大数据处理套件（TBDS）

产品文档

目录

大数据处理套件（TBDS） 7
- 产品简介 7
  - 产品概述 7
  - 产品优势 8
  - 应用场景 9
  - 产品架构 15
  - 产品功能 16
    - 功能列表 16
    - 组件列表 24
  - 客户案例 28
- 操作指南 30
  - 快速入门 30
  - 操作约束与限制 38
  - 主机管理 43
    - 查看主机 43
    - 添加主机 46
    - 重置主机 48
    - 修改主机 50
    - 磁盘修复 53
    - 编辑标签 55
    - 删除主机 57
  - 集群管理 59
    - 集群说明 59
    - 公共集群 60
      - 集群概述 60
      - 安装公共集群 61
      - 启用企业KDC 64
      - 编辑标签 66
      - 新增组件 68
      - 负载检查 70
      - 资源调配 72
      - 服务重启 73
      - 访问Ranger UI 75
      - 卸载组件（销毁服务） 77
    - 经典集群 78
      - 集群列表 78
        - 部署集群 78
        - 集群启停 81
        - 编辑标签 85
        - 软件配置 87
        - 补丁安装 93
        - 集群卸载 94
      - 集群资源 95
        - 节点状态 95
        - 集群扩容 99
        - 集群缩容 102
      - 登录集群 104
      - 客户端管理 107
      - 集群服务 110
        - 服务健康状态 110
        - 服务状态 112
        - WebUI 访问管理 117
        - 配置管理 119
          - 配置更新 119
          - 配置状态 123
          - 配置回滚 125
          - 配置组管理 127
          - Trino资源组配置 129
        - 新增组件 132
        - 卸载组件 133
        - 重启服务 134
        - 启停服务 141
        - 暂停服务 143
        - 客户端管理 145
        - 静态服务池 146
          - 静态服务资源 146
          - 配置集群静态资源 147
        - YARN组件服务 150
          - YARN资源调度 150
            - YARN 资源调度概述 150
            - 配置 Fair Scheduler 151
            - 配置 Capacity Scheduler 160
            - 查看调度历史 171
            - 使用示例 172
          - YARN 作业查询 175
          - 刷新队列 180
        - Impala组件服务 182
          - Impala资源调度 182
          - Impala作业查询 186
        - HDFS组件服务 190
          - HDFS联邦管理 190
          - HDFS挂载表管理 194
          - 联邦使用建议 198
          - RBF组件示例 199
          - 服务操作 209
          - NN主备切换 212
          - 数据均衡 213
          - HDFS 存储策略 215
        - Hive 作业查询 221

- HBase 数据表分析 223
- 集群监控 227
- 集群概览 227
- 操作日志 235
- 日志搜索 236
- 集群事件 240
- 配置告警 245
- 告警历史 247
- 集群巡检 248
- SR集群高级配置 255
- SR 冷热数据分离 255
- 查看JDBC连接串 261
- 数据库表分析 262
- 查询管理 264
- 导入任务管理 266
- ES 集群高级配置 268
- ES 热温冷分层存储策略 268
- 插件管理 275
- 同义词管理 278
- Beats 下载 279
- 数据管理 284
- 数据管理概述 284
- 库表管理 285
- 数据目录管理 285
- 数据库管理 287
- 数据表管理 293
- 表优化配置 303
- 文件管理 307
- 资源管理 309
- 资源组概述 309
- 默认资源组 311
- 创建 YARN 资源组 312
- 编辑 YARN 资源组 315
- 删除 YARN 资源组 317
- 用户权限 319
- 多租户机制介绍 319
- 用户和密码类型 320
- 用户登录 321
- 添加用户 325
- 编辑用户 327
- 删除用户 329
- 为用户授权 330
- 为用户授权资源组权限 335
- 查看用户访问凭证 336
- 重置用户集群访问密码 337
- 设置用户有效期 339
- 重置用户登录密码 341
- 用户组管理 343
- 标签管理 347
- 审计中心 349
- 概述 349
- 操作日志 350
- 访问日志 352
- 日志转储 354
- 升级中心 356
- 安全配置 359
- 配置动态脱敏 359
- 开启云鼎 KMS 365
- HDFS 透明加密 369
- 开启组件审计日志 372
- 开发者指南 373
- 开发规范约束 373
- 应用开发流程 375
- 开发调试模式 375
- 开发发布流程 376
- 开发环境准备 379
- 使用maven仓库 379
- TBDS Maven 下载离线包 379
- 使用TBDS的公共maven开发库 381
- 如何编写Pom.xml文件 384
- 部署TBDS客户端 385
- 身份认证和授权 388
- 新建TBDS用户 388
- 为用户授权 390
- Ranger授权（经典集群） 390
- 获取用户认证 405
- 组件开发指南 406
- TBDS示例工程 406
- 工程目录 406
- 使用流程 410
- HDFS开发 411
- 概述 411
- 示例工程开发 412
- 示例说明 425
- api接口 426
- 常用命令 427
- 常见问题 429
- 权限策略配置 431
- 机架感知配置 434
- Spark开发 436
- 概述 436

- 示例工程开发 437
- 示例说明 451
- api接口 452
- 常用命令 453
- 权限策略配置 455
- 开发规范 462
- 性能调优 471
- 最佳实践 474
- 常见问题 483

- Hive开发 490
  - 概述 490
  - 示例工程开发 492
  - 示例说明 504
  - api接口 505
  - 常用操作 506
  - 最佳实践 511
  - 常见问题 517
  - 权限策略配置 521
- Trino开发 529
  - 概述 529
  - 示例工程开发 532
  - 示例说明 544
  - api接口 546
  - 常用命令 547
  - 常见问题 551
- Impala开发 555
  - 概述 555
  - 示例工程开发 556
  - 示例说明 562
  - api接口 566
  - 常用命令 567
  - 常见问题 571
- Flink开发 572
  - 概述 572
  - 示例工程开发 573
  - 示例说明 581
  - api接口 583
  - 常用操作 584
  - 开发规范 589
    - 术语和数据类型 589
    - DDL数据定义语句 594
    - DML数据操作语句 601
    - SET 控制语句 609
    - 运算符和内置函数 611
    - 标志符和保留字 636
    - 上下游开发指南 641
  - 最佳实践 693
    - SQL解析分段优化方案 693
    - Group Aggregate最佳实践 695
    - TopN最佳实践 697
    - 高效去重最佳实践 698
    - UDF最佳实践 699
  - 常见问题 700
- YARN开发 702
  - 概述 702
  - 示例工程开发 704
  - 示例说明 709
  - api接口 710
  - 常用命令 714
  - 常见问题 715
  - 权限策略配置 719
- HBase开发 722
  - 概述 722
  - 示例工程开发 723
  - 示例说明 731
  - api接口 734
  - 常用命令 735
  - 权限策略配置 737
  - 开发规范 741
  - 最佳实践 768
  - 性能调优 771
  - 常见问题 772
- Phoenix开发 780
  - 概述 780
  - 示例工程开发 781
  - 示例说明 788
  - api接口 790
  - 常用命令 791
  - 常见问题 793
- Kafka开发 796
  - 概述 796
  - 示例工程开发 798
  - 示例说明 801
  - api接口 805
  - 常用命令 813
  - 常见问题 817
  - 权限策略配置 819
  - Kafka机架配置 826
- Kyuubi开发 828
  - 概述 828
  - 示例工程开发 829

- 示例说明 838
- api接口 839
- 常用命令 840
- 常见问题 842
- Iceberg开发 845
- 概述 845
- 快速使用 848
- 性能优化 856
- 常用参数 863
- 最佳实践 868
- 关键功能 877
- 常见问题 878
- ElasticSearch开发 880
- 概述 880
- ES 用户体系介绍 881
- 示例工程开发 884
- Logstash 开发说明 900
- Kibana 开发说明 903
- 高级功能说明 911
- Index 管理 911
- CCR 跨集群数据复制 916
- 【非标】kNN（k-Nearest neighbor）search 919
- API接口说明 927
- 常见问题处理 928
- StarRocks开发 929
- 概述 929
- 快速使用 931
- 表，分区，分桶，索引 931
- CREATE INDEX 931
- CREATE TABLE 933
- DELETE 958
- DROP TABLE 967
- DESC分区键 (Partition Key) 968
- DROP INDEX 971
- REFRESH EXTERNAL TABLE 972
- SELECT 974
- UPDATE 996
- 视图 1000
- ALTER VIEW 1000
- CREATE VIEW 1001
- DROP VIEW 1003
- SHOW CREATE VIEW 1004
- 物化视图 1007
- ALTER MATERIALIZED VIEW 1007
- CANCEL REFRESH MATERIALIZED VIEW 1010
- CREATE MATERIALIZED VIEW 1011
- DROP MATERIALIZED VIEW 1031
- REFRESH MATERIALIZED VIEW 1033
- SHOW CREATE MATERIALIZED VIEW 1035
- 数据导入 1037
- Stream Load数据导入示例 1037
- Broker Load导入 1039
- INSERT数据导入示例 1043
- Kafka connector导入数据到SR 1045
- Kafka Routine Load导入 1050
- 数据湖 1054
- SR对接Hive Catalog 1054
- SR对接Iceberg Catalog 1056
- 数据导出 1058
- Export导出示例 1058
- Spark Connector导出示例 1060
- Flink Connector读取数据 1065
- 性能调优 1068
- 开发规范 1071
- 常见问题 1085
- MapReduce开发 1088
- 概述 1088
- 示例工程开发 1089
- 示例说明 1097
- api接口 1098
- 常用操作 1099
- 常见问题 1101
- Hudi开发 1103
- 概述 1103
- 快速使用 1104
- 常用参数 1114
- 关键功能 1118
- 开发规范 1122
- 最佳实践 1141
- 常见问题 1152
- API文档 1154
- 腾讯大数据套件（tbds） 1154
- 版本（2020-01-16） 1154
- API 概览 1154
- 调用方式 1155
- 接口签名v1 1155
- 接口签名v3 1162
- 请求结构 1171
- 返回结果 1172
- 公共参数 1175
- 腾讯大数据套件 1177

• bms创建集群 ····· 1177
• 根据实例名称获取bms实例信息 ····· 1179
• 判断集群名是否存在 ····· 1180
• 创建集群 ····· 1181
• 获取集群详情 ····· 1183
• 获取集群节点 ····· 1184
• 销毁集群 ····· 1186
• 获得集群列表 ····· 1187
• 获得集群部署进度 ····· 1189
• bms创建实例接口测试 ····· 1190
• 数据结构 ····· 1193
• 错误码 ····· 1196

产品简介
产品概述

大数据处理套件（Big Data Suite，TBDS）是基于云服务商多年大数据实践，面向数据全生命周期，对外提供的安全、可靠、易用的一站式、高性能、企业级大数据存储计算分析平台。

TBDS基于Lakehouse打造的类数仓体验的全景湖仓架构，具备计算隔离、存算按需、智能调优等高可靠特性，借助统一元数据以及权限服务可实现异构存储集群数据互访，数据零搬迁，降低用户整体使用成本。

TBDS按照技术栈不同，可进一步分为如下子产品。

- 基于Hadoop生态的大数据处理技术的TBDS-数据湖(简称TBDS)，通过开源体系的大数据组件，构建轻量易用的大数据底座，最小规模6节点，适用于统一数据湖构建、海量数据存算分析场景；
- 基于PG(Postgres)的分析型数据仓库TBDS-数据仓库(简称 TCHouse)，最小规模3节点，适合中小型客户的数仓、数据集市建设以及OLAP数据分析场景；
- 基于Elasticsearch内核的检索引擎 TBDS-Elasticsearch(简称 ES),可构建全文搜索、日志分析等业务场景，云服务商在ES内核方面的贡献是亚太第一。
  同时TBDS底座可以和数据开发治理平台WeData、BI进行对接。
- WeData包含数据集成/开发/测试/运维的全链路DataOps数据开发能力，以及数据建模/质量/安全/资产/服务等一系列数据管理和治理能力，产品方案符合研发运营一体化设计思路；
- BI工具帮助企业进行可视化的经营分析、财务分析、决策支持。

产品优势

DATA+AI

提供全栈大数据引擎，按需灵活搭配，完全兼容开源 Hadoop 标准生态；升级多模态大数据平台，统一管理非结构/结构化数据，结合UDF函数对模型封装，实现SQL对多模态数据混合分析。

全景湖仓创新架构

基于Lakehouse数据架构打造的类数仓体验的全景湖仓产品，具备计算隔离强、存算按需伸缩、存储智能调优等高可靠特性，通过利用统一元数据和统一安全，能够在无需数据迁移的情况下，轻松实现实现跨集群数据互访与分析，从而显著减少用户的整体使用成本。

极致的性能与可扩展性

支持单项目10万+节点，单集群万节点的超大规模部署，具备日接入数据 百万亿条，日实时计算百万亿次的极致处理能力;通过深度优化开源计算组件，综合性能优于开源50%-200%。

智能管控服务

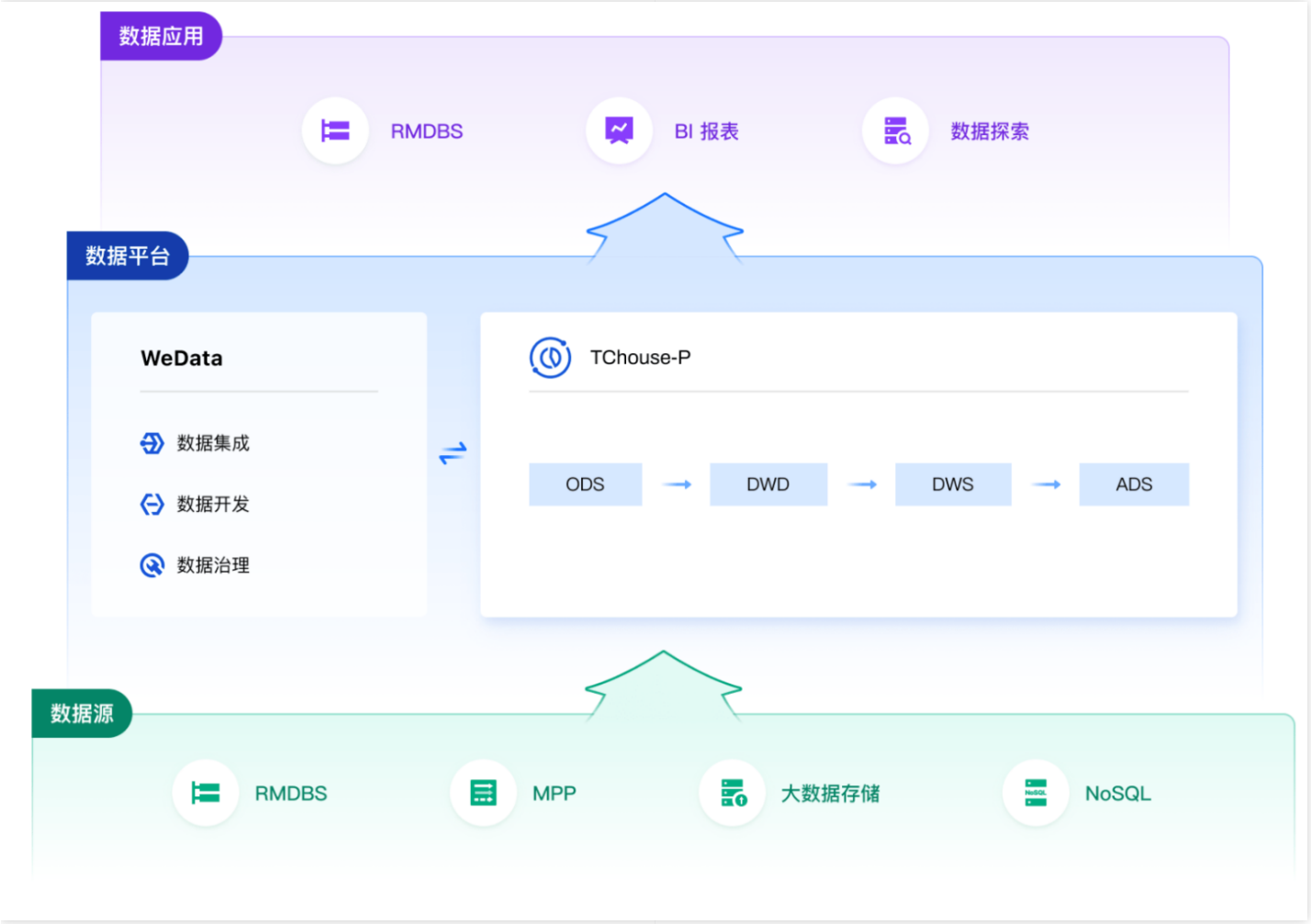
提供开箱即用的极速启动体验，支持集群和产品的一键部署、监控、预警、安全管控、容灾备份和迁移。通过AI加持的智能运维能力，结合运维知识库与系统运行指标，实现故障事前预警、自愈和事后根因分析，有效降低综合运维成本。

融合创新安全可信

全链路芯片、操作系统、服务器生态兼容，支持x86、ARM双架构以及 IPv6 部署环境，同时可提供大数据平滑迁移工具和方案、通过软硬件专项性能优化、国密等能力，构建安全创新的大数据平台。

应用场景

OLAP 数仓分析



需求场景：OLAP数仓分析场景，主要面向政府企业，在数据统一汇聚的基础上，希望深度挖掘数据价值，通过建设数仓，以数据可视化和数据应用的方式赋能业务，TCHouse提供数仓开发处理、分层建模，搭配海量数据查询引擎，提升海量数据查询性能，满足实时/在线数据分析的需要 。

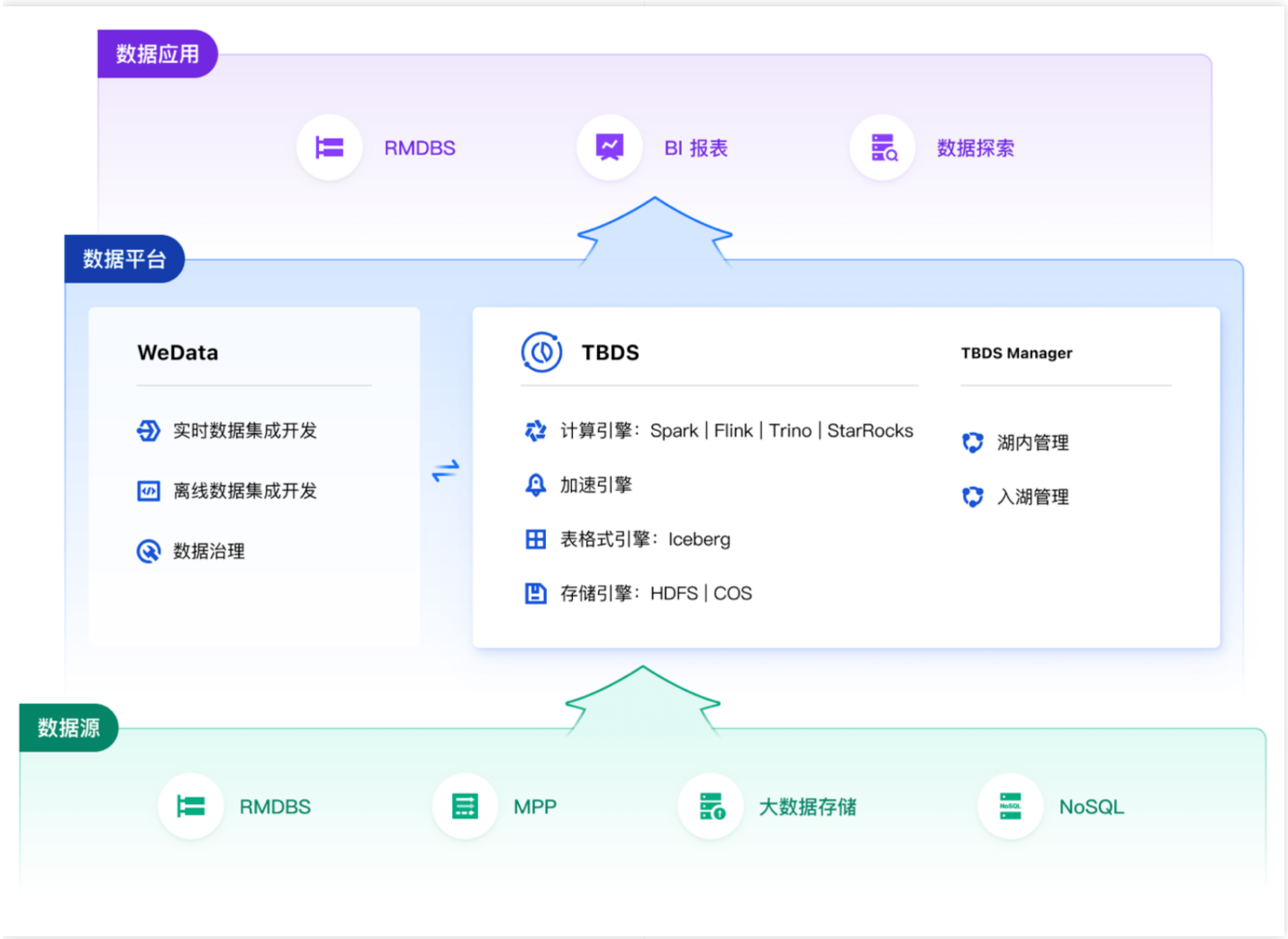
产品优势：【极致性能】全并行分布式架构，节点间、节点内、算子间全并行处理，高效向量化执行引擎，延迟物化技术，万亿级关联查询秒级返回；【行列混合存储】支持高效行列混合计算，列存引擎支持多种压缩算法压缩级别，提供自适应压缩能力和高压缩比。【安全高可用】支持三权分立、数据透明加密、数据脱敏，强制访问控制及全方位审计能力；支持多级容灾高可用；【自动化运维】自动化管控平台对集群、节点、资源、租户等信息进行全方位管控。

多模态大数据平台

需求场景：随着企业数据平台的发展，需要在云环境下构建高效数据存储和计算分析能力，用于： 支持数据分析人员进行海量数据的探索性分析； 为数据科学家存储和管理训练数据集，训练和部署机器学习模型； 通过数据管道为企业提供实时分析和决策支持； 为业务智能平台提供数据存储计算能力保障； 方便不同业务部门之间的数据共享和协作。

产品优势：【多模数据管理】采用Lakehouse架构统一接入和管理多模态数据，并内置向量库高效存储向量数据。【智能分析】基于函数实现对大模型能力的封装，可基于SQL实现对结构化和非结构化数据的融合分析；同时针对业务模型和业务知识库进行自动智能优化，通过优化后的模型可实现智能问数。【智能运维】整合大数据运维知识库与系统运行指标，实现部分故障的自动化恢复、事前预警以及事后根因分析，有效降低运维成本，提高平台的整体可靠性和维护效率。

云原生数据湖

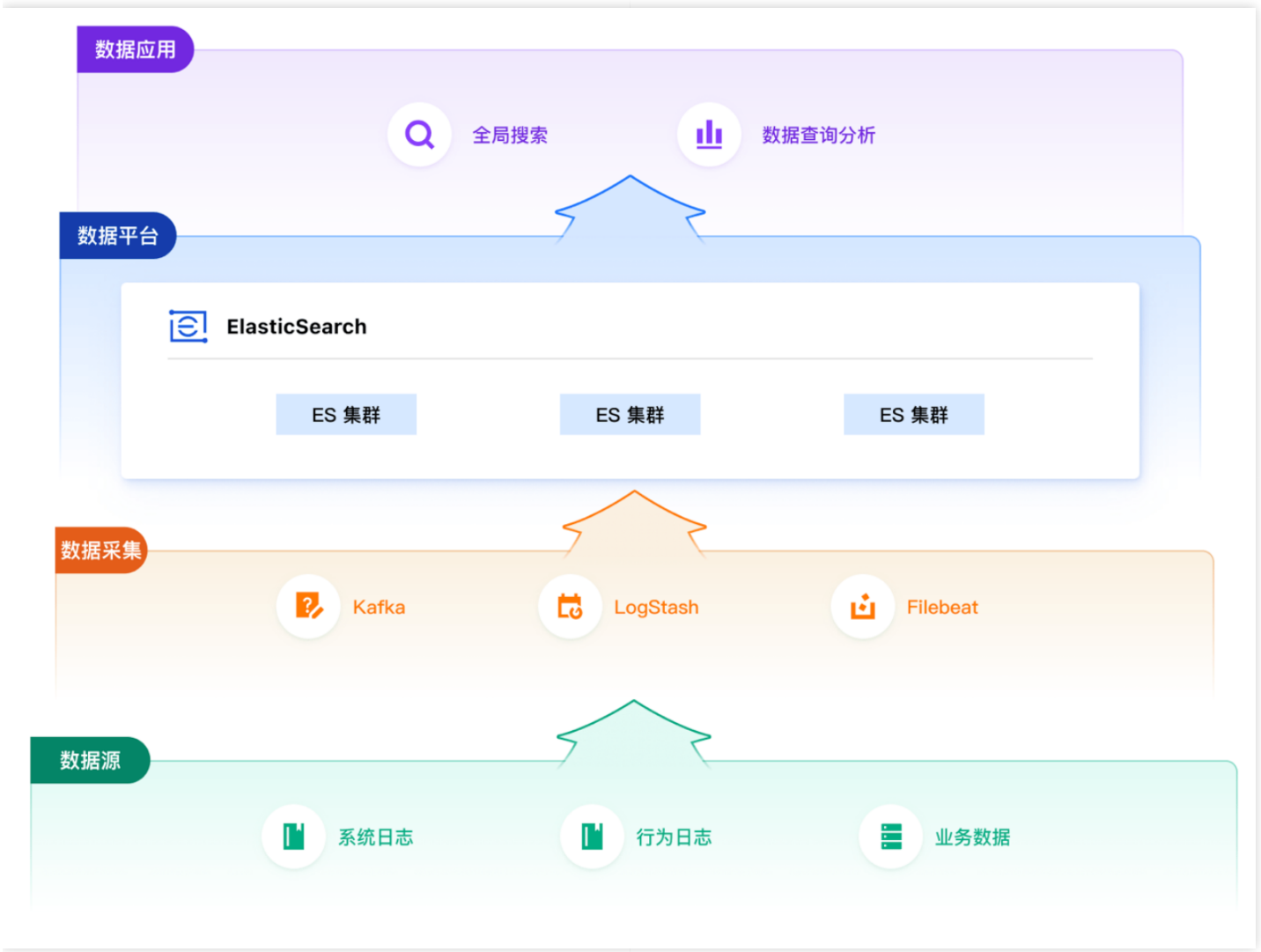


需求场景：随着企业数据平台的发展，需要基于海量数据构建高效存储和计算分析能力，用于支持数据分析人员进行探索性分析；为数据科学家存储和管理训练数据集，以及训练和部署机器学习模型；为企业提供实时分析和决策支持；为业务智能平台提供数据存储计算能力保障；方便不同业务部门之间的数据共享和协作。
产品优势：【创新架构】采用对象存储、Iceberg、Luoshu 等组件搭建高性能数据湖架构；【极致性能】从数据入湖、存储、计算、探索分析等维度大幅提升数据接入和计算的实时性、湖内外流转的性能和稳定性；【降本增效】资源弹性综合利用率提升30%+；存储成本降低20%+。

湖仓一体

需求场景：随着企业AI场景发展，存在大量创新性数据分析和应用需求，但传统数仓式建设的层级结构复杂、修改困难，而企业缺乏高效的自助开发能力，导致无法快速响应需求。 企业在不同发展周期会产生独立的数据仓、数据湖以满足需求，同时也带来了数据的冗余和混乱，缺乏统一的湖仓平台能力对孤岛数据进行联动管理和应用。
产品优势：【开放敏捷】采用开放标准分层解耦设计，架构敏捷灵活，易扩展；【弹性伸缩】云原生架构设计，极致弹性；【混合负载】实时离线一体化，计算隔离无干扰，减少数据移动；【集约成本】存算按需伸缩，智能调优，易用性高，维护成本低；【融合架构】统一数据存储和元数据管理，严格事务一致性，以融合架构支持湖仓平台建设。

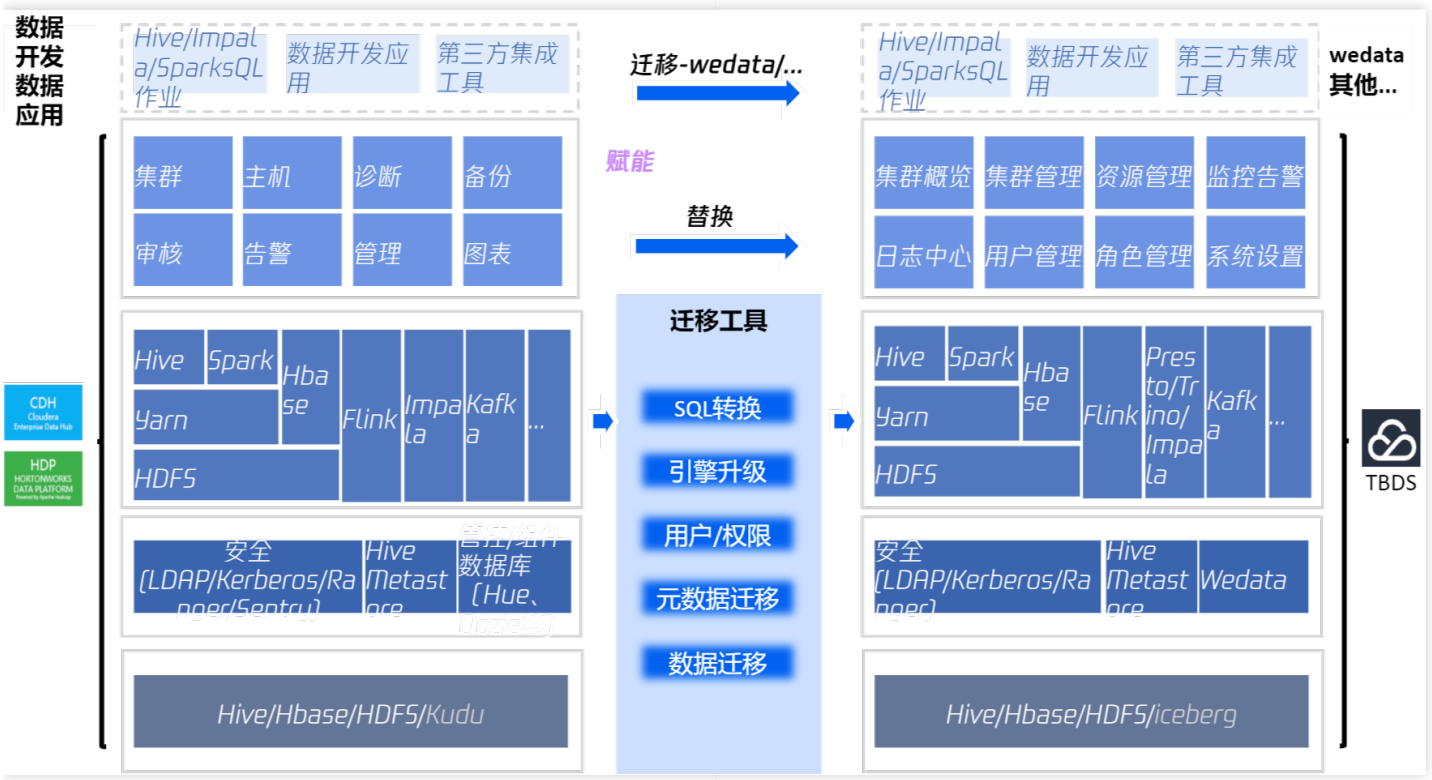
信息检索分析



需求场景：企业在数据应用过程中，往往存在大量日志分析、信息检索、商业智能构建的业务场景，这些业务往往面临数据存储分散、种类繁多、规模庞大问题，以致应用困难；需要一款检索引擎来提供精确查询、全文查询、向量检索的能力。

产品优势：【高效可靠】总体效果：写入性能提升10%-50%+，查询性能提升 10%-50%+，GC 下降接近一倍，存储成本降低30%-80%；【运维管控】提供易运维的、可观测的、多维管理的ES集群管控能力；【开箱即用】提供了开箱即用的可视化分析工具及强大的聚合分析能力，方便用户实现海量数据的实时检索分析和结果呈现。

数据迁移



需求场景：在市场导向的大趋势下，大量金融行业、国有企事业单位需要将原有存量的CDH大数据平台迁移到国产XC大数据平台，但存量数据平台存在大量数据生产应用，不允许出现大面积停服带来的风险；需要一套高效、安全、渐进、自动化、可回溯的迁移方案和有经验的专业团队，保障平台迁移升级动作不会中断/影响业务应用。

产品优势：【平滑适配】TBDS开源兼容自主创新，能无缝对接CDH开源hadoop生态；【端到端产品】提供了一套CDH平滑迁移标准产品服务，包括组件迁移、数据迁移、工作流业务迁移方案；【高效可靠】同时考虑了迁移效率、准确性、容错性、自动化能力，迁移效率提升80%，全链路数据校验，准确性达99%。

产品架构

TBDS大数据平台是提供的面向海量数据的接入、存储、计算、分析统一平台，提供接入、存储、计算、服务、分析、管理等多个组件，支持用户本地部署、云上全托管和混合云多种部署模式，并提供多种运维管控工具实现统一管理，主要提供功能如下：

- 数据湖仓：提供标准存算引擎和虚拟计算引擎，可快速覆盖客户大部分数据中台场景需求；通过提供公共集群统一元数据服务和统一安全管理能力，支持动态计算资源调度、虚拟计算、对象存储，可实现全新湖仓能力支持。
- 平台管控：多架构统一智能运维管控平台:多维度精准洞察集群、服务、主机、容器运行情况，充分满足项目对指标监控的需求。支持更多服务专有资源分析和作业查询。多租户资源管控，细粒度资源管控体系全面覆盖各个服务，资源管控更精准。运维链路全面优化，功能更丰富，引入智能健康诊断体系，问题自动发现，专家故障处理建议，更易使用。
- 信息技术适配：全面支持XC本地化生态，涵盖主流硬件架构（x86/ARM）、操作系统、服务器、数据库等领域，并针对性进行性能稳定性调优。
- 集群迁移：一站式数据迁移校验工具，杜绝手工模式，结合专家服务标准流程提供整体解决方案支持，针对CDH典型迁移场景可提升综合迁移效率达80%。
- 灾备：零侵入设计，可视化操作，单集群/多集群跨AZ/跨机房容灾能力（两地三中心），数据备份、同城灾备、异地灾备方案全方位支持，系统业务级容灾能力。
- 性能：进行组件级参数调优，提供更高的性能表现，Spark、Iceberg等组件提供丰富特性，提升场景化性能。
- 安全：平台通过支持Ranger、Kerberos、LDAP组件，提供认证、权限管控、审计等4A级全方位安全保障。数据层面支持国密算法，在数据存储、传输、应用层实现加密。

产品功能
功能列表

> 说明：
>
> ES集群、SR集群需要单独采购。

模块	一级	二级	三级	功能描述	说明
概览	/	/	使用指引	展示TBDS平台使用引导与快捷入口。	
			集群数量概览	聚合展示用户经典集群数量和处于不同状态下的集群数量。	
			集群状态聚合展示	聚合展示用户全部经典集群的集群状态和集群服务运行状态。	
			集群事件聚合展示	聚合展示用户全部经典集群的集群事件信息。	
主机列表	/	/	主机列表	提供主机列表可视化展示，包括主机ID、主机名、在线状态、重置状态、IP地址（IPv4/IPv6）、机型/机架、硬件架构、可用区、网络信息、配置规格、操作系统、自定义标签及创建时间等核心信息。同时支持多维度字段筛选与排序。	
			主机管理	提供主机管理维护功能，支持批量增删和重置主机。重置流程含可视化磁盘管理（如格式化、挂载/卸载等），可查看主机详情，并支持批量设置主机属性（如机架、机型及自定义标签等）。	
			机架感知	支持HDFS、YARN、Kafka等组件的机架感知能力，并基于机架策略实现组件滚动重启与升级，保障服务高可用。	
集群列表	经典集群	集群列表	集群管理	列表化展示平台当前管理的集群，支持使用标签对集群类型进行分类管理，支持导出软件配置。	
			集群基础信息展示	列表化展示平台当前管理的集群的运行状态、创建时间等基本信息。	
			集群一键启停	支持集群维度的一键启动、停止和重启。	
		集群安装	安装部署	支持自动化、图形化、流程化的集群安装部署，覆盖多种集群类型，提供场景/组件/网络（IP单双栈）/节点类型选择，并能基于主机属性（如机型、自定义标签等）快速筛选目标主机。支持可视化自定义软件配置与系统配置。支持自定义部署拓扑。支持HBase、	

部署		Trino、RSS类型集群的独立部署。	
集群概览	集群运行状态概览	支持展示集群运行状态，节点数量，元数据库运行状态、Kerberos模式状态、引擎版本等信息。	
	集群服务状态概览	支持集群服务状态的展示，展示集群的全部服务组件和组件健康状态。	
	关键服务负载状态概览	支持对HDFS和YARN运行和负载状态的展示和时间维度的负载和用量趋势展示。	
	集群节点进程概览	支持集群节点进程状态的展示，节点按类型区分，展示已部署的标准进程，缺失进程、非法进程。	
	集群节点状态监控	支持对节点运行指标进行监控，包含节点的CPU核数、CPU使用率、内存使用情况、内存使用率、磁盘空间使用率、磁盘IOPS、磁盘IO、进程、网络等监控信息。	
集群信息	/	展示集群的基础信息和软件信息，包括集群版本、节点数量、部署组件等信息。	
集群服务管理	集群服务列表	支持展示集群已安装的全部组件信息，包括组件版本、即时状态和WebUI地址。	
	集群服务WebUI管理	支持集群服务WebUI的展示和鉴权访问， 展示服务所有 WebUI 地址。	
	服务状态监控	支持展示服务列表，查看各服务运行状态、服务基础信息监控等内容。	.
	服务运维操作	支持对服务进行新增、启停、重启、端口查看等基础操作，支持按服务和角色重启，支持设置服务滚动重启时间间隔。	
	角色实例管理	支持对服务角色实例的展示，包含角色实例的运行状态、配置状态、配置组、节点类型、维护状态、最近重启时间和节点IP；支持按时间维度查看角色实例的监控指标。	
	配置管理	组件配置的可视化管理，支持对服务配置进行新增、编辑、删除等操作；支持按照集群维度、节点维度、配置组维度进行配置批量操作；支持各个维度的参数对比能力；支持配置历史管理，可以查看每次配置的变更原因、变更详情，支持配置回滚能力；支持新增配置文件、定义配置文件的名称、存储路径、操作权限等信息；支持新增配置项，配置项值域提示、配置描述、默认值还原等能力。	

			YARN可视化管理	支持对YARN组件资源调度进行可视化管理，支持Fair Scheduler和Capacity Scheduler两种调度器的选择，两种模式均支持多配置集，支持多级队列的配置，支持新增、编辑、克隆资源池，在Fair Scheduler支持基于时间的计划模式、放置规则和用户ACL控制，支持DRF、Fair和FIFO三种调度策略并支持任务抢占模式；在Capacity Scheduler模式下支持基于时间的计划模式，支持配置绝对资源配额和用户权重配置，支持新增、编辑、克隆资源池，支持资源池映射，权限模式配置。支持计划模式配置，支持按日、周、月调度配置队列资源配额。提供配置集切换功能，不同配置集支持队列运行在不同参数下。	
			Impala资源调度	支持对Impala组件资源调度进行可视化管理，支持对Impala组件集群队列资源的管理，包括创建、编辑、显示当前队列等；支持放置规则配置。	
			Impala 角色分离	支持部署独立的Coordinator和Executor角色，减轻大集群情况下元数据同步负担	
			静态服务池配置	支持配置HDFS、YARN、Trino、Impala、HBase等服务的CPU/内存/IO资源使用限制，实现服务级别的资源隔离；同时支持日、周、月三种周期化的调度方式，能够使集群在不同的时间段根据配置动态调整不同组件可以使用的资源。	
			HDFS联邦管理	支持对集群进行联邦拓展，为HDFS集群添加多个独立的HDFS NameService，实现HDFS元数据隔离。 安装部署阶段支持RBF配置自定义，挂载表支持编辑和挂载策略自定义，数据管理、权限、授权服务兼容多 NameService。	
			服务依赖重启	支持服务变更提醒及一键重启过期服务。	
		存储资源分析	HDFS存储分析	支持对HDFS的存储进行分析，包括按天统计存储的总文件数、总存储量、文件数量分布、文件存储量分布等内容；支持按时间段分析文件数据和存储量的使用趋势，包含Top大小文件的展示信息。	
			Hive表分析	支持对hive的库表、存储量、数据表分布等信息进行统计分析。	
			HBase表分析	支持对HBase的读写请求、MemStore、StoreFile、Region和RegionServer信息进行监控。	
			StarRocks数据表分析	帮助用户全面了解数据、管理数据。数据库表分析，分区分片的多维度分析及下钻协助用户根据需要对分区分片进行调整，以优化数据存储和提高查询性能	申请SR后可见

		服务作业管理	Hive作业管理	支持查看Hive组件的任务运行情况，包含作业执行时长；支持Hive查询任务详情查看，包括查看查询语句、查询计划、执行总览和Profile信息。	
			YARN作业管理	支持查看YARN任务运行情况，包含作业提交量、top用户使用的CPU核数、内存消耗量、作业列表等信息，包含用户使用情况的趋势分析；支持YARN作业洞察，包括任务运行阶段时间解析、作业对比、运行参数展示和节点运行参数；支持YARN查询任务详情查看，包括查看作业信息和元数据信息详情；支持YARN任务查杀。	
			Impala查询管理	支持查看Impala查询引擎的任务运行情况，包含查询执行时间、执行状态、查询扫描行数、CPU/内存使用情况等信息；支持通过查询ID和语句检索Impala作业；支持Impala查询任务洞察，包括任务运行阶段时间解析、运行参数展示和查询节点运行参数；支持Impala查询任务详情查看，包括查看查询语句、查询计划、执行总览、内存使用和Profile信息。	
			StarRocks导入任务管理	提供批量数据导入 Broker Load 和实时数据导入 Routine Load 类型的任务信息，可查看任务的状态、进度和详细信息方便监控和管理。	申请SR后可见
			StarRocks查询管理	提供 StarRocks 查询多维度指标及详情展示，查询列表可快速查看查询语句、查询开始时间、查询状态、查询热点、查询持续时间、用户、获取行数、CPU 总时间、消耗总内存等多项明细指标，并可通过查询详情查看相关 SQL、查询计划、Profile、执行拓扑。	
		集群资源管理	集群负载状态概览	支持聚合展示集群负载状态，包括集群CPU使用率、内存使用情况、内存使用率、磁盘空间使用率、磁盘IOPS、磁盘IO、进程、网络等指标。	
			节点列表	支持展示集群全部节点，支持按照节点类型展示。	
			节点扩容	支持按照节点类型对节点进行新增、删除操作，支持批量添加节点。	
			节点缩容	支持按照节点类型对符合缩容条件的节点进行卸载操作， 支持重点组件优雅缩容和强制缩容。	
			节点配置展示	支持展示节点配置信息，包括CPU核数，内存物理容量和磁盘物理容量信息。	
			节点进程监控	支持按节点查看服务进程的部署状态。	
			节点运行指标监控	支持对节点运行指标进行监控，包含节点的CPU核数、CPU使用率、内存使用情况、内存使用率、磁盘空间使用率、磁盘IOPS、磁盘IO、进程、网络等监控信息。	

		集群监控	监控大盘	支持自定义监控面板，支持在面板中自定义配置已采集的监控指标。 新增 HBase Replication 相关指标的监控。	
			集群事件	支持对集群和各组件运行进行集群事件告警，支持按时间维度查看集群事件历史。	
			日志搜索	支持按时间维度、组件维度、节点维度、日志级别维度及关键字查询日志，支持定位到日志上下文。	
			告警历史	支持对集群和各组件运行情况配置监控告警，支持按时间维度查看告警历史。	
			Java分析	支持Java进程的GC在线分析，提供可视化的GC 视图及点位信息分布，并支持展示HDFS、YARN、HBase、Hive等服务的GC明细列表。	
		客户端管理	客户端节点列表	支持展示安装客户端的节点，查看安装路径、安装时间、更新时间、安装状态等信息。	
			客户端安装	支持按需选择组件客户端，自动安装至指定主机并记录信息，同时支持客户端更新。	
			下载客户端	支持按需下载组件客户端至指定节点路径，支持仅下载组件客户端配置文件。	
		操作日志	操作记录列表	支持对操作日志进行审计，记录操作对象、操作详情、操作时间、操作人等信息。	
		ES高级配置	插件管理	支持的内置插件包括 ES 开源插件和特有插件在内的10余款插件，支持安装、卸载、重装插件，其中 IK 分词插件还支持词典管理。	申请ES后可见
			同义词管理	支持上传同义词文件、直接引用同义词进行配置。	
		集群巡检	/	Hadoop集群可以即时或定时（按天/周）选择巡检项对集群节点和服务进行健康检查，并生成巡检报告；平台预置集群/节点/组件维度巡检项，并支持巡检报告定制化。	
公共集群	基础服务	安装部署	/	支持主机物理化部署形态，自动化、图形化、流程化、自定义部署流程，支持配置企业Kerberos认证信息。	

		服务信息	/	展示基础服务的基本信息、资源使用概览、服务状态和资源监控，支持新增和卸载组件及集群打标。通过“编辑标签”功能，便于对不同业务集群进行更精细化的权限管理。	
		服务列表	服务监控	支持对服务角色的展示，包含角色的POD健康状态、运行/期望POD数、资源配置、最近重启时间等信息；支持对服务角色进行重启、调整POD数量、变更配置操作。	
			服务运维	支持服务重启，POD数量调整和配置变更。	
			配置管理	组件配置的可视化管理，支持对服务配置进行新增、编辑、删除等操作；支持配置历史管理，可以查看每次配置的变更原因、变更详情，支持配置回滚能力；支持新增配置文件、定义配置文件的名称、存储路径、操作权限等信息；支持新增配置项，配置项值域提示、配置描述、默认值还原等能力。变更配置中“节点调度策略”，可根据调度规则，将Pod调度到符合预期的Label节点中。	
			日志	支持按时间维度、服务角色维度、节点维度、日志级别维度查询日志，支持定位到日志上下文。	
			销毁服务	支持对非 LDAP、 Kerberos Ranger 组件销毁服务，避免垃圾数据干扰。	
		集群监控	监控大盘	支持自定义的监控面板，支持在面板中自定义配置已采集的监控指标。	
			监控告警	支持对各组件运行情况配置监控告警，支持按时间维度查看告警历史。	
平台管理	用户权限	用户管理	/	支持对平台用户进行新建、删除、查看详情等操作，支持对用户进行用户组、角色、策略、数据权限、资源权限的关联操作。支持配置认证有效期，支持账号状态显示。	
		用户组管理	/	支持对用户组进行新建、删除、添加用户等操作，同时进行角色、策略、数据权限、资源权限的关联操作。	
		策略管理	/	支持查看当前系统的预设和自定义策略及策略详情，同时支持进行策略权限关联操作。自定义策略可对集群操作、资源权限进行细粒度管理及授权；支持自定义菜单权限；	
	资源管	资源组	/	提供资源组各类信息查看，包括资源组类型、状态、全部配额、创建时间等。	

理	列表			
	资源组操作	/	提供自定义资源组，包括对YARN资源组的新建、编辑、删除等操作。	
数据管理	数据目录	数据目录管理	提供经典集群中Iceberg 和 Hive数据目录（catalog）的刷新、编辑、查询。	
		数据库管理	提供Iceberg 、Hive数据库的新建、编辑、删除、查询。	
		表管理	提供Iceberg 、Hive数据表的新建、编辑、删除、查询和授权，当前支持可视化建表，支持查看表详情，包括表的基础信息和属性、字段和分区、快照、变更历史。	
		数据权限	平台统一权限模型，支持库表级联、对数据目录、数据库、表进行增加、删除和编辑，支持将权限批量授予用户和用户组；平台权限信息与 Ranger联动并保持一致。用户权限配置支持脱敏配置。	
		表优化	展示表优化列表，包含表优化状态、耗时、文件大小和文件个数。支持经典集群小文件治理。	
		文件目录管理	展示用户在经典集群中有权限的HDFS文件目录信息，管理员拥有所有权限，可查看所有文件和目录，支持新建、查询、重命名、删除、授权回收、复制路径等操作。	
		文件管理	展示用户在经典集群中有权限的HDFS文件列表信息，管理员拥有所有权限，可查看所有文件和目录，提供文件的浏览、复制、移动、重命名、上传下载、授权回收、删除操作。	
审计中心	操作日志	/	提供完善的集群操作日志记录，保证数据操作的可追溯性；支持操作日志生命周期设置，超出生命周期的操作日志将被清理	
	访问日志	/	访问日志需通过Ranger UI Audit查看。支持访问日志生命周期设置，超出生命周期的访问日志将被清理。	
	日志转储	/	支持定时定量将审计日志转储至外部FTP地址，支持转储调度的新增、编辑、状态变更。	

	文档中心	/	/	提供“文档中心”功能，可直接离线查看网页版产品文档。	
工具箱	升级中心	补丁包记录	补丁包列表	展示补丁包物料信息，包括补丁包名称、补丁包状态、支持集群版本、补丁包组件、补丁版本、已安装集群、存储容量、上传时间。	
			补丁包管理	提供对补丁包物料的添加、删除等操作。	
		集群升级记录	升级记录列表	展示集群升级信息，包括集群ID/名称、当前集群版本、补丁升级、最近补丁升级时间、最近集群升级时间。	

组件列表

经典版组件

> 注意：
>
> Flink组件：默认仅提供运行时支撑，不提供任何生态 Connector，如需构建复杂的业务场景，建议申请 WeData 数据集成、实时开发模块来满足。

分类	组件名称	组件说明	类别	变更类型	组件版本
数据接入	Kafka	分布式消息中间件：可提供高吞吐量的分布式发布订阅消息系统	用户组件		2.8.2
	KafkaManager（CMAK）	用于管理Kafka集群的可视化工具	用户组件		3.0.0.6
数据存储	HDFS	Hadoop分布式文件系统：支持高吞吐量、高度容错性，适合运行在通用硬件上的海量数据分布式文件系统	用户组件		3.2.2
	HBase	基于HDFS的分布式列式数据库：一个高可靠性、高性能、面向列、可伸缩的分布式存储系统	用户组件		2.4.5
	Uniffle	一个远程 Shuffle 服务，为 Apache Spark 应用程序提供了在远程服务器上存储 Shuffle 数据的功能	用户组件	新增	0.10.0
	Phoenix	HBase的SQL查询引擎，为HBase数据提供关系型数据库一样的SQL查询功能	用户组件		5.1.2

	Iceberg	面向海量数据分析场景的开放表格式数据组织方案，与数据存储和计算深度融合提供数据实时流批一体处理的能力	用户组件		1.6.1
计算分析	MapReduce	MapReduce是一个基于集群的高性能并行计算平台	用户组件		3.2.2
	Tez	Tez 是支持 DAG 作业的计算框架，一般和 Hive 组合使用。	用户组件		0.10.2
	Spark	集交互SQL查询、批处理、流式计算、图计算、机器学习为一身的新一代大数据计算框架	用户组件	升级	3.5.4
	Flink	一个面向数据流处理和批量数据处理的可分布式的开源计算框架	用户组件		1.16.1
	Hive	基于Hadoop的数据仓库工具：将结构化的数据文件映射为数据库表，并提供简单的SQL查询功能，可以将SQL语句转换为MapReduce任务运行	用户组件		3.1.3
	Trino	开源的MPP架构的OLAP查询引擎，可针对不同数据源执行大容量数据集的一款分布式SQL执行引擎	用户组件		435
	Impala	Impala是一个分布式SQL查询引擎，主要用于对存储在Hadoop HDFS或HBase中的大数据进行低延迟的SQL查询与分析。	用户组件		4.1.0
	StarRocks	是一款高性能分析型数据仓库，使用向量化、MPP 架构、CBO、智能物化视图、可实时更新的列式存储引擎等技术实现多维、实时、高并发的数据分析	用户组件	升级	3.3.11
	ElasticSearch	Elasticsearch 是位于 Elastic Stack 核心的分布式搜索和分析引擎	用户组		7.10.1

			件		
	Kibana	数据分析与可视化工具	用户组件		7.10.1
资源调度	Yarn	提供统一的资源管理和调度能力	用户组件		3.2.2
开发工具	Hue	Hue是一款基于Web的交互式查询编辑器	用户组件		4.10.0
公共支撑	Kyuubi【可选】	统一的多租户JDBC接口，用于大规模数据处理和分析，建立在Apache Spark™之上	用户组件		1.9.0
	Knox	后台服务-Hadoop API/UI网关	系统组件		1.6.1
	ZooKeeper	分布式协调服务：为分布式应用提供一致性服务，提供配置维护、名字服务、分布式同步、组服务等功能	系统组件		3.7.2
	Ranger	集中式安全管理，可在Web UI或使用REST API管理所有与安全相关的任务；基于策略（Policy-Based）的访问权限模型,支持大部分Hadoop生态组件，支持审计，支持与LDAP、Linux系统的用户同步	系统组件		2.3.0
	Kerberos	网络认证协议，其设计目标是通过密钥系统为客户机/服务器应用程序提供强大的认证服务	系统组件		1.21.2
	LDAP	后台服务-轻量目录访问协议	系统组件		2.4.44
	MetaService	统一元数据管理服务，支持多Catalog集中管理	系统		0.7.0

			组件		
	Luoshu	数据湖优化服务，支持包括文件合并、文件清理等	系统组件	新增	0.7.1

客户案例

X银行数据湖及大数据应用项目

- 客户痛点
  长期以来数据价值挖掘能力欠缺、数据孤岛严重是该银行在推进数据应用上面临的现实困境。严重阻碍了科技创新和数字化转型。
- 解决方案
  构建UDP数据湖平台：数据湖一期以数据为中心，以“覆盖BDP/EDP数据源、一次接入分批加载”为与原则，搭载以离线文本接入、文件湖、缓冲区和贴源层为主体的数据治理体系，离线接入超过一万张源表加载至贴源层、逐步替代EDP、BDP提供数据支撑。
- 客户价值
  数据湖平台帮助银行首次实现了全行数据资产汇集一处，能够为银行统一数据分析层、展现层、数据沙箱等探索提供平台支撑，也为沉淀和深度挖掘全行数据资产打下了坚实技术基础。
- 项目总结与借鉴
  本项目是TBDS在国有银行体系的首个案例，包含了大数据平台、人工智能平台、数据湖及银行数据层面整体规划的综合性项目。

X商业银行湖仓一体项目

- 客户痛点
  - 原友商数据库难以满足业务发展需求，如自助分析、实时数据处理、AI分析等场景
  - 数据开发治理工具分散，未形成一站式数据敏捷开发能力，开发效率低
  - 数据建模体系陈旧，灵活性及扩展性不足，维护成本高，业务响应慢查询
- 解决方案
  - 湖仓一体架构体系，全面覆盖客户湖仓融合的场景需求
  - 数据敏捷化开发工具能力，降本增效
  - 云平台团队互联网建模思路输出，提升数据响应
- 客户价值
  - 实现业务价值六大突破：数据存算更高效，数据管理更简单，数据资产更清晰，业务驱动更智能，数据运营更高效
  - 帮客户构建全公司级统一数据平台，实现数据统一赋能，全链路一体化管理，包括：
    - 大数据湖仓一体化，构建离线、实时计算能力资源池
    - 全链路数据治理工具链能力打造，数据集成、开发、管理核心能力赋能，每日任务数量5000+
    - 基于Iceberg数据湖、StarRocks等技术栈实现湖仓一体新架构转型，有效提升离线场景、实时场

景分析效率和即席数据分析能力

X能源企业大数据平台与数据湖建设项目

- 客户痛点
  - 原有平台独立建设，烟囱耸立，资源管理和利用效率低下
  - 架构陈旧，运维能力不足，无法支撑微信账单、设备台账等业务性能要求
  - 难以实现公共数据积淀，无法形成企业统一管理视图
- 客户价值
  - 支撑了电网管理平台、客户服务平台、企业运管平台、调度运行平台的数据实时采集入湖，整体规模达到1200+节点
  - 我们基于Flink+iceberg实时入湖，TB级天增量入湖，入湖效率提升30%
  - 各业务数据统一归集到数据湖底座，数据服务层作为数据中心对外提供服务能力的应用层，加速了数据应用的融合，满足智能抄表，智能配电房，环境监测，视频监测应用场景
- 项目总结与借鉴

  卡位能源电力行业，作为X电网集团的数据底座，后续扩容规模大。对于他各网省规划大数据平台方案具备示范性标杆作用。

X银行多业务实时检索

- 客户需求

  为风控系统、催收系统、监管报送、信贷系统提供日志分析与检索。
- 解决方案
  - 集群架构：两地三中心数据容灾架构，通过 TM统一管控；客户规模约 50 套 ES 集群，总共近 1000 节点
  - 性能支撑：ES 内核性能增强、稳定性提升、计算成本优化三个方面优化点 100+个
- 客户价值
  - 稳定+价格便宜：集群安全认证、多集群备份、审计、组件 HA、等方面保障集群稳定；价格比原厂便宜 50% 以上
  - 性能提升：相比开源&其他友商，云平台团队内核研发能力，在聚合查询、scroll查询、过滤查询等方面高性能，保障业务稳定性
  - 易管控：TBDS-Manager 管控，在安装部署/扩容/可观测/es 高级特性等方面提升易用性

操作指南
快速入门

TBDS提供了一整套安全创新的企业级大数据集群服务，用户可轻松使用HDFS、Spark、Flink、ES、StarRocks等存算分析组件。
TBDS由TM（TBDS Manager）管控平台和TE（TBDS Engine）组件两部分构成，通过操作TM管控平台，用户可以快速进行集群安装部署、运维管控，并高效访问TE组件服务。通过使用在集群中的各种存算组件，运行大数据作业任务，进而满足数据应用需求。

在正式使用TBDS之前，需要先完成管控平台的部署安装，TM是创建和管理集群的基础。完成管控平台安装后，可以基于TM对业务集群进行配置管理了，TBDS的基本使用流程如下：

1. **主机管理，**集群部署依赖主机环境，用户需要先完成主机规划和配置，才能创建集群；同样在卸载集群后，建议删除主机释放资源。

2. 创建集群，用户可以指定集群类型用于数据存储计算任务，TBDS提供了多种集群类型，用户可自定义其节点类型、资源规格、要安装的组件等。
   集群创建过程需按照公共集群--经典集群的依赖关系依次进行部署。

3. 访问集群，提供经典集群访问；用户可通过TM控制台进行集群服务资源管理、访问组件WebUI，可通过客户端进行节点组件操作。

4. 提交作业，TBDS为用户提供程序执行平台，程序由用户自身开发，TBDS负责程序的提交、执行和监控。

5. 管理集群，TBDS Manager作为用户企业级的大数据集群的统一管理平台，可帮助用户快速掌握服务及主机健康状态，通过图形化的指标监控及定制及时的获取系统的关键信息，根据实际业务的性能需求修改服务属性的配置，对集群、服务、角色实例等实现启停、配置下发等操作。

6. 卸载集群，如果当前存算集群不再承载业务，或需要释放资源用于重新搭建，可以选择卸载集群。

主机管理

TBDS提供统一的主机管理功能，集中维护平台所有主机信息。通过集成化的查看和管理视图，用户可高效完成主机规划和配置：主机管理

创建集群

部署公共集群

公共集群为整个平台提供了统一且强大的公共基础能力，包括安全服务和元数据服务，是创建其他业务集群的基础，一个租户（主账号）仅包含一个公共集群。
步骤1：通过管理员用户首次登录 TBDS Manager后（参考：用户登录），可以看到公共集群部署引导页面。
步骤2：选择部署方式：支持物理机部署方式；高级配置支持内置KDC和企业KDC模式。
步骤3：设置集群密码，元数据库信息：公共集群服务依赖的数据库存储其元数据和配置等信息，需要确保输入账号具有创建和管理数据库的权限。
步骤4：确认信息后，点击"开始部署"，等待部署完成后，查看公共集群详情是否正常，若正常则完成部署。
详细操作参考公共集群：安装公共集群

创建经典集群

采用物理机部署模式；按照组件类型进一步提供四种集群类型，包括：
Hadoop集群，大数据分布式系统基础框架，适用于离线/实时分析等各类大数据场景。包含了经典的存算分析组件，如Hive、Spark、Flink、Trino组件。
StarRocks集群，极速统一的OLAP分析数据库，适用多维分析，实时分析，高并发等场景。
Elasticsearch集群，大数据分布式搜索引擎，适用于对海量数据进行存储、全文搜索、日志分析等场景。
Kafka集群，高吞吐消息处理系统，适用于异步消息和流式数据的接收和分发场景。

创建经典集群基本步骤参考：部署集群

访问集群

访问TBDS Manager管控平台

TBDS提供了租户端和运营端入口，两个入口的子域名前缀不同。TBDS Manager管控平台通过租户端进入（注意：租户端子域名以tbds.xxx开头，登录前请确保打开Manager页面浏览器所在的客户端机器与该集群网络互通）：

1. 登录方式支持主账号、子账号登录（主账号是内置账号，可以创建多个子账号进行资源共享和逻辑隔离（基于资源组），主账号之间进行物理资源隔离），第一次登录通过主账号方式，进入后通过用户管理自定义创建子账号。
2. 子账号登录需要增加输入主账号的UIN信息（主账号登录后，查看账号信息中的账号ID，即为UIN）。

详细操作参考：用户登录

使用客户端

用户可以通过客户端访问集群内节点、组件服务，TBDS提供了客户端管理服务，方便进行客户端连接配置：

1. 登录 TBDS Manager管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 单击左侧客户端管理，即可进入客户端管理页。
3. 下载安装包：可手动下载集群内客户端安装包，支持仅下载配置文件。点击下载之后左上角会有下载进度，下载完毕之后点击安装客户端。
4. 安装客户端：输入节点IP、端口、用户名、密码等信息，安装远程客户端；注意是覆盖式安装。节点类型支持X86/ARM，安装前需进行节点初始化操作。
5. 安装后即可以访问集群节点的方式进行登录使用。
   功能详细参考：客户端管理

   > 注意：
   >
   > 客户端安装注意：
   >
   > i. 下载节点和安装节点要保证SSH通信正常
   > ii. 保证下载节点已经安装Hadoop、hive、hbase、zookeeper、kafka、spark、Sqoop
   > iii. 下载节点有cluster.info文件、/etc/krb5.conf、/var/krb5kdc/emr.keytab、/etc/chrony.conf、/usr/local/JDK

访问集群内节点

TBDS集群由多个主机节点组成，根据节点上部署的组件角色的不同，集群内的节点类型可以分为管理节点（Master）和数据节点（Core、Task），为了更好进行集群节点运维管理，往往需要登录访问不同类型的节点。TBDS提供客户端管理功能，可以支持用户通过本地客户端快速访问经典集群某个节点服务器；下面以 PuTTY 为例，介绍本地电脑如何使用远程登录软件通过密钥或密码方式(SSH方式)登录 TBDS 经典集群：

1. 登录TM控制台，点击进入需要操作的经典集群，在“集群资源-节点状态-节点列表”可查看集群节点资源信息，进程部署状态；
2. 复制需要登录操作的节点IP地址，打开 PuTTY 客户端，输入IP/Port，选择【SSH】，输入节点管理员账号密码进行登录。
   详细操作参考：登录集群

访问集群内组件 Web UI

TBDS集群默认托管了标准组件的Web UI界面，用户可以通过这些Web UI界面查看组件相关信息。 WebUI 默认登录

账号密码为创建集群时设置的账号密码。

针对经典集群组件访问WebUI步骤：

1. 在TBDS Manager首页 > 经典集群 > 集群服务,点击对应组件下方“WebUI地址”；可以看到访问地址链接（由于组件采用高可用模式，因此会有多个地址链接）
2. 点击访问地址链接将进行页面跳转（注意自动跳转是通过内网IP地址链接，若部署的集群采用公网和内网IP分离，则可能根据组网模式需要手动替换为当前客户端能访问的公网IP）。

针对公共集群组件访问WebUI步骤：

1. 在TBDS Manager首页 > 公共集群>服务信息，点击进入需要操作的组件，进入组件管理页，点击“查看信息-查看WebUI”；
2. 点击访问地址链接将进行页面跳转（注意自动跳转是通过内网IP地址链接，若部署的集群采用公网和内网IP分离，则可能根据组网模式需要手动替换为当前客户端能访问的公网IP）；
3. 注意Ranger组件的登录账号特殊，只能通过root登录，密码为创建集群时设置的密码。

详细参考：WebUI 访问管理

提交作业

经典集群提交作业

您可以通过经典集群创建并提交作业；本文以提交 Spark 任务为例，操作如下：

1. 客户端使用 SSH 登录并连接集群。
2. 切换到操作用户（注意需要提前获取该用户keytab），并进入 Spark 安装目录/usr/local/service/spark，进行Kerberos认证（集群为simple认证跳过）：

```
[test@10 ~]$ cd /usr/local/service/spark
[test@10 spark]$ klist -kt {keytab}
[test@10 spark]$ kinit -kt {keytab} {principal}
```

3. 如下示例代码，演示了提交任务并输出运行结果（更多组件示例代码请参考组件开发指南）：

```
# 2. spark-submit提交任务
# 2.1 计算Pi，client模式，结果将输出到控制台
[test@10 spark]$ bin/spark-submit --master yarn --deploy-mode client --class org.apache.spark.examp
les.SparkPi examples/jars/spark-examples_2.12-3.2.2.jar
```

```
Pi is roughly 3.142435712178561

# 2.2 WordCount，cluster模式，输出部分日志到控制台，结果可查看yarn日志
[test@10 spark]$ vim /tmp/wordcount.txt
hello world
hello spark
hello hadoop
scala java
java kyuubi
[test@10 spark]$ /usr/local/service/hadoop/bin/hdfs dfs -put /tmp/wordcount.txt /tmp/
[test@10 spark]$ bin/spark-submit --master yarn --deploy-mode cluster --class org.apache.spark.exam
ples.JavaWordCount examples/jars/spark-examples_2.12-3.2.2.jar hdfs：///tmp/wordcount.txt
```

4. 提交作业后可在TM页面，单击目标集群所在行的集群服务；单击 YARN UI 所在行的 WebUI 链接，登录认证后即可进入YARN UI 页面；单击目标作业的 ID，可以查看作业运行的详情。

管理集群

通过TM管控平台可以高效进行集群管理、数据管理、资源管理和用户权限管理。

公共集群管理

确保公共集群的高可用性对于整个平台的稳定运行至关重要。在日常运维工作中，运维团队会定期对集群执行健康检查和负载监控，以评估其性能并决定是否需要进行资源的扩展或缩减。

操作	相关文档
对公共集群组件管理	新增组件卸载组件
资源使用和趋势分析	负载检查
对高负载的服务进行资源调配	资源调配
对异常服务进行重启	服务重启

经典集群管理

经典集群管理操作和相关文档如下表：

操作	描述	相关文档
集群列表	通过列表页可执行集群部署、启停、卸载、软件配置、补丁安装等操作	[部署集群](/document/product/326814/194503)[集群启停](/document/product/326814/587645)[软件配置](/document/product/326814/600838)[补丁安装](/document/product/326814/210304)[集群卸载](/document/product/326814/806472)
集群资源	包括集群信息概览，节点状态监控、扩缩容	[集群扩容](/document/product/326814/374258)[集群缩容](/document/product/326814/975616)[节点状态](/document/product/326814/805665)
集群服务管理	包括新增卸载组件	新增组件卸载组件
	服务健康状态监控、服务启停	服务健康状态启停服务重启服务
	配置管理：关键配置参数的修改，可以根据实际需要以集群维度、节点维度、配置组维度对服务的配置进行修改；包括配置更新、配置状态同步、配置回滚、分组配置管理	配置管理
	关键服务操作：如 HDFS 、YARN、Impala、HBase、Hive服务管理	YARN组件服务HDFS组件服务Impala组件服务Hive作业查询HBase数据表分析
	支持Trino、Impala、HBase等服务的在CPU/内存/IO资源方面的策略使用限制	静态服务池
集群监控	平台运行日志采集、搜索	操作日志日志搜索
	集群概览，集群事件监控，告警配置查询，巡检	集群概览配置告警集群事件集群巡检
高级配置管理	针对 ES 集群，SR 集群，支持高级配置管理	SR集群高级配置 ES 集群高级配置

数据管理

数据管理基于统一元数据为用户提供了统一视图，用户可以集中管理和查看结构化的数据库表，以及半结构化或非结构化的文件。TBDS为经典集群上的 Hive/Iceberg（共享一个 Hive Metastore）和 HDFS 提供了集中的数据管理入口。当经典集群安装后，会自动将自身注册到数据管理服务中。

1. 库表管理：按照库表层级结构展示库表信息，点击数据目录、数据库和数据表，右侧会展示对应信息；参考：库表管理。
2. 点击左侧文件夹，右侧将展示文件夹下的文件和子文件夹，可对文件夹进行复制路径、复制、重命名、移动、授权、回收、删除等操作。也可对文件进行文本或二进制预览；参考：文件管理。

资源管理

资源组是对TBDS平台上Hadoop资源进行逻辑划分的单元，由存储和计算资源组成。通过资源管理可以高效实现团队间资源隔离（资源组下YARN 队列可以通过配置实现资源的共享或独占）和用户授权。
资源组包括如下两类：

1. 默认资源组：平台部署完成后，会提供YARN 默认资源组（关联了YARN 集群的 default 队列）；参考：默认资源组。
2. 自定义资源组：
   YARN资源类型（将 YARN 的队列绑定到该资源组下实现计算配额分配，将经典集群的HDFS存储路径绑定到该资源组下实现存储配额的分配）；参考：创建 YARN 资源组、编辑 YARN 资源组、删除 YARN 资源组。

   > 注意：
   >
   > 1、一旦用户被授予资源组权限，他们将自动获得该资源组内所有资源的访问权限，且无法进行更细粒度的权限控制。
   >
   > 2、如果资源组中包含了 YARN 队列，那么其所包含的存储资源必须与 YARN 队列位于同一集群。

用户权限

TBDS 为用户提供了多级租户管理机制，包括支持物理隔离的一级租户和逻辑隔离的二级和以下租户。
为此TM提供了三类用户的管理：主账号（租户管理员，默认内置），子账号（租户成员，自定义创建），保留用户（默认内置，如root、hadoop），用户密码分为：TM控制台登录密码、集群密码（集群组件的访问密码）；不同用户的管理和使用方式可参考：用户权限

平台管理

模块功能	描述	相关文档
审计中心	审计日志管理，通过审计中心对操作日志、组件访问日志进行管理，支持审计日志的检索导出、存储生命周期和日志转储	审计中心

模块功能	描述	相关文档
安全配置	常用安全配置包括：脱敏、加密，KMS配置，开启审计等	动态脱敏HDFS 透明加密Ranger授权开启组件审计开启云鼎 KMS
升级中心	版本内补丁升级操作	升级中心

卸载集群（可选）

集群卸载功能支持用户通过平台界面化操作解除服务集群和TBDS Manager管理平台间的管理关系，平台将停止集群中的组件服务并清理集群TBDS Manager平台中的元数据信息。
集群卸载流程不可恢复，请谨慎操作。
经典集群卸载：集群卸载。

> 注意：
>
> 公共集群的卸载属于高危操作，不支持用户直接卸载；
>
> 集群卸载操作将不会删除集群中存储的数据，业务数据将被保留。若需要进一步清理需在主机管理中进行重置主机操作。

操作约束与限制

在使用TBDS 大数据平台之前，请您详细阅读并了解以下使限制：

1. 新建集群为保证集群网络安全和稳定，集群将放置在同一网络环境中，请勿随意变更已有集群或节点的网络，避免造成集群络不互通。
2. 请根据业务需要提前规划节点的存储空间，并及时扩充存储节点，避免因存储空间不足造成数据及节点运行风险。
3. 使用TBDS 大数据平台时，请避免直接操作主机或容器，如关机、重启、网络切换、安全组调整等，以防集群异常。推荐在 TBDS 大数据管理平台内完成所有必要的集群维护操作。
4. 在创建集群时，TBDS 大数据平台提供了满足通用场景的组件初始化参数，在使用组件服务前，建议您检查并微调相关组件参数以确保匹配您的业务场景。如需相关组件初始化指南，可以联系技术支持人员获取。
5. 请您妥善保管TBDS 大数据平台集群的主机登录密码。

禁止操作

在使用及维护 TBDS 大数据平台集群时，一些非预期的操作可能会导致集群不可用或不稳定，在控制台执行部分操作前会有相应的风险提示，本节列举了些禁止及高危操作：

操作	操作风险
使用开源版本/低版本TBDS 的客户端/未经确认的第三方工具进行组件操作	使用非配套版本的客户端进行操作容易导致：集群隐蔽性故障、元数据不一致、数据丢失与损坏等
在 TBDS 集群节点中修改节点内网 IP	节点通信异常、集群不可用
在集群运行中修改 TBDS 集群节点的访问白名单	节点通信异常、组件服务不可用
删除节点上已有进程/应用程序/文件	集群/组件服务不可用
删除或者修改/etc 目录下的 hosts 文件	集群关联不到节点上的服务，导致服务异常
删除或者修改 HDFS 元数据文件 edit log	导致 HDFS 集群不可用
手动修改 Hive 元数据库的数据	Hive 数据解析错误，服务异常
删除 ZooKeeper 相关数据目录	相关依赖组件无法运行

高危操作

操作	操作风险	操作建议

对 TBDS 集群节点进行关机、重启	重启、关机导致服务不可用	确认操作必要性，并详细评估相关操作风险
对 TBDS 集群节点挂载磁盘	TBDS 集群节点无法识别和初始化，导致磁盘不可用	建议在技术人员指导下进行
对 TBDS 集群节点卸载磁盘	会导致数据丢失或集群不可用	建议在技术人员指导下进行
直接在 TBDS 集群节点上修改组件配置文件的参数	服务重启后，导致修改的参数被覆盖	通过 TBDS 大数据管理平台上修改参数配置，特殊情况请在技术支持人员指导下进行
删除或者修改/etc目录下的 resolv.conf 文件	集群关联不到节点上的服务，导致服务异常	确认操作必要性，并在技术指导下进行
修改 MetaDB 密码	TBDS 集群依赖 MetaDB 中配置的密码，修改后导致Hive/Ranger 等服务不可用	在TBDS 大数据管理平台同步修改配置，并在技术人员指导下进行
修改 MetaDB 浮动 IP	TBDS 集群依赖 MetaDB 中配置的 IP，修改后导致 Hive/Ranger等服务不可用	在TBDS 大数据管理平台同步修改配置，并在技术人员指导下进行
修改 MetaDB 安全组	导致 MetaDB 与集群通信受阻，Hive/Ranger 等服务不可用	在技术人员指导下进行

高可用说明

TE组件高可用：

故障场景	高可用说明	限制说明
HDFS namenode-active节点失效	Standby节点自动转换状态到active	自动切换存在分钟级恢复，原active节点恢复后, 状态为standby
HDFS namenode-standby节点失效	Standby节点失效后, active节点不受影响	原standby节点恢复后, 状态为standby
HDFS任意一台DataNode失效	DataNode失效后, 功能正常	DataNode恢复后, datanode重新加入集群
HDFS-任意一个journal node失效	使用quorum机制，任意节点失效功能不受影响	不受影响

Hive- HiveMetaStore/ HiveServer2/ HiveWebHCat 任意一个失效	采用ActiveActive模式，相关功能不受影响	相关组件不受影响
HBase- HMaster active节点失效	active节点失效后, standby节点转换状态到active, 功能正常	自动切换存在分钟级恢复，原active节点恢复后, 状态为standby
HBase- HMaster standby节点失效	Standby节点失效后, active节点不受影响	原standby节点恢复后, 状态为standby
HBase- HBaseThrift	采用ActiveActive模式，相关功能不受影响	不受影响
HBase-RegionServer其中一台失效	相关功能不受影响	不受影响
Trino/Presto-Coordinator/ Worker其中一台失效	采用ActiveActive模式，相关功能不受影响	相关组件不受影响
Impala-Daemon任意一个失效	Daemon无状态服务，任意一个impalad失效不影响功能	不受影响
Impala-catalog/ statestore	非关键服务无高可用	需要监控异常手工启动，不影响业务流程，
YARN-ResourceManager 任意一节点失效	active/standby模式，当active节点失效，standby节点转换状态到active, standby节点失效后, active节点不受影响	原active节点恢复后, 状态为standby；原standby节点恢复后, 状态为standby；恢复时间分钟级
YARN- NodeManager任意一节点失效	不受影响	不受影响
Kafka其中一个broker失效	一个broker上存在partition leader ，停止这个broker，这个partition所在的topic可以被正常使用.	不受影响

TM管控平台高可用：

故障场景	高可用说明	限制说明
POD故障-双活服务 tm-woodpecker-taskcenter,tm-darwintaskcenter,tm-woodpecker-ems,tm-grafana,tm-platform,logstash,tm-web	服务pod挂掉会自动拉起或者漂移恢复，过程中不影响服务的正常使用。	因资源节点问题导致不能拉起恢复但是至少保有一个实例可用，服务仍然可以正常使用。

POD故障-主备服务 tm-emrcc,tm-woodpecker-server,tm-woodpecker-cmdserver,tm-woodpecker-native	主备模式下，当主机pod挂掉后，首先备机升级为主机运行任务。服务pod挂掉会自动拉起或者漂移恢复，过程中不影响服务的正常使用。	因资源节点问题导致不能拉起恢复但是至少保有一个实例可用，服务仍然可以正常使用。
POD故障-agent类服务 woodpecker-agent,woodpecker-bootstrap,woodpecker-ems-agent,Filebeat	服务挂掉后能够自动拉起	可以自动恢复，影响分钟级
硬件故障-单节点	同一组件的两个pod不会部署在同一节点上；当任意worker节点挂了，此节点上对应的pod会迁移至随机的其他worker节点上	可以自动恢复，不影响服务正常使用。
组件更新	采用滚动更新，新建2个pod，完成后再依次销毁旧pod	不影响业务

网络丢包场景下：组件目前机制在非断网状态下不会进行主备切换的，但业务可能会受损，建议由上层业务平台进行监控。

规格限制

TBDS建议规格限制如下：

类型	指标名称	规格	说明
账号/租户	最大主账号数	20	
	单主账号下的最大用户数	5000	
	单主账号下的最大用户组数	300	
	用户关联的最大用户组数	50	
集群	系统最大节点数	5000	通用X86/ARM服务器
HDFS	单NameNode最大文件数	3亿	
	NameNode的最大连接数	3000	
	单DataNode最大block数	500万	
	单个DataNode磁盘最多block数	50万	
	单个目录下最多文件目录数	100万	

	文件路径最大长度	8000	
HBase	单个RegionServer实例的Region数量	2000	单RS实例支持的最大Region数
	单个RegionServer支持的活跃Region数量	200	单RS实例支持的最大活跃Region数
Hive	最大分区数量	1亿	单个Hive服务建议最大分区个数
	单HiveServer最大并发数	500	单个HiveServer实例支持的最大并发数
Elasticsearch	单Elasticsearch实例最大内存配置	31GB	
	单shard支持的记录数	4亿	
ZooKeeper	最大znode数	400000	
	单个znode大小	4M	
Ranger	Ranger策略数	100万	

主机管理
查看主机

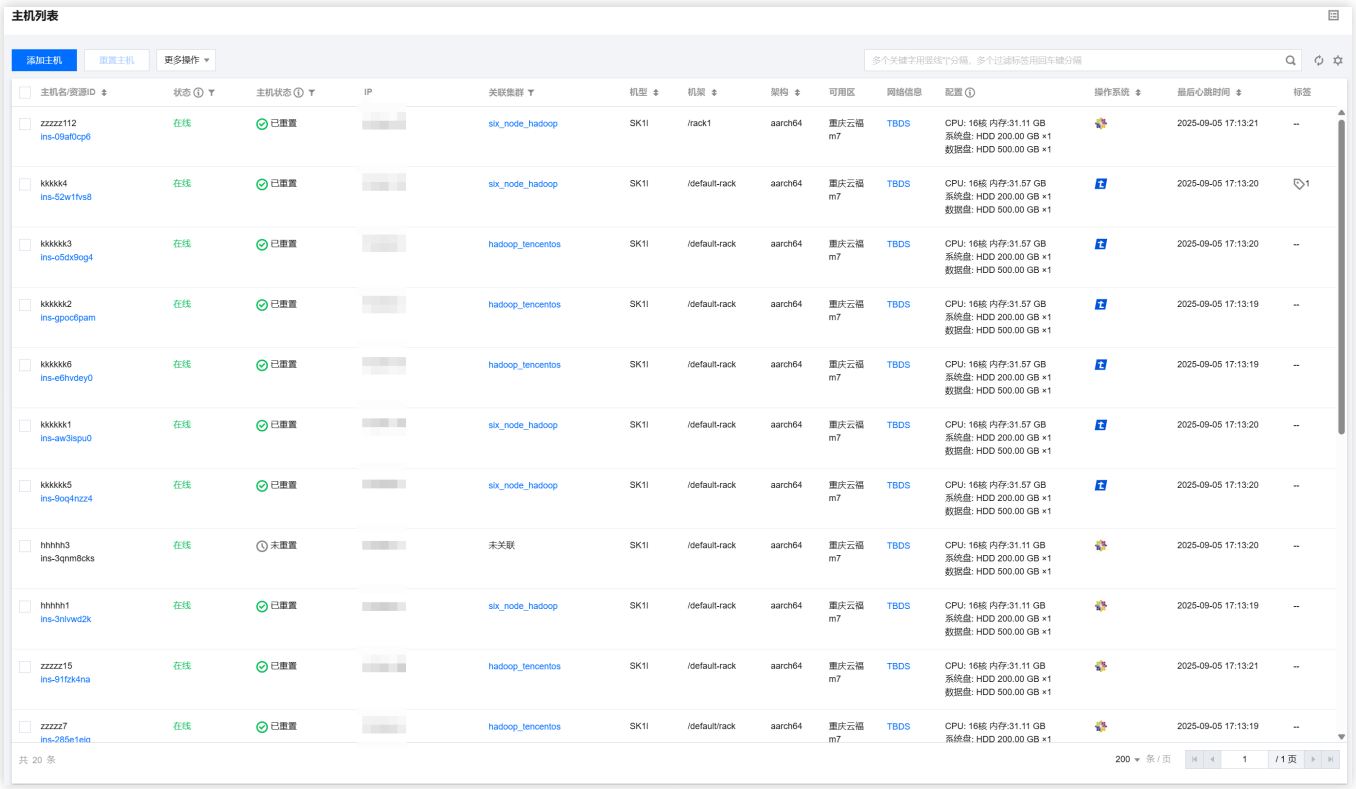
功能介绍

TBDS提供统一的主机管理功能，集中维护系统所有主机信息。通过集成化的查看和管理视图，帮助用户高效完成主机规划和配置，简化运维操作流程。

操作步骤

主机列表

1. 用户登录 TBDS，进入 TBDS Manager首页 > 主机列表，该列表展示当前租户下所有主机信息，可用于集群创建。



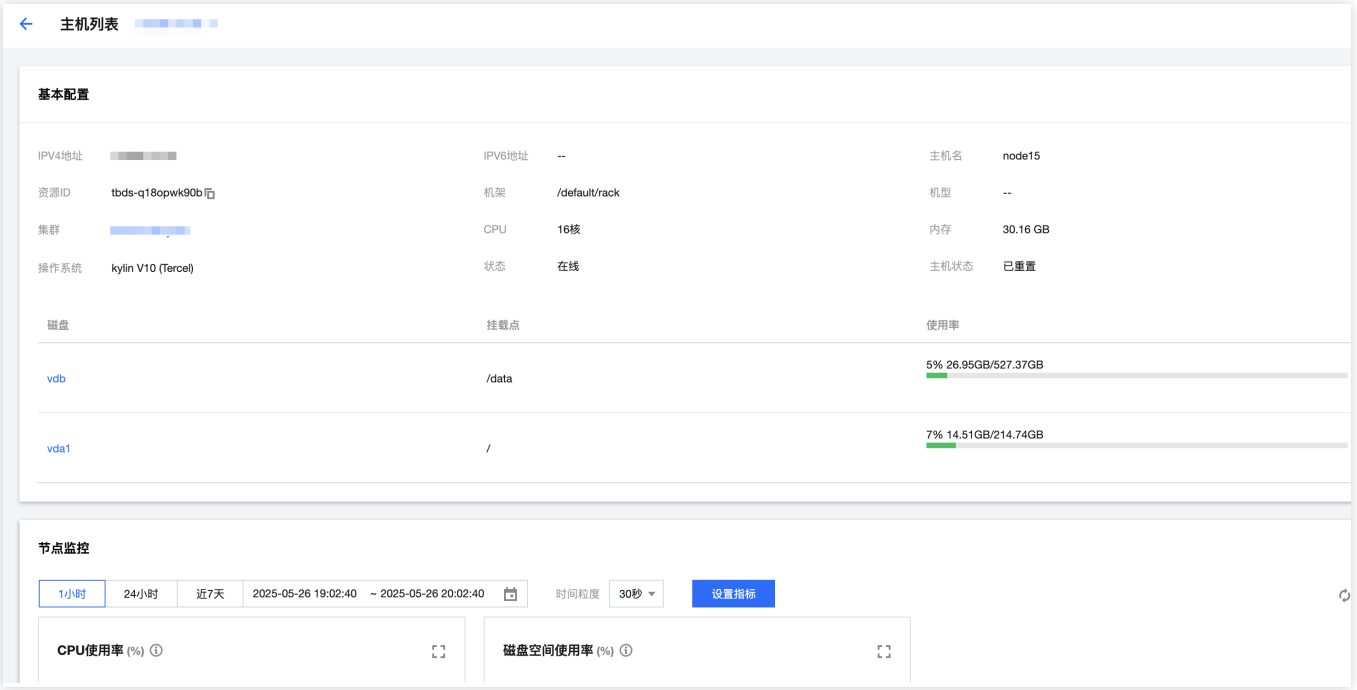
主机信息说明如下：

信息	详情
主机名/ID	主机名用户添加时可指定，不指定则会自动生成，主机添加完成后会为主机生成为一个ID标识

信息	详情
状态	主机在线状态，若5分钟内未收到主机心跳信息，即视为主机离线
主机状态	主机的重置状态，只有重置后的主机方能用于集群创建，主机中的重置包括磁盘挂载、参数设置等
IP	支持 IPv4 和 IPv6
关联集群	如果该主机已经用于集群，则会显示集群名称，否则显示为“未关联”
机型	支持用户自定义，方便同类配置的机器进行管理
机架	默认为/default，支持自定义，需要满足 Unix路径的命名规范
架构	自动识别，比如 x86
可用区	购买的主机所在的可用区
网络信息	主机的网络信息，点击可跳转至私有化网络查看详情
配置	自动识别主机的配置信息，在配置信息中，若磁盘条目显示为灰色，则表明该磁盘尚未挂载
操作系统	自动识别，比如 CentOS 7.9
标签	支持为主机设置标签和按照标签筛选

主机详情

1. 用户登录 TBDS，进入 TBDS Manager首页 > 主机列表，点击主机ID 查看主机详情，可查看主机的基本配置和监控信息。



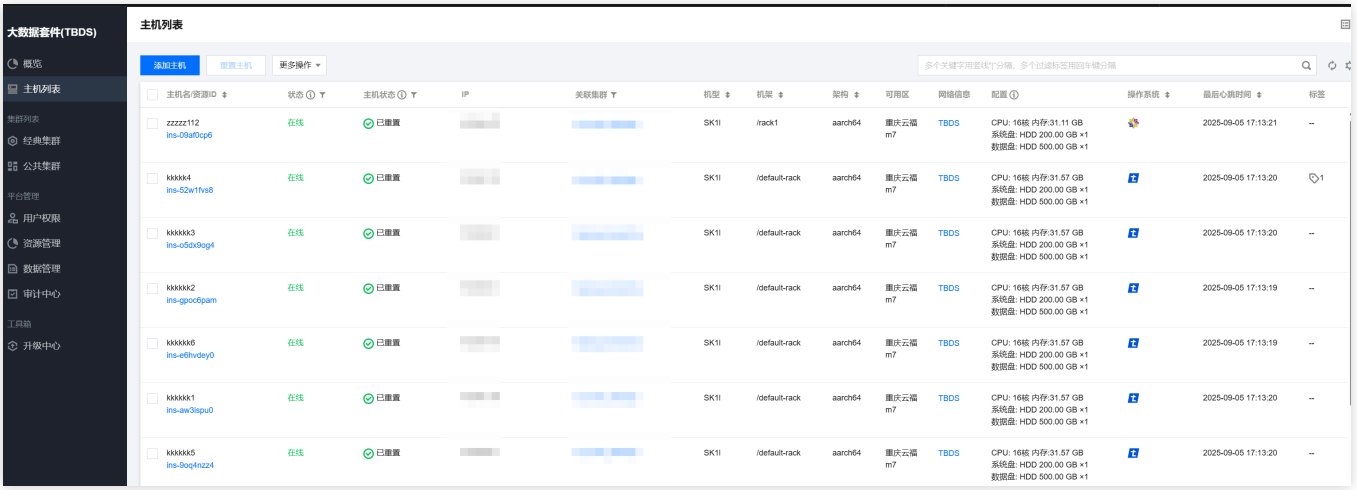
添加主机

操作场景

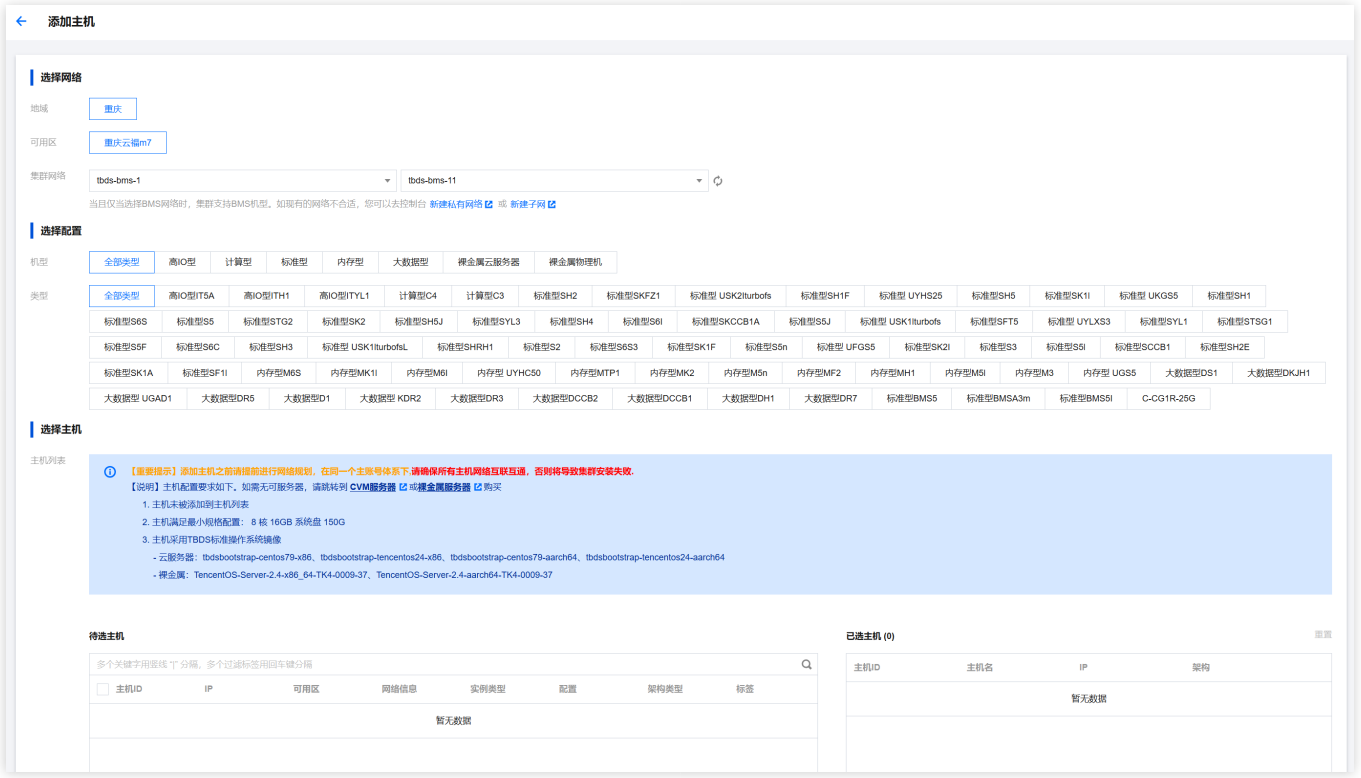
TBDS经典集群和公共集群（主机形态）的部署依赖于主机环境。在安装集群前，用户需先在主机管理界面完成主机的添加和配置。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager首页 > 主机列表，点击“添加主机”按钮。



2. 在添加主机的页面：
   1）选择合适网络和子网，可直接跳转到网络配置页新建网络和子网：VPC网络配置，管理子网。
   2）筛选主机配置类型，具体的机型类型规格可参考：CVM主机规格。
   3）在筛选条件下待选主机会自动展示在列表中，若待选列表不满足要求，也可以跳转至服务器购买页进行购买后刷新。
   4）选择需要的主机确认后，即可进入主机列表查看和管理。



> 说明：
>
> - 需要确保所有主机的网络互通互联，否则可能导致集群安装失败。
> - 若存在主机名重复、主机名不满足格式要求，添加主机会报错。
> - 若主机满足最小规格配置： 8 核 32GB 系统盘 150G，或者未采用TBDS标准操作系统镜像，待选主机列表中不会展现此主机。

后续步骤

1. 用户可对已添加的主机进行重置操作。重置完成后，这些主机可用于创建经典集群或进行集群扩容。

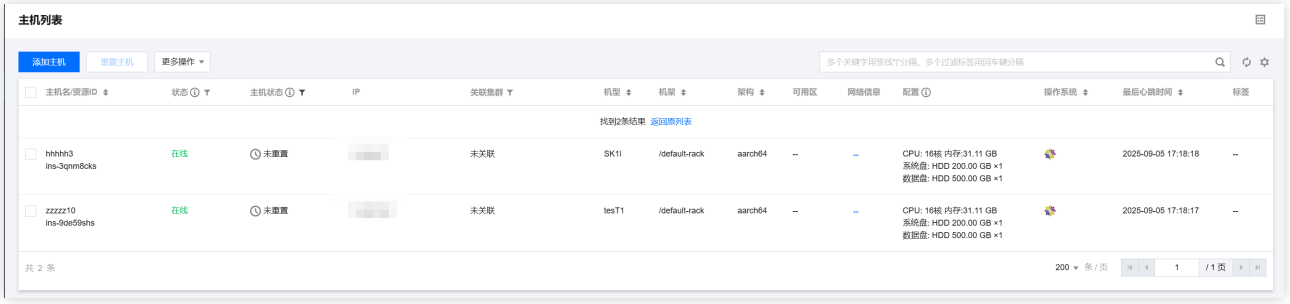
重置主机

操作场景

主机用于集群部署之前需要完成初始化，TBDS提供的主机重置功能可一键可视化完成集群部署前的初始化操作（包括操作系统参数配置、磁盘挂载及安装包准备等），该功能仅适用于状态为"未关联"集群的主机。

操作步骤

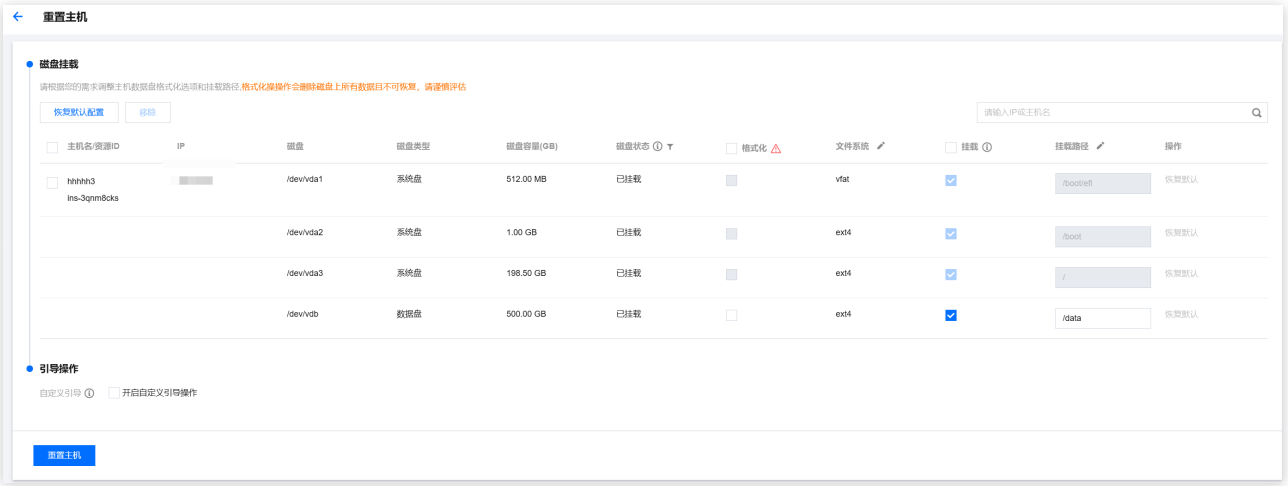
1. 以管理员用户登录 TBDS，进入 TBDS Manager首页 > 主机列表，按照"未重置"对主机进行筛选。



2. 在主机列表中选择目标主机并点击【重置主机】按钮，即可进入重置配置页面，支持对主机机型、磁盘挂载及引导配置等关键参数进行自定义设置。

> 说明：
>
> 磁盘挂载：支持对主机数据盘进行格式化和挂载操作。
>
> 注意事项：
>
> - 格式化操作将清空磁盘所有数据，请谨慎选择。
> - 未格式化的磁盘无法挂载，需先完成格式化。
> - 挂载路径建议：遵循 /data、/data1 等标准化命名规范。
> - 卸载规则：若磁盘已挂载但在页面取消勾选，系统将自动卸载该磁盘。
>   引导操作：支持在磁盘挂载前/后执行用户自定义的 Shell 脚本，用于完成以下操作：
> - 操作系统参数配置
> - 挂载前后的环境初始化或清理任务

i. 磁盘挂载



ii. 引导操作

3. 确认后点击【重置主机】，然后在【主机列表】-> 【任务中心】查看执行进度。

修改主机

操作场景

若已添加的主机信息存在不符合预期的情况，可通过以下两种方式进行更新：

1. 重新添加主机（覆盖更新）

- 操作方式：对目标主机执行重新添加操作，新提交的信息将覆盖原有信息。
- 前提条件：目标主机未被重置
- 适用场景：需全面更新主机配置信息时

2. 批量修改主机属性

- 操作路径：在主机列表中选择目标主机，执行修改操作。
- 可修改属性：
  - 主机名称
  - 硬件机型
  - 机架位置
  - 资源标签
- 适用场景：局部信息调整时
  本章将重点阐述第二种修改方式的详细操作流程。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager首页 > 主机列表，按照"未关联"对主机进行筛选。

2. 勾选目标主机后，点击【更多操作】即可修改主机属性（如主机名、机型和机架），依次说明如下：

   i. 修改主机名

      a. 表格修改：勾选主机后，通过【更多操作】→【修改主机名】进入编辑界面，输入新主机名并确认。

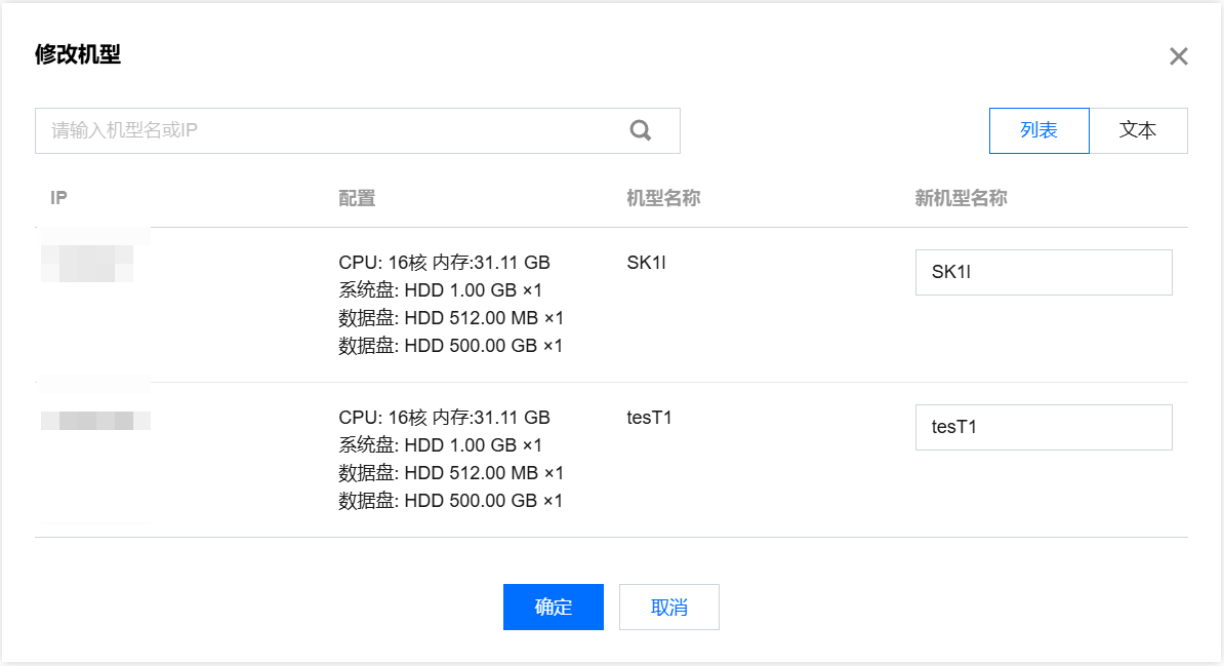
b. 批量修改（文本模式）：在修改主机名界面切换至"文本模式"，按格式批量编辑主机名后粘贴提交，批量操作时建议先校验格式。

> 说明：
>
> 每行填写一个主机的信息，格式为 IP,主机名（用英文逗号分隔）

ii. 修改机型

勾选主机后，通过【更多操作】→【修改机型】进入编辑界面，输入新机型名并确认。

> 说明：
>
> 说明: IP 和机型支持用逗号或制表符（Tab）分隔，每行一个 IP。下请按照"IP,机型"格式填写，机型总长度≤64，仅允许包含字母、数字、连字符-、下划线_、正斜杠/或空格。



iii. 修改机架

勾选主机后，通过【更多操作】→【分配机架】进入编辑界面，输入分配机架并确认。

> 说明：
>
> 机架名称要求类似于文件系统路径。例如，“/rack1”和“/sw/rack2”

3. 完成操作。

磁盘修复

功能介绍

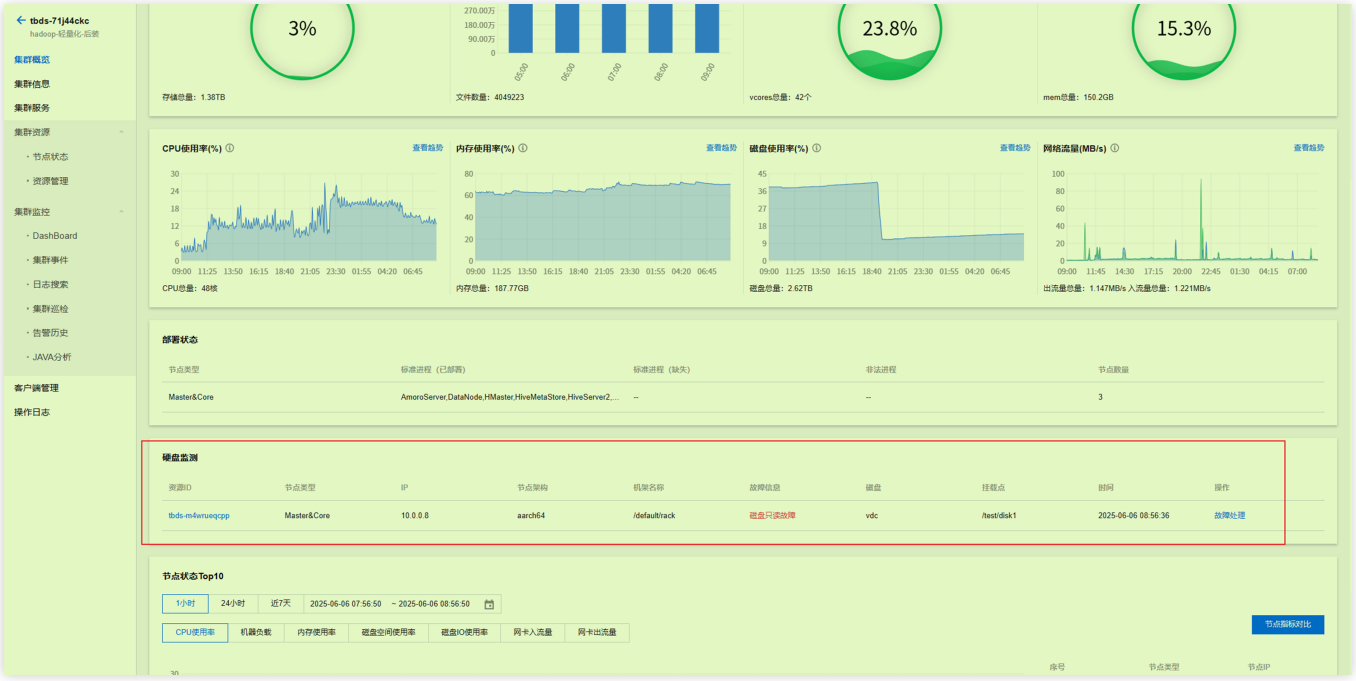
1. 跳转到对应节点，增加磁盘替换处理入口。
2. 支持为故障的磁盘选择新的未挂载磁盘进行替换。

前提条件

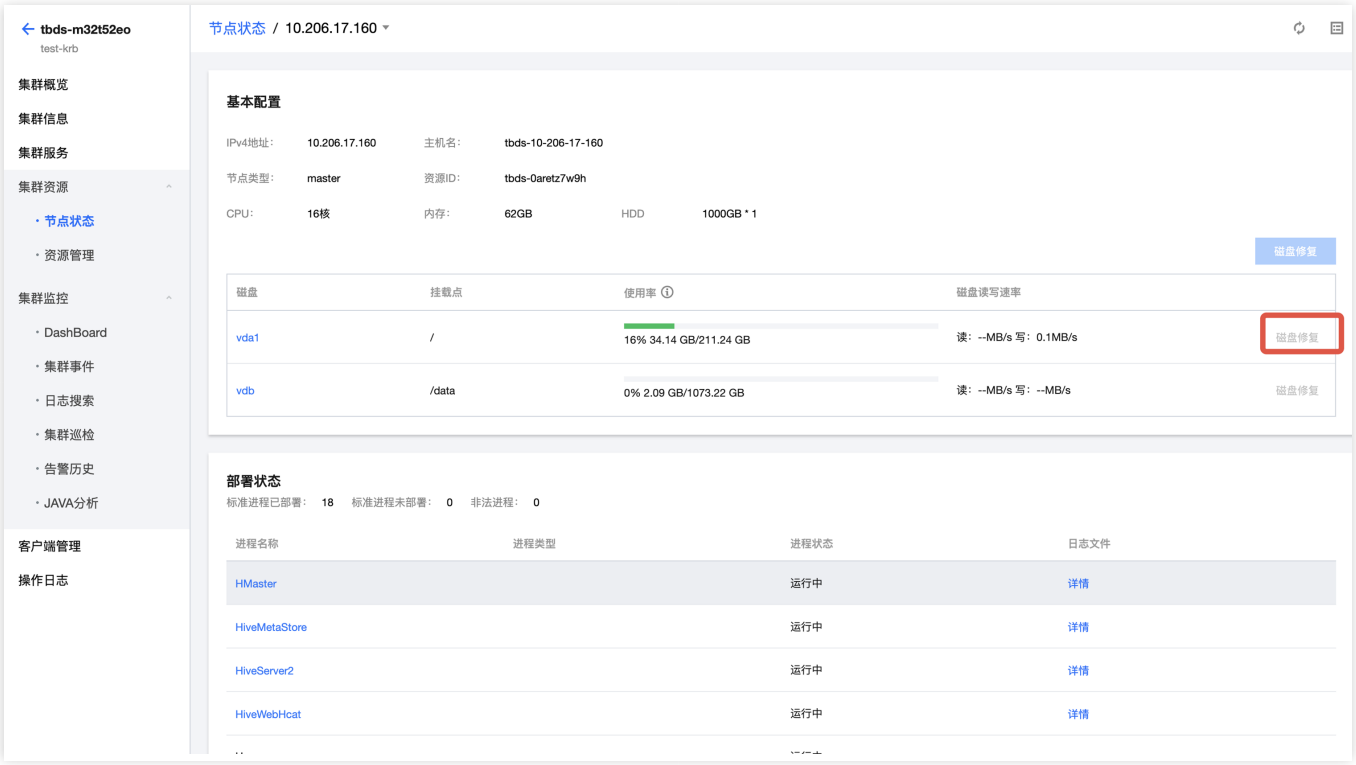
1. 坏盘条件：主机的某一磁盘只读或者某一磁盘掉盘。
2. 可用磁盘：主机上存在可用磁盘，即物理磁盘无分区且无挂载点的磁盘或分区磁盘无挂载点的磁盘。
3. 插拔磁盘：如果给机器插入了磁盘，或挂载了云硬盘，但是没设置挂载点，此时拔出或卸载磁盘，并不会视为掉盘；但是如果插入了磁盘并设置了挂载点，此时拔出或卸载磁盘，会视为掉盘。

操作步骤

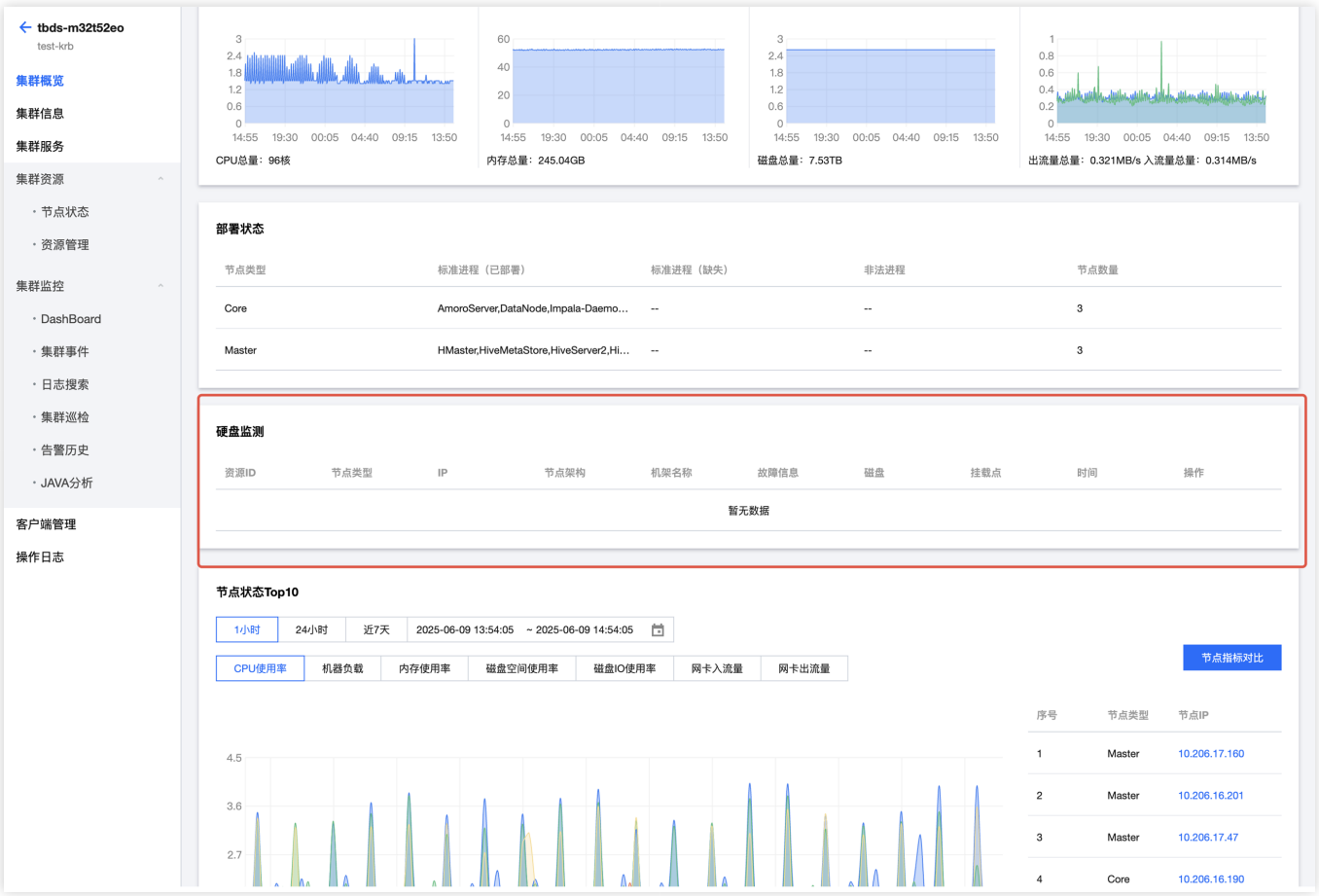
1. 登录 TBDS Manager 管理平台进集群概览页，查看硬盘检测，检查是否存在坏盘问题上报。



2. 点击坏盘的故障处理按钮，跳转到主机节点内部详情页，点击“磁盘修复”选择“可用磁盘”，可使用“可用磁盘”去替换原磁盘，新盘将挂载到原坏盘的挂载点，并格式化为原磁盘的文件系统。



3. 修复完成之后，查看集群概览页，观察探测的坏盘数据是否已被删除。



编辑标签

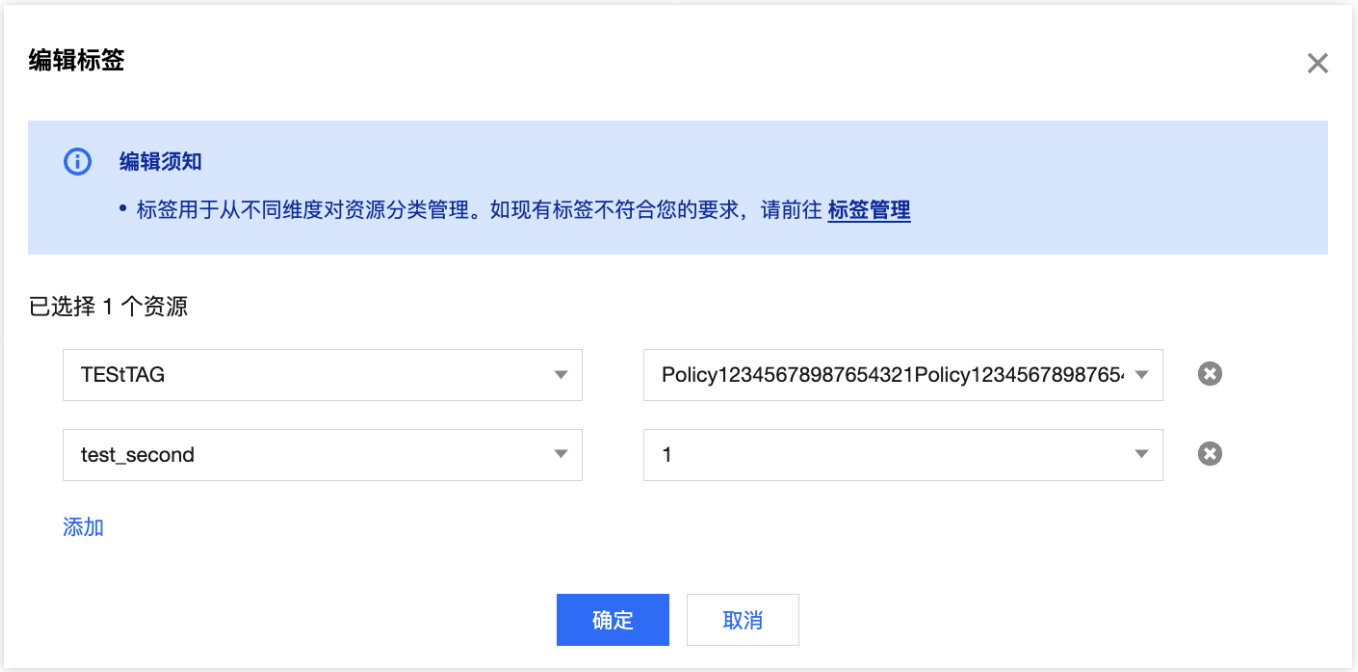
标签介绍详情请参考经典集群-编辑标签，创建标签请参考标签管理。

操作步骤

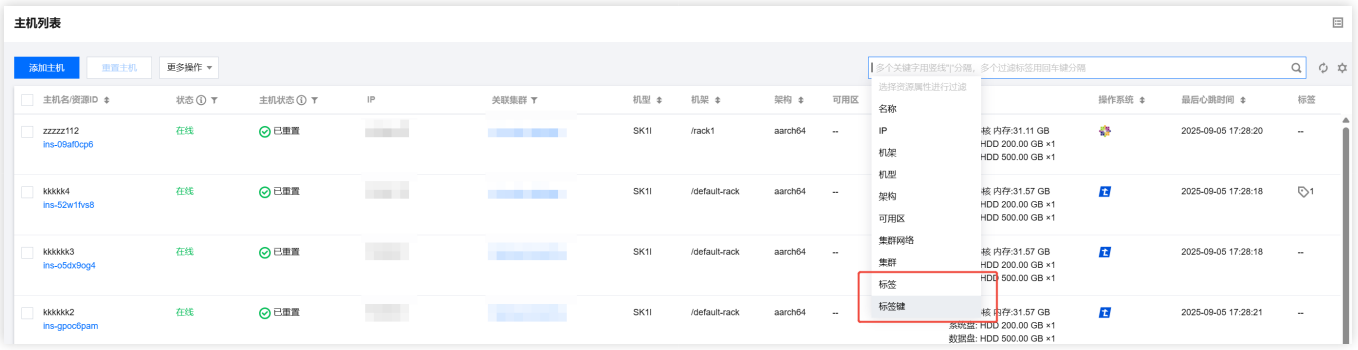
对集群编辑标签

1. 以管理员用户登录 TBDS，进入 TBDS Manager首页 > 主机列表，选中目标主机后点击【更多操作】->【编辑标签】。

2. 在弹出编辑标签对话框中为主机设置一个或多个标签键值，点击确定完成标签设置。



3. 为主机设置完成标签后，支持按照标签和标签键筛选。



删除主机

操作场景

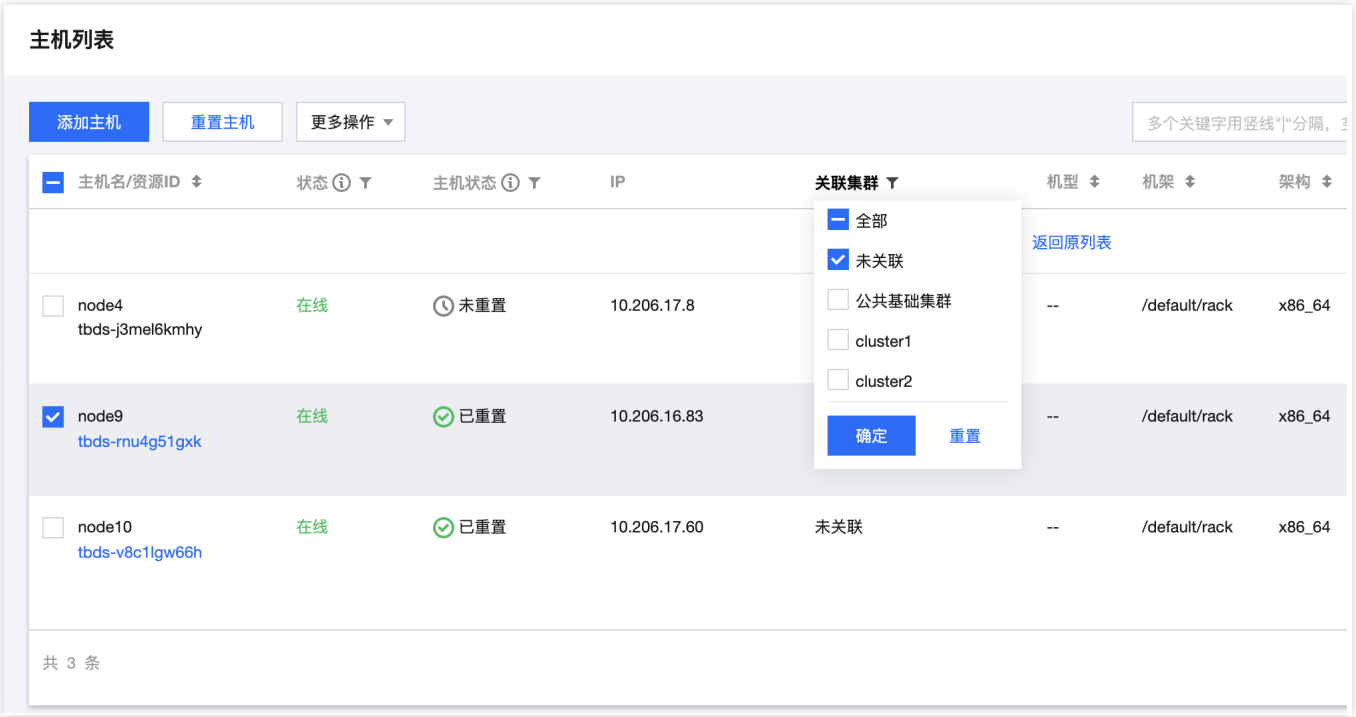
支持删除不再使用或添加有误的主机，删除后仍可重新添加。

前置条件

待删除主机未关联集群。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager首页 > 主机列表，按照“未关联”对主机进行筛选。



2. 在【更多操作】中选择【删除主机】并确认即可完成删除。

> 说明：
>
> 仅删除：仅从TM管控平台移除服务器，主机仍保留在云服务器/裸金属服务器中，会继续产生计费；

> 删除并销毁：从TM管控平台移除服务器并彻底销毁资源，销毁成功后停止计费。

集群管理
集群说明

> 说明：
>
> 本节全面概述TBDS 531版本系列全部功能，旨在帮助您全面了解产品并最大化利用产品优势。请注意，实际交付的软件功能可能与本手册描述略有差异，部分高级或特定功能可能需要您额外选购并获取相应授权许可后方能使用。若遇功能受限，请及时联系售前咨询选购详情并获取必要的支持。

公共集群
集群概述

公共集群是使用 TBDS 平台的必要条件，它为整个平台提供了统一且强大的公共基础能力，包括安全服务和元数据服务，包含服务说明如下：

类别	服务	服务说明	部署约束
安全	OpenLDAP	账号管理服务，存储集群访问账号	必选
	KRB5	Kerberos 安全认证服务	必选
	Ranger	安全管理服务，支持鉴权和审计等	可选，Hadoop/Kafka场景的经典集群和虚拟集群依赖此组件
网关	Knox	大数据服务的安全网关	可选，Hadoop/Kafka场景的经典集群和虚拟集群依赖此组件，同时公共集群的Ranger依赖此组件
	Kyuubi	大数据服务访问网关	可选，虚拟集群的依赖此组件（湖仓版），主机形态下支持。
湖管理	METASERVICE	统一元数据管理服务，支持多 Catalog 集中管理	可选，Hadoop/Kafka场景的经典集群和虚拟集群依赖此组件
	LUOSHU	数据湖优化服务，支持包括文件合并、文件清理等	可选，虚拟集群的依赖此组件（湖仓版）
其它	ZooKeeper	为分布式应用提供一致性服务，提供配置维护、名字服务、分布式同步、组服务等功能	可选，虚拟集群的依赖此组件（湖仓版），同时公共集群的Kyuubi依赖此组件

安装公共集群

前提条件

- 已完成 TBDS Manager 安装。

操作场景

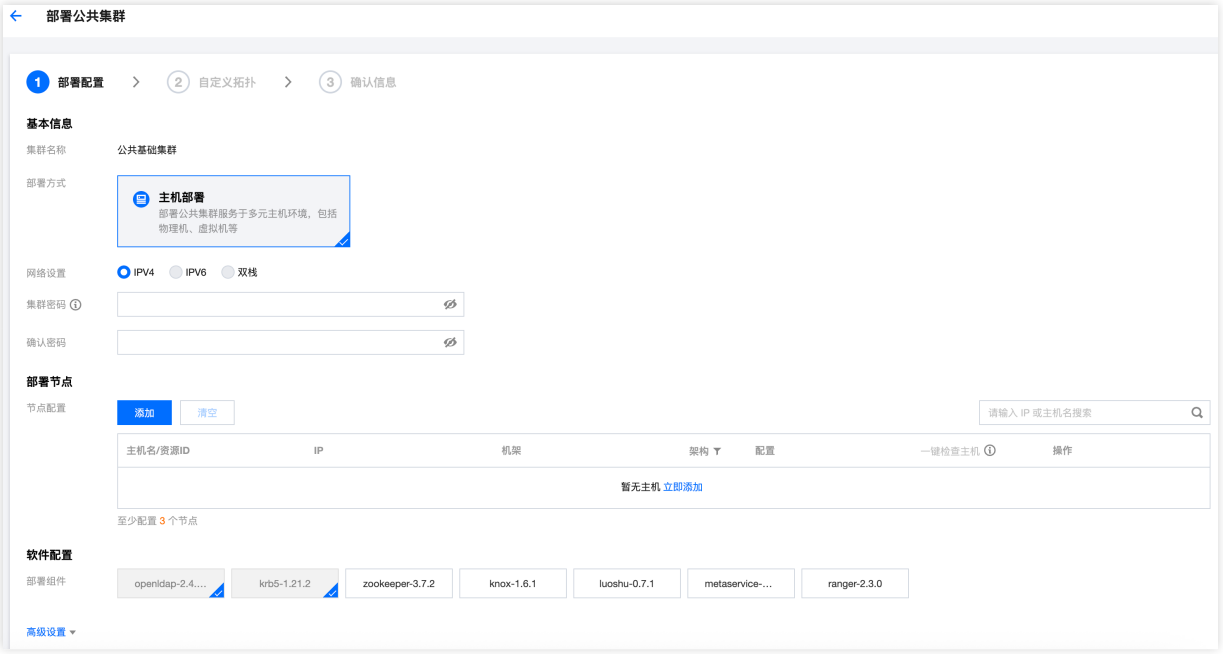
TBDS Manager 完成安装后需要先部署公共集群，一个主账号仅包含一个公共集群。在安装公共集群时，系统会默认启用内置的KDC服务。公共集群支持按照主机形态部署：

- 主机形态：将公共集群的服务组件部署在主机上，包括物理机、虚拟机等。

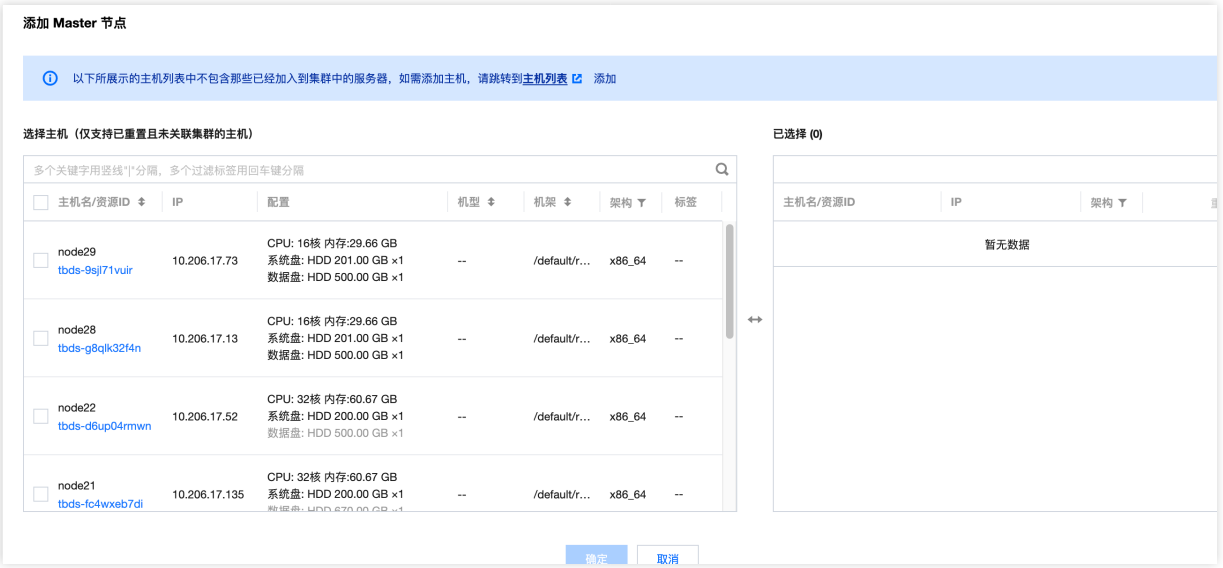
操作步骤

1. 安装完成 TBDS Manager 后，首次登录 TBDS 后可以看到如下公共集群部署引导页面。

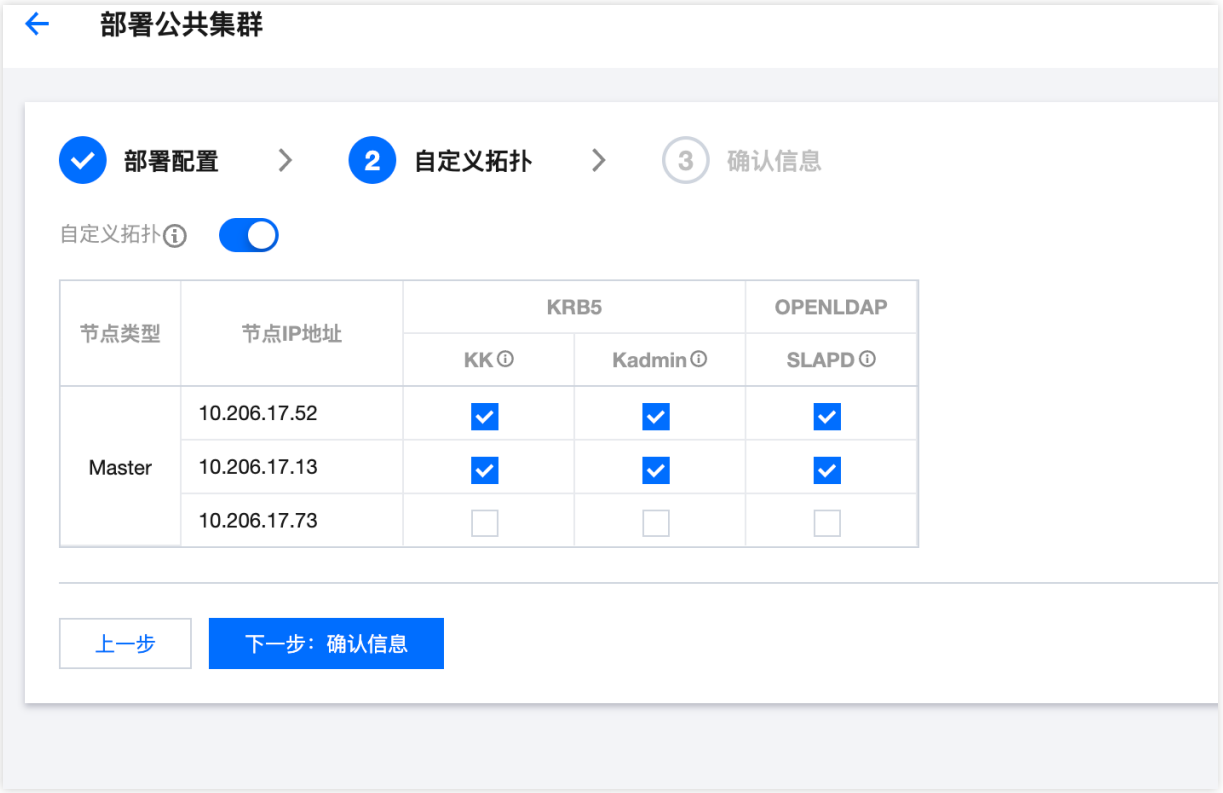
2. 单击部署公共集群，进入安装配置页面，完成信息填写，完成后点击下一步。



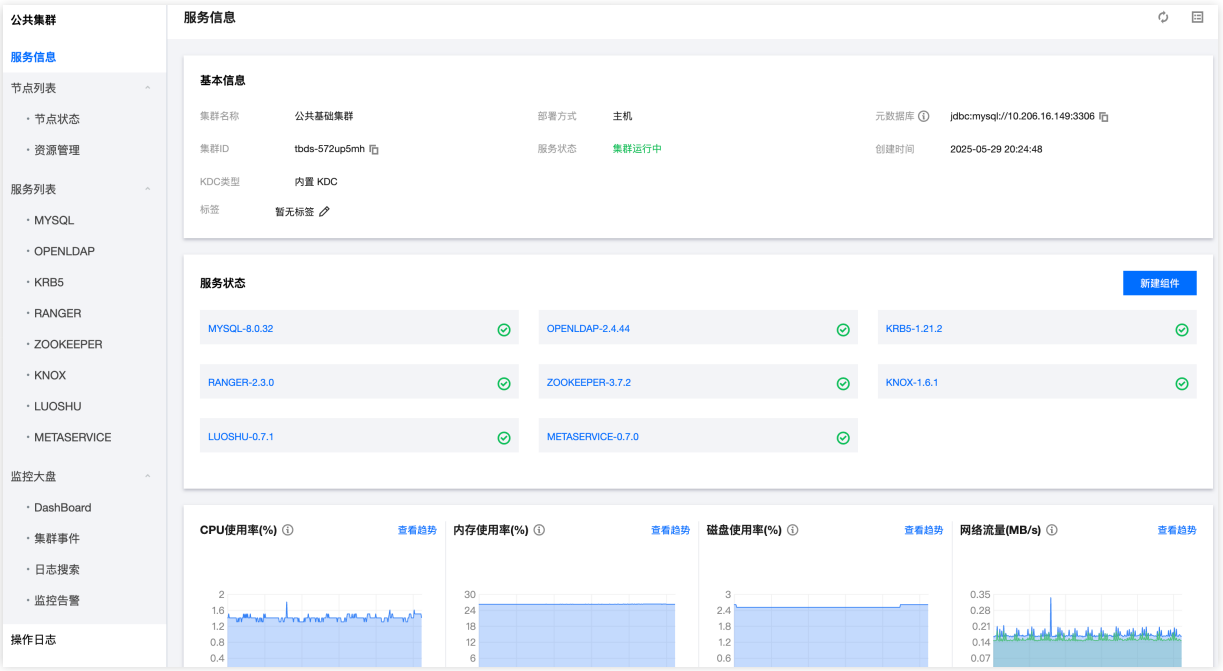
配置项说明如下：

信息	详情
网络设置	当前仅支持 IPv4
集群密码	密码用于设置公共集群服务Web UI的初始访问密码。
节点配置	从主机列表添加用于部署公共集群的节点，至少 3台，只有已重置的主机方可添加。参考 重置主机 
部署组件	按需勾选公共集群需要的组件，参考 集群概述 里面对于不同公共集群组件用途说明。
KDC类型	支持内置和企业 KDC，默认内置，如需要配置企业 KDC，参考 启用企业KDC

3. 公共集群服务组件默认按照所选主机自动配置。如需调整，可开启自定义拓扑开关进行手动配置。



4. 完成部署后可查看公共集群详情。



5. 操作完成。

启用企业KDC

操作场景

在安装公共集群时，系统会默认启用内置的KDC服务，然而，如果企业已经部署了自己的KDC，并希望TBDS服务能够与之集成，以便复用现有的安全基础设施，您可以在安装公共集群的过程中选择指定。

使用限制

1. 企业 KDC 仅支持MIT KDC。
2. 当多个TBDS平台或多个TBDS主账号（租户）共享同一个KDC时，对于同名用户，任何一处的修改都会自动同步到其他关联的地方。
3. 启用企业KDC认证功能后，若存在同名用户跨多个主账号，单个账号下keytab过期时间的更改将影响所有账号，会导致相关用户的keytab无法使用，而其他账号界面上的过期时间显示却不会同步更新的情况。

操作步骤

1. 登录 TBDS 控制台，安装公共集群，在部署公共集群页面展开“高级设置”，切换为“企业 KDC”。

2. 配置企业的 KDC 和 KAdmin 信息，说明如下。

配置项	说明
KDC 地址	输入企业的 KDC 地址，支持填写多个，用逗号分隔
Realm名称	输入企业 KDC Realm 的名称
KAdmin	企业 KDC 服务的地址包含多个时用逗号分隔
KAdmin principal	请输入管理员的 principal，用于 keytab 导出等场景。
KAdmin密码	请输入管理员的密码

3. 按照指引完成后续操作。

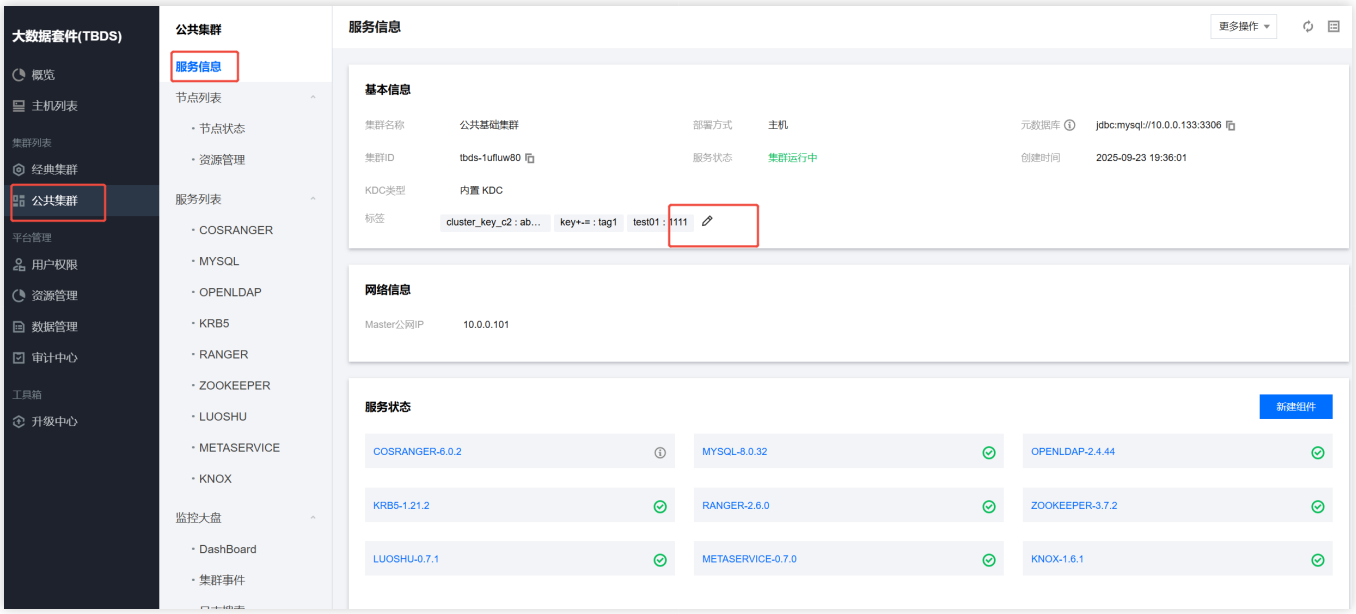
编辑标签

标签介绍详情请参考经典集群-编辑标签，创建标签请参考标签管理。

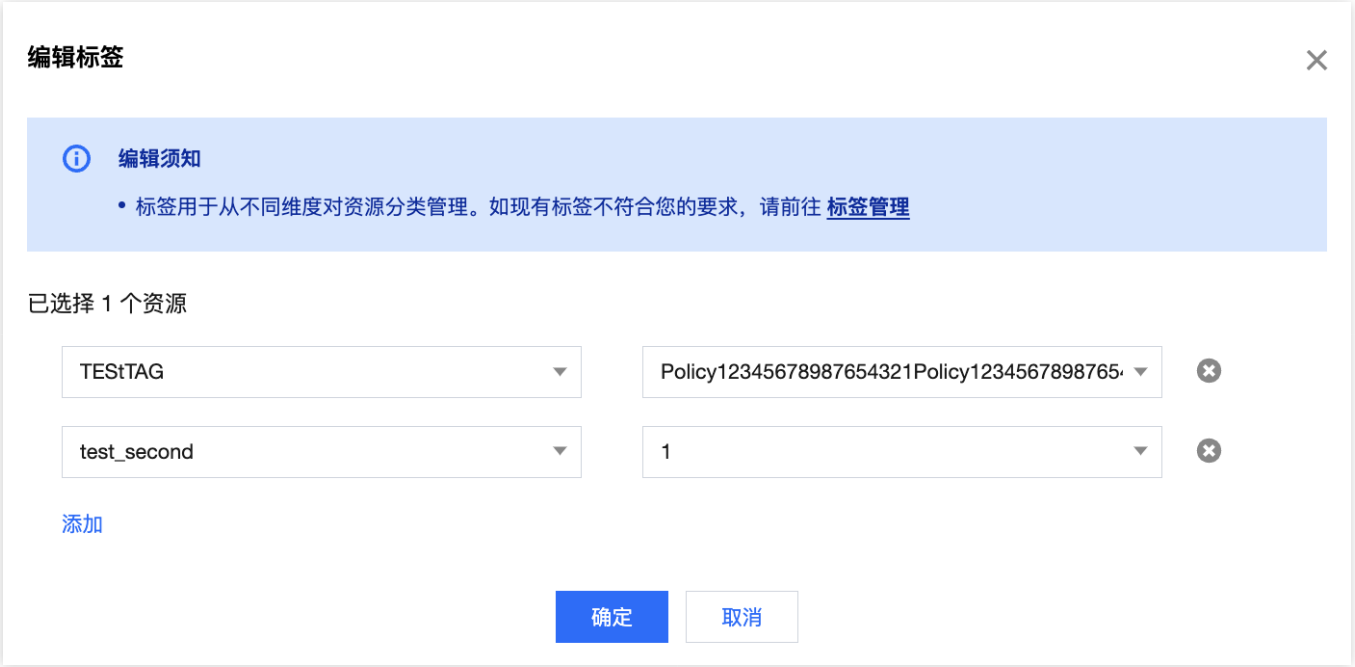
操作步骤

对集群编辑标签

1. 登录TBDS，在【公共集群】->【服务信息】->【基本信息】页面中，单击编辑标签。如下图所示：



2. 在弹出编辑标签对话框中为公共集群设置一个或多个标签键值。



3. 点击确定完成标签设置。

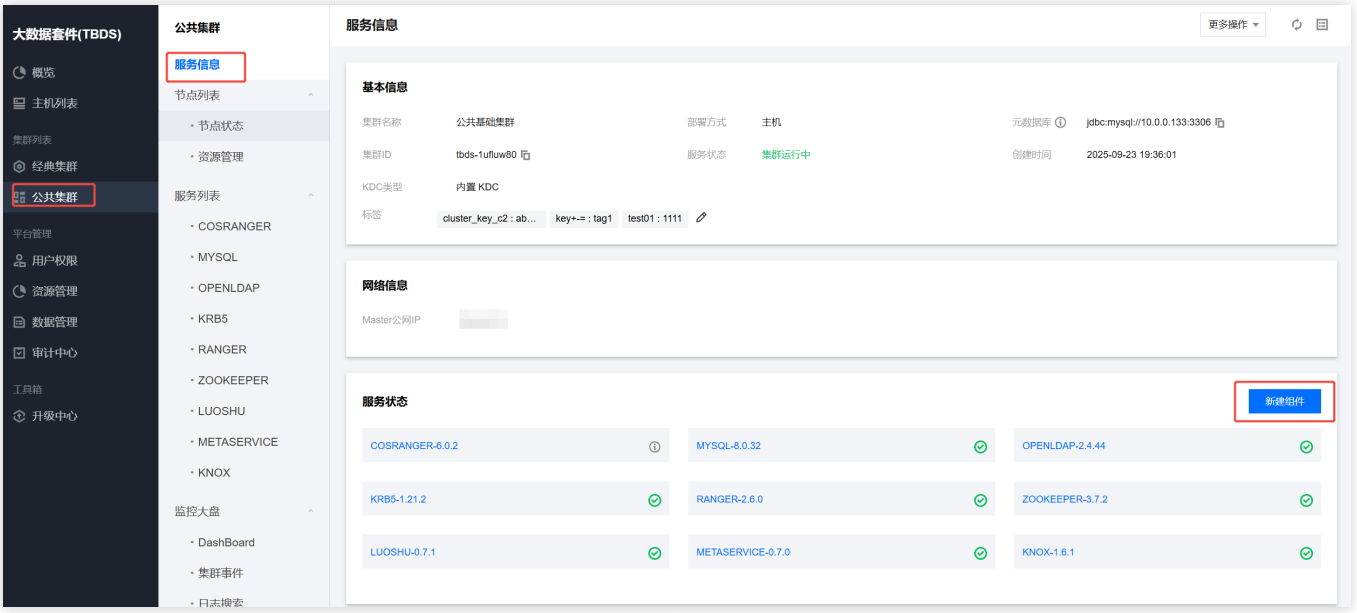
新增组件

操作场景

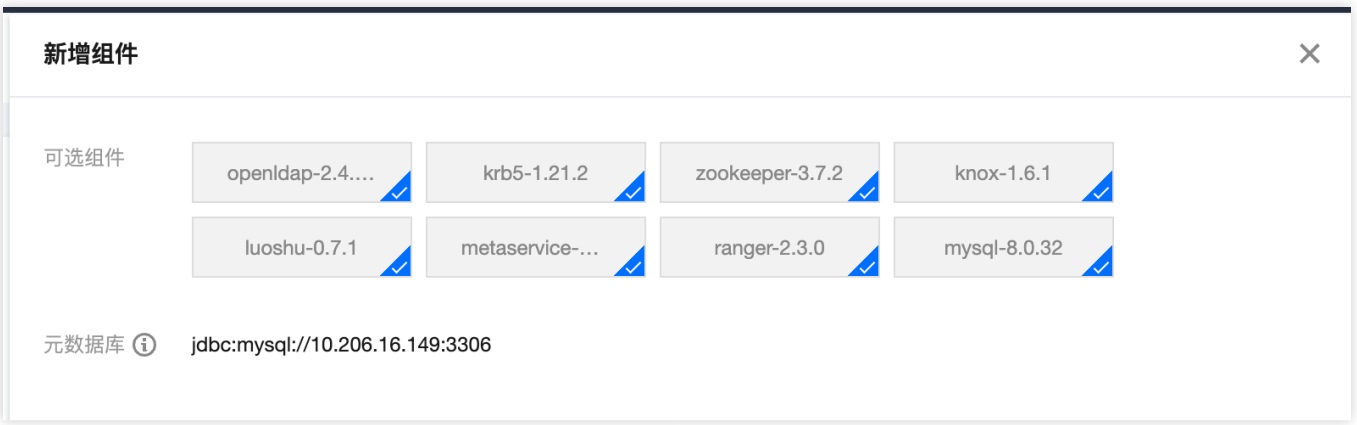
在部署公共集群的初始阶段，如果您只选择了部分组件进行安装（例如OpenLDAP和Kerberos），您可以通过后续的“添加组件”功能来扩展并包括其他公共集群组件。

操作步骤

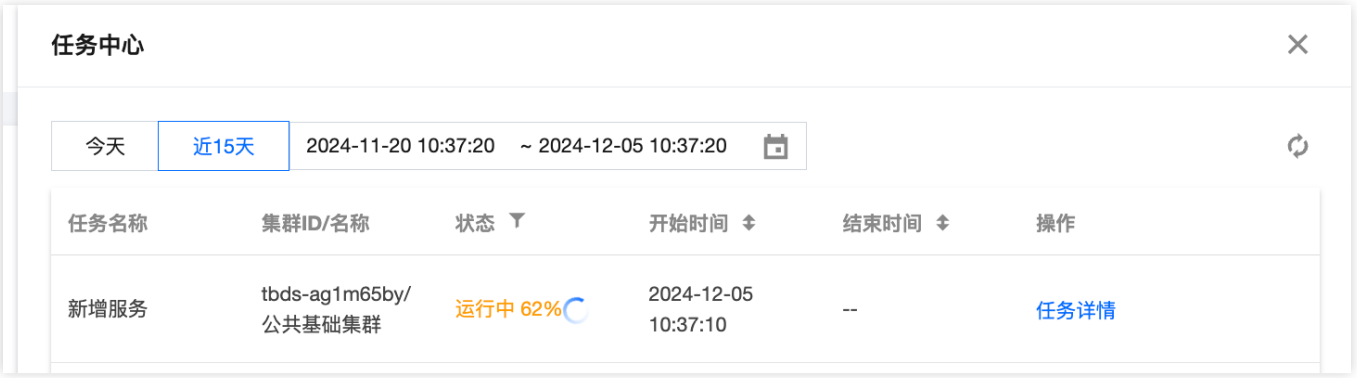
1. 登录TBDS控制台，进入公共集群 > 服务信息 > 服务状态，在集群服务面板点击“新建组件”。



2. 在新建组件页面选择需要扩展的公共集群组件，然后点击创建，如果组件已经全部部署，则不允许新增。



3. 点击公共集群页面图标 查看执行进度。



4. 操作完成。

负载检查

前提条件

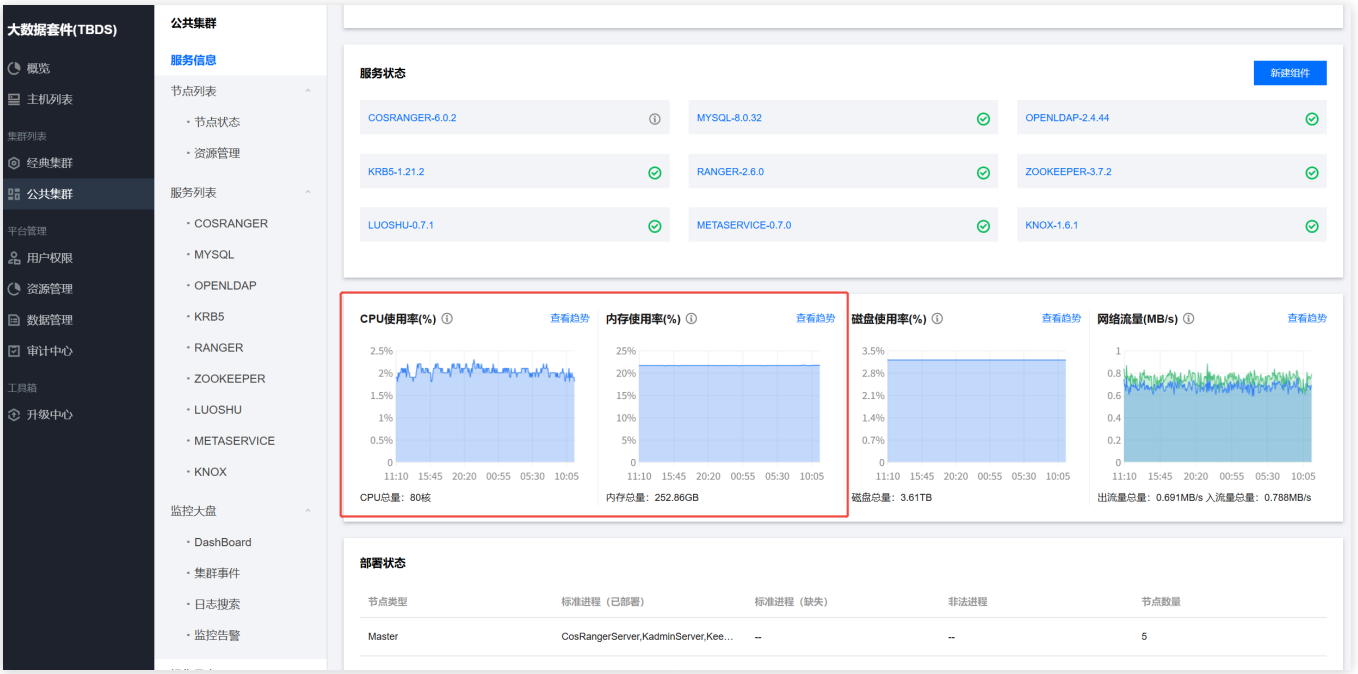
- 已完成公共集群安装。

操作场景

公共集群作为 TBDS 平台的核心共享基础设施，承载着安全等关键公共组件的部署。因此，确保公共集群的高可用性对于整个平台的稳定运行至关重要。在日常运维工作中，运维团队会定期对集群执行健康检查和负载监控，以评估其性能并决定是否需要进行资源的扩展或缩减。本节内容将详细阐述如何对公共集群的负载状况进行深入分析和评估，以确保其持续满足业务需求。

操作步骤

1. 查看公共集群资源使用趋势
   进入 TBDS Manager 首页 > 公共集群 >服务信息，可以查看当前公共集群整体资源平均使用率，如果利用率持续超过 80%，则需要考虑扩容。



2. 查看公共集群服务资源使用
   通过步骤 1 可以查看到公共集群资源的平均使用率，但并不能真实反映公共集群下服务的资源使用率，可进入到节点

状态查看详情。

资源调配

前提条件

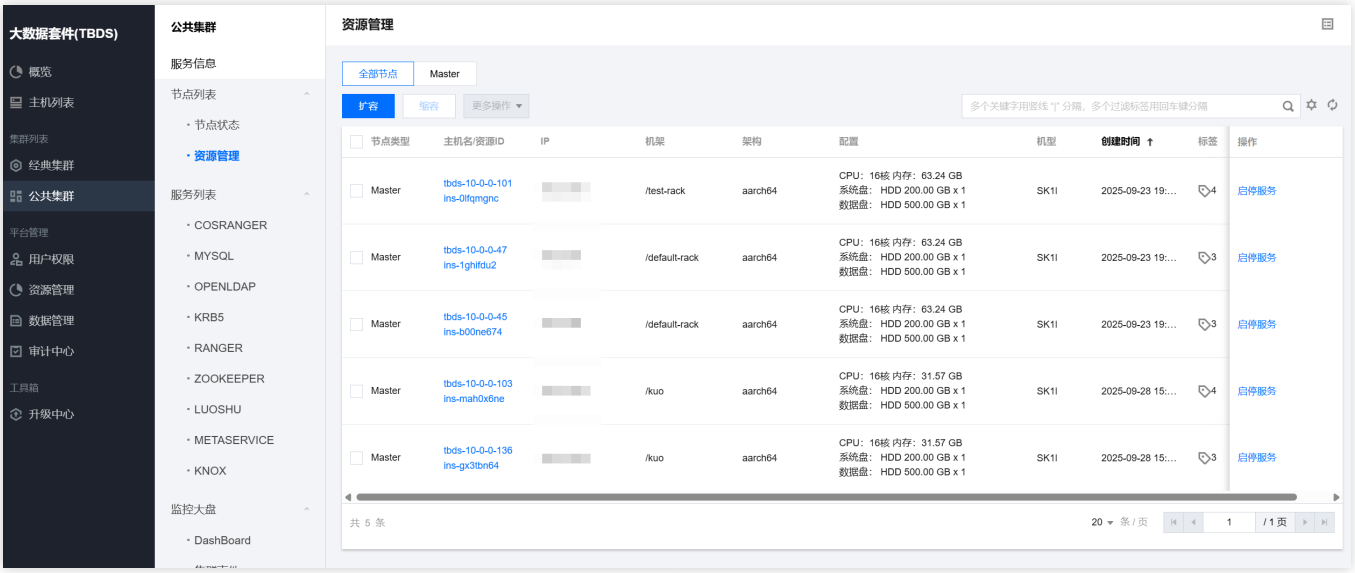
- 已完成公共集群安装。

操作场景

当业务负载较高、公共集群资源不足时，可找到高负载的服务进行资源调配。

操作步骤

1. 登录TBDS，在【公共集群】->【节点信息】->【资源管理】页面中，进行扩缩容操作。如下图所示：



具体参考 集群缩容集群扩容。

服务重启

前提条件

- 已完成公共集群安装。

注意事项

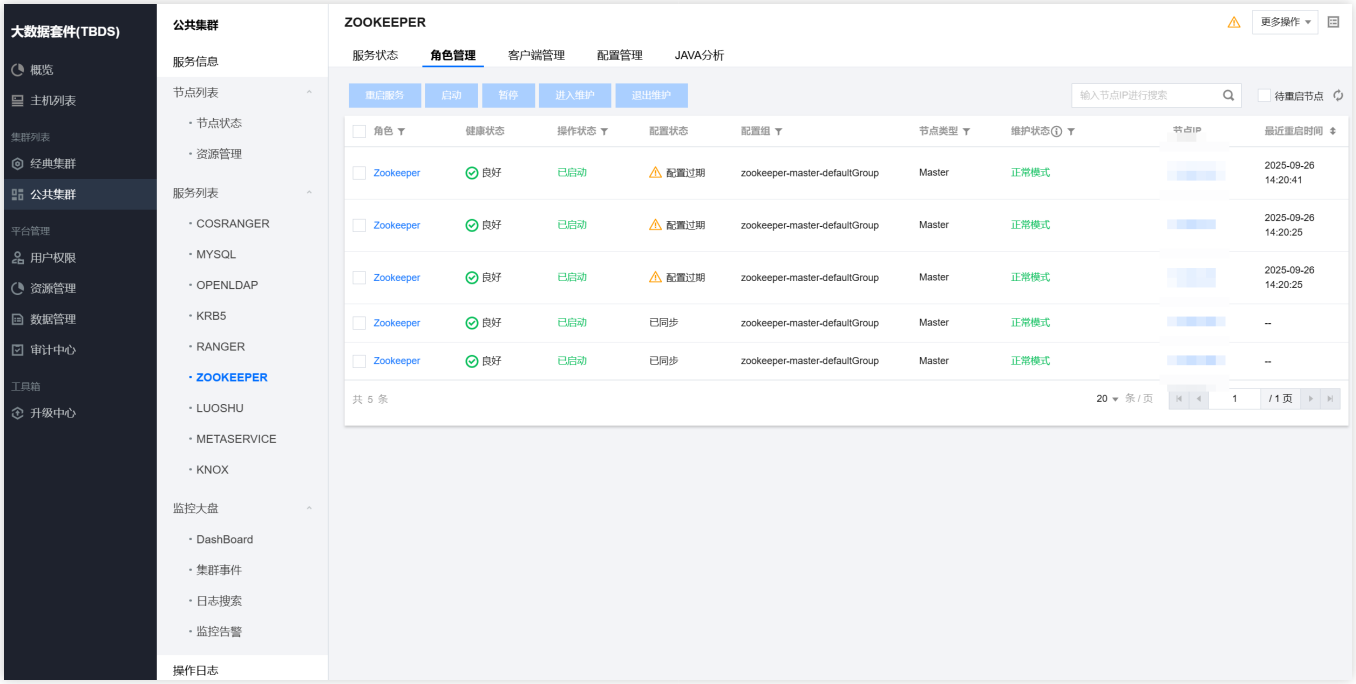
公共集群作为 TBDS 平台的核心共享基础设施，承载着安全等关键公共服务，在生产环境请务必按照滚动重启方式，避免造成平台不可用。

操作场景

当公共集群进行配置调整或资源优化后，需要通过重启来确保这些更改生效。

操作步骤

1. 登录TBDS，在【公共集群】->【服务列表】->页面中，进行重启操作。如下图所示：



详情参考经典集群 重启服务。

访问Ranger UI

前提条件

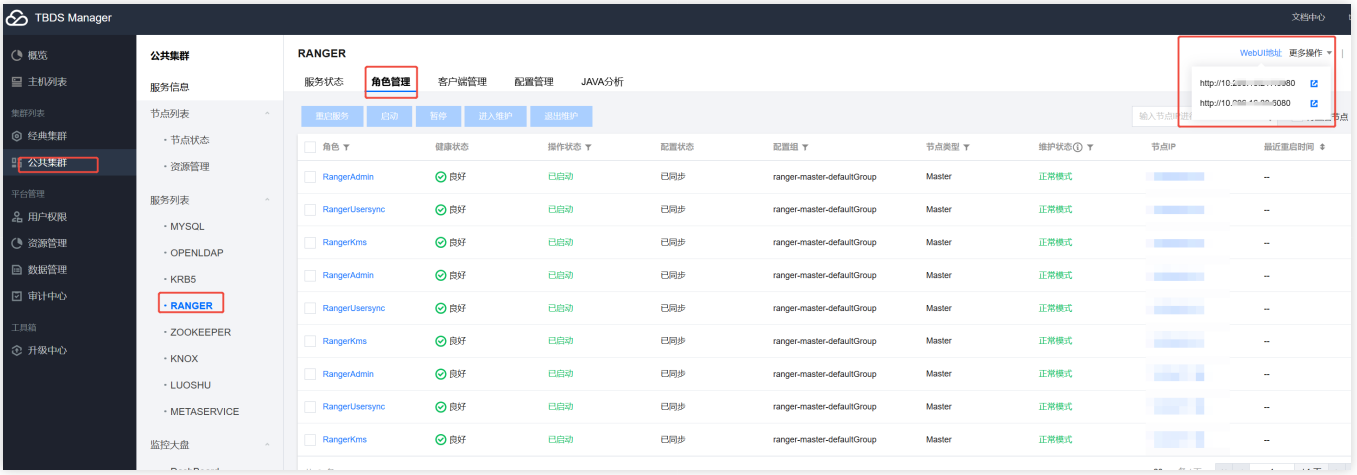
- 已完成公共集群安装。

操作场景

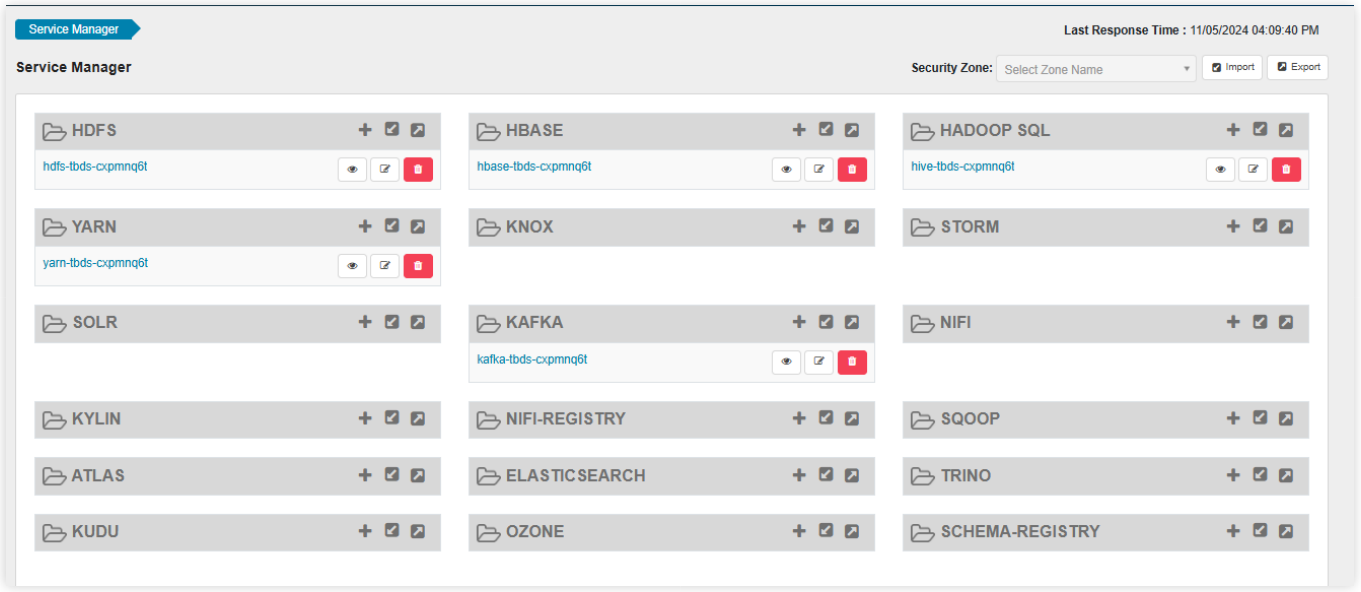
需要对用户和用户组授予经典集群下组件的访问权限，比如 HDFS 文件目录的读写权限，Hive 库表的增删改查权限，YARN 的提交权限等。

操作步骤

1. 查看并打开 webui
   进入 TBDS Manager 首页 > 公共集群 > 服务列表，单击 Ranger 组件，在页面右上方单击webui。



2. 登录 Ranger UI
   在登录输入用户名和密码，其中用户名为 root，密码为公共集群安装时输入的集群密码。登录后可按需为用户或用户组添加策略权限。


Service Manager
Last Response Time : 11/05/2024 04:09:40 PM
Service Manager
Security Zone:
Select Zone Name
Import
Export
HDFS
hdfs-tbds-cxpmnq6t
HBASE
hbase-tbds-cxpmnq6t
HADOOP SQL
hive-tbds-cxpmnq6t
YARN
yarn-tbds-cxpmnq6t
KNOX
STORM
SOLR
KAFKA
kafka-tbds-cxpmnq6t
NIFI
KYLIN
NIFI-REGISTRY
SQOOP
ATLAS
ELASTICSEARCH
TRINO
KUDU
OZONE
SCHEMA-REGISTRY

3. 操作完成。

卸载组件（销毁服务）

操作场景

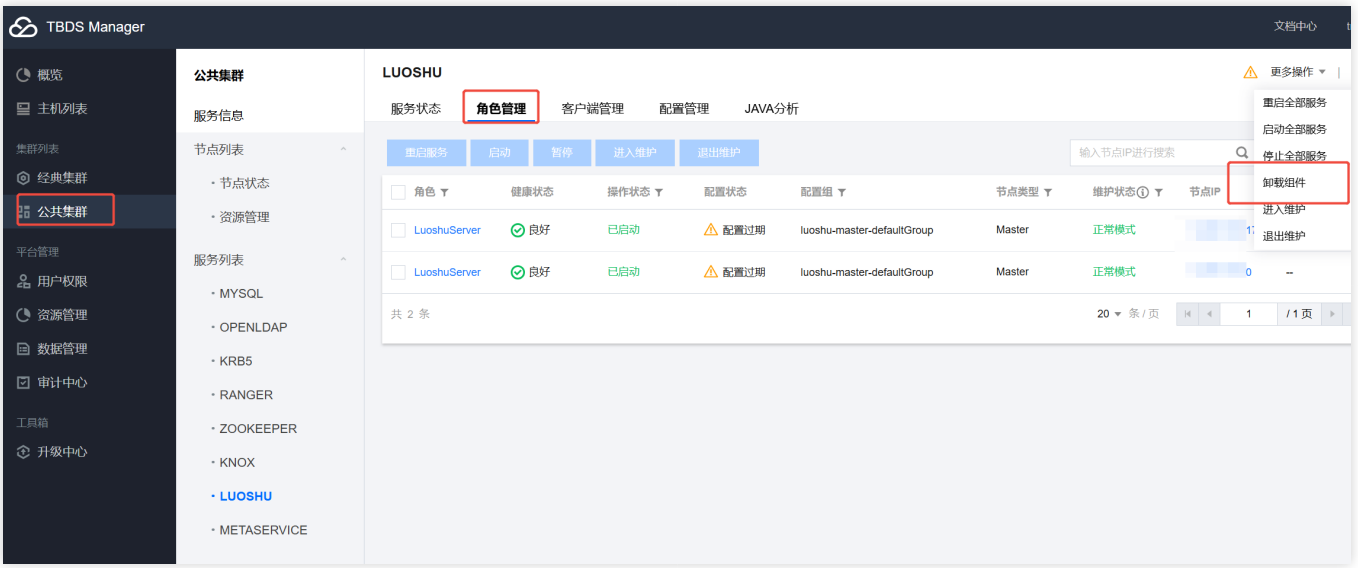
在特定的业务场景中，例如使用Elasticsearch或Starrocks时，并非所有公共集群组件都是必需的。在这种情况下，您可以根据实际需求，在公共集群页面灵活地选择并销毁（卸载）那些不需要的组件。这种按需管理和优化资源配置的方法，有助于提高集群的运行效率，并确保资源得到合理利用。

操作约束

- 主机形态下MySQL、Ranger、Knox、KRB5 和 LDAP 不允许卸载。
- 服务卸载需满足无其他服务依赖的条件。

操作步骤

1. 登录 TBDS 控制台，进入公共集群->服务列表 -> 服务X -> 角色管理 -> 更多操作，点击“卸载组件”。



经典集群
集群列表
部署集群

经典集群包括了经典的 Hadoop 集群，支持 Hadoop、StarRocks、Kafka、Elasticsearch 集群类型。可根据具体应用场景灵活选择并部署所需组件。

前提条件

- 已完成TBDS Manager安装。
- 已完成公共集群安装。
- 准备符合MySQL协议标准的元数据库。

操作场景

本节为您介绍通过 TBDS Manager 管理平台创建一个TBDS 集群的操作。

操作步骤

登录 TBDS Manager 管理平台，在集群列表页单击创建集群。

1. 软件配置

- 产品版本和部署组件：TBDS 推荐了常用的 Hadoop 、StarRocks、ElasticSearch 集群类型和应用场景下的组件搭配，可以根据需求组合，注意：不同类型场景下可选组件不同（可参见组件清单说明），包含必选和可选组件，可选组件存在依赖关系，注意依赖组件需要一并选择；也可以在集群创建后进行可选标准组件的增加。
- 网络设置：支持IPv4/IPv6/双栈模式，注意支持IPv6/双栈模式的前提是云平台和TM部署模式为双栈，否则经典集群只支持ipv4。
- 安全管理配置，默认集群将开启kerberos认证，注意认证模式只能在创建集群时选择，集群部署完成后不支持开启或关闭，后续关联集群的认证模式依赖首个集群的Kerberos认证状态，需结合集群规划谨慎选择。
- 软件配置：按照要求填写参数可实现自定义软件参数创建集群，同时兼容访问外部集群功能，在参数中正确配置访问地址信息即可读写外部集群的数据。具体操作请参考软件配置。

2. 节点部署配置

- 节点配置：TBDS 提供了多种节点类型，可以根据业务需要为不同节点类型选择合适机型配置。Master节点数量最大100台，不同服务Master进程完全独立的场景下可支持4集群联邦。
- 混合部署：TBDS 在开启或不开启混合部署的场景下均为高可用部署。关闭混合部署时，不同类型节点将不允许被部署在同一物理节点上，集群稳定性将得到最大保障；开启混合部署时，Master 节点将支持与 Core 节点部署在同一物理节点上，集群将获得更高的资源利用率。

  > 注意：
  >
  > Impala，Alluxio，Amoro，Kudu，Flume，Sqoop 组件不支持混合部署模式，选择相关组件后混合部署开关将无法开启。

TBDS提供了如下节点类型：
Master节点：为管理节点，保证集群的调度正常进行；主要部署 NameNode、ResourceManager、HMaster 等进程；数量≥3。
Core 节点：为计算及存储节点，您在 HDFS 中的数据全部存储于 Core 节点中，因此为了保证数据安全，扩容 Core 节点后不允许缩容；主要部署 DataNode、NodeManager、RegionServer 等进程。数量≥3。
Task 节点：为纯计算节点，不存储数据，被计算的数据来自 Core 节点及 COS 中，因此 Task 节点往往被作为弹性节点，可随时扩容和缩容；主要部署 NodeManager、Trino-Worker 等进程；可随时更改 Task 节点数，实现集群弹性伸缩，最小值为0。
Router 节点：用以分担 Master 节点的负载或者作为集群的任务提交机，可以随时扩容和缩容；可随时更改 Router 节点数，最小值为0。

3. 元数据库配置

当部署 Hive、Ranger 等依赖外部元数据库的组件时，需指定访问配置。用户可选择自建的、符合 MySQL 协议的数据库作为元数据库，用于存储元数据。配置需包括 jdbc:mysql:// 开头的访问地址、数据库名、用户名及密码，并确保网络连通性。

4. 部署配置

TBDS默认会根据用户选择的主机自动生成部署方案。如需调整，可开启自定义部署功能进行手动配置。开启后，系统支持在 Core 和 Task 类型节点上自定义部署部分关键组件进程。Master和Core 节点支持 Yarn、Trino、Impala、HBase、Kafka 等服务进行自定义节点部署，满足隔离负载部署需求,其他角色将按照默认方式部署在集群中。

5. 基础配置

- 集群名称：通过设置集群名称，来区分不同 TBDS 集群。
- 集群密码：通过设置集群密码来初始化服务 WebUI 密码。
- 高级设置：通过此功能可以安装其它第三方软件，或修改集群运行环境，引导操作会在集群创建和集群扩容时运行引导脚本（Router 节点除外）。

  > 说明：
  >
  > 引导操作的脚本将按照添加的顺序依次执行，累计不超过16个。
  >
  > 目前控制台只支持集群创建和销毁集群时指定引导操作：

> 一、集群创建（含扩容）时指定的引导操作支持在如下三个时机执行。
>
> 1、集群安装前：安装集群软件之前。
>
> 2、集群启动前：在集群服务启动之前。
>
> 3、集群启动后：在集群服务启动之后。
>
> 二、集群销毁（含缩容）时指定的引导操作支持在如下一个时机执行。
>
> 1、服务下线前：在集群服务下线前。

6. 确认配置信息

信息确认：确认集群软件配置信息，节点部署方案和基础配置，完成以上配置后，单击部署集群进创建过程，等待集群进部署流程后即可在 TBDS Manager 集群列表中找到新建的集群。

集群启停

功能介绍

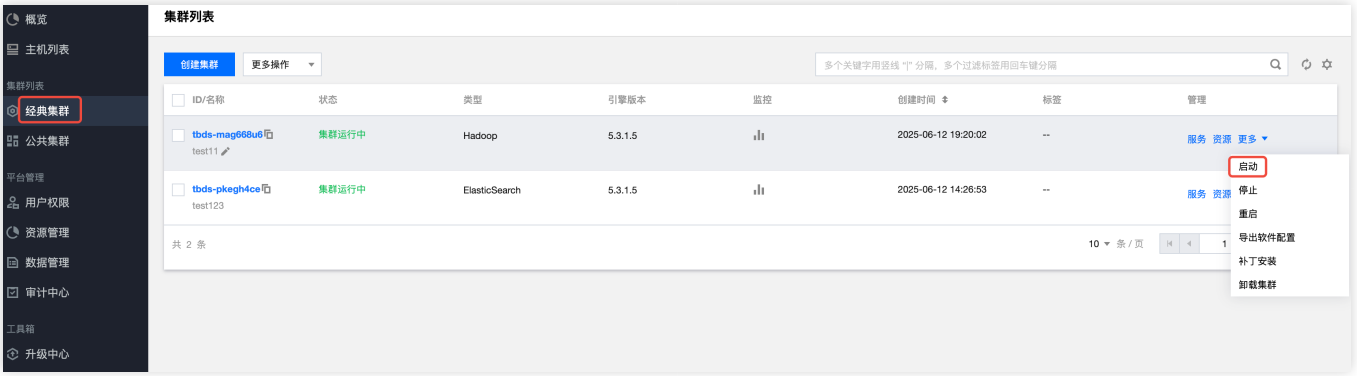
支持对经典集群进行整体操作，包括启动、停止和重启，具体说明如下：

- 启动：按服务依赖顺序启动，例如若YARN依赖HDFS，则先启动HDFS再启动YARN。
- 停止：按服务依赖逆序停止，例如若YARN依赖HDFS，则先停止YARN再停止HDFS。
- 重启：支持滚动重启/非滚动重启，同时可按过期配置的服务进行选择性重启。
  - 非滚动重启：先停止集群，再启动集群，重启期间服务不可用
  - 滚动重启：按服务依赖顺序滚动重启组件，例如若YARN依赖HDFS，则先滚动重启HDFS再滚动重启YARN。TBDS 所有服务均已支持高可用，重启期间不影响服务可用性

操作步骤

启动集群

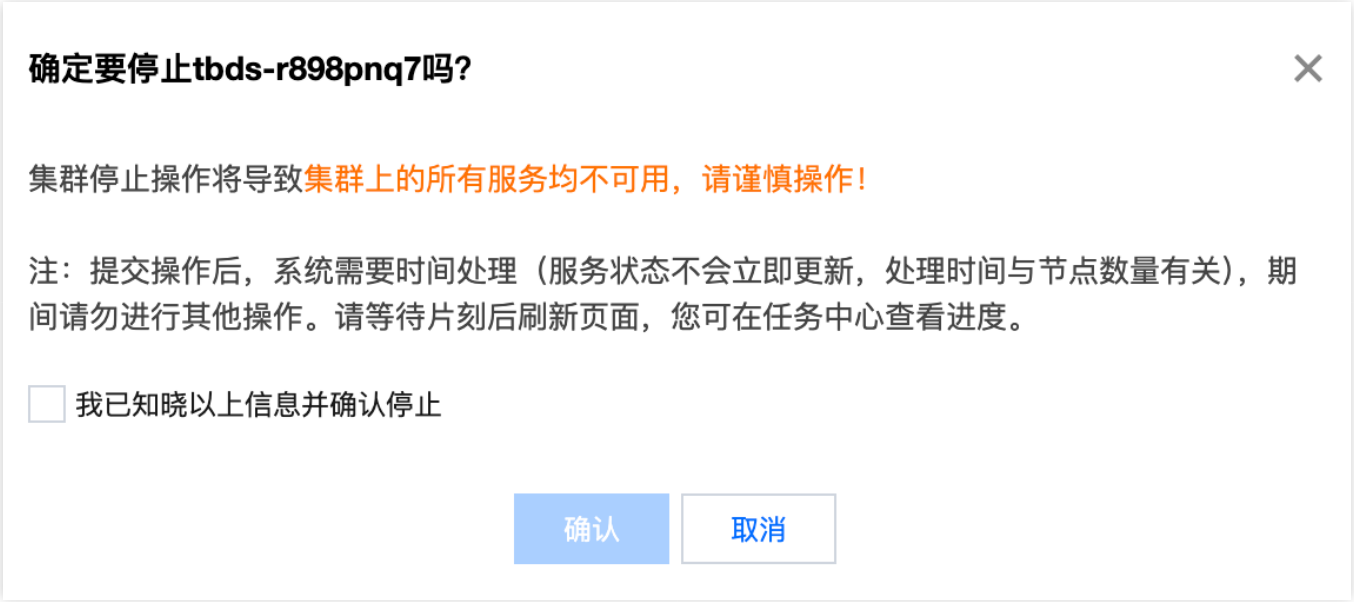
1. 以管理员用户登录 TBDS，进入 TBDS Manager首页 > 经典集群，在目标集群的管理下，点击【更多】->【启动】。



2. 点击【启动】后，系统将自动按服务依赖关系顺序启动各组件（已启动的服务会自动跳过）。

停止集群

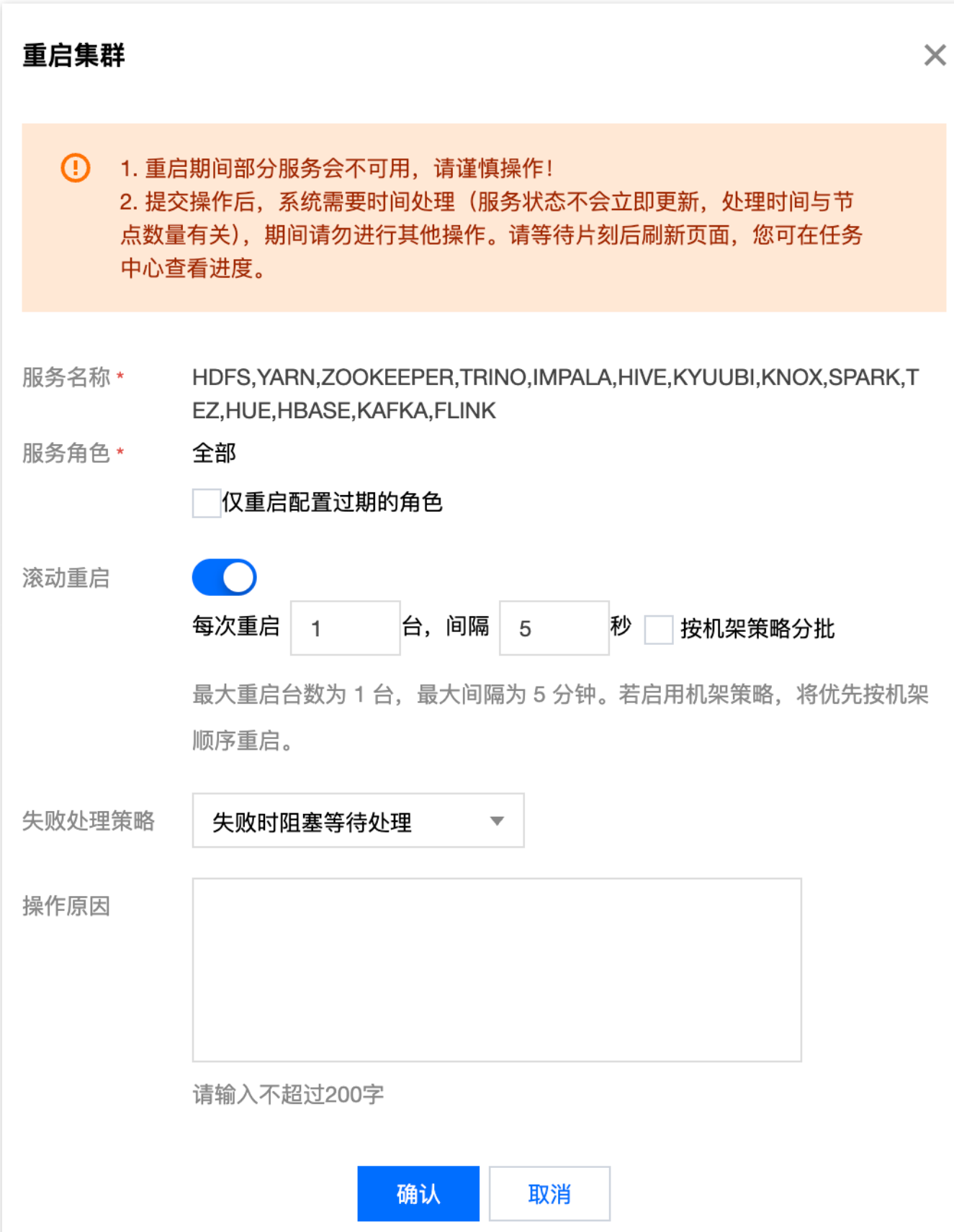
1. 以管理员用户登录 TBDS，进入 TBDS Manager首页 > 经典集群，在目标集群的管理下，点击【更多】->【停止】。
2. 停止集群将导致集群不可访问，请务必评估业务影响后再执行操作。



重启集群

1. 以管理员用户登录 TBDS，进入 TBDS Manager首页 > 经典集群，在目标集群的管理下，点击【更多】->【重启】
2. 支持以下两种重启选项，可最大限度减少服务中断：

- 配置过期角色重启：仅重启配置发生变更的服务角色
- 滚动重启：分批次逐步重启，保障服务持续可用



注意：

集群重启复用了单组件滚动重启功能，目前取消操作不保证回滚。取消重启任务后需人工确认组件状态（例如：

> 取消时若组件已停止但还未启动，会导致组件状态处于已停止状态）

编辑标签

功能介绍

标签是TBDS提供的用于标识集群类型的标记，是一个键值对（Key-Value）。
您可根据各种维度（例如业务、用途、负责人等）使用标签对集群类型进行分类管理。也可通过标签非常方便的标识集群或节点资源。标签键值对对TBDS没有任何语义意义，会严格按字符串进行解析匹配。

使用限制

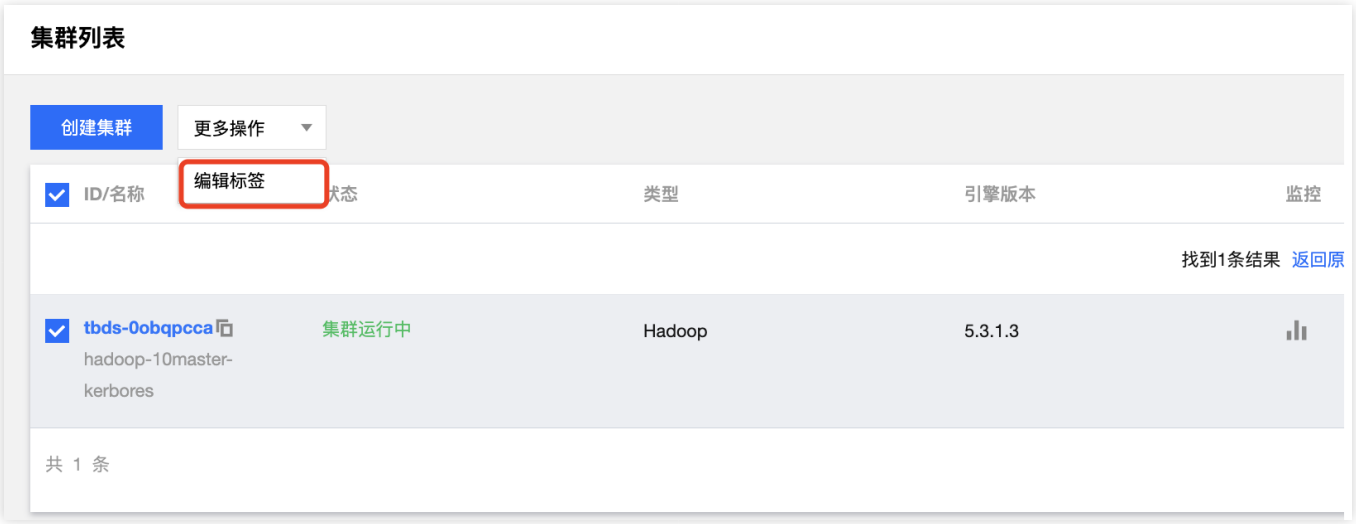
标签是一个键值对（Key-Value），您可以通过对TBDS集群设置标签实现集群分类管理。通过标签可以非常方便地查看标识对应的集群，您可以在TBDS对集群或节点资源进行标签的编辑。
编辑标签时，需注意以下限制条件：

- 数量限制：每个集群或节点允许的最大标签数是50（一次最多添加5个）。
- 标签键限制：
  - 以 qcs:、project:、项目等开头的标签键为系统预留标签键，系统预留标签键禁止创建。
  - 在 UTF-8 中，标记键必须最少为1，最多为127个 Unicode 字符。
  - 支持 UTF-8 格式表示的字符、空格和数字以及特殊字符，不支持以空格开头或结尾：
    - 英文状态下支持： + - = . _ : / @ ( ) [ ] , ; > <
    - 中文状态下支持： + - = / @ （ ） 【 】 ：
- 标签值限制：只能为空字符串、数字、字母、+=.@-，且标签值最大长度为127个字符。
  - 在 UTF-8 中，标记值必须最少为1，最多为255个 Unicode 字符。
  - 支持 UTF-8 格式表示的字符、空格和数字以及特殊字符，不支持以空格开头或结尾：
    - 英文状态下支持： + - = . _ : / @ ( ) [ ] , ; > <
    - 中文状态下支持： + - = / @ （ ） 【 】 ：

操作步骤

对集群编辑标签

1. 登录TBDS，在经典集群列表页面中，选择需要编辑标签的集群，单击顶部的更多操作 > 编辑标签。如下图所示：



2. 在弹出编辑标签对话框中为标准设置一个或多个标签键值。

3. 点击确定完成标签设置。

软件配置

功能说明

软件配置支持您创建集群时自定义 hdfs、yarn、hive 等组件的配置。

自定义软件配置

软件配置支持用户创建集群时自定义 hdfs、yarn、hive 等组件的配置。Hadoop、Hive 等软件含有大量配置，通过软件配置功能您可以在新建集群的过程中自主配置组件参数。配置过程需要您按要求提供相应的 JSON 文件，文件可以由您自定义，也可以将存量集群的软件配置参数导出，然后快速新建一个集群。导出软件配置参数，详情请参见 导出软件配置。
JSON文件示例及说明：

```
[
  {
    "serviceName": "HDFS",
    "classification": "hdfs-site.xml",
    "serviceVersion": "2.8.4",
    "properties": {
      "dfs.blocksize": "67108864",
      "dfs.client.slow.io.warning.threshold.ms": "900000",
      "output.replace-datanode-on-failure": "false"
    }
  },
  {
    "serviceName": "YARN",
    "classification": "yarn-site.xml",
    "serviceVersion": "2.8.4",
    "properties": {
      "yarn.app.mapreduce.am.staging-dir": "/emr/hadoop-yarn/staging",
      "yarn.log-aggregation.retain-check-interval-seconds": "604800",
      "yarn.scheduler.minimum-allocation-vcores": "1"
    }
  },
  {
    "serviceName": "YARN",
    "classification": "capacity-scheduler.xml",
    "serviceVersion": "2.8.4",
    "properties": {
      "content": "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<?xml-stylesheet type=\"text/xsl\" href=\"configuration.xsl\"?>\n<configuration><property>\n <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>\n <value>0.8</value>\n</property>\n<property>\n <name>yarn.scheduler.capacity.maximum-applications</name>\n <value>1000</value>\n</property>\n<property>\n <name>yarn.scheduler.capacity.root.default.capacity</name>\n <value>100</value>\n</property>\n<property>\n <name>yarn.scheduler.capacity.root.default.maximum-capacity</name>\n <value>100</value>\n</property>\n<property>\n <name>yarn.scheduler.capacity.root.default.user-limit-factor</name>\n <value>1</value>\n</property>\n<property>\n <name>yarn.scheduler.capacity.root.queues</name>\n <value>default</value>\n</property>\n</configuration>"
```

```
        }
      }
  ]
```

配置参数说明：
serviceName 组件名，必须大写。
classification 文件名，必须使用全称，包含后缀。
serviceVersion 版本名，为组件版本，该版本必须与 TBDS 产品版本中对应的组件版本一致。
properties 中填写需要自行配置的参数。
如需修改 capacity-scheduler.xml、fair-scheduler.xml 中配置参数，properties 中的属性key需指定为 content，value 为整个文件的内容。
如果您需要调整存量集群的组件配置，您可以进行 配置更新。

访问外部集群

配置外部集群 HDFS 的访问地址信息后，可以读取外部集群的数据。

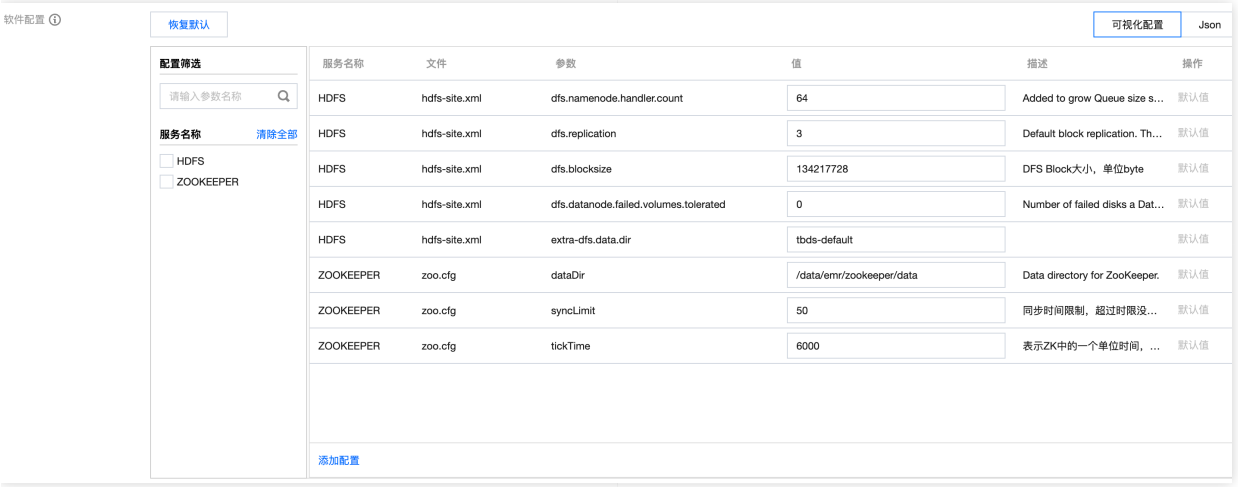
部署时配置

TBDS Manager支持新建集群时，配置访问外部集群，只需在软件配置处输入符合要求的 JSON 文件进行配置即可。下面以假设条件为例进行说明：
假设条件：假设需要访问外部集群的 nameservice 为 ${nameservice_name}，其访问方式为：

```
  <property>
    <name>dfs.ha.namenodes.${nameservice_name}</name>
    <value>nn1,nn2</value>
  </property>
  <property>
    <name>dfs.namenode.http-address.${nameservice_name}.nn1</name>
    <value>${ip:port}</value>
  </property>
  <property>
    <name>dfs.namenode.https-address.${nameservice_name}.nn1</name>
    <value>${ip:port}</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.${nameservice_name}.nn1</name>
    <value>${ip:port}</value>
  </property>
  <property>
    <name>dfs.namenode.http-address.${nameservice_name}.nn2</name>
    <value>${ip:port}</value>
  </property>
  <property>
    <name>dfs.namenode.https-address.${nameservice_name}.nn2</name>
    <value>${ip:port}</value>
  </property>
  <property>
```

```
    <name>dfs.namenode.rpc-address.${nameservice_name}.nn2</name>
    <value>${ip:port}</value>
  </property>
```

如需在新建集群中就可访问外部集群，进入软件配置页后，打开高级设置。



JSON文件及说明：以假设条件为例，框内应该填入 JSON 文件内容（json 内容要求同自定义软件配置）。

```
[
  {
    "serviceName": "HDFS",
    "classification": "hdfs-site.xml",
    "serviceVersion": "2.7.3",
    "properties": {
      "newNameServiceName": "newEmrCluster",
      "dfs.ha.namenodes.${nameservice_name}": "nn1,nn2",
      "dfs.namenode.http-address.${nameservice_name}.nn1": "${ip:port}",
      "dfs.namenode.https-address.${nameservice_name}.nn1": "${ip:port}",
      "dfs.namenode.rpc-address.${nameservice_name}.nn1": "${ip:port}",
      "dfs.namenode.http-address.${nameservice_name}.nn2": "${ip:port}",
      "dfs.namenode.https-address.${nameservice_name}.nn2": "${ip:port}",
      "dfs.namenode.rpc-address.${nameservice_name}.nn2": "${ip:port}"
    }
  }
]
```

配置参数说明：
serviceName 组件名，必须为“HDFS”。
classification 文件名，必须为“hdfs-site.xml”。
serviceVersion 版本名，为组件版本，该版本必须与 TBDS 产品版本中对应的组件版本一致。
properties 中填写的内容与假设条件一致。
newNameServiceName（选填）表示当前新建集群 nameservice。如为空，则由系统生成；如非空，只能由字符串 + 数字 + 中划线组成。

> 注意：
>
> 访问的外部集群只支持高可用集群。

> 访问的外部集群只支持未开启 Kerberos 的集群。

部署后配置

在集群创建后，支持通过 配置更新 功能来访问外部集群。
配置项中填写需要访问外部集群的 nameservice 名称和value（如果是非高可用集群，一般是 masterIp:rpcport）。参考配置为：

```
<property>
  <name>dfs.ha.namenodes.${nameservice_name}</name>
  <value>nn1,nn2</value>
</property>
<property>
  <name>dfs.namenode.http-address.${nameservice_name}.nn1</name>
  <value>${ip:port}</value>
</property>
<property>
  <name>dfs.namenode.https-address.${nameservice_name}.nn1</name>
  <value>${ip:port}</value>
</property>
<property>
  <name>dfs.namenode.rpc-address.${nameservice_name}.nn1</name>
  <value>${ip:port}</value>
</property>
<property>
  <name>dfs.namenode.http-address.${nameservice_name}.nn2</name>
  <value>${ip:port}</value>
</property>
<property>
  <name>dfs.namenode.https-address.${nameservice_name}.nn2</name>
  <value>${ip:port}</value>
</property>
<property>
  <name>dfs.namenode.rpc-address.${nameservice_name}.nn2</name>
  <value>${ip:port}</value>
<property>
```

如果这些信息是经典集群中的，可在 配置更新 管理页查看，或者登录到机器查看 /usr/local/service/hadoop/etc/hadoop/hdfs-site.xml文件。

1. 进入 配置更新 页面，选择 hdfs 组件的hdfs-site.xml文件。
2. 修改配置项 dfs.nameservices 为具体nameservice 名。
3. 增加配置项及值

配置项	配置值
dfs.ha.namenodes. ${nameservice_name}	nn1，nn2
fs.namenode.http-address. ${nameservice_name}.nn1	${ip:port}

dfs.namenode.https-address.${nameservice_name}.nn1	${ip:port}
dfs.namenode.rpc-address.${nameservice_name}.nn1	${ip:port}
fs.namenode.http-address.${nameservice_name}.nn2	${ip:port}
dfs.namenode.https-address.${nameservice_name}.nn2	${ip:port}
dfs.namenode.rpc-address.${nameservice_name}.nn2	${ip:port}
dfs.client.failover.proxy.provider.${nameservice_name}	org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider
dfs.internal.nameservices	${nameservice_name}

> 注意：
>
> dfs.internal.nameservice 需要新增配置项，否则扩容集群后可能导致 datanode 上报异常而被 namenode 标记为 dead。

4. 下发配置，使用 配置更新 功能下发配置。
   更多相关配置详情及原理，请参考 社区文档。

导出软件配置

功能介绍

通过 TBDS Manager 管理平台，可以导出存量集群的软件配置参数，便于软件配置的二次开发、备份、迁移和恢复；导出软件配置参数可作为新集群的预设配置，从而快速新建一个熟悉的集群。

- 导出维度：默认集群维度，支持配置组维度或节点维度选择（单选）。
- 导出模式：默认“全部配置”，支持选择只导出自定义和修改过的配置（二选一）；节点维度无导出模式选择，默认全部配置。
- 导出格式：集群维度和配置组维度仅支持JSON格式；节点维度仅支持原文件格式。

  > 说明：
  >
  > - 若新集群想要复用老集群配置，建议导出老集群有修改的配置文件，需导出全部配置文件。
  > - 节点维度仅支持原文件格式下载（客户端配置），可通过客户端执行管理操作、运行业务或进行二次开发。

操作步骤

1. 登录 TBDS Manager 管理平台进入集群列表页。
2. 在待导出集群的管理栏选择更多 > 导出软件配置。
3. 勾选需要导出的文件，选择导出配置即可下载得到软件配置文件。

导出软件配置
1、导出软件配置请注意规避配置项参数敏感信息泄漏，如：密码、ID等。
2、集群软件配置参数涉及的组件有：FILEBEAT, HBASE, HDFS, HIVE, IMPALA, KNOX, KRB5, OPENLDAP, RANGER, SPARK, TEZ, YARN, ZOOKEEPER
3、若新集群想要复用老集群配置，建议导出老集群有修改的配置文件，无需导出全部配置文件。
4、节点维度仅支持原文件格式下载（客户端配置），可通过客户端执行管理操作、运行业务或进行二次开发。
导出维度
集群维度
配置组维度
节点维度
配置文件
全选
FILEBEAT
filebeat.yml
HBASE
hadoop-metrics2-hbase.properties
hbase-env.sh
hbase-site.xml
导出模式
全部配置
只导出自定义和修改过的配置
文件格式
JSON
导出配置
取消

补丁安装

经典集群支持大版本内选择补丁包进行集群升级，详细操作可参考：升级中心

集群卸载

功能介绍

集群卸载功能支持用户通过平台界面化操作解除服务集群和 TBDS Manager 管理平台间的管理关系，平台将停止集群中的组件服务并清理集群 TBDS Manager 平台中的元数据信息。

操作步骤

1. 登录 TBDS Manager 管控平台，在集群列表中找到目标的集群条目。
2. 选择更多 > 卸载集群进集群卸载流程。
3. 确认需要进行卸载操作的集群和节点信息后，单击确认卸载集群，集群卸载流程不可恢复，请谨慎操作。

4. TBDS Manager 将进入集群卸载流程，流程结束后集群将从 TBDS Manager 中移除。请注意集群卸载操作将不会删除集群中存储的数据，业务数据将被保留。

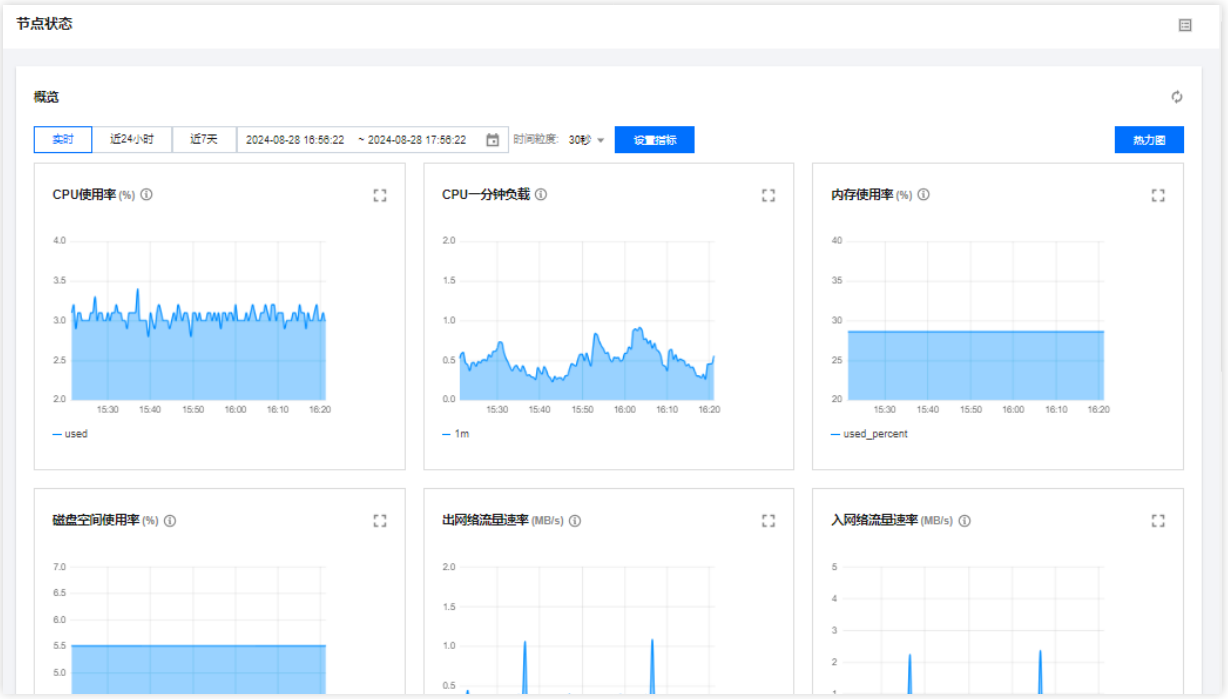
集群资源
节点状态

功能介绍

节点状态页面展示了当前集群所有节点监控概览和所有节点列表，并支持查看所有节点热点图。用户可以在日常使中，通过 TBDS Manager 管理平台，管理节点的状态及指标信息。

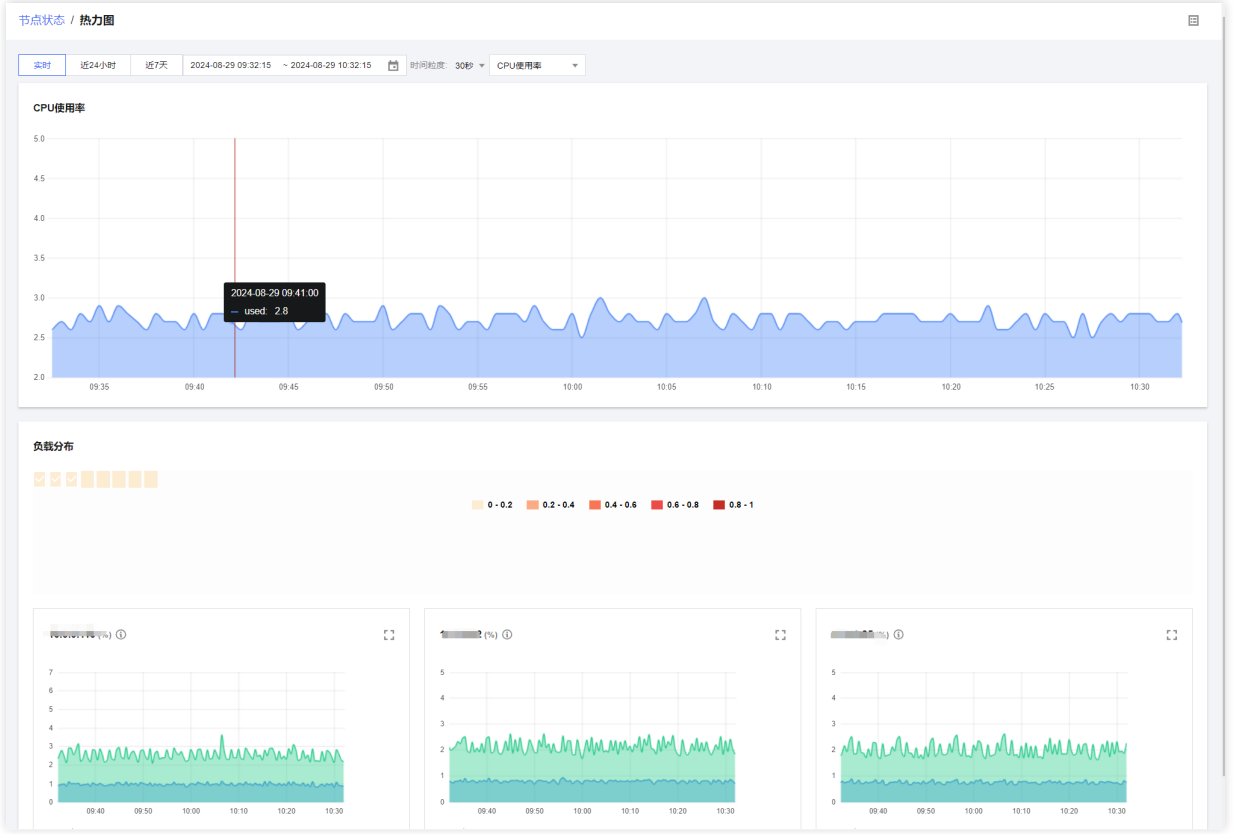
操作步骤

1. 登录 TBDS Manager 管控平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 选择集群资源 > 节点状态可查看集群中所有节点监控信息。
3. 在节点监控中，可查看当前集群所有节点聚合监控指标概览和所有节点列表。

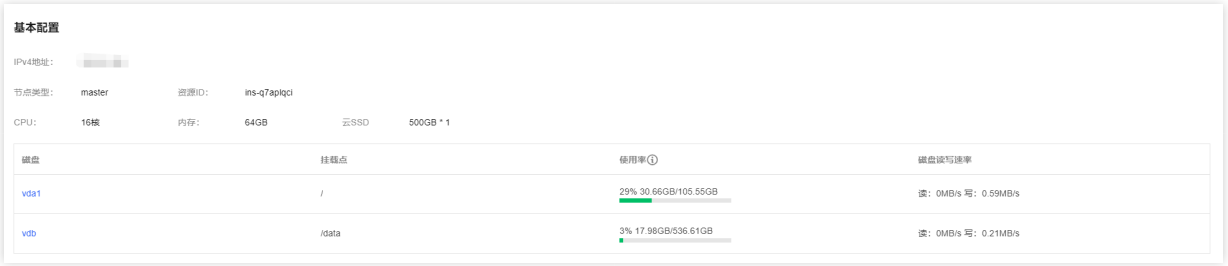


- **概览：**可直观查看对应时间段所有节点聚合监控指标及指标各项统计规则。可单击设置指标自定义展示指标。

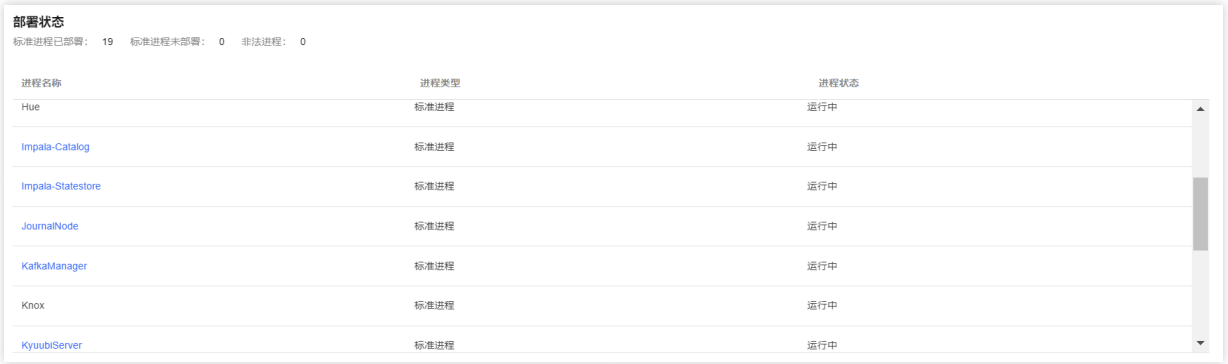
- **热力图：**负载热点图更加直观的展示了节点的负载情况，同时可指定时间段和负载条件进查看。负载热点图主要分为两部分，一部分为当前集群所有节点聚合负载图；部分为所有节点单个热点图，可直观查看所有节点的负载情况。



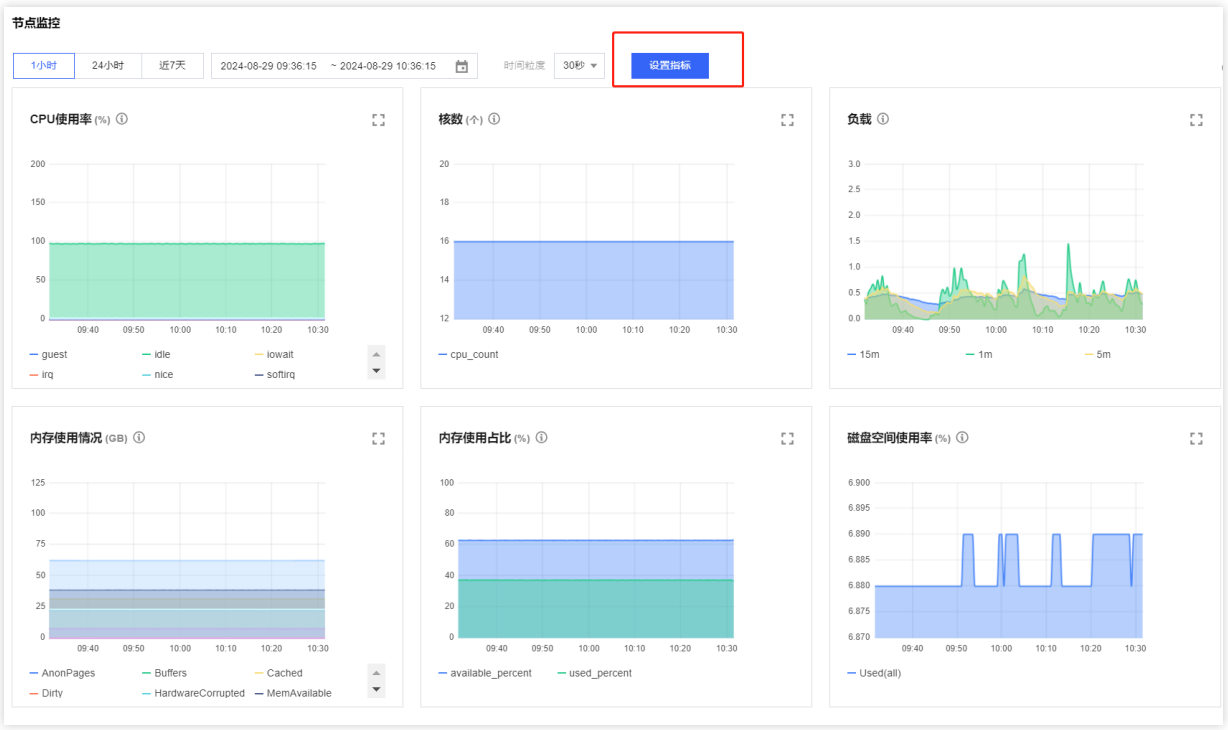
- **节点列表：**展示了当前所有节点和部署节点类型、CPU 利用率、内存利用率、磁盘利用率。单击对应节点 IP 可查看单个节点基本配置、部署状态、负载状态、节点监控等。
  - 基本配置：可查看当前节点的基本信息，例如节点类型、资源 ID、CPU 及内存等。



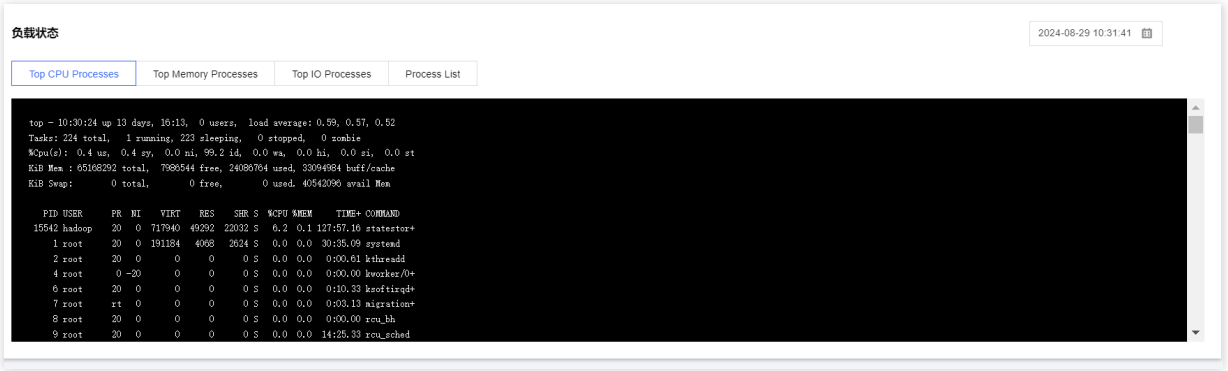
  - 部署状态：可查看当前节点的服务部署情况，是否为标准进程、进程状态是否运行正常等。



  - 节点监控：可查看当前节点各分组指标负载趋势图，默认展示6个指标，最多可展示12个指标。可单击设置指标自定义展示指标。



- 负载状态：可查看当前节点维度 TOP N 进程情况，同时可根据指定时间进行查看。



集群扩容

功能介绍

当 TBDS 集群计算资源、存储资源不足时，可以通过控制台对 Core 节点和 Task 节点进行扩容。当集群主节点（ master 节点）负载较高或不够使时，可以通过扩容或新增路由节点（Router 节点）分担 Master 节点的负载，或作为集群的任务提交机，并支持随时扩容。

> **说明：**所选扩容组件默认继承集群维度配置，且扩容节点将归属该节点类型默认配置组。如需调整扩容组件配置，可通过指定配置设置。

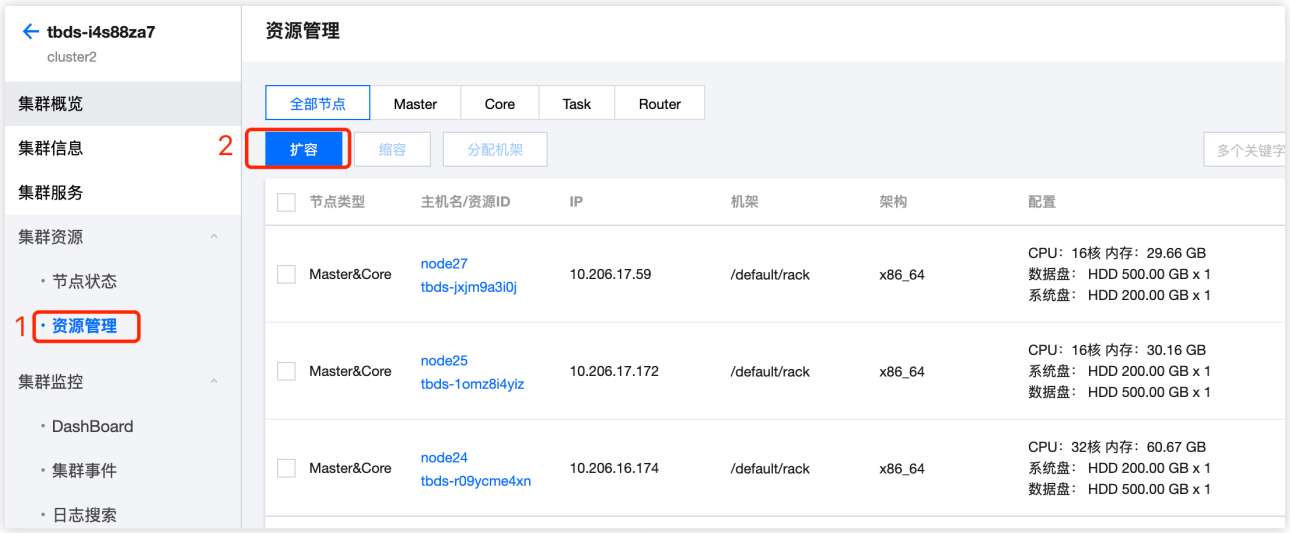
前提条件

扩容 Router 节点：Router 节点可作为提交机使用，可以在 Router 节点上向集群正常提交计算 YARN、Hive、Spark 等计算任务，因此建议配置不低于 Master 规格。

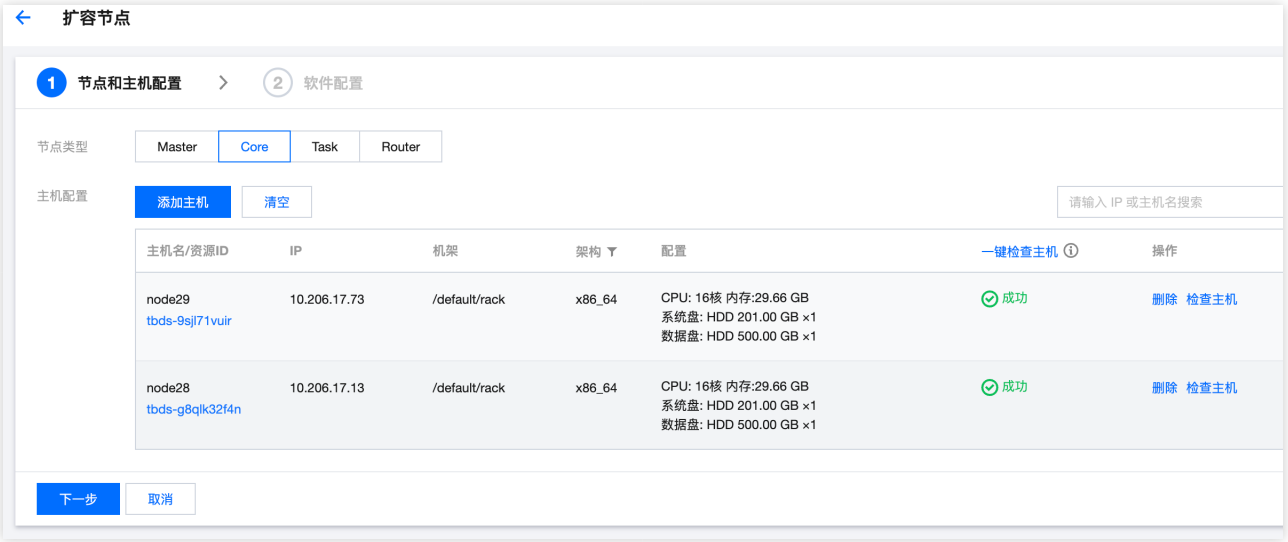
操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群资源 > 资源管理 > 扩容，根据业务需要选择需要扩容的节点类型（Core、Task、Router）、扩容可选服务、扩容数量等操作配置。扩容前请根据安装部署手册完成节点初始化操作。

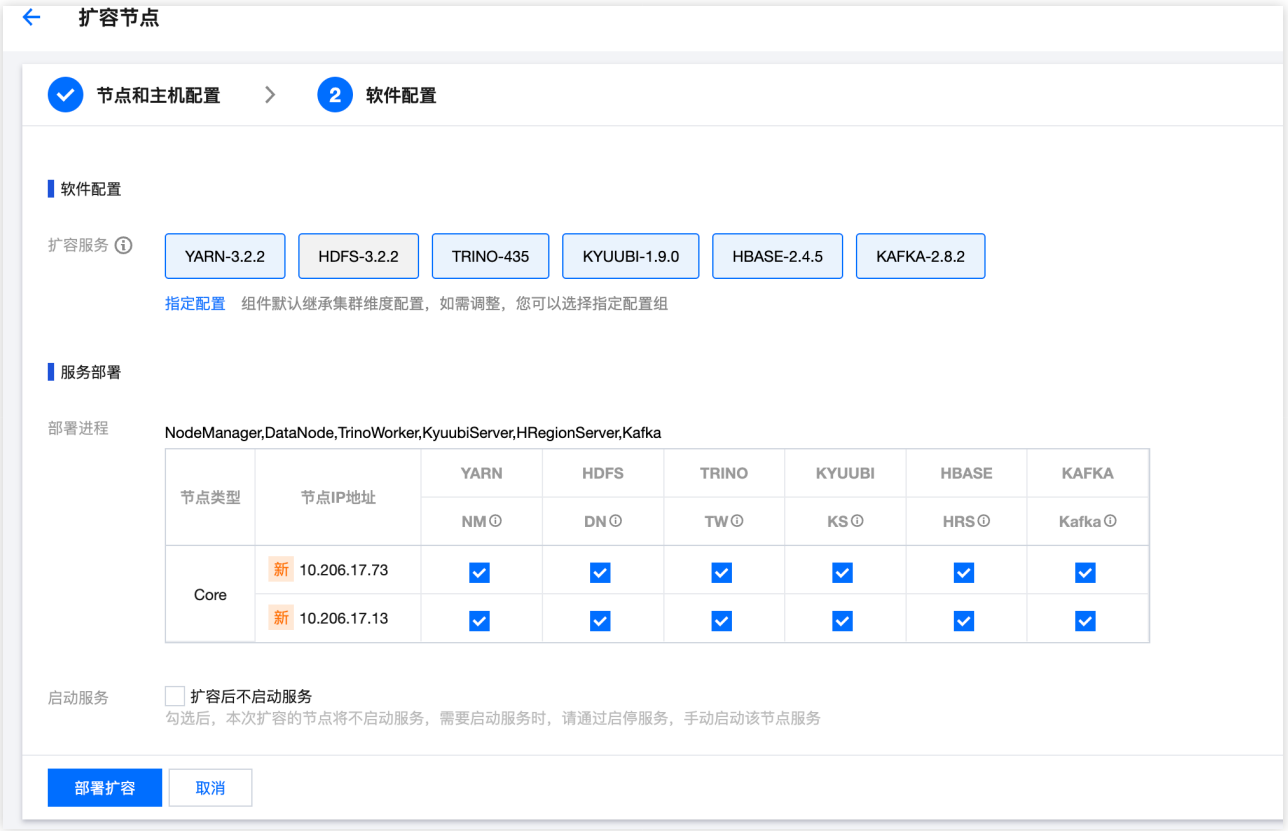
i. 扩容入口



ii. 扩容操作-节点和主机配置



iii. 扩容操作-软件配置



- 扩容服务：服务选择后，新增节点将默认部署服务客户端。
- 指定配置：找到需要指定配置的组件，然后为其选择想要继承的配置维度。
  - 若选择继承集群维度配置，则扩容节点将继承集群维度配置，且该节点将归属该节点类型默认配置组。
  - 若选择继承配置组维度配置，则扩容的节点将继承所选配置组配置，且该节点将属于所选配置组。
- 部署进程：是指不同节点类型选择扩容组件后对应部署的服务进程信息。如需调整部署进程，可以编辑进程。
- 扩容后不启动服务：扩容时在勾选此选项后，本次扩容的节点将不启动服务，需要启动服务时，请通过集群服务 > 选中的具体服务 > 角色管理 > 启动对应节点的服务角色。

3. 选择扩容节点所需的组件和数量后，单击部署扩容后，集群会开始扩容操作，扩容操作一般需要10 - 20分钟。

集群缩容

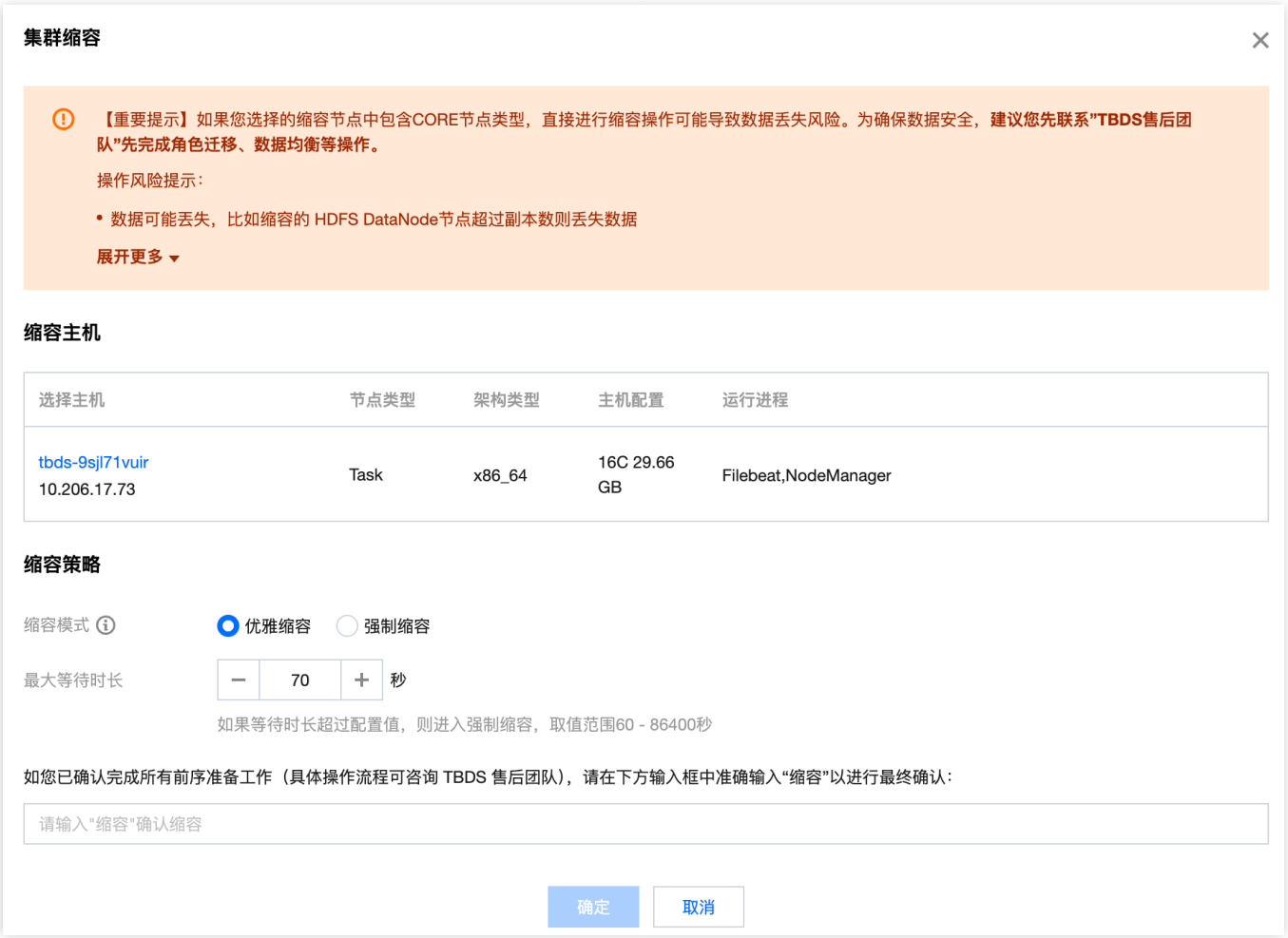
功能介绍

节点卸载功能支持用户通过 TBDS Manager 管理平台界面化操作缩容集群节点，TBDS Manager 管理平台将对符合缩容条件的集群中移除选中的。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。选择 更多 > 卸载集群进集群卸载流程。

2. 在集群详情页中选择集群资源 > 资源管理 > 缩容，根据业务需要选择需要缩容的节点类型（Core、Task、Router）选中缩容节点。缩容前请根据运维手册完成节点数据传输和节点准备操作。



配置项说明：

信息	详情
缩容模式	支持优雅缩容和强制缩容，优雅缩容仅支持 Yarn、Trino 和 Hbase。 - 优雅缩容：开启优雅缩容模式后，如果缩容动作触发时节点正在执行任务，节点不会立即释放，而是在自定义时间内等待任务执行完成后进行缩容；若自定义时间内任务未执行完成也将进行缩容。此模式支持的组件包括Yarn、HBase和Trino。请注意，Trino在Ranger、Kerberos和OpenLDAP集成场景下暂不支持优雅缩容。 - 强制缩容：在此模式下，系统将忽略节点缩容的校验结果和节点健康状态，直接强制进行节点缩容。这种方式适用于需要立即释放资源的紧急情况，但可能会影响正在运行的任务。
最大等待时间	在优雅缩容模式下有效，当超过最大等待时长后则强制缩容。

3. 平台将校验集群是否符合节点缩容的基本条件（缩容后服务角色要满足服务的最小节点数量），节点仅在满足全部基础条件的情况下可被缩容，选择确认缩容节点并点击确认。节点缩容流程不可逆，请谨慎操作。

4. TBDS Manager 将进入节点缩容流程，流程结束后节点将从集群中移除。

登录集群

经典集群提供了完整的存算组件，可通过客户端快速访问服务，常用登录方式如下。

远程登录软件登录（本地系统为 Windows ）

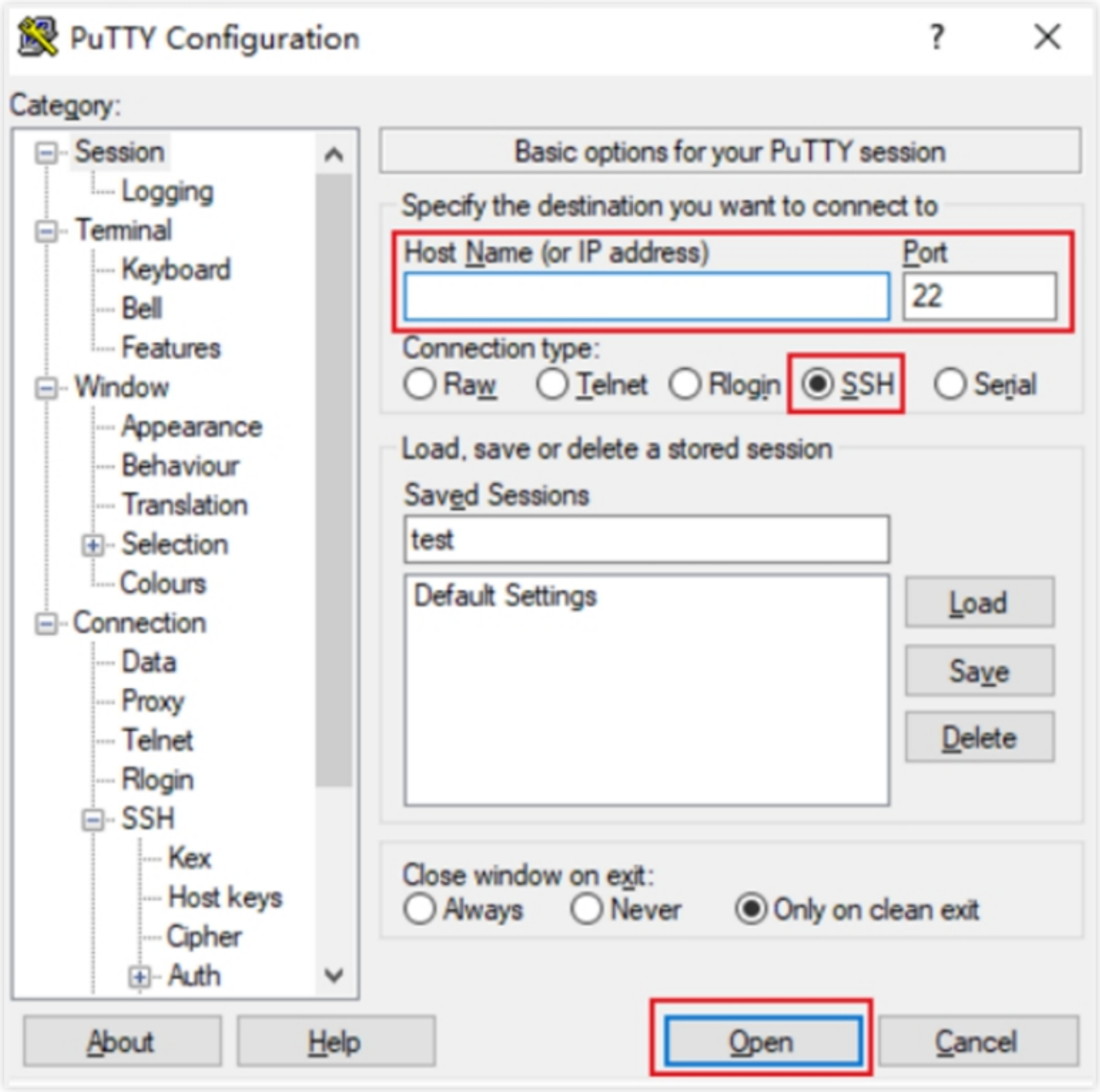
以 Xshell 为例，介绍本地为 Windows 系统的电脑如何使用远程登录软件通过密码登录 TBDS 集群。

适用本地操作系统

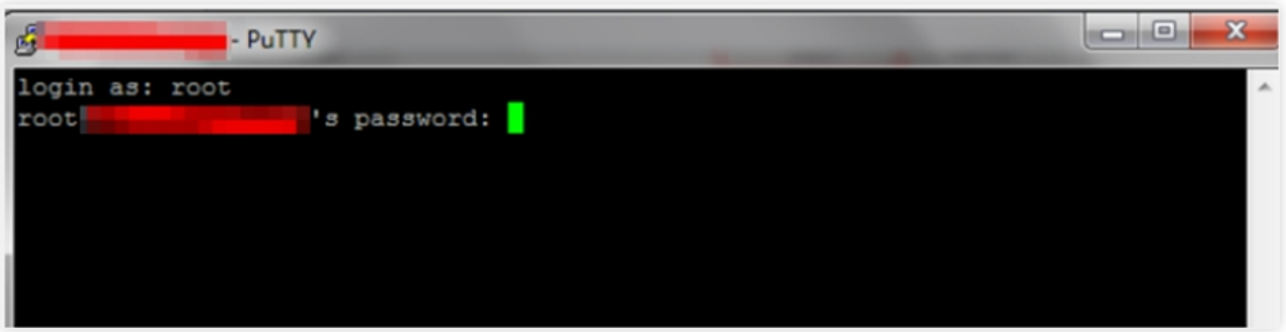
Windows

使用密码登录

1. 安装 Windows 远程登录软件，即 PuTTY。PuTTY 的获取方式参见获取链接。
2. 打开 PuTTY 客户端，在 PuTTY Configuration 窗口中输入以下内容，并单击 Open，创建一个新对话。如下图所示：
    - Host Name：TBDS 集群的 IP，登录TBDS 大数据管理平台，可在列表页及详情页中获取集群 IP。
    - Port：云服务器的远程登录端口，按操作系统 SSH 端口地址填写。请确保云服务器远程登录端口已开放。
    - Connection type：选择**【SSH】**。



3. 在 PuTTY 会话窗口中，输入已获取的管理员账号，按 Enter 键。
4. 输入已获取的登录密码，按 Enter 键，即可完成登录。如下图所示：



使用SSH登录（本地系统为 Linux/Mac OS ）

介绍本地为 Linux/Mac OS 系统的电脑通过 SSH 登录 TBDS 集群。

适用本地操作系统

Linux 或 Mac OS

使用密码登录

1. Mac OS 用户请打开系统自带的终端（Terminal）并执行以下命令，Linux 用户请直接执行以下命令：

   ssh <username>@<hostname or IP address>

   - username：即用户账号，例如 root。
   - hostname or IP address：为您的 TBDS 节点 IP 或自定义域名。

2. 输入已获取的密码（此时仅有输入没有显示输出），按 Enter 键，即可完成登录。

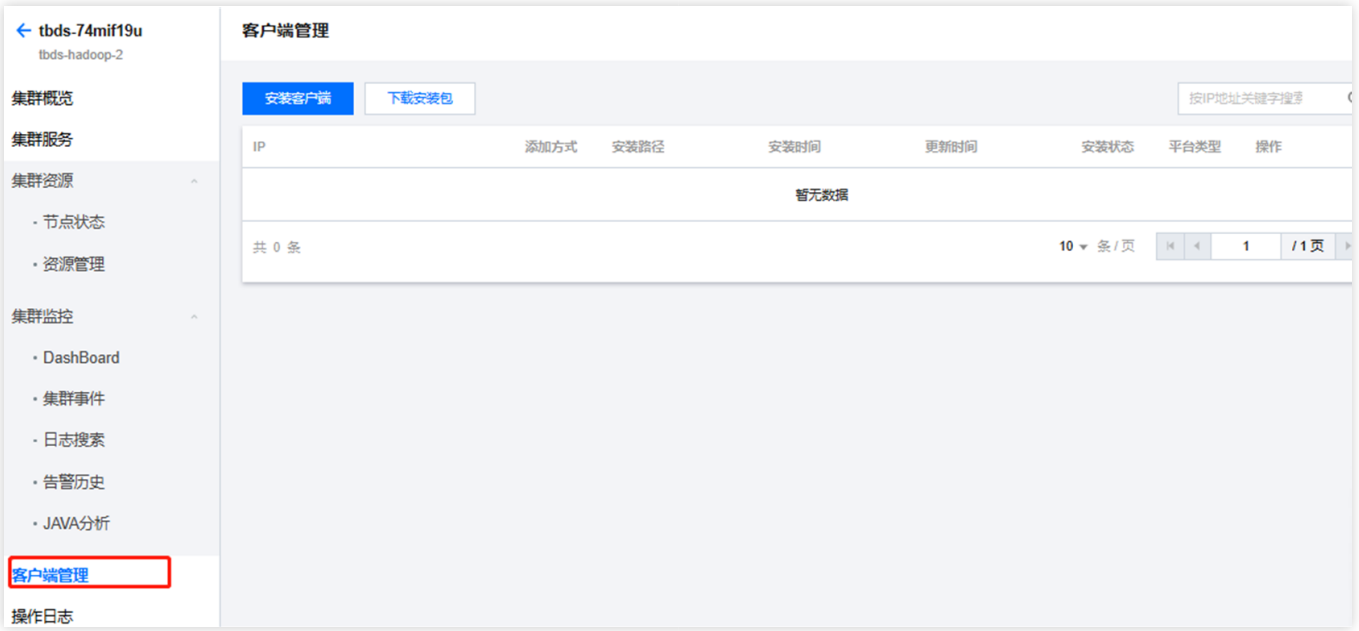
客户端管理

功能介绍

客户端管理提供了集群外部的远程客户端安装信息，包括 IP、添加方式、安装路径、安装时间、更新时间、安装状态、平台类型信息，同时支持集群内安装包的下载和集群外客户端的安装。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 单击左侧客户端管理，即可进入客户端管理页。



下载安装包：可手动下载集群内客户端安装包，下载节点、路径、客户端服务支持可配置，同时也支持仅下载客户端配置。

安装客户端：输入节点 IP、端口、用户名、密码等信息，安装远程客户端。

安装客户端
安装客户端前，请确保目标主机满足安装所需环境，请参考集群运维指南操作
节点类型 *
x86
arm
IP/端口
请输入IP
：
请输入端口号
用户名 *
请输入用户名
密码 *
请输入密码
连通测试
开始测试
安装到路径 *
/usr/local/service
客户端选择
请选择
安装
取消

集群服务
服务健康状态

功能介绍

服务列表展示了集群已安装的服务，以及服务的健康状态、配置状态、版本信息等。同时提供了服务便捷运维管理工具，包含通的服务操作以及部分服务特有的指令类运维操作。

服务健康状态

服务健康状态展示当前服务的运行状态是否正常，由各角色的健康状态聚合而成。

组件健康状态	健康状态说明	组件聚合规则
绿色：良好	服务运行正常	全部角色实例健康状态是良好。
橙色：存在隐患	服务可用，部分角色实例健康状态为不可或存在隐患，需关注处理。	该组件某角色的部分实例健康状态为不可或存在隐患。例如，HDFS 有1个 NameNode 角色实例和2个 DataNode 角色实例，其中1个 DataNode 角色实例健康状态为不可，另1个 DataNode 角色实例和 NameNode 角色实例健康状态为良好，HDFS 健康状态为存在隐患。

组件健康状态	健康状态说明	组件聚合规则
红色：不可用	服务不可用，某角色的全部实例健康状态不可 ，请及时处理。	该组件某角色的全部实例健康状态不可 。例如，HDFS 有1个 NameNode 角色实例和2个 DataNode 角色实例，其中2个 DataNode 角色实例健康状态为不可用，1个 NameNode 角色实例的健康状态为良好，HDFS 健康状态为不可用。
灰色：未知或者未探测	服务健康状态未知或未探测。 无进程组件 健康状态为未探测，有进程组件如进 维护模式或操作状态已停止为未探测； 有进程组件如无法正确获取角色实例健康状态信息为未知。如排查业务 问题， 需关注。	1. 该组件全部角色实例健康状态非存在隐患或不可 的角色，且至少有 个角色实例健康状态为未知。例如，HDFS 有1个 NameNode 角色实例和2个 DataNode 角色实例，其中1个 DataNode 角色实例健康状态为未知，另1个 DataNode 角色实例和 NameNode 角色实例健康状态为良好，HDFS 健康状态为未知； 1. 该服务全部角色实例健康状态为未探测。当服务全部角色实例进 维护模式或操作状态已停止时，其健康状态不做探测。 1. 该组件无进程，则其健康状态不做探测，如 Iceberg、Hudi、Flink 等。

服务状态

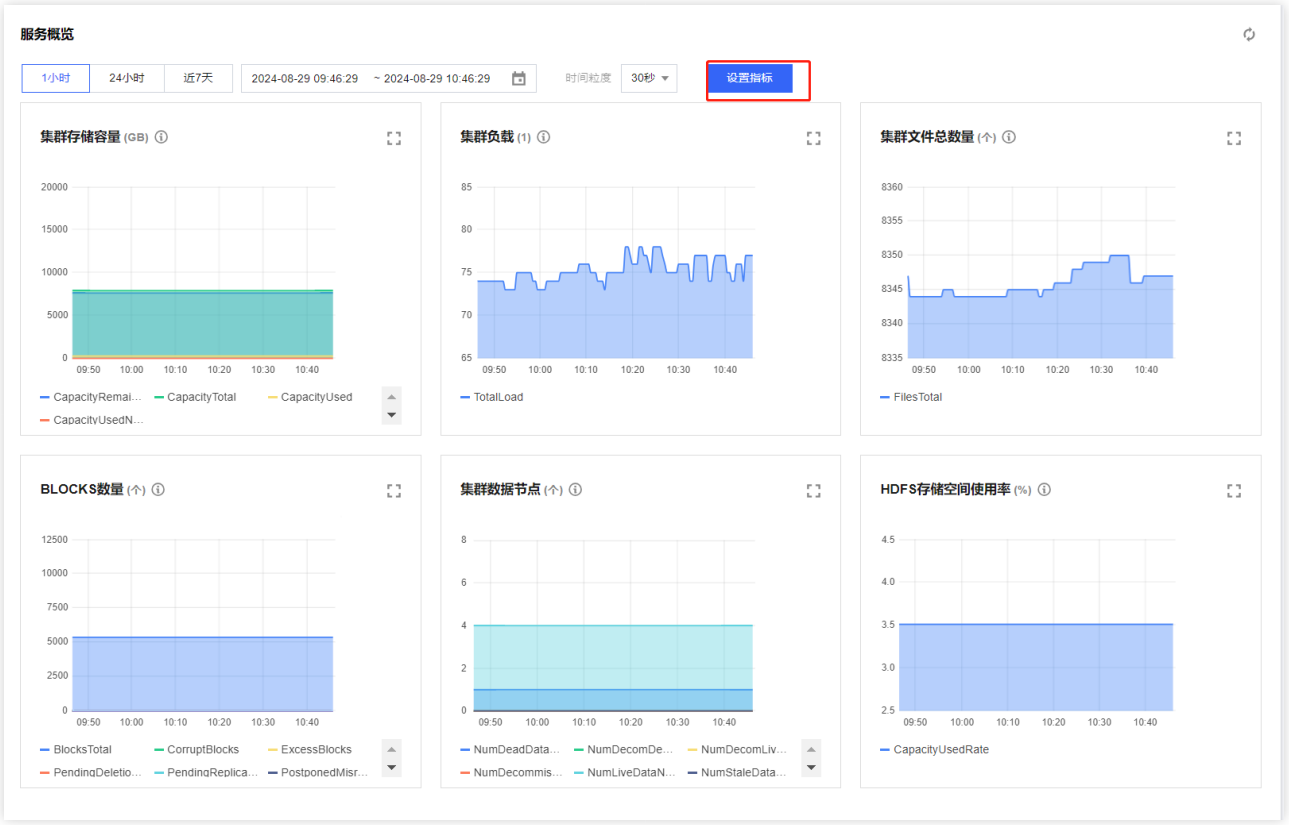
功能介绍

服务状态提供对集群上安装的主要服务的详细监控功能，包括 HDFS、Yarn、Hive、ZooKeeper、Spark、HBase和 Trino 等。本节为您介绍通过控制台查看集群服务状态操作。

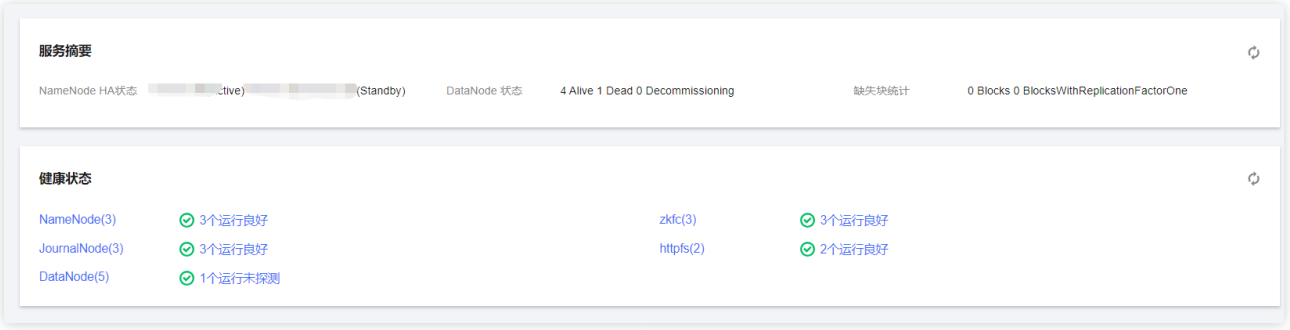
操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进[redacted]集群详情页。
2. 在集群详情页面中选择集群服务，单击对应组件右上角操作 > 服务状态，以 HDFS 为例。

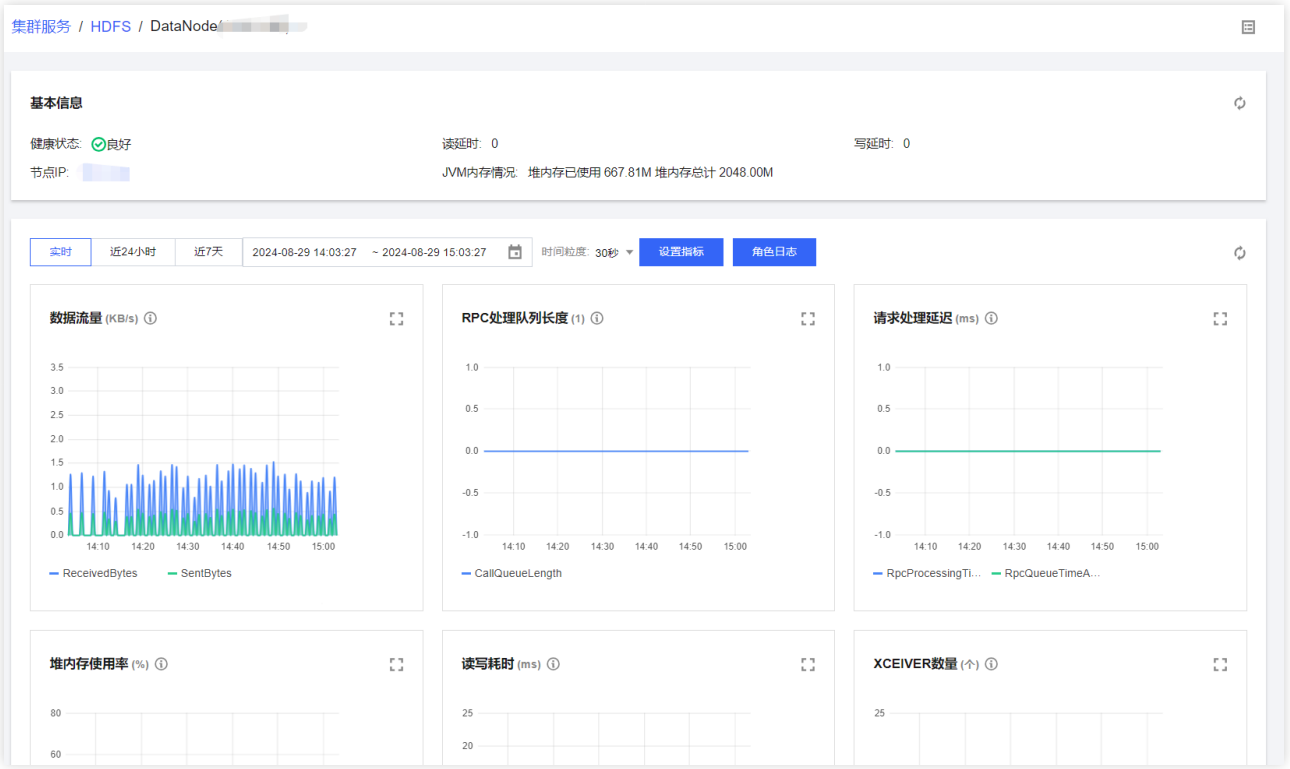
3. 服务状态页面提供了三部分服务维度的监控视图，分别为服务摘要、健康状态、服务概览。因各服务组件服务不同，展示部分维度不同。
4. 服务概览可直观查看对应时间段服务组件的各项指标及指标各项统计规则，系统默认展示6个指标项，可单击设置指标自定义展示指标。


服务概览
1小时
24小时
近7天
2024-08-29 09:46:29 ~ 2024-08-29 10:46:29
时间粒度
30秒
设置指标
集群存储容量 (GB)
20000
15000
10000
5000
0
09:50 10:00 10:10 10:20 10:30 10:40
CapacityRemai...
CapacityTotal
CapacityUsed
CapacityUsedN...
集群负载 (1)
85
80
75
70
65
09:50 10:00 10:10 10:20 10:30 10:40
TotalLoad
集群文件总数量 (个)
8360
8355
8350
8345
8340
8335
09:50 10:00 10:10 10:20 10:30 10:40
FilesTotal
BLOCKS数量 (个)
12500
10000
7500
5000
2500
0
09:50 10:00 10:10 10:20 10:30 10:40
BlocksTotal
CorruptBlocks
ExcessBlocks
PendingDeletio...
PendingReplica...
PostponedMisr...
集群数据节点 (个)
8
6
4
2
0
09:50 10:00 10:10 10:20 10:30 10:40
NumDeadData...
NumDecomDe...
NumDecomLiv...
NumDecommis...
NumLiveDataN...
NumStaleData...
HDFS存储空间使用率 (%)
4.5
4.0
3.5
3.0
2.5
09:50 10:00 10:10 10:20 10:30 10:40
CapacityUsedRate

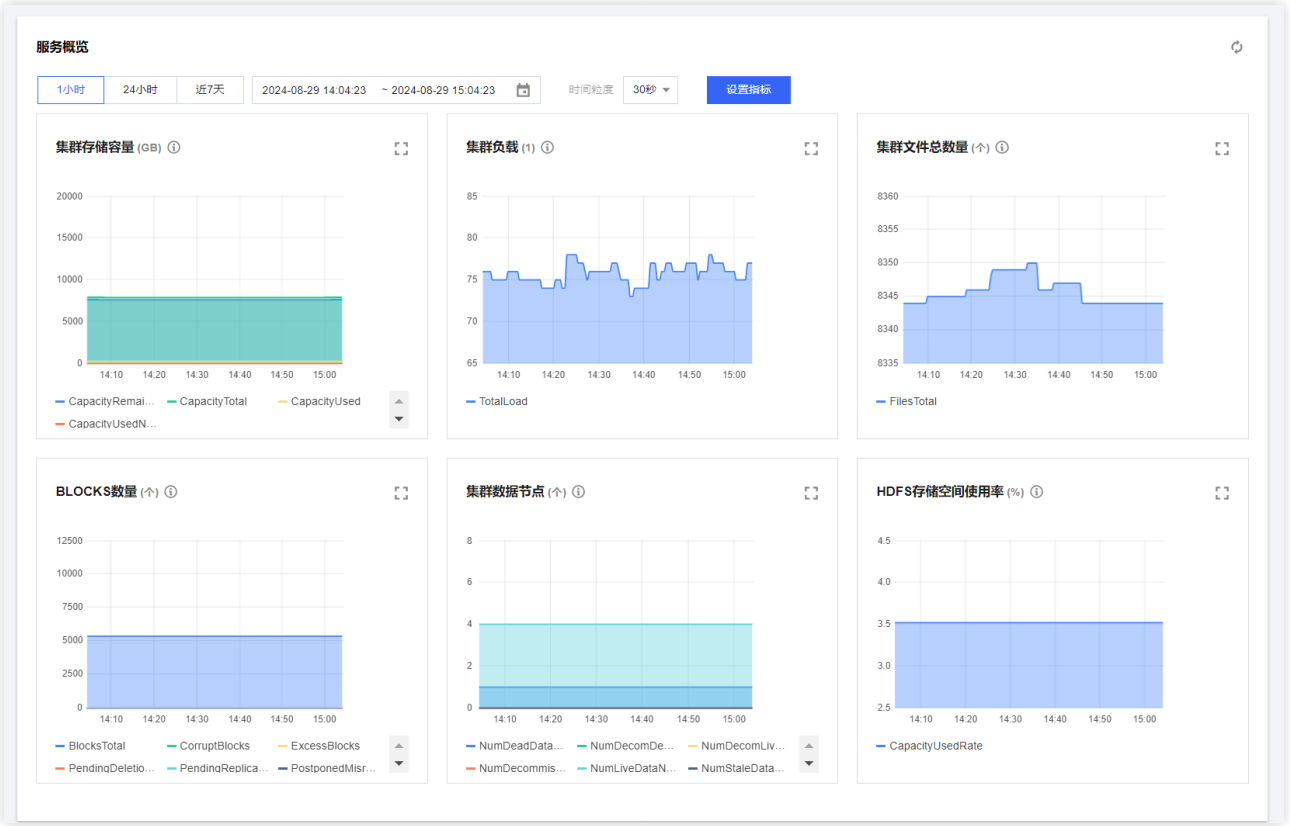
5. 服务摘要展示服务当前整体使状态。
6. 健康状态展示当前服务各组件运行概况。单击角色名称或运行概况可跳转至角色管理或角色状态页。



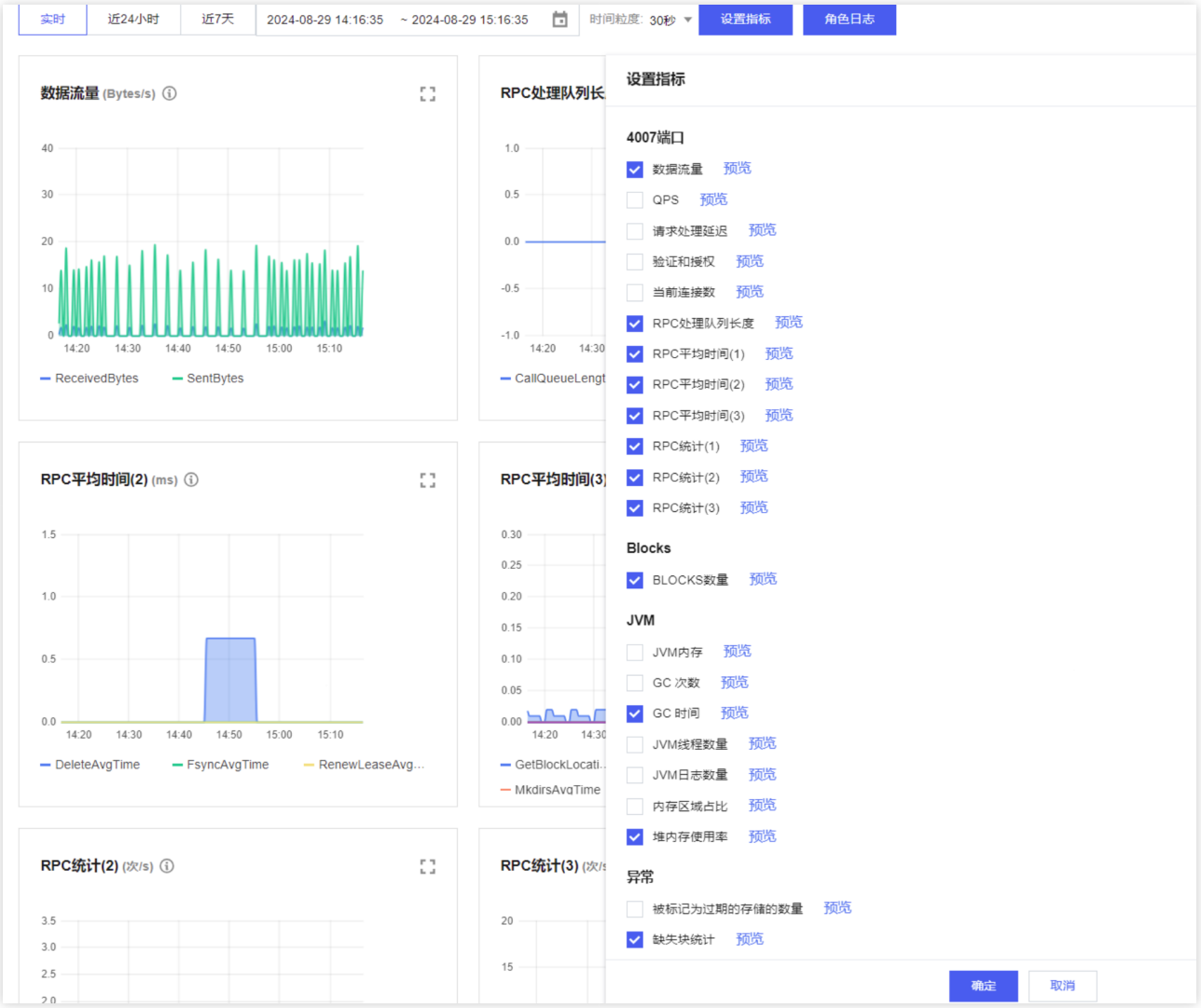
进某角色状态页面，可单击设置指标自定义展示指标。



7. 服务概览可查看集群维度统计指标，单击设置指标可自定义展示指标。



- 服务监控默认展示 HDFS 服务组件，您可手动调整查看其它服务组件。
- 因各服务组件服务性质不同，所以服务监控维度部分有所不同，如 HBase 支持表级监控维度部分。


实时
近24小时
近7天
2024-08-29 14:16:35 ~ 2024-08-29 15:16:35
时间粒度: 30秒
设置指标
角色日志
数据流量 (Bytes/s)
ReceivedBytes
SentBytes
RPC处理队列长
CallQueueLengt
RPC平均时间(2) (ms)
DeleteAvgTime
FsyncAvgTime
RenewLeaseAvg...
RPC平均时间(3)
GetBlockLocati...
MkdirsAvgTime
RPC统计(2) (次/s)
RPC统计(3) (次/
设置指标
4007端口
数据流量 预览
QPS 预览
请求处理延迟 预览
验证和授权 预览
当前连接数 预览
RPC处理队列长度 预览
RPC平均时间(1) 预览
RPC平均时间(2) 预览
RPC平均时间(3) 预览
RPC统计(1) 预览
RPC统计(2) 预览
RPC统计(3) 预览
Blocks
BLOCKS数量 预览
JVM
JVM内存 预览
GC 次数 预览
GC 时间 预览
JVM线程数量 预览
JVM日志数量 预览
内存区域占比 预览
堆内存使用率 预览
异常
被标记为过期的存储的数量 预览
缺失块统计 预览
确定
取消

WebUI 访问管理

功能介绍

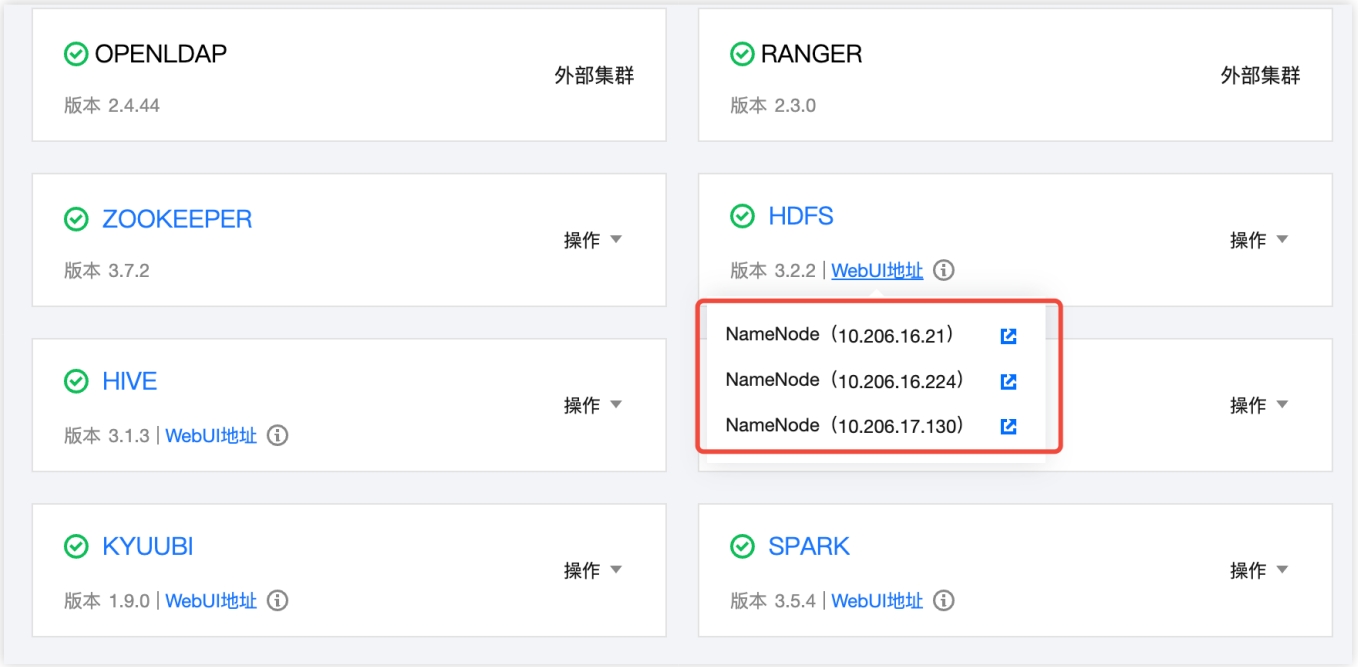
WebUI 入口功能是 TBDS Manager 提供的组件原生 UI 访问能力，通过配置WebUI 访问地址为 Master 节点的外网 IP（建议及时配置安全策略），可以快捷访问组件原生 UI。如果集群内网与企业网络互通，可关闭该外网 IP，直接通过内网访问组件原生 UI。

前提条件

TBDS 集群创建时，如果没有配置 WebUI 访问地址，将不能通过组件管理页面的原 WebUI 访问地址进相关组件的 WebUI 界面。本节介绍来在没有开启 Master 节点公网的集群如何访问组件 WebUI 和完成配置 WebUI 访问地址的集群通过外网访问两种 WebUI 访问方式。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 在集群详情页中选择集群服务，然后单击对应的组件卡页下方WebUI 地址即可访问。



3. 也可以进入组件服务后，通过“WebUI地址”链接进行访问。

WebUI 访问：

TBDS 默认在访问组件原生 WebUI 时经过 Knox，访问链接详情可参见以下 WebUI 访问链接列表。

WebUI 访问链接：

组件名	链接
HDFS UI	https://{集群Master节点IP}:30002/gateway/tbds/hdfs
YARN UI	https://{集群Master节点IP}:30002/gateway/tbds/YARN
Hue UI	https://{集群Master节点IP}:30002/gateway/tbds/hue
Hive UI	https://{集群Master节点IP}:30002/gateway/tbds/hive
HBase UI	https://{集群Master节点IP}:30002/gateway/tbds/HBase/webui
Presto UI	https://{集群Master节点IP}:30002/gateway/tbds/presto/
Alluxio UI	https://{集群Master节点IP}:30002/gateway/tbds/alluxio/
Impala UI	https://{集群Master节点IP}:30002/gateway/tbds/impalastore/
Spark UI	https://{集群Master节点IP}:30002/gateway/tbds/sparkhistory/
Tez UI	https:///{集群Master节点IP}:30002/gateway/tbds/tez/
StarRocks UI	https:///{集群Master节点IP}:8030
ElasticSearch UI	http://{集群Master节点IP}:9200
Kibana UI	http://{集群Master节点IP}:5601

若不通过 Knox 代理访问 yarn webui，即访问原生 yarn 原生 webui，则需做如下前置操作：

在访问的节点配置 yarn 所在集群的 hostname 域名解析。

配置管理
配置更新

功能介绍

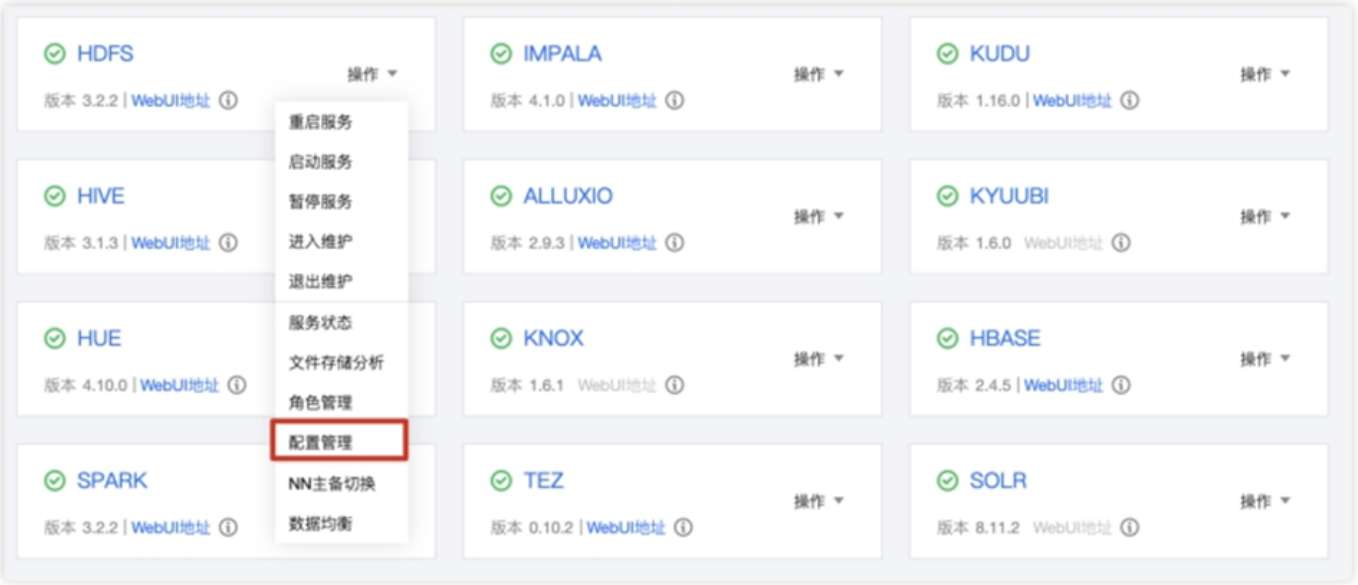
配置管理支持 HDFS、YARN、HIVE、SPARK 等常用开源组件的关键配置参数的修改，可以根据实际需要以集群维度、节点维度、配置组维度对服务的配置进行修改。本节为您介绍通过控制台配置各服务参数的操作。

- 在控制台配置管理中删除自定义配置文件，基于安全性考量，客户端不会同步删除操作。
- 如果您有删除客户端上的自定义配置文件的场景，可使用集群脚本的能力，对存量集群批量执行操作。
- 修改配置后，如果角色管理 > 配置状态变为“配置过期”，则需要重启此服务使配置生效。

操作步骤

编辑配置项

1. 登录 TBDS Manager 管理平台，在集群列表中选择对应的集群单击集群服务进入集群服务列表。
2. 在集群服务列表中，选择待修改配置服务面板右上角操作 > 配置管理。



3. 进入配置管理页后，根据需要选择配置维度范围，默认选择集群维度。

- 若需要对集群内所选服务的所有节点进行参数配置，选择集群维度。
- 若需要对所选服务指定多个节点进行参数配置，可创建配置组，并选择对应配置组。

- 若需要对所选服务某个节点进行参数配置，选择节点维度。

4. 若您想搜索某个配置项或者缩小配置项查找范围，可通过左侧筛选器进行过滤。
5. 根据需要选择配置文件，单击编辑配置进入编辑状态，根据需要进行新增、编辑、删除配置项等操作。

- 选择需要修改的参数填入新的参数值，如有需要可单击复原恢复为原始值，也可单击默认值恢复到系统推荐默认值。
- 部分参数支持删除操作，如需删除该配置，选择删除 > 确认即可。
- 若该文件下没有您想要配置的参数，可单击新增配置项打开新增配置项弹框页面，填写新增配置项的参数名、参数值。

6. 确认无误后单击保存配置，配置下发成功后单击重启服务，修改配置项设置完成。

- 若修改的是服务进程配置，保存后需要重启服务才可配置生效。
- 若修改的是客户端配置，保存后无需重启服务即可生效。

新增配置文件

1. 登录 TBDS Manager 管理平台，在集群列表中选择对应的集群单击服务进入集群服务列表。
2. 在集群服务列表中，选择待修改配置服务面板右上角操作 > 配置管理。
3. 若没有您想要配置的配置文件，可单击右侧新增配置文件进入配置文件配置页面，填写配置文件。
4. 单击保存配置后，参数下发并更新配置文件列表中的配置文件名。

5. 自定义配置文件下发生效后支持修改和删除操作。

配置维度对比

在多个维度修改配置后，如需对比集群维度、配置组维度和节点维度的配置参数差异，可单击切换至维度对比列表，即可查看各维度的配置差异值。若您需要使某维度配置与其上一级维度配置一致，可以覆盖差异值并保存修改。

配置内容说明

每个组件具体配置项详细含义，可参见对应官方文档，比如Elasticsearch 的yml文件配置项说明：

参数	说明	默认值
indices.fielddata.cache.size	指定分配到字段数据的 Java 堆空间的百分比	15%

参数	说明	默认值
indices.query.bool.max_clause_count	指定 Lucene 布尔查询中允许的子句的最大数量	1024
reindex.remote.whitelist	远程 Elasticsearch 集群访问地址白名单	[]
thread_pool.bulk.queue_size	文档写入队列大小，适用于5.6.4版本	1024
thread_pool.write.queue_size	文档写入队列大小，适用于6.4.3及以上版本	1024
thread_pool.search.queue_size	文档搜索队列大小	1024
http.cors.enabled	跨域资源共享配置项，true表示启用跨域资源访问，false表示不启用	false
http.cors.allow-origin	域资源配置项，可设置接受来自哪些域名的请求	“”
http.cors.max-age	获取的 CORS 配置信息在浏览器中的缓存时间	1728000（20天）
http.cors.allow-methods	跨域允许的请求方法	OPTIONS,HEAD,GET,POST,PUT,DELETE
http.cors.allow-headers	跨域允许的请求头信息	X-Requested-With,Content-Type,Content-Length
http.cors.allow-credentials	是否允许响应头中返回 Access-Control-Allow-Credentials 信息	false
xpack.watcher.enabled	true表示开启 X-Pack 的 Watcher 功能	true

官方文档 Elasticsearch Reference [5.6]

如果有其他配置项自定义设置需求，可通过售后支持。

配置状态

功能介绍

组件配置修改并保存下发后，可通过配置状态感知是否存在服务配置下发失败或配置过期。 持查看配置失败和配置过期详情，以便于了解各角色具体配置变更项。

名称解释

已同步：无配置修改，或配置项修改后已重启服务。
配置过期：配置项修改且下发后，未重启服务导致配置未生效。
配置失败：配置项修改后，存在节点配置下发失败。

状态说明

1. 角色实例的配置状态有3种：已同步、配置失败、配置过期。
    - 初始配置状态为已同步，当角色进程配置下发失败时，角色配置状态变为配置失败。
    - 配置下发成功时，角色配置状态变为配置过期。
    - 非配置失败的前提下，重启服务成功后，角色配置状态变为已同步。
2. 客户端的配置状态有2种：已同步、配置失败。
    - 初始配置状态为已同步，当客户端配置或角色进程配置下发失败时，客户端配置状态变为配置失败。
    - 配置下发成功时，客户端配置状态变为已同步。
3. 组件的配置状态只展示2种：配置失败、配置过期。
    - 初始状态不显示，当角色实例或客户端配置失败时，组件配置状态展示为配置失败。
    - 当角色实例配置过期时，组件配置状态展示为配置过期。
    - 当角色实例、客户端同时存在配置失败和配置过期时，组件配置状态优先展示配置失败。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进 集群详情页。
2. 在集群详情页中选择集群服务，选择配置状态为配置失败或配置过期的组件卡片。
3. 以配置过期为例，进组件详情页后，可以通过页面右上角的配置状态图标查看过期配置详情。

4. 确认需要生效的过期配置后，重启相应服务即可。

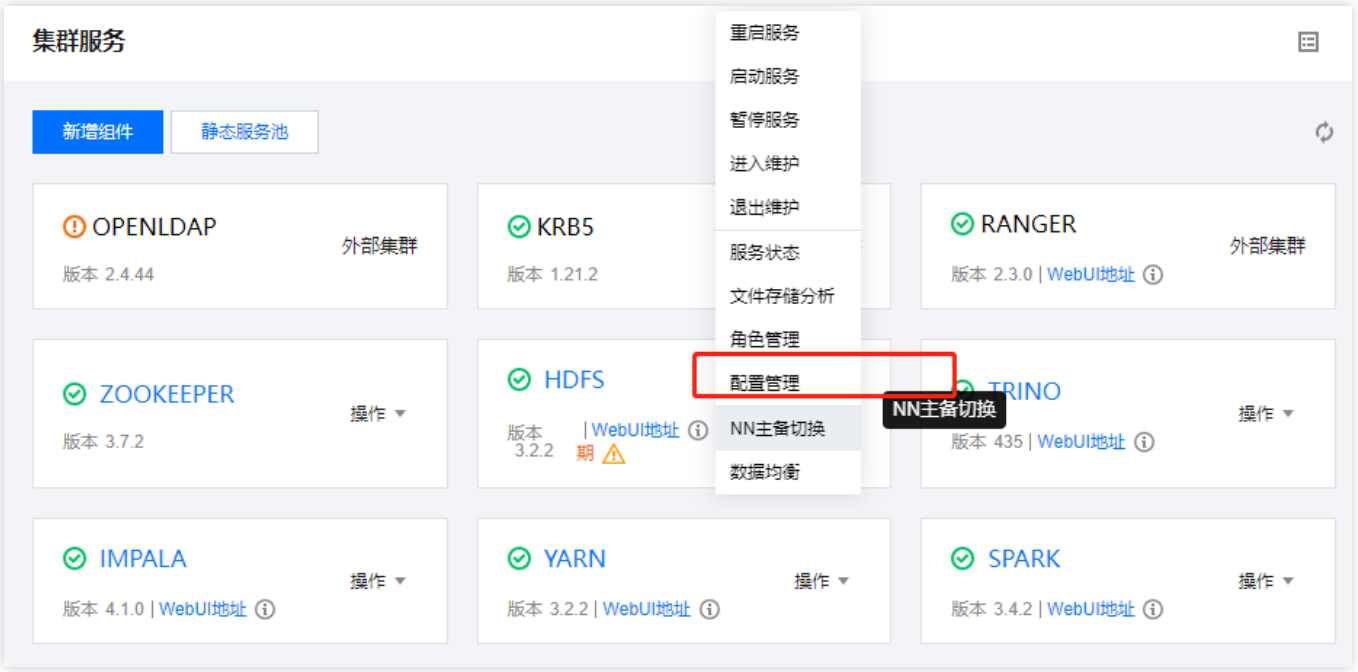
配置回滚

功能介绍

TBDS 支持在控制台对各组件参数新增、修改、删除配置项等操作进行配置回滚，本节介绍如何通过控制台回滚各组件参数配置。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中选择对应的集群单击详情进入集群详情页。
2. 在集群详情页中选择集群服务单击对应的组件卡页右上角操作 > 配置管理，以 HDFS 为例。



3. 在配置管理页选择配置历史，默认展示当前组件的配置历史，可切换查看所有组件的配置历史。单击详情可以看到配置变更前后的参数值对比，单击回滚可对该条记录的参数配置变更回滚。选择回滚 > 确认回滚，回滚成功重启组件，稍作等待回滚即可生效。

新增、修改、删除配置项持回滚，新增配置文件和删除新增配置件不持回滚。

配置组管理

功能介绍

配置组用于将部署同一服务的不同规格或用途的节点进行分组配置管理，使用配置组对服务配置进行管理规则如下：

- 集群维度修改配置项，如果该配置项在配置组维度和节点维度未独立修改过，默认会覆盖组维度和节点维度的配置项。
- 配置组维度修改配置项，如果组内节点该配置项未独立修改过，默认会覆盖该组所有节点的配置项。修改后集群维度配置将不会覆盖该配置组配置。
- 节点维度修改配置项，只更新该节点的配置项。修改后从集群维度和配置组维度下发将不会覆盖该节点配置。
  [illegible]个节点只能属于[illegible]个配置组，组与组之间节点不重复。

操作步骤

1. 登录 TBDS Manager 管理平台 ，在集群列表中选择对应的集群单击详情进入集群详情页。
2. 在集群详情页中选择集群服务单击对应的组件卡页右上角操作 > 配置管理，以 HDFS 为例。
3. 单击配置管理，设置“维度”为配置组，并选择对应的配置组，单击管理配置组，即可进入“管理配置组”页。
   集群创建完后，每个组件会生成以节点类型划分的若干默认的配置组，默认配置组仅支持修改，不支持删除。默认配置组初始化时继承集群维度配置，可根据需要增加新的配置组，每个组件配置组不超过15个。

4. 在配置组管理页，可查看当前组件的所有配置组，同时，支持添加、修改和删除配置组。

- 新增配置组可选择继承配置组配置，若不选继承配置组，则默认继承集群维度配置参数。
- 删除自定义配置组后，该配置组节点将移动到同节点类型的默认配置组中。

5. 若配置组与集群维度配置不一致，可在集群维度直接单击保存，即可强制覆盖配置组配置；同理，若配置组中存在节点配置与当前配置组配置不一致，可在配置组维度直接单击保存，即可强制覆盖所属节点配置。

Trino资源组配置

应用场景

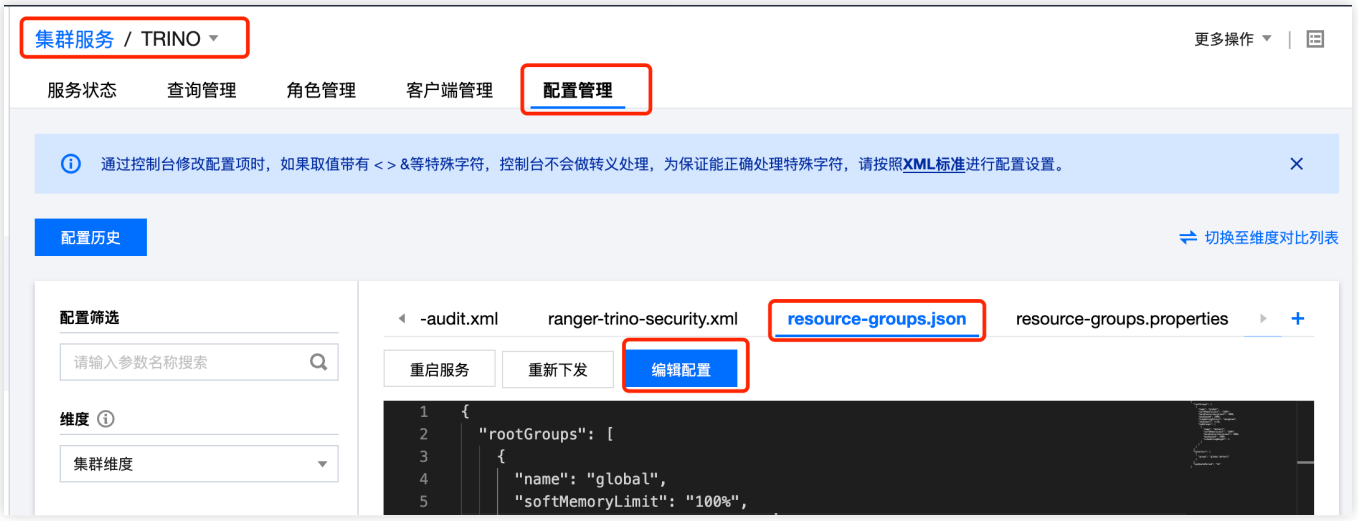
Trino 资源组典型应用场景如下：

1. 保障核心的业务的资源配额。
2. 多租户场景的资源隔离，为不同的用户或团队创建独立的资源组，并设置资源限制，避免单个用户或团队过度占用资源，影响其他用户的体验。

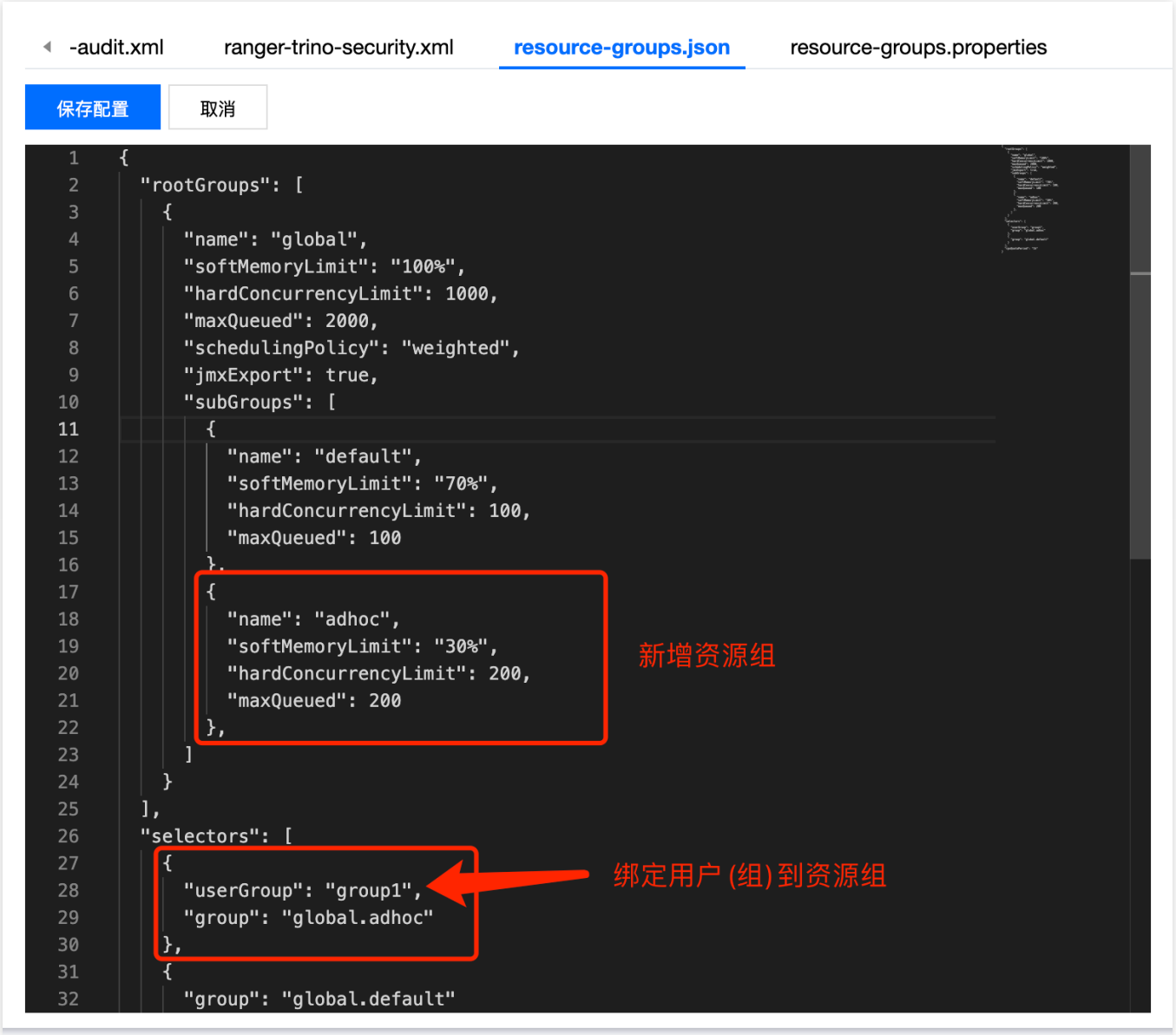
配置方式

当前资源组的配置需要用户详细阅读 Trino 官方文档：Resource Group 配置，了解 resource-groups.json 的配置方式后再进行配置。
当前默认已经开启了 Resource Group，但是只有一个资源组（队列），所有任务跑在默认的资源组下，并没有隔离和限制资源配额。
修改资源组配置方式：

1. Trino 服务页面，单击配置管理，配置文件选择 resource-groups.json



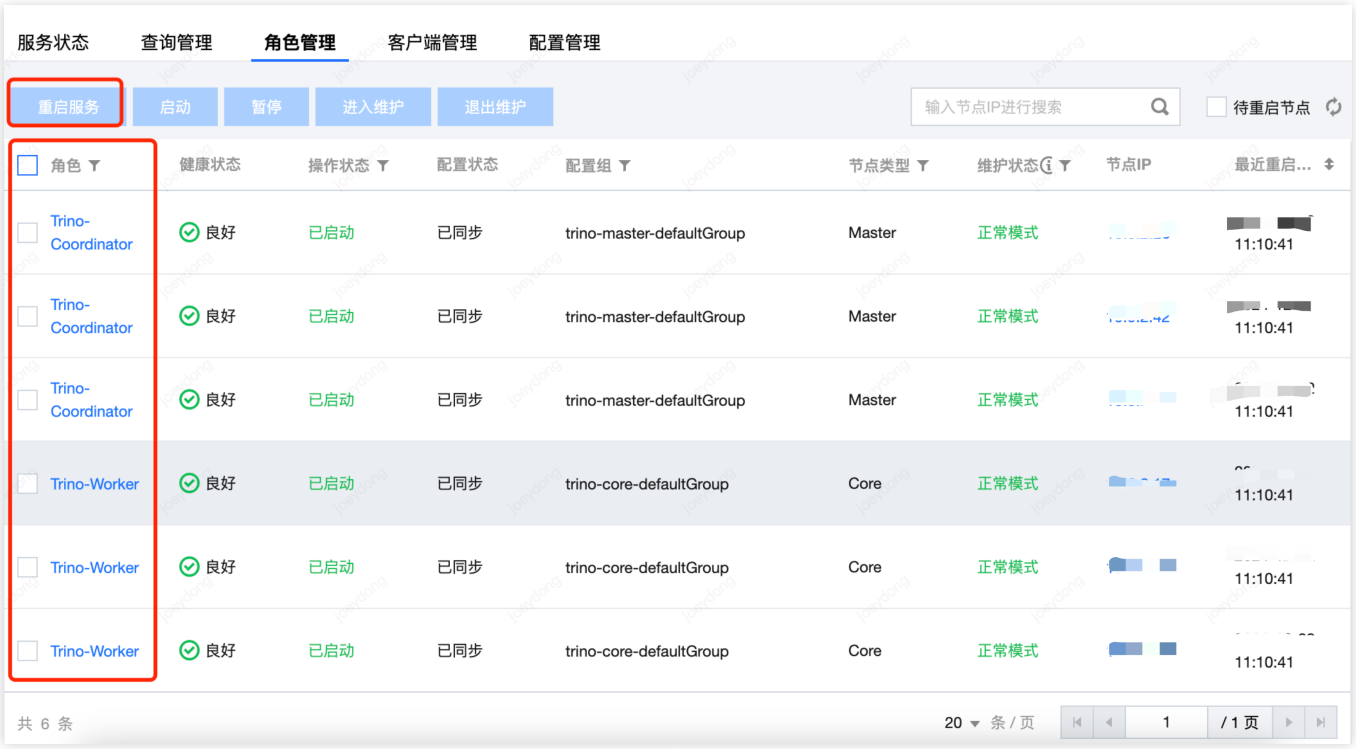
2. 单击编辑配置按钮，对资源组配置进行修改后，点击保存配置。




```
{
  "rootGroups": [
    {
      "name": "global",
      "softMemoryLimit": "100%",
      "hardConcurrencyLimit": 1000,
      "maxQueued": 2000,
      "schedulingPolicy": "weighted",
      "jmxExport": true,
      "subGroups": [
        {
          "name": "default",
          "softMemoryLimit": "70%",
          "hardConcurrencyLimit": 100,
          "maxQueued": 100
        },
        {
          "name": "adhoc",
          "softMemoryLimit": "30%",
          "hardConcurrencyLimit": 200,
          "maxQueued": 200
        },
      ]
    }
  ],
  "selectors": [
    {
      "userGroup": "group1",
      "group": "global.adhoc"
    },
    {
      "group": "global.default"
```

3. 重启 Trino 服务以使配置生效。

> 注意：
>
> 如果改动后的 resource-groups.json 格式异常，服务启动后会因配置问题无法启动，重启后需要关注 Trino 服务的状态是否正常。



常用配置项

配置项	释义
name	资源组名称
softMemoryLimit	内存配额，可以设置为百分比如“10%”，也可设置为绝对值如128GB
hardConcurrencyLimit	最大并发查询数
maxQueued	最大排队查询数
subGroups	定义子级资源组
schedulingPolicy	调度策略。默认为 fair。 - fair：各个子队列需要轮流执行查询，在排队的查询按先进先出的顺序执行。 - weighted：各个子队列按照配置的权重获得执行资源，排队中的查询按照优先级顺序执行。

新增组件

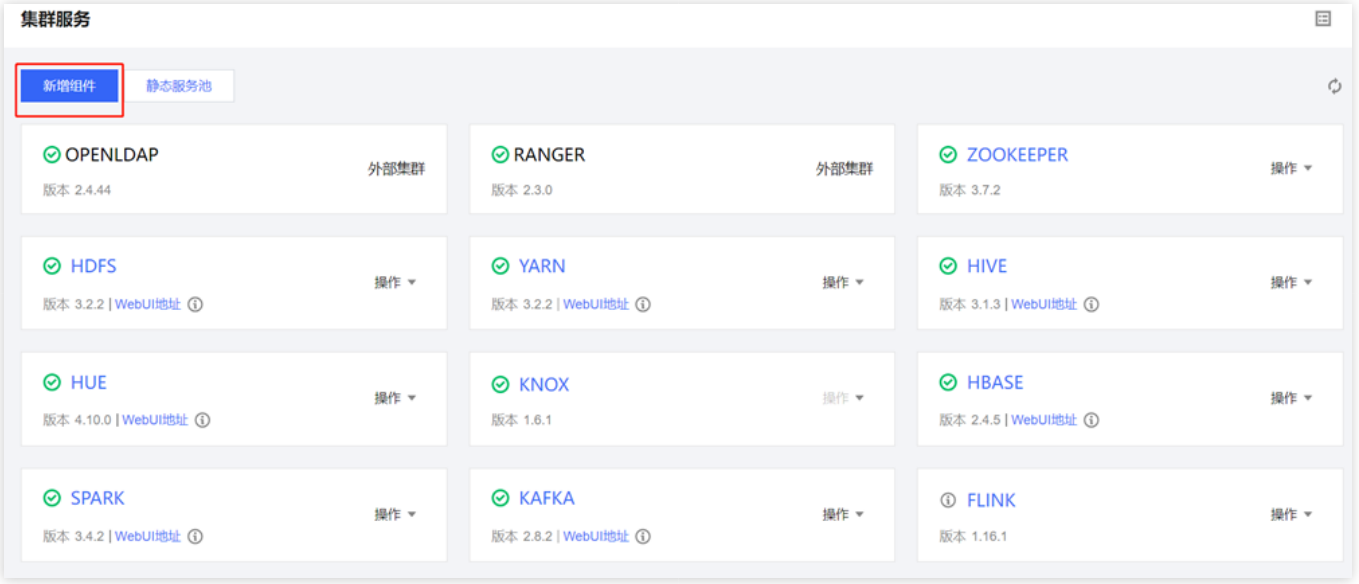
功能介绍

TBDS 支持在控制台新增组件，本节介绍通过控制台新增组件的操作。

> 注意：
>
> - 需通过集群服务中的功能对组件进行管理。登录节点直接做的组件变更（如新增组件），将不会同步到控制台，无法进行进一步的管理。
> - 新增的组件不支持跨 TBDS 产品版本新增，仅支持新增当前 TBDS 产品版本中版本的组件。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 在集群详情页中选择集群服务 > 新增组件新增集群中未安装的组件。



选择集群服务中的新增组件并安装 Hive 组件时，Hive 元数据库将默认使集群安装时配置的数据库。

3. 勾选组件后，单击确认即可，新增组件操作通常需要10-20分钟。

卸载组件

功能介绍

TBDS 持在控制台卸载组件，本节介绍通过控制台卸载组件的操作。

> 注意：
>
> 需通过集群服务中的功能对组件进管理。登录节点直接做的组件变更（如卸载组件），将不会同步到控制台，无法进进一步的管理。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群服务 > 操作 > 卸载组件卸载集群中安装的组件。提示：HDFS，YARN，Ranger，LDAP，Knox，ZK，Kerberos 组件作为集群基础依赖，不支持卸载操作。
3. 确认卸载组件后操作流程将下发执行，卸载组件操作通常需要10-20分钟。

重启服务

功能介绍

组件配置项修改后，需要重启对应的服务使配置生效。为确保服务重启过程中，尽量减少或不影响业务运行，可通过滚动重启服务。对于有主备状态的实例，会先重启备实例，再重启主实例。滚动重启时间比普通重启时间久。

- 控制台持重启服务，默认勾选滚动重启方式。滚动重启关闭后所有节点同时重启可能导致服务不可用，请谨慎选择。
- 失败处理策略持两种方式：失败时阻塞等待处理和单节点失败继续处理。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 若您需要重启整个组件服务，可以在集群服务页，选择需要重启的组件卡片操作 > 重启服务，或者进组件详情页，在页面右上角选择更多操作 > 重启全部服务；若您需要重启某个角色实例，可以在集群服务页，选择需要重启的组件卡片操作 > 角色管理，勾选需要重启的服务角色并单击重启服务。
   在组件卡片或组件详情页更多操作中重启服务时，需要选择重启的服务角色、重启方式、是否滚动重启、失败处理策略等。当您选择服务角色为全部时，将重启整个组件。

   > 说明：
   >
   > TBDS 5315版以上支持按机架策略分批进行重启。

重启服务

1.重启后，该服务可能会不可用，请谨慎操作。
2.任务提交后需要一些时间执行（服务状态不会立即更新，等待时间与节点数量相关），期间不能进行其他操作，请等待一段时间后刷新页面。

服务名称 * HBASE

服务角色 * ALL

重启范围 所有符合条件的节点共（9）个

重启方式 默认重启模式

滚动重启

每次重启 1 台，间隔 5 秒 按机架策略分批

最大重启台数为 99999 台，最大间隔为 5 分钟。若启用机架策略，将优先按机架顺序重启。

失败处理策略 失败时阻塞等待处理

操作原因

请输入不超过200字

确认 取消

在角色管理页中重启服务时，只需要选择重启方式、失败处理策略等。

重启服务

1.重启后，该服务可能会不可用，请谨慎操作。
2.任务提交后需要一些时间执行（服务状态不会立即更新，等待时间与节点数量相关），期间不能进行其他操作，请等待一段时间后刷新页面。

服务名称 * HBASE

服务角色 * RegionServer

重启范围 所有符合条件的节点共（3）个

重启方式 安全重启模式

滚动重启 每次重启 1 台，间隔 5 秒 按机架策略分批

最大重启台数为 1 台，最大间隔为 5 分钟。若启用机架策略，将优先按机架顺序重启。

失败处理策略 失败时阻塞等待处理

操作原因

请输入不超过200字

确认 取消

服务组件支持重启方式：

组件	服务	重启模式	描述	备注
HDFS	NameNode	快速重启模式	通过 ebcha-daemon.sh stop | start namenode 进行重启	-
	NameNode	安全重启模式	在 HA 集群中，首先在 StandbyNameNode 上做 saveNamespace 操作，然后通过 ebcha-daemon.sh stop | start namenode 进行重启。非 HA 集群与快速重启模式一致	只支持滚动重启
	DataNode	默认重启模式	通过 ebcha-daemon.sh stop |start datanode 进行重启	
	JournalNode	默认重启模式	通过 ebcha-daemon.sh stop |start JournalNode 进行重启	
	zkfc	默认重启模式	通过 ebcha-daemon.sh stop |start zkfc 进行重启	
YARN	ResourceManager	默认重启模式	通过 sbin/YARN-daemon.sh stop | start ResourceManager 进行重启	只支持滚动重启
	NodeManager	默认重启模式	通过 sbin/YARN-daemon.sh stop | start NodeManager 进行重启	-
	JobHistoryServer	默认重启模式	通过 sbin/YARN-daemon.sh stop| start HistoryServer 进行重启	-
	TimelineServer	默认重启模式	通过 sbin/YARN-daemon.sh stop | start TimelineServer 进行重启	-
HBase	HBaseThrift	默认重启模式	通过 HBase-daemon.sh stop |start Thrift 进行重启	-
	HMaster	默认重启模式	通过 HBase-daemon.sh stop | start-master 进行重启	-
	HRegionServer	快速重启模式	通过 HBase-daemon.sh stop | start-regionserver 进行重启	-
	HRegionServer	安全重启模式	通过 graceful_stop.sh --restart --reload 进行重启	-

组件	服务	重启模式	描述	备注
Hive	HiveMetaStore	默认重启模式	通过 hcat_server.sh stop | start进行重启	-
	HiveServer2	默认重启模式	通过 hive-daemon.sh stop-h2 | start-h2 进行重启	-
	HiveWebHcat	默认重启模式	通过 hcat_server.sh stop |start 进行重启	-
Presto	PrestoCoordinator	默认重启模式	通过 bin/launcher stop | start 进行重启	只支持滚动重启
	PrestoWorker	默认重启模式	通过 bin/launcher stop | start 进行重启	-
ZooKeeper	QuorumPeerMain	默认重启模式	通过 bin/zkServer.sh stop | start进行重启	-
SPARK	SparkJobHistoryServer	默认重启模式	通过 sbin/stop-history-server.sh| sbin/start-history-server.sh 进行重启	-
Hue	Hue	默认重启模式	通过 build/env/bin/start.sh 和build/env/bin/stop.sh 进行重启	-
Oozie	Oozie	默认重启模式	通过 oozied.sh stop | start 进行重启	-
RANGER	Ranger	默认重启模式	通过 sbin/ranger-daemon.sh stop | start 进行重启	-
Elasticsearch	ESMaster/ESData	默认重启模式	bin/elasticsearch-daemon.sh start	
Kibana	Kibana	默认重启模式	bin/kibana-daemon.sh start	
StarRocks	FE	默认重启模式	docker stop sr-fe; docker rm sr-fe sh fe/bin/stop_fe_container.sh sh fe/bin/start_fe_container.sh	前端页面先停止fe 组件
	BE	默认重启模式	docker stop sr-be; docker rm sr-be sh be/bin/stop_be_container.sh sh be/bin/start_be_container.sh	前端页面先停止be 组件

组件	服务	重启模式	描述	备注
	Broker	默认重启模式	docker stop sr-broker; docker rm sr-broker sh apache_hdfs_broker/bin/stop_broker_container.sh sh apache_hdfs_broker/bin/start_broker_container.sh	前端页面先停止broker组件

启停服务

功能介绍

启停服务可以启停单节点上的所有服务。本节为您介绍通过 TBDS Manager 管理平台进[illegible]启停服务的相关操作。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进[illegible]集群详情页。
2. 在集群详情页中选择集群资源，单击资源管理进[illegible]资源管理页。选择需要启停的节点，单击更多 > 启停服务。
3. 在启停服务页，确认相关服务信息。

4. 确认信息无误后，单击启动全部服务或者停止全部服务即可启动或停[illegible]节点上的全部服务。

暂停服务

功能介绍

经典集群的组件服务暂停功能主要用于在运维操作期间安全地停止特定服务，避免数据损坏或服务中断，同时保持集群整体可用性。

操作步骤

在集群服务中，选择 对应组件卡操作 > 暂停服务进操作。

组件暂停支持列表：

组件	暂停方式	描述
HDFS	快速暂停	直接停止服务
YARN	快速暂停	直接停止服务
HBase	快速暂停	直接停止服务
Hive	快速暂停	直接停止服务
Kafka	快速暂停	直接停止服务
Spark	快速暂停	直接停止服务
Impala	快速暂停	直接停止服务
Kyuubi	快速暂停	直接停止服务
Hue	快速暂停	直接停止服务
ZooKeeper	快速暂停	直接停止服务
Trino	快速暂停	直接停止服务
Knox	快速暂停	直接停止服务

客户端管理

功能介绍

客户端管理提供了组件部署的客户端信息，用户可以在组件详情中的客户端管理页查看客户端的配置状态、配置组、节点类型、节点 IP 信息。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 在集群服务模块点击组件名称，进入组件详情页。
3. 单击客户端管理，即可进入客户端管理页。

静态服务池
静态服务资源

功能介绍

集群分配给各个服务的资源是静态服务资源，这些服务包括Trino、Impala、HBase、HDFS和Yarn。每个服务的计算资源总量固定，不与其他服务共享，是静态的。租户通过独占或共享一个服务来获取这个服务运行时需要的资源。
静态服务池用来指定服务资源的配置。在服务级别上，静态服务池对各服务可使用的资源进行统一管理：

- 限制服务使用的资源总量，支持配置Trino、Impala、HBase、HDFS和Yarn在部署节点可使用的CPU、I/O和内存总量。
- 实现服务级别的资源隔离，可将集群中的服务与其他服务隔离，使一个服务上的负载对其他服务产生的影响有限。

调度机制

静态服务资源支持基于时间的动态调度机制，可以在不同时间段为服务配置不同的资源量，优化客户业务运行环境，提高集群的效率。
在一个复杂的集群环境中，多种服务共享使用集群资源，但是各服务的资源使用周期可能会有比较大的区别。
例如以下业务场景，对于一个金融客户：

- 在白天HBase查询服务的业务多。
- 在晚上查询服务的业务少而Hive分析服务业务多。
  如果只给每个服务设置固定的资源可能会导致：
- 白天查询服务的资源不够用，分析服务的资源空闲。
- 晚上分析服务的资源不够用，查询服务的资源空闲。
  集群资源利用率不高，而且服务能力也打了折扣。因此：
- 白天多配置HBase服务资源。
- 晚上多配置Hive服务资源。
  TBDS Manager同时支持日、周、月三种周期化的调度方式，能够使集群在不同的时间段根据配置动态调整不同组件可以使用的资源。这种基于时间的动态调度机制可以更高效的利用资源，运行任务。

配置集群静态资源

操作场景

当需要控制集群服务可以使用节点资源的情况，或者控制集群服务在不同时间段节可用配额的CPU与I/O资源时，管理员可以在TBDS Manager调整资源基数，并自定义资源配置组。

> 注意：
>
> 配置静态服务池后，各服务及角色实例使用的最大资源将不能超过限制。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群服务 > 静态服务池打开配置列表。
3. 点击新增服务池配置，在弹窗中配置基础信息、权重配置及调度周期。当前仅支持对 Trino、Impala、HBase、HDFS 和Yarn进行 CPU/IO 的使用限制。

说明：

- "CPU Limit(%)"用于配置服务可使用的CPU核数与节点可分配的CPU核数占比,服务使用的资源配置总和可以大于100%。
- "CPU Share(%)"用于配置服务在与其他服务使用同一个CPU核的时间占比，即多个服务在使用同一个CPU核发生争抢时的时间占比。
- 所有服务使用的"I/O(%)"资源配置总和可以大于100%，设置为0时不作限制。

4. 对自定义的服务池可以进行编辑和删除，对默认的服务池只能进行编辑操作。

YARN组件服务
YARN资源调度
YARN 资源调度概述

功能介绍

YARN 资源调度提供了交互式的 YARN 资源队列调度管理能力，较文件式配置管理操作更便捷，目前持 FairScheduler （公平调度器）和 Capacity Scheduler （容量调度器）两种类型的调度配置。

注意事项

1. 资源调度器默认使用公平调度器，配置管理 YARN 组件 Fair-scheduler.xml 配置文件中的相关配置项参数保持与资源调度页一致。切换调度器为容量调度器，配置管理 YARN 组件 Capacity-scheduler.xml 配置件中的相关配置参数也保持与资源调度页致。
2. 资源调度页切换调度器类型后或设置策略后需单击部署生效才会进行对应策略的配置下发，该操作可能会重启 ResourceManager。
3. 如果用户已经在配置管理中开启了标签调度，当开启容量调度时，会进行同步操作。
4. 资源池持嵌套资源子池，子池受其父池设置规则限制。
5. 关闭资源调度器时，选择同步修改后配置，将覆盖其配置管理中对应调度器的配置文件及配置参数。

配置 Fair Scheduler

功能介绍

Fair Scheduler 是公平调度器，公平调度器将资源公平地分配给 YARN 上的各个作业，通过权重来调整资源的分配。

名称解释

配置集：定义在给定时间内处于活动状态的资源池之间的资源分配。
计划模式：定义配置集何时处于活动状态。
放置规则：通过放置规则将不同的用户提交的作业自动分配到指定的资源池中。
用户限制：定义用户可以同时提交的最多应用程序数量。

操作步骤

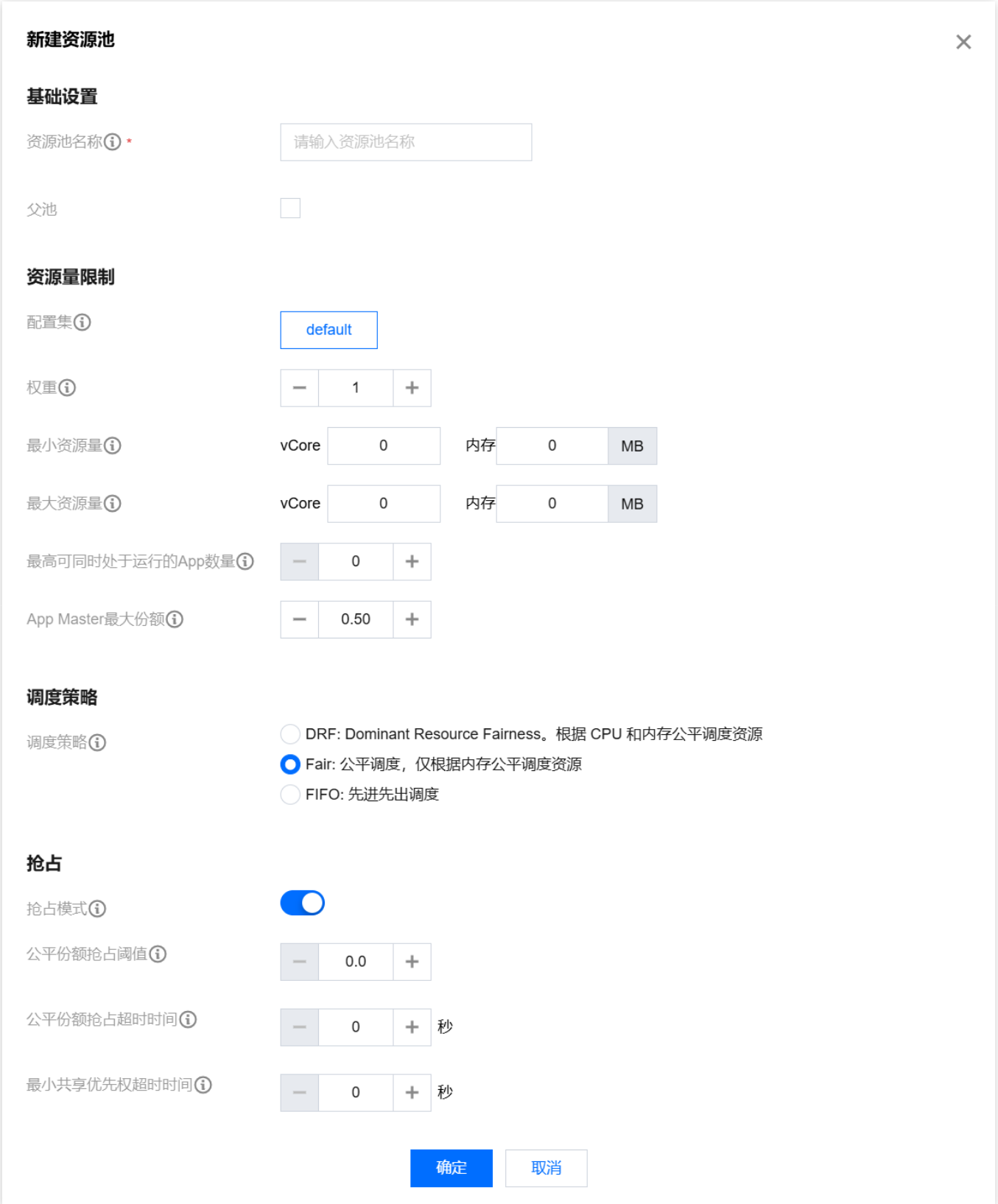
新建资源池

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群服务 > YARN 组件卡页右上角操作 > 资源调度进资源调度页面。

3. 单击资源调度器开关，打开开关后即可进行相关调度器配置。

4. 新建 Fair Scheduler 资源池

   调度器类型选择 Fair Scheduler 即可进 Fair Scheduler 策略设置页面，单击新建资源池即可新建资源池，可对已有资源池进行编辑、新建子池、克隆、删除等操作。配置字段详情解析请见字段与配置项对照表。



配置计划模式

1. 单击策略设置中的计划模式即可进计划模式页面，单击新建计划模式即可进行计划模式的新建。

配置集状态用于标记计划模式是否开启，默认为开启状态，若不需要使用自定义计划模式但仍想保留配置集，可将配置集状态设置为关闭。

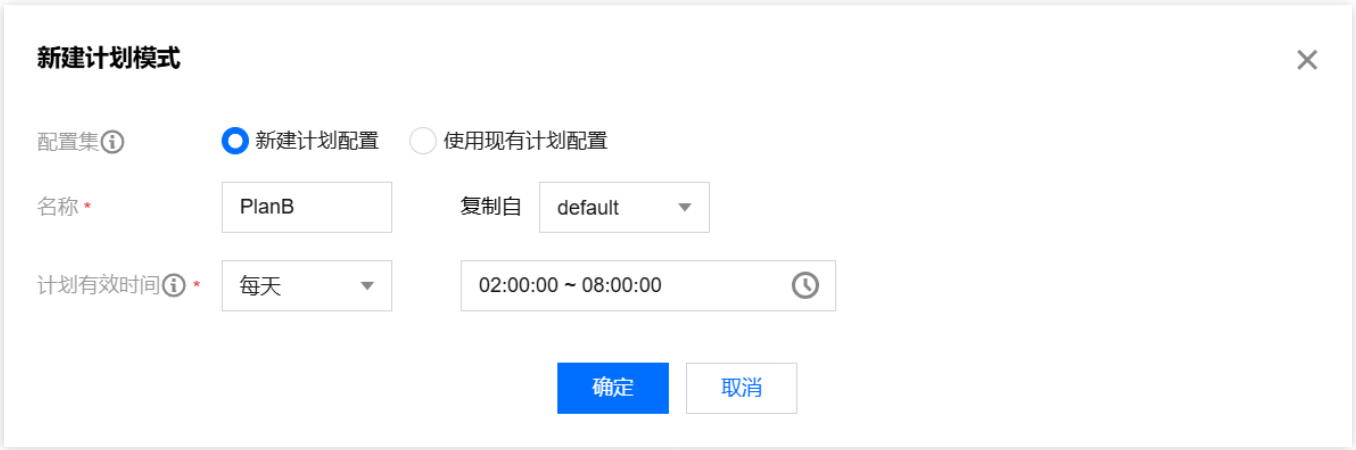
2. 在新建计划模式中选择/填写配置集、名称和计划有效时间。

示例配置集

资源池在不同配置集中的资源量限制相互独立，即可以按照业务分别进行配置，相互不影响。

1. 登录 TBDS Manager 管理平台，在集群列表中选择对应的 Hadoop 集群单击详情进集群详情页。
2. 在集群详情页中选择集群服务 > YARN 组件卡页右上角操作 > 资源调度进资源调度页面。
3. 单击资源调度器开关，调度器类型选择 Fair Scheduler 。
4. 单击新建资源池，根据实际需求进行配置。

5. 在资源调度页中选择计划模式>新建计划模式，根据业务需要调整计划有效时间。



6. 在资源调度页中选择资源池，在配置集下拉选项中，选择个配置集。

7. 在配置集下选择之前创建的资源池，按照业务进行资源量限制的编辑调整。
8. 资源池调整完后，单击部署生效，即可使设置生效。

配置放置规则

1. 单击策略设置中的放置规则即可进放置规则页面，单击新建放置规则即可进行放置规则的新建。

2. 填写放置类型和池名称。配置规则类型详情请见文末的 配置规则类型对照表。放置规则配置有两种类型：
   已在运行时指定：该放置规则主要使用在运行时指定的资源池。
   放置规则的判断方式，根据放置规则的顺序1、2、3...进行判断，判断到满足条件的放置规则后，后续的规则不再进行匹配。

配置用户限制

1. 单击策略设置中的用户限制即可进用户限制页面，单击新建用户限制即可进行户限制的新建。

2. 填写用户名称和同时运行应用程序上限：

配置权限模式

> 说明：
>
> 此功能于 TBDS 5315 版本开始支持

参考配置 Capacity Scheduler中对应章节。

字段与配置项对照表

字段名称	对应参数名称	参数含义
资源池名称	name	资源池的名称或队列的名称。资源池名称只能由含数字、字母，-或_组成，不能以-或_开头。
父池	type 的值为 parent	表示该资源池虽然底下没有子池，但它也不是叶子节点，在 hadoop 2.8 及以后父池不能有子池。
配置集	无	YARN 没有此参数，表示定时任务的集合。
权重	weight	在父池中的资源占比，权重越大分配的资源越多。
最小资源量	minResources	最少资源保证量，当个队列的最少资源保证量未满足时，它将优先于其他同级队列获得资源。
最大资源量	maxResources	最多可以使用的资源量，每个队列可使用的资源量不会超过该值。
最可同时处于运的 App	maxRunningApps	最多同时运行的应用程序数目，该限制可以防止超量 MapTask 同时运行时产生的中间输出结果撑爆磁盘。

字段名称	对应参数名称	参数含义
数量		
AppMaster 最份额	maxAMShare	限制队列用于运行 Application Master 的资源比例。这个属性只能于叶子队列。
调度策略	schedulingPolicy	任队列都可以设置调度策略，取值为 FIFO、Fair、DRF，其中 FIFO、Fair 在资源分配时只考虑内存，DRF 考虑内存和核数。
抢占模式	allowPreemptionFrom	在 hadoop 3.x 之后才生效，2.x只能由全局配置 yarn.scheduler.fair.preemption 控制。
公平份额抢占阈值	fairSharePreemptionThreshold	队列的公平共享抢占阈值。如果队列等待 fairSharePreemptionTimeout 之后没有接收到 fairSharePreemptionThreshold*fairShare 的资源，它被允许从其他队列抢占资源。如果不设置，队列将会从其父队列继承这个值。
公平份额抢占超时时间	fairSharePreemptionTimeout	队列处在最公平共享阈值之下，在尝试抢占其他队列的资源之前的秒数。如果不设置，队列将会从其父队列继承这个值。
最小共享优先权超时时间	minSharePreemptionTimeout	队列处在最小共享之下，在尝试抢占其他队列的资源之前的秒数。如果不设置，队列将会从其父队列继承这个值。
提交访问控制	aclSubmitApps	可以提交 apps 到队列的用户的列表。
管理访问控制	aclAdministerApps	可以管理队列的用户的列表。

配置规则类型说明

配置规则类型	规则类型含义
root.[pool name]	该规则始终满足，在其它规则不匹配的情况下使用，因此该规则默认要放置在所有匹配规则之后。
root.[pool name].[username]	该放置规则会判断资源池中是否存在相应的 pool name，存在则在该资源池下创建与用户名相同的资源池（勾选池不存在时创建池的情况下）。
root.[primarygroup]	该规则使用与该用户主要组匹配的资源池。Linux 中用户默认的主要组与用户名致，匹配时会通过用户的主要组与资源池名称比对。
root.[primarygroup].[username]	该放置规则会优先使用用户的主要组匹配的资源池，然后使用与该用户名匹配的子池，如果勾选池不存在时创建池则会在该池下创建个与用户名一致的子池。
root.[secondarygroup]	该放置规则用于匹配用户的次要组，使用与次要组之一匹配的资源池。
root.[secondarygroup].[username]	该放置规则首先匹配用户的次要组，然后使用与该用户名匹配的资源池。

配置规则类型	规则类型含义
root.[username]	该放置规则用于匹配与用户名一致的资源池。（不推荐使用）

配置 Capacity Scheduler

Capacity Scheduler 是容量调度器，容量调度器以分层的方式组织资源，可通过多层级的资源限制条件让多户共享集群资源。

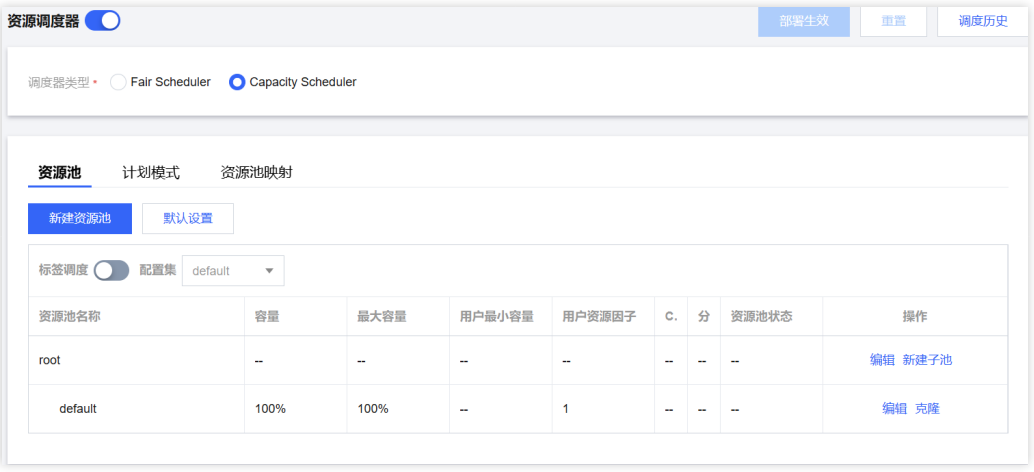
新建资源池

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。

2. 在集群详情页中选择集群服务 > YARN 组件卡页右上角操作 > 资源调度进资源调度页面。

3. 单击资源调度器开关，打开开关后即可进行相关调度器配置。

4. 新建 Capacity Scheduler
   调度策略类型选择 Capacity Scheduler 即可进 Capacity Scheduler 的配置页面，单击新增资源池即可新建资源池。可对已有资源池进行编辑、新建子池、克隆等操作；也可单击默认设置进设置容量调度的延迟调度次数。



配置资源池映射

1. 单击策略设置中的资源池映射即可进入资源池映射页面，单击新建资源池映射即可进行新建资源池映射。
2. 是否覆盖用户指定队列选项默认关闭，假如用户在资源池映射中定义了映射的队列，且用户在提交任务时指定了队列，但是该队列与映射队列不同时：当用户指定的队列为 default 或者开启了覆盖，则会使映射队列，否则使用户指定的队列。

标签调度

功能介绍

标签管理提供新建、编辑、删除标签，以及绑定节点等功能。给集群各节点打上不同的标签，有助于在 Capacity Scheduler 之上进行更细粒度的资源划分，并支持应用程序指定运行的位置。

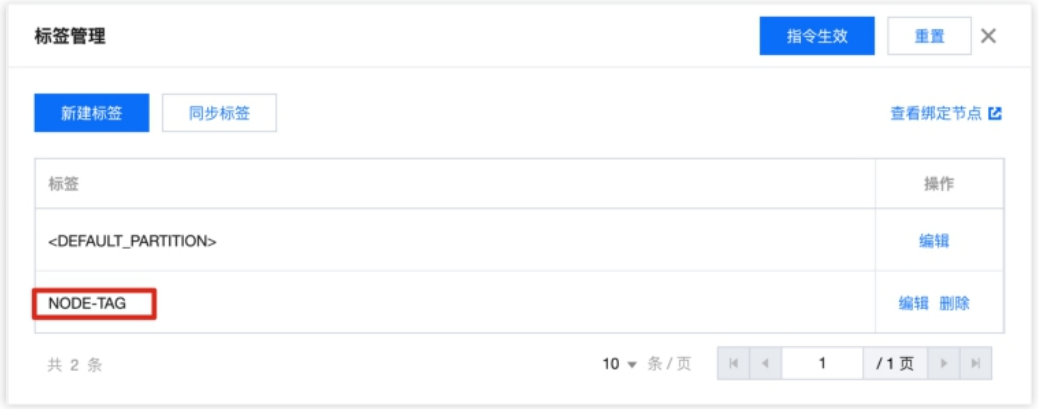
前提条件

开启标签调度。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中选择对应的 Hadoop 集群单击详情进集群详情页。
2. 在集群详情页中选择集群服务 > YARN 组件卡页右上角操作 > 资源调度进资源调度页面。
3. 单击资源调度器开关，调度器类型选择 Capacity Scheduler。
4. 单击标签调度开关，打开标签调度后单击标签管理，进标签管理页。

5. 单击新建标签，填写标签名称，并根据需要设置标签类型和该标签绑定的节点。



6. 标签设置完成后，单击指令生效，即可在资源池中编辑查看该标签的资源队列。

7. 在资源调度页中单击新建资源池，在**资源量限制>标签设置>添加，**配置资源池使用特定标签节点资源的配额。资源池在不同标签中的容量、最大容量相互独立，即可以按照业务分别进行配置，相互不影响。

8. 任务提交到队列（资源池）后，与标签的资源配额的交互逻辑如下：

- 如果任务提交时没有指定NodeLabel并且队列没有配置默认标签表达式，就调度到DEFAULT_PARTITION。
- 如果任务提交时没有指定NodeLabel，但队列配置了默认标签表达式，就调度到默认标签表达式设置的NodeLabel节点上。
- 如果任务提交时指定了NodeLabel，但队列没有配置对应NodeLabel的资源，则除AM外的Container会处于Pending状态。
- 如果任务提交时指定了NodeLabel，且队列也配置了对应NodeLabel的资源，则会调度到NodeLabel对应的节点上。

计划模式

> 说明：
>
> 此功能于 TBDS 5313 版本开始支持

计划模式是 TBDS 相较于社区提供的企业特性，其能力为：按照计划（每天/每周/每月），使用预配置的队列资源配额而非使用固定的资源配额。

典型使用场景

1. 潮汐调度，多数业务白天夜晚的负载模式不一样，可以通过计划模式，更好分配白天/晚上各队列的资源配额。
2. 月批场景，一些行业有月级的高优先级任务（如银行监管报送）。可以对月批任务所处队列，在月初配置较大的资源池，而在预计执行完后的时间点自动释

放队列资源以供其他队列使用，以优化资源使用。

使用示例

假定场景：集群配置了两个队列 queue2、queue1。其中 queue2 晚上负载较重，但白天基本没有负载，queue1 负载主要集中在白天。我们想要白天 queue2 可以降低资源配额，将资源让给 queue1。

下面我们演示如何使用 TBDS 计划模式实现这一目标。

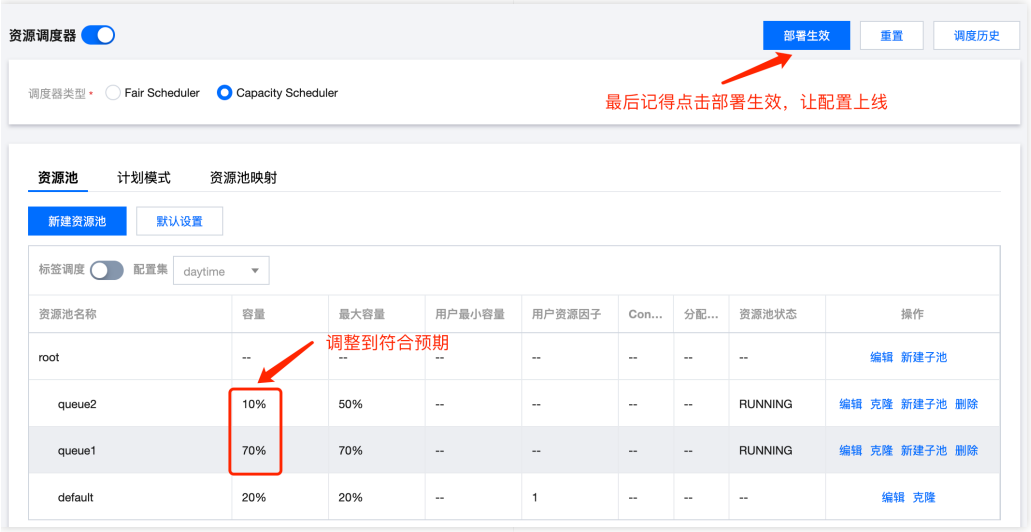
单击资源调度>Capacity Scheduler>计划模式>新建计划模式以新增计划模式。

填写相关字段配置，这里我们配置每天 10点 ~ 18点，使用另一套容量调度配置（配置集），并将这套配置集命名为 daytime，单击确认提交。

提交完成后，可以看到计划模式新增了一行，如下。

下一步，我们修改 daytime 配置集对应的资源配额。
单击 资源池 Tab 页，配置集下拉选项，选择我们刚创建的 daytime 配置集。
选择配置集完成后，对需要进行配额调整的队列进行调整。

这里进行如下调整： 将queue2白天的配额由50%调整为10%，将queue1白天配额由30%调整为70%。
调整后如下：



最后确认无误后，单击 部署生效 让新增的配置集上线。

> 说明：
>
> 需要到配置集预定的调度时间，对应的配置集才会生效，可以在 Yarn UI 进行验证。

使用限制

请勿对于相同调度周期，重复添加多种不同的配置集，只有首次添加的配置集会生效。

绝对值配额

> 说明：
>
> 此功能于 TBDS 5313 版本开始支持

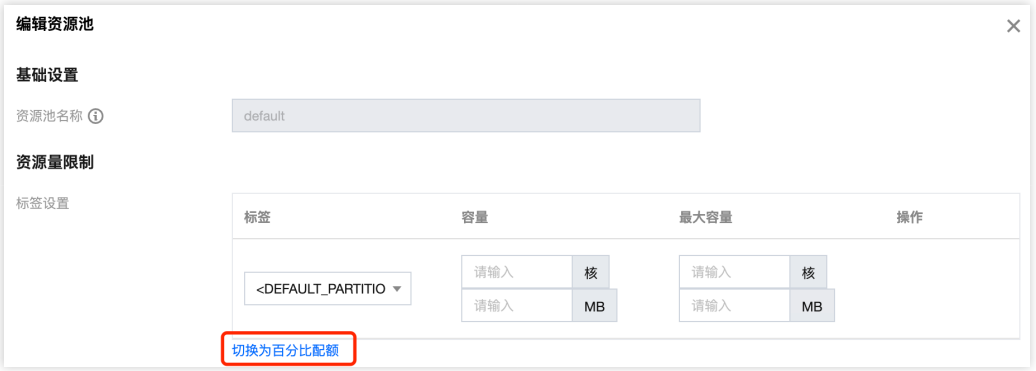
绝对值配额指相比于百分比的资源配额，配置绝对的资源配额（例如 32 core，128 GB memory）。

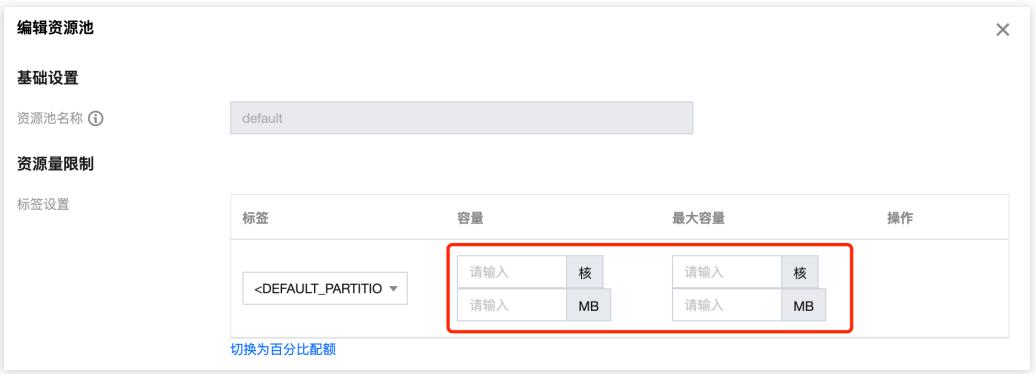
典型应用场景

1. 期望分配给业务的资源是固定的，而非随着Yarn节点扩容随百分比自动增长，考虑到扩容出来的资源是要分配给新业务场景。
2. 期望保障重点业务的资源需求，使其不受集群节点缩容影响。

使用示例

1. 编辑资源池页面，单击“切换为绝对值配额”
2. 容量、最大容量的配置便会由百分比切换为绝对值配额。





使用限制

1. 绝对值的配额和百分比配额可以混用，但是仅限于一级队列，二级及以下队列需跟随一级队列的配置方式。例如：sub1、sub2 都是一级队列，sub1 队列是绝对值配额，sub2 是百分比配额。那么 sub1 的所有子队列都需要是绝对值配额，sub2 的所有子队列都需要是百分比配额。
2. 绝对值配额和百分比配额混用时，绝对值配额并不能使用集群所有的资源，扣除所有百分比配额后的集群资源可以分配给绝对值配额。过大的绝对值配额会导致生效配置时报错配置资源过大。

用户权重

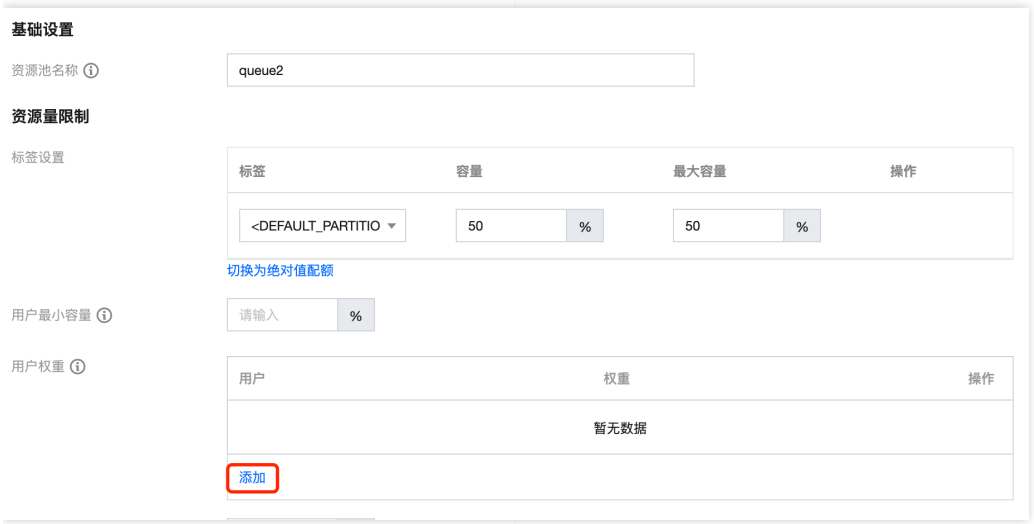
> 说明：
>
> 此功能于 TBDS 5313 版本开始支持

典型应用场景

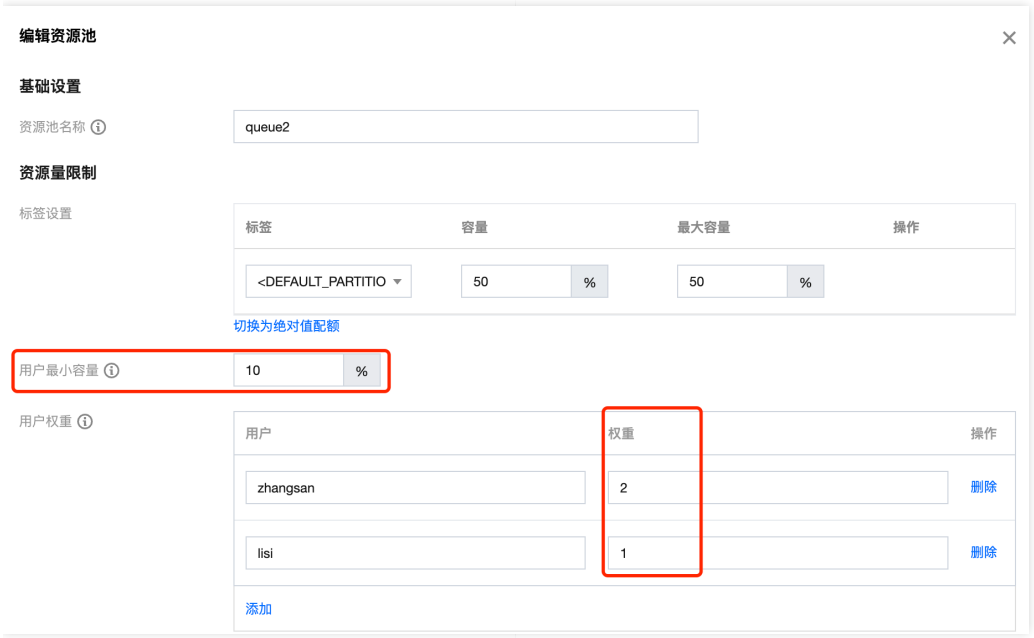
一个队列可能被多个用户共享使用，如果希望一些用户可以获得更多资源，同时又不想划分队列非常细难以管理，那么可以使用用户权重功能。

使用示例

编辑资源池页面，用户权重 字段对应位置，单击添加，以新增用户权重的配置。



配置思路为：如果希望某个人需要更多资源，可以增大权重，默认权重为 1，如果希望某个人资源是其他人 2 倍，就设置为 2。



以上只是权重配置方式的大体思路，但是实际上权重计算逻辑较为复杂，实际获得的资源差别可能不会恰好是权重的倍数。尤其是用户最小容量规定了用户可以获得的最小资源，当用户非常多的时候，会优先保障用户拥有最小容量，再保障权重高的用户获取符合权重的资源，实际获得的资源就不会严格的权重倍数关系。

权限模式

> 说明：
>
> 此功能于 TBDS 5315 版本开始支持

提供两种模式选择（默认/推荐统一模式），这里对差异比对说明如下：

维度	统一模式（Ranger）	混合模式（YARN + Ranger）
权限决策主体	完全通过 Ranger ACL进行细粒度访问控制	Ranger ACL与YARN ACL合并生效，且Ranger ACL策略优先
规则存储位置	Ranger Admin统一存储策略	队列基础权限存于YARN配置，细粒度策略存Ranger
生效层级	全栈权限（队列/作业/操作）通过Ranger策略覆盖	YARN控制队列访问，Ranger控制作业级权限

典型应用场景

一般YARN ACL用于快速紧急权限控制，Ranger ACL 用于长期策略管理。

场景	推荐模式	原因
全栈Ranger化环境	统一模式	统一管控HDFS/Hive/YARN等组件权限，避免多系统维护
历史YARN集群升级迁移	混合模式	保留原有YARN ACL配置，逐步迁移细粒度权限到Ranger
金融/政府强合规要求	统一模式	Ranger审批流和完整审计链满足合规需求
跨云混合部署环境	混合模式	降低Ranger中心化依赖，避免网络延迟影响调度性能

使用示例

混合模式配置示例参考（实际配置需要在TBDS运维工程师指导下进行）

> 注意：
>
> 混合模式下需通过capacity-scheduler.xml显式关闭与Ranger重叠的ACL项（如yarn.scheduler.capacity.root.dev.acl_administer_queue），避免权限越界。

1. YARN 队列 ACL 配置 (capacity-scheduler.xml)

名称 (name)	值 (value)	说明
yarn.acl.enable	true	启用YARN原生ACL权限检查
yarn.scheduler.capacity.root.queues	dev,prod	定义根队列下的子队列名称
yarn.scheduler.capacity.root.dev.acl_submit_applications	dev_group	dev队列提交权限：仅允许dev_group成员提交作业
yarn.scheduler.capacity.root.dev.acl_administer_queue	dev_admin	dev队列管理权限：dev_admin用户可修改队列配置
yarn.scheduler.capacity.root.prod.acl_submit_applications	prod_approvers	prod队列提交权限：仅prod_approvers组可提交（生产环境严格管控）

2. YARN 服务配置 (yarn-site.xml)

名称 (name)	值 (value)	说明
yarn.authorization-provider	org.apache.ranger.authorization.yarn.authorizer.RangerYarnAuthorizer	权限控制器：接入Ranger插件实现细粒度授权
ranger.plugin.yarn.policy.cache.ttl	30000	策略缓存时间：30秒刷新Ranger策略（平衡实时性与性能）
ranger.plugin.yarn.policy.evaluateDelegateAdmin	false	权限越界防护：禁止Ranger策略覆盖

名称 (name)	值 (value)	说明
		YARN ACL管理员权限
yarn.resourcemanager.primary.enabled	true	主RM高可用：启用主ResourceManager（混合模式需保障HA）

3. Ranger 策略配置

名称 (name)	值 (value)	说明
策略基础属性		
policyName	dev-queue-resource-limit	策略名称（需唯一标识）
resources	{"queue": {"values": ["root.dev"]}}	管控资源：针对root.dev队列生效
权限条目		
accesses.type	submit-job	操作类型：作业提交权限
accesses.isAllowed	true	允许提交作业
users	["dev_group"]	授权对象：应用dev_group成员

配置依赖说明：

配置层级	依赖项	混合模式作用
YARN ACL	Ranger策略未覆盖的基础权限	防止未授权用户访问队列（安全兜底）
Ranger策略	YARN队列名称一致性	确保root.dev等资源标识在ACL和Ranger中一致
服务配置	所有权限模块的开关	yarn.authorization-provider是混合模式的总入口

附录：容量调度字段与配置项对照表

字段名称	对应参数名称	参数含义
资源池名称	yarn.scheduler.capacity..queues	资源池的名称或队列的名称。
标签设置	无	设置队列可以访问的特定标签。
容量	yarn.scheduler.capacity..capacity	可以使的资源，同资源池的池容量总和为100，能使的资源=资源池*容量%。如果该队列需要这个例更的资源，其他队列有空闲资源的话，可以占这个例更的资源。
最容量	yarn.scheduler.capacity..maximum-capacity	队列的资源使上限（百分）。由于存在资源共享，因此个队列使的资源量可能超过其容量，最多使资源量可通过该参数限制。
默认标签表达式	yarn.scheduler.capacity..default-nodelabel-expression	当资源请求未指定节点标签时，应将被提交到该值对应的分区。默认情况下，该值为空，即应程序将被分配没有标签的节点上的容器。
户最容量	yarn.scheduler.capacity..minimum-user-limit-percent	每个户最低资源保障（百分）。任何时刻，个队列中每个户可使的资源量均有定的限制。当个队列中同时运多个户的应程序时，每个户的使资源量在个最值和最值之间浮动，其中，最值取决于正在运的应程序数，最值则由 minimum-user-limit-percent 决定。

字段名称	对应参数名称	参数含义
户资源因	yarn.scheduler.capacity..user-limit-factor	每个户最多可使的资源量（百分）。例如，假设该值为30，则任何时刻，每个户使的资源量不能超过该队列容量的30%。
分配Container 最内存数量	yarn.scheduler.capacity..maximum-allocation-mb	每个 container 的最内存值，这个配置会覆盖 YARN.scheduler.maximum-allocation-mb 值，但是该值必须于等于系统的 YARN.scheduler.maximum-allocation-mb 的值。
Container 最 vCore 数量	yarn.scheduler.capacity..maximum-allocation-vcores	每个 container 的最核数，这个配置会覆盖YARN.scheduler.maximum-allocation-vcores 值，但是该值必须于等于系统的 YARN.scheduler.maximum-allocation-vcores 的值。
资源池状态	yarn.scheduler.capacity..state	队列的状态。可以是正在运或已停。如果队列处于停状态，则法向其或其任何队列提交新的应程序。
最应数 Max-Applications	yarn.scheduler.capacity..maximum-applications	最 AM 例 yarn.scheduler.capacity..maximum-am-resource-percent群集中可于运应程序主机的最资
资源池优先级	yarn.scheduler.capacity.root..default-application-priority	配置资源队列的优先级，默认为0，设置值越，优先级越。
提交访问控制	yarn.scheduler.capacity.root..acl_submit_applications	可以提交 apps 到队列的户的列表。
管理访问控制	YARN.scheduler.capacity.root..acl_administer_queue	可以管理队列的户的列表。
延迟调度	yarn.scheduler.capacity.node-locality-delay	保证任务本地化执行，可以延迟调度的次数。如果值为 -1，将禁延迟调度。

查看调度历史

功能介绍

调度历史可查看资源队列配置的操作记录、任务状态等信息。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群服务 > YARN 组件卡页右上角操作 > 资源调度进资源调度页面。
3. 在资源调度页中单击调度历史，可查看任务开始时间、结束时间、状态以及操作信息等。

4. 可按执时间段筛选调度记录，在操作类型下单击详情可查看详细信息。

使用示例

YARN 提供的 Capacity Scheduler 和 Fair Scheduler 调度策略都是为了在多租户环境中公平、有效地分配资源，但它们的工作方式和适用场景有所不同：

- Capacity Scheduler：Capacity Scheduler 的设计目标是将大型、多租户的集群划分为一组独立的队列（queues），每个队列都有一定的容量。这种调度策略适用于那些需要严格的资源隔离和优先级保证的场景。适用于集群需要支持多租户需要严格隔离资源，工作负载主要是长期运行和更重视集群的利用率和效率的场景。
- Fair Scheduler：Fair Scheduler 的设计目标是在所有运行的应用程序之间公平地分配资源。与 Capacity Scheduler 不同的是，Fair Scheduler 的队列并不是静态的。也就是说，如果一个队列当前没有运行的应用程序，那么它的资源可以被其他队列使用。适用于企业更重视公平性、需要动态调整资源分配的场景。

Fair调度示例

场景说明

某企业 TBDS 大数据平台管理员正在按照业务需求划分资源，该企业后 4 个业务部门，不同部门的资源诉求如下：

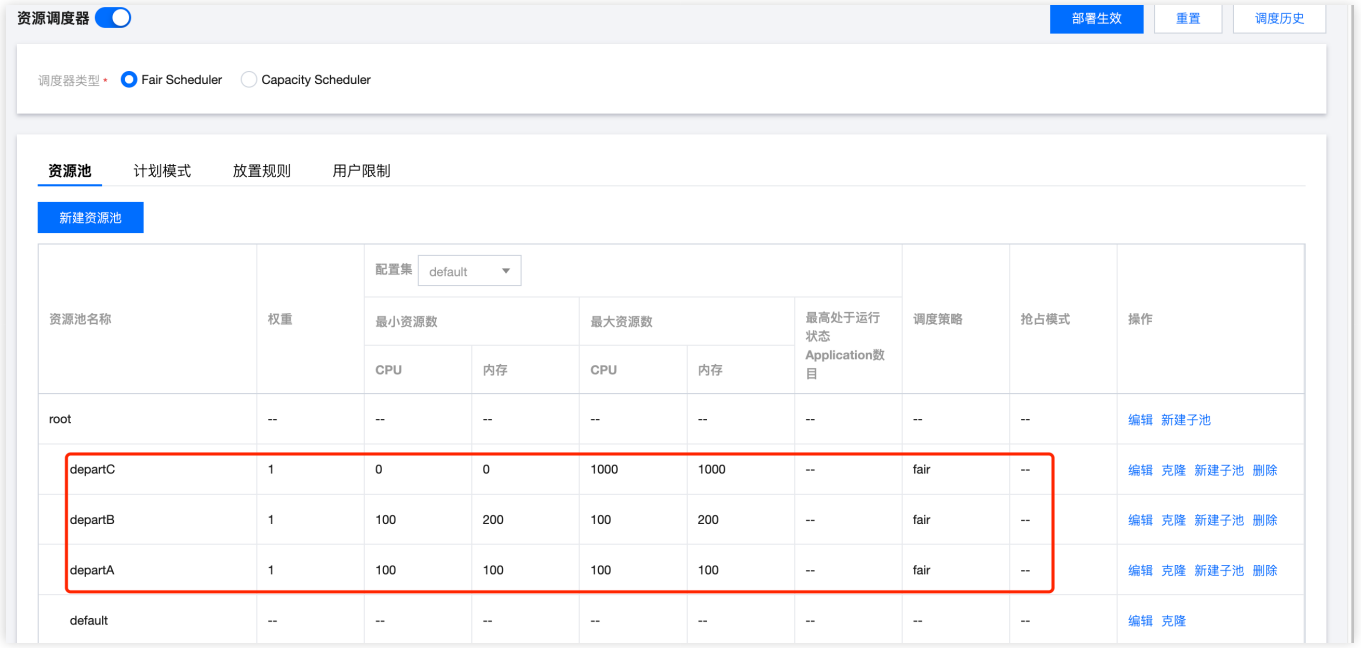
- 资源总量：1000 核，4000GB。
- 部门诉求：
  - A 业务部门：期望占用固定资源：100 核，100GB。
  - B 业务部门：期望最小占用资源为100 核，100GB，最大可弹性获取 200 核，200GB。
  - C 业务部门：仅使用平台公共资源，没有最大资源限制。

前提条件

- 完成经典集群安装，集群包含 YARN 组件。
- 集群运行正常。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群服务 > YARN 组件卡页右上角操作 > 资源调度进资源调度页面。
3. 单击资源调度器开关，打开开关后将调度类型设置为 Fair Scheduler。
4. 为三个部门配置不同的资源池，配置项为最小资源数和最大资源数最终配置如下：



5. 完成配置后点击“部署生效”按钮进行部署，操作后即可生效。

Capacity 调度示例

场景说明

某企业 TBDS 大数据平台管理员正在按照业务需求划分资源，该企业后 4 个业务部门，不同部门的资源诉求如下：

- 资源总量：1000 核，4000GB。
- 部门诉求：
  - A 业务部门：期望占用平台资源的60%
  - B 业务部门：期望占用平台资源的30%，限制单用户可以最多使用部门总资源的10%。
  - C 业务部门：期望占用平台资源的10%，最多可占用 20%

前提条件

- 完成经典集群安装，集群包含 YARN 组件。
- 集群运行正常。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。

2. 在集群详情页中选择集群服务 > YARN 组件卡页右上角操作 > 资源调度进入资源调度页面。
3. 单击资源调度器开关，打开开关后将调度类型设置为 Fair Scheduler。
4. 为三个部门配置不同的资源池，配置项为容量、最大容量和用户资源因子。最终配置如下：

资源池名称	容量	最大容量	用户最小容量	用户资源因子	Container最大vCore数量	分配Container最大内存数量	资源池状态	操作
root	--	--	--	--	--	--	--	编辑 新建子池
departC	10%	20%	--	--	--	--	RUNNING	编辑 克隆 新建子池 删除
departB	30%	30%	--	0.1	--	--	RUNNING	编辑 克隆 新建子池 删除
departA	60%	60%	--	--	--	--	RUNNING	编辑 克隆 新建子池 删除

5. 完成配置后点击“部署生效”按钮进行部署，操作后即可生效。

YARN 作业查询

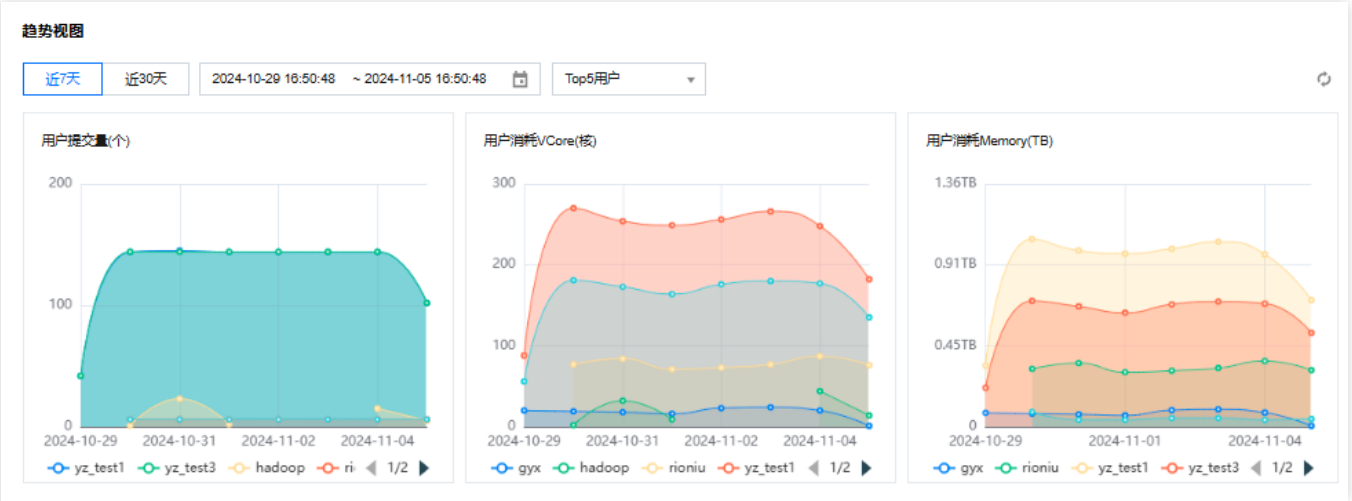
功能介绍

提供查看用户粒度提交量、Memory 量和 Vcore 消耗量等信息，快速查看 YARN 作业的提交队列、状态、持续时间等多项明细指标，并持作业级历史任务对比、作业洞察、任务执信息等。

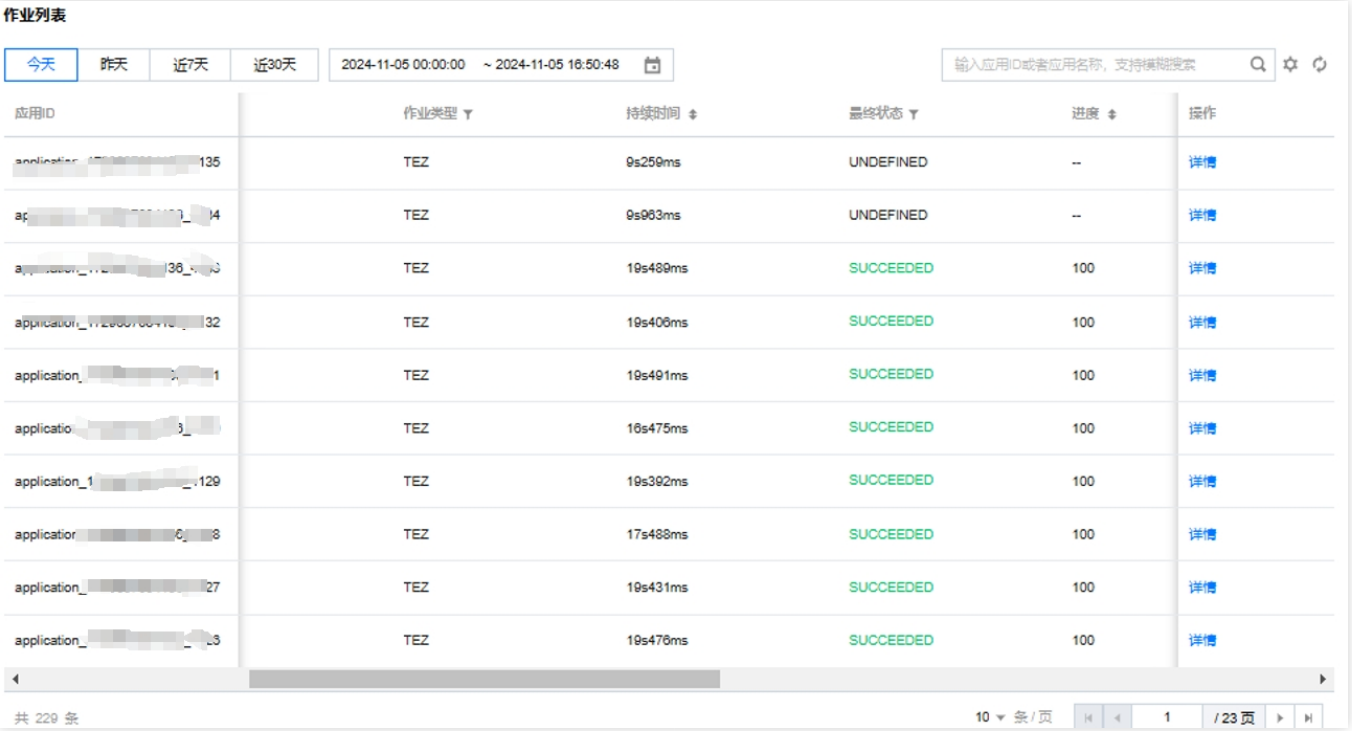
操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中单击集群服务，然后选择 YARN 组件右上角操作 > 作业查询，即可查看作业统计视图、资源消耗趋势，查询相关作业信息、任务信息查看、应用执行结果洞察及应用监控对比等。

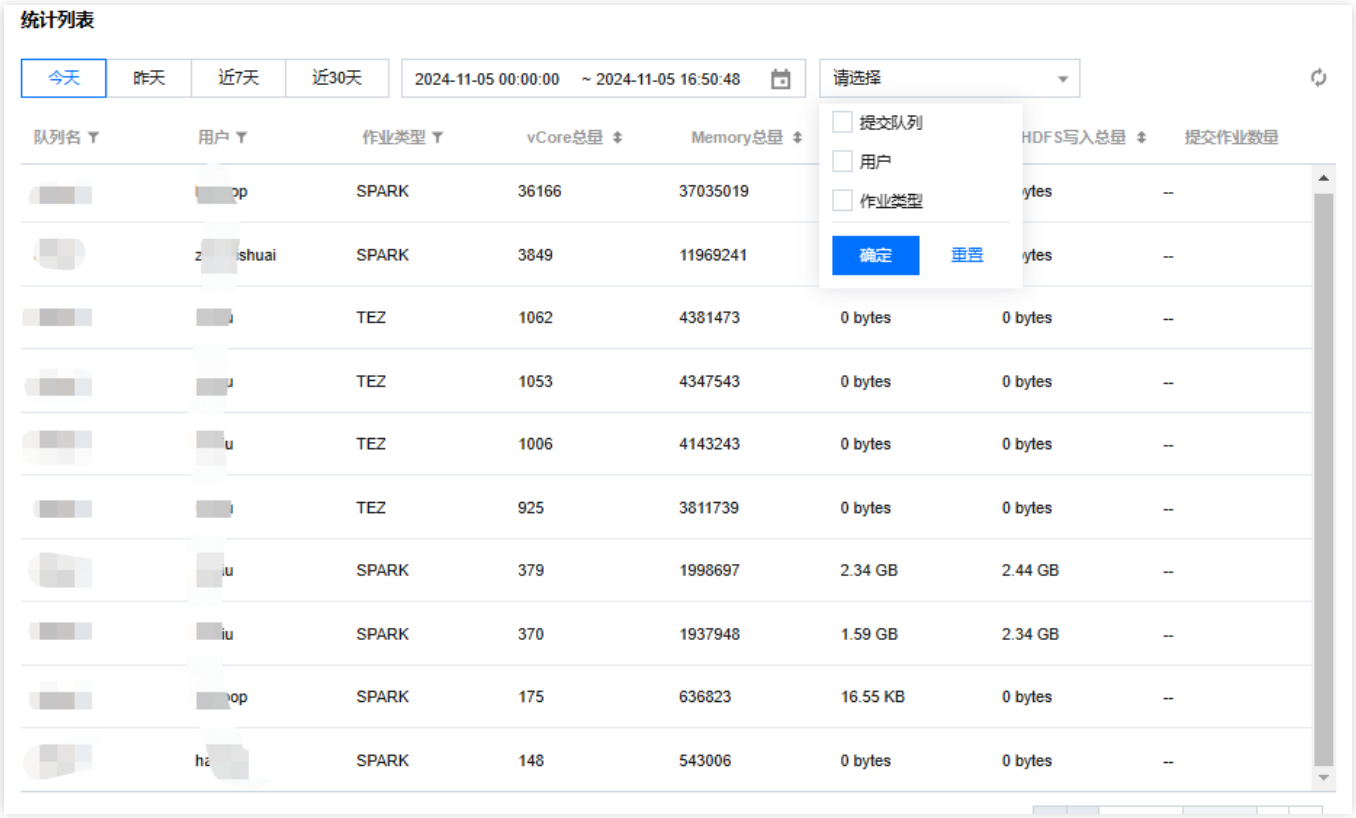
- 用户粒度的提交量、Memory、VCore 的消耗量视图及分布，持近期内的相关趋势查看。



- 作业级别提供用户、应用名、队列名、作业类型、持续时间及相关吞吐资源等多维信息筛查，并支持强制终止特定作业。



- 统计列表可按照指定的用户、队列等信息统计其资源消耗量，帮助统计资源开销情况辅助成本核查（接口同步持）。



- 作业查询将每 30s 采集[1]次 ResourceManager 数据，采集操作对集群业务影响微小可忽略。

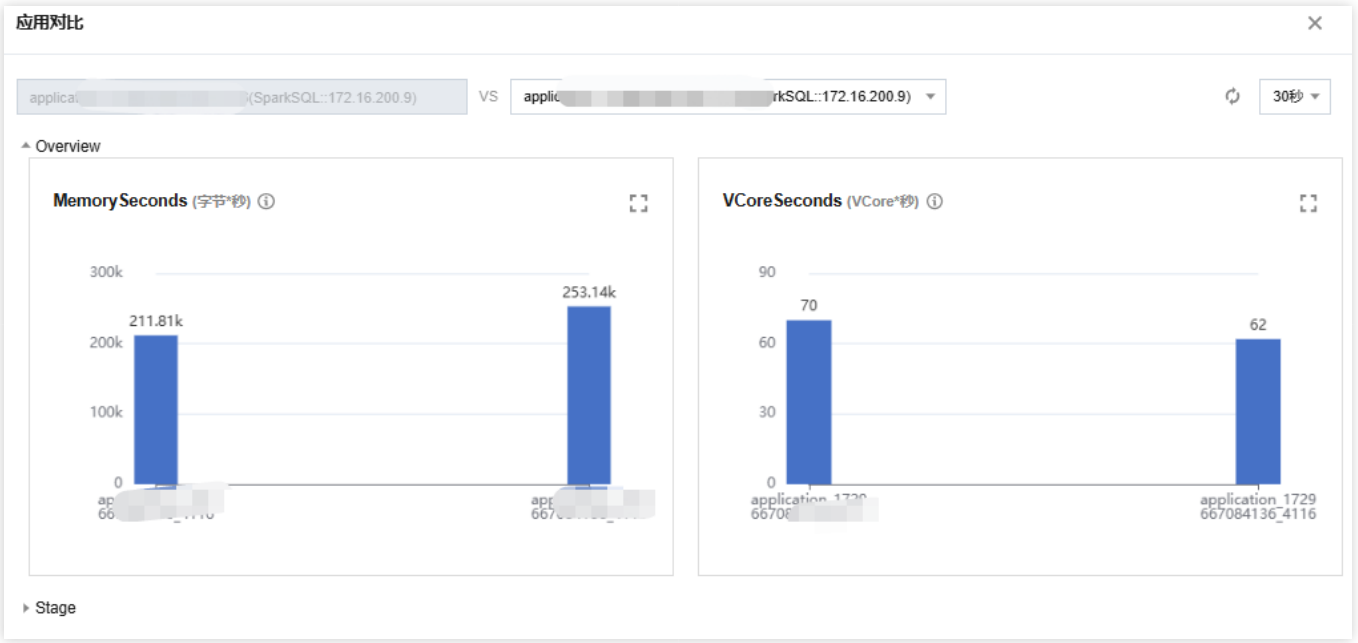
3. 在作业列表中单击更多 > 应用洞察，查看应[2]的详细洞察项及相关的洞察规则、结果、建议。

保障集群稳定运行，洞察功能采集策略满足以下任一规则将被降级忽略采集：

- 默认运行时长小于 10min 的 App 将被降级忽略。
- 默认采集时子任务大于 3W 的 App 将被降级忽略。

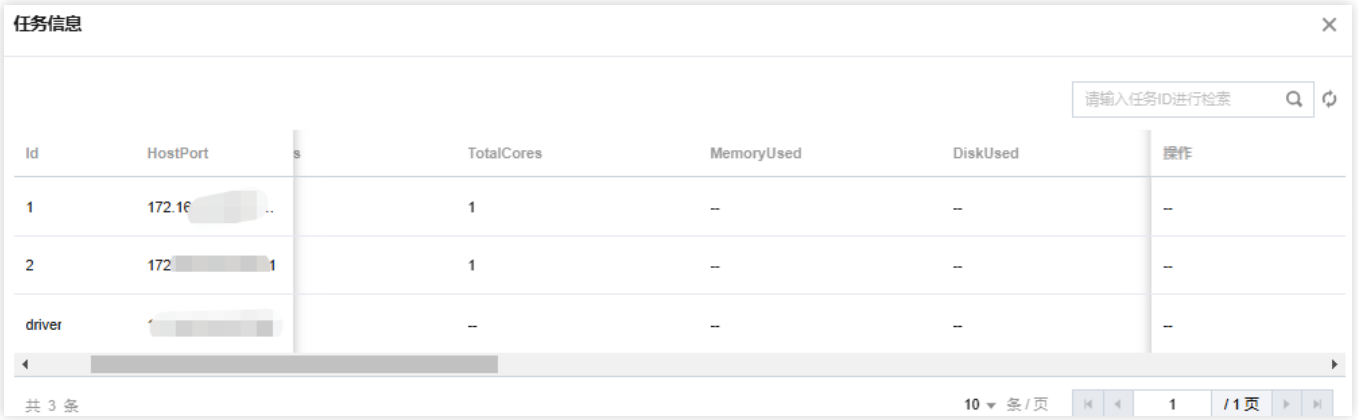
- 默认延迟采集时间大于 24h 的 App 将被降级忽略。
  YARN 应用洞察会分别采集 Spark History、Job History、Timeline Server 相关应用数据进行分析，如若发现上述服务请求量持续突破负载瓶颈可联系TBDS 产品专家关闭该功能。

4. 在作业列表中单击更多 > 应用对比，可以选择当前应用与同类型应用的业务指标对比信息。



- 仅 MR、Spark、Tez 类型且最终状态为 SUCCEEDED 的应用持应对比。
- 默认页面按照同类型相同应用名已做过滤，应对比的选择筛选范围仅限于同类型应，筛选持实时查询后台。

5. 在作业列表中单击更多 > 任务信息，查看作业的任务列表、Hosts 对比及任务的运行日志。



支持清单

功能的覆盖范围如下：

作业类型	任务信息	任务对比	任务日志

作业类型	任务信息	任务对比	任务日志
MR	支持	支持	支持
Spark	支持	不支持	支持
Tez	支持	支持	不支持
其他	不支持	不支持	不支持

刷新队列

功能介绍

当 capacity-scheduler.xml、fair-scheduler.xml 配置文件新增或更新内容时，YARN刷新队列可以使这些内容在 ResourceManager 中生效。

前提条件

YARN 刷新队列时，不要删除调度器的配置文件中已生效的队列。

操作步骤

1. 在集群服务中，选择 YARN组件卡片操作 > 刷新队列进行操作。
2. 在弹窗中选择操作范围，可选择默认节点，也可手动指定节点。

> 注意：
>
> 不要去删除 capacityscheduler.xml、fair-scheduler.xml 中定义的已生效的队列。

刷新队列
不要删除调度器的配置文件中已生效的队列
服务名称
YARN
服务角色
ResourceManager
操作范围
默认节点
指定节点
操作原因 *
请输入操作原因
请输入不超过200字
确定
取消

Impala组件服务
Impala资源调度

功能介绍

Impala资源调度功能支持用户对Impala集群进行资源池划分，为Impala添加多个独立的计算资源池，实现资源隔离。TBDS支持通过管控平台进行Impala资源池配置操作。

操作步骤

新建资源池

1. 登录 TBDS Manager管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。

2. 在集群详情页中选择集群服务 > Impala 组件进入资源调度管理页面。

3. 点击开启资源调度启动Impala资源池管理能力，Impala服务重启后资源调度将正式启用。

4. 点击新建资源池，配置资源池资源后点击确定。

配置项	说明
资源池名称	自定义资源池名称，只允许包含字母、数字、-、_, 长度255字符以内
最大内存限制	资源池查询最大内存限制，建议配合单查询内存限制使用
单查询内存限制	不允许超出最大内存限制
查询选项	默认关闭，若启用MEM_LIMIT查询选项，将允许用户通过该选项自定义单查询内存限制
最大并发查询数	默认值为无限制
最大排队查询数	默认值见default_pool_max_queued配置
排队超时时间	默认值见queue_wait_timeout_ms配置，单位ms
提交访问控制	访问控制列表可以控制向本资源池提交查询的用户

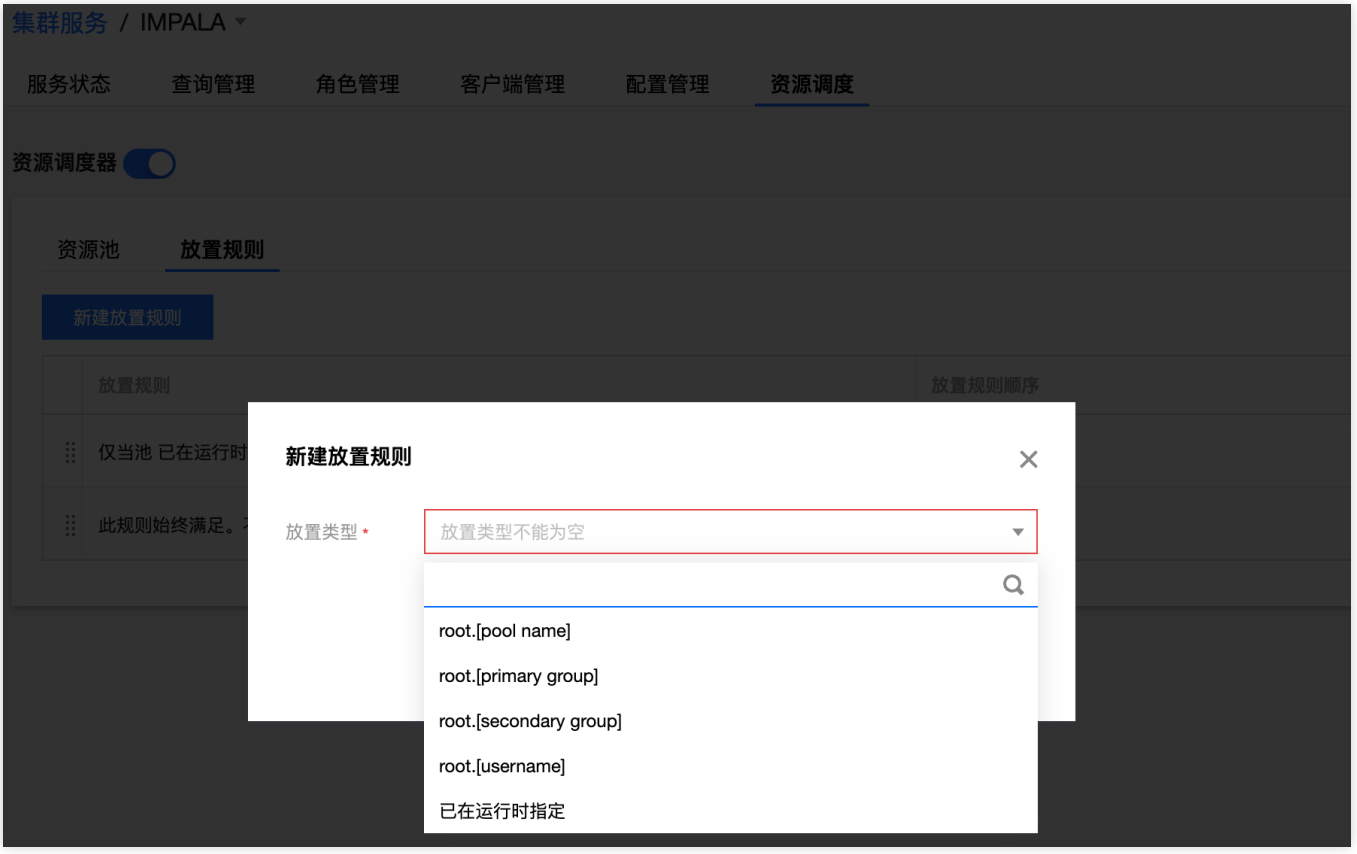
5. 点击部署生效后，资源池配置将下发生效，用户可以开始将查询提交到新增的资源池。

> 说明：
>
> - 重置操作将还原新建后尚未下发的资源池。
> - 若启用MEM_LIMIT查询选项，服务将允许用户通过该选项自定义单查询内存限制。

配置放置规则

1. 单击放置规则即可进入放置规则页面，单击新建放置规则即可进行放置规则的新建。

2. 填写放置类型和池名称。



配置规则类型说明：

- root.[pool name]：该规则始终满足，在其它规则不匹配的情况下使用，因此该规则默认要放置在所有匹配规则之后。
- root.[primary group]：该规则使用与该用户主要组匹配的资源池。Linux 中用户默认的主要组与用户名一致，匹配时会通过用户的主要组与资源池名称比对。
- root.[secondarygroup]：该放置规则用于匹配用户的次要组，使用与次要组之一匹配的资源池。
- root.[username]：该放置规则用于匹配与用户名一致的资源池。
- 已在运行时指定：该放置规则主要使用在运行时指定的资源池。
  放置规则的判断方式，根据放置规则的顺序1、2、3...进行判断，判断到满足条件的放置规则后，后续的规则不再进行匹配。

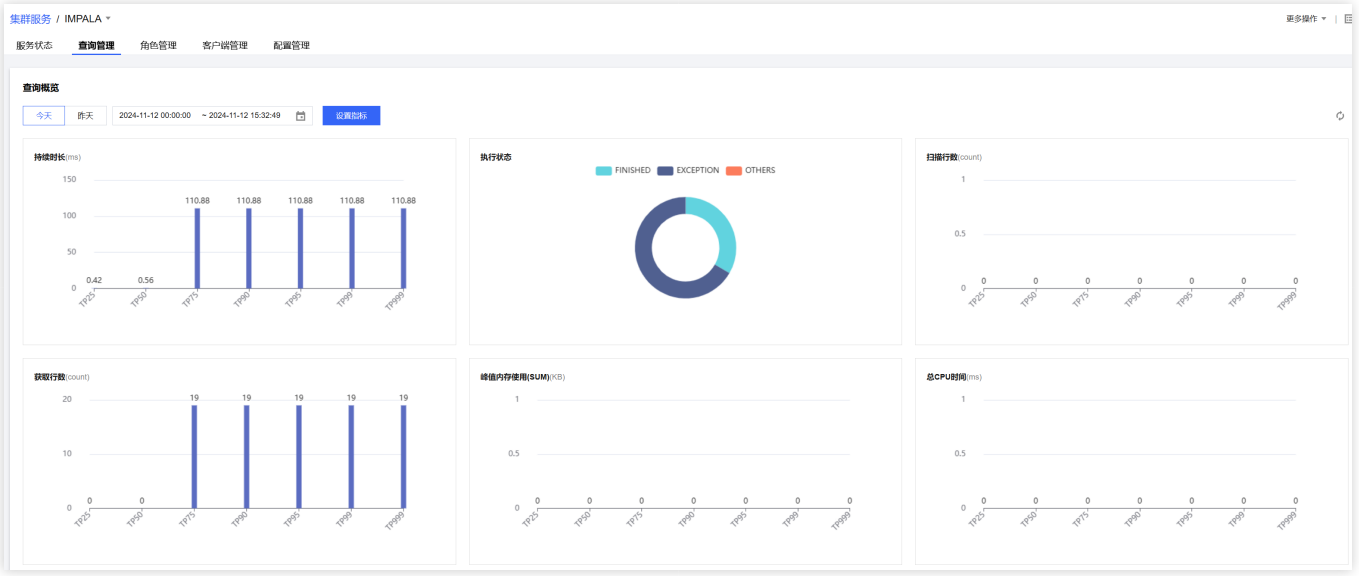
Impala作业查询

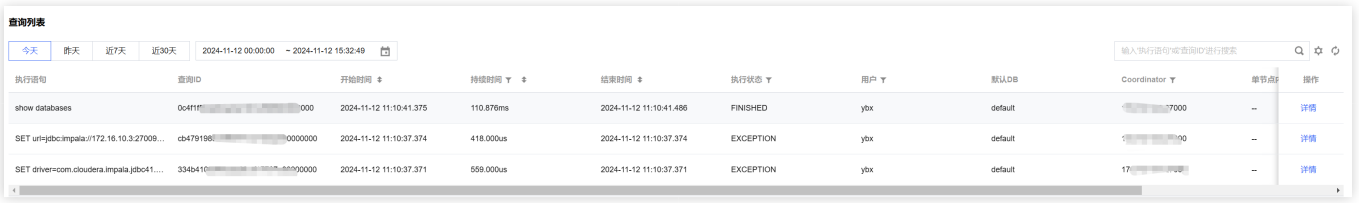
功能介绍

持 Impala 查询多种维度指标的分位分布视图，Impala 列表可快速查看查询语句、查询状态、用户、数据库、扫描行数、峰值内存使、总读取/发送 Bytes 量、HDFS 扫描数等多项明细指标。

操作步骤

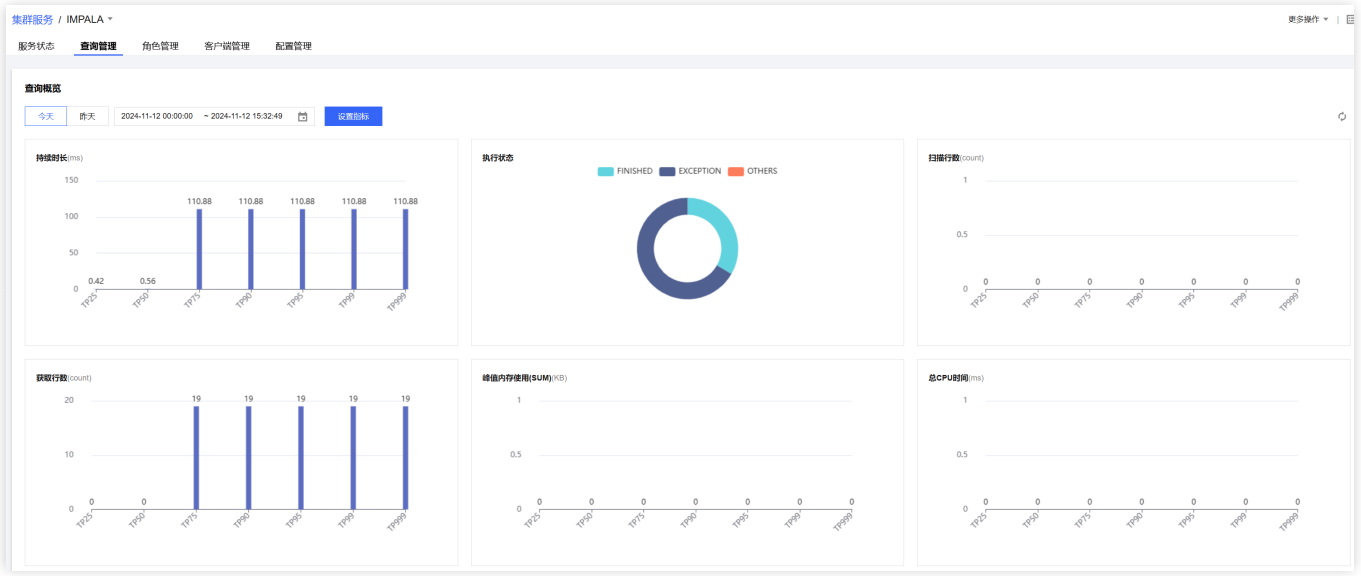
1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中单击集群服务，然后选择 Impala 组件右上角操作 > 查询管理，即可进行相关视图查看和 Impala 作业查询。





查询概览

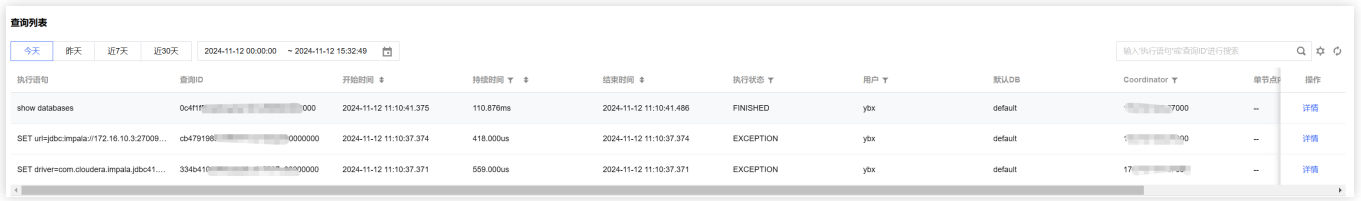
1. 提供 Impala 运行信息图表，持作业持续时间、执状态、扫描行数等信息概览。
   示例：以持续时长为例当前筛选时间范围内，TP90 分位时长为 6.86k(ms) 表示 90% 的查询时长在 6.86s 以内。



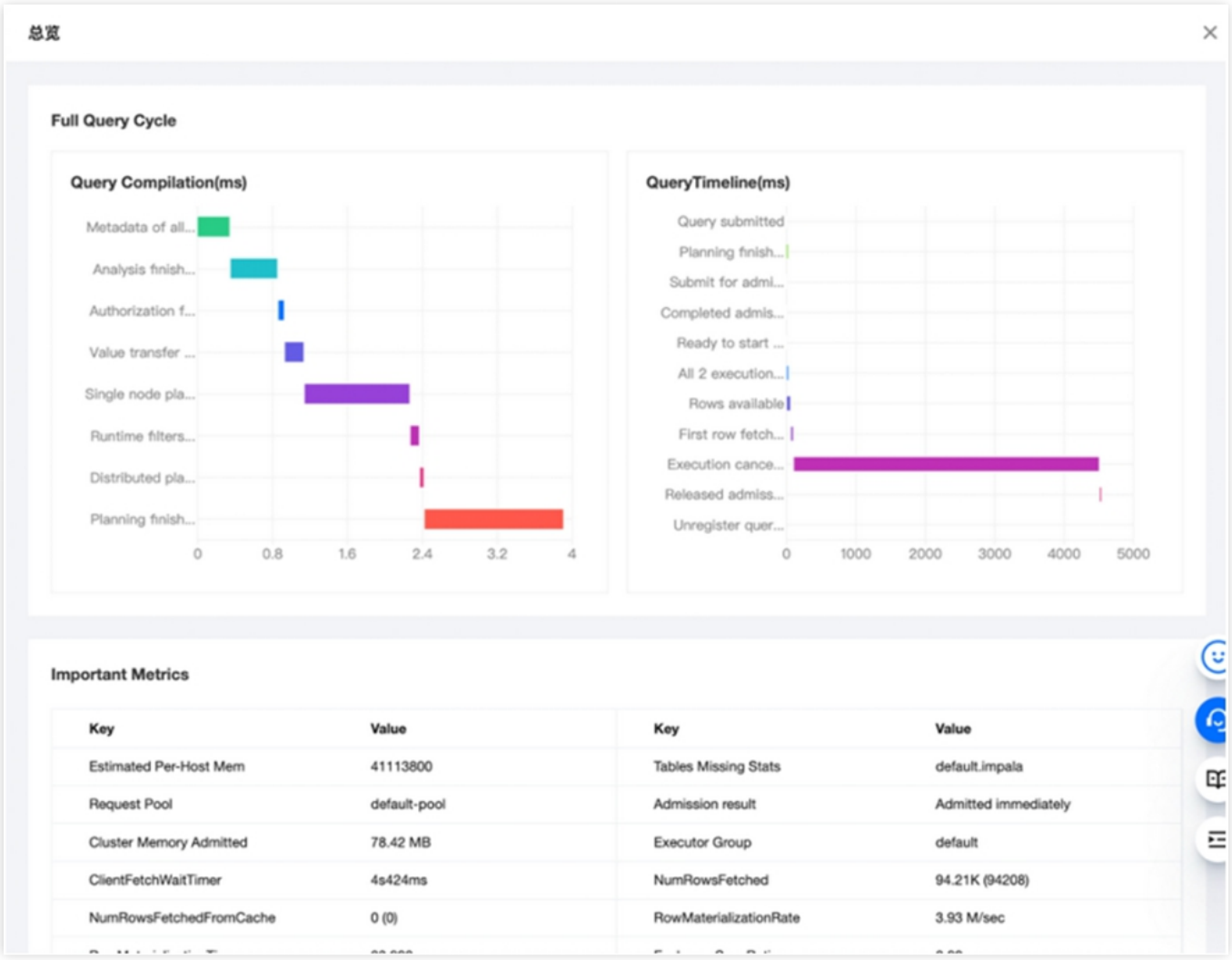
查询列表

执行时长超过 3s 的 Impala 查询提供查看总览和详情中的 Profile 功能。

1. 提供 Impala 查询列表信息，部分列头字段持筛选或排序功能，持多种维度的复合筛选操作。

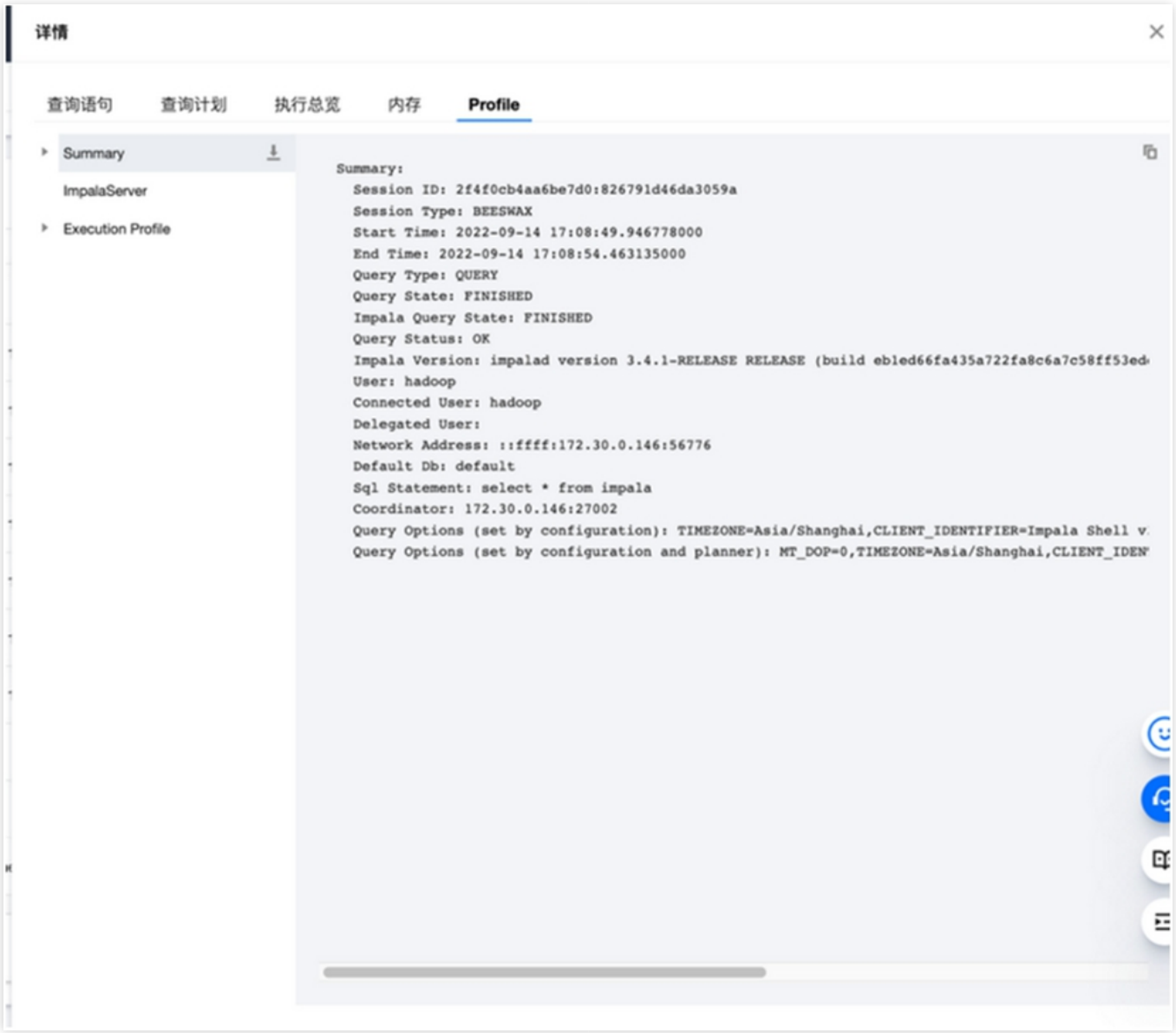


2. 操作列 > 总览可查看 Impala 查询的全生命周期的时间分布信息、重点指标信息及运行时的部分节点信息。



3. 操作列 > 详情中可查看查询语句、查询计划、执行总览、Profile、内存信息。

> 说明：
>
> 采集impala查询的Profile最大大小为100KB，如需采集超过该限制大小的Profile请联系相关运维人员调整参数。


详情
查询语句
查询计划
执行总览
内存
Profile
Summary
ImpalaServer
Execution Profile
Summary:
Session ID: 2f4f0cb4aa6be7d0:826791d46da3059a
Session Type: BEESWAX
Start Time: 2022-09-14 17:08:49.946778000
End Time: 2022-09-14 17:08:54.463135000
Query Type: QUERY
Query State: FINISHED
Impala Query State: FINISHED
Query Status: OK
Impala Version: impalad version 3.4.1-RELEASE RELEASE (build eb1ed66fa435a722fa8c6a7c58ff53ed
User: hadoop
Connected User: hadoop
Delegated User:
Network Address: ::ffff:172.30.0.146:56776
Default Db: default
Sql Statement: select * from impala
Coordinator: 172.30.0.146:27002
Query Options (set by configuration): TIMEZONE=Asia/Shanghai,CLIENT_IDENTIFIER=Impala Shell v
Query Options (set by configuration and planner): MT_DOP=0,TIMEZONE=Asia/Shanghai,CLIENT_IDEN

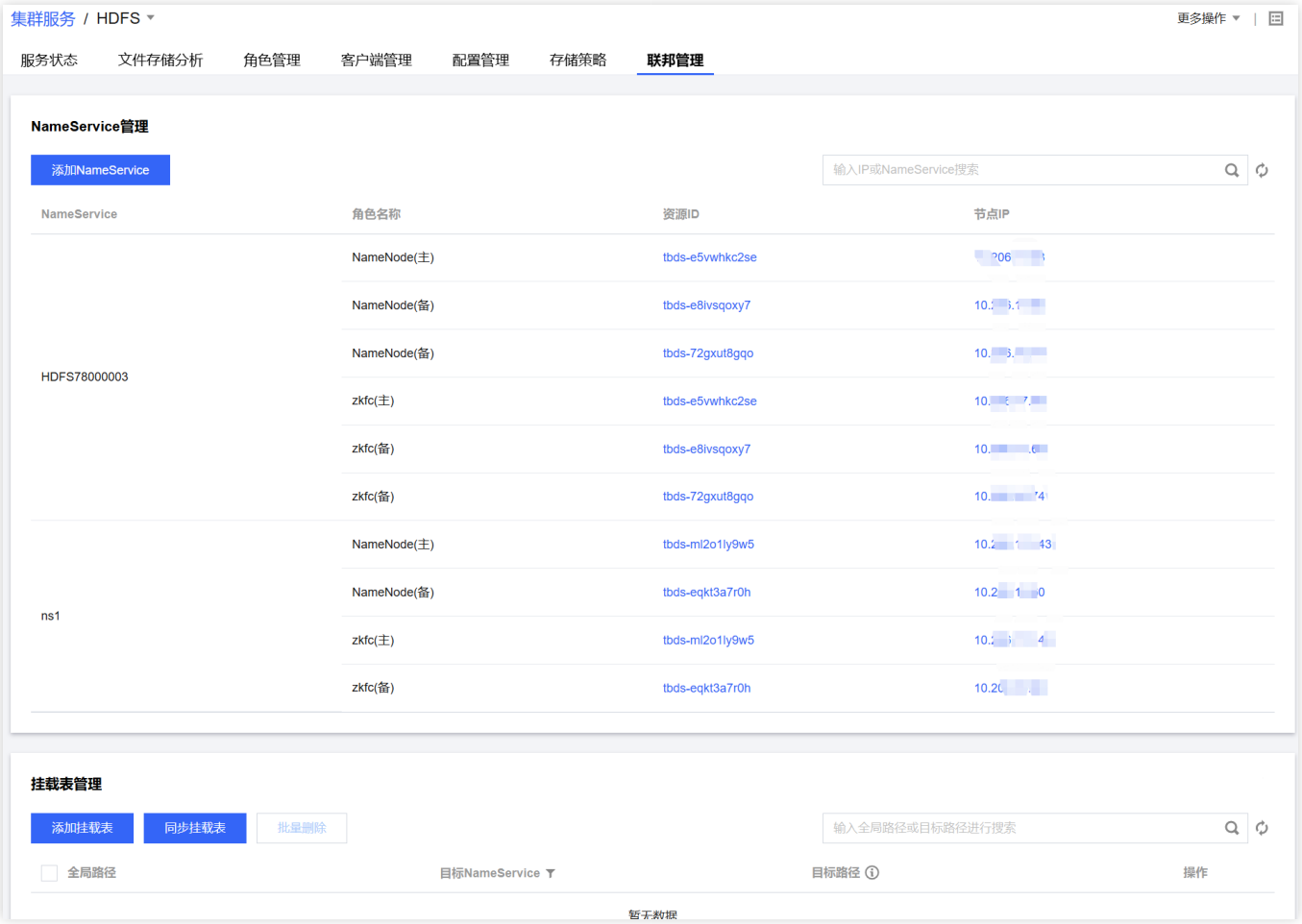
HDFS组件服务
HDFS联邦管理

功能介绍

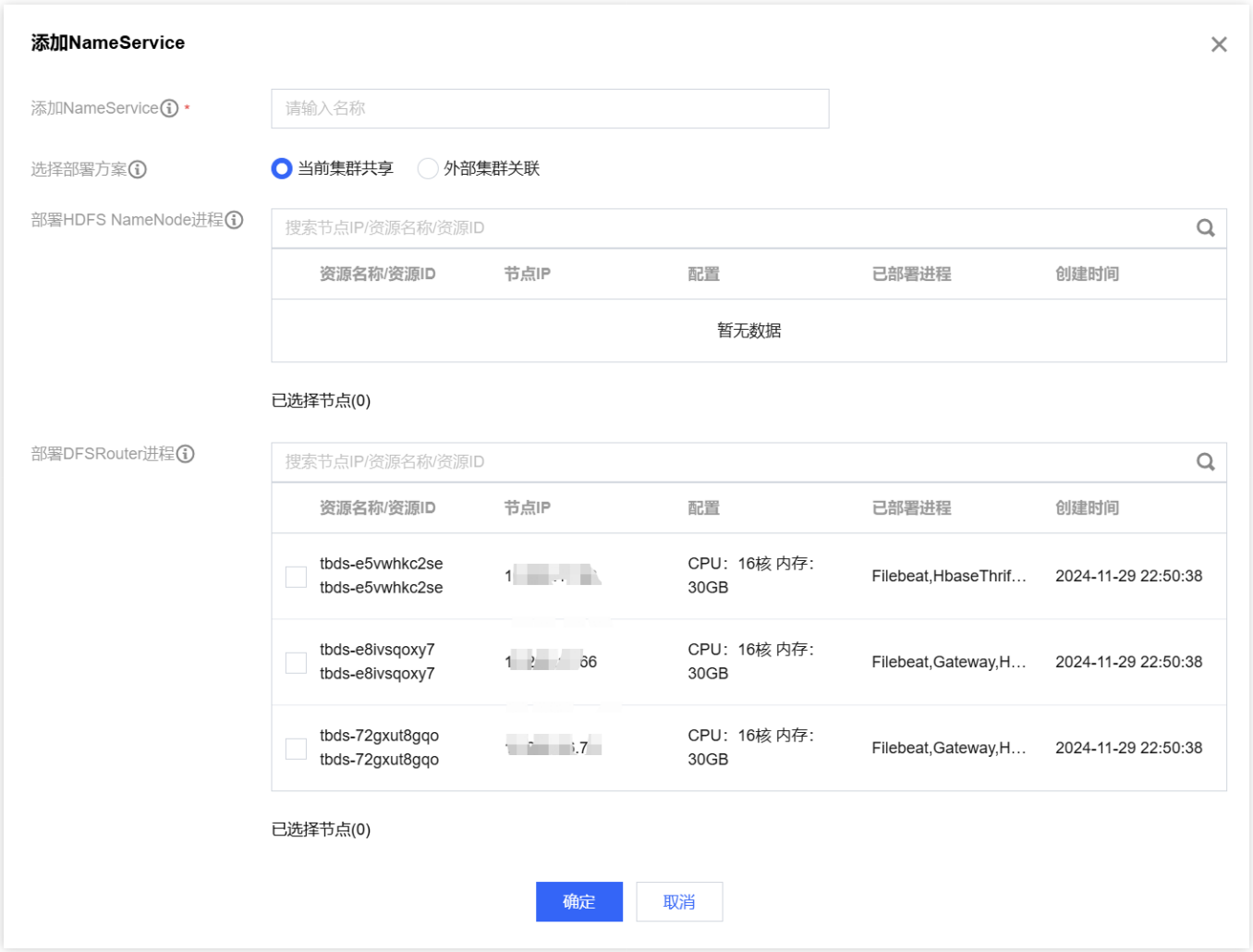
联邦管理功能支持用户对集群进行联邦拓展，为HDFS集群添加多个独立的HDFS NameService，实现HDFS元数据隔离，打破集群规模限制和性能瓶颈。TBDS支持通过管控平台进行HDFS NameService和Router的部署操作。

操作步骤

1. 登录 TBDS Manager管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 在集群详情页中选择集群服务 > HDFS 组件进入联邦管理页。



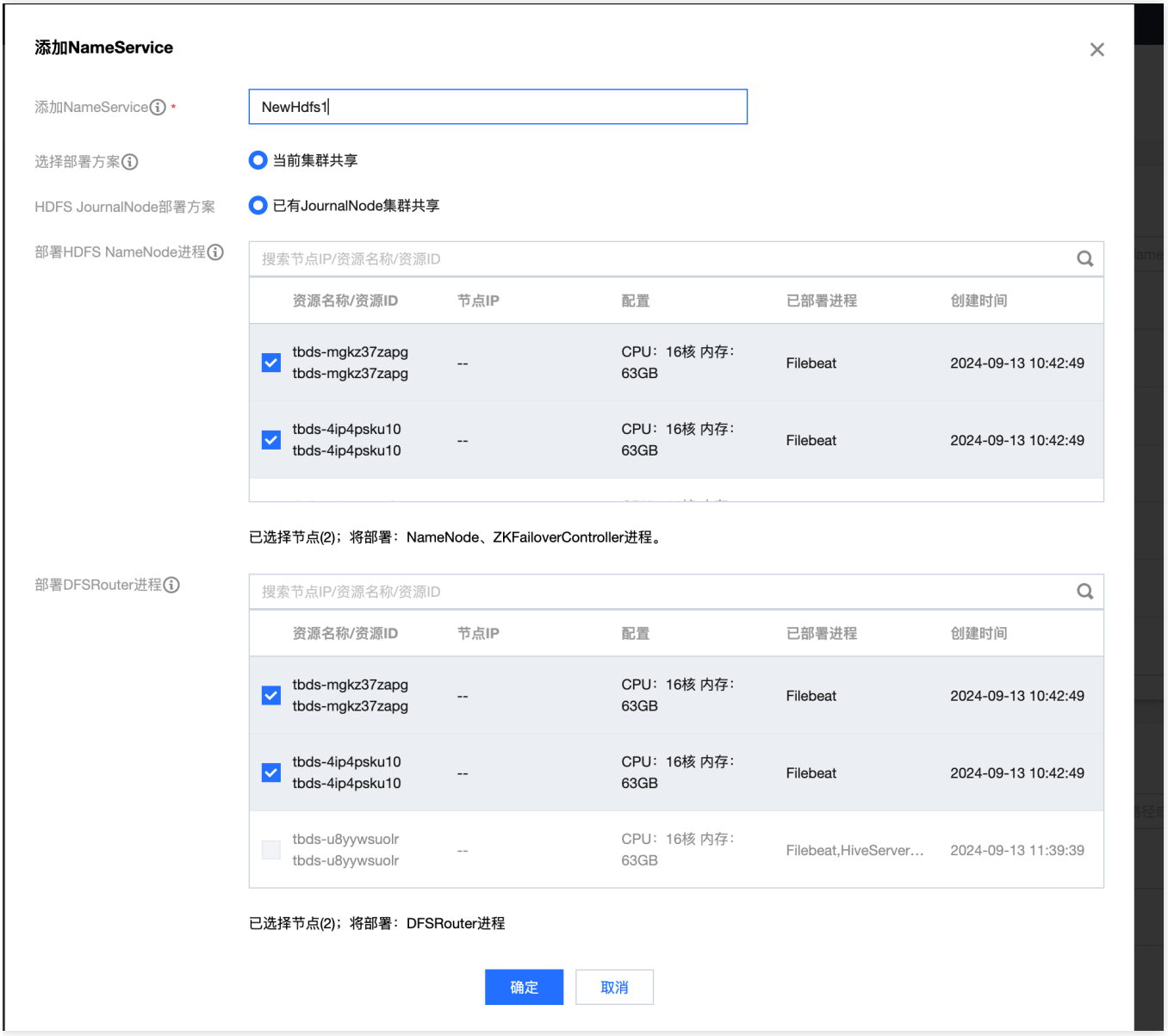
3. 点击添加NameService按钮，根据弹窗中的流程指引进行新增HDFS NameService流程。



4. 输入新建HDFS NameService的名称后，用户可以根据规划选择Master节点进行新HDFS NameService的配置，每个新增的HDFS NameService需要新增部署两个HDFS NameNode，支持部署在集群中任意尚未部署HDFS NameNode的Master节点。HDFS DFS Router默认与新增的HDFS NameNode 1:1部署在同个节点。
5. 点击确定后，任务中心将启动HDFS联邦添加NameService的工作流，进行相关进程的部署，并对更新集群内的HDFS和与HDFS相关的Spark，Hive，Amoro等服务的配置。
6. 待任务完成后，在联邦管理卡页中，用户可见新增的HDFS NameService信息。

> 说明：
>
> i. 新建联邦后，请重启集群部署的依赖HDFS的各个上层组件以生效配置修改，包括且不限于Yarn，Hive，Spark，Flink，Impala，Amoro，Kyuubi，Hue。
> ii. 当前版本数据管理暂不支持通过联邦管理新增HDFS NameService的查看和管理。
> iii. 当前版本暂不支持通过SDK操作在联邦管理新增的HDFS NameService。
> iv. Hue暂不支持查看在联邦管理新增的HDFS NameService的数据目录。
> v. 访问联邦集群时，请确认新增HDFS NameService已在Ranger中注册且已为用户设置目标对象的访问权限。
> vi. amoro需要在开启联邦后再安装，否则amoro的配置无法更新。

集群内开启



虚拟命名空间默认为nsfed，同时支持自定义。

限制：

1. 第一次要求选择2个router，后续可以选择0-n个router。
2. 联邦新添加的Namenode节点，不与其他组件混布。

集群间开启

虚拟命名空间默认为nsfed，同时支持自定义。

> 注意：
>
> 1. router相关命令，需要在router节点上执行。其他节点不支持运行router相关命令。
> 2. 第一次要求选择2个router，后续可以选择0-n个router。

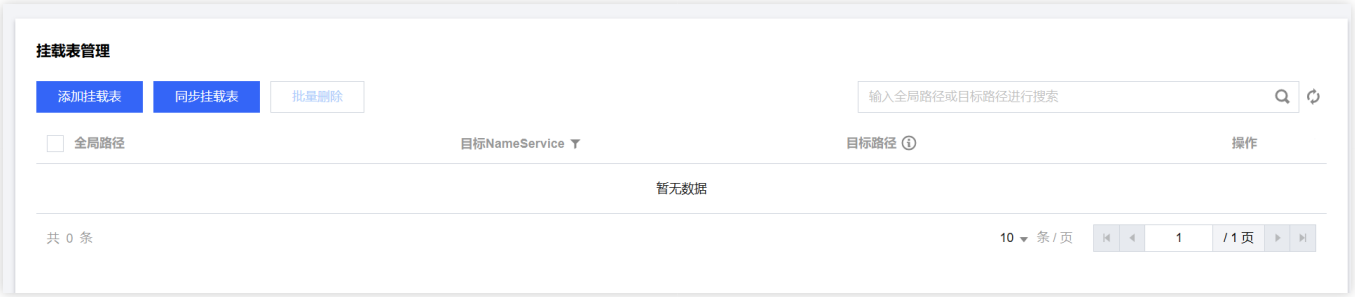
HDFS挂载表管理

功能介绍

在HDFS联邦模式下，挂载表允许用户将一个HDFS NameService的目录映射到另一个HDFS NameService的路径，通过这种方式，可以实现联邦数据访问和HDFS NameService资源共享。TBDS挂载表管理支持通过管控平台进行联邦集群挂载表的查看和管理，支持用户灵活添加、同步和删除挂载表中的挂载点。

操作步骤

1. 登录 TBDS Manager管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群服务 > HDFS 组件进联邦管理页。



3. 点击添加挂载表按钮，根据弹窗中的流程指引，选择目标HDFS NameService和进行添加挂载点流程。

4. 点击确定后，平台将开始下发新的挂载表到HDFS服务中，在添加完成后，用户将可以在挂载表中看到新增的挂载点，并在配置相应的数据访问权限后，通过全局路径访问到目标HDFS NameService中的目标路径。

> 说明：
>
> 配置挂载点时，若目标HDFS NameService中的目标路径不存在，挂载点将无法配置。

挂载表管理指令操作

1. 挂载表添加

   命令：

   hdfs dfsrouteradmin -add <source> <nameservice1, nameservice2, ...> <destination> -order <order>

   - <source>： 全局路径
   - <destination> ：挂载ns对应的目标路径，目标路径是实际存在的路径
   - <order>: 挂载表类型

   举例：

   - 单ns挂载：

   hdfs dfsrouteradmin -add /tmp ns1 /tmp
   hdfs dfsrouteradmin -add /data/app1 ns2 /data/app1
   hdfs dfsrouteradmin -add /data/app2 ns3 /data/app2

   - 多ns挂载：

   hdfs dfsrouteradmin -add /data/app3 ns1,ns2 /data/app3 HASH_ALL

2. 挂载表更新

   hdfs dfsrouteradmin -update /tmp ns1 /tmp2

3. 挂载表删除

   hdfs dfsrouteradmin -rm /tmp

4. 查看挂载表

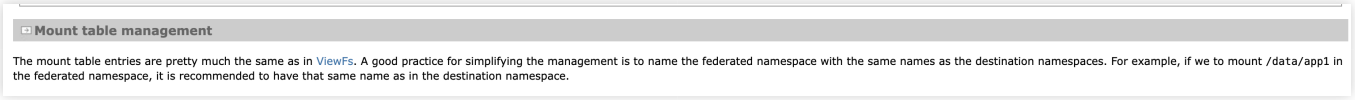
   hdfs dfsrouteradmin -ls
   -d ：显示order类型，默认不显示

目录/文件删除失败解决：

> 说明：
>
> 将所有用户的回收站目录挂载到所有ns上，比如当前集群有用户user1、user2和user3，ns有ns1和ns2，需要将这个三个用户回收站目录挂载到所有ns上。

```
hdfs dfsrouteradmin -add /user/user1/.Trash ns1,ns2   /user/user1/.Trash   -order HASH_ALL
hdfs dfsrouteradmin -add /user/user2/.Trash ns1,ns2   /user/user2/.Trash   -order HASH_ALL
hdfs dfsrouteradmin -add /user/user2/.Trash ns1,ns2   /user/user2/.Trash   -order HASH_ALL
```

- 如果某个目录没有进行挂载表挂载，访问这个目录时，将默认访问由dfs.federation.router.default.nameserviceId配置的ns目录。
- HDFS官方建议，RBF最好的实践是全局路径和目标路径同名。



RBF具体还可参考hadoop社区官方文档：
Apache Hadoop 3.3.6 – HDFS Router-Based Federation

Quota开启和设置

1. 开启quota，设置参数：dfs.federation.router.quota.enable=true，重启所有Router。

2. 设置quota命令：

   hdfs dfsrouteradmin -setQuota <path> -nsQuota <nsQuota> -ssQuota <ssQuota> <quota in bytes or quota size string>

   - -nsQuota ： 设置路径下目录和文件数配额。
   - -ssQuota ：设置空间配额，可以设置数字（字节为单位），也可以设置k/m/g为单位的字符串。
     举例：
   - 同时设置nsQuota和ssQuota：
     hdfs dfsrouteradmin -setQuota /path1 -nsQuota 100 -ssQuota 1024
     解释：表示给/path1路径设置了100目录和文件数配额，以及1024bytes空间配额。
   - 只设置nsQuota：
     hdfs dfsrouteradmin -setQuota /path2 -nsQuota 1000
     解释：表示给/path2路径设置了1000目录和文件数配额

- 只设置ssQuota：
  hdfs dfsrouteradmin -setQuota /path3 -ssQuota 10240
  解释：表示给/path3路径设置了10240bytes空间配额

3. 取消quota命令

```
hdfs dfsrouteradmin -clrQuota <path>
```

举例：
hdfs dfsrouteradmin -clrQuota /path

RBF具体还可参考hadoop社区官方文档：
Apache Hadoop 3.3.6 – HDFS Router-Based Federation

Router使用注意点

1. 子集群上被挂载的路径，不允许被移动/删除。
   > 说明：
   >
   > hdfs dfsrouteradmin /path1 ns1 /path2，则hdfs://ns1/path2不允许被移动/删除。

联邦使用建议

NameService 数量的规划建议

1. 单 NameService 存储容量超过安全数量（操作约束与限制），该 NameService 的访问速度、读写吞吐量都会严重降低。所以，需要新增 NameService。
2. 对于 HDFS 的重度使用（即大量读写 HDFS 中文件）应用，有必要单独划分一个 NameService 来处理其请求，以保证该应用的数据可以独占该 NameService 的所有处理能力，也避免了该应用对其他应用的影响。
3. 对于可靠性要求严苛的应用，可以划分一个 NameService，以免其他应用的过高的读写频率导致该应用无法访问 HDFS，造成该应用业务的不稳定。

业务数据目录挂载方式的规划建议

1. 业务数据的数据目录，尽量挂载在同一 NameService 下面。否则，跨 NameService 的文件读写速率较慢，会降低应用的文件存储性能。
2. 数据量大，且对其他服务的业务无关联的，可以直接使用一个 NameService。
3. 对业务量较小的业务，建议直接将其目录挂载在默认的 NameService，这样就不用做数据迁移，可以降低配置为 Federation 的复杂性。
4. 建议只对全局一级目录进行 NameService 映射以减少配置复杂度。

RBF组件示例

TBDS的RBF NameService默认名称为nsfed，要使用RBF，就需要组件使用nsfed来读写数据。

Router示例

1. 任意挂载nameservice 路径到全局路径。
   hdfs dfsrouteradmin -add /xxx NameService /path/
2. hdfs dfs -ls hdfs://nsfed/xxx.
   hdfs dfs -ls hdfs://NameService/path/
3. 访问Router webui

MR示例

1. 提交yarn任务，输入输出均为NameService的hdfs。

```
yarn jar /usr/local/service/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar word count hdfs://NameService1/xxx hdfs://NameService1/NS/yarn_wc_output1/
yarn jar /usr/local/service/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar word count hdfs://nsfed/xxx hdfs://NameService1/NS/yarn_wc_output1/
```

Hive示例

进入beeline，参考：

```
/usr/local/service/hive/bin/beeline -u "jdbc:hive2://{ip}:7001/;principal=hadoop/{ip}@{realm}"
```

1. hive on tez（tez未使用可忽略）

```
set hive.execution.engine=tez;
create database if not exists nsfed_db;
use nsfed_db;
drop table if exists nsfed_db.hive_hdfs_nsfed_tez;
create table hive_hdfs_nsfed_tez( id bigint, name string, price double) LOCATION 'hdfs://nsfed/path/to/directory';
insert into hive_hdfs_nsfed_tez values(100, '草莓', 80.5), (101, '石榴', 90);
select * from hive_hdfs_nsfed_tez order by id ASC;
```

2. hive on mr指定nsfed读写表

```
set hive.execution.engine=mr;
create database if not exists nsfed_db;
```

```
use nsfed_db;
drop table if exists nsfed_db.hive_hdfs_nsfed_tez;
create table hive_hdfs_nsfed_tez( id bigint, name string, price double) LOCATION 'hdfs://nsfed/path/to/directory';
insert into hive_hdfs_nsfed_tez values(100, '草莓', 80.5), (101, '石榴', 90);
select * from hive_hdfs_nsfed_tez order by id ASC;
```

Spark 示例

spark-sql 进入，参考：

```
./bin/./spark-sql --master yarn --principal hadoop/{ip}@{realm} --keytab /var/krb5kdc/emr.keytab
```

1. spark-sql创建iceberg表指定nsfed某路径

```
create database if not exists nsfed_db;
use nsfed_db;
drop table if exists nsfed_db.sparksql_hdfs_nsfed;
create table sparksql_hdfs_nsfed( id bigint, name string, price double) USING iceberg LOCATION 'hdfs://NameService/path/to/directory';
insert into sparksql_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
select * from sparksql_hdfs_nsfed order by id ASC;
```

2. spark-sql创建hive表指定nsfed某路径

```
create database if not exists nsfed_db;
use nsfed_db;
drop table if exists nsfed_db.sparksql_hdfs_nsfed;
create table sparksql_hdfs_nsfed( id bigint, name string, price double) LOCATION 'hdfs://nsfed/path/to/directory';
insert into sparksql_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
select * from sparksql_hdfs_nsfed order by id ASC;
```

3. spark-submit基本操作指定nsfed某路径
4. 准备输入数据并上传至hdfs

```
vim wordcount.txt
...
hello world
hello spark
hello hadoop
scala java
java Kyuubi
```

...
hdfs dfs -mkdir hdfs://nsfed/spark/resource
hdfs dfs -put ./wordcount.txt hdfs://nsfed/spark/resource
2、bin/spark-submit --master yarn --deploy-mode client --class org.apache.spark.examples.JavaWordCount /usr/local/service/spark/examples/jars/spark-examples*.jar hdfs://nsfed/spark/resource/wordcount.txt
（--conf spark.kerberos.access.hadoopFileSystems=hdfs://nsfed/ 该配置文件可添可不添加）

Flink示例

准备：

#进入 Flink 目录
cd /usr/local/service/flink

#先修改 conf/flink-conf.yaml 配置，增加 tm 的内存和 slot 数量
#taskmanager.heap.size: 1024m -> 3072m
#taskmanager.numberOfTaskSlots: 1 -> 10

#将 planner & hive connector 移入 lib 目录
mv ./opt/connector/flink-sql-connector-hive-3.1.2_2.12-1.16.1-TBDS-5.3.1_2023p4-SNAPSHOT.jar ./lib
mv ./opt/flink-table-planner_2.12-1.16.1-TBDS-5.3.1_2023p4-SNAPSHOT.jar ./lib/
mv ./lib/flink-table-planner-loader-1.16.1-TBDS-5.3.1_2023p4-SNAPSHOT.jar ./opt/

#清理 yarn-session 模式残留（如果存在残留文件会自动往 yarn-session 提交）
rm -rf /tmp/.yarn-properties-root

进入flink参考：./sql-client.sh embedded

1. flink创建hive表指定nsfed某路径

1、CREATE CATALOG hive_catalog WITH ('type'='hive','default-database'='default','hive-conf-dir'='/usr/local/service/hive/conf/', 'hive-version'='3.1.2');

#设置 checkpoint
set 'execution.runtime-mode' = 'streaming';
set 'execution.checkpointing.interval'='10000';

#设置 hive 方言
SET table.sql-dialect = hive;

#创建数据库
CREATE DATABASE hive_catalog.hive_db;

drop table hive_catalog.hive_db.t1;

```
-- 创建hive表
CREATE TABLE  hive_catalog.hive_db.t1 (uuid string, name string, age int, ts TIMESTAMP)
PARTITIONED BY (pt string)
LOCATION 'hdfs:/nsfed/tmp/hudi_flink/t1'
TBLPROPERTIES (
    'sink.partition-commit.policy.kind'='metastore'
);
2、插入数据：
INSERT INTO hive_catalog.hive_db.t1 PARTITION (pt = 'par1')  select'id1','Danny',23,TIMESTAMP '197
0-01-01 00:00:01';
SELECT * from hive_catalog.hive_db.t1;
3、启动flink集群
/usr/local/service/flink/bin/start-cluster.sh
#执行 kinit 认证
kinit -kt /var/krb5kdc/emr.keytab hadoop/*.*.*.*@TBDS-*******
开启flink 客户端
/usr/local/service/flink/bin/sql-client.sh
4、/usr/local/service/flink/bin/stop-cluster.sh
关闭flink集群
```

2. flink创建hudi表指定nsfed某路径

1、创建 catalog 信息

```
CREATE CATALOG hudi_catalog WITH ('type' = 'hudi','mode' = 'hms','default-database' = 'default','hive.
conf.dir' = '/usr/local/service/hive/conf','table.external' = 'true');

CREATE DATABASE hudi_catalog.hudi_db;
-- sets up the result mode to tableau to show the results directly in the CLI
set execution.result-mode=tableau;

CREATE TABLE hudi_catalog.hudi_db.t2 (
  id int PRIMARY KEY NOT ENFORCED,
  name VARCHAR(10),
  price int,
  ts int,
  dt VARCHAR(10)
)
PARTITIONED BY (dt)
WITH (
  'connector' = 'hudi',
  'path' = 'hdfs://HDFS78000004/tmp/hudi/t2',
  'table.type' = 'MERGE_ON_READ',
  'hoodie.datasource.write.keygenerator.class' = 'org.apache.hudi.keygen.ComplexAvroKeyGenerator',
  'hoodie.datasource.write.recordkey.field' = 'id',
  'hoodie.datasource.write.hive_style_partitioning' = 'true',
```

```
  'hive_sync.conf.dir'='/usr/local/service/hive/conf',
  'spark.version'='spark3.4.2',
  'hoodie.datasource.write.precombine.field'='ts'
);
```

2、插入数据：

```
INSERT INTO hudi_catalog.hudi_db.t2 VALUES(1,'Danny',23,15,'par1');
SELECT * from hudi_catalog.hudi_db.t2;
```

3、启动flink集群

```
/usr/local/service/flink/bin/start-cluster.sh
#执行 kinit 认证
kinit -kt /var/krb5kdc/emr.keytab hadoop/*.*.*.*@TBDS-*******
#开启flink 客户端
/usr/local/service/flink/bin/sql-client.sh
```

4、关闭flink集群

```
/usr/local/service/flink/bin/stop-cluster.sh
```

Trino示例

进入Trino 交互参考：

```
/usr/local/service/trino/client/trino-cli --server https://{ip}:9443 --user hadoop --krb5-config-path /et
c/krb5.conf --krb5-principal hadoop/{ip}@{realm} --krb5-keytab-path /var/krb5kdc/emr.keytab --krb5-
disable-remote-service-hostname-canonicalization --krb5-remote-service-name hadoop --insecure
```

1. Trino基本操作指定nsfed某路径

```
#1、库支持nsfed:
create SCHEMA if not exists hive.nsfed_db with (LOCATION='hdfs://nsfed/path/to/directory');
use hive.nsfed_db;
drop table if exists hive.nsfed_db.trino_hdfs_nsfed;
create table trino_hdfs_nsfed( id bigint, name varchar, price double);
insert into trino_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
select * from trino_hdfs_nsfed order by id ASC;

hdfs dfs -ls hdfs://nsfed/path/to/directory/trino_hdfs_nsfed
```

```
#2、表支持nsfed:
# ps：该表为外表，不可通过 Trino插入
create SCHEMA if not exists hive.nsfed_db_common;
use hive.nsfed_db_common;
drop table if exists trino_hdfs_nsfed;
create table trino_hdfs_nsfed( id bigint, name varchar, price double) with (EXTERNAL_LOCATION='hdf
s://nsfed/path/to/directory');
SHOW CREATE TABLE trino_hdfs_nsfed;
```

2. Trino操作iceberg指定联邦某路径

```
#1、库支持nsfed:
create SCHEMA if not exists iceberg.nsfed_db with (LOCATION='hdfs://nsfed/path/to/directory');
use iceberg.nsfed_db;
drop table if exists iceberg.nsfed_db.trino_hdfs_nsfed;
create table trino_hdfs_nsfed( id bigint, name varchar, price double);
insert into trino_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
select * from trino_hdfs_nsfed order by id ASC;

hdfs dfs -ls hdfs://nsfed/path/to/directory/xx/xx
#2、表支持nsfed:
# ps：该表为外表，不可通过 Trino插入
create SCHEMA if not exists iceberg.nsfed_db_common;
use iceberg.nsfed_db_common;
drop table if exists trino_hdfs_nsfed;
create table trino_hdfs_nsfed( id bigint, name varchar, price double) WITH (EXTERNAL_LOCATION='hdf
s://nsfed/path/to/directory');
SHOW CREATE TABLE trino_hdfs_nsfed;
```

Impala示例

进入Impala参考：

```
/usr/local/service/impala/shell/impala-shell -i {ip}:27009 -s hadoop

INVALIDATE METADATA;
```

1. Impala基本操作指定nsfed某路径

```
create database if not exists nsfed_db;
use nsfed_db;
drop table if exists nsfed_db.sparksql_hdfs_nsfed;
create table sparksql_hdfs_nsfed( id bigint, name string, price double) stored by 'org.apache.iceberg.mr
.hive.HiveIcebergStorageHandler' LOCATION 'hdfs://nsfed/path/to/directory';
spark-sql:
```

```
insert into sparksql_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
select * from sparksql_hdfs_nsfed order by id ASC;
```

Kyuubi示例

进入Kyuubi交互参考：

```
spark：
bin/beeline -u "jdbc:hive2:/{ip}:10009/default;principal=hadoop/{ip}@{realm};#Kyuubi.engine.type=SP
ARK_SQL;Kyuubi.engine.share.level=CONNECTION"

Trino：
bin/beeline -u "jdbc:hive2://{ip}:10009/default;principal=hadoop/{ip}@{realm};#Kyuubi.engine.type=TR
INO;Kyuubi.engine.share.level=CONNECTION"
```

1. Kyuubi on spark基本操作指定nsfed某路径

```
create database if not exists nsfed_db;
use nsfed_db;
drop table if exists nsfed_db.sparksql_hdfs_nsfed;
create table sparksql_hdfs_nsfed( id bigint, name string, price double) stored by 'org.apache.iceberg.mr
.hive.HiveIcebergStorageHandler' LOCATION 'hdfs://nsfed/path/to/directory';
spark-sql:
insert into sparksql_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
select * from sparksql_hdfs_nsfed order by id ASC;
```

2. Kyuubi on Trino基本操作iceberg指定nsfed某路径

```
create database if not exists nsfed_db;
use nsfed_db;
drop table if exists nsfed_db.trino_hdfs_nsfed;
create table trino_hdfs_nsfed( id bigint, name varchar, price double) USING iceberg LOCATION 'hdfs://n
sfed/path/to/directory';
insert into trino_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
select * from trino_hdfs_nsfed order by id ASC;
```

Hue示例

打开hue原生页面，在hue页面操作Trino。

```
create database if not exists nsfed_db;
use nsfed_db;
drop table if exists nsfed_db.sparksql_hdfs_nsfed;
create table sparksql_hdfs_nsfed( id bigint, name string, price double) LOCATION 'hdfs://nsfed/path/t
```

```
o/directory';
insert into sparksql_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
select * from sparksql_hdfs_nsfed order by id ASC;
```

Amoro示例

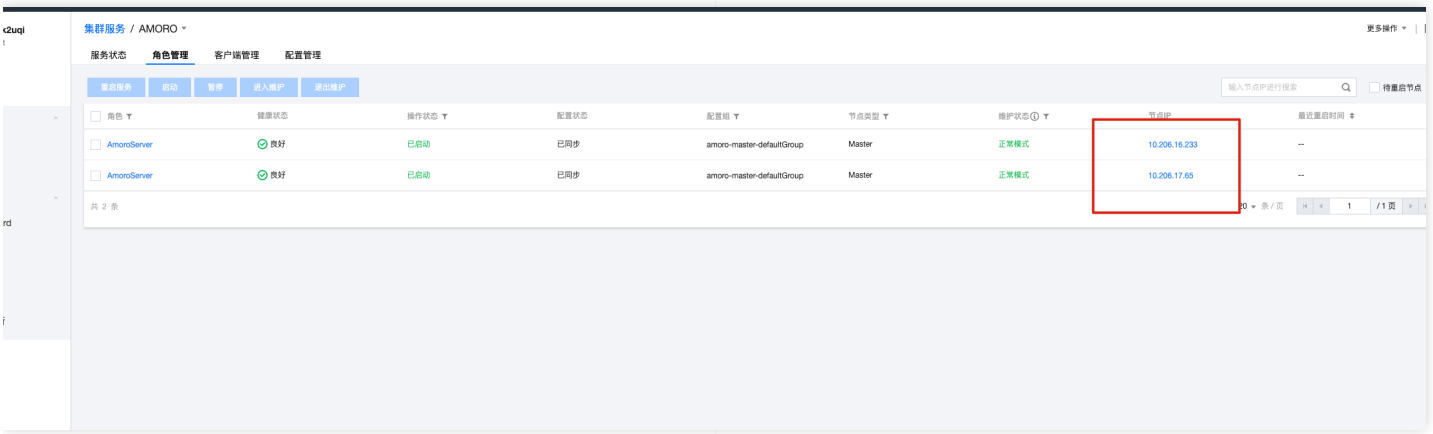
1. Amoro任务提交指定nsfed路径

```
#1、创建nsfed下的hive表并插入数据
set hive.execution.engine=tez;
create database if not exists nsfed_db;
use nsfed_db;
drop table if exists nsfed_db.hive_hdfs_nsfed_tez;
create table hive_hdfs_nsfed_tez( id bigint, name string, price double) LOCATION 'hdfs://nsfed/path/t
o/directory';
insert into hive_hdfs_nsfed_tez values(100, '草莓', 80.5), (101, '石榴', 90), (102, '石榴x', 91), (103, '石榴1', 9
2);
#2、Amoro页面对该表进行优化任务设置，
yarn页面任务运行正常
#3、创建新增NameService下的hive表并插入数据
set hive.execution.engine=tez;
create database if not exists nsfed_db;
use nsfed_db;
drop table if exists nsfed_db.hive_hdfs_nsfed_tez;
create table hive_hdfs_nsfed_tez( id bigint, name string, price double) LOCATION 'hdfs://NameService/
path/to/directory';
insert into hive_hdfs_nsfed_tez values(100, '草莓', 80.5), (101, '石榴', 90), (102, '石榴x', 91), (103, '石榴1', 9
2);
#4、Amoro页面对该表进行优化任务设置，
yarn页面任务运行正常
#5、Trino创建nsfed下的iceberg表并插入数据
create SCHEMA if not exists iceberg.nsfed_db with (LOCATION='hdfs://nsfed/path/to/directory');
use iceberg.nsfed_db;
drop table if exists iceberg.nsfed_db.trino_hdfs_nsfed;
create table trino_hdfs_nsfed( id bigint, name varchar, price double);
insert into trino_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
#6、Amoro页面对该表进行优化任务设置，
yarn页面任务运行正常
#7、trino创建新增NameService下的iceberg表并插入数据
create SCHEMA if not exists iceberg.nsfed_db with (LOCATION='hdfs://NameService/path/to/director
y');
use iceberg.nsfed_db;
drop table if exists iceberg.nsfed_db.Trino_hdfs_nsfed;
create table trino_hdfs_nsfed( id bigint, name varchar, price double);
insert into trino_hdfs_nsfed values(100, '草莓', 80.5), (101, '石榴', 90);
#8、Amoro页面对该表进行优化任务设置，
```

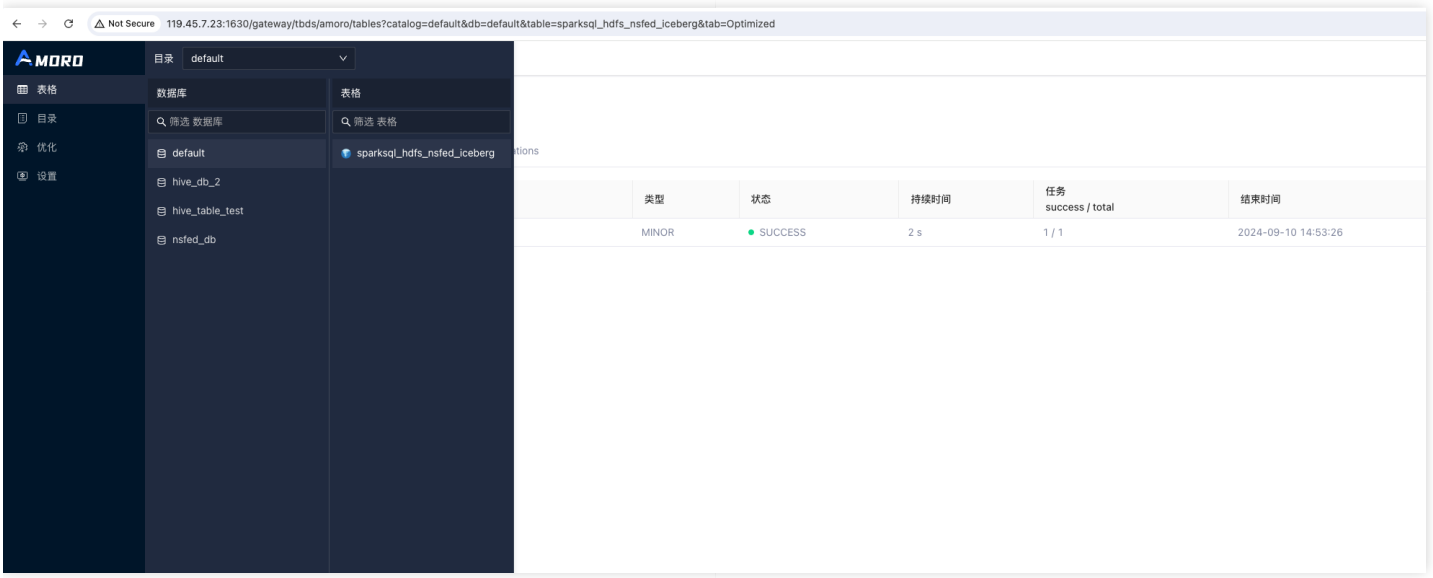
yarn页面任务运行正常

页面操作可参考：

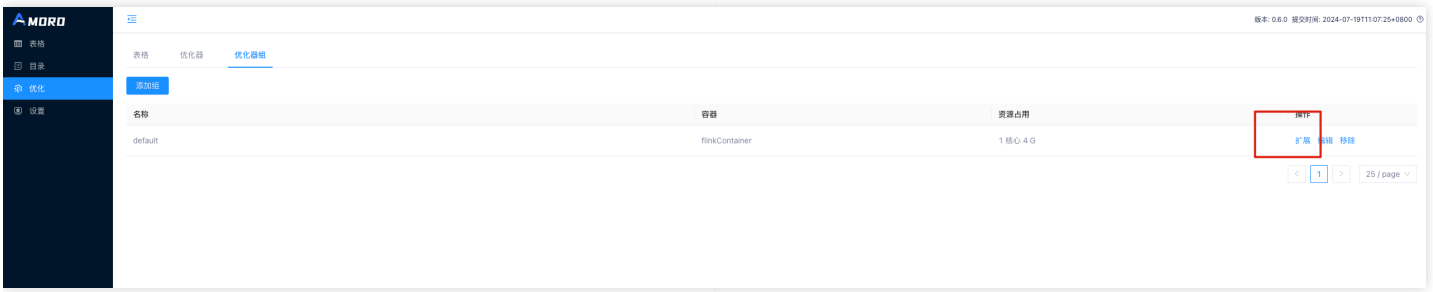
第一步：找到 Amoro 安装节点的外网 IP。



第二步：外网ip:1630 访问 Amoro webui。



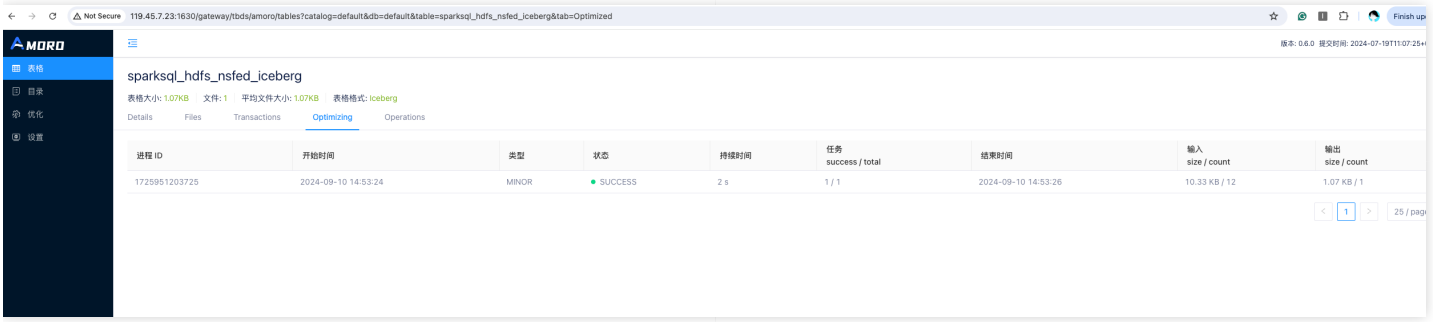
第三步：优化器组扩展一个优化器。



第四步：建 iceberg 表，不停插入数据。

```
 Iceberg 1.4.3 (commit 9a5d24fee239352021a9a73f6a4cad8ecf464f01)}} (org.apache.iceberg.metrics.LoggingMetr
2024-09-10 14:52:11,257 [INFO] [main] Committed in 236 ms (org.apache.iceberg.spark.source.SparkWrite(org.
2024-09-10 14:52:11,257 [INFO] [main] Data source write support IcebergBatchWrite(table=spark_catalog.defa
Time taken: 0.326 seconds
2024-09-10 14:52:11,262 [INFO] [main] Time taken: 0.326 seconds (org.apache.spark.sql.hive.thriftserver.Sp
spark-sql (default)> insert into sparksql_hdfs_nsfed_iceberg values(1, "a", 1);
```

第五步： Amoro 监测到有小文件，自动开启优化。



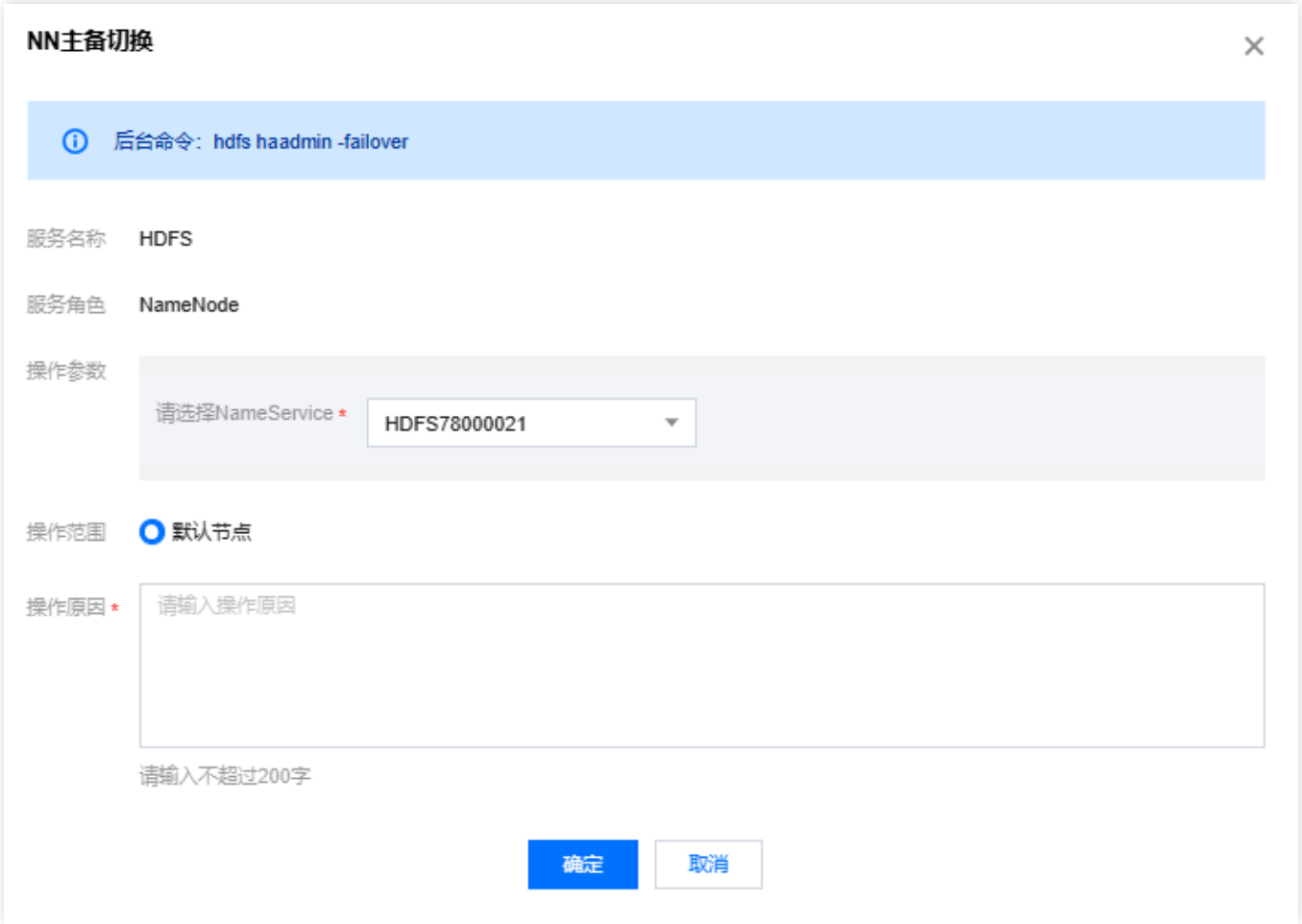
服务操作

功能介绍

服务操作是向组件提供的便捷运维管理具集合，包含通的服务重启，以及部分组件特有的指令类运维操作，如 HDFS NameNode 主备切换、HDFS 数据均衡、YARN ResourceManager 主备切换、YARN 刷新队列等。

前提条件

1. YARN ResourceManager 主备切换需禁 YARN.resourcemanager.ha.automatic-failover.enabled。若 RM 主备切换未在 YARN 卡操作下拉框中显，请在 YARN 配置管理-配置件 YARN-site.xml 中找到 YARN.resourcemanager.ha.automatic-failover.enabled，并对其进禁。
2. YARN 刷新队列时，不要删除调度器的配置件中已效的队列。
3. Ranger 修改元数据库，当前持 Mysql 和 MariaDB 数据库，且测试连接仅测试管理员户的连接。



操作步骤

1. 登录TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群服务，选择需要操作的组件卡。
3. 以 HDFS NN 主备切换为例，在集群服务中，选择 HDFS 组件卡操作 > NN 主备切换进主备切换操作。

服务操作清单

通的服务操作包含服务重启、启动、暂停、进/退出维护模式、查看端；指令类服务操作包含 HDFSNameNode 主备切换、HDFS 数据均衡、YARN ResourceManager 主备切换、YARN 刷新队列等，操作说明如下：

服务操作	说明
HDFS NameNode 主备切换	简称 NN 主备切换，将当前处于 Active 状态的 NameNode 转成 StandBy 状态，并将原先处于 StandBy 状态的 NameNode 转成 Active 状态。
HDFS 数据均衡	通常需要在有新 DataNode 加时执，本操作会使数据分布均匀，避免热点问题，使集群读写负载更均衡。
YARNResourceManager 主备切换	简称 RM 主备切换，将当前处于 Active 状态的 ResourceManager 转成 StandBy 状态，并将原先处于 StandBy 状态的 ResourceManager 转成 Active 状态。RM 主备切换只有当 YARN.resourcemanager.ha.automatic-failover.embedded 禁时才允许操作。 若 RM 主备切换未在 YARN 卡操作下拉框中显，请在 YARN 配置管理-配置件 YARN-site.xml 中找到 YARN.resourcemanager.ha.automatic-failover.enabled，并对其进禁。
YARN 刷新队列	当 capacity-scheduler.xml、fair-scheduler.xml 新增或更新内容时，本操作可以使这些内容在 ResourceManager 中效。注意，不要去删除 capacityscheduler.xml、fair-scheduler.xml 中定义的已效的队列。
Ranger 修改元数据库	当需要更改 Ranger 底层的数据库时，需要修改 conf/install.properties 件，然后在本地执 setup.sh 脚本，本操作提供键配置元数据库功能，避免户修改 Ranger 元数据库地址时因改漏配置导致服务异常。 本操作当前持 MySQL 和 MariaDB 数据库，且测试连接功能仅于测试管理员户的连接。 本操作将数据库的信息同步到本地的 ranger-admin-site.xml 配置件中，但是不会同步修改配置管理中 ranger-admin-site.xml 的内容，若户因为额外的需求在配置管理页修改并下发 ranger-admin-site.xml，会导致数据库信息被覆盖，从导致异常。

名称解释

- HDFS NameNode 主备切换：简称 NN 主备切换，将当前处于 Active 状态的 NameNode 转成 StandBy 状态，

并将原先处于 StandBy 状态的 NameNode 转成 Active 状态。

- HDFS 数据均衡：通常需要在有新 DataNode 加[illegible]时执[illegible]，本操作会使数据分布均匀，避免热点问题，使集群读写负载更均衡。
- YARN ResourceManager 主备切换：简称RM主备切换，将当前处于 Active 状态的 ResourceManager 转成 StandBy 状态，并将原先处于 StandBy 状态的 ResourceManager 转成 Active 状态。RM 主备切换只有当 YARN.resourcemanager.ha.automatic-failover.embedded 禁[illegible]时才允许操作。
- YARN 刷新队列：当 capacity-scheduler.xml、fair-scheduler.xml 新增或更新内容时，本操作可以使这些内容在 ResourceManager 中[illegible]效。注意，不要去删除capacity-scheduler.xml、fair-scheduler.xml中定义的队列。
- Ranger 修改元数据库：当需要更改 Ranger 底层的数据库时，需要修改 conf/install.properties[illegible]件，然后在本地执[illegible] setup.sh 脚本，该脚本的功能之[illegible]就是将数据库的信息同步到本地的 ranger-adminsite.xml 配置[illegible]件中，但是这并没有同步修改配置管理中 ranger-admin-site.xml 的内容，当[illegible]户因为额外的需求在配置管理页修改并下发 ranger-admin-site.xml，则会导致数据库信息被覆盖，导致异常；本操作提供[illegible]键配置元数据库功能，避免[illegible]户修改Ranger 元数据库地址时因改漏配置导致服务异常。

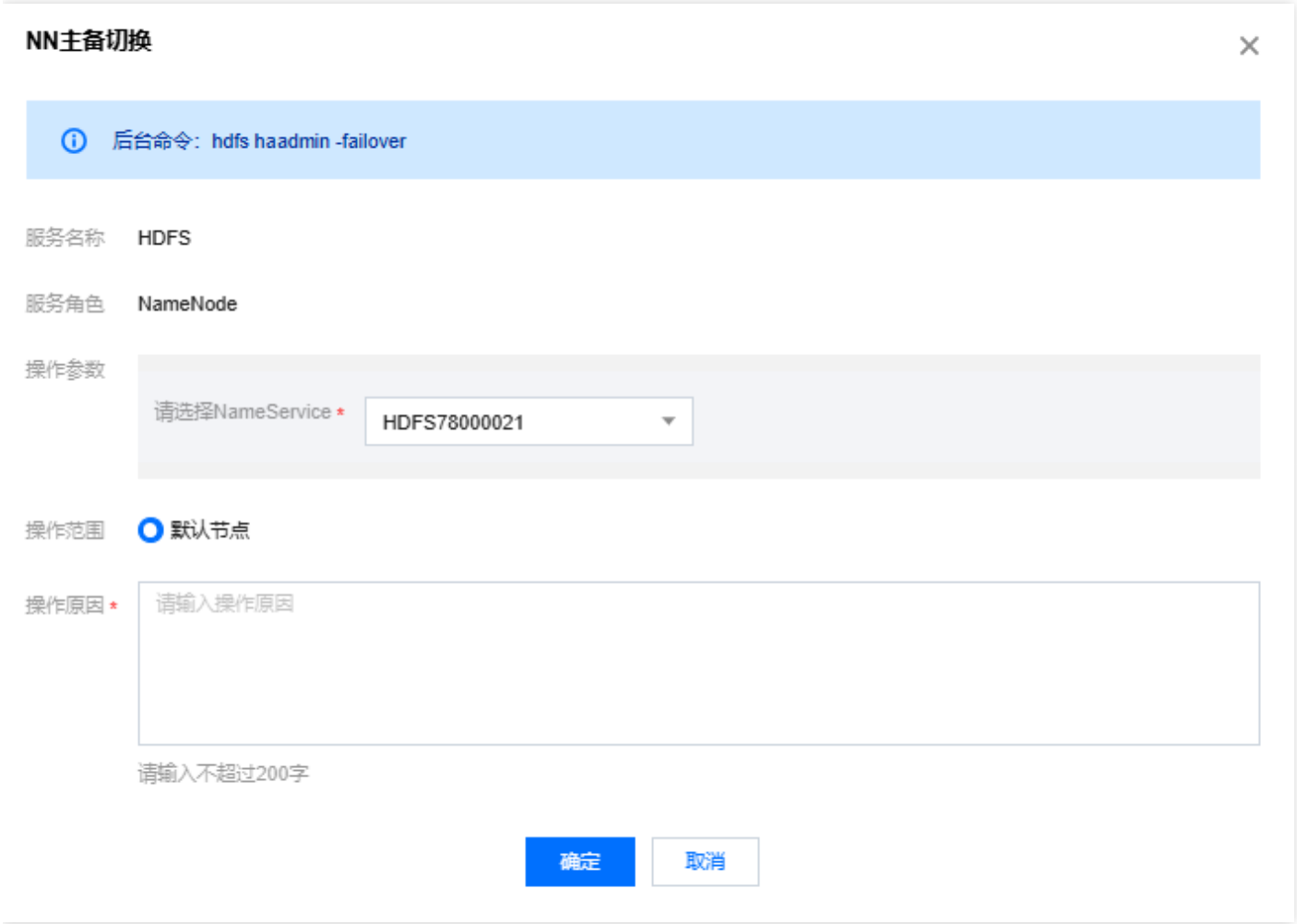
NN主备切换

功能介绍

HDFS NameNode 主备切换：简称 NN 主备切换，将当前处于 Active 状态的 NameNode 转成 StandBy 状态，并将原先处于 StandBy 状态的 NameNode 转成 Active 状态。

操作步骤

HDFS NN 主备切换，在集群服务中，选择 HDFS 组件卡片操作 > NN 主备切换进主备切换操作。



数据均衡

功能介绍

通常需要在有新 DataNode 加入时执行，本操作会使数据分布均匀，避免热点问题，使集群读写负载更均衡。

操作步骤

1. 在集群服务中，选择 HDFS 组件卡片操作 > 数据均衡进行操作。

2. 操作参数填写说明。

阈值(-threshold)	值为大于1且小于100的整数
平衡策略(-policy)	datanode(默认)或者blockpool
排除的主机列表(-exclude)	排除指定的datanode，参数：由逗号分隔的主机列表或者-f [指定主机列表的hdfs文件路径]
包含的主机列表(-include)	只包含指定的datanode，参数：由逗号分隔的主机列表或者-f 指定主机列表的hdfs文件路径
数据源的主机列表(-source)	只选择指定的datanode作为源节点，参数：由逗号分隔的主机列表或者-f 指定主机列表的hdfs文件路径
块池(-blockpools)	要均衡的块池，多个块池由逗号分隔
退出前的空闲迭代次数(-idleiterations)	退出前的最大空闲迭代次数，默认值为5
升级集群时是否运行Balancer(-runDuringUpgrade)	是/否

3. 操作范围：可选择默认节点，也可手动指定节点。

数据均衡

服务名称 HDFS

服务角色 NameNode

操作参数

阈值(-threshold) * 10

高级设置

平衡策略(-policy) datanode

排除的主机列表(-exclude) 请输入

包含的主机列表(-include) 请输入

数据源的主机列表(-source) 请输入

块池(-blockpools) 请输入

退出前的空闲迭代次数(-idleiterations) 请输入

升级集群时是否运行balancer(-runDuringUpgrade) true false

操作范围 默认节点 指定节点

操作原因 * 请输入操作原因

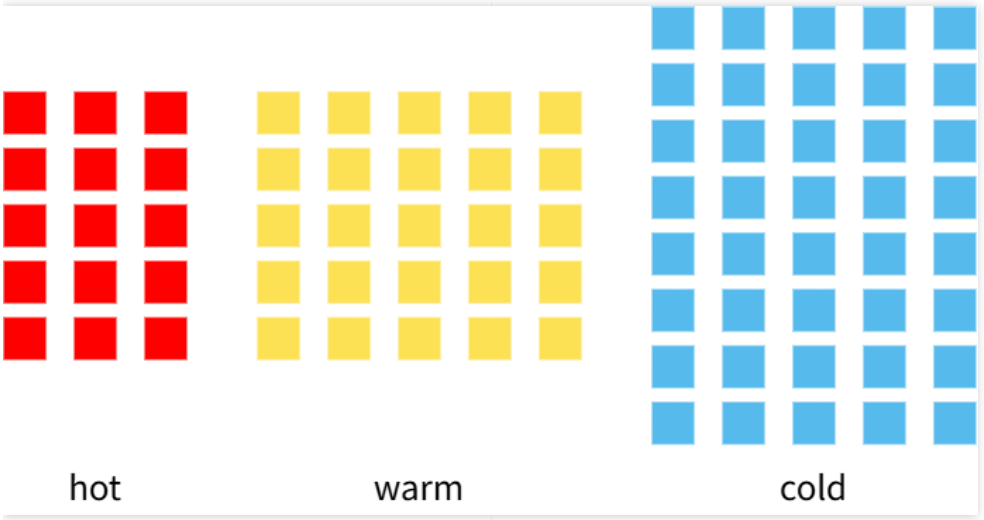
请输入不超过200字

确定 取消

HDFS 存储策略

简要介绍

HDFS 存储策略是结合不同类型的存储介质产出的一种文件按照异构存储自动化调配的解决方案。TBDS 集群管理支持 HDFS 的存储策略可视化配置，支持用户使用 HDFS 存储策略在大规模集群中合理利用数据的热、温、冷生命周期来对 Hadoop 集群降本增效。



存储类型和存储策略概述

- 存储类型包括 ARCHIVE、DISK、SSD 和 RAM_DISK ，不同类型适用于不同场景。
- 存储策略包括热（Hot）、温（Warm）、冷（Cold）、全 SSD（All_SSD）、单 SSD（One_SSD）、延迟持久化（Lazy_Persist）和 智能策略（TBDS优化），每种策略规定了数据在不同存储类型上的存放方式。

策略	副本分布规则	适用场景
Hot（默认）	所有副本存于DISK或SSD	高频访问数据
Cold	所有副本存于ARCHIVE	归档/极少访问数据
Warm	1副本存于DISK，其余存ARCHIVE	温数据
All_SSD	所有副本存于SSD	极高IO需求数据
One_SSD	其中一个副本存储在 SSD，其余副本存储在DISK/HDD	高频访问数据，且业务 IO 要求高
Lazy_Persist	用于写入内存中具有单个副本的块，该副本首先写入RAM_DISK，然后延迟持久化到DISK中	高吞吐日志/临时数据处理
智能策略（TBDS优化）	将根据最近访问时间或最后修改时间对 HDFS 存储策略进行调度，针对选中的目录，周期内没有访问记录子目录将被降级存储策略等级，启用智能策略时热数据存储策略为 Hot，温数据存储策略为 Warm，冷数据存储策略为 Cold。	智能调配场景

TBDS 提供对 HDFS 组件存储策略的可视化配置和管理功能，供用户在业务使用中，提高使用效率。
HDFS 社区对存储策略能力说明更多可参考：HDFS存储策略

参考实践

> 注意：
>
> 下文为示例参考，实际项目请联系交付工程师结合业务情况综合评估后实施。

实施前提：完成服务器资源规划准备，按节点类型分配硬件，用于 集群部署安装/扩容阶段使用。例如：

- 节点磁盘规划（如果仅用到热温类型，可以简单规划两种类型，即热（SSD）、温（SATA））

节点类型	节点角色（HDFS + 计算）	磁盘配置（单节点）	关联存储策略（核心适配）	节点内存储类型配置（dfs.datanode.data.dir 示例）	用途说明	适用场景	规划建议（硬件 / 规模）
热数据节点	Combined Node（DataNode + YARN NodeManager）	磁盘类型：SATA/SAS DISK（平衡性能与成本） 磁盘数量：12-24 块（多盘提升吞吐量） 单盘大小：1-4TB（避免单盘故障影响过大）	1. 核心：Hot（默认策略） 1. 兼容：Warm（DISK 部分）、One_SSD（DISK 部分）	[DISK]file:///data/disk0,[DISK]file:///data/disk1,...,[DISK]file:///data/disk23	存储高频访问、需计算的数据，承担 YARN 计算任务	实时计算（Flink/Spark Streaming）输入输出、在线查询（Hive/Impala）结果存储、业务核心数据	1. 硬件：选带 RAID 卡的服务器（RAID 0/10，平衡性能与可靠性） 1. 规模：占集群总节点 30%-50%（业务核心）
冷数据节点	Pure DataNode（仅存储，无计算）	磁盘类型：大容量 SATA DISK（ARCHIVE 类型，低成本高密度） 磁盘数量：4-8 块（单盘容量大，减少盘位占用） 单盘大小：8-16TB（最大化存储密度）	1. 核心：Cold（归档策略） 1. 兼容：Warm（ARCHIVE 部分）	[ARCHIVE]file:///data/archive0,[ARCHIVE]file:///data/archive1,...,[ARCHIVE]file:///data/archive7	存储低频访问、无需计算的归档数据，不承担计算任务	历史日志归档（如 3 个月以上 Nginx 日志）、离线分析结果备份（如年度报表数据）、冷备份数据	1. 硬件：选低成本服务器（无需高性能 CPU / 内存，重点堆磁盘容量） 1. 规模：占集群总节点 40%-60%（存储成本敏感）
高性能节点	Combined Node（DataNode + 高性能计算节点）	磁盘类型 1：NVMe SSD（低延迟，All_SSD 策略） 磁盘数量 1：4-8 块 单盘大小 1：1-2TB	核心：All_SSD（全 SSD 存储）、One_SSD（单 SSD 副本）	[SSD]file:///data/ssd0,[SSD]file:///data/ssd1,...,[SSD]file:///data/ssd7	存储高 IO 需求、低延迟要求的数据，承担高性能计算任务	实时推荐系统特征数据、高频查询元数据（如 HBase RegionServer 存储）、低延迟分析场景	1. 硬件：配高主频 CPU（如 Intel Xeon Gold）+ 大内存（128-256GB），匹配 SSD I/O 能力 1. 规模：占集群总节点 5%-15%（核心高性能场景）
临时数据节点	Combined Node（DataNode + 临时计算节点）	磁盘类型 1：RAM_DISK（内存盘，Lazy_Persist 策略） 容量 1：64-256GB（通过 tmpfs 挂载） 磁盘类型 2：SATA DISK（持久化备份） 磁盘数量 2：4-8 块 单盘大小	核心：Lazy_Persist（单副本临时存储）	[RAM_DISK]file:///data/ram0,[DISK]file:///data/disk0,[DISK]file:///data/disk1,...,[DISK]file:///data/disk7	存储短期临时数据（如计算中间结果），支持延迟持久化	Spark/Flink 作业中间结果、临时文件缓存（如 MapReduce Shuffle 临时数据）	1. 硬件：重点配大内存（256-512GB），RAM_DISK 容量不超过物理内存的 50%（避免 OOM） 1. 规模：占集群总节点 5%-10%（临时计算场景）

节点类型	节点角色（HDFS + 计算）	磁盘配置（单节点）	关联存储策略（核心适配）	节点内存储类型配置（dfs.datanode.data.dir 示例）	用途说明	适用场景	规划建议（硬件 / 规模）
		2：2-4TB					
混合存储节点	Combined Node（DataNode + YARN NodeManager）	磁盘类型 1：SSD（1-2 块，One_SSD 策略） 单盘大小 1：1-2TB 磁盘类型 2：SATA DISK（8-12 块，Hot/Warm 策略） 单盘大小 2：2-4TB	1. 核心：One_SSD（单 SSD + 多 DISK）、Warm（DISK+ARCHIVE，若含 ARCHIVE 盘） 1. 兼容：Hot（纯 DISK）	[SSD]file:///data/ssd0,[DISK]file:///data/disk0,[DISK]file:///data/disk1,...,[DISK]file:///data/disk11	平衡性能与成本，适配多类型数据需求	中小规模集群（节点数 < 10）的通用节点，同时支撑热数据计算与低 IO 高性能需求	1. 硬件：性价比优先，SSD 选 SATA 接口（成本低于 NVMe），DISK 选常规容量 1. 规模：仅用于中小集群（大规模集群建议拆分专用节点，便于管理）

- 目录规划

数据目录	数据类别	匹配策略	策略逻辑（以 3 副本为例）	目的
/user/project/realtime	热数据	Hot	3 副本全存 DISK	支持高频计算与读写
/user/project/report	温数据	Warm	1 副本存 DISK，2 副本存 ARCHIVE	平衡访问速度与成本
/user/project/archive	冷数据	Cold	3 副本全存 ARCHIVE	低成本长期归档
/user/project/high_io	高性能数据	All_SSD	3 副本全存 SSD	超低延迟读写
/user/project/tmp	临时数据	Lazy_Persist	1 副本先存 RAM_DISK，延迟存 DISK	加速临时数据写入
/user/project/external	外部数据	Provided	1 副本存外部存储，2 副本存 DISK	避免重复存储

（1）环境配置（确保 HDFS 支持异构存储）

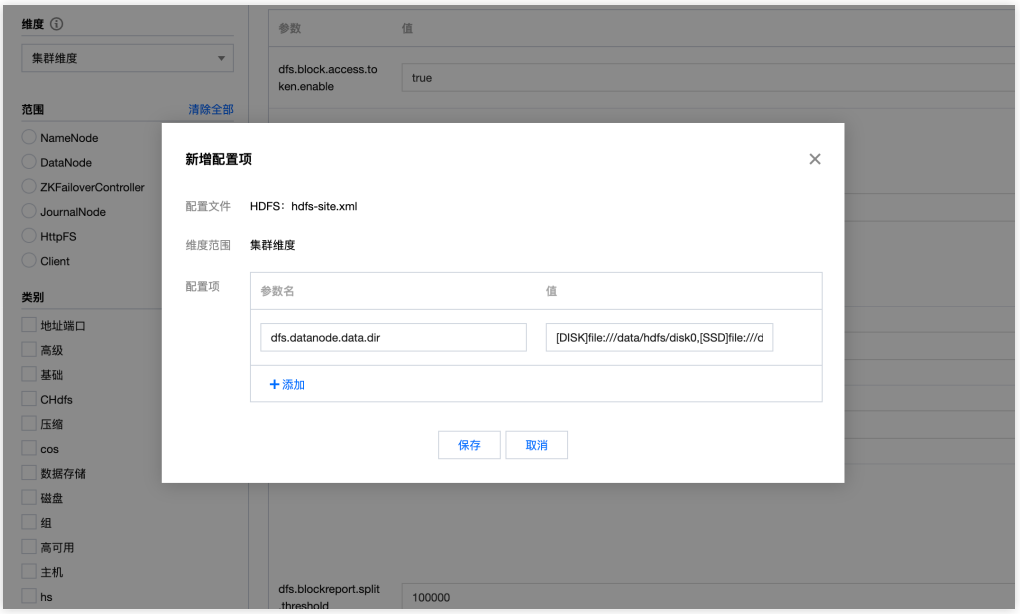
- 单击 HDFS 组件-配置管理，即可进配置管理页面。
- 选择 hdfs-site.xml 文件**。**
- 查找配置项 extra-dfs.data.dir，来进行配置。

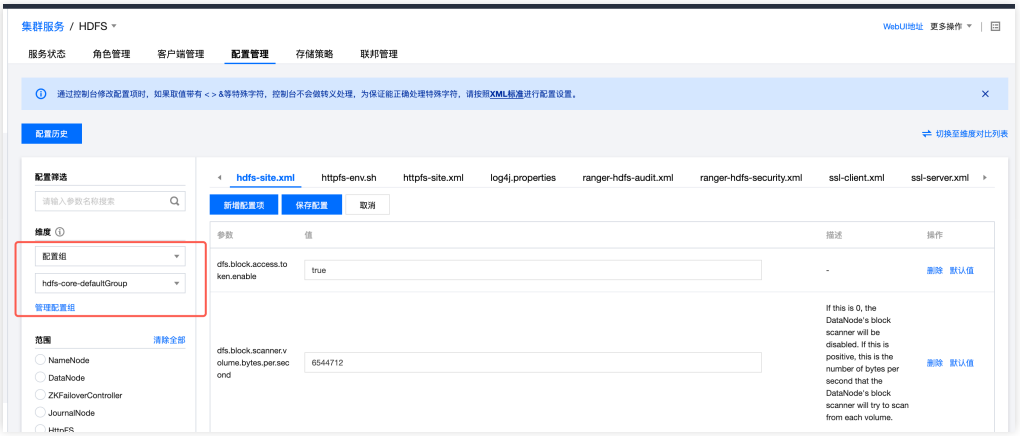
> 说明：
>
> 系统的默认值：tbds-default，表示系统自动识别逻辑。系统会自动识别节点挂载的数据盘。比如节点挂载了/data1，/data2数据目录。则系统会自动将该节点的datanode数据目录配置为/data1/qcloud/data/hdfs,/data2/qcloud/data/hdfs。默认的介质类型为DISK

- 对于复杂场景，用户需要自定义，例如：

```
<property>
  <name>extra-dfs.data.dir</name>
  <value>
   [DISK]file:///data/hdfs/disk0,  <!-- 普通硬盘 -->
   [SSD]file:///data/hdfs/ssd0,    <!-- SSD硬盘 -->
   [ARCHIVE]file:///data/hdfs/archive0,  <!-- 高密度归档盘（如SATA大容量盘） -->
   [RAM_DISK]file:///data/hdfs/ram0,    <!-- 内存盘（需提前挂载tmpfs） -->
  </value>
</property>
```

- 配置方式有两种，基于集群维度、基于配置组维度





（2）策略创建（基于 TBDS-HDFS 存储策略管理页面）

- 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
- 在集群服务模块点击 HDFS 组件，进入组件详情页。
- 单击存储策略，即可进存储策略列表页。
- 点击新增策略，可对 HDFS 存储路径配置存储配额和存储策略。

参数说明：

信息	详情
策略名称	存储策略名称，长度限制为6-36个字符，只允许包含中文、字母、数字、“-”、“_”
HDFS 存储路径	选定 HDFS 路径进行策略配置
策略类型	支持配额策略和存储策略配置
策略配置	**配额策略：**配置存储容量和文件数，限制选定路径下HDFS存储容量和文件【这里配额策略的应用不在本次冷热温使用范围，不做详细介绍】 存储策略： 对 HOT，COLD，WARM，ALL_SSD，ONE_SSD和LAZY_PERSIST 策略，将直接设定选中目录的存储策略并按照策略进行存储块的移动； 智能策略将根据最近访问时间或最后修改时间对 HDFS 存储策略进行调度，针对选中的目录，周期内没有访问记录子目录将被降级存储策略等级，启用智能策略时热数据存储策略为 Hot，温数据存储策略为 Warm，冷数据存储策略为 Cold。 EC纠删码策略：存储策略配置将为选中的数据目录设置HDFS EC纠删码策略。

- 配置完成后可在列表中查看路径的已用存储容量和已用文件数量，同时支持对策略进行修改和删除。

（3）验证策略是否生效

通过hdfs storagepolicies -getStoragePolicy命令，确认目录的策略已正确设置：

```
# 查看冷数据目录的策略
hdfs storagepolicies -getStoragePolicy -path /user/project/archive
# 预期输出：Storage policy of /user/project/archive is COLD
```

通过hdfs fsck命令查看块的存储类型，确认是否符合策略：

```
# 查看冷数据目录下某文件的块存储信息
hdfs fsck /user/project/archive/202401.log -files -blocks -locations
# 预期输出：块的存储位置（Storage Type: ARCHIVE），即所有副本都在ARCHIVE介质上
```

（4）说明

TBDS 内置的 HDFS 存储策略有如下约定，需要注意。

> 注意：
>
> 1. 当前支持的EC纠删码策略的使用限制如下：
>
> - RS-10-4-1024K需要至少14个 HDFS DataNode以开启
> - RS-3-2-1024K需要至少5个 HDFS DataNode以开启
> - RS-6-3-1024K需要至少9个 HDFS DataNode以开启
> - RS-LEGACY-6-3-1024K需要至少9个 HDFS DataNode以开启
> - XOR-2-1-1024K需要至少3个 HDFS DataNode以开启
>
> 1. 存储策略设置的HOT，COLD，WARM，ALL_SSD，ONE_SSD和LAZY_PERSIST策略将即时触发目标路径下的数据块移动操作。
> 2. 智能策略的数据目录策略调度将在每日8:00执行。
> 3. 设置目标路径到EC纠删码策略时，目标路径下的存量数据不会被转化为EC存储。

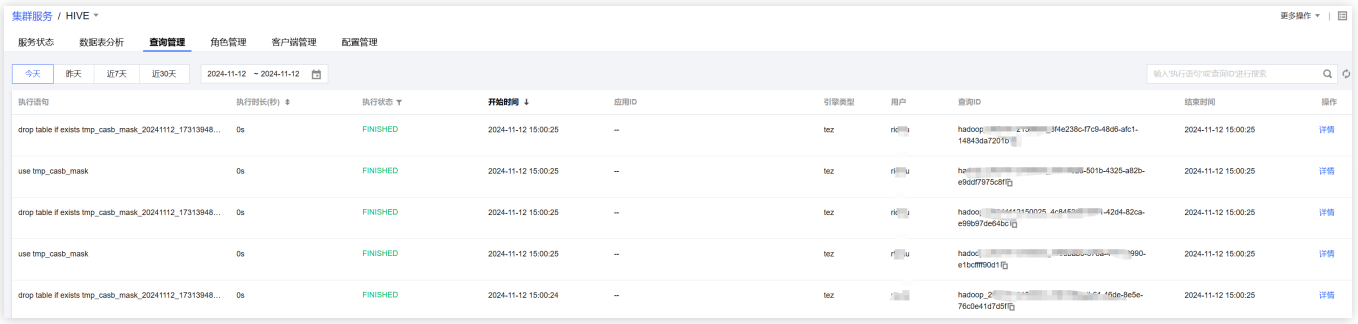
Hive 作业查询

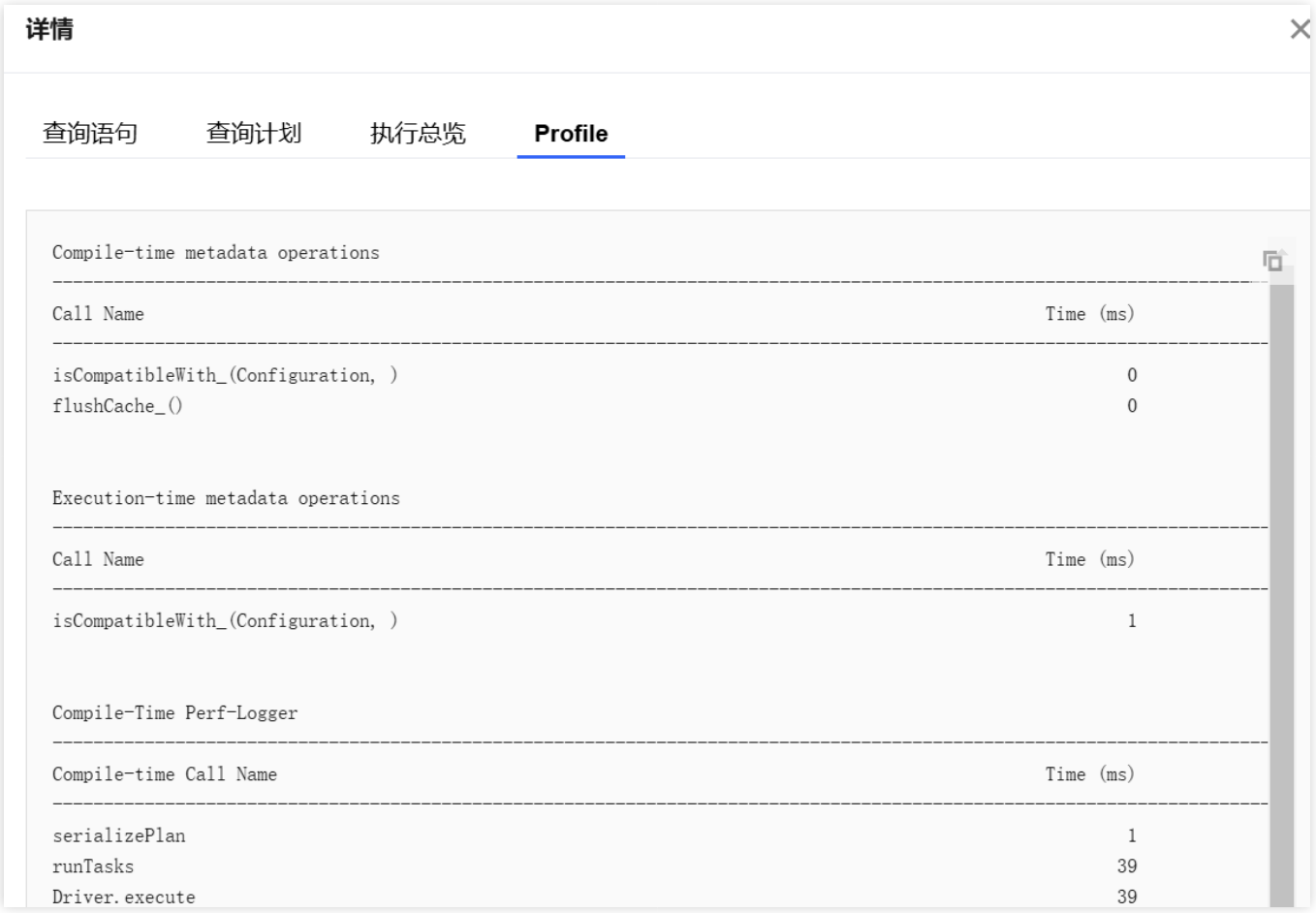
功能介绍

提交后的 Hive 查询可通过查询管理快速查看查询的运行状况。查询列表展示了相关查询的执信息、执状态等信息，同时可帮助快速关联查询得到执作业。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中单击集群服务目录，然后选择 Hive 服务名称或 Hive 服务右上角操作 > 查询管理，即可进相关作业查询、查询语句、查询计划、执总览、Profile 等信息。





3. 单击应用 ID 可跳转至 YARN 应用管理并选中 Hive 查询所关联的 YARN 作业集。

HBase 数据表分析

功能介绍

数据表分析提供 HBase 表级、表内 Regions、RegionServers 的读写请求量和存储情况等维度信息；同时提供 Region 分析，结合实际场景持对所属表或所属 RegionServer 分析读取 QPS、写入 QPS 信息及历史变化趋势。

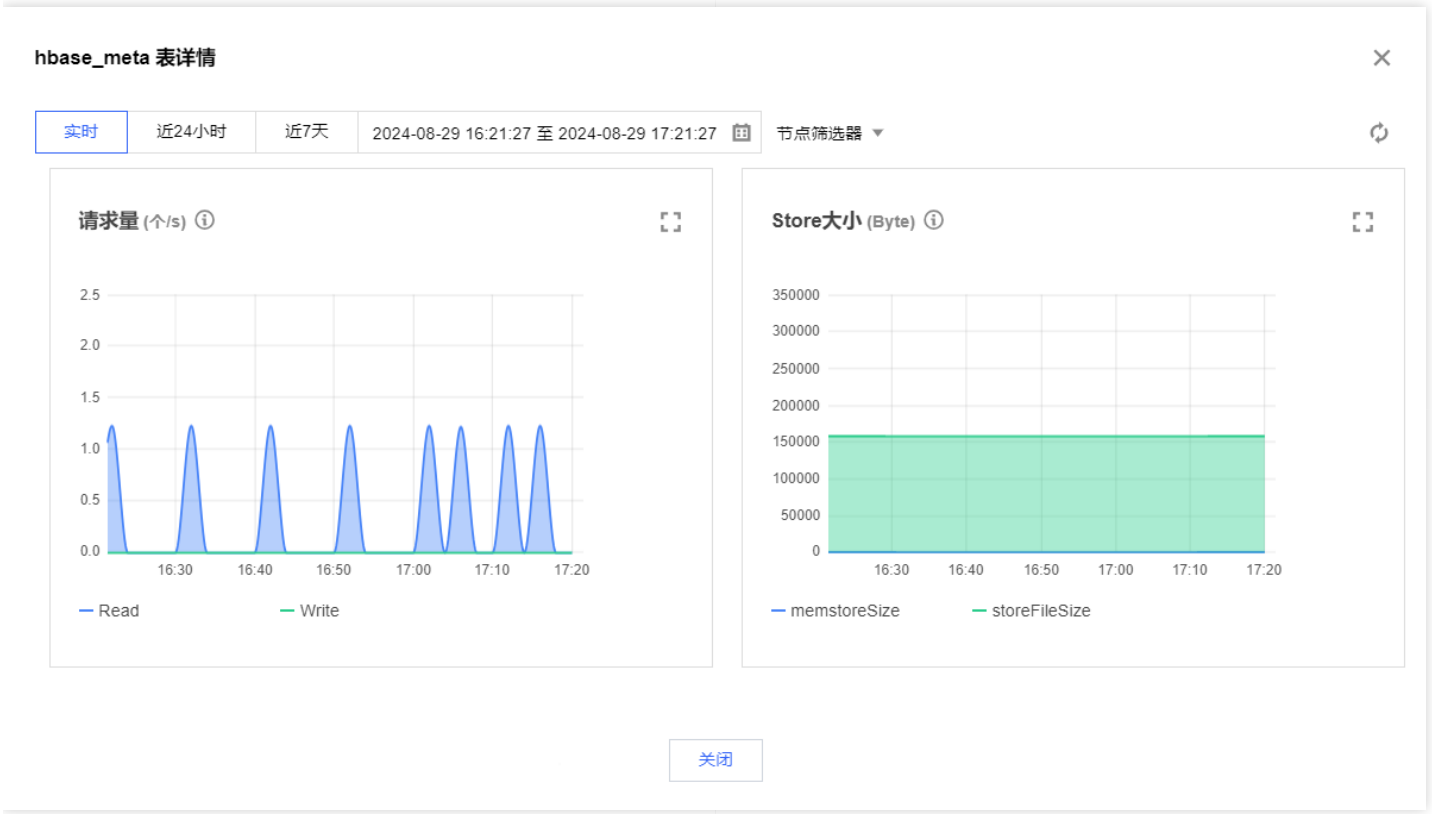
操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中单击集群服务，然后选择 HBase 组件右上角操作 >数据表分析，即可进相关 HBase 数据表负载查询。

数据表列表

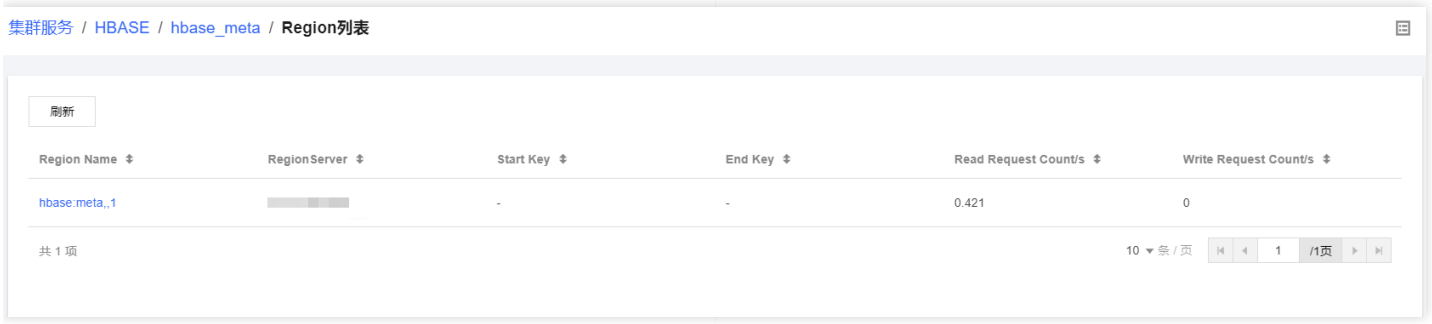
HBase 数据表列表可查看表级请求 QPS、写入 QPS、MemStore 存储量、StoreFile 大小等信息，通过列 title 的排序按钮可定位集群 Top 数据表。

单击对应表名，即可弹出表详情。详情页可按整个表、节点维度展示所选择表的请求量（包括读和写）、store 大小（包括 memstore 和 StoreFile）两个指标数据，选择右上角的节点筛选器可切换节点查看。



Region 操作

单击 Regions 操作，即可查看表所包含的各个 Region 的读写请求量，定位表内 Region 热点情况。

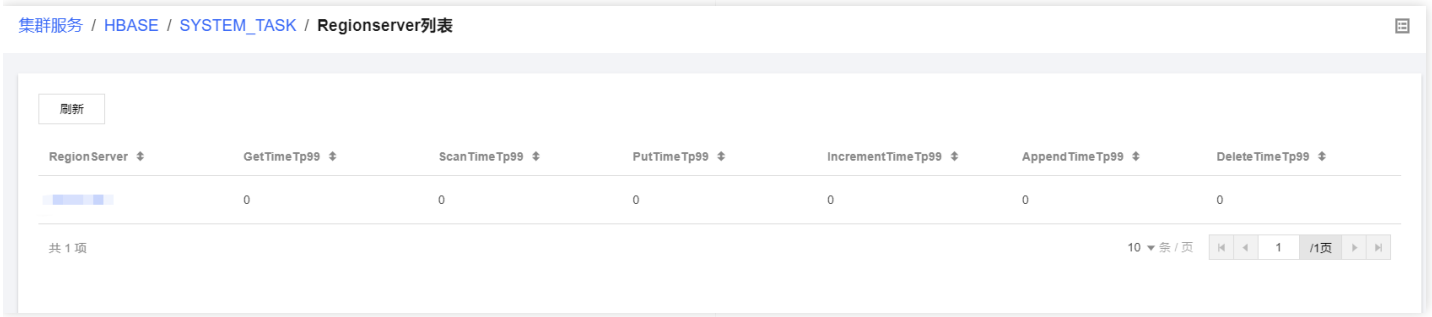


Region 详情

单击对应 Region 名，即可弹出 Region 详情，查看指标趋势。详情页可按不同时间粒度展示所选择表的请求量（包括读和写）指标数据，选择右上角的时间粒度可切换粒度查看。

RegionServers 操作

单击 RegionServers 操作，即可查看表所分布的各个 RegionServer 的请求延迟。



HBase Region 分析功能

Region 分析可检索所属表或筛选所属 RegionServer，通过平均请求 QPS、平均读写 QPS 信息定位集群热点请求分布。

Region分析
1小时
6小时
12小时
2024-08-29 16:26:01 ~ 2024-08-29 17:26:01
请输入表名
Region
所属表
所属RegionServer
平均读取QPS
平均写入QPS
hbase:meta,,1
hbase_meta
0.44
0
f1f8a7d3d137aed605f33cc158ed55e6
SYSTEM_CATALOG
0.02
0
65563f033ef389f4baa6b292da85f70d
SYSTEM_TASK
0.02
0
8fb9cdb7b1bf928bded8cecc3b97a527
SYSTEM_CHILD_LINK
0
0

指标趋势对比
1小时
6小时
12小时
2024-08-29 16:37:06 ~ 2024-08-29 17:37:06
Region读请求 (个/s)
1
0.5
0
16:35
16:40
16:45
16:50
16:55
17:00
17:05
17:10
17:15
17:20
17:25
17:30
17:35
region_id=026dc4c906630d157667312139558044
region_id=031a3ede102d62b513323e4d4a6252a
1/8

集群监控
集群概览

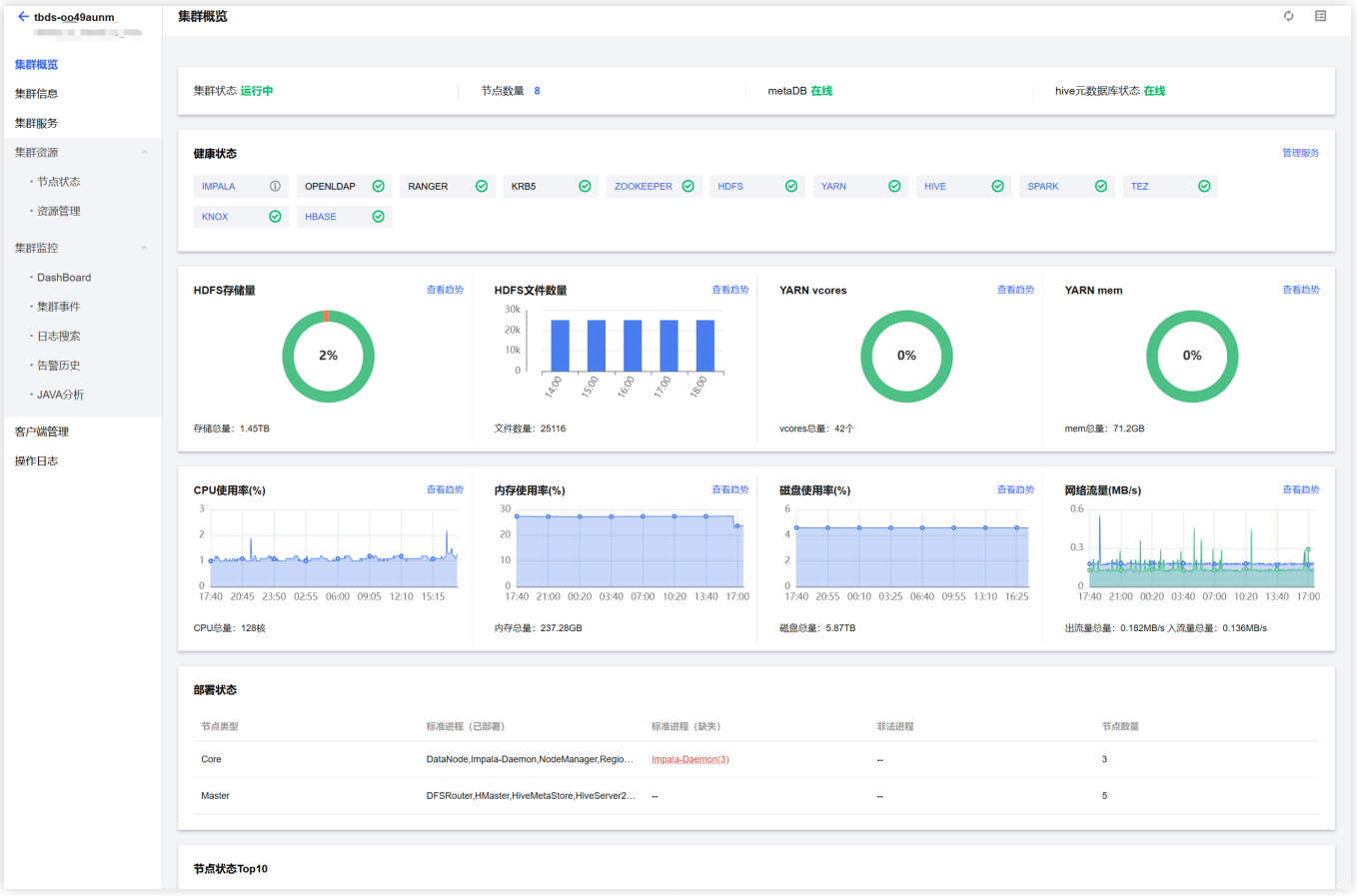
功能介绍

集群概览展示集群运行状态的总体视图，可以获取集群运状态、核心服务指标、核节点指标以及节点负载 TOP10 情况。

操作步骤

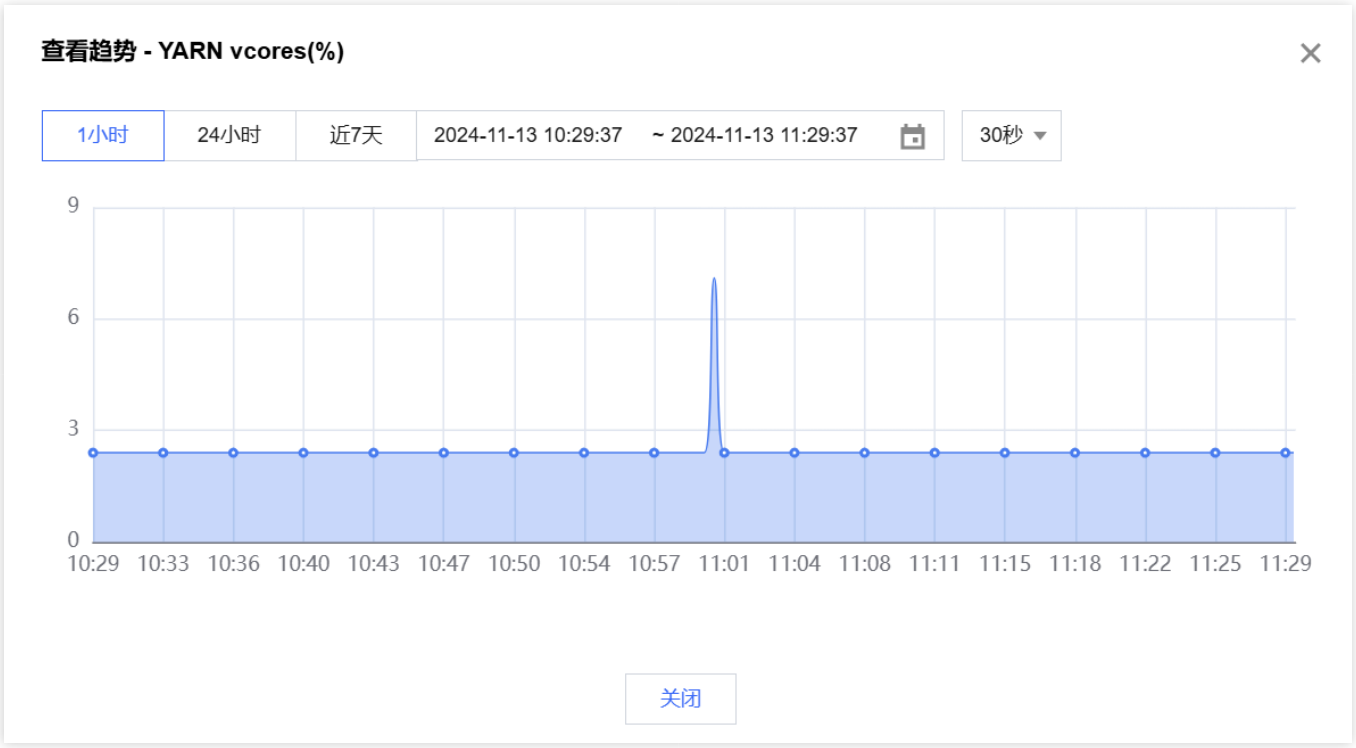
1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群概览，可直接查看当前集群总体情况。在集群概览页中，提供了四部分集群维度的监控视图，分别为集群总体情况、集群重要指标、集群部署状态、集群节点负载 TOP10。

- 集群总体情况：可直接查看当前集群状态是否异常、节点数量、元数据库是否在线以及组件健康状态。



- 集群重要指标：可直观查看当前集群 HDFS、YARN、CPU 总使用率、内存总使用率、磁盘总使用率以及网络总流量

情况，且每个使用情况可单击右上角查看趋势，选择对应时间段进查看。

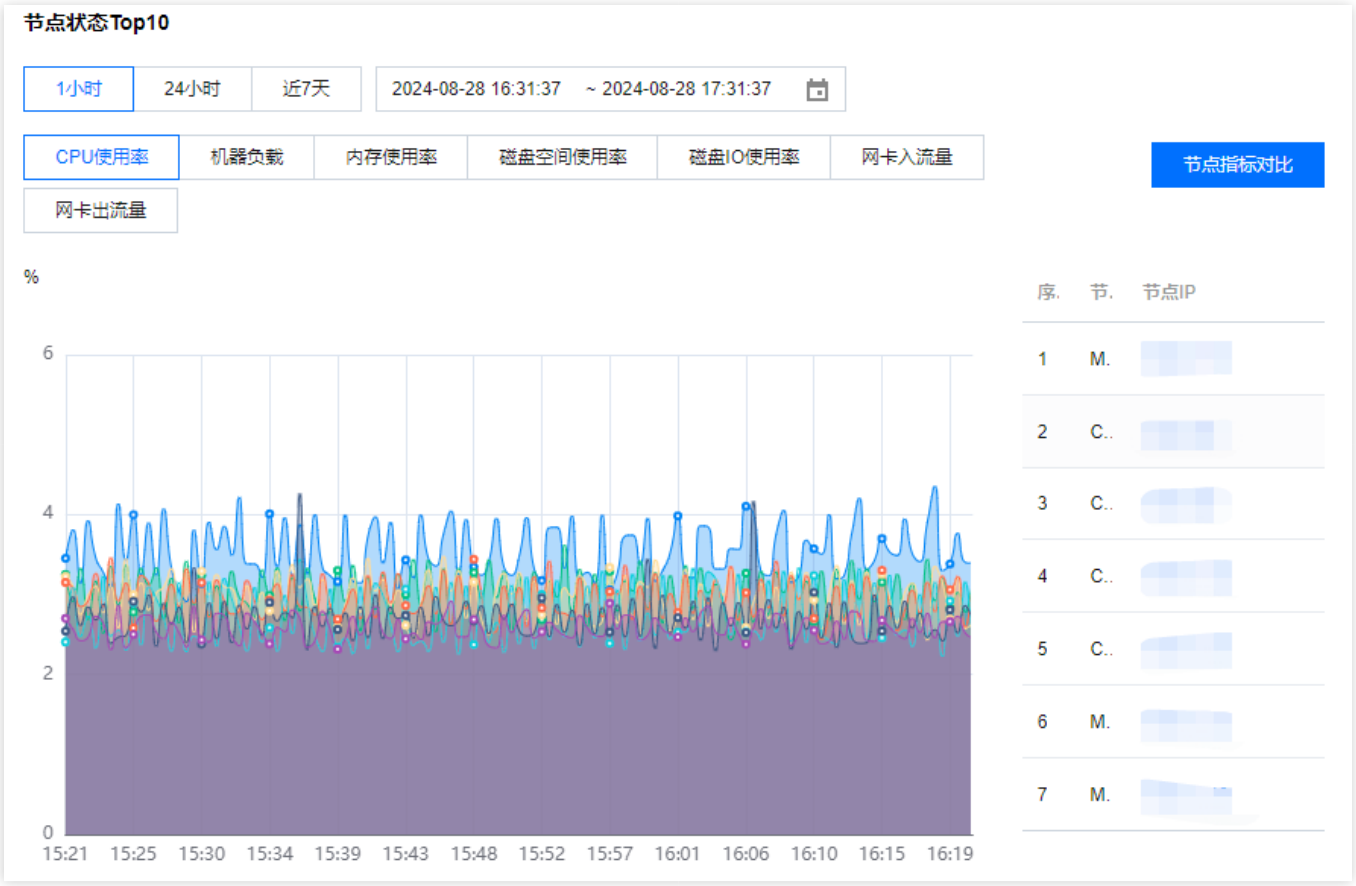


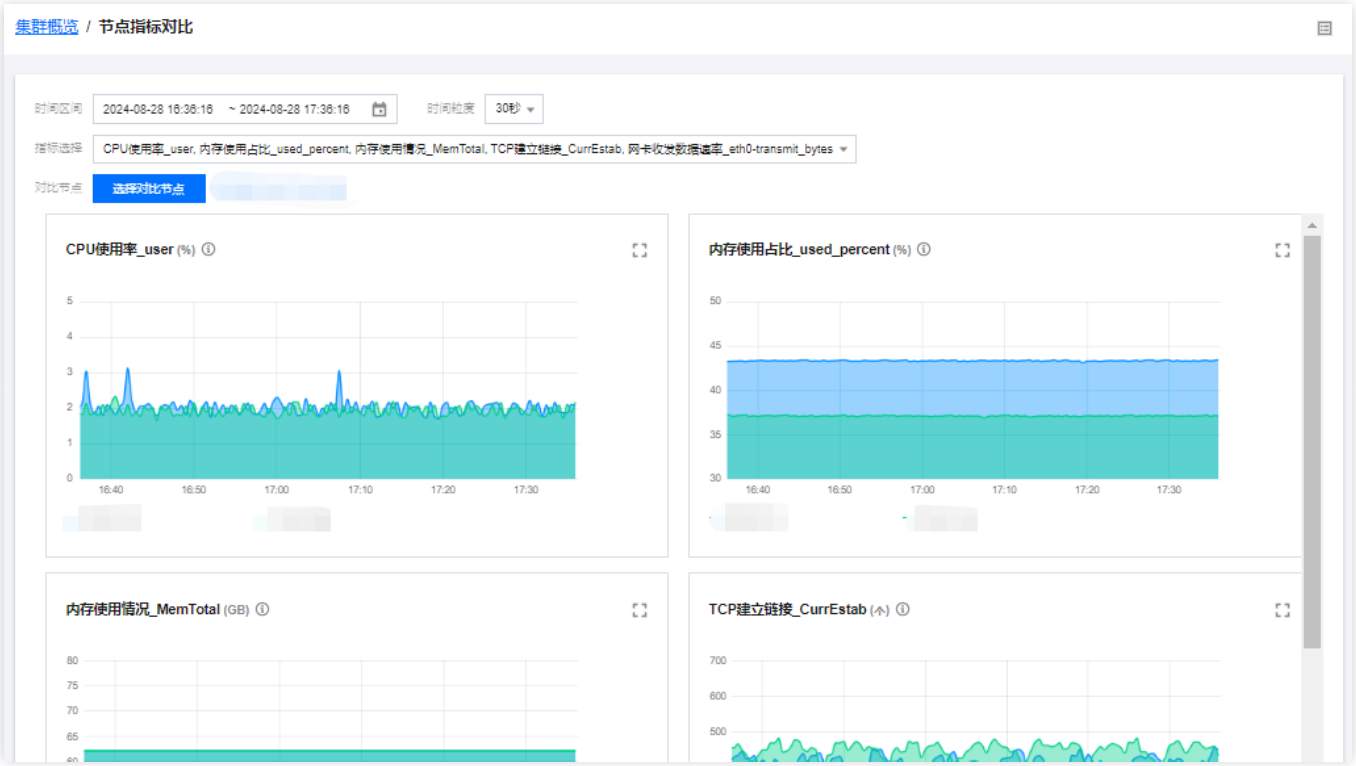
- 集群部署状态：可直接查看当前集群部署节点类型中部署进程是否异常、缺失、非法和节点数量，以便于正确调整。

部署状态

节点类型	标准进程（已部署）	标准进程（缺失）	非法进程	节点数量
Core	DataNode,Impala-Daemon,NodeManager,Regio…	Impala-Daemon(3)	--	3
Master	DFSRouter,HMaster,HiveMetaStore,HiveServer2…	--	--	5

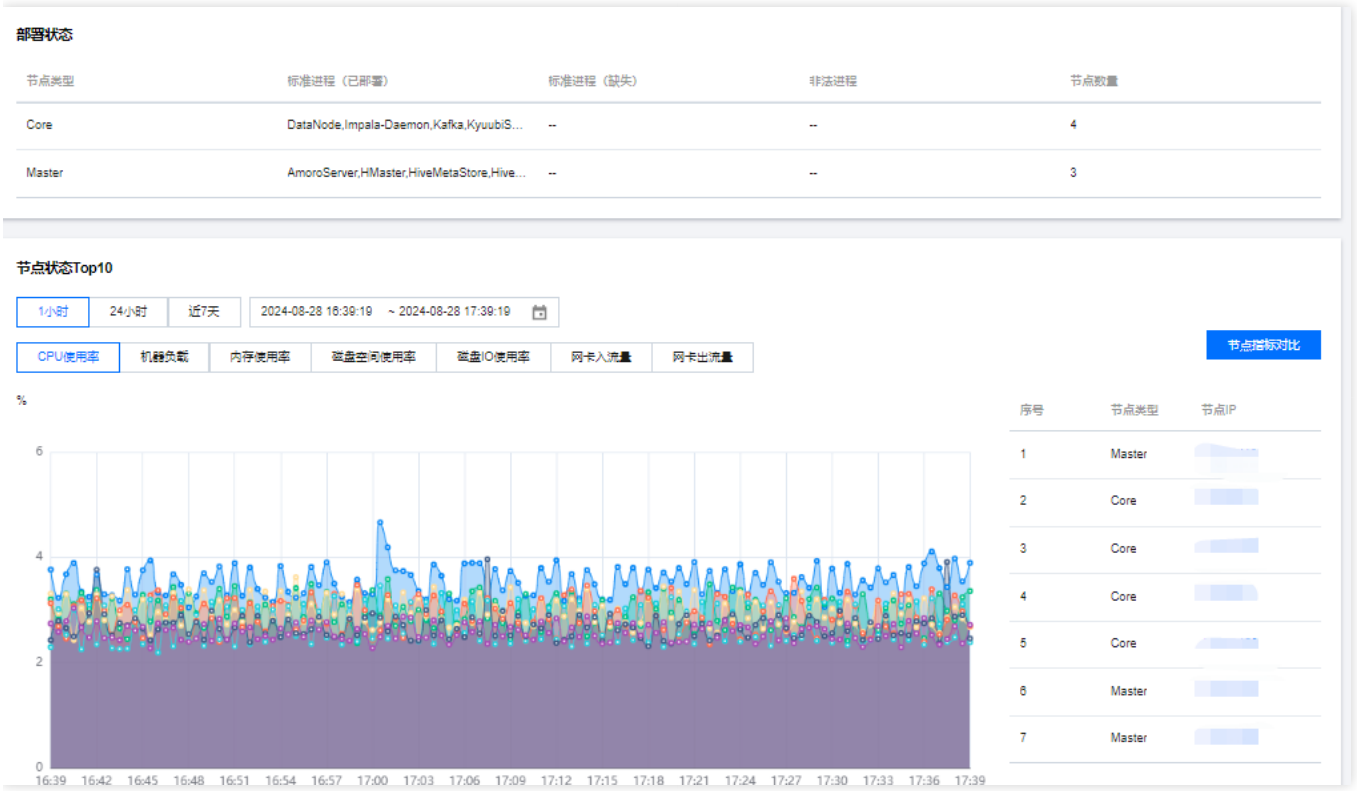
- 集群节点负载 TOP10：可查看核心指标下当前集群中节点负载趋势变化情况，可选择多个节点对同指标的负载趋势进比较。





以下对核心指标进行说明：

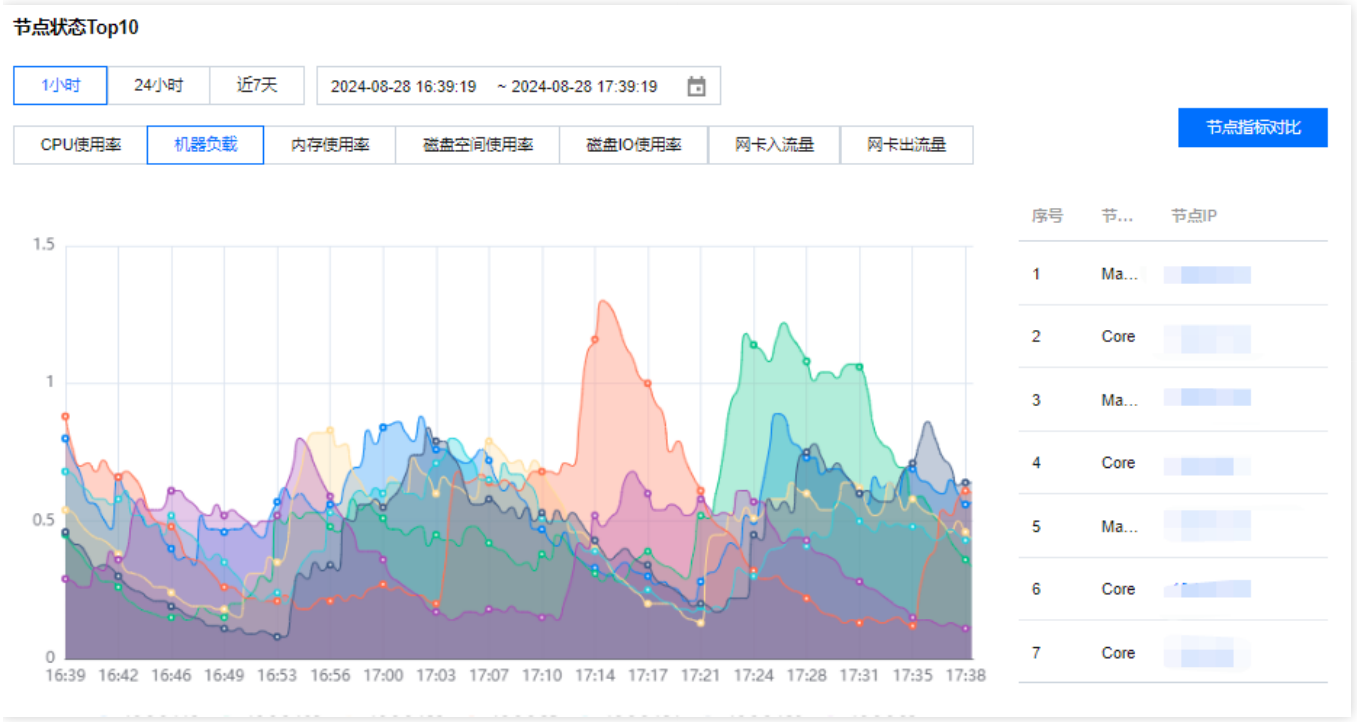
cpu 使用率



当集群各节点处理的读写任务超出节点 CPU 的负载能力时，该指标就会过高，CPU 使用率过高会导致集群节点处理能力下降，甚至宕机。您可以从以下几点解决平均 CPU 使用率过高的问题：

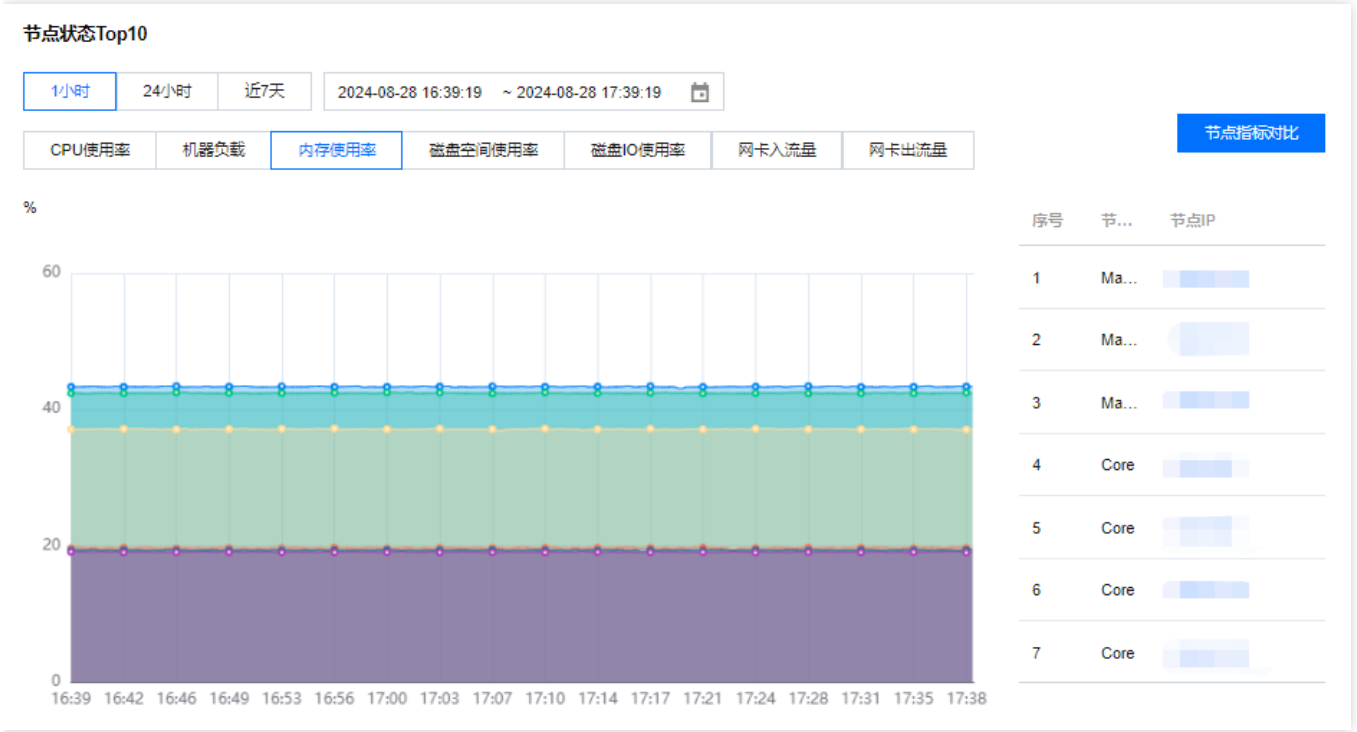
- 观察该指标是持续性较高，还是临时飙升。若是临时飙升，确定是否有临时性复杂任务正在执行。
- 若该指标持续较高，分析业务对集群的读写操作是否可以优化，降低读写频率，减小数据量，从而减轻节点负载。
- 对于节点配置无法满足业务吞吐量的情况，建议对集群节点进行纵向扩容，提高单节点的负载能力。

机器负载



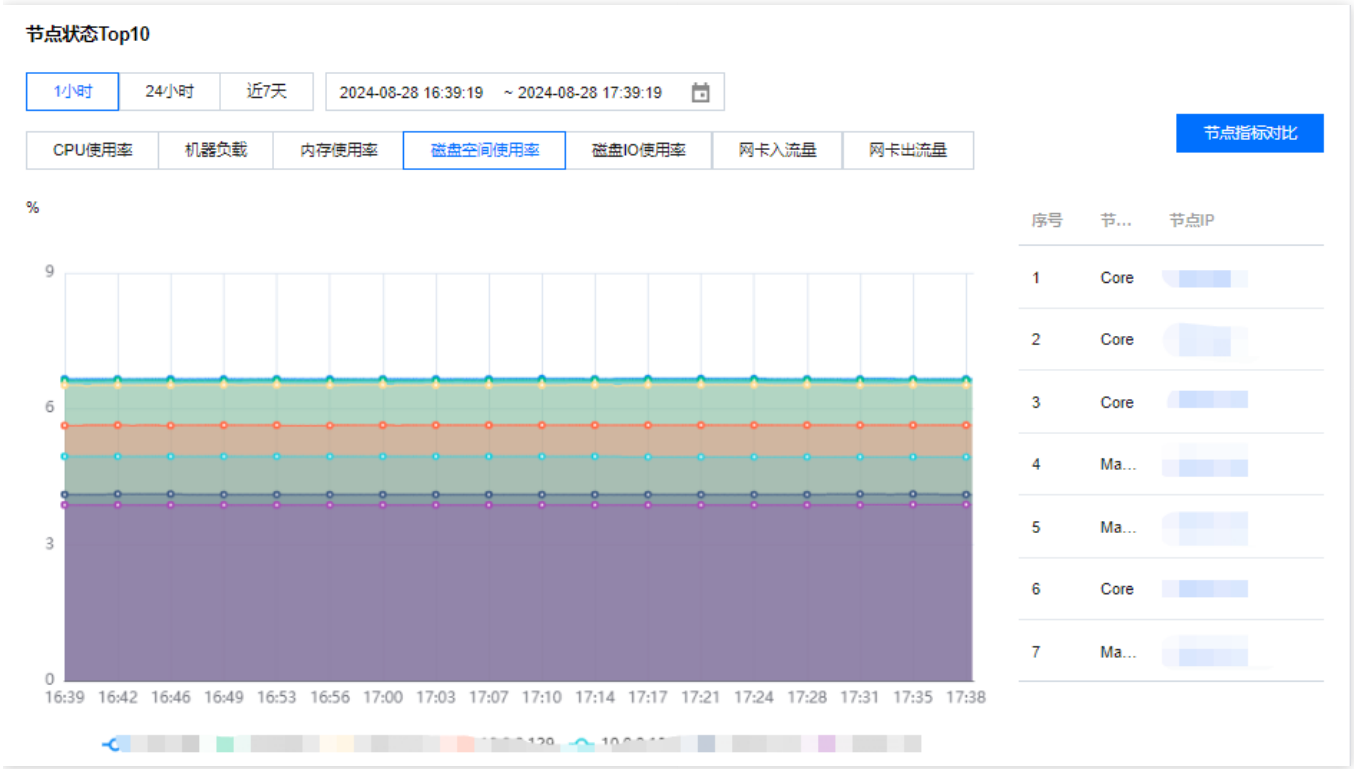
机器负载过高时，建议降低集群负载或调大集群节点规格。

内存使用率



该值过高会导致集群节点 GC 频繁，甚至有出现 OOM。导致该值过高的原因，一般是节点上 ES 处理任务超出节点 JVM 的负载能力。您需要注意观察集群正在执行的任务，或调整集群的配置。

磁盘空间使用率



磁盘使用率过高会导致数据无法正常写入，解决方法：及时清理无用的索引。对集群进行扩容，增加单节点的磁盘容量或增加节点个数。

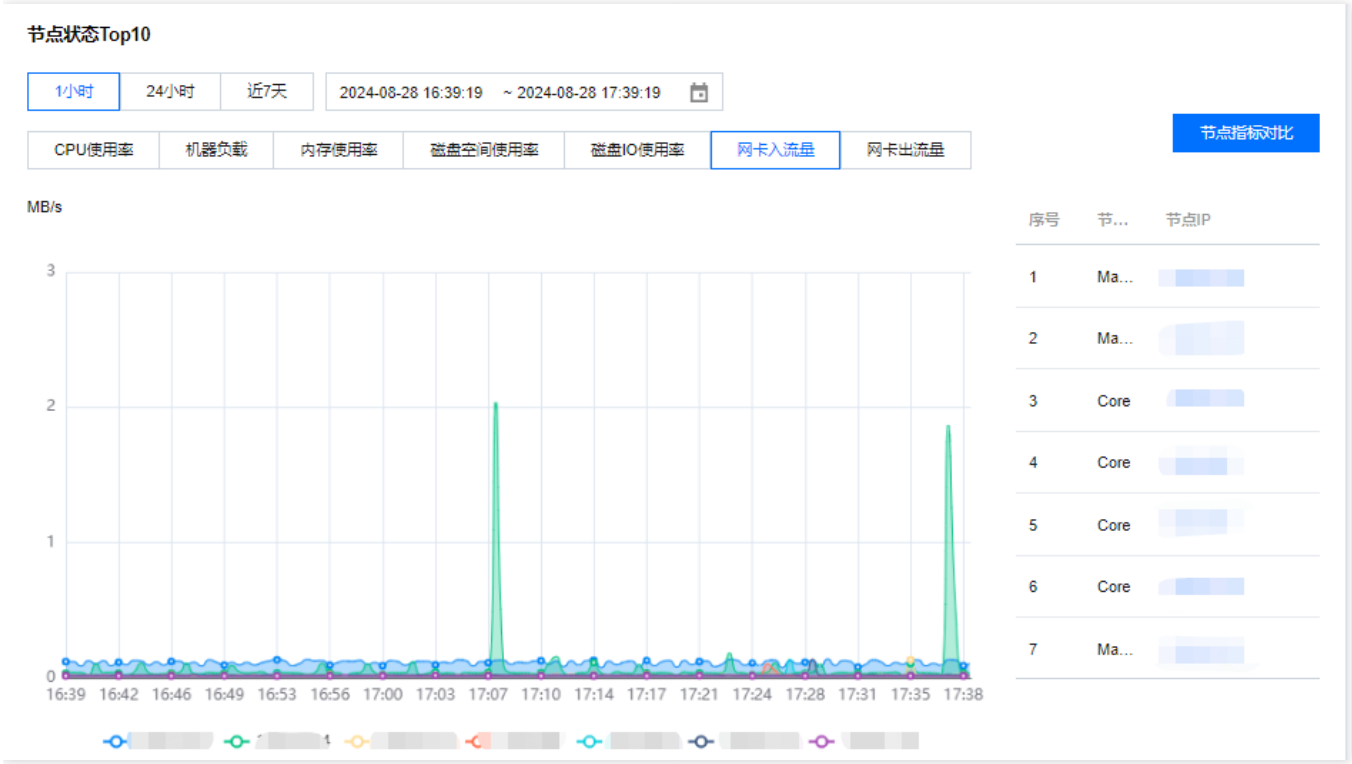
磁盘 IO 使用率

写入 QPS 过大，CPU、内存、磁盘使用率过高时，可能会造成集群写入拒绝率增加。一般地，是集群当前配置无法满足业

务写操作需求。对于节点配置过低的场景，可以通过提高节点规格或降低写入操作次数来解决。对于磁盘使用率过高的情况，可以通过扩容集群磁盘或删除无用数据来解决。

网卡入/出流量

网卡入/出流量是对节点的网卡在特定时间段内接收和发送的数据量。监控这两个指标对管控网络流量和网络性能具有重要作用，可以提供关于网络带宽使用情况、故障排除、网络优化和安全监控方面的重要信息，帮助进行网络管控和保障网络性能、稳定性和安全性。



网卡入流量是指通过网络接口接收到的数据量，例如从外部网络或其他设备发送给本机的数据包。这可以是下载文件、接收电子邮件、接收网络请求等。

网卡出流量是指通过网络接口发送出去的数据量，例如将数据包发送到外部网络或其他设备。这可以是上传文件、发送电子邮件、响应网络请求等。出流量也通常以比特或字节为单位进行计量。

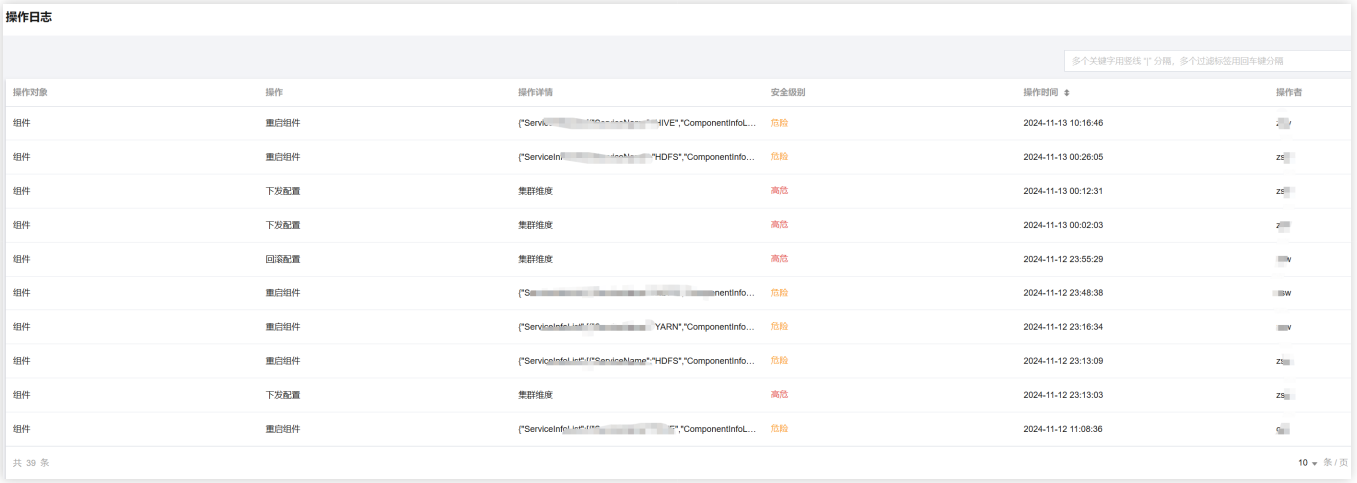
操作日志

功能介绍

操作日志统一记录用户在 TBDS Manager 管理平台对集群执行的操作，方便用户查看。例如创建集群、扩缩容集群等，都会记录在操作日志中。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 单击左侧菜单栏操作日志页。支持展示操作，操作对象，安全级别等操作审计信息。



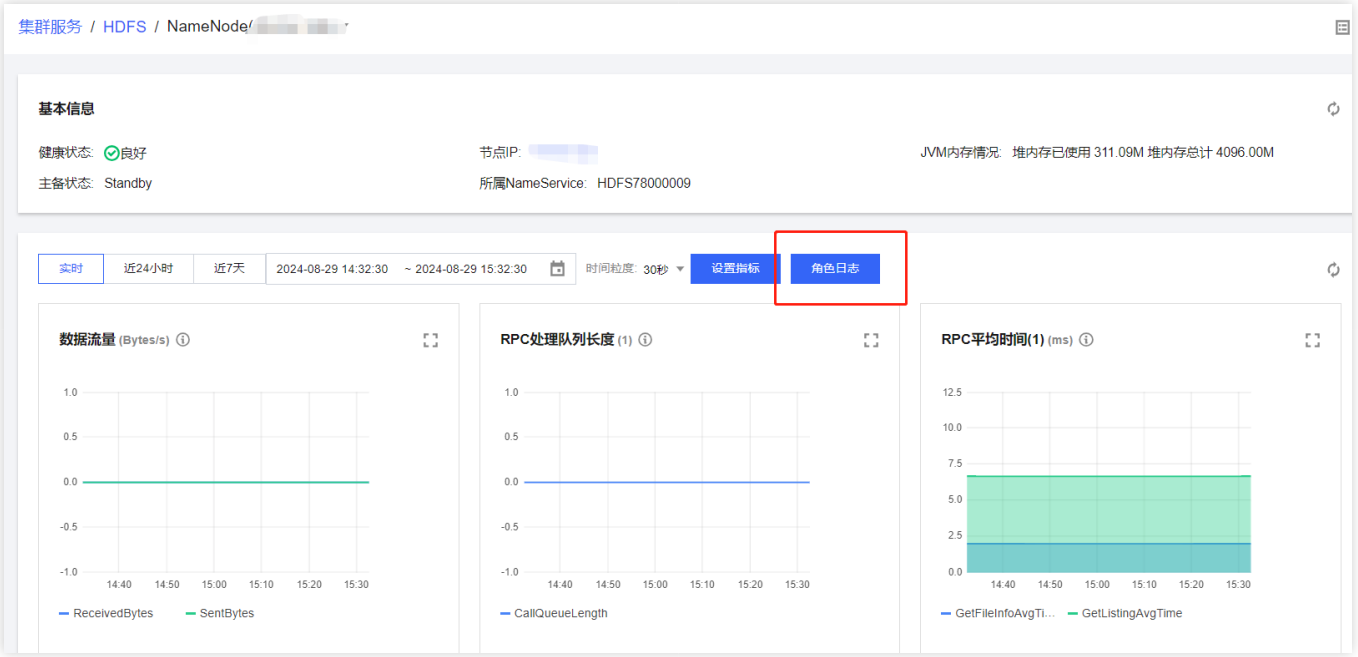
日志搜索

功能介绍

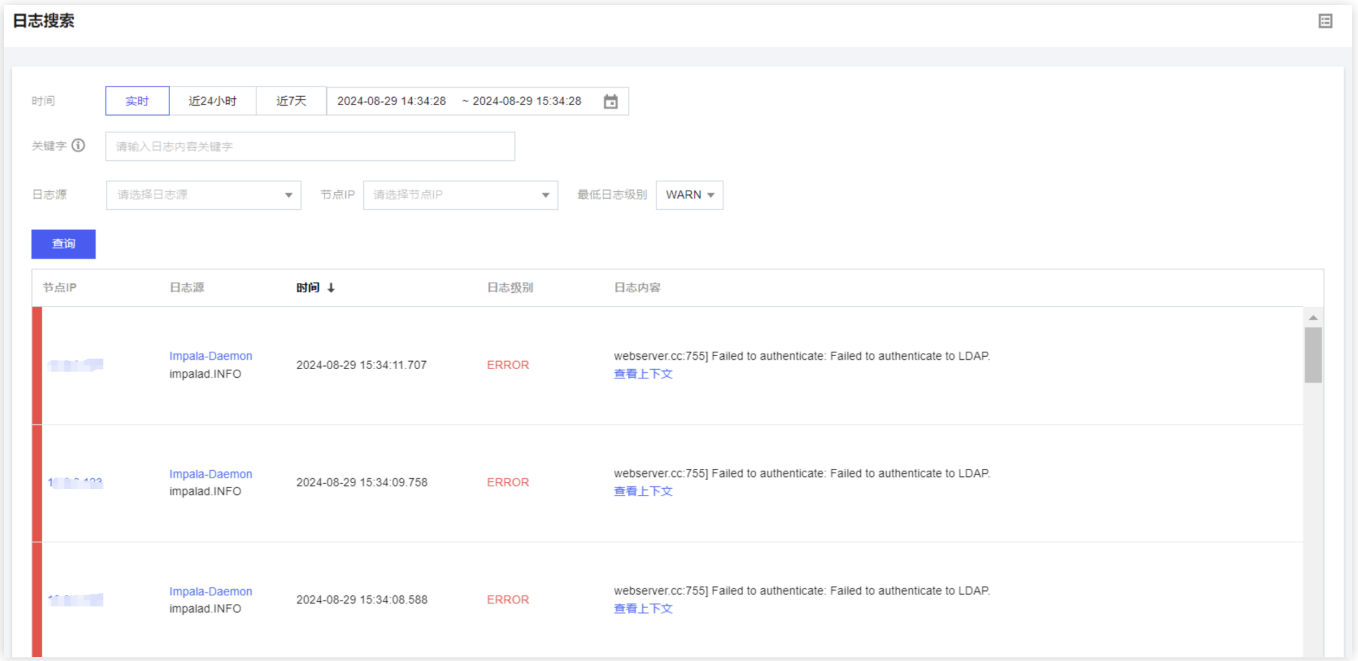
日志搜索功能提供组件的运行日志采集和搜索功能，支持当前集群核心服务日志和节点系统日志进行关键词搜索，可以在不登录节点的情况下快速查看服务关键日志。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 在集群详情页中选择集群监控 > 日志搜索可根据当前集群、日志文件、节点 IP 和时间范围条件过滤，查看日志内容。
   或者在集群详情页中选择集群服务>组件卡片>角色管理，查看角色列表，选择某节点IP单击跳转，进入节点监控指标展示页，单击角色日志，可跳转日志搜索页面。



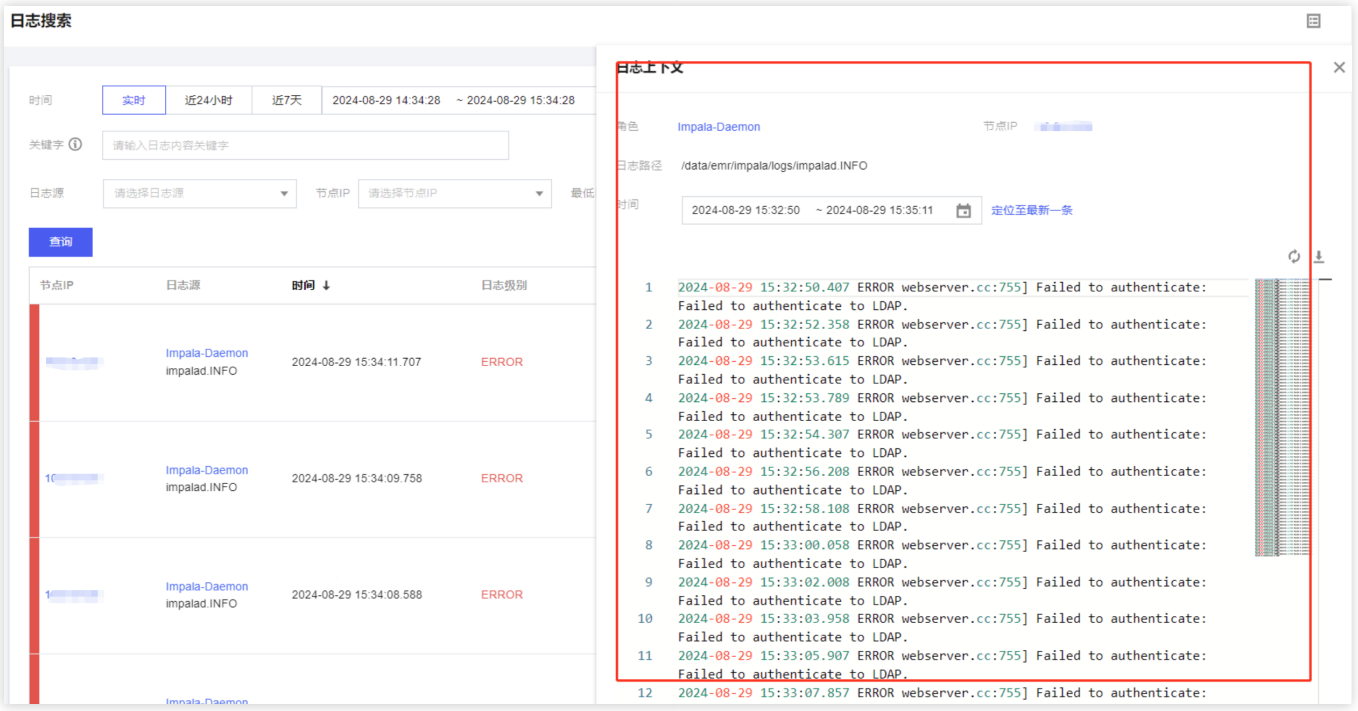
单击节点 IP，可跳转到对应节点状态页面。单击日志源，可跳转到对应节点监控指标展示页。



关键字说明：

- 支持关键字全文检索。
- 支持特殊字符 - . * > < = ! () {} / 检索。
- 支持短语检索，例如：address=/ip:port。

3. 在排查问题的时候，经常需要关注关键词的上下文日志，在日志搜索页单击查看上下文，进入日志上下文页面。



日志搜索支持的服务类型：

日志搜索支持15天内的历史数据搜索。

组件	角色	日志	说明
HDFS	NameNode	/data/emr/hdfs/logs/hadoop-hadoopnamenode.log	NameNode 的运行时日志
	ZKFC	/data/emr/hdfs/logs/hadoop-hadoopzkfc.log	ZKFC 的运行时日志
	DataNode	/data/emr/hdfs/logs/hadoop-hadoopdatanode.log	DataNode 的运行时日志
	JournalNode	/data/emr/hdfs/logs/hadoop-hadoopjournalnode.log	JournalNode 的运行时日志
	DFSRouter	/data/emr/hdfs/logs/hadoop-hadoopdfsrouter.log	DFSRouter 的运行时日志
YARN	ResourceManager	/data/emr/YARN/logs/YARN-hadoopresourcemanager.log	ResourceManager的运行时日志
	NodeManager	/data/emr/YARN/logs/YARN-hadoopnodemanager.log	NodeManager 的运行时日志
	JobHistoryServer	/data/emr/YARN/logs/mapred-hadoophistoryserver.log	JobHistoryServer 的运行时日志
HBase	HMaster	/data/emr/HBase/logs/HBase-hadoopmaster. log	HMaster 的运行时日志
	ThriftServer	/data/emr/HBase/logs/HBase-hadoopthrift.log	ThriftServer的运行时日志
	RegionServer	/data/emr/HBase/logs/HBase-hadoopregionserver.log	RegionServer 的运行时日志
ClickHouse	ClickHouse-server	/data/clickhouse/clickhouseserver/logs/clickhouse-server.log	ClickHouse-server的运行时日志
ZooKeeper	ZooKeeper	/data/emr/ZooKeeper/logs/ZooKeeperroot-server.log	ZooKeeper 的运行时日志
Hive	HiveServer2	/data/emr/hive/logs/hadoop-hive	HiveServer2 的运行时日志
Ranger	EmbeddedServer	/data/emr/ranger/logs/ranger-admin.log	EmbeddedServer 的运行时日志
Impala	catalogd	/data/emr/impala/logs/catalogd.INFO	catalogd 的运行时日志
	statestored	/data/emr/impala/logs/statestored.INFO	statestored 的运行时日志

组件	角色	日志	说明
	impalad	/data/emr/impala/logs/impalad.INFO	impalad 的运行时日志
Spark	HistoryServer	/data/emr/spark/logs/spark-hadoop.log	HistoryServer 的运行时日志
Zeppelin	ZeppelinServer	/data/emr/zeppelin/logs/zeppelinhadoop.log	ZeppelinServer 的运行时日志
Knox	Gateway	/data/emr/knox/logs/gateway.log	Gateway 的运行时日志
Kafka	Kafka	/usr/local/service/kafka/logs/server.log	Kafka 的运行时日志

服务支持最低日志级别：

服务	默认采集的最低日志级别
Impala	INFO
其他服务	WARN

最低日志级别查询规则：

最低日志级别	可查询的日志级别
INFO	INFO、WARN、ERROR、FATAL
WARN	WARN、ERROR、FATAL
ERROR	ERROR、FATAL
FATAL	FATAL

集群事件

功能介绍

集群事件中包含事件列表和事件策略。

- 事件列表：记录集群发生的关键变化事件或异常事件。
- 事件策略：支持根据业务情况自定义事件监控触发策略，已开启监控的事件可设置为集群巡检项。

查看事件列表

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 在集群详情页中选择集群监控 > 集群事件 > 事件列表，可直接查看当前集群所有操作事件。严重程度说明如下：
   - 致命：节点或服务的异常事件，人工干预处理，否则服务不可用，这类事件可能持续一段时间。
   - 严重：暂时未造成服务或节点不可用问题，属于预警类，如果一直不处理会产生致命事件。
   - 一般：记录集群发生的常规事件，一般无需特别处理。
3. 单击当日触发次数列值可查看事件的触发记录，同时可查看事件记录相关指标、日志。

设置事件策略

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 在集群详情页中选择集群监控 > 集群事件 > 事件策略，可以自定义设置事件监控触发策略。
3. 事件配置列表包含：事件名、事件发现策略、严重程度（致命/严重/一般）、开启监控，支持修改和保存。
4. 事件发现策略分两类：一类事件为系统固定策略事件，不支持用户修改；另一类事件会因客户业务标准的不同而变化，支持用户设置。
5. 事件策略可自定义是否开启事件监控，已开启监控的事件才支持在集群巡检的巡检项中选择。部分事件默认开启，部分事件默认开启且不可关闭。

集群事件清单

类别	事件名称	事件含义	建议&措施	默认值	严重程度	允许关闭	默认开启
节点	节点磁盘 IO 错误	磁盘 IO 发生错误	更换磁盘		致命	是	是
	元数据库 Ping 失败	CDB 心跳未定时上报	人工排查		-	-	-
HDFS	HDFS 文件总数持续高于阈值	集群文件总数量 >= m，持续时间 t 秒（300<=t<=2592000）	调大 namenode 内存	m=50,000,000, t=1800	严重	是	否
	HDFS 总 block 数量持续高于阈值	集群 Blocks 总数量 >= m，持续时间 t 秒（300<=t<=2592000）	调大 namenode 内存或调大 block size	m=50,000,000, t=1800	严重	是	否
	HDFS 标记为 Dead 状态的数据节点数量持续高于阈值	标记为 Dead 状态的数据节点数量 >= m，持续时间 t 秒（300<=t<=2592000）	人工排查	m=1,t=1800	一般	是	否
	HDFS 存储空间使用率持续高于阈值	HDFS 存储空间使用率 >= m，持续时间 t 秒（300<=t<=2592000）	清理 HDFS 中的文件或对集群扩容	m=85, t=1800	严重	是	是
	NameNode 发生主备切换	NameNode 发生主备切换	排查 NameNode 切换的原因	-	严重	是	是
	NameNode RPC 请求处理延迟持续高于阈值	RPC 请求处理延迟 >= m 毫秒，持续时间 t 秒（300<=t<=2592000）	人工排查	m=300, t=300	严重	是	否
	NameNode 当前连接数持续高于阈	NameNode 当前连接数 >= m，持续时间 t 秒	人工排查	m=2000, t=1800	一般	是	否

类别	事件名称	事件含义	建议&措施	默认值	严重程度	允许关闭	默认开启
	值	（300<=t<=2592000）					
	NameNode 发生 full GC	NameNode 发生 full GC	参数调优	-	严重	是	是
	NameNode JVM 内存使用率持续高于阈值	NameNode JVM 内存使用率持续 >= m，持续时间 t 秒（300<=t<=2592000）	调整 NameNode 堆内存大小	m=85, t=1800	严重	是	是
	DataNode RPC 请求处理延迟持续高于阈值	RPC 请求处理延迟 >= m 毫秒，持续时间 t 秒（300<=t<=2592000）	人工排查	m=300, t=300	一般	是	否
	DataNode 当前连接数持续高于阈值	DataNode 当前连接数 >= m，持续时间 t 秒（300<=t<=2592000）	人工排查	m=2000, t=1800	一般	是	否
	DataNode 发生 full GC	NameNode 发生 full GC	参数调优	-	一般	是	否
	DataNode JVM 内存使用率持续高于阈值	NameNode JVM 内存使用率持续 >= m，持续时间 t 秒（300<=t<=2592000）	调整 DataNode 堆内存大小	m=85, t=1800	一般	是	是
	HDFS 两个 NameNode 服务状态均为 Standby	两个 NameNode 角色同时处于 Standby 状态大于 t 秒	人工排查	t=90	严重	是	是
	HDFS MissingBlocks数量持续高于阈值	集群 MissingBlocks 数量 >=m，持续时间t秒 (300<=t<=604800)	建议排查 HDFS 出现数据块损坏，使用命令 hadoop fsck / 检查 HDFS 文件分布的情况	m=1,t=1800	严重	是	是
	HDFS NameNode 进入安全模式	NameNode 进入安全模式(持续300s)	建议排查 HDFS 出现数据块损坏，使用命令 hadoop fsck / 检查 HDFS 文件分布的情况	-	严重	是	是
	HDFS NameNode 长时间未做 Checkpoint	HDFS NameNode 长时间未做 Checkpoint，持续时间m 小时	1. 检查 SecondaryNameNode(Standby NameNode) 的状态 2. 检查 HDFS 配置文件 hdfs-site.xml 中的 dfs.namenode.checkpoint.period 和dfs.namenode.checkpoint.txns 参数 3. 查看 HDFS 集群的日志信息	m=24	一般	是	是
	HDFS 小文件占比超过指定阈值	小文件比率>=50%,每天巡检一次	合并相同类型的小文件或定时清理掉小文件或用对象存储来存小文件	m=50	一般	是	是
YARN	集群当前丢失的 NodeManager 的个数持续高于阈值	集群当前丢失的 NodeManager 的个数 >= m，持续时间 t 秒（300<=t<=2592000）	检查 NM 进程状态，检查网络是否畅通	m=1, t=1800	一般	是	否
	Pending Containers 个数持续高于阈值	pending Containers 个数 >= m个，持续时间 t 秒（300<=t<=2592000）	合理指定 YARN 任务可用资源	m=90, t=1800	一般	是	否
	集群内存使用率持续高于阈值	内存使用率 >= m，持续时间 t 秒（300<=t<=2592000）	集群扩容	m=85, t=1800	严重	是	是
	集群 CPU 使用率持续高于阈值	CPU 使用率 >= m，持续时间 t 秒（300<=t<=2592000）	集群扩容	m=85, t=1800	严重	是	是

类别	事件名称	事件含义	建议&措施	默认值	严重程度	允许关闭	默认开启
	各队列中可用的 CPU 核数持续低于阈值	任意队列中可用 CPU 核数 <= m，持续时间 t 秒（300<=t<=2592000）	给队列分配更多资源	m=1, t=1800	一般	是	否
	各队列中可用的内存持续低于阈值	任意队列中可用内存 <= m，持续时间 t 秒（300<=t<=2592000）	给队列分配更多资源	m=1024, t=1800	一般	是	否
	ResourceManager 发生主备切换	ResourceManager 发生了主备切换	检查 RM 进程状态，查看 standby RM 日志查看主备切换原因	-	严重	是	是
	ResourceManager 发生 full GC	ResourceManager 发生了 full GC	参数调优	-	严重	是	是
	ResourceManager JVM 内存使用率持续高于阈值	RM JVM 内存使用率持续 >= m，持续时间 t 秒（300<=t<=2592000）	调整 ResourceManager 堆内存大小	m=85, t=1800	严重	是	是
	NodeManager 发生 full GC	NodeManager 发生 full GC	参数调优	-	一般	是	否
	NodeManager 可用的内存持续低于阈值	单个 NM 可用内存持续 <= m，持续时间 t 秒（300<=t<=2592000）	调整 NodeManager 堆内存大小	m=1, t=1800	一般	是	否
	NodeManager JVM 内存使用率持续高于阈值	NM JVM 内存使用率持续 >= m，持续时间 t 秒（300<=t<=2592000）	调整 NodeManager 堆内存大小	m=85, t=1800	一般	是	否
	YARN ResourceManager 无 active 状态	YARN ResourceManager 无 active 状态	人工排查	t=90	严重	是	是
	Yarn Application 作业运行失败次数持续高于阈值	Yarn Application 作业运行失败	人工排查	m=1,t=300	一般	是	否
	YARN 当前不健康的 NodeManager 的个数持续高于阈值	Unhealthy NodeManager 个数 >=m，持续时间t秒（300<=t<=2592000）	人工排查	m=1,t=1800	一般	是	否
	YARN application 运行时长高于阈值	Yarn application 运行时间>=m(min)	人工排查	t=30min	一般	是	否
HBase	被拉黑的NodeManager阈值	App 被拉黑的节点数大于阈值	人工排查	m=0	一般	是	否
	集群 dead RS 数量持续高于阈值	集群 dead RegionServer 数量 >= m，持续时间 t 秒（300<=t<=2592000）	人工排查	m=1, t=60	一般	是	是
	集群每个 RS 平均 REGION 数持续高于阈值	集群每个 RegionServer 平均 region 数 >= m，持续时间 t 秒（300<=t<=2592000）	节点扩容或升配	m=300, t=1800	一般	是	是
	HMaster 发生 full GC	HMaster 发生了 full GC	参数调优	m=5, t=300	一般	是	是
	HMaster JVM 内存使用率持续高于阈值	HMaster JVM 内存使用率 >= m，持续时间 t 秒（300<=t<=2592000）	调整 HMaster 堆内存大小	m=85, t=1800	严重	是	是
	HMaster 当前连接数持续高于阈值	HMaster 当前连接数 >= m，持续时间 t 秒（300<=t<=2592000）	人工排查	m=1000, t=1800	一般	是	否

类别	事件名称	事件含义	建议&措施	默认值	严重程度	允许关闭	默认开启
	RegionServer 发生 full GC	RegionServer 发生 full GC	参数调优	m=5, t=300	严重	是	否
	RegionServer JVM 内存使用率持续高于阈值	RegionServer JVM 内存使用率 >= m，持续时间 t 秒（300<=t<=2592000）	调整 RegionServer 堆内存大小	m=85, t=1800	一般	是	否
	RegionServer 当前 RPC 连接数持续高于阈值	RegionServer 当前 RPC 连接数 >= m，持续时间 t 秒（300<=t<=2592000）	人工排查	m=1000, t=1800	一般	是	否
	RegionServer Storefile 个数持续高于阈值	RegionServer StoreFile 个数 >= m，持续时间 t 秒（300<=t<=2592000）	建议执行 major compaction	m=50000, t=1800	一般	是	否
	HBASE 两个 HMaster 服务状态均为 Standby	两个 HMaster 角色同时处于 Standby 状态	人工排查	-	严重	是	是
	HMaster发生主备切换	HMaster发生主备切换	通过 HMaster 服务日志进行排查	-	严重	是	是
	HBaseThrift 发生 full GC	HBaseThrift 发生 full GC	参数调优	m=5, t=300	严重	是	否
	HBaseThrift JVM 内存使用率持续高于阈值	HBaseThrift JVM内存使用率>= m，持续时间 t 秒	调整 HBaseThrift 堆内存大小	m=85, t=1800	一般		
Hive	HiveServer2 发生 full GC	HiveServer2 发生 full GC	参数调优	m=5, t=300	严重	是	是
	HiveServer2 JVM 内存使用率持续高于阈值	HiveServer2 JVM 内存使用率 >= m，持续时间 t 秒（300<=t<=2592000）	调整 HiveServer2 堆内存大小	m=85, t=1800	严重	是	是
	HiveMetaStore 发生 full GC	HiveMetaStore 发生 full GC	参数调优	m=5, t=300	一般	是	是
	HiveWebHcat 发生 full GC	HiveWebHcat 发生 full GC	参数调优	m=5, t=300	一般	是	是
	HIVE SQL 编译时间高于阈值	HIVE SQL 编译时间高于阈值	人工排查	t=60	一般	是	否
ZooKeeper	ZooKeeper 连接数持续高于阈值	ZooKeeper 连接数 >= m，持续时间 t 秒（300<=t<=2592000）	人工排查	m=65535, t=1800	一般	是	否
	znode 节点数量持续高于阈值	znode 节点数 >= m，持续时间 t 秒（300<=t<=2592000）	人工排查	m=100000, t=1800	一般	是	否
	ZooKeeper 发生 leader 切换	ZooKeeper发生leader切换	通过 Zookeeper 服务日志进行排查	-	严重	是	是
集群	JVM OLD 区异常	JVM OLD 区异常	人工排查	1. old 区连续5分钟80%或者 2. JVM 内存使用率达到90%	严重	是	是
	节点角色进程重启	节点角色进程重启	人工排查	/	一般	否	是

类别	事件名称	事件含义	建议&措施	默认值	严重程度	允许关闭	默认开启
	服务角色健康状态异常	服务角色健康状态异常，持续时间t秒(180<=t<=604800)	服务角色健康状态连续分钟级不可用。 处理方式：查看对应服务角色日志信息，根据日志处理。	t=300	严重	是	是
	服务角色健康状态超时	服务角色健康状态超时，持续时间t秒(180<=t<=604800)	服务角色健康状态连续分钟级超时。 处理方式：查看对应服务角色日志信息，根据日志处理。	t=300	一般	是	否

配置告警

操作场景

TBDS 已接入告警平台，用户通过告警平台在配置 TBDS 节点和服务监控指标的告警策略。

- TBDS 已接入告警平台，集群创建时会同步自动默认告警策略。
- 持手动创建告警策略。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。
2. 在集群详情页中选择集群监控 > 告警历史 > 告警配置 跳转到监控系统告警策略页，点击新建策略进行告警策略配置，自定义配置步骤详见：自定义告警。
3. 在弹出的"新建告警策略"窗口中，进行告警策略配置。
4. 填写完后，单击完成即可完成新告警策略配置。

告警字段与配置项对照表

配置类型	配置项		说明
基本信息	策略名称		自定义策略名称
	备注		自定义策略备注
	监控类型		持云产品监控类型
	策略类型		选择您需要监控的云产品策略类型
	策略所属项目		设置所属项目后，您可以在告警策略列表快速筛选该项目下的告警策略
配置告警规则	告警对象		选择产品，则该告警策略绑定所选产品和模块 选择对象，即该告警策略绑定所选对象 选择全部对象，则该告警策略绑定当前账号拥有权限的全部对象
	触发条件	手动配置（指标告	告警触发条件：指标、比较关系、阈值、统计周期和持续周期组成的一个有语义的条件。您可以展开触发条件详情，根据图表中

配置类型	配置项		说明
		警）	指标变化趋势设置合适的告警阈值
		手动配置（事件告警）	在节点或底层基础设施服务发生异常时，可以创建事件告警及时通知您采取措施
		选择模板	选择模板按钮，并在下拉列表选择已配置的模板，配置触发条件模板
	配置告警通知（可选）	通知模板	默认绑定系统预设通知模板，每个告警策略最多只能绑定三个通知模板

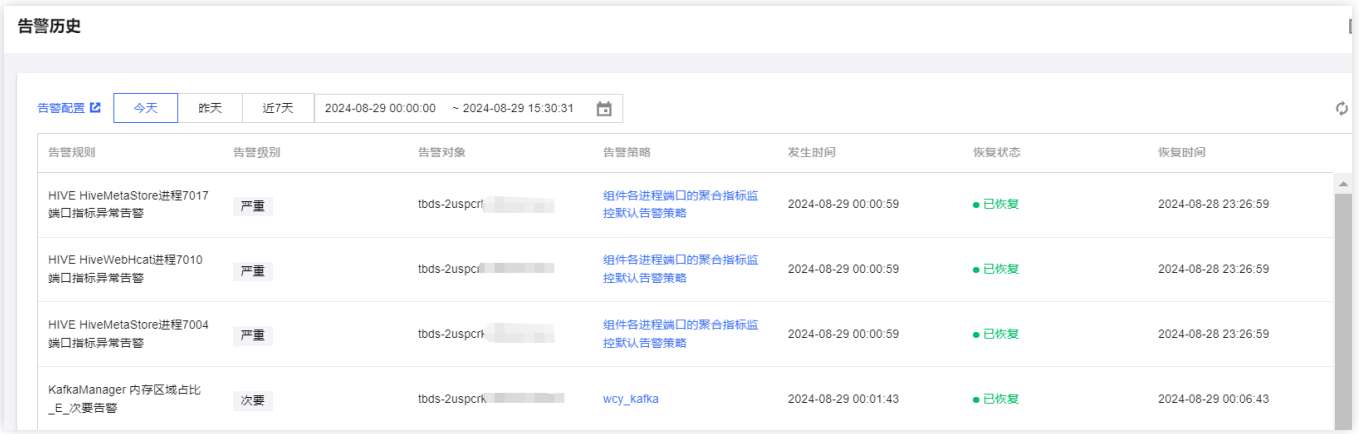
告警历史

功能介绍

告警历史功能可以帮您回溯和查看当前集群的告警历史记录。您还可以在告警历史页通过策略类型快速进对应告警策略并进行订阅。

操作步骤

1. 登录 TBDS Manager 管理平台，在集群列表中单击对应的集群 ID/名称进集群详情页。在集群详情页中选择集群监控 > 告警历史，可根据告警内容模糊搜索对应告警，也可根据时间范围和告警状态进行筛选，可以通过开始时间进行排序。



2. 单击告警配置即可快速进监控系统告警配置页面，进行告警策略配置。

集群巡检

功能介绍

1. 经典集群可即时或定时（按天、按周）根据已选的巡检项对集群的节点和服务进行健康检查，以便周期性掌握集群健康情况，及时对异常风险点进行处理。
2. 平台提供默认巡检项，用户可按需勾选需要增加的巡检项目。
3. 每次巡检任务完成后生成 PDF 格式的巡检报告，用户可以下载或删除巡检报告。

> 说明：
>
> - 单个集群仅可配置一个定期巡检任务。
> - 服务功能类巡检会消耗集群性能，不推荐在业务高峰期进行有耗损的巡检。
> - 每个集群最多可保留200份巡检报告，超过保存的最大限额将会从最早期报告开始滚动删除。
> - 定时巡检任务正在执行中时，不能修改保存配置。

巡检范围

5315版本支持的巡检范围包括：

1. 集群类型：支持Hadoop 类型集群。
2. 服务类型：支持 HDFS、YARN、HBase、Hive、Zookeeper组件。
3. 预置巡检项：

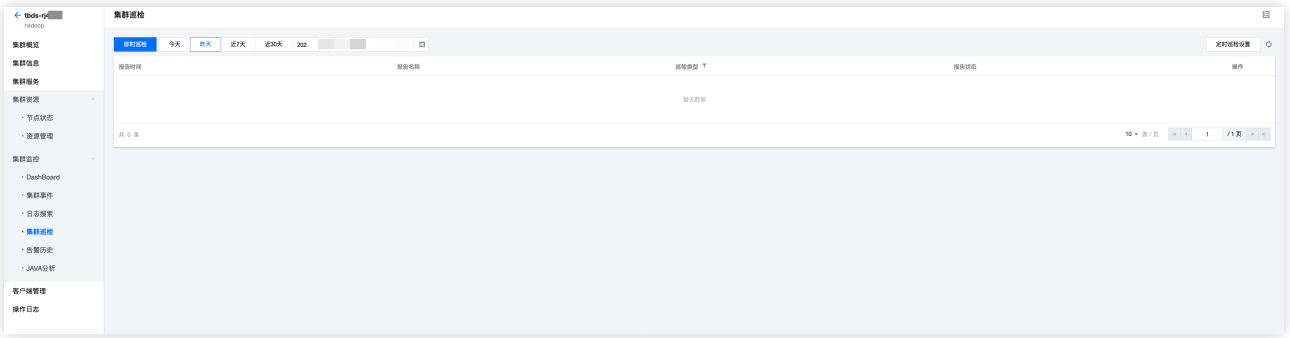
维度	服务类型	巡检项
集群	-	JVM OLD区异常
		节点角色进程重启
		自动伸缩策略过期
		服务角色健康状态异常
		自动伸缩策略执行失败
		自动伸缩策略未触发
		自动伸缩策略执行超时
		自动伸缩扩容部分成功
		引导脚本执行失败
		进程被OOM killer kill
节点	-	CPU利用率连续高于阈值

		CPU IOwait平均值高于阈值
		CPU1分钟负载连续高于阈值
		内存使用率持续高于阈值
		系统进程总数连续高于阈值
		元数据库异常
		单盘INODES使用率持续高于阈值
		单盘I/O设备利用率持续高于阈值
		单盘空间使用率持续高于阈值
		子机UTC时间和NTP时间差值高于阈值
		节点磁盘I/O异常
		节点故障
		故障节点自动补偿
		实例硬盘异常待授权
		实例运行异常待授权
		子机nvme设备error
		机器重启
		内存OOM
		内核故障
		磁盘只读
		ping不可达
组件	HDFS	HDFS存储空间使用率持续高于阈值
		NameNode 发生主备切换
		NameNode 发生Full GC
		NameNode JVM内存使用率持续高于阈值
		DataNode JVM内存使用率持续高于阈值
		HDFS 两个NameNode服务状态均为Standby
		HDFS NameNode进入安全模式
		HDFS MissingBlocks数量持续高于阈值
		HDFS NameNode长时间未做 Checkpoint

	YARN	ResourceManager 发生主备切换
		ResourceManager 发生Full GC
		ResourceManager JVM内存使用率持续高于阈值
		YARN ResourceManager无active状态
	HBase	集群处于RIT Region个数持续高于阈值
		集群dead RS数量持续高于阈值
		集群每个RS平均REGION数持续高于阈值
		HMaster 发生Full GC
		HMaster JVM内存使用率持续高于阈值
		HBASE 两个HMaster服务状态均为Standby
		HMaster 发生主备切换
	Hive	HiveServer2 Full GC
		HiveServer2 JVM内存使用率持续高于阈值
		HiveMetaStore Full GC
		HiveWebHcat Full GC
	ZK	Zookeeper 发生Leader切换

操作步骤

1. 登录 TBDS Manager，在集群列表中单击对应的集群名称进入集群详情页。
2. 在集群详情页中选择集群监控 > 集群巡检可根据当前集群的节点和服务进行健康检查，用户可单击“即时巡检”进行巡检；也可单击“定时巡检设置”，配置定时巡检任务。



即时巡检

即时巡检是检查集群从某个时刻到当前时间节点和服务的健康状态并生成巡检报告。

定时巡检

1. 定期巡检策略开启后，系统将自动检测每个巡检周期内集群节点和服务的健康状态并生成巡检报告。每个集群可配置一个定期巡检策略，配置更新后将覆盖历史。

i. 巡检项：默认支持所有已开启的事件监控策略，若需调整巡检项可参考 集群事件 进行设置。初始巡检项系统默认勾选所有已开启监控的事件，修改后第二次设置巡检项，默认会勾选上一次已选择的巡检项。

ii. 巡检报告：每次的即时巡检/定时巡检任务结束后，会自动生成一份巡检报告，该报告支持在线预览、下载、删除。

TBDS 巡检报告

集群ID:tbds-rj4d8v4v　　巡检时间:2025年

1 建议与措施

- 人工排查

巡检总数: 43　异常数: 1

2 基本信息

节点个数: 6 个　CPU总核数: 96 核　磁盘总容量: 3.15 TB　内存总量: 180.90 GB

部署服务: Kerberos, Ranger, Zookeeper, YARN, HDFS, Trino, Impala, Hive, Kudu, Kyuubi, Knox, Spark, Tez, Hue, HBase, Kafka, Flink

资源使用聚合图

(a) CPU使用率(%)　(b) 内存使用率(%)　(c) 磁盘使用率(%)　(d) 网络流量(MB/s)

3 节点巡检

巡检类别	巡检项	阈值	时间(s)	阈值生效时间	巡检结果	异常情况
NODE	CPU IOwait平均值高于阈值	60	1800	:25至今	正常	-
NODE	CPU5分钟负载连续高于阈值	8	1800	:25至今	正常	-
NODE	内存使用率持续高于阈值	85	1800	:25至今	正常	-
NODE	系统进程总数连续高于阈值	10000	1800	:25至今	正常	-
NODE	元数据库Ping失败	-	-	:25至今	正常	-
CLUSTER	JVM OLD区异常	-	-	:25至今	正常	-
NODE	单盘INODES使用率持续高于阈值	85	1800	:25至今	正常	-
NODE	单盘IO设备利用率持续高于阈值	85	1800	:25至今	正常	-
NODE	单盘空间使用率持续高于阈值	85	1800	:25至今	正常	-
NODE	节点进程重启(HUE)	-	-	:25至今	异常	
CLUSTER	服务角色健康状态异常	-	300	:25至今	正常	-
NODE	节点磁盘IO异常	-	-	:26至今	正常	-
NODE	节点磁盘异常	-	-	:26至今	正常	-
NODE	CPU利用率连续高于阈值	85	1800	:26至今	正常	-

4 服务巡检

巡检类别	巡检项	阈值	时间(s)	阈值生效时间	巡检结果	异常情况
HDFS	HDFS文件总数持续高于阈值	50000000	1800	25至今	正常	-
HDFS	HDFS总block数量持续高于阈值	50000000	1800	25至今	正常	-
HDFS	HDFS标记为 Dead 状态的数据节点数量持续高于阈值	1	1800	25至今	正常	-
HDFS	HDFS存储空间使用率持续高于阈值	85	1800	25至今	正常	-
HDFS	NameNode 发生主备切换	-	-	25至今	正常	-
HDFS	NameNode RPC请求处理延迟持续高于阈值	300	300	25至今	正常	-
HDFS	NameNode 发生full GC	-	-	25至今	正常	-
HDFS	NameNode JVM内存使用率持续高于阈值	85	1800	25至今	正常	-
HDFS	DataNode JVM内存使用率持续高于阈值	85	1800	25至今	正常	-

1

YARN	ResourceManager 发生主备切换	-	-	8:25至今	正常	-

YARN	ResourceManager 发生full GC	-	-	8:25至今	正常	-
YARN	ResourceManager JVM内存使用率持续高于阈值	85	1800	8:25至今	正常	-
HIVE	HiveServer2 发生full GC	-	-	8:25至今	正常	-
HIVE	HiveServer2 JVM内存使用率持续高于阈值	85	1800	8:25至今	正常	-
HIVE	HiveMetaStore 发生full GC	-	-	8:25至今	正常	-
HIVE	HiveWebHcat 发生full GC	-	-	8:25至今	正常	-
HDFS	HDFS 两个NameNode服务状态均为Standby	-	300	8:25至今	正常	-
HDFS	HDFS NameNode进入安全模式	-	-	8:25至今	正常	-
HDFS	HDFS MissingBlocks数量持续高于阈值	1	1800	8:25至今	正常	-
ZOOKEEPER	Zookeeper 发生Leader切换	-	-	8:26至今	正常	-
HDFS	HDFS NameNode长时间未做 Checkpoint	24	-	8:26至今	正常	-
YARN	YARN ResourceManager无active状态	-	90	8:26至今	正常	-
HBASE	集群处于RIT Region个数持续高于阈值	1	60	2:42至今	正常	-
HBASE	集群dead RS数量持续高于阈值	1	60	2:42至今	正常	-
HBASE	集群每个RS平均REGION数持续高于阈值	300	1800	2:42至今	正常	-

SR集群高级配置
SR 冷热数据分离

简要介绍

在大数据业务场景中，数据访问通常具有明显的时效性特征：近期数据（如最近一周或一个月）被频繁查询和更新，属于高价值的热数据；而历史久远的数据访问频率极低，称为冷数据。为兼顾查询性能与存储成本，冷热数据分层存储成为主流解决方案。TBDS-SR作为大数据平台的存储计算组件，支持基于时间策略的自动化数据分层管理。通过将热数据存储在SSD、冷数据自动迁移至HDD，TBDS-SR可在保障关键业务查询性能的同时，显著降低整体存储成本。该功能特别适用于日志分析、实时数仓、用户行为分析等具有强时间序列特征的业务场景。

属性说明

设置初始存储介质和自动存储降冷时间

```
PROPERTIES (
    "storage_medium" = "[SSD|HDD]",
    { "storage_cooldown_ttl" = "<num> { YEAR | MONTH | DAY | HOUR } "
    | "storage_cooldown_time" = "yyyy-MM-dd HH:mm:ss" }
)
```

- storage_medium ：初始存储介质，可以设置为 SSD 或 HDD。确保您明确指定的存储介质类型与 BE 静态参数 storage_root_path 为您的 StarRocks 集群指定的 BE 磁盘类型一致。

  如果 FE 配置项 enable_strict_storage_medium_check 设置为 true，系统在创建表时严格检查 BE 磁盘类型。如果您在 CREATE TABLE 中指定的存储介质与 BE 磁盘类型不一致，则返回错误 "Failed to find enough host in all backends with storage medium is SSD|HDD." ，并且表创建失败。
  如果 enable_strict_storage_medium_check 设置为 false，系统会忽略此错误并强制创建表。但是，加载数据后，集群磁盘空间可能会分布不均。

  如果未明确指定 storage_medium ，系统会根据 BE 磁盘类型自动推断存储介质。

  - 在以下场景中，系统会自动将此参数设置为 SSD：
    - BEs 报告的磁盘类型（ storage_root_path ）仅包含 SSD。
    - BEs 报告的磁盘类型（ storage_root_path ）同时包含 SSD 和 HDD。请注意，当 BEs 报告的

storage_root_path 同时包含 SSD 和 HDD 且属性 storage_cooldown_time 被指定时，系统会将 storage_medium 设置为 SSD。

- 在以下场景中，系统会自动将此参数设置为 HDD：
  - BEs 报告的磁盘类型（ storage_root_path ）仅包含 HDD。
  - 当 BEs 报告的 storage_root_path 同时包含 SSD 和 HDD 且属性 storage_cooldown_time 未指定时，系统会将 storage_medium 设置为 HDD。

- storage_cooldown_ttl 或 storage_cooldown_time ：自动存储降冷时间或时间间隔。自动存储降冷是指将数据从 SSD 自动迁移到 HDD。此功能仅在初始存储介质为 SSD 时有效。
  - storage_cooldown_ttl ：此表中分区的自动存储降冷的时间间隔。如果您需要保留最近的分区在 SSD 上，并在一定时间间隔后自动将较旧的分区降冷到 HDD，可以使用此参数。每个分区的自动存储降冷时间是使用此参数的值加上分区的上限时间计算的。

  支持的值为 <num> YEAR 、 <num> MONTH 、 <num> DAY 和 <num> HOUR 。 <num> 是一个非负整数。默认值为 null，表示不自动执行存储降冷。

  例如，您在创建表时将值指定为 "storage_cooldown_ttl"="1 DAY" ，并且存在范围为 [2023-08-01 00:00:00,2023-08-02 00:00:00) 的分区 p20230801 。此分区的自动存储降冷时间为 2023-08-03 00:00:00 ，即 2023-08-02 00:00:00 + 1 DAY 。如果您在创建表时将值指定为 "storage_cooldown_ttl"="0 DAY" ，则此分区的自动存储降冷时间为 2023-08-02 00:00:00 。
  - storage_cooldown_time ：表从 SSD 降冷到 HDD 的自动存储降冷时间（绝对时间）。指定的时间需要晚于当前时间。格式： "yyyy-MM-dd HH:mm:ss" 。当您需要为指定分区配置不同的属性时，可以执行 ALTER TABLE ... ADD PARTITION 或 ALTER TABLE ... MODIFY PARTITION 。

参考指南

要基于TBDS-SR实现冷热数据管理，需完成存储介质配置与表/分区策略设置两步核心操作。

1. 配置存储介质

在BE节点的 be.conf 配置文件中，通过 storage_root_path 参数显式声明各存储路径的介质类型。示例如下：

```
storage_root_path = /data1,medium:HDD;/data2,medium:SSD;/data3
```

该配置定义了三个存储路径：

- /data1 ：指定为HDD存储；

- /data2 ：指定为SSD存储；
- /data3 ：未指定介质，默认使用HDD。

配置完成后重启BE节点，系统将根据此配置进行数据分布与迁移。

> 说明：
>
> 在TBDS-SR【集群部署】阶段。选择【高级设置】可以界面化配置存储介质。

2. 建表与分区策略设置示例

方式1：创建分区表时统一设置冷热策略

```
CREATE TABLE part_exp (
    c1 INT,
    c2 INT,
    c3 DECIMAL(10,2),
    c4 DATETIME NOT NULL,
    c5 DECIMAL(18,2) SUM DEFAULT "0"
) AGGREGATE KEY(c1,c2,c3,c4)
PARTITION BY RANGE(c4) (
    PARTITION p1 VALUES LESS THAN ("2022-01-01"),
    PARTITION p2 VALUES LESS THAN ("2022-01-02"),
    PARTITION p3 VALUES LESS THAN ("2022-01-03")
)
DISTRIBUTED BY HASH(c1) BUCKETS 8
PROPERTIES (
    "replication_num" = "1",
    "storage_medium" = "SSD",
    "storage_cooldown_time" = "2023-01-01 23:59:59"
);
```

该语句创建的表中所有分区初始存储在SSD上，当系统时间超过 2023-01-01 23:59:59 后，数据将自动迁移至HDD。

方式2：为不同分区设置独立冷却时间

```
CREATE TABLE part_exp_1 (
    c1 INT,
    c2 INT,
    c3 DECIMAL(10,2),
    c4 DATETIME NOT NULL,
    c5 DECIMAL(18,2) SUM DEFAULT "0"
) AGGREGATE KEY(c1,c2,c3,c4)
PARTITION BY RANGE(c4) (
```

```
    PARTITION p1 VALUES LESS THAN ("2022-01-01"),
    PARTITION p2 VALUES LESS THAN ("2022-01-02") ("storage_medium"="SSD", "storage_cooldown_time" = "2023-01-01 23:59:59"),
    PARTITION p3 VALUES LESS THAN ("2022-01-03") ("storage_medium"="SSD", "storage_cooldown_time" = "2024-01-01 23:59:59")
)
DISTRIBUTED BY HASH(c1) BUCKETS 8
PROPERTIES (
    "replication_num" = "1"
);
```

方式3：动态分区表自动启用冷热管理

```
CREATE TABLE part_exp_2 (
    c1 INT,
    c2 INT,
    c3 DECIMAL(10,2),
    c4 DATETIME NOT NULL,
    c5 DECIMAL(18,2) SUM DEFAULT "0"
) AGGREGATE KEY(c1,c2,c3,c4)
PARTITION BY RANGE(c4) (
    PARTITION p1 VALUES LESS THAN ("2022-08-26"),
    PARTITION p2 VALUES LESS THAN ("2022-08-27"),
    PARTITION p3 VALUES LESS THAN ("2022-08-28")
)
DISTRIBUTED BY HASH(c1) BUCKETS 8
PROPERTIES (
    "replication_num" = "1",
    "dynamic_partition.enable" = "true",
    "dynamic_partition.time_unit" = "DAY",
    "dynamic_partition.start" = "-60",
    "dynamic_partition.end" = "1",
    "dynamic_partition.prefix" = "p",
    "dynamic_partition.buckets" = "8",
    "storage_medium" = "SSD",
    "storage_cooldown_time" = "2024-01-01 23:59:59"
);
```

> 前提：需在 fe.conf 中设置 default_storage_medium=SSD 和 storage_cooldown_second=2592000（30天），以确保动态创建的分区自动继承冷热策略。

方式4：修改已有分区策略

```
-- 修改分区p3的冷却时间
ALTER TABLE part_exp MODIFY PARTITION p3 SET(
    "storage_medium" = "SSD",
```

```
    "storage_cooldown_time" = "2023-03-11 10:29:01"
);

-- 新增分区并指定冷热策略
ALTER TABLE part_exp ADD PARTITION p4 VALUES LESS THAN ("2022-01-04") (
    "storage_medium" = "SSD",
    "storage_cooldown_time" = "2023-03-11 10:29:01"
);
```

最佳实践

1. 合理设置冷却时间：根据业务访问规律设定 storage_cooldown_time ，例如将最近7天数据保留在SSD，30天后迁移至HDD。对于长期高频访问的维度表，可设置远期时间（如2030年）避免被冷却。
2. 动态分区结合默认策略：在使用动态分区时，建议在 fe.conf 中将 default_storage_medium=SSD ，并配合表级 storage_medium 和 storage_cooldown_time 确保新分区自动进入冷热管理流程。
3. 灰度验证与监控：首次启用冷热策略时，建议先在非核心表或分区进行测试，观察数据迁移对集群IO的影响。通过TBDS-SR监控界面跟踪迁移任务状态、完成率及资源占用情况。
4. 定期审计存储分布：使用 SHOW PARTITIONS FROM table_name 检查各分区的 StorageMedium 和 CooldownTime ，确保策略按预期执行。
5. 避免主键模型限制：目前主键模型不支持tablet跨磁盘迁移，无法实现冷热分区自动迁移，建议在需要冷热分离的场景优先使用聚合表或更新模型。

用法说明

- 自动存储降冷相关参数之间的比较如下：
    - storage_cooldown_ttl ：一个表属性，指定表中分区的自动存储降冷的时间间隔。系统会在 此参数的值加上分区的上限时间 时自动降冷分区。因此，自动存储降冷在分区粒度上执行，更加灵活。
    - storage_cooldown_time ：一个表属性，指定此表的自动存储降冷时间（绝对时间）。此外，您可以在创建表后为指定分区配置不同的属性。
    - storage_cooldown_second ：一个静态 FE 参数，指定集群内所有表的自动存储降冷延迟。
- 表属性 storage_cooldown_ttl 或 storage_cooldown_time 优先于 FE 静态参数 storage_cooldown_second 。
- 配置这些参数时，需要指定 "storage_medium = "SSD" 。
- 如果您不配置这些参数，则不会自动执行自动存储降冷。
- 执行 SHOW PARTITIONS FROM <table_name> 查看每个分区的自动存储降冷时间。

使用限制

- 不支持表达式和 List 分区。
- 分区列需要是日期类型。
- 不支持多个分区列。
- 不支持主键表。
- 不支持表达式和 List 分区。
- 分区列需要是日期类型。
- 不支持多个分区列。
- 不支持主键表。

查看JDBC连接串

SR集群支持在"集群服务-更多操作"中，查看JDBC连接串信息，方便用户通过JDBC连接集群。

数据库表分析

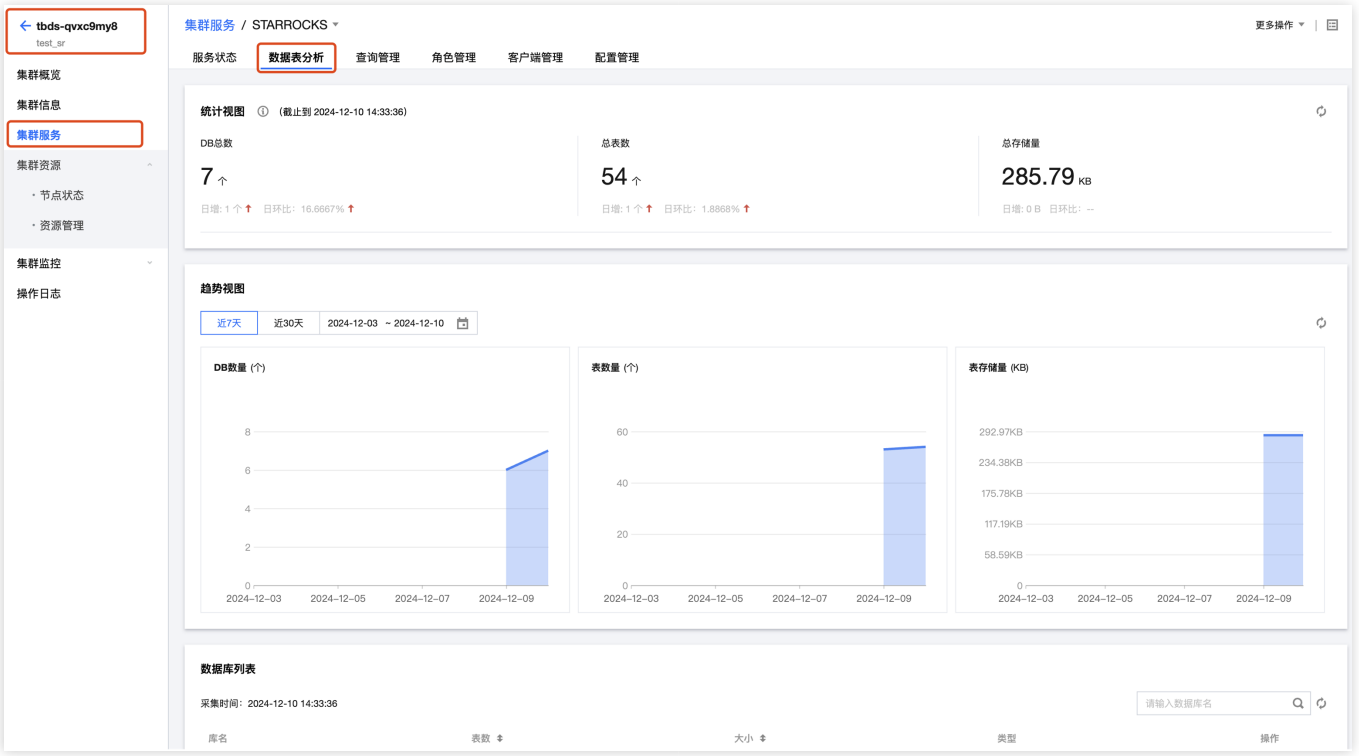
功能介绍

StarRocks 数据表分析帮助用户全面了解数据、管理数据。数据库表分析，分区分片的多维度分析及下钻协助用户根据需要对分区分片进行调整，以优化数据存储和提高查询性能。

操作步骤

1. 登录 TM 控制台，在集群列表中单击对应的 StarRocks 集群的【服务】链接，进入集群服务页。

2. 在集群服务页中单击数据表分析 Tab 页，即可进行相关数据分析查看。DB 总数、表总数及总存储量的日增、日环比及近期的趋势视图快速查看。



3. 提供数据库存储量信息、任务执行记录多维度信息查询；所属表下钻分区分片的存储量及分布信息查看。

> 说明：
>
> i. TBDS 5313 版本之后的StarRocks集群默认支持数据库表分析相关功能。
>
> ii. 数据库表分析信息采集频率为1小时。

查询管理

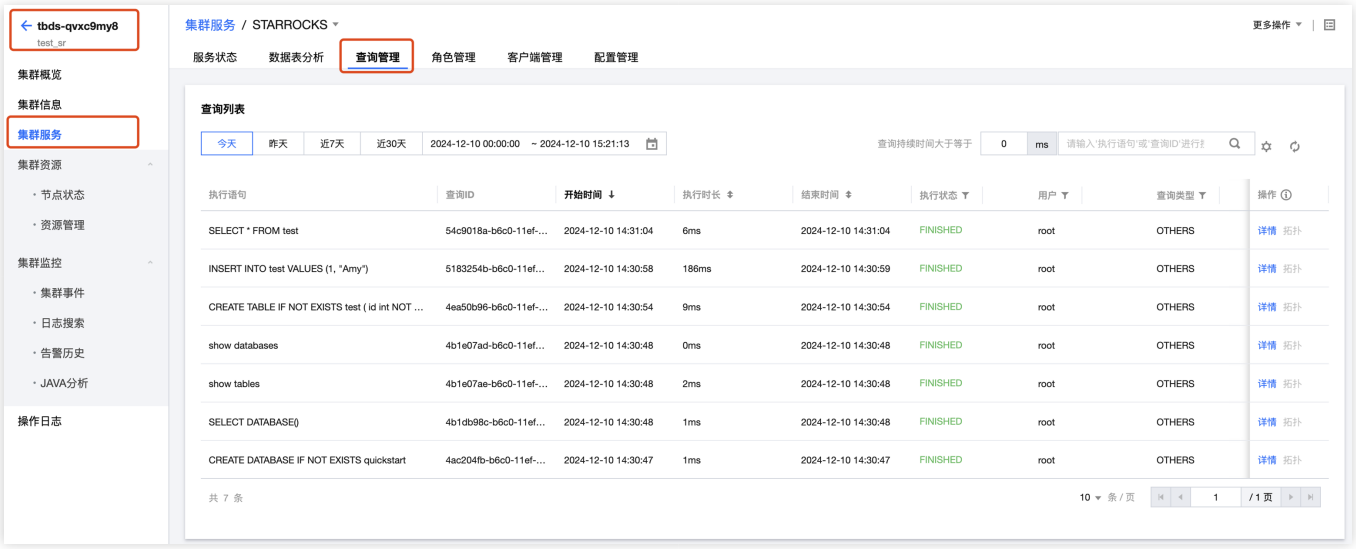
功能介绍

提供 StarRocks 查询多维度指标及详情展示，查询列表可快速查看查询语句、查询开始时间、查询状态、查询持续时间、用户、获取行数、CPU 总时间、消耗总内存等多项明细指标，并可通过查询详情查看相关 SQL、查询计划、Profile、执行拓扑。

操作步骤

1. 登录 TM 控制台，在集群列表中单击对应的 StarRocks 集群的【服务】链接，进入集群服务页。

2. 在集群服务页中单击查询管理 Tab 页，即可进行相关查询列表查看，部分列头字段支持筛选或排序功能，支持多种维度的复合筛选操作，并可自定义查询持续时间进行慢查询筛选。
   查询列表：查看执行语句信息，最右侧操作列 > 详情中可查看查询语句、查询计划、Profile、执行 DAG 和算子排序。



查询热点：根据库表维度，通过统计查询的次数，查询的字节数，查询的行数信息，进行热点排序。

> 注意：
>
> StarRocks 查询管理功能在 TBDS 5313 及以上版本支持。查询热点功能在TBDS 5315 及以上版本支持。
>
> 执行查询前在命令中输入 *set enable_profile=true* 即可打印当前会话查询 SQL 的 Profile，同时采集服务将自动实现该会话提交 SQL 的 Profile 采集。
>
> StarRocks 查询管理中的执行 DAG 及算子排序的可视化功能当前仅在 TBDS 5313 及以上版本支持，该功能需要开启 StarRocks 集群的 Profile 采集。

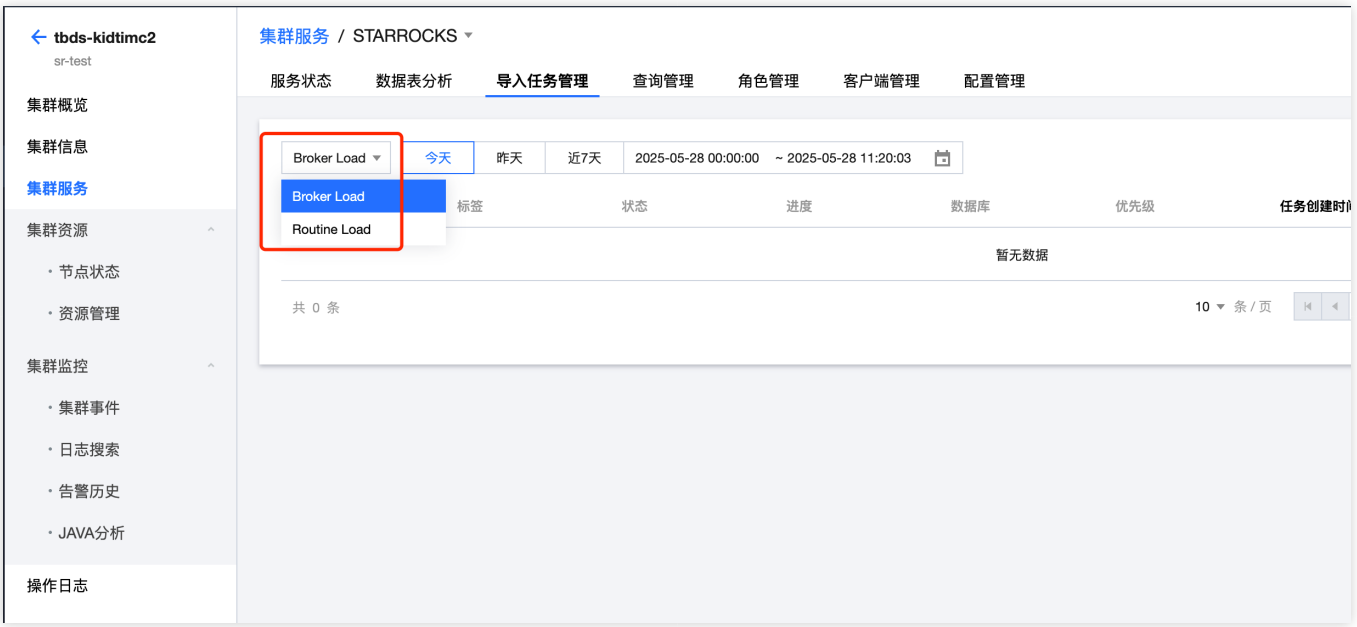
导入任务管理

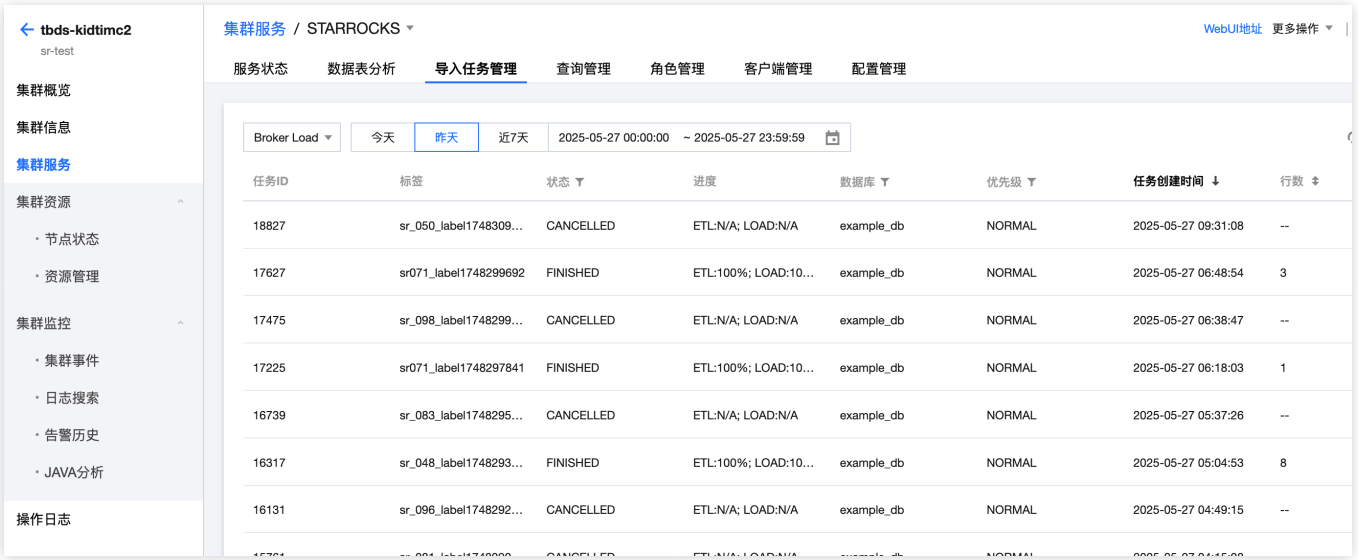
功能介绍

提供批量数据导入 Broker Load 和实时数据导入 Routine Load 类型的任务信息，可查看任务的状态、进度和详细信息方便监控和管理。

操作步骤

1. 登录TM管控平台，在集群列表中单击对应的集群 ID/名称进入集群详情页。
2. 在集群详情页中单击集群服务，然后选择 StarRocks 组件- > 导入任务管理 ，即可查看Broker Load和Routine Load两种导入任务的执行情况和详细导入任务信息。



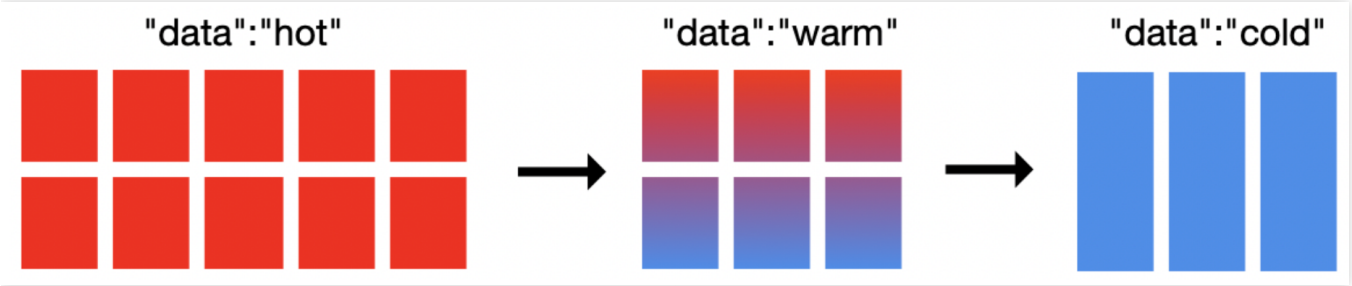


> 说明：
>
> i. TBDS 5315版本开始支持导入任务管理功能，其他版本不支持该功能。
> ii. 导入任务管理信息采集频率为1小时。

ES 集群高级配置
ES 热温冷分层存储策略

简要介绍

TBDS-ES 支持冷热温分层存储策略，核心通过 节点属性标记 + 索引生命周期管理（ILM） 实现数据按访问频率自动分配到不同性能的节点，平衡性能与成本。
热温冷架构常用于日志或指标类的时序数据。例如，假设正在使用 ES 聚合来自多个系统的日志文件，今天的日志正在频繁地被索引，且本周的日志搜索量最大（热）。上周的日志可能会被频繁搜索，但频率没有本周日志那么高（温）。上月日志的搜索频率可能较高，也可能较低，但最好保留一段时间以防万一（冷）。



ES 热温冷数据生命周期示例
如上图集群有 19 个节点：10 个热节点、6 个温节点和 3 个冷节点（实际使用 ILM 实现热温冷架构至少需要有 2 个节点），其中冷节点是可选的，它只是多提供一个建模级别来表示数据的存放位置。ES 允许用户定义哪些节点是热节点、温节点或冷节点，ILM 允许用户定义何时在两个阶段之间移动，以及在进入哪个阶段时如何处理索引。比如在日志、监控、时序数据等场景中，数据的访问频率随时间递减（“热数据”→“温数据”→“冷数据”）：

- 热数据：最近 7 天内的数据，需高并发写入、低延迟查询（如实时监控告警），需高性能硬件。
- 温数据：7~30 天的数据，查询频率降低（如周级报表），可接受稍高延迟，用中性能硬件。
- 冷数据：30 天以上的数据，极少查询（如合规归档），可牺牲查询速度换存储成本，用低成本硬件。

分层架构通过将不同数据分配到对应节点，避免 “用 SSD 存储冷数据” 的资源浪费，同时避免 “用 HDD 存储热数据” 的性能瓶颈。

对于热温冷架构，ES 集群中的节点规格配置并不是一个标准的设置，通常而言遵循：
热节点需要较多的 CPU/内存资源和较快的 IO 带宽资源
对于温节点和冷节点来说，通常每个节点会需要更多的磁盘空间，分配较小规格的 CPU/内存 资源和 IO 带宽资源。

参考实践

> 注意：

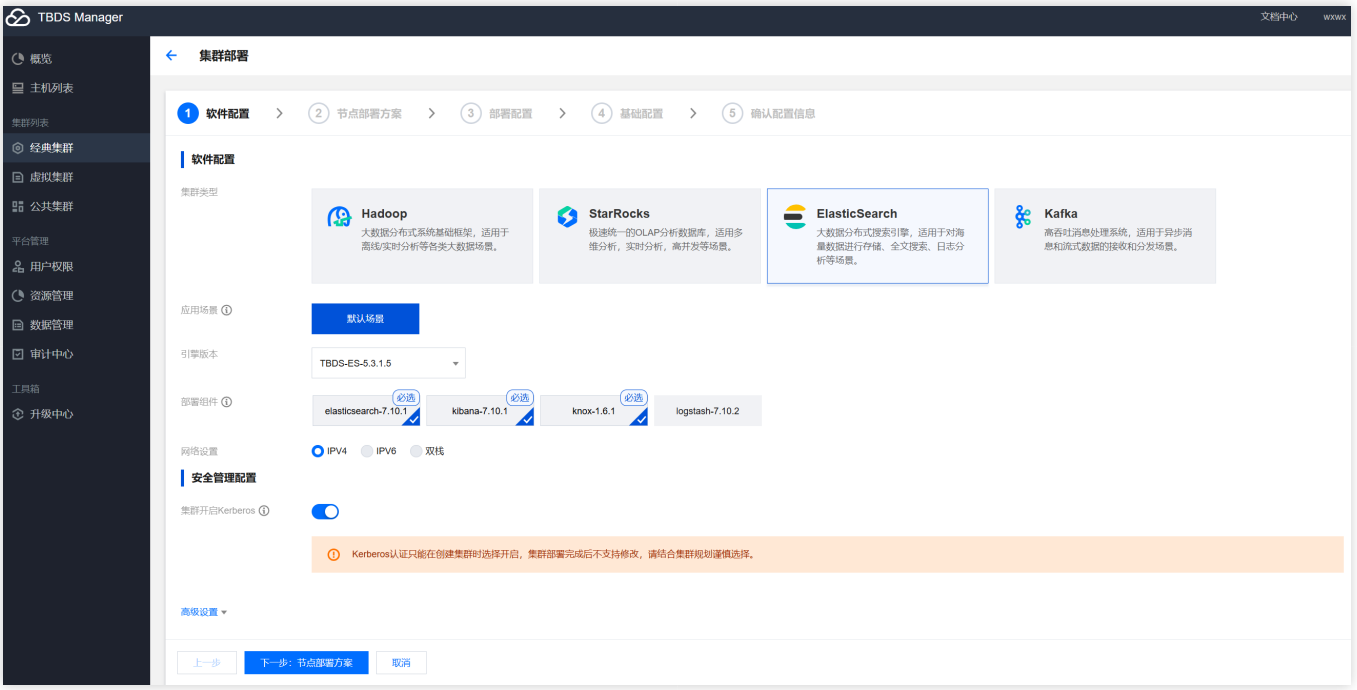
> 下文为示例参考，实际项目请联系交付工程师结合业务情况综合评估后实施。

实施前提：完成服务器资源规划准备，按节点类型分配硬件，用于 ES 集群部署安装/扩容阶段使用。例如：

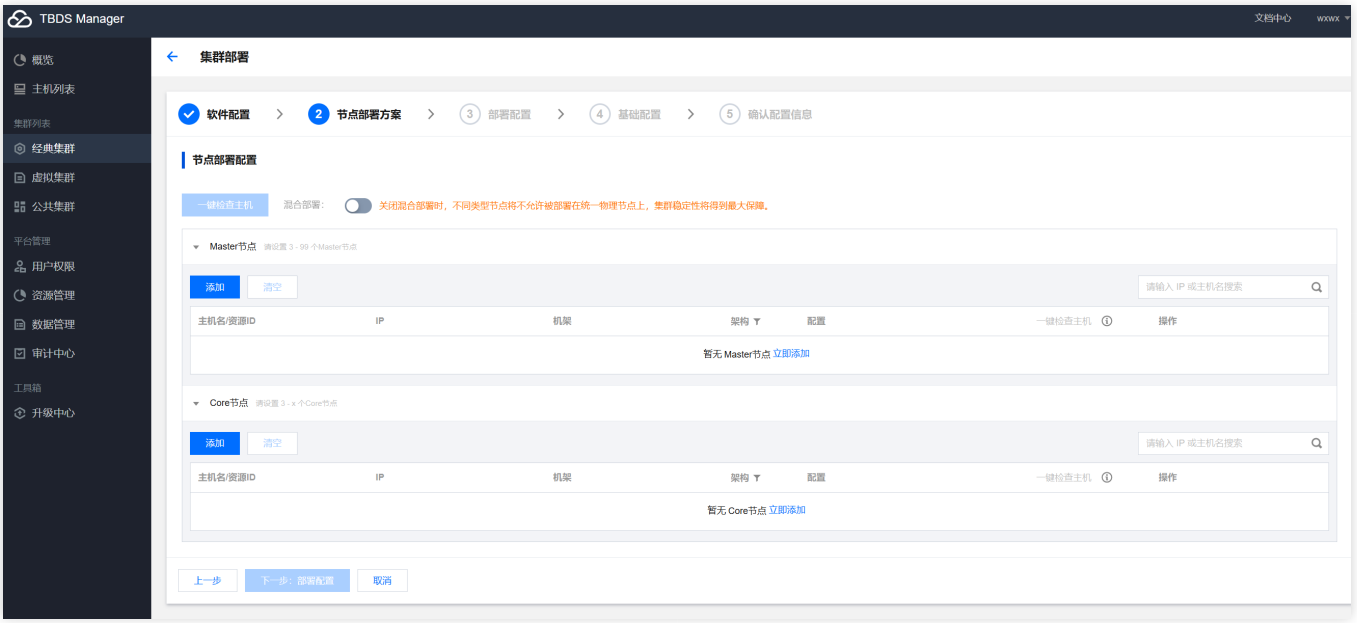
节点类型	CPU	内存	存储介质	核心用途
热节点（Hot）	高核心数（如 16 核）	大内存（如 64GB，缓存数据）	SSD（IOPS ≥ 1000）	写入 + 高频查询
温节点（Warm）	中核心数（如 8 核）	中内存（如 32GB）	HDD（7200 转，大容量）	低频查询
冷节点（Cold）	低核心数（如 4 核）	小内存（如 16GB）	超大规模 HDD	归档查询

（1）创建 TBDS-ES 集群，指定节点属性

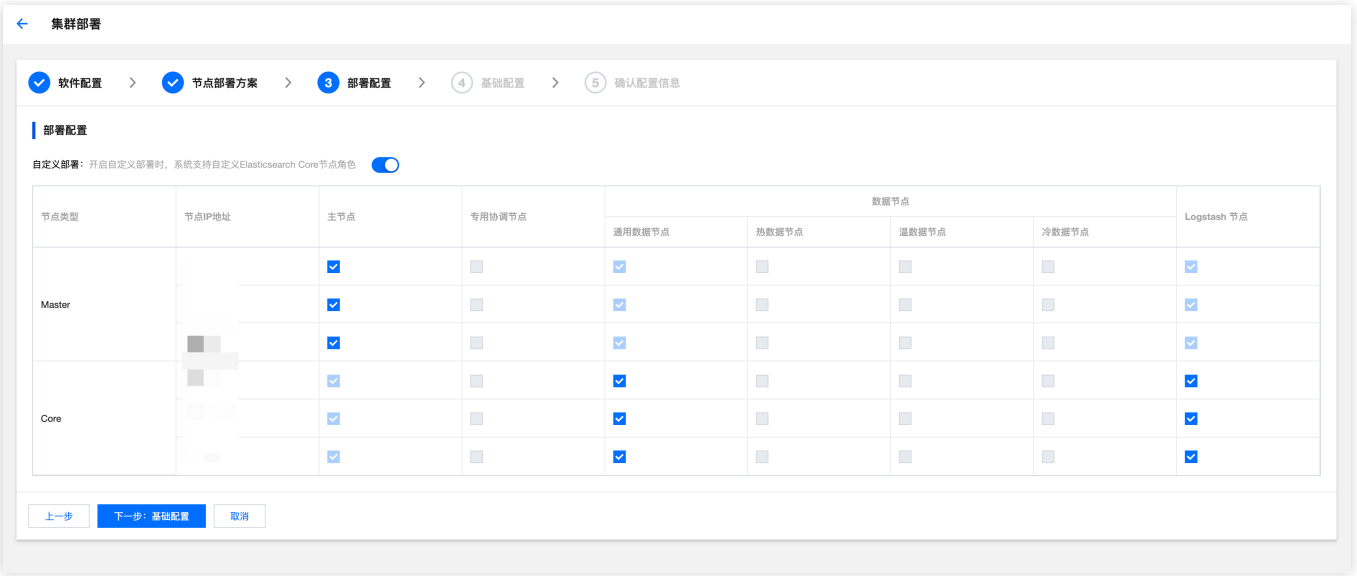
1. 创建集群，选择《ES》集群，按需选择部署的组件，及是否开启Kerberos认证。



2. 点击《下一步：节点部署方案》，根据部署场景选择是否《混合部署》，以及填写主机信息后点击下一步。



3. 点击《下一步：部署配置》，支持打开“自定义部署”，可以调整各个节点的角色，数据节点也支持设置冷热温节点角色。



这里底层会对相应节点做打标处理，如：

节点类型标签（核心，用于分片分配）
node.attr.temp: hot

4. 点击《下一步：基础配置》，填写集群名称

5. 点击《下一步：确认配置信息》，检查集群配置。

6. 点击《部署集群》，等待进度100%之后即可完成部署。

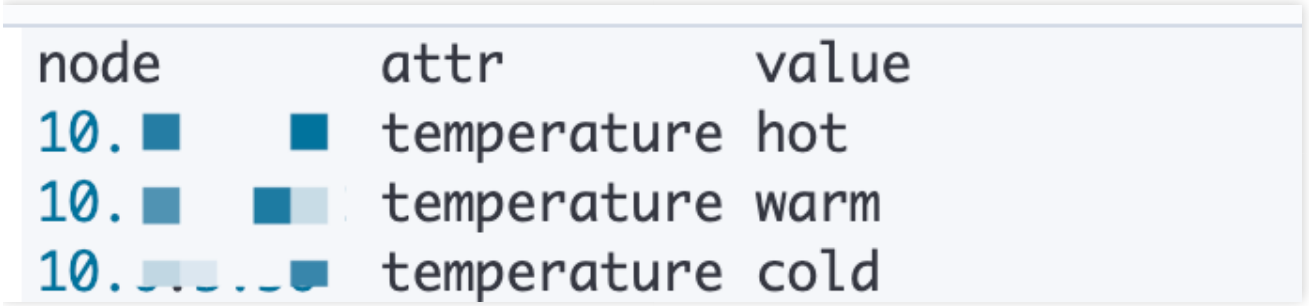
（2）配置索引生命周期管理（ILM）

> 说明：
>
> TBDS-ES 的 ILM 插件可以参考：https://opendistro.github.io/for-elasticsearch-docs/docs/im/ism/policies/

1. 验证节点属性是否生效（通过 API 或 Kibana）：

```
# API查询节点属性curl -X GET "http://<es-master-ip>:9200/_cat/nodeattrs?v&h=node,attr,value"
```

正常返回示例：



2. 创建 ILM 策略（定义数据流转规则）
通过 Kibana 或 API 创建策略，以下是典型的日志数据策略

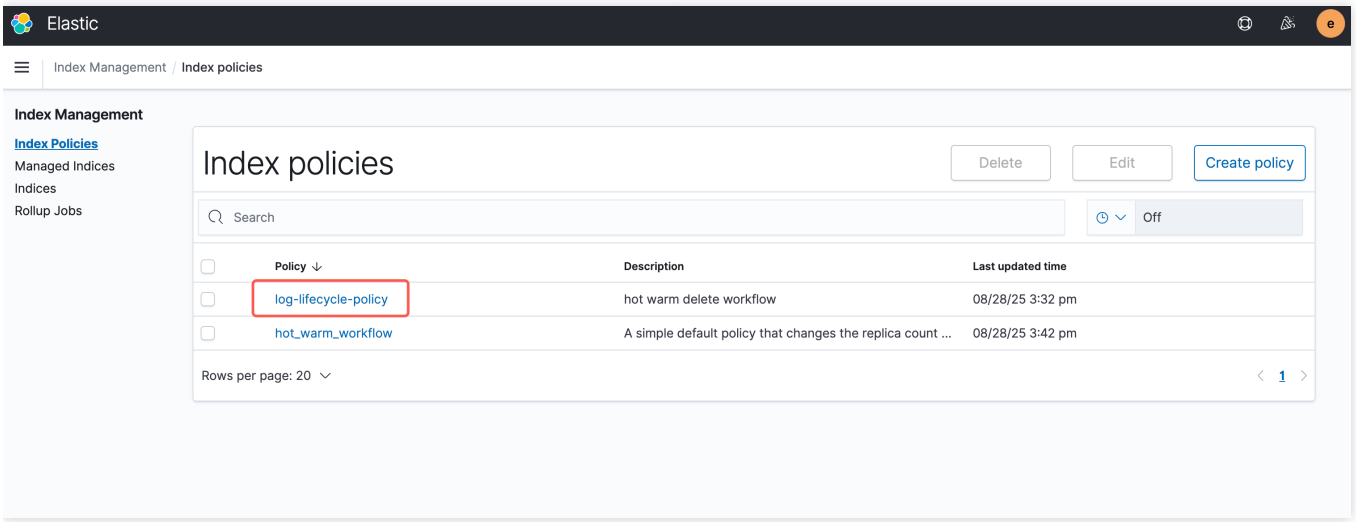
> 说明：
>
> 以下示例策略实现了热（Hot）、温（Warm）、删（Delete）工作流，您可以将此策略用作模板，根据索引的活跃程度为其分配优先资源。

> 在此场景中，索引初始处于热状态（Hot State）。1 天后，索引转为温状态（Warm State），此时副本数量会增加至 5 个，以提升读取性能。
>
> 30 天后，该策略会将索引移入删除状态（Delete State），随后永久删除该索引。

```
PUT _opendistro/_ism/policies/log-lifecycle-policy
{
   "policy": {
      "description": "hot warm delete workflow",
      "default_state": "hot",
      "schema_version": 1,
      "ism_template": {
         "index_patterns": ["log-*"]
      },
      "states": [{
            "name": "hot",
            "actions": [{
               "rollover": {
                  "min_index_age": "1d"
               }
            }],
            "transitions": [{
               "state_name": "warm"
            }]
         },
         {
            "name": "warm",
            "actions": [{
               "replica_count": {
                  "number_of_replicas": 5
               }
            }],
            "transitions": [{
               "state_name": "delete",
               "conditions": {
                  "min_index_age": "30d"
               }
            }]
         },
         {
            "name": "delete",
            "actions": [
               {
                  "delete": {}
               }
            ]
         }
      ]
```

```
    }
  }
```

登录 Kibana 页面，查看 index policies：



3. 创建索引模板（关联 ILM 策略）

索引模板用于让新创建的索引自动应用 ILM 策略，并指定初始分片配置（热节点）。

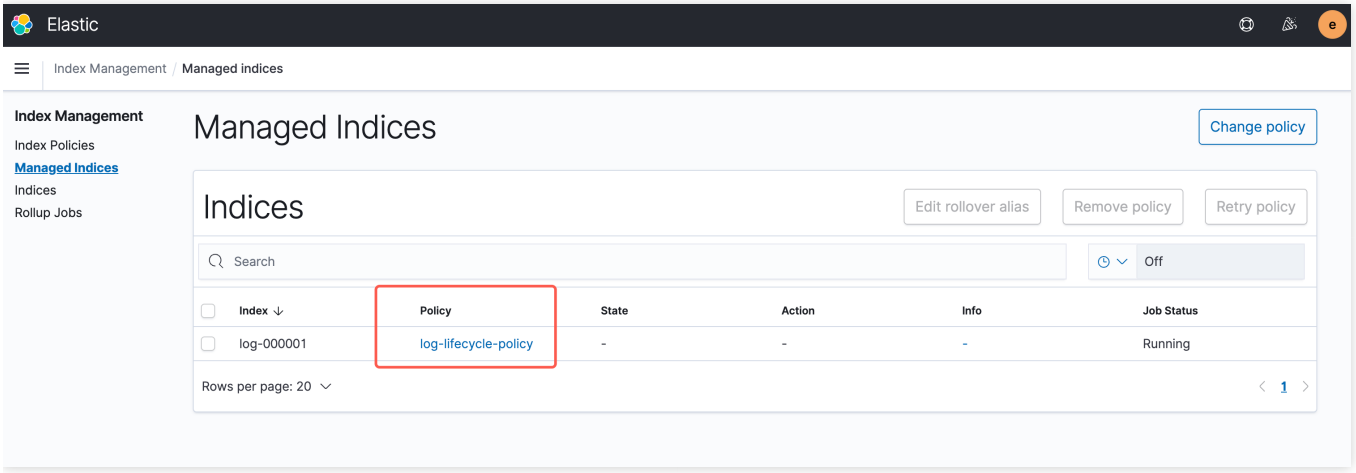
```
PUT _index_template/log-template {
    "index_patterns": ["log-*"],
    "template": {
        "settings": {
            "number_of_shards": 3,
            "number_of_replicas": 1,
            "index.lifecycle.name": "log-lifecycle-policy",
            "index.lifecycle.rollover_alias": "log-alias"
        },
        "mappings": {
            "properties": {
                "timestamp": {
                    "type": "date"
                },
                "log_level": {
                    "type": "keyword"
                },
                "message": {
                    "type": "text"
                }
            }
        }
    }
}
```

4. 创建初始滚动索引

通过 “别名 + 初始索引” 触发 ILM 的 rollover 机制

```
PUT log - 000001 {
    "aliases": {
        "log-alias": {
            "is_write_index": true
        }
    }
}
```

登录 Kibana 页面，查看该索引被相应 Policy 应用

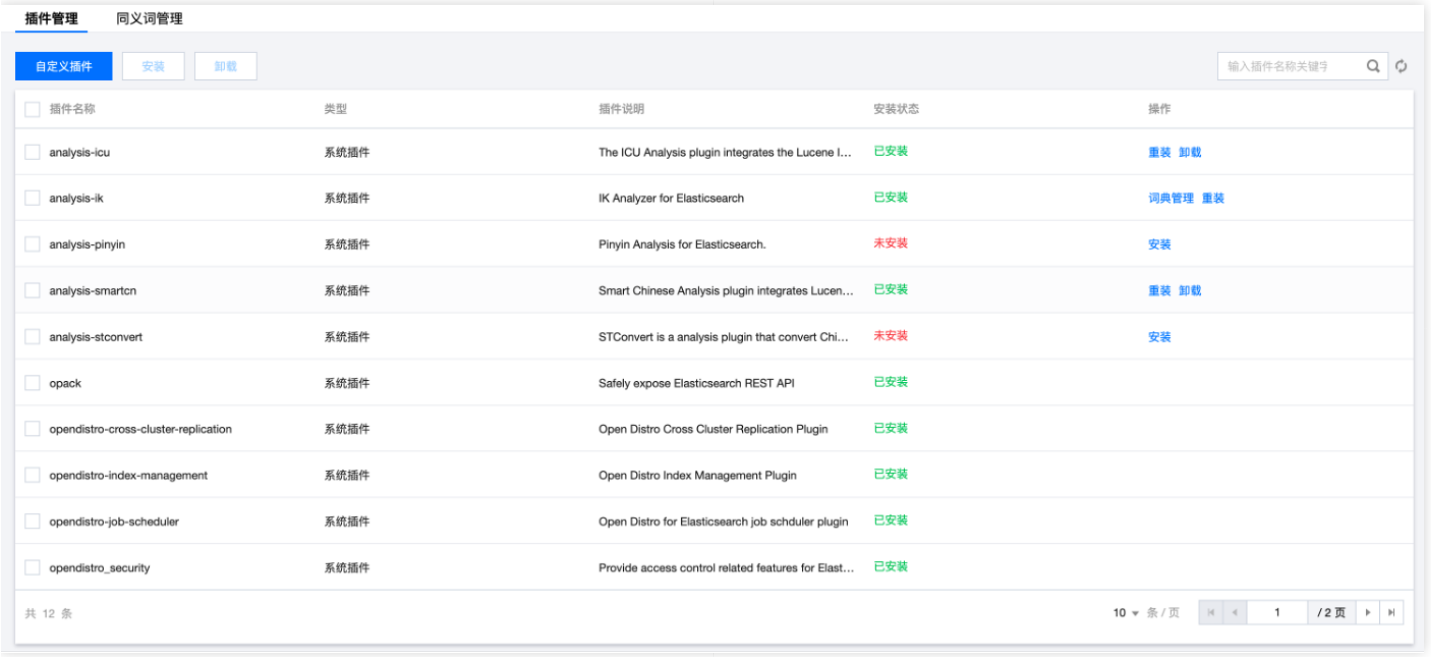


之后，当索引达到rollover条件（7 天或 50GB），ILM 会自动创建log-000002作为新写入索引，旧索引则按策略进入 “温→冷→删除” 阶段。

插件管理

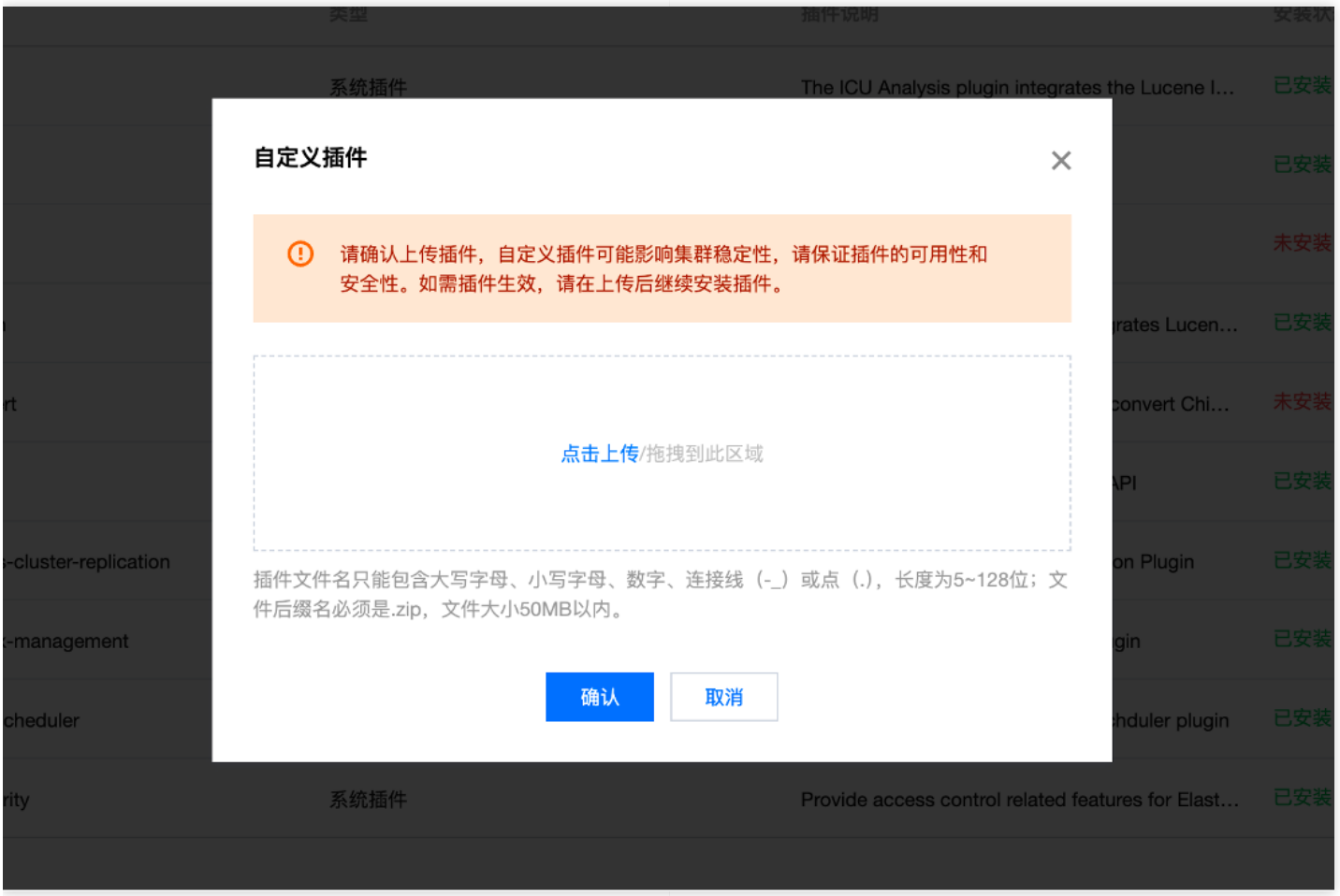
内置插件管理

支持的内置插件包括 ES 开源插件和自有插件在内的10余款插件，为您提供丰富的插件功能。当您部署了TBDS-ES 实例后，您可以根据需求在插件列表页面安装或者卸载这些插件。
支持的功能包括安装、卸载、重装。其中 IK 分词插件还支持词典管理。



自定义插件管理

TBDS-ES 支持用户根据自己的特殊场景安装自定义插件，可以是开源插件，也可以是用户自行开发的插件。不过需要用户自行确认插件的安全合规性，自定义插件可能影响集群稳定性，请保证插件的可用性和安全性。



IK -分词词典管理

系统默认插件中已经预装了 IK 中文分词插件。关于 IK 中文分词插件的介绍，详情可查看 IK Analysis for Elasticsearch https://github.com/infinilabs/analysis-ik，您可以利用该插件对存到 ES 集群数据中的中文关键词建立索引，实现搜索功能。

其中词典管理可以在 web UI 页面进行管理配置，单击“词典管理”更新词典，进入更新词典页面。有分词词典和停用词词典两项，单击本地上传，选择您需要更新的词典文件后，单击保存，即可热更新词典（不需要重启集群）。

关于词典文件的要求及说明如下：

词典类型：有两类词，"分词词典"和"停用词词典"。"分词词典"中的词是用户在向 ES 集群存入数据，建立索引的时候，指定 IK 作为分词工具。如果存入的数据中有这类词，就会建立索引，并能通过关键词查询搜索到。"停用词词典"则会刻意回避不建立索引的词。

限制要求：对于词典文件，也有一些限制和要求，需要一行一个词，UTF-8 编码。为了避免混淆，分词词典和停用词词典文件名不能重复。另外，因为词典文件会加载到内存中，所以对文件的大小和个数也做了一定的限制，单个文件最大为20M，上传文件总数最多为10个。

更新过程：列表会展示历史已经更新上传的词典。新上传的词典，如果不符合要求，会直接限制上传。上传完成后，词典文件会显示成"待生效"状态。所有需要更新的词典上传完成后，单击保存，会保存到用户的集群并生效。如果有上传失败的文件，或不是 UTF-8 格式的文件，会提示失败，需要删除失败的文件后，才能单击保存生效。

同义词管理

TBDS-ES 支持以下两种方式配置同义词：上传同义词文件、直接引用同义词。
操作方式：上传同义词文件

- 最多50个文件，单文件最大 20 M。
- 新上传/新变更的同义词文件对老索引不生效，需要重建索引。
- 同义词文件要求每行一个同义词表达式（表达式支持 Solr 规则 和 WordNet 规则），并且文件需要为utf-8编码，扩展名为.txt。

Beats 下载

功能介绍

Elasticsearch 集群中辅助提供 ELK 生态服务，包括 Logstash、Kibana 以及提供开源 Beats 的下载和安装说明。Beats 集合了多种类型数据采集器。这些采集器安装后可用作轻量型代理，从成百上千或成千上万台机器向Logstash或Elasticsearch发送数据。TBDS-ES 平台提供了三种主要 Beats的下载，包括 Filebeat（收集文件）、Metricbeat（收集服务、系统的指标数据）、Winlogbeat（Windows 时间日志）。

采集器名称	简介	官网参考链接	环境
Filebeat	日志采集器，用于收集和传送日志文件	Metricbeat overview | Metricbeat Reference [7.10] | Elastic	支持 aarch64、x86_64
Metricbeat	指标采集器，输送系统和服务统计数据	Metricbeat overview | Metricbeat Reference [7.10] | Elastic	支持 aarch64、x86_64
Winlogbeat	Windows 事件日志的采集	Winlogbeat Overview | Winlogbeat Reference [7.10] | Elastic	支持 x86_64

> 注意：
>
> 平台仅提供 TBDS-ES 对应版本的 Beats 的下载和说明，用户需要自行安装到采集端并配置，并发送数据到 TBDS 平台部署的 Logstash 或者 ES 进行传输和存储，并在 Kibana 上做可视化分析展示。

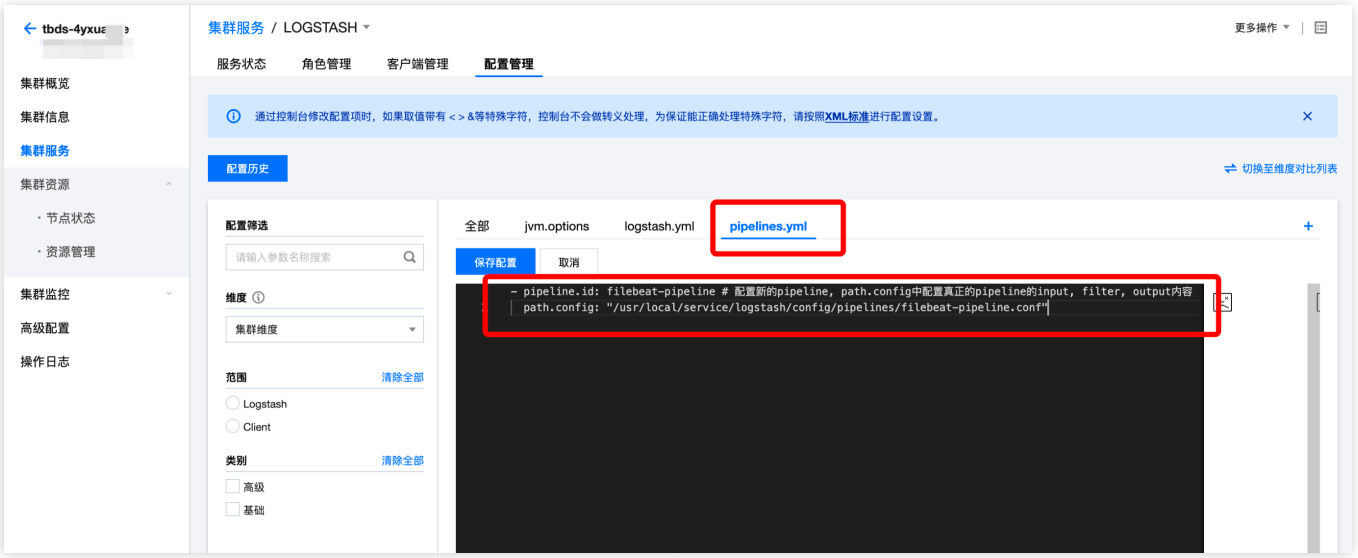
操作步骤

1. 在集群详情页中选择**高级配置 > Beats 下载，**可以看到以上介绍的不同采集器，单击 说明 即可下载浏览对应采集器的安装说明流程。
2. 下载对应环境的采集器部署包，并按照说明文档部署到相应环境。
3. 按照提供的操作说明、采集器官网参考链接进行配置，完成 Beats 的安装部署和配置。

ELK 最佳实践

介绍 Beats 将日志收集发送到 Logstash, 进行数据采集和处理后传输到 ES 存储，然后在Kibana进行展示的完整示例。
前置需要搭建 ES、Logstash、Kibana集群，配置部署好 Filebeat。

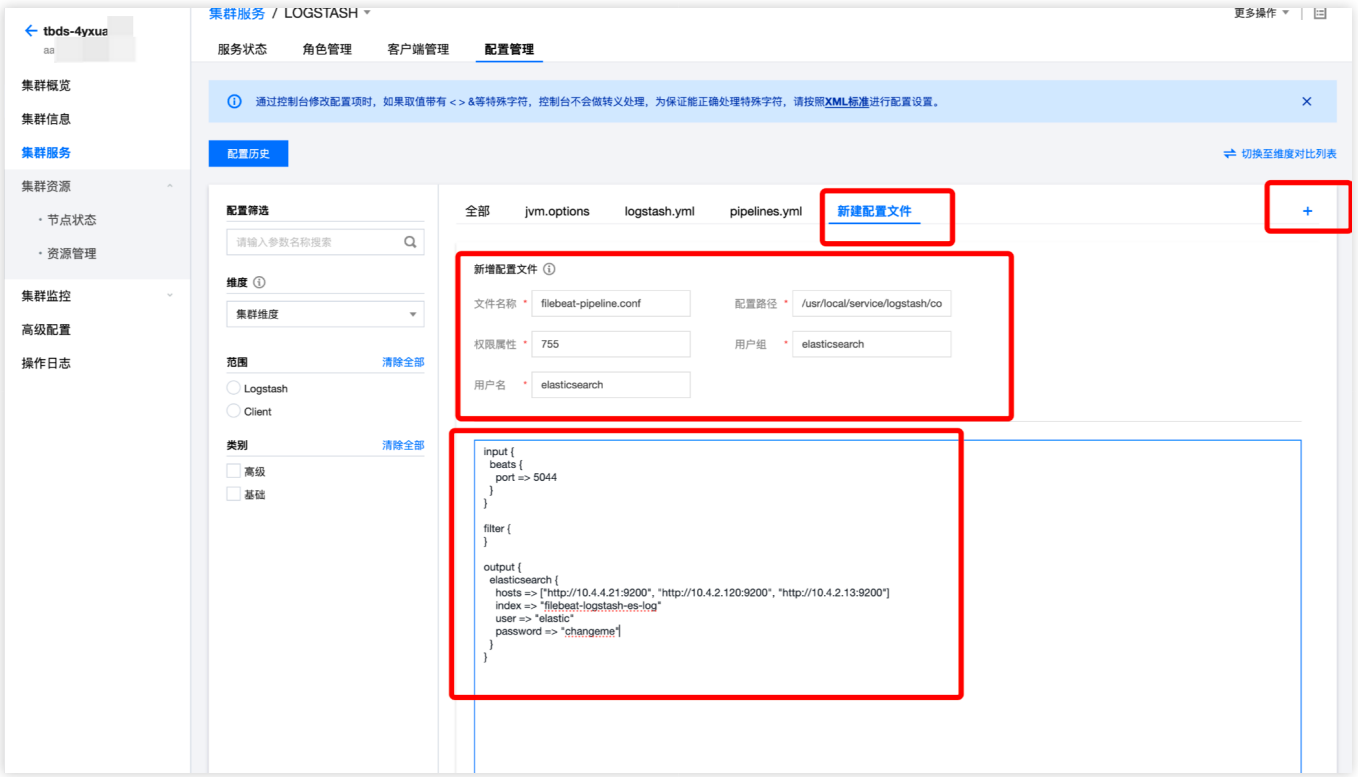
1. Logstash 配置
   在logstash的配置管理页面的pipelines.yml新增配置。



- pipeline.id: filebeat-pipeline # 配置新的pipeline
  path.config: "/usr/local/service/logstash/config/pipelines/filebeat-pipeline.conf" # path.config中配置真正pipeline的input, filter, output内容

> 注意：
>
> 该pipelines.yml文件后台中默认配置了一个alive-pipeline以维持logstash进程能一直在后台运行。

新增pipeline具体配置文件：



文件名称: filebeat-pipeline.conf # pipelines.yml中配置pipeline对应的path.config的最末尾文件名

配置路径: /usr/local/service/logstash/config/pipelines # pipelines.yml中配置pipeline对应的path.config对应的目录(不包括最后的文件名)

权限属性: 755 # 配置755即可

用户组: elasticsearch
用户名: elasticsearch

配置文件具体内容: input: 接收Filebeat输入的端口, filter: 处理的逻辑, output: 输出的es的地址

```
input {
  beats {
    port => 5044
  }
}

filter {
}

output {
  elasticsearch {
    hosts => ["http://10.4.4.21:9200", "http://10.4.2.120:9200", "http://10.4.2.13:9200"] # es的地址
    index => "filebeat-logstash-es-log" # 输出到的索引
    user => "elastic"  # 对该索引有读权限的es用户
    password => "changeme" # 密码
  }
}
```

重启所有的logstash。

2. Filebeat 配置

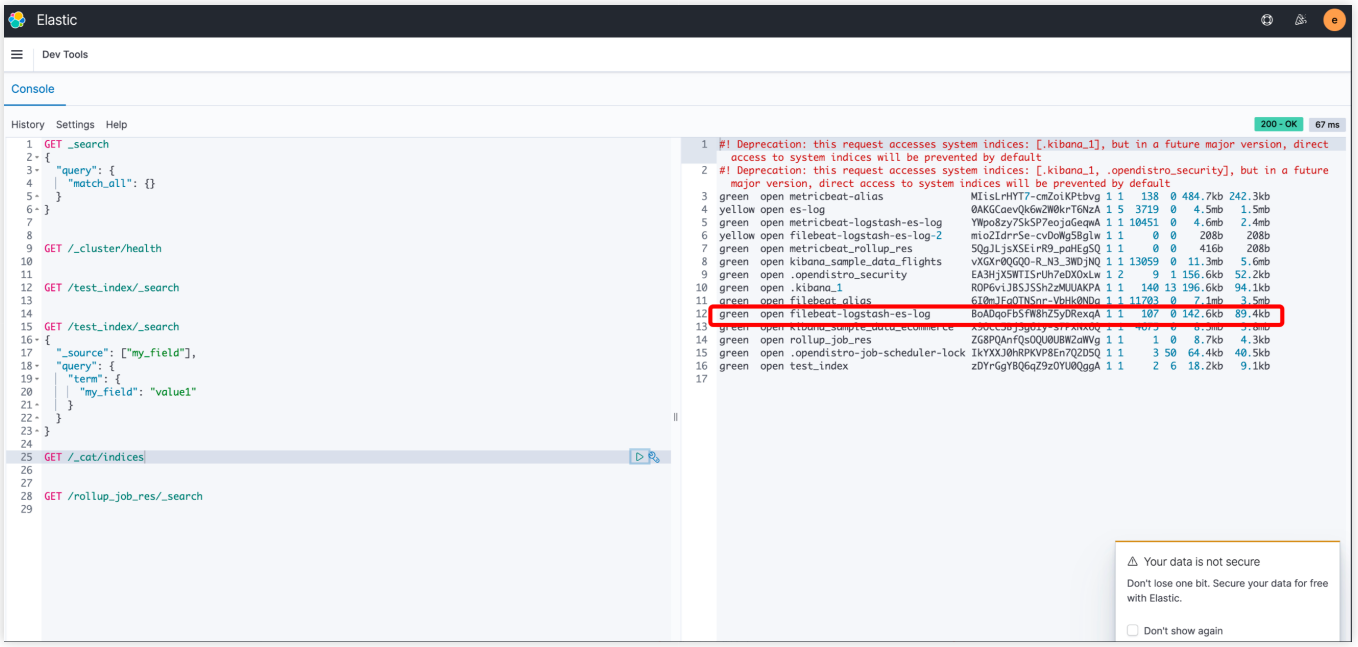
将对应操作系统的安装包和说明文档下载下来。

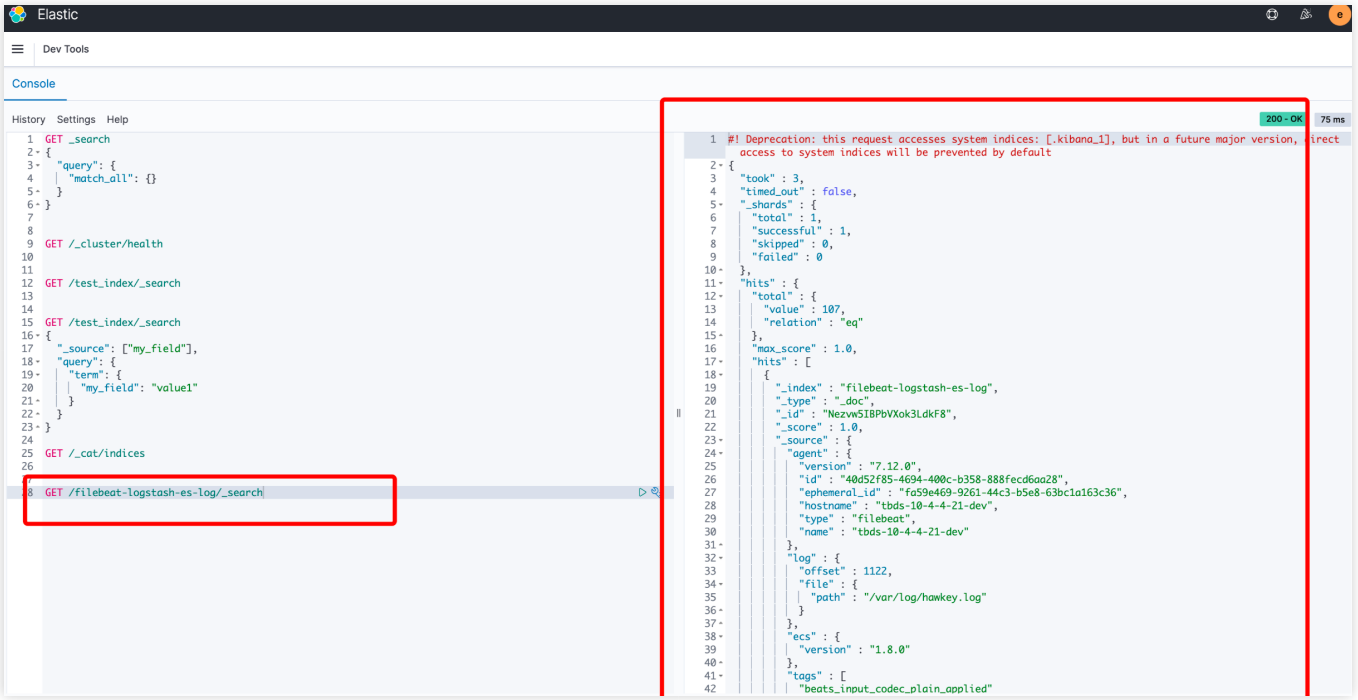
参考说明对Filebeat进行部署配置, 其中 filebeat.yml 配置项中的输出源设置为logstash。

```
output.logstash:
  hosts: ["node1:5044", "node2:5044", "node3:5044"]
```

3. Kibana 配置

可以在Kibana 的图表页面编辑数据的可视化展示，本例仅在Dev Tools 处展示查看对应索引是否创建, 其中是否有数据。




Elastic
Dev Tools
Console
History Settings Help
GET _search
{
"query": {
"match_all": {}
}
}
GET /_cluster/health
GET /test_index/_search
GET /test_index/_search
{
"_source": ["my_field"],
"query": {
"term": {
"my_field": "value1"
}
}
}
GET /_cat/indices
GET /filebeat-logstash-es-log/_search
200 - OK
75 ms
#! Deprecation: this request accesses system indices: [.kibana_1], but in a future major version, direct access to system indices will be prevented by default
{
"took" : 3,
"timed_out" : false,
"_shards" : {
"total" : 1,
"successful" : 1,
"skipped" : 0,
"failed" : 0
},
"hits" : {
"total" : {
"value" : 107,
"relation" : "eq"
},
"max_score" : 1.0,
"hits" : [
{
"_index" : "filebeat-logstash-es-log",
"_type" : "_doc",
"_id" : "NezvwSIBPbVXok3LdkF8",
"_score" : 1.0,
"_source" : {
"agent" : {
"version" : "7.12.0",
"id" : "40d52f85-4694-400c-b358-888fecd6aa28",
"ephemeral_id" : "fa59e469-9261-44c3-b5e8-63bc1a163c36",
"hostname" : "tbds-10-4-4-21-dev",
"type" : "filebeat",
"name" : "tbds-10-4-4-21-dev"
},
"log" : {
"offset" : 1122,
"file" : {
"path" : "/var/log/hawkey.log"
}
},
"ecs" : {
"version" : "1.8.0"
},
"tags" : [
"beats_input_codec_plain_applied"

数据管理
数据管理概述

数据管理概述

数据管理为用户提供了一个统一视图，涵盖了 TBDS 平台上的所有数据。通过这一视图，用户可以集中管理和查看结构化的数据库表，以及半结构化或非结构化的文件，核心能力包括：

- 集中化元数据浏览。
- 数据目录及库表管理。
- 文件管理。
- 表优化配置和监控。

注册机制

TBDS 为经典集群上的 Hive/Iceberg（共享一个 Hive Metastore）和 HDFS 提供了集中的数据管理入口。当经典集群安装后，会自动将自身注册到数据管理服务中，相关操作机制说明如下：

- 安装经典集群/加装组件：
  - 包含Hive：自动将当前经典集群的 Hive/Iceberg 数据目录注册到数据管理服务中。
  - 包含HDFS：自动将当前经典集群 HDFS 目录注册到数据管理服务中。
- 卸载经典集群/卸载组件：
  - 包含Hive：从数据管理服务中自动移除当前集群的 Hive/Iceberg 数据目录。
  - 包含HDFS：从数据管理服务中自动移除当前集群的 HDFS 文件目录。

库表管理
数据目录管理

功能介绍

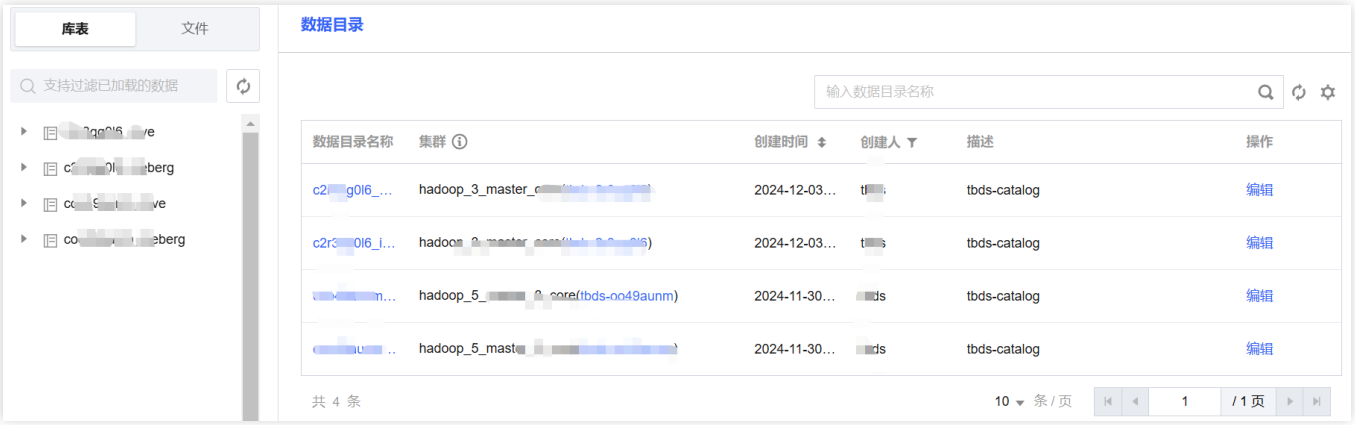
TBDS 支持集中查看和编辑经典集群中已经注册的数据目录，数据目录当前仅支持编辑目录描述。

前提条件

- 已完成 TBDS Manager 安装。
- 已完成公共集群安装。
- 完成经典集群安装，集群需要包含 Hive 服务。

操作步骤

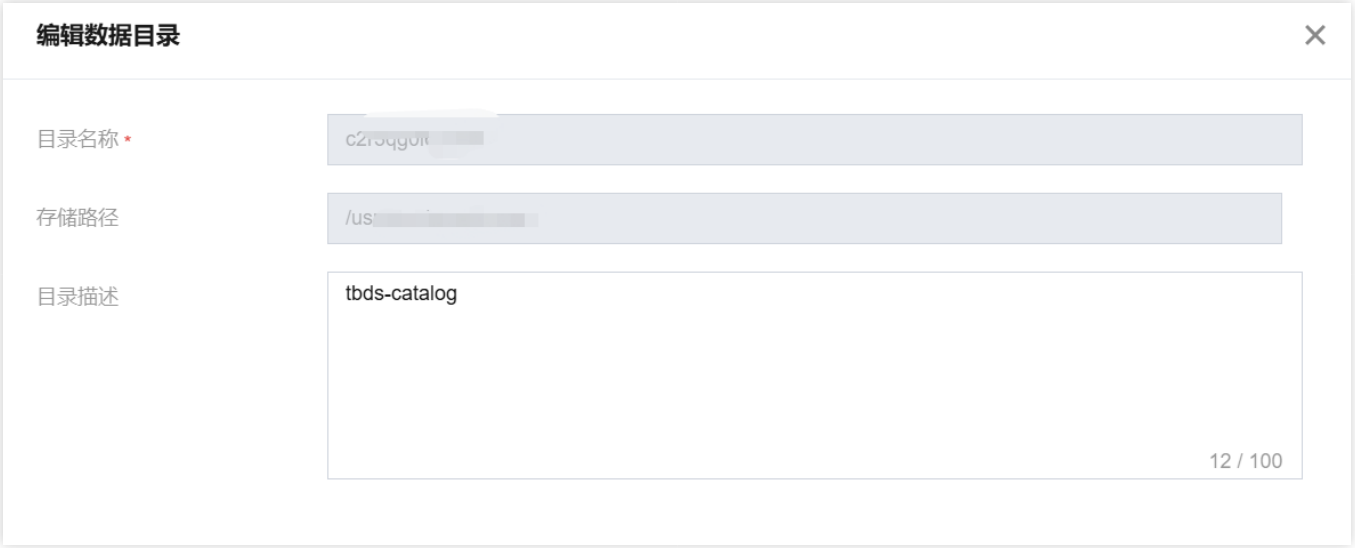
1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 数据管理，默认左侧导航会按照库表层级结构展示库表信息，点击数据目录、数据库和数据表，右侧会展示对应信息。



信息	详情
数据目录名称	标识一个经典集群 Hive/Iceberg 元数据，每个经典集群会自动生成一个 hive 和 iceberg 的数据目录，命名规则是集群数据目录ID_表类型。
集群	仅对经典集群有效，表示数据目录所在的经典集群。
创建时间	当前数据目录创建时间。

信息	详情
创建人	当前数据目录创建人。
描述	当前数据目录备注信息。
搜索	支持按关键词模糊搜索数据目录。

2. 数据目录编辑：库表管理右侧页面中选择需要编辑的数据目录，并点击“编辑”，在右侧弹出页面中可修改目录描述信息。



3. 操作完成。

数据库管理

功能介绍

TBDS支持在数据管理页面完成数据库的创建、编辑和删除，方便用户对平台上的数据库进行便捷管理。

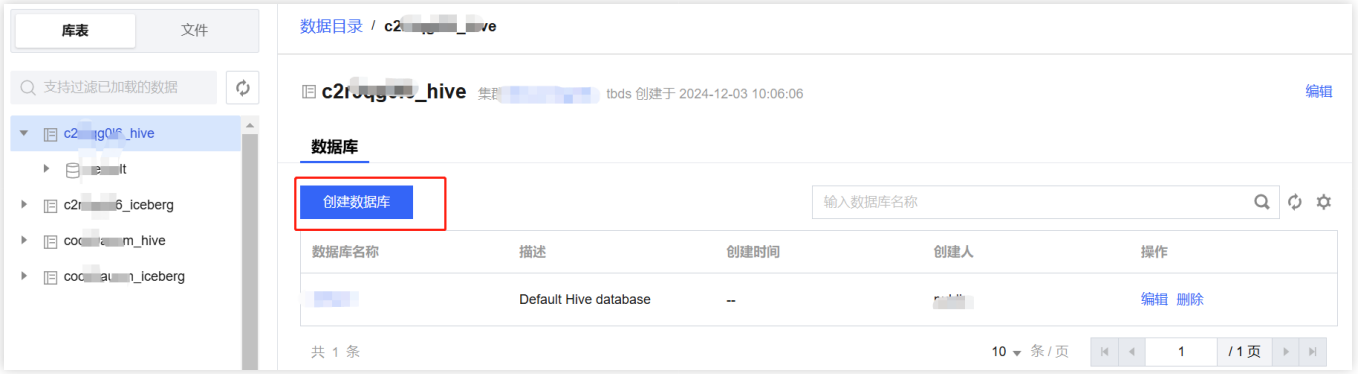
前提条件

- 已完成 TBDS Manager 安装。
- 已完成公共集群安装。
- 完成经典集群安装，集群需要包含 Hive 服务。

操作步骤

1. 以管理员用户登录TBDS，进入 TBDS Manager首页 > 数据管理。
2. 创建数据库：

- 在数据管理-库表下选定数据目录，在右侧页面点击“创建数据库”。



- 在创建数据库页面输入数据库相关信息，点击确定后完成数据库创建。

信息	详情
数据库名称	必填，数据目录下不允许存在重名的数据库
存储路径	可选，默认为/usr/hive/warehouse，可通过修改Hive的hive.metastore.warehouse.dir配置值来调整。
描述	可选，库描述，不超过100个字符

- 创建完成后可在列表中查看。

3. 编辑数据库：

- 在数据管理-库表下选定数据目录，在右侧数据库列表中选择需要编辑的数据库并点击“编辑”，仅支持编辑库描述。

- 编辑完成后点击确定保存。

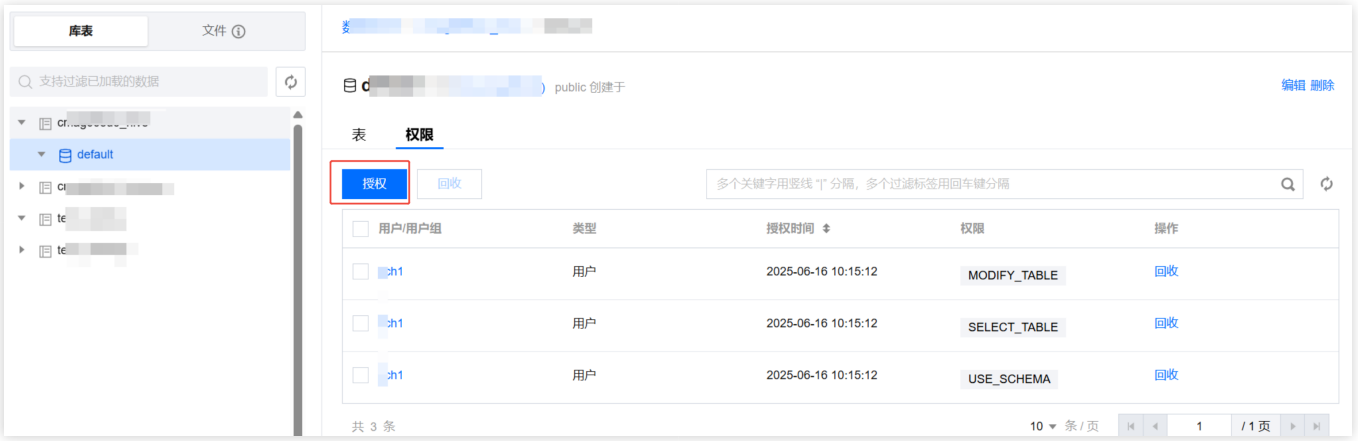
4. 删除数据库：

- 在数据管理-库表下选定数据目录，在右侧数据库列表中选择需要删除的数据库并点击“删除”，当数据库下存在表时不允许删除。

- 在删除确认弹框中点击“删除”完成数据库删除。

5. 库授权和回收：

- 点击数据库名称进入库详情，并切换到权限选项卡，点击“授权”按钮。



- 在授权弹框中选择需要授权的用户/用户组和权限，然后点击确认完成授权操作。

权限点说明：

- 库级权限

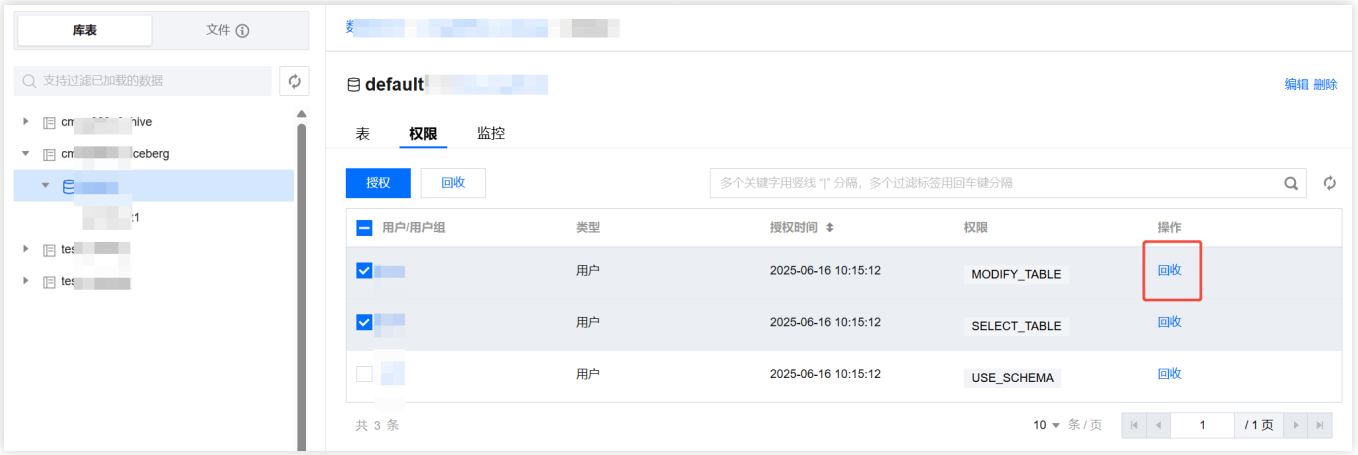
权限点	描述
访问目录	表示授予访问库目录的权限
创建表	表示授予当前库下创建表的权限

修改库	表示授予修改库的权限
库管理	表示授予管理库的权限，包括删除库、修改表和创建表

- 表级权限

权限点	描述
查询表	表示授予当前库下所有表字段的查询权限
编辑表	表示授予当前库下所有表更新、插入和查询权限
修改表	表示授予当前库下所有表编辑、删除记录、更新、插入和查询权限

- 在授权记录中，选择需要回收权限的用户或用户组，点击“回收”。



- 在回收弹框中，请检查并确定回收库的信息，确认无误后请点击“回收”按钮来执行库的回收操作。

回收限
确定回收用户/用户组的权限吗？回收后将失去相关权限！ 收起
用户/用户组
类型
权限
用户
USE_SCHEMA
回收
取消

数据表管理

功能介绍

TBDS支持在数据管理页面完成数据表的创建、编辑和删除，方便用户对平台上的数据表进行便捷管理。

前提条件

- 已完成 TBDS Manager 安装。
- 已完成公共集群安装。
- 完成经典集群安装，集群需要包含 Hive 服务。

操作步骤

1. 以管理员用户登录TBDS，进入 TBDS Manager首页 > 数据管理。
2. 创建数据表：

- 在数据管理-库表下选定数据库，在右侧页面点击“创建数据表”。

- 创建数据表分两个步骤，第一步输入表基本信息：

信息	描述
表名称	必填，一个数据库下不允许表重名。
描述	可选，表描述，不超过 100 个字符。

- 第二步配置表（以下为Iceberg表举例）：

表字段：至少需要一个字段。

信息	描述
字段名	字段名称，支持字母、数字和下划线。
字段类型	支持的字段类型，包括 BOOLEAN,INT,LONG,FLOAT,DOUBLE,DECIMAL(P,S),DATE,TIME,TIMESTAMP,TIMESTAMPTZ,STRING,FIXED(L),BINARY,UUID。
字段配置	仅针对 Fixed 和 Decimal 类型有效： Fixed: 表示固定长度，取值 >= 1。 Decimal: 表示精度，输入格式为P,S，取值：1 <= P <= 38，0 <= S <= P，比如 2,2。

表分区：开启后，至少需要一个分区。

信息	描述
分区字段	Iceberg表仅支持从字段名中选择。

分区转换配置	用于创建隐藏分区的转换表达式。这些转换表达式可以在PARTITIONED BY子句中使用，用于创建隐藏分区。Iceberg支持多种转换表达式，包括year(ts)，month(ts)，day(ts)，hour(ts)，bucket(N, col)，truncate(L, col)等。
分区转换参数	仅适用于 bucket 和 truncate，对应转换函数中参数（L或 N）。

表属性：开启后，至少需要一个属性。

信息	描述
属性名	可填写 Iceberg 表属性名称，即表的TBLPROPERTIES指定，比如write.format.default。
属性值	属性取值，比如Parquet。

表优化：参考表优化配置

- 点击完成按钮完成表的创建。

3. 编辑数据表：

- 在数据管理-库表下选定数据库，在右侧数据表列表中选择需要编辑的数据表并点“编辑”，支持编辑表的描述、属性和优化方式。

- 编辑完成后点击确定完成，相关信息填写要求参考创建表。

- 编辑完成。

4. 删除数据表：

- 在数据管理-库表下选定数据库，在右侧数据表列表中选择需要删除的数据表并点“删除”。

- 在删除确认弹框中点击“删除”完成数据表删除。

5. 查看表详情：支持查看表的基本信息和表属性。

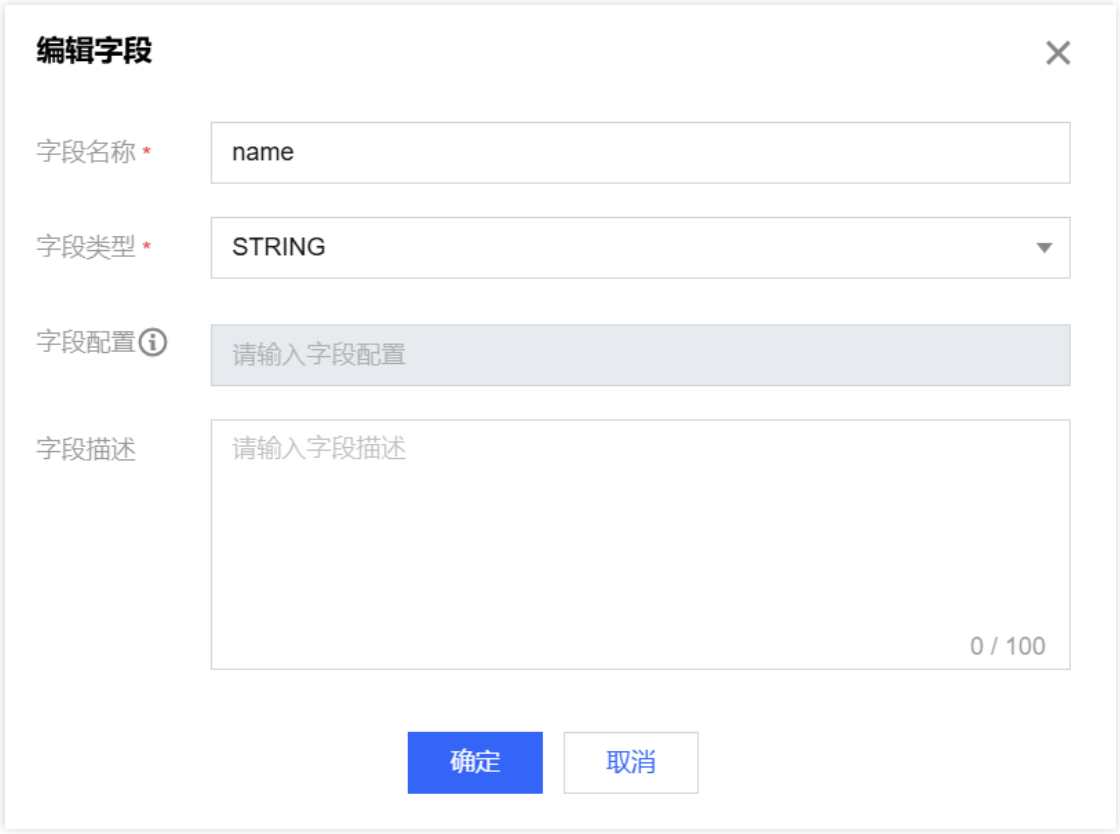
6. 查看表字段和分区：支持查看表的所有字段信息和分区信息。

7. 添加字段：在“字段和分区”选项下，您只需点击“添加字段”按钮，即可轻松添加一个或多个字段。完成添加后，点击“确定”即可完成整个添加过程。

8. 编辑字段：

- 在“字段和分区”选项卡下找到需要编辑的字段，点击编辑：

- 在字段编辑对话框中，您可以根据需要对字段进行相应的修改。目前，您可以调整字段类型、字段配置以及字段描述。请注意，更改字段类型可能会受到兼容性的限制。因此，在修改字段类型时，请务必参照您的表格类型约束，确保修改后的字段类型与表格结构兼容。

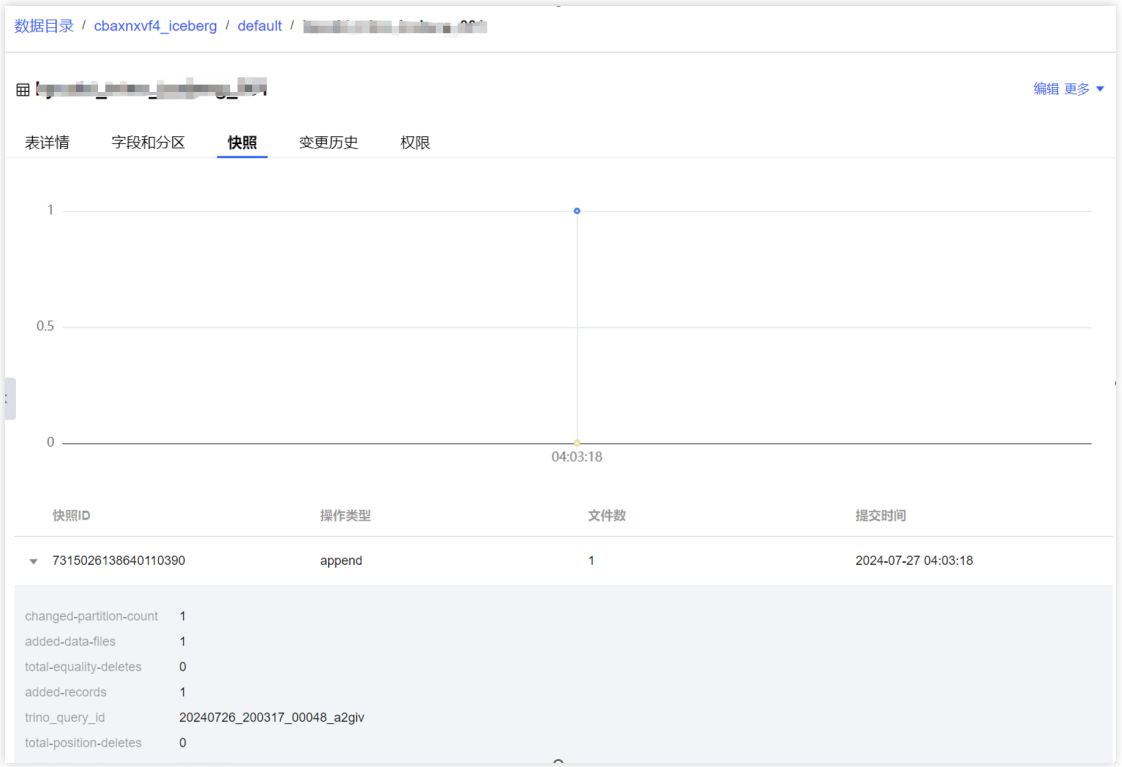


9. 删除字段：

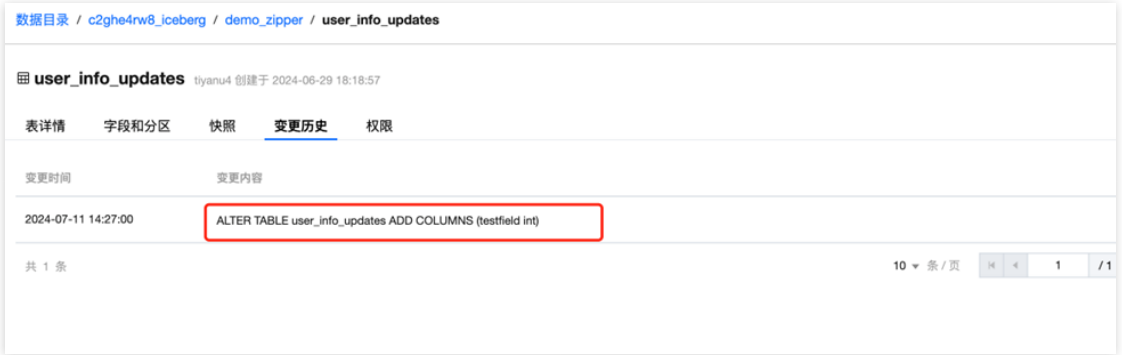
- 在“字段和分区”选项卡下找到需要删除的字段，点击删除，注意：目前仅允许按照表中字段的顺序，从最后一个字段开始，依次向前进行删除。

- 在确认弹框中点击“确认”按钮，即可完成字段的删除操作。

10. 查看表快照：对于Iceberg表，您可以查看其快照信息。



11. 查看表变更历史：对于Iceberg表，您可以查看其表变更信息，包括表字段、分区和属性等。



12. 表授权和回收：

- 为用户和用户组授予当前表的权限，点击“授权”按钮。

- 在授权弹框中选择需要授权的用户/用户组和权限，然后点击确认完成授权操作。

权限点说明：

信息	详情
查询表	表示授予当前表所有字段的查询权限
修改表	表示授予当前表编辑、删除记录、更新、插入和查询权限
所有表	表示授予当前表删除、编辑、删除记录、更新、插入和查询权限

- 在授权记录中，选择需要回收权限的用户或用户组，点击“回收”。

- 在回收弹框中，请检查并确认回收表的信息，确认无误后请点击"回收"按钮来执行表的回收操作。

表优化配置

> 说明：
>
> 仅针对湖仓版，支持表优化配置功能。

前提条件

- 完成经典集群安装。

操作约束

- 仅支持对Iceberg表配置表优化。

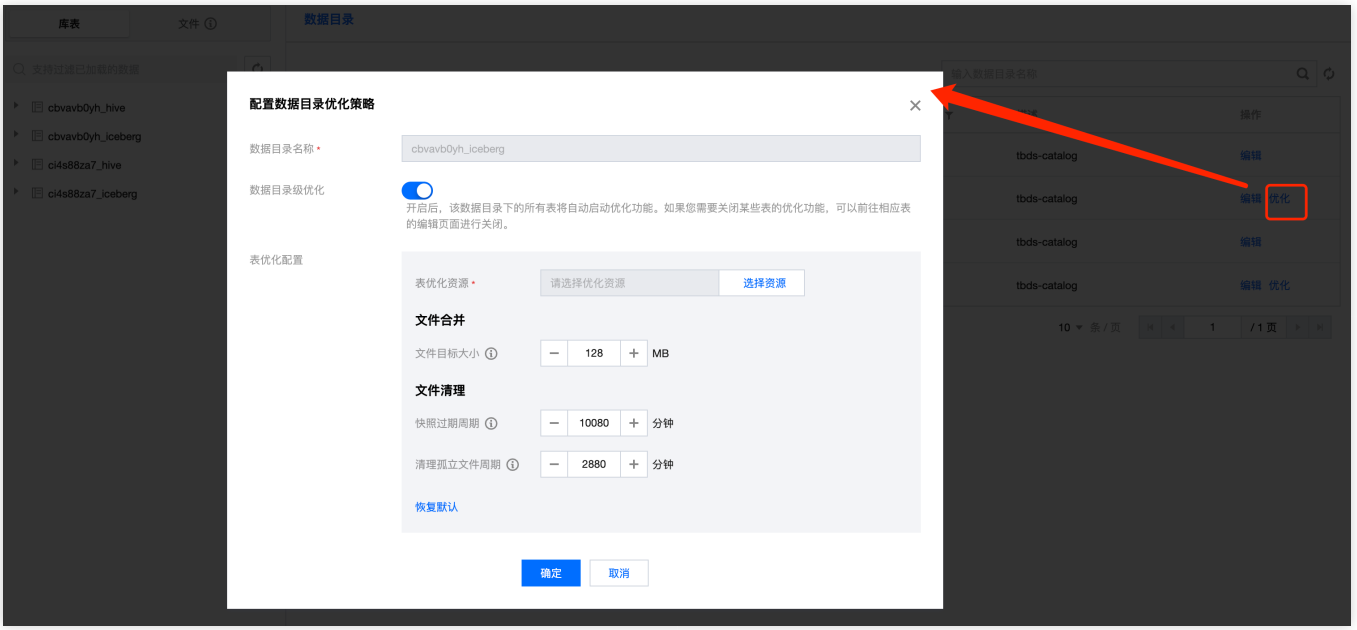
操作场景

针对 Iceberg 表，TBDS平台默认会进行自动优化。然而，如果某些表的优化速度较慢或效果不佳，可以进行针对性的优化，表优化支持数据目录和表两种级别配置，当两者同时存在某个优化配置时，表级配置优先级更高。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager首页 > 数据管理。
2. 数据目录级表优化：

- 在数据目录列表中找到需要调整的数据目录名称，右侧点击编辑链接：



- 在编辑数据目录页面下方找到数据优化，点击自定义，按需调整优化参数：

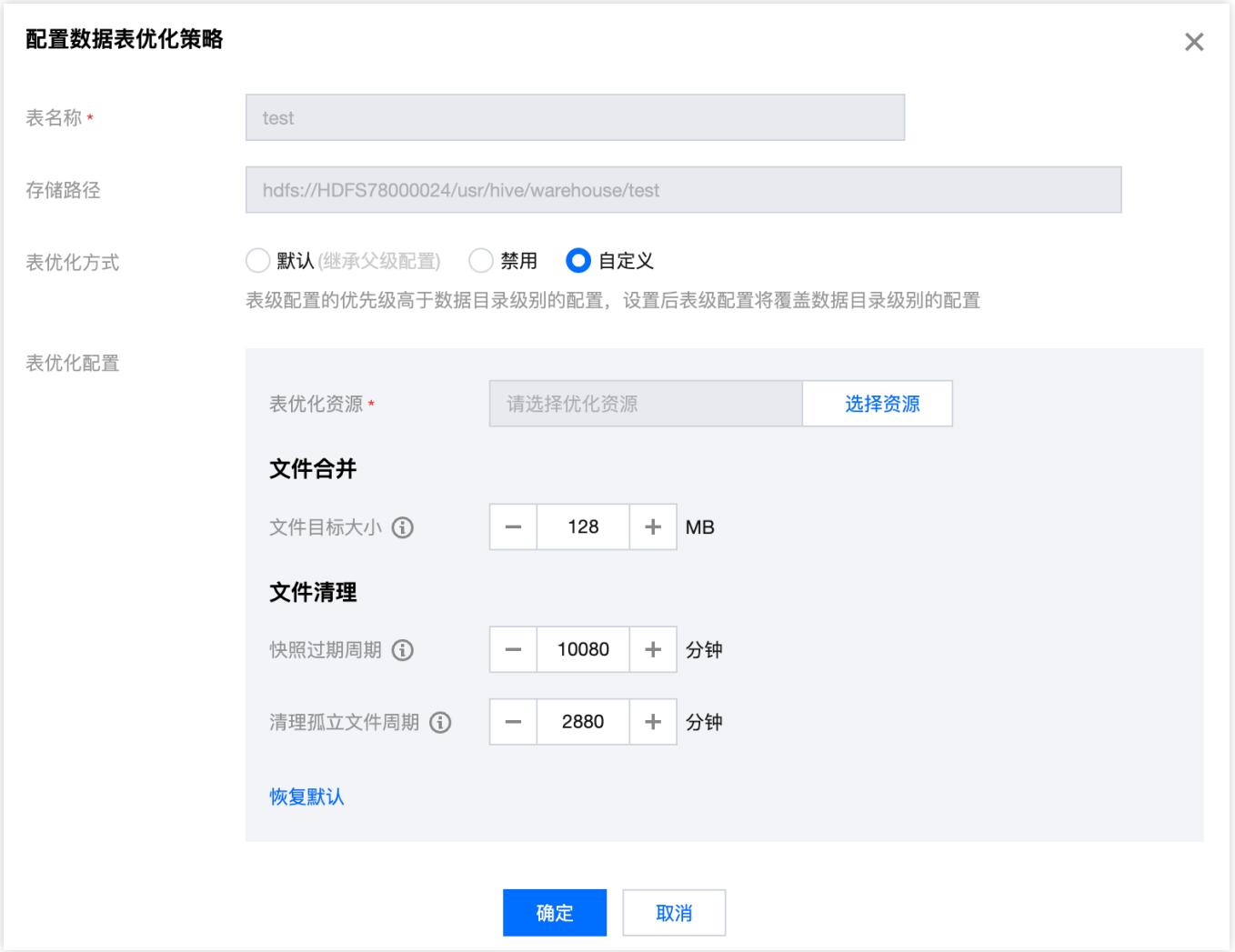
信息	详情
表优化资源	可以选择优化任务运行的YARN资源队列或虚拟集群。
文件目标大小	文件优化合并时，会尽可能将文件合并成目标大小，建议取值128M。
快照过期周期	快照存在时间超过该值时，平台会将该快照标记为过期的快照。快照过期时间取值越长，快照清理的速度越慢，占用存储空间越多。
清理孤立文件周期	平台会周期性扫描并清理孤立文件。执行周期越短，清理孤立文件会更灵敏，但是可能消耗更多资源。

- 点击【数据目录】 -> 【监控】，可以监控表优化任务列表信息。

3. 表级表优化：

- 在表列表中的操作列表，点击【优化】可针对表配置优化策略。

- 通过点击表优化方式相关配置完成表优化修改。



信息	详情
表优化方式	默认：配置会继承使用数据目录级别的配置。 禁用：允许禁用优化功能，系统后台将不再自动执行表优化操作。 自定义：允许用户独立设置表的优化参数，会覆盖目录级的优化参数。

- 操作完成。

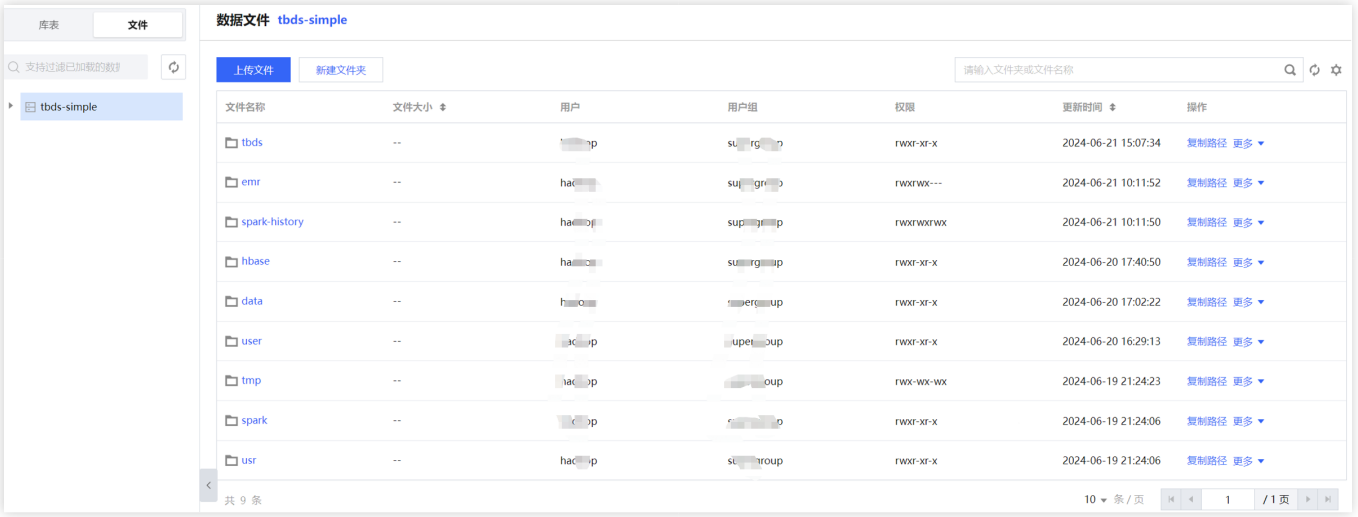
文件管理

前提条件

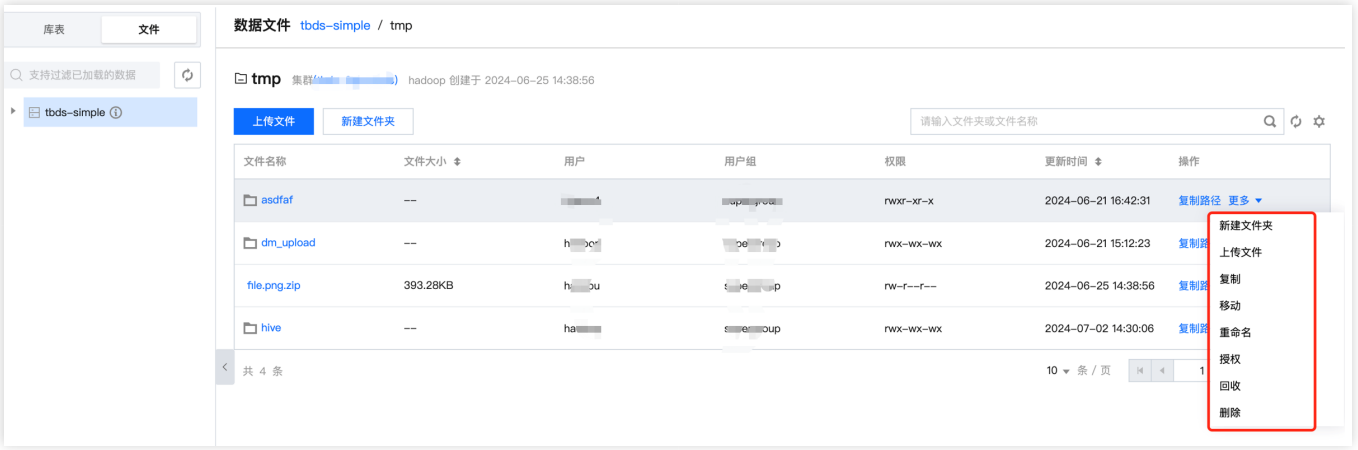
- 完成经典集群安装，集群需要包含 HDFS 服务。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 数据管理，左侧点击“文件”切换到文件管理视图：



2. 文件夹管理：点击左侧文件夹，右侧将展示文件夹下的文件和子文件夹，可对文件夹进行复制路径、复制、重命名、移动、授权、回收、删除等操作。



3. 操作说明如下：

操作名称	描述
复制路径	复制文件或文件夹的全路径
新建文件夹	在当前文件夹下新建子文件夹
上传文件	将本地文件上传到 HDFS 目录下，上传文件大小最大支持 100MB
复制	将选中的文件或文件夹复制到其他目录下
移动	将选中的文件或文件夹移动到其他目录下
重命名	更改文件或文件夹的名称
授权	为用户授予访问权限（基于 Ranger ）
回收	回收用户访问权限（基于 Ranger ）
删除	删除文件或文件夹，当 HDFS 开启回收站时会删除到回收站。

4. 文件管理：点击左侧文件，右侧将展示文件预览信息，支持按照文本和二进制查看。

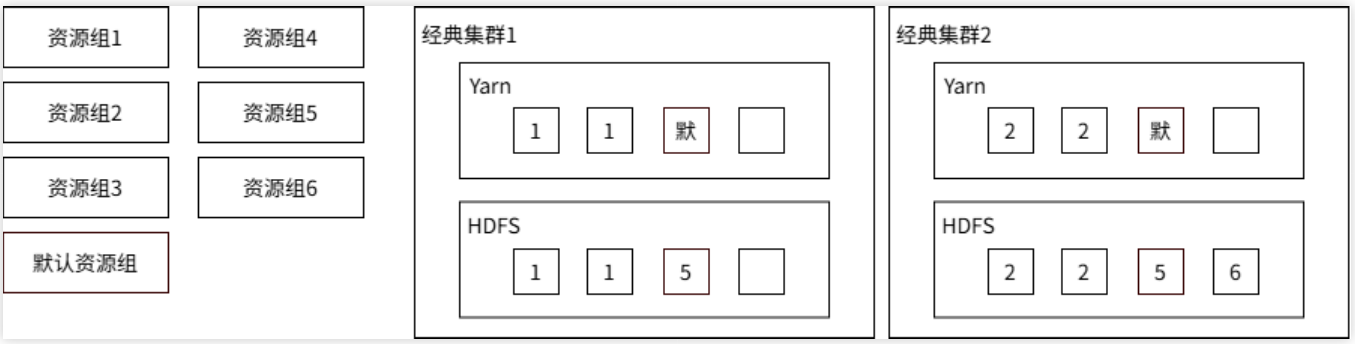
5. 操作完成。

资源管理
资源组概述

资源组概述

资源组是对 TBDS 平台上 Hadoop 资源进行逻辑划分的单元，由存储和计算资源组成。每个资源组至少包含一个存储资源或一个计算资源，设置约束如下：

- 资源组是由存储和计算资源构成的集合，每个资源组至少需要包含一个存储资源或一个计算资源。
- 如果资源组中包含了 YARN 队列，那么其所包含的存储资源必须与 YARN 队列位于同一集群。
  资源组划分模型举例如下：

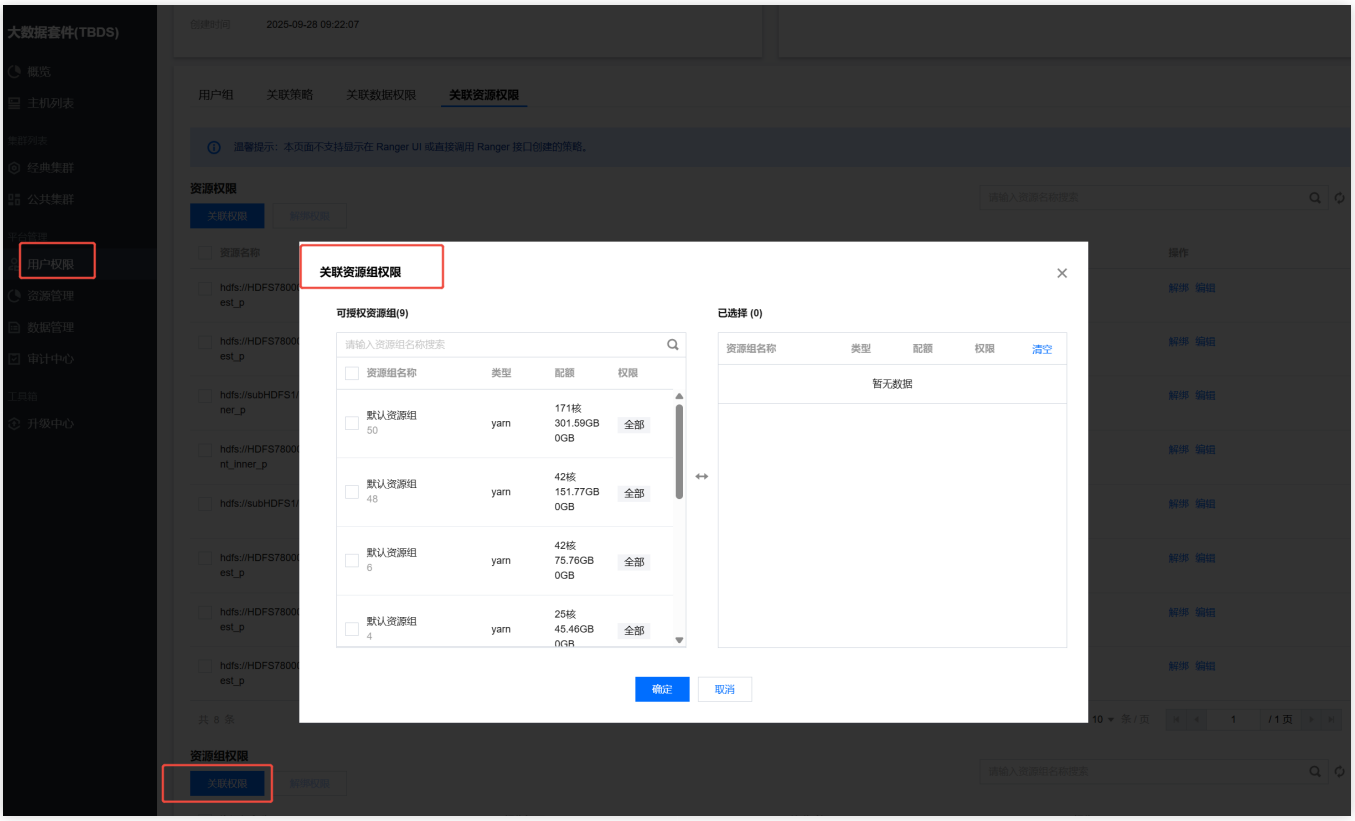


考虑到YARN独特的机制，资源组在创建时的计算配额分配方法存在差异：

- YARN资源组：通过与YARN队列的关联来实现计算资源的分配。为此，需要事先在YARN的调度配置中完成队列的设定。

资源组的用途说明如下：

- 实现团队间资源隔离：企业可以为团队分配一个或多个资源组。在每个资源组内，可以关联多个 YARN 队列，资源组下 YARN 队列可以通过配置实现资源的共享或独占，从而帮助企业更有效地管理团队或应用之间的资源竞争问题。
- 实现基于资源组授权：在用户权限-用户详情-关联资源权限中支持为用户分配资源组权限，请注意：一旦用户被授予资源组权限，他们将自动获得该资源组内所有资源的访问权限，而无法进行更细粒度的权限控制。



- 统一开放 API：提供统一的资源组开放 API，便于上层工具灵活对接，满足其资源隔离需求。

默认资源组

平台部署完成后，为了确保用户能够迅速上手，TBDS 平台预置了一套默认资源组。用户无需进行额外配置，即可实现即开即用的便捷体验，YARN 默认资源组是指每个 YARN 集群都包含一个默认的资源组，该资源组关联了 YARN 集群的 default 队列。

默认资源组

平台部署完成后，为了确保平台可正常使用并让用户能够迅速上手，TBDS预置了一套默认资源组，用户无需进行额外配置，即可实现即开即用的便捷体验。默认资源组不支持删除，定义如下：

- YARN 默认资源组：每个 YARN 集群都包含一个默认的资源组，该资源组关联了YARN 集群的 default 队列。

创建 YARN 资源组

前提条件

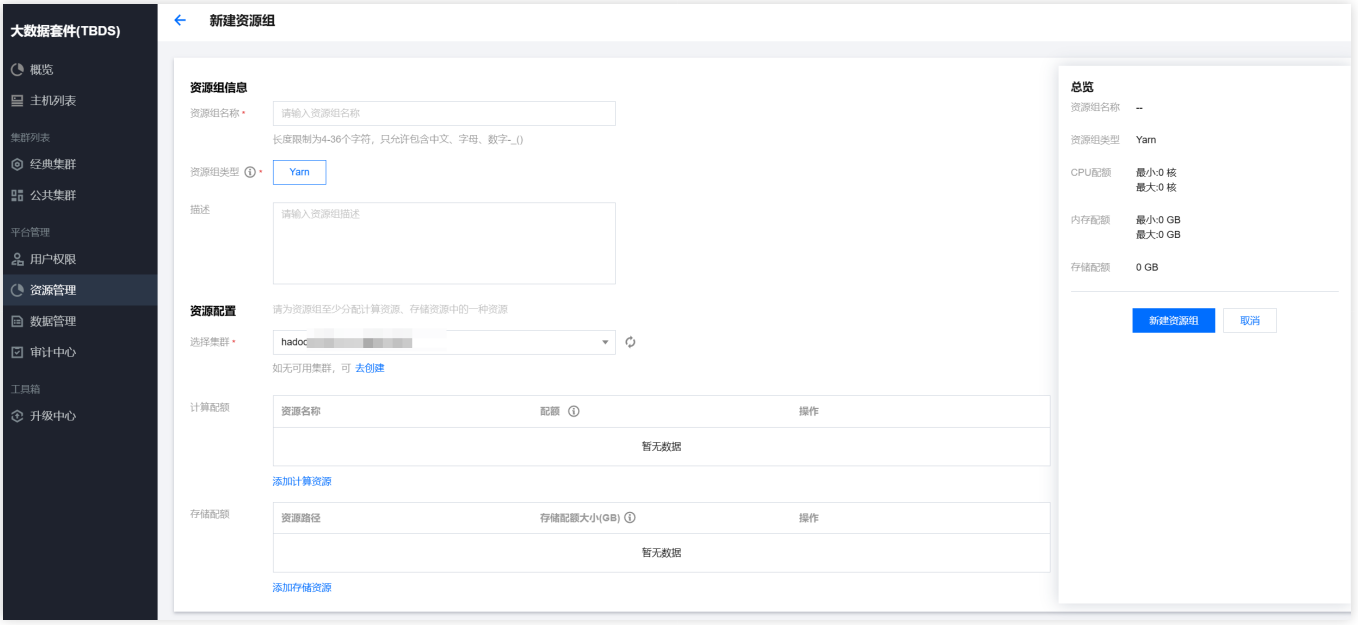
- 已完成经典集群安装，包含 YARN 组件。

操作场景

TBDS 平台默认会为用户生成经典集群的默认资源组。用户在安装完公共集群后，这些默认资源组会自动创建。然而，对于生产环境来说，默认资源组可能无法完全满足用户的业务需求。在这种情况下，用户可以根据具体的业务要求，自定义资源组，以实现更精细的资源使用控制。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 资源管理，点击新建资源组，按照要求填写信息并确认：

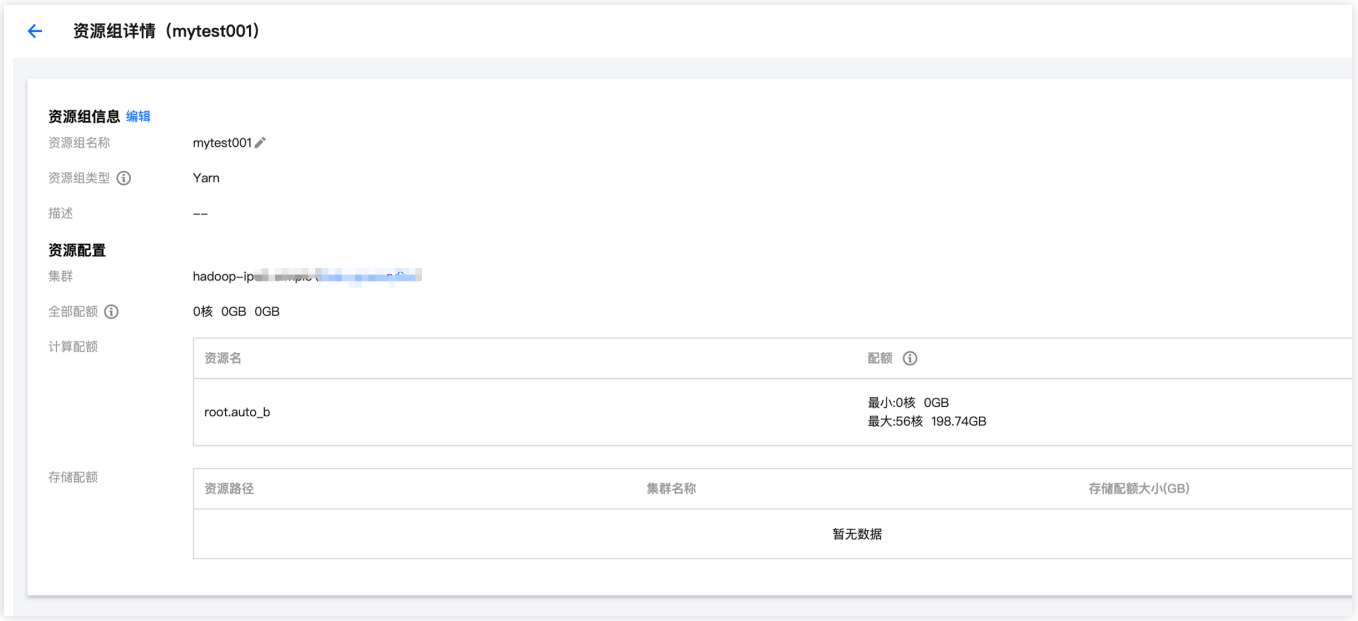


信息	详情
资源组名称	支持名称重复，长度限制为4-36个字符，只允许包含中文、字母、数字-_()。

信息	详情
资源组类型	支持 YARN 计算资源。
描述	资源组描述。
选择集群	选择需要分配资源的经典集群，该集群需要包含 YARN 组件。
计算配额	通过将 YARN 的队列绑定到该资源组下实现计算配额的分配，其中 YARN 的计算配额根据 YARN 队列的配置进行计算。在 Fair 调度模式下，最小值取自 minResource，最大值取自 maxResource。而在 Capacity 调度模式下，最小值取自 Effective Capacity，最大值则取自 Effective Max Capacity。
存储配额	通过将该经典集群的 HDFS 存储路径绑定到该资源组下实现存储配额的分配，支持设置存储路径的存储配额大小。
存储配额大小(GB)	表示当前存储路径所允许的最大存储容量，并受集群整体配额的限制。请注意，多路径的配额总和有可能超出集群的最大配额，如存储超过配额则会阻断数据写入。

2. 填写完成后点右侧新建资源组完成资源组创建：

3. 点击资源名称可查看 YARN 资源组详情：


资源组详情（mytest001）
资源组信息 编辑
资源组名称
mytest001
资源组类型
Yarn
描述
--
资源配置
集群
全部配额
0核 0GB 0GB
计算配额
资源名
配额
root.auto_b
最小:0核 0GB
最大:56核 198.74GB
存储配额
资源路径
集群名称
存储配额大小(GB)
暂无数据

4. 操作完成。

编辑 YARN 资源组

前提条件

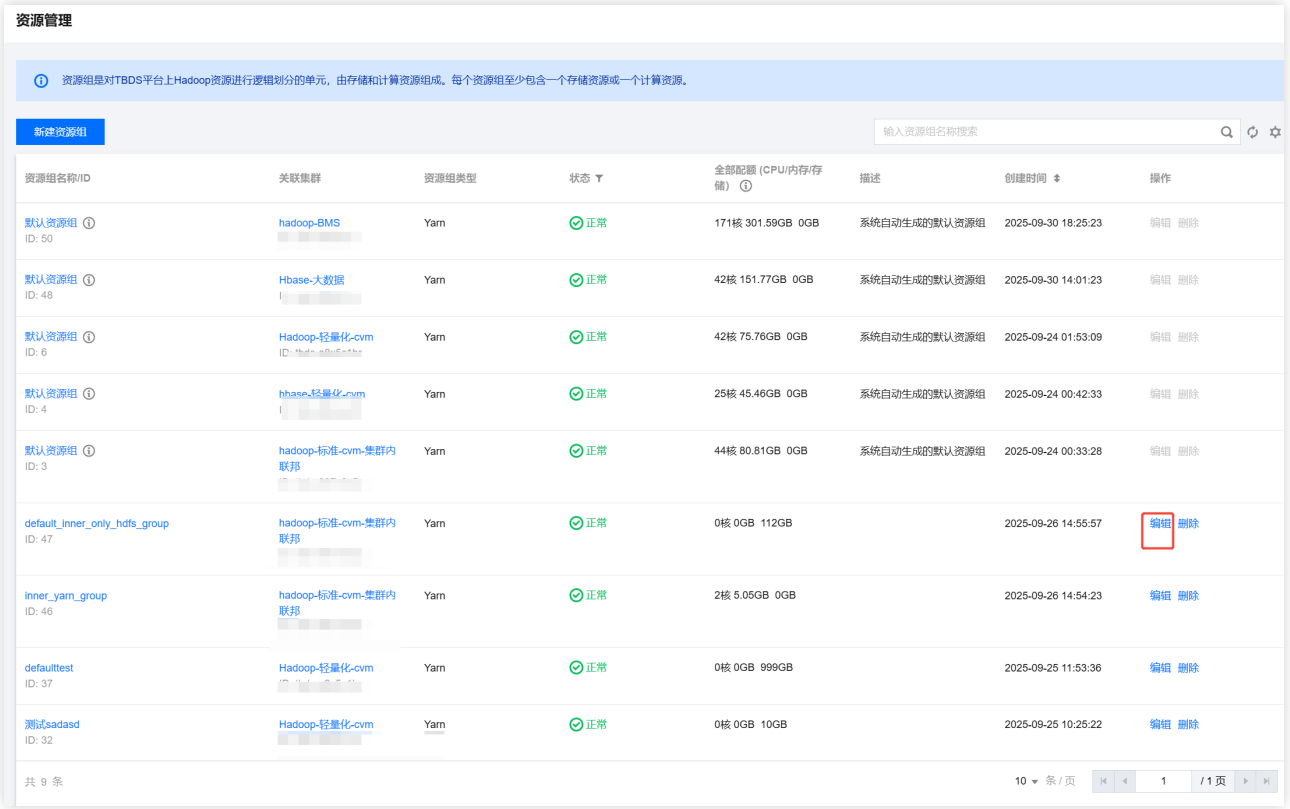
已存在 YARN 资源组。

操作场景

TBDS 支持对资源组的名称、描述和配额进行按需调整，方便用户根据实际业务需求灵活管理资源。

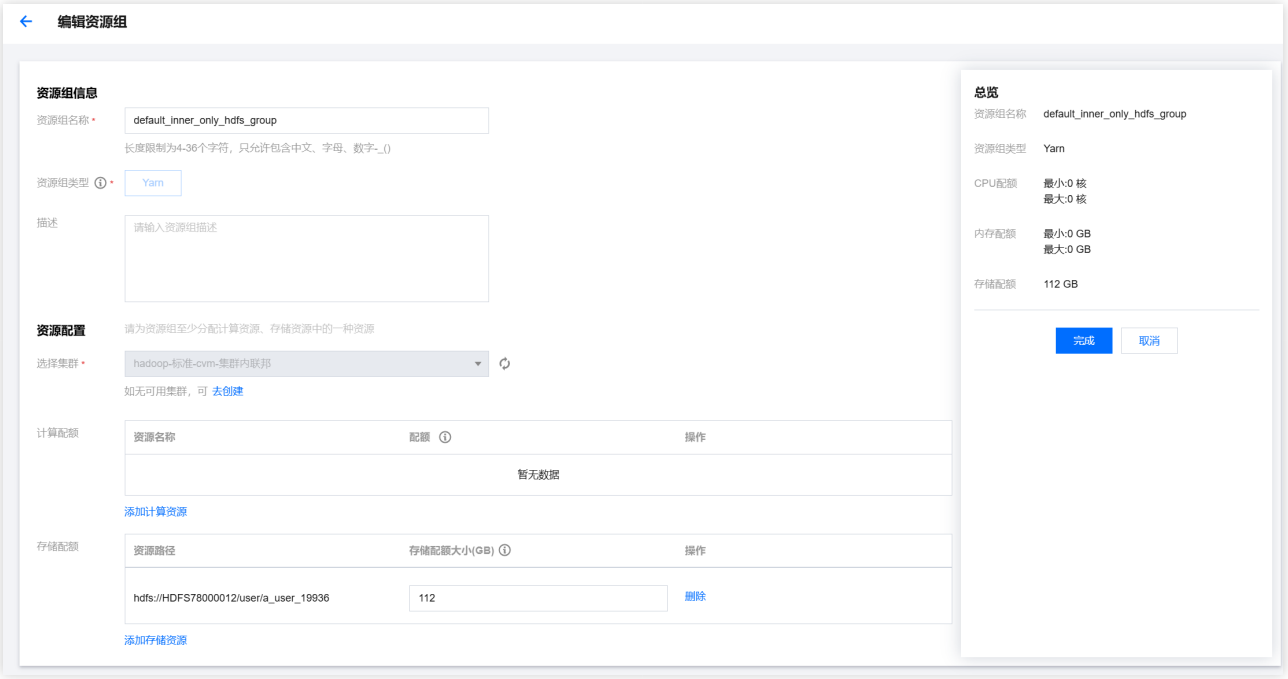
操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 资源管理，在资源组列表中找到需要编辑的资源组，在操作中单击编辑。



2. 进入编辑资源组页面可对资源组名称、描述和资源进行修改，资源配置修改约束说明如下：
    - 资源组关联的标准集群不允许修改。

- 修改计算资源配额时，需要确保业务未使用。
- 在调整或更新存储资源配额时，我们确保所有变更均在不影响业务运行的前提下进行。



3. 单击确定，即可完成资源组的编辑操作。

删除 YARN 资源组

前提条件

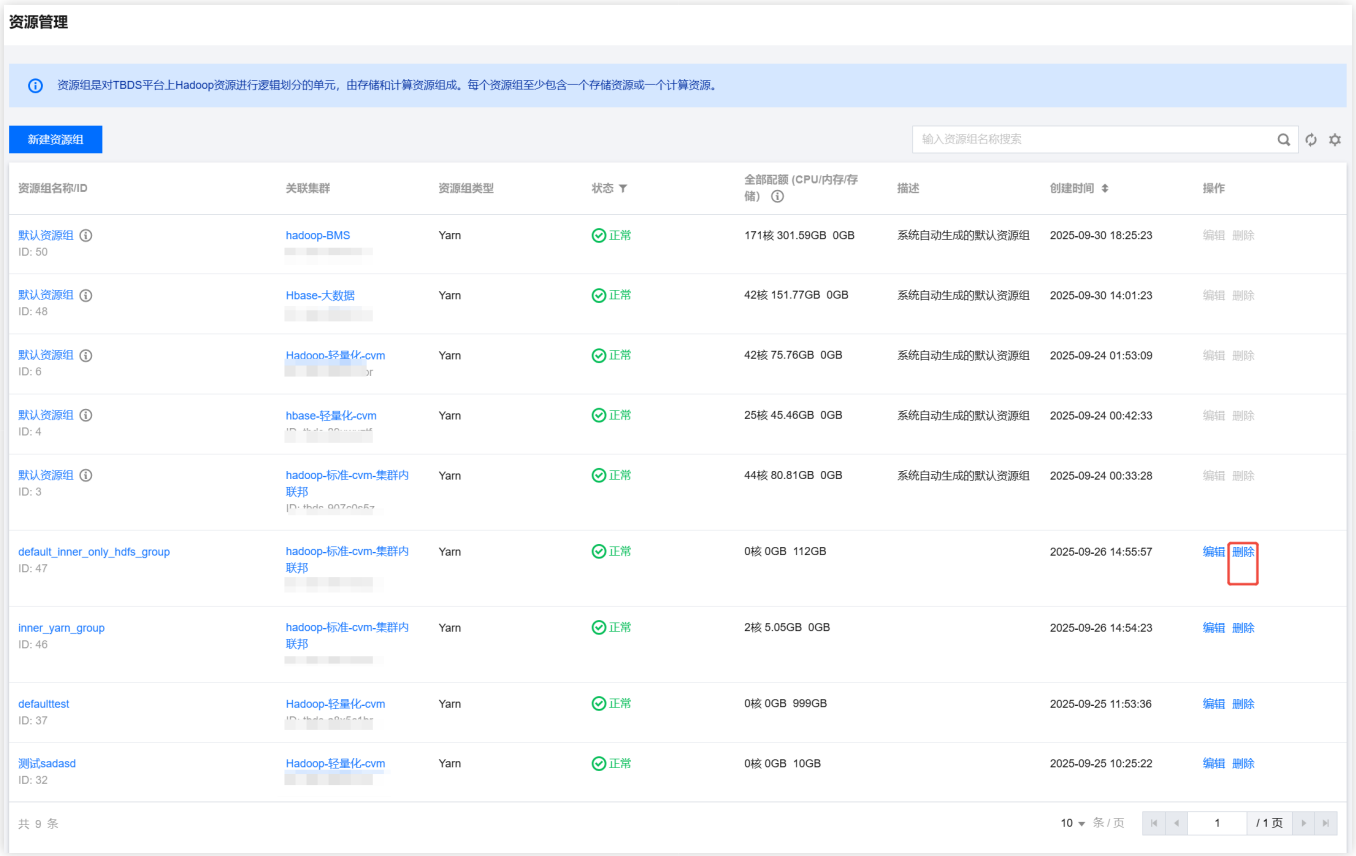
已存在 YARN 资源组。

操作场景

若您有不再使用的资源组，可以在资源管理界面中进行删除操作。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 资源管理，在资源组列表中找到需要删除的资源组，在操作中点击删除。



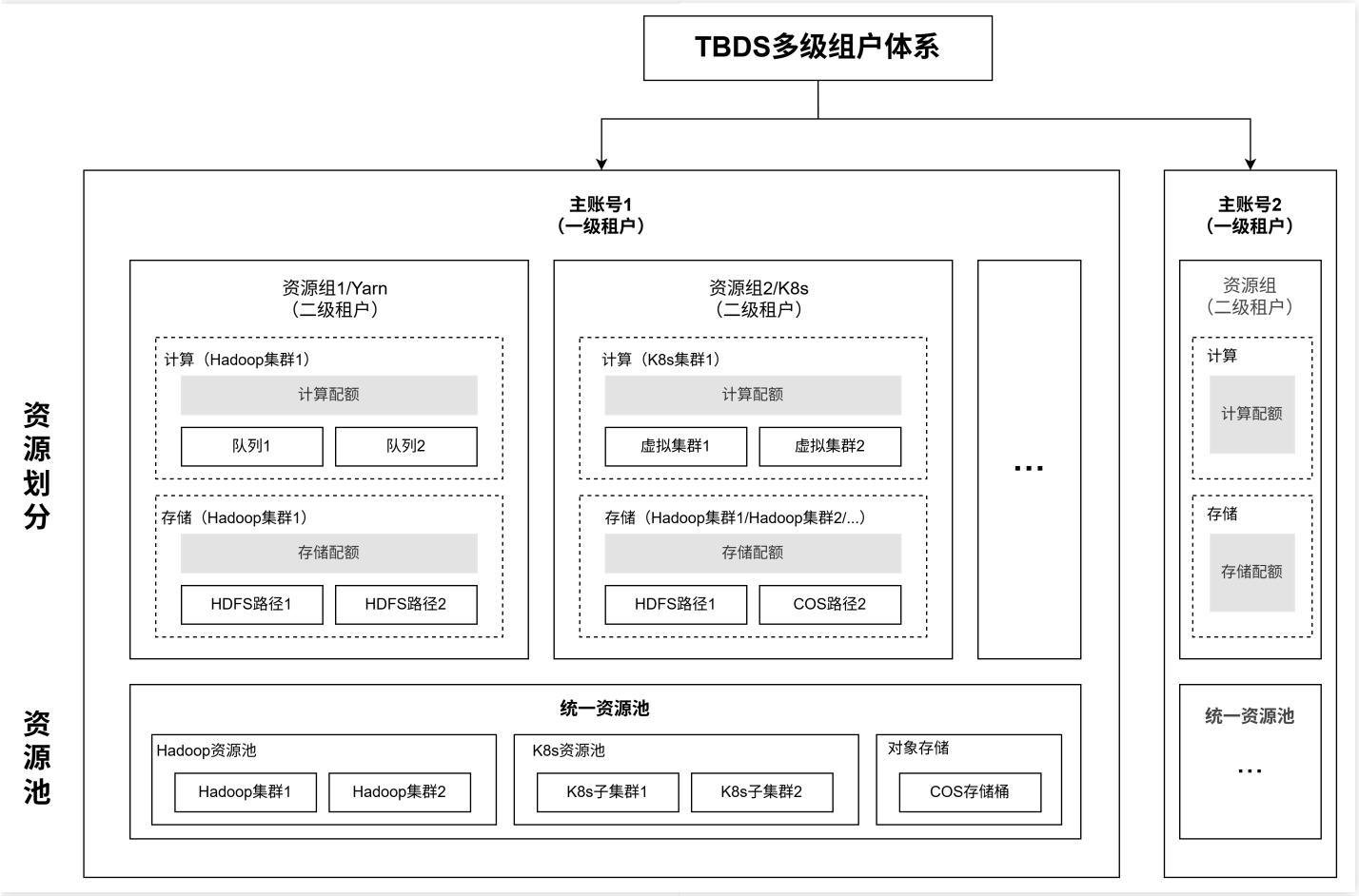
2. 确认后点击删除，说明：如果资源组下关联了 YARN 队列则不允许删除。

确认删除资源组 test 吗?
删除后，使用该资源组的业务受到影响且无法恢复，请谨慎操作！
我已知晓以上信息并确认删除
删除
取消

3. 操作完成。

用户权限
多租户机制介绍

TBDS 为用户提供 2 层级的租户管理机制，包括支持物理隔离的一级租户和逻辑隔离的二级租户。



说明如下：

- 主账号隔离为一级，主账号间完全隔离不互通，支持物理资源隔离；
- 资源组隔离为二级，支持逻辑资源隔离，资源类型包括 HDFS、COS、YARN 资源。
- 资源组是对资源的逻辑划分，由存储和计算资源构成，每个资源组至少包含一个存储资源或一个计算资源。
- 如果资源组中包含了 YARN 队列，那么其所包含的存储资源必须与 YARN 队列位于同一集群。
- 支持基于用户的精细化资源权限控制，可按照资源和资源组进行访问授权。

用户和密码类型

TBDS 平台提供三类用户，第一类是租户账号，第二类是平台内部保留用户，第三类是普通用户：

用户类型	用户	说明
租户管理员（主账号）	默认内置，主账号，即租户管理员	TBDS 部署完成后自动创建，拥有 TBDS 当前租户的所有功能、数据和资源权限
租户成员（子账号）	普通用户	由管理员用户在 TBDS “用户权限-用户”模块添加。
保留用户	默认内置，root、hadoop	仅用于平台内部访问，不允许登录 TBDS 管控平台

TBDS 已经实现了对 TBDS 控制台登录和集群访问用户的统一管理，用户可以分别设置各自的密码：

密码类型	用途
控制台登录密码	TBDS Manager 登录密码。
集群密码	集群组件的访问密码，比如 Hive、Kyuubi 等。

用户登录

以租户管理员（主账号）身份登录

操作场景

适用于 TBDS Manager 首次登录或以超管角色登录和使用的场景。

操作步骤

1. 访问地址：在浏览器中输入 TBDS Manager 访问地址，比如http://tbds.tbds.test.fsphere.cn。
2. 输入用户名和密码：在登录页面输入用户名和密码，注意：这里需要填写租户管理员用户和密码。

3. 点击登录完成。

以租户成员（子账号）身份登录

操作场景

适用于非管理员用户登录，比如运维人员、观察员等。

操作步骤

1. 访问地址：在浏览器中输入 TBDS Manager 访问地址，比如 http://tbds.tbds.test.fsphere.cn。

2. 切换子账号登录：点击“子账号登录”切换到子用户登录方式。

3. 获取子账号信息：租户管理员用户可在 TBDS Manager 右上角获取子账号登录需要的 UIN 信息。
   a. 点击登录名称，在下拉菜单中点击账号信息。

b. 在弹出的页面中复制 账号 ID，作为 UIN。

4. 输入 UIN、用户名和密码：在登录页面输入主账号 UIN，用户名和密码，请联系管理员获取。

5. 点击登录完成。

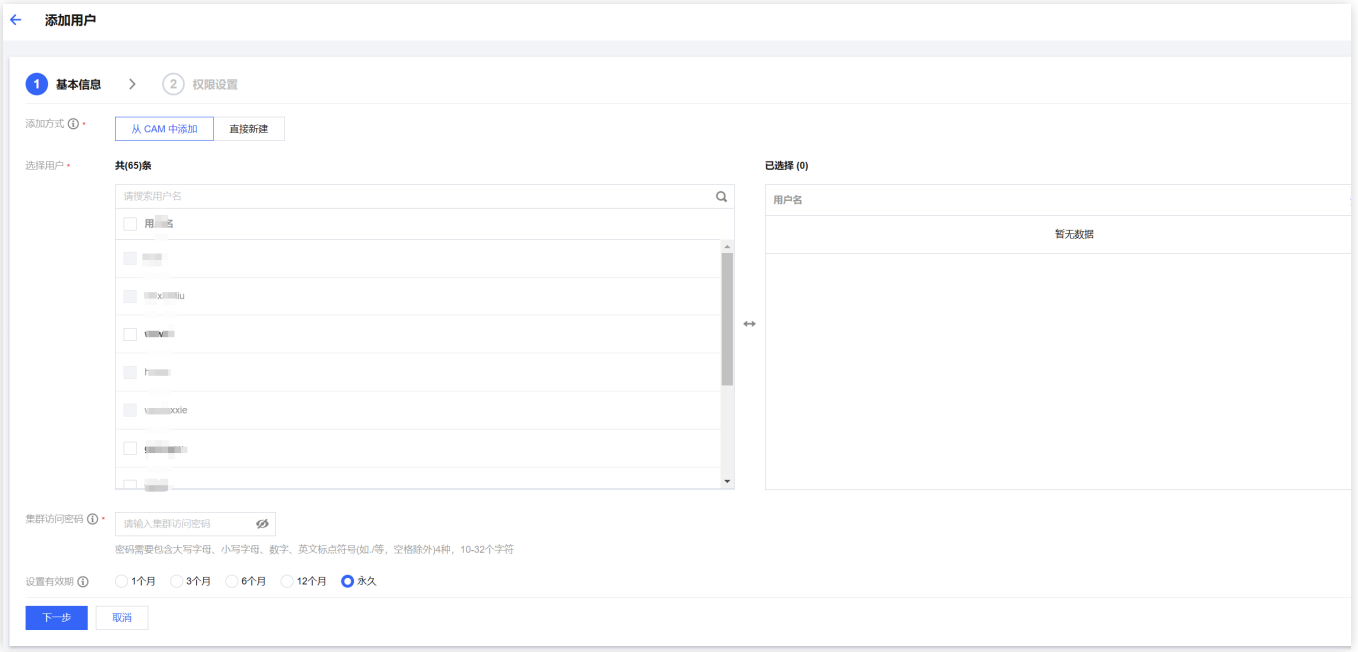
添加用户

操作场景

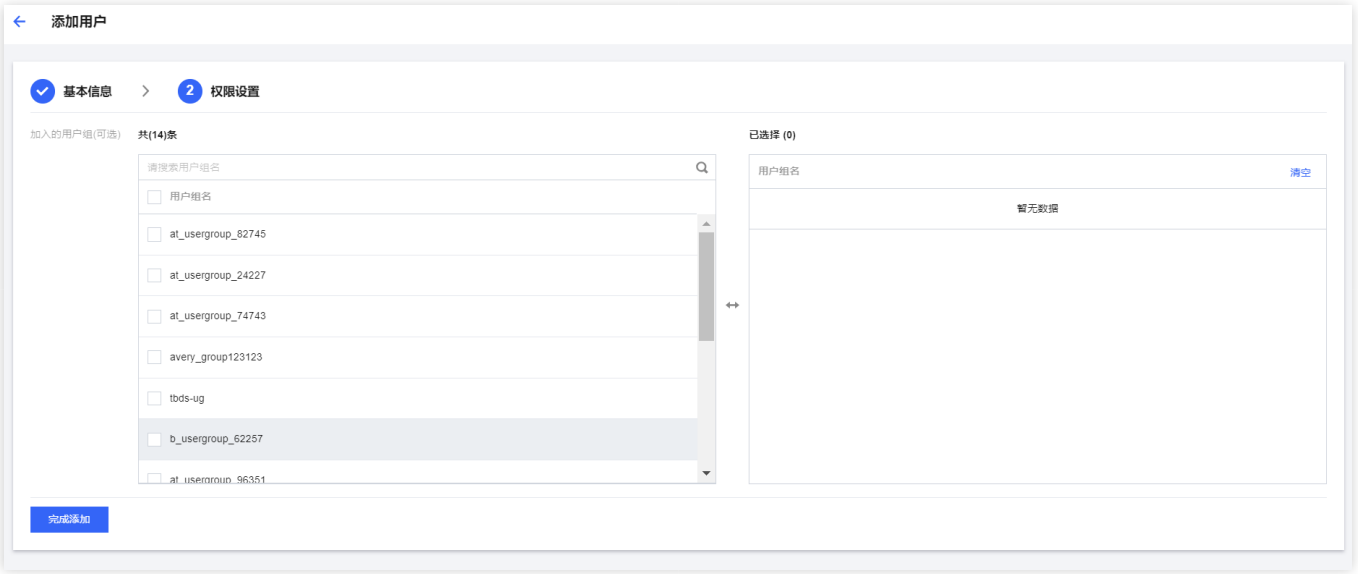
TBDS 平台默认内置的主账号为超级管理员用户。在需要进行平台管理的场景中，通常需要添加其他用户，以确保访问操作可以被审计和追踪。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限，在用户列表上方单击添加用户按钮。



2. TBDS 支持“从 CAM 中添加”和“直接新建”两种添加用户的方式，这里选择“从 CAM 中添加”，表示将 CAM 用户添加为 TBDS 用户，添加后将有权限操作 TBDS 资源。集群访问密码是用户提交任务时的密码，这里需要按要求填写。认证有效期默认为永久，有效期内用户可以访问 TBDS，如需要持续访问大数据服务，请务必在有效期结束前更新认证时间，然后点击下一步进入权限配置。



3. 单击完成添加，即可完成用户添加操作。

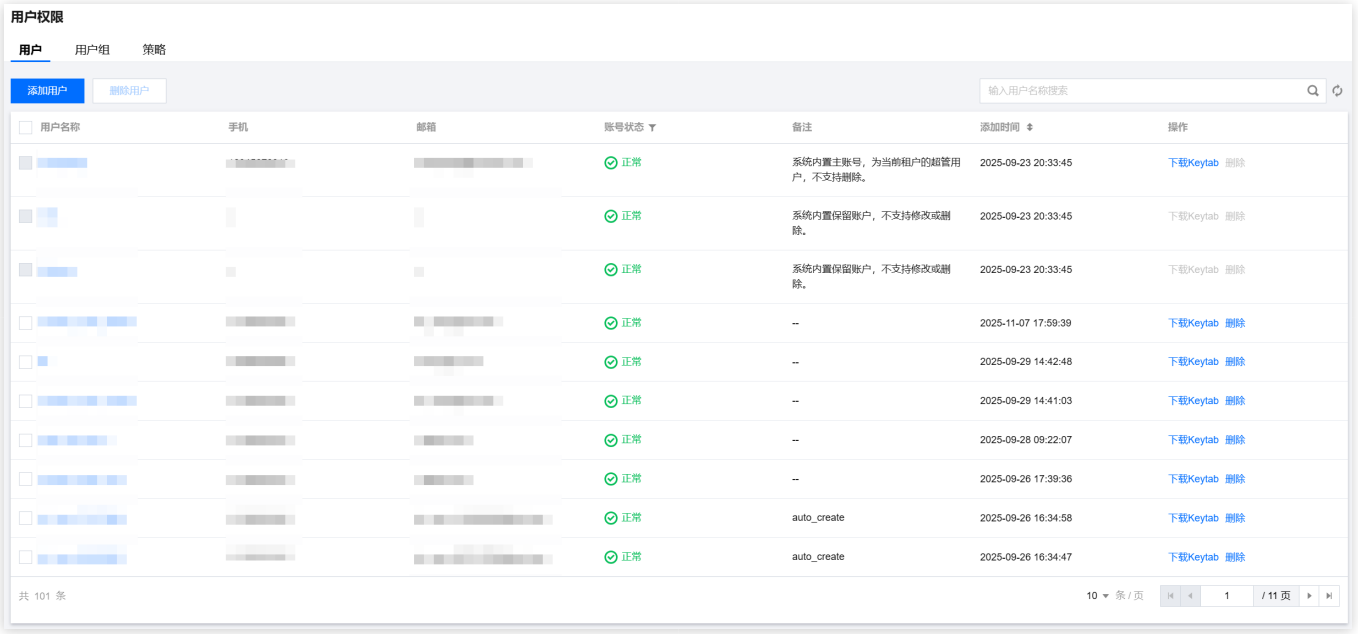
编辑用户

操作场景

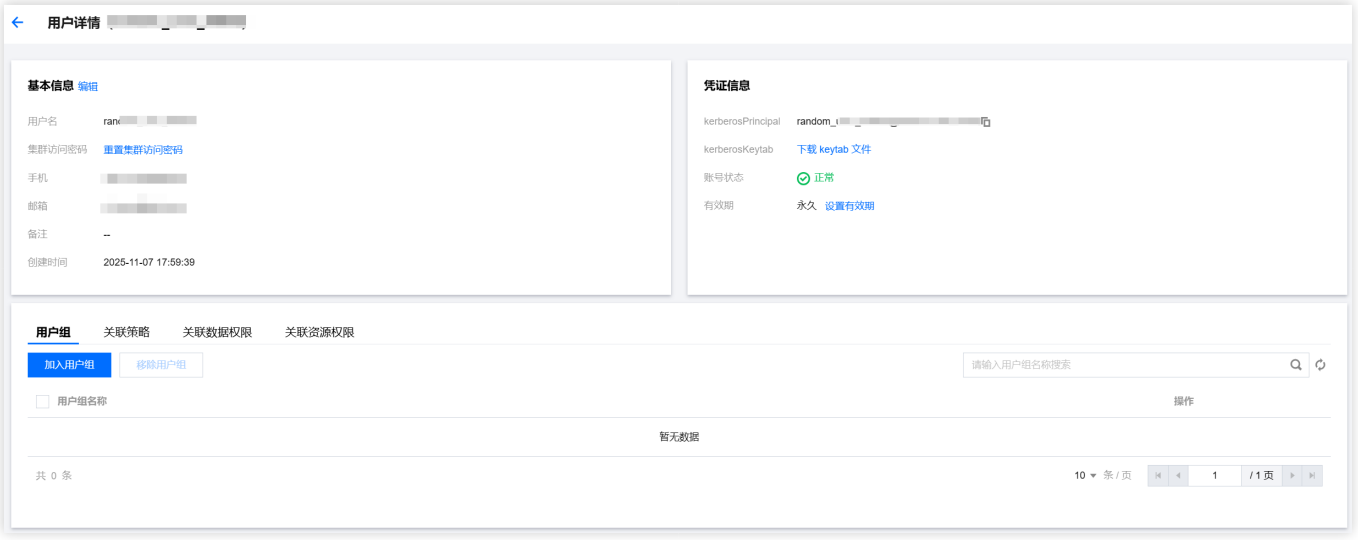
TBDS 支持对普通用户的基本信息进行编辑，仅支持编辑备注信息。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限，在用户列表中找到需要编辑的用户，点击用户名称进入用户详情。



2. 在用户详情-基本信息上点击编辑。

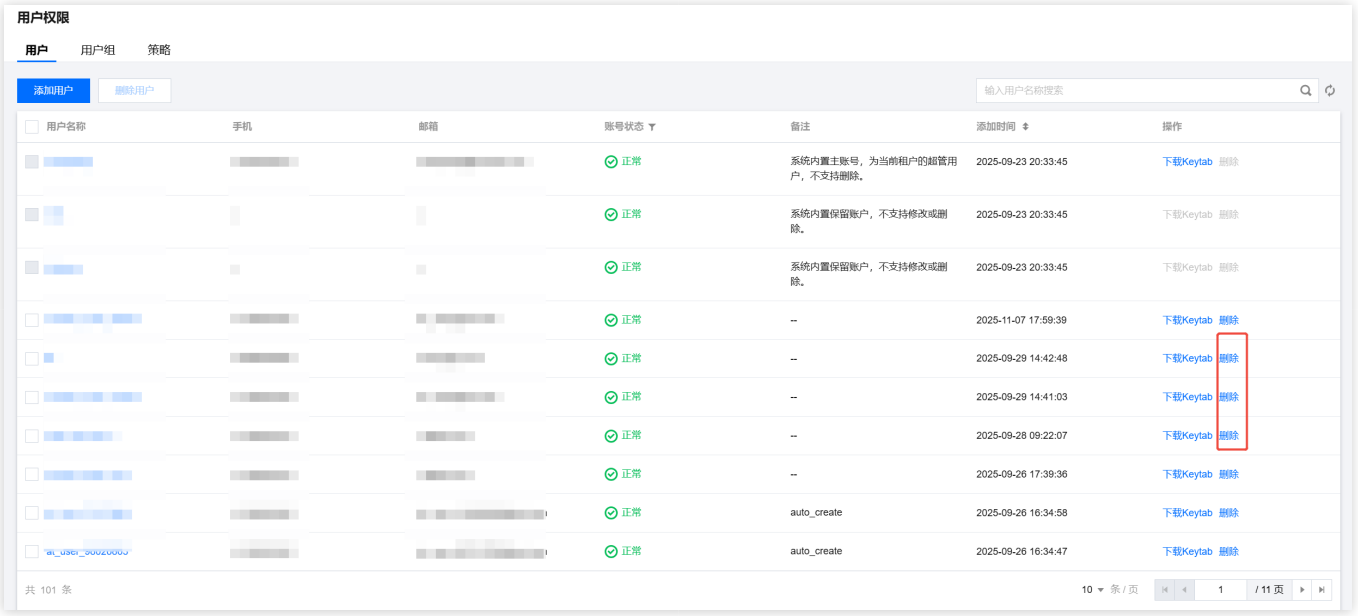


3. 在编辑用户信息弹框中仅可以编辑备注信息，点确定完成操作。

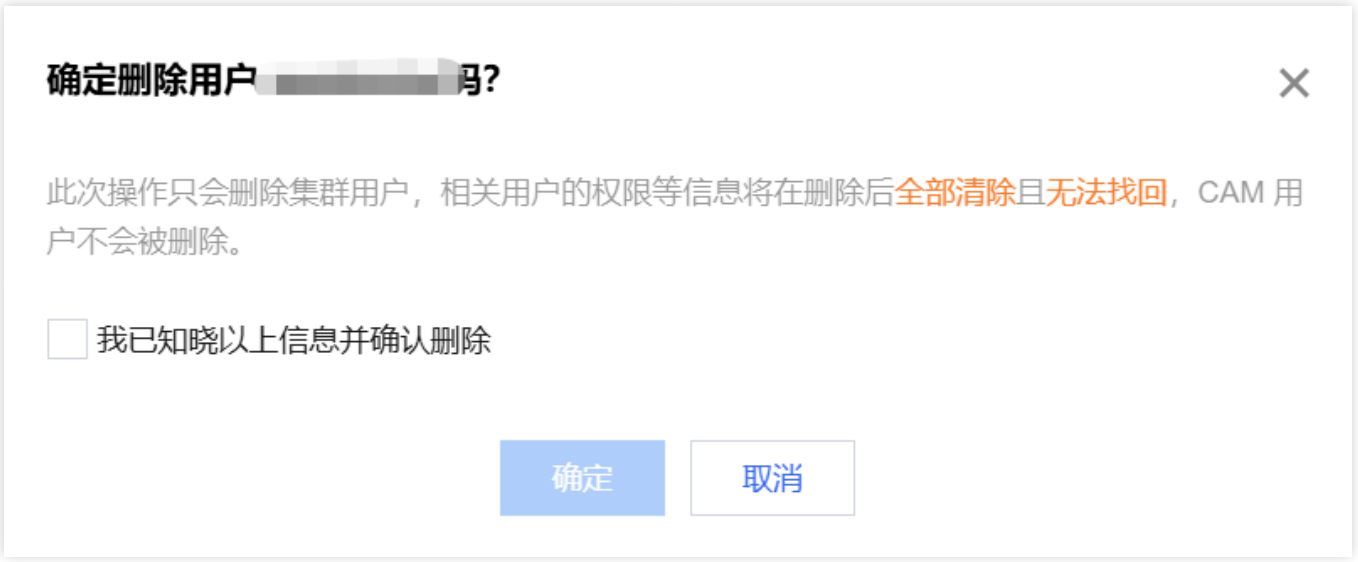
删除用户

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限 > 用户，在用户列表中找到需要删除用户。



2. 在用户列表中的操作列单击删除，勾选“我已知晓以上信息并确认删除”并单击确定。



3. 操作完成。

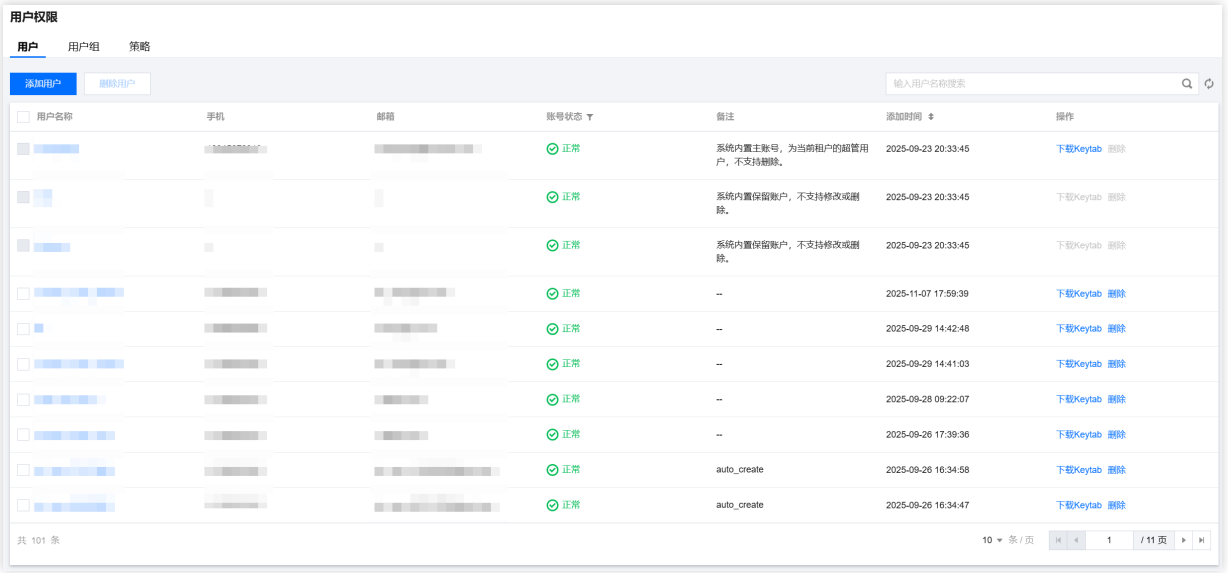
为用户授权

操作场景

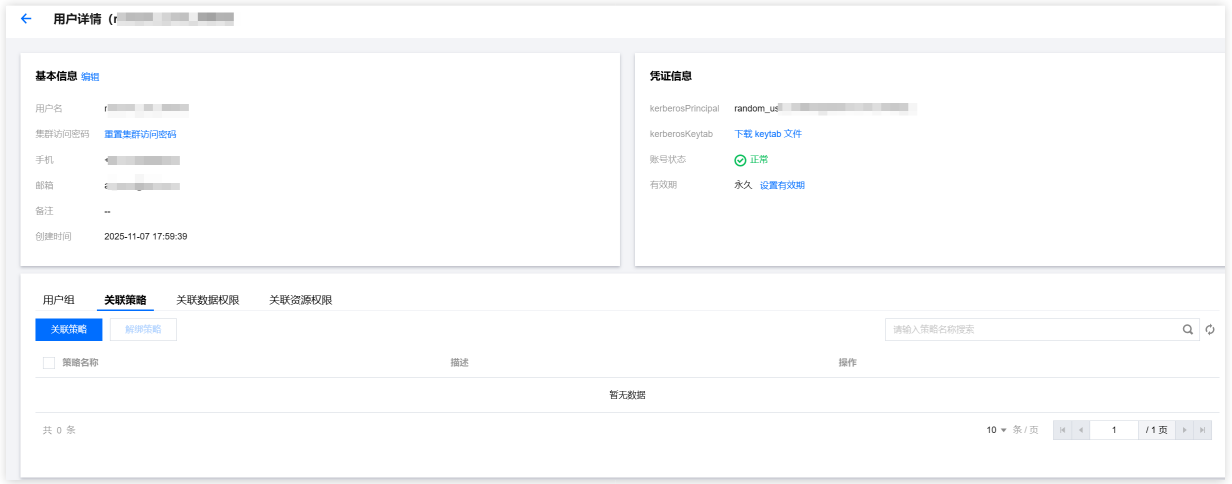
普通用户默认情况下没有平台操作权限，需要为其授予相应的权限后才能进行操作。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限，在用户列表中找到需要授权的用户。



i. 点击用户名称进入用户详情，在用户详情中可对用户授予策略、数据权限和资源权限。



ii. 关联策略：切换到关联策略页签，点击关联策略在弹窗框中选择需要关联的策略，点击确认后完成策略关联。

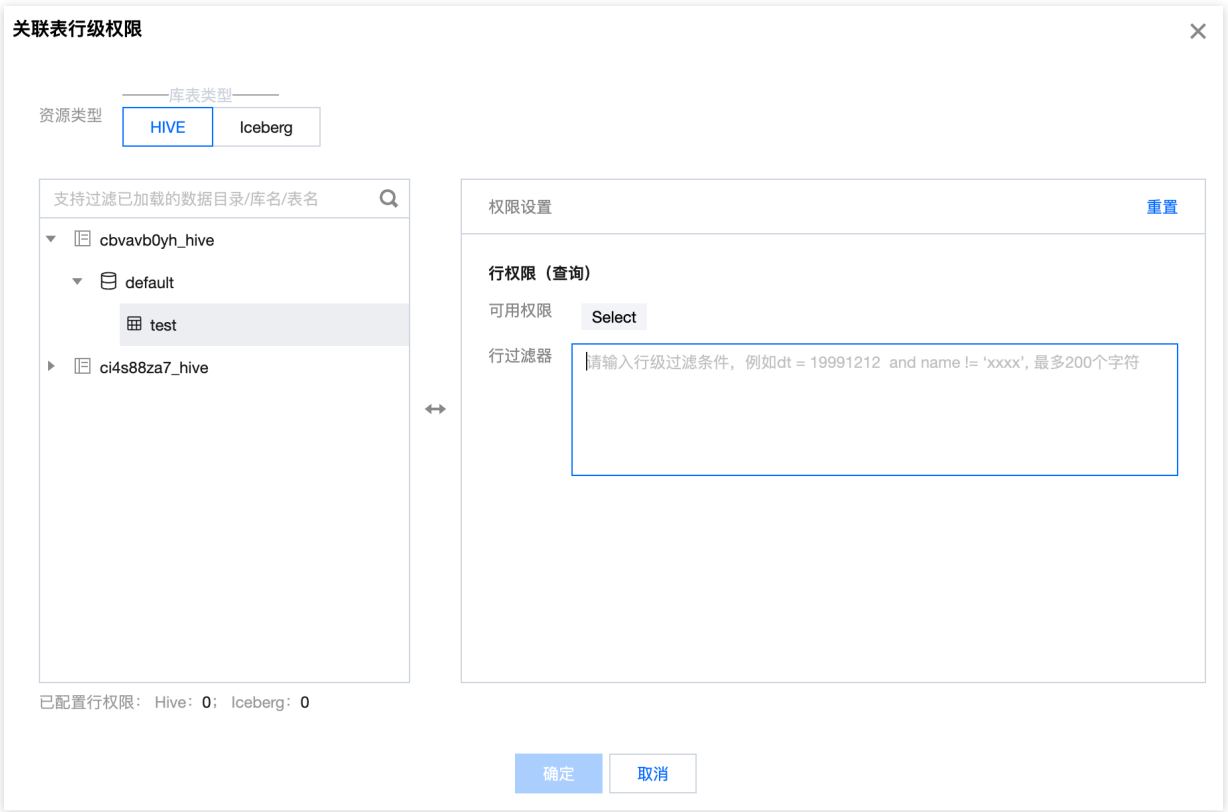
平台策略说明：

策略名	策略类型	策略描述
TBDSNewOpsFullAccess	预设策略	大数据处理套件（TBDS）的全部集群的管理能力
TBDSNewFullAccess	预设策略	大数据处理套件（TBDS）的全部管理能力
TBDSNewPlatformFullAccess	预设策略	大数据处理套件（TBDS）的平台管理权限，包括资源管理、用户权限和数据管理的所有权限
TBDSNewReadOnlyAccess	预设策略	大数据处理套件（TBDS）的只读权限
TBDSNewDevFullAccess	预设策略	大数据处理套件（TBDS）的开发权限，包括数据管理全部权限和其他模块只读权限
TBDSNewAuditAccess	预设策略	大数据处理套件（TBDS）的审计管理员权限，拥有审计中心的访问和管理权限

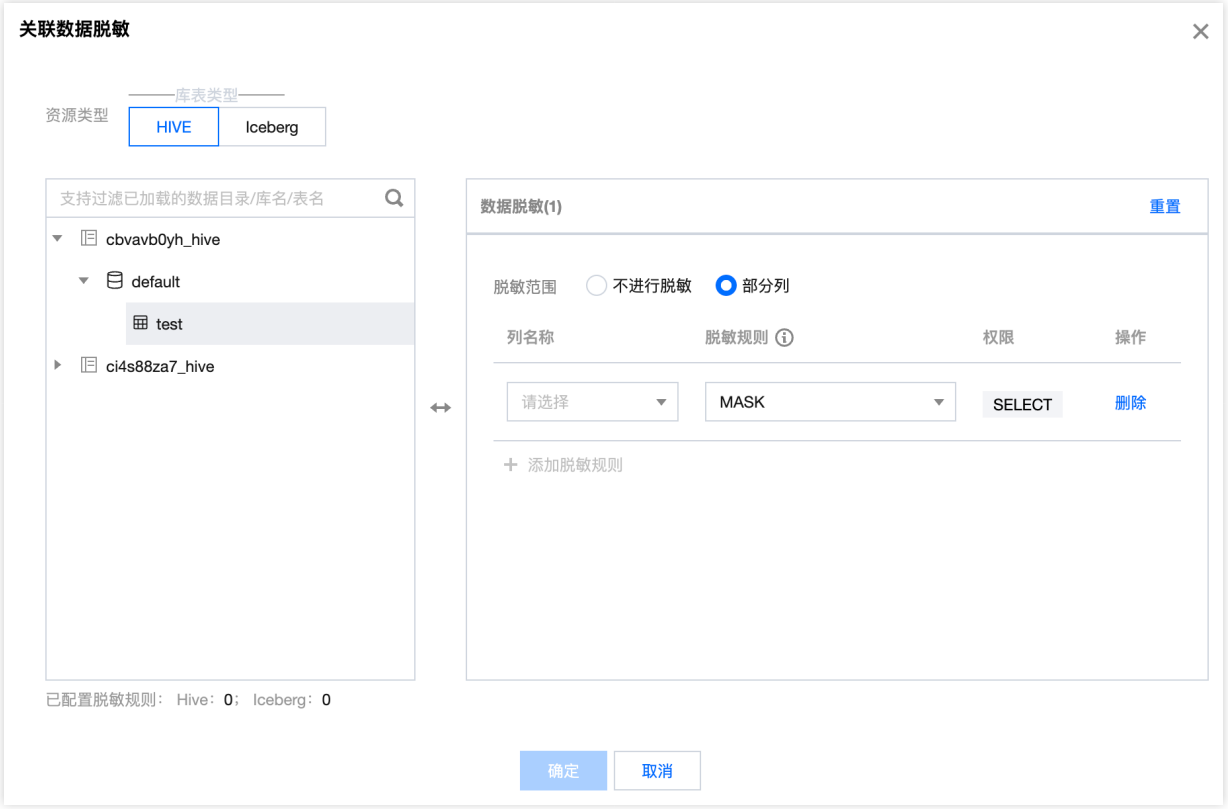
iii. 授予数据权限：切换到关联数据权限页签，支持为用户设置目录库权限、行权限和数据脱敏。

a. 设置目录库表列权限：在左侧选择需要授权的库表列，在右侧按需选择权限点，点击确定后完成设置。

b. 设置行权限：在左侧选择需要授权的库表列，在右侧输入表的 where 条件表达式实现行过滤，点击确定后完成设置。



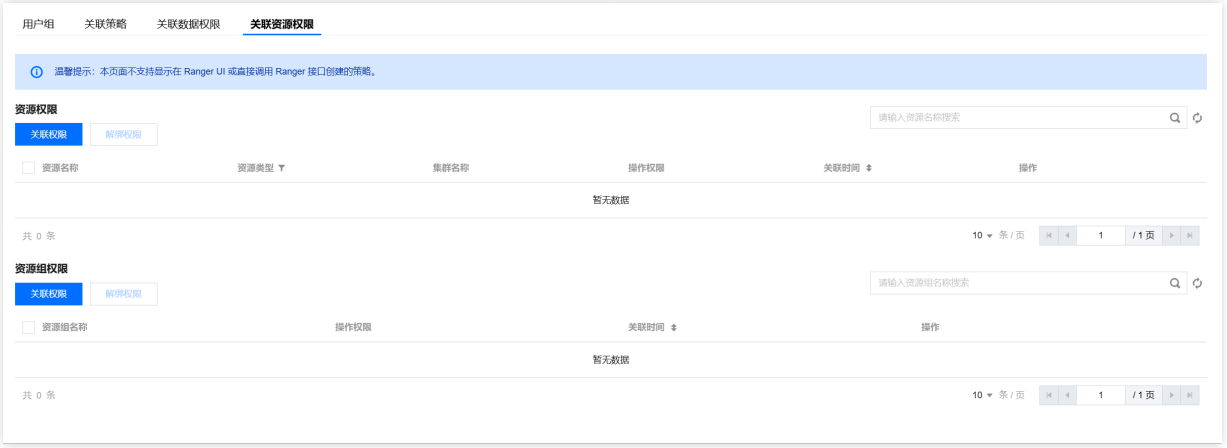
c. 设置数据脱敏：在左侧选择需要授权的库表列，在右侧按需选择脱敏字段和规则，点击确定后完成设置。

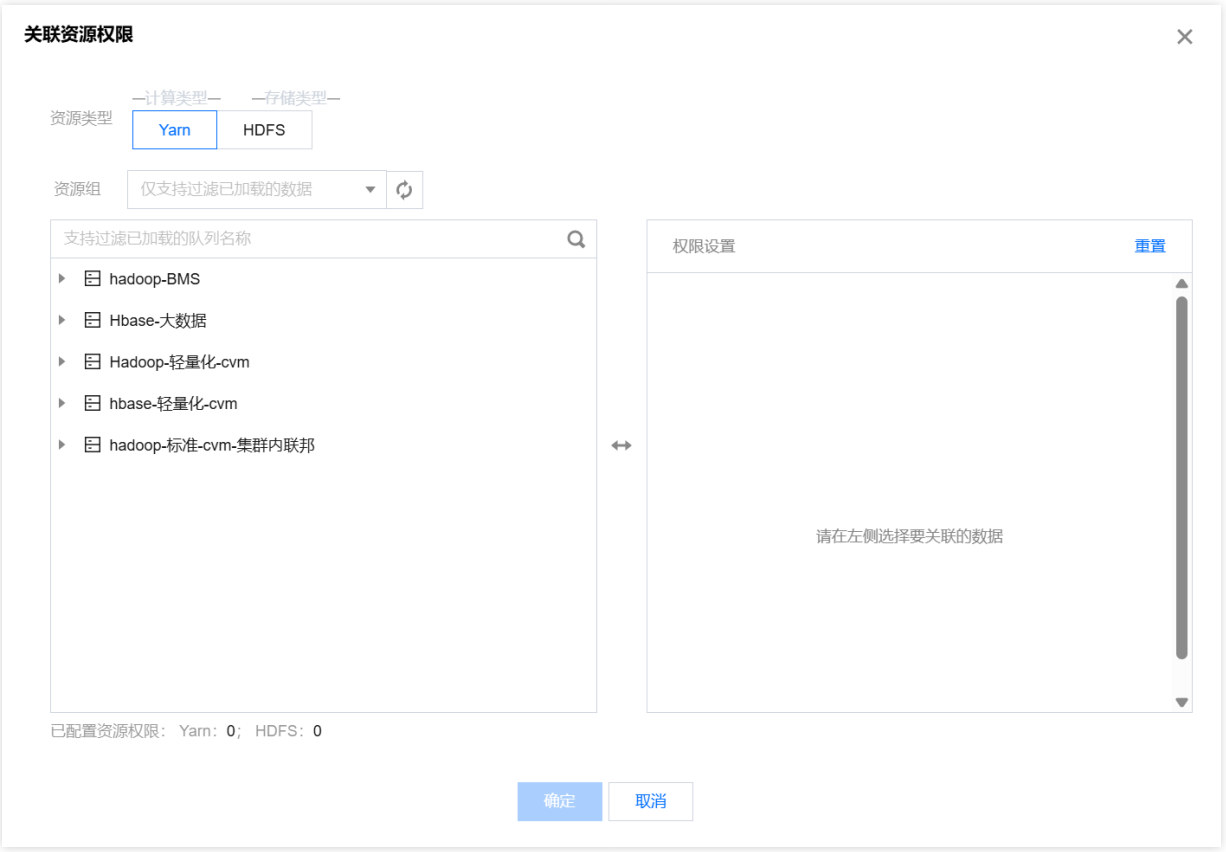


支持的脱敏规则说明如下：

角色	描述
MASK	将所有字母字符掩码为“x”，所有数字字符掩码为“n”。
MASK_SHOW_LAST_4	只显示最后四个字符。
MASK_SHOW_FIRST_4	只显示前四个字符。
MASK_HASH	将所有字符替换为整个单元格值的哈希值。
MASK_NULL	将所有字符替换为NULL值。
MASK_NONE	不进行任何掩码处理，显示原始值。
MASK_DATE_SHOW_YEAR	只显示日期字符串的年份部分，并将月和日默认设置为01/01。

iv. 授予资源权限：切换到关联资源权限页签，支持为用户设置 HDFS、YARN资源访问权限。





2. 返回列表查看授权结果，操作完成。

为用户授权资源组权限

操作场景

如果用户希望拥有资源组内所有资源的访问权限，可以使用资源组快捷授权功能。

操作步骤

1. 以管理员用户登录 TBDS，进入TBDS Manager 首页 > 用户权限,在用户列表中找到需要授权的用户。
2. 点击用户名称进入用户详情，切换到关联资源权限页签，点击关联资源组权限进行设置，支持选择多个，点击配置后完成权限设置。

3. 返回列表查看授权结果，操作完成。

查看用户访问凭证

前提条件

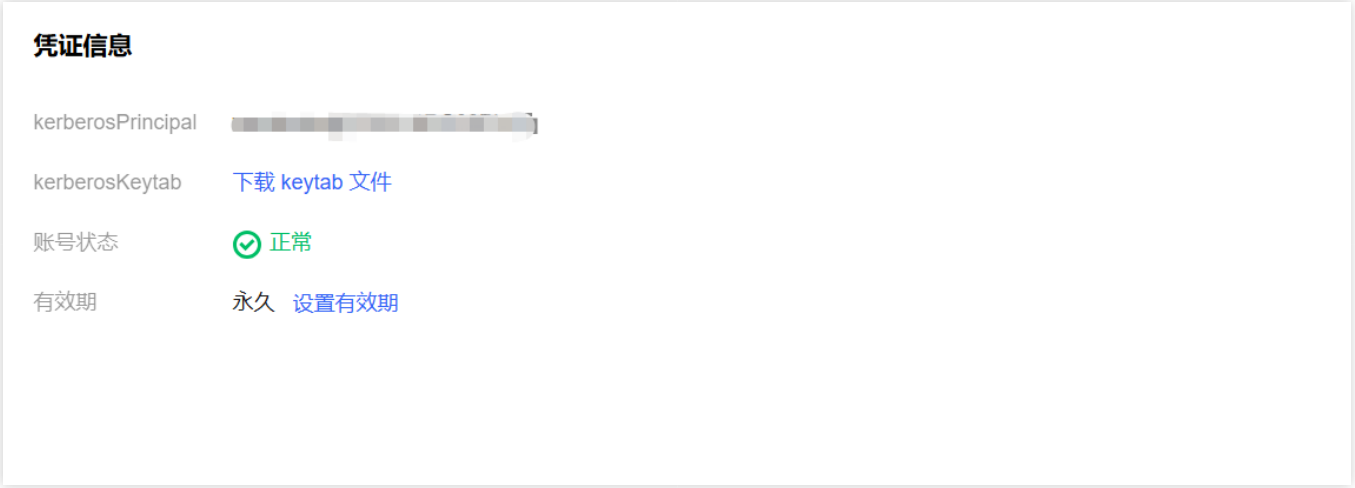
- 已完成用户添加

操作场景

当用户访问启用了 Kerberos 认证的集群时，需要提供 principal 和 keytab 凭证信息，以确保能够顺利通过认证，此时需要查找到用户凭证信息。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限，在用户列表中找到需要查看凭证的信息的用户。
2. 点击用户列表中的用户名称进入用户详情页面，在右上角凭证信息栏可以看到 kerberosPrincipal 和 kerberosKeytab，复制和下载即可。



3. 操作完成。

重置用户集群访问密码

前提条件

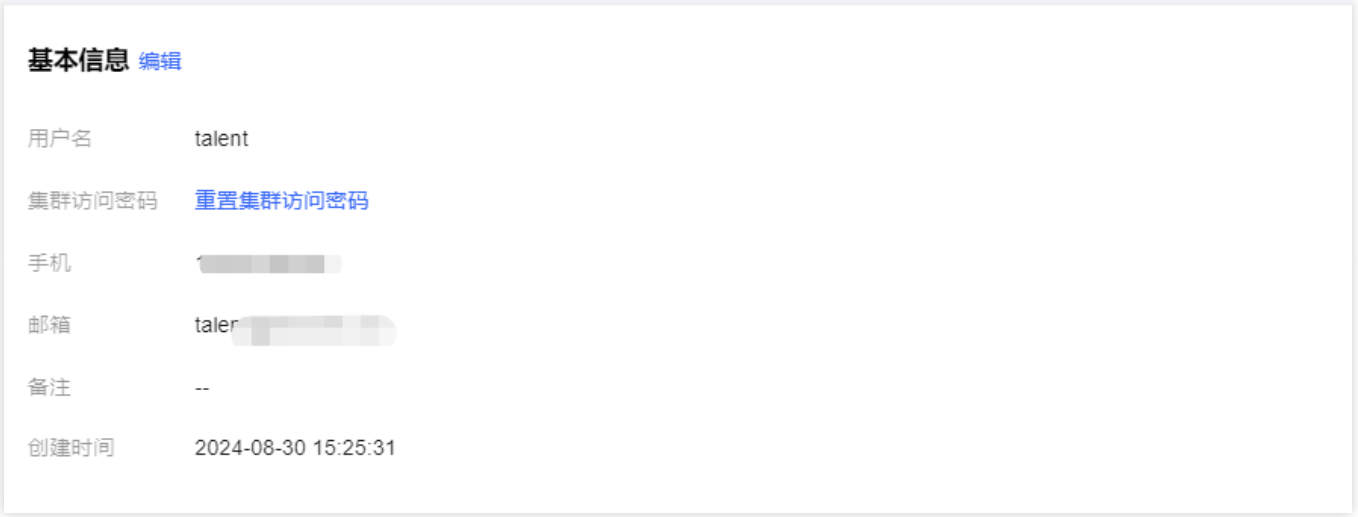
- 已完成用户添加。

操作场景

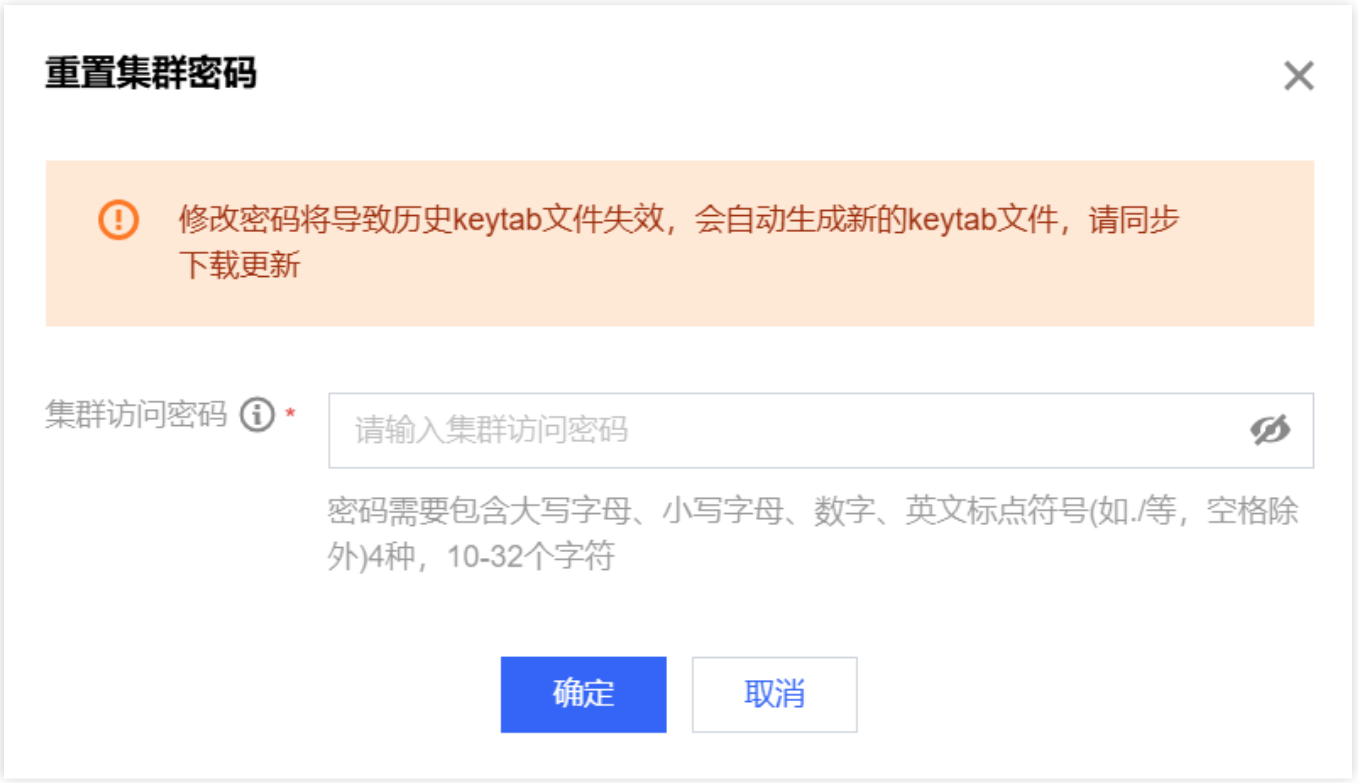
用户访问 TBDS 集群服务依赖“集群访问密码”，您可以在用户详情页面重置该密码，满足企业的安全要求。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限，在用户列表中找到需要查看凭证的信息的用户。
2. 点击用户列表中的用户名称进入用户详情页面，在基本信息中点击重置集群访问密码链接：



3. 在重置集群密码中输入新密码，点击确定。


重置集群密码
修改密码将导致历史keytab文件失效，会自动生成新的keytab文件，请同步下载更新
集群访问密码 *
请输入集群访问密码
密码需要包含大写字母、小写字母、数字、英文标点符号(如./等，空格除外)4种，10-32个字符
确定
取消

4. 操作完成。

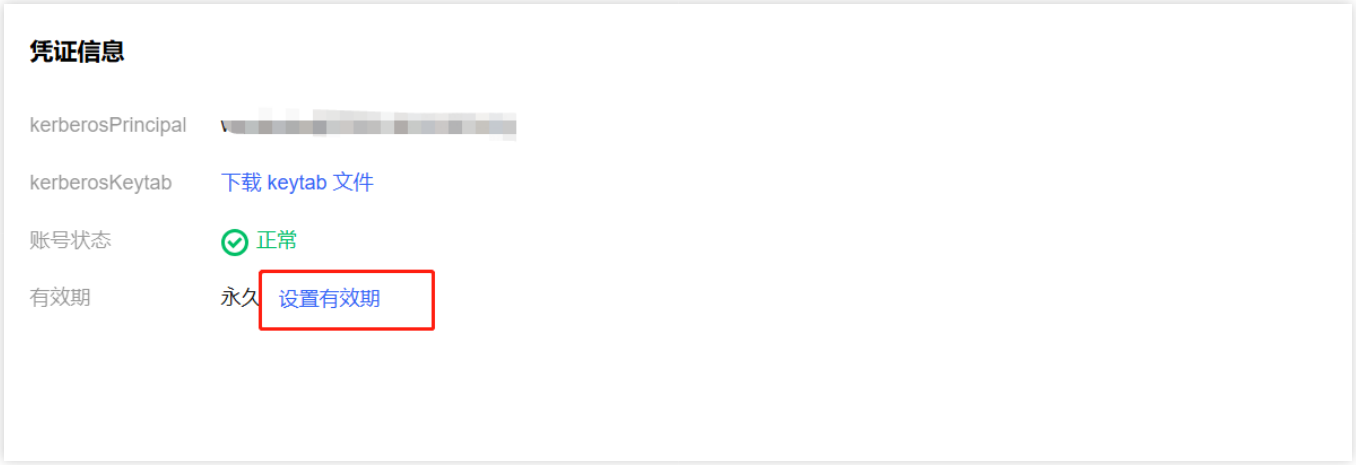
设置用户有效期

操作场景

在使用TBDS创建用户时，默认会生成一个有效期为"永久"的keytab。针对那些对安全性要求较高的业务场景，用户可以自行设定keytab的有效期。本文将指导您如何为用户配置keytab的有效期限制。

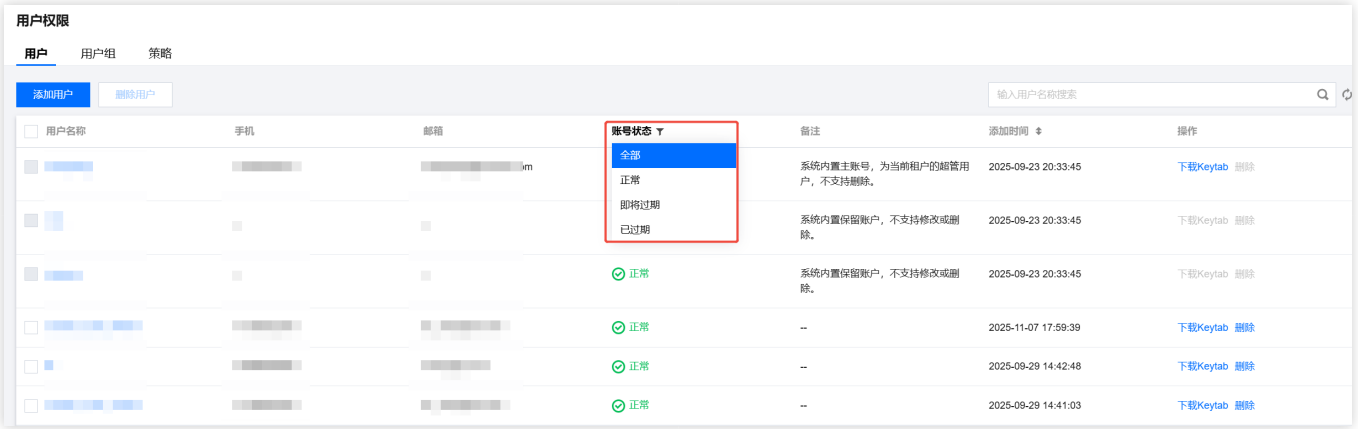
操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限，在用户列表中找到需要更新认证有效期的用户。
2. 点击用户列表中的用户名称进入用户详情页面，在右上角凭证信息栏点击设置有效期：



3. 在用户详情->凭证信息中点击"设置有效期"为用户指定一个有效期，指定有账号的有效期即可生效。

4. 当用户keytab的有效期少于15天时，系统会在所有页面标题栏上方显示提示信息，提示关闭后在当前登录会话中不再显示，但用户下次登录时提示将重新出现。对于过期的账号您也可以在用户列表进行筛选查看。



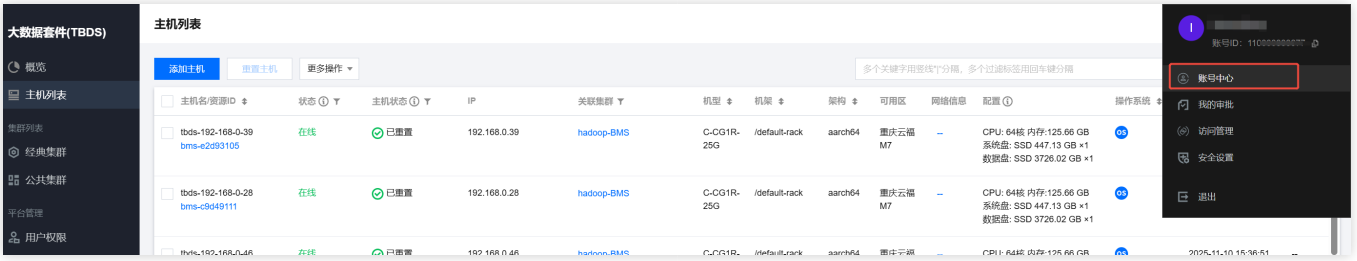
重置用户登录密码

前提条件

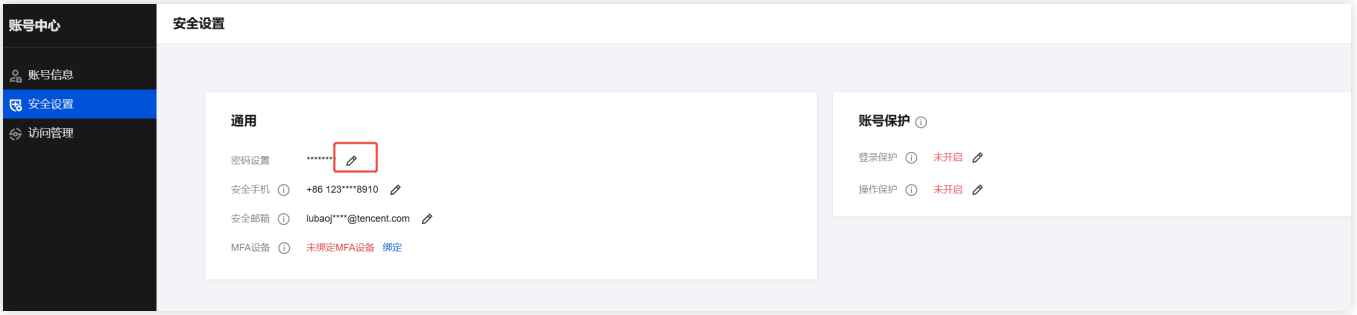
- 已完成用户添加。

操作步骤

1. 用户登录 TBDS，进入 TBDS Manager 首页，右上角点击“用户名称”在下拉选择中找到“账号中心”并点击。



2. 进入账号信息页面，点击“安全设置”菜单。在通用板块点击“修改密码”，并在修改密码弹框中输入新旧密码，点击确定。



修改密码
旧密码 *
请输入旧密码
新密码 *
请输入新密码
密码需要包含大写字母、小写字母、数字、特殊字符(如./_等，空格除外)4种，且不得包含用户名，10-32个字符。
确认密码 *
请输入新密码
确定
关闭

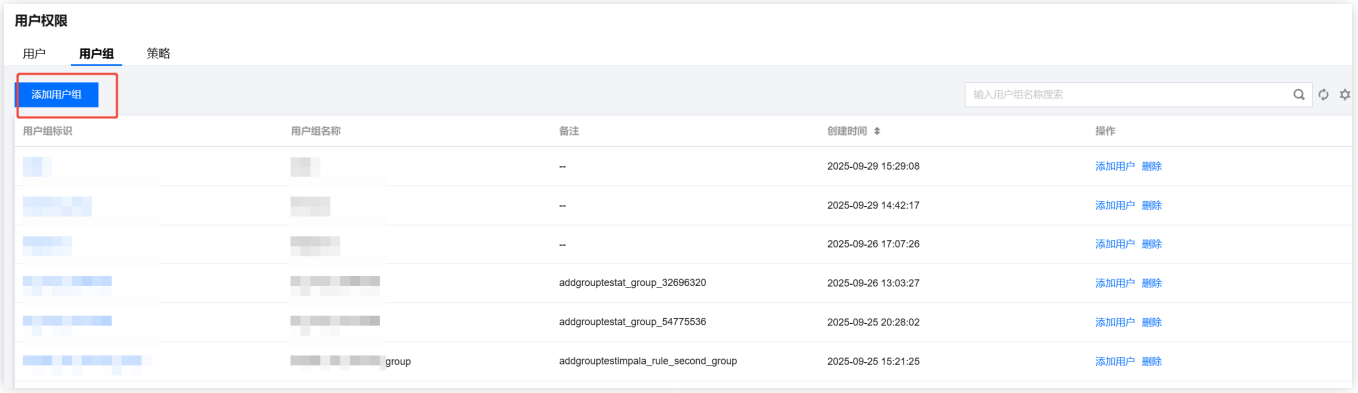
3. 操作完成。

用户组管理

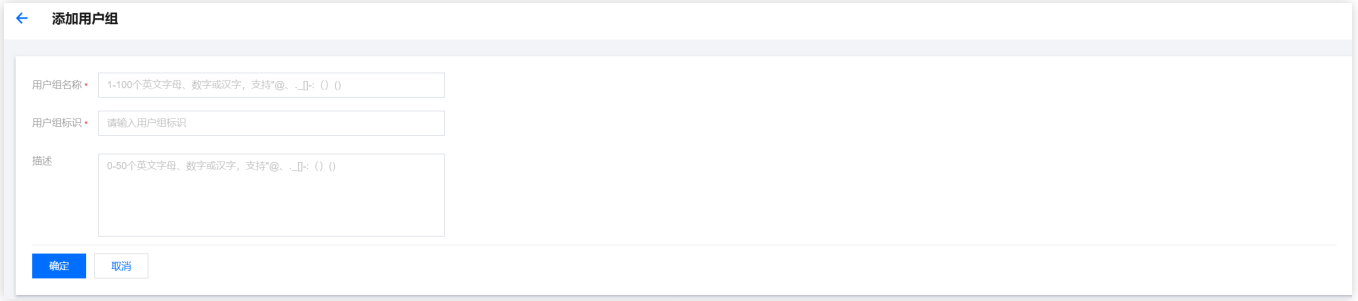
添加用户组

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限 > 用户组，在用户组列表上方点击添加用户组。



2. 在添加用户组页面中填写用户组相关信息，点击确定后完成操作。



信息项	描述
用户组名称	用户组名称支持修改
用户组标识	用户组唯一标识，不允许修改
描述	最长50个字符

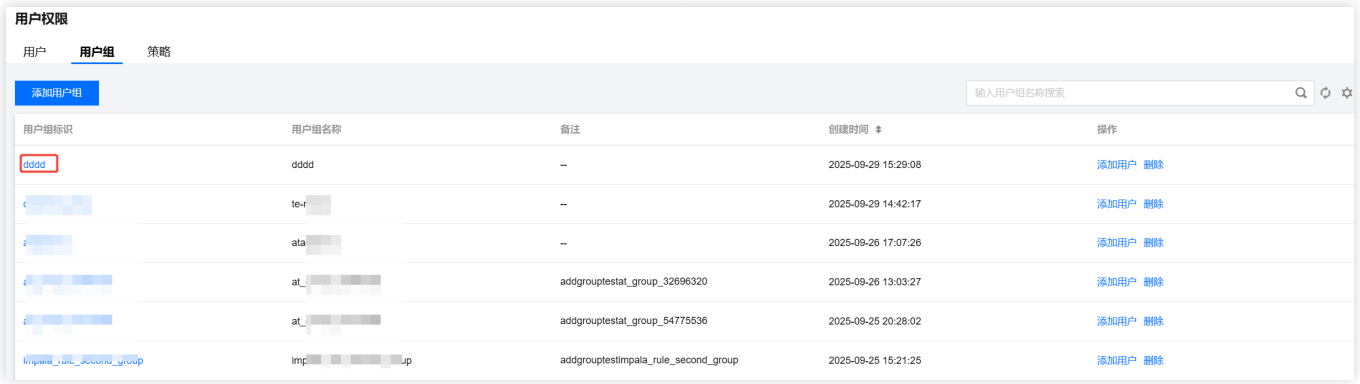
编辑用户组

操作场景

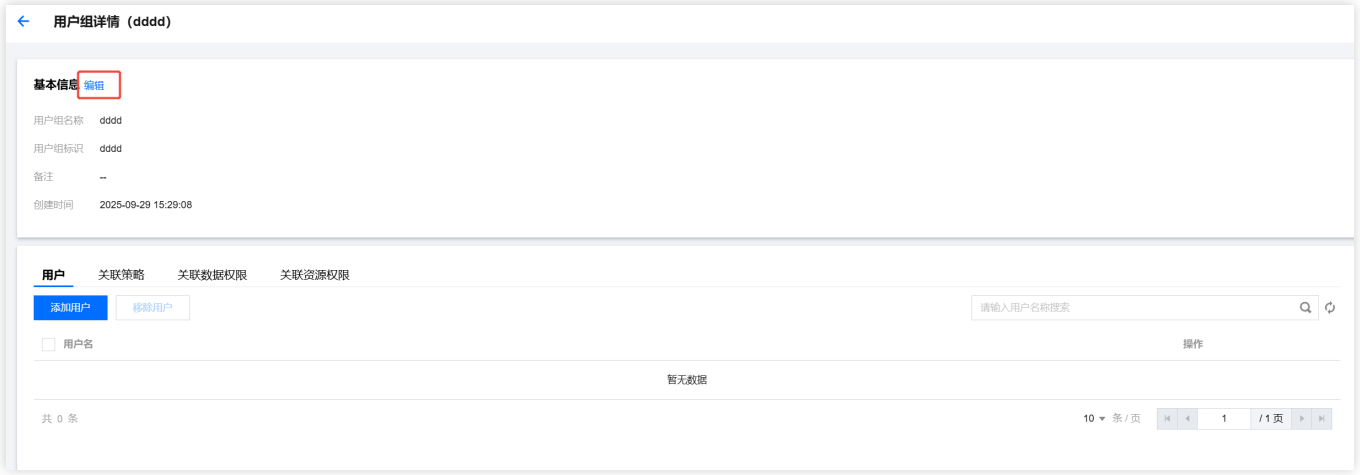
TBDS 支持对用户组名称和备注进行编辑。

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限-用户组，在用户组列表中找到需要编辑的用户组，点击用户组标识进入用户组详情。



2. 在用户组详情-基本信息上点击编辑。

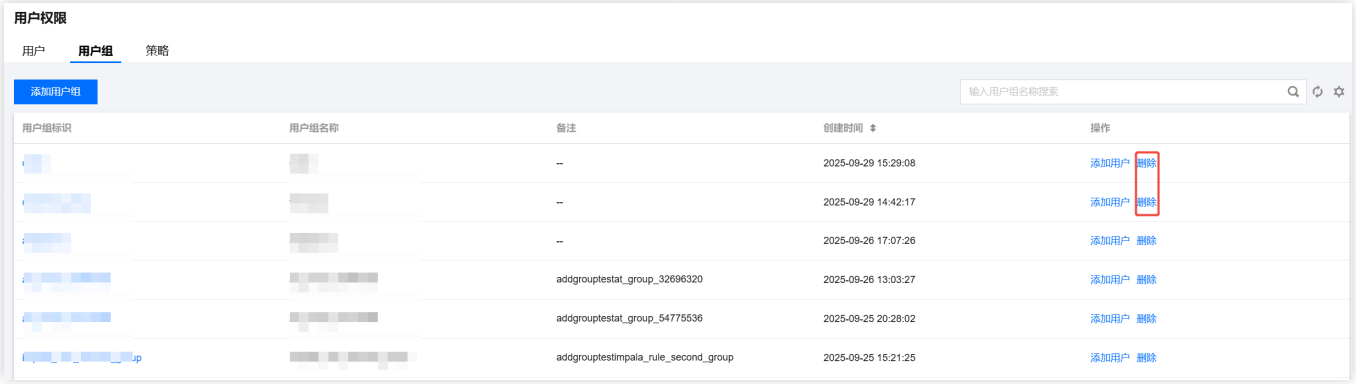


3. 在“编辑用户组信息”弹框中可以修改用户组名称和备注，点确定完成操作。

删除用户组

操作步骤

1. 以管理员用户登录 TBDS，进入 TBDS Manager 首页 > 用户权限 > 用户组，在用户组列表中找到需要删除的用户组。



2. 在用户组列表中的操作列点击删除，确认风险后勾选“我已知晓...”并点击确定。

确定删除用户组 测试用户组 吗?
删除用户组后，用户组相关的权限等信息均会被删除，且无法恢复
我已知晓以上信息并确认删除
确定
取消

3. 操作完成。

标签管理

操作场景

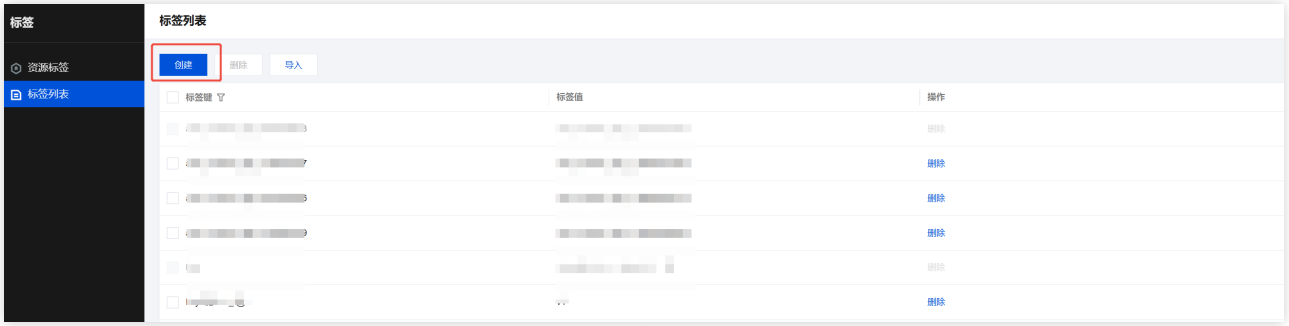
标签（Tag）是一种资源管理工具，您可以从不同维度对具有相同特征的资源进行分类、搜索和聚合，从而轻松管理TBDS资源。标签是由标签键和标签值两个部分组成，您可以为资源创建和绑定标签。一个标签键可以对应多个标签值，一对标签键和标签值可绑定多个资源。

操作步骤

创建标签

1. 登录 TBDS 控制台，通过编辑标签弹窗进入到标签管理页面。

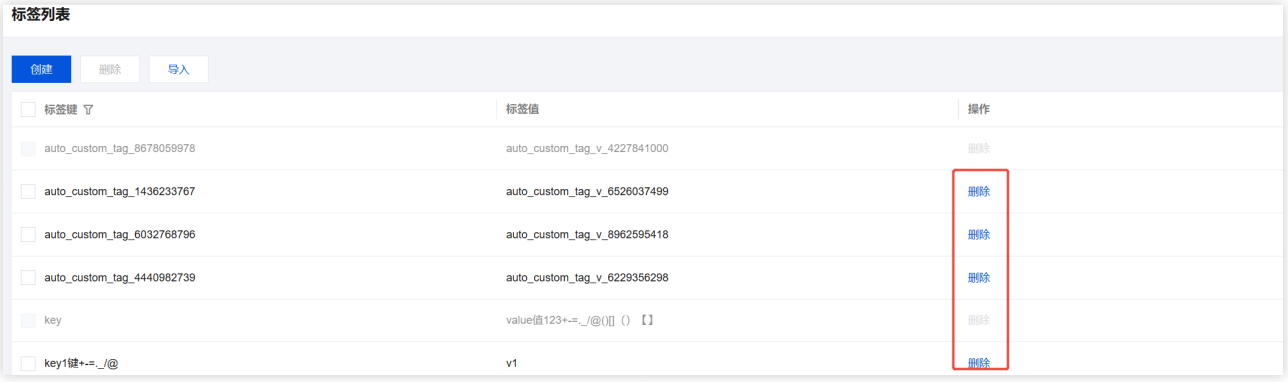
2. 在标签列表页面点击“创建”按钮。



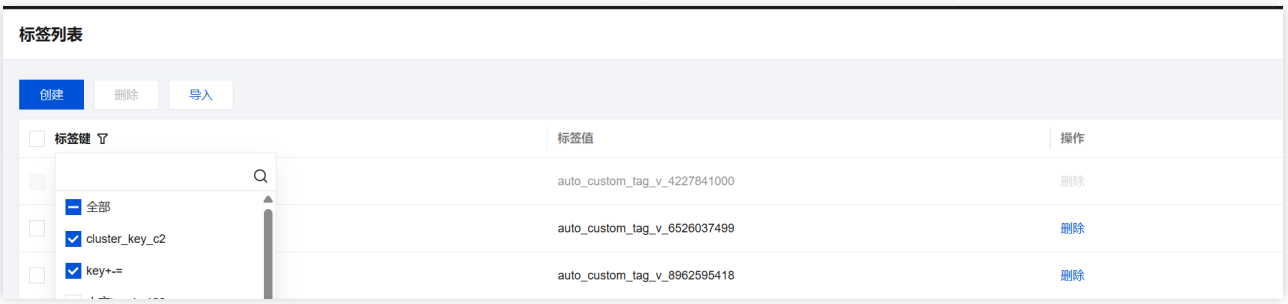
3. 在新建标签对话框填写标签键和标签值。

删除标签

1. 登录 TBDS 控制台，通过编辑标签弹窗进入到标签管理页面。
2. 找到需要删除的标签并点击删除。
   i. 在标签列表找到需要删除的标签，点击操作列的"删除"链接。



   ii. 按照标签键搜索需要删除的标签并勾选，点击删除。



审计中心
概述

操作场景

为满足大数据平台安全合规管理的要求，TM管控平台提供了“审计中心”，集中对全量用户在TM管控的所有集群内进行的功能操作、核心组件访问操作进行审计；审计中心记录的日志详情可参考审计日志。

概述

用户登录TM后，可在平台管理下访问审计中心，审计中心核心功能包括：
1、操作日志：对当前租户下全量用户操作TM管控的所有集群（公共集群、虚拟集群、经典集群）核心操作进行日志查询、筛选导出和生命周期管理；
2、访问日志：通过跳转到Ranger UI Audit 查看当前租户下全量用户操作集群内核心组件的访问日志审计；
3、日志转储：支持用户配置操作日志、访问日志按时间周期或日志数量定期转储到远端FTP。

权限要求

审计中心模块允许系统管理员进行访问和编辑操作，也可给用户分配审计操作策略进行授权。

操作日志

操作场景

管理员可通过审计中心的操作日志模块，对用户的集群、组件级别功能操作进行查询检索、筛选导出功能，达到快速定位用户对集群组件高危操作的目的。

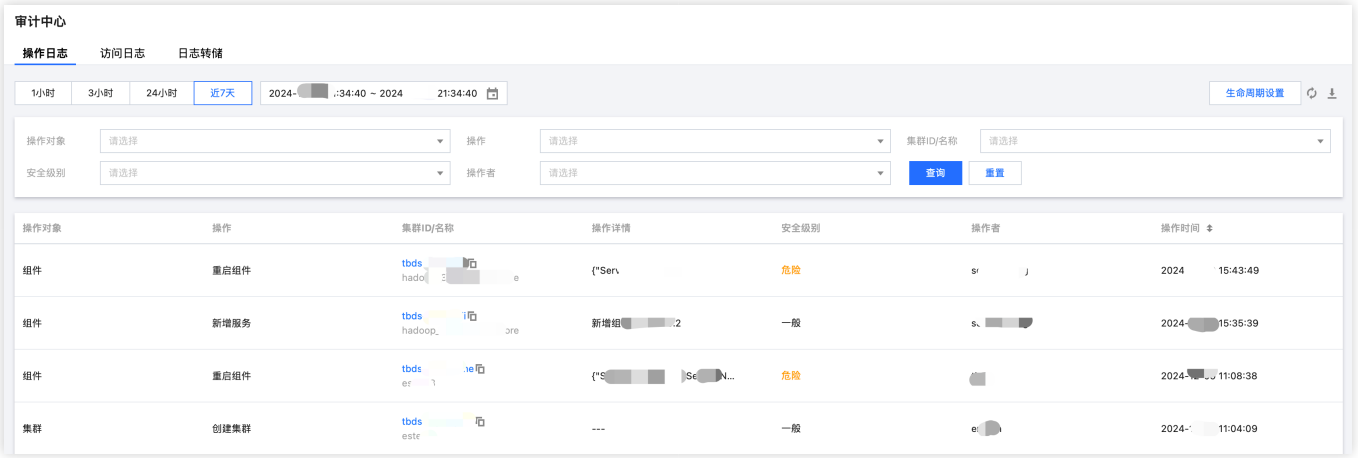
功能说明

1. 操作日志支持筛选操作时间、集群（整合公共集群、虚拟集群、标准集群ID/名称）、操作类型、安全级别、操作者；

2. 支持筛选后的日志清单导出，筛选查询后点击 进行导出（默认格式的CSV文件）；

> 注意：
>
> 导出大小限制在10万行，超出大小将会报错提示："导出内容超出10万行限制，请调整搜索条件后导出"。

3. 支持操作日志生命周期设置，详情参见周期说明。



存储周期

TM操作审计日志默认保存在TBDS内部Mysql数据库，默认生命周期为365天，管理员可以修改生命周期（输入限制>0）。

> 说明：

> 超出生命周期的审计日志将被清理，不会在审计中心界面中展现。

日志存储格式

日志文件名	日志内容格式	示例
op.audit.[yyyy-MM-dd].[编号].log	<操作对象><操作><集群ID/名称><操作详情><安全级别><操作者><时间>	Resource=test, Operation=Kill, ClusterID/name=tbds_xxx,content=xxxxx, Level=WARN,UserName=storm/hadoop, Time=Tue Mar 10 01:15:35 CST 2015

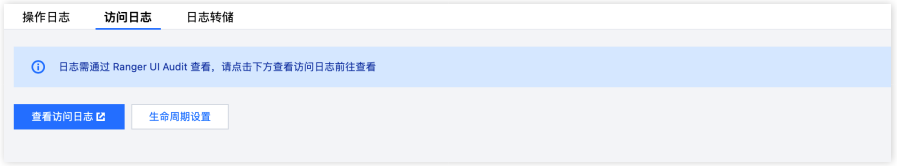
访问日志

操作场景

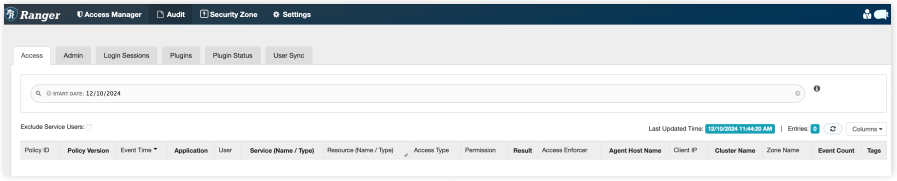
管理员可以通过审计中心的访问日志模块，跳转到Ranger UI Audit进行访问日志查询检索，达到对用户组件级高危操作的跟踪分析和异常定位的目的。

操作步骤

1. 进入审计中心-访问日志，页面功能如下：



2. 点击"查看访问日志"后，会跳转页面进行二次登录，如果集群配置了EIP/VIP地址，需要注意将访问链接的内网ip转换为EIP/VIP。
登录后可直接操作ranger audit页面如下,其中 Access中记录了用户对组件的操作内容和鉴权情况：



3. 生命周期设置
组件审计日志默认保存在TBDS内部 ElasticSearch中间件，默认生命周期为365天，管理员可以修改生命周期（输入限制>0）。

> 说明：
>
> 超出生命周期的审计日志将被清理，不会在审计中心界面中展现。

组件范围

TBDS当前版本已对接Ranger Audit进行日志审计的组件范围包括，审计日志默认不开启，如需开启可以在TM页面上修改对应组件的配置文件，参考开启审计日志：

组件名
HDFS
Hive

Yarn
HBase
Spark
Trino
Kafka

审计日志格式

日志文件名	日志内容格式	示例
component.audit.[yyyy-MM-dd]T[hh-mm-ss.msZ].log	json格式	{"logType":"RangerAudit","reason":null,"cluster":"","policyVersion":20,"agent":"hiveServer2","access":"INSERT","event_count":1,"repo":"hive-tbds-hguf87fe","sess":"ac52588e-e9cc-4148-9f66-b9263a08c5ce","reqUser":"hadoop","seq_num":1,"event_dur_ms":0,"reqData":"INSERT INTO TABLE sales PARTITION (year=2041, month=10) VALUES (1, 'ProductA', 100.0), (2, 'ProductB', 200.0), (3, 'ProductC', 300.0), (4, 'ProductD', 400.0), (5, 'ProductE', 400.0),(6, 'ProductF', 500.0),(7, 'ProductG', 600.0),(8, 'ProductH', 700.0)","result":1,"action":"update","id":"f2756e26-e6b2-412c-8a2d-f03b688d8bc2-0","zoneName":null,"agentHost":"tbds-172-16-0-29-dev","policy":71,"cliIP":"172.16.0.29","resource":"default/sales","resType":"@table","evtTime":"2024-10-28T15:59:41.575Z","tags":[],"repoType":3,"enforcer":"ranger-acl"}

日志转储

操作场景

审计日志默认保存在TBDS自带的mysql和ES数据库中，如果长期保留可能引起数据目录的磁盘空间不足问题，管理员如果需要将审计日志保存到其他归档服务器，可以在日志转储模块设置转储参数及时自动转储，便于管理审计日志信息。

操作步骤

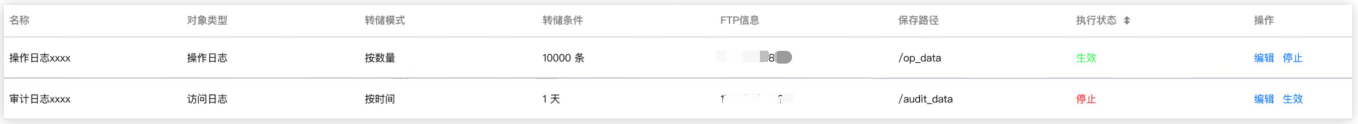
1. 新建转储设置，输入转储名称长度限制：不超过100字符；
   选择类型：TBDS支持对操作日志、组件访问日志的转储；
   转储模式，可以按照：
   时间周期模式：转储时间默认为按天转储前一天数据；
   日志数量模式：默认日志到达指定条数（默认10万条）时开始转储，每个转储文件固定为10w条记录。
   存储路径：FTP路径地址
   IP端口：支持IPv4/IPv6地址格式
   连通性：若输入IP端口或用户密码错误，会导致连通性测试失败，无法保存内容。

   > 说明：
   >
   > 转储记录会进行唯一性检查，相同转储对象类型只能创建一次，查重报错弹窗，访问日志类型："访问日志已存在，请勿重复添加"；操作日志类型："操作日志已存在，请勿重复添加"

2. 查看转储列表
   新建成功后会显示在转储列表中，可以对执行状态进行操作，点击"停止"、"生效"让转储动作进行暂停和启动。

   > 说明：

> 点击生效不会立即让转储执行，转储会按照既定的条件进行定时检查（按天/按量），达到增量标准才会进行转储动作。



名称	对象类型	转储模式	转储条件	FTP信息	保存路径	执行状态	操作
操作日志xxxx	操作日志	按数量	10000 条	[redacted]	/op_data	生效	编辑 停止
审计日志xxxx	访问日志	按时间	1 天	[redacted]	/audit_data	停止	编辑 生效

3. 编辑转储设置

点击列表中“编辑”操作时，会弹出抽屉，对象、类型不能修改，其他内容可编辑；

> 说明：
>
> 弹出抽屉时密码显示为空，每次需要输入密码并确认连通性后才能保存成功。

升级中心

功能介绍

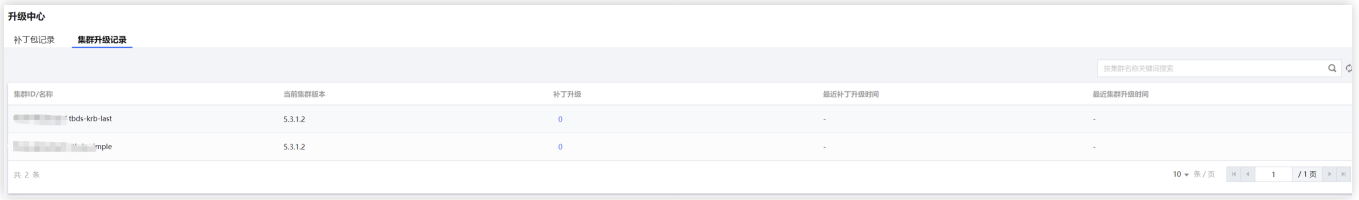
升级中心提供了补丁包及集群升级的信息，用户可以在升级中心中添加、查看补丁包的状态、支持集群版本、补丁版本、补丁包组件、已安装集群、存储容量、上传时间等信息，以及集群版本、集群升级时间等。

操作步骤

1. 登录 TBDS Manager 管理平台，在左侧栏中单击升级中心，进入集群补丁包记录页。
2. 单击添加补丁包， 进入添加补丁包物料界面，输入节点 IP、用户名、密码、物料包路径添加补丁包物料。

3. 查看补丁包记录列表。

4. 单击集群升级记录，查看集群升级信息。

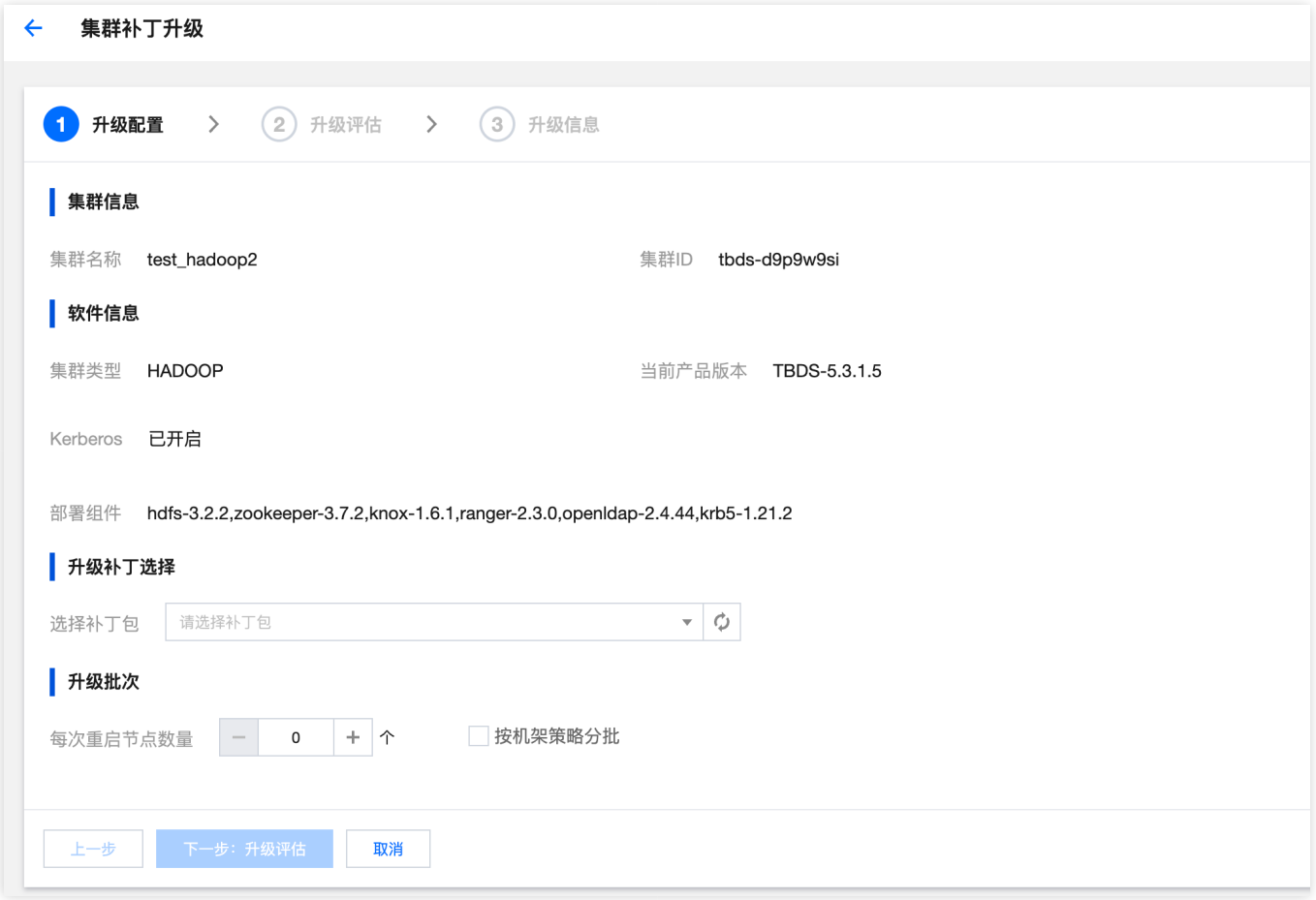


查看补丁包升级记录详情。

5. 选择需要安装升级补丁的集群，点击集群名称跳转到集群列表，选择补丁安装。

6. 选择对应补丁包并按步骤执行.

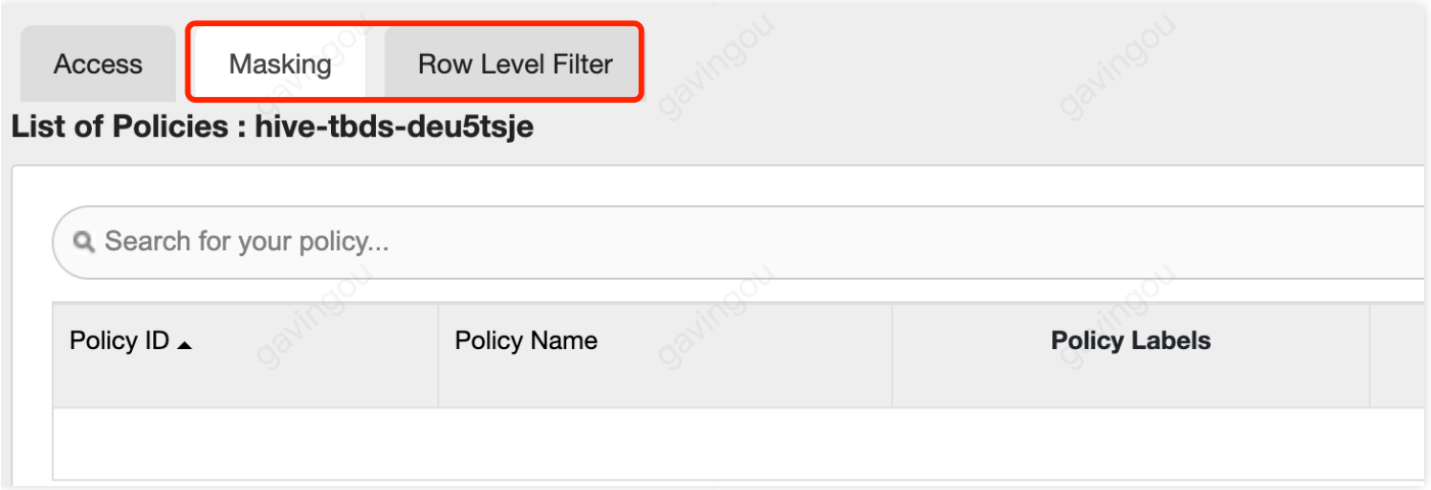
可选择每次重启的节点数量，或按机架策略分批升级。


集群补丁升级
1 升级配置
2 升级评估
3 升级信息
集群信息
集群名称 test_hadoop2
集群ID tbds-d9p9w9si
软件信息
集群类型 HADOOP
当前产品版本 TBDS-5.3.1.5
Kerberos 已开启
部署组件 hdfs-3.2.2,zookeeper-3.7.2,knox-1.6.1,ranger-2.3.0,openldap-2.4.44,krb5-1.21.2
升级补丁选择
选择补丁包 请选择补丁包
升级批次
每次重启节点数量 0 个
按机架策略分批
上一步
下一步：升级评估
取消

安全配置
配置动态脱敏

操作场景

适用于Hive、Spark、Impala或Trino的select操作，指定用户只能获取掩码后的数据。
tbds基于ranger实现动态脱敏，支持行过滤（Row Level Filter）、列屏蔽（Masking）两种脱敏方式，如下



其中，行过滤就是基于where条件做过滤，列屏蔽支持的算法如下表说明
注意：个别算法不支持中文

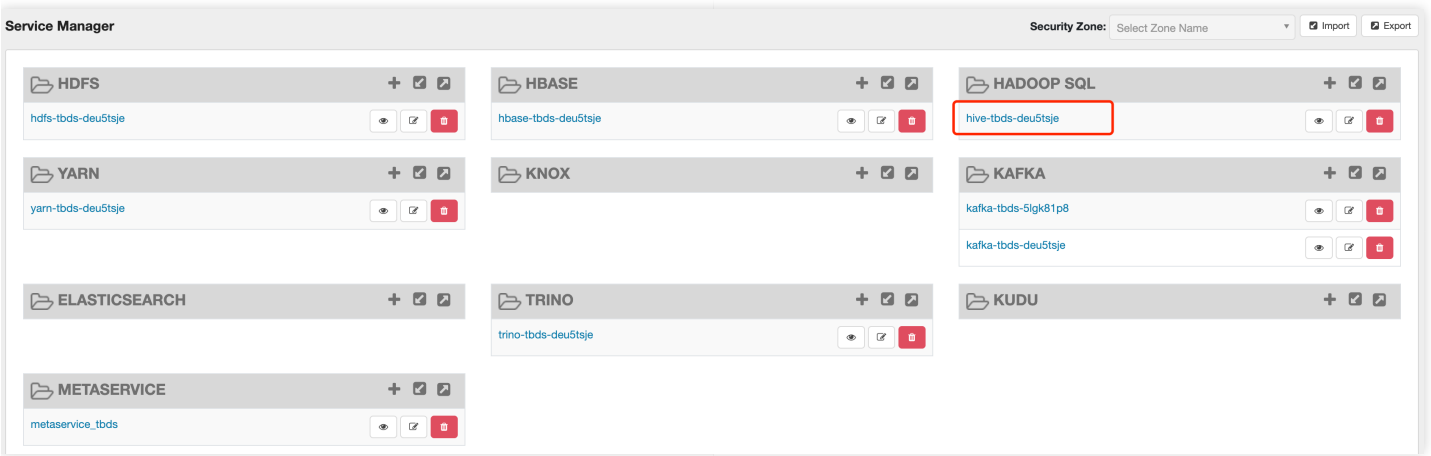
脱敏算法

脱敏算法	Hive	Spark/Impala复用Hive策略	Trino
Redact 用x屏蔽所有字母字符，用n屏蔽所有数字字符	支持	支持	支持
Partial mask: show last 4 只显示最后的4个字符	支持	支持	支持

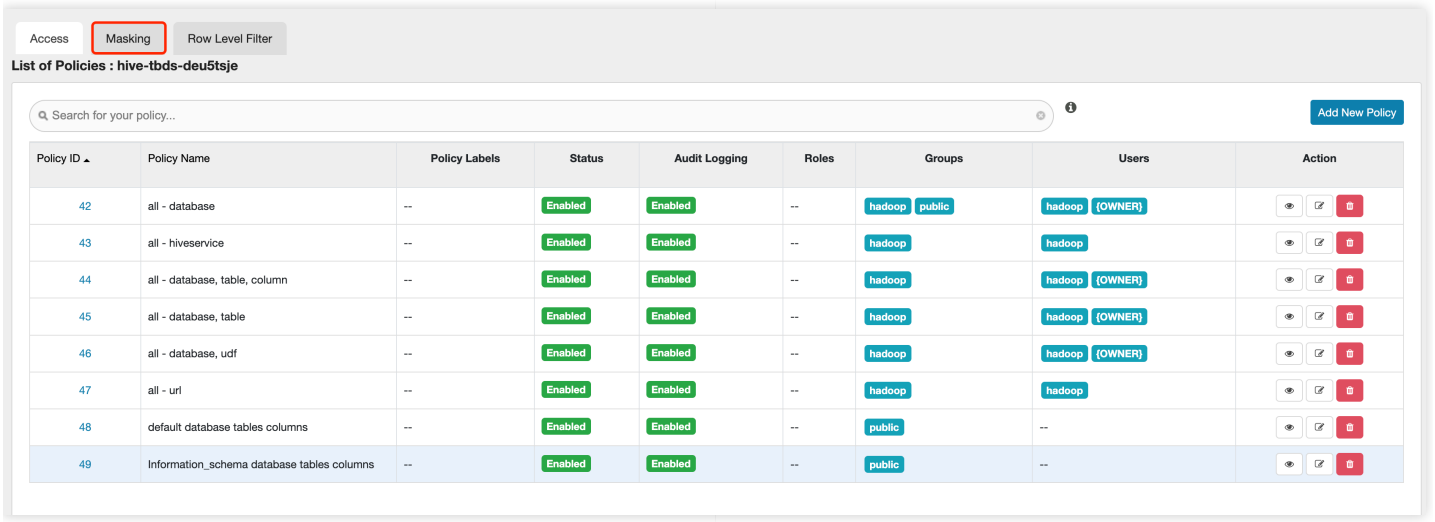
Partial mask: show first 4 只显示开始的4个字符	支持	支持	支持
Hash 用值的哈希值替换原值，仅对文本类型生效	支持	支持	支持
Nullify 替换为NULL值	支持	支持	支持
Unmasked (retain original value) 原样返回	支持	支持	支持
Date: show only year 仅展示年，月份日期设置为01/01	支持	支持	支持
Custom 基于表达式自定义。 例子: cast(concat({col}, “test”) as {type})。 参考：https://cwiki.apache.org/confluence/display/hive/languagemanual+udf#LanguageManualUDF-DataMaskingFunctions 自定义屏蔽可以使用任何有效的 Hive UDF，要求返回与被屏蔽的列中的数据类型相同的数据类型	支持	支持	支持

配置流程

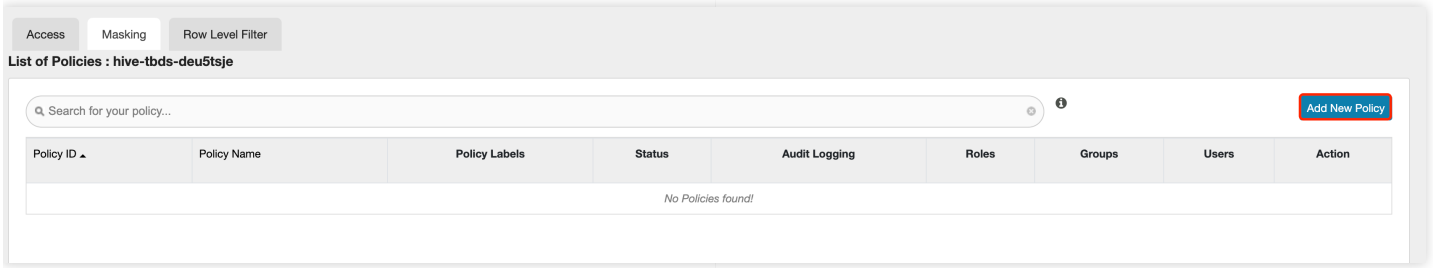
登录到ranger ui，选择hive service或trino service



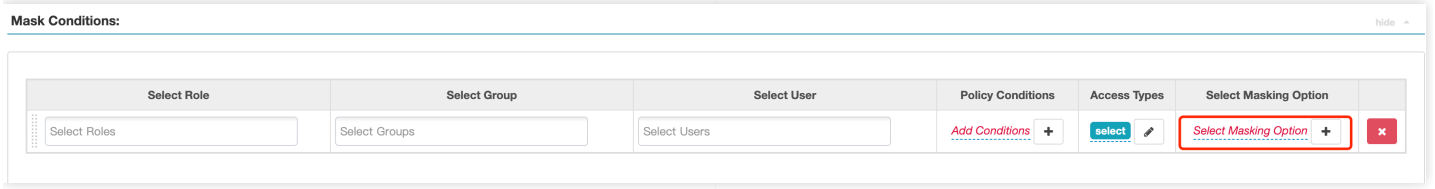
选择脱敏标

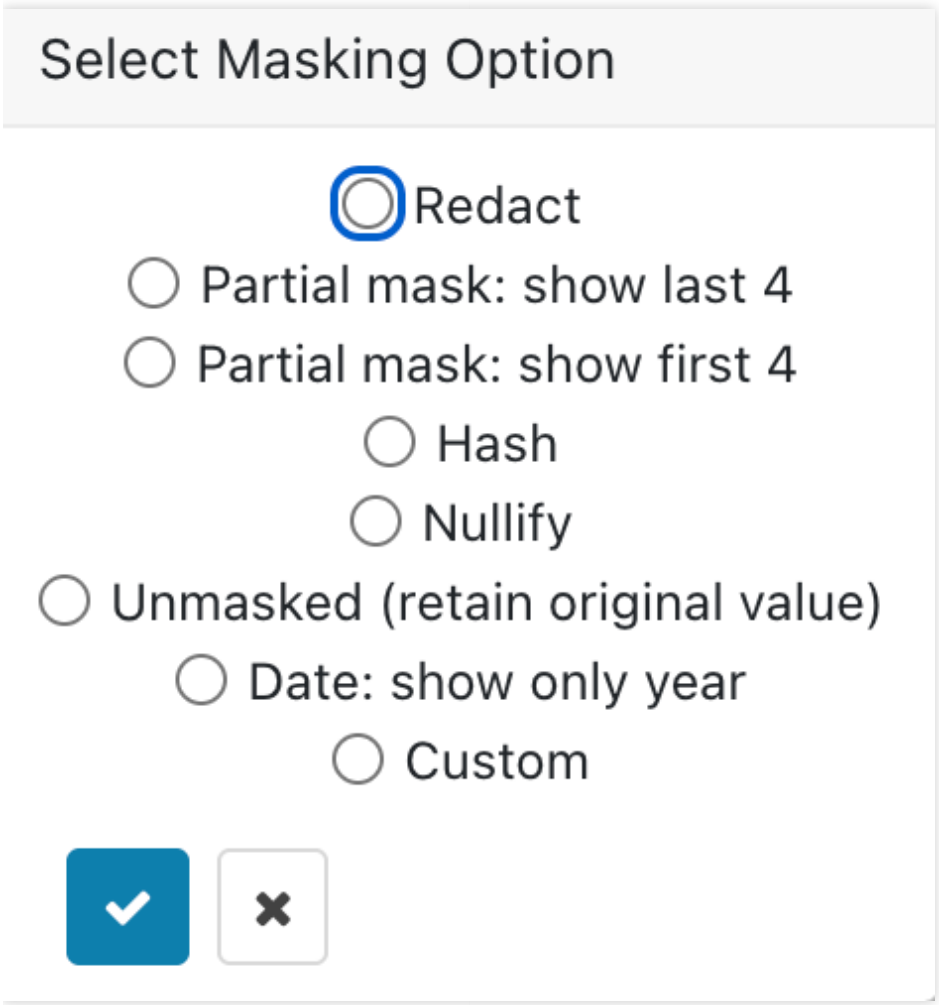


新增脱敏策略

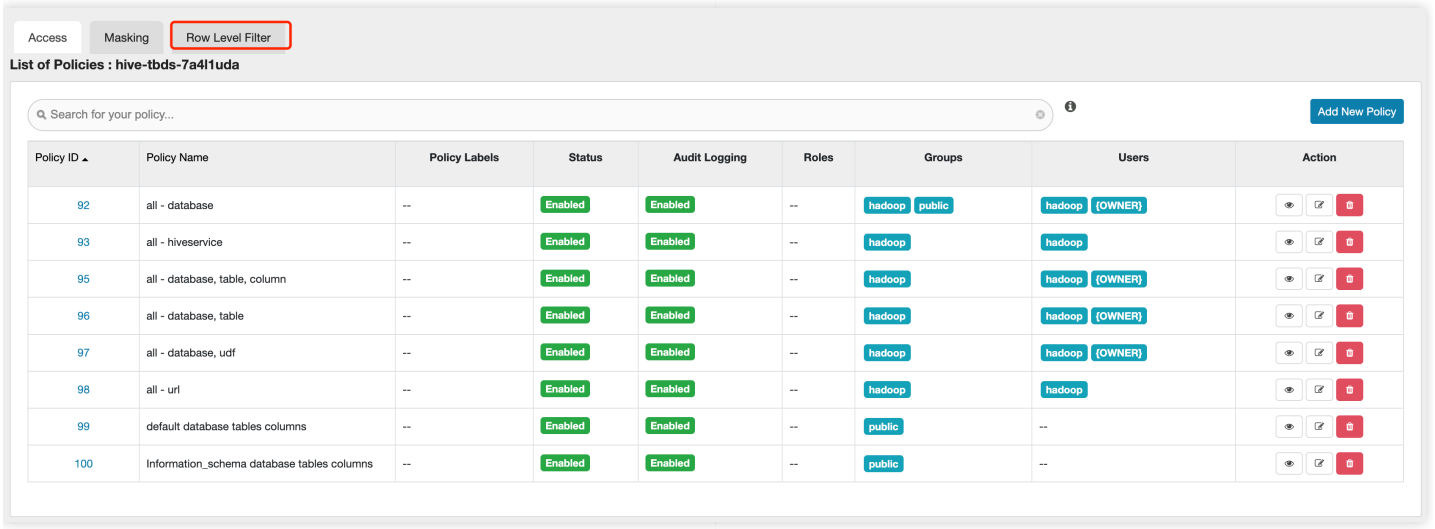


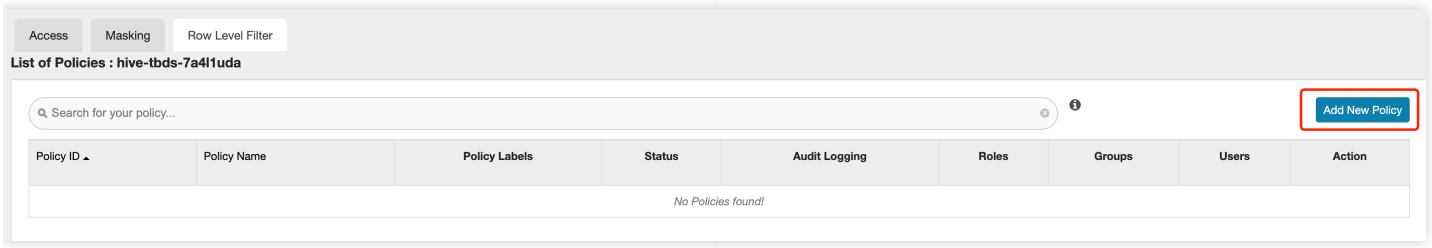
选择脱敏算法

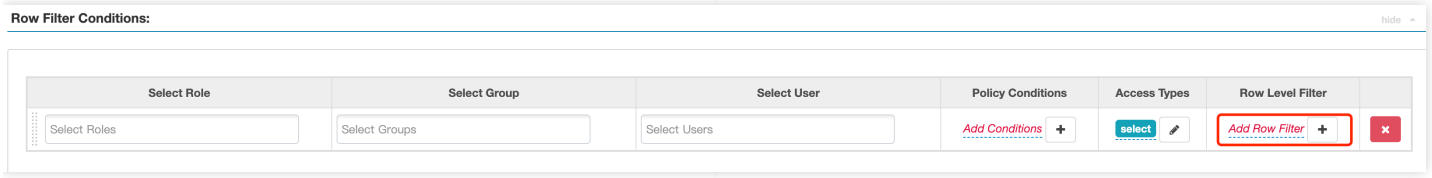



Select Masking Option
Redact
Partial mask: show last 4
Partial mask: show first 4
Hash
Nullify
Unmasked (retain original value)
Date: show only year
Custom

配置行过滤


Access
Masking
Row Level Filter
List of Policies : hive-tbds-7a4l1uda
Search for your policy...
Add New Policy
Policy ID
Policy Name
Policy Labels
Status
Audit Logging
Roles
Groups
Users
Action
92 all - database -- Enabled Enabled -- hadoop public hadoop {OWNER}
93 all - hiveservice -- Enabled Enabled -- hadoop hadoop
95 all - database, table, column -- Enabled Enabled -- hadoop hadoop {OWNER}
96 all - database, table -- Enabled Enabled -- hadoop hadoop {OWNER}
97 all - database, udf -- Enabled Enabled -- hadoop hadoop {OWNER}
98 all - url -- Enabled Enabled -- hadoop hadoop
99 default database tables columns -- Enabled Enabled -- public --
100 Information_schema database tables columns -- Enabled Enabled -- public --





示例

原始数据

```
22      John Doe        2022-01-01 13:14:15
22      222222222222    2024-01-01 13:14:15
22      张三李四        2024-01-01 13:14:15
```

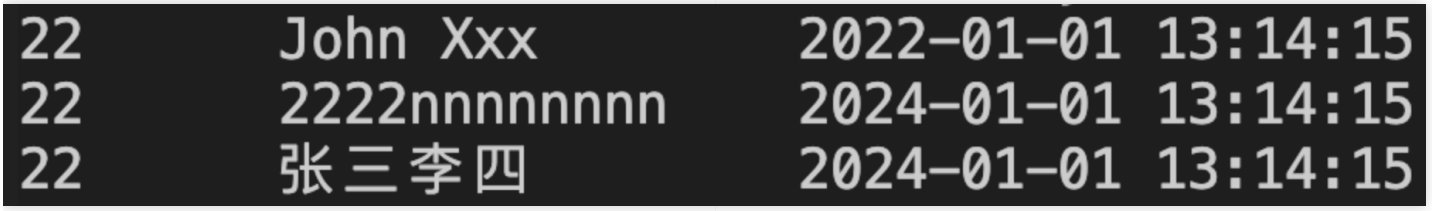
Redact - 用“x”屏蔽所有字母字符，用“n”屏蔽所有数字字符

```
22      Xxxx Xxx        2022-01-01 13:14:15
22      nnnnnnnnnnnn    2024-01-01 13:14:15
22      xxxx    2024-01-01 13:14:15
```

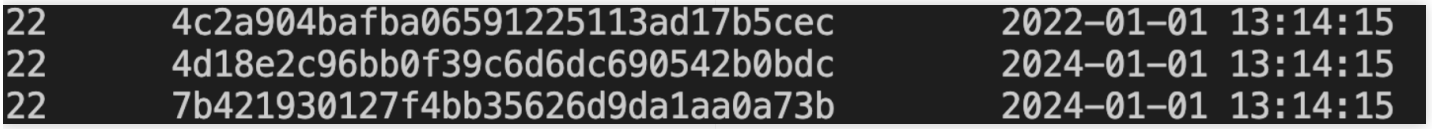
Partial mask - show last 4 只显示最后的4个字符

```
22      Xxxx Doe        2022-01-01 13:14:15
22      nnnnnnnn2222    2024-01-01 13:14:15
22      张三李四        2024-01-01 13:14:15
```

Partial mask - show first 4 只显示开始的4个字符



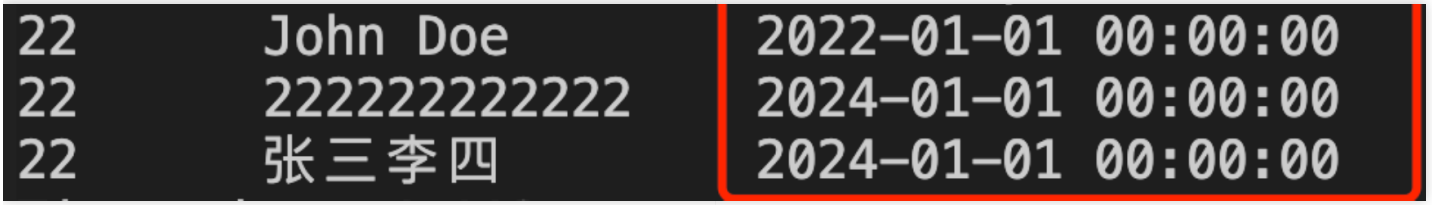
Hash - 用值的哈希值替换原值，仅对文本类型生效



Nullify - 替换为NULL值

Unmasked (retain original value) - 原样返回

Date - show only year 仅展示年，月份日期设置为01/01



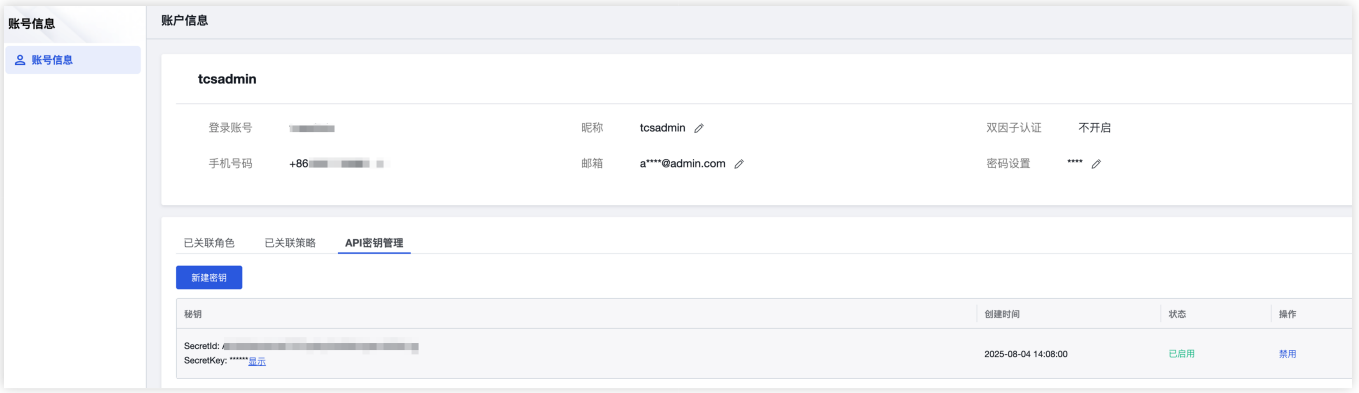
开启云鼎 KMS

> 注意：
>
> TBDS在容器形态下支持选配云鼎KMS，主机形态不支持。KMS密钥需要到独立的管理页面进行配置。

集群默认开启 Ranger KMS，如果需要开启云鼎 KMS（选配），可参考如下配置步骤：

1. 登录管控服务TCS，登录租户端。

2. 获取 API 密钥。



记录下 SecretId 和 SecretKey。

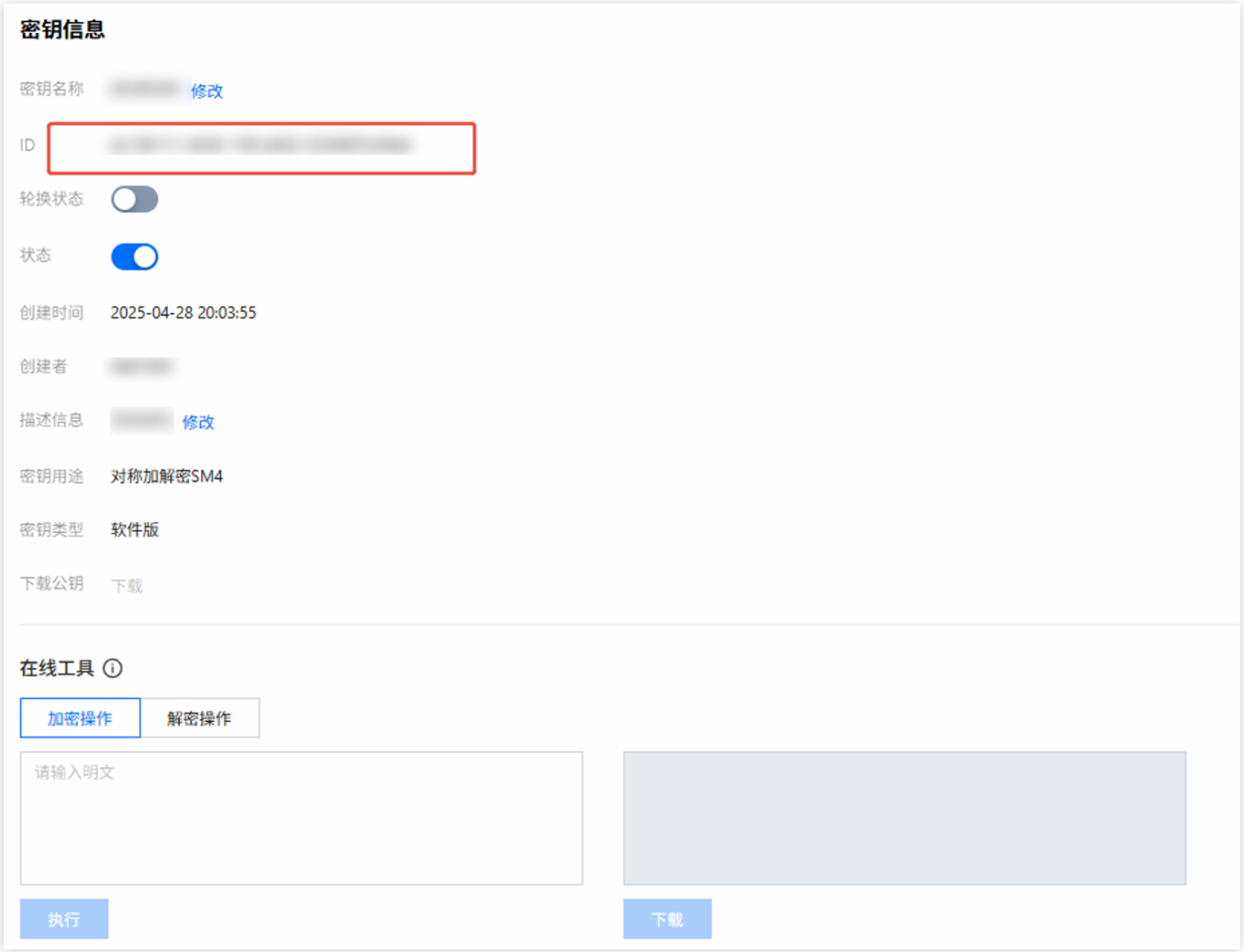
3. 用密钥管理员账号登录密钥管理系统，创建 KMS 主密钥，系统登录网址：http://{IP}:{port}/web/。
点击左侧导航栏用户密钥，进入用户密钥界面。密钥管理员可通过此界面创建和管理密钥信息。

配置参数说明：

配置参数	说明
密钥名称	◦ 必填；同一区域内，密钥的唯一标识； ◦ 密钥名称只能为字母、数字及字符_和-，且不能以 “KMS-” 开头。
描述信息	选填；用来说明您计划保护的数据类型或计划与 CMK 配合使用的应用程序。
密钥用途	必选；选择对称加解密、非对称加解密或者非对称签名验签。
加密算法	当密钥用途为非对称加解密、非对称签名验签时加密算法可选； ◦ 非对称加解密：RSA_2048、Kyber_AES ； ◦ 非对称签名验签：RSA_2048、ECC、Dilithium 。 当密钥用途为对称加解密时，加密算法不可选，默认大陆地区为 SM4，其他地区为 AES_256 。
密钥材料来源	必选；选择密钥生成方式，KMS 生成或者用户自有密钥导入。

> 注意：
>
> 密钥材料来源为外部时，只支持用途为对称加解密。

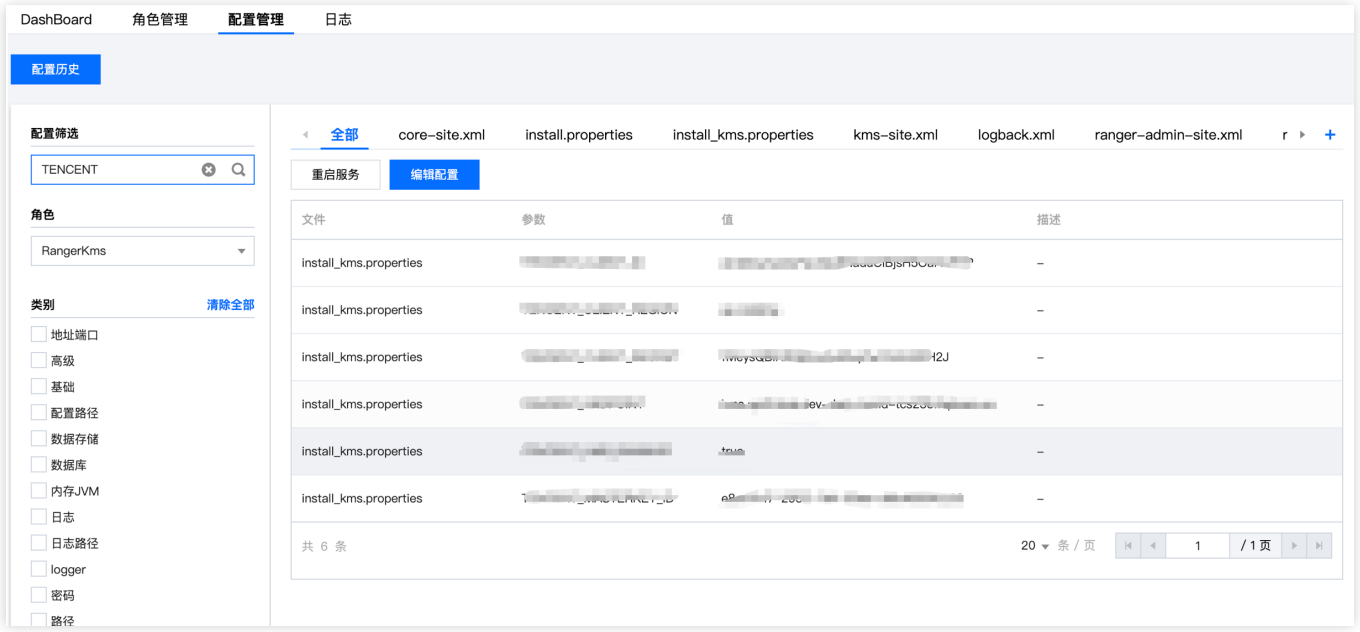
在用户密钥界面，点击目标密钥ID，即可查看该密钥的详细信息。



记录下主密钥 ID。

4. 修改 Ranger KMS 配置。
   install_kms.properties。

```
TENCENT_KMS_ENABLED=true
TENCENT_MASTERKEY_ID=第三步获取的ID
TENCENT_CLIENT_ID=第二步获取的SecretId
TENCENT_CLIENT_SECRET=第二步获取的SecretKey
TENCENT_CLIENT_REGION=没用到,不用管
TENCENT_ENDPOINT=kms.api3.auto-deploy.tcs-master.fsphere.cn # 后四段后缀随环境变化
```



重启 Ranger KMS。

5. 创建 key。

   登录 RangerUI 创建 key。

HDFS 透明加密

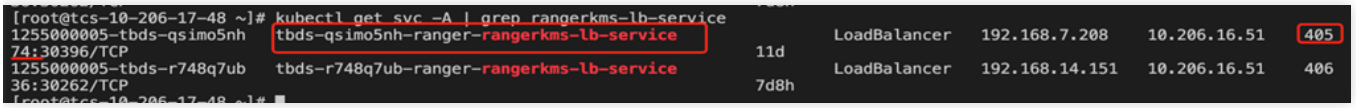
hdfs 配置集成 RangerKMS

修改 hdfs-site 配置：

dfs.encryption.key.provider.uri=kms://http@${KMS_DOMAIN}:${KMS_PORT}/kms

${KMS_DOMAIN}为 公共安全集群id.ranger.rangerkms.vip.com，比如qsimo5nh.ranger.rangerkms.vip.com

${KMS_PORT}为 公共安全集群对应的 rangerkms-lb-service的端口



修改完配置后重启 HDFS 服务和 YARN 服务。

登录 ranger ui

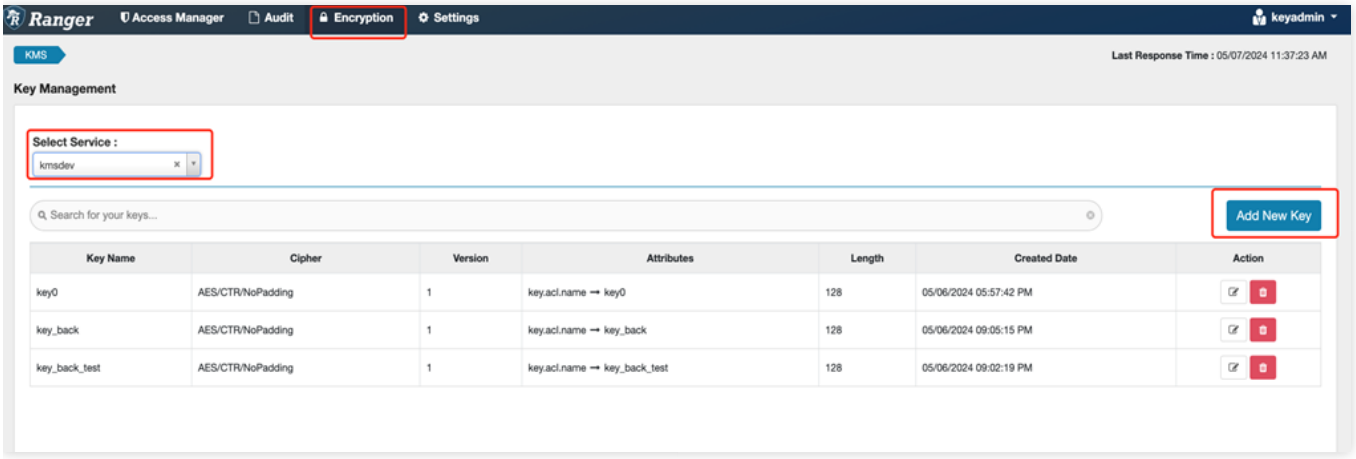
登录 ranger ui 页面

账号：keyadmin。

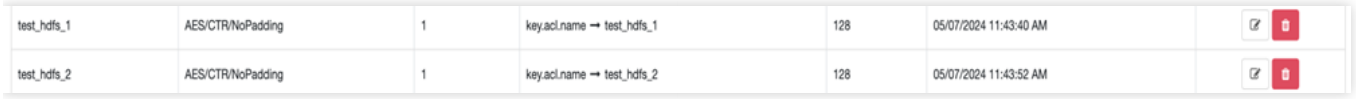
密码：keyadmin。

在【Encryption】页面可创建 key。

选择【Access Manager】中对应的服务名称，比如 kmsdev。



新增Key：



创建加密区

hadoop fs -mkdir /test_kms

hadoop fs -mkdir /test_kms/test_1

hadoop fs -mkdir /test_kms/test_2

加密区1：hdfs crypto -createZone -keyName test_hdfs_1 -path /test_kms/test_1

加密区2：hdfs crypto -createZone -keyName test_hdfs_2 -path /test_kms/test_2

上传对应的文件至加密区1和2，hadoop 用户可直接查看文件内容。

```
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# hadoop fs -cat /test_kms/test_1/1
2024-05-07T16:03:45,940 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-05-07T16:03:46,198 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-05-07T16:03:46,247 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
11111111111111
111111111111111
1111111111111111
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# hadoop fs -cat /test_kms/test_2/2
2024-05-07T16:03:53,212 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-05-07T16:03:53,471 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-05-07T16:03:53,520 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
222222222222
2222222222222
22222222222222
```

加密区文件查看

使用 Kudu 用户，无权限查看：

```
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# kinit -kt /var/krb5kdc/emr.keytab kudu/tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0@TBDS-C
2O588YD
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# hadoop fs -cat /test_kms/test_2/2
2024-05-07T16:05:53,280 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-05-07T16:05:53,507 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-05-07T16:05:53,558 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
cat: User:kudu not allowed to do 'DECRYPT_EEK' on 'test_hdfs_2'
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# hadoop fs -cat /test_kms/test_1/1
2024-05-07T16:05:58,920 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-05-07T16:05:59,135 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-05-07T16:05:59,183 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
cat: User:kudu not allowed to do 'DECRYPT_EEK' on 'test_hdfs_1'
```

ranger ui 上新增 Kudu 用户对 test_hdfs_1 的 key 的解密权限，然后再以 kudu 用户去访问，可以查看 test_hdfs_1 对应的加密区1的文件，但是无法访问加密区2的文件。

```
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# kinit -kt /var/krb5kdc/emr.keytab kudu/tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0@TBDS-C
2O588YD
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]#
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]#
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# hadoop fs -cat /test_kms/test_1/1
2024-05-07T16:15:42,013 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-05-07T16:15:42,227 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-05-07T16:15:42,276 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
11111111111111
111111111111111
1111111111111111
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# hadoop fs -cat /test_kms/test_2/2
2024-05-07T16:15:47,849 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-05-07T16:15:48,064 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-05-07T16:15:48,112 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
cat: User:kudu not allowed to do 'DECRYPT_EEK' on 'test_hdfs_2'
```

底层文件查看

1. hdfs 快照目录文件为加密内容：

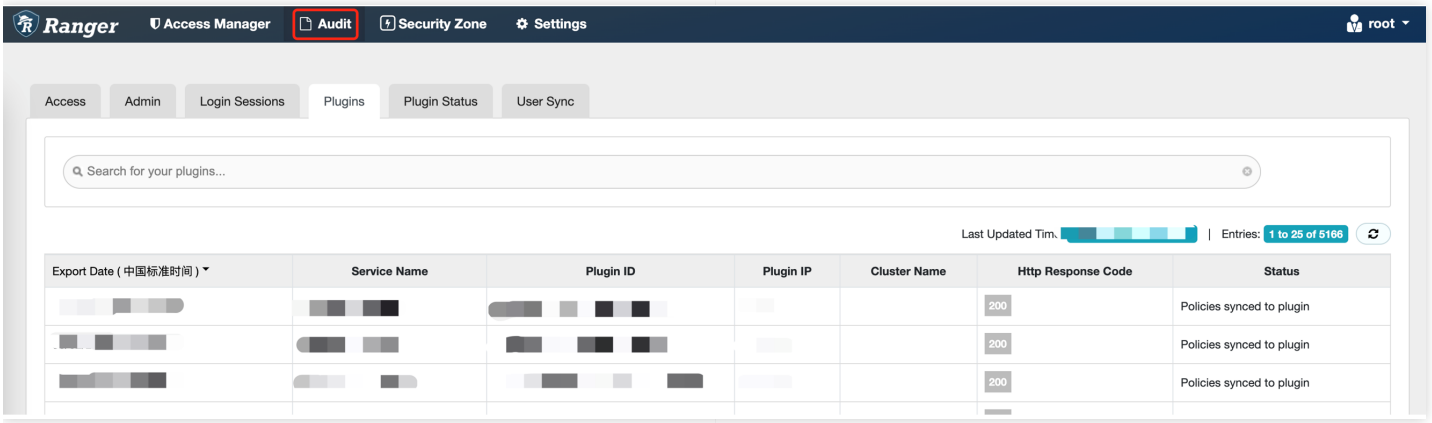
```
[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# hadoop fs -cat /.reserved/raw/test_kms/test_1/1
2024-05-07T16:11:33,413 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-05-07T16:11:33,632 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-05-07T16:11:33,681 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
H6666666P2@.]+0q0      66666]/680a\666666A/66  [root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]# hadoop fs -cat /.reserved/raw/test_kms/
test_2/2
2024-05-07T16:11:39,006 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-05-07T16:11:39,235 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-05-07T16:11:39,285 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
&(V"0xF0^0O[root@tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0 ~]#
```

2. 查看 datanode 数据目录的文件，为加密内容：

```
[root@tbds-2402-4e00-140b-4c01-0-9b9f-8417-54e6 ~]# cat /data/qcloud/data/hdfs/current/BP-564943269-\[2402.4e00.140b.4c01.0.9b9f.7058.e5f0\
]-1714275594040/current/finalized/subdir0/subdir31/blk_1073757973
[root@tbds-2402-4e00-140b-4c01-0-9b9f-8417-54e6 ~]#o^Ctbds-2402-4e00-140b-4c01-0-9b9f-8417-54e6 ~]#
[root@tbds-2402-4e00-140b-4c01-0-9b9f-8417-54e6 ~]# ^C
[root@tbds-2402-4e00-140b-4c01-0-9b9f-8417-54e6 ~]# ^C
[root@tbds-2402-4e00-140b-4c01-0-9b9f-8417-54e6 ~]# cat /data/qcloud/data/hdfs/current/BP-564943269-\[2402.4e00.140b.4c01.0.9b9f.7058.e5f0\
]-1714275594040/current/finalized/subdir0/subdir31/blk_1073757973
H6666666P2@.]+0q0      66666]/680a\666666A/66  [root@tbds-2402-4e00-140b-4c01-0-9b9f-8417-54e6 ~]#
[root@tbds-2402-4e00-140b-4c01-0-9b9f-8417-54e6 ~]#
[root@tbds-2402-4e00-140b-4c01-0-9b9f-8417-54e6 ~]# hdfs fsck /test_kms/test_1/1 -files -blocks -locations
2024-05-07T16:28:29,519 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-05-07T16:28:29,767 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-05-07T16:28:29,816 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
Connecting to namenode via https://tbds-2402-4e00-140b-4c01-0-9b9f-7058-e5f0:4009/fsck?ugi=hadoop&files=1&blocks=1&locations=1&path=%2Ftest
_kms%2Ftest_1%2F1
FSCK started by hadoop (auth:KERBEROS_SSL) from /2402:4e00:140b:4c01:0:9b9f:8417:54e6 for path /test_kms/test_1/1 at Tue May 07 16:28:28 CS
T 2024

/test_kms/test_1/1 48 bytes, replicated: replication=3, 1 block(s):  OK
0. BP-564943269-[2402.4e00.140b.4c01.0.9b9f.7058.e5f0]-1714275594040:blk_1073757973_17254 len=48 Live_repl=3  [DatanodeInfoWithStorage[[240
2:4e00:140b:4c01:0:9b9f:8417:54e6]:4001,DS-b46a2435-0460-4c07-b300-25698aaa812b,DISK], DatanodeInfoWithStorage[[2402:4e00:140b:4c01:0:9b9f:
83f8:c686]:4001,DS-28204cc9-6131-4e40-bfa1-d85a76981f69,DISK], DatanodeInfoWithStorage[[2402:4e00:140b:4c01:0:9b9f:83c6:a5e0]:4001,DS-a4cff
32f-cd7f-4837-acca-54929c71211f,DISK]]
```

开启组件审计日志

TBDS 支持对以下组件开放审计日志：HDFS，YARN，Hive，HBase，Spark，Kafka，Kudu，Trino。审计日志默认不开启，如需开启可以在 TM 页面上修改对应组件的配置文件。
以 YARN 为例，在配置文件中找到 ranger-yarn-audit.xml，修改 xasecure.audit.destination.elasticsearch 的值为 true，然后重启组件。在 Ranger UI 的审计页面可以看到审计信息。



开发者指南
开发规范约束

核心原则

为保障大数据集群的稳定性、数据完整性及服务连续性，所有操作必须遵循以下核心原则：

1. 严格版本配套原则：严禁使用非当前集群版本配套的客户端（Client/SDK）连接服务。
2. 最小权限原则：禁止在非必要情况下使用hadoop、root等超级用户身份运行业务代码。
3. 变更窗口原则：涉及元数据变更、重启、配置修改的操作必须在规定的维护窗口内进行。

严禁操作项

> 警告：
>
> 违反以下规定导致的服务故障、数据丢失，属于人为责任事故，不在平台 SLA 保障范围内。

客户端与版本管理

【禁止】使用开源社区版本客户端：严禁直接下载 社区版 Jar 包或客户端工具连接TBDS集群。厂商版本通常包含私有补丁、安全加固及协议修改，混用极易导致协议握手失败或隐蔽的数据错误。
【禁止】跨版本操作（低配高）：严禁使用低版本 TBDS 客户端操作高版本集群服务。
【禁止】使用未经验证的第三方工具：严禁使用未经平台官方认证的 GUI 管理工具（如某些开源的 Kafka Manager、HBase UI 工具）直接对生产环境进行写操作。

元数据与存储安全

【禁止】直接操作底层元数据：严禁绕过组件 API，直接修改 Zookeeper 中的 Znode 数据、MySQL 中的 Hive Metastore 表数据或 HDFS 上的 `fsimage / edits` 文件。
【禁止】非标路径写入：严禁在 HDFS 的 `/tmp` 、 `/user/hive/` 等系统关键目录下随意存放临时数据或业务数据，必须规划独立的业务目录。
【禁止】关闭安全认证：在开启 Kerberos 的集群中，严禁在业务代码中硬编码关闭认证检查或试图绕过安全协议。

Kafka 专项规范

【禁止】低版本客户端管理高版本 Topic：

- 风险：使用低版本客户端执行 `Add Partition` 操作，会导致新分区缺乏 Topic ID。当集群重启或发生 Leader 切换时，Broker 间因元数据不一致拒绝同步，导致 ISR 列表收缩、消费组卡死、部分分区不可用。
- 规范：所有 Producer/Consumer 及管理脚本（ `kafka-topics.sh` ）必须使用与 Broker 版本完全一致的客户端包。

HBase 专项规范

- 【禁止】 在业务高峰期手动触发 Major Compaction 。
- 【禁止】 修改列族属性时未同步修改所有 Region，可能导致 Split 失败。

Hive/Spark 专项规范

- 【禁止】 开启过大的动态分区（Dynamic Partition），如一次 Insert 产生超过 2000 个分区，会压垮 NameNode。
- 【禁止】 在 Driver 端收集过大结果集（如 collect() 操作），会导致 Driver OOM。

慎重操作项

以下操作需经过技术团队评审，并在低峰期执行。

资源与性能

【慎重】全表扫描与无索引查询：

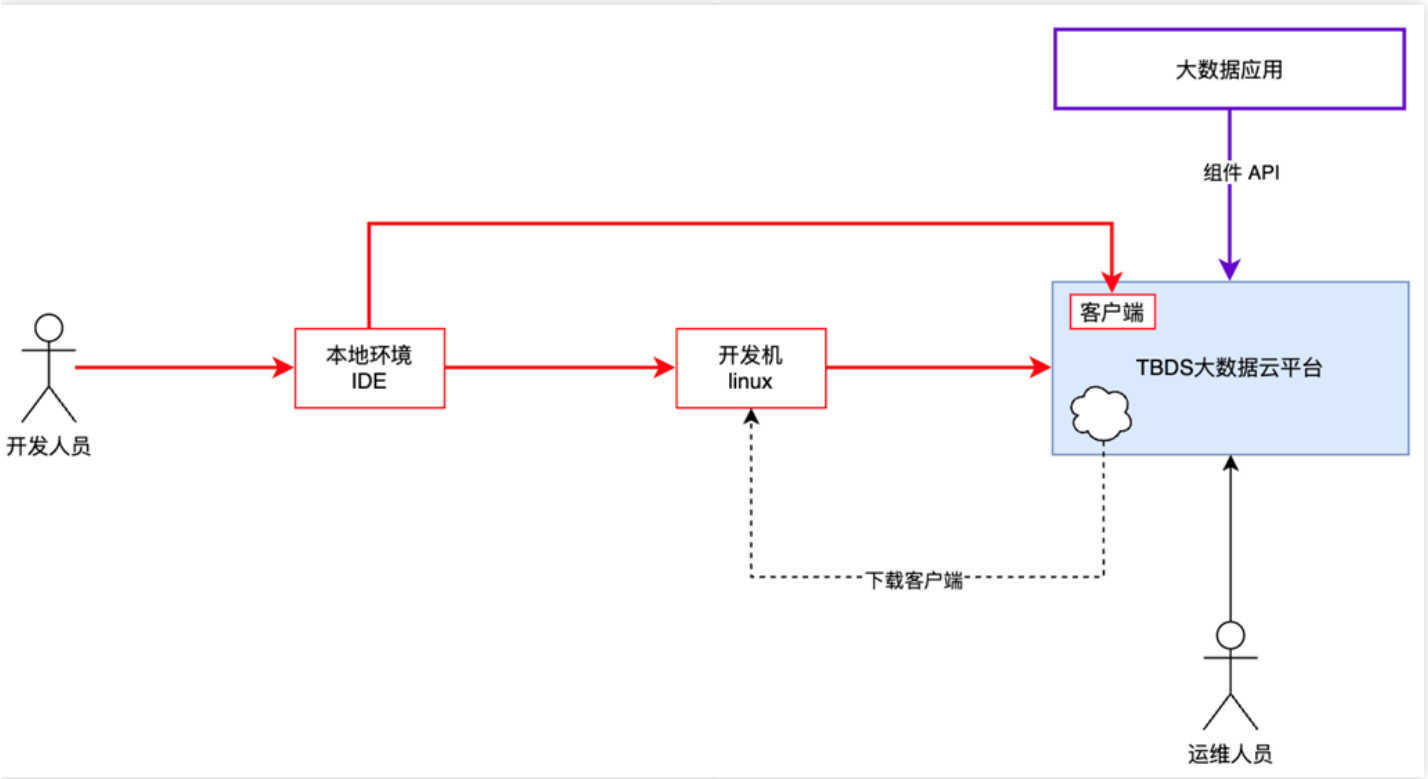
- Hive/Spark SQL 禁止在无分区过滤的情况下对 PB 级大表进行全表扫描。
- HBase 禁止在无 RowKey 规划的情况下使用 Scan 操作或使用非前缀匹配的 Filter。
  【慎重】高并发小文件写入：避免在 HDFS/Hive 中生成大量（百万级）低于 128MB 的小文件，这会耗尽 NameNode 内存并拖垮集群性能。
  【慎重】自定义代码（UDF/Coprocessor）：
- Hive UDF 必须经过内存泄漏测试。
- HBase Coprocessor部署前必须在测试环境验证，错误的协处理器会导致 RegionServer 级联宕机。

运维变更

【慎重】强制重启与 Kill：严禁使用 kill -9 强制杀停有状态组件（HDFS, Kafka, HBase, ZK）的进程，必须使用标准的 stop 脚本或平台控制台操作，否则可能导致数据损坏。
【慎重】大规模数据删除：执行 hadoop fs -rm -r 或 DROP TABLE 前必须进行“双人复核”，建议开启 HDFS 回收站功能。

应用开发流程
开发调试模式

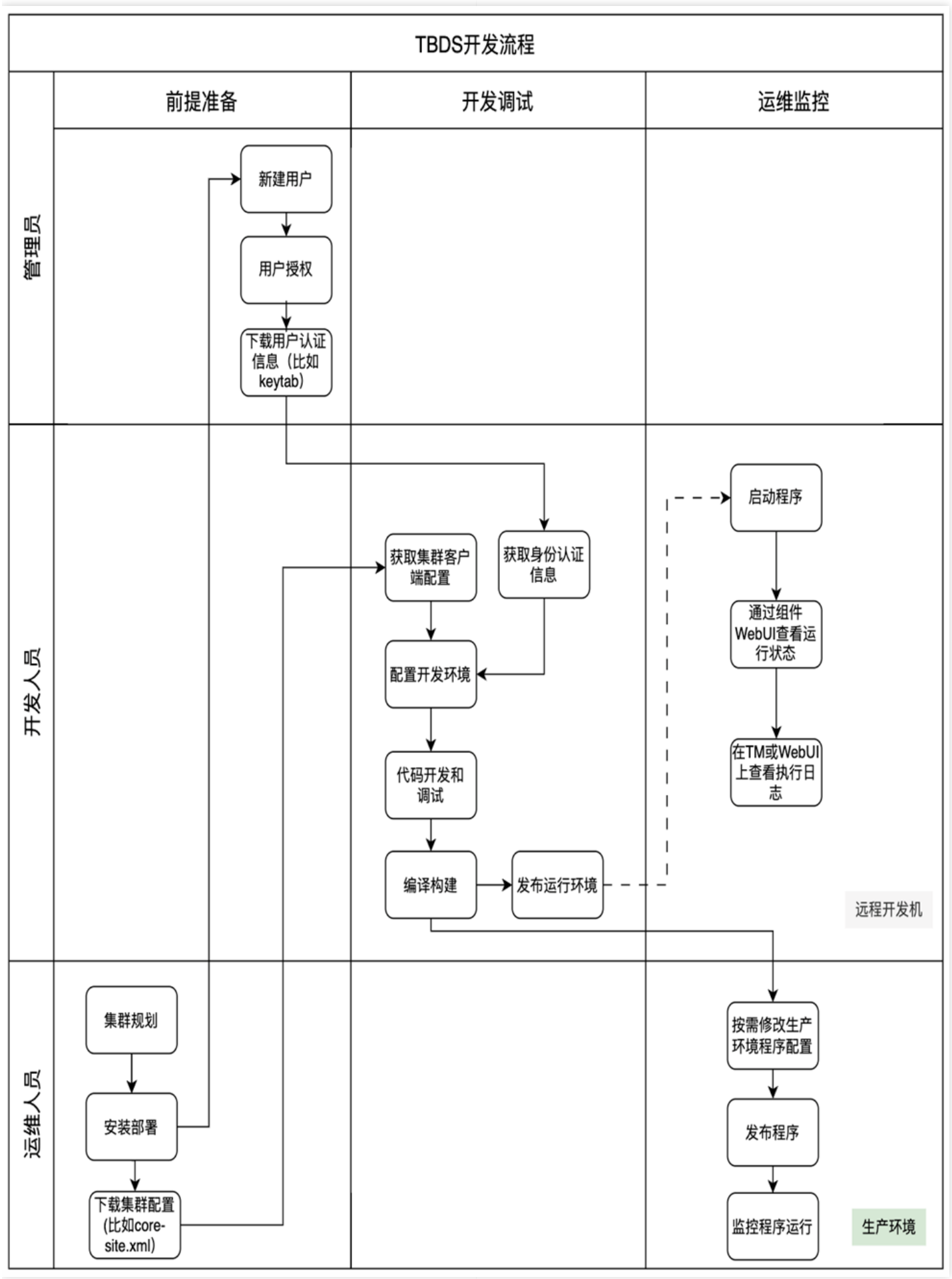
TBDS为应用开发商提供与开源社区完全兼容的组件API,使应用开发商在大数据开发过程中获得与开源社区一致的体验。



基于TBDS 平台的大数据应用开发调试主要有两种方式：

- 本地调试：开发人员使用自己的 IDE 在本地完成代码的开发和调试。如果需要访问 TBDS 平台上的集群资源,需要在本地配置好相关的连接方式。这种方式门槛低,适用于简单应用场景。
- 远程调试：对于依赖复杂或涉及性能优化的大数据应用,这时开发人员需要先在本地开发完成代码,然后将应用程序打包上传到 TBDS 提供的开发机进行调试,直至应用调试稳定后再进行生产部署。这种方式可以提供与生产环境一致的运行环境,适用于对资源和性能要求较高的复杂应用开发。

开发发布流程


TBDS开发流程
前提准备
开发调试
运维监控
管理员
开发人员
运维人员
新建用户
用户授权
下载用户认证信息（比如keytab）
获取集群客户端配置
获取身份认证信息
配置开发环境
代码开发和调试
编译构建
发布运行环境
启动程序
通过组件WebUI查看运行状态
在TM或WebUI上查看执行日志
远程开发机
集群规划
安装部署
下载集群配置(比如core-site.xml)
按需修改生产环境程序配置
发布程序
监控程序运行
生产环境

TBDS 开发流程可分为准备、开发调试、运维监控三个阶段：

- 准备阶段：包括集群准备、用户准备、开发环境配置等，主要目的是让开发者准备好开发所需的用户身份、集群信息、开发环境等前提条件。
- 开发调试阶段：这个阶段开发者基于准备好的环境，通过代码开发、编译构建、上传到开发机、查看执行情况等步骤完成程序开发和调试。
- 运维监控阶段：开发好的程序部署到开发机，开发者可以通过各种运维监控工具查看程序运行状态、执行日志等信息，并可以按需修改和更新程序配置。

开发环境准备
使用maven仓库
TBDS Maven 下载离线包

背景

一些客户可能开发环境不能连接公网，这里提供了下载 TBDS Maven 离线包的方案，下载好之后客户可按需上传到内部 Maven 仓库使用。

下载离线包

新建目录存放 Jar。

```
mkdir tbds-jars
cd tbds-jars
```

创建 POM 文件 pom.xml，声明包依赖（在 dependencies 中填写想下载的依赖包），POM 中加上 TBDS 的 Maven 仓库源，示例如下：
vim pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-client</artifactId>
  <version>3.2.2-TBDS-5.3.1.2</version>

  <dependencies>
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-common</artifactId>
      <version>3.2.2-TBDS-5.3.1.2</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-client-api</artifactId>
      <version>3.2.2-TBDS-5.3.1.2</version>
    </dependency>
```

```
  </dependencies>

  <repositories>
    <repository>
      <id>tbds2</id>
      <url>https://tbdsrepo.cloud.tencent.com/repository/maven-public/</url>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
      </snapshots>
      <releases>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
      </releases>
    </repository>
  </repositories>
</project>
```

执行下面命令，下载POM声明的所有包及所有相关依赖及对应的 POM 文件到当前目录。

```
mvn dependency:copy-dependencies -Dmdep.copyPom=true -DoutputDirectory=./
```

如果不想下载包相关依赖，只下载对应包，执行：

```
mvn dependency:copy -Dartifact=org.apache.hadoop:hadoop-common:3.2.2-TBDS-5.3.1.2 -DoutputDirectory=./
```

dependency:copy 命令只能单条下载一个包，且不能下载 POM文件，要下载多个包需执行多遍。

使用TBDS的公共maven开发库

为了让代码程序能够顺利在TBDS大数据集群上提交并运行，建议使用TBDS提供的maven仓库。部分客户端依赖JAR包只有在TBDS maven仓库中才能拉取到。

云服务商对外提供了 TBDS 的公共 maven 仓库，repository 地址为: https://tbdsrepo.cloud.tencent.com/repository/maven-public/
如果是配置 maven 的 settings.XML 配置文件，则典型的配置片段样例：

```
<profiles>
<!-- define a profile and tbds public repository -->
  <profile>
  <id>tbds-profile</id>
    <repositories>
      <repository>
        <id>tbds_public</id>
        <name>tbds_public_repository</name>
        <url>https://tbdsrepo.cloud.tencent.com/repository/maven-public/</url>
        <layout>default</layout>
       </repository>
      <repository>
        <id>maven_public</id>
        <name>maven_public_repository</name>
        <url>https://tbdsrepo.cloud.tencent.com/repository/maven_public/</url>
        <layout>default</layout>
       </repository>

    </repositories>
  </profile>
</profiles>

 <activeProfiles>
 <!-- make sure tbds-profile active -->
   <activeProfile>tbds-profile</activeProfile>
 </activeProfiles>
```

以下是完整的样例：

```
<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/sett
ings-1.0.0.xsd">
```

```
<!-- localRepository
 | The path to the local repository maven will use to store artifacts.
 |
 | Default: ${user.home}/.m2/repository
<localRepository>/path/to/local/repo</localRepository>
-->
<pluginGroups>

</pluginGroups>

<proxies>

</proxies>

<servers>

</servers>
<!--
<mirrors>
   <mirror>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
     <url>http://repo1.maven.org/maven2/</url>
     <mirrorOf>central</mirrorOf>
   </mirror>
   <mirror>
      <id>repo2</id>
     <mirrorOf>central</mirrorOf>
      <name>Human Readable Name for this Mirror.</name>
     <url>http://repo2.maven.org/maven2/</url>
   </mirror>
</mirrors>
-->
<profiles>
  <!-- define a profile and tbds public repository -->
   <profile>
       <id>tbds-profile</id>
     <repositories>
         <repository>
           <id>tbds_public</id>
           <name>tbds_public_repository</name>
           <url>https://tbdsrepo.cloud.tencent.com/repository/maven-public/</url>
           <layout>default</layout>
         </repository>
         <repository>
           <id>maven_public</id>
```

```
            <name>maven_public_repository</name>
            <url>https://tbdsrepo.cloud.tencent.com/repository/maven-public/</url>
            <layout>default</layout>
          </repository>
        </repositories>
      </profile>
    </profiles>

    <activeProfiles>
    <!-- make sure tbds-profile active -->
      <activeProfile>tbds-profile</activeProfile>
    </activeProfiles>

  </settings>
```

如果开发者使用的是 maven 进行程序开发和编译，那么需要：
步骤1：替换 ${maven_home}\conf\settings.XML 文件为我们上面提供的settings.xml 文件。 主要是让 maven 能够识别到 TBDS 的公共 maven 库;${maven_home}表示的是开发机上面 maven 的安装目录。比如D:\Program Files\apache-maven-3.3.9\。需要换成开发者自己部署maven 的目录。
步骤2：如果是 IDEA 进行代码编写和开发，还需要将 IDEA 里面设置 settings.XML 的配置 修改成最新的 settings.xml 路径。

如何编写Pom.xml文件

TBDS 的 maven 仓库规则：部分 jar、pom 等文件在开源社区的文件名后面增加了 TBDS-5.3.1.X 后缀，可以参考后面样例工程代码中的pom.xml添加依赖。

这里举个例子：比如 hbase-client-1.2.1.jar 社区jar包修改为 hbase-client-1.2.1-TBDS-5.3.1.X.jar。因此，在工程代码编译的时候，在 pom.xml 文件里面建议将开源的依赖替换为 TBDS 的依赖。即将开源的依赖：

```
<dependency>
  <groupId>org.apache.hbase</groupId>
  <artifactId>hbase-client</artifactId>
   <version>2.4.5</version>
</dependency>
```

改为：

```
<dependency>
  <groupId>org.apache.hbase</groupId>
  <artifactId>hbase-client</artifactId>
  <version>2.4.5-TBDS-5.3.1.X</version>
</dependency>
```

部署TBDS客户端

> 警告：
>
> 在 TBDS 环境中进行任何组件（包括但不限于 Kafka, HBase, Hive, Spark, Elasticsearch 等）的运维管理、数据读写或应用开发时，必须严格使用当前 TBDS 版本所提供的配套客户端及 SDK。
>
> - 严禁使用开源社区下载的通用客户端直接连接 TBDS 组件。
> - 严禁使用低版本 TBDS 的客户端操作高版本 TBDS 的服务（跨版本混用）。
> - 严禁在未确认兼容性的情况下，使用第三方工具进行开发应用操作。
>   原则： 需下载使用与TBDS 集群同版本的控制台/安装包中提供的 Client。

在提交程序到大数据集群之前，首先需要在开发机上面部署 TBDS 自带的客户端， 针对应用开发者的开发机，有两种情况:

TBDS 集群内开发机

应用程序提交以及运行在 TBDS 大数据集群内的服务器， 因为大数据集群内的服务器默认都安装了客户端，不需要再单独安装。

TBDS 集群外开发机

应用程序提交以及运行在 TBDS 大数据集群外的服务器， 因为大数据集群外的服务器并不受 TBDS 管理，因此无法自动安装上客户端。
对于集群外服务器部署 TBDS 的客户端，有两种方式：

方式1：

直接安装 TBDS 的hdfs、hive、hbase、yarn、spark、kafka 等 组件的客户端，具体安装方法如下：
下载安装工具：https://tbds-mirror-1258756906.cos.ap-guangzhou.myqcloud.com/tools/TBDS_clients_install_tools.tgz
工具组成：每个版本均提供如下组成部分：
|-- install-TBDS-client.sh //客户端安装脚本
|-- prepare-TBDS-client-installpackage.sh //客户端安装包准备工具

> 注意：

> 以上文档链接如无权限请联系运维工程师获取。

客户端安装包准备工具：客户端安装包准备工具即prepare-TBDS-client-installpackage.sh，在TBDS集群内任意一台完整安装了hdfs,hbase,hive,spark,kafka,hbase,tez等所有客户端的节点执行，如图所示，直接执行，会运行脚本，到最终执行成功：

```
function red() {
    echo -e "\033[31m$*\033[0m" >&2
}

function green() {
    echo -e "\033[32m$*\033[0m"
}

function yellow() {
    echo -e "\033[33m$*\033[0m"
}
###
#  check the client environment
###
DATADIR="data"

if [ ! -d "/${DATADIR}" ]; then

  echo "Your datadir is "/${DATADIR}" but not exist!"
  exit 1
fi
```

执行成功后，将相关压缩包同步到TBDS集群外需要安装客户端的节点上。节点最好是环境比较干净的状态，如没有JDK，没有生产程序等。

【注意】此处2个配置项请在使用前确认正确并符合预期。

以TBDS531为例：

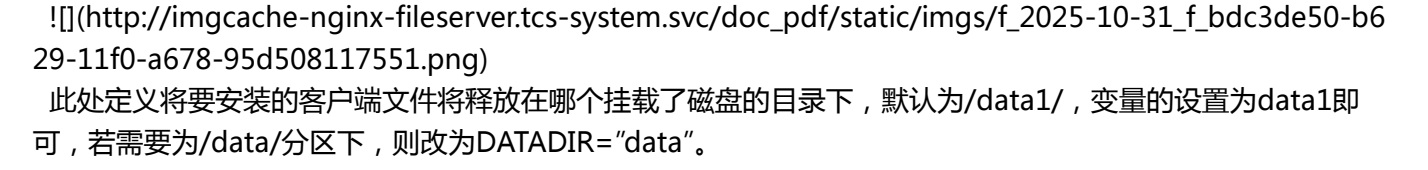
prepare-TBDS531-client-installpackage.sh

如图：

如需要将打包的客户端文件放到其他目录下，则修改此处即可。默认打包保存在/data/下，如果需要打包保存在/data1下则修改为DATADIR=”data1”。

prepare-TBDS-client-installpackage.sh检查中 输出Sqoop目录有误/usr/local/service/apps/sqoop1.4.7，需要改为/usr/local/service/apps/sqoop-1.4.7。

如图：



此处定义将要安装的客户端文件将释放在哪个挂载了磁盘的目录下，默认为/data1/，变量的设置为data1即可，若需要为/data/分区下，则改为DATADIR=”data”。

客户端安装工具：执行完成准备工作相关操作后，就需要安装客户端了，安装客户的脚本为install-TBDS-client.sh，执行位置为TBDS集群外需要安装客户端的节点的任意位置，保证准备步骤打包的客户端文件已经传到

该节点。
该脚本支持全部客户端安装，以及按需安装，如下为使用提示：

在选择所有客户端都安装后，会自动安装，直到完成所有操作。

TBDS53XX版本开启kerberos的特殊处理
TBDS版本从530以上版本可支持kerberos认证，该认证情况下，需要额外添加该认证的相关配置文件。
- 针对TBDS530X版本需要把集群的/etc/krb5.conf文件同步到集群外服务器，同时安装kerberos客户端，如有yum源情况下安装yum install krb5-workstation；关于认证文件keytab的获取，TBDS530X版本需要平台侧提供；
- 针对TBDS531X版本客户端会自动同步krb5文件，但仍需安装kerberos客户端；TBDS531X版本用户可以在TM集群管理的用户管理页面下载对应用户的keytab文件。

方式2

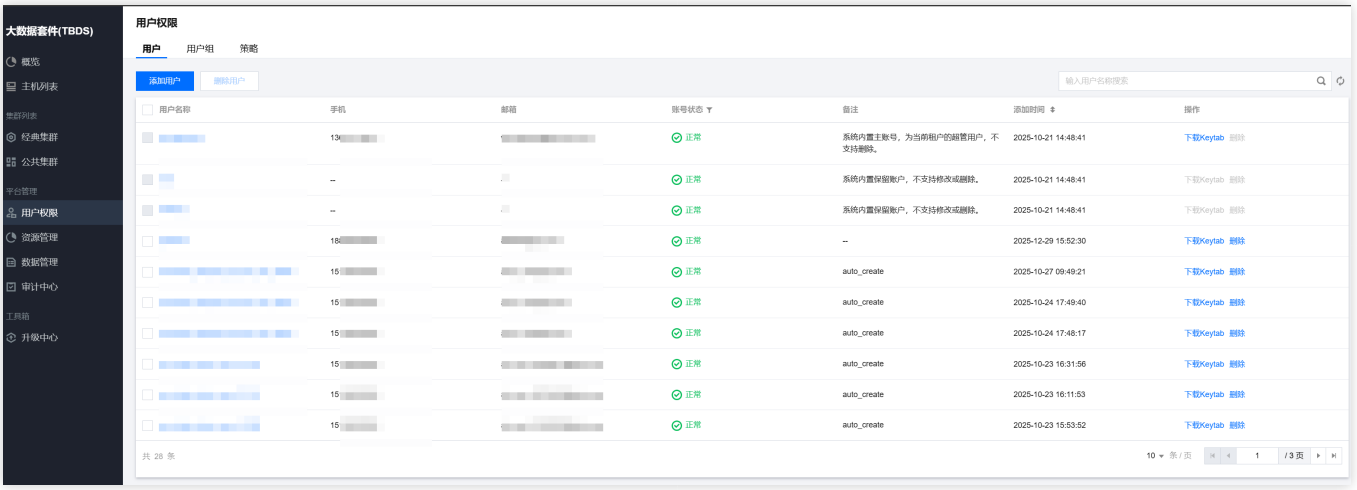
应用开发者已经安装好开源社区的组件客户端(要求版本与 TBDS 内部的组件 版本一致，或者兼容)，此时可以直接复用社区版的客户端进行访问。

> 注意：
>
> 建议开发者采用方式1；由于不同类型的 HADOOP 客户端有一定差异，若采用上述的方法2部署客户端，可能会存在不兼容问题。

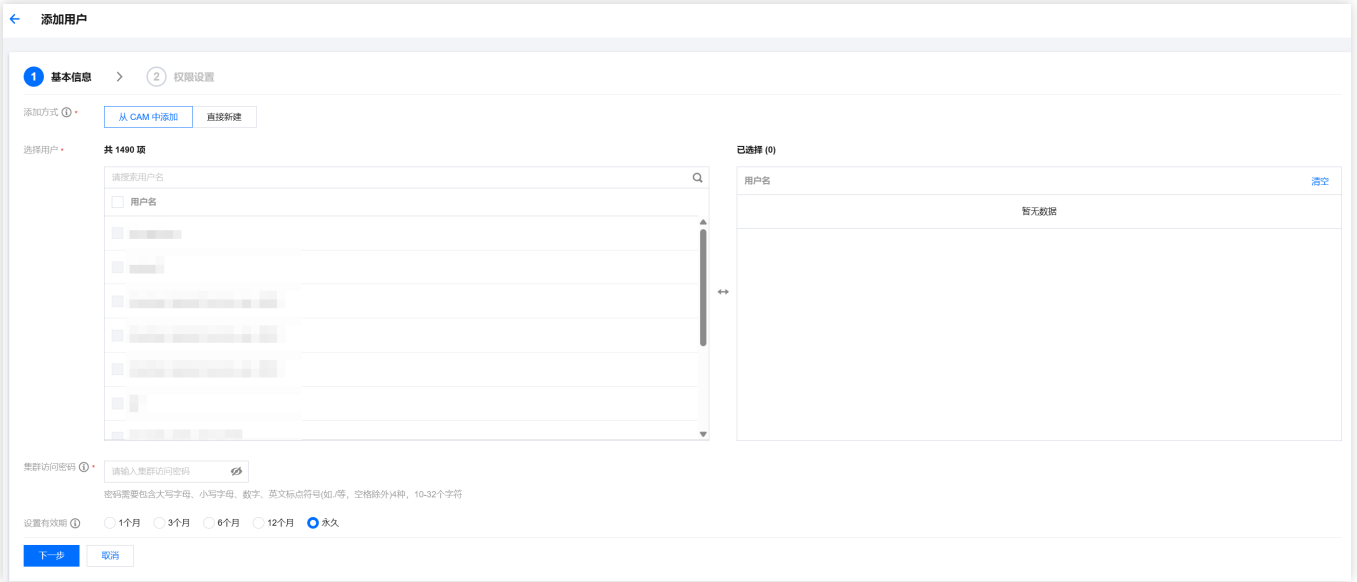
身份认证和授权
新建TBDS用户

TBDS平台通过整合CAM用户与集群用户，构建了一个统一的用户体系，从而优化了用户的操作流程。目前，平台提供了两种用户创建方式：第一种是"通过CAM添加"，即允许用户将已存在于CAM系统中的账户直接同步为集群用户；第二种是"直接新建"，这允许用户同时在CAM和集群中创建具有相同名称的新账户。这两种方法都旨在简化用户管理，提高效率。操作说明如下：

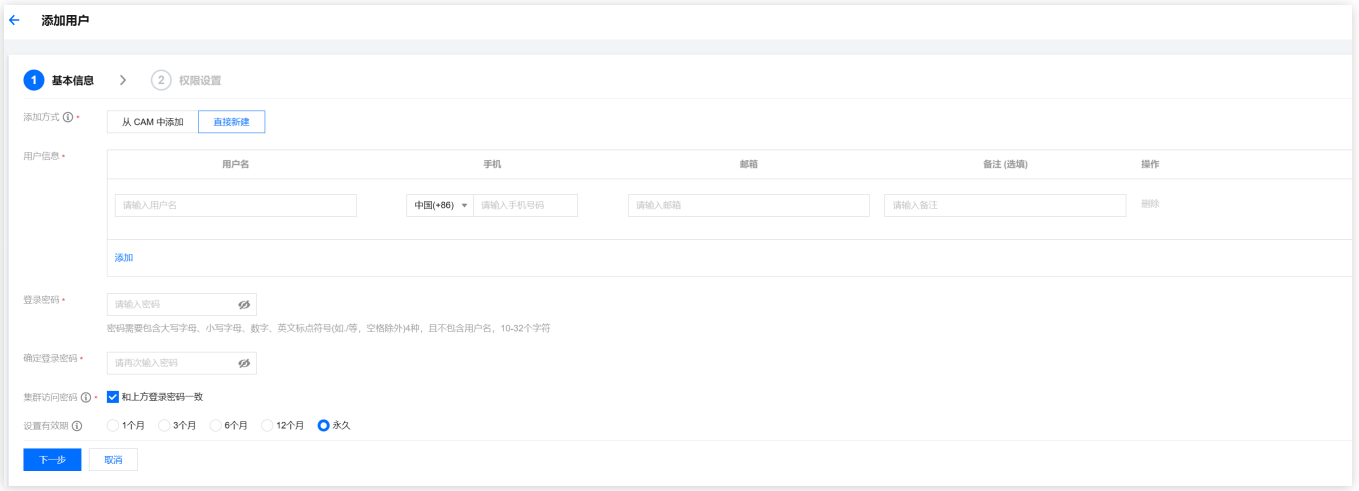
1. 以租户管理员或系统管理员身份登录TBDS Manager，点击左侧“用户权限”菜单进入列表页面。



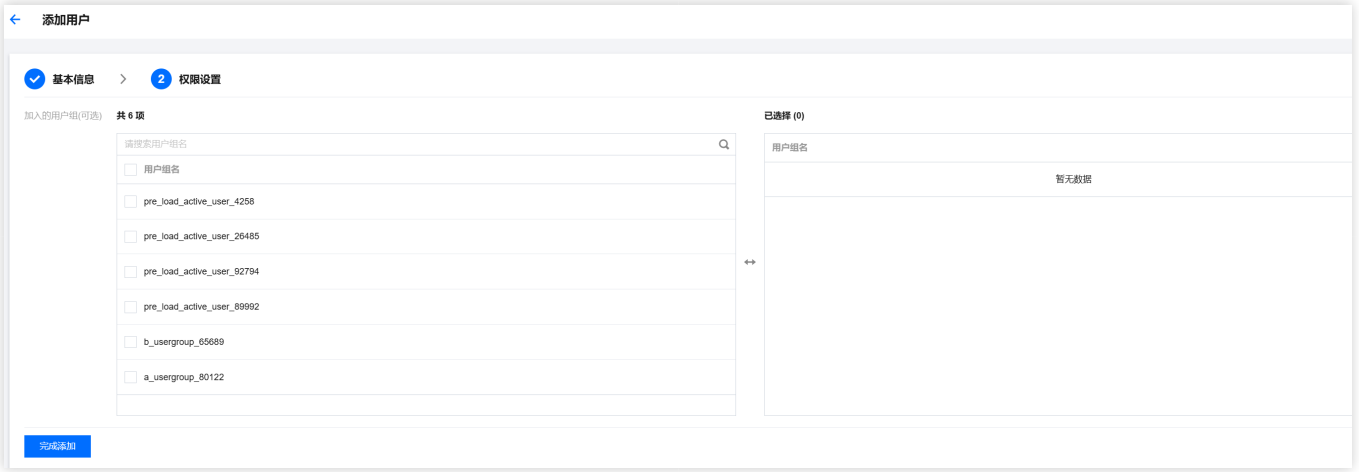
2. 点击“添加用户”，选择“从 CAM 中添加”，搜索需要添加的用户，并输入集群访问密码（说明此密码是访问大数据平台组件的密码，可能和登录密码不一样），点击下一步。



3. 如果上一步没有想要的用户，可以点击“直接新建”添加方式，输入登录密码，注意默认情况下TBDS Manager 登录密码和集群访问密码相同，如果希望不同，可取消“和上方登录密码一致”勾选单独输入，完成后点击下一步。



4. 经过步骤 2 或 3 后已经完成用户创建，点击下一步进入权限配置页面进行用户组设置。



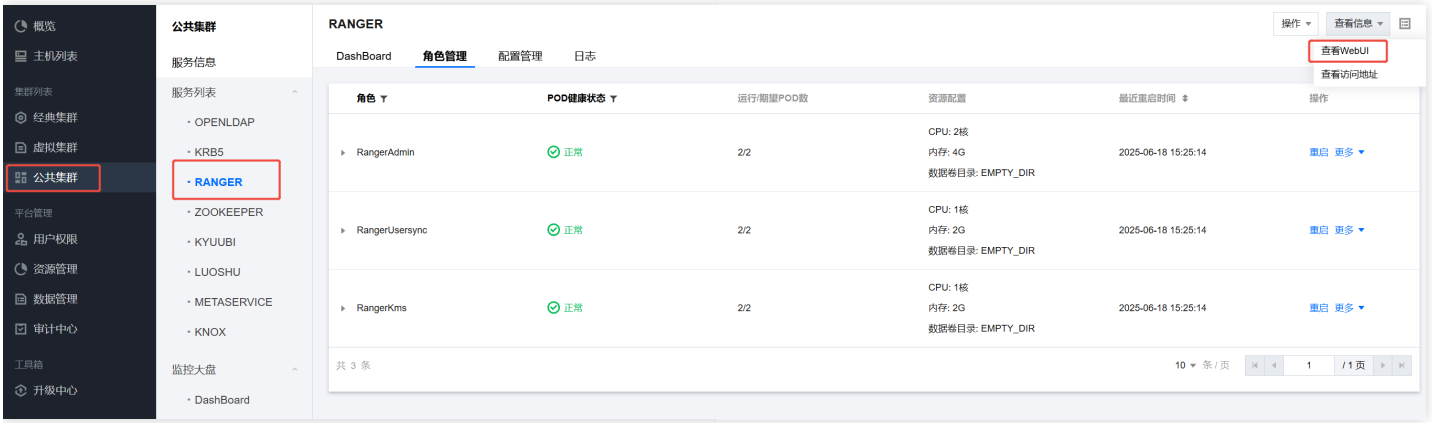
5. 添加用户完成。

为用户授权
Ranger授权（经典集群）

Ranger服务提供了集中式的权限管理框架，可以对各类组件进行细粒度的权限访问控制，并可以通过Web UI进行在线操作。

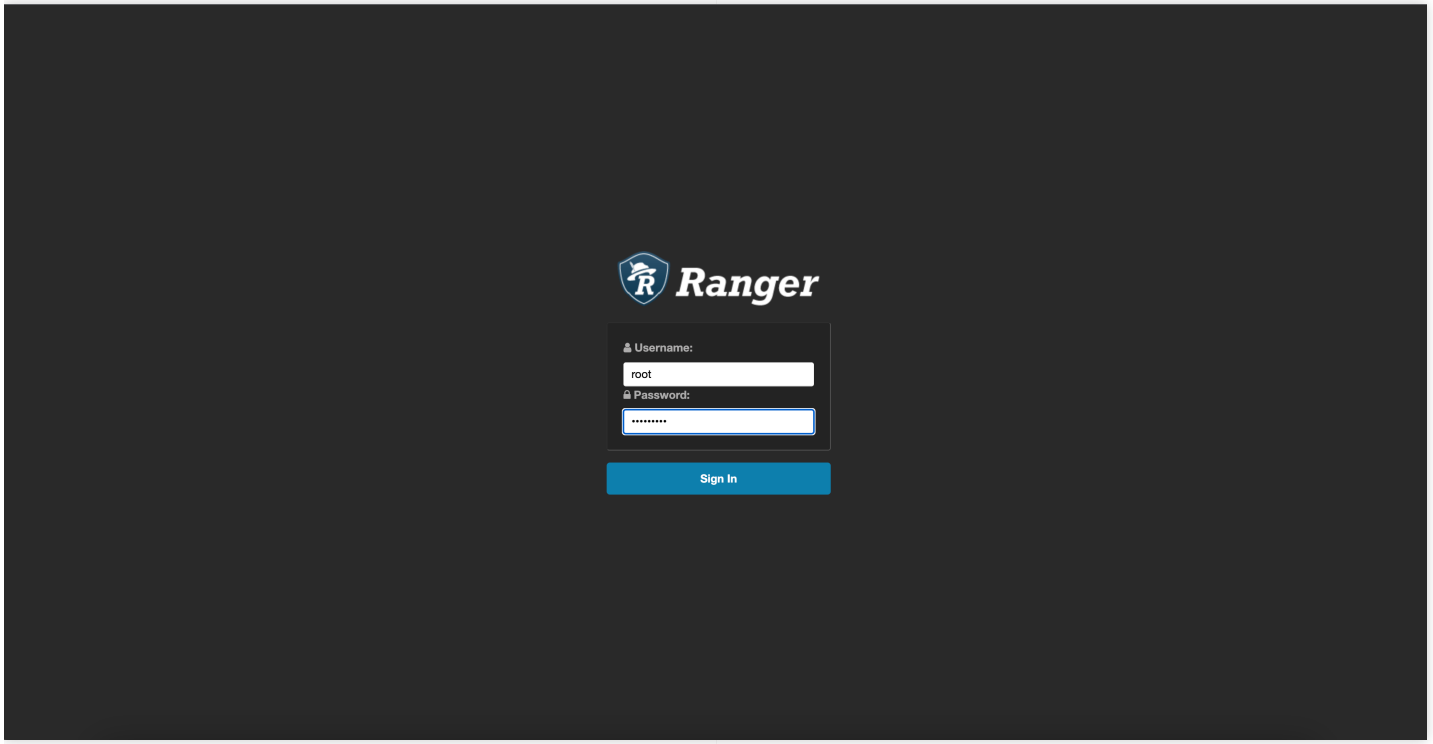
从集群管理页面进入Ranger UI

通过TBDS Manager的集群管理页面，进入公共集群找到对应的ranger服务，并访问WebUI的地址。

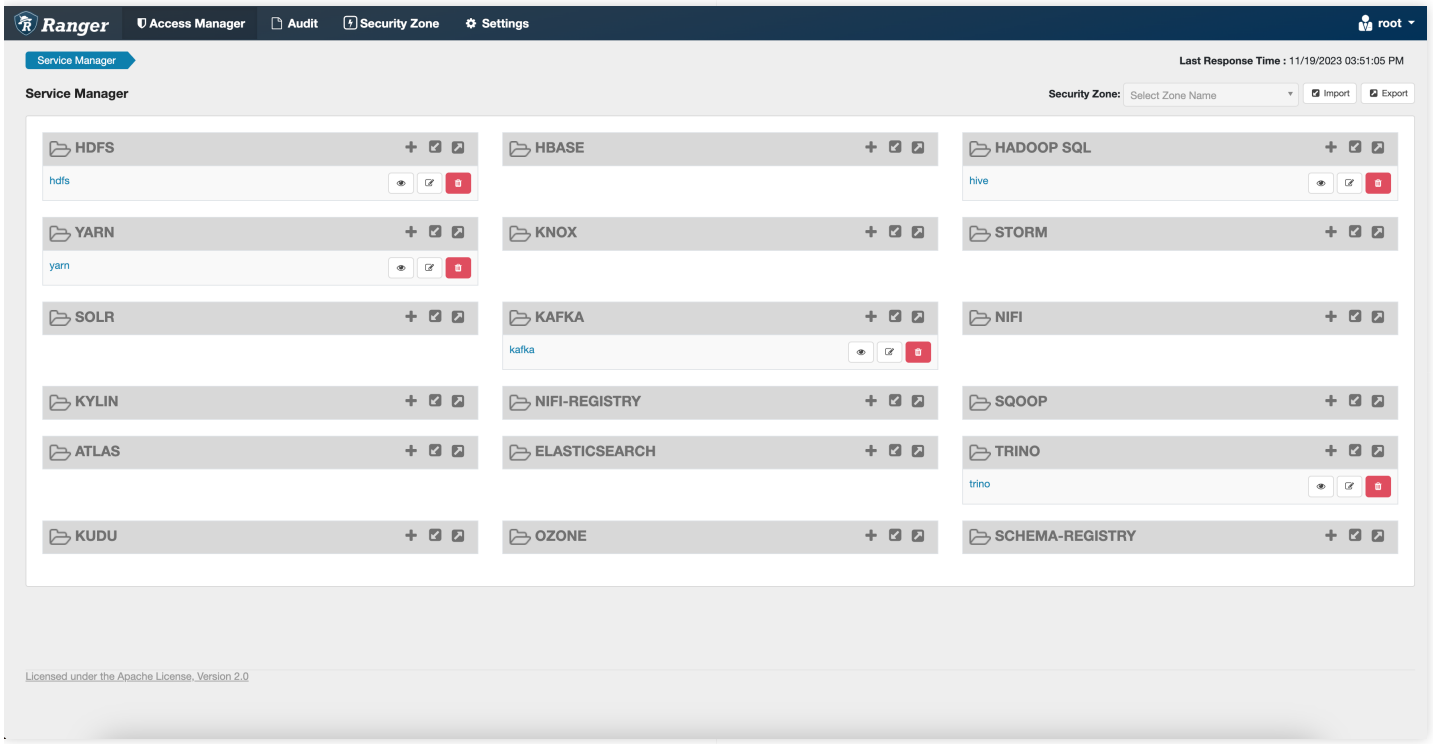


以管理员账号登录Ranger

使用管理员账号登录，管理员账号为root，密码为创建集群时填写的集群密码。

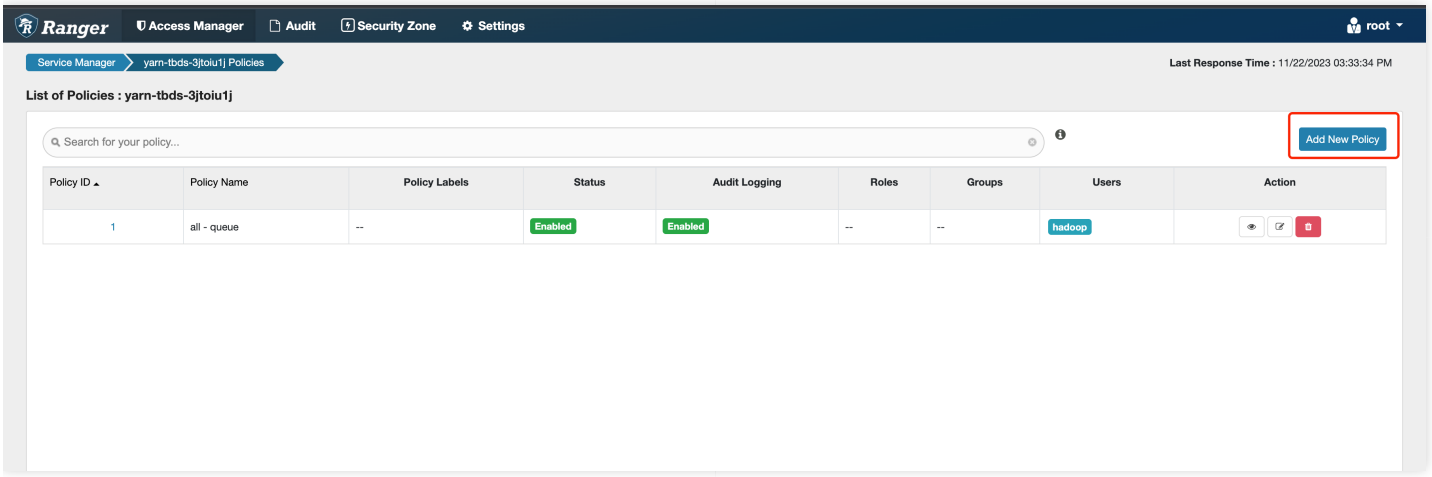


登录后在Ranger首页的“Service Manager”区域内，单击组件名称下的权限插件名称，即可进入组件安全访问策略列表页面。



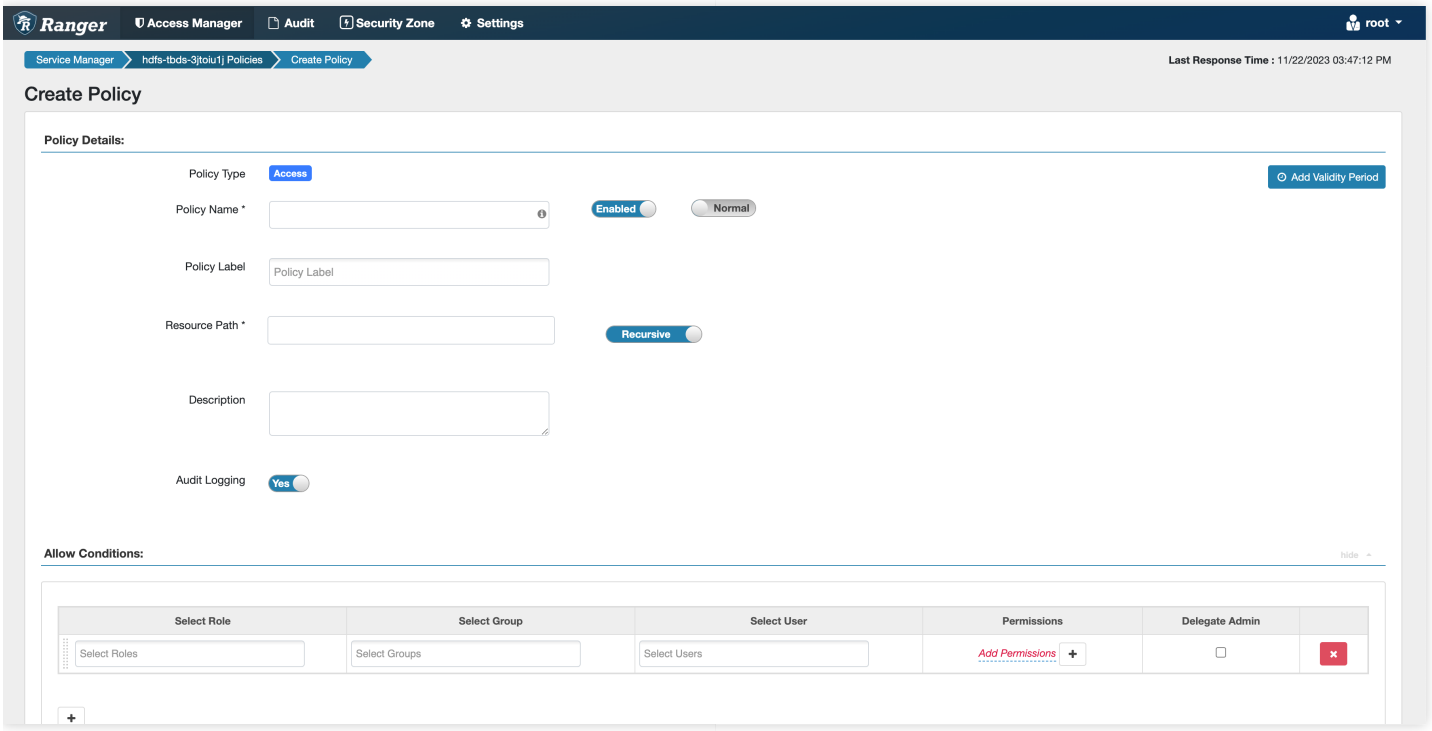
添加授权策略

进入到需要授权的组件服务后，点击“Add New Policy”，填写对应组件的权限策略后保存，即可完成授权。



创建HDFS策略

进入hdfs组件的service页面，添加策略。

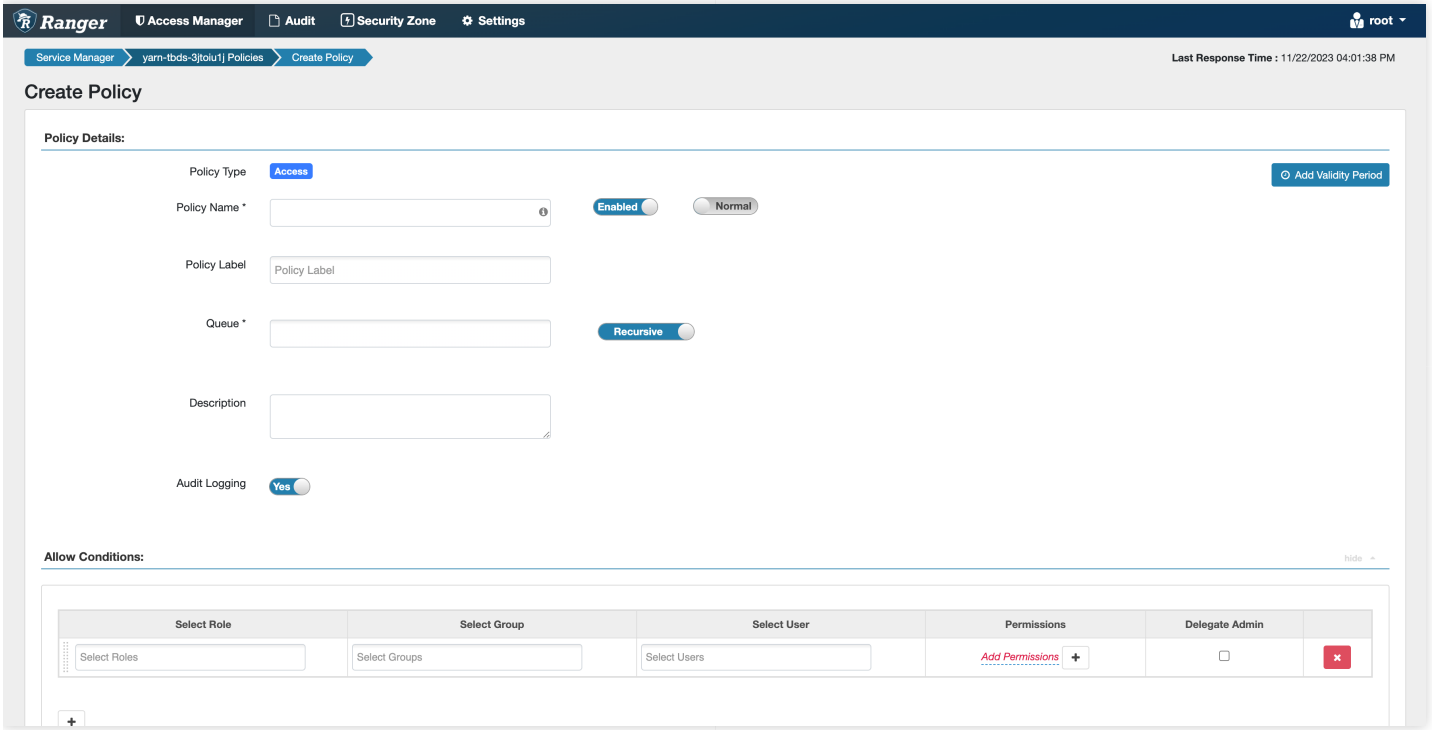


参数名称	描述
Policy Name	策略名称
Policy Label	策略标签

参数名称	描述
Resource Path	hdfs资源路径，可填写多个值，支持通配符"*" 并且可以设置是否递归，如果开启，则同时子目录也配置该策略 non-recursive: 关闭递归 recursive: 开启递归
Description	策略描述
Audit Logging	是否审计此策略
Allow Conditions	策略允许条件设置，并通过"+"号进行添加 Select Role: 授予的角色 Select Group: 授予的用户组 Select User: 授予的用户 Permission: 授权策略 Read(读权限)、Write(写权限)、Execute(执行权限)、Select/Deselect All(全选/取消全选) Delegate Admin: 是否让这些用户或用户组能够管理该条策略 Exclude from Allow Conditions：配置排除在允许条件之外的例外规则。
Deny All Other Access	是否拒绝其它所有访问。 True：拒绝其它所有访问。 False：设置为false，可配置Deny Conditions。
Deny Conditions	策略拒绝条件设置，配置方法与"Allow Conditions"相同，此条件的优先级将高于"Allow Conditions"设置的条件 Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

创建YARN策略

进入YARN组件service，添加策略。

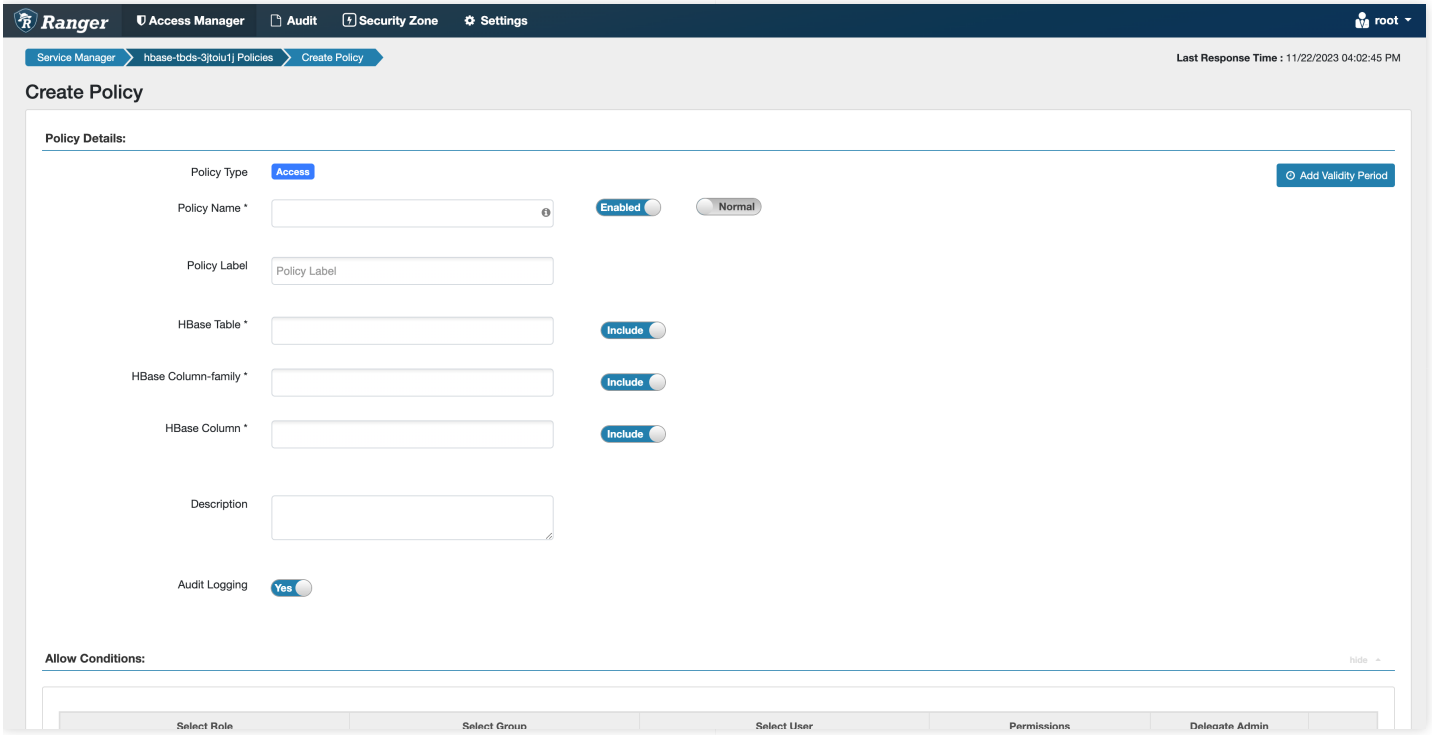


参数名称	描述
Policy Name	策略名称
Policy Label	策略标签
Queue	YARN资源队列，可填写多个值，支持通配符"*" 并且可以设置是否递归，如果开启，则同时子队列也配置该策略 non-recursive: 关闭递归 recursive: 开启递归
Description	策略描述
Audit Logging	是否审计此策略
Allow Conditions	策略允许条件设置，并通过"+"号进行添加 Select Role: 授予的角色 Select Group: 授予的用户组 Select User: 授予的用户 Permission: 授权策略 submit-app(提交队列任务权限)、admin-queue(管理队列任务权限)、Select/Deselect All(全选/取消全选) Delegate Admin: 是否让这些用户或用户组能够管理该条策略 Exclude from Allow Conditions：配置排除在允许条件之外的例外规则。
Deny All Other Access	是否拒绝其它所有访问。 True：拒绝其它所有访问。 False：设置为false，可配置Deny Conditions。

参数名称	描述
Deny Conditions	策略拒绝条件设置，配置方法与"Allow Conditions"相同，此条件的优先级将高于"Allow Conditions"设置的条件 Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

创建HBase策略

进入HBase组件service，添加策略。

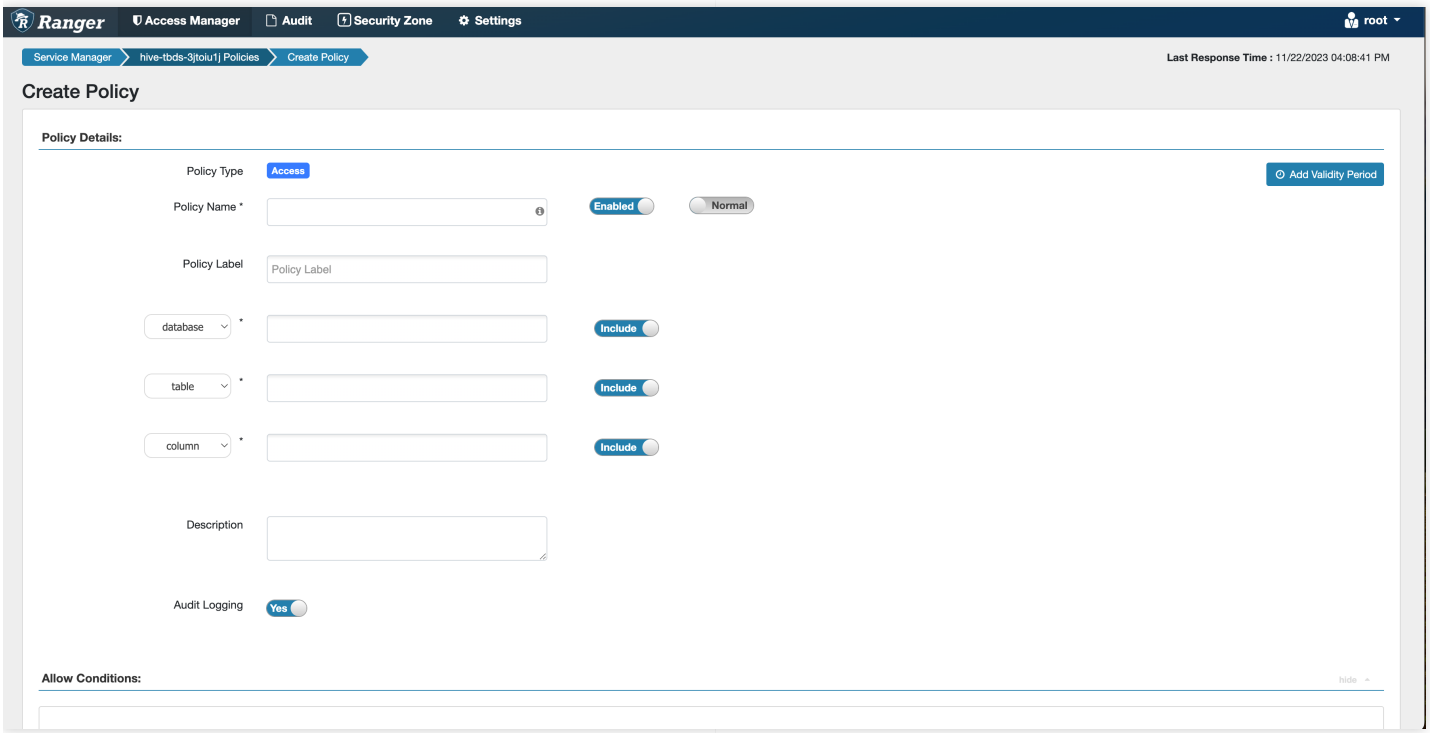


参数名称	描述
Policy Name	策略名称
Policy Label	策略标签
HBase Table	适用该策略的HBase表，支持通配符"*" “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象
HBase Column-family	适用该策略的HBase列族，支持通配符"*" “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象

参数名称	描述
HBase Column	适用该策略的HBase列，支持通配符"*" “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象
Description	策略描述
Audit Logging	是否审计此策略
Allow Conditions	策略允许条件设置，并通过"+"号进行添加 Select Role: 授予的角色 Select Group: 授予的用户组 Select User: 授予的用户 Permission: 授权策略 Read(读权限)、Write(写权限)、Create(创建权限)、Admin(管理权限)、Select/Deselect All(全选/取消全选) Delegate Admin: 是否让这些用户或用户组能够管理该条策略 Exclude from Allow Conditions：配置排除在允许条件之外的例外规则。
Deny All Other Access	是否拒绝其它所有访问。 True：拒绝其它所有访问。 False：设置为false，可配置Deny Conditions。
Deny Conditions	策略拒绝条件设置，配置方法与"Allow Conditions"相同，此条件的优先级将高于"Allow Conditions"设置的条件 Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

创建Hive策略

进入hive组件service，添加策略。

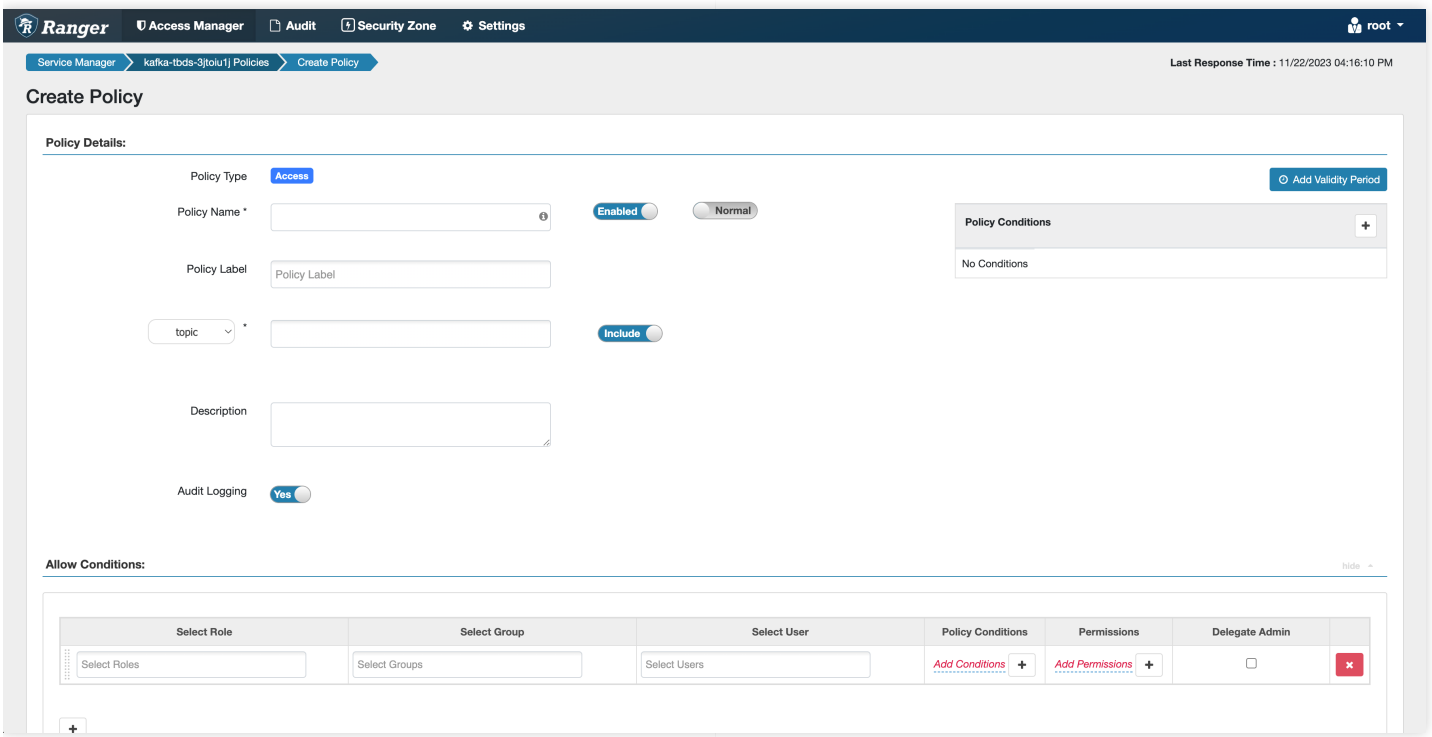


参数名称	描述
Policy Name	策略名称
Policy Label	策略标签
database	适用该策略的Hive数据库名称 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
table	适用该策略的Hive表名称 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
column	适用该策略的Hive列名 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
Description	策略描述
Audit Logging	是否审计此策略
Allow Conditions	策略允许条件设置，并通过"+"号进行添加 Select Role: 授予的角色 Select Group: 授予的用户组 Select User: 授予的用户 Permission: 授权策略 Select(查询权限)、Update(更新权限)、Create(创建权限)、

参数名称	描述
	Drop(drop操作权限)、Alter(alter操作权限)、Index(索引操作权限)、Lock(lock操作权限)、All(所有执行权限)、Read(只读权限)、Write(写权限)、ReplAdmin(ReplAdmin权限)、Service Admin(Service Admin权限)、Temporary UDF Admin(临时UDF管理权限)、Refresh(刷新权限)、Select/Deselect All(全选/取消全选) Delegate Admin: 是否让这些用户或用户组能够管理该条策略 Exclude from Allow Conditions：配置排除在允许条件之外的例外规则。
Deny All Other Access	是否拒绝其它所有访问。 True：拒绝其它所有访问。 False：设置为false，可配置Deny Conditions。
Deny Conditions	策略拒绝条件设置，配置方法与"Allow Conditions"相同，此条件的优先级将高于"Allow Conditions"设置的条件 Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

创建Kafka策略

进入kafka组件service，添加策略。

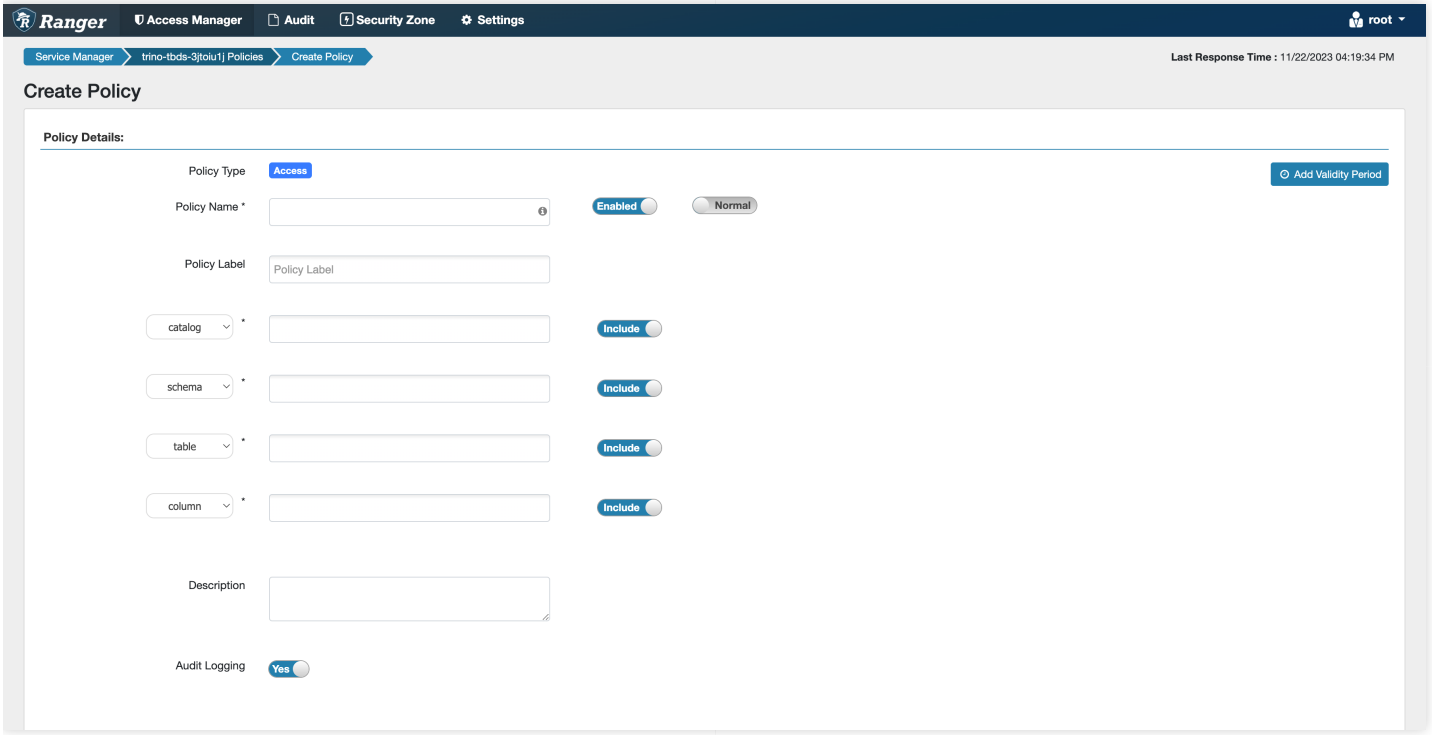


参数名称	描述
Policy Name	策略名称

参数名称	描述
Policy Label	策略标签
topic	topic名，支持通配符 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
Description	策略描述
Audit Logging	是否审计此策略
Allow Conditions	策略允许条件设置，并通过"+"号进行添加 Select Role: 授予的角色 Select Group: 授予的用户组 Select User: 授予的用户 Permission: 授权策略 Publish(生产权限)、Consume(消费权限)、Describe(查询权限)、Create(创建主题权限)、Delete(删除主题权限)、Describe Configs(查询配置权限)、Alter(修改topic的权限)、Alter Configs(修改配置权限)、Select/Deselect All(全选/取消全选) Delegate Admin: 是否让这些用户或用户组能够管理该条策略 Exclude from Allow Conditions：配置排除在允许条件之外的例外规则。
Deny All Other Access	是否拒绝其它所有访问。 True：拒绝其它所有访问。 False：设置为false，可配置Deny Conditions。
Deny Conditions	策略拒绝条件设置，配置方法与"Allow Conditions"相同，此条件的优先级将高于"Allow Conditions"设置的条件 Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

创建Trino策略

进入trino组件service，添加策略。

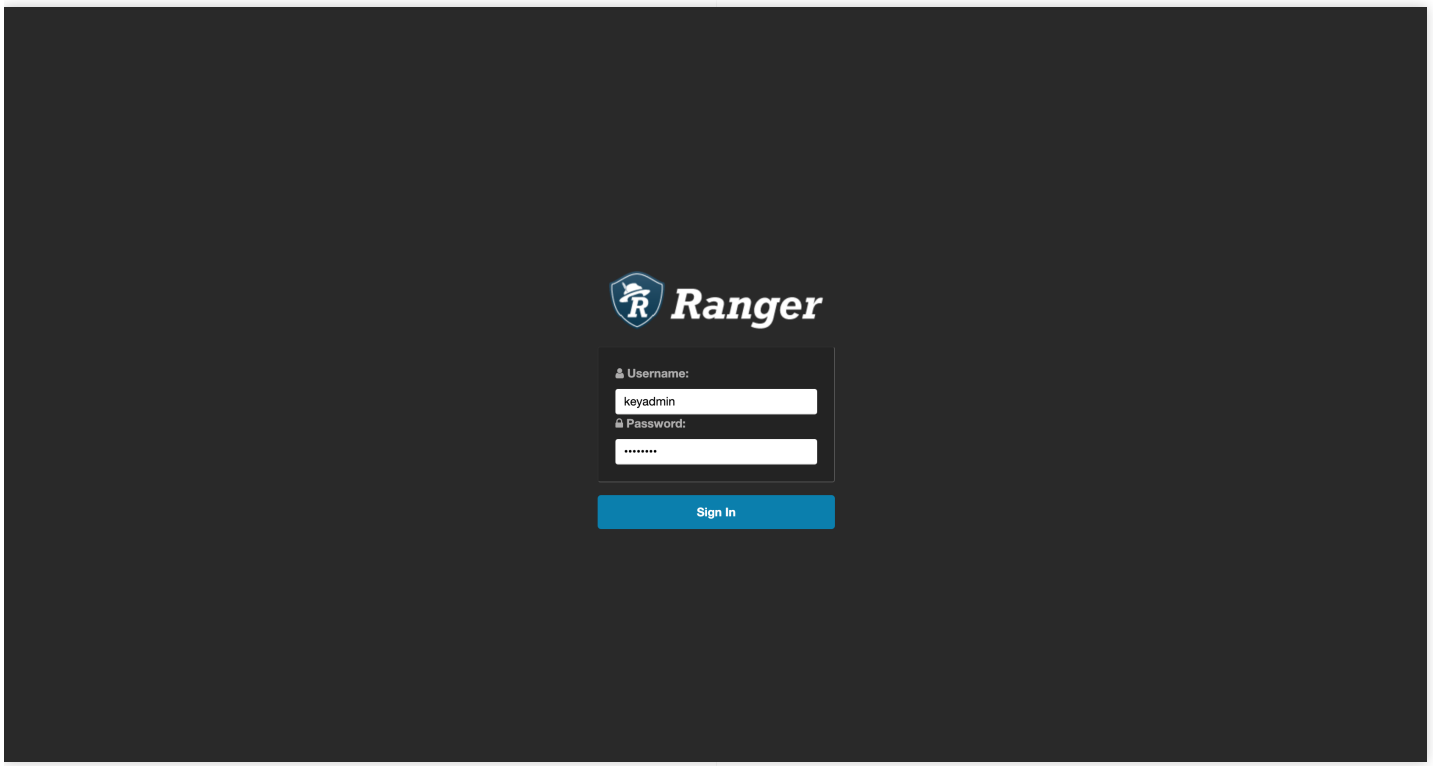


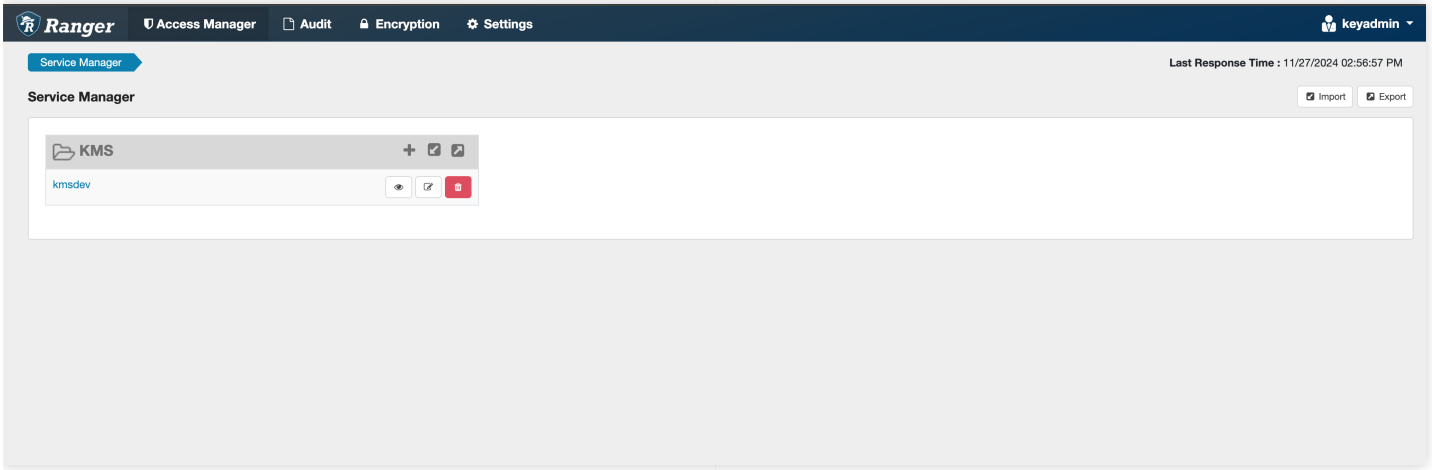
参数名称	描述
Policy Name	策略名称
Policy Label	策略标签
catalog	catalog名 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
schema	schema名 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
table	table名 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
column	column名 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
Description	策略描述
Audit Logging	是否审计此策略

参数名称	描述
Allow Conditions	策略允许条件设置，并通过"+"号进行添加 Select Role: 授予的角色 Select Group: 授予的用户组 Select User: 授予的用户 Permission: 授权策略 Select(查询权限)、Insert(插入权限)、Create(创建权限)、Drop(drop操作权限)、Delete(删除权限)、Use(use操作权限)、Alter(修改权限)、 Grant(grant操作权限)、Revoke(revoke操作权限)、Show(show操作权限)、Impersonate(impersonate操作权限)、All(全部权限)、execute(执行权限)、Select/Deselect All(全选/取消全选) Delegate Admin: 是否让这些用户或用户组能够管理该条策略 Exclude from Allow Conditions：配置排除在允许条件之外的例外规则。
Deny All Other Access	是否拒绝其它所有访问。 True：拒绝其它所有访问。 False：设置为false，可配置Deny Conditions。
Deny Conditions	策略拒绝条件设置，配置方法与"Allow Conditions"相同，此条件的优先级将高于"Allow Conditions"设置的条件 Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

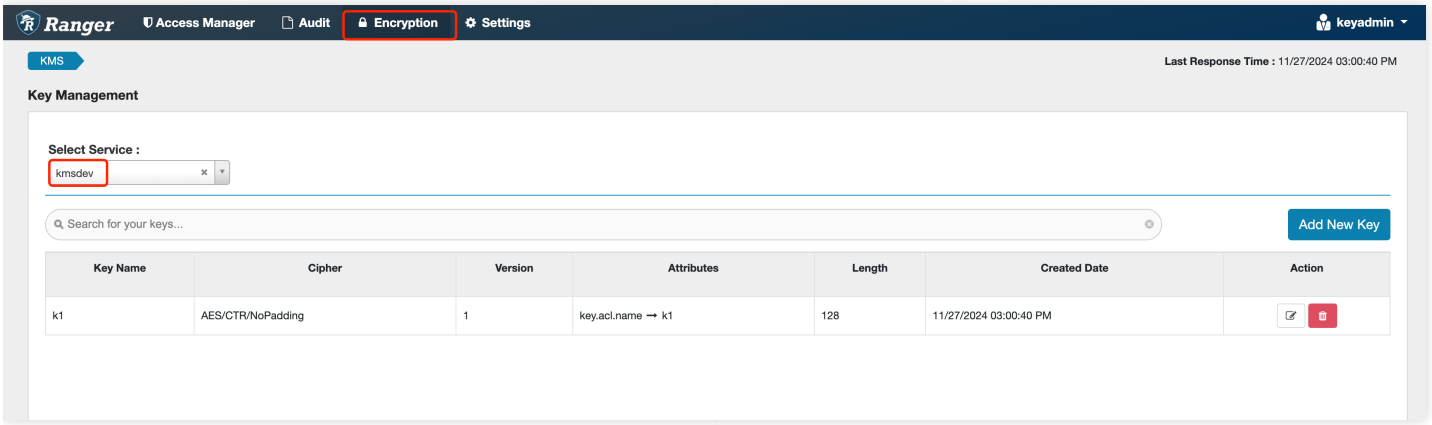
创建KMS密钥

在Ranger UI界面，使用keyadmin账号登录，密码为keyadmin。

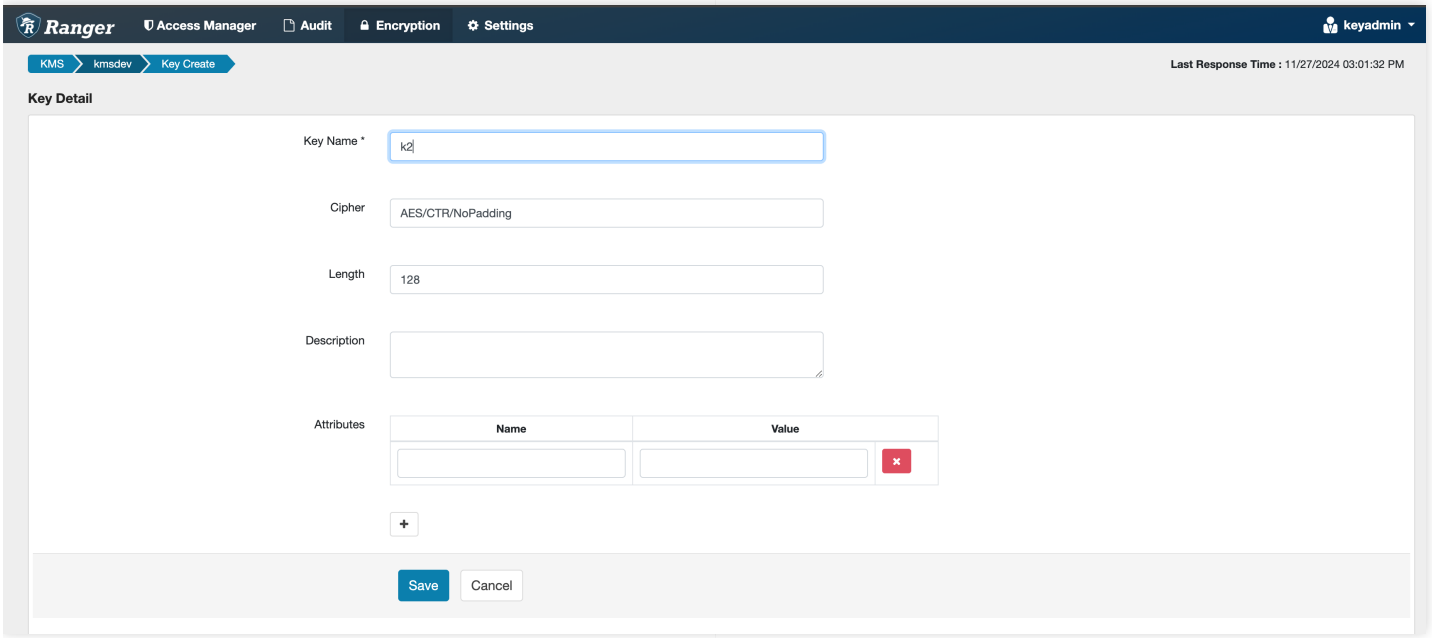


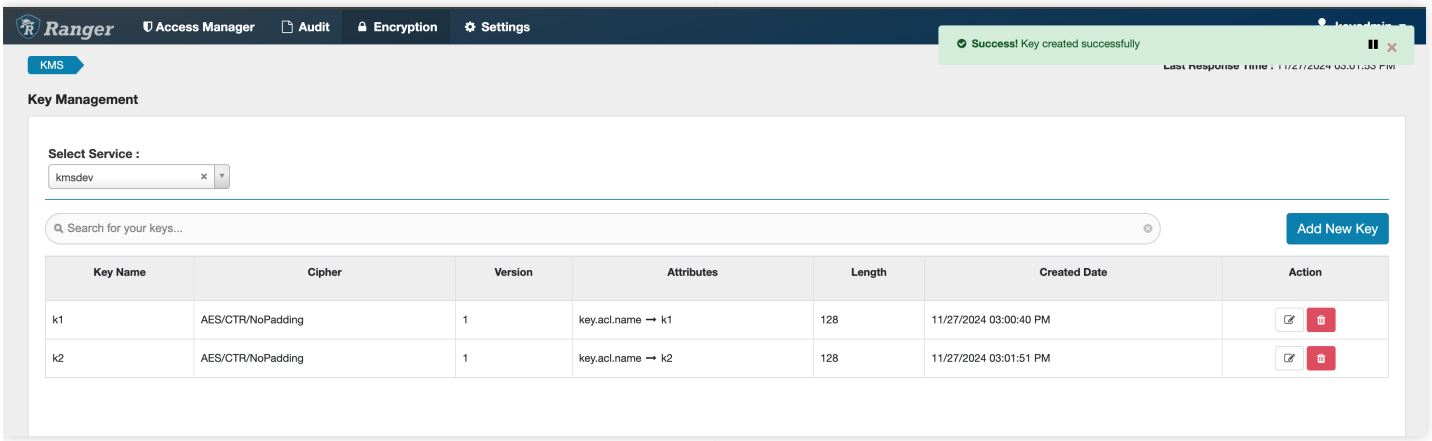


点击Encryption按钮，选择kmsdev服务，可以查看当前密钥列表。



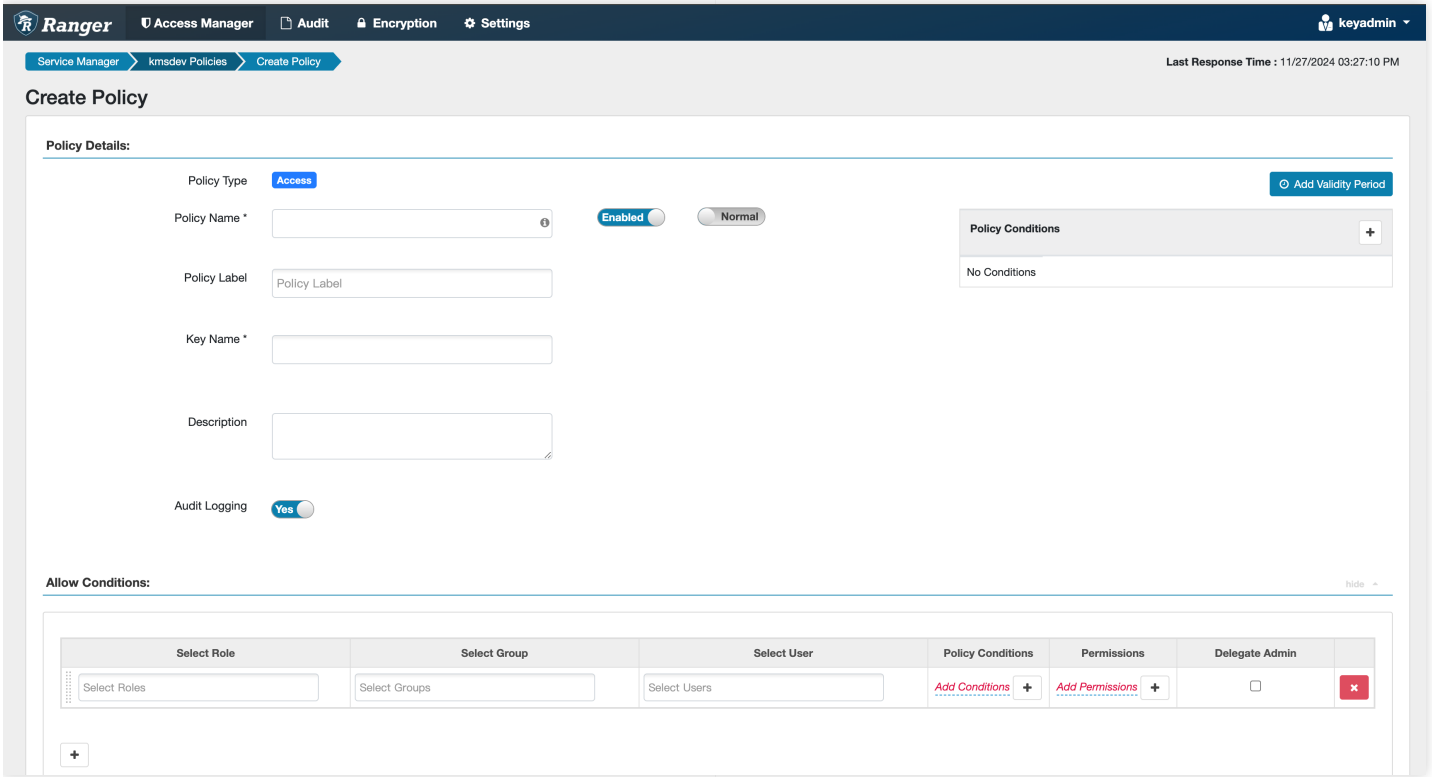
点击Add New Key可以新增密钥





创建KMS策略

进入KMS组件service，添加策略。



参数名称	描述
Policy Name	策略名称
Policy Label	策略标签

参数名称	描述
Key Name	密钥名称，可填写多个值，支持通配符"*"
Description	策略描述
Audit Logging	是否审计此策略
Allow Conditions	允许策略，可以对选择的用户、用户组或者角色配置相应的权限。包括： Create：创建Key Delete：删除key Rollover：更新key Set Key Material：设置密钥密文 Get：读取某个key Get keys：读取所有key（策略中配置的key） Get Metadata：获取key的元信息 Generate EEK：生成EEK Decrypt EEK：解密EEK Exclude from Allow Conditions：配置允许条件之外的例外规则。
Deny All Other Access	是否拒绝其它所有访问。 True：拒绝其它所有访问。 False：设置为false，可配置Deny Conditions。
Deny Conditions	策略拒绝条件设置，配置方法与"Allow Conditions"相同，此条件的优先级将高于"Allow Conditions"设置的条件 Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

获取用户认证

Kerberos认证方式

启用了Kerberos认证的安全模式集群，为应用开发提供了必要的安全认证机制。Kerberos系统采用经典的"客户端/服务器"架构，并结合了AES等高级加密技术，实现了双方的相互认证能力——即客户端和服务器端都能对对方的身份进行验证。这种设计不仅能有效防止数据在传输过程中被窃听，还能抵御重放攻击，确保数据的完整性。Kerberos系统通过应用对称密钥体制，对密钥进行精细管理，从而构建了一个既安全又高效的认证环境。

用户在安装TBDS集群时如果开启了Kerberos认证，则在开发使用时需要提前准备凭证信息，用户凭证信息获取方式如下：

1. krb5.conf文件获取方式：集群机器部署自带，文件在/etc/krb5.conf，客户端部署时同步。
2. keytab文件获取方式：
   步骤1：进入用户权限-点击用户名称。
   步骤2：在用户详情页面找到凭证信息，复制KerberosPrincipal和下载 keytab。

Simple认证方式

如果用户在安装TBDS集群时未开启Kerberos认证，则默认使用Simple认证方式。在这种情况下，用户只需要使用创建的用户名即可访问集群，无需密码或keytab。

组件开发指南
TBDS示例工程
工程目录

组件	示例工程	示例工程描述	Git仓库访问地址	备注 （最常用的用户操作场景）
HDFS	hdfs-examples	使用HDFS Java API实现读取、写入、删除、追加操作	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/hdfs-examples	包含以下示例： 1、包含 HDFS Java API文件读取、写入、删除、追加等接口调用
Spark	spark-examples	调用Spark API编写常用数据处理任务	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/spark-examples	包含以下示例： 1、Spark-Core，常用算子操作 2、SparkSQL，包含与Hive 集成的 SQL 操作，连接方式包括自定义代码, 命令行, Thrift协议三种 3、Spark-Streaming&Structured Streaming（低） 4、Spark ML（低）
Hive	hive-examples	使用JDBC连接Hive，进行SQL操作	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/hive-examples	包含以下示例： 1、通过JDBC连接执行SQL操作，包括连接、SQL执行（包括DDL、DML 常见语法）、日志查看、作业监控等，优先 Java 和 Scala，Python 可选 2、Hive Beeline 命令行连接，包括连接、SQL执行（包括DDL、DML 常见语法）、日志查看、作业监控等
Trino	trino-examples	使用JDBC连接Trino，进行SQL操作	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-	包含以下示例： 1、JDBC 代码连接，包括连接、SQL执行（包括DDL、DML 常见语法）、

组件	示例工程	示例工程描述	Git仓库访问地址	备注 （最常用的用户操作场景）
			examples/git/files/master/src/trino-examples	日志查看、作业监控等，优先 Java 和 Scala，Python 可选 2、命令行连接方式，包括连接、SQL执行（包括 DDL、DML 常见语法）、日志查看、作业监控等
Impala	impala-examples	使用JDBC连接Impala，进行SQL操作	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/impala-examples	包含以下示例： 1、JDBC 代码连接，包括连接、SQL执行（包括 DDL、DML 常见语法）、日志查看、作业监控等 2、命令行连接，包括连接、SQL执行（包括DDL、DML 常见语法）、日志查看、作业监控等
Flink	flink-examples	调用Flink API编写常用实时数据处理任务	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/flink-examples	包含以下示例： 1、Flink-Core，常用算子操作 2、FlinkSQL，包含与 Hive 集成的 SQL 操作，连接方式包括自定义代码, 命令行, Thrift协议三种
Yarn	yarn-examples	使用Yarn API自定义编写运行在Yarn上的应用程序	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/yarn-examples	包含以下示例： 1、使用Yarn 提供的Java API进行大数据分布式程序编写
HBase	hbase-examples	HBase创建集群连接、创建表、读写、删除表的基本功能操作示例	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/hbase-examples	包含以下示例： 1、HBase Java API操作，包括创建 namespace、表、写入数据、scan 数据，读取数据等常用操作
Phoenix	phoenix-examples	执行Phoenix SQL的基本功能	https://g-necm8077.coding.net/public/tencentcloud-	包含以下示例： 1、JDBC 代码连接，包括连接、SQL执行

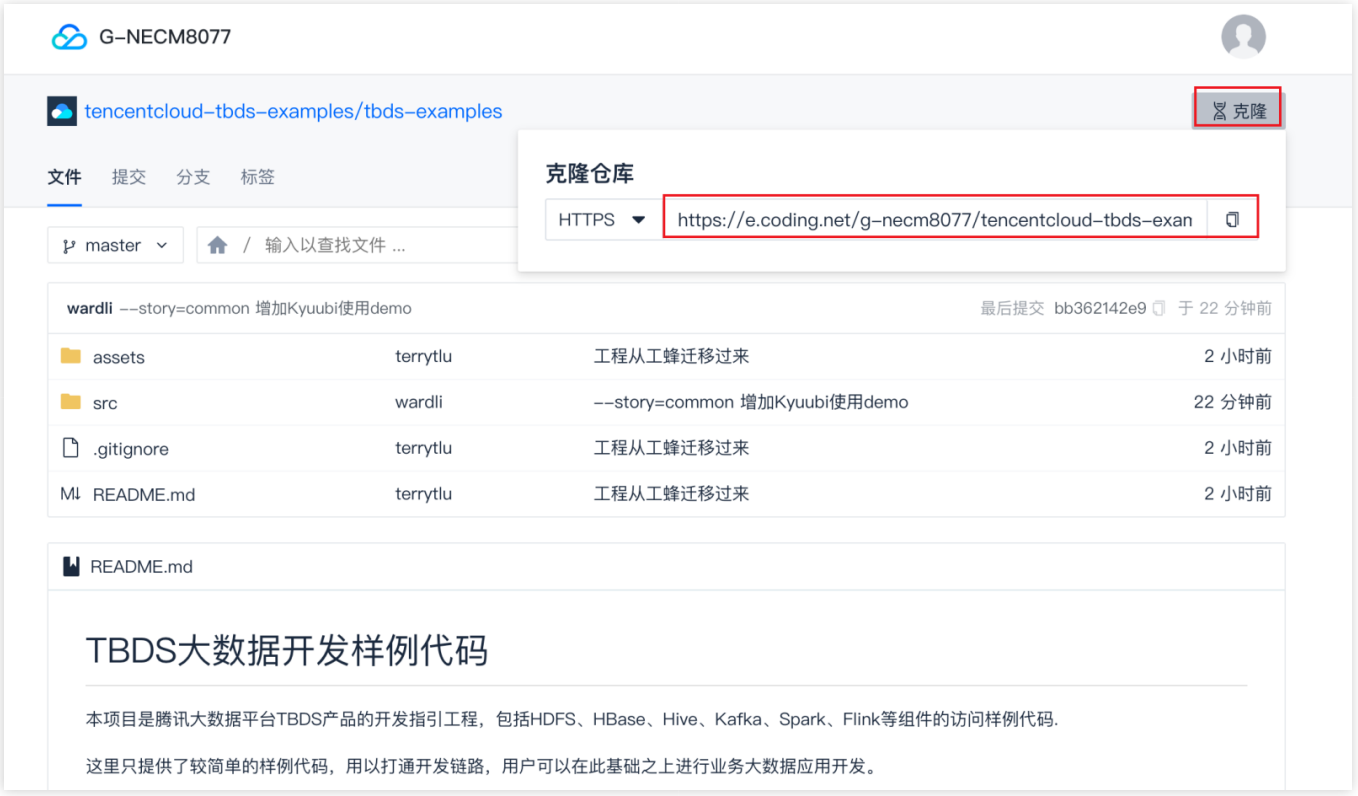
组件	示例工程	示例工程描述	Git仓库访问地址	备注 （最常用的用户操作场景）
			tbds-examples/tbds-examples/git/files/master/src/phoenix-examples	2、命令行连接方式，包括连接、SQL执行
Kafka	kafka-examples	使用Kafka API进行Kafka 消息的读取与写入	https://g-necm8077.coding.net/p/tencentcloud-tbds-examples/d/tbds-examples/git/tree/master/src/kafka-examples	包含以下示例： 1、Kafka 发布和订阅功能示例，开发语言 Java 和 Scala，包括消息写入和消息消费
Kyuubi	kyuubi-examples	使用Kyuubi API通过REST或者JDBC连接提交相关应用程序	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/kyuubi-examples	包含以下示例： 1、JDBC 代码连接，包含 Spark 和 Trino集成的SQL操作，包括连接、SQL执行（包括DDL、DML 常见语法）、日志查看、作业监控等 2、命令行连接，包含 Spark 和 Trino集成的SQL操作，包括连接、SQL执行（包括DDL、DML 常见语法）、日志查看、作业监控等
Iceberg	iceberg-examples	使用Iceberg API进行Spark以及Flink的集成开发	https://g-necm8077.coding.net/p/tencentcloud-tbds-examples/d/tbds-examples/git/tree/master/src/iceberg-examples	包含以下示例： 1、Spark 集成开发SQL和Java API示例，包含 DDL 和 DML 2、Flink 集成开发SQL和Java API示例，包含 DDL 和 DML
Elasticsearch	elasticsearch-example	使用REST API操作es的数据导入与读取	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/elasticsearch-examples	包含以下示例： 1、通过REST API访问ES 创建索引, 查询索引, 删除索引
StarRocks	StarRocks开发	StarRocks常见的使用方式	StarRocks开发	包含以下示例： 1、对StarRocks进行相关的

组件	示例工程	示例工程描述	Git仓库访问地址	备注 （最常用的用户操作场景）
		与配置		配置 2、StarRocks相关最佳实践
MapReduce	mapreduce-examples	使用Hadoop API编写常见MapReduce程序开发	https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master/src/mapreduce-examples	包含以下示例： 1、MapReduce 常见Java API编写指导 2、MapReduce 常见程序参数解析
Hudi	Hudi开发	Hudi常见的使用方式与最佳实践	Hudi开发	包含以下示例： 1、对Hudi进行关键功能与常用参数解析 2、Hudi最佳实践与开发规范

使用流程

示例工程使用流程包括如下步骤:

1. 下载样例工程的工程源码：使用Git clone命令可以下载完整样例工程代码。
2. 根据TBDS产品版本选择对应的分支便可以进行示例代码的查看。
   TBDS示例仓库: https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master



HDFS开发
概述

HDFS（Hadoop Distributed File System）是Apache Hadoop生态系统的一部分，是一个分布式文件系统，旨在处理大规模数据集的存储和处理。
以下是HDFS的一些关键特点和概念：

- 分布式存储：HDFS将大文件切分为较小的数据块，并将这些数据块分布存储在Hadoop集群的多个节点上。这种分布式存储方式提供了高容错性和可伸缩性。
- 冗余存储：HDFS通过数据复制机制提供数据冗余。每个数据块默认会被复制到集群中的多个节点上，以确保数据的可靠性和容错性。
- 主从架构：HDFS采用主从架构，其中有一个称为NameNode的主节点负责管理文件系统的元数据，包括文件和目录的命名空间、权限和数据块的位置等信息。数据块实际存储在称为DataNode的从节点上。
- 高吞吐量：HDFS的设计目标之一是提供高吞吐量的数据访问。它适用于一次写入多次读取的大型数据集场景，如批处理任务和数据分析。
- 数据本地性：HDFS倾向于将计算任务分配给存储数据的节点，以减少数据的网络传输，提高数据访问性能。
- 支持多种数据处理模型：HDFS作为Hadoop生态系统的核心组件，能够与其他Hadoop生态系统工具（如MapReduce、Apache Spark、Hive等）无缝集成，支持多种数据处理和分析模型。

示例工程开发

环境准备（考虑共用）

1. 开发环境准备

准备项	说明
安装JDK	JDK8/JDK11，推荐使用KonaJDK，下载地址
安装和配置IDE	按需选择，比如IntelliJ IDEA或Eclipse
安装Maven	开发环境基础配置，负责构建Java应用程序
Maven配置准备	如果需要本地调试，需要配置Maven pom.xml，推荐Maven 3.6.3，下载地址

2. 运行环境准备
   以下使用Linux环境作为开发机进行应用调试说明。

   i. 准备开发机（可选）：建议使用Linux操作系统；
   ii. 部署客户端：参考部署TBDS客户端 在开发机上执行客户端部署，HDFS开发涉及的配置如下：

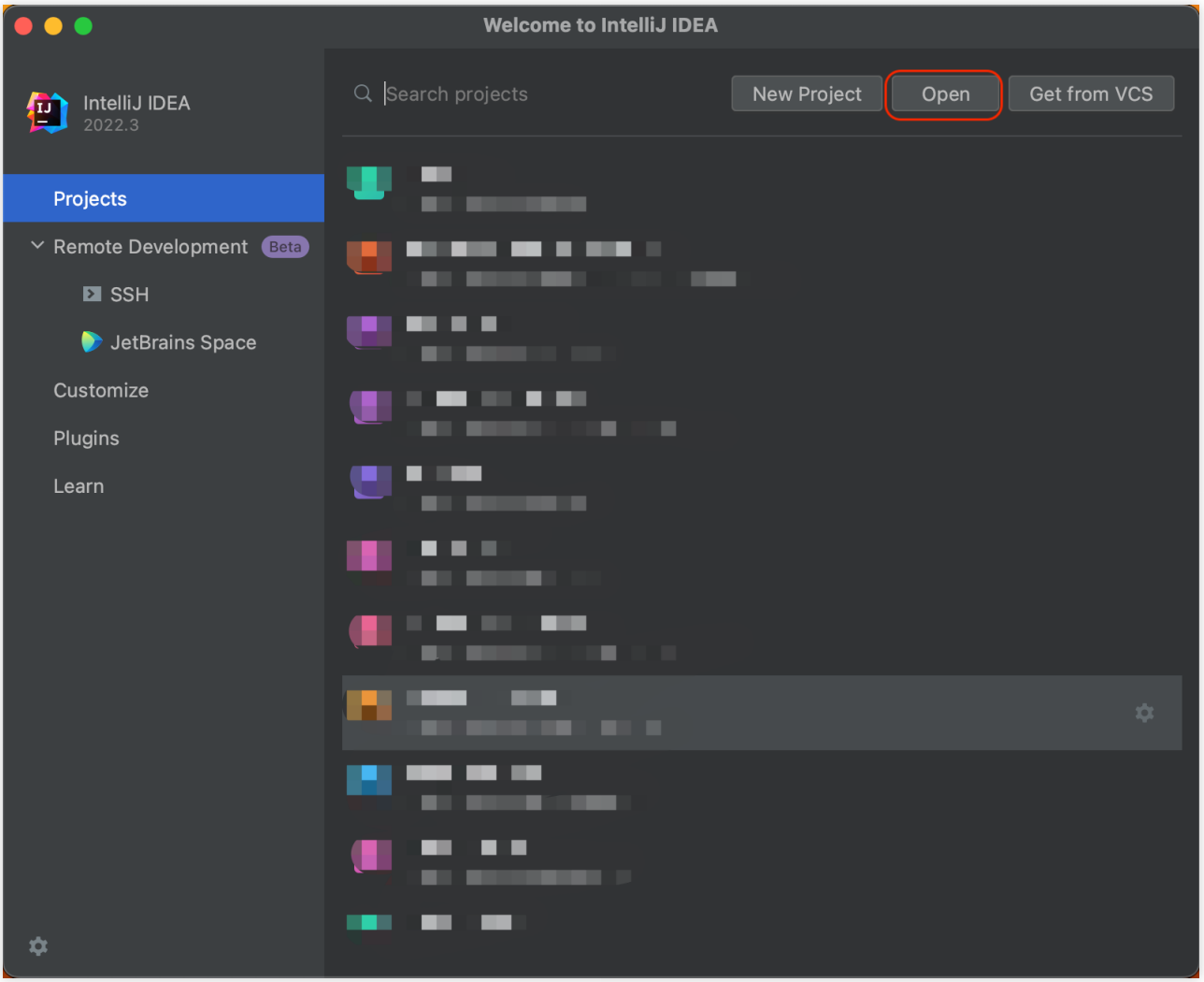
文件名称	作用
core-site.xml	配置Hadoop集群的核心配置参数。
hdfs-site.xml	配置HDFS组件的详细参数。
emr.keytab	对于Kerberos安全认证提供用户信息。
krb5.conf	Kerberos Server配置信息。

3. 导入示例工程代码
   以下以IntelliJ IDEA举例，将HDFS示例工程代码导入进行说明。

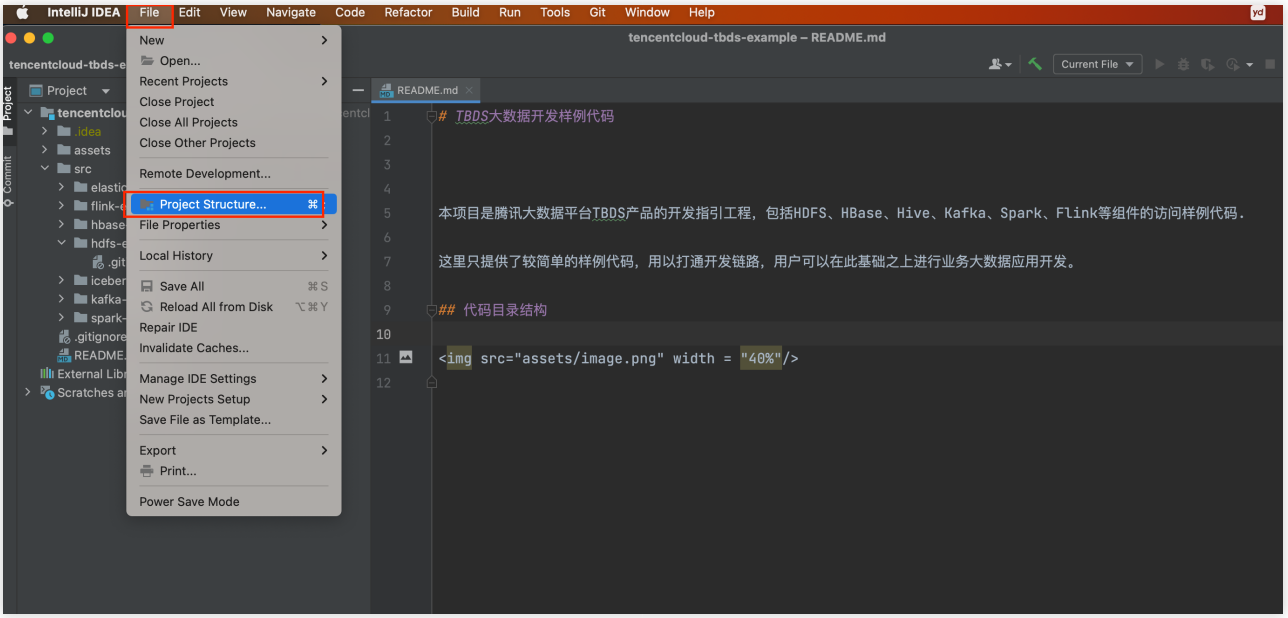
   i. 在GitHub获取样例代码：仓库地址：https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master

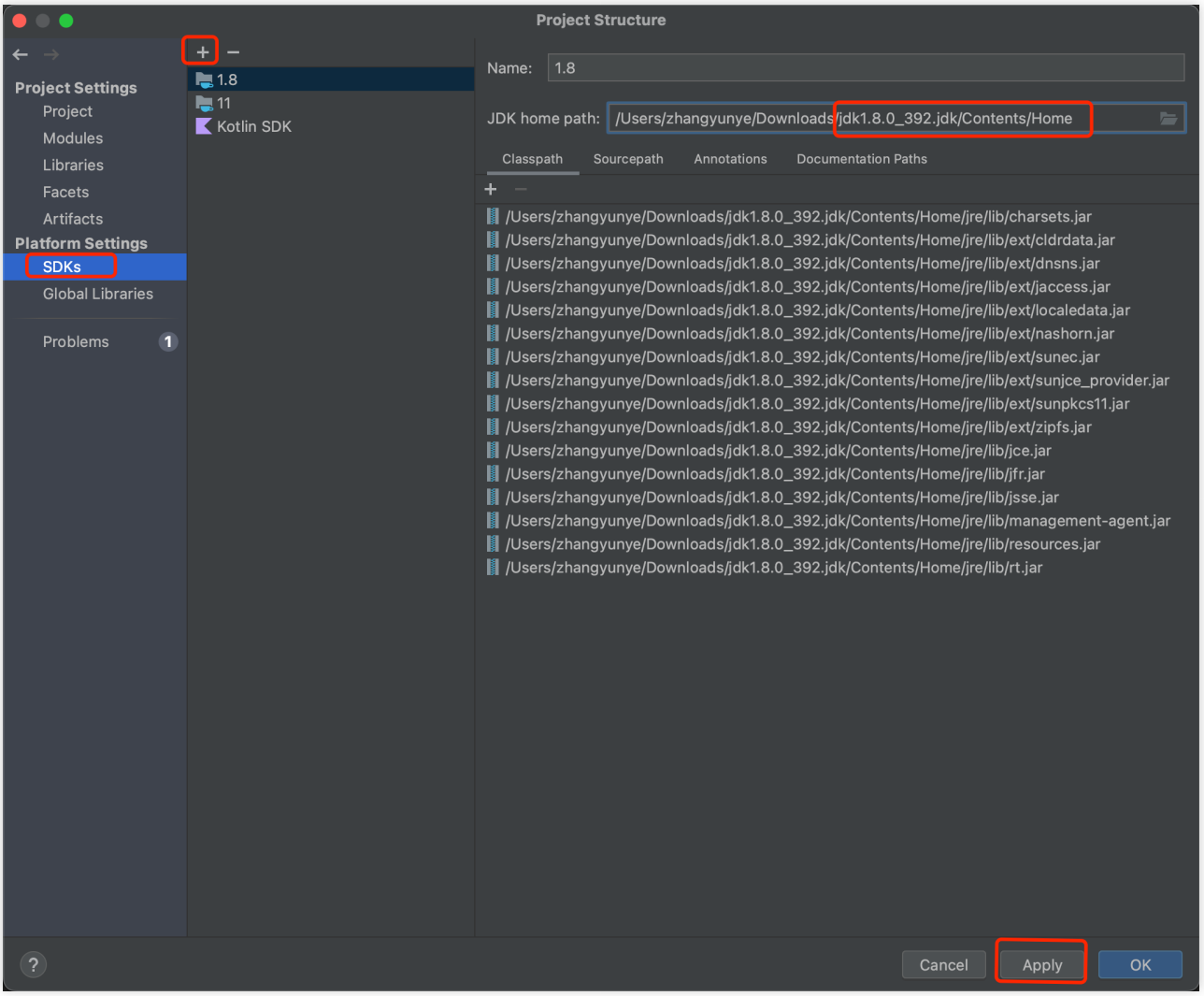
   ii. 参考身份认证和授权 下载的“user.keytab”和“krb5.conf”文件，放到样例工程的“conf”目录下。

   iii. 导入样例工程到IntelliJ IDEA开发环境。



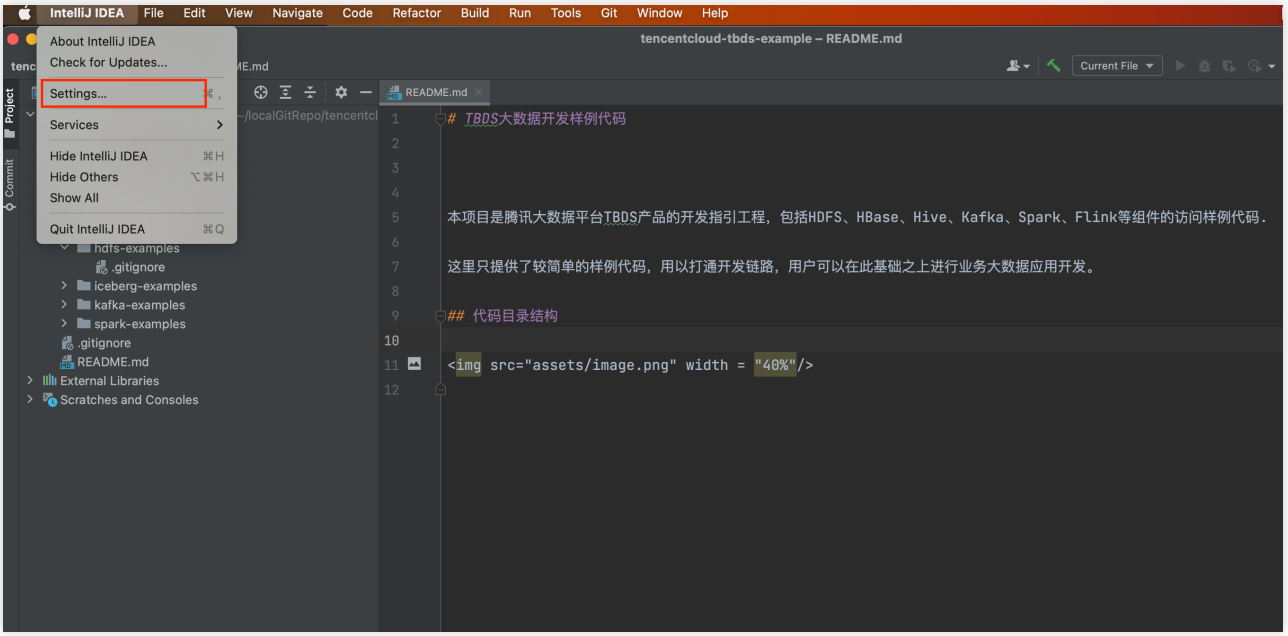
iv. 安装完IntelliJ IDEA和JDK工具后,需要在IntelliJ IDEA中配置JDK。选择 File 下的 Project Structure，点击 SDKs，选择 JDK 1.8，点击 Apply，再点击 OK。若没有 JDK 1.8，则点击 + 号进行添加，点击 Add JDK 后选择 JDK 1.8 安装目录，然后点击 OK 即可。

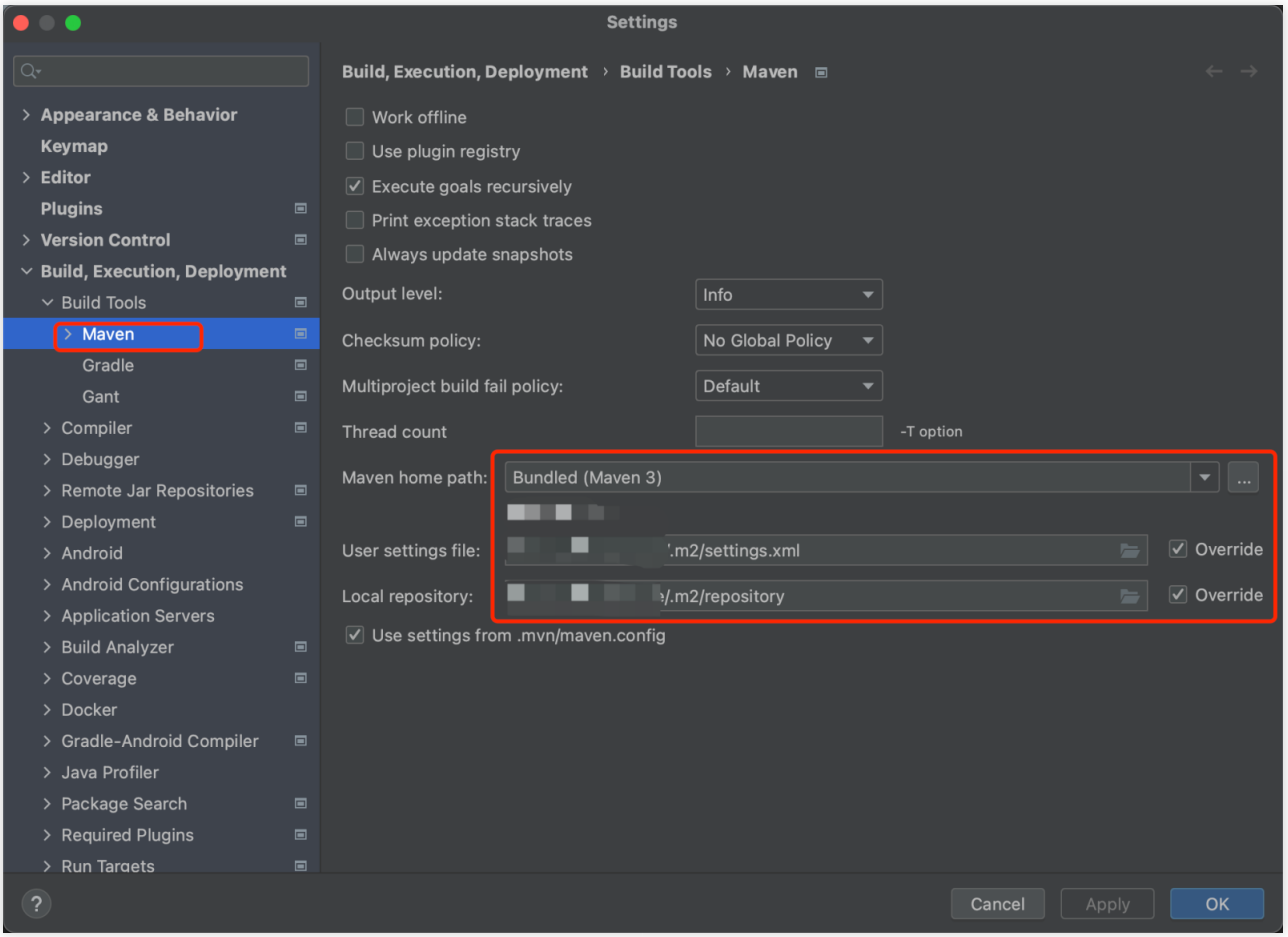




v. 将工程依赖的jar包添加到类路径。需确保在本地安装了 Maven，并配置好了环境变量和 settings.xml 文件。IDEA 点击 Settings 进入配置页面，左上角输入 maven 进行搜索，点击 Build Tools 下的 Maven 配置项，修改 “Maven

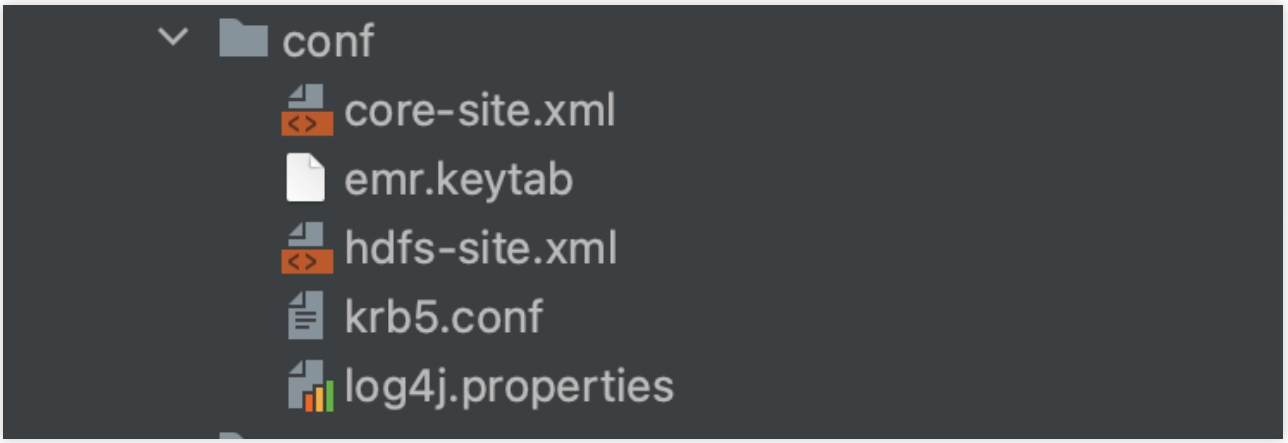
home path” 为本地 Maven 的安装目录，修改 “User settings file” 为本地 Maven settings.xml 配置文件的文件路径，并勾选 Override，此时 “Local repository” 将自动设置为 settings.xml 文件中配置的本地 Maven 仓库的目录。最后点击 Apply，再点击 OK。





vi. 将配置文件放入工程的conf目录

从6.3.1.Kerberos认证方式获取认证信息后，将.keytab文件和krb5.conf文件拷贝到工程的conf目录下；Hadoop的配置文件的路径在/usr/local/service/hadoop/etc/hadoop/，将hdfs的相关配置文件core-site.xml和hdfs-site.xml拷贝到工程的conf目录下。



vii. 认证相关配置修改

基础安全认证

```
    public static Configuration confLoad(String HADOOP_CONF_HOME){
        Configuration conf = new Configuration();
        conf.addResource(new Path(HADOOP_CONF_HOME + "/core-site.xml"));
        conf.addResource(new Path(HADOOP_CONF_HOME + "/hdfs-site.xml"));
        conf.set("fs.hdfs.impl.disable.cache", "true");
        conf.set("dfs.client.failover.max.attempts", "3");
        conf.set("dfs.client.retry.max.attempts", "3");
        conf.set("fs.hdfs.impl", "org.apache.hadoop.hdfs.DistributedFileSystem");
        LOG.info("配置文件: " + conf);
        return conf;
    }
    public static FileSystem authentication(String HADOOP_CONF_HOME, String principal, String keytab)
  throws Exception {
        Configuration conf = confLoad(HADOOP_CONF_HOME);
        if(conf.get("hadoop.security.authentication").equalsIgnoreCase("kerberos")) {
            System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
            UserGroupInformation.setConfiguration(conf);
            UserGroupInformation.loginUserFromKeytab(principal, keytab);//当前用户认证
            LOG.info("kerberos认证成功");
        }
        FileSystem fileSystem = FileSystem.get(conf);
        return fileSystem;
    }
```

代码逻辑说明

1. 功能说明

在使用HDFS提供的API之前，需要先进行HDFS初始化操作。过程为：

- 加载HDFS服务配置文件，并进行Kerberos安全认证。
- 认证通过后，实例化FileSystem对象。
- 调用FileSystem对象，实现HDFS的上传、下载、文件读取、写入、删除、追加功能。

2. 配置文件说明

文件名称	作用
core-site.xml	配置Hadoop集群的核心配置参数。
hdfs-site.xml	配置HDFS详细参数。
EMR.keytab	对于Kerberos安全认证提供用户信息。
krb5.conf	Kerberos Server配置信息。

3. 代码逻辑说明

```
/**
 * 获取初始化配置
 * 如果程序运行在Linux上，则需要core-site.xml、hdfs-site.xml的路径修改
 * 为在Linux下客户端文件的绝对路径
 */
 public static Configuration confLoad(String HADOOP_CONF_HOME){
     Configuration conf = new Configuration();
     conf.addResource(new Path(HADOOP_CONF_HOME + "/core-site.xml"));
     conf.addResource(new Path(HADOOP_CONF_HOME + "/hdfs-site.xml"));
     conf.set("fs.hdfs.impl.disable.cache", "true");
     conf.set("dfs.client.failover.max.attempts", "3");
     conf.set("dfs.client.retry.max.attempts", "3");
     conf.set("fs.hdfs.impl", "org.apache.hadoop.hdfs.DistributedFileSystem");
     LOG.info("配置文件: " + conf);
     return conf;
  }

/**
 *安全认证
 */
 public static FileSystem authentication(String HADOOP_CONF_HOME, String principal, String keytab) throw
s Exception {
     Configuration conf = confLoad(HADOOP_CONF_HOME);
     if(conf.get("hadoop.security.authentication").equalsIgnoreCase("kerberos")) {
         System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
         UserGroupInformation.setConfiguration(conf);
         UserGroupInformation.loginUserFromKeytab(principal, keytab);//当前用户认证
         LOG.info("kerberos认证成功");
     }
     FileSystem fileSystem = FileSystem.get(conf);
```

```
        return fileSystem;
    }

    /**
     *创建用例
     */
    public static void main(String[] args) throws Exception {
        LOG.setLevel(Level.ALL);
        GetProperties getProperties = new GetProperties();
        Properties properties = getProperties.getRequestProperties("demo.hdfs.");
        String confDir = properties.getProperty("confDir");
        String keytab = properties.getProperty("keytab");
        String principal = properties.getProperty("principal");
        LOG.info("confDir: " + confDir +"keytab: " +keytab +"principal: "+principal);
        FileSystem fileSystem = authentication(confDir,principal,keytab);
         //上传文件
        uploadFile(fileSystem);
        //下载文件
        downloadFile(fileSystem);
        //创建并写入文件
        writeFile(fileSystem);
        //读取文件
        readFile(fileSystem);
        //对文件追加内容
        appendContext(fileSystem);
        //删除文件
        deleteFile(fileSystem);
        fileSystem.close();
    }
```

代码调试

以下使用IntelliJ IDEA说明示例工程代码调试过程。

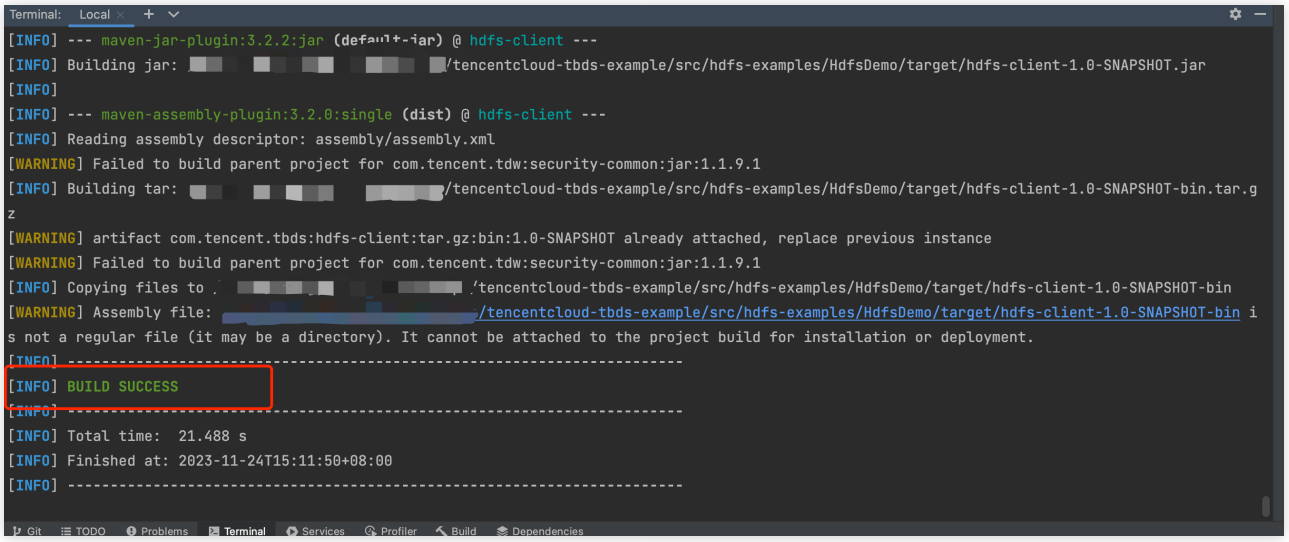
1. 编译代码
   打开IntelliJ IDEA终端，进入HdfsDemo目录。

   ```
   cd src/hdfs-examples/HdfsDemo
   ```

   使用Maven命令打包：

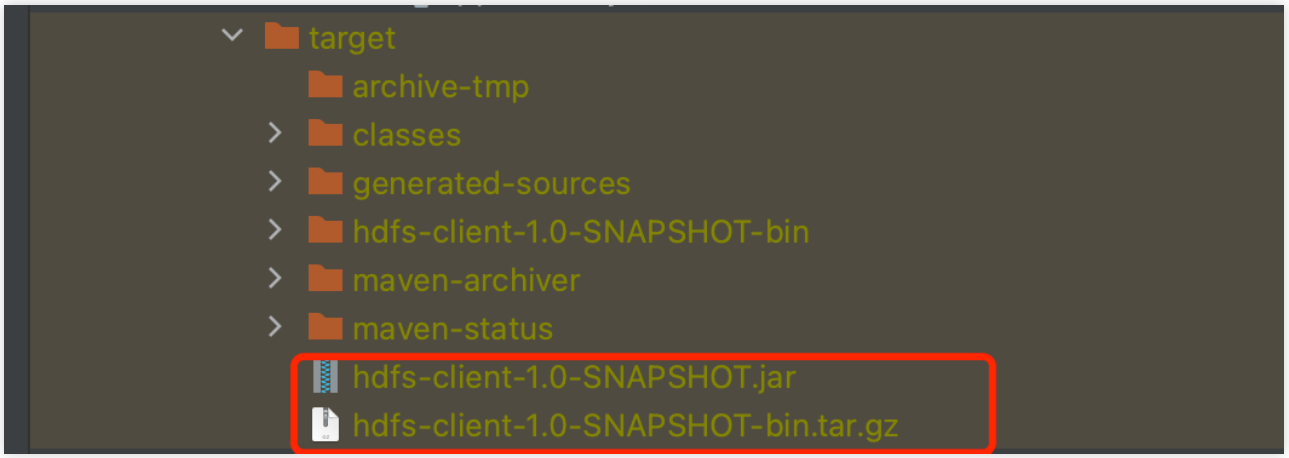
   ```
   mvn install clean package
   ```

   编译成功界面如下：



编译过程中遇到报错需要根据错误信息进行排查和修改，打印“BUILD SUCCESS”为编译成功。

2. 开发机调试
   编译成功后在target目录下得到hdfs-client-1.0-SNAPSHOT-bin.tar.gz和hdfs-client-1.0-SNAPSHOT.jar文件，上传到开发机新建的/data/demo目录进行调试运行。具体的运行方法见下一节打包发布中的介绍。



打包发布

1. 打包
   打包的具体操作见上一节代码调试部分。
2. 发布
   通过上述编译后得到hdfs-client-1.0-SNAPSHOT-bin.tar.gz和hdfs-client-1.0-SNAPSHOT.jar。其中hdfs-client-1.0-SNAPSHOT-bin.tar.gz包含运行过程中依赖的目录和文件；hdfs-client-1.0-SNAPSHOT.jar为hdfs demo的java执行文件。将两个文件上传到开发机，并解压tar包。
   命令如下：

```
#开发机中新建目录
mkdir -p /data/demo
#本地上传压缩文件和jar包
scp hdfs-client-1.0-SNAPSHOT-bin.tar.gz hdfs-client-1.0-SNAPSHOT.jar root@{ip}:/data/demo
#开发机上解压压缩文件
cd /data/demo
tar zxf hdfs-client-1.0-SNAPSHOT-bin.tar.gz
mv conf/application.yml ./
```

开发机demo目录下的文件如下：

```
[test@10 root]$ cd /data/demo/
[test@10 demo]$ ll
total 0
[test@10 demo]$ rz
Sent - hdfs-client-1.0-SNAPSHOT-bin.tar.gz  4.08 MB/s Spend: 14 seconds
Sent - hdfs-client-1.0-SNAPSHOT.jar 135.14 KB/s Spend: 0 seconds
[test@10 demo]$ tar zxf hdfs-client-1.0-SNAPSHOT-bin.tar.gz
[test@10 demo]$ ll
total 60412
drwxr-xr-x 2 test test       53 Nov 28 19:48 conf
-rw-r--r-- 1 test test 61836012 Nov 28 19:52 hdfs-client-1.0-SNAPSHOT-bin.tar.gz
-rwxr-xr-x 1 test test    10102 Nov 28 19:51 hdfs-client-1.0-SNAPSHOT.jar
drwxrwxr-x 2 test test     8192 Dec  7 11:39 lib
[test@10 demo]$ ll conf/
total 8
-rwxr-xr-x 1 test test 484 Nov 28 19:48 application.yml
-rwxr-xr-x 1 test test 662 May 15  2023 log4j.properties
```

3. 开发机运行

1. 准备认证文件和配置文件：参考获取用户认证信息。若是 Kerberos 环境，需要将用户的keytab文件上传至开发机conf。这里将测试用户的test.keytab文件上传至开发机的conf目录。然后将hadoop的配置文件core-site.xml和hdfs-site.xml放入conf目录下。

   ```
   mv test.keytab conf/
   cp /usr/local/service/hadoop/etc/hadoop/hdfs-site.xml conf/
   cp /usr/local/service/hadoop/etc/hadoop/core-site.xml conf/
   ```

   ```
   [test@10 demo]$ ll conf/
   total 36
   -rwxr-xr-x 1 test test   484 Nov 28 19:48 application.yml
   -rwxr-xr-x 1 test test 10941 Dec  7 11:44 core-site.xml
   -rwxr-xr-x 1 test test  9504 Dec  7 11:43 hdfs-site.xml
   -rwxr-xr-x 1 test test   662 May 15  2023 log4j.properties
   -rw-r--r-- 1 test test   138 Dec  7 11:23 test.keytab
   [test@10 demo]$
   ```

2. 准备测试文件：准备名为download.txt，read.txt，append.txt和delete.txt的测试文件，然后将这几个文件上传至 HDFS 的/hdfs_demo目录。注意，如果是 Kerberos 环境，需要首先进行认证。然后准备upload.txt，放在开发机的demo目录下。

操作命令：

```
#kerberos认证
klist -kt /data/demo/conf/test.keytab
kinit -kt /data/demo/conf/test.keytab test@TBDS-xxx
/usr/local/service/hadoop/bin/hdfs dfs -mkdir /hdfs_demo
/usr/local/service/hadoop/bin/hdfs dfs -put delete.txt read.txt append.txt download.txt /hdfs_demo
```

```
[test@10 demo]$ ll
total 60436
-rw-r--r-- 1 test test       17 Nov 28 17:41 append.txt
drwxr-xr-x 2 test test       53 Nov 28 19:48 conf
-rw-r--r-- 1 test test       17 Nov 28 17:40 delete.txt
-rw-r--r-- 1 test test       18 Nov 28 17:41 download.txt
-rw-r--r-- 1 test test 61836012 Nov 28 19:52 hdfs-client-1.0-SNAPSHOT-bin.tar.gz
-rwxr-xr-x 1 test test    10102 Nov 28 19:51 hdfs-client-1.0-SNAPSHOT.jar
drwxrwxr-x 2 test test     8192 Dec  7 11:39 lib
-rw-r--r-- 1 test test       15 Nov 28 17:41 read.txt
-rw-r--r-- 1 test test      138 Dec  7 11:23 test.keytab
-rw-r--r-- 1 test test       16 Nov 28 17:41 upload.txt
```

```
[test@10 demo]$ hdfs dfs -ls /hdfs_demo
2023-12-07T11:42:48,462 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2023-12-07T11:42:48,678 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SE
CONDS
2023-12-07T11:42:48,721 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SE
CONDS
Found 4 items
-rwxrwxrwx   3 test supergroup         33 2023-12-04 10:45 /hdfs_demo/append.txt
-rwxrwxrwx   3 test supergroup         18 2023-11-28 19:30 /hdfs_demo/download.txt
-rwxrwxrwx   3 test supergroup         15 2023-12-04 10:41 /hdfs_demo/read.txt
-rwxrwxrwx   3 test supergroup         18 2023-12-04 10:45 /hdfs_demo/write.txt
```

修改application.yml的内容，填写demo文件夹中测试文件的路径和hdfs中测试文件的路径。

```
[test@10 demo]$ cat application.yml
demo:
  hdfs:
    confDir: "conf"
    keytab: "conf/test.keytab"
    principal: "test@TBDS-[redacted]"
    localFile: "/data/demo/upload.txt"
    uploadPath: "/hdfs_demo/"
    localPath: "/data/demo/"
    hdfsFile: "/hdfs_demo/download.txt"
    writeFile: "/hdfs_demo/write.txt"
    writeContent: "Hello Hadoop World"
    readFile: "/hdfs_demo/read.txt"
    appendFile: "/hdfs_demo/append.txt"
    appendContent: "Append Something"
    deleteFile: "/data/demo/delete.txt"
[test@10 demo]$
```

3. Ranger 授权：参考Ranger授权（经典集群），确保 Hadoop用户具有读写权限。

4. 安全认证：HDFS在Kerberos 环境下认证方式有两种，如下表所示。

认证方式	认证说明

命令认证	在提交HDFS 任务前，使用 kinit -kt {keytab} {principal} 命令进行认证
代码认证	获取用户的 keytab 和 principal 后，在应用程序的代码中进行认证

5. 运行样例：在开发机上进入/data/demo目录，执行命令运行jar文件，更多参数说明参考常用命令。该demo演示了对HDFS JAVA API的基本实现，包括：上传、下载、读取、写入、追加、删除等功能。
   /usr/local/jdk/bin/java -jar hdfs-client-1.0-SNAPSHOT.jar
   开发机上的运行过程：

```
[test@10 demo]$ java -jar hdfs-client-1.0-SNAPSHOT.jar
11:47:18.676 [main] INFO  com.tencent.tbds.HdfsClient - confDir: confkeytab: conf/test.keytabprincipal: test@TBDS-
11:47:18.775 [main] INFO  com.tencent.tbds.HdfsClient - 配置文件: Configuration: core-default.xml, core-site.xml, conf/
core-site.xml, conf/hdfs-site.xml, conf/yarn-site.xml, conf/mapred-site.xml
11:47:18.913 [main] INFO  com.tencent.tbds.HdfsClient - kerberos认证成功
11:47:19.595 [main] INFO  com.tencent.tbds.HdfsClient - 上传本地文件: /data/demo/upload.txt 到hdfs路径: /hdfs_demo/
11:47:19.641 [main] INFO  com.tencent.tbds.HdfsClient - 下载hdfs文件: /hdfs_demo/download.txt 到本地路径: /data/demo/
11:47:19.664 [main] INFO  com.tencent.tbds.HdfsClient - 写入文本到hdfs文件: /hdfs_demo/write.txt
11:47:19.670 [main] INFO  com.tencent.tbds.HdfsClient - 读取hdfs文件: read from hdfs
11:47:19.765 [main] INFO  com.tencent.tbds.HdfsClient - 追加内容到hdfs文件:/hdfs_demo/append.txt
```

   HDFS上的运行结果：

```
[test@10 demo]$ hdfs dfs -ls /hdfs_demo
2023-12-07T12:02:56,511 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2023-12-07T12:02:56,713 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2023-12-07T12:02:56,757 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
Found 5 items
-rwxrwxrwx   3 test supergroup         49 2023-12-07 11:47 /hdfs_demo/append.txt
-rwxrwxrwx   3 test supergroup         18 2023-11-28 19:30 /hdfs_demo/download.txt
-rwxrwxrwx   3 test supergroup         15 2023-12-04 10:41 /hdfs_demo/read.txt
-rw-r--r--   3 test supergroup         16 2023-12-07 11:47 /hdfs_demo/upload.txt
-rw-r--r--   3 test supergroup         18 2023-12-07 11:47 /hdfs_demo/write.txt
[test@10 demo]$ hdfs dfs -cat /hdfs_demo/append.txt
2023-12-07T12:03:21,509 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2023-12-07T12:03:21,716 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2023-12-07T12:03:21,762 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
append test file
Append SomethingAppend Something[test@10 demo]$
[test@10 demo]$ hdfs dfs -cat /hdfs_demo/write.txt
2023-12-07T12:03:51,304 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2023-12-07T12:03:51,503 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2023-12-07T12:03:51,546 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
Hello Hadoop World[test@10 demo]$
[test@10 demo]$
```

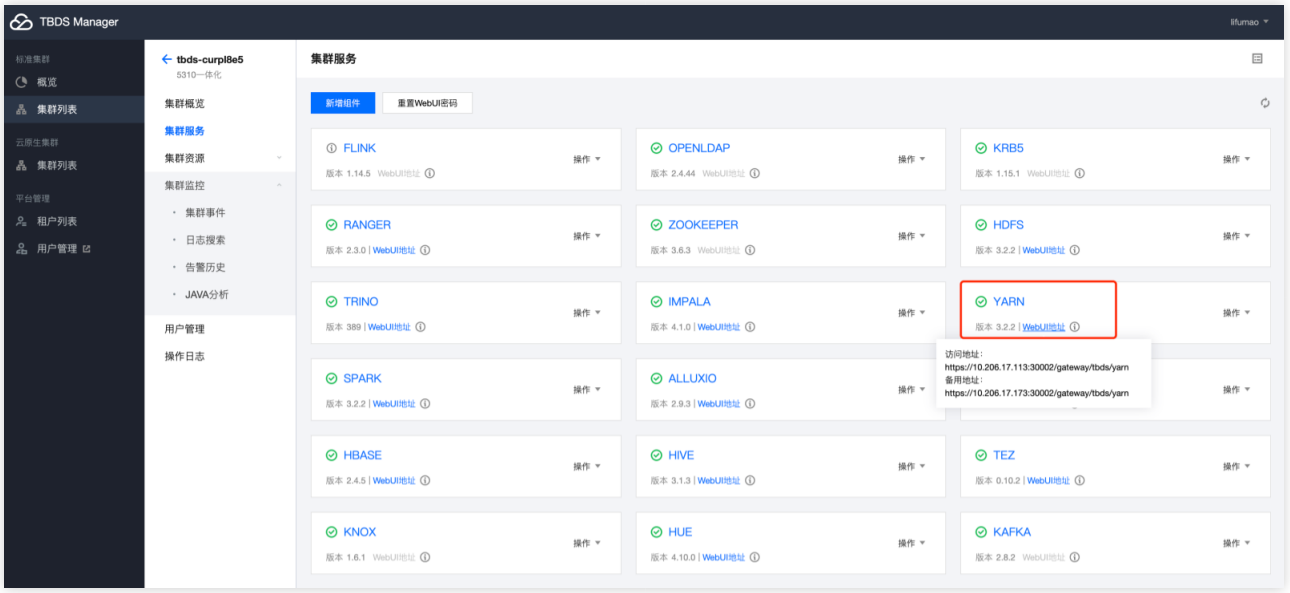
程序运维监控

HDFS提交YARN任务的时候可以通过进入YARN UI界面查看运行状态和运行日志。具体可以使用示例说明中的命令进行测试，然后体验程序的运维监控功能。本小节以提交MapReduce任务为例，具体命令如下：

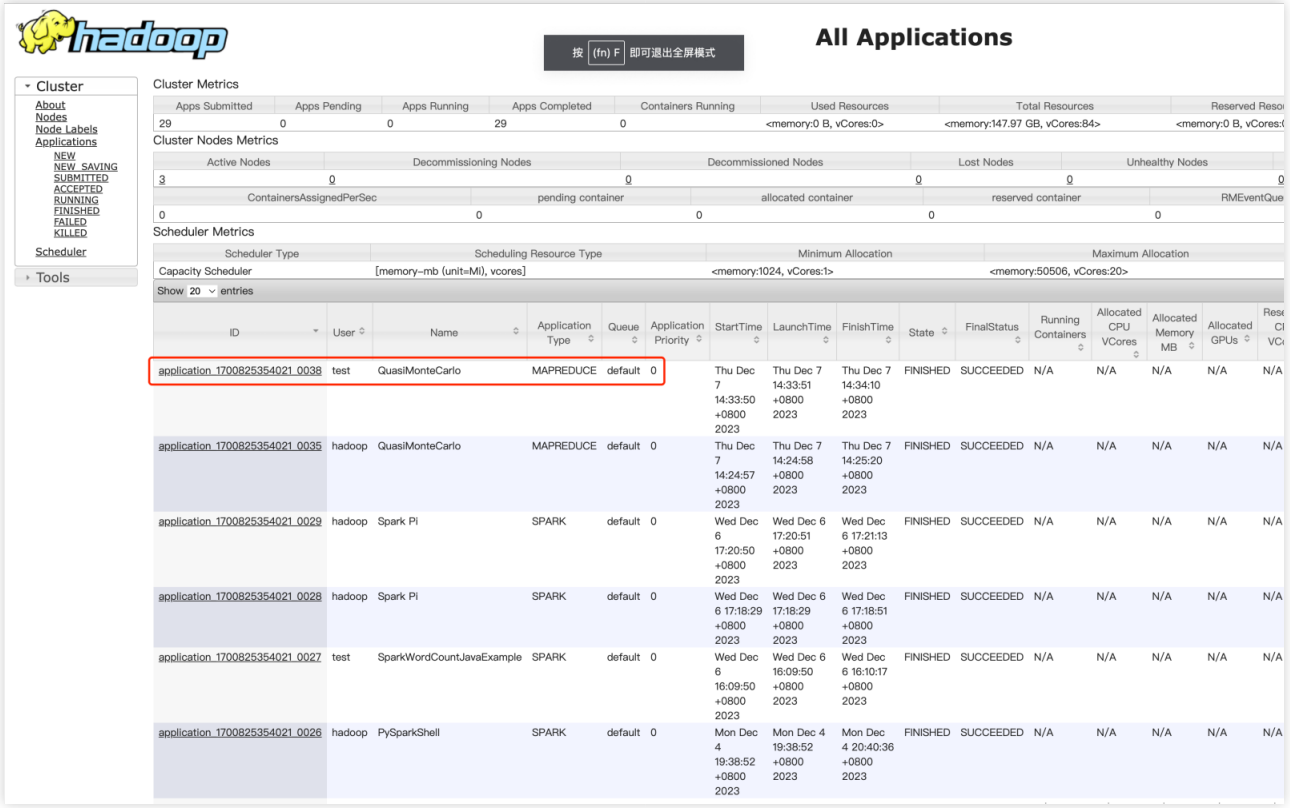
```
#kerberos认证
/usr/bin/klist -kt /var/krb5kdc/emr.keytab
/usr/bin/kinit -kt /var/krb5kdc/emr.keytab hadoop/xxx@xxxxx

#提交任务
cd /usr/local/service/hadoop/bin
./hadoop jar ../share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar wordcount  /test  /output
./hdfs dfs -cat /output/*
```

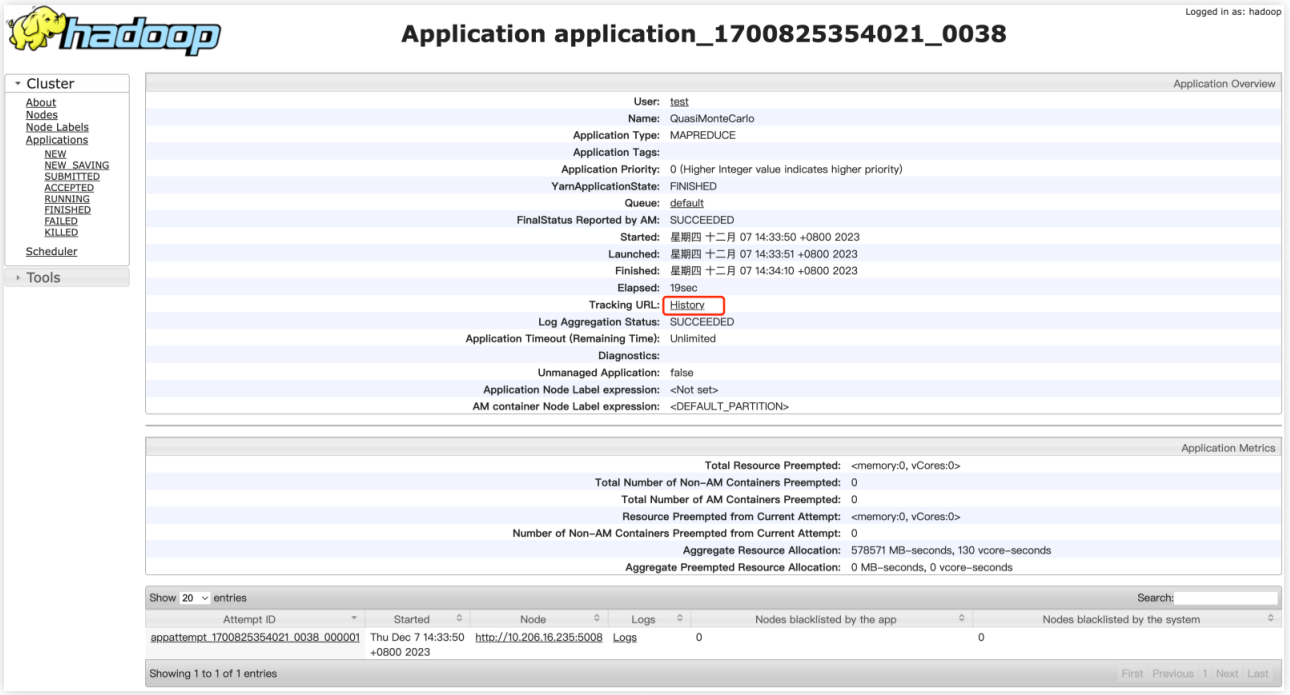
1. 监控

登录 TBDS Manager 管控平台，点击“集群列表”，选择 HDFS 程序运行所对应的集群。点击“集群服务”，选择 YARN 服务，点击 WebUI 地址跳转至 YARN UI 界面（注意YARN UI的ip地址需要替换为公网ip）。



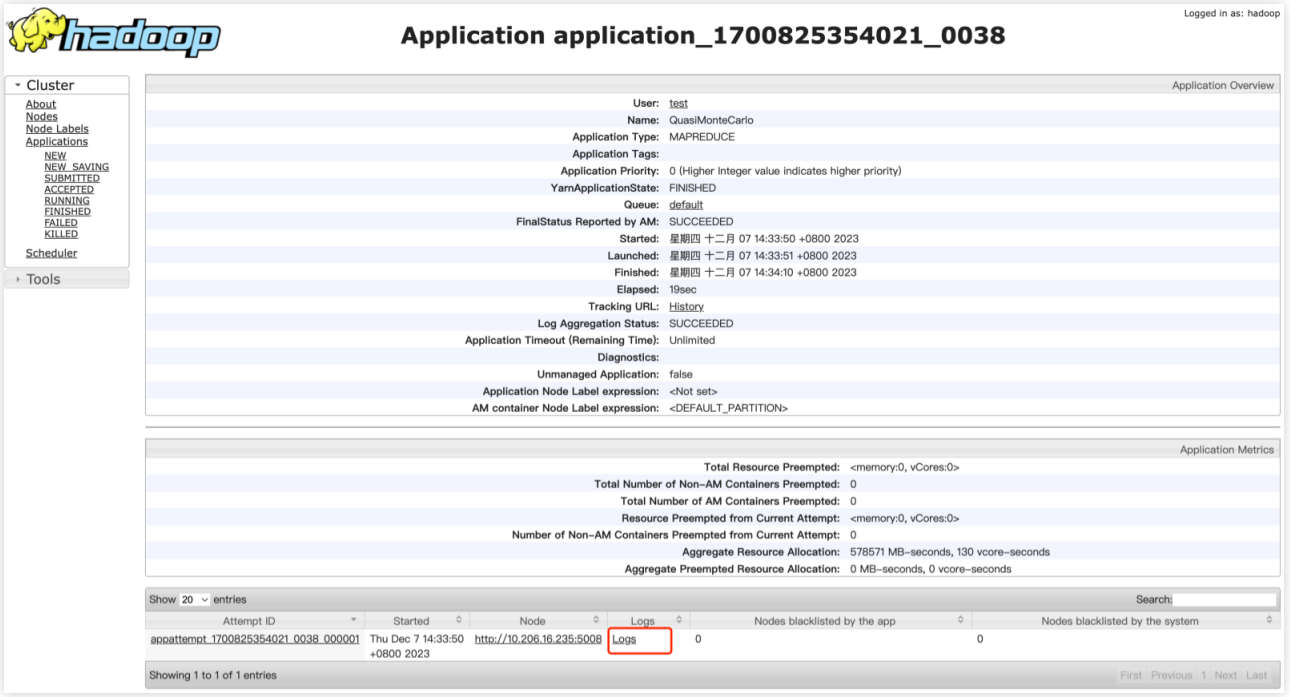
YARN UI 页面根据 ApplicationID、User、Name 等信息，找到对应的 MapReduce 任务。由于 HDFS 任务的 ApplicationID 在日志级别为 INFO 及以下才会输出至控制台，而HDFS默认的日志级别为 WARN，因此这里可以根据任务类型等信息进行查找对应的Mapreduce任务，点击 ApplicationID 链接，然后点击 Tracking URL 链接，跳转作业监控页面。





2. 日志查看

在 YARN UI 页面根据 ApplicationID、Name 等信息，找到对应的MapReduce任务，点击 Logs 查看日志。



3. 作业操作

与其它提交至YARN的任务类似，可以通过YARN相关命令查看、停止任务，参考常用命令。可以通过命令 yarn application -kill 停止MapReduce任务；通过命令 yarn logs -applicationId 查看Mapreduce的任务日志。

示例说明

hadoop-MapReduce-examples

kerberos认证
/usr/bin/klist -kt /var/krb5kdc/emr.keytab
/usr/bin/kinit -kt /var/krb5kdc/emr.keytab hadoop/xxx@xxxxx

yarn提交任务
cd /usr/local/service/hadoop/bin
./yarn jar ../share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar pi -Dmapreduce.job.queuename=default 16 1000
预期结果
Estimated value of Pi is 3.14250000000000000000

mr提交任务
cd /usr/local/service/hadoop/bin
./hadoop jar ../share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar wordcount /test /output
./hdfs dfs -ls /output/*

生产随机数
./hadoop jar ../share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar randomwriter random

排序
./hadoop jar ../share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2.jar sort random sort

联邦示例说明

参考：RBF组件示例

api接口

TBDS大数据平台上的HDFS访问接口与开源兼容，可以参考Apache Hadoop Main 3.2.2 API。

常用命令

命令说明

	组件	操作	命令	描述
hadoop命令	HDFS 文件系统操作基本命令	查看目录	ls	查看hdfs目录下的文件和目录
		查看文件	cat	查看hdfs文件的内容输出至终端
		上传文件	put	将本地文件上传至hdfs文件系统
		下载文件	get	下载hdfs文件至本地机器
		创建目录	mkdir	创建目录到hdfs系统
		删除目录/文件	rm	将hdfs系统上的文件/目录删除
		计算使用量	du	计算hdfs系统上的文件/目录使用量
		修改权限	chmod	修改hdfs系统上的文件/目录权限
		修改属主	chown	修改hdfs系统上的文件/目录属主
	HDFS 文件系统管理命令（dfsadmin）	获取集群报告	report	显示集群的总体健康状况
		元数据的导出	metasave	将NameNode的元数据信息保存到一个文件
	YARN 调度基本命令	作业操作命令	app	打印应用程序报告/终止应用程序/管理长期运行的应用程序
		打印日志	logs	打印容器日志
		打印Node报告	Node	打印Node相关的详细报告
	YARN 调度管理命令（rmadmin）	刷新队列配置	refreshQueues	用于刷新ResourceManager中的队列配置
	联邦管理命令（dfsrouteradmin）	挂载表查看	ls	查看联邦挂载表详细条目
		挂载表添加	add	添加联邦挂载表至服务端
		挂载表更新	update	更新联邦已有挂载表
		挂载表删除	rm	删除已有挂载表

命令示例说明

hdfs基础功能
cd /usr/local/service/hadoop/bin
查看目录 ./hadoop fs -ls /
读取文件 ./hadoop fs -cat /test/tesst.txt
创建文件 ./hadoop fs -put /tmp/tesst.txt /test
下载文件 ./hadoop fs -get /tmp/tesst.txt
创建目录 ./hadoop fs -mkdir /test
删除目录 ./hadoop fs -rm -skipTrash -r /test
删除文件 ./hadoop fs -rm -skipTrash /tmp/tesst.txt
计算使用量 ./hadoop fs -du /test
修改权限 ./hadoop fs -chmod 777 /test
修改属主 ./hadoop fs -chown root:root /test

hdfs管理功能
查看当前集群报告 ./hdfs dfsadmin -report
统计信息 ./hdfs dfsadmin -metasave filename

版权所有：TCloudFinanceZone

yarn基础功能
cd /usr/local/service/hadoop/bin
显示作业列表 ./yarn app -list
打印作业容器日志 ./yarn logs -applicationId <ApplicationId>

yarn管理功能
查看node列表 ./yarn node -all -list
查看node状态 ./yarn node -status {ip}:{port}
管理命令查看 ./yarn rmadmin
刷新队列配置 ./yarn rmadmin -refreshQueues

联邦管理命令
挂载表查看 hdfs dfsrouteradmin -d -ls
挂载表添加 hdfs dfsrouteradmin -add /user/user1/.Trash ns1,ns2 /user/user1/.Trash -order HASH_ALL
挂载表更新 hdfs dfsrouteradmin -update /user/user1/.Trash ns1,ns2 /user/user1/.Trash -order HASH_ALL
挂载表删除 hdfs dfsrouteradmin -rm /data

详细命令参考：Apache Hadoop 3.2.2 – Hadoop Commands Guide。

联邦常用配置说明

配置项	值	说明
dfs.federation.router.default.nameserviceId	无	要监视的默认子集群的
dfs.federation.router.store.driver.class	org.apache.hadoop.hdfs.server.federation.store.driver.impl.StateStoreZooKeeperImpl	StateStore的实现类
dfs.federation.router.monitor.namenode	无	Router检测的Namenode列表
dfs.federation.router.rpc-address	0.0.0.0:8888	RPC地址处理所有客户端请求。
dfs.federation.router.handler.count	10	路由器处理RPC客户端请求的handler数量。
dfs.federation.router.handler.queue.size	1	Router处理RPC客户端请求的reader数量。

联邦命令说明

参考：RBF组件示例

常见问题

NameNode或JournalNode的editlogs目录占用较大磁盘空间

1. 问题根因：HDFS依赖FsImage Checkpoint进行editlogs合并，当FsImage Checkpoint出现异常时，会导致editlogs无法合并，通常异常情况是由于FsImage目录写满或磁盘异常等情况引起的，此时NameNode重启时间也会变得很长。NameNode FsImage Checkpoint失败后，该FsImage目录默认不会再次使用，需要设置开启FsImage目录自恢复功能。
2. 解决方案：

```
#手动开启FsImage目录自恢复选项
su - hdfs
hdfs dfsadmin -restoreFailedStorage true
#开启成功后NameNode后续自动FsImage Checkpoint时，会先尝试恢复FsImage目录
restoreFailedStorage is set to true
```

3. 默认开启FsImage目录自恢复功能：
   在TM的HDFS服务页面，选择配置 > hdfs-site.xml ，添加参数为dfs.namenode.name.dir.restore，值为true的自定义配置，下次重启NameNode后FsImage目录自恢复功能会自动打开。

写入文件提示DataXceiver Premature EOF from InputStream

1. 具体报错：DataXceiver error processing WRITE_BLOCK operation src: /10.*...:35692 dst: /10.*...*:50010 java.io.IOException: Premature EOF from inputStream。
2. 问题根因：通常为了不断地向HDFS写入新的数据，作业会打开较多的HDFS文件写入流（Stream）。但HDFS允许同时打开的文件数量是有限的，受限于DataNode参数，超过限制就会出现DataXceiver Premature EOF from inputStream异常。
3. 解决办法：

- 查看DataNode日志

```
java.io.IOException: Xceiver count 4097 exceeds the limit of concurrent xcievers: 4096
at org.apache.hadoop.hdfs.server.datanode.DataXceiverServer.run(DataXceiverServer.java:150)
```

- 如果存在以上报错，进行如下修改。
  在TM的HDFS服务页面，选择配置，搜索参数dfs.datanode.max.transfer.threads，并适当地调大该参数值，

一般建议翻倍增加，例如8192、16384。

权限策略配置

配置步骤

1. 安装 Ranger Plugin：

   i. 下载并安装 Ranger HDFS 插件。

2. 配置 HDFS：

   i. 修改 hdfs-site.xml 文件，添加以下配置：

```
<property>
  <name>dfs.namenode.inode.attributes.provider.class</name>
  <value>org.apache.ranger.authorization.hadoop.RangerHdfsAuthorizer</value>
</property>
```

3. 配置 Ranger：

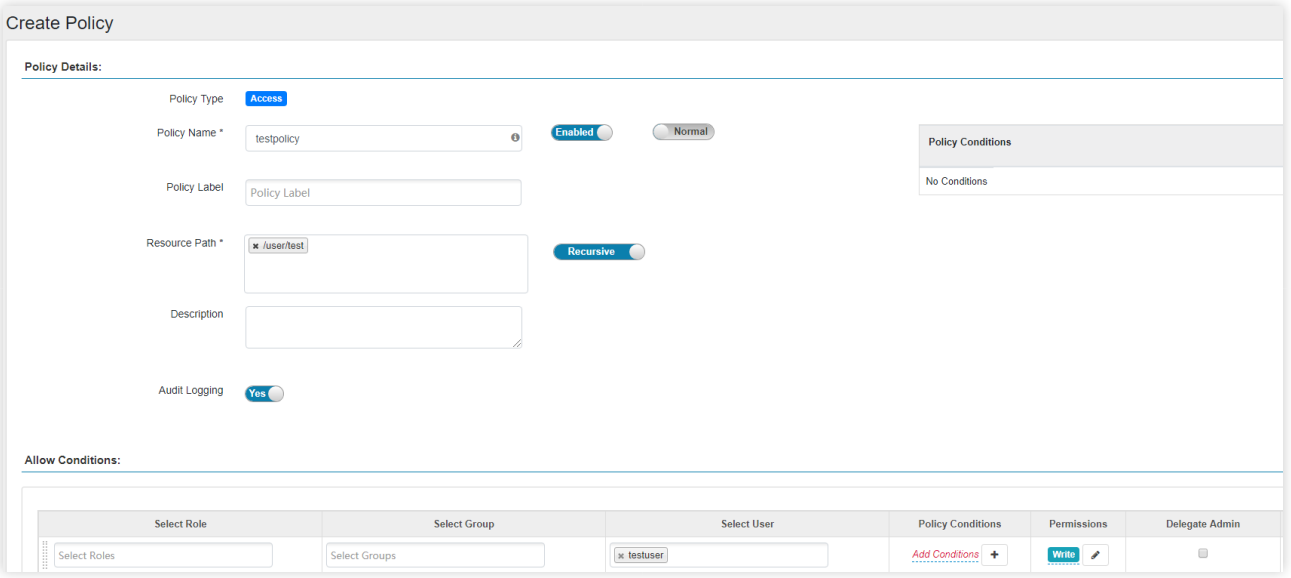
   i. 使用Ranger管理员用户rangeradmin登录Ranger管理页面。
   ii. 在首页中单击“HDFS”区域的组件插件名称，例如“tbdscluster”。
   iii. 单击“Add New Policy”，添加HDFS权限控制策略。
   iv. 根据业务需求配置相关参数。

   HDFS权限参数如下：

参数名称	描述
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。
Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持“*”通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。
Policy Label	为当前策略指定一个标签，可以根据这些标签搜索报告和筛选策略。
Resource Path	资源路径，配置当前策略适用的HDFS路径文件夹或文件，可填写多个值，支持使用通配符“*”（例如“/test/*”）。 如需子目录继承上级目录权限，可打开递归开关按钮。 如果父目录开启递归，同时子目录也配置了策略，以子目录策略为准。 ▪ non-recursive：关闭递归 ▪ recursive：打开递归
Description	策略描述信息。
Audit Logging	是否审计此策略。
Allow Conditions	策略允许条件，配置本策略内允许的权限及例外，例外条件优先级高于正常条件。 在“Select Role”、“Select Group”、“Select User”列选择已创建好的需要授予权限的Role、用户组或用户，单击“Add Conditions”，添加策略适用的IP地址范围，单击“Add Permissions”，添加对应权限。

	▪ Read：读权限 ▪ Write：写权限 ▪ Execute：执行权限 ▪ Select/Deselect All：全选/取消全选 如需让当前条件中的用户或用户组管理本条策略，可勾选“Delegate Admin”使这些用户或用户组成为受委托的管理员。被委托的管理员可以更新、删除本策略，还可以基于原始策略创建子策略。 如需添加多条权限控制规则，可单击 + 按钮添加。如需删除权限控制规则，可单击 ✖ 按钮删除。 Exclude from Allow Conditions：配置排除在允许条件之外的例外规则。
Deny All Other Access	是否拒绝其他所有访问。 ▪ True：拒绝其他所有访问。 ▪ False：设置为false，可配置Deny Conditions。
Deny Conditions	策略拒绝条件，配置本策略内拒绝的权限及例外，配置方法与“Allow Conditions”类似，拒绝条件的优先级高于“Allow Conditions”中配置的允许条件。 Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

例如为用户“testuser”添加“/user/test”目录的写权限，配置如下：



设置权限场景如下：

任务场景	角色授权操作
设置HDFS管理员权限	1. 在首页中单击“HDFS”区域的组件插件名称，例如“hacluster”。 2. 选择“Policy Name”为“all - path”的策略，单击 按钮编辑策略。

	3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。
设置用户执行HDFS检查和HDFS修复的权限	1. 在“Resource Path”配置文件夹或文件。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Read”和“Execute”。
设置用户读取其他用户的目录或文件的权限	1. 在“Resource Path”配置文件夹或文件。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Read”和“Execute”。
设置用户在其他用户的文件写入数据的权限	1. 在“Resource Path”配置文件夹或文件。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Write”和“Execute”。
设置用户在其他用户的目录新建或删除子文件、子目录的权限	1. 在“Resource Path”配置文件夹或文件。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Write”和“Execute”。
设置用户在其他用户的目录或文件执行的权限	1. 在“Resource Path”配置文件夹或文件。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Execute”。
设置子目录继承上级目录权限	1. 在“Resource Path”配置文件夹或文件。 2. 打开递归开关按钮，“Recursive”即为打开递归。

5. （可选）添加策略有效期。在页面右上角单击“Add Validity period”，设置“Start Time”和“End Time”，选择“Time Zone”。单击“Save”保存。如需添加多条策略有效期，可单击按钮添加。如需删除策略有效期，可单击按钮删除。
6. 单击“Add”，在策略列表可查看策略的基本信息。等待策略生效后，验证相关权限是否正常。
   如需禁用某条策略，可单击按钮编辑策略，设置策略开关为“Disabled”。
   如果不再使用策略，可单击按钮删除策略。
7. 重启 HDFS：
   i. 重启 HDFS 服务以应用配置。

机架感知配置

使用场景

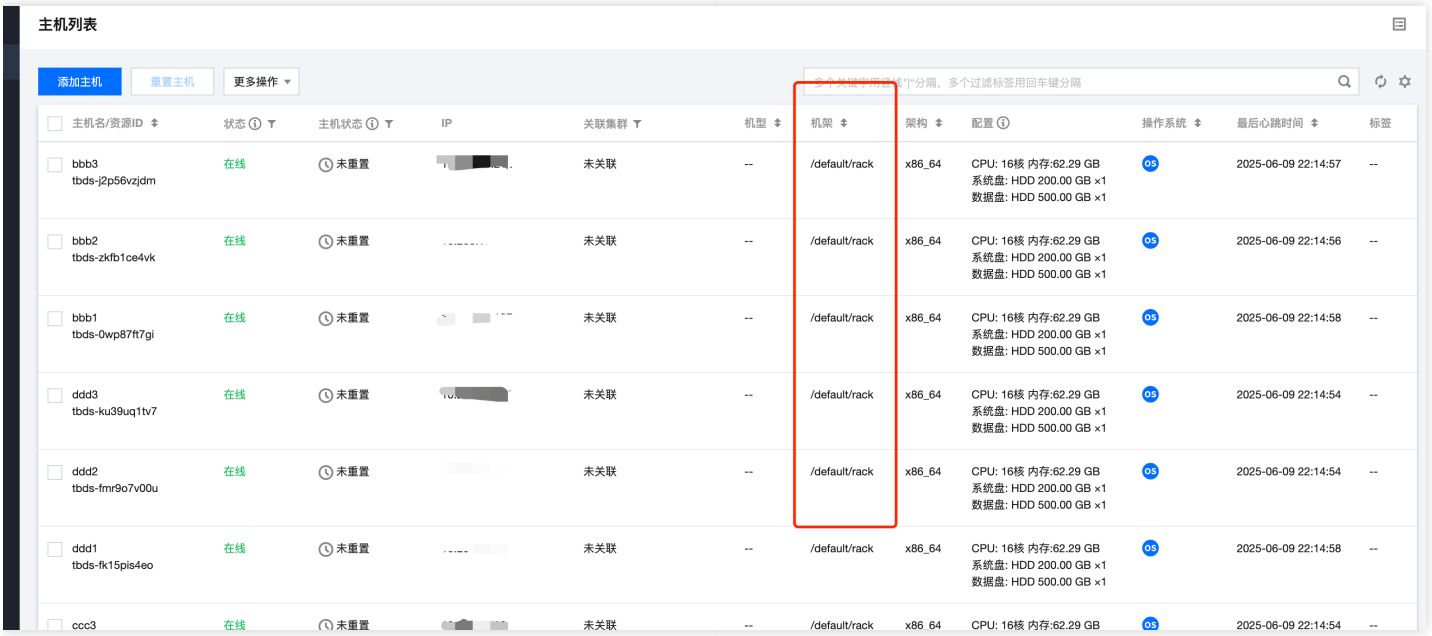
HDFS 的机架感知（Rack Awareness）是一种策略，用于优化数据块的放置和读取，以提高数据的可靠性和读写性能。通过机架感知，HDFS 可以确保数据块的副本分布在不同的机架上，从而在机架故障时仍能保证数据的可用性。

配置方法

默认情况下，HDFS 的机架感知没有被启用，所有节点都被认为在同一个默认机架/default-rack下。这种配置可能导致数据块的副本放置在同一个机架上，增加了单点故障的风险。
我们可以通过如下配置，打开机架感知。

```
<property>
  <name>net.topology.script.file.name</name>
  <value>/usr/local/service/hadoop/etc/hadoop/topology.py</value>
</property>
```

默认机架信息文件： /usr/local/service/rack/topology_map.py，无需后台修改，由TM管控页面的主机列表机架信息决定（具体设置方式参看主机运维章节）



HDFS要启用机架感知，需要在TM页面的core-site.xml 配置文件中设置 net.topology.script.file.name，设置为/

usr/local/service/hadoop/etc/hadoop/topology.py（该文件已在后台存在，无需修改），并滚动重启HDFS。

1. 机架感知副本放置策略。

- 第一个副本放在客户端所在的节点（如果客户端不在集群中，则随机选择一个节点）。
- 第二个副本放在与第一个副本不同机架的节点上。
- 第三个副本放在与第二个副本相同机架的不同节点上。

2. 机架配置限制。

- 层级结构：HDFS 机架感知的网络拓扑结构通常是一个树状结构，每个节点的层级数需要一致。例如，如果一个节点的路径是 /dc1/rack1，那么所有节点的路径都应该遵循类似的层级结构。
- 副本数限制：默认情况下，HDFS 的副本数为 3。如果配置了机架感知，副本的放置策略会根据机架感知的配置来决定。副本数不应超过 DataNode 的数量。
- 脚本文件限制：net.topology.script.file.name 指定的脚本文件必须具有可执行权限，并且能够正确返回每个 DataNode 的机架信息。

生效验证

通过如下命令获取机架感知是否生效。

```
hdfs dfsadmin -printTopology
```

Spark开发
概述

Spark 作为 Apache 高级的开源项目，是一个快速、通用的大规模数据处理引擎，与 Hadoop 的 MapReduce 计算框架类似，但是相对于 MapReduce，Spark 凭借其可伸缩、基于内存计算等特点以及可以直接读写 Hadoop 上任何格式数据的优势，进行批处理时更加高效，并有更低的延迟。实际上，Spark 已经成为轻量级大数据快速处理的统一平台，各种不同的应用，如实时流处理、机器学习、交互式查询等，都可以通过 Spark 建立在不同的存储和运行系统上。

Spark 是基于内存计算的大数据并行计算框架。Spark 基于内存计算，提高了在大数据环境下数据处理的实时性，同时保证了高容错性和高可伸缩性，允许用户将 Spark 部署在大量廉价硬件之上，形成集群。

示例工程开发

环境准备

1. 开发环境准备（Java/Scala）

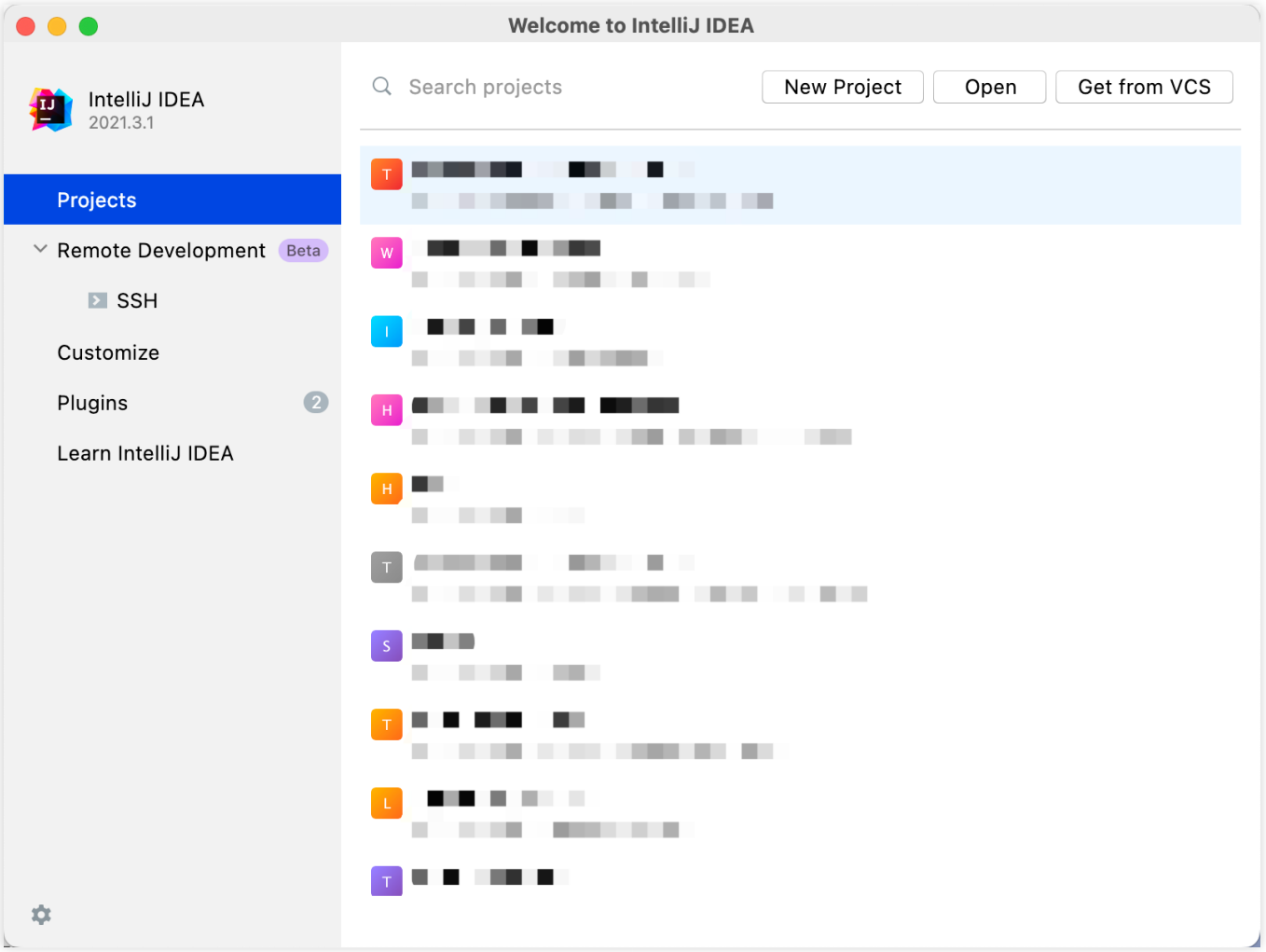
准备项	说明
安装JDK	JDK 8、JDK 11 ，推荐使用 konaJDK，下载地址
安装 Scala	Scala 2.12，下载地址
安装和配置 IDE	按需选择，比如 IntelliJ IDEA 或 Eclipse
安装 Maven	开发环境基础配置，负责构建 Java 应用程序
Maven 配置准备	如果需要本地调试，需要配置 Maven pom.xml，推荐 Maven 3.6.3，下载地址

2. 远程调试环境准备
   以下使用 Linux 环境作为开发机进行应用调试说明。

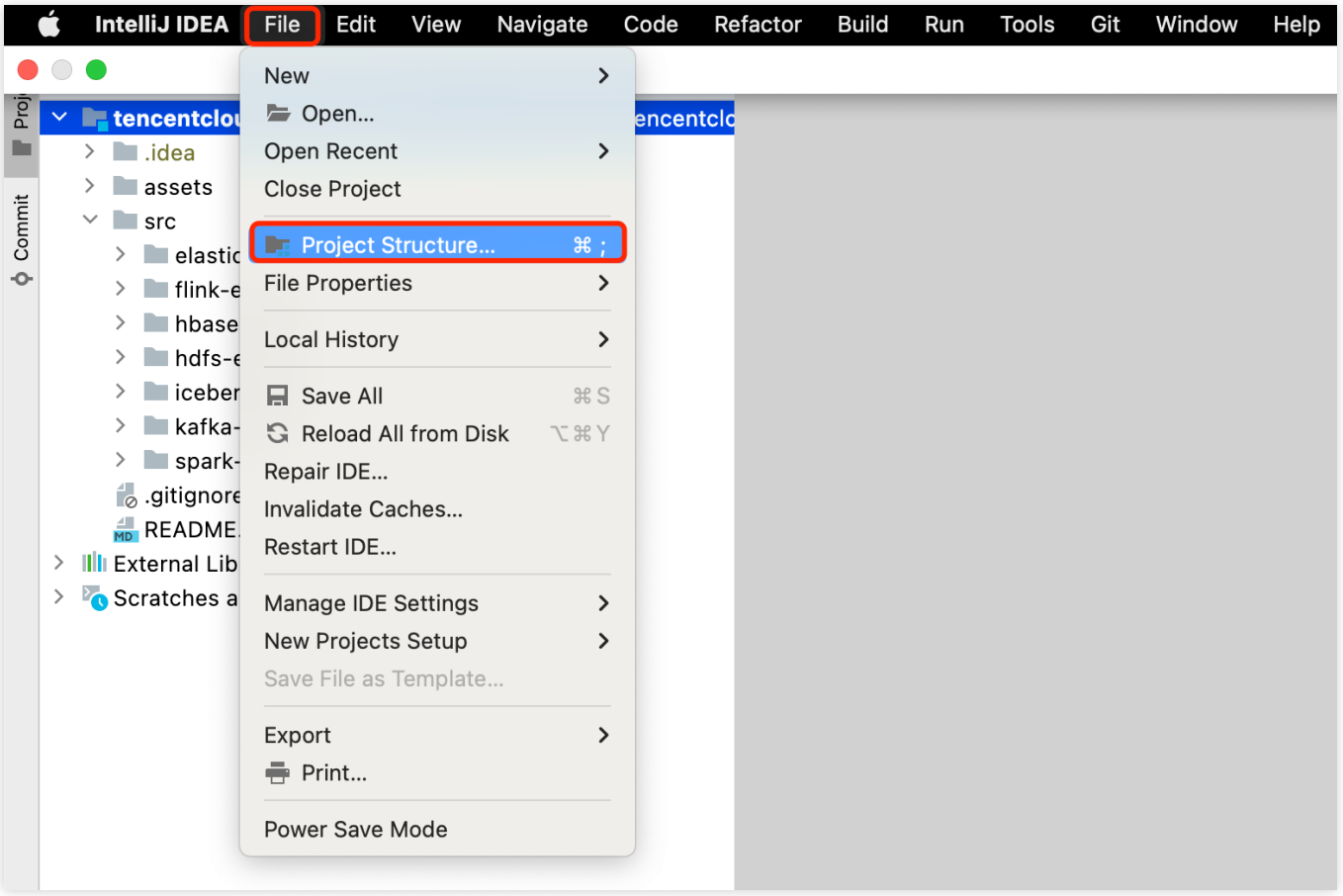
- 准备开发机（可选）：建议使用 Linux 操作系统
- 部署客户端：在开发机上执行客户端部署

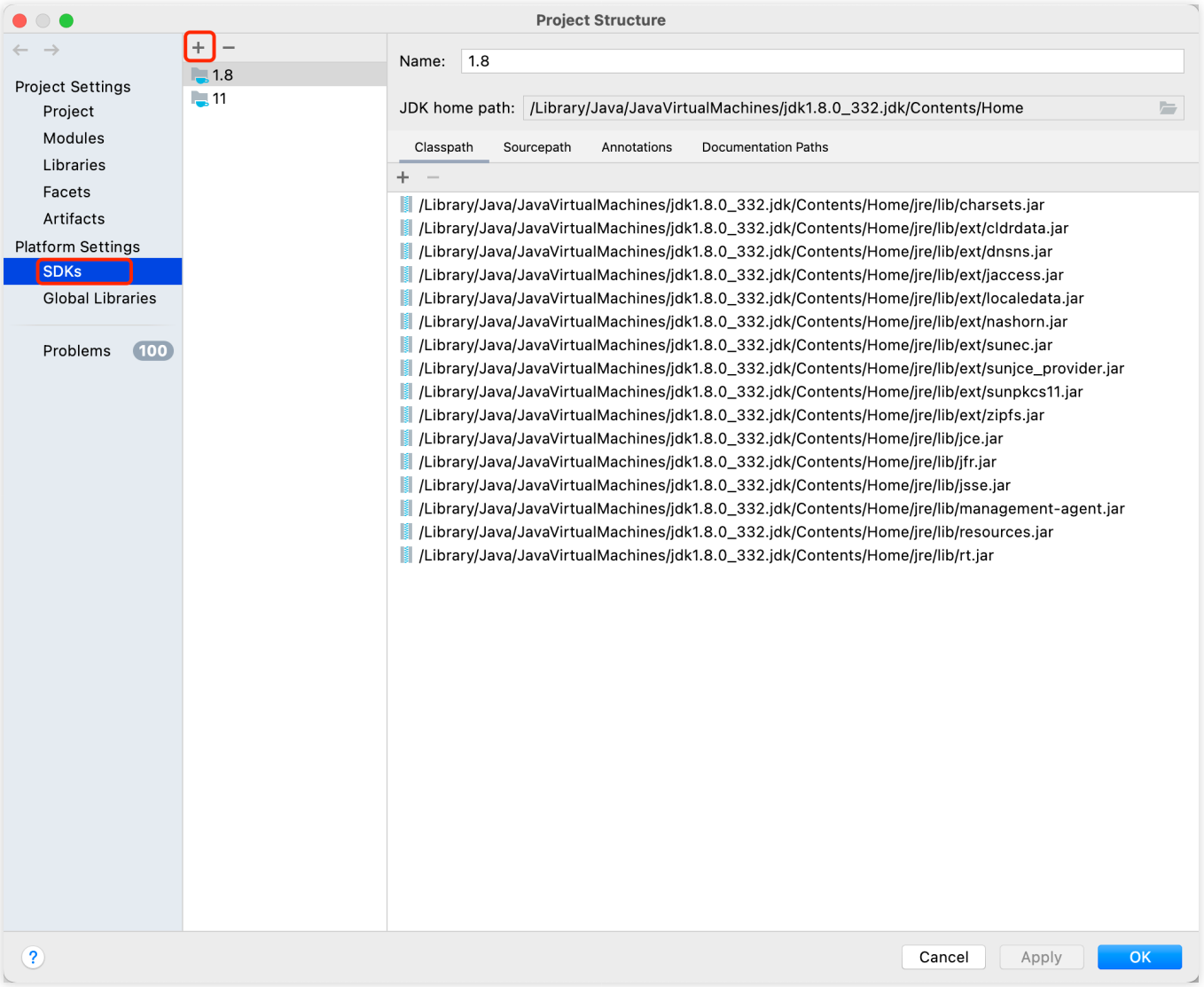
3. 导入示例工程代码
   以 IntelliJ IDEA 为例，将示例工程代码导入进行说明。

- 下载样例代码，选择对应迭代分支：https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master
- 导入项目：安装完 IntelliJ IDEA 和 JDK 工具后，导入样例工程到 IntelliJ IDEA 开发环境。点击 Open 后，选择上一步下载的项目地址打开

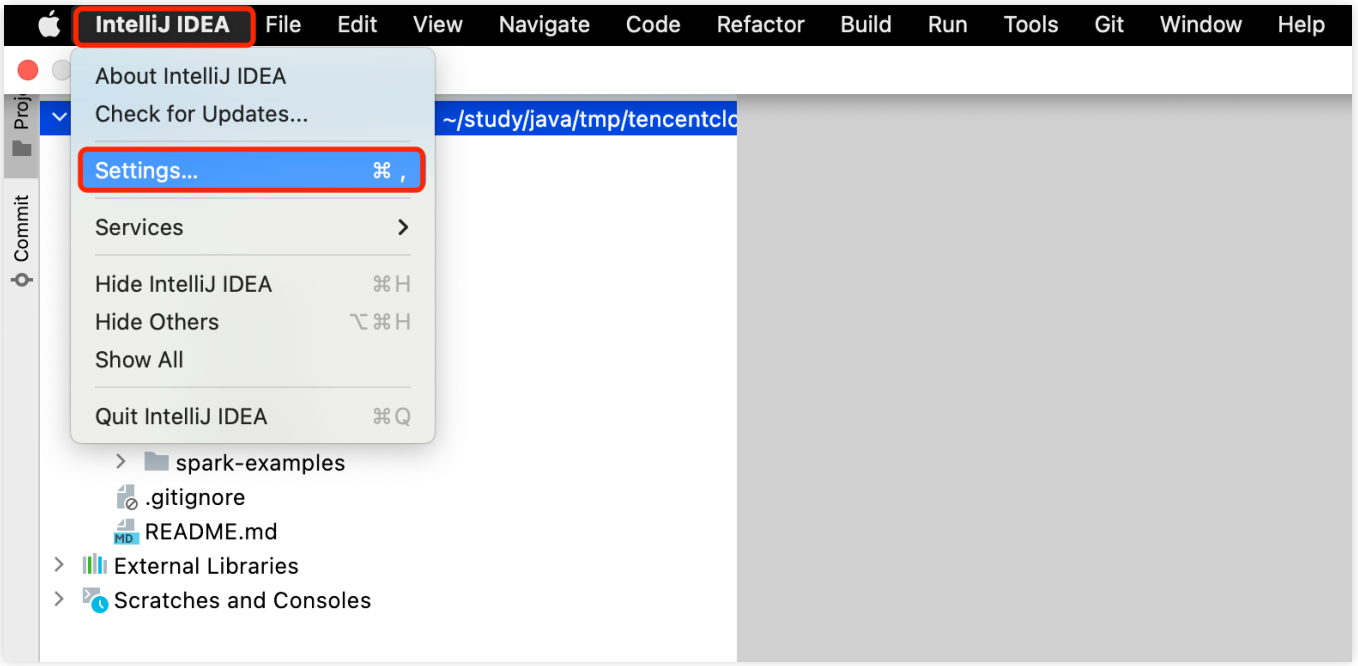


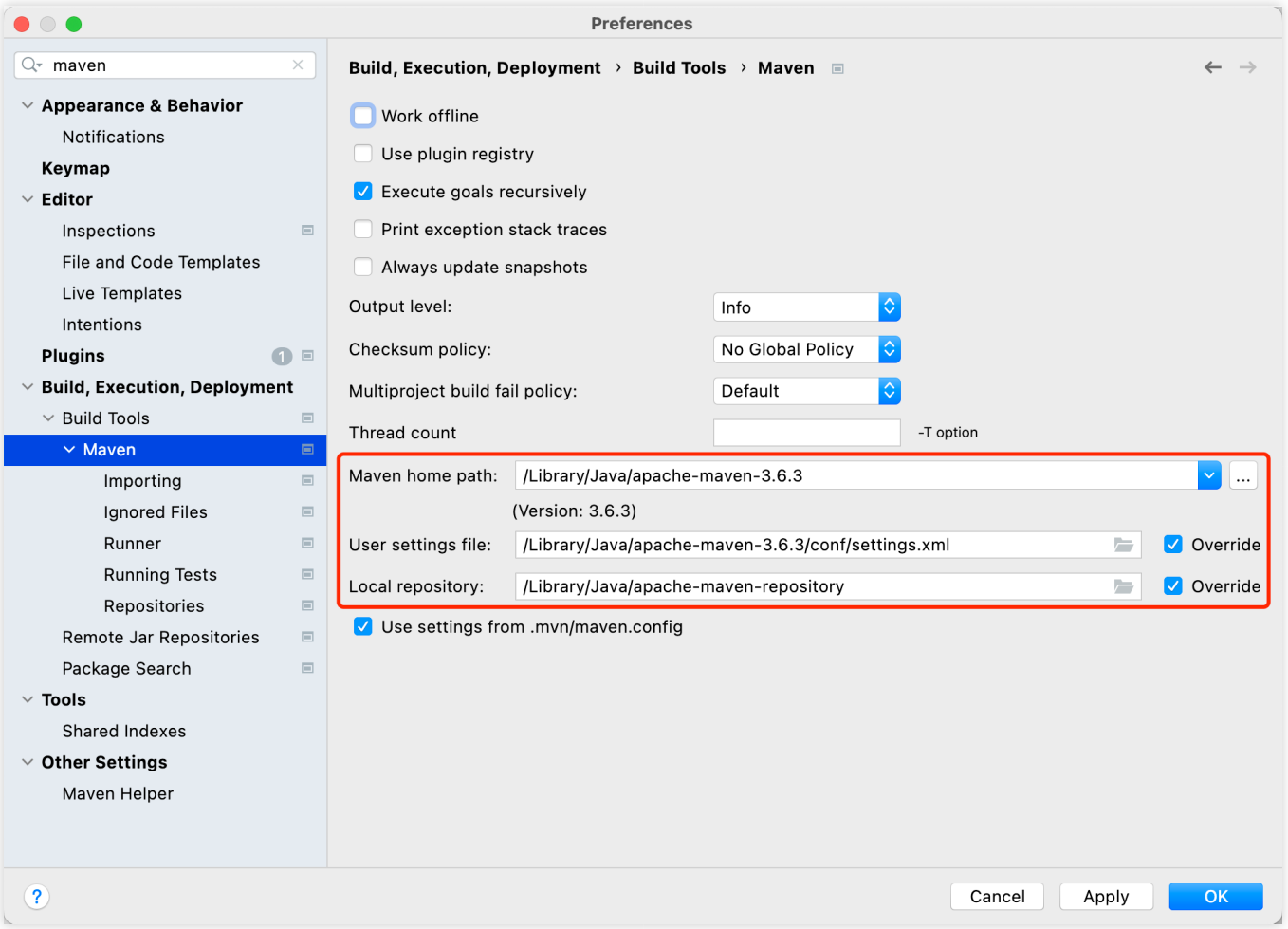
- IDEA 配置 JDK：首先确保在本地安装了 JDK 1.8，并配置好了环境变量。IDEA 选择 File 下的 Project Structure，点击 SDKs，选择 JDK 1.8，点击 Apply，再点击 OK。若没有 JDK 1.8，则点击 + 号进行添加，点击 Add JDK 后选择 JDK 1.8 安装目录，然后点击 OK 即可。


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
New
Open...
Open Recent
Close Project
Project Structure... ⌘ ;
File Properties
Local History
Save All ⌘S
Reload All from Disk ⌥⌘Y
Repair IDE...
Invalidate Caches...
Restart IDE...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
Commit
.idea
assets
src
.gitignore
README
External Lib
Scratches a



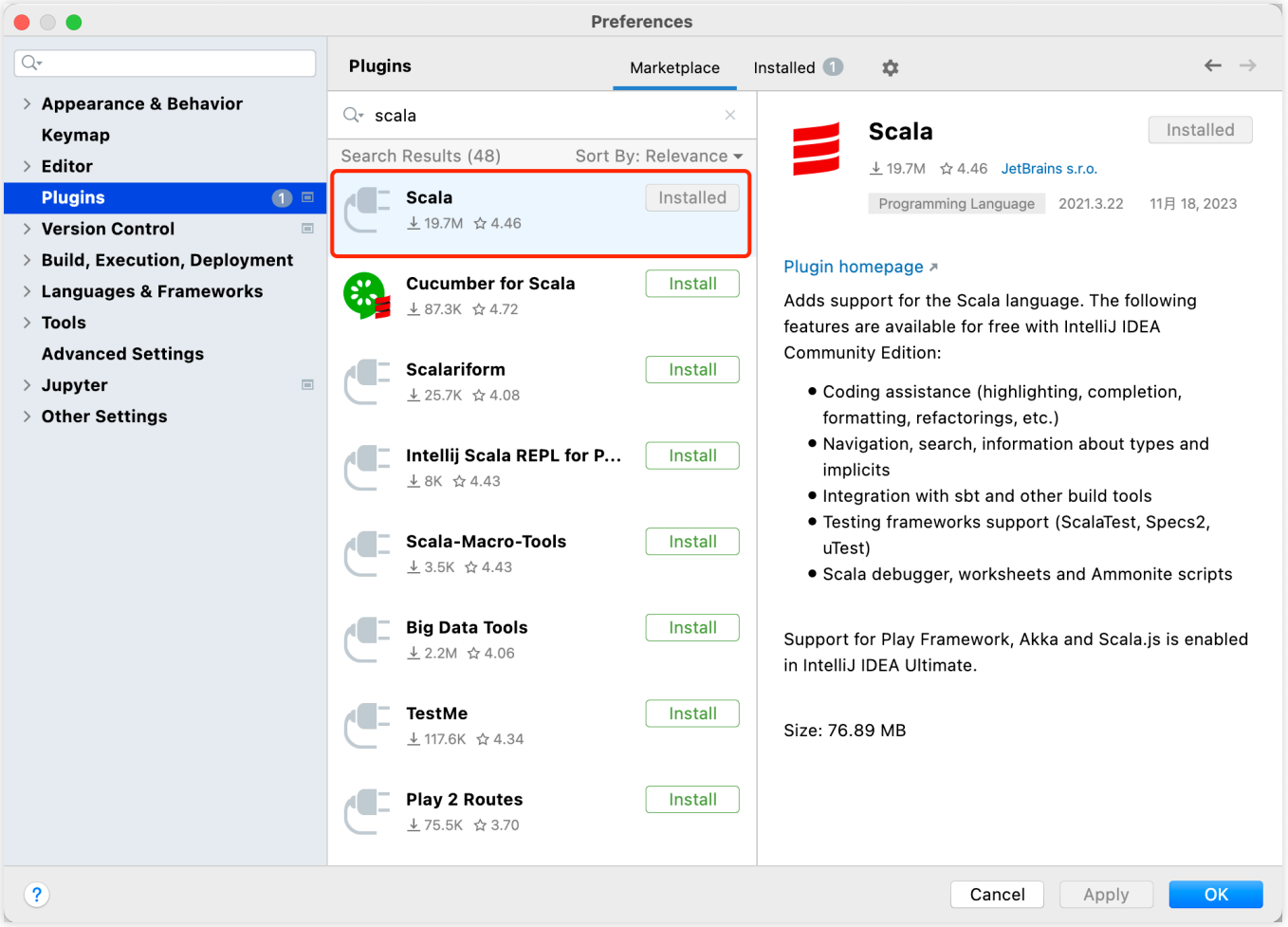
- IDEA 配置 Maven：首先确保在本地安装了 Maven，并配置好了环境变量和 settings.xml 文件。IDEA 点击 Settings 进入配置页面，左上角输入 maven 进行搜索，点击 Build Tools 下的 Maven 配置项，修改 “Maven home path” 为本地 Maven 的安装目录，修改 “User settings file” 为本地 Maven settings.xml 配置文件的文件路径，并勾选 Override，此时 “Local repository” 将自动设置为 settings.xml 文件中配置的本地 Maven 仓库的目录。最后点击 Apply，再点击 OK。

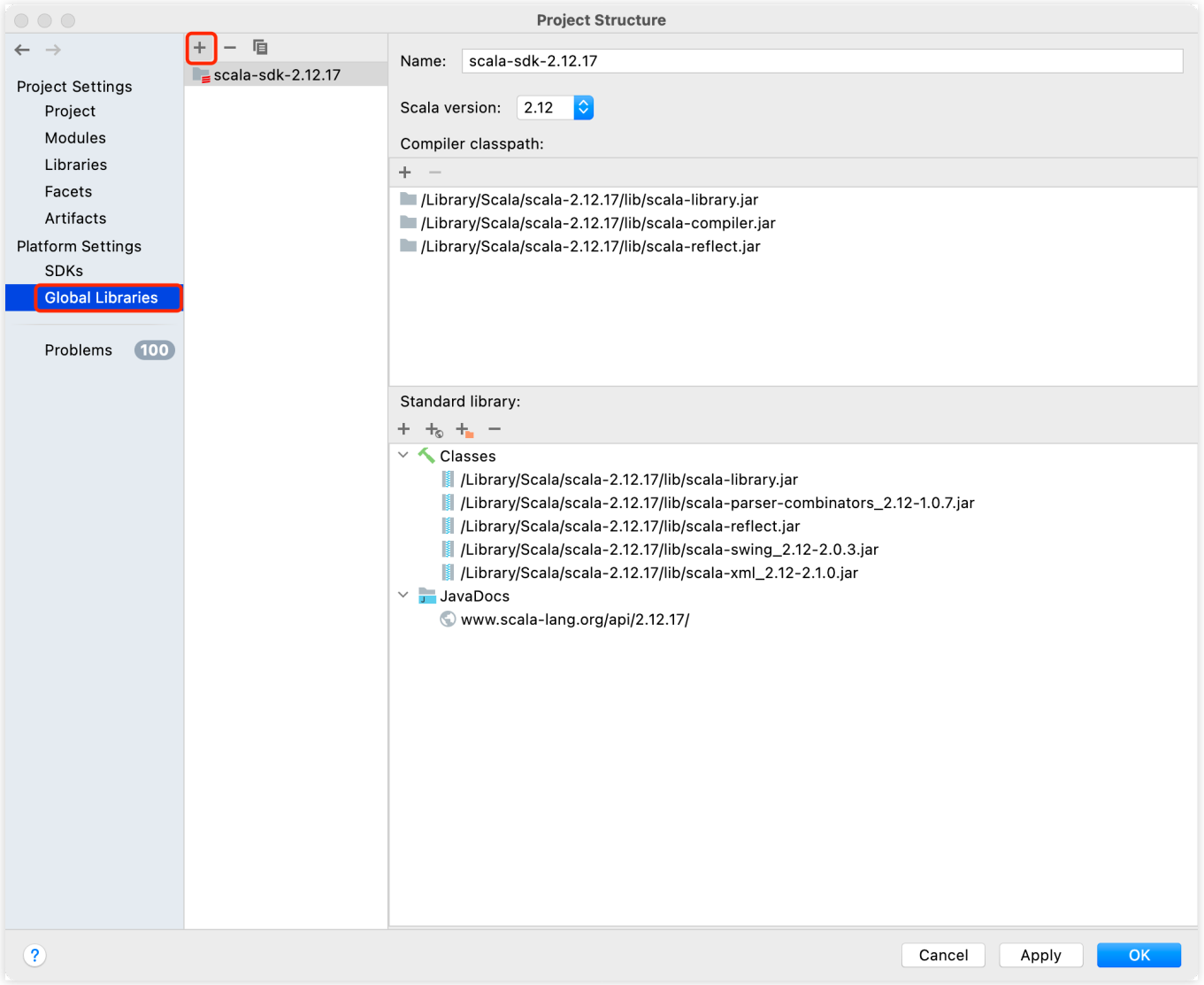




- IDEA 配置 Scala（可选）：首先确保在本地安装了 Scala，并配置好了环境变量。同样地，IDEA 点击 Settings 进入配置页面，点击 Plugins，右侧窗口选择 Marketplace，搜索 scala 插件并安装，安装后重启。重启后，继续选择 File 下的 Project Structure，点击 "Global Libraries"，点击 + 号添加 Scala SDK，选择本地 Scala 2.12 安装目录，

点击 Apply，再点击 OK 即可。





代码调试

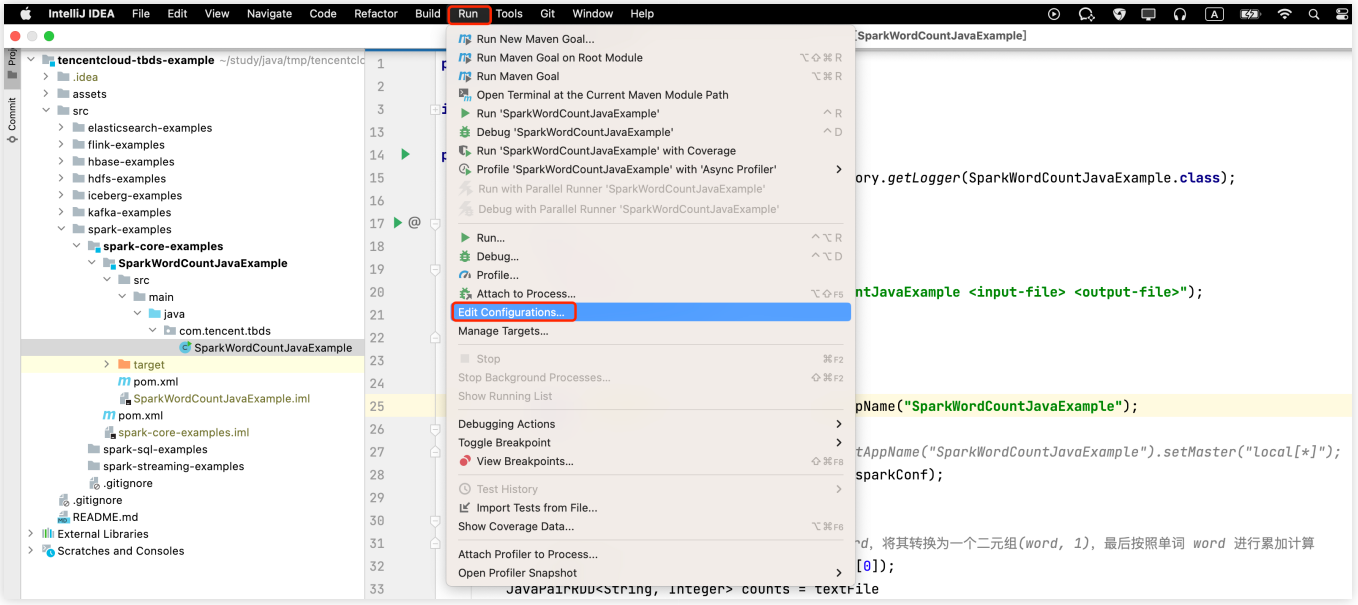
以下使用 IntelliJ IDEA 说明示例工程代码调试过程。

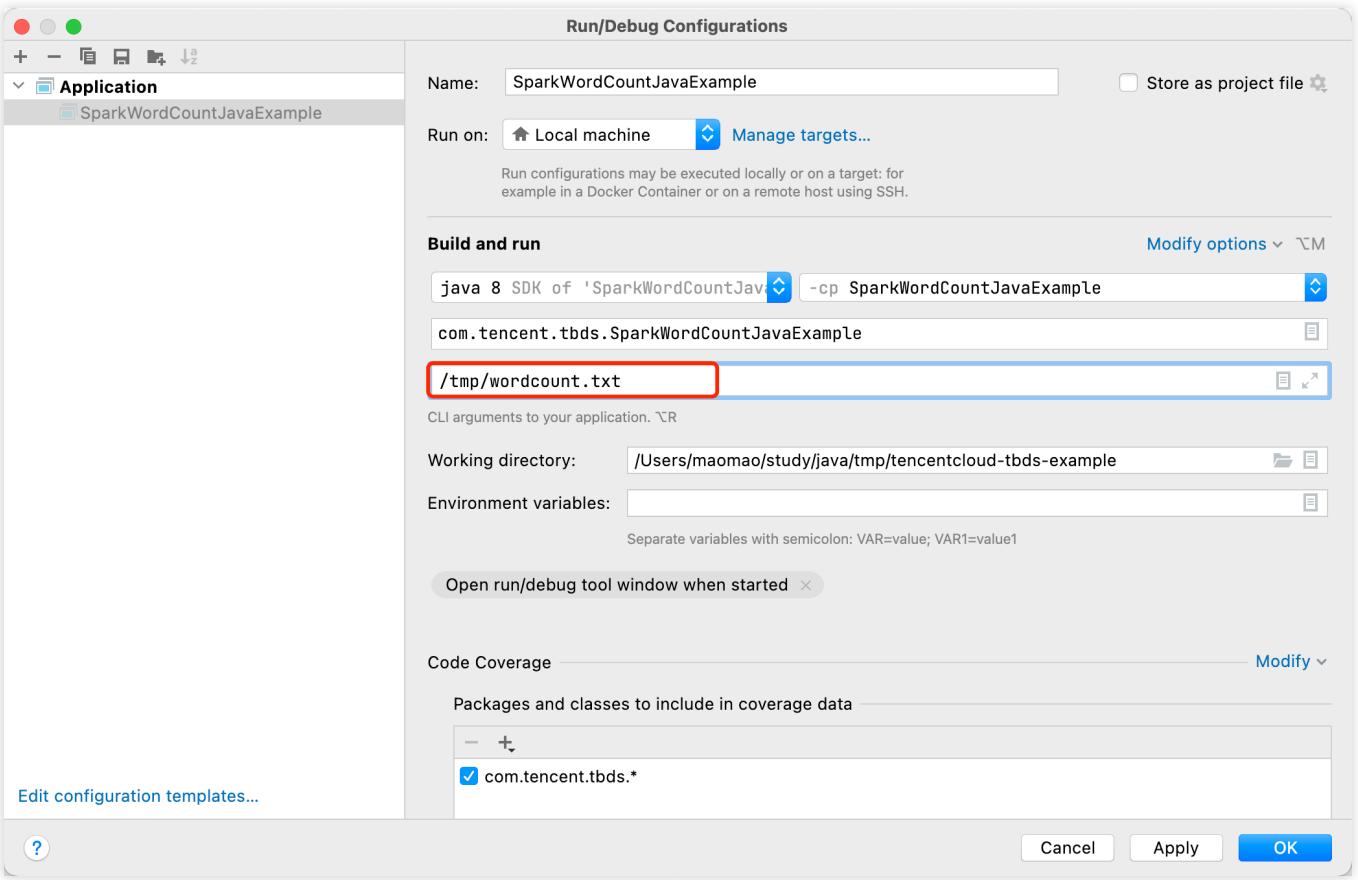
1. 编译和运行代码

- 先在本地准备一份需要单词统计的文件，文件内容如下。假设文件名为 wordcount.txt，文件绝对路径为 /tmp/wordcount.txt。

```
hello world
hello spark
hello hadoop
scala java
java kyuubi
```

- 选择 Run下面的 Edit Configurations，若不存在 Application，则点击“+”号新建，内容如下图所示，并在 Program arguments 一列中输入参数，多个参数使用空格分隔，这里的参数表示要进行单词统计的本地文件路径，以及单词统计最终结果的保存目录（可选），这里填写的是 /tmp/wordcount.txt，然后点击 Apply。





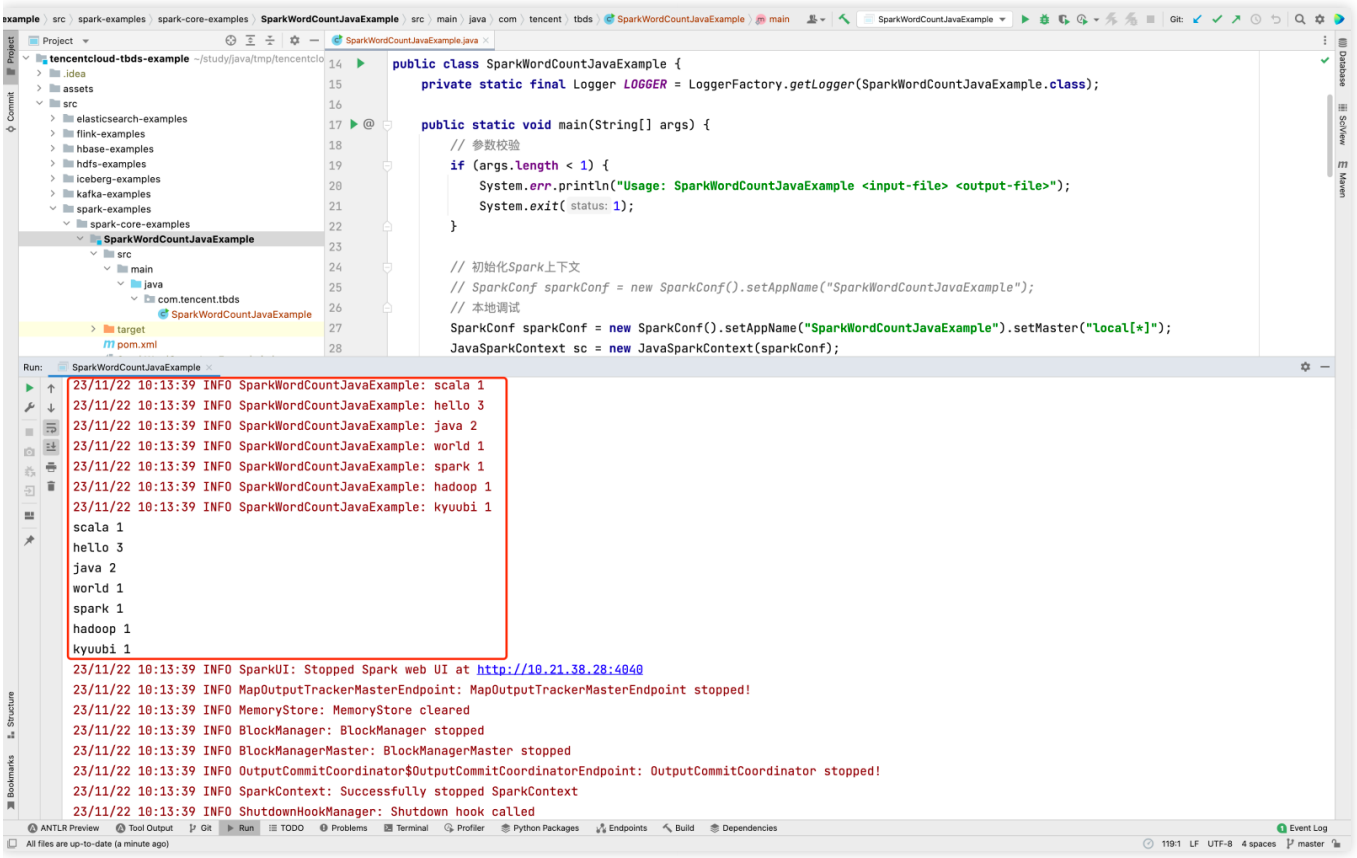
2. 本地调试

将示例代码中的本地调试注释打开，同时注释上一行代码，如下所示。local[*] 表示使用与逻辑核心数相同的工作线

程在本地运行 Spark，若在代码中未进行设置，则需要在提交任务时或配置文件中进行指定。这是由于参数具有优先级：直接在 SparkConf 中设置的属性优先，然后是传递给 spark-submit 或 spark-shell 的参数，最后是 spark-defaults.conf 配置文件中的选项参数。

```
// 初始化Spark上下文
// SparkConf sparkConf = new SparkConf().setAppName("SparkWordCountJavaExample");

// 本地调试
SparkConf sparkConf = new SparkConf().setAppName("SparkWordCountJavaExample").setMaster("local[*]");
```

IDEA 点击 Run 即可在本地调试运行代码，调试结果如下图所示。在代码本地运行过程中，还可以查看 Spark web UI 监控页面，即图中日志输出的蓝色字体 URL，一旦代码运行结束，该 URL 将失效。



3. 开发机调试

确保代码中有必要的日志输出，开发机调试参考下一节【打包发布】。 注意，开发机调试和本地调试使用场景略有不同，总的来说，开发机调试更解决实际的应用场景。比如，这里本地调试没有 HDFS 配置文件，无法访问 HDFS，因此读取的是本地文件；而开发机上可以直接访问 HDFS，读取的是 HDFS 文件。

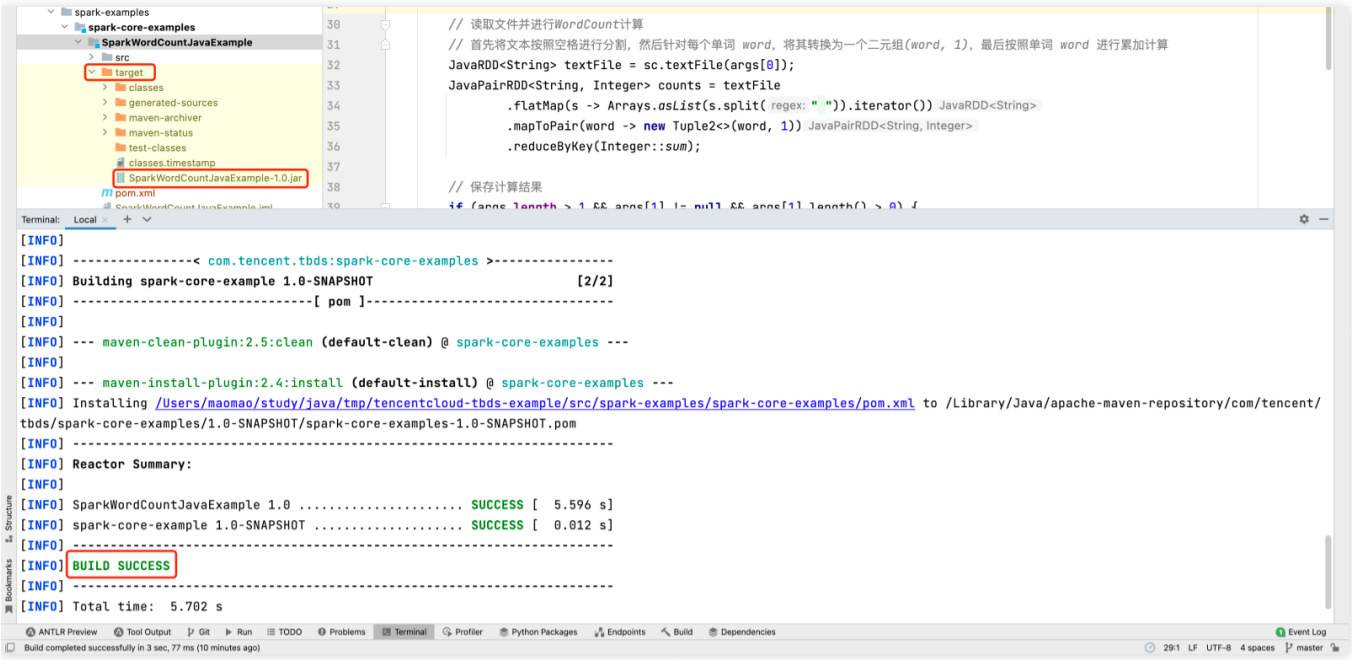
打包发布

1. 打包

以下使用 IntelliJ IDEA 说明示例工程代码编译过程。点击 IDEA 下方 Terminal 打开终端，切换到示例工程的 Spark 工程目录下，然后使用命令 mvn clean install 对工程进行打包，运行过程中可能还需要下载一些文件，直到出现 build success 表示打包成功。

```
cd src/spark-examples/spark-core-examples/
mvn clean install
```

通过上述编译打包后，将在工程目录下 target 文件夹中看到打好的 jar 包，如图示中的 SparkWordCountJavaExample-1.0.jar。注意，这里没有将项目依赖的第三方 JAR 一起打包，若开发机上不存在项目中依赖的第三方 JAR，读者可参考网上教程将项目打包成 FatJar，或将依赖的第三方 JAR 上传至开发机，然后在使用 spark-submit 提交任务时，通过 --jars 参数进行指定，参数说明参考【常用命令】。



2. 开发机运行

- 准备用户：获取用户认证信息。若是 Kerberos 环境，需要将用户的 keytab 文件下载到本地，然后上传至开发机。这里将 test 用户的 test.keytab 文件上传至开发机的 /tmp 目录。
- 准备文件：准备一份名为 wordcount.txt 的单词统计文件，然后将该文件上传至 HDFS /tmp 目录。注意，如果是 kerberos 环境，需要首先进行认证。

```
# 1. Kerberos认证（Simple认证跳过该步）
[test@10 ~]$ klist -kt /tmp/test.keytab
Keytab name: FILE:/tmp/test.keytab
KVNO Timestamp           Principal
---- ------------------- ------------------------------------------------------
   1 11/22/2023 17:00:59 test@TBDS-CURPL8E5
   1 11/22/2023 17:00:59 test@TBDS-CURPL8E5
```

```
[test@10 ~]$ kinit -kt /tmp/test.keytab test@TBDS-CURPL8E5

# 2. 准备单词统计文件，内容如下，编辑后保存退出
[test@10 ~]$ vim /tmp/wordcount.txt
hello world
hello spark
hello hadoop
scala java
java kyuubi

# 3.上传单词统计文件至HDFS
[test@10 ~]$ /usr/local/service/hadoop/bin/hdfs dfs -put /tmp/wordcount.txt /tmp
```

- Ranger 授权：确保 test 用户具有提交 yarn default 队列的权限。

- 安全认证：Spark Kerberos 环境认证方式有三种，如下表所示。

认证方式	认证说明
命令认证	在提交 Spark 任务前，使用 kinit -kt {keytab} {principal} 命令进行认证
配置认证	以下三种方式任选其一，推荐方式一和方式二 i. 在 spark-submit 命令提交任务时，通过 --conf spark.kerberos.keytab={keytab} --conf spark.kerberos.principal={principal} 参数指定 ii. 在 spark-submit 命令提交任务时，通过 --keytab {keytab} --principal {principal} 参数指定 iii. 在 spark-defaults.conf 配置文件中，配置 spark.kerberos.keytab 和 spark.kerberos.principal 参数
代码认证	获取用户的 keytab 和 principal 后，在应用程序的代码中进行认证

- 运行样例：将该 jar 包上传到开发机 /tmp 目录，进入 Spark 安装目录，一般位于 /usr/local/service/spark，然后使用 spark-submit 命令提交，最后的 HDFS 路径参数表示需要进行单词统计的文件的绝对路径。若项目依赖第三方 JAR，可以通过 --jars 参数进行指定，更多参数说明参考【常用命令】。注意，这里演示的是通过“配置认证”的方式提交 Spark 任务，读者可根据自身情况自行选择认证方式。

```
[test@10 ~]$ cd /usr/local/service/spark
[test@10 spark]$ bin/spark-submit \
--master yarn \
--deploy-mode client \
--queue default \
--keytab /tmp/test.keytab \
--principal test@TBDS-CURPL8E5 \
--class com.tencent.tbds.spark.SparkWordCountJavaExample \
```

/tmp/SparkWordCountJavaExample-1.0.jar hdfs:///tmp/wordcount.txt

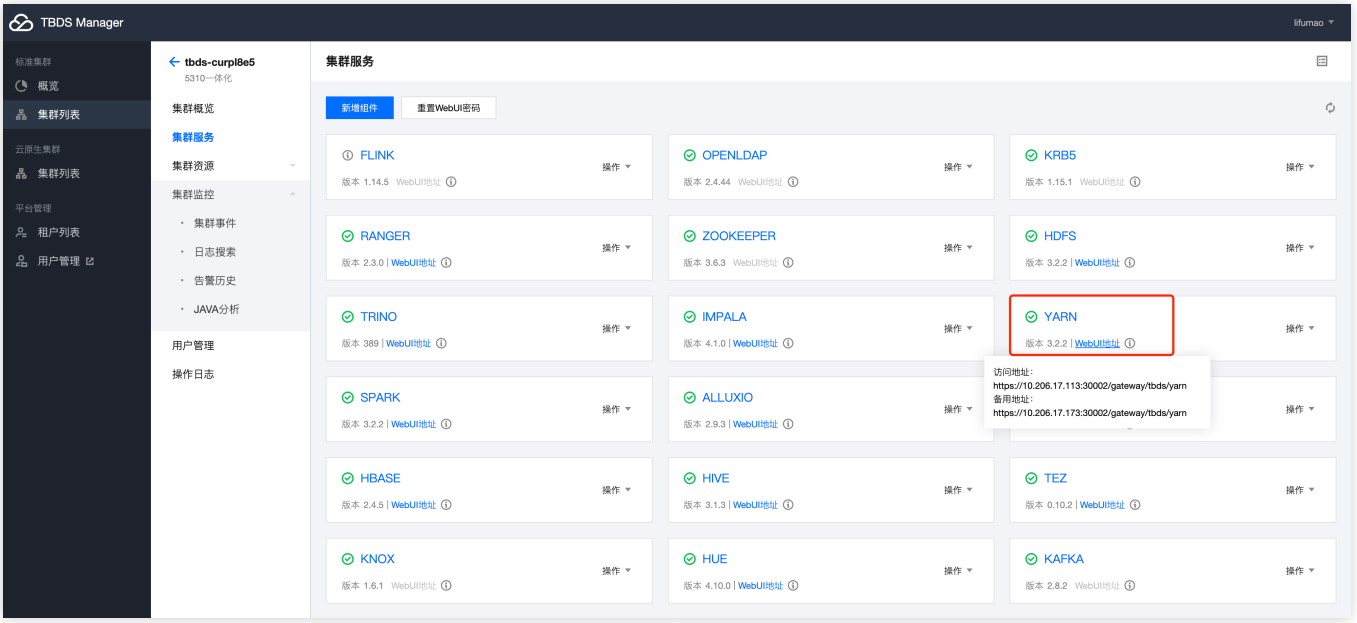
- 当以 yarn client 模式提交，运行过程中的日志和结果将直接输出至控制台，如下图所示。由于日志级别为 WARN，因此这里没有输出日志。如果是在集群外进行作业提交，两个集群间没有做 hostname 通信的话需要指定 Driver 实际的 IP 地址，例如：--conf spark.driver.host=xx.xx.xx.xx。

```
[test@10 spark]$ bin/spark-submit \
> --master yarn \
> --deploy-mode client \
> --queue default \
> --keytab /tmp/test.keytab \
> --principal test@TBDS-CURPL8E5 \
> --class com.tencent.tbds.SparkWordCountJavaExample \
> /tmp/SparkWordCountJavaExample-1.0.jar hdfs:///tmp/wordcount.txt
world 1
java 2
kyuubi 1
hadoop 1
scala 1
hello 3
spark 1
[test@10 spark]$
```

程序运维监控

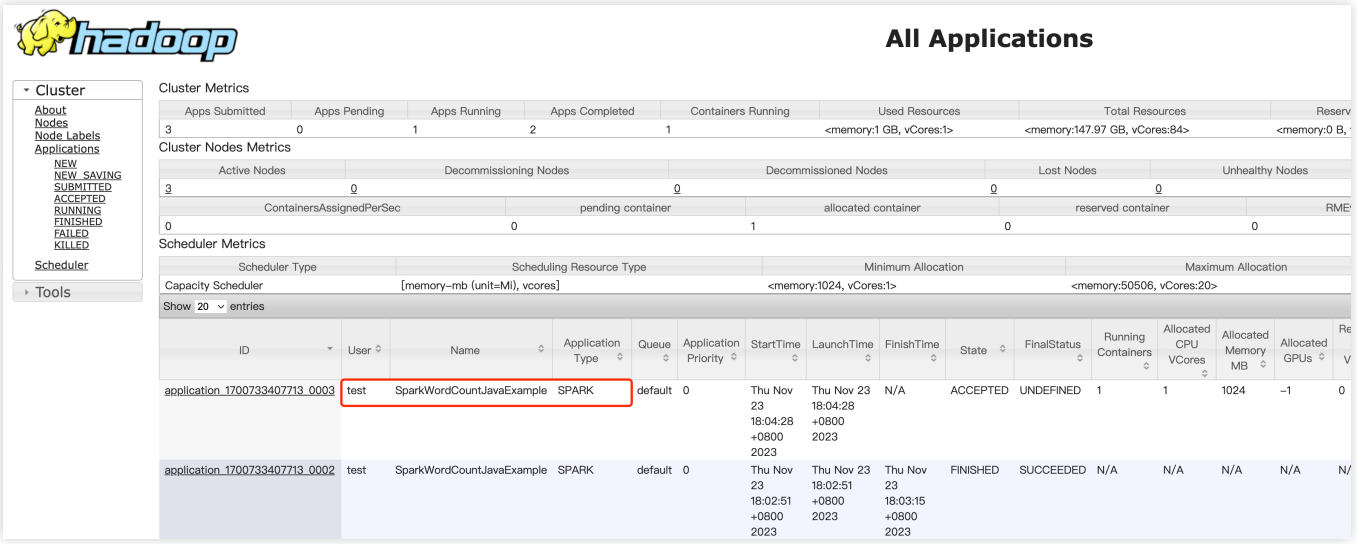
1. 监控

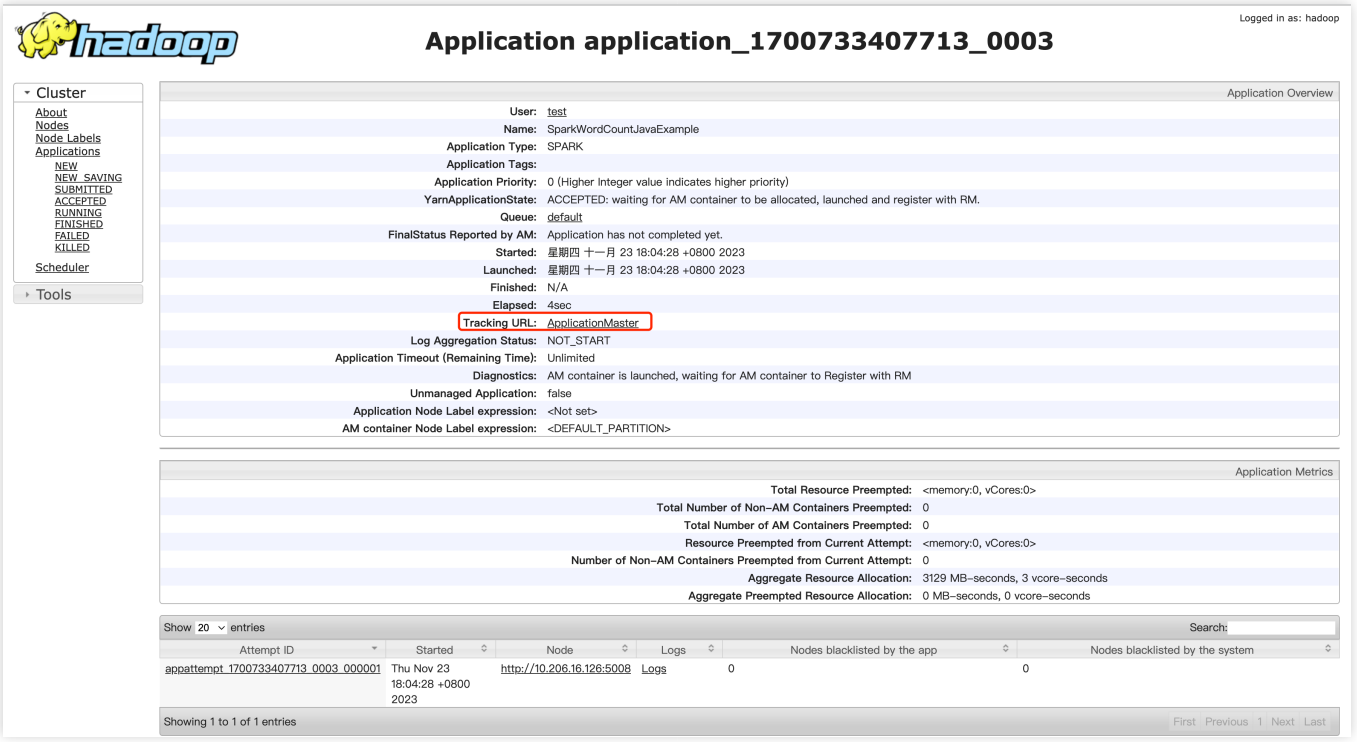
- 登录 TBDS Manager 管控平台，点击“集群列表”，选择 Spark 程序运行所对应的集群。点击“集群服务”，选择 YARN 服务，点击 WebUI 地址跳转至 YARN UI 界面。



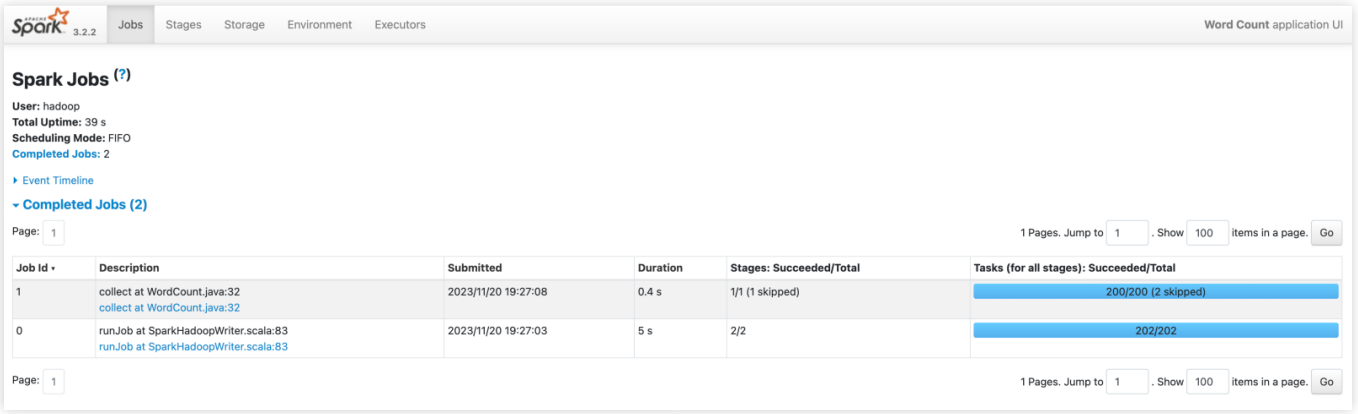
- YARN UI 页面根据 ApplicationID、User、Name 等信息，找到对应的 Spark 任务。Spark 任务的 ApplicationID

在日志级别为 INFO 及以下会输出至控制台，也可以根据任务名 SparkWordCountJavaExample 等信息进行查找对应的 Spark 任务，点击 applicationID 链接，然后点击 Tracking URL 链接，跳转 Spark 作业监控页面。



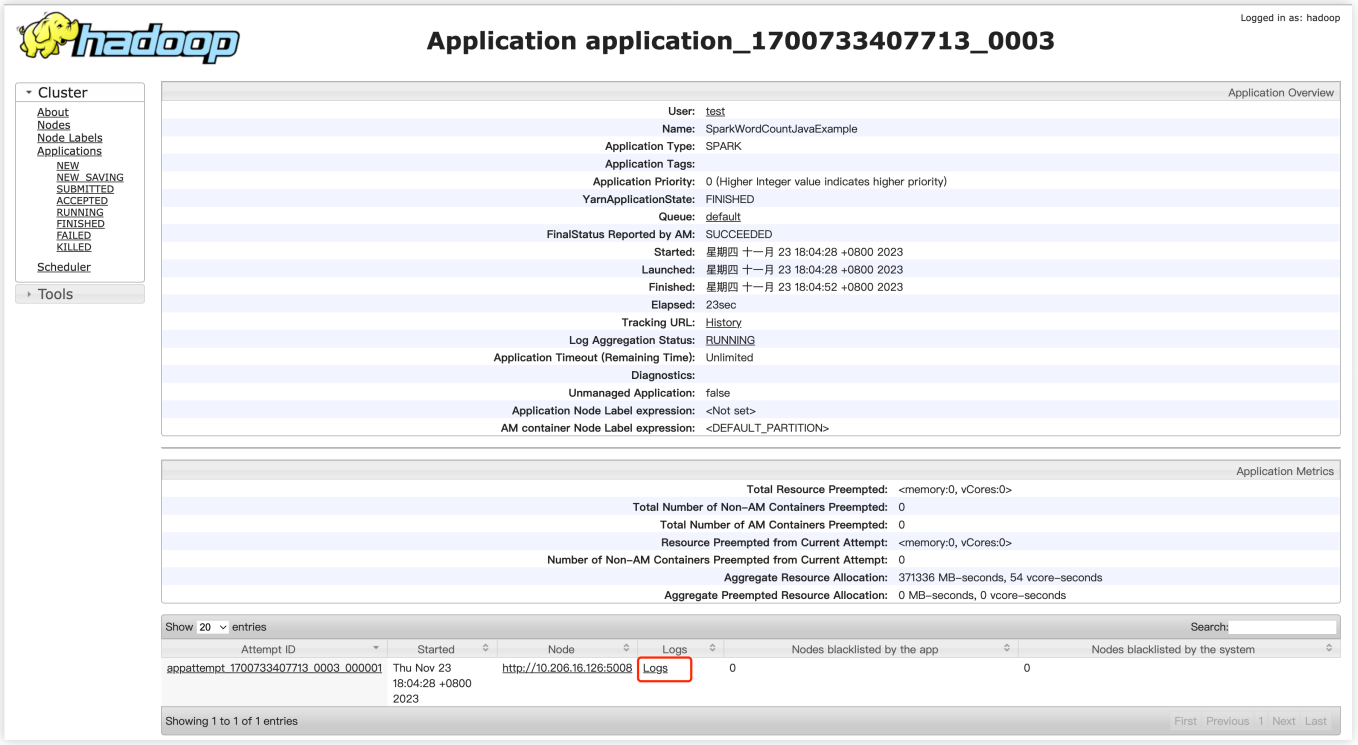


- Spark 监控页面大致如下，更多介绍详见 Spark 官网。



2. 日志查看

若任务以 yarn client 模式提交，则日志将直接输出至控制台。若任务以 yarn cluster 模式提交，则需要到 YARN UI 页面根据 ApplicationID、Name 等信息，找到对应的 Spark 任务，点击 Logs 查看日志。



3. 作业操作

与其它提交至 YARN 的任务类似，可以通过 YARN 相关命令查看、停止任务。例如，通过命令 yarn application -kill 停止 Spark 应用；通过命令 yarn logs -applicationId 查看 Spark 任务日志（Cluster 模式）。

示例说明

本教程演示的是：提交的任务为 WordCount 任务，即统计单词个数，需要提前在集群中准备需要统计的文件，可以是一个 HDFS 文件，或 Yarn 集群所有节点上都存在的本地文件。更多说明请查看示例工程 README.md。

```
// 初始化Spark上下文
SparkConf sparkConf = new SparkConf().setAppName("SparkWordCountJavaExample");
JavaSparkContext sc = new JavaSparkContext(sparkConf);

// 读取文件并进行WordCount计算
// 首先将文本按照空格进行分割，然后针对每个单词 word，将其转换为一个二元组(word, 1)，最后按照单词 word 进行累加计算
JavaRDD<String> textFile = sc.textFile(args[0]);
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey(Integer::sum);

// 保存计算结果
if (args.length > 1 && args[1] != null && args[1].length() > 0) {
    counts.saveAsTextFile(args[1]);
}

// 收集计算结果，打印日志并输出控制台
List<Tuple2<String, Integer>> output = counts.collect();
    for (Tuple2<?, ?> tuple : output) {
        System.out.println(tuple._1 + " " + tuple._2);
        LOGGER.info(tuple._1 + " " + tuple._2);
}

// 关闭Spark上下文
sc.close();
```

api接口

TBDS 大数据平台上的 SPARK 访问接口与开源兼容，可分别参考 Scala API、Java API、Python API。

常用命令

<table>
<tr><th>操作</th><th>命令</th><th>描述</th></tr>
<tr><td>提交任务</td><td>bin/spark-submit \
--class <main-class> \
--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key>=<value> \
... # other options
<application-jar> \
[application-arguments]</td><td>• --class：应用程序的入口（如：org.apache.spark.examples.SparkPi）。
• --master：集群的master URL（如：spark://23.195.26.187:7077）。
• --deploy-mode：cluster表示driver位于集群内部，client表示driver位于本地（默认 client）。
• --conf：key=value格式的任意Spark配置属性。对于包含空格的值，用引号将 "key=value"包起来。多个配置应作为单独参数传递。(如：--conf <key>=<value> --conf <key2>=<value2>）
• application-jar：包含应用程序和所有依赖项的捆绑 jar 路径。该URL必须在集群内全局可见，例如，一个 hdfs:// 路径或所有节点上都有的 file:// 路径。对于Python应用程序，只需用 .py 文件代替 application-jar，并用 --py-files 将 Python .zip、.egg 或 .py 文件添加到搜索路径。
• application-arguments：传递给主类main方法的参数（如果有的话）</td></tr>
<tr><td rowspan="2">交互式命令</td><td>bin/spark-shell [options]</td><td rowspan="2">options 说明如下：
• --master MASTER_URL：spark://host:port、mesos://host:port、yarn、k8s://host:port或local（默认值：local[*]）。
• --deploy-mode DEPLOY_MODE：是否在本地启动driver程序（"client"）或在集群内的一个工作节点上启动（"cluster"）（默认值：client）
• --class CLASS_NAME：应用程序的主类（适用于Java/Scala应用程序）。
• --jars JARS：要包含在driver和executor类路径中的以逗号分隔的jar文件列表。
• --packages：要包含在driver和executor类路径中的以逗号分隔的jar文件的Maven坐标列表。将搜索本地Maven仓库，然后是Maven中央仓库和通过--repositories指定的任何其他远程仓库。坐标的格式应为groupId:artifactId:version。
• --exclude-packages：要在解析--packages中提供的依赖项时排除的以逗号分隔的groupId:artifactId列表，以避免依赖冲突。
• --repositories：要搜索与--packages给定的Maven坐标对应的其他远程仓库的以逗号分隔的列表。
• --py-files PY_FILES：要放置在PYTHONPATH上的以逗号分隔的.zip、.egg或.py文件列表（适用于Python应用程序）。
• --files FILES：要放置在每个executor工作目录中的以逗号分隔的文件列表。可以通过SparkFiles.get(fileName)在执行程序中访问这些文件的文件路径。
• --archives ARCHIVES：要解压缩到每个executor工作目录中的以逗号分隔的存档文件列表。</td></tr>
<tr><td>bin/pyspark [options]</td></tr>
</table>

		• --conf, -c PROP=VALUE：任意的Spark配置属性。 • --properties-file FILE：加载额外属性的文件路径。如果未指定，则会查找 conf/spark-defaults.conf。 • --driver-memory MEM：driver的内存。 • --driver-java-options：传递给driver的额外Java选项。 • --driver-library-path：传递给driver的额外库路径条目。 • --driver-classpath：传递给driver的额外类路径条目。注意，使用--jars添加的 jar 文件会自动包含在类路径中。 • --executor-memory MEM：每个executor的内存。 • --proxy-user NAME：提交应用程序时要模拟的用户。此参数与--principal / --keytab不兼容。
	bin/spark-sql [options] [cli options]	cli options 说明如下： • -d,--define <key=value>：对 Hive 命令应用的变量替换。如：-d A=B 或 --define A=B。 • --database <databasename>：指定要使用的数据库。 • -e <quoted-query-string>：命令行中的 SQL 查询语句。 • -f <filename>：来自文件的 SQL 查询语句。 • --hiveconf <property=value>：使用给定属性的值。 • --hivevar <key=value>：对 Hive 命令应用的变量替换。如：--hivevar A=B。 • -i <filename>：初始化 SQL 文件。 • -S,--silent：交互式 shell 中的静默模式。 • -v,--verbose：详细模式（将执行的 SQL 输出到控制台）。

权限策略配置

配置步骤

1. 安装 Ranger Plugin：

   i. 下载并安装 Ranger Spark插件。

2. 配置 Spark：

   i. 修改 spark-site.xml 文件，添加以下配置：

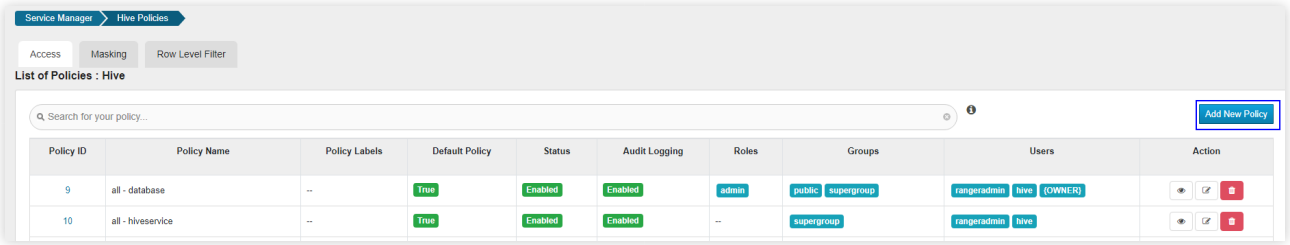
```
spark.sql.extensions=org.apache.ranger.authorization.spark.authorizer.RangerSparkSQLExtension
spark.sql.hive.metastore.sharedPrefixes=org.apache.ranger
```

3. 配置 Ranger：

   i. 使用Ranger管理员用户rangeradmin登录Ranger管理页面。

   ii. 在首页中单击“HADOOP SQL”区域的组件插件名称，例如“hive”。

   iii. 在“Access”页签单击“Add New Policy”，添加Spark权限控制策略。



   iv. 根据业务需求配置相关参数。

   Spark权限参数如下：

参数名称	描述
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。
Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持“*”通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。
Policy Label	为当前策略指定一个标签，可以根据这些标签搜索报告和筛选策略。

database	将适用该策略的Spark数据库名称。 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
table	将适用于该策略的Spark表名称。 如果需要添加基于UDF的策略，可切换为UDF，然后输入UDF的名称。 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
Hive Column	将适用于该策略的列名，填写*时表示所有列。 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
Description	策略描述信息。
Audit Logging	是否审计此策略。
Allow Conditions	策略允许条件，配置本策略内允许的权限及例外。 在“Select Role”、“Select Group”、“Select User”列选择已创建好的需要授予权限的Role、用户组或用户，单击“Add Conditions”，添加策略适用的IP地址范围，然后再单击“Add Permissions”，添加对应权限。 ▪ select：查询权限 ▪ update：更新权限 ▪ Create：创建权限 ▪ Drop：drop操作权限 ▪ Alter：alter操作权限 ▪ Index：索引操作权限 ▪ All：所有执行权限 ▪ Read：可读权限 ▪ Write：可写权限 ▪ Temporary UDF Admin：临时UDF管理权限 ▪ Select/Deselect All：全选/取消全选 如需添加多条权限控制规则，可单击 按钮添加。 如需当前条件中的用户或用户组管理本条策略，可勾选“Delegate Admin”，这些用户将成为受委托的管理员。被委托的管理员可以更新、删除本策略，它还可以基于原始策略创建子策略。
Deny Conditions	策略拒绝条件，配置本策略内拒绝的权限及例外，配置方法与“Allow Conditions”类似。

设置权限场景如下：

任务场景	角色授权操作

role admin操作	1. 在首页中单击“Settings”，选择“Roles>Add New Role”。 2. 单击Role Name为admin的角色，在“Users”区域，单击“Select User”，选择对应用户名。 3. 单击Add Users按钮，在对应用户名所在行勾选“Is Role Admin”，单击“Save”保存配置。 说明： Ranger页面的“Settings”选项只有rangeradmin用户有权限访问。用户绑定Hive管理员角色后，在每个维护操作会话中，还需要执行以下操作： 1. 以客户端安装用户，登录安装Hive客户端的节点。 2. 执行以下命令配置环境变量。 例如，Spark客户端安装目录为“/opt/client”，执行source/opt/client/bigdata_env。 1. 执行以下命令认证用户。 kinit Spark业务用户 1. 执行以下命令登录客户端工具。 spark-beeline 1. 执行以下命令更新用户的管理员权限。 set role admin;
创建库表操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写并选择对应的数据库（如果是创建库，需填写将要创建的库名称，或填写“*”表示任意名称的数据库，然后选择所写名称），在“table”与“column”右侧填写并选择对应的表名称、列名称，均支持通配符（“*”）匹配。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Create”。
删除库表操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写并选择对应的数据库（如果是删除库，需填写将要创建的库名称，或填写“*”表示任意名称的数据库，然后选择所写名称），在“table”与“column”右侧填写并选择对应的表名称、列名称，均支持通配符（“*”）匹配。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Drop”。 说明： 对于CarbonData表，只有对应表的OWNER，才能执行“drop”操作。
Alter操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写并选择对应的数据库，在“table”右侧填写并选择对应的表，在“column”右侧填写并选择对应的列名称，支持通配符（“*”）匹配。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Alter”。

LOAD操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写并选择对应的数据库，在“table”右侧填写并选择对应的表，在“column”右侧填写并选择对应的列名称，支持通配符（“*”）匹配。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“update”。
INSERT操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写并选择对应的数据库，在“table”右侧填写并选择对应的表，在“column”右侧填写并选择对应的列名称，支持通配符（“*”）匹配。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“update”。 5. 用户还需要具有Yarn任务队列的“submit-app”权限，默认情况下，hadoop用户组具有向所有Yarn任务队列“submit-app”权限。
GRANT操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写并选择对应的数据库，在“table”右侧填写并选择对应的表，在“column”右侧填写并选择对应的列名称，支持通配符（“*”）匹配。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 勾选“Delegate Admin”。
ADD JAR操作	1. 在“Policy Name”填写策略名称。 2. 单击“database”并在下拉菜单中选择“global”。在“global”右侧填写或选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Temporary UDF Admin”。
VIEW与INDEX权限	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写并选择对应的数据库，在“table”右侧填写并选择对应的VIEW或INDEX名称，在“column”右侧填写并选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，参照表格上述相关操作，根据需求，给用户勾选相应权限。
其他用户库表操作	1. 参照表格上述相关操作添加对应权限。 2. 给用户添加其他用户库表的HDFS路径的读、写、执行权限。

> **说明：**
>
> 说明：在Ranger上为用户添加Spark SQL的访问策略后，需要在HDFS的访问策略中添加相应的路径访问策略，否则无法访问数据文件。
>

- Ranger策略中global策略仅用于联合Temporary UDF Admin权限，用来控制UDF包的上传。
- 通过Ranger对Spark SQL进行权限控制时，不支持empower语法。
- Ranger策略不支持本地路径以及HDFS上带空格的路径。
- 开启Ranger鉴权后，对视图操作时，默认需要具备相关表的权限，如果需要对视图进行独立鉴权，不依赖相关表的权

限，需要将参数spark.ranger.plugin.viewaccesscontrol.enable设置为true。
- 使用非Spark-beeline方式提交作业时，需要在“客户端安装目录/Spark/spark/conf/spark-defaults.conf”中设置该参数。
- 使用Spark-beeline方式提交作业时，需要在“客户端安装目录/Spark/spark/conf/spark-defaults.conf”和服务端“JDBCserver > 自定义配置” 中设置该参数。

5. 单击“Add”，在策略列表可查看策略的基本信息。等待策略生效后，验证相关权限是否正常。如需禁用某条策略，可单击 按钮编辑策略，设置策略开关为“Disabled”。

如果不再使用策略，可单击 按钮删除策略。

6. 重启 Spark：
   i. 重启 Spark 服务以应用配置。

Spark表数据脱敏

Ranger支持对Spark数据进行脱敏处理（Data Masking），可对用户执行的select操作的返回结果进行处理，以屏蔽敏感信息。

1. 修改服务端和客户端spark.ranger.plugin.masking.enable参数值为true。
   - 服务端：登录TBDS Manager页面，选择“标准集群 > 集群服务 > Spark > 配置管理”，搜索并修改所有的spark.ranger.plugin.masking.enable参数值为true，保存配置并重启服务。
   - 客户端：登录Spark客户端节点，进入目录“{客户端安装目录}/Spark/spark/conf/spark-defaults.conf”，修改spark.ranger.plugin.masking.enable参数值为true。
2. 登录Ranger WebUI界面，在首页单击“HADOOP SQL”区域的组件插件名称如“Hive”。
3. 在“Masking”页签单击“Add New Policy”，添加Spark权限控制策略。
4. 根据业务需求配置相关参数。
   Spark数据脱敏参数：

参数名称	描述
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。
Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持“*”通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。
Policy Label	为当前策略指定一个标签，可以根据这些标签搜索报告和筛选策略。
Hive Database	配置当前策略适用的Spark中的数据库名称。
Hive Table	配置当前策略适用的Spark中的表名称。
Hive Column	配置当前策略适用的Spark中的列名称。
Description	策略描述信息。

Audit Logging	是否审计此策略。
Mask Conditions	在“Select Group”、“Select User”列选择已创建好的需要授予权限的用户组或用户，单击“Add Conditions”，添加策略适用的IP地址范围，然后再单击“Add Permissions”，勾选“select”权限。 单击“Select Masking Option”，选择数据脱敏时的处理策略： ◦ Redact：用x屏蔽所有字母字符，用0屏蔽所有数字字符。 ◦ Partial mask: show last 4：只显示最后的4个字符。 ◦ Partial mask: show first 4：只显示开始的4个字符。 ◦ Hash：对数据进行Hash处理。 ◦ Nullify：用NULL值替换原值。 ◦ Unmasked(retain original value)：不脱敏，显示原数据。 ◦ Date: show only year：日期格式数据只显示年份信息。 ◦ Custom：可使用任何有效Hive UDF（返回与被屏蔽的列中的数据类型相同的数据类型）来自定义策略。 如需添加多列的脱敏策略，可单击 按钮添加。
Deny Conditions	策略拒绝条件，配置本策略内拒绝的权限及例外，配置方法与“Allow Conditions”类型。

Spark行级别数据过滤

Ranger支持用户对Spark数据表执行select操作时进行行级别的数据过滤。

1. 修改服务端和客户端spark.ranger.plugin.rowfilter.enable参数值为true。
   - 服务端：登录TBDS Manager页面，选择“标准集群 > 集群服务 > Spark > 配置管理”，搜索并修改所有的spark.ranger.plugin.rowfilter.enable参数值为true，保存配置并重启服务。
   - 客户端：登录Spark客户端节点，进入目录“{客户端安装目录}/Spark/spark/conf/spark-defaults.conf”，修改spark.ranger.plugin.rowfilter.enable参数值为true。
2. 登录Ranger WebUI界面，在首页单击“HADOOP SQL”区域的组件插件名称如“Hive”。
3. 在“Row Level Filter”页签单击“Add New Policy”，添加行数据过滤策略。
4. 根据业务需求配置相关参数。
   Spark行数据过滤参数：

参数名称	描述
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。
Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持“*”通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。

Policy Label	为当前策略指定一个标签，可以根据这些标签搜索报告和筛选策略。
Hive Database	配置当前策略适用的Spark中的数据库名称，仅支持设置一个数据库名，并且不支持“*”通配符。
Hive Table	配置当前策略适用的Spark中的表名称，仅支持设置一个表名，并且不支持“*”通配符。
Description	策略描述信息。
Audit Logging	是否审计此策略。
Row Filter Conditions	在“Select Role”、“Select Group”、“Select User”列选择已创建好的需要授予权限的对象，单击“Add Conditions”，添加策略适用的IP地址范围，然后再单击“Add Permissions”，勾选“select”权限。 单击“Row Level Filter”，填写数据过滤规则。 例如过滤表A中“name”列“zhangsan”行的数据，过滤规则为：name <> 'zhangsan'。更多信息可参考Ranger官方文档。 如需添加更多规则，可单击 按钮添加。

5. 单击“Add”，在策略列表可查看策略的基本信息。

6. 用户通过Spark客户端对配置了数据脱敏策略的表执行select操作，系统将对数据进行处理后进行展示。

开发规范

RDD

利用缓存复用RDD

RDD 中的数据只会在计算中产生，在计算完成后就会消失，这意味着对同一个 RDD 执行算子操作时，都需要从头开始计算。可以使用 persist() 或 cache() 函数将 RDD 缓存在内存或磁盘中，以便在后续的计算中复用，这使得后面的操作速度大大加快（通常快 10 倍以上）。
Spark 缓存级别如下表所示。注意：在 Python 中，存储对象总是被 pickle 库序列化，所以是否选择序列化级别并不重要。

缓存级别	含义
MEMORY_ONLY	在JVM中将RDD存储为反序列化的Java对象。如果内存中没有足够的RDD，某些分区将不会被缓存，而是在每次需要时重新计算。这是默认级别，cache() 函数使用的就是这种缓存策略。
MEMORY_AND_DISK	在JVM中将RDD存储为反序列化的Java对象。如果内存中容纳不下RDD，则将容纳不下的部分存储在磁盘上，需要时从磁盘上读取。
MEMORY_ONLY_SER (Java and Scala)	将RDD存储为序列化的Java对象（每个分区一个字节数组）。这通常比反序列化对象更节省空间，特别是在使用快速序列化器时，但读取时更耗费CPU。
MEMORY_AND_DISK_SER (Java and Scala)	类似于MEMORY_ONLY_SER，但是将内存中无法容纳的分区溢出到磁盘，而不是在每次需要时重新计算。
DISK_ONLY	仅在磁盘上存储 RDD 分区。
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	与上述级别相同，但在两个群集节点上复制每个分区。
OFF_HEAP (experimental)	与MEMORY_ONLY_SER类似，但将数据存储在堆外内存中。这要求启用堆外内存。

Spark 的存储级别是为了在内存使用和 CPU 效率之间提供不同的权衡。建议通过以下过程来选择一个：

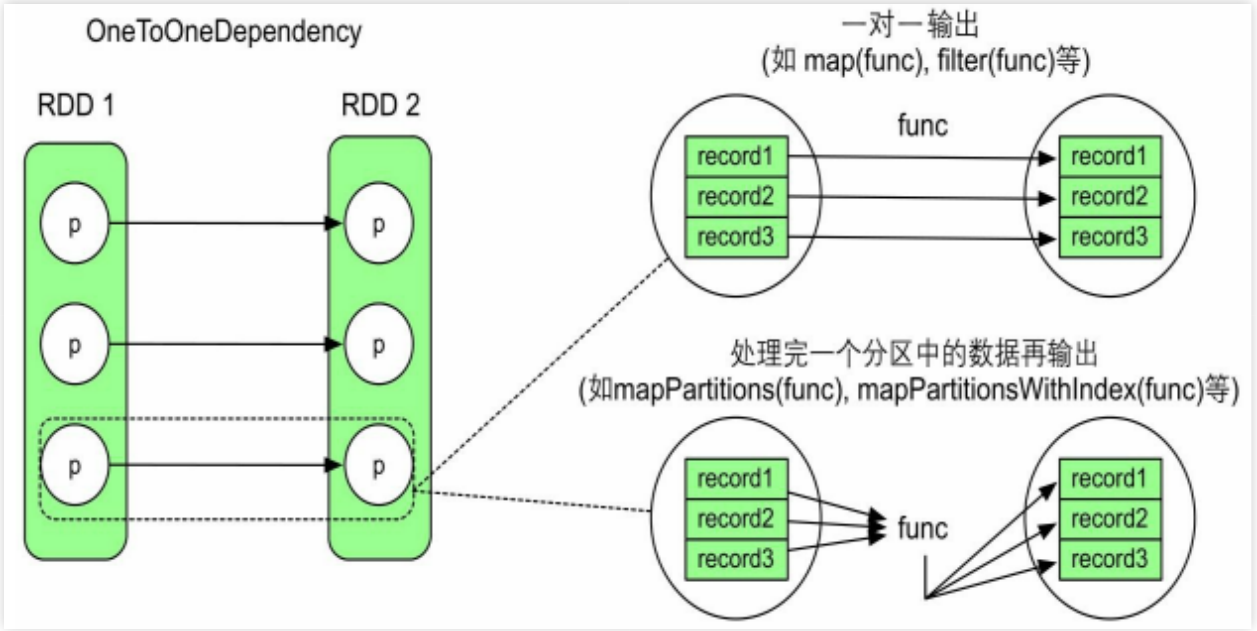
1. 如果 RDD 适合默认存储级别 MEMORY_ONLY，保持该级别。这是最节省 CPU 的选项，可使 RDD 上的操作尽可能快地运行。
2. 如果不行，尝试使用 MEMORY_ONLY_SER，并选择一个快速序列化库，使对象更节省空间，但访问速度仍相

当快。

3. 不要溢出到磁盘，除非计算数据集的函数非常昂贵，或者过滤了大量数据。否则，重新计算一个分区可能和从磁盘读取数据一样快。
4. 如果希望快速故障恢复，使用复制存储级别（例如，如果使用 Spark 来服务来自 Web 应用程序的请求）。所有存储级别都通过重新计算丢失的数据提供完全容错，但复制存储级别可让你继续在 RDD 上运行任务，而无需等待重新计算丢失的分区。

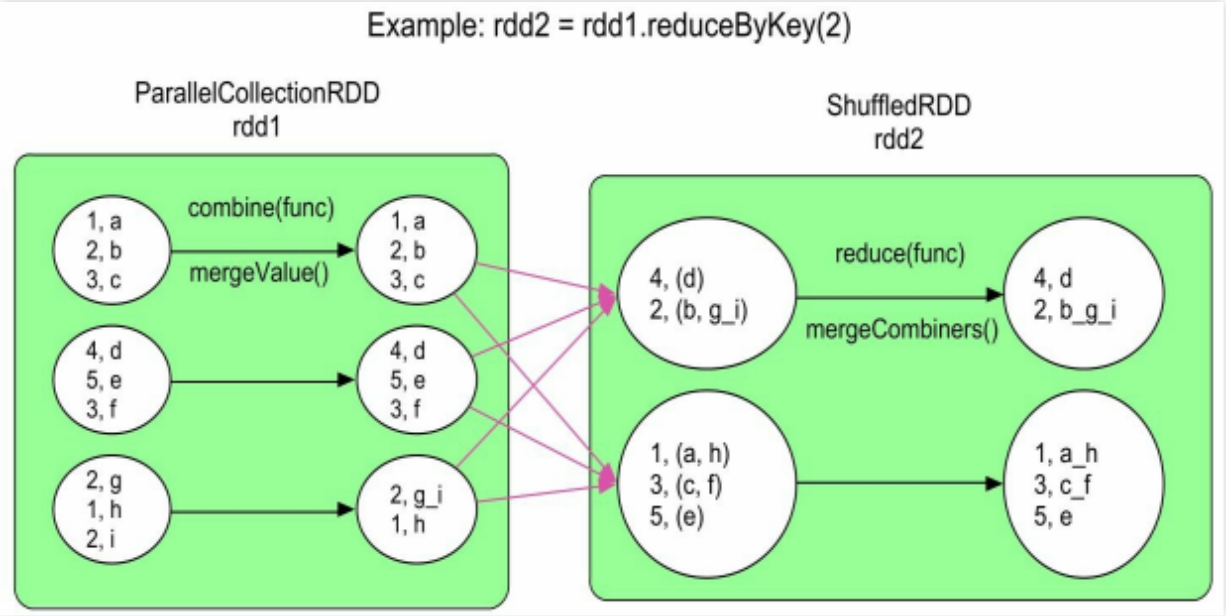
合理选择RDD算子

1. mapPartitions 替换 map
   map 算子是分区内一个个数据的执行，类似于串行操作，所以性能较低；而 mapPartitions 算子是以分区为单位进行批处理操作，所以性能较高。但是 mapPartitions 算子会长时间占用内存，导致内存可能不够用，出现 OOM，所以在内存有限的情况下，不推荐使用，而应使用 map 操作。



2. foreachPartition 替换 foreach
   foreachPartition 和 foreach 的关系类似于 mapPartitions 和 map 的关系。

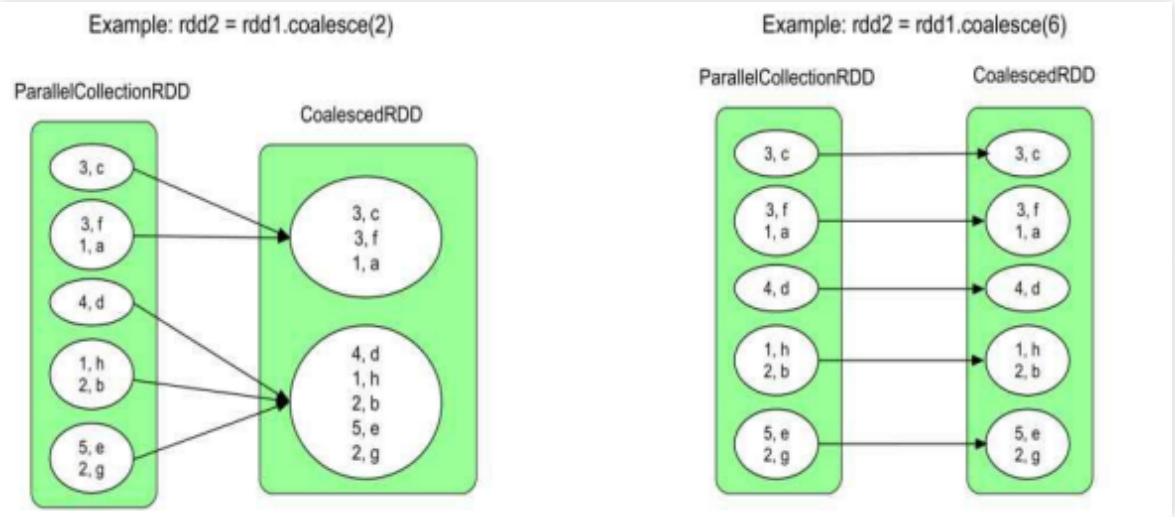
3. reduceByKey/aggregateByKey 替换 groupByKey
   与 groupByKey 不同，reduceByKey(func, numPartitions) 实际包含两步聚合。第一步，在 Shuffle 之前对每个分区的数据进行一个本地化的 combine() 聚合操作，也称为mini-reduce 或 map 端 combine，这一步由 Spark 自动完成，并不形成新的 RDD，减少了数据传输量和内存用量，效率比 groupByKey 高。第二步，reduceByKey 生成新的 ShuffledRDD，将来自不同分区且具有相同 key 的数据聚合在一起，利用 func 进行 reduce() 聚合操作。整个过程中，combine() 和reduce() 的计算逻辑一样，采用同一个 func。



4. coalesce 与 repartition 重分区

   两者均是将 RDD 中的数据进行重新分区，repartition(numPartitions) 语义与 coalesce(numPartitions, true) 一致（第二个参数表示是否 shuffle），即 repartition 一定会进行 shuffle。而 coalesce 将相邻的分区直接合并在一起，形成的数据依赖关系是多对一的窄依赖，不会触发数据的 shuffle。因此在不进行 shuffle 的情况下，coalesce 不能将一个分区拆分为多份，且当 RDD 中不同分区中的数据量差别较大时，直接合并容易造成数据倾斜。为了增加分区或解决数据倾斜问题，可以指定 shuffle=true 将数据打乱。

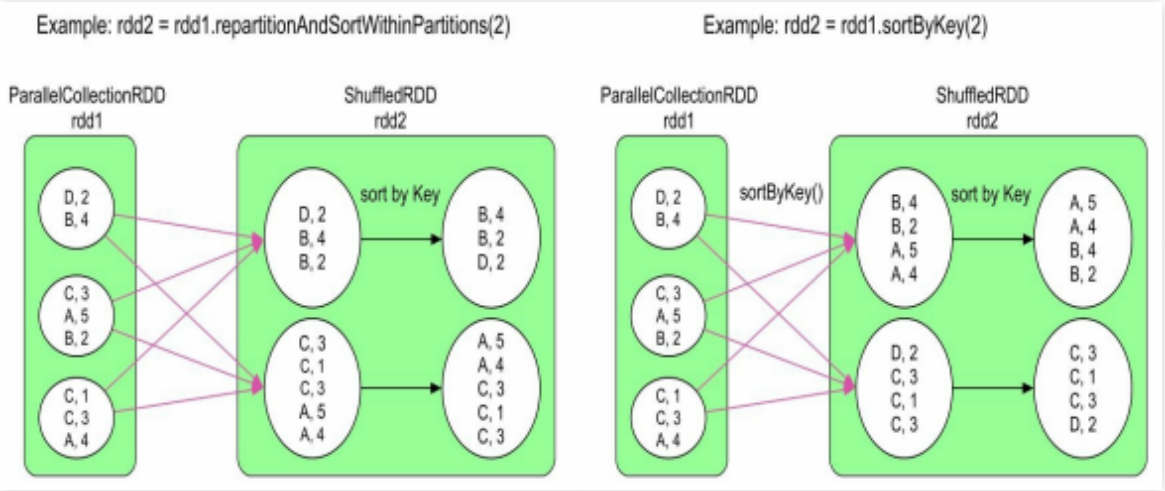
   当数据集较小，而分区数量较多时，可以使用 coalesce 来减少分区数量，从而减少资源消耗和网络传输开销。例如，使用 filter 算子过滤掉 RDD 较多数据后，可以使用 coalesce 减少分区数，根据经验值，分区数一般是核心数的 2-3 倍。



5. repartitionAndSortWithinPartitions 替换 repartition 与 sortByKey

   repartitionAndSortWithinPartitions 可以灵活使用各种分区器，且对于结果 RDD 中的每个分区，对其中的

数据按照 key 进行排序，该操作比 repartition + sortByKey 效率高，因为它可以将排序下放到 shuffle 机制中。不过由于 repartitionAndSortWithinPartitions 可定义分区器，不一定是 sortByKey 默认的 RangePartitioner，因此它得到的结果不能保证 key 全局有序。



减少不必要的action算子

Spark 将数据操作分为两种，transformation 操作（转换算子）和 action 操作（行动算子），两者的区别是行动算子一般是对结果数据进行后处理，产生输出结果，且会触发 Spark 提交 job 真正执行数据处理任务。
当应用程序出现 action 操作时，表示应用会生成一个 job。如果应用程序中有很多 action 操作，那么 Spark 会按照顺序为每个 action 操作生成一个 job，每个 job 的逻辑处理流程都是从输入数据到最后 action 操作。 因此，减少不必要的 action 算子可以提高 Spark 的性能和效率。

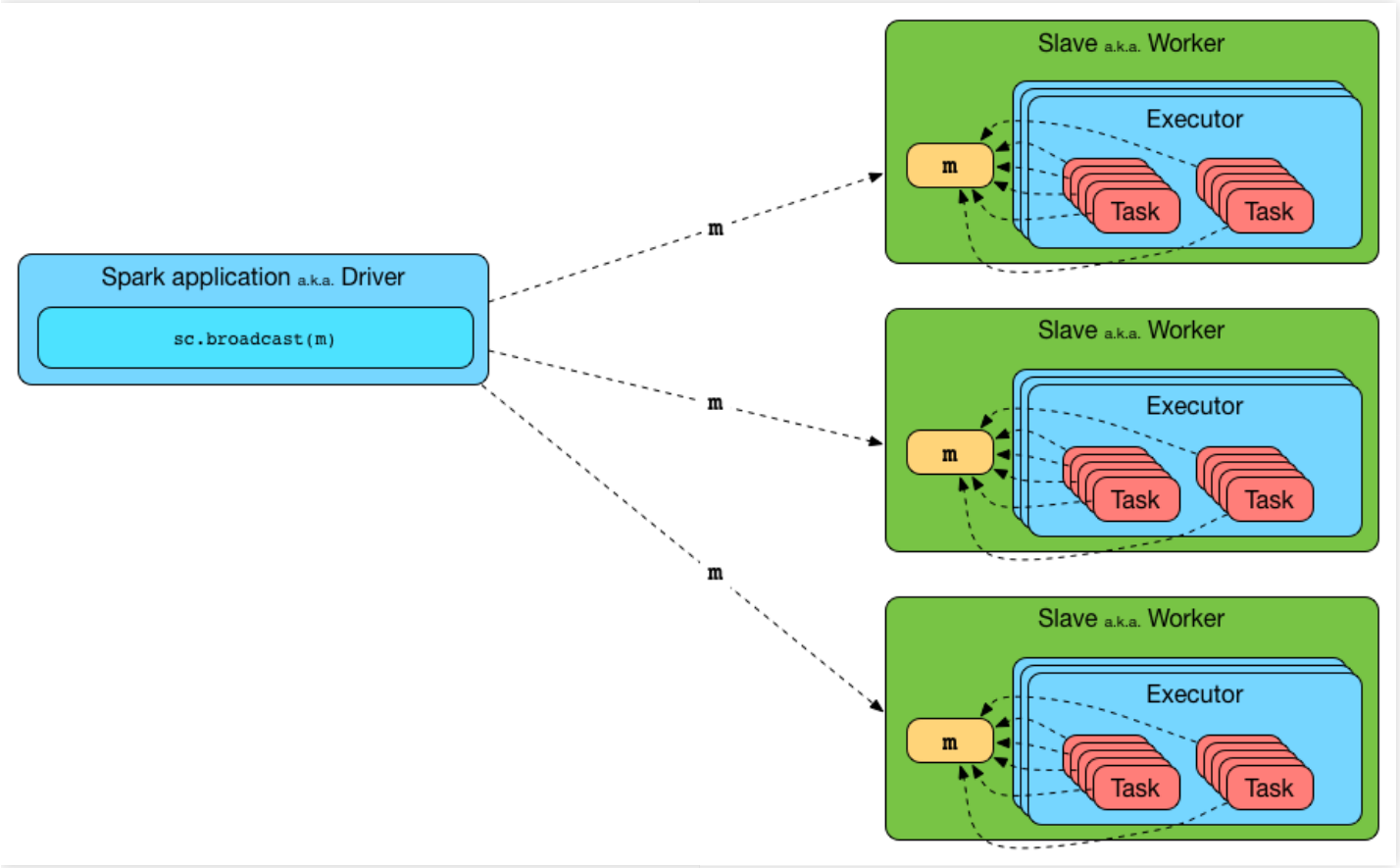
尽量避免使用shuffle算子

Spark 作业运行过程中，最消耗性能的地方就是 shuffle。shuffle 分为 shuffle write 和 shuffle read 两个阶段，前者将 map 端的输出数据按照 key 进行分区，并将输出数据写入到本地磁盘或者网络中，后者读取 shuffle write 阶段输出的数据，并进行合并和计算，最终生成结果。由此可见，磁盘 IO 和网络数据传输是 shuffle 性能较差的主要原因。
因此，应该尽量避免使用 shuffle 算子，如 repartition、reduceByKey、join 等，尽量使用非 shuffle 类算子。

广播大变量

广播变量是一个只读变量，通过它可以将一些共享数据集或者大变量缓存在 Spark 集群中的各个机器上，而不用每个 task 都需要复制一个副本，后续计算可以重复使用，减少了数据传输时网络带宽的使用，提高效率。相比于 Hadoop 的分布式缓存，广播的内容可以跨作业共享。广播变量要求广播的数据不可变、不能太大但也不能太小（一般几十M

以上）、可被序列化和反序列化、并且必须在 Driver 端声明广播变量，适用于广播多个 Stage 公用的数据，存储级别目前是 MEMORY_AND_DISK。




```
val rdd = sc.makeRDD(List(("a", 1), ("b", 2), ("c", 3)))
val map = mutable.Map(("a", 4), ("b", 5), ("c", 6))
// 封装广播变量，如果直接使用map，则每个Task都各自持有该变量，数据重复且占用大量内存
val bc: Broadcast[mutable.Map[String, Int]] = sc.broadcast(map)
rdd.map {
    case (w, c) => {
        val l: Int = bc.value.getOrElse(w, 0)
        (w, (c, l))
    }
}.collect().foreach(println)
```

使用Kryo序列化

序列化在任何分布式应用程序的性能中都起着重要作用，序列化对象速度慢或消耗大量字节的格式将大大降低计算速度。Spark 的目标是在便利性（允许在操作中使用任何 Java 类型）和性能之间取得平衡，它提供了两个序列化库：

1. Java序列化：默认情况下，Spark 使用 Java 的 ObjectOutputStream 框架对对象进行序列化，并可与创建的

任何实现 java.io.Serializable 的类协同工作。还可以通过扩展 java.io.Externalizable 更紧密地控制序列化的性能。Java 序列化非常灵活，但通常相当缓慢，而且会导致许多类的序列化格式过大。

2. Kryo序列化：Spark 还可以使用 Kryo 库更快地序列化对象。Kryo 比 Java 序列化要快得多，也更紧凑（通常是 Java 序列化的 10 倍），但不支持所有可序列化类型，而且需要提前注册程序中使用的类，以获得最佳性能。
   通过使用 SparkConf 初始化作业，并调用 conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer") 可以切换到使用 Kryo。该设置不仅可以配置在worker 节点间 shuffle 数据时使用的序列化器，还可以配置将 RDD 序列化到磁盘时使用的序列化器。Kryo 不是默认设置的唯一原因是需要自定义注册，但建议在任何网络密集型应用中尝试使用。自 Spark 2.0.0 以来，在内部使用 Kryo 序列化器来处理简单类型、简单类型数组或字符串类型的 RDD。
   如果对象较大，可能还需要增加 spark.kryoserializer.buffer 配置。这个值必须足够大，以容纳你要序列化的最大对象。最后，如果你不注册自定义类，Kryo 仍然可以工作，但它必须在每个对象中存储完整的类名，这就造成了浪费。

```
val conf = new SparkConf().setMaster(...).setAppName(...)
// 设置序列化器为KryoSerializer
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
// 向Kryo注册自己的自定义类
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

优化数据结构

默认情况下，Java 对象的访问速度很快，但占用的空间却比其字段中的"原始"数据多出 2-5 倍。原因如下：

1. 每个不同的 Java 对象都有一个约 16 字节的"对象头"，其中包含指向其类的指针等信息。对于数据量极少的对象（例如一个 int 字段）来说，这个头可能比数据量还大。
2. Java 字符串与原始字符串数据相比有大约 40 个字节的开销（因为它们将数据存储在一个 char 数组中，并保留了额外的数据，如长度），并且由于字符串内部使用 UTF-16 编码，每个字符存储为两个字节。因此，一个 10 个字符的字符串可以轻松占用 60 个字节。
3. 常见的集合类，如 HashMap 和 LinkedList，使用链接数据结构，其中每个条目都有一个"封装"对象（如 Map.Entry）。这个对象不仅有一个标头，还有指向列表中下一个对象的指针（通常每个指针 8 字节）。
4. 原始类型的集合通常将它们存储为"包装"对象，如 java.lang.Integer。

减少内存消耗的第一种方法是避免使用会增加开销的 Java 特性，Spark 官方建议：

1. 设计数据结构时，优先选择对象数组和原始类型，而不是标准的 Java 或 Scala 集合类（如 HashMap）。
2. 尽可能避免使用包含大量小对象和指针的嵌套结构。
3. 考虑使用数字 ID 或枚举对象代替字符串作为键。

Spark SQL

注意 null 的特殊性

SQL 对 null 的处理比较特殊，一些计算逻辑中经常会直接跳过 null。但用户经常将 null 当作一个具体的值，因此会出现一些困惑的情况。例如，null 既不参与 in 表达式计算，也不参与 not in 表达式计算，若数据中存在 null，则两个表达式得到的结果之和并不等于总的数据结果。
另外，需要注意过滤 key 为空字符串和 null 的情形，因为如果空字符串或 null 的记录数较多，就可能导致大量数据分配到同一个 task 执行，引起数据倾斜，进而导致任务执行时间变长。

多表 join 顺序调整

多表 join 场景中，数据表的顺序对性能影响比较大，例如，A join B join C 与 A join C join B，两者中间产生的数据量可能差别很大。实际上，多表 join 一直都是数据库中基于代价优化机制（Cost-Based Optimization，简称 CBO）的重要针对对象，如果是针对数据相对固定的表进行 SQL 查询，建议打开该配置，相关参数说明如下表所示。不过要使用该功能，需确保相关表和列的统计信息已经生成，并定期更新和维护。

参数	默认值	参数说明
spark.sql.cbo.enabled	false	是否启用 CBO
spark.sql.cbo.joinReorder.enabled	false	在 CBO 中启用 join 重排

```
-- 生成表级别统计信息
ANALYZE TABLE 表名 COMPUTE STATISTICS
-- 生成列级别统计信息
ANALYZE TABLE 表名 COMPUTE STATISTICS FOR COLUMNS 列 1, 列 2, 列 3

-- 显示表统计信息
DESC FORMATTED 表名
-- 显示列统计信息
DESC FORMATTED 表名 列名
```

复用数据

数据复用的场景较多，绝大部分以 union 操作为主。如果读取同一份数据的两个任务之间没有依赖关系，可以想办法合并任务逻辑，使得只需要读取一次数据，减少 IO 代价。

```
-- 数据表t被读取两次，当表数据量非常大时，对性能影响大
select *
from (
    select value form t where key = k1
    union all
    select value form t where key = k2
)
-- 优化后，只需要读取一次数据表，减少IO代价
select *
from (
    select value form t where key = k1 or key = k2
)
```

处理数据倾斜

数据倾斜是经常碰到的一类问题，可通过以下方法来优化或规避：

1. 过滤无关数据：大量的 null 数据没有过滤，参与了 join 的执行；存在“脏数据”，不满足原有的数据类型。
2. 广播小表：若参与 join 操作的两个表是大小表，可以采用 BroadcastJoin 方式，即将小表广播到大表所在的 executor 上，避免数据倾斜。Spark 支持在 SQL 中通过添加 Hint 的方式强制采用 BroadcastJoin，不过需要注意，对于外连接，基表不能被广播，因此左外连接中左表不可以是小表，右外连接中右表不可以是小表。

```
-- 将小表t1广播到大表t2所在的Executor上
select /*+ BROADCAST(t1) */ * from t1, t2 where t1.key = t2.key
```

3. 分离倾斜数据：假设参加 join 操作的两个表分别为 t1、t2，其中表 t1 有数据倾斜。可以将 t1 的数据分为两部分：不含倾斜数据的 t11、只包含倾斜数据的 t12。数据表 t11 和 t12 分别与 t2 进行 join 操作，然后将结果合并。首先，t11 与 t2 的 join 操作不存在数据倾斜；其次，由于 t12 通常不会很大，所以 t12 与 t2 的 join 操作可以采用第二种方法执行 BroadcastJoin。

```
select * from (
    select * from t11, t2 where t11.key = t2.key
    union all
    select /*+ BROADCAST(t12) */ * from t12, t2 where t12.key = t2.key
)
```

4. 打散数据：假设表 A 和表 B 都有 id、value 字段，现在对这两个表按照 id 进行 join 操作，即 A.id = B.id。此时，因为 id 都为 a，所有数据会在一个 task 上进行关联操作，这样就出现了数据倾斜。处理方法就是将大表 A 中的 id 加上后缀 0 - n，起到打散的作用，为了结果正确，小表 B 中的 id 需要将每条数据都复制 n 份。如下图所示，正是由于小表 B 复制了多份，所以无论大表 A 打上哪个随机后缀，都可以保证能跟小表 B 中的某

一条数据 join 上。此时再进行 join 操作，将会产生 3 个 task，每个 task 只需要关联一条数据，起到分散的作用。

```
-- 表A
select id, value, concat(id, (rand() * 10000) % 3) as new_id
from A
-- 表B
select id, value, concat(id, suffix) as new_id
form (
    select id, value, suffix
    from B Lateral View explode(array(0, 1, 2)) tmp  as suffix
)
```

自适应查询执行AQE

Spark 3.0 优化查询性能的另一利器是自适应查询执行（Adaptive Query Execution，简称 AQE），它可以在查询执行过程中动态地调整查询计划，以提高查询性能。一旦启用 AQE（spark.sql.adaptive.enabled=true），Spark SQL 会采用以下策略来优化查询：

1. 动态调整 shuffle 分区数：在执行 shuffle 操作时，Spark SQL 会根据数据量和集群资源动态调整 shuffle 分区数，以避免资源浪费和数据倾斜。
2. 动态调整 join 策略：Spark SQL 会根据数据大小和 join 条件的复杂度动态选择 join 策略，以提高查询性能。
3. 动态调整 Broadcast Join 阈值：Spark SQL 会根据数据大小和集群资源动态调整 Broadcast Join 的阈值，以避免数据倾斜和 OOM。
4. 动态调整数据倾斜处理策略：当查询出现数据倾斜时，Spark SQL 会自动调整数据倾斜处理策略，以避免查询失败。
   总之，启用 AQE 可以大大提高 Spark SQL 查询的性能和稳定性。但需要注意的是，自适应查询优化器可能会造成一些额外的开销，因此在使用时需要根据实际情况进行权衡。

性能调优

1. 内存 & CPU

参数名	默认值	说明	调优建议
spark.driver.memory	1g	Driver 进程的堆内存大小，命令行使用 --driver-memory。运行 Driver 的容器的最大内存为 spark.driver.memoryOverhead 和 spark.driver.memory 的总和。	当使用 collect 等算子将数据收集到 Driver 端，需要加大内存，否则容易造成 OOM。通常设置 1G - 4G 较为合适。
spark.driver.memoryOverhead	driverMemory * spark.driver.memoryOverheadFactor，最小 384m	Driver 进程的非堆内存大小，非堆内存包括：堆外内存（spark.memory.offHeap.enabled=true 时）、其他 Driver 进程（如与 PySpark Driver 一起运行的 python 进程）使用的内存以及在同一容器中运行的其他非 Driver 进程使用的内存。注意，spark.driver.memoryOverheadFactor 默认值为 0.10。	建议保持默认值。
spark.executor.memory	1g	Executor 进程的堆内存大小，命令行使用 --executor-memory。运行 Executor 的容器的最大内存大小为 spark.executor.memory、spark.executor.memoryOverhead、spark.memory.offHeap.size 和 spark.executor.pyspark.memory 的总和。	num-executors * executor-memory < Yarn 资源队列最大内存，若与其他用户共享 Yarn 资源队列，则最好不要超过资源队列的 1/3 - 1/2。通常设置 4G - 8G 较为合适。
spark.executor.memoryOverhead	executorMemory * spark.executor.memoryOverheadFactor，最小 384m	为每个 Executor 分配的额外内存量，额外内存包括 PySpark Executor 内存（当未配置 spark.executor.pyspark.memory 时）和在同一容器中运行的其他非 Executor 进程使用的内存。注意，spark.executor.memoryOverheadFactor 默认值为 0.10。	建议保持默认值。
spark.memory.offHeap.enabled	false	如果为 true，Spark 将尝试在某些操作中使用堆外内存。如果启用了堆外内存，则 spark.memory.offHeap.size 必须为正值。	建议保持默认值。
spark.memory.offHeap.size	0	堆外内存大小。	
spark.driver.cores	1	Driver 进程使用的内核数，仅在 Cluster 模式下使用，命令行使用 --driver-cores。	建议保持默认值。
spark.executor.cores	1（Yarn 模式）	Executor 进程使用的内核数，命令行使用 --executor-cores。	num-executors * executor-cores < Yarn 资源队列最大 vCores，若与其他用户共享 Yarn 资源队列，则最好不要超过资源队列的 1/3 - 1/2。通常设置 2 - 5 个较为合适。
spark.executor.instances	2	静态分配的 Executor 数量，命令行使用 --num-executors。	见 spark.executor.cores、spark.executor.memory 调优建议。
spark.yarn.am.memory	512m	在 Client 模式下，YARN Application Master 使用的内存量，单位使用小写后缀。在 Cluster 模式下，请使用 spark.driver.memory。	建议保持默认值。
spark.yarn.am.memoryOverhead	AM memory * 0.10，最小 384m	与 spark.driver.memoryOverhead 相同，但适用于 Client 模式下的 YARN Application Master。	建议保持默认值。

注 1：参数 spark.executor.memoryOverhead 与 spark.memory.offHeap.size 区别，在 Spark 2.4.0 之前，Executor 非堆内存 = spark.executor.memoryOverhead（如果指定了参数 spark.memory.offHeap.size，需要手动将其添加到 Yarn 的 memoryOverhead 中）；在 Spark 2.4.0 至 Spark 3.0.0 之前，Executor 非堆内存 = spark.executor.memoryOverhead（同前） + spark.executor.pyspark.memory；自 Spark 3.0.0 开始，Executor 非堆内存 = spark.executor.memoryOverhead + spark.memory.offHeap.size + spark.executor.pyspark.memory。
注 2：参数 spark.driver.memory 与 spark.yarn.am.memory 区别，Yarn Cluster 模式，ApplicationMaster 在任意一台 NodeManager 上启动，此方式 ApplicationMaster 包含 Driver，AM 内存为：spark.driver.memory + spark.driver.memoryOverhead；Yarn Client 模式，Driver 在提交任务的节点启动，而 ApplicationMaster 在任意一台 NodeManager 上启动，此方式 Driver 和 AM 是分开的，AM 内存为：spark.yarn.am.memory + spark.yarn.am.memoryOverhead。

2. 动态分配

参数名	默认值	说明	调优建议
spark.dynamicAllocation.enabled	false	是否启用动态资源分配，动态资源分配可根据工作负载上下调整应用程序注册的 Executor 数量。这需要设置 spark.shuffle.service.enabled 或 spark.dynamicAllocation.shuffleTracking.enabled。	可让用户免于繁琐的 Executor 数目的预估和设置，增加业务运行的稳定性，提高集群资源的利用率，建议设置为 true。
spark.shuffle.service.enabled	false	是否启用 ESS。该服务会保留 Executor 写入的 Shuffle 文件，这样就可以安全地移除 Executor，或者在 Executor 发生故障时继续进行 Shuffle。	建议设置为 true。
spark.dynamicAllocation.initialExecutors	spark.dynamicAllocation.minExecutors	如果启用动态分配，要运行 Executor 的初始数量。如果设置了 --num-executors（或 spark.executor.instances）且大于此值，则会将其用作初始 Executor 数量。	参数较小时，任务需要等待向 Yarn 申请资源，造成任务运行有较长的爬坡阶段；参数较大时，对于不需要那么多 Executor 的任务来说，会造成资源浪费。该参数值的选取可以根据历史任务 Executor 数目的统计，按照二八原则来设置，例如 80% 历史业务的 Executor 数目都不大于参数值。若无法确认历史任务的 Executor ，建议先设置为 1。
spark.dynamicAllocation.minExecutors	0	如果启用动态分配，Executor 数量的下限。	建议与 spark.dynamicAllocation.initialExecutors 保持一致。
spark.dynamicAllocation.maxExecutors	infinity（无穷大）	如果启用动态分配，Executor 数量的上限。	为了防止大业务独占资源，造成小任务没有资源的情况，需要将该参数值设置为一个合理值，如 200。
spark.dynamicAllocation.executorIdleTimeout	60s	如果启用动态分配，且某个 Executor 的空闲时间超过这一期限，则该 Executor 将被移除。	参数较小时，利于集群资源共享，但会影响业务执行时，在 Executor 被删除后，可能需要重新申请新的 Executor 来执行任务；参数较大时，不利于资源共享，若一些较大的任务占用资源，迟迟不释放，就会造成其他任务得不到资源。建议保持默认值。

3. 自适应查询 AQE

参数名	默认值	说明	调优建议
spark.sql.adaptive.enabled	true	是否启用 AQE，在查询执行过程中，根据准确的运行时统计数据重新优化查询计划。	建议保持默认值。
spark.sql.adaptive.advisoryPartitionSizeInBytes	64 MB	自适应优化期间 Shuffle 分区的建议大小，它在 Spark 合并小的 Shuffle 分区或拆分倾斜的 Shuffle 分区时生效。	
spark.sql.adaptive.coalescePartitions.enabled	true	当启用 AQE 且该值为 true 时，Spark 将根据目标大小（由 spark.sql.adaptive.advisoryPartitionSizeInBytes 指定）合并连续的 Shuffle 分区，以避免过多的小任务。	建议保持默认值。
spark.sql.adaptive.coalescePartitions.parallelismFirst	true	该值为 true 时，Spark 在合并 Shuffle 分区时会忽略 spark.sql.adaptive.advisoryPartitionSizeInBytes 指定的目标大小，而只尊重 spark.sql.adaptive.coalescePartitions.minPartitionSize 指定的最小分区大小，以最大化并行性。这是为了避免在启用 AQE 执行时出现性能下降。	官方建议将此配置设为 false，并遵守 spark.sql.adaptive.advisoryPartitionSizeInBytes 指定的目标分区大小。

参数名	默认值	说明	调优建议
spark.sql.adaptive.coalescePartitions.minPartitionSize	1MB	合并后 Shuffle 分区的最小大小。其值最多为 spark.sql.adaptive.advisoryPartitionSizeInBytes 的 20%。当目标大小在分区合并过程中被忽略（默认情况）时，这个值就非常有用。	
spark.sql.adaptive.coalescePartitions.initialPartitionNum	spark.sql.shuffle.partitions	合并前的初始 Shuffle 分区数。只有同时启用 spark.sql.adaptive.enabled 和 spark.sql.adaptive.coalescePartitions.enabled 时，此配置才会生效。	建议保持默认值。
spark.sql.adaptive.autoBroadcastJoinThreshold	spark.sql.autoBroadcastJoinThreshold	当任何 Join 方的运行时统计数据小于该值时，AQE 会将 Sort-Merge Join 连接转换为 Broadcast Hash Join。注意，此配置仅在 AQE 中使用。	建议保持默认值。
spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold	0	如果该值不小于 spark.sql.adaptive.consultativePartitionSizeInBytes，且所有分区的大小都不大于该配置，则无论 spark.sql.join.preferSortMergeJoin 的值如何，AQE 会将 Sort-Merge Join 转换为 Shuffle Hash Join。	
spark.sql.adaptive.skewJoin.enabled	true	当启用 AQE 且该值为 true 时，Spark 会通过拆分（必要时复制）倾斜分区来动态处理 Sort-Merge Join 中的倾斜。	建议保持默认值。
spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes	256MB	如果分区的大小大于此阈值，且大于 spark.sql.adaptive.skewJoin.skewedPartitionFactor 乘以分区大小中值，则该分区被视为倾斜分区。理想情况下，此配置应大于 spark.sql.adaptive.consultivePartitionSizeInBytes。	

4. Shuffle

参数名	默认值	说明	调优建议
spark.shuffle.file.buffer	32k	每个 Shuffle 文件输出流的内存缓冲区大小，这些缓冲区可减少创建中间 Shuffle 文件时的磁盘寻道和系统调用次数。	若内存资源充足，可适当调大该参数（如 64k、128k），从而减少 Shuffle Write 溢写磁盘的次数，进而减少磁盘I/O次数，提升性能。
spark.reducer.maxSizeInFlight	48m	从每个 Reduce 任务中同时获取的 Map 输出的最大大小。由于每个输出都需要我们创建一个缓冲区来接收，这代表了每个 Reduce 任务的固定内存开销，因此除非有大量内存，否则请将其保持在较小的范围内。	若内存资源充足，可适当调大该参数（如 96m、128m），从而减少 Shuffle Read 拉取数据的次数，进而减少网络传输的次数，提升性能。
spark.shuffle.io.maxRetries	3	（仅限 Netty）自动重试因 I/O 相关异常而失败的最大拉取次数。这种重试逻辑有助于在出现长时间 GC 停顿或瞬时网络连接问题时稳定大型 Shuffle。	对于特别耗时的大型 Shuffle 操作，可适当调大该参数（如 60），以避免长时间 GC 停顿或瞬时网络连接问题导致数据拉取失败。
spark.shuffle.io.retryWait	5s	（仅限 Netty）重试拉取之间的等待时间。重试造成的最大延迟默认为 15 秒，计算公式为 maxRetries * retryWait。	对于特别耗时的大型 Shuffle 操作，可适当调大该参数（如 60s），有助于稳定大型 Shuffle。
spark.shuffle.push.enabled	false	设为 true 可在客户端启用基于推送的 Shuffle 功能，并与服务器端标志 spark.shuffle.push.server.mergedShuffleFileManagerImpl 配合使用。	官方说明基于推送的 Shuffle 可提高长时间运行的作业/查询的性能，因为在 Shuffle 过程中会涉及大量磁盘 I/O。但目前，它不太适合处理较少 Shuffle 数据的快速运行作业/查询。建议在涉及大量 Shuffle 时开启。

5. Spark SQL

参数名	默认值	说明	调优建议
spark.sql.shuffle.partitions	200	为连接或聚合而 Shuffle 数据时使用的默认分区数。	官方建议参数值为 num-executors * executor-cores 的 2-3 倍较为合适。
spark.sql.autoBroadcastJoinThreshold	10MB	配置执行 Join 时向所有 Worker 节点广播的表的最大大小。将该值设置为-1，可以禁用广播。注意，目前只有运行了 ANALYZE TABLE COMPUTE STATISTICS NOSCAN 命令的 Hive Metastore 表和直接在数据文件上计算统计数据的基于文件的数据源表才支持统计数据。	若内存资源充足，可适当调大该参数（如 64M），以提高 Join 效率。
spark.sql.broadcastTimeout	300	广播的超时时间（秒）。	建议根据广播大小进行相应调整。
spark.sql.optimizer.runtime.bloomFilter.enabled	true	该值为 true 时，如果 Shuffle Join 的一侧有选择性谓词，会尝试在另一侧插入 Bloom Filter，以减少 Shuffle 数据量。	Spark 3.4 之前默认为 false，建议设置为 true。
spark.sql.optimizer.runtimeFilter.semiJoinReduction.enabled	false	该值为 true 时，如果 Shuffle Join 的一侧有选择性谓词，会尝试在另一侧插入 Semi Join，以减少 Shuffle 数据量。	
spark.sql.files.maxPartitionBytes	128MB	读取文件时打包到单个分区的最大字节数。此配置仅在使用 Parquet、JSON 和 ORC 等基于文件的源时有效。	建议保持默认值。
spark.sql.cbo.enabled	false	是否启用基于成本的优化（Cost-Based Optimization，CBO）。CBO 可以基于表和列的统计信息，进行一系列估算，最终选择出最优的查询计划，比如：Build 侧选择、Join 类型优化、多表 Join 顺序优化等。Spark 自 2.2.0 支持 CBO，之前都使用基于规则的优化器（Rule-Based Optimization，RBO）。	CBO 目前有许多限制，比如，数据统计信息缺失，统计信息不准确，UDF 成本估计困难等。因此，Spark 3.3.0 基于运行时统计信息实现了 AQE，可以动态调整 Join 类型。建议设置为 false，若要启用该功能，需确保相关表和列的统计信息已经生成，并定期更新和维护，同时调整 spark.sql.cbo.joinReorder.enabled 等 CBO 相关配置。
spark.sql.parquet.writeLegacyFormat	false	该值为 true 时，数据将以 Spark 1.4 及更早版本的方式写入，例如，十进制值将以 Apache Parquet 的固定长度字节数组格式写入，而 Apache Hive 和 Apache Impala 等其他系统都使用这种格式。该值为 false 时，则使用 Parquet 中较新的格式，例如，小数将以基于 int 的格式写入。	官方建议如果 Parquet 输出要用于不支持这种较新格式的系统，则应设置为 true。

注 1：使用 ANALYZE TABLE 语句可收集表的统计信息，基本语法请参阅说明。

6. 其它参数

参数名	默认值	说明	调优建议
spark.default.parallelism	对于 reduceByKey 和 join 等 Shuffle 操作，取决于父 RDD 中的最大分区数。对于无父 RDD 的并行化等操作，则取决于集群管理器，Yarn 模式下为所有 Executor 节点的内核数与 2 的最大值	当用户未设置时，由 join、reduceByKey 和 parallelize 等转换返回的 RDD 中的默认分区数。	官方建议参数值为 num-executors * executor-cores 的 2-3 倍较为合适。
spark.locality.wait	3s	在放弃并在本地化程度较低的节点上启动任务之前，等待启动数据本地化任务的时间。相同的等待时间将用于跨越多个本地化级别（进程本地化 process-local、节点本地化 node-local、机架本地化 rack-local、然后是任意级别），还可以通过设置 spark.locality.wait.node 等自定义每个级别的等待时间。	官方建议如果任务时间较长，本地性较差，则应增加这一设置，但默认设置通常效果不错。
spark.speculation	false	该值为 true 时，会启用 Task 的推测执行。这意味着，如果某个 Task 执行缓慢时，Spark 会启动一个备份 Task 来替代，哪个 Task 先完成，就取该 Task 的结果，并 Kill 掉另一个 Task。	推测执行本质是以空间换时间，但 Task 执行缓慢的原因有很多，若推测 Task 也变为缓慢 Task，则会导致情况进一步恶化。对于集群内有不同性能的机器，或执行慢的 Task 集中在同一个机器，建议设置为 true。
spark.speculation.quantile	0.75	在启用对特定 Stage 的推测执行之前，必须完成的 Task 比例。	

参数名	默认值	说明	调优建议
spark.serializer	org.apache.spark.serializer.JavaSerializer	用于序列化需要通过网络发送或需要以序列化形式缓存对象的类。默认的 Java 序列化适用于任何可序列化的 Java 对象，但速度较慢。	官方建议在需要速度时使用 org.apache.spark.serializer.KryoSerializer 并配置 Kryo 序列化，可以是 org.apache.spark.Serializer 的任何子类。
spark.kryo.unsafe	true	是否使用基于不安全的 Kryo 序列化器，使用基于不安全的 IO 可以大大提高速度。	

注 1：spark.sql.shuffle.partitions 与 spark.default.parallelism 区别，前者是在连接和聚合时使用，适用于 SQL；后者只适用于原始 RDD，如果正在执行的任务不是连接或聚合，且使用的是 DataFrame，那么上述参数均不会有任何影响，若实在分不清两者差别，可以同时设置。

7. 关联组件参数

参数名	默认值	说明	调优建议
spark.yarn.shuffle.stopOnFailure	[yarn-site.xml] false	当 Spark Shuffle 服务初始化失败时，是否停止 NodeManager。这可以防止在 Spark Shuffle 服务未运行的 NodeManager 上运行容器导致的应用程序故障。	建议保持默认值。
spark.shuffle.push.server.mergedShuffleFileManagerImpl	[yarn-site.xml] org.apache.spark.network.shuffle.NoOpMergedShuffleFileManager	管理基于推送 Shuffle 的 MergedShuffleFileManager 实现类名。默认情况下，服务器端禁用基于推送的 Shuffle。若要启用，将此配置设置为 org.apache.spark.network.shuffle.RemoteBlockPushResolver。	建议在涉及大量 Shuffle 时开启。
yarn.nodemanager.local-dirs	[yarn-site.xml] ${hadoop.tmp.dir}/nm-local-dir	存储本地化文件的目录列表。应用程序的本地化文件目录位于 ${yarn.nodemanager.local-dirs}/usercache/${user}/appcache/application_${appid}，单个容器的工作目录（称为 container_${contid}）将是该目录的子目录。	Spark 会将 Shuffle 数据写在此目录，如果只配置一个盘，当 Shuffle 数据较多时，会影响整个 Shuffle 读写性能，需要配置成多个盘。
yarn.nodemanager.log-dirs	[yarn-site.xml] ${yarn.log.dir}/userlogs	存储容器日志的目录列表。应用程序的本地化日志目录位于 ${yarn.nodemanager.log-dirs}/application_${appid}，单个容器的日志目录（称为 container_{$contid}）将是该目录的子目录，每个容器目录都将包含由该容器生成的 stderr、stdin 和 syslog 文件。	同样需要配置多个盘，如果只配置一个盘，有可能任务较多时，日志会把磁盘写满。
YARN_NODEMANAGER_HEAPSIZE	[yarn-env.sh] HADOOP_HEAPSIZE_MAX	指定 NodeManager 的最大堆大小。这个值将被 HADOOP_OPTS或 YARN_NODEMANAGER_OPTS 中指定的 Xmx 设置覆盖，默认值与 HADOOP_HEAPSIZE_MAX（mapred-env.sh，默认 1000MB）相同。	Spark ESS 是附属在 NodeManager 上的一个服务，随 NodeManager 一起启动，若 NodeManger JVM 内存配置太小，会影响 ESS 的稳定性。建议该值大于等于 4G。
yarn.nodemanager.resource.memory-mb	[yarn-site.xml] -1	可分配给容器的物理内存（单位：MB）。如果设置为-1，且 yarn.nodemanager.resource.detect-hardware-capabilities 为 true，则会自动计算（适用于 Windows 和 Linux）。在其他情况下，默认值为 8192MB。	建议节点总内存的 80%，预留部分内存。
yarn.scheduler.maximum-allocation-mb	[yarn-site.xml] 8192	RM 中单个容器可申请的最大内存（单位：MB），大于此值的内存申请将引发 InvalidResourceRequestException。	建议节点总内存的 80%，最大不超过 yarn.nodemanager.resource.memory-mb。
yarn.nodemanager.resource.cpu-vcores	[yarn-site.xml] -1	可分配给容器的 vcores 数量。RM 调度器在为容器分配资源时使用此值，它不用于限制 YARN 容器使用的 CPU 数量。如果设置为 -1，且 yarn.nodemanager.resource.detect-hardware-capabilities 为 true，则在 Windows 和 Linux 环境下，该值将根据硬件自动确定。在其他情况下，vcores 数量默认为 8。	建议节点总核数的 90%，预留部分 Core。

8. 辅助调优

有以下几个辅助调优工具，重要性依次递减。

1、Spark History Server（SHS）：重点关注 Jobs（包括 Job 状态、数量、Event Timeline，以及包含的 Stage 汇总信息）、Stages（包括 Stage 读取/写入/Shuffle Write/Shuffle Read 数据量，以及包含的 Task 汇总信息）、Executors（包括内存、磁盘、内核数使用情况，以及 Task 和 Shuffle 信息）、SQL（包括 Job、执行计划等信息），具体参考 WEB-UI 说明。

2、EXPLAIN 执行计划：EXPLAIN 语句用于为 SQL 提供逻辑/物理计划，默认只提供有关物理计划的信息。语法为： EXPLAIN [EXTENDED | CODEGEN | COST | FORMATTED] statement ，具体参考 EXPLAIN 说明。其中 Parsed Logical Plan 表示未解析的逻辑计划，Analyzed Logical Plan 表示解析后的逻辑计划，Optimized Logical Plan 表示优化后的逻辑计划，Physical Plan 表示物理计划。物理计划中，常见类型节点含义如下表所示。

类型节点	说明
HashAggregate	数据聚合，一般 HashAggregate 成对出现，第一个 HashAggregate 是将执行节点本地的数据进行局部聚合，另一个 HashAggregate 是将各个分区的数据进一步进行聚合计算。
Exchange	数据 Shuffle，表示需要在集群上移动数据，很多时候 HashAggregate 会以 Exchange 分隔开来。
Project	投影操作，即选择列，如 select name, age...。
BroadcastHashJoin	基于广播方式进行 HashJoin。

3、任务日志（Yarn & 本地）：主要获取 Driver 日志，Cluster 模式日志位于 Yarn，Client 默认日志位于本地。

最佳实践

Spark Structured Streaming

本教程演示的是：在 Kerberos 环境下，TBDS 集群运行 Spark Structured Streaming 作业来消费 Kafka 数据，实现统计单词个数，需要提前在集群中创建 Kafka Topic。更多说明请查看示例工程 README.md。

准备应用程序

1. 参考【Spark 示例工程开发】，下载样例代码，选择对应迭代分支，将示例工程进行编译打包。
2. 在 SparkStructuredKafkaWordCountJavaExample 示例工程中，通过使用 Spark Structured Streaming 调用 Kafka 接口来获取单词记录，然后把单词记录分类统计，最后输出每个单词的记录数。

```
// 创建SparkSession
SparkSession spark = SparkSession
  .builder()
  .appName("JavaStructuredKafkaWordCount")
  .getOrCreate();

// 创建DataSet，表示从Kafka接收到的输入行流
Dataset<String> lines = spark
  .readStream()
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option(subscribeType, topics)
   // 设置kafka kerberos认证（如有必要）
  .option("kafka.security.protocol", "SASL_PLAINTEXT")
  .option("kafka.sasl.mechanism", "GSSAPI")
  .option("kafka.sasl.kerberos.service.name", "hadoop")
  .load()
  .selectExpr("CAST(value AS STRING)")
  .as(Encoders.STRING());

// 生成运行中的单词计数
Dataset<Row> wordCounts = lines.flatMap(
    (FlatMapFunction<String, String>) x -> Arrays.asList(x.split(" ")).iterator(),
    Encoders.STRING()).groupBy("value").count();

// 开始运行查询，该查询将运行中的单词计数打印到控制台
StreamingQuery query = wordCounts.writeStream()
  .outputMode("complete")
  .format("console")
  .start();
```

```
query.awaitTermination();
```

运行 Spark Structured Streaming 作业

1. 准备主题：创建一个名为 spark_structured_streaming_topic 的 topic。
2. 准备用户：获取用户认证信息，需要将用户的 keytab 文件下载到本地，然后上传至开发机。这里将 test 用户的 test.keytab 文件上传至开发机的 /tmp 目录，并进行 kerberos 认证。

```
# Kerberos认证
[test@10 ~]$ klist -kt /tmp/test.keytab
Keytab name: FILE:/tmp/test.keytab
KVNO Timestamp           Principal
---- ------------------- ------------------------------------------------------
   1 11/28/2024 22:05:22 test@TBDS-JYOKXQY8
   1 11/28/2024 22:05:22 test@TBDS-JYOKXQY8
[test@10 ~]$ kinit -kt /tmp/test.keytab test@TBDS-JYOKXQY8
```

3. Ranger 授权：确保 test 用户具有提交 yarn default 队列的权限。
4. 运行样例：将编译后的 jar 包上传到开发机 /tmp 目录，进入 Spark 安装目录，一般位于 /usr/local/service/spark，然后使用 spark-submit 命令提交，根据需要选择 Cluster 模式或 Client 模式。若项目依赖第三方 Jar，可以通过 --jars 参数进行指定。
5. Cluster 模式提交：应用程序参数依次为：brokers、subscribe-type、topics，具体说明如下。等 Spark Structured Streaming 任务正常运行起来后，使用 Kafka 客户端同时向 spark_structured_streaming_topic 生产一些数据。

- brokers：即 Kafka 配置 bootstrap.servers，host:port 之间使用逗号分隔。
- subscribe-type：有三种类型，Kafka 源只能指定 assign、subscribe 或 subscribePattern 中的一个。其中 assign 指定要消费的 TopicPartitions，值为 JSON 字符串，例如 {"topicA":[0,1],"topicB":[2,4]}。subscribe 指定要订阅的 Topic，值为逗号分隔的 Topic 列表。 subscribePattern 指定订阅 Topic 的模式，值为 Java 正则表达式字符串。
- topics：根据 subscribe-type 的值，采用不同的格式。

```
# 1. 准备sasl认证配置文件，内容如下，编辑后保存退出
[test@10 ~]$ vim /tmp/kafka_jaas.conf
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="./test.keytab"
    principal="test@TBDS-JYOKXQY8";
};

# 2. 以Cluster模式提交Spark Structured Streaming任务，参数依次为：brokers、subscribe-type、topics
[test@10 ~]$ /usr/local/service/spark/bin/spark-submit \
```

```
--master yarn \
--deploy-mode cluster \
--files /tmp/kafka_jaas.conf,/tmp/test.keytab,/etc/krb5.conf \
--conf spark.driver.extraJavaOptions="-Djava.security.auth.login.config=./kafka_jaas.conf -Djava.security.krb
5.conf=./krb5.conf" \
--conf spark.executor.extraJavaOptions="-Djava.security.auth.login.config=./kafka_jaas.conf" \
--class com.tencent.tbds.spark.SparkStructuredKafkaWordCountJavaExample \
SparkStructuredKafkaWordCountJavaExample-1.0.jar \
broker1:9092,broker2:9092,broker3:9092 subscribe spark_structured_streaming_topic
```

```
Log Type: stdout
Log Upload Time: 周二 12月 17 18:10:18 +0800 2024
Log Length: 546
-------------------------------------------
Batch: 0
-------------------------------------------
+-----+-----+
|value|count|
+-----+-----+
+-----+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+-----+-----+
|value|count|
+-----+-----+
|kafka|    2|
|hello|    4|
|spark|    2|
+-----+-----+
```

6. Client 模式提交：Driver sasl 认证配置文件 kafka_driver_jaas.conf（由 Yarn 客户端使用）和 kafka_jaas.conf 基本相同，除了 keyTab 处有一些差异。在 kafka_driver_jaas.conf 中，keyTab 应为 {keytab_path}/kafka.service.keytab。

```
# 1. 准备driver sasl认证配置文件，内容如下（KafkaClient用于Kafka认证，Client使用ZK认证），编辑后保存退出
[test@10 ~]$ vim /tmp/kafka_driver_jaas.conf
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/tmp/test.keytab"
    principal="test@TBDS-JYOKXQY8";
};
Client {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/tmp/test.keytab"
    principal="test@TBDS-JYOKXQY8";
};

# 2. Client模式提交Spark Structured Streaming任务，参数依次为：brokers、subscribe-type、topics
[test@10 ~]$ /usr/local/service/spark/bin/spark-submit \
--master yarn \
--deploy-mode client \
--files /tmp/kafka_jaas.conf,/tmp/test.keytab \
--conf spark.driver.extraJavaOptions="-Djava.security.auth.login.config=/tmp/kafka_driver_jaas.conf" \
--conf spark.executor.extraJavaOptions="-Djava.security.auth.login.config=./kafka_jaas.conf" \
```

--class com.tencent.tbds.spark.SparkStructuredKafkaWordCountJavaExample \
SparkStructuredKafkaWordCountJavaExample-1.0.jar \
broker1:9092,broker2:9092,broker3:9092 subscribe spark_structured_streaming_topic

```
2024-12-17 18:01:53,236 [WARN] [stream execution thread for [id = db8a4817-7112-46e6-9ae5-d56bbd8f3c9f, runId = 022d8cb3-8fb3-47a8-9ebe-1cff8512
62e2]] The configuration 'auto.offset.reset' was supplied but isn't a known config. (org.apache.kafka.clients.admin.AdminClientConfig(org.apache
.kafka.common.config.AbstractConfig.logUnused:380))
-------------------------------------------
Batch: 0
-------------------------------------------
+-----+-----+
|value|count|
+-----+-----+
+-----+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+-----+-----+
|value|count|
+-----+-----+
|kafka|    2|
|hello|    4|
|spark|    2|
+-----+-----+
```

Spark Streaming

本教程演示的是：在 Kerberos 环境下，TBDS 集群运行 Spark Streaming 作业来消费 Kafka 数据，实现统计单词个数，需要提前在集群中创建 Kafka Topic。更多说明请查看示例工程 README.md。
注：Spark Streaming 是 Spark 的上一代流处理引擎，Spark Streaming 已经不再更新，它是一个遗留项目。在 Spark 中有一个更新且更易于使用的流处理引擎，叫做 Structured Streaming，建议在流式应用中使用 Spark Structured Streaming，具体可以参考 Structured Streaming 编程指南。

2.1 准备应用程序

1. 参考【Spark 示例工程开发】，下载样例代码，选择对应迭代分支，将示例工程进行编译打包。
2. 在 SparkDirectKafkaWordCountJavaExample 示例工程中，通过使用 Spark Streaming 调用 Kafka 接口来获取单词记录，然后把单词记录分类统计，最后输出每个单词的记录数。

```
// 创建2秒批处理间隔的上下文
SparkConf sparkConf = new SparkConf().setAppName("SparkDirectKafkaWordCountJavaExample");
JavaStreamingContext jssc = new JavaStreamingContext(sparkConf, Durations.seconds(5));

Set<String> topicsSet = new HashSet<>(Arrays.asList(topics.split(",")));
Map<String, Object> kafkaParams = new HashMap<>();
kafkaParams.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, brokers);
kafkaParams.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
kafkaParams.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
kafkaParams.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);

// 设置kafka kerberos认证（如有必要）
```

```
kafkaParams.put("security.protocol", "SASL_PLAINTEXT");
kafkaParams.put("sasl.mechanism", "GSSAPI");
kafkaParams.put("sasl.kerberos.service.name", "hadoop");

// 使用brokers和topics创建direct kafka stream
JavaInputDStream<ConsumerRecord<String, String>> messages = KafkaUtils.createDirectStream(
        jssc,
        LocationStrategies.PreferConsistent(),
        ConsumerStrategies.Subscribe(topicsSet, kafkaParams));

// 获取行，将其拆分为单词，计算单词数并打印
JavaDStream<String> lines = messages.map(ConsumerRecord::value);
JavaDStream<String> words = lines.flatMap(x -> Arrays.asList(SPACE.split(x)).iterator());
JavaPairDStream<String, Integer> wordCounts = words.mapToPair(s -> new Tuple2<>(s, 1))
        .reduceByKey((i1, i2) -> i1 + i2);
wordCounts.print();

// 开始计算
jssc.start();
jssc.awaitTermination();
```

2.2 运行 Spark Streaming 作业

1. 参考【Kafka 常用命令】，在 TBDS 集群创建一个名为 spark_streaming_topic 的 topic。
2. 准备用户：获取用户认证信息，需要将用户的 keytab 文件下载到本地，然后上传至开发机。这里将 test 用户的 test.keytab 文件上传至开发机的 /tmp 目录，并进行 kerberos 认证。

```
# Kerberos认证
[test@10 ~]$ klist -kt /tmp/test.keytab
Keytab name: FILE:/tmp/test.keytab
KVNO Timestamp           Principal
---- ------------------- ------------------------------------------------------
   1 11/28/2024 22:05:22 test@TBDS-JYOKXQY8
   1 11/28/2024 22:05:22 test@TBDS-JYOKXQY8
[test@10 ~]$ kinit -kt /tmp/test.keytab test@TBDS-JYOKXQY8
```

3. Ranger 授权：确保 test 用户具有提交 yarn default 队列的权限。
4. 运行样例：将该 jar 包上传到开发机 /tmp 目录，进入 Spark 安装目录，一般位于 /usr/local/service/spark，然后使用 spark-submit 命令提交，根据需要选择 Cluster 模式或 Client 模式。若项目依赖第三方 JAR，可以通过 --jars 参数进行指定，更多参数说明参考【Spark 常用命令】。

- Cluster 模式提交：等 Spark Streaming 任务正常运行起来后，使用 Kafka 客户端同时向 spark_streaming_topic 生产一些数据，参考【Kafka 常用命令】。注意，应用程序参数依次为：brokers、groupId、topic。

```
# 1. 准备sasl认证配置文件，内容如下，编辑后保存退出
[test@10 ~]$ vim /tmp/kafka_jaas.conf
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="./test.keytab"
    principal="test@TBDS-JYOKXQY8";
};

# 2. 以Cluster模式提交Spark Streaming任务，参数依次为：brokers、groupId、topic
[test@10 ~]$ /usr/local/service/spark/bin/spark-submit \
--master yarn \
--deploy-mode cluster \
--files /tmp/kafka_jaas.conf,/tmp/test.keytab,/etc/krb5.conf \
--conf spark.driver.extraJavaOptions="-Djava.security.auth.login.config=./kafka_jaas.conf -Djava.security.krb
5.conf=./krb5.conf" \
--conf spark.executor.extraJavaOptions="-Djava.security.auth.login.config=./kafka_jaas.conf" \
--class com.tencent.tbds.spark.SparkDirectKafkaWordCountJavaExample \
SparkDirectKafkaWordCountJavaExample-1.0.jar \
broker1:9092,broker2:9092,broker3:9092 tmp_group spark_streaming_topic
```

- YARN UI 页面根据 ApplicationID、User、Name 等信息，查看对应的 Spark 任务日志。可以看到，任务日志中有输出单词统计相关信息，如下图所示。

```
Log Type: stdout
Log Upload Time: 星期四 十一月 28 22:03:33 +0800 2024
Log Length: 926
-------------------------------------------
Time: 1732802480000 ms
-------------------------------------------
(kafka,3)
(hello,6)
(spark,3)

-------------------------------------------
Time: 1732802482000 ms
-------------------------------------------
```

- Client 模式提交：注意，Driver sasl 认证配置文件 kafka_driver_jaas.conf（由 Yarn 客户端使用）和 kafka_jaas.conf 基本相同，除了 'keyTab' 处有一些差异。在 kafka_driver_jaas.conf 中，'keyTab' 应为“{keytab_path}/kafka.service.keytab”。

```
# 1. 准备driver sasl认证配置文件，内容如下（KafkaClient用于Kafka认证，Client使用ZK认证），编辑后保存退出
[test@10 ~]$ vim /tmp/kafka_driver_jaas.conf
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
```

```
    storeKey=true
    keyTab="/tmp/test.keytab"
    principal="test@TBDS-JYOKXQY8";
};
Client {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/tmp/test.keytab"
    principal="test@TBDS-JYOKXQY8";
};

# 2. Client模式提交Spark Streaming任务，参数依次为：brokers、groupId、topic
[test@10 ~]$ /usr/local/service/spark/bin/spark-submit \
--master yarn \
--deploy-mode client \
--files /tmp/kafka_jaas.conf,/tmp/test.keytab \
--conf spark.driver.extraJavaOptions="-Djava.security.auth.login.config=/tmp/kafka_driver_jaas.conf" \
--conf spark.executor.extraJavaOptions="-Djava.security.auth.login.config=./kafka_jaas.conf" \
--class com.tencent.tbds.spark.SparkDirectKafkaWordCountJavaExample \
SparkDirectKafkaWordCountJavaExample-1.0.jar \
broker1:9092,broker2:9092,broker3:9092 tmp_group spark_streaming_topic
```

- 不同于 Cluster 模式，单词统计相关信息将直接输出在控制台，如下图所示。

```
-------------------------------------------
Time: 1732806134000 ms
-------------------------------------------
(kafka,1)
(hello,2)
(spark,1)

-------------------------------------------
Time: 1732806136000 ms
-------------------------------------------

-------------------------------------------
Time: 1732806138000 ms
-------------------------------------------

^C2024-11-28 23:02:19,401 [WARN] [kafka-kerberos-refresh-thread-test@TBDS-JYOKXQY8] [Principal=test@TBDS-JYOKXQY8]: TGT renewal
thread has been interrupted and will exit. (org.apache.kafka.common.security.kerberos.KerberosLogin(org.apache.kafka.common.secu
rity.kerberos.KerberosLogin.lambda$login$0:196))
```

3. Spark SQL

本教程演示的是：Spark Hive SQL 任务，需要提前将示例工程 resources 目录下的 kv1.txt 文件放至 HDFS。更多说明请查看示例工程 README.md。

3.1 准备应用程序

1. 参考【Spark 示例工程开发】，下载样例代码，选择对应迭代分支，将示例工程进行编译打包。

2. 在 SparkHiveJavaExample 示例工程中，首先创建了一张 Hive 表，并从 HDFS 中加载数据；然后进行基本查询和聚合查询；最后通过 DataFrame 访问每列数据，并创建临时视图。

```
// 初始化SparkSession, 并开启Hive支持
SparkSession spark = SparkSession
  .builder()
  .appName("Spark Hive Java Example")
  .enableHiveSupport()
  .getOrCreate();

// 创建Hive表
spark.sql("CREATE TABLE IF NOT EXISTS tmp_src (key INT, value STRING) USING hive");
// 从HDFS加载数据
spark.sql("LOAD DATA INPATH '" + args[0] + "' INTO TABLE tmp_src");
// 基本查询
spark.sql("SELECT * FROM tmp_src").show();
// 聚合查询
spark.sql("SELECT COUNT(*) FROM tmp_src").show();

// SQL查询结果类型是Dataset<Row>，即DataFrame，并且支持所有常规函数
Dataset<Row> sqlDF = spark.sql("SELECT key, value FROM tmp_src WHERE key < 10 ORDER BY key");
// DataFrame中的每个元素都是Row类型，可以通过序号访问每列
Dataset<String> stringsDS = sqlDF.map(
    (MapFunction<Row, String>) row -> "Key: " + row.get(0) + ", Value: " + row.get(1),
    Encoders.STRING());
stringsDS.show();

// 可以使用DataFrame在SparkSession中创建临时视图
List<Record> records = new ArrayList<>();
for (int key = 1; key < 100; key++) {
  Record record = new Record();
  record.setKey(key);
  record.setValue("val_" + key);
  records.add(record);
}
Dataset<Row> recordsDF = spark.createDataFrame(records, Record.class);
recordsDF.createOrReplaceTempView("tmp_records");
// 可以将DataFrame的数据与存储在Hive中的数据进行join
spark.sql("SELECT * FROM tmp_records r JOIN tmp_src s ON r.key = s.key").show();

// 关闭SparkSession
spark.stop();
```

3.2 运行 Spark SQL 作业

1. 准备用户：获取用户认证信息，需要将用户的 keytab 文件下载到本地，然后上传至开发机。这里将 test 用户的

test.keytab 文件上传至开发机的 /tmp 目录，并进行 kerberos 认证。

2. 准备文件：将 resources 目录下的 kv1.txt 文件上传至开发机 /tmp 目录，然后将其上传至 HDFS /tmp 目录。

```
# 1. Kerberos认证（Simple认证跳过该步）
[test@10 ~]$ klist -kt /tmp/test.keytab
Keytab name: FILE:/tmp/test.keytab
KVNO Timestamp           Principal
---- ------------------- ------------------------------------------------------
   1 11/28/2024 22:05:22 test@TBDS-JYOKXQY8
   1 11/28/2024 22:05:22 test@TBDS-JYOKXQY8
[test@10 ~]$ kinit -kt /tmp/test.keytab test@TBDS-JYOKXQY8

# 2.上传kv1.txt文件至HDFS
[test@10 ~]$ /usr/local/service/hadoop/bin/hdfs dfs -put /tmp/kv1.txt /tmp
```

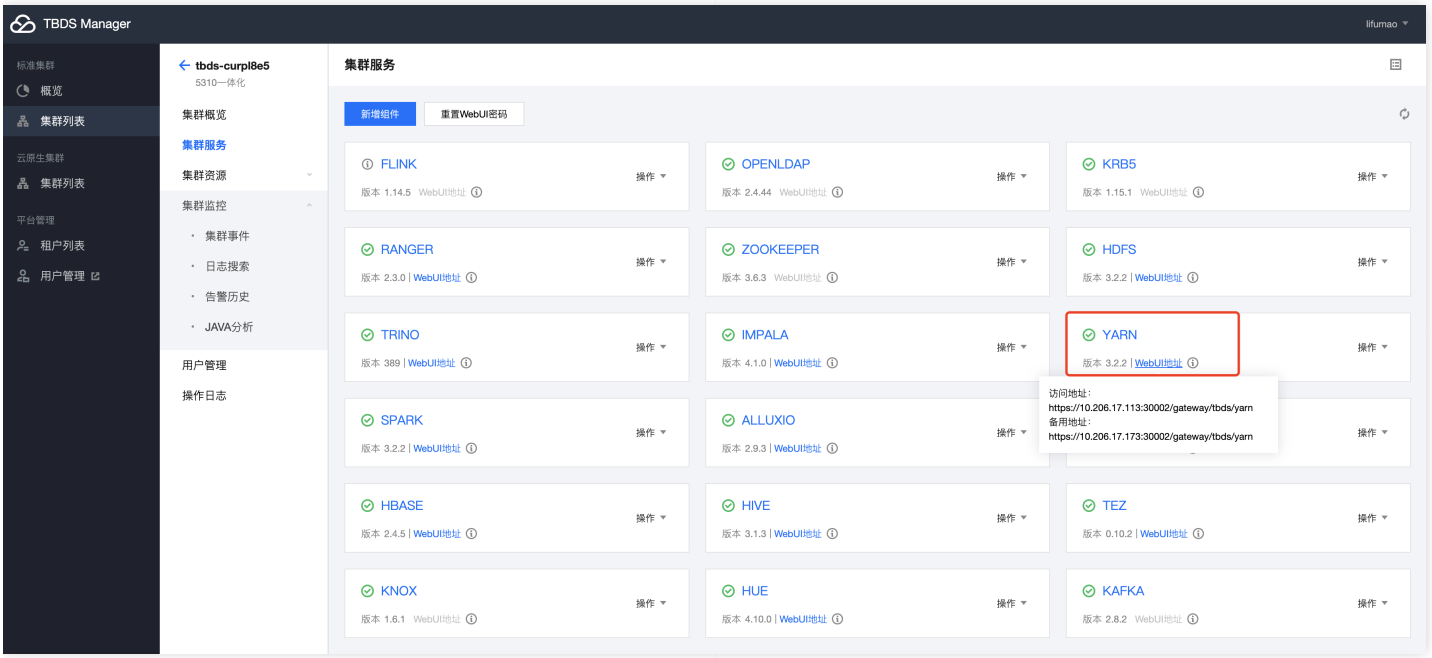
3. Ranger 授权：确保 test 用户具有提交 yarn default 队列的权限。
4. 运行样例：将编译后的 jar 包上传到开发机 /tmp 目录，进入 Spark 安装目录，一般位于 /usr/local/service/spark，然后使用 spark-submit 命令提交，根据需要选择 Cluster 模式或 Client 模式。若项目依赖第三方 Jar，可以通过 --jars 参数进行指定。

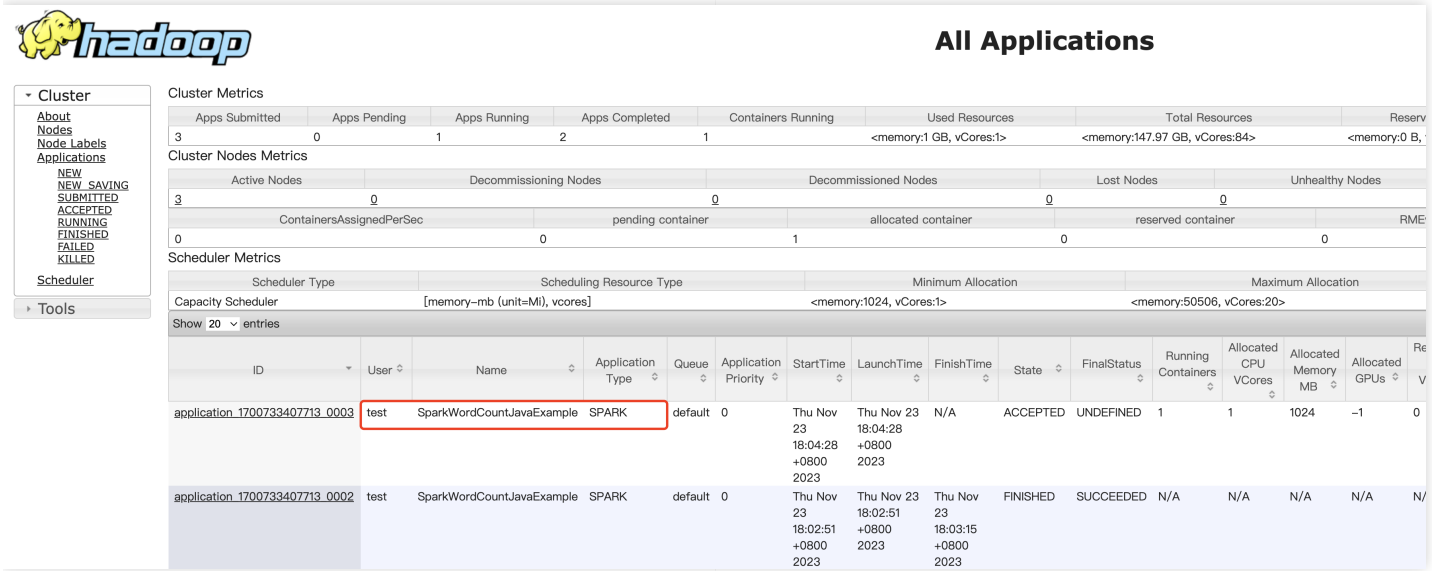
```
[test@10 ~]$ cd /usr/local/service/spark
[test@10 spark]$ bin/spark-submit \
--master yarn \
--deploy-mode client \
--queue default \
--keytab /tmp/test.keytab \
--principal test@TBDS-JYOKXQY8 \
--class com.tencent.tbds.spark.SparkHiveJavaExample \
/tmp/SparkHiveJavaExample-1.0.jar hdfs:///tmp/kv1.txt
```

常见问题

Spark on YARN 如何查看日志

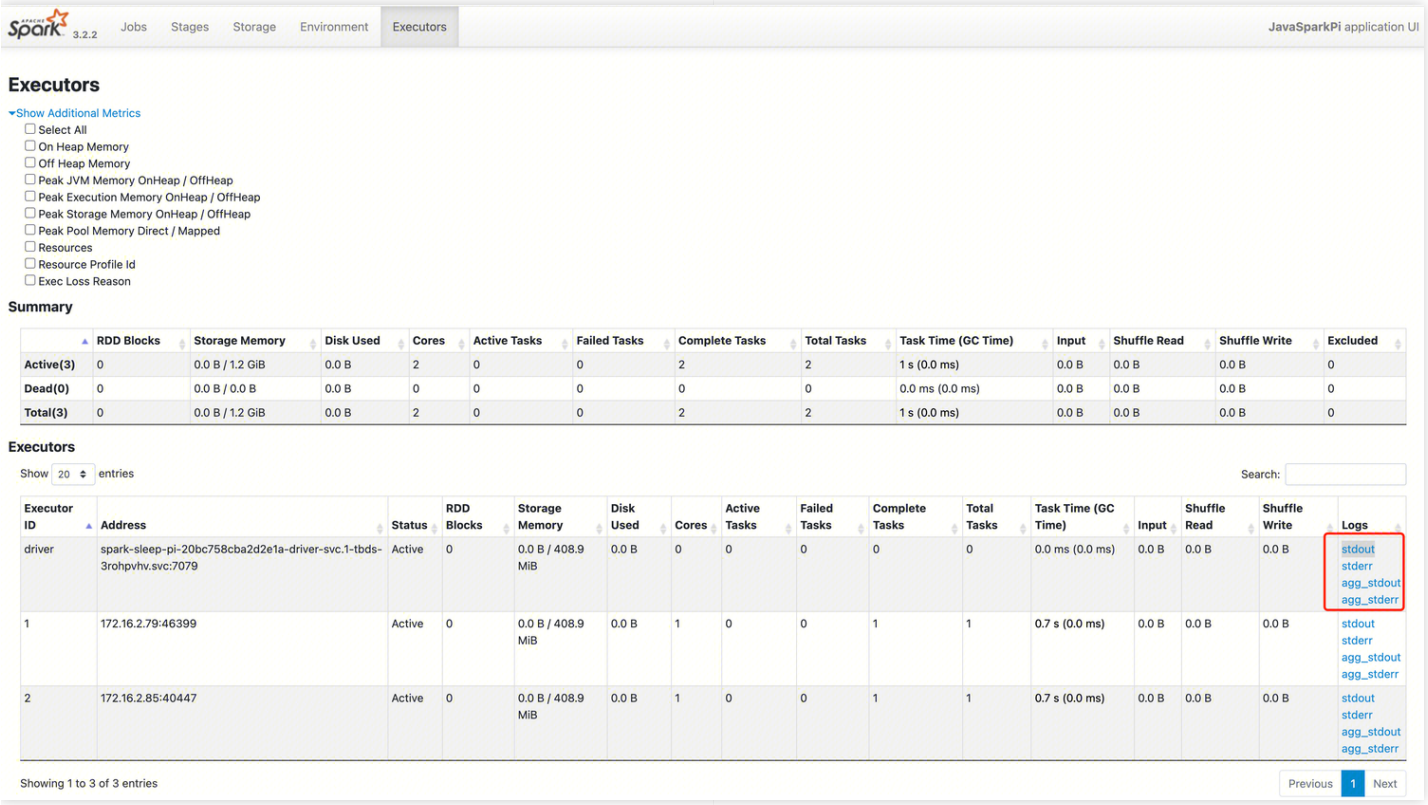
登录 TBDS Manager 管控平台，点击“集群列表”，选择 Spark 程序运行所对应的集群。点击“集群服务”，选择 YARN 服务，点击 WebUI 地址跳转至 YARN UI 界面。YARN UI 页面根据 ApplicationID、User、Name 等信息，找到对应的 Spark 任务。Spark 任务的 ApplicationID 在日志级别为 INFO 及以下会输出至控制台，也可以根据任务名等信息进行查找对应的 Spark 任务，然后点击 ApplicationID 链接，最后点击 Logs 查看日志。





Spark on K8s 如何查看日志

如下图所示，Spark History Server 页面 Driver、Executor 将返回以下 4 个日志跳转链接 stdout、stderr、agg_stdout、agg_stderr。其中前两个链接只能查看运行中任务的日志，一旦任务运行结束，再点击查看，将报 500 错误，属于正常现象；此时可以点击后两个链接，下载运行结束后任务的全部日志，同样，若运行过程中点击后两个链接，也可能报 500 错误，属于正常现象。另外，为防止日志文件过大，任务运行过程中，容器中的日志会定期上传。若日志大于 16M，将上传 stdout、stderr 日志至 HDFS 等底层存储，然后清空该日志，因此有时 stdout、stderr 只显示了部分运行中的日志，其余部分日志可点击 agg_stdout、agg_stderr 查看。



Spark 如何减少小文件产生

小文件通常是指文件大小显著小于 HDFS Block 块（默认 128MB）的文件，小文件过多会给 HDFS 带来严重的性能瓶颈（主要表现在 NameNode 节点元数据的管理以及客户端的元数据访问请求），并且对用户作业的稳定性和集群数据的维护也会带来很大的挑战。

对于 Spark 来说，由于静态分区写入，Hive 表分区只有 1 个，而动态分区写入，Hive 表分区字段一般是固定的业务字段，因此减少最后一个 Stage 的 RDD 分区数是 Spark 小文件合并的主要手段。然而，RDD 分区数的减少意味着任务并行度的减少，因此需要权衡小文件的个数与任务的并行度。

1. 降低并行度：调整 spark.sql.shuffle.partitions 和 spark.default.parallelism，默认值 200，该参数只在 shuffle 时才会生效，需要根据集群大小和数据量来调整，从小文件角度，可设置为输出数据量 / (128M ~ 1G)

作为初始值；从资源利用角度，可设置为核心数的 2-3 倍作为初始值，然后根据实际效果进行调整。

2. 重分区：Spark RDD 使用 coalesce 与 repartition 算子，两者均是将 RDD 中的数据进行重新分区，repartition(numPartitions) 语义与 coalesce(numPartitions, true) 一致（第二个参数表示是否 shuffle），即 repartition 一定会进行 shuffle。而 coalesce 将相邻的分区直接合并在一起，形成的依赖关系是多对一的窄依赖，不会触发数据的 shuffle，因此性能更好。在不进行 shuffle 的情况下，coalesce 不能将一个分区拆分为多份，且当 RDD 中不同分区中的数据量差别较大时，直接合并容易造成数据倾斜。为了增加分区或解决数据倾斜问题，可以指定 shuffle=true 将数据打乱。
   自 Spark 2.4 开始，Spark SQL 可使用 Coalesce Hints 控制输出文件的数量，就像 Dataset API 中的 coalesce、repartition 和 repartitionByRange 一样，它们可用于性能调整和减少输出文件的数量。其中，COALESCE 只有一个分区数作为参数；REPARTITION 参数包括分区数、列或两者。
   REPARTITION_BY_RANGE 必须包含列名，分区数是可选参数。

   ```
   INSERT ... SELECT /*+ COALESCE(3) */ * FROM t
   INSERT ... SELECT /*+ REPARTITION(3) */ * FROM t
   INSERT ... SELECT /*+ REPARTITION(c) */ * FROM t
   INSERT ... SELECT /*+ REPARTITION(3, c) */ * FROM t
   INSERT ... SELECT /*+ REPARTITION_BY_RANGE(c) */ * FROM t
   INSERT ... SELECT /*+ REPARTITION_BY_RANGE(3, c) */ * FROM t
   ```

3. distribute by：通过控制 map 输出结果的分发，相同字段的 map 输出会发到一个 reduce 节点处理，如果字段均匀随机，能保证每个分区的数量基本一致，用法一般为：

   ```
   INSERT ... SELECT ... FROM ... DISTRIBUTE BY 分区, ceil(rand() * 15))
   ```

4. 自适应查询 AQE：通过设置 Spark shuffle partition 的上限和下限对不同作业不同阶段的 reduce 个数进行动态调整，也可以通过参数对 join 和数据倾斜问题进行优化，针对小文件合并，主要设置以下几个参数。需要注意的是此方法并不会严格保证按每个文件 128M 进行合并，也不能指定最终合成多少个文件。

   ```
   -- 是否开启AQE，默认false，Spark 3.2默认true
   set spark.sql.adaptive.enabled=true;

   -- 当启用AQE且该值为true时，Spark将根据目标大小（由spark.sql.adaptive.advisoryPartitionSizeInBytes指定）合并连续的Shuffle分区，以避免过多的小任务，默认true
   set spark.sql.adaptive.coalescePartitions.enabled=true;

   -- 自适应优化期间Shuffle分区的建议大小，默认64M，Spark 3.0使用spark.sql.adaptive.advisoryPartitionSizeInBytes
   set spark.sql.adaptive.shuffle.targetPostShuffleInputSize=128M;
   set spark.sql.adaptive.advisoryPartitionSizeInBytes=128M;
   ```

```
-- 合并后，建议（但不保证）的Shuffle分区的最小数量，Spark 3.2使用spark.sql.adaptive.coalescePartitions.minPartitionSize，合并后Shuffle分区的最小大小，其值最多为spark.sql.adaptive.advisoryPartitionSizeInBytes的20%，默认1M
set spark.sql.adaptive.coalescePartitions.minPartitionNum=1;
set spark.sql.adaptive.coalescePartitions.minPartitionSize=10M;

-- 仅Spark 3.2，该值为true（默认）时，Spark在合并Shuffle分区时会忽略spark.sql.adaptive.advisoryPartitionSizeInBytes指定的目标大小，而只尊重spark.sql.adaptive.coalescePartitions.minPartitionSize指定的最小分区大小，以最大化并行性。这是为了避免在启用AQE执行时出现性能下降
set spark.sql.adaptive.coalescePartitions.parallelismFirst=false;
```

5. 集成 Kyuubi 插件：注入自定义规则来添加额外的 Shuffle，然后通过 AQE 合并小文件。需要将 Kyuubi Spark SQL 扩展 kyuubi-extension-spark-*.jar（一般位于 /usr/local/service/kyuubi/extension 目录）放入 $SPARK_HOME/jars 目录，并在 $SPARK_HOME/conf/spark-defaults.conf 中添加配置 spark.sql.extensions=org.apache.kyuubi.sql.KyuubiSparkSQLExtension（新参数值在原来值的基础上追加，逗号分隔）。如果配置了 spark.yarn.archive 参数，还需要将 $SPARK_HOME/jars 目录重新打包，并上传至 HDFS 对应目录（默认为 /tbds/spark/jars/spark_jars.tar.gz），打包压缩命令为： tar -czf /usr/local/service/spark/spark_jars.tar.gz -C /usr/local/service/spark/jars .

```
-- 在查询计划的顶部添加repartition节点，默认true
set spark.sql.optimizer.insertRepartitionBeforeWrite.enabled=true

-- 当设置为true时，即使原始计划没有Shuffle操作，也会添加repartition，默认false
set spark.sql.optimizer.insertRepartitionBeforeWriteIfNoShuffle.enabled=true;

-- AQE相关参数
set spark.sql.adaptive.advisoryPartitionSizeInBytes=128M;
set spark.sql.adaptive.coalescePartitions.minPartitionSize=10M;
```

Spark 启动时频繁报 HBase错误

由于 Spark 启动时需要向 HBase 获取 Token，因此如果获取 Token 失败（如 HBase服务异常），Spark 将不断尝试，报错信息类似：Call exception, tries=6, retries=16, started=4731 ms ago, cancelled=false, msg=org.apache.hadoop.hbase.ipc.ServerNotRunningYetException。如果 Spark 不需要访问 HBase，可以在提交任务时，设置参数：--conf spark.security.credentials.hbase.enabled=false。

Spark 如何解决内存不足

Spark 任务运行内存不足，常见报错包括：

1. java.lang.OutOfMemoryError: Java heap space
2. java.lang.OutOfMemoryError: GC overhead limit exceeded
3. Cannot allocate memory
4. The job has been killed by "OOM Killer", please check your job's memory usage
   可尝试调整如下参数：
5. spark.driver.memory：设置 Driver 进程的堆内存大小，通常与 spark.driver.cores 保持 1:4 设置即可。当使用 collect 等算子将数据收集到 Driver 端，或抛出 java.lang.OutOfMemoryError 异常时，可以适当调大该值。
6. spark.driver.memoryOverhead：Driver 进程的非堆内存大小，默认大小为 spark.driver.memory * 0.1，最小 384 MB。通常不需要额外设置，当 Driver 日志出现 Cannot allocate memory 报错，可以适当调大该值。
7. spark.executor.memory：Executor 进程的堆内存大小，通常与 spark.executor.cores 保持 1:4 设置即可。当抛出 java.lang.OutOfMemoryError 异常时，可以适当调大该值。
8. spark.executor.memoryOverhead：为每个 Executor 分配的额外内存量，主要用于 JVM 自身、字符串、NIO Buffer 等开销。默认大小为spark.executor.memory * 0.1，最小 384 MB。通常不需要额外设置，当 Executor 日志出现 Cannot allocate memory 或 OOM Killer 报错时，可以适当调大该值。

Ranger 策略过多导致 Driver OOM

尝试去掉 spark-defaults.conf 文件中 spark.sql.extensions 配置参数值 org.apache.kyuubi.plugin.spark.authz.ranger.RangerSparkExtension，若 Spark Driver 正常运行，则确认是由于 Ranger 策略过多导致 Driver OOM。
由于 Ranger 插件需要将策略加载到 Driver 内存中，所以当 Ranger 策略数量增大时，需要对应调大 Spark Driver Memory（参数：spark.driver.memory），建议每增加 10w 条策略增加约 500M 内存。

Spark 如何开启审计日志

Spark 审计日志默认关闭，若要开启，将 ranger-spark-audit.xml 配置文件中的参数 xasecure.audit.is.enabled 的对应值修改为 true。

Spark IPv6 运行 Python 报错

若报错信息为：RuntimeError: could not open socket: ["tried to connect to ('127.0.0.1', 44373), but an error

occurred: [Errno 111] Connection refused"]，可尝试在运行 Python 任务前，导入环境变量：export SPARK_PREFER_IPV6=true

Spark Client 模式提交报错，Cluster 模式正常

若报错信息为：java.net.NoRouteToHostException: No Route to Host（没有到主机的路由），可尝试检查集群外客户端所在节点的防火墙，iptables 防火墙使用命令 service iptables status ；firewall 防火墙使用 systemctl status firewalld 。由于 Client 模式 Driver 位于客户端所在节点，而 Driver 需要与集群内的 Executor 进行网络通信，因此客户端所在节点需要向集群内所有节点关闭防火墙。

Spark Drop Hive 表不删除底层 HDFS 数据

若要删除底层 HDFS 数据，SQL 需要加上 PURGE 关键字，例如 drop table test_1 purge 。或者提交 Spark 任务时指定参数 --conf spark.sql.iceberg.drop-table-rollback=true 。否则只会删除元数据，而不会删除底层 HDFS 数据，这是由于将 Iceberg 作为默认 Catalog 导致的。

Spark 如何对接 RSS

若所有 Spark 任务均使用 RSS，则在 TM 页面修改【经典集群】Spark 组件 spark-defaults.conf 配置文件，新增如下参数，无需重启服务。
若只是单个 Spark 任务使用 RSS，可在提交 Spark 任务时，直接通过 --conf 指定如下参数（coordinatorIp 替换为具体 IP），例如：/usr/local/service/spark/bin/spark-sql --conf
spark.serializer=org.apache.spark.serializer.KryoSerializer --conf
spark.shuffle.manager=org.apache.spark.shuffle.RssShuffleManager --conf
spark.rss.coordinator.quorum=:19999,:19999 --conf
spark.shuffle.sort.io.plugin.class=org.apache.spark.shuffle.RssShuffleDataIo 。
更多详情参考：【运维管理】-> 【运维管理指南】->【Uniffle运维】

```
# Uniffle 通过网络传输序列化的洗牌数据，因此应使用支持序列化对象重定位的序列化器（spark-defaults.conf默认已设置，因此可不设置）
spark.serializer org.apache.spark.serializer.KryoSerializer

# 开启 RssShuffleManager
spark.shuffle.manager org.apache.spark.shuffle.RssShuffleManager

# 指定 Coordinator 地址（coordinatorIp替换为具体IP）
```

spark.rss.coordinator.quorum <coordinatorIp1>:19999,<coordinatorIp2>:19999

开启 RSS 时，使用 Spark 动态分配（默认开启），仅 Spark 3.5（Spark 3.4 及以下，需要更新补丁，暂不支持）
spark.shuffle.sort.io.plugin.class org.apache.spark.shuffle.RssShuffleDataIo

Hive开发

概述

Hive是建立在Hadoop上的数据仓库基础架构。它提供了一系列的工具，可以用来进行数据提取、转化和加载（ETL），这是一种可以存储、查询和分析存储在Hadoop中的大规模数据的机制。Hive定义了一种简单的类SQL查询语言，称为HQL（Hive Query Language），它允许熟悉SQL的用户查询数据。Hive的数据计算依赖于MapReduce、Spark和Tez。
使用新的执行引擎Tez代替原先的MapReduce，性能有了显著提升。Tez可以将多个有依赖的作业转换为一个作业（这样只需写一次HDFS，且中间节点较少），从而大大提升DAG（有向无环图）作业的性能。

Hive主要特点如下：

- 海量结构化数据分析汇总：Hive能够处理和分析存储在Hadoop中的大规模结构化数据。
- 将复杂的MapReduce编写任务简化为SQL语句：用户可以使用类似SQL的HQL来编写查询，而不需要编写复杂的MapReduce代码。
- 灵活的数据存储格式：Hive支持多种数据存储格式，包括JSON、CSV、TEXTFILE、RCFILE、SEQUENCEFILE和ORC（Optimized Row Columnar）。

Hive体系结构：

- 用户接口：
    - Beeline：Beeline是Hive的推荐CLI工具，基于JDBC，提供了更好的用户体验和更强的功能。它可以连接到HiveServer2，支持多用户并发访问。
    - CLI（Command Line Interface）：传统的CLI工具，直接与Hive交互，但在最新版本中，Beeline逐渐取代了CLI。
    - Client：Hive的客户端，用户通过Client连接至HiveServer2。在启动Client模式时，需要指定HiveServer2所在节点，并在该节点启动HiveServer2。
    - WebUI：通过浏览器访问Hive的用户接口。
    - JDBC/ODBC：通过JDBC或ODBC驱动程序，用户可以使用各种BI工具和应用程序连接到Hive。
- 元数据存储：Hive将元数据存储在关系型数据库中，如MySQL、PostgreSQL等。元数据包括表的模式、分区信息、列的数据类型、表和列的统计信息等。元数据存储是Hive架构的关键部分，它使得Hive能够快速访问和管理数据。
- 执行引擎：Hive的执行引擎负责将HQL查询转换为底层的MapReduce、Tez或Spark作业，并在Hadoop集群上执行这些作业。执行引擎优化查询计划，选择最佳的执行路径，以提高查询性能。

- 编译器：Hive编译器将HQL查询解析为抽象语法树（AST），然后将AST转换为逻辑计划。逻辑计划经过优化后，生成物理计划，最终转换为可执行的MapReduce、Tez或Spark作业。
- 优化器：Hive优化器对逻辑计划进行优化，包括谓词下推、列裁剪、分区修剪等优化技术，以提高查询性能。
- 存储和计算：Hive依赖Hadoop的分布式存储和计算能力。数据存储在HDFS（Hadoop Distributed File System）中，计算任务由MapReduce、Tez或Spark执行。

Hive主要组件：

Hive作为一个基于HDFS和MapReduce架构的数据仓库，其主要能力是通过对HQL（Hive Query Language）编译和解析，生成并执行相应的MapReduce任务或者HDFS操作。随着Hive的发展，最新版本引入了更多的执行引擎（如Tez和Spark）和改进的用户接口（如Beeline），进一步提升了性能和用户体验。

- Metastore：Metastore负责对表、列和分区等的元数据进行读写及更新操作。元数据存储在关系型数据库中，如MySQL、PostgreSQL等。Metastore是Hive架构的关键部分，使得Hive能够快速访问和管理数据。
- Driver：Driver管理HQL执行的生命周期，并贯穿Hive任务的整个执行期间。它负责接收用户的查询请求，协调各个组件的工作，最终返回查询结果。
- Compiler：Compiler负责将HQL编译并转化为一系列相互依赖的MapReduce、Tez或Spark任务。编译过程包括语法解析、语义分析、逻辑计划生成和物理计划生成。
- Optimizer：Optimizer是优化器，分为逻辑优化器和物理优化器。逻辑优化器对HQL生成的执行计划进行优化，如谓词下推、列裁剪、分区修剪等。物理优化器对生成的MapReduce、Tez或Spark任务进行优化，以提高执行效率。
- Executor：Executor按照任务的依赖关系分别执行MapReduce、Tez或Spark任务。它负责调度和监控任务的执行，确保任务按计划完成。
- ThriftServer：ThriftServer提供Thrift接口，作为JDBC和ODBC的服务端，并将Hive与其他应用程序集成起来。它支持多用户并发访问，提供了高效的查询服务。
- Clients：Beeline：Beeline是Hive推荐的CLI工具，基于JDBC，提供了更好的用户体验和更强的功能。它可以连接到HiveServer2，支持多用户并发访问。WebUI：通过浏览器访问Hive的用户接口，提供了图形化的查询和管理功能。JDBC/ODBC：通过JDBC或ODBC驱动程序，用户可以使用各种BI工具和应用程序连接到Hive。

示例工程开发

环境准备

开发环境准备

准备项	说明
安装JDK	JDK 8、JDK11 ，推荐使用 konaJDK，下载地址
安装和配置 IDE	按需选择，比如 IntelliJ IDEA 或 Eclipse，示例使用IDEA
安装 Maven	开发环境基础配置，负责构建 Java 应用程序
Maven 配置准备	如果需要本地调试，需要参考 6.开发环境准备 配置 Maven settings.xml，推荐 Maven 3.6.3，下载地址

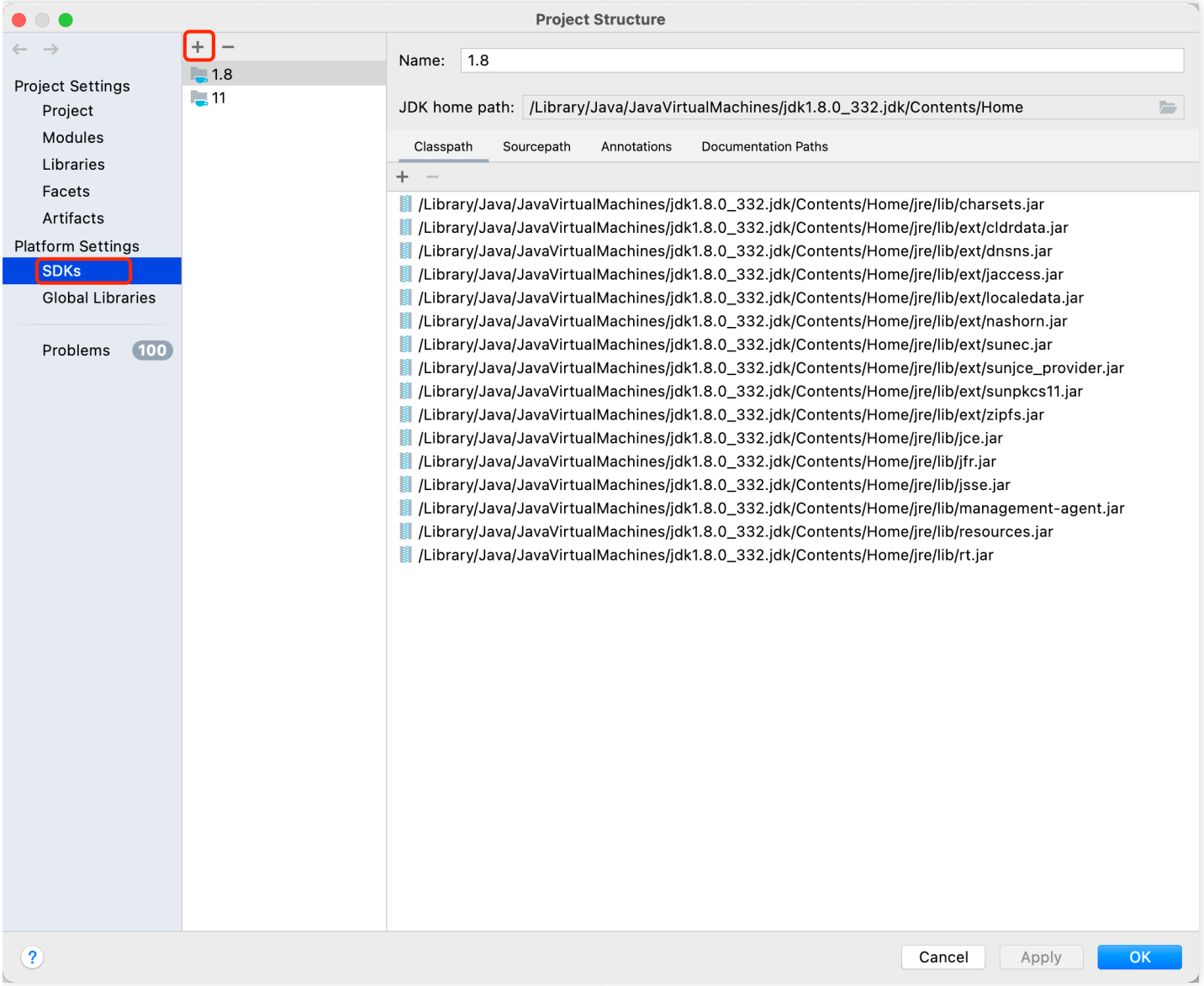
导入示例工程代码

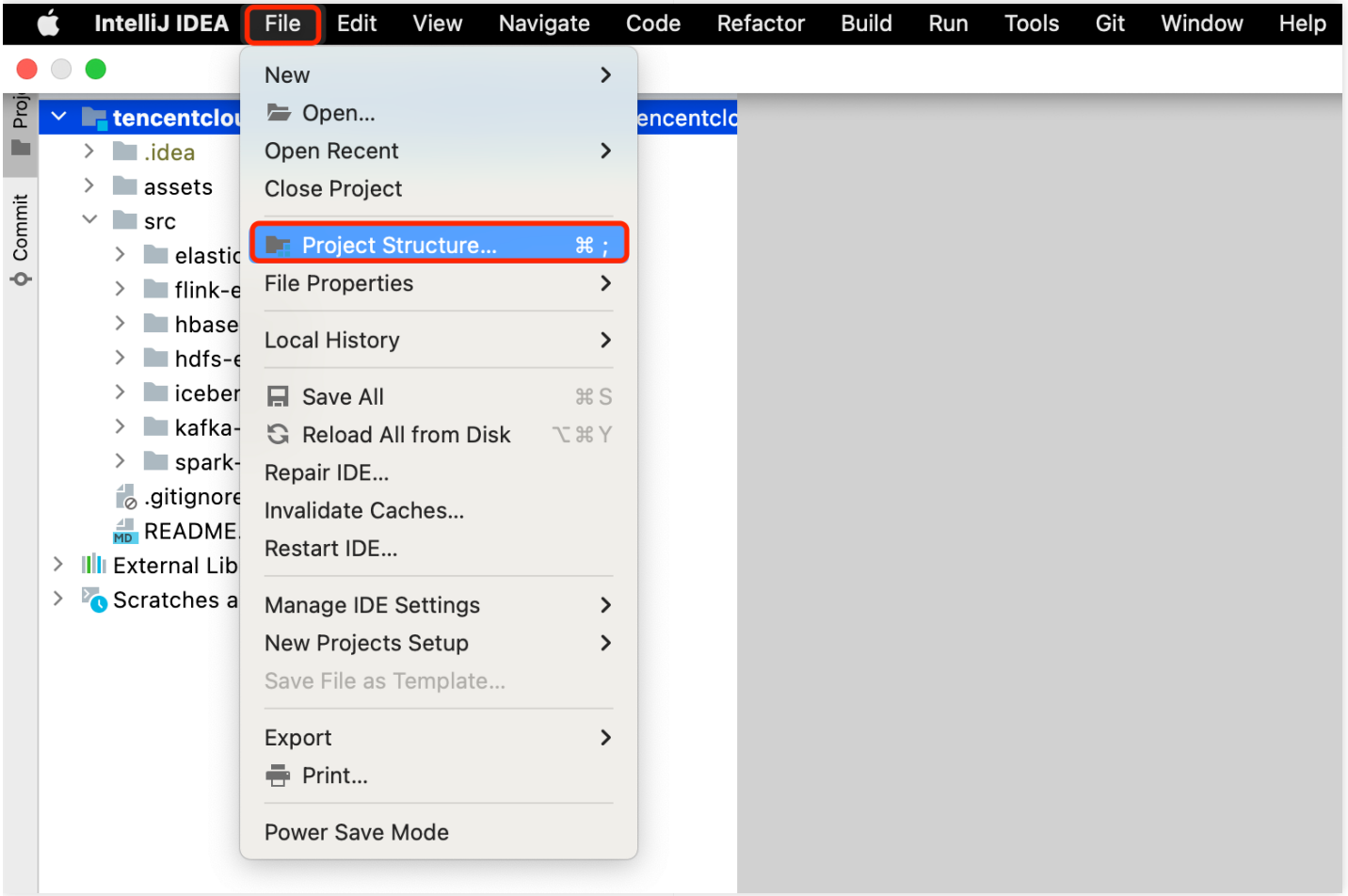
以下以 IntelliJ IDEA 举例，将示例工程代码导入进行说明。
2.1 下载样例代码：https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master

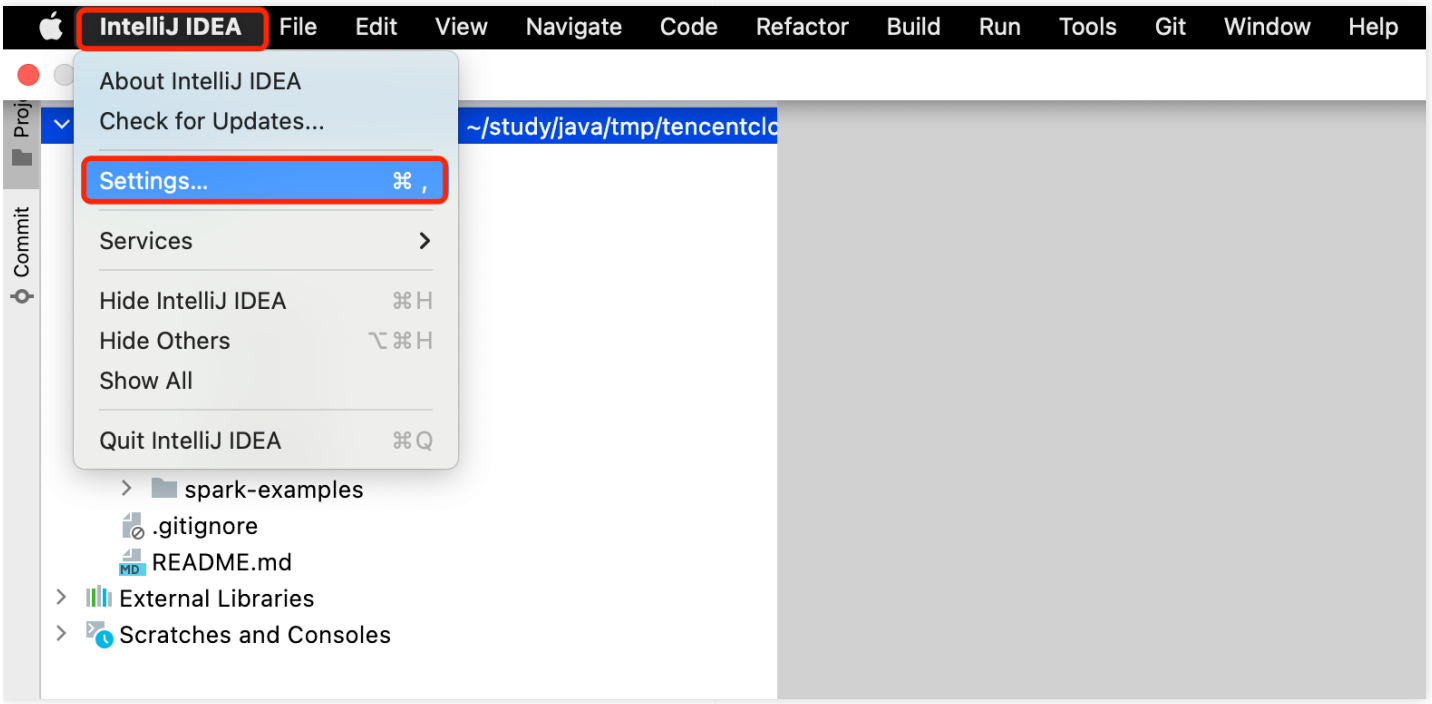
克隆或者直接下载master代码都可以

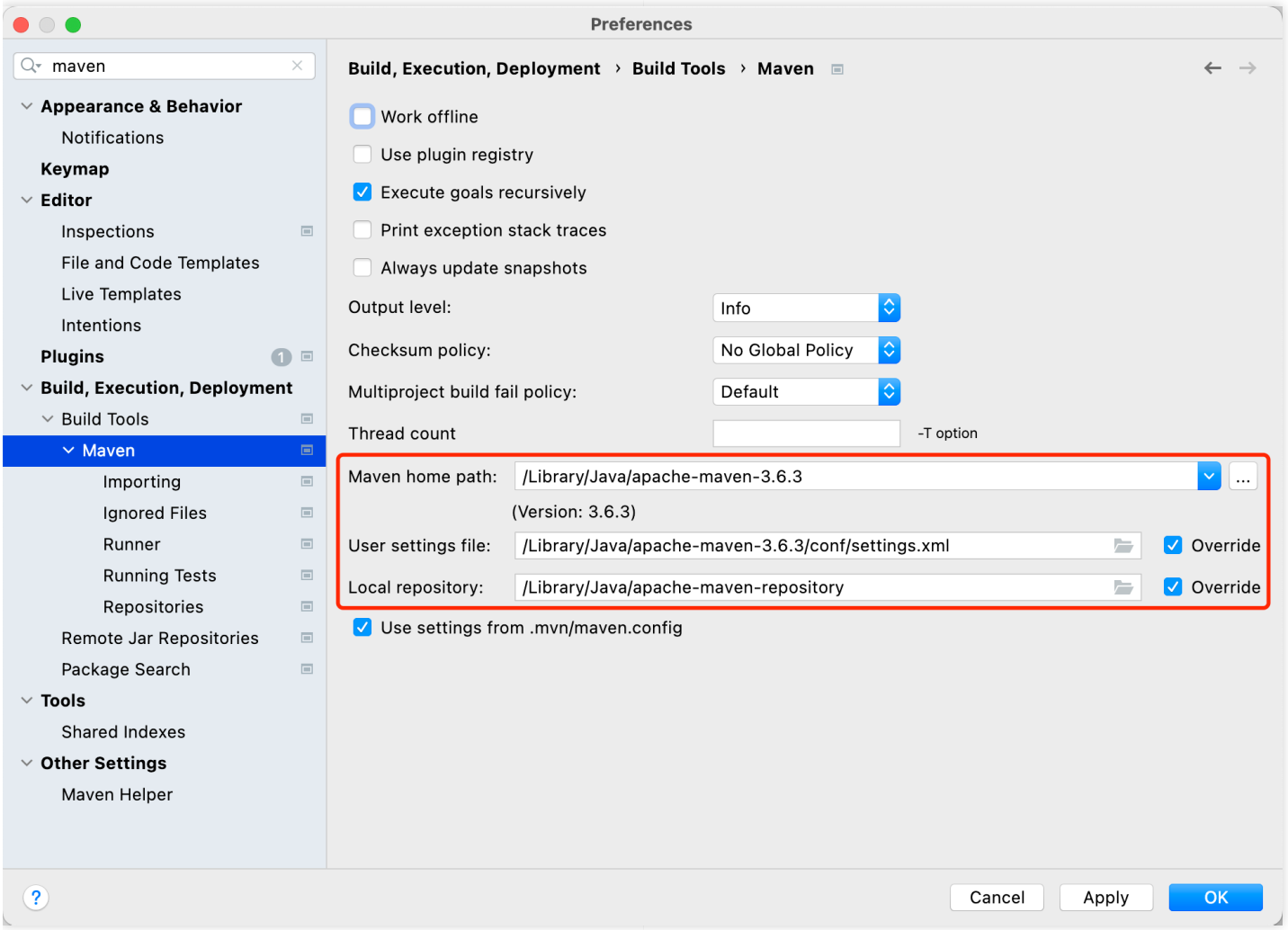
git clone https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples

2.2 导入项目，然后选择JDK、MAVEN和settings文件。


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
100
1.8
11
Name:
1.8
JDK home path:
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home
Classpath
Sourcepath
Annotations
Documentation Paths
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/charsets.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/cldrdata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/dnsns.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/jaccess.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/localedata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/nashorn.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunec.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunjce_provider.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunpkcs11.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/zipfs.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jce.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jfr.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jsse.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/management-agent.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/resources.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/rt.jar
Cancel
Apply
OK


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
New
Open...
Open Recent
Close Project
Project Structure... ⌘ ;
File Properties
Local History
Save All ⌘S
Reload All from Disk ⌥⌘Y
Repair IDE...
Invalidate Caches...
Restart IDE...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
.idea
assets
src
.gitignore
README
External Lib
Scratches a
Proj
Commit


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
About IntelliJ IDEA
Check for Updates...
Settings... ⌘ ,
Services
Hide IntelliJ IDEA ⌘H
Hide Others ⌥⌘H
Show All
Quit IntelliJ IDEA ⌘Q
~/study/java/tmp/tencentclo
spark-examples
.gitignore
README.md
External Libraries
Scratches and Consoles
Proj
Commit



样例代码

功能说明

演示的实例是样例代码通过JDBC的方式连接和访问Hive引擎

POM 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/mave
```

```
n-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.tencent.tbds</groupId>
    <artifactId>HiveExample</artifactId>
    <version>1.0-SNAPSHOT</version>
    <properties>
        <hadoop.version>3.2.2-TBDS-5.3.1.3</hadoop.version>
        <hive.version>3.1.3-TBDS-5.3.1.3</hive.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-auth</artifactId>
            <version>${hadoop.version}</version>
            <exclusions>
                <exclusion>
                    <artifactId>log4j-1.2-api</artifactId>
                    <groupId>org.apache.logging.log4j</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>log4j-slf4j-impl</artifactId>
                    <groupId>org.apache.logging.log4j</groupId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-common</artifactId>
            <version>${hive.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-shims</artifactId>
            <version>${hive.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>${hadoop.version}</version>
            <exclusions>
                <exclusion>
                    <artifactId>log4j-1.2-api</artifactId>
                    <groupId>org.apache.logging.log4j</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>log4j-slf4j-impl</artifactId>
                    <groupId>org.apache.logging.log4j</groupId>
```

```
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-jdbc</artifactId>
    <version>${hive.version}</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>*</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.apache.hbase</groupId>
            <artifactId>*</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
        <exclusion>
            <groupId>log4j</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
    <!--<scope>compile</scope>-->
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.32</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.17.2</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.17.2</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.17.2</version>
```

```
        </dependency>

    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>2.3</version>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>com.tencent.tbds.HiveExample</mainClass>
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
                <executions>
                    <execution>
                        <id>make-assemble</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>6</source>
                    <target>6</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

示例代码

支持HS2直连；zk方式连接。

支持simple认证，krbs认证。

```
public class HiveExample {
    public static final org.slf4j.Logger logger = LoggerFactory.getLogger(HiveExample.class.getName());
    private static final String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";

    private static final String ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME = "Client";
    private static final String ZOOKEEPER_SERVER_PRINCIPAL_KEY = "zookeeper.server.principal";
    private static String ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL = null;

    private static Configuration CONF = null;
    private static String KRB5_FILE = null;
    private static String USER_NAME = null;
    private static String USER_KEYTAB_FILE = null;

    /* zookeeper节点ip和端口列表 */
    private static Boolean isZookeeper= false;
    private static String zkQuorum = null;
    private static String auth = null;
    private static String sasl_qop = null;
    private static String zooKeeperNamespace = null;
    private static String serviceDiscoveryMode = null;
    private static String principal = null;
    private static String AUTH_HOST_NAME = null;
    private static String host = null;
    private static String port = null;

    /**
     * Get user realm process
     */
    public static String getUserRealm() {
        String serverRealm = System.getProperty("SERVER_REALM");
        if (serverRealm != null && serverRealm != "") {
            AUTH_HOST_NAME = "hadoop." + serverRealm.toLowerCase();
        } else {
            serverRealm = KerberosUtil.getKrb5DomainRealm();
            if (serverRealm != null && serverRealm != "") {
                AUTH_HOST_NAME = "hadoop." + serverRealm.toLowerCase();
            } else {
                AUTH_HOST_NAME = "hadoop";
            }
        }
        return AUTH_HOST_NAME;
    }

    private static void init() throws IOException {
```

```
        CONF = new Configuration();

        Properties clientInfo = null;
        InputStream fileInputStream = null;
        try {
            clientInfo = new Properties();
            String hiveclientProp = System.getProperty("config.file", "hiveclient.properties");
            File propertiesFile = new File(hiveclientProp);
            fileInputStream = new FileInputStream(propertiesFile);
            clientInfo.load(fileInputStream);
        } catch (IOException e) {
            throw new IOException(e);
        } finally {
            if (fileInputStream != null) {
                fileInputStream.close();
                fileInputStream = null;
            }
        }
        zkQuorum = clientInfo.getProperty("zk.quorum");
        auth = clientInfo.getProperty("auth");
        sasl_qop = clientInfo.getProperty("sasl.qop");
        zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
        serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
        principal = clientInfo.getProperty("principal");
        host = clientInfo.getProperty("host");
        port = clientInfo.getProperty("port");
        isZookeeper = Boolean.valueOf(clientInfo.getProperty("isZookeeper"));
        // 设置新建用户的USER_NAME，其中"xxx"指代之前创建的用户名，例如创建的用户为user，则USER_N
AME为user
        USER_NAME = clientInfo.getProperty("user");

        if ("KERBEROS".equalsIgnoreCase(auth)) {
            // 设置客户端的keytab和krb5文件路径
            USER_KEYTAB_FILE = clientInfo.getProperty("keytab");
            KRB5_FILE = System.getProperty("java.security.krb5.conf", "krb5.conf");
            System.setProperty("java.security.krb5.conf", KRB5_FILE);
            ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL = "zookeeper/" + getUserRealm();
            System.setProperty(ZOOKEEPER_SERVER_PRINCIPAL_KEY, ZOOKEEPER_DEFAULT_SERVER_PRIN
CIPAL);
        }
        System.setProperty("javax.security.auth.useSubjectCredsOnly", "false");
    }

    public static void main(String[] args) throws ClassNotFoundException, SQLException, IOException {
        logger.info("Starting HiveExample application...");
        // 参数初始化
        init();
```

```
        // 定义HQL，HQL为单条语句，不能包含";"
        String[] sqls = {
                "create table if not exists hive_test (id int, name string) row format delimited fields terminate
d by ',' ",
                "SELECT * FROM hive_test",
                "DROP TABLE hive_test"
        };
        // 拼接JDBC URL
        StringBuilder strBuilder = new StringBuilder("jdbc:hive2://");
        if (isZookeeper) {
            // 使用 ZooKeeper 方式连接
            strBuilder.append(zkQuorum)
                    .append("/")
                    .append(zooKeeperNamespace)
                    .append(";serviceDiscoveryMode=")
                    .append(serviceDiscoveryMode)
                    .append(";zooKeeperNamespace=")
                    .append(zooKeeperNamespace);
        } else {
            // 使用 HS2 方式连接
            strBuilder.append(host)
                    .append(":")
                    .append(port)
                    .append("/default");
        }
        if ("KERBEROS".equalsIgnoreCase(auth)) {
            strBuilder
                    .append(";principal=")
                    .append(principal)
                    .append("");
        } else {
            /* 普通模式 */
            strBuilder.append(";");
        }

        String url = strBuilder.toString();
        Class.forName(HIVE_DRIVER);
        Connection connection = null;
        try {
            // 获取JDBC连接
            connection = DriverManager.getConnection(url, USER_NAME, "");
            execDDL(connection, sqls[0]);
            logger.info("Create table success!");
            // 查询
            execDML(connection, sqls[1]);
            // 删表
            execDDL(connection, sqls[2]);
```

```
            logger.info("Delete table success!");
        } catch (SQLException e) {
            logger.error("Create connection failed : " + e.getMessage());
        } finally {
            // 关闭JDBC连接
            if (null != connection) {
                connection.close();
            }
        }
    }

    /**
     * Execute DDL Task process
     */
    public static void execDDL(Connection connection, String sql) throws SQLException {
        PreparedStatement statement = null;
        try {
            statement = connection.prepareStatement(sql);
            statement.execute();
        } finally {
            if (null != statement) {
                statement.close();
            }
        }
    }

    /**
     * Execute DML Task process
     */
    public static void execDML(Connection connection, String sql) throws SQLException {
        PreparedStatement statement = null;
        ResultSet resultSet = null;
        ResultSetMetaData resultMetaData = null;

        try {
            // 执行HQL
            statement = connection.prepareStatement(sql);
            resultSet = statement.executeQuery();

            // 输出查询的列名到控制台
            resultMetaData = resultSet.getMetaData();
            int columnCount = resultMetaData.getColumnCount();
            String resultMsg = "";
            for (int i = 1; i <= columnCount; i++) {
                resultMsg += resultMetaData.getColumnLabel(i) + '\t';
            }
            logger.info(resultMsg);
```

```
            // 输出查询结果到控制台
            while (resultSet.next()) {
                String result = "";
                for (int i = 1; i <= columnCount; i++) {
                    result += resultSet.getString(i) + '\t';
                }
                logger.info(result);
            }
        } finally {
            if (null != resultSet) {
                resultSet.close();
            }

            if (null != statement) {
                statement.close();
            }
        }
    }
}
```

示例说明

- 创建hiveclient.properties文件，必须包含host和port参数，其他可根据实际情况修改；完整示例如下。

```
host = {IP}
port = 7001
user = hadoop
auth = KERBEROS
principal = hadoop/tbds-12-0-0-1@ABC-4PT4G9W6
keytab = /etc/emr.keytab
isZookeeper = false
serviceDiscoveryMode = zooKeeper
zooKeeperNamespace = hiveserver2
zk.quorum = {IP1}:2181,{IP2}:2181,{IP3}:2181
```

- krb环境指定krb5.conf，一般都在/etc目录下。
- 执行命令

```
#kerbeos环境，先krbs认证
java  -Djava.security.krb5.conf=/etc/krb5.conf -Dconfig.file=hiveclient.properties -jar hive-test-1.0-SNA
PSHOT-jar-with-dependencies.jar
```

```
[redacted]#   java  -Djava.security.krb5.conf=/etc/krb5.conf -Dconfig.file=hiveclient.properties -jar hive-test-1.0-SNAPSHOT-jar-with-dependencies.jar
[redacted]  INFO [main] tbds.HiveExample: Starting HiveExample application...
[redacted]  INFO [main] security.KerberosUtil: Get default realm successfully, the realm is : [redacted]
[redacted]  INFO [main] security.UserGroupInformation: Hadoop UGI authentication : SIMPLE
[redacted]  INFO [main] security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
[redacted]  INFO [main] jdbc.HiveStatement: The url to track the operation: [redacted]7003/query_page?operationId=42c091df-b4f2-4120-81e1-0a53359e2e43
[redacted]  INFO [main] tbds.HiveExample: Create table success!
[redacted]  INFO [main] jdbc.HiveStatement: The url to track the operation: [redacted]7003/query_page?operationId=cdd461e0-f3bb-41ee-98be-9e9798aa02b2
[redacted]  INFO [main] tbds.HiveExample: hive_test.id      hive_test.name
[redacted]  INFO [main] jdbc.HiveStatement: The url to track the operation: [redacted]7003/query_page?operationId=d53cc595-520f-45c0-8f12-7763dc814af3
[redacted]  INFO [main] tbds.HiveExample: Delete table success!
[redacted]#
```

api接口

TBDS大数据平台上的HIVE访问接口与开源兼容，可参考 Hive API。

常用操作

连接方式

Kerberos环境

- kerberos认证

klist -kt /var/krb5kdc/emr.keytab
kinit -kt /var/krb5kdc/emr.keytab hadoop/{IP}@TBDS-{域名}

- 直连模式

/usr/local/service/hive/bin/beeline -u 'jdbc:hive2://{IP}:7001/;principal=hadoop/{IP}@TBDS-{域名}'

- ZK模式

/usr/local/service/hive/bin/beeline -u 'jdbc:hive2://{IP1}:2181,{IP2}:2181,{IP3}:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2;principal=hadoop/_HOST@TBDS-{域名}'

#参数说明
-d ---使用一个驱动类：beeline -d driver_class
-e ---使用一个查询语句：beeline -e "query_string"
-f ---加载一个文件：beeline -f filepath 多个文件用-e file1 -e file2
-n ---加载一个用户名：beeline -n valid_user
-p ---加载一个密码：beeline -p valid_password
-u ---加载一个JDBC连接字符串：
--autoCommit=[true/false] ---进入一个自动提交模式
--autosave=[true/false] ---进入一个自动保存模式
--color=[true/false] ---显示用到的颜色
--delimiterForDSV= DELIMITER ---分隔值输出格式的分隔符
--fastConnect=[true/false] ---在连接时，跳过组建表等对象
--force=[true/false] ---是否强制运行脚本
--headerInterval=ROWS ---输出的表间隔格式，默认是100: beeline --headerInterval=50
--help ---帮助

simple模式

- 直连模式

/usr/local/service/hive/bin/beeline -u 'jdbc:hive2://{IP}:7001/;'

- ZK模式

/usr/local/service/hive/bin/beeline -u 'jdbc:hive2://{IP1}:2181,{IP2}:2181,{IP3}:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2;'、

常用命令

```
# 查看数据库
show databases;

# 创建数据库：
create database test;

# 创建表：
create table hive_test (a int, b string) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
#创建数据表 hive_test, 并指定列分割符为','

# 查看表
show tables;

# 插入数据
insert into hive_test values(1,'gtc');

# 查询表中的前10个数据
select * from hive_test limit 10;

# 删除 Hive 表
drop table hive_test;
```

经典案例

创建不同类型表

```
# 创建Hive表（orc表）
CREATE TABLE IF NOT EXISTS tbtest_orc (
  `id_card` int,
  `tran_time` string,
  `name` string,
  `cash` int
  )
partitioned by(ds string)
stored as orc;

# 创建Hive表（orc+snappy压缩表）
CREATE TABLE IF NOT EXISTS tbtest_orc_snappy (
  `id_card` int,
  `tran_time` string,
  `name` string,
  `cash` int
  )
partitioned by(ds string)
stored as orc
TBLPROPERTIES ("orc.compression"="SNAPPY");

# 创建Hive表（parquet表）
CREATE  TABLE `tbtest_parquet_snappy`(
  `interfaceName` string,
  `uri` string,
  `reqMethod` string,
  `status` int,
  `isGzip` boolean,
  `response` string,
  `serverIp` string,
  `reqCost` float,
  `time` bigint,
  `schema` string)
STORED AS PARQUET
TBLPROPERTIES('parquet.compression'='SNAPPY')

# 创建Hive表（json表）,可以解析json接口的数据
create external table if not exists tbtest_json(
id int,
```

```
name string
)
row format serde 'org.apache.hive.hcatalog.data.JsonSerDe'
stored as textfile;
##数据样例如下##
{"id":1,"name":"zhangsan1"}
{"id":2,"name":"zhangsan2"}
{"id":3,"name":"zhangsan3"}
{"id":4,"name":"zhangsan4"}

# 创建hive外表，使用external关键字，外表删除时不删除数据
create external table hive_test1 (
a int,
b string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

Hive事务表

```
set hive.support.concurrency = true;
set hive.exec.dynamic.partition.mode = nonstrict;
set hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;
set hive.compactor.initiator.on = true;
set hive.compactor.worker.threads = 1;
create database if not exists hivetest;
use hivetest;
drop table if exists hive_trans_tbl;
create table hive_trans_tbl(id int, name String, age int)
clustered by (id) into 2 buckets
stored as orc
TBLPROPERTIES('transactional'='true');
insert into hive_trans_tbl values (1, '张三', 18);
select * from hivetest.hive_trans_tbl;
```

导入导出Hive表/分区数据

数据准备

```
#连接hive，创建数据
/usr/local/service/hive/bin/beeline -u 'jdbc:hive2://{IP}:7001/;principal=hadoop/{IP}@TBDS-{域名}'
```

```
create table export_test(id int,name string);
insert into export_test values(1,"HIVE");

#hdfs上创建数据目录
hdfs dfs -mkdir /tmp/export
```

简单导出导入

```
#在源端集群执行以下命令将表“export_test”的元数据和业务数据导出到创建的目录下。
export table export_test to 'hdfs:///tmp/export';

#在目标集群执行以下命令，将导出的表数据导入到表“export_test”中。
import from '/tmp/export'
```

在导入时重命名表

```
在源端集群执行以下命令将表“export_test”的元数据和业务数据导出到创建的目录下。
export table export_test to 'hdfs:///tmp/export';

在目标集群执行以下命令将导出的表数据导入到表“import_test”中。
import table import_test from '/tmp/export';
```

分区数据表准备

```
CREATE TABLE export_test_part (
  id INT,
  name STRING
)
PARTITIONED BY (pt1 STRING, pt2 STRING)
STORED AS PARQUET;

INSERT INTO export_test_part PARTITION (pt1='in', pt2='ka') VALUES (1, 'HIVE');
INSERT INTO export_test_part PARTITION (pt1='in', pt2='ka') VALUES (2, 'SPARK');
```

导出分区数据并导入

```
在源端集群执行以下命令将表“export_test”的pt1和pt2分区导出到创建的目录下。
EXPORT TABLE export_test_part PARTITION (pt1='in', pt2='ka') TO 'hdfs:///tmp/export/partition';

在目标集群执行以下命令将导出的表数据导入到表“export_test”中。
import from '/tmp/export/partition';
```

最佳实践

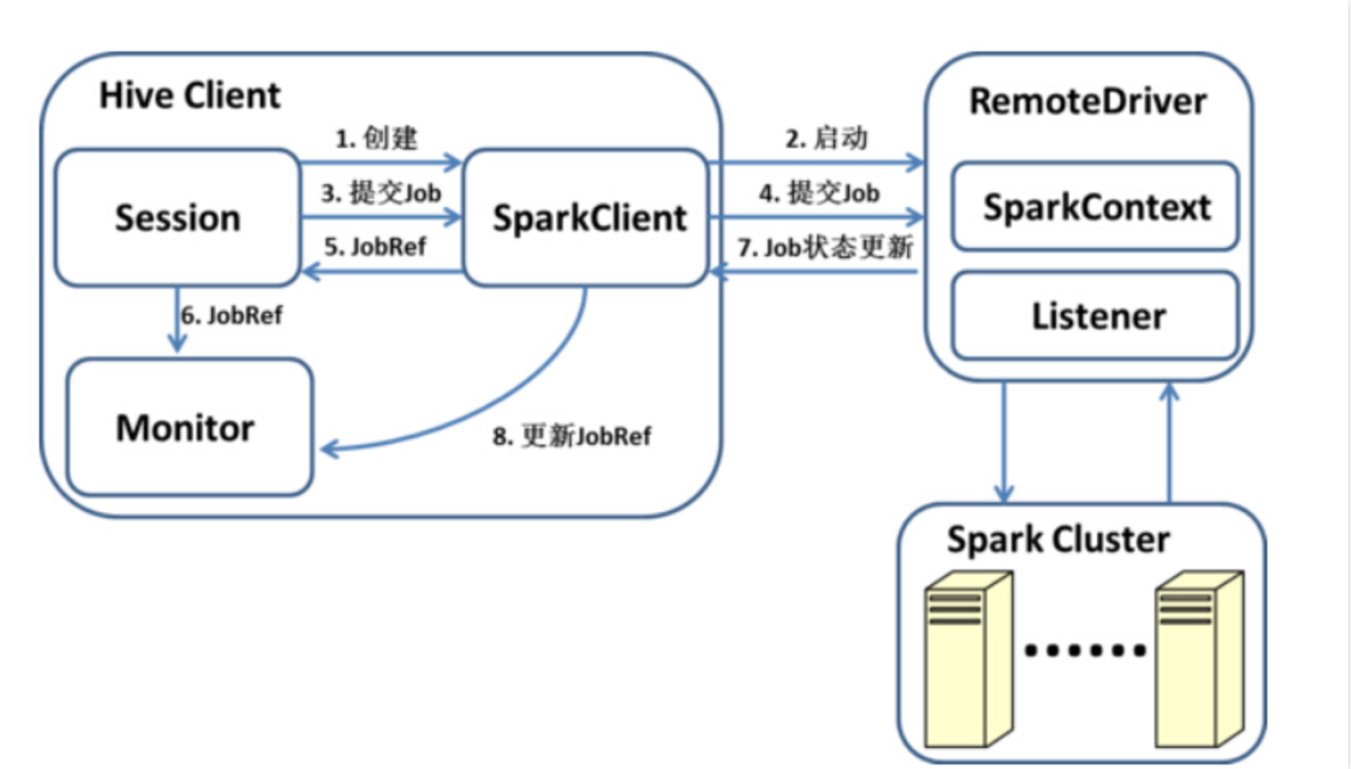
Hive on Spark任务

```
set hive.execution.engine=spark;
create database if not exists hive_db_spark;
use hive_db_spark;
drop table hive_table_spark;
create table hive_table_spark (id int, name string, age int) ROW FORMAT DELIMITED FIELDS TERMINATED BY
',';

insert into hive_table_spark values(1,'gtc',25),(2,'tom',35),(3,'danny',24),(4,'hive',26),(5,'test',24);
select * from hive_table_spark;
```

任务执行过程图

```
0: jdbc:hive2://tbds-10-4-4-19:7001/> insert into hive_table_spark values(1,'gtc',25),(2,'tom',35),(3,'danny',24),(4,'hive',26),(5,'test',24);
INFO  : Compiling command(queryId=hadoop_20241210114014_300aa497-3d09-4bb3-a456-a477a8ea8922): insert into hive_table_spark values(1,'gtc',25),(2,'tom',35),(3,'danny',24),(4,'hive',26),(5,'test',24)
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Created Hive schema: Schema(fieldSchemas:[FieldSchema(name:col1, type:int, comment:null), FieldSchema(name:col2, type:string, comment:null), FieldSchema(name:col3, type:int, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hadoop_20241210114014_300aa497-3d09-4bb3-a456-a477a8ea8922); Time taken: 0.286 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=hadoop_20241210114014_300aa497-3d09-4bb3-a456-a477a8ea8922): insert into hive_table_spark values(1,'gtc',25),(2,'tom',35),(3,'danny',24),(4,'hive',26),(5,'test',24)
INFO  : Query ID = hadoop_20241210114014_300aa497-3d09-4bb3-a456-a477a8ea8922
INFO  : Total jobs = 1
INFO  : Launching Job 1 out of 1
INFO  : Starting task [Stage-1:MAPRED] in parallel
----------------------------------------------------------------------------------------------
          STAGES   ATTEMPT        STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED
----------------------------------------------------------------------------------------------
Stage-0                0       RUNNING      1          0        1        0       0
Stage-1                0       PENDING      1          0        0        1       0
----------------------------------------------------------------------------------------------
STAGES: 00/02    [>>--------------------------] 0%    ELAPSED TIME: 3.08 s
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
          STAGES   ATTEMPT        STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED
----------------------------------------------------------------------------------------------
Stage-0 ........         0      FINISHED      1          1        0        0       0
Stage-1 ........         0      FINISHED      1          1        0        0       0
----------------------------------------------------------------------------------------------
STAGES: 02/02    [==========================>>] 100%  ELAPSED TIME: 10.98 s
----------------------------------------------------------------------------------------------
INFO  : Spark job[0] finished successfully in 10.06 second(s)
INFO  : Starting task [Stage-0:MOVE] in serial mode
INFO  : Loading data to table hive_db_spark.hive_table_spark from hdfs://HDFS78000005/usr/hive/warehouse/hive_db_spark.db/hive_table_spark/.hive-staging_hive_2024-12-10_11-40-14_935_8318939830613063587-1/-ext-10000
INFO  : Starting task [Stage-2:STATS] in parallel
INFO  : Completed executing command(queryId=hadoop_20241210114014_300aa497-3d09-4bb3-a456-a477a8ea8922); Time taken: 40.529 seconds
5 rows affected (40.859 seconds)
```

- 用户的每个Session会创建一个SparkClient，SparkClient会启动RemoteDriver进程，并由RemoteDriver创 建SparkContext。SparkTask执行时，通过Session提交任务，任务的主体就是对应的SparkWork。
- SparkClient 将任务提交给RemoteDriver，并返回一个SparkJobRef，通过该SparkJobRef，客户端可以监控任务执行进度，进行错误处理， 以及采集统计信息等。由于最终的RDD计算没有返回结果，因此客户端只需要监控执行进度而不需要处理返回值。
- RemoteDriver通过 SparkListener收集任务级别的统计数据，通过Accumulator收集Operator级别的统计数据（Accumulator被包装为 SparkCounter），并在任务结束时返回给SparkClient。
- SparkClient 与RemoteDriver之间通过基于Netty的RPC进行通信。除了提交任务，SparkClient还提供了诸如添加JAR包、获取集群信息等接口。如果客户端需要使用更一般的SparkContext的功能，可以自定义一个任务并通过SparkClient发送到RemoteDriver上执行
- Hive on Spark对于Spark集群的部署方式没有特别的要求，RemoteDriver可以连接到任意的Spark集群来执行任务。在我们的测试中，Hive on Spark在Standalone和Spark on YARN的集群上都能正常工作。



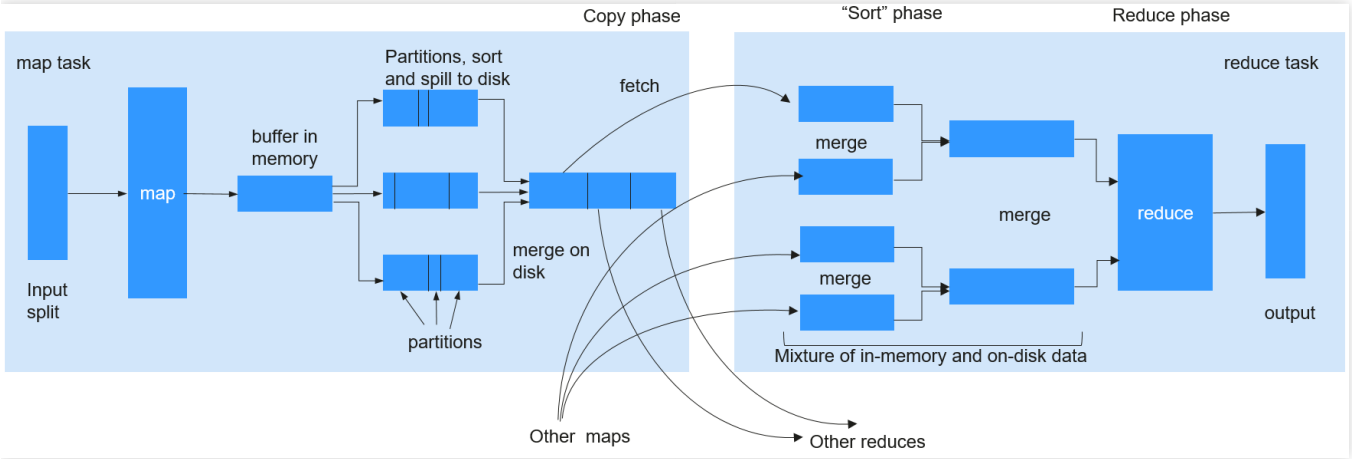
常见任务类型

Hive on MR任务

```
set hive.execution.engine=mr;
create database if not exists hive_db_mr;
use hive_db_mr;
drop table hive_table_mr;
create table hive_table_mr (id int, name string, age int) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

insert into hive_table_mr values(1,'gtc',25),(2,'tom',35),(3,'danny',24),(4,'hive',26),(5,'test',24);
select * from hive_table_mr;
```

任务执行过程图

```
0: jdbc:hive2://tbds-10-4-4-19:7001/> insert into hive_table_mr values(1,'gtc',25),(2,'tom',35),(3,'danny',24),(4,'hive',26),(5,'test',24);
INFO  : Compiling command(queryId=hadoop_20241210113749_b72e5422-0ec1-4efb-adef-b17f4b97738c): insert into hive_table_mr values(1,'gtc',25),(2,'tom',35),(3,'danny',24),(4,'hive',26),(5,'test',24)
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Created Hive schema: Schema(fieldSchemas:[FieldSchema(name:col1, type:int, comment:null), FieldSchema(name:col2, type:string, comment:null), FieldSchema(name:col3, type:int, comment:null)], properties:null)
INFO  : Completed compiling command(queryId=hadoop_20241210113749_b72e5422-0ec1-4efb-adef-b17f4b97738c); Time taken: 0.347 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=hadoop_20241210113749_b72e5422-0ec1-4efb-adef-b17f4b97738c): insert into hive_table_mr values(1,'gtc',25),(2,'tom',35),(3,'danny',24),(4,'hive',26),(5,'test',24)
WARN  : Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or using Hive 1.X releases.
INFO  : Query ID = hadoop_20241210113749_b72e5422-0ec1-4efb-adef-b17f4b97738c
INFO  : Total jobs = 3
INFO  : Launching Job 1 out of 3
INFO  : Starting task [Stage-1:MAPRED] in parallel
INFO  : Number of reduce tasks determined at compile time: 1
INFO  : In order to change the average load for a reducer (in bytes):
INFO  :   set hive.exec.reducers.bytes.per.reducer=<number>
INFO  : In order to limit the maximum number of reducers:
INFO  :   set hive.exec.reducers.max=<number>
INFO  : In order to set a constant number of reducers:
INFO  :   set mapreduce.job.reduces=<number>
INFO  : job_1731568354545_0038handle token cost:47ms
INFO  : job_1731568354545_0038 hand resources cost:9177ms
INFO  : job_1731568354545_0038 write splits cost:1310ms
INFO  : number of splits:1
INFO  : job_1731568354545_0038 queue: default
INFO  : job_1731568354545_0038 write conf cost:70ms
INFO  : Submitting tokens for job: job_1731568354545_0038
INFO  : Executing with tokens: [Kind: HIVE_DELEGATION_TOKEN, Service: HiveServer2ImpersonationToken, Ident: 00 06 68 61 64 6f 6f 70 06 68 61 64 6f 6f 70 23 68 61 64 6f 6f 70 2f 74 62 64 73 2d 31 30 2d 34 2d 34 2d 3
1 39 40 54 42 44 53 2d 4a 38 38 44 33 51 38 4a 8a 01 93 ae a2 53 0f 8a 01 93 d2 ae d7 0f 02 8e 34 99, Kind: HDFS_DELEGATION_TOKEN, Service: ha-hdfs:HDFS78000005, Ident: (token for hadoop: HDFS_DELEGATION_TOKEN owne
r=hadoop, renewer=hadoop, realUser=hadoop/tbds-10-4-4-19@TBDS-J88D3Q8J, issueDate=1733801842635, maxDate=1734406642635, sequenceNumber=535, masterKeyId=59)]
INFO  : job_1731568354545_0038 submitJob cost:454ms
INFO  : The url to track the job: https://tbds-10-4-4-19:5005/proxy/application_1731568354545_0038/
INFO  : Starting Job = job_1731568354545_0038, Tracking URL = https://tbds-10-4-4-19:5005/proxy/application_1731568354545_0038/
INFO  : Kill Command = /usr/local/service/hadoop/bin/mapred job  -kill job_1731568354545_0038
INFO  : Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
INFO  : 2024-12-10 11:38:12,473 Stage-1 map = 0%,  reduce = 0%
INFO  : 2024-12-10 11:38:28,952 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 4.06 sec
INFO  : 2024-12-10 11:38:41,294 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 9.71 sec
INFO  : MapReduce Total cumulative CPU time: 9 seconds 710 msec
INFO  : Ended Job = job_1731568354545_0038
INFO  : Starting task [Stage-7:CONDITIONAL] in parallel
INFO  : Stage-4 is selected by condition resolver.
INFO  : Stage-3 is filtered out by condition resolver.
INFO  : Stage-5 is filtered out by condition resolver.
INFO  : Starting task [Stage-4:MOVE] in serial mode
INFO  : Moving data to directory hdfs://HDFS78000005/usr/hive/warehouse/hive_db_mr.db/hive_table_mr/.hive-staging_hive_2024-12-10_11-37-49_686_7896714984599479507-1/-ext-10000 from hdfs://HDFS78000005/usr/hive/ware
house/hive_db_mr.db/hive_table_mr/.hive-staging_hive_2024-12-10_11-37-49_686_7896714984599479507-1/-ext-10002
INFO  : Starting task [Stage-0:MOVE] in serial mode
INFO  : Loading data to table hive_db_mr.hive_table_mr from hdfs://HDFS78000005/usr/hive/warehouse/hive_db_mr.db/hive_table_mr/.hive-staging_hive_2024-12-10_11-37-49_686_7896714984599479507-1/-ext-10000
INFO  : Starting task [Stage-2:STATS] in parallel
INFO  : MapReduce Jobs Launched:
INFO  : Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 9.71 sec   HDFS Read: 24355 HDFS Write: 386 HDFS EC Read: 0 SUCCESS
INFO  : Total MapReduce CPU Time Spent: 9 seconds 710 msec
INFO  : Completed executing command(queryId=hadoop_20241210113749_b72e5422-0ec1-4efb-adef-b17f4b97738c); Time taken: 56.086 seconds
5 rows affected (56.486 seconds)
```



Reduce过程分为三个不同步骤：Copy、Sort（实际应当称为Merge）及Reduce。在Copy过程中，Reducer尝试从NodeManagers获取Maps的输出并存储在内存或硬盘中。紧接着进行Shuffle过程（包含Sort及Reduce），这个过程将获取到的Maps输出进行存储并有序地合并然后提供给Reducer。当Job有大量的Maps输出需要处理的时候，Shuffle过程将变得非常耗时。对于一些特定的任务（例如hash join或hash aggregation类型的SQL任务），Shuffle过程中的排序并非必需的。但是Shuffle却默认必须进行排序，所以需要对此处进行改进。

Hive Join数据优化

操作场景

使用Join语句时，如果数据量大，可能造成命令执行速度和查询速度慢，此时可进行Join优化。
Join优化可分为以下方式：

- Map Join
- Sort Merge Bucket Map Join
- Join顺序优化

Map Join

Hive的Map Join适用于能够在内存中存放下的小表（指表大小小于25MB），通过“hive.mapjoin.smalltable.filesize”定义小表的大小，默认为25MB。
Map Join的方法有两种：

- 使用/*+ MAPJOIN(join_table) */。
- 执行语句前设置如下参数，当前版本中该值默认为“true”。set hive.auto.convert.join=true;
  使用Map Join时没有Reduce任务，而是在Map任务前起了一个MapReduce Local Task，这个Task通过TableScan读取小表内容到本机，在本机以HashTable的形式保存并写入硬盘上传到HDFS，并在Distributed Cache中保存，在Map Task中从本地磁盘或者Distributed Cache中读取小表内容直接与大表join得到结果并输出。
  使用Map Join时需要注意小表不能过大，如果小表将内存基本用尽，会使整个系统性能下降甚至出现内存溢出的异常。

Sort Merge Bucket Map Join

使用Sort Merge Bucket Map Join必须满足以下2个条件：

- join的两张表都很大，内存中无法存放。
- 两张表都按照join key进行分桶（clustered by (column)）和排序（sorted by(column)），且两张表的分桶数正好是倍数关系。
  通过如下设置，启用Sort Merge Bucket Map Join：
  set hive.optimize.bucketmapjoin=true;
  set hive.optimize.bucketmapjoin.sortedmerge=true;
  这种Map Join也没有Reduce任务，是在Map任务前启动MapReduce Local Task，将小表内容按桶读取到本地，在本机保存多个桶的HashTable备份并写入HDFS，并保存在Distributed Cache中，在Map Task中从本地磁盘或者Distributed Cache中按桶一个一个读取小表内容，然后与大表做匹配直接得到结果并输出。

Join顺序优化

当有3张及以上的表进行Join时，选择不同的Join顺序，执行时间存在较大差异。使用恰当的Join顺序可以有效缩短任务执行时间。
Join顺序原则：

- Join出来结果较小的组合，例如表数据量小或两张表Join后产生结果较少，优先执行。
- Join出来结果大的组合，例如表数据量大或两张表Join后产生结果较多，在后面执行。
  例如，customer表的数据量最多，orders表和lineitem表优先Join可获得较少的中间结果。
  原有的Join语句如下：

```
select
  l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  o_orderdate,
  o_shippriority
from
  customer,
  orders,
  lineitem
where
  c_mktsegment = 'BUILDING'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < '1995-03-22'
  and l_shipdate > '1995-03-22'
limit 10;
```

Join顺序优化后如下：

```
select
  l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  o_orderdate,
  o_shippriority
from
  orders,
  lineitem,
  customer
where
  c_mktsegment = 'BUILDING'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < '1995-03-22'
  and l_shipdate > '1995-03-22'
limit 10;
```

注意事项

- Join数据倾斜问题。执行任务的时候，任务进度长时间维持在99%，这种现象叫数据倾斜。
- 数据倾斜是经常存在的，因为有少量的Reduce任务分配到的数据量和其他Reduce差异过大，导致大部分Reduce都已

完成任务，但少量Reduce任务还没完成的情况。

- 解决数据倾斜的问题，可通过设置“set hive.optimize.skewjoin=true”并调整“hive.skewjoin.key”的大小。“hive.skewjoin.key”是指Reduce端接收到多少个key即认为数据是倾斜的，并自动分发到多个Reduce。

Hive Group By语句优化

操作场景

优化Group by语句，可提升命令执行速度和查询速度。
Group by的时候， Map端会先进行分组， 分组完后分发到Reduce端， Reduce端再进行分组。可采用Map端聚合的方式来进行Group by优化，开启Map端初步聚合，减少Map的输出数据量。

操作步骤

在Hive客户端进行如下设置：
set hive.map.aggr=true;

注意事项

- Group By数据倾斜Group By也同样存在数据倾斜的问题，设置“hive.groupby.skewindata”为“true”，生成的查询计划会有两个MapReduce Job，第一个Job的Map输出结果会随机的分布到Reduce中，每个Reduce做聚合操作，并输出结果，这样的处理会使相同的Group By Key可能被分发到不同的Reduce中，从而达到负载均衡，第二个Job再根据预处理的结果将Group By Key分发给Reduce任务，以完成最终的聚合操作。
- Count Distinct聚合问题当使用聚合函数count distinct完成去重计数时，处理值为空的情况会使Reduce产生很严重的数据倾斜，可以将空值单独处理，如果是计算count distinct，可以通过where子句将该值排除掉，并在最后的count distinct结果中加1。如果还有其他计算，可以先将值为空的记录单独处理，再和其他计算结果合并。

常见问题

Hive3的时间转换差8个小时

在Hive3中，由于时间戳默认使用UTC时区，转换过程少8小时。我们在hive中禁用了cast函数进行时间类型与数字类型的转化，如果有相关场景使用（例如从Hive2迁移Hive3），需要在业务侧进行改造，使用from_utc_timestamp 和to_utc_timestamp函数替代。
改造前：

```
SELECT CAST(1597217764557 AS TIMESTAMP);
```

```
+--------------------------+
|           _c0            |
+--------------------------+
| 2020-08-12 07:36:04.557  |
+--------------------------+
```

改造后：

```
select from_utc_timestamp(1597217764557,'GMT+8');
```

```
+--------------------------+
|           _c0            |
+--------------------------+
| 2020-08-12 15:36:04.557  |
+--------------------------+
```

hive.groupby.skewindata 导致任务失败重试结果错

误

问题现象：手动设置 hive.groupby.skewindata 参数为 true（这个默认是 false 的），就会在任务失败重试场景下，输出的结果少部分 Group By 相关的字段值。
解决方案：
【首选】设置 hive.groupby.skewindata 为 false，关闭 GroupBy 数据倾斜的优化，社区尚未修复此问题，可尝试使用 hive.map.aggr 解决数据倾斜问题。
【不推荐】如果非常需要 hive.groupby.skewindata 设置为 true 解决数据倾斜问题，设置 tez.am.task.max.failed.attempts = 0 让失败后不再进行 task 级别重试，避免出现成功但是数据出错情况。

分区表新增字段后，往旧分区插入数据，新字段值为 NULL

问题现象：查询分区表，数据文件字段值不为空情况下，查询出来的字段值为 NULL。
原因：

- 这个表最近做了表结构变更，新加了一个字段。
- 之后又写了新的字段到旧的分区，但是旧分区在 COLUMNS_V2 表存储的字段列表并没有包含新的字段名（这是默认的Hive行为），
- 导致旧分区就算包含新字段的数据，也无法读出来。
  解决方案：
- 修复 COLUMNS_V2 表，对旧的 CD_ID 新增最近添加的字段。

SELECT * FROM COLUMNS_V2 WHERE CD_ID='1764367';

INSERT INTO COLUMNS_V2(CD_ID,COLUMN_NAME,TYPE_NAME,INTEGER_IDX) values (旧的CD_ID, xxx, xxx, 和新分区INTEGER_IDX一致);

- 如果想要新增字段对所有历史分区有效，那应该在 ALTER TABLE 时使用 CASCADE 关键字。

ALTER TABLE dbname.table_name ADD columns (column1 string,column2 string) CASCADE;

提示 Multiple partitions for one merge mapper

如果表存储格式为ORC。并且由于小文件问题hive.merge.tezfiles改成了true进行小文件输出合并，这里是hive的bug。

```
set hive.merge.tezfiles=false;
```

drop table异常提示You might need to increase max_locks_per_transaction.)FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask success

查看 metastore日志关键字：“Can not delete hdfs path”
解决：drop table 执行时，在一个事务里需要删除partitions的元数据，分区太多的话，会占用比较多的shared memory建议先TRUNCATE TABLE清空表数据，再删除表。

读取Snappy文件出现OOM

原因分析：LogService等服务写入的标准Snappy文件和Hadoop生态的Snappy文件格式不同，EMR默认处理的是Hadoop修改过的Snappy格式，处理标准格式时会报错OutOfMemoryError。
解决方法：对Hive作业配置如下参数。

```
set io.compression.codec.snappy.native=true;
```

Hive和Spark时区不一致导致结果不一致

异常现象：Hive的from_unixtime时区固定为UTC，而Spark使用的是本地时区，如果两者的时区不一致会导致结果不一致。
解决方法：修改Spark时区为UTC，在Spark SQL里面插入如下代码：

```
set spark.sql.session.timeZone=UTC;
```

或者修改Spark配置文件，添加新的配置：

```
spark.sql.session.timeZone=UTC
```

权限策略配置

配置步骤

1. 安装 Ranger Plugin：

   i. 下载并安装 Ranger Hive 插件。

2. 配置 Hive：

   i. 修改 hive-site.xml 文件，添加以下配置：

```
<property>
  <name>hive.security.authorization.manager</name>
  <value>org.apache.ranger.authorization.hive.authorizer.RangerHiveAuthorizerFactory</value>
</property>
<property>
  <name>hive.security.authenticator.manager</name>
  <value>org.apache.hadoop.hive.ql.security.SessionStateUserAuthenticator</value>
</property>
```

3. 配置 Ranger：

   i. 使用Ranger管理员用户rangeradmin登录Ranger管理页面。
   ii. 在首页中单击“HADOOP SQL”区域的组件插件名称，例如“hive”。
   iii. 在“Access”页签单击“Add New Policy”，添加Hive权限控制策略。
   iv. 根据业务需求配置相关参数。
   Hive权限参数如下：

参数名称	描述
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。
Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持“*”通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。
Policy Label	为当前策略指定一个标签，可以根据这些标签搜索报告和筛选策略。
database	将适用该策略的列Hive数据库名称。 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。

table	将适用该策略的Hive表名称。 如果需要添加基于UDF的策略，可切换为UDF，然后输入UDF的名称。 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
Hive Column	将适用该策略的列名，填写*时表示所有列。 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
Description	策略描述信息。
Audit Logging	是否审计此策略。
Allow Conditions	策略允许条件，配置本策略内允许的权限及例外。 在“Select Role”、“Select Group”、“Select User”列选择已创建好的需要授予权限的Role、用户组或用户，单击“Add Conditions”，添加策略适用的IP地址范围，然后再单击“Add Permissions”，添加对应权限。 ● select：查询权限 ● update：更新权限 ● Create：创建权限 ● Drop：drop操作权限 ● Alter：alter操作权限 ● Index：索引操作权限 ● All：所有执行权限 ● Read：可读权限 ● Write：可写权限 ● Temporary UDF Admin：临时UDF管理权限 ● Select/Deselect All：全选/取消全选 如需添加多条权限控制规则，可单击 按钮添加。 如需当前条件中的用户或用户组管理本条策略，可勾选“Delegate Admin”，这些用户将成为受委托的管理员。被委托的管理员可以更新、删除本策略，它还可以基于原始策略创建子策略。
Deny Conditions	策略拒绝条件，配置本策略内拒绝的权限及例外，配置方法与“Allow Conditions”相同。

设置权限场景如下：

任务场景	角色授权操作

role admin操作	1. 在首页中单击“Settings”，选择“Roles”。 2. 单击Role Name为admin的角色，在“Users”区域，单击“Select User”，选择对应用户名。 3. 单击Add Users按钮，在对应用户名所在行勾选“Is Role Admin”，单击“Save”保存配置。 说明： Ranger页面的“Settings”选项只有rangeradmin用户有权限访问。用户绑定Hive管理员角色后，在每个维护操作会话中，还需要执行以下操作： 1. 以客户端安装用户，登录安装Hive客户端的节点。 2. 执行以下命令配置环境变量。 例如，Hive客户端安装目录为“/opt/hiveclient”，执行source/opt/hiveclient/bigdata_env 1. 执行以下命令认证用户。 kinit Hive业务用户 1. 执行以下命令登录客户端工具。 beeline 1. 执行以下命令更新用户的管理员权限。 set role admin;
创建库表操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库(如果是创建表则在“table”右侧填写或选择对应的表)，在“column”右侧填写或选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Create”。
删除库表操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库（如果是删除表则在“table”右侧填写或选择对应的表），在“column”右侧填写并选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Drop”。
查询操作(select、desc、show)	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库(如果是表则在“table”右侧填写或选择对应的表)，在“column”右侧填写并选择对应的列（*代表所有列）。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“select”。

Alter操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写并选择对应的数据库(如果是表则在“table”右侧填写或选择对应的表)，在“column”右侧填写或选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Alter”。
LOAD操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库，在“table”右侧填写或选择对应的表，在“column”右侧填写并选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“update”。
INSERT、DELETE操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库，在“table”右侧填写或选择对应的表，在“column”右侧填写并选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“update”。 5. 用户还需要具有Yarn任务队列的“submit”权限。
Import/Export操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库，在“table”右侧填写或选择对应的表，在“column”右侧填写并选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“select”。
Repl Dump/Load操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库，在“table”右侧填写或选择“*”，在“column”右侧填写并选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“ReplAdmin”。
GRANT、REVOKE操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库，在“table”右侧填写或选择对应的表，在“column”右侧填写并选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 勾选“Delegate Admin”。
ADD JAR操作	1. 在“Policy Name”填写策略名称。 2. 单击“database”并在下拉菜单中选择“global”。在“global”右侧填写或选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Temporary UDF Admin”。

UDF操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库，“UDF”右侧填写对应的UDF函数名。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，根据需求，给用户勾选相应权限（UDF支持Create，SELECT， Drop）。
VIEW操作	1. 在“Policy Name”填写策略名称。 2. “database”右侧填写或选择对应的数据库，在“table”右侧填写或选择对应的VIEW名称，在“column”右侧填写并选择“*”。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，参照表格上述相关操作，根据需求，给用户勾选相应权限。
dfs命令操作	执行set role admin操作才可使用。
其他用户库表操作	1. 参照表格上述相关操作添加对应权限。 2. 给用户添加其他用户库表的HDFS路径的读、写、执行权限。

5. 单击“Add”，在策略列表可查看策略的基本信息。等待策略生效后，验证相关权限是否正常。如需禁用某条策略，可

单击 按钮编辑策略，设置策略开关为“Disabled”。

如果不再使用策略，可单击 按钮删除策略。

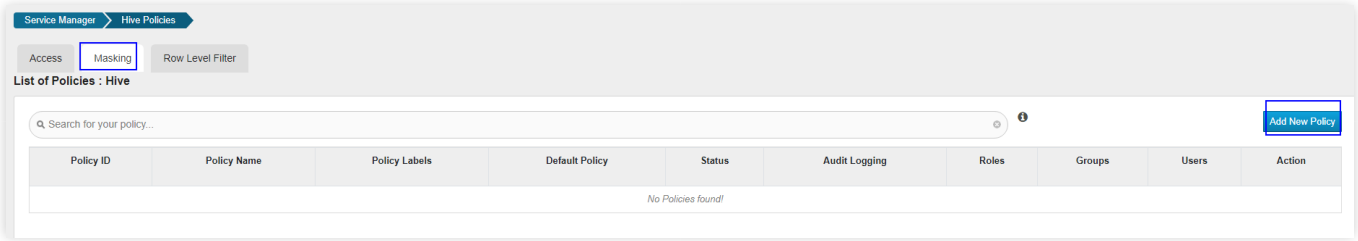
6. 重启 Hive：
   i. 重启 Hive 服务以应用配置。

Hive数据脱敏

Ranger支持对Hive数据进行脱敏处理（Data Masking），可对用户执行的SELECT操作的返回结果进行处理，以屏蔽敏感信息。

1. 登录Ranger WebUI界面，在首页中单击“HADOOP SQL”区域的“Hive”

2. 在“Masking”页签单击“Add New Policy”，添加Hive权限控制策略。



3. 根据业务需求配置相关参数。

Hive数据脱敏参数：

参数名称	描述
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。
Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持“*”通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。
Policy Label	为当前策略指定一个标签，可以根据这些标签搜索报告和筛选策略。
Hive Database	配置当前策略适用的Hive中数据库名称，支持设置多个数据库名，并且填写支持“*”通配符，例如：aa、a*、*b、a*b或者*。
Hive Table	配置当前策略适用的Hive中的表名称，支持设置多个表名，并且填写支持“*”通配符，例如：aa、a*、*b、a*b或者*。
Hive Column	可添加列名，支持设置多个列名，并且填写支持“*”通配符，例如：aa、a*、*b、a*b或者*。
Description	策略描述信息。
Audit Logging	是否审计此策略。
Mask Conditions	在“Select Role”、“Select Group”、“Select User”列选择已创建好的需要授予权限的对象，单击“Add Conditions”，添加策略适用的IP地址范围，然后再单击“Add Permissions”，勾选“SELECT”权限。 单击“Select Masking Option”，选择数据脱敏时的处理策略： ● Redact：用x屏蔽所有字母字符，用0屏蔽所有数字字符。 ● Partial mask: show last 4：只显示最后的4个字符，其他用x代替。 ● Partial mask: show first 4：只显示开始的4个字符，其他用x代替。 ● Hash：用值的哈希值替换原值，采用的是hive的内置mask_hash函数，只对string、char、varchar类型的字段生效，其他类型的字段会返回NULL值。 ● Nullify：用NULL值替换原值。 ● Unmasked(retain original value)：原样显示。 ● Date: show only year：仅显示日期字符串的年份部分，并将月份和日期默认为01/01。 ● Custom：可使用任何有效返回与被屏蔽的列中的数据类型相同的数据类型来自定义策略。 如需添加多列的脱敏策略，可单击 按钮添加。

4. 单击“Add”，在策略列表可查看策略的基本信息。

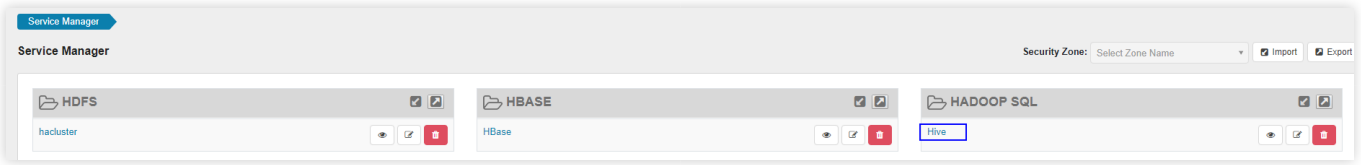
5. 用户通过Hive客户端对配置了数据脱敏策略的表执行SELECT操作，系统将对数据进行处理后进行展示。

> 说明：
>
> 处理数据需要用户同时具有向Yarn队列提交任务的权限。

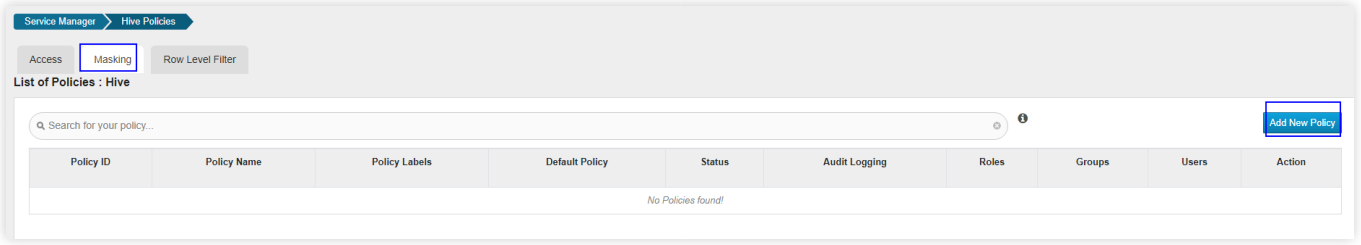
Hive行级别数据过滤

Ranger支持用户对Hive数据表执行SELECT操作时进行行级别的数据过滤。

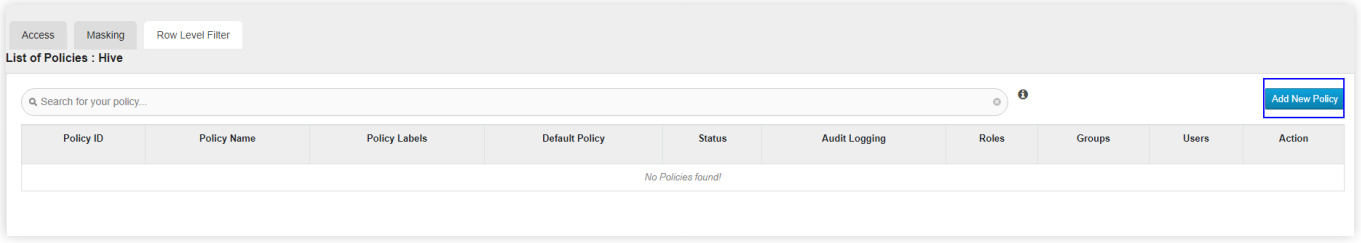
1. 登录Ranger WebUI界面，在首页中单击“HADOOP SQL”区域的“Hive”。



2. 在“Row Level Filter”页签单击



“Add New Policy”，添加行数据过滤策略。



3. 根据业务需求配置相关参数。

Hive行数据过滤参数：

参数名称	描述
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。

Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持“*”通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。
Policy Label	为当前策略指定一个标签，可以根据这些标签搜索报告和筛选策略。
Hive Database	配置当前策略适用的Hive中数据库名称。
Hive Table	配置当前策略适用的Hive中的表名称。
Description	策略描述信息。
Audit Logging	是否审计此策略。
Row Filter Conditions	在“Select Role”、“Select Group”、“Select User”列选择已创建好的需要授予权限的对象，单击“Add Conditions”，添加策略适用的IP地址范围，然后再单击“Add Permissions”，勾选“SELECT”权限。 单击“Row Level Filter”，填写数据过滤规则。 例如过滤表A中“name”列“zhangsan”行的数据，过滤规则为：name <> 'zhangsan'。更多信息可参考Ranger官方文档。 如需添加更多规则，可单击 按钮添加。

4. 单击“Add”，在策略列表可查看策略的基本信息。

5. 用户通过Hive客户端对配置了数据脱敏策略的表执行SELECT操作，系统将对数据进行处理后进行展示。

> 说明：
>
> 处理数据需要用户同时具有向Yarn队列提交任务的权限。

Trino开发

概述

Trino是一个开源的用户交互式分析查询的SQL查询引擎，用于针对各种大小的数据源进行交互式分析查询。其主要应用于海量结构化数据/半结构化数据分析、海量多维数据聚合/报表、ETL、Ad-Hoc查询等场景。
Trino允许查询的数据源包括Hadoop分布式文件系统（HDFS），Hive，HBase，Cassandra，关系数据库甚至专有数据存储。一个Trino查询可以组合不同数据源，执行跨数据源的数据分析。

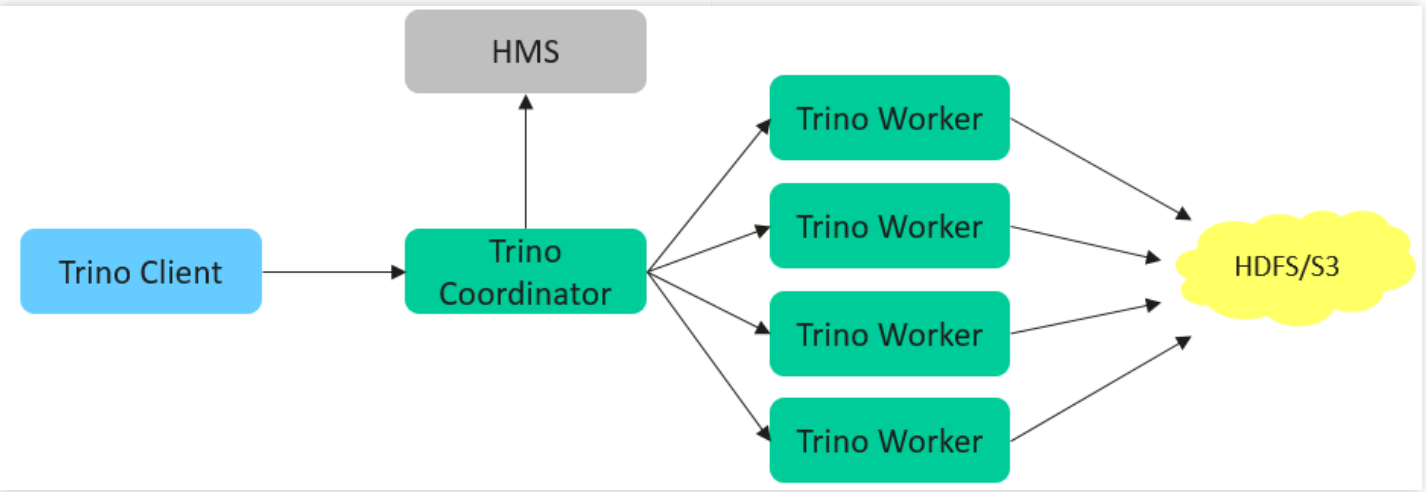
基本特性

- 高性能：Trino使用了许多优化技术，如基于成本的查询优化、管道执行、动态代码生成等，以提高查询性能。
- 分布式架构：Trino采用了MPP架构，将查询任务分解为多个子任务，然后在多个工作节点上并行执行。这使得Trino能够在大规模数据集上进行高效查询。
- 完整的 ANSI SQL 支持能力；
  Trino支持大部分ANSI SQL标准，包括复杂的查询、连接、聚合、窗口函数等。这使得用户可以使用熟悉的SQL语言进行数据分析。
- 支持丰富的数据源：
  - ClickHouse
  - Cassandra
  - ElasticSearch
  - Hive
  - Hudi
  - Iceberg
  - Kafka
  - MySQL
  - Oracle
  - PostgreSQL
  - SQL Server
  - Redis
  - JMX
  - 本地文件
- 支持高级数据结构：
  - 数组和Map数据
  - JSON数据
  - GIS数据
- 功能扩展能力强，提供了多种扩展机制：

    - 扩展数据连接器
    - 自定义数据类型
    - 自定义SQL函数
- 流水线：基于Pipeline处理模型数据在处理过程中实时返回给用户。
- 监控接口完善：
    - 提供友好的Web UI，可视化的呈现查询任务执行过程。
    - 支持JMX协议。

系统架构

Trino分布式地运行在一个集群中，包含一个Coordinator和多个Worker进程，查询从客户端（例如CLI）提交到Coordinator，Coordinator进行SQL的解析和生成执行计划，然后分发到多个Worker进程上执行。



Trino是典型的M/S架构的系统，由一个Coordinator节点和多个Worker节点组成。 Coordinator负责如下工作：

- 接收用户查询请求，解析并生成执行计划，下发任务到Worker节点执行。
- 监控Worker节点运行状态，各个Worker节点与Coordinator节点保持心跳连接，汇报节点状态。
- 维护MetaStore数据。
  Worker节点负责执行下发的任务，通过连接器读取外部存储系统的数据，进行处理，并将处理结果发送给Coordinator节点。

基本概念

数据模型

数据模型即数据的组织形式。Trino使用Catalog、Schema和Table三层结构来管理数据。

- Catalog
  一个Catalog可以包含多个Schema，物理上指向一个外部数据源，可以通过Connector访问该数据源。一次查询可以访问一个或多个Catalog。
- Schema
  相当于一个数据库实例，一个Schema包含多张数据表。
- Table
  数据表，与一般意义上的数据库表相同。

常用 connector

Trino通过各种connector来接入多种外部数据源。Trino提供了一套标准的SPI接口，用户可以使用这套接口开发自己的Connector，以便访问自定义的数据源。
一个Catalog通常会绑定一种类型的connector，在Catalog的Properties文件中设置。Trino内置了多种Connector。

参考

https://trino.io/docs/current/

示例工程开发

环境准备

开发环境准备

准备项	说明
安装JDK	JDK 17，推荐使用 konaJDK，下载地址
安装和配置 IDE	按需选择，比如 IntelliJ IDEA 或 Eclipse，示例使用IDEA
安装 Maven	开发环境基础配置，负责构建 Java 应用程序
Maven 配置准备	如果需要本地调试，需要参考 6.开发环境准备 配置 Maven settings.xml，推荐 Maven 3.6.3，下载地址

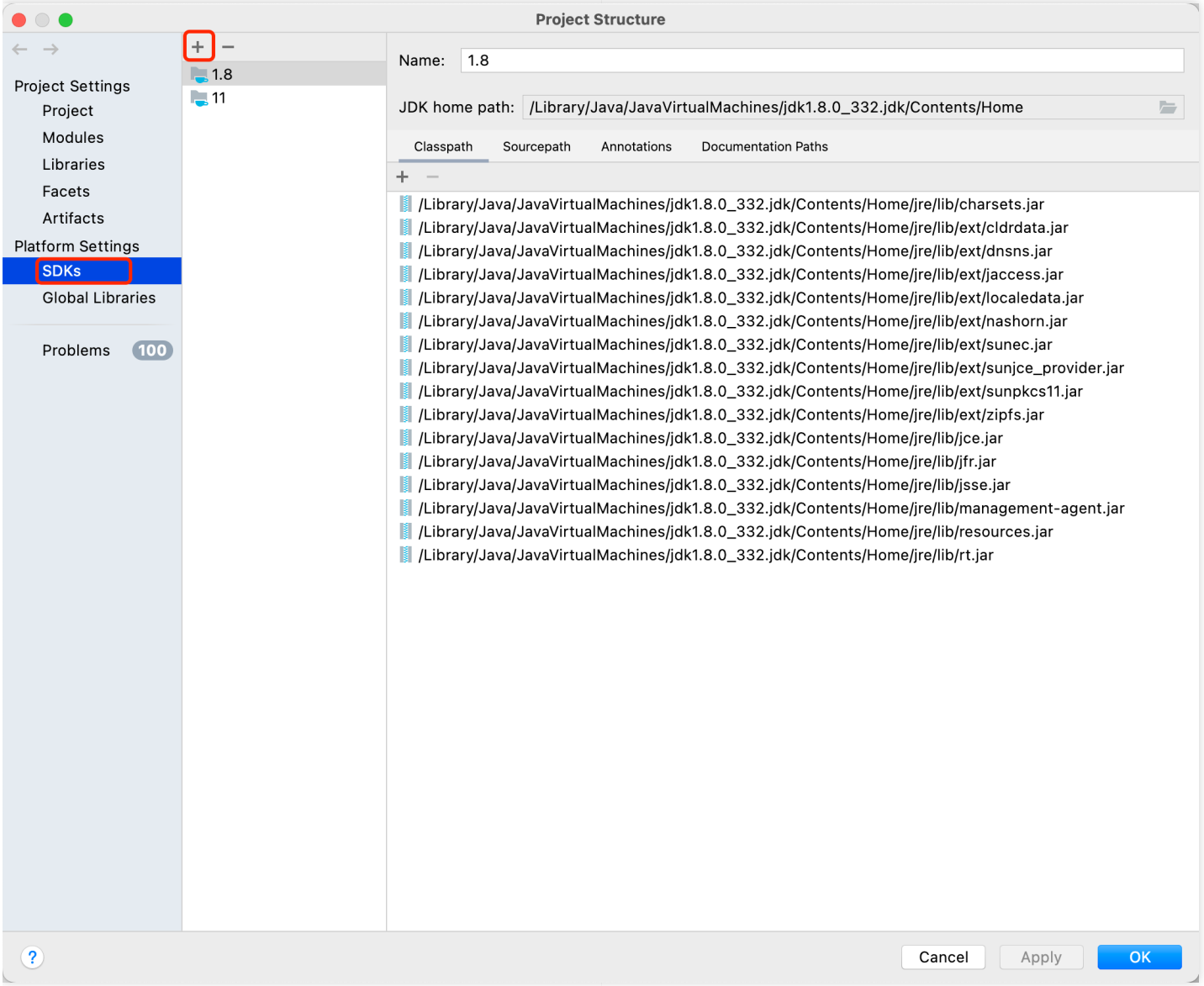
导入示例工程代码

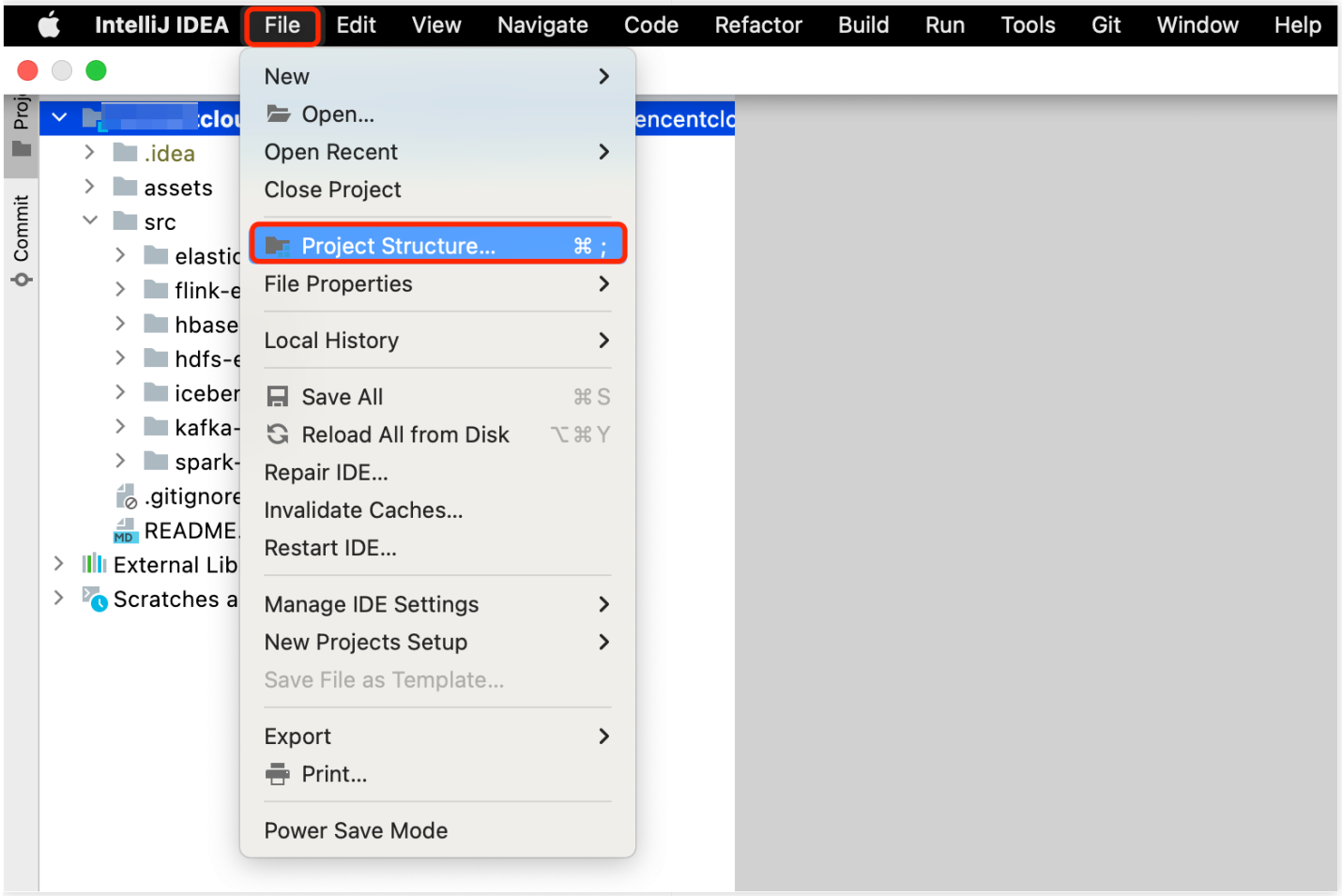
以下以 IntelliJ IDEA 举例，将示例工程代码导入进行说明。
2.1 下载样例代码：https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master

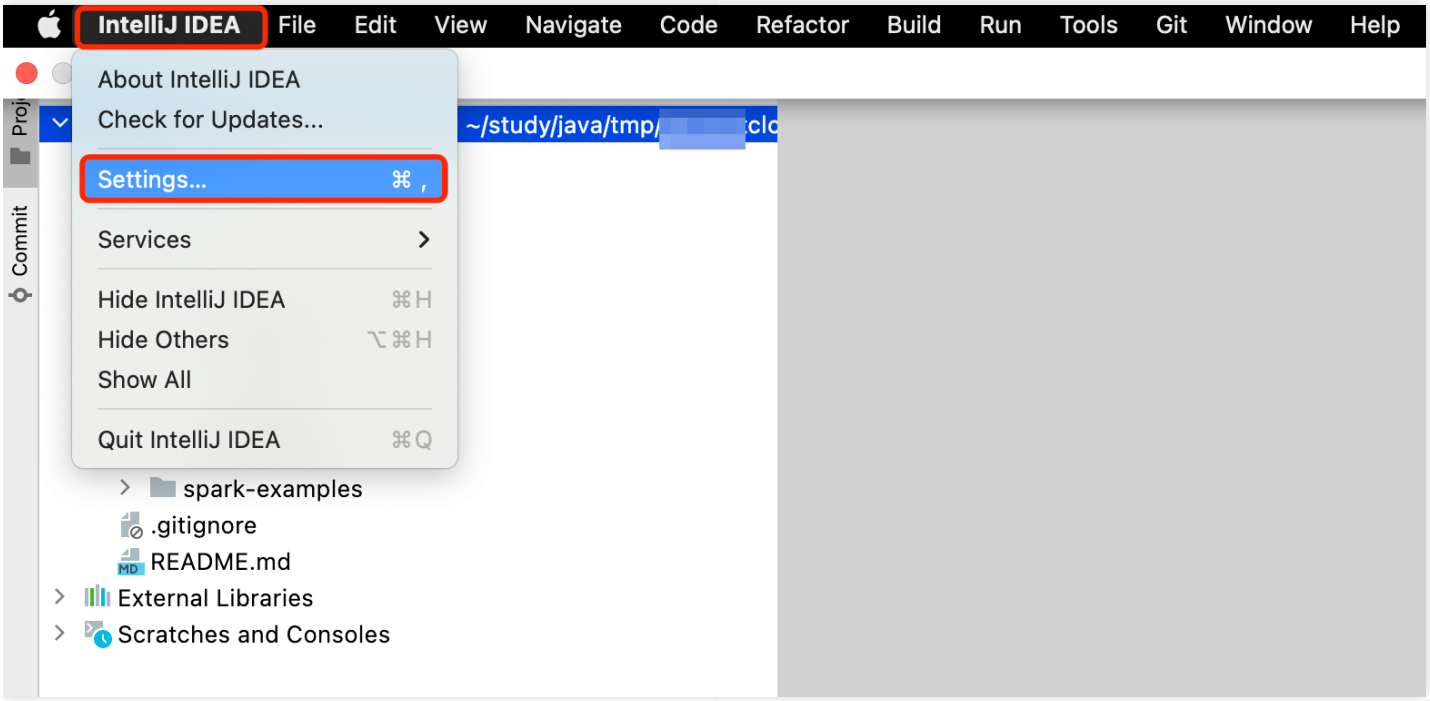
克隆或者直接下载master代码都可以

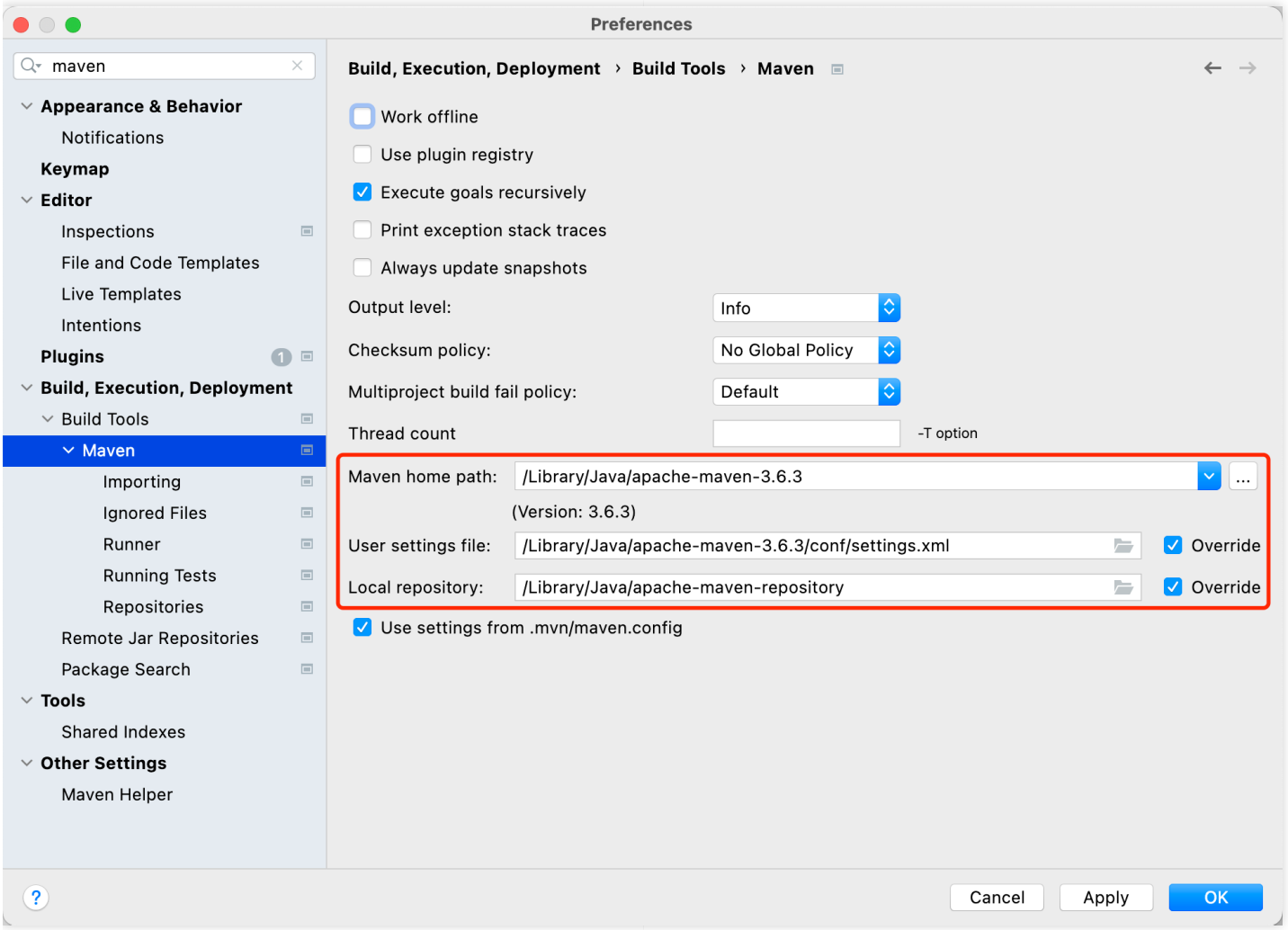
git clone https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples

2.2 导入项目，然后选择JDK、MAVEN和settings文件。


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
100
1.8
11
Name:
1.8
JDK home path:
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home
Classpath
Sourcepath
Annotations
Documentation Paths
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/charsets.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/cldrdata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/dnsns.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/jaccess.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/localedata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/nashorn.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunec.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunjce_provider.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunpkcs11.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/zipfs.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jce.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jfr.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jsse.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/management-agent.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/resources.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/rt.jar
Cancel
Apply
OK


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
New
Open...
Open Recent
Close Project
Project Structure... ⌘ ;
File Properties
Local History
Save All ⌘S
Reload All from Disk ⌥⌘Y
Repair IDE...
Invalidate Caches...
Restart IDE...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
.idea
assets
src
.gitignore
README
External Lib
Scratches a
Proj
Commit


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
About IntelliJ IDEA
Check for Updates...
Settings... ⌘ ,
Services
Hide IntelliJ IDEA ⌘H
Hide Others ⌥⌘H
Show All
Quit IntelliJ IDEA ⌘Q
~/study/java/tmp/
spark-examples
.gitignore
README.md
External Libraries
Scratches and Consoles
Proj
Commit



样例代码说明

功能说明

演示的实例是样例代码通过JDBC的方式连接和访问Trino引擎

代码逻辑说明

JDBC具体访问代码
Simple认证

```
package com.tencent.tbds.trino;
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;

public class TrinoJDBCExample {
    public static void main(String[] args) {
        try {
            args = new String[]{"hadoop", "1.2.3.4", "9000", "hive", "default"};
            String url = "jdbc:trino://" + args[1] + ":" + args[2] +"/" + args[3] + "/" + args[4];
            Properties properties = new Properties();
            properties.setProperty("user", args[0]);
            Connection connection = DriverManager.getConnection(url, properties);
            Statement stmt = connection.createStatement();
            try {
                ResultSet rs = stmt.executeQuery("select * from city");
                System.out.println("select * from city");
                while(rs.next()) {
                    int id = rs.getInt(1);
                    String name = rs.getString(2);
                    System.out.printf("id=%d, name=%s%n", id, name);
                }
            } catch (Exception e) {
                System.out.println(e);
            }
            finally {
                stmt.close();
                connection.close();
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Kerberos 认证

```
package com.tencent.tbds.trino;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;
```

```
public class TrinoJDBCKrbsExample {
    public static void main(String[] args) {
        try {
            // 构造访问trino coordinator的jdbc url，hive表示catalog，default表示schema/database
            String url = "jdbc:trino://1.2.3.4:9443/hive/default/";
            System.out.println("url: " + url);
            Properties properties = new Properties();
            // 配置开启https协议访问
            properties.setProperty("SSL", "true");
            properties.setProperty("SSLVerification", "NONE");

            // 指定使用的用户
            properties.setProperty("user", "hadoop");

            // 运行jdbc client环境的krbs配置
            properties.setProperty("KerberosConfigPath", "/etc/krb5.conf");
            properties.setProperty("KerberosPrincipal", "hadoop/10.206.17.113@TBDS-2NGVYEHH");
            properties.setProperty("KerberosKeytabPath", "/var/krb5kdc/emr.keytab");
            properties.setProperty("KerberosRemoteServiceName", "hadoop");
            properties.setProperty("KerberosUseCanonicalHostname", "false");

            Connection connection = DriverManager.getConnection(url, properties);
            Statement stmt = connection.createStatement();
            try {
                ResultSet rs = stmt.executeQuery("select * from city");
                System.out.println("select * from city");
                while(rs.next()) {
                    int id = rs.getInt(1);
                    String name = rs.getString(2);
                    System.out.printf("id=%d, name=%s%n", id, name);
                }
            } catch (Exception e) {
                System.out.println(e);
            }
            finally {
                stmt.close();
                connection.close();
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

LDAP 认证

```
package com.tencent.tbds.trino;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;

public class TrinoJDBCLdapExample {
    public static void main(String[] args) {
        try {
            // 构造访问trino coordinator的jdbc url，hive表示catalog，default表示schema/database
            String url = "jdbc:trino://1.2.3.4:9443/hive/default/";
            System.out.println("url: " + url);
            Properties properties = new Properties();
            // 配置开启https协议访问
            properties.setProperty("SSL", "true");
            properties.setProperty("SSLVerification", "NONE");

            // ldap用户与密码
            properties.setProperty("user", "sn124");
            properties.setProperty("password", "Tbds@2023!");

            Connection connection = DriverManager.getConnection(url, properties);
            Statement stmt = connection.createStatement();
            try {
                ResultSet rs = stmt.executeQuery("select * from city");
                System.out.println("select * from city");
                while(rs.next()) {
                    int id = rs.getInt(1);
                    String name = rs.getString(2);
                    System.out.printf("id=%d, name=%s%n", id, name);
                }
            } catch (Exception e) {
                System.out.println(e);
            }
            finally {
                stmt.close();
                connection.close();
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

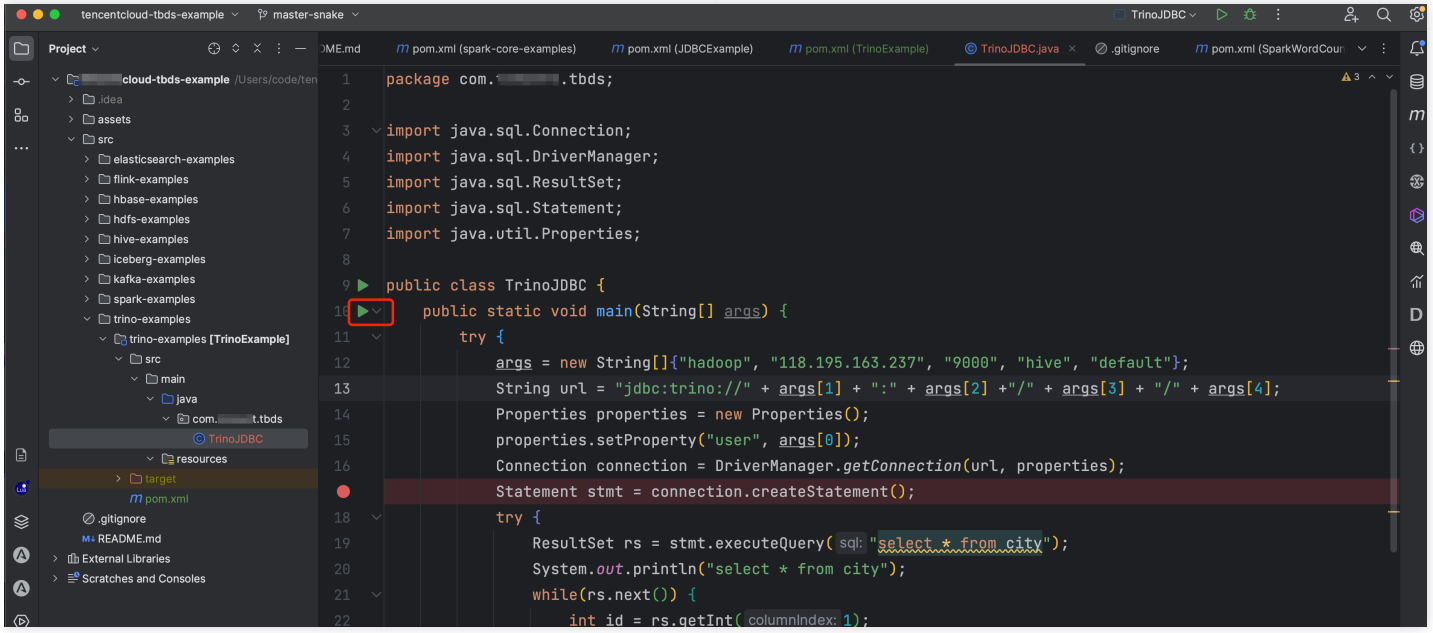
pom文件中需要引入Trino的JDBC驱动包

```
...
    <dependencies>
        <!--> 引入trino版本435的驱动 <-->
        <dependency>
            <groupId>io.trino</groupId>
            <artifactId>trino-jdbc</artifactId>
            <version>435</version>
        </dependency>
        ...
    </dependencies>
...
```

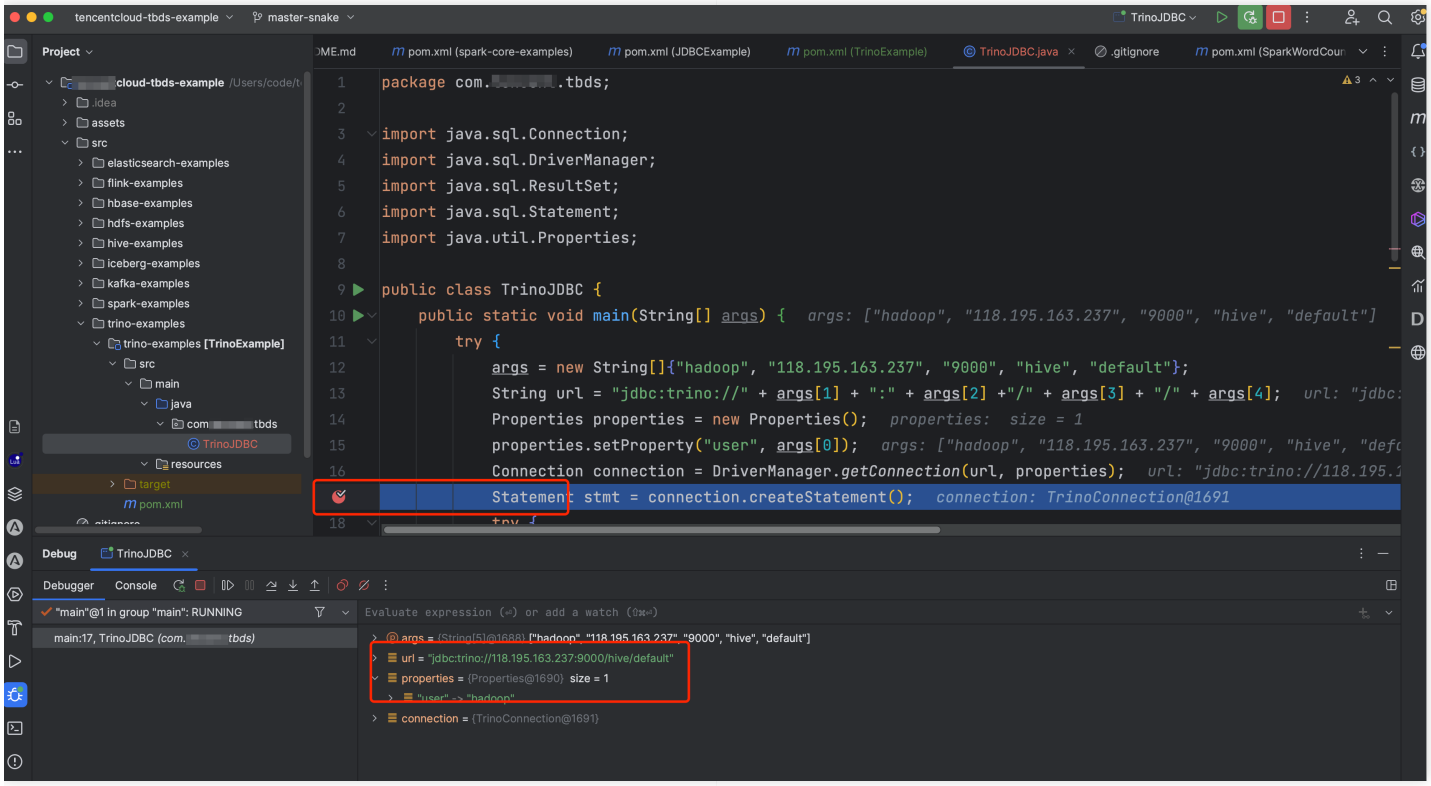
代码调测

使用idea并配置好环境之后，即可进行调试，需要注意的是，保证开发环境和Trino组件的网络互通，如果网络不互通，将代码打包之后上传到作业机，使用Java -jar TrinoExample-1.0-SNAPSHOT-jar-with-dependencies.jar 运行代码，注意，需要在pom.xml中标注mainClass。

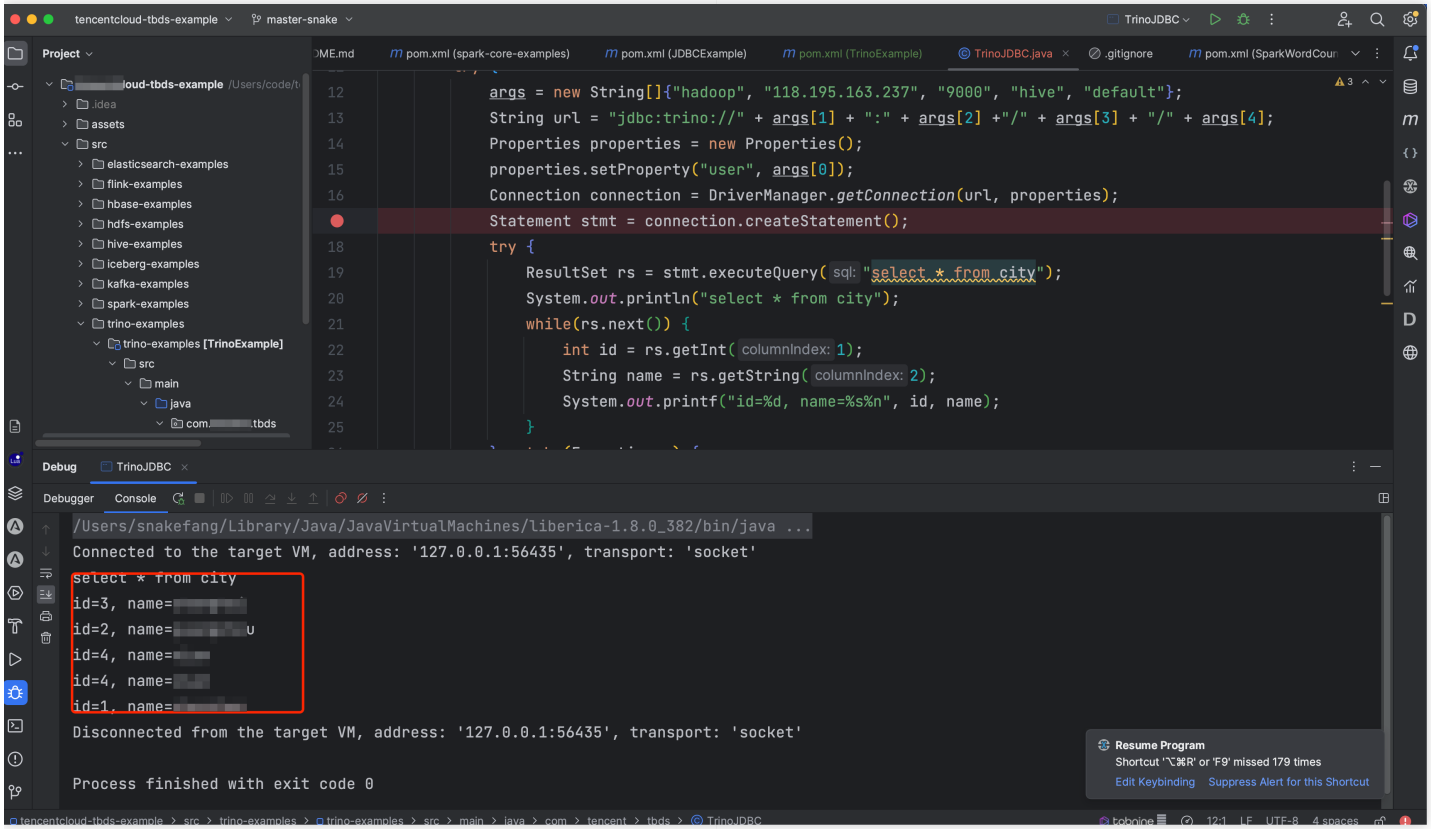
在IDEA中打好断点之后，点击红框中标志，并选择debug模式



程序运行至断点处，可以查询相关变量的参数

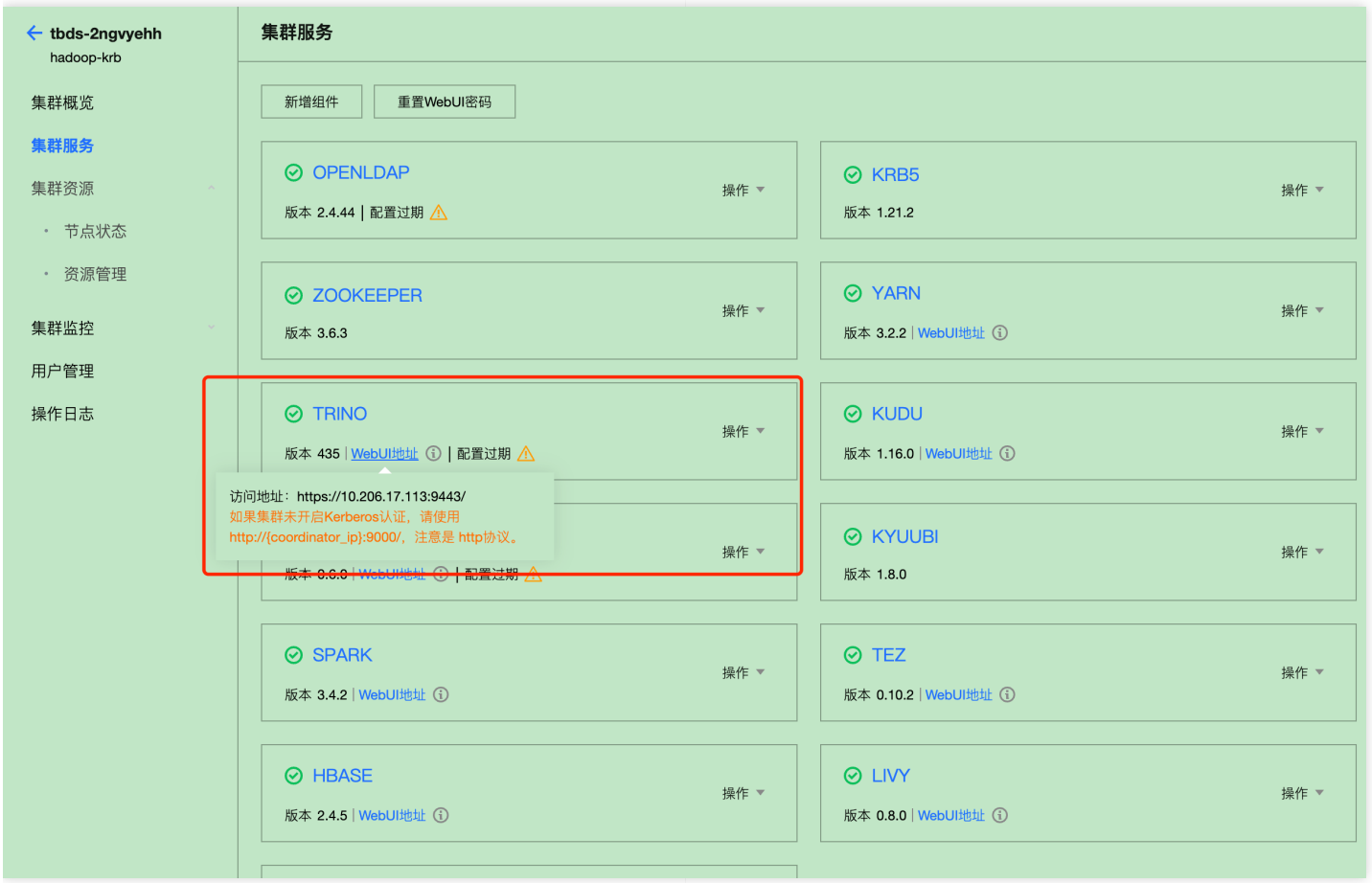


继续执行程序查看，执行结果

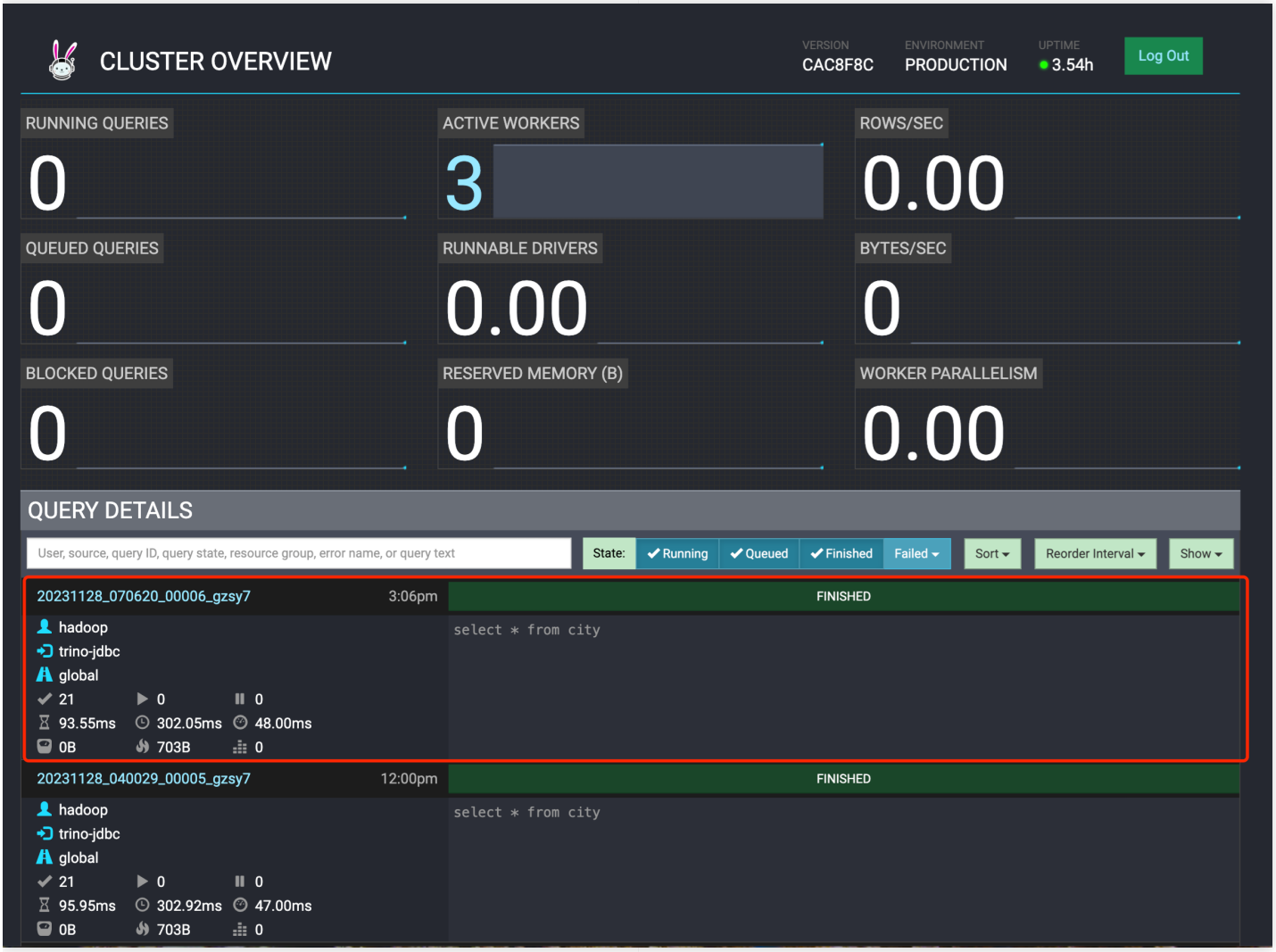


查询任务详细进度和执行计划

登录TBDS Manager 管控平台，点击“集群列表”，选择 Trino 程序运行所对应的集群。点击“集群服务”，选择 TRINO 服务，点击 WebUI 地址跳转至 Web UI 界面。



输入用户名和密码，通过查询WebUI可以查看查询任务的具体执行状态。


CLUSTER OVERVIEW
VERSION
CAC8F8C
ENVIRONMENT
PRODUCTION
UPTIME
3.54h
Log Out
RUNNING QUERIES
0
ACTIVE WORKERS
3
ROWS/SEC
0.00
QUEUED QUERIES
0
RUNNABLE DRIVERS
0.00
BYTES/SEC
0
BLOCKED QUERIES
0
RESERVED MEMORY (B)
0
WORKER PARALLELISM
0.00
QUERY DETAILS
User, source, query ID, query state, resource group, error name, or query text
State:
Running
Queued
Finished
Failed
Sort
Reorder Interval
Show
20231128_070620_00006_gzsy7
3:06pm
FINISHED
hadoop
trino-jdbc
global
21
0
0
93.55ms
302.05ms
48.00ms
0B
703B
0
select * from city
20231128_040029_00005_gzsy7
12:00pm
FINISHED
hadoop
trino-jdbc
global
21
0
0
95.95ms
302.92ms
47.00ms
0B
703B
0
select * from city

示例说明

Simple 认证

```
# 在trino节点上，执行命令如下，其中{coordinator_ip}更换成trino coordinator组件所在节点ip
# catalog用与指定要访问源，例如hive；schema则表示数据库，例如default库
[hadoop@10 ~]# /usr/local/service/trino/client/trino-cli --server http://{coordinator_ip}:9000 --user ha
doop --catalog hive --schema default
trino:default>

# 执行SQL，进行表查询
trino:default> select * from city;
 id |   name
----+-----------
  1 | city1
  3 | city2

  4 | city3
  2 | city4
  4 | city5
(5 rows)

Query 20231128_090616_00007_bbiwe, FINISHED, 3 nodes
Splits: 21 total, 21 done (100.00%)
1.10 [5 rows, 1.86KB] [4 rows/s, 1.69KB/s]
```

Kerberos 认证

```
# 在trino节点上，执行命令如下，其中{coordinator_ip}更换成trino coordinator组件所在节点ip
# krb5-principal指定客户端的principal，krb5-keytab-path指定客户端keytab的路径
# krb5-remote-service-name表示trino coordinator的服务名
# catalog用与指定要访问源，例如hive；schema则表示数据库，例如default库
[root@10 ~]# /usr/local/service/trino/client/trino-cli --server https://{coordinator_ip}:9443  --user had
oop --krb5-config-path /etc/krb5.conf --krb5-principal  hadoop/10.206.17.113@TBDS-2NGVYEHH --kr
```

```
b5-keytab-path /var/krb5kdc/emr.keytab --krb5-remote-service-name hadoop --krb5-disable-remote-service-hostname-canonicalization --catalog hive --schema default --insecure
trino:default> select * from city;
 id |   name
----+----------
  1 | city1
  2 | city2
  3 | city3
(3 rows)

Query 20240204_170407_00021_fwpar, FINISHED, 1 node
Splits: 1 total, 1 done (100.00%)
0.31 [3 rows, 386B] [9 rows/s, 1.2KB/s]
```

LDAP认证

```
# 在trino节点上，执行命令如下，其中{coordinator_ip}更换成trino coordinator组件所在节点ip
# 指定用户名后需要输入密码
# catalog用与指定要访问源，例如hive；schema则表示数据库，例如default库
[root@10 ~]# /usr/local/service/trino/client/trino-cli --server https://{coordinator_ip}:9443  --user hadoop --catalog hive --schema default --insecure  --password
Password:
trino:default> select * from city;
 id |   name
----+----------
  1 | city1
  2 | city2
  3 | city3
(3 rows)

Query 20240204_165709_00015_fwpar, FINISHED, 1 node
Splits: 1 total, 1 done (100.00%)
0.35 [3 rows, 386B] [8 rows/s, 1.07KB/s]
```

api接口

TBDS大数据平台上的Trino访问接口与开源兼容, 参考 JDBC、COMMAND。

常用命令

命令说明

操作	命令	描述
通过客户端访问trino组件/Simple认证	/usr/local/service/trino/client/trino-cli --server http://{coordinator_ip}:9000 --user hadoop --catalog hive --schema default --file ~/test.sql --execute "select * from city;" --debug	user: 必选，指定用户名 catalog：可选，指定默认访问的数据源 server：必选，指定访问coordinator URL schema：可选，指定默认访问的数据库 file: 可选，指定需要执行sql的文件路径 execute: 可选，指定执行sql命令 debug： 可选，开启客户端debug日志，方便定位问题
通过客户端访问trino组件/ldap认证	/usr/local/service/trino/client/trino-cli --server https://{coordinator_ip}:9443 --user hadoop --catalog hive --schema default --insecure --password	user: 必选，指定用户名 insecure: 必选，开启https访问 password：必选，表示需要输入密码进行身份认证 server：必选，指定访问coordinator URL catalog：可选，指定默认访问的数据源 schema：可选，指定默认访问的数据库 file: 可选，指定需要执行sql的文件路径 execute: 可选，指定执行sql命令 debug： 可选，开启客户端debug日志，方便定位问题
通过客户端访问trino组件/	/usr/local/service/trino/client/trino-cli --server https://10.206.17.113:9443 --user hadoop --krb5-config-path /etc/krb5.conf --krb5-principal	user: 必选，指定用户名 insecure: 必选，开启https访问

操作	命令	描述
kerberos认证	hadoop/10.206.17.113@TBDS-2NGVYEHH --krb5-keytab-path /var/krb5kdc/emr.keytab --krb5-remote-service-name hadoop --krb5-disable-remote-service-hostname-canonicalization --catalog hive --schema default --insecure	krb5-config-path：必选，客户端的krb5.conf配置路径，缺省值/etc/krb5.conf krb5-principal：必选，客户端的principal krb5-keytab-path: 必选，客户端的keytab路径 krb5-remote-service-name：必选，coordinator用于kerberos认证的服务名 krb5-disable-remote-service-hostname-canonicalization：可选，对于coordinator 的kerberos认证，若未使用规范化的主机名，一般需要带上该参数 server：必选，指定访问coordinator URL catalog：可选，指定默认访问的数据源 schema：可选，指定默认访问的数据库 file: 可选，指定需要执行sql的文件路径 execute: 可选，指定执行sql命令 debug： 可选，开启客户端debug日志，方便定位问题

命令示例

Simple认证

```
# 1 直接执行sql
[root@10 ~]# /usr/local/service/trino/client/trino-cli --server http://10.206.0.67:9000 --user hadoop --
```

```
catalog hive --schema default --execute "select * from city;"
"4","city4"
"3","city3"
"1","city1"
"2","city2"
"4","city4"

# 2 进入交互终端，执行sql
[root@10 ~]# /usr/local/service/trino/client/trino-cli --server http://10.206.0.67:9000 --user hadoop --
catalog hive --schema default
trino:default>
trino:default> select * from city;
 id |   name
----+-----------
  4 | city4
  3 | city3
  2 | city2
  1 | city1
  4 | city4
(5 rows)

Query 20231128_091610_00011_bbiwe, FINISHED, 3 nodes
Splits: 21 total, 21 done (100.00%)
0.29 [5 rows, 1.86KB] [17 rows/s, 6.44KB/s]
trino:default>
trino:default>
```

Kerberos认证

```
[root@10 etc]# /usr/local/service/trino/client/trino-cli --server https://10.206.17.113:9443  --user hado
op --krb5-config-path /etc/krb5.conf --krb5-principal  hadoop/10.206.17.113@TBDS-2NGVYEHH --krb
5-keytab-path /var/krb5kdc/emr.keytab --krb5-remote-service-name hadoop --krb5-disable-remote-s
ervice-hostname-canonicalization --catalog hive --schema default --insecure --execute "select * from c
ity;"
"1","city1"
"2","city2"
"3","city3"

[root@10 etc]# /usr/local/service/trino/client/trino-cli --server https://10.206.17.113:9443  --user hado
op --krb5-config-path /etc/krb5.conf --krb5-principal  hadoop/10.206.17.113@TBDS-2NGVYEHH --krb
5-keytab-path /var/krb5kdc/emr.keytab --krb5-remote-service-name hadoop --krb5-disable-remote-s
ervice-hostname-canonicalization --catalog hive --schema default --insecure
trino:default> select * from city;
 id |   name
```

```
----+----------
  1 | city1
  2 | city2
  3 | city3
(3 rows)

Query 20240204_174308_00029_fwpar, FINISHED, 1 node
Splits: 1 total, 1 done (100.00%)
0.31 [3 rows, 386B] [9 rows/s, 1.22KB/s]
```

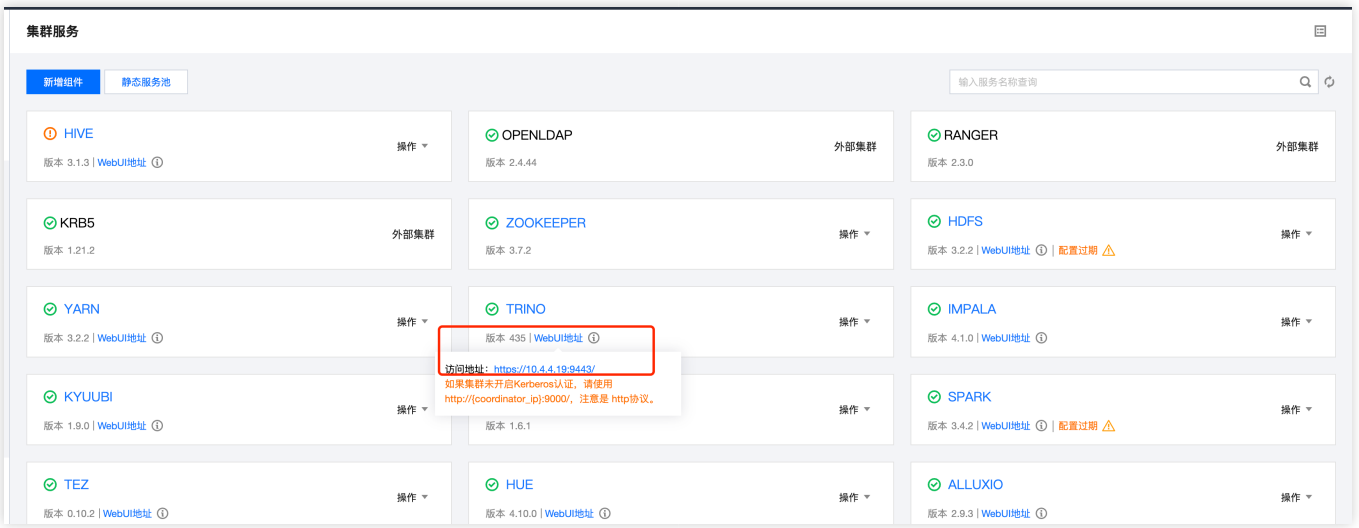
LDAP认证

```
[root@10 etc]# /usr/local/service/trino/client/trino-cli --server https://10.206.17.113:9443  --user hado
op --catalog hive --schema default --insecure  --password --execute "select * from city;"
Password:
"1","city1"
"2","city2"
"3","city3"

trino:default> select * from city;
 id |   name
----+----------
  1 | city1
  2 | city2
  3 | city3
(3 rows)

Query 20240204_174059_00025_fwpar, FINISHED, 1 node
Splits: 1 total, 1 done (100.00%)
0.27 [3 rows, 386B] [11 rows/s, 1.4KB/s]
```

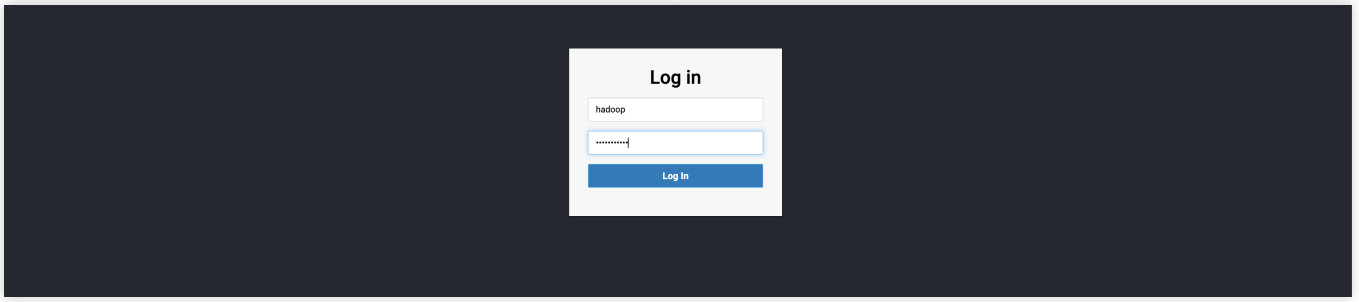
常见问题

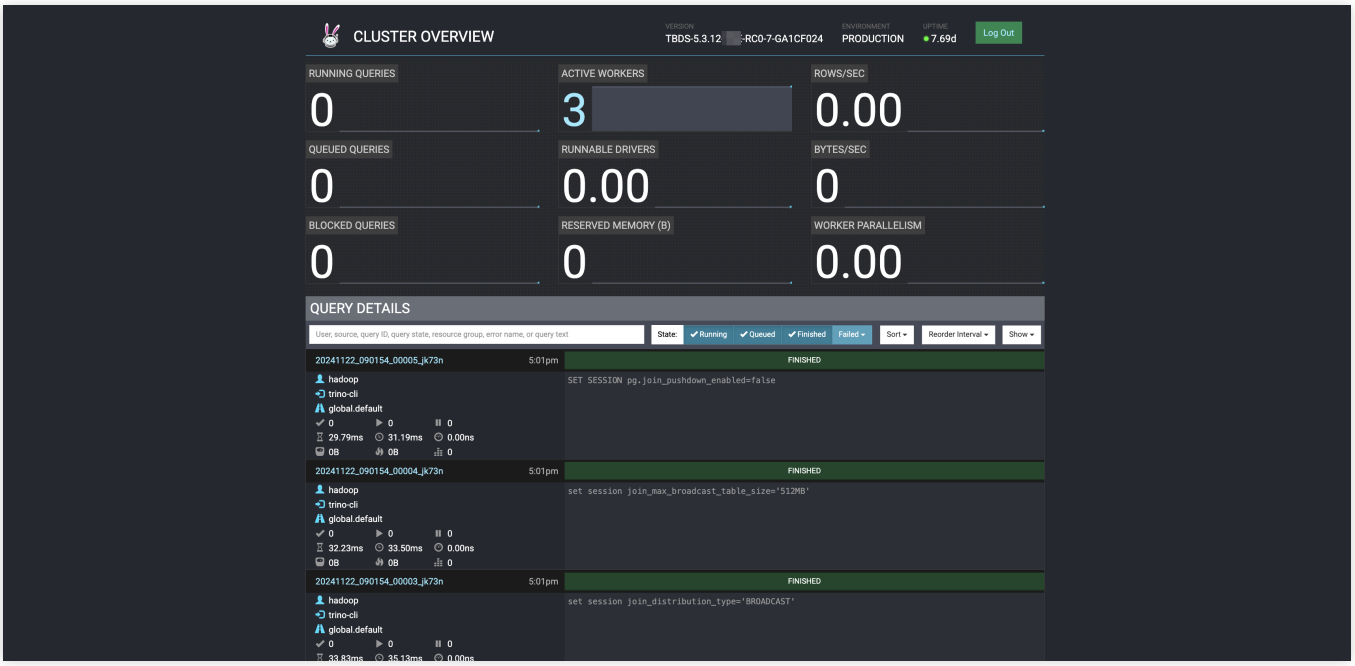
如何访问Trino UI

1. 通过 Knox 登录到 Trino UI。

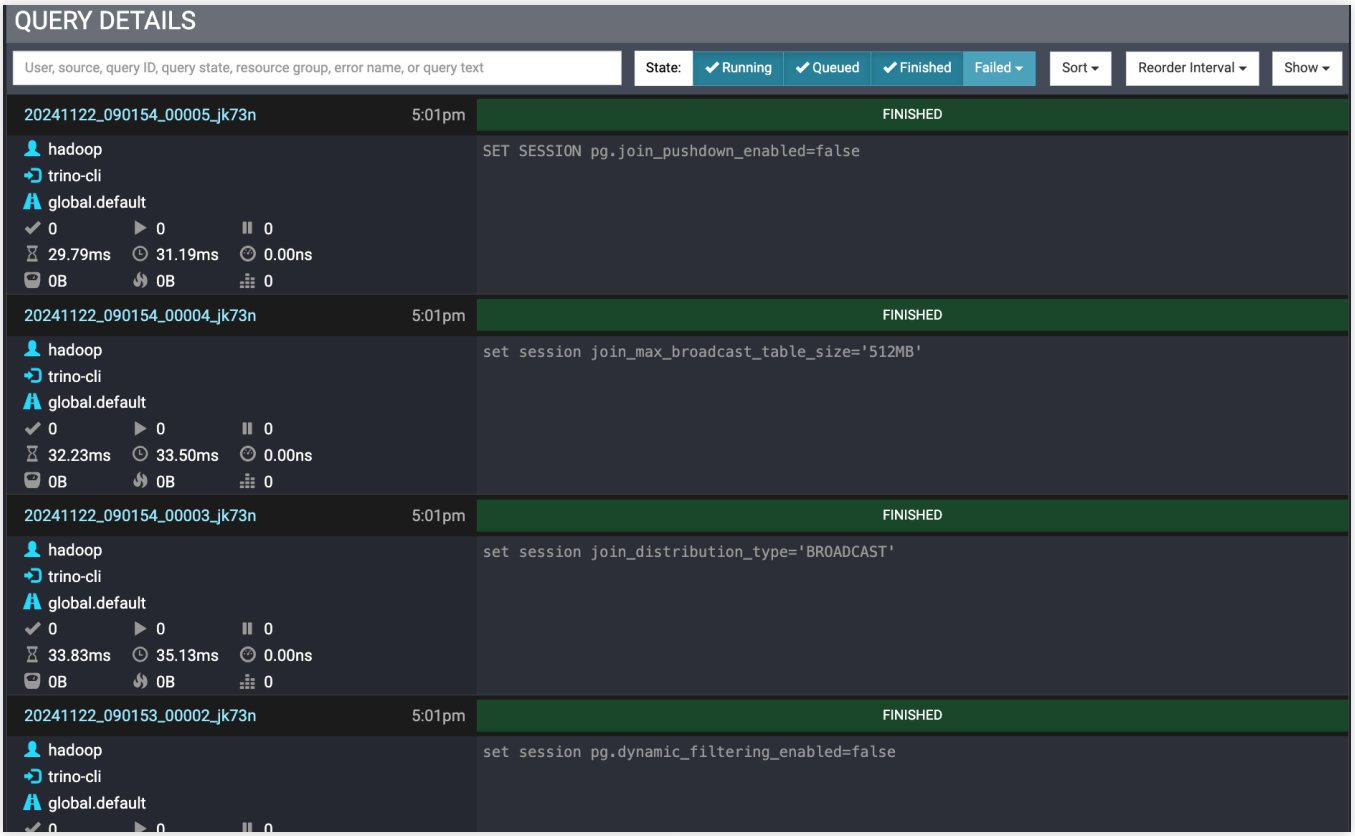


2. 输入集群账号密码。

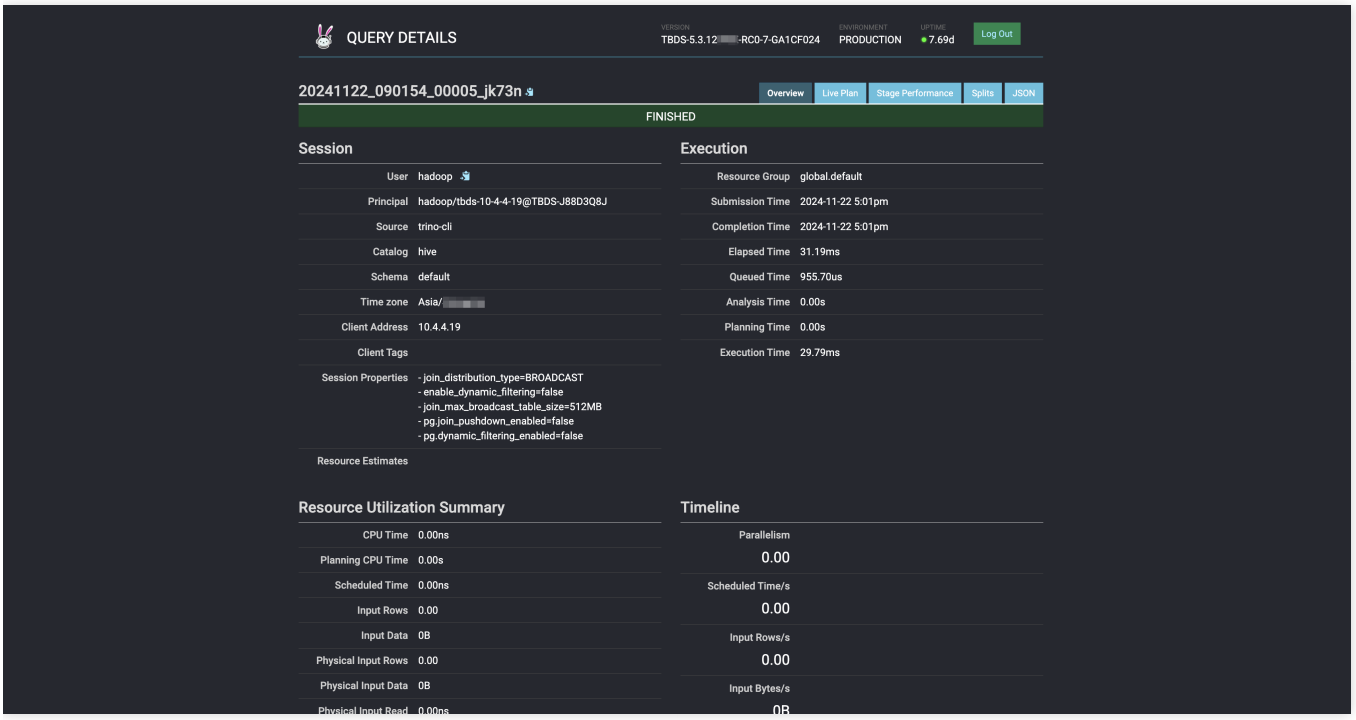




可以查看最近执行的查询任务列表，



具体的任务详情，包括：SQL ，执行计划等；



Trino只会保存最近的不超过200条的查询，可调整query.max-history配置（默认为100）以增加保存的SQL数量。

Worker节点挂掉导致报错

如果发现报错信息为：Could not communicate with the remote task. The node may have crashed or be under too much load. This is probably a transient issue, so please retry your query in a few minutes. 或No handle resolver for connector: hive ... Unrecognized token 'io': was expecting (JSON String, Number, Array, Object or token 'null', 'true' or 'false')，表示Worker节点负载太高已无法提供服务或Worker节点出现了自动重启。可能是某个Worker上的进程被系统终止，需要根据实际情况调整配置，尤其是内存相关配置，或限制并发请求数。

如何查看Trino日志

Trino的日志文件默认保存在/data/emr/trino/var/log路径下。其中，输出及异常堆栈信息均在server.log文件中。
如果仅想查看某个查询的报错详情。可以在进入client时添加--debug命令，此时即可打印异常堆栈。
Trino的Coordinator和Worker之间通过HTTP协议通信，当Coordinator节点出现HTTP返回异常时，说明报错可能出现在某台Worker节点上。此时，如果没有其他明显的异常信息，您需要逐一排查各Worker节点。

查询数据失败

请按照以下方式排查：

1. 使用Hive、Spark等其他引擎访问或查询数据。如果不能访问，需要确认数据源是否连通、数据是否完好。
2. 如果仅Trino无法访问或无法执行查询，需要检查配置的元数据信息是否正确。
3. 如果元数据正常，但查询一个有数据的表的结果为空，需要先检查您是否有数据访问权限。

- 如果数据所在的HDFS开启了proxyuser，Trino也需要开启hive.hdfs.impersonation.enabled配置。
- 如果开启了Ranger，请确认Ranger权限配置是否正确。
- 如果集群进行过扩容等操作，需要检查新增的节点组或节点是否具有访问相应文件的权限或能力。

新增配置后重启Trino失败

如果Server.log中包含了Error: Configuration property 'xxxxx' was not used，则说明您新增配置的位置不正确，或缺乏必要的前置配置。Trino对配置项的校验非常严格，如果新增的配置不存在、配置写错，或配错文件，都会导致配置无法识别，Trino无法启动，所以请您仔细检查新增的配置是否正确，或进行回滚操作。

使用Hive连接器查询Iceberg、Hudi或Delta Lake表时会报错Cannot query xxx table

针对Iceberg、Hudi和Delta Lake，Trino分别提供了单独的连接器。建议您使用各自的独立连接器来执行查询。如果您的作业必须使用Hive连接器，请使用提供的Table Redirection功能将查询转发到相应的独立连接器上。

Impala开发
概述

Apache Impala 项目为存储在 Apache Hadoop 文件格式的数据提供高性能、低延迟的 SQL 查询。它对查询进行快速响应，同时支持对分析查询进行交互式的数据探索和查询调整，而不是传统上那种与 SQL-on-Hadoop 技术相关联的长时间批量作业。Impala 不同于 hive，hive 底层执行使用的是 MapReduce 引擎，仍然是一个批处理过程。而 impala 的中间结果不写入磁盘，即时通过网络以流的形式传递，大大降低了节点的 IO 开销。Impala 与 Apache Hive 数据库集成，在两个组件之间共享数据库和表。通过与 Hive 的高度集成，以及与 HiveQL 语法的兼容性，您可以使用 Impala 或 Hive 创建表、发起查询、加载数据等。

示例工程开发

环境准备

开发环境准备

准备项	说明
安装JDK	JDK 8、JDK11 ，推荐使用 konaJDK，下载地址
安装和配置IDE	按需选择，比如 IntelliJ IDEA 或 Eclipse，示例使用IDEA
安装 Maven	开发环境基础配置，负责构建 Java 应用程序
Maven 配置准备	如果需要本地调试，需要参考 示例说明 配置 Maven settings.xml，推荐 Maven 3.6.3，下载地址

导入示例工程代码

以下以 IntelliJ IDEA 举例，将示例工程代码导入进行说明。

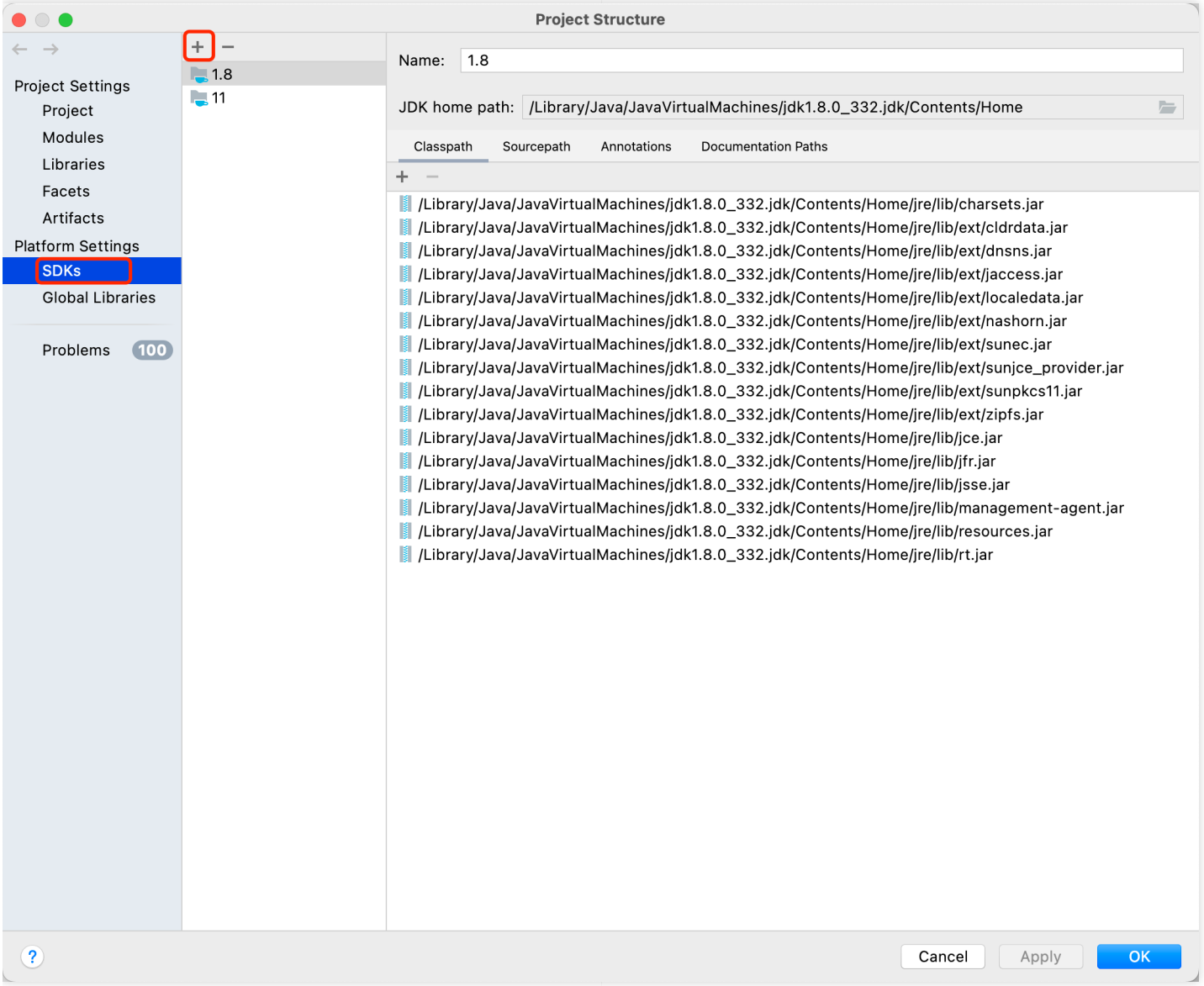
下载样例代码

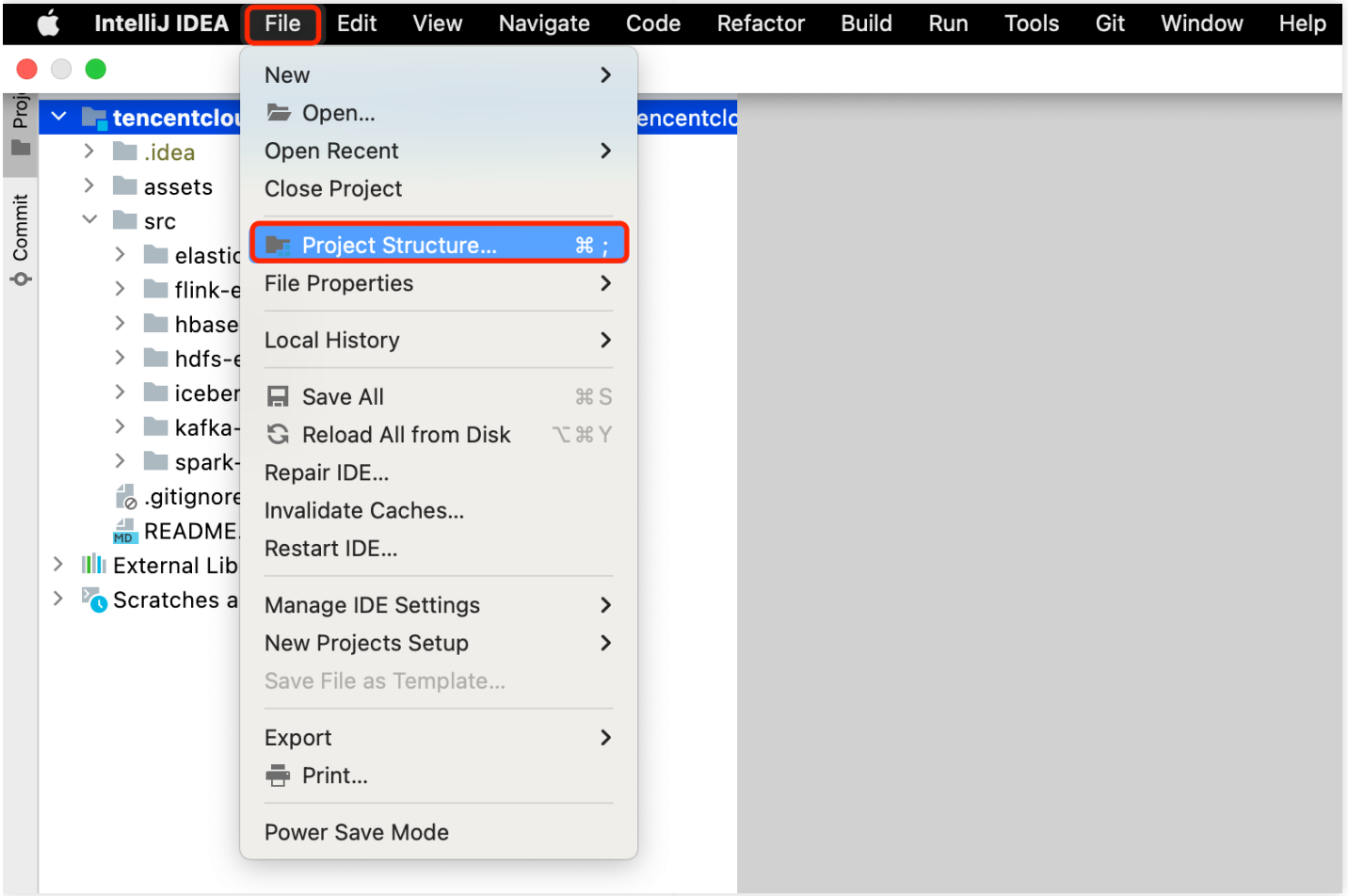
https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master

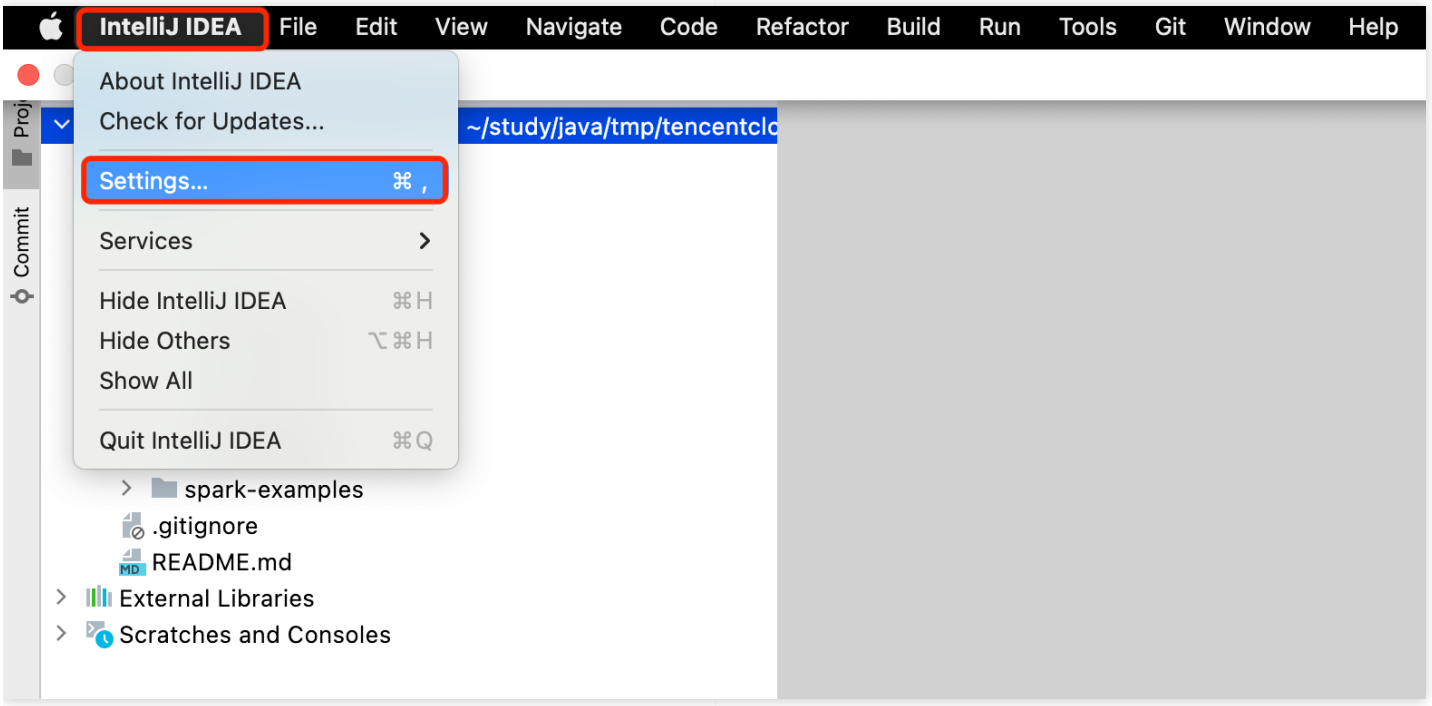
克隆或者直接下载master代码都可以

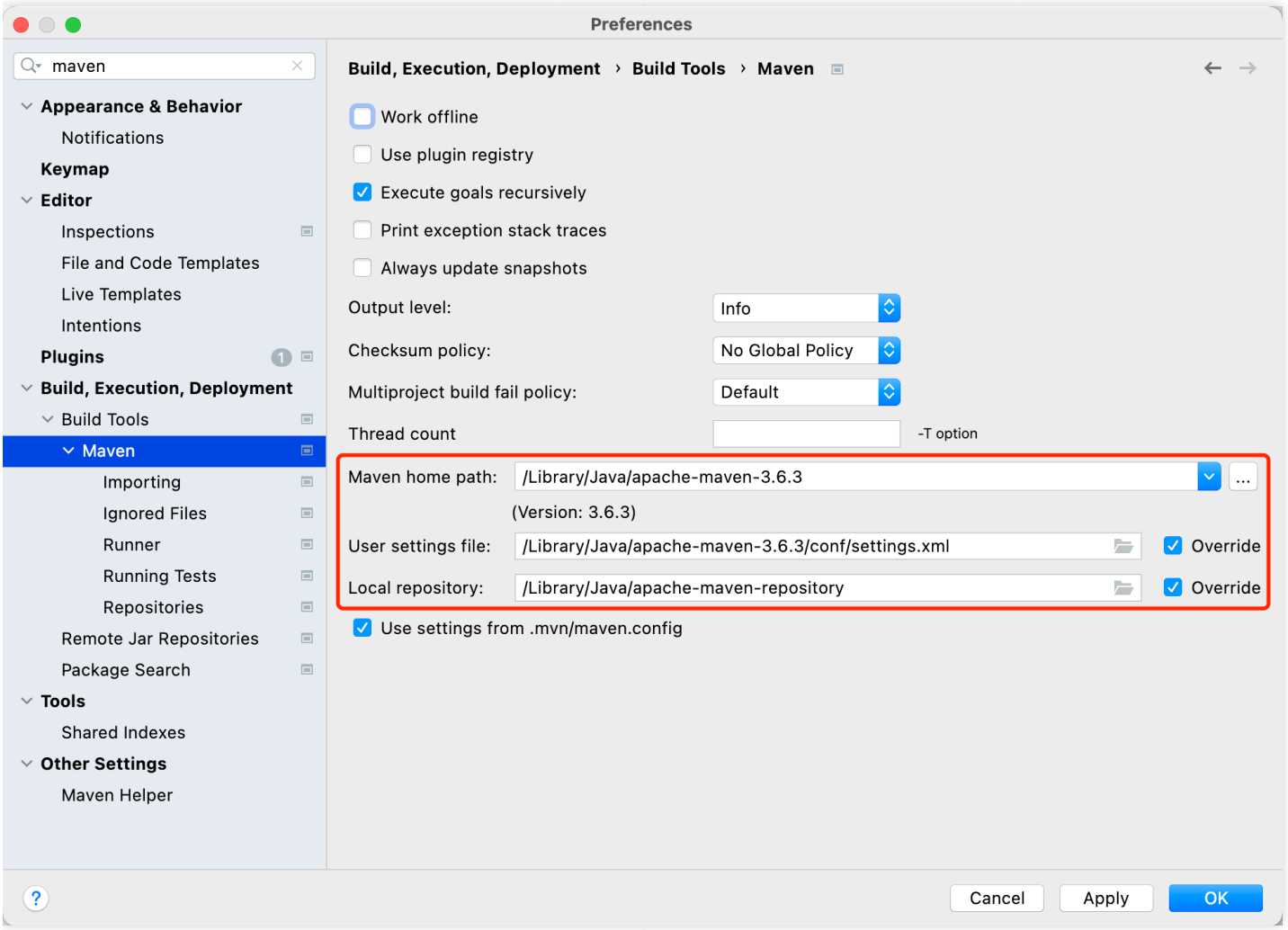
git clone https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples

导入项目，然后选择JDK、MAVEN和settings文件


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
100
1.8
11
Name:
1.8
JDK home path:
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home
Classpath
Sourcepath
Annotations
Documentation Paths
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/charsets.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/cldrdata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/dnsns.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/jaccess.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/localedata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/nashorn.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunec.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunjce_provider.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunpkcs11.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/zipfs.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jce.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jfr.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jsse.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/management-agent.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/resources.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/rt.jar
Cancel
Apply
OK


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
New
Open...
Open Recent
Close Project
Project Structure... ⌘ ;
File Properties
Local History
Save All ⌘S
Reload All from Disk ⌥⌘Y
Repair IDE...
Invalidate Caches...
Restart IDE...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
.idea
assets
src
.gitignore
README
External Lib
Scratches a
Proj
Commit


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
About IntelliJ IDEA
Check for Updates...
Settings... ⌘ ,
Services
Hide IntelliJ IDEA ⌘H
Hide Others ⌥⌘H
Show All
Quit IntelliJ IDEA ⌘Q
~/study/java/tmp/tencentclo
spark-examples
.gitignore
README.md
External Libraries
Scratches and Consoles
Proj
Commit



样例代码说明

功能说明

演示的实例是样例代码通过JDBC的方式连接和访问Impala引擎

代码逻辑说明

请将 src/impala-examples/impala-jdbc-examples/pom.xml 中的 hadoop.version 升级到您使用的版本。
5313 版本对应：<hadoop.version>3.2.2-TBDS-5.3.1.3</hadoop.version>

```
public class ImpalaJDBCExample {
```

```
    private static final Logger LOGGER = LoggerFactory.getLogger(ImpalaJDBCExample.class);

    public static void main(String[] args) {
        // 参数校验
        if (args.length < 5) {
            System.err.println(
                    "Kerberos Usage: ImpalaJDBCExample <impala-host> <impala-port> <principal> <keytab
> <sql>");
            System.exit(1);
        }
        // args 共5个
        // args[0]、args[1]表示impala daemon的ip和端口，拼接jdbc的url
        // args[2]、args[3]分别是用户使用的principal和keytab，用于kerberos认证
        // args[4] 为要执行的 sql 语句

        final String impalaHost = args[0];
        final String impalaPort = args[1];
        String principal = args[2];
        String keytab = args[3];
        final String sql = args[4];
        final String krbRealm = principal.substring(principal.lastIndexOf("@") + 1);

        // kerberos认证
        LOGGER.info("kerberos auth begin");
        try {
            Configuration conf = new Configuration();
            conf.set("hadoop.security.authentication", "Kerberos");
            UserGroupInformation.setConfiguration(conf);
            UserGroupInformation.loginUserFromKeytab(principal, keytab);
        } catch (IOException e) {
            LOGGER.error("kerberos auth error", e);
        }
        LOGGER.info("kerberos auth end");
        UserGroupInformation loginUser = null;
        try {
            loginUser = UserGroupInformation.getLoginUser();
        } catch (IOException e) {
            LOGGER.error("get login user error", e);
            System.exit(1);
        }

        loginUser.doAs(new PrivilegedAction<Object>() {
            public Object run() {
                // 加载驱动
                try {
                    Class.forName("com.cloudera.impala.jdbc.Driver");
                } catch (Exception e) {
```

```
                LOGGER.error("driver class load error", e);
            }

            Connection connection = null;
            try {
                // 创建连接
                String url = String.format(
                        "jdbc:impala://%s:%s/;AuthMech=1;KrbRealm=%s;KrbHostFQDN=%s;KrbServiceNam
e=hadoop",
                        impalaHost, impalaPort, krbRealm, impalaHost);
                LOGGER.info("url = {}", url);
                connection = DriverManager.getConnection(url);
                LOGGER.info("jdbc connection created");

                // 执行SQL
                Statement statement = connection.createStatement();
                ResultSet resultSet = statement.executeQuery(sql);
                int columnSize = resultSet.getMetaData().getColumnCount();
                LOGGER.info("======execute sql result======");
                while (resultSet.next()) {
                    for (int j = 1; j < columnSize; j++) {
                        LOGGER.info(resultSet.getString(j));
                    }
                }
            } catch (Exception e) {
                LOGGER.error("execute sql error", e);
            } finally {
                try {
                    if (connection != null) {
                        connection.close();
                    }
                } catch (SQLException e) {
                    LOGGER.error("close connection error", e);
                }
            }

            return null;
        }
    });
  }
}
```

编译出包，拷贝到 TE 机器，执行以下命令连接 impala 查询。

```
java -jar ImpalaJDBCExample-1.0.jar ${impala_daemon_hostname} 27009 hadoop/10-206-0-90@TBD
S-3AP6176L /var/krb5kdc/emr.keytab  "show databases"
```

示例说明

pom.xml 配置

src/impala-examples/impala-jdbc-examples/pom.xml 中的 hadoop.version 升级到：<hadoop.version>3.2.2-TBDS-5.3.1.3</hadoop.version>。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tencent.tbds</groupId>
    <artifactId>ImpalaJDBCExample</artifactId>
    <version>1.0</version>
    <name>impala-jdbc-examples</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <hadoop.version>3.2.2-TBDS-5.3.1.3</hadoop.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>Impala</groupId>
            <artifactId>ImpalaJDBC42</artifactId>
            <version>2.6.26.1031</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-client</artifactId>
            <version>${hadoop.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.17.2</version>
        </dependency>
    </dependencies>

    <build>
```

```
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.4</version>
        <configuration>
          <keepDependenciesWithProvidedScope>true</keepDependenciesWithProvidedScope>
          <createDependencyReducedPom>true</createDependencyReducedPom>
          <filters>
            <filter>
              <artifact>*:*</artifact>
              <excludes>
                <exclude>META-INF/*.SF</exclude>
                <exclude>META-INF/*.DSA</exclude>
                <exclude>META-INF/*.RSA</exclude>
                <exclude>**/Log4j2Plugins.dat</exclude>
              </excludes>
            </filter>
          </filters>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <transformers>
                <transformer
                    implementation="org.apache.maven.plugins.shade.resource.ManifestResourc
eTransformer">
                  <mainClass>com.tencent.tbds.ImpalaJDBCExample</mainClass>
                </transformer>
              </transformers>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Maven settings.xml配置

```
<?xml version="1.0" encoding="UTF-8"?>

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/set
tings-1.0.0.xsd">

    <!-- localRepository
    | The path to the local repository maven will use to store artifacts.
    |
    | Default: ${user.home}/.m2/repository
    <localRepository>/path/to/local/repo</localRepository>
    -->
    <pluginGroups>

    </pluginGroups>

    <proxies>

    </proxies>

    <servers>

    </servers>
    <!--
    <mirrors>
      <mirror>
        <id>central</id>
        <name>Maven Repository Switchboard</name>
        <url>http://repo1.maven.org/maven2/</url>
        <mirrorOf>central</mirrorOf>
      </mirror>
      <mirror>
        <id>repo2</id>
        <mirrorOf>central</mirrorOf>
        <name>Human Readable Name for this Mirror.</name>
        <url>http://repo2.maven.org/maven2/</url>
      </mirror>
    </mirrors>
    -->
    <profiles>
    <!-- define a profile and tbds public repository -->
      <profile>
      <id>tbds-profile</id>
        <repositories>
```

```
            <repository>
                <id>tbds_public</id>
                <name>tbds_public_repository</name>
                <url>https://tbdsrepo.cloud.tencent.com/repository/tbds/</url>
                <layout>default</layout>
            </repository>
            <repository>
                <id>maven_public</id>
                <name>maven_public_repository</name>
                <url>https://tbdsrepo.cloud.tencent.com/repository/maven-public/</url>
                <layout>default</layout>
            </repository>
        </repositories>
    </profile>
</profiles>

<activeProfiles>
<!-- make sure tbds-profile active -->
    <activeProfile>tbds-profile</activeProfile>
</activeProfiles>

</settings>
```

api接口

TBDS大数据平台上的 Impala 访问接口与开源兼容, 参考 JDBC、Impala Shell

常用命令

连接方式

kerberos 连接

kerberos 认证（Simple 不需要执行）

```
klist -kt /var/krb5kdc/emr.keytab
kinit -kt /var/krb5kdc/emr.keytab hadoop/{IP}@TBDS-{域名}
```

使用 hostname 连接

```
impala-shell -i {impalad_hostname}:27009 -s hadoop
```

使用 ip 连接

- hostname 为 ip

```
impala-shell -i {impalad_ip}:27009 -s hadoop
```

- hostname 不为 ip

```
impala-shell -i {impalad_ip}:27009 -s hadoop -b {impalad_hostname}
```

ldap 连接

```
impala-shell -i {impalad_hostname} -l -u user --auth_creds_ok_in_clear
```

命令示例

- 查看数据库

```
[10.206.0.90:27009] default> show databases;
Query: show databases
+------------------+-----------------------------------------------+
| name             | comment                                       |
+------------------+-----------------------------------------------+
| _impala_builtins | System database for Impala builtin functions |
| default          | Default Hive database                         |
+------------------+-----------------------------------------------+
```

- 创建数据库

```
[10.206.0.90:27009] default> create database impala_test;
Query: create database impala_test
+---------------------------+
| summary                   |
+---------------------------+
| Database has been created. |
+---------------------------+
```

- 切换到 impala_test 数据库

```
[10.206.0.90:27009] default> use impala_test;
Query: use impala_test
```

- 查看当前用户

```
[10.206.0.90:27009] impala_test> select current_user();
Query: select current_user()
Query submitted at: 2023-11-28 19:21:57 (Coordinator: http://10.206.0.90:27004)
Query progress can be monitored at: http://10.206.0.90:27004/query_plan?query_id=8f41325d0ded69
d8:ca8b3db500000000
+---------------------------------+
| current_user()                  |
+---------------------------------+
| hadoop/10.206.0.67@TBDS-3AP6176L |
+---------------------------------+
```

- 创建hive表

```
[10.206.0.90:27009] impala_test> create table t1 (id int, name string);
Query: create table t1 (id int, name string)
+------------------------+
| summary                |
```

```
+------------------------+
| Table has been created. |
+------------------------+
```

- 查看库中的表

```
[10.206.0.90:27009] impala_test> show tables;
Query: show tables
+------+
| name |
+------+
| t1   |
+------+
```

- 查看hive表结构

```
[10.206.0.90:27009] impala_test> show create table t1;
Query: show create table t1
+----------------------------------------------------------------------------+
| result                                                                     |
+----------------------------------------------------------------------------+
| CREATE TABLE impala_test.t1 (                                              |
|   id INT,                                                                  |
|   name STRING                                                              |
| )                                                                          |
| STORED AS TEXTFILE                                                         |
| LOCATION 'hdfs://HDFS78000002/usr/hive/warehouse/impala_test.db/t1'    |
| TBLPROPERTIES ('OBJCAPABILITIES'='EXTREAD,EXTWRITE', 'accessType'='8') |
+----------------------------------------------------------------------------+
```

- 插入数据

```
[10.206.0.90:27009] impala_test> insert into t1 values(1,'a');
```

- 查询hive表

```
[10.206.0.90:27009] impala_test> select * from t1;
Query: select * from t1
Query submitted at: 2023-11-28 19:26:06 (Coordinator: http://10.206.0.90:27004)
Query progress can be monitored at: http://10.206.0.90:27004/query_plan?query_id=ff47e59f75f2c71
8:81190e4900000000
+----+------+
| id | name |
```

```
+----+------+
| 1 | a    |
+----+------+
```

- 增量刷新hive表的元数据

[10.206.0.90:27009] impala_test> refresh t1;

- 销毁impala中的元数据缓存，下次使用该表将全量拉取

[10.206.0.90:27009] impala_test> invalidate metadata t1;

常见问题

为了提升查询性能，impala 使用了元数据缓存进行加速，但同时也带来了元数据不一致的问题，常见有两个问题.

从其引擎新建的库表 Impala 中看不到

在其他引擎执行的DDL，即插入/删除库表、修改表结构等之后，需要在 impala 中执行 invalidate metadata 销毁对应库表的缓存，待下次查询相关表时会重新加载元数据。

```
invalidate metadata table_name;
```

Impala和其他引擎的查询结果不同

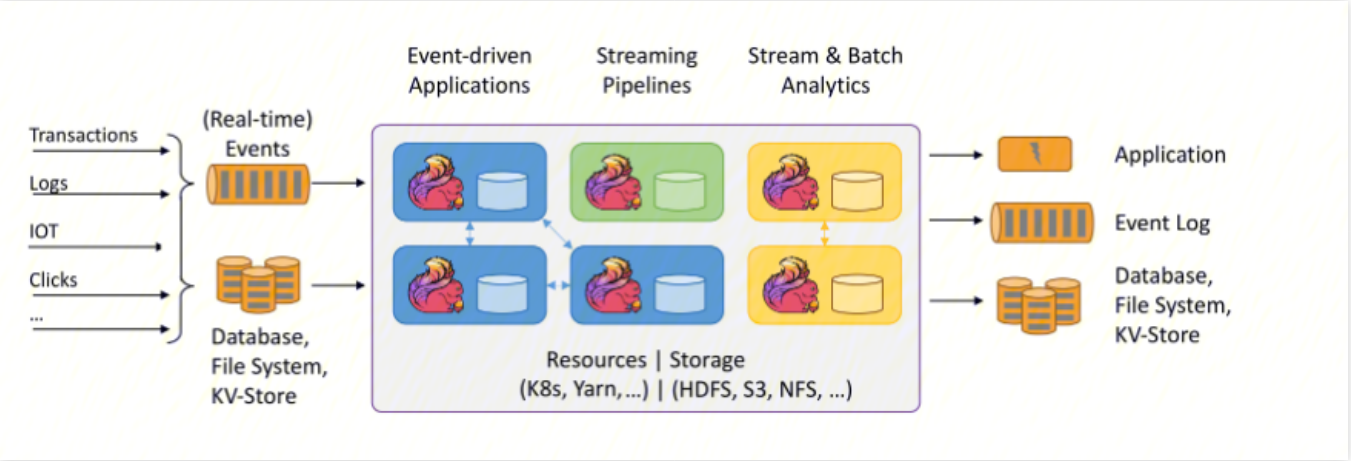
在其他引擎执行的写入数据、修改分区或者通过etl等直接修改文件的方式导入数据，需要在 impala 中执行 refresh 增量更新对应的元数据，该操作的性能会显著优于 invalidate metadata。

```
refresh table_name;
```

Flink开发
概述

Flink 核心是一个开源的分布式、高性能、高可用、准确的数据流执行引擎，其针对数据流的分布式计算提供了数据分布、数据通信以及容错机制等功能。基于流执行引擎，Flink 提供了更高抽象层的 API 以便您编写分布式任务。

- 分布式：表示 Flink 程序可以运行在多台机器上。
- 高性能：表示 Flink 处理性能比较高。
- 高可用：表示 Flink 支持程序的自动重启机制。
- 准确的：表示 Flink 可以保证处理数据的准确性。



上图中左边是数据源，从这里可以看出来，这些数据是实时生产的一些日志，或者是数据库，文件系统，kv 存储系统中的数据。中间是 Flink，负责对数据进行梳理。右边是目的地，Flink 可以将计算好的数据输出到其它应用系统中，或者存储系统中。Flink 的三大核心组件如下：

- Data Source：也就是图中左边的数据源。
- Transformations：算子（负责对数据进行处理）。
- Data Sink：输出组件（负责把计算好的数据输出到其它应用系统中）。

示例工程开发

环境准备

开发环境准备

准备项	说明
安装JDK	JDK 8、JDK11 ，推荐使用 konaJDK，下载地址
安装和配置 IDE	按需选择，比如 IntelliJ IDEA 或 Eclipse，示例使用IDEA
安装 Maven	开发环境基础配置，负责构建 Java 应用程序
Maven 配置准备	如果需要本地调试，需要参考 6.开发环境准备 配置 Maven settings.xml，推荐 Maven 3.6.3，下载地址

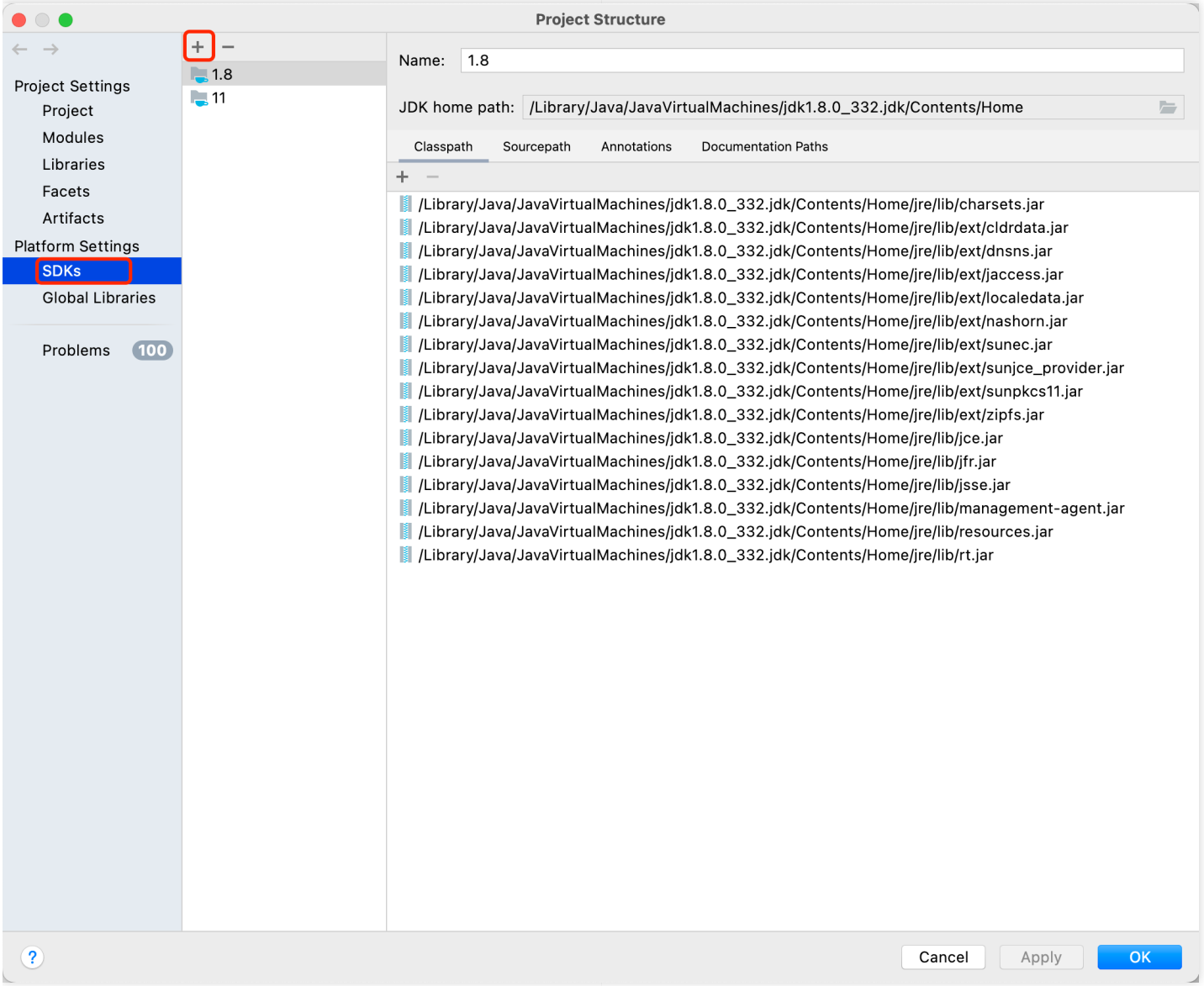
导入示例工程代码

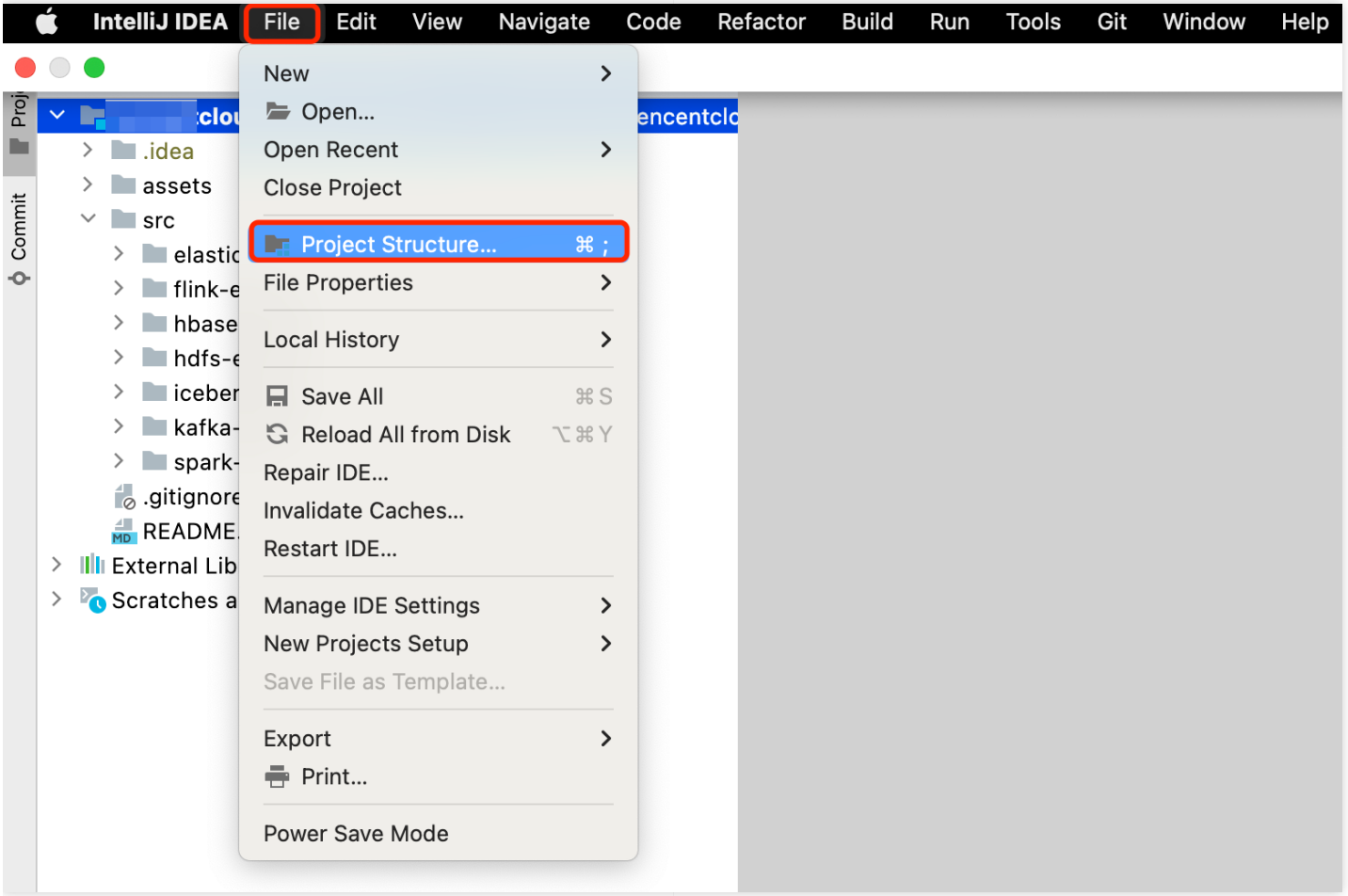
以下以 IntelliJ IDEA 举例，将示例工程代码导入进行说明。
2.1 下载样例代码：https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master

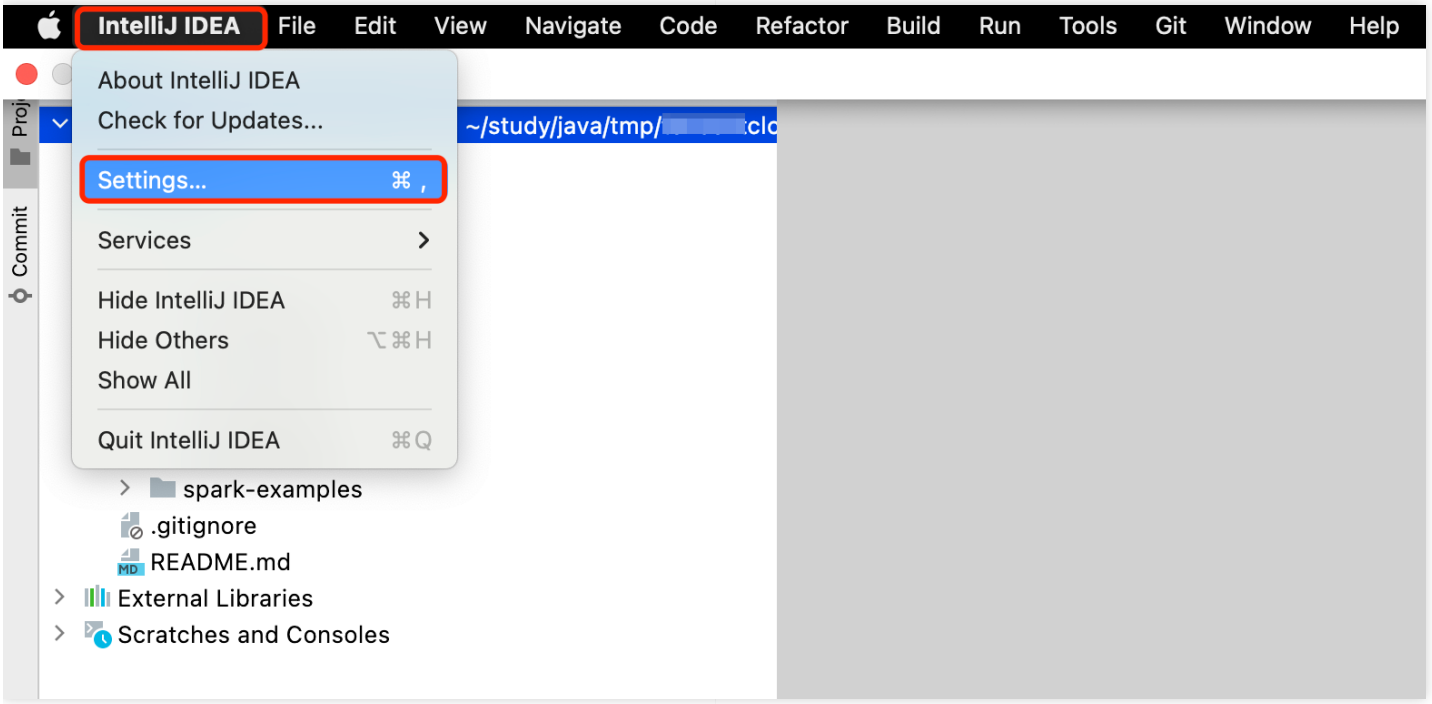
克隆或者直接下载master代码都可以

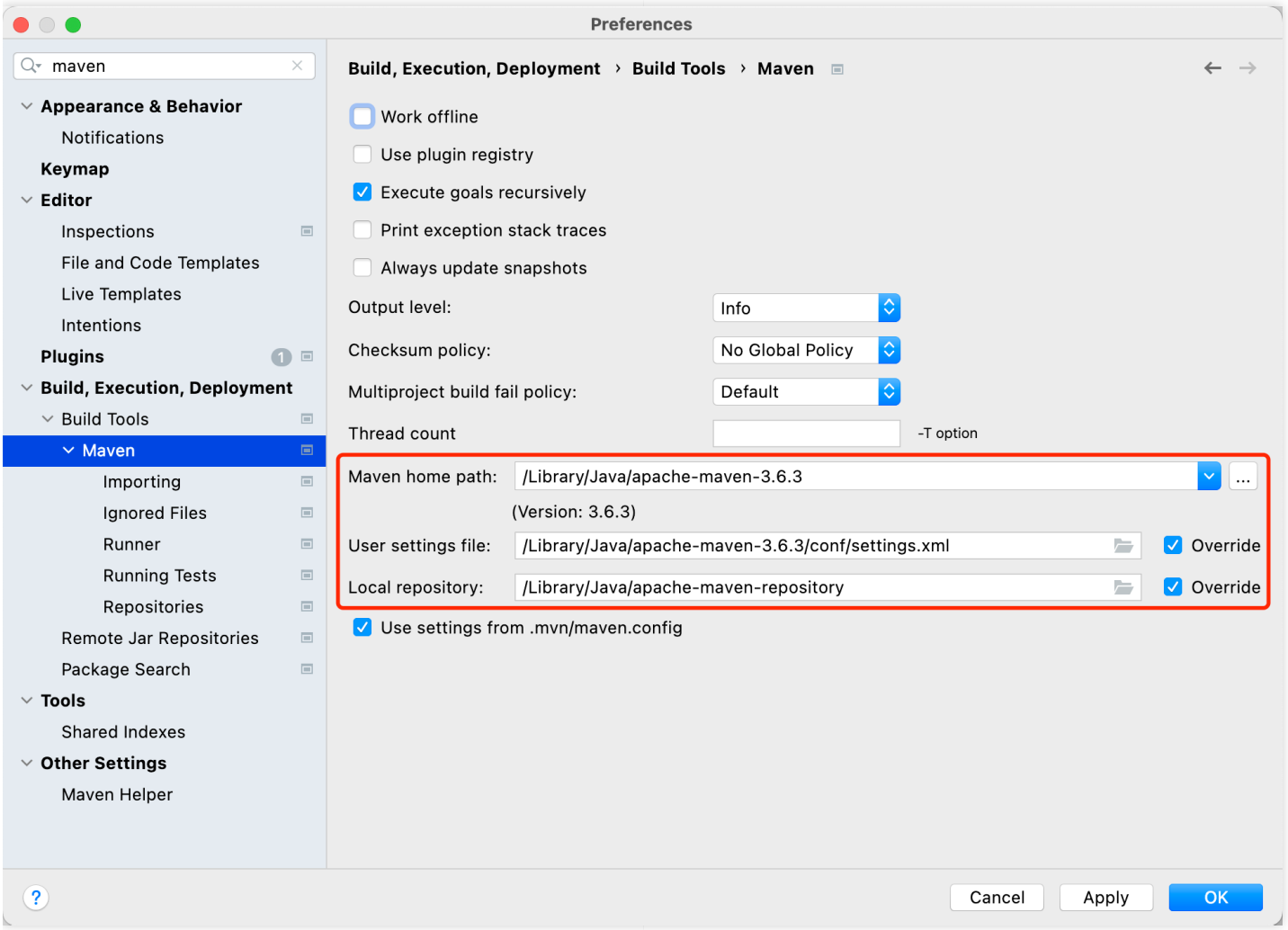
git clone https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples

2.2 导入项目，然后选择JDK、MAVEN和settings文件。


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
100
1.8
11
Name:
1.8
JDK home path:
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home
Classpath
Sourcepath
Annotations
Documentation Paths
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/charsets.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/cldrdata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/dnsns.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/jaccess.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/localedata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/nashorn.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunec.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunjce_provider.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunpkcs11.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/zipfs.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jce.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jfr.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jsse.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/management-agent.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/resources.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/rt.jar
Cancel
Apply
OK


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
New
Open...
Open Recent
Close Project
Project Structure... ⌘ ;
File Properties
Local History
Save All ⌘S
Reload All from Disk ⌥⌘Y
Repair IDE...
Invalidate Caches...
Restart IDE...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
.idea
assets
src
.gitignore
README
External Lib
Scratches a
Proj
Commit


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
About IntelliJ IDEA
Check for Updates...
Settings... ⌘ ,
Services
Hide IntelliJ IDEA ⌘H
Hide Others ⌥⌘H
Show All
Quit IntelliJ IDEA ⌘Q
~/study/java/tmp/
spark-examples
.gitignore
README.md
External Libraries
Scratches and Consoles
Proj
Commit



样例代码

POM 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tencent.tbds</groupId>
    <artifactId>CountWindowApp</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
<properties>
    <flink.version>1.16.1-TBDS-5.3.1.3</flink.version>
    <java.version>1.8</java.version>
    <maven.compiler.source>${java.version}</maven.compiler.source>
    <maven.compiler.target>${java.version}</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-clients</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
        <version>2.11.12</version>
    </dependency>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.1.1</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <createDependencyReducedPom>false</createDependencyReducedPom>

                        <artifactSet>
                            <excludes>
                                <exclude>com.google.code.findbugs:jsr305</exclude>
                                <exclude>org.slf4j:*</exclude>
```

```
                    <exclude>log4j:*</exclude>
                </excludes>
            </artifactSet>
            <filters>
                <filter>
                    <!-- Do not copy the signatures in the META-INF folder.
                    Otherwise, this might cause SecurityExceptions when using the JAR. -->
                    <artifact>*:*</artifact>
                    <excludes>
                        <exclude>META-INF/*.SF</exclude>
                        <exclude>META-INF/*.DSA</exclude>
                        <exclude>META-INF/*.RSA</exclude>
                    </excludes>
                </filter>
            </filters>
            <transformers>
                <!-- 注意不要漏掉这个transformer，否则运行时会报类似"Could not find a suitable ta
ble factory
                for 'org.apache.flink.table.delegation.ExecutorFactory" 的错误-->
                <transformer
                        implementation="org.apache.maven.plugins.shade.resource.ServicesResource
Transformer"/>

                <transformer
                        implementation="org.apache.maven.plugins.shade.resource.ManifestResourc
eTransformer">
                    <mainClass>com.tencent.tbds.CountWindowApp</mainClass>
                </transformer>
            </transformers>
        </configuration>
    </execution>
</executions>
</plugin>
</plugins>
</build>
</project>
```

示例代码

```
public class CountWindowApp {

    private static final Logger LOGGER = LoggerFactory.getLogger(CountWindowApp.class);
    private static Boolean cancel = false;
```

```
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
                StreamExecutionEnvironment.getExecutionEnvironment();
        // env.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
        DataStreamSource<String> ds =
                env.addSource(new RichParallelSourceFunction<String>() {
                    @Override
                    public void run(SourceContext<String> sourceContext)
                            throws Exception {
                        while (!cancel) {
                            String source = RandomStringUtils.random(8, true, true);
                            sourceContext.collect(source);
                            LOGGER.info("generate source: {} ,subtask: {}", source,
                                    getRuntimeContext().getIndexOfThisSubtask());
                            TimeUnit.SECONDS.sleep(5);
                        }
                    }
                    @Override
                    public void cancel() {
                        LOGGER.info("source cancel new ... ");
                        cancel = true;
                    }
                }).setParallelism(1);
        // 滚动窗口每 5 个数据产生一个窗口
        AllWindowedStream<String, GlobalWindow> countWindowAll = ds.countWindowAll(5);
        // 滑动窗口每 3 个数据产生一个包含前 5 个数据的窗口
        // AllWindowedStream<String, GlobalWindow> countWindowAll = ds.countWindowAll(5, 3);
        SingleOutputStreamOperator<List<Tuple2<String, Integer>>> aggregate =
                countWindowAll.aggregate(
                        new AggregateFunction<String, List<Tuple2<String, Integer>>, List<Tuple2<String, Inte
ger>>>() {
                            @Override
                            public List<Tuple2<String, Integer>> createAccumulator() {
                                return new ArrayList<>();
                            }
                            @Override
                            public List<Tuple2<String, Integer>> add(String s,
                                                                     List<Tuple2<String, Integer>> acc) {
                                acc.add(Tuple2.apply(s, 1));
                                return acc;
                            }
                            @Override
                            public List<Tuple2<String, Integer>> getResult(
                                    List<Tuple2<String, Integer>> acc) {
                                return acc;
                            }
                            @Override
```

```
                public List<Tuple2<String, Integer>> merge(
                        List<Tuple2<String, Integer>> acc1,
                        List<Tuple2<String, Integer>> acc2) {
                    acc1.addAll(acc2);
                    return acc1;
                }
            });

        aggregate.addSink(new RichSinkFunction<List<Tuple2<String, Integer>>>() {
            @Override
            public void invoke(List<Tuple2<String, Integer>> value,
                               Context context) throws Exception {
                LOGGER.info("sink => {}", value);
            }

        });
        env.execute("CountWindowApp");
    }
}
```

示例说明

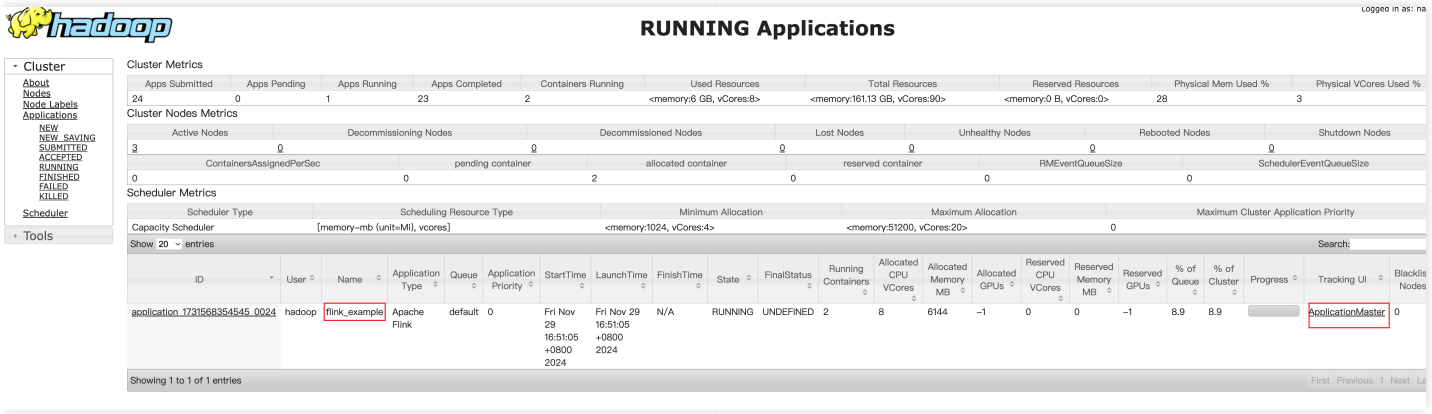
启动任务

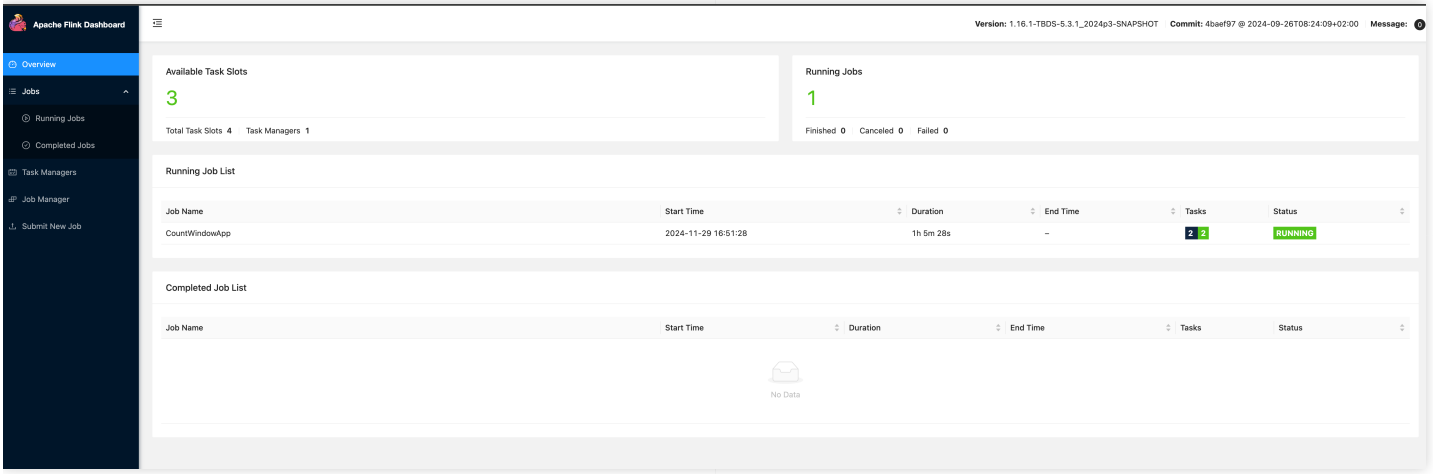
```
/usr/local/service/flink/bin/yarn-session.sh -s 4 -jm 2048 -tm 4096 -nm flink_example -d
/usr/local/service/flink/bin/flink run CountWindowApp-1.0-SNAPSHOT.jar -m yarn-cluster -d
```

查询任务详细进度和执行计划

YARN UI 上找到对应的 Application，也可在 YARN UI界面跳转到Flink UI。

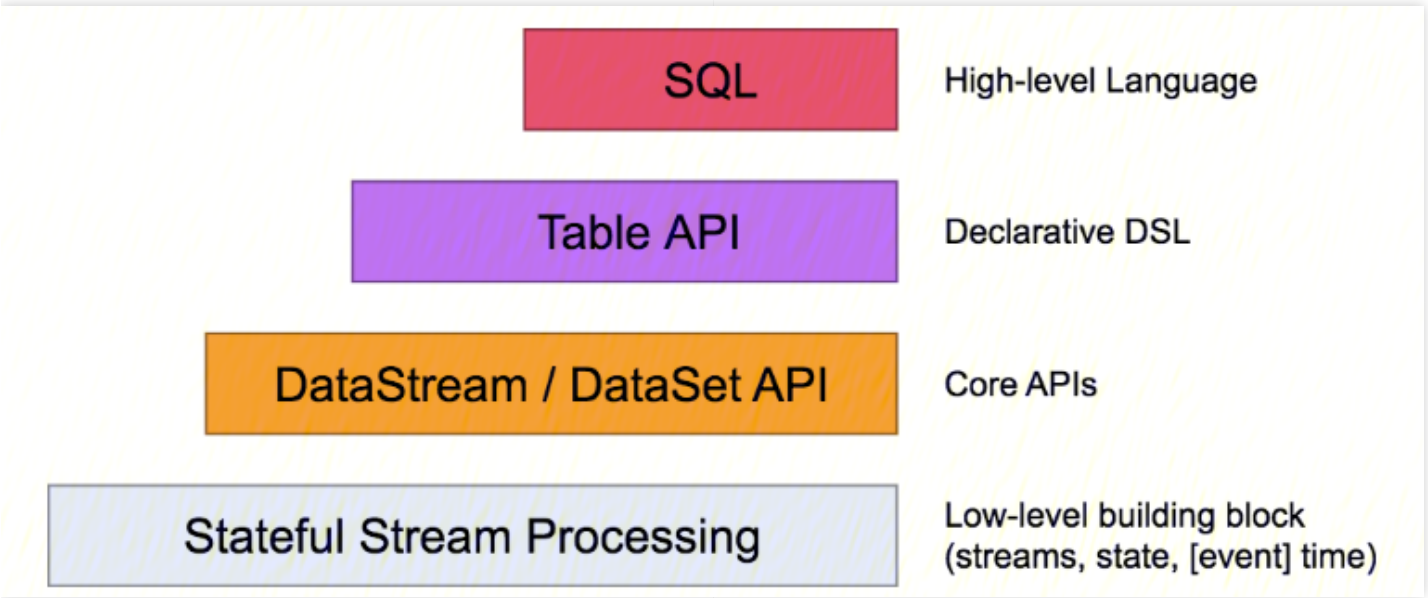
```
2024-11-29 16:51:05,093 INFO  org.apache.flink.yarn.YarnClusterDescriptor                  [] - Submitting application master application_1731568354545_0024
2024-11-29 16:51:05,324 INFO  org.apache.hadoop.yarn.client.api.impl.YarnClientImpl        [] - Submitted application application_1731568354545_0024
2024-11-29 16:51:05,324 INFO  org.apache.flink.yarn.YarnClusterDescriptor                  [] - Waiting for the cluster to be allocated
2024-11-29 16:51:05,325 INFO  org.apache.flink.yarn.YarnClusterDescriptor                  [] - Deploying cluster, current state ACCEPTED
2024-11-29 16:51:19,393 INFO  org.apache.flink.yarn.YarnClusterDescriptor                  [] - YARN application has been deployed successfully.
2024-11-29 16:51:19,394 INFO  org.apache.flink.yarn.YarnClusterDescriptor                  [] - Found Web Interface tbds-10-4-4-47:46661 of application 'application_1731568354545_0024'.
JobManager Web Interface: https://tbds-10-4-4-47:46661
2024-11-29 16:51:19,760 INFO  org.apache.flink.yarn.cli.FlinkYarnSessionCli                [] - The Flink YARN session cluster has been started in detached mode. In order to stop Flink gracefully, use the following
 command:
$ echo "stop" | ./bin/yarn-session.sh -id application_1731568354545_0024
If this should not be possible, then you can also kill Flink via YARN's web interface or via:
$ yarn application -kill application_1731568354545_0024
Note that killing Flink might not clean up all job artifacts and temporary files.
[root@tbds-10-4-4-19 ~]# /usr/local/service/flink/bin/flink run CountWindowApp-1.0-SNAPSHOT.jar -m yarn-cluster -d
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/local/service/flink/lib/log4j-slf4j-impl-2.17.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/service/hadoop/share/hadoop/common/lib/log4j-slf4j-impl-2.20.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
2024-11-29 16:51:22,270 INFO  org.apache.hadoop.security.UserGroupInformation              [] - Hadoop UGI authentication : KERBEROS
2024-11-29 16:51:22,447 INFO  org.apache.flink.yarn.cli.FlinkYarnSessionCli                [] - Found Yarn properties file under /tmp/.yarn-properties-root.
2024-11-29 16:51:22,447 INFO  org.apache.flink.yarn.cli.FlinkYarnSessionCli                [] - Found Yarn properties file under /tmp/.yarn-properties-root.
2024-11-29 16:51:22,661 INFO  org.apache.hadoop.security.UserGroupInformation              [] - Hadoop UGI authentication : KERBEROS
2024-11-29 16:51:22,676 INFO  org.apache.hadoop.security.UserGroupInformation              [] - Login successful for user hadoop/tbds-10-4-4-19@TBDS-J88D3Q8J using keytab file /var/krb5kdc/emr.keytab
2024-11-29 16:51:23,023 WARN  org.apache.flink.yarn.configuration.YarnLogConfigUtil        [] - The configuration directory ('/usr/local/service/flink/conf') already contains a LOG4J config file.If you want to use
logback, then please delete or rename the log configuration file.
2024-11-29 16:51:23,624 INFO  org.apache.hadoop.yarn.client.AHSProxy                       [] - Connecting to Application History server at tbds-10-4-4-37/10.4.4.37:10200
2024-11-29 16:51:23,633 INFO  org.apache.flink.yarn.YarnClusterDescriptor                  [] - No path for the flink jar passed. Using the location of class org.apache.flink.yarn.YarnClusterDescriptor to locate th
e jar
2024-11-29 16:51:23,636 WARN  org.apache.flink.yarn.YarnClusterDescriptor                  [] - Neither the HADOOP_CONF_DIR nor the YARN_CONF_DIR environment variable is set.The Flink YARN Client needs one of these
 to be set to properly load the Hadoop configuration for accessing YARN.
2024-11-29 16:51:23,783 INFO  org.apache.flink.yarn.YarnClusterDescriptor                  [] - Found Web Interface tbds-10-4-4-47:46661 of application 'application_1731568354545_0024'.
Job has been submitted with JobID 94b23e2a39f8e0b553de1b87b7e3ac3d
```




Apache Flink Dashboard
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
Version: 1.16.1-TBDS-5.3.1_2024p3-SNAPSHOT Commit: 4baef97 @ 2024-09-26T08:24:09+02:00 Message:
Available Task Slots
3
Total Task Slots 4 Task Managers 1
Running Jobs
1
Finished 0 Canceled 0 Failed 0
Running Job List
Job Name Start Time Duration End Time Tasks Status
CountWindowApp 2024-11-29 16:51:28 1h 5m 28s – 2 2 RUNNING
Completed Job List
Job Name Start Time Duration End Time Tasks Status
No Data

api接口

TBDS 大数据平台上的 Flink 访问接口与开源兼容，可参考：



- Table API & SQL ：Table API 一般与 DataSet 或者 DataStream 紧密关联，可以通过一个 DataSet 或者 DataStream 创建出一个 Table，然后再使用类似 filter、sum、join、select 等这种操作。最近还可以将一个 Table 对象转换成 DataSet 或者 DataStream。SQL API 的底层是基于 Apache Calcite，Apache Calcite 实现了标准 SQL，使用起来比其它 API 更加灵活，因为可以直接使用 SQL 语句。Table API 和 SQL API 可以很容易地结合在一块使用。因为它们都返回 Table 对象。
- DataStream API & DataSet API ：主要提供针对流数据和批数据的处理，是对低级 API 进行了一些封装，提供了 filter、sum、max、min等高阶函数，简单易用，所以这些 API 在实际生产中应用还是比较广泛的。
- Stateful Stream Processing ：提供了对时间和状态的细粒度控制，简洁性和易用性较差，主要应用在一些复杂事件处理逻辑上。

常用操作

内置JAR作业

- Kerberos认证

#当前 flink 默认配置 hadoop 用户 principal 信息,不需要再做认证操作
#如果需要使用其他用户认证,修改 flink-conf.yaml 中以下信息为期望认证信息
security.kerberos.login.conf: /etc/krb5.conf
security.kerberos.login.keytab: /var/krb5kdc/emr.keytab
security.kerberos.login.principal: hadoop/172.16.0.32@TBDS-6QAEUIB9

- 提交流任务到 yarn

/usr/local/service/flink/bin/flink run -m yarn-cluster ./examples/streaming/TopSpeedWindowing.jar

- 其他操作命令

使用命令确认任务状态
/usr/local/service/flink/bin/flink list -m yarn-cluster -yid $YarnAppID

触发作业 savepoint，完成后会输出 savepoint 的路径. Savepoint completed. Path: hdfs:///flink/savepoints/$SavePath
/usr/local/service/flink/bin/flink savepoint $JobID hdfs:///flink/savepoints -yid $YarnAppID

停止作业
/usr/local/service/flink/bin/flink cancel $JobID -yid $YarnAppID

从 savepoint 恢复
/usr/local/service/flink/bin/flink run -m yarn-cluster -s hdfs:///flink/savepoints/$SavePath ./examples/streaming/TopSpeedWindowing.jar

内置SQL作业

运行 SQL 作业前，需要先启动本地集群 / yarn-session，下文以本地集群为例。

- Kerberos认证

#当前 flink 默认配置 hadoop 用户 principal 信息,不需要再做认证操作
#如果需要使用其他用户认证,修改 flink-conf.yaml 中以下信息
security.kerberos.login.conf: /etc/krb5.conf
security.kerberos.login.keytab: /var/krb5kdc/emr.keytab
security.kerberos.login.principal: hadoop/172.16.0.32@TBDS-6QAEUIB9

- 修改 conf/flink-conf.yaml 配置，增加 tm 的内存和 slot 数量。

taskmanager.heap.size: 1024m -> 3072m
taskmanager.numberOfTaskSlots: 1 -> 10

Hive表

- 配置hadoop相关配置，启动 flink 本地集群。

#将 planner & hive connector 移入 lib 目录
mv ./opt/connector/flink-sql-connector-hive-3.1.2_2.12-1.16.1-TBDS-5.3.1.2.jar ./lib
mv ./opt/flink-table-planner_2.12-1.16.1-TBDS-5.3.1.2.jar ./lib/
mv ./lib/flink-table-planner-loader-1.16.1-TBDS-5.3.1.2.jar ./opt/

#清理 yarn-session 模式残留 （如果存在残留文件会自动往 yarn-session 提交）
rm -rf /tmp/.yarn-properties-root

#启动本地集群
bin/start-cluster.sh

- 启动 flink sql-client

/usr/local/service/flink/bin/sql-client.sh embedded

- sql 操作

创建 hive catalog
CREATE CATALOG hive_catalog WITH ('type'='hive','default-database'='default','hive-conf-dir'='/usr/local/service/hive/conf/');

#设置 checkpoint
set 'execution.runtime-mode' = 'streaming';
set 'execution.checkpointing.interval'='10000';

```
#设置 hive 方言
SET table.sql-dialect = hive;

# 使用 hive catalog
USE CATALOG hive_catalog;
CREATE DATABASE hive_catalog.hive_db;
CREATE TABLE hive_catalog.hive_db.t1 (id BIGINT COMMENT 'unique id',data STRING);
INSERT INTO hive_catalog.hive_db.t1 values(1, 'tom');
SELECT * from hive_catalog.hive_db.t1;
```

- 恢复集群

```
#退出sql client 后关闭集群
./bin/stop-cluster.sh

#恢复依赖
mv ./lib/flink-sql-connector-hive-3.1.2_2.12-1.16.1-TBDS-5.3.1.2.jar ./opt/connector/
mv ./lib/flink-table-planner_2.12-1.16.1-TBDS-5.3.1.2.jar ./opt/
mv ./opt/flink-table-planner-loader-1.16.1-TBDS-5.3.1.2.jar ./lib/
```

Iceberg表

- 配置hadoop相关配置，启动 flink 本地集群。

```
#将 iceberg & hive connector 移入 lib 目录
mv ./opt/connector/iceberg-flink-runtime-1.16-1.4.2.jar ./lib
mv ./opt/connector/flink-sql-connector-hive-3.1.2_2.12-1.16.1-TBDS-5.3.1.2.jar ./lib

#清理 yarn-session 模式残留 （如果存在残留文件会自动往 yarn-session 提交）
rm -rf /tmp/.yarn-properties-root

#启动本地集群
./bin/start-cluster.sh
```

- 启动 flink sql-client

```
/usr/local/service/flink/bin/sql-client.sh embedded
```

- sql 操作

```
#创建 catalog
CREATE CATALOG iceberg_catalog WITH ('type'='iceberg','catalog-type'='hive','uri'='thrift://172.16.0.8
6:7004,thrift://172.16.0.132:7004','clients'='5','property-version'='1','warehouse'='hdfs:///usr/hive/ware
house/');
```

```
#设置 checkpoint
set 'execution.runtime-mode' = 'streaming';
set 'execution.checkpointing.interval'='10000';

CREATE DATABASE iceberg_catalog.iceberg_db;
CREATE TABLE iceberg_catalog.iceberg_db.t1 (id BIGINT COMMENT 'unique id',data STRING);
INSERT INTO iceberg_catalog.iceberg_db.t1 values(1, 'tom');
SELECT count(*) from iceberg_catalog.iceberg_db.t1;
```

- 恢复集群

```
#退出sql client 后关闭集群
./bin/stop-cluster.sh

#恢复依赖
mv ./lib/iceberg-flink-runtime-1.16-1.4.3.jar ./opt/connector
mv ./lib/flink-sql-connector-hive-3.1.2_2.12-1.16.1-TBDS-5.3.1.2.jar ./opt/connector
```

常用命令

```
#cd /usr/local/service/flink
# 创建standalone集群
./bin/start-cluster.sh

# 提交作业到standalone集群
./bin/flink run ./examples/streaming/TopSpeedWindowing.jar

# 任务状态查看
./bin/flink list

# 任务终止（不会保存savepoint）
./bin/flink cancel $JobID

# 任务终止（默认会保存savepoint）
./bin/flink stop $JobID

# 创建作业保存点savepoint
./bin/flink savepoint $JOB_ID /tmp/flink-savepoints

# 将任务提交到yarn集群上
# 提交到yarn上的作业无法使用上述的3条命令
```

```
./bin/flink run -m yarn-cluster -d examples/streaming/TopSpeedWindowing.jar
```

开发规范
术语和数据类型

术语表

流计算常用术语如下：

术语	详细说明
流计算	流计算是面向流式数据的计算，它从一个或多个流式数据源读取持续不断产生的数据，经过引擎中多个算子的组合进行高效计算，再根据实际需要，将结果输出至下游的多种数据目的，例如消息队列、数据库、数据仓库、存储服务等。
数据源（Source）	为流计算系统持续提供输入数据，例如CKafka 等。
数据目的（Sink）	流计算系统处理结果输出的地方，例如 CKafka、云数据库 MySQL、PostgreSQL 等。
Schema	表示一个表的结构信息，例如各个列名、列类型等。对于 PostgreSQL 而言，Schema 是介于 Database 和 Table 之间的一个层级，可以理解成数据库内部的命名空间。
时间模式	指导系统处理数据时如何获取时间戳，目前支持 Event Time、Processing Time、Source Time 三种时间模式。
Event Time	Event Time 时间模式下，时间戳由输入数据的某个字段提供，可以用 WATERMARK FOR 语句指定该字段并启用 Event Time 时间模式，适用于数据源包含精确时间戳的场合。
Watermark	表示一个特定的时间点，在该时间点之前的所有数据已经得到妥善处理。 Watermark 由系统自动生成，用户可通过 WATERMARK FOR BOUNDED 语句指定时间戳的最大容差。
Processing Time	Processing Time 时间模式下，时间戳由系统自动生成并添加到数据源中（以PROCTIME命名，SELECT *时不可见，使用时必须显式指定）。它以每条数据被系统处理的时间作为时间戳，因而有一定的不可控性，适用于对时间精度要求不是很高的场合。
Source Time	在 Source Time 时间模式下，可使用 Kafka 每条记录所含元数据的时间戳作为流计算处理所使用的时间戳（以SOURCETIME 命名，SELECT * 时不可见，使用时必须显式指定），避免了输入数据没有时间戳字段时，使用 Processing Time 模式带来的不可控性。
时间窗口	定义了多个时间段以及各个时间段之间的关系（例如是否可重叠、是否固定大小）。目前系统支持 TUMBLE、HOP、SESSION 三种时间窗口。具体见 时间窗口函数。

术语	详细说明
CKafka	CKafka 是云服务商提供的一个分布式的、高吞吐量、高可扩展性的消息系统，完全兼容0.10版本的 Kafka API。流计算目前支持 CSV 和 JSON 两种输入输出格式。
Tuple 与 Append 流	Tuple（又称为 Append）为数据流类型的一种，可以存放不含主键的流数据。用户可以不断追加新数据到这种数据流中，它不涉及到对之前已发出数据的更新操作。目前各种数据源和数据目的均支持 Append 流的输入输出。
Upsert 流	Upsert 是 Update OR Insert 的简写，由 DISTINCT、不含时间窗口的 GROUP BY 语句、不含时间范围的 JOIN 语句等查询产生，它具有主键定义，如果后续发出的数据与之前的某条数据具有相同主键，则更新该条记录为新值；反之则新增一行数据。它可以确保之前发出的数据被更新以反映最新的值。
DDL 语句	DDL 即数据定义语言（Data Definition Language），为 SQL 语言的一个子集，由 CREATE 语句组成。它可以用来定义表、视图、以及用户自定义函数（UDF）等。
DML 语句	DML 即数据操作语言（Data Manipulation Language），为 SQL 语言的一个子集，包括 INSERT 和 SELECT 语句，可以用来对数据表、视图等进行选择、变换、筛选、插入等操作。

数据类型

Flink 采用符合 ANSI SQL 规范的定义，支持丰富的数据类型。用户在使用 CREATE TABLE 语句定义一个数据表时，可以用这些数据类型来定义每个字段的类型。

支持的类型列表

类型名称	使用说明
CHAR CHAR(n)	定长字符串。n 表示容纳的字符数，默认为1，即 CHAR 等价于 CHAR(1)。
VARCHAR VARCHAR(n) STRING	可变长度字符串。n 表示最多容纳的字符数，默认为1，即 VARCHAR 等价于 VARCHAR(1)。STRING 等价于 VARCHAR(2147483647)。
BINARY BINARY(n)	固定长度的二进制字符串。n 表示容纳的字节数量，默认为1，即 BINARY 等价于 BINARY(1)。
VARBINARY VARBINARY(n) BYTES	可变长度的二进制字符串。n 表示容纳的字节数量，默认为1，即 VARBINARY 等价于 VARBINARY(1)。BYTES 等价于 VARBINARY(2147483647)。
DECIMAL DECIMAL(p) DECIMAL(p, s)	固定精度的实数（定点数）。 p 表示数字的总位数（精度），取值区间为[1，38]，默认值是10。 s 表示小数点右边的位数（尾数），取值区间为[0，p]，默认值是0。DEC 和

类型名称	使用说明
DEC DEC(p) DEC(p, s) NUMERIC NUMERIC(p) NUMERIC(p, s)	NUMERIC 是 DECIMAL 的别名，可以任意互换使用，即 DECIMAL(p, s) 等价于 DEC(p, s) 也等价于 NUMERIC(p, s)。
TINYINT	1个字节的整数。等价于 Java 的 Byte 类型，取值范围是[-128，127]。
SMALLINT	2个字节的整数。等价于 Java 的 Short 类型，取值范围是[-32768，32767]。
INT	4个字节的整数。等价于 Java 的 Integer 类型，取值范围是[-2147483648，2147483647]。
BIGINT	8个字节的整数。等价于 Java 的 Long 类型，取值范围是[-9223372036854775808，9223372036854775807]。
FLOAT	4个字节的单精度浮点数。等价于 Java 的 Float 类型。
DOUBLE	8个字节的双精度浮点数。等价于 Java 的 Double 类型。
DATE	日期类型，包含年-月-日。取值范围是[0000-01-01，9999-12-31]。
TIME TIME(p)	不含时区信息的时间类型，包含时-分-秒及纳秒信息。 取值范围是[00:00:00.000000000，23:59:59.999999999]。 p 表示秒的小数位精度，取值范围是[0，9]。如果未指定，默认为0。 此类型不支持闰秒。类似于 Java 的 LocalTime 类型。
TIMESTAMP TIMESTAMP(p) TIMESTAMP WITHOUT TIME ZONE TIMESTAMP(p) WITHOUT TIME ZONE	不含时区信息的时间戳类型，精度可以达到纳秒级别。 取值范围是[0000-01-01 00:00:00.000000000，9999-12-31 23:59:59.999999999]。 p 表示秒的小数位精度，取值范围是[0，9]。如果未指定，默认为6。 此类型不支持与 BIGINT（Java 的 Long 类型）之间相互转换。TIMESTAMP WITHOUT TIME ZONE 类型等价于 TIMESTAMP 类型。 此类型不支持闰秒。类似于 Java 的 Timestamp 类型。
TIMESTAMP WITH TIME ZONE TIMESTAMP(p) WITH TIME ZONE	含时区信息的时间戳类型。 取值范围是[0000-01-01 00:00:00.000000000 +14:59，9999-12-31 23:59:59.999999999 -14:59]。 p 表示秒的小数位精度，取值范围是[0，9]。如果未指定，默认为6。 每条该类型的数据，都含有各自的时区信息。 此类型不支持闰秒。类似于 Java 的 OffsetDateTime 类型。
TIMESTAMP WITH LOCAL TIME ZONE TIMESTAMP(p) WITH LOCAL TIME ZONE	含 本地时区 信息的时间戳类型。 取值范围是[0000-01-01 00:00:00.000000000 +14:59，9999-12-31

类型名称	使用说明
	23:59:59.999999999 -14:59]。 p 表示秒的小数位精度，取值范围是[0，9]。如果未指定，默认为6。 时区数据不存储在每条数据中，而是遵循全局的时区设置。 此类型不支持闰秒。类似于 Java 的 OffsetDateTime 类型。
INTERVAL YEAR INTERVAL YEAR(p) INTERVAL YEAR(p) TO MONTH INTERVAL MONTH	表示以年和月表示的一段粗粒度的时间间隔，精度为月份。 语法为+年数-月数，例如+04-02。 取值范围是[-9999-11，+9999-11]。 p 表示年的精度位数，取值范围是[1，4]，默认为2。
INTERVAL DAY INTERVAL DAY(p1) INTERVAL DAY(p1) TO HOUR INTERVAL DAY(p1) TO MINUTE INTERVAL DAY(p1) TO SECOND(p2) INTERVAL HOUR INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND(p2) INTERVAL MINUTE INTERVAL MINUTE TO SECOND(p2) INTERVAL SECOND INTERVAL SECOND(p2)	表示以天、时、分、秒、纳秒表示的细粒度时间间隔，最高精度为纳秒。 取值范围是[-999999 23:59:59.999999999，+999999 23:59:59.999999999]。 p1 表示天数精度的位数，p1 的取值范围是[1，6]，默认为2。 p2 表示秒的小数位数，p2 的取值范围是[0，9]，默认为6。
ARRAY t ARRAY	数组类型，大小固定为2147483647。 t 表示数组中元素的类型。 ARRAY 等价于 t ARRAY，例如 ARRAY 与 INT ARRAY 含义一致。
MAP<kt, vt>	键值对映射类型，其中 kt 是键（key）的类型，vt 是值（value）的类型。
MULTISET t MULTISET	允许重复元素的集合类型，别称为 Bag。同样地，MULTISET 等价于 t MULTISET。
ROW<n0 t0, n1 t1, ...> ROW<n0 t0 'd0', n1 t1 'd1', ...> ROW(n0 t0, n1 t1, ...) ROW(n0 t0 'd0', n1 t1 'd1', ...)	允许包含多个字段的复合类型，每个字段有自己的类型，类似于其他语言的 Struct 及 Tuple 类型。 n 表示字段名。 t 是字段的逻辑类型。 d 是字段的描述。 尖括号和圆括号的两种写法是等价的，例如 ROW(field1 INT, field2 BOOLEAN) 等

类型名称	使用说明
	同于 ROW<field1 INT, field2 BOOLEAN>。
BOOLEAN	三值布尔型，可选值为 TRUE、FALSE 和 UNKNOWN。如果不允许出现 UNKNOWN，可以定义为 BOOLEAN NOT NULL 类型。
RAW('class', 'snapshot')	可表示任意类型，例如 Flink 无法识别或者不需要识别的类型。 class 表示原始类型。 snapshot 表示 Base64 编码的序列化后的 TypeSerializerSnapshot 定义。
NULL	空值，类似 Java 等语言中的 null 值。

DDL数据定义语句

CREATE TABLE

CREATE TABLE 语句用来描述数据源（Source）或者数据目的（Sink）表，并将其定义为一张表，以供后续语句引用。

表定义语法

语法结构

```
CREATE TABLE 表名
  (
    { <列定义> | <计算列定义> }[ , ...n]
    [ <Watermark 定义> ]
    [ <表约束定义, 例如 Primary Key 等> ][ , ...n]
  )
  [COMMENT 表的注释]
  [PARTITIONED BY (分区列名1, 分区列名2, ...)]
  WITH (键1=值1, 键2=值2, ...)
  [ LIKE 其他的某个表 [( <LIKE 子句选项> )] ]
```

符号含义

CREATE TABLE 语句创建的表，既可以作为数据源表，也可以作为数据目的表。但是如果没有对应的 Connector，则会在运行时报错。

```
<列定义>:
  列名 列类型 [ <列的约束定义> ] [COMMENT 列的注释]

<列的约束定义>:
  [CONSTRAINT 约束名] PRIMARY KEY NOT ENFORCED

<表的约束定义>:
  [CONSTRAINT 约束名] PRIMARY KEY (列名1, 列名2, ...) NOT ENFORCED

<计算列定义>:
  列名 AS 计算列表达式 [COMMENT 列的注释]

<Watermark 定义>:
  WATERMARK FOR 某个Rowtime类型的列名 AS 某个Watermark策略表达式

<LIKE 子句选项>:
{
    { INCLUDING | EXCLUDING } { ALL | CONSTRAINTS | PARTITIONS }
```

```
  | { INCLUDING | EXCLUDING | OVERWRITING } { GENERATED | OPTIONS | WATERMARKS }
}[, ...]
```

子句功能说明

计算列

计算列是一种虚拟列，它是逻辑上的定义而非数据源中实际存在的列，通常由同一个表的其他列、常量、变量、函数等计算而来。例如，如果数据源中定义了 price（商品单价）和 quantity（采购量），那么就可以新定义一个 cost（总成本）字段，即 cost AS price * quantity，即可在后续查询中直接使用 cost 字段。

更常见的用法是使用计算列来实现非标准时间戳的标准化。例如在一个数据源中，时间戳字段 mytime 为 Unix 格式（例如以毫秒为单位的1599469771494），则可以通过计算列的方式ts AS TO_TIMESTAMP(FROM_UNIXTIME(mytime / 1000, 'yyyy-MM-dd HH:mm:ss'))，将时间戳处理为 Flink 可识别的 Timestamp(3) 类型。另外，如果数据源时间戳字段虽然是 Timestamp(3) 格式，但是嵌套在 JSON 的其他字段中，也可以用计算列的方式将其解析出来。

> 注意：
>
> 计算列只允许在 SELECT 语句中使用。
>
> 对于 INSERT 语句，目的表中的计算列会被自动忽略。

WATERMARK

Watermark 定义

Watermark 决定着 Flink 作业的时间模式（详见下文的 Event Time/Processing Time 介绍小节），定义方式：

```
WATERMARK FOR 某个Rowtime类型的列名 AS 某个Watermark策略表达式
```

例如 WATERMARK FOR my_time_field AS my_time_field - INTERVAL '3' SECOND 表示定义一个容差为 3 秒的 Watermark 策略。

某个 Rowtime 类型的列名：必须是 Flink 可识别的 Timestamp(3) 类型，且不是嵌套列。如果类型不对或者属于嵌套字段，则需要使用上文提到的"计算列"功能，创建一个转换后的虚拟列，作为 Rowtime 类型的列。

某个 Watermark 策略表达式：用于定义 Watermark 的生成策略，可以用各种表达式来描述一个 Timestamp(3) 类型的值，以作为每次生成 Watermark 时的依据。

定义示例如下：

```
CREATE TABLE StudentRecord (
  Id BIGINT,
  StudentName STRING,
  RegistrationTime TIMESTAMP(3),
  WATERMARK FOR RegistrationTime AS RegistrationTime - INTERVAL '3' MINUTE
) WITH ( ... ... );
```

Watermark 生成策略

单调递增时间戳的 Watermark 策略
如果时间戳可以确保是单调递增的，不存在乱序的情况，则可以用如下语法，以期得到最低的数据处理延迟。

```
WATERMARK FOR 某个Rowtime类型的列名 AS 某个Rowtime类型的列名
```

下面语句的含义是将每个输入数据中最大时间戳作为 Watermark 的取值。因此如果存在乱序，就会造成晚到的数据未达到 Watermark 的界限而被丢弃。例如：

```
WATERMARK FOR my_time AS my_time
```

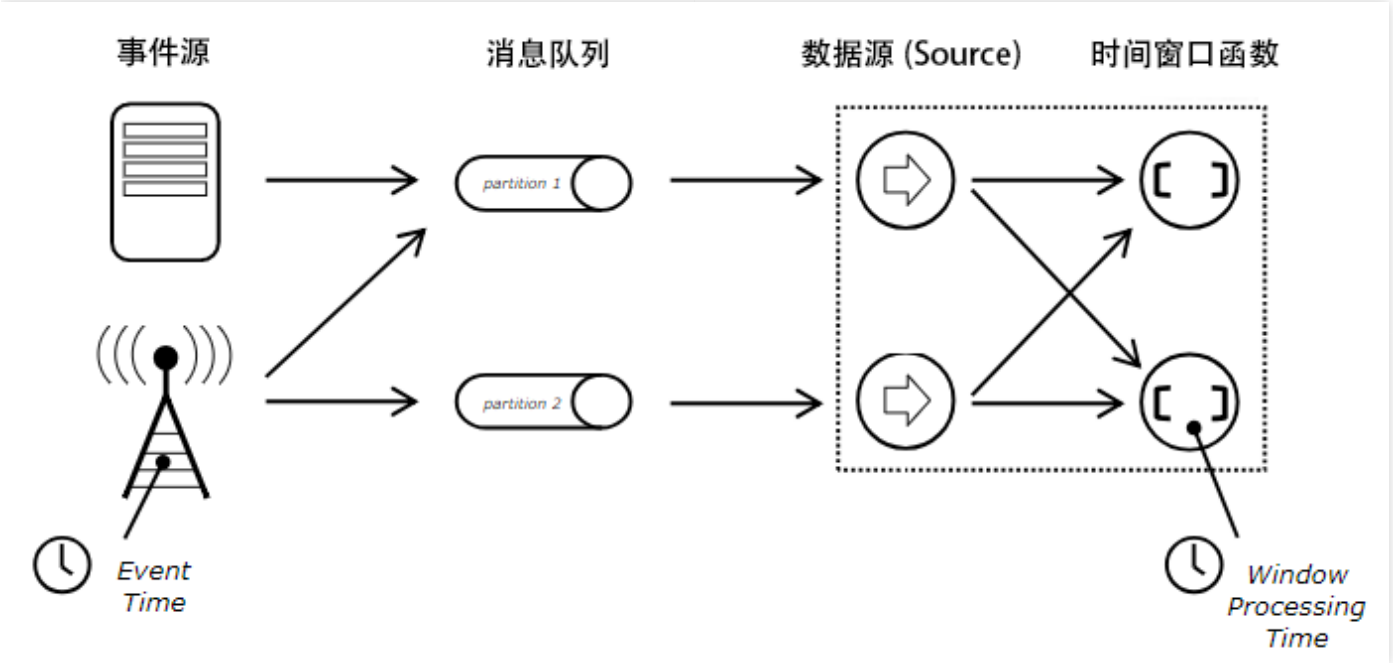
有限容忍乱序的 Watermark 策略
与上述的策略不同，本策略允许数据中存在一定范围的乱序。这个乱序范围由用户自行控制，如果设置的较大，则会带来较长的延迟（数据积压、等待）；如果设置的较小，则超过阈值的数据则可能被丢弃（造成结果不准确）。

```
WATERMARK FOR 某个Rowtime类型的列名 AS 某个Rowtime类型的列名 - INTERVAL '时间长度' 时间单位
```

下面语句的含义是每个输入数据中最大的时间戳减去3秒的容差作为 Watermark 的取值。因此如果存在乱序，但是后来的数据比之前最大值相差不到3秒，也会被允许加入计算。

```
WATERMARK FOR my_time AS my_time - INTERVAL '3' SECOND
```

Event Time/Processing Time 介绍
对于基于窗口的操作（例如 GROUP BY、OVER、JOIN 条件中时间段的指定），流计算 Flink 支持 Event Time 和 Processing Time 两种时间处理模式。



Event Time 模式使用输入数据自带的时间戳，容忍一定程度的乱序数据输入（例如，更早的数据由于各节点处理能

力、网络波动等不可预知的原因，来的却更晚），这个参数可以通过 BOUNDED 的第二个参数指定，单位是毫秒。该处理模式最精确，但要求输入数据自带时间戳。目前只支持数据源中以 timestamp 类型定义的字段，未来将会支持虚拟列，可将其他类型的列应用处理函数转换为系统接受的时间戳。
Processing Time 处理模式不要求输入数据有时间戳，而是将该条数据被处理的时间戳自动加入数据，并以 PROCTIME（必须全为大写）字段命名。该列是隐藏的，SELECT *时不会出现，只有用户手动使用时才会被读取。

> 注意：
>
> 对于同一个任务的所有数据源，只允许采用一种时间模式。若某个使用 Event Time 模式，则必须要求所有定义的 Table Source 都定义时间戳并声明 WATERMARK 时间戳字段。

主键 PRIMARY KEY
定义表或视图时，可以声明某些字段为主键（PRIMARY KEY），表示这些字段的值不会重复且不会为 NULL（即 SQL 的 NOT NULL + UNIQUE）。
主键的定义可以在列上，也可以单独使用 CONSTRAINT 语句定义。不能对同一个表多次定义不同的主键。Flink 因为无法保证数据源的每条数据主键不重复，目前只支持 PRIMARY KEY NOT ENFORCED 语法，即提醒用户需自行保证主键语义。
分区 PARTITIONED BY
如果在某个列上定义了 PARTITIONED BY 子句，则表明允许 Flink 对该列进行分区。主要影响 FileSystem Sink，它会根据分区的不同，为每个数据分区创建一个单独的目录。
流计算 Flink 不建议用户使用 FileSystem Sink，因为所有 TaskManager 运行结束后，文件系统的数据会被自动清理。
WITH 参数
WITH 参数通常用于指定数据源和数据目的的 Connector 所需参数，语法为'key1'='value1', 'key2' = 'value2'的键值对。
例如要写入 Kafka（ CKafka 或自建 Kafka）时，需要指定服务器地址、消费的 Topic、消费的起始时间点等信息。

LIKE 子句
LIKE 子句允许用于创建表（下文称为 B 表）时，引用其他表（下文称为 A 表）的结构，这样可以大幅节省 CREATE TABLE 语句的代码量，做到代码复用。例如，把同样的数据一份写入 Kafka Sink，另一份写入 Elasticsearch Sink，还有一部分写入 MySQL，那么就可以通过 LIKE 语句来实现定义三张表，同时复用列定义的效果。
定义一张 A 表：

```
CREATE TABLE A (
  Id BIGINT,
  StudentName STRING,
  RegistrationTime TIMESTAMP(3)
) WITH ( ... 某些参数 ... );
```

再定义一个含 Watermark 的 B 表，则可以直接基于上面的表，创建一个新的表：

```
CREATE TABLE B (
```

```
    WATERMARK FOR RegistrationTime AS RegistrationTime - INTERVAL '3' MINUTE
) WITH ( ... 另一些参数 ... ) LIKE `A`;
```

默认情况下 LIKE 语句与 WITH 参数无关，所以两个表允许使用完全不同的 WITH 参数集。如果希望继承原表的 WITH 参数信息，则需要通过 LIKE 子句选项来实现。
LIKE 子句选项
目前 LIKE 子句提供了如下的选项，可以控制引用（继承）某个 A 表的内容：
CONSTRAINTS：主键（PRIMARY KEY）等约束
GENERATED：计算列
OPTIONS：WITH 参数
PARTITIONS：PARTITIONED BY 定义
WATERMARK：WATERMARK FOR 定义
ALL：以上所有
同时，Flink 提供了三种不同的合并策略：
INCLUDING：继承 A 表的所有指定属性，但如果 A 表和 B 表的某些定义有冲突（例如含有相同字段定义）则报错。
EXCLUDING：B 表中不会包含任何 A 表中已有的指定属性。
OVERWRITING：继承 A 表的所有指定属性，如果 A 表和 B 表定义有冲突，则 B 表的定义会覆盖 A 表的定义。
如果未提供 LIKE 子句选项，默认行为是INCLUDING ALL OVERWRITING OPTIONS，即 B 表会继承 A 表的所有定义和设置，但是会覆盖掉 WITH 参数。

Create View

用户可以使用 CREATE VIEW 语句创建视图。视图是一个虚拟表，基于某条 SELECT 语句。视图可以用在定义新的虚拟数据源（类型转换、列变换和虚拟列等），拆分过长代码等场景。
语法

```
CREATE VIEW 视图名 AS
SELECT 子句
```

示例一
创建一个名为 MyView 的视图：

```
CREATE VIEW MyView AS
SELECT s1.time_, s1.client_ip, s1.uri, s1.protocol_version, s2.status_code, s2.date_
FROM KafkaSource1 AS s1, KafkaSource2 AS s2
WHERE s1.time_ = s2.time_ AND s1.client_ip = s2.client_ip;
```

示例二
在计算中由于数据量较大、函数方法类型匹配要求等原因，必须使用 TINYINT、SMALLINT 等类型。当 Kafka 等输入类型不符合需求时，可通过 CREATE VIEW 语句配合 CAST() 类型转换函数，实现定义虚拟视图作为新的数据源。

通过定义一个名为 KafkaSource2 的视图，实现将 KafkaSource1 数据源中的 BIGINT 类型的 status_code 列转为 VARCHAR 类型的列，命令如下：

```
CREATE VIEW KafkaSource2 AS
SELECT
  `time_`,
  `client_ip`,
  `method`,
  CAST(`status_code` AS VARCHAR) AS status_code,
FROM KafkaSource1;
```

> 注意：
>
> 不当的数据转换 CAST() 可能会导致精度损失，例如由 BIGINT 转为 INTEGER 或 TINYINT 等，请谨慎使用。
>
> 如果需要进行字符串（VARCHAR）和时间戳（TIMESTAMP）之间的类型转换，可参考时间函数中 TO_TIMESTAMP、DATE_FORMAT 等函数。

CREATE FUNCTION

对于 SQL 作业，用户可以上传自定义程序包，然后在作业分析开发页的参数设置中，引用该程序包。目前支持从本地上传，也可以引用账户下现有 COS 存储中的资源（仅限相同地域）。
这里的程序包既可以用来扩展 Connector 的功能，也可以创建自定义函数（UDF）。
语法
目前流计算 Flink 支持 Java 和 Scala 两种语言编写的程序包。当用户上传了自定义程序包后，在界面上关联后即可用下面的 CREATE FUNCTION 语句来声明：

```
CREATE TEMPORARY SYSTEM FUNCTION 函数名
  AS '函数类全名' [LANGUAGE JAVA|SCALA]
```

其中的函数名可以自行定义，但不要与现有的冲突。函数类全名为 Java 或 Scala 类的类全名（例如 'com.example.flink.MyCustomFunction'）。
命名覆盖
如果存在系统内置的同名函数时，用户使用上述语法创建的 UDF 会覆盖系统内置的函数。因此除非有意改变系统函数的功能，请不要创建与系统内置函数同名的自定义函数。
函数类型
目前 Flink 支持下面多种函数定义。
标量函数（Scalar Function）
标量函数简称 UDF，作用是将一个值转换为另一个值（一对一），例如系统内置的 SUBSTRING、REPLACE 等字符串操作函数，都属于标量函数。

表函数（Table Function）

表函数简称 UDTF，作用是将一个值转为表中的一行数据（一变多），这样可以在后续 JOIN 操作中作为右表。

聚合函数（Aggregate Function）

聚合函数简称 UDAF，作用是将多行数据的一组值，聚合为一个最终值（多变一），例如系统内置的 MAX、MIN、AVG 等都属于聚合函数。

表聚合函数（Table Aggregate Function）

表聚合函数的作用是将多行数据的一组值，聚合为新的多行数据（多对多）。

异步表函数（Async Table Function）

异步表函数可作为一种特殊的数据源，例如可以通过它来对接外部的数据库、数据存储。

UDF 开发指南

由于 Flink 不同版本间 API 和文档迭代频繁，可参考 Flink 官方文档中的 UDF 开发指南。

DML数据操作语句

查询语句

SELECT FROM

SELECT 不能单独使用，必须配合CREATE VIEW ... AS或 INSERT INTO 使用，否则系统会提示没有合适的 Operator。

语法

```
SELECT 以逗号分隔的需要选中的字段
FROM 数据源或视图
WHERE 过滤条件
其他子查询
```

示例

```
SELECT s1.time_, s1.client_ip, s1.uri, s1.protocol_version, s2.status_code, s2.date_
FROM KafkaSource1 AS s1, KafkaSource2 AS s2
WHERE s1.time_ = s2.time_ AND s1.client_ip = s2.client_ip;
```

WHERE

WHERE 用来过滤查询条件（谓词），多个并列的条件可以用 AND、OR 来连接。在与外部数据库 TencentDB 的表 JOIN 时，条件的连接只支持 AND。如需使用 OR 的功能，请参见 UNION ALL。

HAVING

HAVING 用于过滤 GROUP BY 之后的结果。WHERE 在 GROUP BY 之前过滤，而 HAVING 在 GROUP BY 分组之后过滤。

```
SELECT SUM(amount)
FROM Orders
WHERE price > 10
GROUP BY users
HAVING SUM(amount) > 50
```

GROUP BY

在流计算 Flink 中，GROUP BY 用于对结果进行分组聚合。目前有含时间窗口（Window）类型的 GROUP BY 和不含窗口的 GROUP BY（即持续查询）。

含时间窗口（Window）类型的 GROUP BY 不会更新之前的结果，因而会产生 Append（Tuple）类型的数据流，只允许写入不带主键的 MySQL、PostgreSQL、Kafka、Elasticsearch 等数据目的。

不含窗口的 GROUP BY 会更新之前发出的记录，因而会产生 Upsert 类型的数据流，只允许写入含主键的云数据库 MySQL、PostgreSQL 数据目的表（Sink）、Elasticsearch 等，且主键必须与 GROUP BY 语句里 Upsert 字段一

致。
含时间窗口的 GROUP BY
本示例定义了一个包含时间窗口的 GROUP BY 查询语句。关于时间窗口函数的使用方法，请参见 时间窗口函数章节。

```
SELECT user, SUM(amount)
FROM Orders
GROUP BY TUMBLE(rowtime, INTERVAL '1' DAY), user
```

在 Event Time 时间模式下，使用 WATERMARK FOR 定义时间戳字段，那么 TUMBLE 窗口函数的第一个参数必须为该字段。HOP 和 SESSION 窗口同理。
在 Processing Time 时间模式下，TUMBLE 窗口函数的第一个参数必须为表proctime()声明的字段。HOP 和 SESSION 窗口同理。
不含时间窗口的 GROUP BY（持续查询）
本示例定义了一个不包含时间窗口的 GROUP BY 查询语句，这种查询叫做持续查询，因为它会根据每条新到的数据来计算并决定是否更新之前发出的结果，因而会产生一个 Upsert 流。

```
SELECT a, SUM(b) as d
FROM Orders
GROUP BY a
```

> 注意：
>
> 这种方式可能会因为 key 的数量过大或数据过多而发生内存溢出。因而请谨慎设置对象超时时间，不要过长。

JOIN
目前流计算 Flink 系统只支持等值连接 Equi-JOIN，即 JOIN 条件内包含至少一条令左右表某字段相等的过滤条件。
流和流的 Inner Equi-JOIN
目前流和流的连接也分为两种：含时间范围的和不含时间范围的。前者会生成 Append（Tuple）类型的流，而后者会生成 Upsert 类型的流。
含时间范围的 Inner JOIN
含时间范围的 JOIN 也称为 Interval Join，它的 WHERE 条件中需要至少一个等值连接的 JOIN 条件和一个指定的时间范围。这个时间范围可以用 <、<=、>=、> 或 BETWEEN ... AND 等来表示。

```
ltime = rtime
ltime >= rtime AND ltime < rtime + INTERVAL '10' MINUTE
ltime BETWEEN rtime - INTERVAL '10' SECOND AND rtime + INTERVAL '5' SECOND
```

示例：

```
SELECT *
```

```
FROM Orders o, Shipments s
WHERE o.id = s.orderId AND
o.ordertime BETWEEN s.shiptime - INTERVAL '4' HOUR AND s.shiptime
```

不含时间范围的 Inner JOIN

不含时间范围的流-流 JOIN 的特点是只要求有至少一个等值连接，而不要求指定时间范围。也就是说，它会将历史以来所有的活跃数据参与计算（可以通过指定超时时间来去除不活跃的元素）。

注意

可能会导致非常大的内存占用，需要谨慎使用。通常需要设置合适的对象超时时间，以及时清除失活的对象。

这种查询会产生一个 Upsert 流，只能使用含主键的 MySQL、PostgreSQL、Elasticsearch 等数据目的（Sink）来接收数据。

示例：

```
SELECT *
FROM Orders INNER JOIN Product ON Orders.productId = Product.id
```

Outer Equi-JOIN

Outer Equi-JOIN 会产生 Upsert 流，因此需要注意数据目的（Sink）必须可以接受 Upsert 数据流，例如 MySQL、PostgreSQL、Elasticsearch 等。

> 注意：
>
> 由于不对 JOIN 的顺序做优化，JOIN 操作会依次按照 FROM 语句后定义的各表进行，因此可能会产生非常大的状态压力，甚至可能执行失败。

```
SELECT *
FROM Orders LEFT JOIN Product ON Orders.productId = Product.id

SELECT *
FROM Orders RIGHT JOIN Product ON Orders.productId = Product.id

SELECT *
FROM Orders FULL OUTER JOIN Product ON Orders.productId = Product.id
```

维表 JOIN

流计算 Flink 也支持流与 MySQL、PostgreSQL 数据库中维表（Temporal table，即随时间不断变化的表）的 JOIN，语法同上面介绍的完全一致，只是要求维表必须放在 JOIN 条件的右表。

```
SELECT
  o.amout, o.currency, r.rate, o.amount * r.rate
FROM
  Orders AS o
  JOIN LatestRates FOR SYSTEM_TIME AS OF o.proctime AS r
```

```
ON r.currency = o.currency
```

> 注意：
>
> 一定要加入 FOR SYSTEM_TIME AS OF 语句，否则虽然仍然可以执行 JOIN，但是只会全量读取一次数据库，结果可能不符合预期。

与 UDTF 的 JOIN
如果用户自定义了表函数（UDTF，即 User-Defined Table Function），则可以将表函数作为 JOIN 的右表，语法与普通 JOIN 类似，只是需要加上 LATERAL TABLE( ) 关键字，将 UDTF 包围起来。
Inner UDTF JOIN

```
SELECT users, tag
FROM Orders, LATERAL TABLE(unnest_udtf(tags)) t AS tag
```

Left Outer UDTF JOIN

```
SELECT users, tag
FROM Orders LEFT JOIN LATERAL TABLE(unnest_udtf(tags)) t AS tag ON TRUE
```

> 注意：
>
> 目前 Left Outer UDTF JOIN 只支持ON TRUE语法，类似于 CROSS JOIN。

与数组进行 JOIN
流计算 Flink 系统支持和一个已定义的数组对象（可通过 值构造函数 构造数组对象 ARRAY）做 JOIN 操作。
示例：假设 tags 是一个已定义的数组。

```
SELECT users, tag
FROM Orders CROSS JOIN UNNEST(tags) AS t (tag)
UNION ALL
UNION ALL 用来合并两个查询的结果。

SELECT *
FROM (
    (SELECT user FROM Orders WHERE a % 2 = 0)
  UNION ALL
    (SELECT user FROM Orders WHERE b = 0)
)
```

目前流计算 Flink 只支持 UNION ALL 而暂不支持 UNION，即不会对相同的行进行去重操作。
如果需要实现去重以达到 UNION 的效果，请配合 DISTINCT 使用。DISTINCT 会让结果从 Append（Tuple）流变

为 Upsert 流，因而只能使用含主键的 MySQL、PostgreSQL、Elasticsearch 数据目的（Sink）来接收数据。

OVER Window 聚合

如果需要对数据流做基于滑动窗口的聚合（不使用 GROUP BY 的聚合），那么可以使用 OVER 来进行滑动窗口的聚合操作。在 OVER 中可以指定 PARTITION、ORDER、窗口范围等。

下面的示例定义了一个滑动窗口聚合查询，统计一个大小为3的滑动窗口的交易总额（amount）。其中对之前的行使用 PRECEDING，而目前还不支持 FOLLOWING。

另外 ORDER BY 后面只允许一个时间戳字段，本例使用数据源中声明的 proctime 字段。

```
SELECT SUM(amount) OVER (
  PARTITION BY user
  ORDER BY proctime
  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM Orders

SELECT COUNT(amount) OVER w, SUM(amount) OVER w
FROM Orders
WINDOW w AS (
  PARTITION BY user
  ORDER BY proctime
  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
```

ORDER BY

ORDER BY 用来对查询的结果做排序，默认是 ASC（升序排列），也可以显式指定 DESC（降序排列）。

> 注意：
>
> 要求第一个排序项必须是升序的时间列（Event Time 时间戳或 Processing Time 时间戳，即 PROCTIME），之后的排序项可以自由指定。

```
SELECT *
FROM Orders
ORDER BY `orderTime`, `username` DESC, `userId` ASC
```

DISTINCT

DISTINCT 用来对查询结果进行去重，它必须放在 SELECT 后面。

```
SELECT DISTINCT users FROM Orders
```

DISTINCT 会产生一个 Upsert 流，因而只有 Upsert 类型的数据目的（Sink）才可以接收其结果。而且长时间查询可能会导致内存占用过大，请谨慎使用。

通过设置合适的对象过期时间，可以及时清除失活对象来节省内存。

IN

可以使用 IN 关键字对判断指定集合（例如子查询）中是否存在某元素。

> 注意：
>
> 该操作的内存压力可能较大，请谨慎使用。

```
SELECT user, amount
FROM Orders
WHERE product IN (
    SELECT product FROM NewProducts
)
```

EXISTS

如果 EXISTS 后面子查询的结果大于或等于一行（存在数据），则返回true。

> 注意：
>
> 该操作的内存压力可能较大，请谨慎使用。

```
SELECT user, amount
FROM Orders
WHERE product EXISTS (
    SELECT product FROM NewProducts
)
```

ORDER BY

ORDER BY 可以对某个字段进行排序后输出。

注意

该操作的内存压力可能较大，请谨慎使用。

```
SELECT *
FROM Orders
ORDER BY orderTime
Grouping Sets、Rollup、Cube
```

对于 Grouping Sets、Rollup、Cube 操作，产生的是一个 Upsert 流，因此只有 Upsert 类型的数据目的才可以接收其结果。

```
SELECT SUM(amount)
FROM Orders
```

```
GROUP BY GROUPING SETS ((user), (product))
```

模式匹配

目前流计算 Flink 支持 MATCH_RECOGNIZE 语句对一条输入流进行模式匹配，用户可以用 SQL 语句来描述 CEP（复杂事件处理）的逻辑。

```
SELECT T.aid, T.bid, T.cid
FROM MyTable
MATCH_RECOGNIZE (
  PARTITION BY userid
  ORDER BY proctime
  MEASURES
    A.id AS aid,
    B.id AS bid,
    C.id AS cid
  PATTERN (A B C)
  DEFINE
    A AS name = 'a',
    B AS name = 'b',
    C AS name = 'c'
) AS T
```

上述例子定义了 A、B、C 三个事件，分别表示 name 字段等于 a、b、c 时的事件。PATTERN 表示事件触发的规则，即 A、B、C 三个事件连续出现时触发。MEASURES 指定了事件触发后的输出格式。

更多内容，可参见 Flink 官方文档的 表中的模式检测。

Top-N

Top-N 查询可以在一批流数据中，不断输出当前最新的前 N 大或者前 N 小的记录，因此输出类型为 Upsert 流，需要写入支持 Upsert 数据流的数据目的（Sink）。

Top-N 语法详情请参见 Flink 官方文档的 Top-N 查询。

相邻数据去重

有时候上游输入的数据可能会包含连续的重复值，该查询语句可以将重复的数据删除，只保留一个。去重的语法，可参见 Flink 官方文档的 去重。

INSERT 语句

INSERT INTO

INSERT INTO 语句必须和 SELECT 子查询联用，SELECT 的数据会写入到指定的数据目的表（Table Sink）中。

语法

```
INSERT INTO 数据目的表
SELECT 子句
```

示例

将 SELECT 查询的结果插入名为 KafkaSink1 的数据目的表（Sink）。

```
INSERT INTO KafkaSink1
SELECT s1.time_, s1.client_ip, s1.uri, s1.protocol_version, s2.status_code, s2.date_
FROM KafkaSource1 AS s1, KafkaSource2 AS s2
WHERE s1.time_ = s2.time_ AND s1.client_ip = s2.client_ip;
```

Table Sink 注意事项

选择合适的 Connector 程序包

如果在 WITH 参数里指定了某个 Sink，那么请务必勾选相应的【内置 Connector】，或自行上传相应的 Connector 程序包。

如果缺少符合条件的 Connector 程序包，作业启动时会抛出org.apache.flink.table.api.ValidationException: Could not find any factory异常信息。

对于读写 Kafka 的场景，推荐使用不带版本号的 flink-connector-kafka程序包，并将 connector.version参数设置为 universal，以获得最新的功能适配。不建议选择 flink-connector-kafka-0.11 等带版本号的旧版程序包。

排除计算列

对于 INSERT INTO 的数据目的表，计算列是不考虑在内的。例如，某个 Sink 的定义如下，那么在 INSERT INTO MySink 后的 SELECT 语句，必须包含 a (VARCHAR) 和 b (BIGINT) 两个字段，且不允许加入 c 字段，因为 c 是虚拟的计算列。

```
CREATE TABLE MySink (
  a VARCHAR,
  b BIGINT,
  c AS PROCTIME()
) WITH ( ... ... );
```

Tuple 和 Upsert 数据流的区别

确保 Sink Table 定义了合适的 WITH 参数。例如有些 Connector 只支持作为数据源，不支持作为数据目的；还有的只支持 Tuple 类型的数据流，不支持 Upsert 数据流等。

SET 控制语句

Flink 配置项

简介

SET 语句可以调整作业的关键运行参数。目前大多数参数都可以通过 作业高级参数 功能来配置，只有极少数参数需要使用 SET 语句。

> 注意：
>
> SET 命令属于高级用法，请谨慎使用，避免参数配置不当而引起的运行时异常。
>
> SET 命令不支持注释，请不要在其后增加 -- 注释信息。
>
> SET 语句行尾需加上分号。

语法

SET 语句中字符串类型的参数值必须用半角单引号括起来，布尔值和数值型的可以不加单引号。

```
SET 配置项 = '参数值';
```

示例

配置 SQL 作业状态保留时间

针对 GROUP BY 和 JOIN 等大状态的语句，Flink 提供了 Idle State Retention Time 机制，用户可以通过设置状态的最短保留时间和最长保留时间，避免状态的无限制增长造成作业崩溃（OOM）。例如，下面的语句指定了状态的最短保留时间是5小时，最长保留时间是6小时，Flink 会在这两个时间点之间自行选择状态的清理时机。

> 注意：
>
> 状态的最短保留时间和最长保留时间之间的差值必须大于5分钟，否则 Flink 会报错并忽略该设置。
>
> 时间单位的写法遵循 Flink 配置的规范，例如可以写10min、也可以写3h、3hour、7day、7d等。数值和单位之间的空格为可选项。

```
SET execution.min-idle-state-retention = '5 h';
SET execution.max-idle-state-retention = '6 h';
```

配置 SQL 作业快照超时时间

对于 SQL 作业，默认情况下，快照超时时间最短为10分钟，最长为2倍的快照周期。例如，对于快照周期为60秒的作业，其快照超时时间是10分钟；而对于快照周期为10分钟的作业，其快照超时时间是20分钟。

如果您需要自定义快照的超时时间，可以使用如下的 SET 语句。

```
SET CHECKPOINT_TIMEOUT = '300 s';
```

> 注意：
>
> Flink 配置项（高级参数）execution.checkpointing.timeout 对 SQL 作业可能不生效，请使用本文中的 SET 语句设置 SQL 作业的快照超时时间。

启用 Table 的 mini-batch 支持

Flink SQL 对聚合提供了 mini-batch 支持，可以显著提升吞吐量。默认没有开启，因为会增加处理时延。如果希望使用 Mini-Batch，需要在 SQL 作业的编辑页面中添加以下语句，通过下面的设置项启用此功能（批次大小和延迟参数可以自行设置，但不可省略）：

```
set table.exec.mini-batch.enabled = true;
set table.exec.mini-batch.size = 5000;
set table.exec.mini-batch.allow-latency = '200 ms';
```

运算符和内置函数

概述

本节介绍TBDS Flink 提供了一组内置的数据转换函数。本页简要介绍了它们。如果你需要的函数尚不支持，可参考官网实现 用户自定义函数。

比较函数

比较函数的函数名和功能描述如下：

函数名	功能描述
value1 = value2	比较 value1 和 value2 是否相等，如果相等则返回 TRUE，如果不相等则返回 FALSE。 NULL 与任何值比较的结果均为 NULL，在 WHERE 条件中会被当作 FALSE。因此请使用 IS NULL 而不是 = NULL 来与 NULL 进行比较。 = 和 IS NOT DISTINCT FROM 的区别主要在于对 NULL 值的处理方式不同。
value1 <> value2	比较 value1 和 value2 是否不相等。如果不相等则返回 TRUE，否则返回 FALSE。
value1 > value2	比较 value1 是否大于 value2。如果大于返回 TRUE，否则返回 FALSE。
value1 >= value2	比较 value1 是否大于等于 value2。如果大于等于则返回 TRUE，否则返回 FALSE。
value1 < value2	比较 value1 是否小于 value2。如果小于则返回 TRUE，否则返回 FALSE。
value1 <= value2	比较 value1 是否小于等于 value2。如果小于等于则返回 TRUE，否则返回 FALSE。
value IS NULL	如果 value 为 NULL 则返回 TRUE，否则返回 FALSE。
value IS NOT NULL	如果 value 不为 NULL 则返回 TRUE，否则返回 FALSE。
value1 IS DISTINCT FROM value2	如果两个值不等（所有 NULL 值视为彼此相等），则返回 TRUE，否则返回 FALSE。

value1 IS NOT DISTINCT FROM value2	如果两个值相等（所有 NULL 值视为彼此相等），则返回 TRUE，否则返回 FALSE。
value1 BETWEEN [ASYMMETRIC | SYMMETRIC] value2 AND value3	如果 value1 大于等于 value2 且小于等于 value3，则返回 TRUE，否则返回 FALSE。 [ ] 内为可选参数，默认行为是 ASYMMETRIC。如果显式声明 SYMMETRIC，则 value2 和 value3 的顺序可以任意对调，不影响结果。
value1 NOT BETWEEN [ASYMMETRIC | SYMMETRIC] value2 AND value3	如果 value1 小于 value2 或者大于 value3，则返回 TRUE，否则返回 FALSE。 [ ] 内为可选参数，默认行为是 ASYMMETRIC。如果显式声明 SYMMETRIC，则 value2 和 value3 的顺序可以任意对调，不影响结果。
string1 LIKE string2	如果 string1 符合 string2 表示的 pattern 则返回 TRUE，否则返回 FALSE。
string1 NOT LIKE string2	如果 string1 不符合 string2 表示的 pattern 则返回 TRUE，否则返回 FALSE。
string1 SIMILAR TO string2	如果 string1 符合 string2 表示的正则表达式，则返回 TRUE，否则返回 FALSE。
string1 NOT SIMILAR TO string2	如果 string1 不符合 string2 表示的正则表达式，则返回 TRUE，否则返回 FALSE。
value IN (listItem [, listItem]*)	如果 value 处于 IN 后面的值列表中，则返回 TRUE。该语句等价于多个 OR 表达式的连接。 如果值列表中包含 NULL，则找不到时会返回 NULL，否则找不到时返回 FALSE。 如果 value 是 NULL，则结果永远为 NULL。
value NOT IN (listItem, [, listItem]*)	如果 value 不在 IN 后的值列表中，则返回 TRUE，否则返回 FALSE。
EXISTS (某个子查询)	如果子查询返回至少一行，则返回 TRUE，否则返回 FALSE。 该查询对内存压力较大，请谨慎使用。
value IN (某个子查询)	如果子查询返回的多行结果中，有一条等于 value 的值，则返回 TRUE，否则返回 FALSE。 该查询对内存压力较大，请谨慎使用。
value NOT IN (某个子查询)	如果子查询返回的多个结果中，没有一条等于 value，则返回 TRUE，否则返回 FALSE。 该查询对内存压力较大，请谨慎使用。

逻辑函数

逻辑函数用来执行逻辑运算，结果是布尔值（Boolean）。
逻辑状态有 TRUE、FALSE、UNKNOWN 三种（NULL 值的逻辑状态是 UNKNOWN），因而 NOT TRUE 不一定是 FALSE，还可能是 UNKNOWN。

函数名	功能描述
boolean1 OR boolean2	如果 boolean1 或者 boolean2 任意一个为 TRUE，则返回 TRUE，否则返回 FALSE。
boolean1 AND boolean2	当且仅当 boolean1 和 boolean2 均为 TRUE 时才返回 TRUE，否则返回 FALSE。
NOT boolean	如果 boolean 为 TRUE 则返回 FALSE；如果为 FALSE 则返回 TRUE；如果为 UNKNOWN 则返回 UNKNOWN。
boolean IS FALSE	如果 boolean 为 FALSE 则返回 TRUE；如果 boolean 为 UNKNOWN 则返回 FALSE。
boolean IS NOT FALSE	如果 boolean 不为 FALSE 则返回 TRUE；如果为 UNKNOWN 则返回 TRUE。
boolean IS UNKNOWN	如果 boolean 为 UNKNOWN 则返回 TRUE，否则返回 FALSE。
boolean IS NOT UNKNOWN	如果 boolean 不为 UNKNOWN 则返回 TRUE，否则返回 FALSE。

算术函数

算术函数的函数名和功能描述如下：

函数名	功能描述
+numeric	返回 numeric 本身。
-numeric	返回 0-numeric 的值，即反转符号。
numeric1 + numeric2	计算 numeric1 加 numeric2 的结果。
numeric1 - numeric2	计算 numeric1 减 numeric2 的结果。
numeric1 * numeric2	计算 numeric1 乘 numeric2 的结果。
numeric1 / numeric2	计算 numeric1 除以 numeric2 的结果。

POWER(numeric1, numeric2)	计算 numeric1 的 numeric2 次方。
ABS(numeric)	返回 numeric 的绝对值。
MOD(numeric1, numeric2)	返回 numeric1 除以 numeric2 的余数。如果 numeric1 是负数，那么余数也为负数。
SQRT(numeric)	计算 numeric 的平方根。
LN(numeric)	计算 numeric 的自然对数（以 e 为底）。
LOG10(numeric)	计算 numeric 以10为底的对数。
LOG2(numeric)	计算 numeric 以2为底的对数。
LOG(numeric2)LOG(numeric1, numeric2)	如果提供一个参数，计算 numeric2 的自然对数（等价于 LN）。如果提供两个参数，计算 numeric2 以 numeric1 为底的对数。
EXP(numeric)	计算 e 的 numeric 次方。
CEIL(numeric)CEILING(numeric)	返回 numeric 向上取整的值。
FLOOR(numeric)	返回 numeric 向下取整的值。
TRUNCATE(numeric1, numeric2)	对 numeric1 的小数部分以截断的方式取整，取整的位数由 numeric2 决定。 如果参数为 NULL，则结果也是 NULL。 如果 numeric2 为0或不填，则结果没有小数部分。numeric2 可为负数，此时对整数部分取整。例如 TRUNCATE(42.345, 2) 返回 42.34；TRUNCATE(42.345) 返回42.0；TRUNCATE(42.345, -1) 返回40.0。
SIN(numeric)	计算 numeric 的正弦值。
SINH(numeric)	计算 numeric 的双曲正弦值（返回值为 DOUBLE 类型）。
COS(numeric)	计算 numeric 的余弦值。
COSH(numeric)	计算 numeric 的双曲余弦值（返回值为 DOUBLE 类型）。
TAN(numeric)	计算 numeric 的正切值。
TANH(numeric)	计算 numeric 的双曲正切值（返回值为 DOUBLE 类型）。
COT(numeric)	计算 numeric 的余切值。
ASIN(numeric)	计算 numeric 的反正弦值。
ACOS(numeric)	计算 numeric 的反余弦值。

ATAN(numeric)	计算 numeric 的反正切值。
ATAN2(numeric1, numeric2)	计算 (numeric1、numeric2) 坐标点的四象限反正切值。
DEGREES(numeric)	将 numeric 从弧度转为角度。
RADIANS(numeric)	将 numeric 从角度转为弧度。
SIGN(numeric)	得到 numeric 的符号，负数是-1，0返回0，正数是1。
ROUND(numeric, int)	对 numeric 取整，位数由 int 值给定，可正也可负。
PI()	返回一个可以代表 π 的值。
E()	返回一个可以代表自然对数的底数 e 的值。
RAND()RAND(种子值)	返回一个0.0 - 1.0（不包含）的伪随机数，可以指定一个整数作为种子值。
RAND_INTEGER(上限值) RAND_INTEGER(种子值, 上限值)	返回一个0.0 - 指定上限值（不包含）的伪随机数。
UUID()	返回一个随机生成的 Type-4 UUID 字符串。
BIN(numeric)	获取 numeric 的二进制表示的字符串，例如输入4，则返回"100"。
HEX(numeric)HEX(string)	获取 numeric 或 string 的十六进制表示的字符串，例如输入15或"15" 则返回"F"。

条件函数

条件函数的函数名和功能描述如下：

函数名	功能描述
CASE valueWHEN value1 [, value11]* THEN result1[WHEN valueN [, valueN1]* THEN resultN]*[ELSE resultZ]END	当满足 value1 ~ value11 的任意值时，返回result1。 当满足 valueN ~ valueN1 的任意值时，返回resultN。 否则返回 resultZ。
CASEWHEN condition1 THEN result1[WHEN conditionN THEN resultN]*[ELSE resultZ]END	当满足 condition1 时返回 result1。 当满足 conditionN 时返回 resultN。 否则返回 resultZ。

NULLIF(value1, value2)	如果 value1 与 value2 相同，则返回 NULL，否则返回第一个值。例如 NULLIF(5, 5) 返回 NULL，而 NULLIF(5, 0) 返回5。
COALESCE(value, value [, value]*)	如果前值是 NULL，则提供一个后续的值，例如 COALESCE(NULL, 5) 则返回5。
IF(condition, true_value, false_value)	如果 condition 的条件满足，返回 true_value，否则返回 false_value。例如，IF(2 > 1, 2, 1) 返回2，而 IF (1 > 2, 99, 100) 返回100。
IS_ALPHA(string)	判断字符串是不是仅由纯字母组成。如果是，则返回 true，否则返回 false。
IS_DECIMAL(string)	判断字符串是不是一个合法的数字（整数、小数、负数均可）。如果是，则返回 true，否则返回 false。
IS_DIGIT(string)	判断字符串是不是仅由纯数字组成（即无符号整数）。如果是，则返回 true，否则返回 false。
IF_NULL_STR(str, defaultValue)	如果 str 不为 NULL，则返回 str 本身；如果 str 为 NULL，则返回第二项参数 defaultValue。

字符串操作函数

函数

函数名	功能描述
string1 || string2	连接两个字符串，返回两个字符串拼接后的结果，等同于 CONCAT(string1, string2)。
CHAR_LENGTH(string)	返回字符串的长度。
CHARACTER_LENGTH(string)	与 CHAR_LENGTH(string) 相同。
UPPER(string)	返回 string 的全大写字母形式。
LOWER(string)	返回 string 的全小写字母形式。
POSITION(string1 IN string2)	获取 string1 在 string2 中第一次出现的位置（位置从1开始计数）。当 string1 在 string2 中找不到时，返回0。
TRIM({BOTH |LEADING | TRAILING }string1 FROM string2)	从 string2 中除去字符串首尾/首/末尾的 string1。默认情况下，首尾的空格都被删除。

LTRIM(string)	去掉 string 字符串最左边的所有空格。例如 LTRIM(' Hello') 会返回 'Hello'。
RTRIM(string)	去掉 string 字符串最右边的所有空格。例如 RTRIM(' World ') 会返回 ' World'。
REPEAT(string, integer)	将 string 字符串重复 integer 次。例如 REPEAT('Meow', 3) 会返回 'MeowMeowMeow'。
REGEXP_REPLACE(string1, string2, string3)	对 string1 字符串以 string2 表示的正则表达式进行替换，替换内容是 string3。例如 REGEXP_REPLACE('banana', 'a|n', 'A') 返回 'bAAAAA'。
REPLACE(string1, string2, string3)	将 string1 字符串中所有的 string2 替换为 string3。例如 REPLACE('banana', 'a', 'A') 返回 'bAnAnA'。
OVERLAY(string1 PLACING string2 FROM start_pos [FOR length])	将 string1 从第 start_pos 位（start_pos 从1开始计数）开始的子串替换为 string2。可以指定替换的长度。
SUBSTRING(string from pos [FOR length])	获取从 pos 位开始的子串，默认行为是直到源字符串的最后，可以使用 FOR 来指定子串的长度。其中字符串起始 pos 从1开始计数，而不是 0。
REGEXP_EXTRACT(string1, string2[, integer])	从 string1 中提取正则分组，正则表达式为 string2，第一个括号为第一组，以此类推。可通过第三个参数 integer 来指定所需的分组号（从1开始）。如果不指定分组号或者分组号为0，则表示返回整个正则表达式匹配到的字符串。例如，REGEXP_EXTRACT('foothebar', 'foo(.*?)(bar)', 2) 返回 'bar'。
INITCAP(string)	将 string 中的单词，转为以大写字母开头，其他是小写字母（首字母大写）的形式。例如 INITCAP('i have a dream') 返回 'I Have A Dream'。
CONCAT(string1, string2 ...)	连接多个字符串。若任意字符串为 NULL，则结果为 NULL。
CONCAT_WS(separator, string1, string2, ...)	使用指定的分隔符 separator 连接多个字符串。如果 separator 为 NULL，则结果为 NULL。如果某个字符串为 NULL，则跳过它；但是不会跳过空字符串。例如 CONCAT_WS('~', 'AA','BB', '', 'CC') 会返回 AA~BB~~CC。
LPAD(text, length, padding)	使用 padding 指定的字符串从左侧填充 text 字符串到指定长度 length。如果 text 比 length 更长，则会截断到 length 的长度。
RPAD(text, length, padding)	使用 padding 指定的字符串从右侧填充 text 字符串到指定长度 length。如果 text 比 length 更长，则会截断到 length 的长度。
FROM_BASE64(string)	将 Base64 编码的 string 字符串解码为字符串。如果 string 为 NULL，则返回 NULL。

TO_BASE64(string)	将 string 表示的字符串编码为 Base64 字符串。
ASCII(string)	返回 string 字符串中第一个字符的 ASCII 码。如果 string 为 NULL，则返回 NULL。例如 ASCII('an apple') 返回97，因为首字母 'a' 的 ASCII 编码是97。
CHR(integer)	返回编码为 integer 的 ASCII 字符。例如 CHR(97) 返回 'a'。
ENCODE(string, charset)	将 string 字符串转码为 charset 指定的字符集编码的 BINARY 类型，例如 ENCODE(hello, 'GBK')。
DECODE(binary, charset)	将 binary 表示的 BINARY 类型以 charset 指定的字符集解码，例如 DECODE(binary_field, 'UTF-16LE')。
INSTR(string1, string2)	返回 string2 在 string1 字符串中首次出现的位置。如果任意参数为 NULL，结果为 NULL。
LEFT(string, n)	返回 string 从左起前 n 个字符。如果 n 为负数，则返回空字符串。如果任意参数为 NULL，结果为 NULL。
RIGHT(string, n)	返回 string 从右起后 n 个字符。如果 n 为负数，则返回空字符串。如果任意参数为 NULL，结果为 NULL。
LOCATE(string1, string2[, integer])	返回跳过 integer 个字符后，string1 在 string2 中首次出现的位置（参数顺序与 INSTR 函数相反）。如果未找到，则返回0。如果任意参数为 NULL，结果为 NULL。
REGEXP(string, regex)	如果 regex 表示的正则表达式可以匹配 string 中的字符串的任意子串，那么返回 TRUE，否则返回 FALSE。如果任意参数为 NULL，结果为 NULL。
REVERSE(string)	反转 string 字符串。如果任意参数为 NULL，结果为 NULL。
SPLIT_INDEX(string, separator, index)	将 string 表示的字符串以 separator 指定的分隔符拆分，并获取第 index 项，返回值为字符串 VARCHAR 类型。其中 index 从0开始计数。
SPLIT(string, separator)	将 string 表示的字符串以 separator 指定的分隔符拆分，并返回一个 Row 类型的对象。
STR_TO_MAP(string1[, string2, string3])	将 string1 字符串用 string2 提供的数据分隔符（默认为半角逗号，）和 string3 提供的键值间分隔符（默认为半角等号 =）进行拆分，结果为键值对 MAP<string, string> 类型。例如 STR_TO_MAP('k1=v1,k2=v2,k3=v3') 返回键值对（非字符串）{'k1': 'v1', 'k2': 'v2', 'k3': 'v3'}。
SUBSTR(string[, pos[, length]])	返回 string 字符串从 pos 位置开始，长度为 length 的子串。如果不提供 length，则默认到该字符串尾部。

示例

||

功能描述：连接两个字符串，返回两个字符串拼接后的结果，等同于 CONCAT(string1, string2)语法：string1 || string2示例测试语句：SELECT string1 || string2 FROM Test ;测试数据和结果：

测试数据（VARCHAR string1）	测试数据（VARCHAR string2）	测试结果（VARCHAR）
Oce	anus	Oceanus

CHAR_LENGTH

功能描述：返回字符串的长度。语法：CHAR_LENGTH(string)示例测试语句：SELECT CHAR_LENGTH(var1) AS length FROM Test;测试数据和结果：

测试数据（VARCHAR var1）	测试结果（INT length）
Oceanus	7

CHARACTER_LENGTH

功能描述：与 CHAR_LENGTH(string) 相同。语法：CHARACTER_LENGTH(string)示例测试语句：SELECT CHAR_LENGTH(var1) AS length FROM Test;测试数据和结果：

测试数据（VARCHAR var1）	测试结果（INT length）
Oceanus	7

LOWER

功能描述：返回小写字符的字符串。语法：LOWER(string)示例测试语句：SELECT LOWER(var1) AS lower FROM Test;测试数据和结果：

测试数据（VARCHAR var1）	测试结果（VARCHAR lower）
Flink	flink

UPPER

功能描述：返回大写字符的字符串。语法：UPPER(string)示例测试语句：SELECT UPPER(var1) AS upper FROM Test;测试数据和结果：

测试数据（VARCHAR var1）	测试结果（VARCHAR upper）
Flink	FLINK

TRIM

功能描述：从 string2 中除去字符串首尾/首/末尾的 string1。默认情况下，首尾的空格都被删除。语法：TRIM({BOTH | LEADING | TRAILING } string1 FROM string2)示例测试语句：SELECT TRIM(BOTH string1 FROM string2) AS res FROM Test;测试数据和结果：

测试数据（VARCHAR string1）	测试数据（VARCHAR string2）	测试结果（VARCHAR res）
a	aflinka	flink

CONCAT

功能描述：拼接两个或多个字符串值从而组成一个新的字符串。如果任一参数为 NULL 时，则跳过该参数。语法：CONCAT(string1, string2 ...) 示例测试语句：SELECT CONCAT('123', '456', 'abc', 'def') AS res FROM Test;测试数据和结果：'123456abcdef'

测试数据（VARCHAR string1）	测试数据（VARCHAR string2）	测试数据（VARCHAR string3）	测试数据（VARCHAR string4）	测试结果（VARCHAR res）
123	456	abc	def	123456abcdef

CONCAT_WS

功能描述：使用指定的分隔符 separator 连接多个字符串。如果 separator 为 NULL，则结果为 NULL。如果某个字符串为 NULL，则跳过它。但是不会跳过空字符串。语法：CONCAT_WS(separator,string1,string2, ...)示例测试语句：SELECT CONCAT_WS(separator, string1,string2, string3) AS res FROM Test;测试数据和结果：

测试数据（VARCHAR separator）	测试数据（VARCHAR string1）	测试数据（VARCHAR string2）	测试数据（VARCHAR string3）	测试结果（VARCHAR res）
-	AA	BB	CC	AA-BB-CC

INITCAP

功能描述：将 string 中的单词，转为以大写字母开头，其他是小写字母（首字母大写）的形式。语法：INITCAP(string)示例测试语句：SELECT INITCAP(var1) AS str FROM Test;数据和结果：

测试数据（VARCHAR var1）	测试结果（VARCHAR str）
i have a dream	I Have A Dream

IS_ALPHA

功能描述：判断字符串是否只包含字母。语法：IS_ALPHA(content)示例测试语句：SELECT IS_ALPHA(content) AS result FROM Test;测试数据和结果：

测试数据（VARCHAR content）	测试结果（BOOLEAN result）
flink	true
flink123	false
''	false
null	false

IS_DIGITS

功能描述：判断字符串是否只包含数字。语法：IS_DIGITS(content)示例测试语句：SELECT IS_DIGITS(content) AS result FROM Test;测试数据和结果：

测试数据（VARCHAR content）	测试结果（BOOLEAN case_result）
58.0	true
58	true
58pl	false
''	false
null	false

LPAD

功能描述：使用 padding 指定的字符串从左侧填充 text 字符串到指定长度 length。如果 text 比 length 更长，则会截断到 length 的长度。语法：LPAD(text , length , padding)示例测试语句：SELECT LPAD(test, length, padding) AS res FROM Test;测试数据和结果：

测试数据（VARCHAR text）	测试数据（INT length）	测试数据（VARCHAR padding）	测试结果（VARCHAR res）
flink	3	hello	hel
flink	-1	hello	''
flink	12	hello	helloflink

RPAD

功能描述：使用 padding 指定的字符串从右侧填充 text 字符串到指定长度 length。如果 text 比 length 更长，则会截断到 length 的长度。语法：RPAD(text , length , padding)示例测试语句：SELECT RPAD(text, length, padding) AS res FROM Test;测试数据和结果：

测试数据（VARCHAR text）	测试数据（INT length）	测试数据（VARCHAR padding）	测试结果（VARCHAR res）
flink	3	hello	oce
flink	-1	hello	''
flink	12	hello	flinkhello

类型转换函数

函数

> 注意：
>
> 如果使用 CAST() 函数，将时间段 INTERVAL 转为数字，则结果会是字面值（可能不符合预期）。例如 CAST(INTERVAL '1234' MINUTE AS BIGINT)，则结果会是字面值1234，而非时间段表示的毫秒值。

函数名	功能描述
CAST(value AS type)	将某个值转为 type 类型，例如 CAST(hello AS VARCHAR) 会将 hello 字段转为 VARCHAR 类型。
TRY_CAST(value AS type)	TRY_CAST 行为与 CAST 相似, 但在遇到错误时 TRY_CAST 会返回 NULL 而不是使作业失败。 例如：TRY_CAST('42' AS INT) 返回 42; TRY_CAST(NULL AS STRING) 返回字符串类型的 NULL; TRY_CAST('non-number' AS INT) 返回 INT 类型的 NULL; COALESCE(TRY_CAST('non-number' AS INT), 0) 返回 INT 类型的 0。
TYPEOF(input) | TYPEOF(input, force_serializable)	返回输入表达式的数据类型的字符串表示。默认情况下返回的字符串是一个摘要字符串，可能会为了可读性而省略某些细节。 如果 force_serializable 设置为 TRUE，则字符串表示可以持久化保存在 catalog 中的完整数据类型。 请注意，特别是匿名的内联数据类型没有可序列化的字符串表示。在这种情况下返回 NULL。

示例

CAST

功能描述：将某个值转为 type 类型。语法：CAST(value AS type)示例测试语句：SELECT CAST(var1 AS VARCHAR) FROM TEST;测试数据和结果：

测试数据（INT var1）	测试结果 VARCHAR
58	'58'

时间相关函数

时间相关函数的函数名和功能描述如下：

函数名	功能描述
DATE string	将 "yyyy-MM-dd" 形式表示的字符串转为 SQL 日期（DATE）类型。
TIME string	将 "HH:mm:ss" 形式表示的字符串转为 SQL 时间（TIME）类型。
TIMESTAMP string	将 "yyyy-MM-dd HH:mm:ss[.SSS]" 形式的字符串转为 SQL 时间戳（TIMESTAMP）类型。
INTERVAL string range	接受 "dd hh:mm:ss.fff" 形式的字符串（毫秒），或者 "yyyy-MM" 形式的字符串（月）。 对于毫秒，可以接受 DAY、MINUTE、DAY TO HOUR、DAY TO SECOND 作为 range。 对于月，接受 YEAR 或 YEAR TO MONTH 作为 range。例如 INTERVAL '10 00:00:00.004' DAY TO SECOND、INTERVAL '10' DAY、INTERVAL '2-10' YEAR TO MONTH。
CURRENT_DATE	返回当前的 SQL DATE 格式表示的日期（UTC）。
CURRENT_TIME	返回当前的 SQL TIME 格式表示的时间（UTC）。
CURRENT_TIMESTAMP	返回当前 SQL TIMESTAMP 格式表示的时间戳（UTC）。
LOCALTIME	返回本地时区表示的 SQL 时间。
LOCALTIMESTAMP	返回本地时区表示的 SQL 时间戳。
EXTRACT(timeintervalunit FROM temporal)	获取时间点或时间段字符串中的某项。例如 EXTRACT(DAY FROM DATE '2006-06-05') 返回5，而 EXTRACT(YEAR FROM DATE '2018-06-12') 则返回2018。
YEAR(date)	返回指定日期中的年份，等价于 EXTRACT(YEAR FROM date)。例如 YEAR(DATE '2020-08-12') 返回2020。
QUARTER(date)	返回指定日期的季度，等价于 EXTRACT(QUARTER FROM date)。例如 QUARTER(DATE '2012-09-10')返回3。
MONTH(date)	返回指定日期的月份，等价于 EXTRACT(MONTH FROM date)。例如 MONTH(DATE '2012-09-10') 返回9。
WEEK(date)	返回指定日期的周数，即当年的第几周，等价于 EXTRACT(WEEK FROM date)。例如 WEEK(DATE '1994-09-27') 返回39。
DAYOFYEAR(date)	返回指定日期在当年的天数，即当年的第几天（范围是[1，366]），等价于 EXTRACT(DOY FROM date)。例如 DAYOFYEAR(DATE

	'1994-09-27') 返回270。
DAYOFMONTH(date)	返回指定日期在当月的天数，即当月的第几天（范围是[1，31]），等价于 EXTRACT(DAY FROM date)。例如 DAYOFMONTH(DATE '1994-09-27') 返回27。
DAYOFWEEK(date)	返回指定日期在本周的天数，即本周的第几天（范围是[1，7]），等价于 EXTRACT(DOW FROM date)。例如 DAYOFWEEK(DATE '1994-09-27') 返回3。
HOUR(timestamp)	返回指定时间戳的小时部分（范围是[0，23]）。等价于 EXTRACT(HOUR FROM timestamp)。例如 HOUR('2017-10-02 12:25:44') 返回12。
MINUTE(timestamp)	返回指定时间戳的分钟部分（范围是[0，59]）。等价于 EXTRACT(MINUTE FROM timestamp)。例如 MINUTE('2017-10-02 12:25:44') 返回25。
SECOND(timestamp)	返回指定时间戳的秒部分（范围是[0，59]）。等价于 EXTRACT(SECOND FROM timestamp)。例如 SECOND('2017-10-02 12:25:44') 返回44。
FLOOR(timepoint TO timeintervalunit)	将一个时间点向下取整，例如 FLOOR(TIME '12:44:31' TO MINUTE) 返回12:44:00。
CEIL(timepoint TO timeintervalunit)	将一个时间点向上取整，例如 CEIL(TIME '12:44:31' TO MINUTE) 返回12:45:00。
(timepoint, temporal) OVERLAPS (timepoint, temporal)	判断两个时间段是否重叠。例如 (TIME'2:55:00', INTERVAL '1' HOUR) OVERLAPS (TIME'3:30:00', INTERVAL '2' HOUR) 返回 TRUE；而 (TIME'9:00:00', TIME '10:00:00') OVERLAPS (TIME'10:15:00', INTERVAL '3' HOUR) 返回 FALSE。
TO_TIMESTAMP(string, simple_format)	将字符串格式的时间戳转为 Timestamp 类型。
DATE_FORMAT(timestamp, format)	将 Timestamp 类型的时间戳格式化为字符串。
TO_DATE(string1[, string2])	将 string1 字符串以 string2 的格式（可选，默认是 'yyyy-MM-dd HH:mm:ss'）转为 DATE 格式。
TO_TIMESTAMP(string1[, string2])	将 string1 字符串以 string2 的格式（可选，默认是 'yyyy-MM-dd HH:mm:ss'）转为 TIMESTAMP 格式。默认使用 UTC+8 时区。

聚合函数

函数名	功能描述
COUNT([ALL] expression | DISTINCT expression1 [, expression2]*)	默认情况和 ALL 时，返回 expression 表达式筛选后，非 NULL 值的输入行数。如果是 DISTINCT，则会先对数据进行去重，然后再进行统计总行数。
COUNT(*) COUNT(1)	返回输入的总行数，含 NULL 值。
AVG([ALL | DISTINCT] expression)	默认情况和 ALL 时，返回 expression 表达式筛选后，所有输入的算术平均值。如果是 DISTINCT，则会先对数据进行去重，然后再进行统计求平均。
SUM([ALL | DISTINCT] expression)	默认情况和 ALL 时，返回 expression 表达式筛选后，所有输入和。如果是 DISTINCT，则会先对数据进行去重，然后再进行统计求和。
MAX([ALL | DISTINCT] expression)	默认情况和 ALL 时，返回 expression 表达式筛选后，所有输入的最大值（不可用于 TIMESTAMP 类型）。如果是 DISTINCT，则会先对数据进行去重，然后再进行统计求最大值。
MIN([ALL | DISTINCT] expression)	默认情况和 ALL 时，返回 expression 表达式筛选后，所有输入的最小值（不可用于 TIMESTAMP 类型）。如果是 DISTINCT，则会先对数据进行去重，然后再进行统计求最小值。
STDDEV_POP([ALL | DISTINCT] expression)	默认情况和 ALL 时，返回 expression 表达式筛选后，所有输入的总体标准差。如果是 DISTINCT，则会先对数据进行去重，然后再进行统计求总体标准差。
STDDEV_SAMP([ALL | DISTINCT] expression)	默认情况和 ALL 时，返回 expression 表达式筛选后，所有输入的样本标准差。如果是 DISTINCT，则会先对数据进行去重，然后再进行统计求样本标准差。
VAR_POP([ALL | DISTINCT] expression)	默认情况和 ALL 时，返回 expression 表达式筛选后，所有输入的总体方差。如果是 DISTINCT，则会先对数据进行去重，然后再进行统计求总体方差。
VAR_SAMP([ALL | DISTINCT] expression)	默认情况和 ALL 时，返回 expression 表达式筛选后，所有输入的样本方差。如果是 DISTINCT，则会先对数据进行去重，然后再进行统计求样本方差。两种写法等价。
COLLECT([ALL | DISTINCT] expression)	默认情况和 ALL 时，返回 expression 表达式筛选后，所有输入的非 NULL 输入的 MULTISET 集合（允许重复值）。如果所有值都是 NULL，则返回一个空集。
RANK()	返回某个数据在一组数据中的排名，前后调用的结果可能不连续。例如有五个数据，其中两个并列第二，那么 RANK() 的结果是1、2、2、4、5。

DENSE_RANK()	返回某个数据在一组数据中的排名，前后调用的结果保证连续。例如有五个数据，其中两个并列第二，那么 RANK() 的结果是1、2、2、3、4。
ROW_NUMBER()	为一组数据的每行分配一个递增且连续的值，从1开始，不会重复。例如有五个数据，其中两个并列第二，那么 RANK() 的结果是1、2、3、4、5。
LEAD(expression [, offset] [, default])	在窗口计算中，访问当前行之后 offset 行的数据，默认 offset 为1，即访问下一行的数据。default 表示无数据时的默认值，如果不提供，默认为 NULL。
LAG(expression [, offset] [, default])	在窗口计算中，访问当前行之前 offset 行的数据，默认 offset 为1，即访问上一行的数据。default 表示无数据时的默认值，如果不提供，默认为 NULL。
FIRST_VALUE(expression)	返回一系列数据中，第一个数据。
LAST_VALUE(expression)	返回一系列数据中，最后一个数据。
LISTAGG(expression [, separator])	将一组数据使用给定的分隔符进行连接，最终返回一个连接后的字符串。默认分隔符是半角逗号“,”，类似于其他语言的 String.join() 方法。

示例

为方便演示，建立示例测试数据表 Test：

id	site_id	count	date
1	1	45	2021-07-10
2	3	100	2021-07-13
3	1	230	2021-07-14
4	2	10	2021-07-14
5	5	205	2021-07-14
6	4	13	2021-07-15
7	3	220	2021-07-15
8	5	545	2021-07-16
9	3	201	2021-07-17

COUNT

功能描述：统计总行数。语法：COUNT([ALL] expression | DISTINCT expression1 [, expression2] *)示例测试语句：SELECT COUNT(*) FROM Test;测试结果：

测试语句	测试结果（nums）
SELECT COUNT(*) AS nums FROM Test;	9
SELECT COUNT(DISTINCT site_id) AS nums FROM Test;	5

AVG

功能描述： 统计求平均。语法：AVG([ALL | DISTINCT] expression)示例测试语句： SELECT AVG(count) FROM Test;测试数据和结果：

测试语句	测试结果（nums）
SELECT AVG(count) AS nums FROM Test;	176.3

SUM

功能描述：统计求和。语法： SUM([ALL | DISTINCT] expression)示例测试语句： SELECT SUM(count) FROM Test;测试数据和结果：

测试语句	测试结果（nums）
SELECT SUM(count) AS nums FROM Test;	1569

MAX

功能描述：统计求最大值。语法： MAX([ALL | DISTINCT] expression)示例测试语句： SELECT MAX(count) FROM Test;测试数据和结果：

测试语句	测试结果（nums）
SELECT MAX(count) AS nums FROM Test;	545

MIN

功能描述：统计求最小值。语法： MIN([ALL | DISTINCT] expression)示例测试语句： SELECT MIN(count) FROM Test;测试数据和结果：

测试语句	测试结果（nums）
SELECT MIN(count) AS nums FROM Test;	13

STDDEV_POP

功能描述：统计求总体标准差。语法： STDDEV_POP([ALL | DISTINCT] expression)示例测试语句： SELECT

STDDEV_POP(count) FROM Test;测试数据和结果：

测试语句	测试结果（pop）
SELECT STDDEV_POP(count) AS pop FROM Test;	156.18

VAR_POP
功能描述：统计求总体方差。语法： VAR_POP([ALL | DISTINCT] expression)示例测试语句： SELECT VAR_POP(count) FROM Test;测试数据和结果：

测试语句	测试结果（pop）
SELECT VAR_POP(count) AS pop FROM Test;	24390.7

FIRST_VALUE
功能描述：返回一系列数据中，第一个数据。语法： FIRST_VALUE(expression)示例测试语句： SELECT FIRST_VALUE(count) FROM Test;测试数据和结果：

测试语句	测试结果（first）
SELECT FIRST_VALUE(count) AS first FROM Test;	45

LAST_VALUE
功能描述：返回一系列数据中，最后一个数据。语法： LAST_VALUE(expression)示例测试语句： SELECT LAST_VALUE(count) FROM Test;测试数据和结果：

测试语句	测试结果（last）
SELECT LAST_VALUE(count) AS last FROM Test;	201

时间窗口函数

在流式计算中，流通常是无穷无尽的，我们无法知道什么时候数据源会继续/停止发送数据，所以在流上处理聚合事件（count、sum 等）的处理方式与批处理中的处理方式会有所差异。在流上一般用窗口（Windows）来限定聚合的范围，例如“过去2分钟网站点击量的计数”、“在最近100个人中点赞这个视频的总人数”。窗口的概念相当于帮我们收集了一张有限数据的动态表，我们可以对表中的数据进行聚合计算。
窗口函数是一种特殊的函数，它并不在 SELECT 的投影列表中使用，而是在 GROUP BY 子句中使用。流计算 Flink 支持三种类型的窗口函数 TUMBLE、HOP 和 SESSION。
Flink 流处理介绍详见 及时流处理。
TUMBLE WINDOW

TUMBLE WINDOW（滚动窗口）将每个进入的数据分配到一个指定窗口大小的窗口中。滚动窗口可以自定义固定的大小，并且不会出现重叠。我们可以对窗口内的数据进行计算。
语法

```
TUMBLE(time_attr, interval)
```

time_attr 参数表示时间戳字段，表示每条记录被处理的时间戳。如果指定为 PROCTIME 是自动生成的时间戳，记录了数据被 flink 处理的时刻，一般用在 Processing Time 模式下。
interval 参数用来设置窗口大小。例如，设置为1天：INTERVAL '1' DAY；设置为2小时：INTERVAL '2' HOUR，其他用法可参见 时间相关函数。

> 注意：
>
> 如果在 Event Time 时间模式下（使用 WATERMARK FOR 语句定义了时间戳字段），那么 TUMBLE、HOP、SESSION 窗口函数的第一个参数必须为该字段。
>
> 如果在 Processing Time 时间模式下，则 TUMBLE、HOP、SESSION 窗口函数的第一个参数必须为 proctime() 函数生成的计算列，下文用 PROCTIME 举例，请在实际作业中替换为实际的列名。

标识函数
标识函数用来表示窗口的起始与结束时间。

函数名	功能描述
TUMBLE_START(time-attr, size-interval)	返回该窗口的起始时间
TUMBLE_END(time-attr, size-interval)	返回该窗口的结束时间

模拟用例
下文以 TUMBLE WINDOW 为例，帮助您更容易地理解 TUMBLE WINDOW。使用 Event Time 模拟统计每小时各用户收入金额。
示例数据：

username（VARCHAR）	income(BIGINT)	times(TIMESTAMP)
Tom	20	2021-11-11 10:30:00.0
Jack	10	2021-11-11 10:35:00.0
Tom	10	2021-11-11 10:35:00.0
Tom	10	2021-11-11 10:40:00.0
Tom	15	2021-11-11 11:30:00.0

Jack	10	2021-11-11 11:30:00.0
Jack	15	2021-11-11 11:40:00.0

SQL 语句：

```
CREATE TABLE user_income (
    username VARCHAR,
    Income INT,
    times TIMESTAMP(3),
    WATERMARK FOR times AS times - INTERVAL '3' SECOND
) WITH (
    ...
  );

CREATE TABLE output (
        win_start TIMESTAMP,
        win_end TIMESTAMP,
        username VARCHAR,
        hour_income BIGINT
)WITH(
    ...
);

INSERT INTO output
SELECT
TUMBLE_START(times,INTERVAL '1' HOUR),
TUMBLE_END(times,INTERVAL '1' HOUR),
username,
SUM(Income)
FROM user_income
GROUP BY TUMBLE(times,INTERVAL '1' HOUR),username;
```

输出结果：

win_start(TIMESTAMP)	win_end(TIMESTAMP)	username(VARCHAR)	hour_income(BIGINT)
2021-11-11 10:00:00.0	2021-11-11 11:00:00.0	Tom	40
2021-11-11 10:00:00.0	2021-11-11 11:00:00.0	Jack	10
2021-11-11 11:00:00.0	2021-11-11 12:00:00.0	Tom	15
2021-11-11 11:00:00.0	2021-11-11 12:00:00.0	Jack	25

HOP WINDOW

HOP WINDOW（滑动窗口）将元素分配到固定长度的窗口中，与滚动窗口类似，窗口的大小由窗口大小参数来配置，另一个窗口滑动参数控制滑动窗口开始的频率。

HOP WINDOW（滑动窗口）保持窗口大小（Size）不变，每次滑动指定的时间周期（Slide），因而允许窗口之间的相互重叠。

Slide 的大小决定了 Flink 创建新窗口的频率。

当 Slide 小于 Size 时（如图 window1 与 window2），相邻窗口会重叠，一个时间会被分配到多个窗口。

当 Slide 大于 Size 时（如图 window1 与 window4），可能会导致有些事件被丢弃。

当 Slide 等于 Size 时（如图 window1 与 window3），等于是 TUMBLE WINDOW。

语法

HOP(time_attr, sliding_interval, window_size_interval)

time_attr 参数表示时间戳字段，表示每条记录被处理的时间戳。如果指定为 PROCTIME 是自动生成的时间戳，记录了数据被 flink 处理的时刻，一般用在 Processing Time 模式下。

window_size_interval 参数用来设置窗口大小。例如，设置为1天：INTERVAL '1' DAY；设置为2小时：INTERVAL '2' HOUR，其他用法可参见 时间相关函数。

sliding_interval 参数用来设置滑动时间周期大小。例如，设置为1天：INTERVAL '1' DAY；设置为2小时：INTERVAL '2' HOUR，其他用法可参见 时间相关函数。

标识函数

标识函数用来表示窗口的起始与结束时间。

函数名	功能描述
HOP_START(time-attr, slide-interval,size-interval)	返回该窗口的起始时间
HOP_END(time-attr, slide-interval,size-interval)	返回该窗口的结束时间

模拟用例

下文以 HOP WINDOW 为例，帮助您更容易地理解 HOP WINDOW。使用 Event Time 模拟统计每小时各用户收入金额，每30分钟更新一次。1小时的窗口，10分钟滑动一次。

样例数据：

username（VARCHAR）	income(BIGINT)	times(TIMESTAMP)
Tom	20	2021-11-11 10:30:00.0
Jack	10	2021-11-11 10:35:00.0
Tom	10	2021-11-11 10:35:00.0
Tom	10	2021-11-11 10:40:00.0
Tom	15	2021-11-11 11:35:00.0

Jack	10	2021-11-11 11:30:00.0
Jack	15	2021-11-11 11:40:00.0

SQL 语句：

```
CREATE TABLE user_income (
    username VARCHAR,
    Income INT,
    times TIMESTAMP(3),
    WATERMARK FOR times AS times - INTERVAL '3' MINUTE
)WITH(
    ...
 );

CREATE TABLE output (
        win_start TIMESTAMP,
        win_end TIMESTAMP,
        username VARCHAR,
        hour_income BIGINT
)WITH(
    ...
 );

INSERT INTO output
SELECT
HOP_START(times,INTERVAL '30' MINUTE,INTERVAL '1' HOUR),
HOP_END(times,INTERVAL '30' MINUTE,INTERVAL '1' HOUR),
username,
SUM(income)
FROM user_income
GROUP BY HOP(times,INTERVAL '30' MINUTE,INTERVAL '1' HOUR),username;
```

输出结果：

win_start(TIMESTAMP)	win_end(TIMESTAMP)	username(VARCHAR)	hour_income(BIGINT)
2021-11-11 10:00:00.0	2021-11-11 11:00:00.0	Tom	40
2021-11-11 10:00:00.0	2021-11-11 11:00:00.0	Jack	10
2021-11-11 10:30:00.0	2021-11-11 11:30:00.0	Jack	10
2021-11-11 10:30:00.0	2021-11-11 11:30:00.0	Tom	40
2021-11-11 11:00:00.0	2021-11-11 12:00:00.0	Tom	15

2021-11-11 11:00:00.0	2021-11-11 12:00:00.0	Jack	25
2021-11-11 11:30:00.0	2021-11-11 12:30:00.0	Jack	25
2021-11-11 11:30:00.0	2021-11-11 12:30:00.0	Tom	15

SESSION WINDOW

SESSION WINDOW（会话窗口）通过 session 活动对元素进行分组，session 窗口与滚动窗口和滑动窗口相比，不会有重叠和固定的开始时间和结束时间的情况，相反，当它在一个固定的时间周期内不再收到元素，即非活动间隔产生，那么这个窗口就会关闭。一个 session 窗口通过一个 session 间隔来配置。这个 session 间隔定义了非活跃周期的长度，当这个非活跃周期产生，那么当前的 session 将关闭并且后续的元素将被分配到新的 session 窗口中。

Session Window 并非以长度来划分窗口，而是以非活跃时间来划分。例如超过30分钟不活跃（没有新数据），则之前的窗口结束，下一个来到的数据将会形成一个新窗口。

语法

```
SESSION(time_attr, interval)
```

time_attr 参数表示时间戳字段，表示每条记录被处理的时间戳。如果指定为 PROCTIME 是自动生成的时间戳，记录了数据被 flink 处理的时刻，一般用在 Processing Time 模式下。

interval 参数用来设置窗口大小。例如，设置为1天：INTERVAL '1' DAY；设置为2小时：INTERVAL '2' HOUR，其他用法可参见 时间相关函数。

标识函数

标识函数用来表示窗口的起始与结束时间。

函数名	功能描述
SESSION_START(time-attr, size-interval)	返回该窗口的起始时间
SESSION_END(time-attr, size-interval)	返回该窗口的结束时间

模拟用例

下文以 SESSION WINDOW 为例，帮助您更容易地理解 SESSION WINDOW。使用 Event Time 模拟统计每小时各用户收入金额，会话超时时长为30分钟。

样例数据：

username（VARCHAR）	income(BIGINT)	times(TIMESTAMP)
Tom	20	2021-11-11 10:30:00.0
Jack	10	2021-11-11 10:35:00.0
Tom	10	2021-11-11 10:35:00.0

Tom	10	2021-11-11 10:40:00.0
Tom	15	2021-11-11 11:50:00.0
Jack	10	2021-11-11 11:40:00.0
Jack	15	2021-11-11 11:45:00.0

SQL 语句：

```
CREATE TABLE user_income (
    username VARCHAR,
    Income INT,
    times TIMESTAMP(3),
    WATERMARK FOR times AS times - INTERVAL '3' MINUTE
)WITH(
    ...
 );

CREATE TABLE output (
        win_start TIMESTAMP,
        win_end TIMESTAMP,
        username VARCHAR,
        hour_income BIGINT
)WITH(
    ...
 );

INSERT INTO output
SELECT
SESSION_START(times,INTERVAL '30' MINUTE),
SESSION_END(times,INTERVAL '30' MINUTE),
username,
SUM(Income)
FORM user_income
GROUP BY SESSION(times,INTERVAL '30' MINUTE),username;
```

输出结果：

win_start(TIMESTAMP)	win_end(TIMESTAMP)	username(VARCHAR)	hour_income(BIGINT)
2021-11-11 10:30:00.0	2021-11-11 11:10:00.0	Tom	40
2021-11-11 10:35:00.0	2021-11-11 11:05:00.0	Jack	10

2021-11-11 11:30:00.0	2021-11-11 12:00:00.0	Tom	15
2021-11-11 11:30:00.0	2021-11-11 12:10:00.0	Jack	25

更多说明

以上三种窗口都有对应的辅助函数。以 TUMBLE 窗口为例（HOP、SESSION 也一样，只是前缀不同），辅助函数如下：

TUMBLE_ROWTIME：表示 TUMBLE 窗口的末端界限（包含，可用作 JOIN 或 GROUP 以及 OVER 条件，Event Time 时间模式下使用）。示例如下：

```
SELECT user,
TUMBLE_START(rowtime, INTERVAL '12' HOUR) AS sStart,
TUMBLE_ROWTIME(rowtime, INTERVAL '12' HOUR) AS snd,
SUM(amount)
FROM Orders
GROUP BY TUMBLE(rowtime, INTERVAL '12' HOUR), user
```

TUMBLE_PROCTIME：表示 TUMBLE 窗口的末端界限（包含，可用作 JOIN 或 GROUP 以及 OVER 条件，Processing Time 时间模式下使用）。示例如下：

```
SELECT user,
TUMBLE_START(PROCTIME, INTERVAL '12' HOUR) AS sStart,
TUMBLE_PROCTIME(PROCTIME, INTERVAL '12' HOUR) AS snd,
SUM(amount)
FROM Orders
GROUP BY TUMBLE(PROCTIME, INTERVAL '12' HOUR), user
```

标志符和保留字

命名规则

标识符用来唯一地表示表名或列名。当前流计算 Flink 中标识符的命名规则如下：
不能使用 数据类型 中所指定的保留字。若必须使用这些保留字作为表名或列名，请使用反引号（``）将其括起来，例如 time 。
请不要使用 PROCTIME 和 SOURCETIME 作为列名，以避免与系统自动生成的时间戳发生冲突。
不能以 *DataStreamTable* 开头。
如果表名或列名含有空格或者特殊字符，需使用反引号将其括起来，例如 HELLO WORLD 。
长度必须小于等于128个英文（半角）字符（一个汉字或全角字符等价于两个英文字符）。

保留字

TBDS Flink内部的保留字如下所示。在表名或列名中使用这些保留字时，必须使用反引号（``）括起来，否则语法检查时会报错。

> 注意：
>
> 在 Processing Time 时间模式下（即未使用 WATERMARK FOR 来定义数据源的时间戳字段），请不要使用 PROCTIME 作为列名，以免与系统自动生成的时间戳发生命名冲突。

A

A, ABS, ABSOLUTE, ACTION, ADA, ADD, ADMIN, AFTER, ALL, ALLOCATE, ALLOW,
ALTER, ALWAYS, AND, ANY, ARE, ARRAY, AS, ASC, ASENSITIVE, ASSERTION,
ASSIGNMENT, ASYMMETRIC, AT, ATOMIC, ATTRIBUTE, ATTRIBUTES, AUTHORIZATION, AVG

B

BEFORE, BEGIN, BERNOULLI, BETWEEN, BIGINT, BINARY, BIT, BLOB, BOOLEAN, BOTH BREADTH, BY

C

C, CALL, CALLED, CARDINALITY, CASCADE, CASCADED, CASE, CAST, CATALOG, CATALOG_NAME,
CEIL, CEILING, CENTURY, CHAIN, CHAR, CHARACTER, CHARACTERISTICTS, CHARACTERS,
CHARACTER_LENGTH, CHARACTER_SET_CATALOG, CHARACTER_SET_NAME, CHARACTER_SET_SCHEM
A,
CHAR_LENGTH, CHECK, CLASS_ORIGIN, CLOB, CLOSE, COALESCE, COBOL, COLLATE, COLLATION,

COLLATION_CATALOG, COLLATION_NAME, COLLATION_SCHEMA, COLLECT, COLUMN, COLUMN_NAME,
COMMAND_FUNCTION, COMMAND_FUNCTION_CODE, COMMIT, COMMITTED,CONDITION, CONDITION_NUMBER,
CONNECT, CONNECTION, CONNECTION_NAME, CONSTRAINT, CONSTRAINTS, CONSTRAINT_CATALOG,
CONSTRAINT_NAME, CONSTRAINT_SCHEMA, CONSTRUCTOR, CONTAINS, CONTINUE, CONVERT, CORR,
CORRESPONDING, COUNT, COVAR_POP, COVAR_SAMP, CREATE, CROSS, CUBE, CUME_DIST, CURRENT,
CURRENT_CATALOG, CURRENT_DATE, CURRENT_DEFAULT_TRANSFORM_GROUP, CURRENT_PATH, CURRENT_ROLE,
CURRENT_SCHEMA, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_TRANSFORM_GROUP_FOR_TYPE,
CURRENT_USER, CURSOR, CURSOR_NAME, CYCLE, DATA, DATABASE, DATE, DATETIME_INTERVAL_CODE

D

DATETIME_INTERVAL_PRECISION, DAY, DEALLOCATE, DEC, DECADE, DECIMAL, DECLARE, DEFAULT,
DEFAULTS, DEFERRABLE, DEFERRED, DEFINED, DEFINER, DEGREE, DELETE, DENSE_RANK, DEPTH, DEREF,
DERIVED, DESC, DESCRIBE, DESCRIPTION, DESCRIPTOR, DETERMINISTIC, DIAGNOSTICS, DISALLOW, DISCONNECT,
DISPATCH, DISTINCT, DOMAIN, DOUBLE, DOW, DOY, DROP, DYNAMIC, DYNAMIC_FUNCTION, DYNAMIC_FUNCTION_CODE

E

EACH, ELEMENT, ELSE, END, END-EXEC, EPOCH, EQUALS, ESCAPE, EVERY, EXCEPT, EXCEPTION,
EXCLUDE, EXCLUDING, EXEC, EXECUTE, EXISTS, EXP, EXPLAIN, EXTEND, EXTERNAL, EXTRACT

F

FALSE, FETCH, FILTER, FINAL, FIRST, FIRST_VALUE, FLOAT, FLOOR, FOLLOWING, FOR, FOREIGN,
FORTRAN, FOUND, FRAC_SECOND, FREE, FROM, FULL, FUNCTION, FUSION

G

G, GENERAL, GENERATED, GET, GLOBAL, GO, GOTO, GRANT, GRANTED, GROUP, GROUPING

H

HAVING, HIERARCHY, HOLD, HOUR

I

IDENTITY, IMMEDIATE, IMPLEMENTATION, IMPORT, IN, INCLUDING, INCREMENT, INDICATOR, INITIALLY, INNER, INOUT, INPUT, INSENSITIVE, INSERT, INSTANCE, INSTANTIABLE, INT, INTEGER, INTERSECT, INTERSECTION, INTERVAL, INTO, INVOKER, IS, ISOLATION

J

JAVA, JOIN

K

K, KEY, KEY_MEMBER, KEY_TYPE

L

LABEL, LANGUAGE, LARGE, LAST, LAST_VALUE, LATERAL, LEADING, LEFT, LENGTH, LEVEL, LIBRARY, LIKE, LIMIT, LN, LOCAL, LOCALTIME, LOCALTIMESTAMP, LOCATOR, LOWER

M

M, MAP, MATCH, MATCHED, MAX, MAXVALUE, MEMBER, MERGE, MESSAGE_LENGTH, MESSAGE_OCTET_LENGTH,
MESSAGE_TEXT, METHOD, MICROSECOND, MILLENNIUM, MIN, MINUTE, MINVALUE, MOD, MODIFIES, MODULE,
MONTH, MORE, MULTISET, MUMPS

N

NAME, NAMES, NATIONAL, NATURAL, NCHAR, NCLOB, NESTING, NEW, NEXT, NO, NONE, NORMALIZE,
NORMALIZED, NOT, NULL, NULLABLE, NULLIF, NULLS, NUMBER, NUMERIC

O

OBJECT, OCTETS, OCTET_LENGTH, OF, OFFSET, OLD, ON, ONLY, OPEN, OPTION, OPTIONS, OR, ORDER, ORDERING, ORDINALITY, OTHERS, OUT, OUTER, OUTPUT, OVER, OVERLAPS, OVERLAY, OVERRIDING

P

PAD, PARAMETER, PARAMETER_MODE, PARAMETER_NAME, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_NAME, PARAMETER_SPECIFIC_SCHEMA, PARTIAL,
PARTITION, PASCAL, PASSTHROUGH, PATH, PERCENTILE_CONT, PERCENTILE_DISC, PERCENT_RANK, PLACING,
PLAN, PLI, POSITION, POWER, PRECEDING, PRECISION, PREPARE, PRESERVE, PRIMARY, PRIOR, PRIVILEGES,
PROCEDURE, PUBLIC

Q

QUARTER

R

RANGE, RANK, READ, READS, REAL, RECURSIVE, REF, REFERENCES, REFERENCING, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY,
RELATIVE, RELEASE, REPEATABLE, RESET, RESTART, RESTRICT, RESULT, RETURN, RETURNED_CARDINALITY,
RETURNED_LENGTH, RETURNED_OCTET_LENGTH, RETURNED_SQLSTATE, RETURNS, REVOKE, RIGHT, ROLE, ROLLBACK,
ROLLUP, ROUTINE, ROUTINE_CATALOG, ROUTINE_NAME, ROUTINE_SCHEMA, ROW, ROWS, ROW_COUNT, ROW_NUMBER

S

SAVEPOINT, SCALE, SCHEMA, SCHEMA_NAME, SCOPE, SCOPE_CATALOGS, SCOPE_NAME, SCOPE_SCHEMA,
SCROLL, SEARCH, SECOND, SECTION, SECURITY, SELECT, SELF, SENSITIVE, SEQUENCE, SERIALIZABLE,
SERVER, SERVER_NAME, SESSION, SESSION_USER, SET, SETS, SIMILAR, SIMPLE, SIZE, SMALLINT,
SOME, SOURCE, SPACE, SPECIFIC, SPECIFICTYPE, SPECIFIC_NAME, SQL, SQLEXCEPTION, SQLSTATE,
SQLWARNING, SQL_TSI_DAY, SQL_TSI_FRAC_SECOND, SQL_TSI_HOUR, SQL_TSI_MICROSECOND, SQL_TSI_MINUTE,
SQL_TSI_MONTH, SQL_TSI_QUARTER, SQL_TSI_SECOND, SQL_TSI_WEEK, SQL_TSI_YEAR, SQRT, START,
STATE, STATEMENT, STATIC, STDDEV_POP, STDDEV_SAMP, STREAM, STRUCTURE, STYLE, SUBCLASS_ORIGIN,
SUBMULTISET, SUBSTITUTE, SUBSTRING, SUM, SYMMETRIC, SYSTEM, SYSTEM_USER

T

TABLE, TABLESAMPLE, TABLE_NAME, TEMPORARY, THEN, TIES, TIME, TIMESTAMP, TIMESTAMPADD,

TIMESTAMPDIFF, TIMEZONE_HOUR, TIMEZONE_MINUTE, TINYINT, TO, TOP_LEVEL_COUNT, TRAILING, TRANSACTION, TRANSACTIONS_ACTIVE, TRANSACTIONS_COMMITTED, TRANSACTIONS_ROLLED_BACK,
TRANSFORM, TRANSFORMS, TRANSLATE, TRANSLATION, TREAT, TRIGGER, TRIGGER_CATALOG, TRIGGER_NAME, TRIGGER_SCHEMA, TRIM, TRUE, TYPE

U

UESCAPE, UNBOUNDED, UNCOMMITTED, UNDER, UNION, UNIQUE, UNKNOWN, UNNAMED, UNNEST, UPDATE, UPPER, UPSERT, USAGE, USER, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_CODE, USER_DEFINED_TYPE_NAME, USER_DEFINED_TYPE_SCHEMA, USING

V

VALUE, VALUES, VARBINARY, VARCHAR, VARYING, VAR_POP, VAR_SAMP, VERSION, VIEW

W

WATERMARK, WEEK, WHEN, WHENEVER, WHERE, WIDTH_BUCKET, WINDOW, WITH, WITHIN, WITHOUT,
WORK, WRAPPER, WRITE

X

XML

Y

YEAR

Z

ZONE

上下游开发指南

消息队列Kafka

Kafka 数据管道是流计算系统中最常用的数据源（Source）和数据目的（Sink）。用户可以把流数据导入到 Kafka 的某个 Topic 中，通过 Flink 算子进行处理后，输出到相同或不同 Kafka 实例的另一个 Topic。
Kafka 支持同一个 Topic 多分区读写，数据可以从多个分区读入，也可以写入到多个分区，以提供更高的吞吐量，减少数据倾斜和热点。
版本说明

Flink 版本	说明
1.11	支持
1.13	支持
1.14	支持
1.16	支持

使用范围
Kafka 支持用作数据源表（Source），也可以作为 Tuple 数据流的目的表（Sink）。
Kafka 还可以与 Debezium、Canal 等联用，对 MySQL、PostgreSQL 等传统数据库的变更进行捕获和订阅，然后 Flink 即可对这些变更事件进行进一步的处理。
DDL 定义
用作数据源（Source）
JSON 格式输入

```
CREATE TABLE `kafka_json_source_table` (
  `id` INT,
  `name` STRING
) WITH (
  -- 定义 Kafka 参数
  'connector' = 'kafka',
  'topic' = 'Data-Input',  -- 替换为您要消费的 Topic
  'scan.startup.mode' = 'latest-offset', -- 可以是 latest-offset / earliest-offset / specific-offsets / group-of
fsets / timestamp 的任何一种
  'properties.bootstrap.servers' = '{IP:Port}',  -- 替换为您的 Kafka 连接地址
  'properties.group.id' = 'testGroup', -- 必选参数, 一定要指定 Group ID

  -- 定义数据格式 (JSON 格式)
```

```
  'format' = 'json',
  'json.fail-on-missing-field' = 'false',  -- 如果设置为 false, 则遇到缺失字段不会报错。
  'json.ignore-parse-errors' = 'true'    -- 如果设置为 true，则忽略任何解析报错。
);
```

CSV 格式输入

```
CREATE TABLE `kafka_csv_source_table` (
  `id` INT,
  `name` STRING
) WITH (
  -- 定义 Kafka 参数
  'connector' = 'kafka',
  'topic' = 'Data-Input',  -- 替换为您要消费的 Topic
  'scan.startup.mode' = 'latest-offset', -- 可以是 latest-offset / earliest-offset / specific-offsets / group-offsets / timestamp 的任何一种
  'properties.bootstrap.servers' = '{IP:Port}',  -- 替换为您的 Kafka 连接地址
  'properties.group.id' = 'testGroup', -- 必选参数, 一定要指定 Group ID

  -- 定义数据格式 (CSV 格式)
  'format' = 'csv'
);
```

Debezium 格式输入

```
CREATE TABLE `kafka_debezium_source_table` (
  `id` INT,
  `name` STRING
) WITH (
  -- 定义 Kafka 参数
  'connector' = 'kafka',
  'topic' = 'Data-Input',  -- 替换为您要消费的 Topic
  'scan.startup.mode' = 'latest-offset', -- 可以是 latest-offset / earliest-offset / specific-offsets / group-offsets / timestamp 的任何一种
  'properties.bootstrap.servers' = '{IP:Port}',  -- 替换为您的 Kafka 连接地址
  'properties.group.id' = 'testGroup', -- 必选参数, 一定要指定 Group ID

  -- 定义数据格式 (Debezium 输出的 JSON 格式)
  'format' = 'debezium-json'
);
```

Canal 格式输入

```
CREATE TABLE `kafka_source`
(
   aid             BIGINT COMMENT 'unique id',
```

```
    charname        string,
    `ts`            timestamp(6),
    origin_database STRING METADATA FROM 'value.database' VIRTUAL,
    origin_table STRING METADATA FROM 'value.table' VIRTUAL,
    origin_es TIMESTAMP(3) METADATA FROM 'value.event-timestamp' VIRTUAL,
    origin_type STRING METADATA FROM 'value.operation-type' VIRTUAL,
    `batch_id` bigint METADATA FROM 'value.batch-id' VIRTUAL,
    `is_ddl` boolean METADATA FROM 'value.is-ddl' VIRTUAL,
    origin_old ARRAY<MAP<STRING, STRING>> METADATA FROM 'value.update-before' VIRTUAL,
    `mysql_type` MAP<STRING, STRING> METADATA FROM 'value.mysql-type' VIRTUAL,
    origin_pk_names ARRAY<STRING> METADATA FROM 'value.pk-names' VIRTUAL,
    `sql` STRING METADATA FROM 'value.sql' VIRTUAL,
    origin_sql_type MAP<STRING, INT> METADATA FROM 'value.sql-type' VIRTUAL,
    `ingestion_ts` TIMESTAMP(3) METADATA FROM 'value.ingestion-timestamp' VIRTUAL
) WITH (
    'connector' = 'kafka', -- 注意选择对应的内置  Connector
    'topic' = '$TOPIC', -- 替换为您要消费的 Topic
    'properties.bootstrap.servers' = '$IP:$PORT', -- 替换为您的 Kafka 连接地址
    'properties.group.id' = 'testGroup', -- 必选参数, 一定要指定 Group ID
    'scan.startup.mode' = 'latest-offset',
    'scan.topic-partition-discovery.interval' = '5s',
    'format' = 'canal-json',
    'canal-json.ignore-parse-errors' = 'false', -- 忽略 JSON 结构解析异常
    'canal-json.source.append-mode' = 'true'  -- 仅支持Flink1.13及以上版本
);
```

用作数据目的（Sink）

JSON 格式输出

```
CREATE TABLE `kafka_json_sink_table` (
  `id` INT,
  `name` STRING
) WITH (
  -- 定义 Kafka 参数
  'connector' = 'kafka',
  'topic' = 'Data-Output',  -- 替换为您要写入的 Topic
  'properties.bootstrap.servers' = '{IP:Port}',  -- 替换为您的 Kafka 连接地址

  -- 定义数据格式 (JSON 格式)
  'format' = 'json',
  'json.fail-on-missing-field' = 'false',  -- 如果设置为 false, 则遇到缺失字段不会报错。
  'json.ignore-parse-errors' = 'true'    -- 如果设置为 true，则忽略任何解析报错。
);
```

CSV 格式输出

```
CREATE TABLE `kafka_csv_sink_table` (
  `id` INT,
  `name` STRING
) WITH (
  -- 定义 Kafka 参数
  'connector' = 'kafka',
  'topic' = 'Data-Output',  -- 替换为您要写入的 Topic
  'properties.bootstrap.servers' = '{IP:Port}',  -- 替换为您的 Kafka 连接地址

  -- 定义数据格式 (CSV 格式)
  'format' = 'csv'
);
```

Canal 格式输出

```
CREATE TABLE `kafka_canal_json_sink_table`
(
    aid             BIGINT COMMENT 'unique id',
    charname        string,
    `ts`            timestamp(6),
    origin_database STRING METADATA FROM 'value.database',
    origin_table STRING METADATA FROM 'value.table',
    origin_ts TIMESTAMP(3) METADATA FROM 'value.event-timestamp',
    `type` STRING METADATA FROM 'value.operation-type',
    `batch_id` bigint METADATA FROM 'value.batch-id',
    `isDdl` BOOLEAN METADATA FROM 'value.is-ddl',
    `old` ARRAY<MAP<STRING, STRING>> METADATA FROM 'value.update-before',
    `mysql_type` MAP<STRING, STRING> METADATA FROM 'value.mysql-type',
    `pk_names` ARRAY<STRING> METADATA FROM 'value.pk-names',
    `sql` STRING METADATA FROM 'value.sql',
    `sql_type` MAP<STRING, INT> METADATA FROM 'value.sql-type',
    `ingestion_ts` TIMESTAMP(3) METADATA FROM 'value.ingestion-timestamp'
) WITH (
    'connector' = 'kafka', -- 注意选择对应的内置  Connector
    'topic' = '$TOPIC', -- 替换为您要消费的 Topic
    'properties.bootstrap.servers' = '$IP:$PORT', -- 替换为您的 Kafka 连接地址
    'properties.group.id' = 'testGroup', -- 必选参数, 一定要指定 Group ID
    'format' = 'canal-json'
);
```

WITH 参数

参数	必填	默认值	描述
connector	是	无	固定值为 'kafka'。

参数	必填	默认值	描述
topic	是	无	要读写的 Kafka Topic 名。
properties.bootstrap.servers	是	无	逗号分隔的 Kafka Bootstrap 地址。
properties.group.id	作为数据源时必选	无	Kafka 消费时的 Group ID。
format	是	无	Kafka 消息的输入输出格式。目前支持 csv、json、avro、debezium-json、canal-json，Flink1.13支持 Maxwell-JSON。
scan.startup.mode	否	group-offsets	Kafka consumer 的启动模式。可以是 latest-offset、earliest-offset、specific-offsets、group-offsets、timestamp 的任何一种。 'scan.startup.specific-offsets' = 'partition:0,offset:42;partition:1,offset:300'，使用 'specific-offsets' 启动模式时需要指定每个 partition 对应的 offsets。 'scan.startup.timestamp-millis' = '1631588815000'，使用 'timestamp' 启动模式时需要指定启动的时间戳（单位毫秒）。
scan.startup.specific-offsets	否	无	如果 scan.startup.mode 的值为'specific-offsets'，则必须使用本参数指定具体起始读取的偏移量。例如 'partition:0,offset:42;partition:1,offset:300'。
scan.startup.timestamp-millis	否	无	如果scan.startup.mode 的值为'timestamp'，则必须使用本参数来指定开始读取的时间点（毫秒为单位的 Unix 时间戳）。
sink.partitioner	否	无	Kafka 输出时所用的分区器。目前支持的分区器如下： fixed：一个 Flink 分区对应不多于一个 Kafka 分区。 round-robin：一个Flink 分区依次被分配到不同的 Kafka 分区。 自定义分区：也可以通过继承 FlinkKafkaPartitioner 类，实现该逻辑。

JSON 格式 WITH 参数

参数	必填	默认值	描述
json.fail-on-missing-field	否	false	如果为 true，则遇到缺失字段时，会让作业失败。如果为 false（默认值），则只会把缺失字段设置为 null 并继续处理。

参数	必填	默认值	描述
json.ignore-parse-errors	否	false	如果为 true，则遇到解析异常时，会把这个字段设置为 null 并继续处理。如果为 false，则会让作业失败。
json.timestamp-format.standard	否	SQL	指定 JSON 时间戳字段的格式，默认是 SQL（格式是yyyy-MM-dd HH:mm:ss.s{可选精度}）。也可以选择 ISO-8601，格式是yyyy-MM-ddTHH:mm:ss.s{可选精度}。

CSV 格式 WITH 参数

参数	必填	默认值	描述
csv.field-delimiter	否	,	指定 CSV 字段分隔符，默认是半角逗号。
csv.line-delimiter	否	U&'\000A'	指定 CSV 的行分隔符，默认是换行符\n，SQL 中必须用U&'\000A'表示。如果需要使用回车符\r，SQL 中必须使用U&'\000D'表示。
csv.disable-quote-character	否	false	禁止字段包围引号。如果为 true，则 'csv.quote-character' 选项不可用。
csv.quote-character	否	"	字段包围引号，引号内部的作为整体看待。默认是"。
csv.ignore-parse-errors	否	false	忽略处理错误。对于无法解析的字段，会输出为 null。
csv.allow-comments	否	false	忽略 # 开头的注释行，并输出为空行（请务必将 csv.ignore-parse-errors 设为 true）。
csv.array-element-delimiter	否	;	数组元素的分隔符，默认是;。
csv.escape-character	否	无	指定转义符，默认禁用转义。
csv.null-literal	否	无	将指定的字符串看作 null 值。

Debezium-json 格式 WITH 参数

参数	必填	默认值	描述
debezium-json.schema-include	否	false	设置 Debezium Kafka Connect 时，如果指定了'value.converter.schemas.enable'参数，那么 Debezium 发来的 JSON 数据里会包含 Schema 信息，该选项需要设置为 true。

参数	必填	默认值	描述
debezium-json.ignore-parse-errors	否	false	忽略处理错误。对于无法解析的字段，会输出为 null。
debezium-json.timestamp-format.standard	否	SQL	指定 JSON 时间戳字段的格式，默认是 SQL（格式是 yyyy-MM-dd HH:mm:ss.s{可选精度}）。也可以选择 ISO-8601，格式是 yyyy-MM-ddTHH:mm:ss.s{可选精度}。

Canal 格式 WITH 参数

参数	必填	默认值	描述
canal-json.source.append-mode	否	false	设置为 true 时支持 append 流，例如，消费 kafka canal-json 数据到 hive，该参数仅支持 Flink1.13 集群
debezium-json.ignore-parse-errors	否	false	忽略处理错误。对于无法解析的字段，会输出为 null。
canal-json.*	否	-	参考 Format Options

Canal 格式支持的元数据（仅支持 Flink1.13 版本集群）

以下元数据只能作为表定义中的只读（VIRTUAL）列，若元数据列与物理列冲突，元数据列可以使用meta.列名：

列	数据类型	描述
database	STRING NOT NULL	包含该 Row 的数据库名称
table	STRING NOT NULL	包含该 Row 的表名称
event-timestamp	TIMESTAMP_LTZ(3) NOT NULL	Row 在数据库中进行更改的时间
batch-id	BIGINT	Binlog 的批 ID
is-ddl	BOOLEAN	是否 DDL 语句
mysql-type	MAP	数据表结构
update-before	ARRAY	未修改前字段的值
pk-names	ARRAY	主键字段名
sql	STRING	暂时为空
sql-type	MAP	sql_type 表的字段到 java 数据类型 ID 的映射
ingestion-timestamp	TIMESTAMP_LTZ(3) NOT NULL	收到该 ROW 并处理的当前时间

列	数据类型	描述
operation-type	STRING	数据库操作类型，例如 INSERT/DELETE 等

代码示例

Json 格式使用示例

```
CREATE TABLE `kafka_json_source_table` (
  `id` INT,
  `name` STRING
) WITH (
  -- 定义 Kafka 参数
  'connector' = 'kafka',
  'topic' = 'Data-Input',  -- 替换为您要消费的 Topic
  'scan.startup.mode' = 'latest-offset', -- 可以是 latest-offset / earliest-offset / specific-offsets / group-offsets / timestamp 的任何一种
  'properties.bootstrap.servers' = '{IP:Port}',  -- 替换为您的 Kafka 连接地址
  'properties.group.id' = 'testGroup', -- 必选参数, 一定要指定 Group ID

  -- 定义数据格式 (JSON 格式)
  'format' = 'json',
  'json.fail-on-missing-field' = 'false',  -- 如果设置为 false, 则遇到缺失字段不会报错。
  'json.ignore-parse-errors' = 'true'    -- 如果设置为 true，则忽略任何解析报错。
);
CREATE TABLE `kafka_json_sink_table` (
  `id` INT,
  `name` STRING
) WITH (
  -- 定义 Kafka 参数
  'connector' = 'kafka',
  'topic' = 'Data-Output',  -- 替换为您要写入的 Topic
  'properties.bootstrap.servers' = '{IP:Port}',  -- 替换为您的 Kafka 连接地址

  -- 定义数据格式 (JSON 格式)
  'format' = 'json',
  'json.fail-on-missing-field' = 'false',  -- 如果设置为 false, 则遇到缺失字段不会报错。
  'json.ignore-parse-errors' = 'true'    -- 如果设置为 true，则忽略任何解析报错。
);
insert into kafka_json_sink_table select * from kafka_json_source_table;
```

复杂嵌套 Json 格式使用示例

json 示例

```
{
  "id": 1234567890,
  "name": "tom",
  "date": "2000-10-25",
  "obj": {
    "time1": "11:11:11",
    "str": "test",
    "lg": 1122334455
  },
  "arr": [
    "aa",
    "bb",
    "cc",
    "dd"
  ],
  "rowinarr": [
    {
      "f1": "f11",
      "f2": 111
    },
    {
      "f1": "f12",
      "f2": 222
    }
  ],
  "time": "19:19:19",
  "timestamp": "1999-01-12 14:14:14",
  "map": {
    "flink": 123
  },
  "mapinmap": {
    "inner_map": {
      "key": 234
    }
  }
}
```

Flink SQL

```
create table kafka_source(
  id            BIGINT,
  name          STRING,
  `date`        DATE,
  obj           ROW<time1 TIME,str STRING,lg BIGINT>,
  arr           ARRAY<STRING>,
  rowinarr      ARRAY<ROW<f1 STRING,f2 INT>>,
  `time`        TIME,
```

```
  `timestamp`   TIMESTAMP(3),
  `map`         MAP<STRING,BIGINT>,
  mapinmap      MAP<STRING,MAP<STRING,INT>>
) with (
  'connector' = 'kafka',
  'topic' = 'test-topic',
  'scan.startup.mode' = 'latest-offset',
  'properties.bootstrap.servers' = '{IP:Port}',
  'properties.group.id' = 'testGroup',
  'format' = 'json'
);

create table logger_sink (
  id            BIGINT,
  name          STRING,
  `date`        DATE,
  str           STRING,
  arr           ARRAY<STRING>,
  nameinarray   STRING,
  rowinarr      ARRAY<ROW<f1 STRING,f2 INT>>,
  f2            INT,
  `time`        TIME,
  `timestamp`   TIMESTAMP(3),
  `map`         MAP<STRING,BIGINT>,
  flink         BIGINT,
  mapinmap      MAP<STRING,MAP<STRING,INT>>,
  `key`         INT
) with (
    'connector' = 'logger'
);

insert into
  logger_sink
select
  id,
  name,
  `date`,
  obj.str,
  arr,
  arr[4],
  rowinarr,
  rowinarr[1].f2,
  `time`,
  `timestamp`,
  `map`,
  `map`['flink'],
  mapinmap,
```

```
  mapinmap['inner_map']['key']
from kafka_source;
```

输出结果

```
+I(1234567890,tom,2000-10-25,test,[aa, bb, cc, dd],dd,[f11,111, f12,222],111,13:13:13,1999-01-12T14:14:14,{flink=123},123,{inner_map={key=234}},234)
```

> 注意：
>
> 1. 各数据类型获取元素的方法：
>
> map：map['key']
>
> array：array[index]
>
> row：row.key
>
> 1. array 的起始下标从 1 开始。

Json 数据类型映射

Flink SQL type	JSON type
CHAR / VARCHAR / STRING	string
BOOLEAN	boolean
BINARY / VARBINARY	string with encoding: base64
DECIMAL	number
TINYINT	number
SMALLINT	number
INT	number
BIGINT	number
FLOAT	number
DOUBLE	number
DATE	string with format: date
TIME	string with format: time
TIMESTAMP	string with format: date-time

TIMESTAMP_WITH_LOCAL_TIME_ZONE	string with format: date-time (with UTC time zone)
INTERVAL	number
ARRAY	array
MAP / MULTISET	object
ROW	object

Canal 使用示例

```
CREATE TABLE `source`
(
    `aid`           bigint,
    `charname`      string,
    `ts`            timestamp(6),
    `database_name` string METADATA FROM 'value.database_name',
    `table_name`    string METADATA FROM 'value.table_name',
    `op_ts`         timestamp(3) METADATA FROM 'value.op_ts',
    `op_type` string METADATA FROM 'value.op_type',
    `batch_id` bigint METADATA FROM 'value.batch_id',
    `is_ddl` boolean METADATA FROM 'value.is_ddl',
    `update_before` ARRAY<MAP<STRING, STRING>> METADATA FROM 'value.update_before',
    `mysql_type` MAP<STRING, STRING> METADATA FROM 'value.mysql_type',
    `pk_names` ARRAY<STRING> METADATA FROM 'value.pk_names',
    `sql` STRING METADATA FROM 'value.sql',
    `sql_type` MAP<STRING, INT> METADATA FROM 'value.sql_type',
    `ingestion_ts` TIMESTAMP(3) METADATA FROM 'value.ts',
    primary key (`aid`) not enforced
) WITH (
    'connector' = 'mysql-cdc' ,
     'append-mode' = 'true',
    'hostname' = '$IP',
    'port' = '$PORT',
    'username' = '$USERNAME',
    'password' = '$PASSWORD',
    'database-name' = 't_wr',
    'table-name' = 't1',
    'server-time-zone' = 'Asia/city',
    'server-id' = '5500-5510'
);

CREATE TABLE `kafka_canal_json_sink`
(
    aid             BIGINT COMMENT 'unique id',
```

```
    charname            string,
    `ts`            timestamp(6),
    origin_database STRING METADATA FROM 'value.database',
    origin_table STRING METADATA FROM 'value.table',
    origin_ts TIMESTAMP(3) METADATA FROM 'value.event-timestamp',
    `type` STRING METADATA FROM 'value.operation-type',
    `batch_id` bigint METADATA FROM 'value.batch-id',
    `isDdl` BOOLEAN METADATA FROM 'value.is-ddl',
    `old` ARRAY<MAP<STRING, STRING>> METADATA FROM 'value.update-before',
    `mysql_type` MAP<STRING, STRING> METADATA FROM 'value.mysql-type',
    `pk_names` ARRAY<STRING> METADATA FROM 'value.pk-names',
    `sql` STRING METADATA FROM 'value.sql',
    `sql_type` MAP<STRING, INT> METADATA FROM 'value.sql-type',
    `ingestion_ts` TIMESTAMP(3) METADATA FROM 'value.ingestion-timestamp'
)
    WITH (
        'connector' = 'kafka',
        'topic' = 'TOPIC', -- 替换为您要消费的 Topic
        'properties.bootstrap.servers' = '$IP:$PORT', -- 替换为您的 Kafka 连接地址
        'format' = 'canal-json'
);

insert into kafka_canal_json_sink select * from source;

 CREATE TABLE `source`
(
    `aid`           bigint,
    `charname`      string,
    `ts`            timestamp(3),
    origin_database STRING METADATA FROM 'value.database' VIRTUAL,
    origin_table STRING METADATA FROM 'value.table' VIRTUAL,
    origin_es TIMESTAMP(3) METADATA FROM 'value.event-timestamp' VIRTUAL,
    origin_type STRING METADATA FROM 'value.operation-type' VIRTUAL,
    `batch_id` bigint METADATA FROM 'value.batch-id' VIRTUAL,
    `is_ddl` boolean METADATA FROM 'value.is-ddl' VIRTUAL,
    origin_old ARRAY<MAP<STRING, STRING>> METADATA FROM 'value.update-before' VIRTUAL,
    `mysql_type` MAP<STRING, STRING> METADATA FROM 'value.mysql-type' VIRTUAL,
    origin_pk_names ARRAY<STRING> METADATA FROM 'value.pk-names' VIRTUAL,
    `sql` STRING METADATA FROM 'value.sql' VIRTUAL,
    origin_sql_type MAP<STRING, INT> METADATA FROM 'value.sql-type' VIRTUAL,
    `ingestion_ts` TIMESTAMP(3) METADATA FROM 'value.ingestion-timestamp' VIRTUAL,
    WATERMARK FOR `origin_es` AS `origin_es` - INTERVAL '5' SECOND
) WITH (
      'connector' = 'kafka', -- 注意选择对应的内置  Connector
      'topic' = '$TOPIC', -- 替换为您要消费的 Topic
      'properties.bootstrap.servers' = '$IP:PORT', -- 替换为您的 Kafka 连接地址
      'properties.group.id' = 'testGroup', -- 必选参数, 一定要指定 Group ID
```

```
    'scan.startup.mode' = 'latest-offset',
    'scan.topic-partition-discovery.interval' = '10s',

    'format' = 'canal-json',
    'canal-json.source.append-mode' = 'true', -- 仅支持Flink1.13
    'canal-json.ignore-parse-errors' = 'false'
);


CREATE TABLE `kafka_canal_json` (
  `aid`          bigint,
  `charname`     string,
  `ts`           timestamp(9),
  origin_database STRING,
  origin_table STRING,
  origin_es TIMESTAMP(9),
  origin_type STRING,
  `batch_id` bigint,
  `is_ddl` boolean,
  origin_old ARRAY<MAP<STRING, STRING>>,
  `mysql_type` MAP<STRING, STRING>,
  origin_pk_names ARRAY<STRING>,
  `sql` STRING,
  origin_sql_type MAP<STRING, INT>,
  `ingestion_ts` TIMESTAMP(9),
  dt STRING,
  hr STRING
) PARTITIONED BY (dt, hr)
with (
    'connector' = 'hive',
    'hive-version' = '3.1.1',
    'hive-database' = 'testdb',
    'partition.time-extractor.timestamp-pattern'='$dt $hr:00:00',
    'sink.partition-commit.trigger'='partition-time',
    'sink.partition-commit.delay'='30 min',
    'sink.partition-commit.policy.kind'='metastore,success-file'
);

insert into kafka_canal_json select *,DATE_FORMAT(`origin_es`,'yyyy-MM-dd'),DATE_FORMAT(`origin_es
`,'HH')
from `source`;
```

SASL 认证授权

SASL/PLAIN 用户名密码认证授权

1. 设置 Topic 按用户名密码访问的 SASL_PLAINTEXT 认证方式。
2. 选择 SASL_PLAINTEXT 接入方式，并以该接入方式下的网络地址访问 Topic。

3. 作业配置 with 参数。

```
CREATE TABLE `YourTable` (
...
) WITH (
  ...
  'properties.sasl.jaas.config' = 'org.apache.kafka.common.security.plain.PlainLoginModule required use
rname="ckafka-xxxxxxxx#YourUserName" password="YourPassword";',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'PLAIN',
  ...
);
```

> 说明：
>
> 1. username 是实例 ID + # + 刚配置的用户名，password 是刚配置的用户密码 。
> 2. flink 版本 sasl 认证配置 对应包：
>
> 1.16版本以下：org.apache.kafka.common.security.plain.PlainLoginModule
>
> 1.16版本及以上：'org.apache.flink.kafka.shaded.org.apache.kafka.common.security.plain.PlainLoginModule'

SASL/GSSAPI Kerberos 认证授权

CKafka 暂时不支持 Kerberos 认证，您的自建 Kafka 如果开启了 Kerberos 认证，可参考如下步骤配置作业。

1. 获取您的自建 Kafka 集群的 Kerberos 配置文件，如果您基于TBDS 集群自建，获取 krb5.conf、emr.keytab 文件，路径如下。

```
/etc/krb5.conf
/var/krb5kdc/emr.keytab
```

2. 对步骤1中获取的文件打 jar 包。

```
jar cvf kafka-xxx.jar krb5.conf emr.keytab
```

3. 校验 jar 的结构（可以通过 vim 命令查看 vim kafka-xxx.jar），jar 里面包含如下信息，请确保文件不缺失且结构正确。

```
META-INF/
```

META-INF/MANIFEST.MF
emr.keytab
krb5.conf

4. 上传 jar 包，并在作业参数配置里引用该程序包。
5. 获取 kerberos principal。

```
klist -kt /var/krb5kdc/emr.keytab
# 输出如下所示，选取第一个即可：hadoop/{IP}@TBDS-XXXX
KVNO Timestamp     Principal
---- ------------------- ------------------------------------------------------
  2 08/09/2021 15:34:40 hadoop/{IP}@TBDS-XXXX
  2 08/09/2021 15:34:40 HTTP/{IP}@TBDS-XXXX
  2 08/09/2021 15:34:40 hadoop/VM-28-51-centos@TBDS-XXXX
  2 08/09/2021 15:34:40 HTTP/VM-28-51-centos@TBDS-XXXX
```

6. 作业 with 参数配置。

```
CREATE TABLE `YourTable` (
...
) WITH (
  ...
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'hadoop',
  ...
);
```

7. 作业 配置。

```
security.kerberos.login.principal: hadoop/{IP}@TBDS-XXXX
security.kerberos.login.keytab: emr.keytab
security.kerberos.login.conf: krb5.conf
security.kerberos.login.contexts: KafkaClient
fs.hdfs.hadoop.security.authentication: kerberos
```

消息队列Upsert Kafka

Upsert Kafka 连接器支持以 upsert 方式从 Kafka topic 中读取数据并将数据写入 Kafka topic。

作为 Source，Upsert Kafka 连接器生产 changelog 流，其中每条数据记录代表一个更新或删除事件。更准确地说，数据记录中的 value 被解释为同一 key 的最后一个 value 的 UPDATE，如果有这个 key（如果不存在相应的 key，则该更新被视为 INSERT）。用表来类比，changelog 流中的数据记录被解释为 UPSERT，也称为 INSERT/UPDATE，因为任何具有相同 key 的现有行都被覆盖。另外，value 为空的消息将会被视作为 DELETE 消息。
作为 Sink，Upsert Kafka 连接器可以消费 changelog 流。它会将 INSERT/UPDATE_AFTER 数据作为正常的 Kafka 消息写入，并将 DELETE 数据以 value 为空的 Kafka 消息写入（表示对应 key 的消息被删除）。Flink 将根据主键列的值对数据进行分区，从而保证主键上的消息有序，因此同一主键上的更新/删除消息将落在同一分区中。
版本说明

Flink 版本	说明
1.11	不支持
1.13	支持
1.14	支持
1.16	支持

DDL 定义

```
CREATE TABLE kafka_upsert_sink_table (
  id INT,
  name STRING,
  PRIMARY KEY (id) NOT ENFORCED
) WITH (
  -- 定义 Upsert Kafka 参数
  'connector' = 'upsert-kafka',  -- 选择 connector
  'topic' = 'topic',  -- 替换为您要写入的 Topic
  'properties.bootstrap.servers' = '...',  -- 替换为您的 Kafka 连接地址
  'key.format' = 'json',  -- 定义 key 数据格式
  'value.format' = 'json'  -- 定义value 数据格式
);
```

> 说明：
>
> Upsert Kafka 确保在 DDL 中定义主键。

WITH 参数

参数	是否必选	默认值	数据类型	描述
connector	必选	(none)	String	指定要使用的连接器，Upsert Kafka 连接器使用：'upsert-kafka'。

参数	是否必选	默认值	数据类型	描述
topic	必选	(none)	String	用于读取和写入的 Kafka topic 名称。
properties.bootstrap.servers	必选	(none)	String	以逗号分隔的 Kafka brokers 列表。
properties.*	可选	(none)	String	该选项可以传递任意的 Kafka 参数。选项的后缀名必须匹配定义在 Kafka 参数文档 中的参数名。Flink 会自动移除选项名中的 "properties." 前缀，并将转换后的键名以及值传入 KafkaClient。例如，您可以通过 'properties.allow.auto.create.topics' = 'false来禁止自动创建 topic。 但是，某些选项，例如 'key.deserializer' 和 'value.deserializer' 是不允许通过该方式传递参数，因为 Flink 会重写这些参数的值。
key.format	必选	(none)	String	用于对 Kafka 消息中 key 部分序列化和反序列化的格式。key 字段由 PRIMARY KEY 语法指定。支持的格式包括 'csv'、'json'、'avro'。
key.fields-prefix	optional	(none)	String	为'key.fields'的所有字段定义自定义前缀，以避免与 'value.fields' 字段名称冲突。默认情况下，前缀为空。如果定义了自定义前缀，则表 schema 和 'key.fields' 将使用前缀名称。构建'key.fields'格式的数据类型时，将删除前缀并使用 key format 中非前缀名称。请注意，此选项要求 'value.fields-include'必须设置为 'EXCEPT_KEY'。
value.format	必选	(none)	String	用于对 Kafka 消息中 value 部分序列化和反序列化的格式。支持的格式包括 'csv'、'json'、'avro'。

参数	是否必选	默认值	数据类型	描述
value.fields-include	可选	'ALL'	String	控制哪些字段应该出现在 value 中。可取值： ALL：消息的 value 部分将包含 schema 中所有的字段，包括定义为主键的字段。 EXCEPT_KEY：记录的 value 部分包含 schema 的所有字段，定义为主键的字段除外。
sink.parallelism	可选	(none)	Integer	定义 upsert-kafka sink 算子的并行度。默认情况下，由框架确定并行度，与上游连接算子的并行度保持一致。
sink.buffer-flush.max-rows	可选	0	Integer	缓存刷新前，最多能缓存多少条记录。当 sink 收到很多同 key 上的更新时，缓存将保留同 key 的最后一条记录，因此 sink 缓存能帮助减少发往 Kafka topic 的数据量，以及避免发送潜在的 tombstone 消息。 可以通过设置为 '0' 来禁用它。默认，该选项是未开启的。注意，如果要开启 sink 缓存，需要同时设置 'sink.buffer-flush.max-rows' 和 'sink.buffer-flush.interval' 两个选项为大于零的值。
sink.buffer-flush.interval	可选	0	Duration	缓存刷新的间隔时间，超过该时间后异步线程将刷新缓存数据。当 sink 收到很多同 key 上的更新时，缓存将保留同 key 的最后一条记录，因此 sink 缓存能帮助减少发往 Kafka topic 的数据量，以及避免发送潜在的 tombstone 消息。 可以通过设置为 '0' 来禁用它。默认，该选项是未开启的。注意，如果要开启 sink 缓存，需要同时设置 'sink.buffer-flush.max-rows'和 'sink.buffer-flush.interval' 两个选项为大于零的值。

代码示例

```
CREATE TABLE `kafka_json_source_table` (
  `id` INT,
```

```
  `name` STRING
) WITH (
  -- 定义 Kafka 参数
  'connector' = 'kafka',
  'topic' = 'Data-Input',  -- 替换为您要消费的 Topic
  'scan.startup.mode' = 'latest-offset', -- 可以是 latest-offset / earliest-offset / specific-offsets / group-of
fsets / timestamp 的任何一种
  'properties.bootstrap.servers' = '{IP:Port}',  -- 替换为您的 Kafka 连接地址
  'properties.group.id' = 'testGroup', -- 必选参数, 一定要指定 Group ID

  -- 定义数据格式 (JSON 格式)
  'format' = 'json',
  'json.fail-on-missing-field' = 'false',  -- 如果设置为 false, 则遇到缺失字段不会报错。
  'json.ignore-parse-errors' = 'true'    -- 如果设置为 true，则忽略任何解析报错。
);

CREATE TABLE kafka_upsert_sink_table (
  id INT,
  name STRING,
  PRIMARY KEY (id) NOT ENFORCED
) WITH (
  -- 定义 Upsert Kafka 参数
  'connector' = 'upsert-kafka',  -- 选择 connector
  'topic' = 'topic',  -- 替换为您要消费的 Topic
  'properties.bootstrap.servers' = '...',  -- 替换为您的 Kafka 连接地址
  'key.format' = 'json',  -- 定义 key 数据格式
  'value.format' = 'json'  -- 定义value 数据格式
);

-- 计算 pv、uv 并插入到 upsert-kafka sink
INSERT INTO kafka_upsert_sink_table
SELECT * FROM kafka_json_source_table;
```

SASL 认证授权

SASL/PLAIN 用户名密码认证授权

1. 设置 Topic 按用户名密码访问的 SASL_PLAINTEXT 认证方式。
2. 选择 SASL_PLAINTEXT 接入方式，并以该接入方式下的网络地址访问 Topic。
3. 作业配置 with 参数。

```
CREATE TABLE `YourTable` (
...
) WITH (
  ...
  'properties.sasl.jaas.config' = 'org.apache.kafka.common.security.plain.PlainLoginModule required use
```

```
rname="ckafka-xxxxxxxx#YourUserName" password="YourPassword";',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'PLAIN',
  ...
);
```

> 说明：
>
> username 是实例 ID + # + 刚配置的用户名，password 是刚配置的用户密码。

SASL/GSSAPI Kerberos 认证授权

CKafka 暂时不支持 Kerberos 认证，您的自建 Kafka 如果开启了 Kerberos 认证，可参考如下步骤配置作业。

1. 获取您的自建 Kafka 集群的 Kerberos 配置文件，如果您基于TBDS 集群自建，获取 krb5.conf、emr.keytab 文件，路径如下。

```
/etc/krb5.conf
/var/krb5kdc/emr.keytab
```

2. 对步骤1中获取的文件打 jar 包。

```
jar cvf kafka-xxx.jar krb5.conf emr.keytab
```

3. 校验 jar 的结构（可以通过 vim 命令查看 vim kafka-xxx.jar），jar 里面包含如下信息，请确保文件不缺失且结构正确。

```
META-INF/
META-INF/MANIFEST.MF
emr.keytab
krb5.conf
```

4. 上传 jar 包，并在作业参数配置里引用该程序包。
5. 获取 kerberos principal。

```
klist -kt /var/krb5kdc/emr.keytab
# 输出如下所示，选取第一个即可：hadoop/{IP}@TBDS-XXXX
KVNO Timestamp     Principal
---- ------------------- ------------------------------------------------------
  2 08/09/2021 15:34:40 hadoop/{IP}@TBDS-XXXX
```

2 08/09/2021 15:34:40 HTTP/{IP}@TBDS-XXXX
2 08/09/2021 15:34:40 hadoop/VM-28-51-centos@TBDS-XXXX
2 08/09/2021 15:34:40 HTTP/VM-28-51-centos@TBDS-XXXX

6. 作业 with 参数配置。

```
CREATE TABLE `YourTable` (
...
) WITH (
  ...
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.sasl.mechanism' = 'GSSAPI',
  'properties.sasl.kerberos.service.name' = 'hadoop',
  ...
);
```

> 说明：
>
> 参数 properties.sasl.kerberos.service.name 的值必须与您选取的 principal 匹配，如果您选择的为 hadoop/${IP}@TBDS-XXXX，那么取值为 hadoop。

7. 作业 高级参数 配置。

```
security.kerberos.login.principal: hadoop/{IP}@TBDS-XXXX
security.kerberos.login.keytab: emr.keytab
security.kerberos.login.conf: krb5.conf
security.kerberos.login.contexts: KafkaClient
fs.hdfs.hadoop.security.authentication: kerberos
```

数据库HBase

HBase Connector 提供了对 HBase 集群的读写支持。TBDS Flink 已经提供了内置的flink-connector-hbase Connector 组件。

版本说明

Flink 版本	说明
1.11	支持 hbase 版本为：1.4.x
1.13	支持 hbase 版本为：1.4.x、2.2.x、2.3.x

Flink 版本	说明
1.14	支持 hbase 版本为：1.4.x、2.2.x
1.16	支持 hbase 版本为: 1.4.x、2.2.x

适用范围

可以作为源表，维表，以及Tuple、Upsert 数据流的目的表。

DDL 定义

```
CREATE TABLE hbase_table (
  rowkey INT,
  cf ROW < school_name STRING >,
  PRIMARY KEY (rowkey) NOT ENFORCED
) WITH (
  'connector' = 'hbase-1.4',                    -- Flink 1.13 支持 hbase-2.2
  'table-name' = 'hbase_sink_table',                    -- Hbase 表名
  'zookeeper.quorum' = 'ip:port,ip:port,ip:port'   -- Hbase 的 zookeeper 地址
);
```

WITH 参数

参数	说明	是否必填	备注
connector	表类型	是	hbase-1.4 或者 hbase-2.2 如果您用了 hbase 2.3.x 版本，那么，connector 参数值需要替换为 hbase-2.2
table-name	HBase 表名	是	-
zookeeper.quorum	HBase 的 zookeeper 地址	是	查看 hbase-site.xml 确定参数值
zookeeper.znode.parent	HBase 在 zookeeper 中的根目录	否	查看 hbase-site.xml 确定参数值
null-string-literal	HBase 字段类型为字符串时，如果 Flink 字段数据为 null，则将该字段赋值为 null-string-literal，并写入 HBase	否	默认为 null
sink.buffer-flush.max-size	写入 HBase 前，内存中缓存的数据量（字节）大小。调大该值有利于提高 HBase 写入性能，但会增加写入	否	默认值为2MB，支持字节单位 B、KB、MB 和 GB，不区分大小写。设置为0表

参数	说明	是否必填	备注
	延迟和内存使用。仅作为 Sink 时使用		示不进行缓存
sink.buffer-flush.max-rows	写入 HBase 前，内存中缓存的数据条数。调大该值有利于提高 HBase 写入性能，但会增加写入延迟和内存使用。仅作为 Sink 时使用	否	默认值为1000，设置为0表示不进行缓存
sink.buffer-flush.interval	将缓存数据周期性写入到 HBase 的间隔，可以控制写入 HBase 的延迟。仅作为 Sink 时使用。	否	默认值为1秒，支持时间单位 ms、s、min、h 和 d。设置为0表示关闭定期写入

类型映射

HBase 将所有的数据存为字节数组。读写操作时需要将数据进行序列化和反序列化。Flink 与 HBase 的数据转换关系如下：

Flink 字段类型	HBase 转换
CHAR / VARCHAR / STRING	byte[] toBytes(String s) String toString(byte[] b)
BOOLEAN	byte[] toBytes(boolean b)boolean toBoolean(byte[] b)
BINARY / VARBINARY	byte[]
DECIMAL	byte[] toBytes(BigDecimal v)BigDecimal toBigDecimal(byte[] b)
TINYINT	new byte[] { val } bytes[0]
SMALLINT	byte[] toBytes(short val)short toShort(byte[] bytes)
INT	byte[] toBytes(int val)int toInt(byte[] bytes)
BIGINT	byte[] toBytes(long val)long toLong(byte[] bytes)
FLOAT	byte[] toBytes(float val)float toFloat(byte[] bytes)
DOUBLE	byte[] toBytes(double val)double toDouble(byte[] bytes)
DATE	将日期转换成自1970.01.01以来的天数，用 int 表示，并通过 byte[] toBytes(int val) 转换成字节数组
TIME	将时间转换成自00:00:00以来的毫秒数，用 int 表示，并通过 byte[] toBytes(int val) 转换成字节数组

Flink 字段类型	HBase 转换
TIMESTAMP	将时间戳转换成自1970-01-01 00:00:00以来的毫秒数，用 long 表示，并通过 byte[] toBytes(long val) 转换成字节数组
ARRAY	不支持
MAP / MULTISET	不支持
ROW	不支持

代码示例

包含 HBase 维表的实时计算作业代码，示例如下：

```
CREATE TABLE datagen_source_table (
  id INT,
  name STRING,
  `proc_time` AS PROCTIME()
) with (
  'connector'='datagen',
  'rows-per-second'='1'
);

CREATE TABLE hbase_table (
  rowkey INT,
  cf ROW < school_name STRING >,
  PRIMARY KEY (rowkey) NOT ENFORCED
) WITH (
  'connector' = 'hbase-1.4',                    -- Flink 1.13 支持 hbase-2.2
  'table-name' = 'hbase_sink_table',                    -- Hbase 表名
  'zookeeper.quorum' = 'ip:port,ip:port,ip:port'   -- Hbase 的 zookeeper 地址
);

CREATE TABLE blackhole_sink(
  id INT,
  name STRING
) with (
  'connector' = 'blackhole'
);

INSERT INTO blackhole_sink
    SELECT id, cf.school_name as name FROM datagen_source_table src
JOIN hbase_table FOR SYSTEM_TIME AS OF src.`proc_time` as h ON src.id = h.rowkey;
```

> 注意：

> HBase Connector 一般会使用 DDL 语句中定义的主键，以 upsert 模式工作，与外部系统交换变更日志信息。因此，必须在 HBase 的 rowkey 字段上定义主键（必须声明 rowkey 字段）。如果未声明 PRIMARY KEY 子句，则 HBase 连接器默认将 rowkey 作为主键。

Kerberos 认证授权

1. 登录集群 Master 节点，获取 krb5.conf、emr.keytab、core-site.xml、hdfs-site.xml、hbase-site.xml 文件，路径如下。

```
/etc/krb5.conf
/var/krb5kdc/emr.keytab
/usr/local/service/hadoop/etc/hadoop/core-site.xml
/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml
/usr/local/service/hbase/conf/hbase-site.xml
```

2. 对获取的配置文件构建 jar 包。

```
jar cvf hbase-xxx.jar krb5.conf emr.keytab core-site.xml hdfs-site.xml hbase-site.xml
```

3. 校验 jar 的结构（可以通过 vim 命令查看），jar 里面包含如下信息，请确保文件不缺失且结构正确。

```
META-INF/
META-INF/MANIFEST.MF
krb5.conf
emr.keytab
core-site.xml
hdfs-site.xml
hbase-site.xml
```

4. 上传 jar 包，并在作业参数配置里引用该程序包。
5. 获取 kerberos principal。

```
klist -kt /var/krb5kdc/emr.keytab
# 输出如下所示，选取第一个即可：hadoop/{IP}@TBDS-XXXX
KVNO Timestamp     Principal
---- ------------------- ------------------------------------------------------
  2 08/09/2021 15:34:40 hadoop/{IP}@TBDS-XXXX
  2 08/09/2021 15:34:40 HTTP/{IP}@TBDS-XXXX
  2 08/09/2021 15:34:40 hadoop/VM-28-51-centos@TBDS-XXXX
  2 08/09/2021 15:34:40 HTTP/VM-28-51-centos@TBDS-XXXX
```

6. 作业高级参数 配置。

```
containerized.taskmanager.env.HADOOP_USER_NAME: hadoop
containerized.master.env.HADOOP_USER_NAME: hadoop
security.kerberos.login.principal: hadoop/{IP}@TBDS-XXXX
security.kerberos.login.keytab: emr.keytab
security.kerberos.login.conf: krb5.conf
```

数据仓库Hive

Hive Connector 支持数据流的目的表，但只支持 append only，不支持 upsert 数据流。数据格式支持包括 Text、SequenceFile、ORC 和 Parquet 等。

版本说明

Flink 版本	说明
1.11	支持 hive 版本 1.1.0、2.3.2、2.3.5、3.1.1 配置项 'connector.type' = 'hive'
1.13	支持 hive 版本 1.0.0 ~ 1.2.2、2.0.0 ~ 2.2.0、2.3.0 ~ 2.3.6、3.0.0 ~ 3.1.2 配置项 'connector' = 'hive'
1.14	不支持
1.16	支持 hive 版本 2.3.0 ~ 2.3.6、3.0.0 ~ 3.1.2 配置项 'connector' = 'hive'

DDL 定义

用作数据目的（Sink）

```
CREATE TABLE hive_table (
  `id` INT,
  `name` STRING,
  `dt` STRING,
  `hr` STRING
) PARTITIONED BY (dt, hr)
with (
    'connector' = 'hive',  -- Flink 1.13 请使用 'connector' = 'hive'
    'hive-version' = '3.1.1',
    'hive-database' = 'testdb',
    'partition.time-extractor.timestamp-pattern'='$dt $hr:00:00',
    'sink.partition-commit.trigger'='partition-time',
```

```
    'sink.partition-commit.delay'='1 h',
    'sink.partition-commit.policy.kind'='metastore,success-file'
);
```

作业配置

在 Hive 数据库中创建 Hive 表。

```
# 在 Hive 的 testdb 数据库创建 hive_table 数据表
USE testdb;
CREATE TABLE `hive_table` (
  `id` int,
  `name` string)
PARTITIONED BY (`dt` string, `hr` string)
STORED AS ORC;
```

对 Hive 表的 HDFS 路径开启写权限。

方式一：可登录 TBDS Hive 集群节点，对目的库 testdb 库的 hive_table 表执行 chmod 操作。

```
hdfs dfs -chmod 777 /usr/hive/warehouse/testdb.db/hive_table
```

方式二：在作业管理 > 作业参数中添加以下高级参数，可以让 hadoop 用户角色获取 HDFS 路径权限。

```
containerized.taskmanager.env.HADOOP_USER_NAME: hadoop
containerized.master.env.HADOOP_USER_NAME: hadoop
```

> 说明：
>
> Flink SQL 中使用 Hive 表 testdb.hive_table，这里 CREATE TABLE 的表名对应 Hive 库的表名（Flink 1.13支持通过hive-table参数配置覆盖该值），库名通过 hive-database 参数指定。

WITH 参数

参数	必填	默认值	描述
connector.type	是	无	Flink-1.11支持，填 'hive' 选择使用 hive connector。
connector	是	无	Flink-1.13支持，填 'hive' 选择使用 hive connector。
hive-version	是	无	TBDS 创建的 Hive 集群对应的版本。
hive-database	是	无	数据要写入的 Hive database。
hive-table	否	无	Flink-1.13支持，填写后该值会作为Hive库的对应表名

参数	必填	默认值	描述
sink.partition-commit.trigger	否	process-time	分区关闭策略。可选值包括： process-time：当分区创建超过一定时间之后将这个分区关闭，分区创建时间为分区创建时的物理时间。 partition-time：当分区创建超过一定时间之后将这个分区关闭，分区创建时间从分区中抽取出来。partition-time 依赖于 watermark 生成，需要配合 watermark 才能支持自动分区发现。当 watermark 时间超过了 从分区抽取的时间与 delay 参数配置时间 之和后会提交分区。
sink.partition-commit.delay	否	0s	分区关闭延迟。当分区在创建超过一定时间之后将被关闭。
sink.partition-commit.policy.kind	是	无	用于提交分区的策略。可选值可以组合使用，可选值包括： success-file：当分区关闭时将在分区对应的目录下生成一个 _success 的文件。 metastore：向 Hive Metastore 更新分区信息。 custom：用户实现的自定义分区提交策略。
partition.time-extractor.timestamp-pattern	否	无	分区时间戳的抽取格式。需要写成 yyyy-MM-dd HH:mm:ss 的形式，并用 Hive 表中相应的分区字段做占位符替换。默认支持第一个字段为 yyyy-mm-dd hh:mm:ss。 如果时间戳应该从单个分区字段 'dt' 提取，可以配置 '$dt'。 如果时间戳应该从多个分区字段中提取，例如 'year'、'month'、'day' 和 'hour'，可以配置 '$year-$month-$day $hour:00:00'。 如果时间戳应该从两个分区字段 'dt' 和 'hour' 提取，可以配置 '$dt $hour:00:00'。
sink.partition-commit.policy.class	否	无	分区提交类，配合 sink.partition-commit.policy.kind = 'custom' 使用，类必须实现 PartitionCommitPolicy。
partition.time-extractor.kind	否	default	分区时间抽取方式。这个配置仅当 sink.partition-commit.trigger 配置为 partition-time 时生效。如果用户有自定义的分区时间抽取方法，配置为 custom。
partition.time-extractor.class	否	无	分区时间抽取类，这个类必须实现 PartitionTimeExtractor 接口。

代码示例1

```
CREATE TABLE datagen_source_table (
  id INT,
  name STRING,
```

```
  log_ts TIMESTAMP(3),
  WATERMARK FOR log_ts AS log_ts - INTERVAL '5' SECOND
) WITH (
  'connector' = 'datagen',
  'rows-per-second' = '10'
);

CREATE TABLE hive_table (
  `id` INT,
  `name` STRING,
  `dt` STRING,
  `hr` STRING
) PARTITIONED BY (dt, hr)
with (
    'connector' = 'hive',  -- Flink 1.13 请使用 'connector' = 'hive'
    'hive-version' = '3.1.1',
    'hive-database' = 'testdb',
    'partition.time-extractor.timestamp-pattern'='$dt $hr:00:00',
    'sink.partition-commit.trigger'='partition-time',
    'sink.partition-commit.delay'='1 h',
    'sink.partition-commit.policy.kind'='metastore,success-file'
);

-- streaming sql, insert into hive table
INSERT INTO hive_table
SELECT id, name, DATE_FORMAT(log_ts, 'yyyy-MM-dd'), DATE_FORMAT(log_ts, 'HH')
FROM datagen_source_table;
```

代码示例2

```
-- hive table 作为source 需要显示指定hive中的database，否则会用默认的数据库default_database
CREATE database testdb;

CREATE TABLE `testdb`.hive_table (
  `user_id` INT,
  `first_name` STRING,
  `dt` STRING,
  `hr` STRING
) PARTITIONED BY (dt, hr) with (
  'connector' = 'hive',
  'hive-version' = '3.1.1',
  'hive-database' = 'testdb',
  'partition.time-extractor.timestamp-pattern' = '$dt $hr:00:00',
  'streaming-source.enable' = 'true',
  'streaming-source.partition.include' = 'all' -- all 、 latest
,
  'streaming-source.monitor-interval' = '1 min',
```

```
  'streaming-source.partition-order' = 'partition-name'
);

CREATE TABLE logger_sink_table (
  `user_id` INT,
  first_name STRING,
  `dt` STRING,
  `hr` STRING
) WITH (
  'connector' = 'logger',
  'print-identifier' = 'DebugData'
);

INSERT into
  logger_sink_table
SELECT
  *
from
  `testdb`.hive_table;
```

Hive 配置

获取 Hive 连接配置 jar 包

Flink SQL 任务写 Hive 时需要使用包含 Hive 及 HDFS 配置信息的 jar 包来连接到 Hive 集群。具体获取连接配置 jar 及其使用的步骤如下：

1. ssh 登录到对应 Hive 集群节点。
2. 获取 hive-site.xml 和 hdfs-site.xml，TBDS 集群中的配置文件在如下位置。

```
/usr/local/service/hive/conf/hive-site.xml
/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml
```

3. 修改 hive-site.xml 文件
   在hive-site增加如下配置，获取配置文件里 hive.metastore.uris 的 value

```
<property>
 <name>hive.metastore.uris</name>
 <value>thrift://ip:7004</value>
</property>
```

4. 对获取到的配置文件 打 jar 包。

```
jar -cvf hive-xxx.jar hive-site.xml hdfs-site.xml
```

5. 校验 jar 的结构（可以通过 vi 命令查看 vi hive-xxx.jar），jar 里面包含如下信息，请确保文件不缺失且结构正

确。

```
META-INF/
META-INF/MANIFEST.MF
hive-site.xml
hdfs-site.xml
```

Kerberos 认证授权

1. 登录集群 Master 节点，获取 krb5.conf、emr.keytab、core-site.xml、hdfs-site.xml、hive-site.xml 文件，路径如下。

```
/etc/krb5.conf
/var/krb5kdc/emr.keytab
/usr/local/service/hadoop/etc/hadoop/core-site.xml
/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml
/usr/local/service/hive/conf/hive-site.xml
```

2. 修改 hive-site.xml 文件。获取配置文件里 hive.metastore.uris 的 value。

```
<property>
 <name>hive.metastore.uris</name>
 <value>thrift://ip:7004</value>
</property>
```

3. 对获取的配置文件打 jar 包。

```
jar cvf hive-xxx.jar krb5.conf emr.keytab core-site.xml hdfs-site.xml hive-site.xml
```

4. 校验 jar 的结构（可以通过 vim 命令查看 vim hive-xxx.jar），jar 里面包含如下信息，请确保文件不缺失且结构正确。

```
META-INF/
META-INF/MANIFEST.MF
emr.keytab
krb5.conf
hdfs-site.xml
```

core-site.xml
hive-site.xml

5. 上传 jar 包，并在作业参数配置里引用该程序包。
6. 获取 kerberos principal，用于作业 高级参数 配置。

```
klist -kt /var/krb5kdc/emr.keytab
# 输出如下所示，选取第一个即可：hadoop/{IP}@TBDS-XXXX
KVNO Timestamp     Principal
---- ------------------- ------------------------------------------------------
  2 08/09/2021 15:34:40 hadoop/{IP}@TBDS-XXXX
  2 08/09/2021 15:34:40 HTTP/{IP}@TBDS-XXXX
  2 08/09/2021 15:34:40 hadoop/VM-28-51-centos@TBDS-XXXX
  2 08/09/2021 15:34:40 HTTP/VM-28-51-centos@TBDS-XXXX
```

7. 作业 高级参数 配置。

```
containerized.taskmanager.env.HADOOP_USER_NAME: hadoop
containerized.master.env.HADOOP_USER_NAME: hadoop
security.kerberos.login.principal: hadoop/{IP}@TBDS-XXXX
security.kerberos.login.keytab: emr.keytab
security.kerberos.login.conf: ${krb5.conf.fileName}
```

注意事项

如果 Flink 作业正常运行，日志中没有报错，但是客户端查不到这个 Hive 表，可以使用如下命令对 Hive 表进行修复（需要将 hive_table_xxx 替换为要修复的表名）。

```
msck repair table hive_table_xxx;
```

文件系统 FileSystem

介绍

FileSystem connector 提供了对 HDFS 等常见文件系统的写入支持。

版本说明

Flink 版本	说明
1.11	支持

Flink 版本	说明
1.13	支持常见的 lzo、snappy 压缩算法
1.14	支持写入到 HDFS，不支持 lzo、snappy 压缩算法
1.16	支持写入到 HDFS，不支持 lzo、snappy 压缩算法

使用范围

FileSystem 支持作为 Append-Only 数据流的目的表 (Sink)，目前还不支持 Upsert 数据流的目的表。FileSystem 目前支持以下格式的数据写入：

- CSV
- JSON
- Avro
- Parquet
- Orc

> 说明：
>
> 目前使用数据格式 Avro、Parquet、Orc 写入时，需要 手动上传额外的 jar 包 才能使用。

DDL 定义

用作数据目的

```
CREATE TABLE `hdfs_sink_table` (
    `id` INT,
    `name` STRING,
    `part1` INT,
    `part2` INT
) PARTITIONED BY (part1, part2) WITH (
    'connector' = 'filesystem',
    'path' = 'hdfs://HDFS10000/data/', -- cosn://${buketName}/path/to/store/data
    'format' = 'json',
    'sink.rolling-policy.file-size' = '1M',
    'sink.rolling-policy.rollover-interval' = '10 min',
    'sink.partition-commit.delay' = '1 s',
    'sink.partition-commit.policy.kind' = 'success-file'
);
```

WITH 参数

参数	必填	默认值	描述
path	是	无	文件写入的路径。

参数	必填	默认值	描述
sink.rolling-policy.file-size	否	128MB	文件最大大小。当写入的文件大小达到设置的阈值时，当前写入的文件将被关闭，并打开一个新的文件进行写入。
sink.rolling-policy.rollover-interval	否	30min	文件最大持续写入时间。当写入的文件写入的时间超过了设置的阈值时，当前写入的文件将被关闭，并打开一个新的文件进行写入。
sink.rolling-policy.check-interval	否	1min	文件检查间隔。FileSystem 按照这个间隔检查文件的写入时间是否已经满足了关闭条件，并将满足条件的文件进行关闭。
sink.partition-commit.trigger	否	process-time	分区关闭策略。可选值包括： process-time：当分区创建超过一定时间之后将这个分区关闭，分区创建时间为分区创建时的物理时间。 partition-time：当分区创建超过一定时间之后将这个分区关闭，分区创建时间从分区中抽取出来。partition-time 依赖于 watermark 生成，需要配合 watermark 才能支持自动分区发现。当 watermark 时间超过了 从分区抽取的时间与 delay 参数配置时间 之和后会提交分区。
sink.partition-commit.delay	否	0s	分区关闭延迟。当分区在创建超过一定时间之后将被关闭。
partition.time-extractor.kind	否	default	分区时间抽取方式。这个配置仅当 sink.partition-commit.trigger 配置为 partition-time 时生效。如果用户有自定义的分区时间抽取方法，配置为 custom。
partition.time-extractor.class	否	无	分区时间抽取类，这个类必须实现 PartitionTimeExtractor 接口。
partition.time-extractor.timestamp-pattern	否	无	分区时间戳的抽取格式。需要写成 yyyy-MM-dd HH:mm:ss 的形式，并用 Hive 表中相应的分区字段做占位符替换。默认支持第一个字段为 yyyy-MM-dd HH:mm:ss。 如果时间戳应该从单个分区字段 'dt' 提取，可以配置 '$dt'。 如果时间戳应该从多个分区字段中提取，例如 'year'、'month'、'day' 和 'hour'，可以配置 '$year-$month-$day $hour:00:00'。 如果时间戳应该从两个分区字段 'dt' 和 'hour' 提取，可以配置 '$dt $hour:00:00'。
sink.partition-commit.policy.kind	是	无	用于提交分区的策略。可选值包括： success-file：当分区关闭时将在分区对应的目录下生成一个 _success 的文件。 custom：用户实现的自定义分区提交策略。

参数	必填	默认值	描述
sink.partition-commit.policy.class	否	无	分区提交类，这个类必须实现 PartitionCommitPolicy。

HDFS 配置

在 HDFS 上创建数据目录后，需为目录开启写权限，才可成功写入数据。流计算 Flink 写入 HDFS 的 user 是 flink。

进行配置前，需要先登录 TBDS 集群下载 Hadoop 集群的 hdfs-site.xml 文件，以获取下列配置中所需的参数值。

HDFS 路径的形式为 hdfs://${dfs.nameservices}/${path}，${dfs.nameservices} 的值可在 hdfs-site.xml 中查找，${path} 为要写入的数据目录。

若目标 Hadoop 集群只有单个 Master，仅需要为 path 参数传入 HDFS 路径即可，无需使用高级参数。

若目标 Hadoop 集群为高可用的双 Master 集群，为 path 参数传入 HDFS 路径后，还需要在作业参数的 高级参数 中对两个 Master 的地址和端口进行配置。以下是一个配置示例，相应的参数值都可在 hdfs-site.xml 中查找并替换。

```
fs.hdfs.dfs.nameservices: HDFS12345
fs.hdfs.dfs.ha.namenodes.HDFS12345: nn2,nn1
fs.hdfs.dfs.namenode.http-address.HDFS12345.nn1: {ip:port}
fs.hdfs.dfs.namenode.https-address.HDFS12345.nn1: {ip:port}
fs.hdfs.dfs.namenode.rpc-address.HDFS12345.nn1: {ip:port}
fs.hdfs.dfs.namenode.http-address.HDFS12345.nn2: {ip:port}
fs.hdfs.dfs.namenode.https-address.HDFS12345.nn2: {ip:port}
fs.hdfs.dfs.namenode.rpc-address.HDFS12345.nn2: {ip:port}
fs.hdfs.dfs.client.failover.proxy.provider.HDFS12345: org.apache.hadoop.hdfs.server.namenode.ha.Confi
guredFailoverProxyProvider
```

> 说明：
>
> Flink 作业默认以 flink 用户操作 HDFS，若没有 HDFS 路径的写入权限，可通过作业 高级参数 设置为有权限的用户，或者设置为超级用户 hadoop。

```
containerized.taskmanager.env.HADOOP_USER_NAME: hadoop
containerized.master.env.HADOOP_USER_NAME: hadoop
```

HDFS Kerberos 认证授权

1. 登录集群 Master 节点，获取 krb5.conf、emr.keytab、core-site.xml、hdfs-site.xml 文件，路径如下。

```
/etc/krb5.conf
/var/krb5kdc/emr.keytab
```

/usr/local/service/hadoop/etc/hadoop/core-site.xml
/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml

2. 对步骤1中获取的文件打 jar 包。

jar cvf hdfs-xxx.jar krb5.conf emr.keytab core-site.xml hdfs-site.xml

3. 校验 jar 的结构（可以通过 vim 命令查看 vim hdfs-xxx.jar），jar 里面包含如下信息，请确保文件不缺失且结构正确。

META-INF/
META-INF/MANIFEST.MF
emr.keytab
krb5.conf
hdfs-site.xml
core-site.xml

4. 上传 jar 包，并在作业参数配置里引用该程序包。
5. 获取 kerberos principal，用于作业 高级参数 配置。

klist -kt /var/krb5kdc/emr.keytab
输出如下所示，选取第一个即可：hadoop/{IP}@TBDS-XXXX
KVNO Timestamp Principal
---- ------------------ --
2 08/09/2021 15:34:40 hadoop/{IP}@TBDS-XXXX
2 08/09/2021 15:34:40 HTTP/{IP}@TBDS-XXXX
2 08/09/2021 15:34:40 hadoop/VM-28-51-centos@TBDS-XXXX
2 08/09/2021 15:34:40 HTTP/VM-28-51-centos@TBDS-XXXX

6. 作业 高级参数 配置。

containerized.taskmanager.env.HADOOP_USER_NAME: hadoop
containerized.master.env.HADOOP_USER_NAME: hadoop
security.kerberos.login.principal: hadoop/{IP}@TBDS-XXXX
security.kerberos.login.keytab: emr.keytab
security.kerberos.login.conf: krb5.conf

如果是 Flink-1.13 版本，需要在高级参数额外增加如下参数，其中参数的值需要为对应 hdfs-site.xml 中的值。

fs.hdfs.dfs.nameservices: HDFS17995
fs.hdfs.dfs.ha.namenodes.HDFS17995: nn2,nn1
fs.hdfs.dfs.namenode.http-address.HDFS17995.nn1: {ip:port}
fs.hdfs.dfs.namenode.https-address.HDFS17995.nn1: {ip:port}
fs.hdfs.dfs.namenode.rpc-address.HDFS17995.nn1: {ip:port}
fs.hdfs.dfs.namenode.http-address.HDFS17995.nn2: {ip:port}
fs.hdfs.dfs.namenode.https-address.HDFS17995.nn2: {ip:port}
fs.hdfs.dfs.namenode.rpc-address.HDFS17995.nn2: {ip:port}
fs.hdfs.dfs.client.failover.proxy.provider.HDFS17995: org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider
fs.hdfs.hadoop.security.authentication: kerberos

代码示例

```
CREATE TABLE datagen_source_table ( id INT, name STRING, part1 INT, part2 INT ) WITH ( 'connector' = 'datagen', 'rows-per-second'='1', -- 每秒产生的数据条数 'fields.part1.min'='1', 'fields.part1.max'='2', 'fields.part2.min'='1', 'fields.part2.max'='2');

CREATE TABLE hdfs_sink_table ( id INT, name STRING, part1 INT, part2 INT) PARTITIONED BY (part1, part2) WITH ( 'connector' = 'filesystem', 'path' = 'hdfs://HDFS10000/data/', 'format' = 'json', 'sink.rolling-policy.file-size' = '1M', 'sink.rolling-policy.rollover-interval' = '10 min', 'sink.partition-commit.delay' = '1 s', 'sink.partition-commit.policy.kind' = 'success-file');

INSERT INTO hdfs_sink_table SELECT id, name, part1, part2FROM datagen_source_table;
```

compressible-fs connector使用说明
只支持在 flink 1.13版本使用。
支持对于 csv 和 json 两种 format 的写入，其它诸如 avro、parquet、orc 文件格式已经自带压缩功能。
支持 LzopCodec、OceanusSnappyCodec 两种压缩算法。
支持写入 hdfs 和 cos 文件，使用方法和 filesystem 一致。
用作数据目的

```
CREATE TABLE `hdfs_sink_table` (
    `id` INT,
    `name` STRING,
    `part1` INT,
    `part2` INT
) PARTITIONED BY (part1, part2) WITH (
    'connector' = 'compressible-fs',
    'hadoop.compression.codec' = 'LzopCodec',
    'path' = 'hdfs://HDFS10000/data/',
```

```
    'format' = 'json',
    'sink.rolling-policy.file-size' = '1M',
    'sink.rolling-policy.rollover-interval' = '10 min',
    'sink.partition-commit.delay' = '1 s',
    'sink.partition-commit.policy.kind' = 'success-file'
);
```

WITH 参数

除上文中 filesystem connector 支持的参数外，compressible-fs 额外特有的参数有以下三个：

参数	必填	默认值	描述
hadoop.compression.codec	否	无	使用的压缩算法，可选值为 LzopCodec 和 OceanusSnappyCodec，不指定时，按照默认的文件格式写入。其中 OceanusSnappyCodec 是由于 snappy 库版本原因，对于 SnappyCodec 的封装，结果完全同 SnappyCodec
filename.suffix	否	无	最终写入文件名，如果没有声明，则会按照支持的压缩算法生成特定的后缀名，如果采用了非 lzop 和 snappy 压缩算法且未声明该值，则文件后缀为空
filepath.contain.partition-key	否	false	写入分区文件时，最终的写入路径是否包括分区字段，默认不包括。例如，假设写入一个按天分区dt=12和按小时分区ht=24的分区路径，默认的分区路径为12/24 而非dt=12/ht=24

模拟上下游 Datagen Logger Blackhole

调试 Source 和 Sink 介绍

当需要检验作业是否可以正常运行、逻辑是否正确时，为了减少外部系统的部署开销，以及避免干扰因素，我们可以使用一些调试专用的 Connector。

版本说明

Flink 版本	说明
1.11	支持
1.13	支持
1.14	支持
1.16	支持

Datagen Source

Datagen 是 Flink 自带的随机数据生成器，它可以作为数据源直接引用。详细的使用方式可参考 Flink 官方文档。

下面是 Datagen 数据源的一个示例，它生成的数据含有两个字段：第一个字段 id 是一个随机数，第二个字段 name 是一个随机字符串。

DDL 定义

```
CREATE TABLE datagen_source_table (
    id INT,
    name STRING
) WITH (
  'connector' = 'datagen',
  'rows-per-second'='1'  -- 每秒产生的数据条数
);
```

WITH 参数

参数	是否必选	默认参数	数据类型	描述
connector	必须	(none)	String	指定要使用的连接器，这里是 'datagen'。
rows-per-second	可选	10000	Long	每秒生成的行数，用以控制数据发出速率。
fields.#.kind	可选	random	String	指定 '#' 字段的生成器。可以是 'sequence' 或 'random'。
fields.#.min	可选	(Minimum value of type)	(Type of field)	随机生成器的最小值，适用于数字类型。
fields.#.max	可选	(Maximum value of type)	(Type of field)	随机生成器的最大值，适用于数字类型。
fields.#.length	可选	100	Integer	随机生成器生成字符的长度，适用于 char、varchar、string。
fields.#.start	可选	(none)	(Type of field)	序列生成器的起始值。
fields.#.end	可选	(none)	(Type of field)	序列生成器的结束值。

Logger Sink

Logger Sink 是TBDS 提供的一个自定义 Logger 示例，它可以将最终的结果数据写入 TaskManager 的日志文件中，后续可以通过 Flink UI 或者控制台的日志面板查看这些日志的输出。TBDS 平台已内置Logger Sink。

DDL 定义

```
CREATE TABLE logger_sink_table (
    id INT,
    name STRING
) WITH (
    'connector' = 'logger',
    'print-identifier' = 'DebugData'
);
```

WITH 参数

参数	是否必选	数据类型	描述
connector	必须	String	指定要使用的连接器，这里是 'logger'。
print-identifier	可选	String	日志打印的前缀信息。
all-changelog-mode	可选	Boolean	启用后，不会过滤 -U 数据，可用来模拟 ClickHouse Collapsing 模式的数据流。
records-per-second	可选	Integer	可指定每秒输出多少条数据，起到限流的作用。
mute-output	可选	Boolean	丢弃所有输出，只做条数统计（类似增强版的 Blackhole Sink）。

监控指标说明

TBDS 为 Logger Sink 增加了很多实用的统计指标。单击 Flink UI 的运行图中的 Logger Sink 算子，即可搜索并查看指标：

numberOfInsertRecords：获取输出的 +I 消息数。

numberOfDeleteRecords：获取输出的 -D 消息数。

numberOfUpdateBeforeRecords：获取输出的 -U 消息数。

numberOfUpdateAfterRecords：获取输出的 +U 消息数。

自定义Connector

介绍

若内置的 Connector 无法满足需求，可以考虑自定义 Connector 功能，即用户可以自行上传实现了相应 Source 和 Sink 接口的类实现，然后作业在运行时会动态加载并调用。

版本说明

Flink 版本	说明
1.11	支持
1.13	支持
1.14	支持
1.16	支持

选择合适的 Connector
用户可以选择第三方提供的 Connector 实现包（例如下面介绍的 Bahir），或者自行通过编程的方式实现。
Apache Bahir 第三方包
Apache Bahir 为 Flink 提供了常见的数据源和数据目的的扩展包。
目前 Bahir 支持如下的第三方组件：
ActiveMQ
Akka
Flume
InfluxDB
Kudu
Redis
Netty
自行编程实现
参见 Flink API。
构建并上传 Connector 包
步骤一：源码构建
建议参考现有的 Connector 的项目，修改其 pom.xml 配置文件，引入相关的依赖包，然后通过 Maven 构建一个 JAR 包。
说明
尽量使用 maven-shade-plugin 将常见的依赖（例如 Apache Commons、Guava 等相关的包）进行 shade 化，以避免引入的库与流计算平台本身的类发生冲突。
步骤二：上传程序包
上传 Connector 的程序包。
步骤三：作业参数引用程序包
在作业的详情页，作业参数选择引用之前上传的程序包和版本。
注意
请务必确认程序包的版本是否符合预期，避免出现各种不可预知的错误。
步骤四：保存并发布
选择程序包后，可以单击保存，也可以选择直接发布草稿。

数据湖Hudi

版本说明

Flink 版本	说明
1.11	不支持
1.13	支持 Source 和 Sink
1.14	不支持
1.16	不支持

使用范围

可以作为 Source/Sink 使用。

DDL 定义

用作数据目的：

```
CREATE TABLE hudi_sink
(
    uuid        VARCHAR(20) PRIMARY KEY NOT ENFORCED,
    name        VARCHAR(10),
    age         INT,
    ts          TIMESTAMP(3),
    `partition` VARCHAR(20)
) WITH (
    'connector' = 'hudi'
    , 'path' = 'hdfs://HDFS1000/data/hudi/mor'
    , 'table.type' = 'MERGE_ON_READ'  --  MERGE_ON_READ 表, 默认值为 COPY_ON_WRITE
    , 'write.tasks' = '3' -- 默认为4
    , 'compaction.tasks' = '4' -- 默认为4
    --  , 'hive_sync.enable' = 'true'  -- 默认值为false
    --  , 'hive_sync.db' = 'default'
    --  , 'hive_sync.table' = 'datagen_mor_1'
    --  , 'hive_sync.mode' = 'jdbc'
    --  , 'hive_sync.username' = ''
    --  , 'hive_sync.password' = ''
    --  , 'hive_sync.jdbc_url' = 'jdbc:hive2://{ip:port}'
    --  , 'hive_sync.metastore.uris' = 'thrift://{ip:port}'
);
```

作为数据源：

```
CREATE TABLE `source`
(
    uuid        VARCHAR(20) PRIMARY KEY NOT ENFORCED,
    name        VARCHAR(10),
    age         INT,
    ts          TIMESTAMP(3),
    `partition` VARCHAR(20)
) WITH (
      'connector' = 'hudi'
      , 'path' = 'hdfs://{ip:port}/path/hudidata'
      , 'table.type' = 'MERGE_ON_READ'  -- MOR 表, 目前无法读取增量数据
      , 'read.tasks' = '1'  -- 读task的并行度,默认值为4
      , 'hoodie.datasource.query.type' = 'snapshot' -- 默认值为snapshot, 可选值为 read_optimized, increm
ental
      , 'read.streaming.enabled' = 'true'  -- this option enable the streaming read
      , 'read.start-commit' = 'earliest' -- specifies the start commit instant time, the commit time format s
hould be 'yyyyMMddHHmmss'
      , 'read.streaming.check-interval' = '4'
      );
```

WITH 参数

通用参数

参数	必填	默认值	描述
connector	是	无	必须填 hudi
path	是	无	数据的存储路径（如果存储到 HDFS，格式为 hdfs://；存储为 COS 为 COSN://$bucket/$path）

作为 Sink 的参数

参数	必填	默认值	描述
table.type	否	COPY_ON_WRITE	Hudi 表类型，可选值为 COPY_ON_WRITE 或者 MERGE_ON_READ

HoodieRecord 字段相关

参数	必填	默认值	描述
hoodie.datasource.write.recordkey.field	否	uuid	key 字段，flink table 如果有 primary key，则采用 flink table 的 pk

参数	必填	默认值	描述
hoodie.datasource.write.partitionpath.field	否	""	分区路径字段，为空表示不分区
write.precombine.field	否	ts	预合并时，相同 key 记录时候，用于比较(Object.compareTo(..))的字段

并行度相关

参数	必填	默认值	描述
write.tasks	否	4	写算子的并行度
write.index_bootstrap.tasks	否	无	index bootstrap 的并行度，默认为作业的并行度
write.bucket_assign.tasks	否	无	bucket assign 的并行度，默认为作业的并行度
compaction.tasks	否	4	compaction 任务的并行度

compaction相关

参数	必填	默认值	描述
compaction.schedule.enabled	否	true	是否启动 compaction
compaction.async.enabled	否	true	compaction 是否采用异步
compaction.trigger.strategy	否	num_commits	num_commits / time_elapsed / num_and_time / num_or_time

hive 元数据同步相关

参数	必填	默认值	描述
hive_sync.enabled	否	false	-
hive_sync.db	否	-	-
hive_sync.table	否	-	-
hive_sync.mode	否	JDBC	可选值 hms，JDBC and HiveQL
hive_sync.username	否	-	-
hive_sync.password	否	-	-
hive_sync.jdbc_url	否	-	-

更多参数

其他更详细的参数，可参见 Flink Options。

作为 Source 的参数

参数	必填	默认值	描述
read.tasks	否	4	读算子的并行度
hoodie.datasource.query.type	否	snapshot	可选值 snapshot / read_optimized / incremental
read.streaming.enabled	否	false	-
read.streaming.check-interval	否	60	单位秒，streaming read的检查时间间隔
read.streaming.skip_compaction	否	false	-
read.start-commit	否	无	格式为 yyyyMMddHHmmss；可设置为 earliest，表示从最早的 commit 开始消费
read.end-commit	否	无	streaming read，无需设置该值

更详细的参数配置

可参见 Flink Options。

HDFS 配置

获取 HDFS 链接配置 jar

Flink SQL 任务写 Hudi，使用 HDFS 存储时需要使用包含 HDFS 配置信息的 jar 包来连接到 HDFS 集群。具体获取连接配置 jar 及其使用的步骤如下：

1. SSH 登录到对应 HDFS 集群节点。
2. 获取 hdfs-site.xml，TBDS 集群中的配置文件在如下位置。

```
/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml
```

3. 对获取到的配置文件打 jar 包。

```
jar -cvf hdfs-xxx.jar hdfs-site.xml
```

4. 校验 jar 的结构（可以通过 vi 命令查看 ），jar 里面包含如下信息，请确保文件不缺失且结构正确。

```
vi hdfs-xxx.jar
```

```
META-INF/
META-INF/MANIFEST.MF
hdfs-site.xml
```

配置写入 HDFS 的用户

> 说明：
>
> Flink 作业默认以 flink 用户操作 HDFS，若没有 HDFS 路径的写入权限，可通过 作业高级参数 设置为有权限的用户，或者设置为超级用户 hadoop。

```
containerized.taskmanager.env.HADOOP_USER_NAME: hadoop
containerized.master.env.HADOOP_USER_NAME: hadoop
```

Kerberos 认证授权

1. 登录集群 Master 节点，获取 krb5.conf、emr.keytab、core-site.xml、hdfs-site.xml 文件，路径如下。

```
/etc/krb5.conf
/var/krb5kdc/emr.keytab
/usr/local/service/hadoop/etc/hadoop/core-site.xml
/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml
```

2. 对获取的配置文件打 jar 包。

```
jar cvf hdfs-xxx.jar krb5.conf emr.keytab core-site.xml hdfs-site.xml
```

3. 校验 jar 的结构（可以通过 vim 命令查看 vim hdfs-xxx.jar），jar 里面包含如下信息，请确保文件不缺失且结构正确。

```
META-INF/
META-INF/MANIFEST.MF
emr.keytab
krb5.conf
hdfs-site.xml
core-site.xml
```

4. 上传 jar 包，并在作业参数配置里引用该程序包。
5. 获取 kerberos principal，用于 作业高级参数 配置。

```
klist -kt /var/krb5kdc/emr.keytab

# 输出如下所示，选取第一个即可：hadoop/{IP}@TBDS-XXXX
KVNO Timestamp     Principal
---- ------------------- ------------------------------------------------------
 2 08/09/2021 15:34:40 hadoop/{IP}@TBDS-XXXX
 2 08/09/2021 15:34:40 HTTP/{IP}@TBDS-XXXX
 2 08/09/2021 15:34:40 hadoop/VM-28-51-centos@TBDS-XXXX
 2 08/09/2021 15:34:40 HTTP/VM-28-51-centos@TBDS-XXXX
```

6. 作业高级参数 配置。

```
containerized.taskmanager.env.HADOOP_USER_NAME: hadoop
containerized.master.env.HADOOP_USER_NAME: hadoop
security.kerberos.login.principal: hadoop/{IP}@TBDS-XXXX
security.kerberos.login.keytab: emr.keytab
security.kerberos.login.conf: krb5.conf
```

数据湖Iceberg

版本说明

Flink 版本	说明
1.11	不支持
1.13	支持 Source 和 Sink
1.14	支持 Source 和 Sink
1.16	支持 Source 和 Sink

使用范围

可以作为 Source/Sink 使用。其中，作为 source 使用时，不支持 upsert 写入的源 iceberg。

DDL 定义

用作数据目的：

```
CREATE TABLE `sink` (
```

```
  `id` bigint,
  `YCSB_KEY` string,
  `FIELD0` string,
  `FIELD1` string,
  `FIELD2` string,
  `database_name` string,
  `table_name` string,
  `op_ts` timestamp(3),
  `date` string
) PARTITIONED BY (`date`) WITH (
  'connector' = 'iceberg-1.1',
  'write.upsert.enabled'='false', -- 是否开启upsert
  'catalog-type' = 'hive',
  'catalog-name'='xxx',
  'catalog-database'='xxx',
  'catalog-table'='xxx',
  'warehouse' = 'hdfs://HDFS14979/usr/hive/warehouse',
  -- Hive metastore 的 thrift URI，可以从hive-site.xml配置文件中获取，对应的Key为：hive-metastore-uris
  'uri'='thrift://ip:port',
  'engine.hive.enabled' = 'true',
  'format-version' = '2'
);
```

作为数据源：

```
CREATE TABLE `icesource` (
  `id` bigint,
  `YCSB_KEY` string,
  `FIELD0` string,
  `FIELD1` string,
  `FIELD2` string,
  `database_name` string,
  `table_name` string,
  `op_ts` timestamp(3),
PRIMARY KEY(id) NOT ENFORCED
) WITH (
  'connector' = 'iceberg-1.1',
  'catalog-name' = 'hive_catalog',
  'catalog-type' = 'hive',
  'catalog-database' = 'database_ta',
  'catalog-table' = 't_p1_hive3_avro_3',
  'warehouse'='hdfs://HDFS14979/usr/hive/warehouse',
  'engine.hive.enabled' = 'true',
  'format-version' = '2',
  'streaming'='true',
  'monitor-interval'='10',
```

```
  -- Hive metastore 的 thrift URI，可以从hive-site.xml配置文件中获取，对应的Key为：hive-metastore-uri
s
  'uri'='thrift://ip:port'
);
```

WITH 参数

通用参数

参数	必填	默认值	描述
connector	是	无	必须填 iceberg-1.1
warehouse	是	无	数据的存储路径（如果存储到 HDFS，格式为 hdfs:// ；存储为 COS 为 COSN://$bucket/$path ）
catalog-name	是	无	自定义的 catalog 名
catalog-type	是	无	catalog 类型，可选值为 hadoop / hive / custom
catalog-database	是	无	iceberg 数据库名称
catalog-table	是	无	iceberg 表名称
catalog-impl	否	无	catalog-type 为 custom时，必填
uri	否	无	Hive metastore 的 thrift URI，可以从 hive-site.xml 配置文件中获取，对应的 Key 为：hive-metastore-uris; e.g. thrift://{ip:port}
format-version	否	1	Iceberg 格式 请参见 Iceberg Table Spec

更多参数请参见 Configuration。

HDFS 配置

获取 HDFS 连接配置 jar

Flink SQL 任务写 Iceberg，使用 HDFS 存储时需要使用包含 HDFS 配置信息的 jar 包来连接到 HDFS 集群。具体获取连接配置 jar 及其使用的步骤如下：

1. SSH 登录到对应 HDFS 集群节点。
2. 获取 hdfs-site.xml，TBDS 集群中的配置文件在如下位置。

```
/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml
```

3. 对获取到的配置文件 打 jar 包。

```
jar -cvf hdfs-xxx.jar hdfs-site.xml
```

4. 校验 jar 的结构（可以通过 vi 命令查看 ），jar 里面包含如下信息，请确保文件不缺失且结构正确。

```
vi hdfs-xxx.jar

META-INF/
META-INF/MANIFEST.MF
hdfs-site.xml
```

配置写入 hdfs 的用户

> 说明：
>
> Flink 作业默认以 flink 用户操作 HDFS，若没有 HDFS 路径的写入权限，可通过 作业高级参数 设置为有权限的用户，或者设置为超级用户 hadoop。

```
containerized.taskmanager.env.HADOOP_USER_NAME: hadoop
containerized.master.env.HADOOP_USER_NAME: hadoop
```

Kerberos 认证授权

1. 登录集群 Master 节点，获取 krb5.conf、emr.keytab、core-site.xml、hdfs-site.xml、hive-site.xml 文件，路径如下：

```
/etc/krb5.conf
/var/krb5kdc/emr.keytab
/usr/local/service/hadoop/etc/hadoop/core-site.xml
/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml
/usr/local/service/hive/conf/hive-site.xml
```

2. 修改 hive-site.xml 文件。在 hive-site.xml 中增加如下配置，获取配置文件里 hive.metastore.uris 的 value。

```
<property>
 <name>hive.metastore.uris</name>
 <value>thrift://ip:7004</value>
</property>
```

3. 对获取的配置文件打 jar 包。

```
jar cvf hive-xxx.jar krb5.conf emr.keytab core-site.xml hdfs-site.xml hive-site.xml
```

4. 校验 jar 的结构（可以通过 vim 命令查看 vim hdfs-xxx.jar），jar 里面包含如下信息，请确保文件不缺失且结构正确。

```
META-INF/
META-INF/MANIFEST.MF
emr.keytab
krb5.conf
hdfs-site.xml
core-site.xml
hive-site.xml
```

5. 上传 jar 包，并在作业参数配置里引用该程序包。
6. 获取 kerberos principal，用于 作业高级参数 配置。

```
klist -kt /var/krb5kdc/emr.keytab

# 输出如下所示，选取第一个即可：hadoop/{IP}@TBDS-XXXX
KVNO Timestamp     Principal
---- ------------------- ------------------------------------------------------
 2 08/09/2021 15:34:40 hadoop/{IP}@TBDS-XXXX
 2 08/09/2021 15:34:40 HTTP/{IP}@TBDS-XXXX
 2 08/09/2021 15:34:40 hadoop/VM-28-51-centos@TBDS-XXXX
 2 08/09/2021 15:34:40 HTTP/VM-28-51-centos@TBDS-XXXX
```

7. 作业高级参数 配置。

```
containerized.taskmanager.env.HADOOP_USER_NAME: hadoop
containerized.master.env.HADOOP_USER_NAME: hadoop
security.kerberos.login.principal: hadoop/{IP}@TBDS-XXXX
security.kerberos.login.keytab: emr.keytab
security.kerberos.login.conf: krb5.conf
```

> 说明：
>
> security.kerberos.login.keytab 和 security.kerberos.login.conf 的值为对应的文件名。

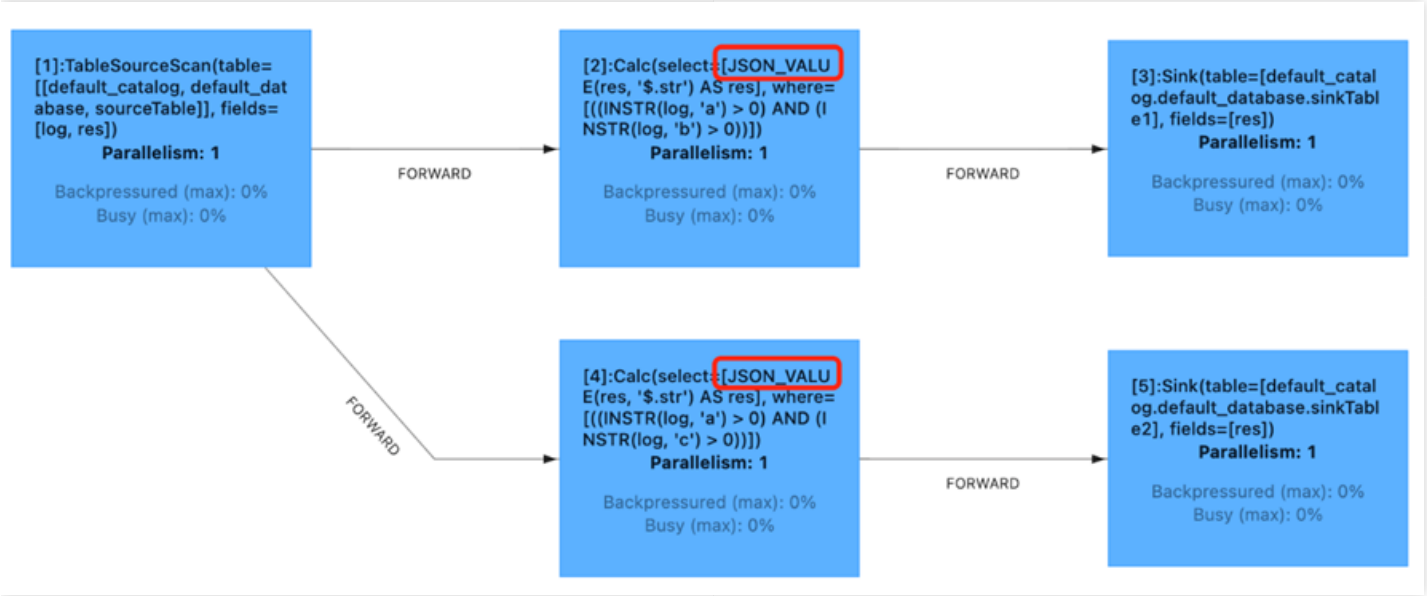
最佳实践
SQL解析分段优化方案

需求背景

在 Flink SQL 作业的开发过程中，如果需要复用某段逻辑，通常就将其定义为 View，希望能够减少重复计算。

```
-- 模拟一个数据源
CREATE TABLE `sourceTable` (`log` String,`res` String) WITH ('connector' = 'kafka');
-- 自定义函数解析
CREATE view rowTable AS SELECT JSON_VALUE (res, '$.str') as res,`log`
FROM sourceTable WHERE INSTR(`log`,'a') > 0;
-- 两个目标表写入
CREATE TABLE sinkTable1 (`res` String) WITH ( 'connector' = 'print');
CREATE TABLE sinkTable2 (`res` String) WITH ( 'connector' = 'print');
INSERT INTO sinkTable1 SELECT res FROM rowTable WHERE INSTR(`log`,'b') > 0;
INSERT INTO sinkTable2 SELECT res FROM rowTable WHERE INSTR(`log`,'c') > 0;
```

开源Flink 在通过 View 消除重复计算上存在不足，例如如上作业，View 中的耗时自定义函数解析 JSON_VALUE 重复计算了两次，降低了作业的性能。



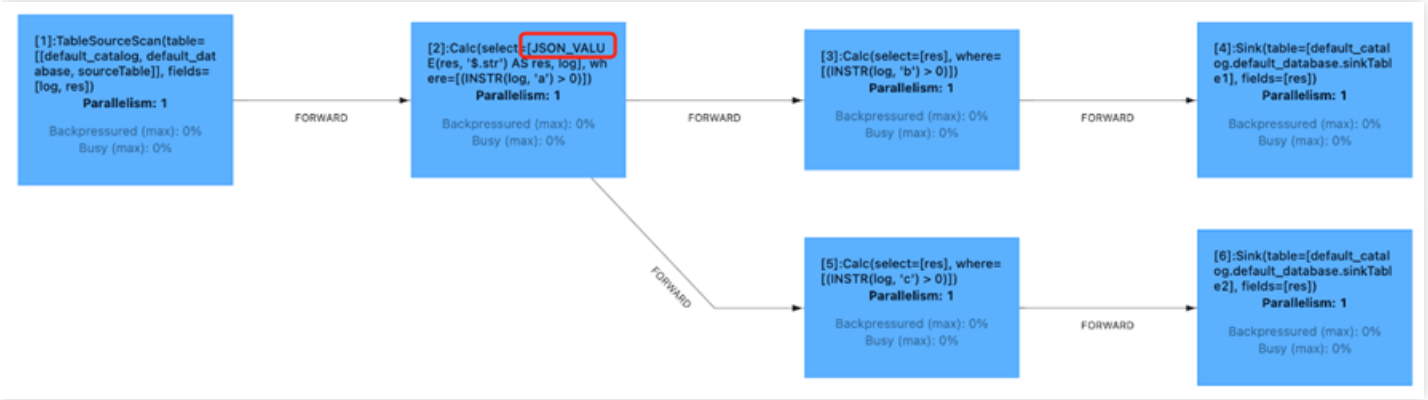
本方案用于消除 View 重复计算。如果客户的作业中使用了 View，且 View 中存在耗时的自定义函数解析，可参考本方案。

方案介绍

Flink SQL 解析阶段，在对 View 展开时，我们对 View 子查询的头节点自动添加特定的 Hint。分段切分时，通过特定 Hint 识别出 View 头节点，并基于该节点划分出新的优化段，从而达到按 View 切分，实现分段优化的效果。在作业中添加如下语句即可开启优化。

```
SET table.optimizer.view-as-breakpoint-enabled = true;
```

方案效果



对上面的示例作业，启用分段优化后，消除了重复计算。我们使用采样数据对客户作业进行了实际测试，性能有成倍的提升。

Group Aggregate最佳实践

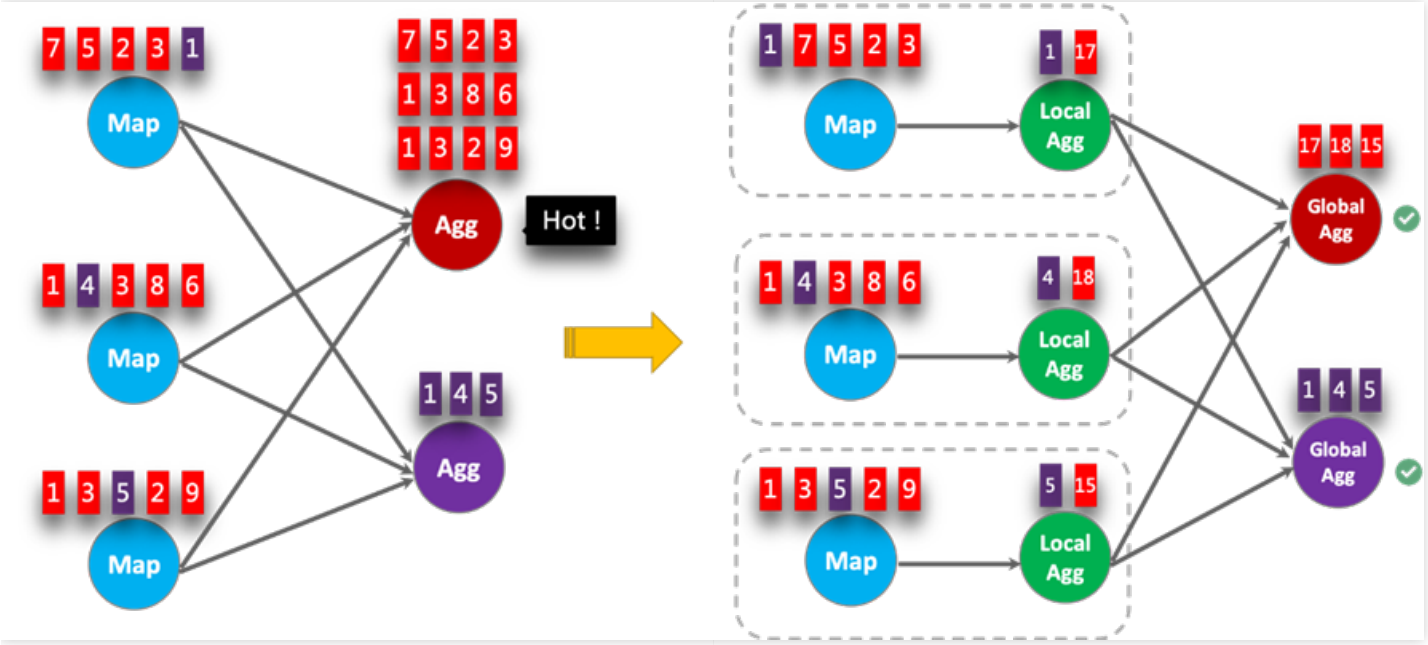
本节最佳实践的更多详情可参考Flink官方文档。

开启MiniBatch，提升吞吐

MiniBatch是缓存一定的数据后再触发处理，以减少对State的访问，从而提升吞吐并减少数据的输出量，通常对于聚合场景，MiniBatch可以显著地提升系统性能，建议开启。

开启LocalGlobal，解决常见数据热点问题

LocalGlobal本质上能够靠LocalAgg的聚合筛除部分倾斜数据，从而降低GlobalAgg的热点，提升性能。LocalGlobal优化将原先的Aggregate分成Local和Global两阶段聚合，即MapReduce模型中的Combine和Reduce两阶段处理模式。第一阶段在上游节点本地攒一批数据进行聚合（LocalAgg），并输出这次微批的增量值（Accumulator）。第二阶段再将收到的Accumulator合并（Merge），得到最终的结果（GlobalAgg）。



LocalGlobal 可用于提升普通聚合（例如SUM、COUNT、MAX、MIN和AVG）的性能，以及解决这些场景下的数据热点问题。

开启Split Distinct Aggregation，解决COUNT DISTINCT热点问题

为了解决COUNT DISTINCT的热点问题，通常需要手动改写为两层聚合（增加按Distinct Key取模的打散层）。Flink提供了COUNT DISTINCT自动打散，即Split Distinct Aggregation优化，您无需自行改写为两层聚合。LocalGlobal优化针对普通聚合（例如SUM、COUNT、MAX、MIN和AVG）有较好的效果，对于COUNT DISTINCT收效不明显，因为COUNT DISTINCT在Local聚合时，对于DISTINCT KEY的去重率不高，导致在Global节点仍然存在热点问题。如果您发现COUNT DISTINCT聚合存在性能瓶颈，建议开启该优化。

FILTER 替换CASE WHEN，提升大量COUNT DISTINCT场景性能

统计作业需要计算各种维度的UV，例如全网UV、来自手机客户端的UV、来自PC的UV等等。

```
SELECT
 day,
 COUNT(DISTINCT user_id) AS total_uv,
 COUNT(DISTINCT CASE WHEN flag IN ('android', 'iphone') THEN user_id ELSE NULL END) AS app_uv,
 COUNT(DISTINCT CASE WHEN flag IN ('wap', 'other') THEN user_id ELSE NULL END) AS web_uv
FROM T
GROUP BY day
```

建议使用标准的AGG WITH FILTER语法来代替CASE WHEN实现多维度统计的功能。实时计算目前的SQL优化器能分析出Filter参数，从而同一个字段上计算不同条件下的COUNT DISTINCT能共享State，减少对State的读写操作，从而提高作业性能。

```
SELECT
 day,
 COUNT(DISTINCT user_id) AS total_uv,
 COUNT(DISTINCT user_id) FILTER (WHERE flag IN ('android', 'iphone')) AS app_uv,
 COUNT(DISTINCT user_id) FILTER (WHERE flag IN ('wap', 'other')) AS web_uv
FROM T
GROUP BY day
```

TopN最佳实践

更新流TopN算法优化

当TopN的输入是更新流时（例如经过了AGG或JOIN计算），TopN有2种算法，性能从高到低分别是：UpdateFastRank和RetractRank。算法名字会显示在拓扑图的节点名字上。RetractRank在某些业务场景下可优化成UpdateFastRank，使用UpdateFastRank算法需要具备3个条件：

1. 输入流为更新流，但不能包含DELETE（D）、UPDATE_BEFORE（UB）类型的消息，否则会影响排序字段的单调性。关于输入流的消息类型，可以通过执行EXPLAIN CHANGELOG_MODE命令来获取对应节点输出的消息类型，语法详情请参见EXPLAIN语句。
2. 输入流有Primary Key信息，例如上游做了GROUP BY聚合操作。
3. 排序字段的更新是单调的，且单调方向与排序方向相反。例如，ORDER BY COUNT/COUNT_DISTINCT/SUM（正数）DESC。对于ORDER BY SUM DESC时，可对 SUM 的字段添加过滤条件，只对正数求和，从而确保 SUM 结果单调递增。

无排名优化

TopN的输出结果不需要显示rownum值，仅需在最终前端显示时进行1次排序，极大地减少输入结果表的数据量。

增加TopN的Cache大小

TopN为了提升性能有一个State Cache层，Cache层能提升对State的访问效率。TopN的Cache命中率的计算公式如下。

```
cache_hit = cache_size*parallelism/top_n/partition_key_num
```

例如，Top100配置缓存10000条，并发50，当您的PartitionBy的Key维度较大时，例如10万级别时，Cache命中率只有10000*50/100/100000=5%，命中率会很低，导致大量的请求都会击中State（磁盘），性能会大幅下降。因此当partitionKey维度特别大时，可以适当加大TopN的cache size，相对应的也建议适当加大TopN节点的heap memory。

```
table.exec.rank.topn-cache-size: 200000
```

调整TopN cache到200000，那么理论命中率能达到200000*50/100/100000 = 100%。

PartitionBy的字段中要有时间类字段

例如每天的排名，要带上Day字段，否则TopN的最终结果会由于State TTL产生错乱。

高效去重最佳实践

Flink的源数据在部分场景中存在重复数据，这时通常需要进行去重。Flink 内置了去重的功能，有保留第一条（Deduplicate Keep FirstRow）和保留最后一条（Deduplicate Keep LastRow）两种去重方案。由于Flink SQL上没有直接支持去重的语法，因此使用了SQL的ROW_NUMBER OVER WINDOW功能来实现去重语法。去重本质上是一种特殊的TopN。详情可参考Flink官网文档。

```
SELECT [column_list]
FROM (
  SELECT [column_list],
    ROW_NUMBER() OVER ([PARTITION BY col1[, col2...]]
      ORDER BY time_attr [asc|desc]) AS rownum
  FROM table_name)
WHERE rownum = 1
```

UDF最佳实践

使用内置函数替换自定义函数

内置函数在持续的优化当中，请尽量使用内置函数替换自定义函数。内置函数优化了数据序列化和反序列化的耗时，并支持直接对字节单位进行操作。

LIKE操作注意事项

1. 如果需要进行StartsWith操作，使用LIKE 'xxx%'。
2. 如果需要进行EndsWith操作，使用LIKE '%xxx'。
3. 如果需要进行Contains操作，使用LIKE '%xxx%'。
4. 如果需要进行Equals操作，使用LIKE 'xxx'，等价于str = 'xxx'。
5. 如果需要匹配下划线（_），请注意要完成转义LIKE '%seller/*id%' ESCAPE '/'。下划线（*）在SQL中属于单字符通配符，能匹配任何字符。如果声明为 LIKE '%seller_id%'，则不单会匹配seller_id，还会匹配seller#id、sellerxid或seller1id等，导致结果错误。

慎用正则函数（REGEXP）

正则表达式是非常耗时的操作，对比加减乘除通常有百倍的性能开销，而且正则表达式在某些极端情况下可能会进入无限循环，导致作业阻塞，具体情况请参见Regex execution is too slow，因此建议使用LIKE。

常见问题

切换其他用户提交任务

在 ranger 中赋予用户 hdfs/yarn/hbase 其他要使用的组件权限(和 admin 用户同级)，更改 flink-conf.yaml 中配置的 principal 和 keytab 信息为对应用户。

数据倾斜

当数据发生倾斜（某一部分数据量特别大），虽然没有GC（Garbagee Collection，垃圾回收），但是task执行时间严重不一致。

- 需要重新设计key，以更小粒度的key使得task大小合理化。
- 修改并行度。
- 调用rebalance操作，使数据分区均匀。

报错java.lang.OutOfMemoryError: GC overhead limit exceeded，该如何处理？

- 报错原因该报错代表为作业设定的内存不够，导致GC超时。常见原因为代码（如UDF）发生内存泄露或者内存大小确实不能满足业务需求。
- 解决方案
- 您可在重新运行问题作业前通过-D方式指定JVM参数，保存OutOfMemoryError发生时的现场-D env.java.opts="-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/dump.hprof"。
- 在flink-conf.yaml中添加参数env.java.opts: -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/dump.hprof来配置OutOfMemoryError发生时进行heap dump。
- 待作业再次报错之后，您可针对HeapDumpPath指定的heap dump文件进行分析，确定问题根因。

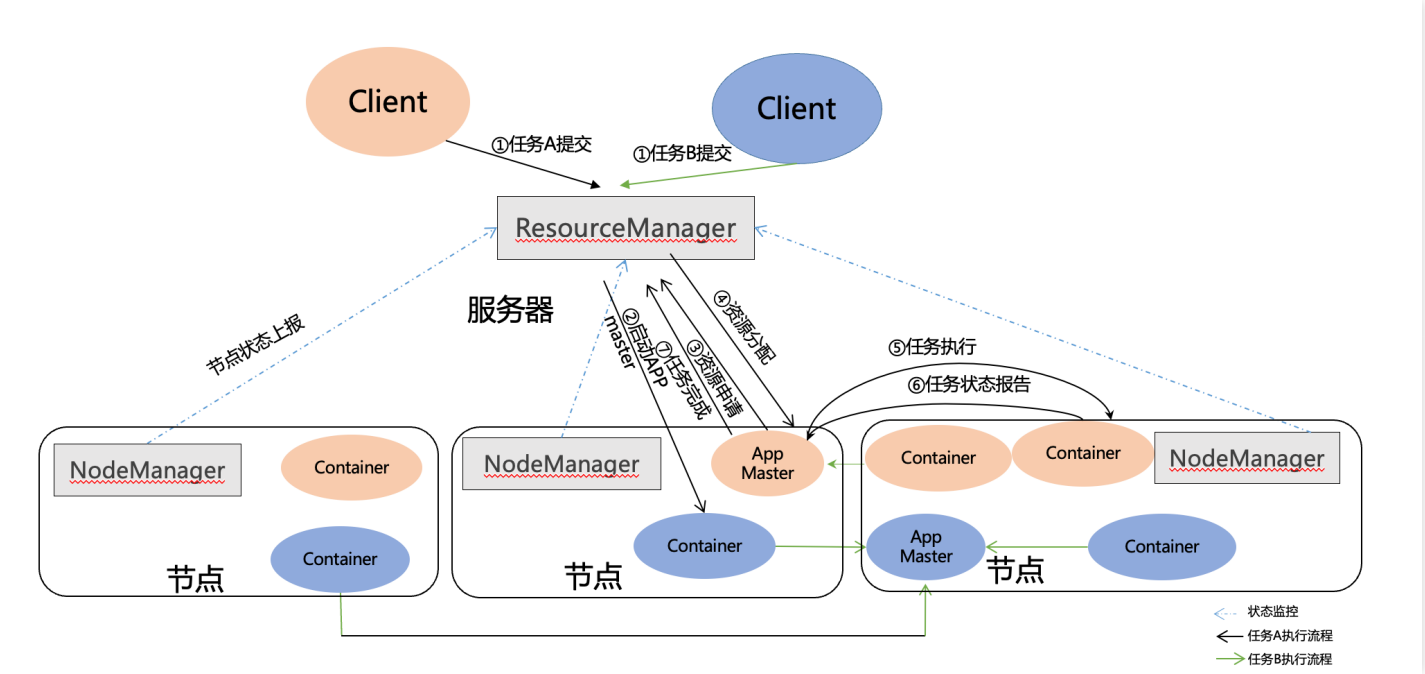
Flink UI上作业只有一个Operator，并且显示Records Received为0，该如何处理？

这是正常现象，Flink的Records Received相关指标用于描述不同Operator之间的数据通信，当作业被优化为一个Operator时，该指标值恒为0。

YARN开发
概述

Yarn架构简介

Hadoop YARN(Yet Another Resource Negotiator)是一个分布式的资源管理系统，用于提高分布式的集群环境下的资源利用率，这些资源包括内存、IO、网络、磁盘等。其产生的原因是为了解决原MapReduce框架的不足。最初MapReduce的committer还可以周期性的在已有的代码上进行修改，可是随着代码的增加以及原MapReduce框架设计的不足，在原MapReduce框架上进行修改变得越来越困难，所以MapReduce的committer决定从架构上重新设计MapReduce，使下一代的MapReduce(MRv2/YARN)框架具有更好的扩展性、可用性、可靠性、向后兼容性和更高的资源利用率，以及能支持除了MapReduce计算框架外的更多的计算框架。YARN(MR v2) 在 MR v1 的基础上发展而来，将资源管理和任务控制解耦，分别由 ResourceManager 和 ApplicationMaster 负责，是一个两层调度系统。

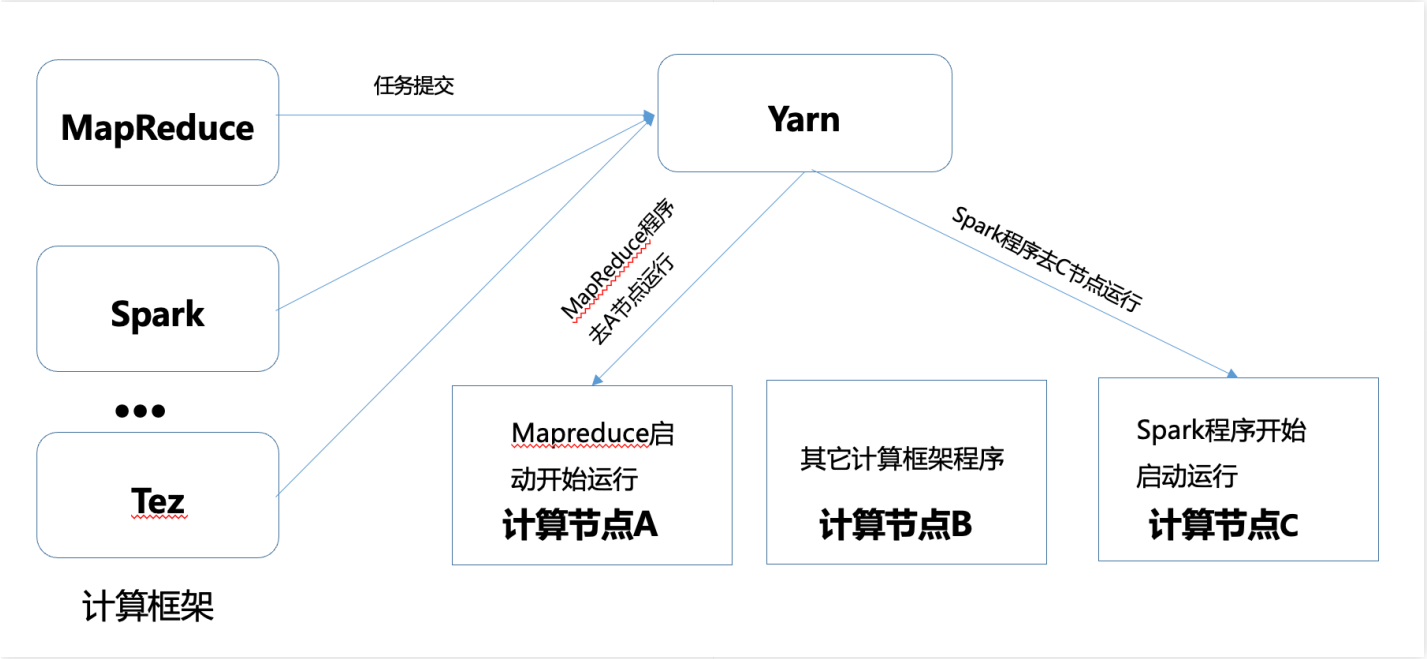


基本概念：

- ResourceManager：以下简称RM。YARN的中控模块，负责统一规划资源的使用。它接收来自NM的汇报，建立AM，并将资源派送给AM。
- NodeManager：以下简称NM。YARN中的资源节点模块，负责启动管理container。
- ApplicationMaster：以下简称AM。YARN中每个应用都会启动一个AM，负责向RM申请资源，请求NM启动container，并告诉container做什么事情。
- Container：对任务运行环境的抽象。它描述一系列信息：任务运行资源（包括节点、内存、CPU）、任务启

动命令、任务运行环境。

MapReduce和Yarn的关系



作用:
Yarn负责集群资源的管理及调度。
MapReduce负责数据的计算过程。
区别:
Yarn是一个服务，时时刻刻在运行着。
MapReduce是个静态的框架，就像程序一样，不用时无需启动。
而YARN 作为 Hadoop 的资源管理框架，提供了一个通用的资源调度平台，支持多种计算框架（如 MapReduce、Spark、Tez 等）。

示例工程开发

环境准备

参考Spark环境准备。

代码逻辑说明

1. 功能说明。
   本示例演示获取yarn app列表的功能。

2. POM依赖。

```
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-common</artifactId>
  <version>3.2.2-${TBDS_VERSION}</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-auth</artifactId>
  <version>3.2.2-${TBDS_VERSION}</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>tq-security</artifactId>
  <version>3.2.2-${TBDS_VERSION}</version>
</dependency>
<dependency>
   <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-yarn-common</artifactId>
  <version>3.2.2-${TBDS_VERSION}</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-yarn-api</artifactId>
  <version>3.2.2-${TBDS_VERSION}</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-yarn-client</artifactId>
  <version>3.2.2-${TBDS_VERSION}</version>
</dependency>
```

3. 代码示例。

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.security.UserGroupInformation;
import org.apache.hadoop.yarn.api.records.ApplicationReport;
import org.apache.hadoop.yarn.api.records.YarnApplicationState;
import org.apache.hadoop.yarn.client.api.YarnClient;
import org.apache.hadoop.yarn.exceptions.YarnException;

import java.io.File;
import java.io.IOException;
import java.util.EnumSet;
import java.util.List;

public class YarnClientExample {

    public static void main(String[] args) throws IOException{

        StringconfPath = "/usr/local/service/hadoop/etc/hadoop";
        Configurationconf = new Configuration();
        System.out.println(confPath + File.separator + "core-site.xml");
        conf.addResource(new Path(confPath + File.separator + "core-site.xml"));
        conf.addResource(new Path(confPath + File.separator + "hdfs-site.xml"));
        conf.addResource(new Path(confPath + File.separator + "yarn-site.xml"));
        YarnClientyarnClient = YarnClient.createYarnClient();
        //kerberos认证
        System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
       conf.setBoolean("hadoop.security.authorization", true);
        conf.set("hadoop.security.authentication", "Kerberos");
        UserGroupInformation.setConfiguration(conf);
        UserGroupInformation.loginUserFromKeytab("xxxx/x.x.x.x@TBDS-XXXXXX", "/var/krb5kdc/emr.key
tab");

        yarnClient.init(conf);
        yarnClient.start();
        try {
            List<ApplicationReport>applications = yarnClient.getApplications(EnumSet.of(YarnApplicationS
tate.RUNNING, YarnApplicationState.FINISHED));
            System.out.println("getApplications..." + applications.size());
            for (ApplicationReportapplication : applications) {
                System.out.println("\nApplicationId ============> " + application.getApplicationId());
                System.out.println("name ============> " + application.getName());
                System.out.println("queue ============> " + application.getQueue());
                System.out.println("user ============> " + application.getUser());
                System.out.println("type ============> " + application.getApplicationType());
                System.out.println("status ============> " + application.getFinalApplicationStatus());
            }
            System.out.println("end...");
```

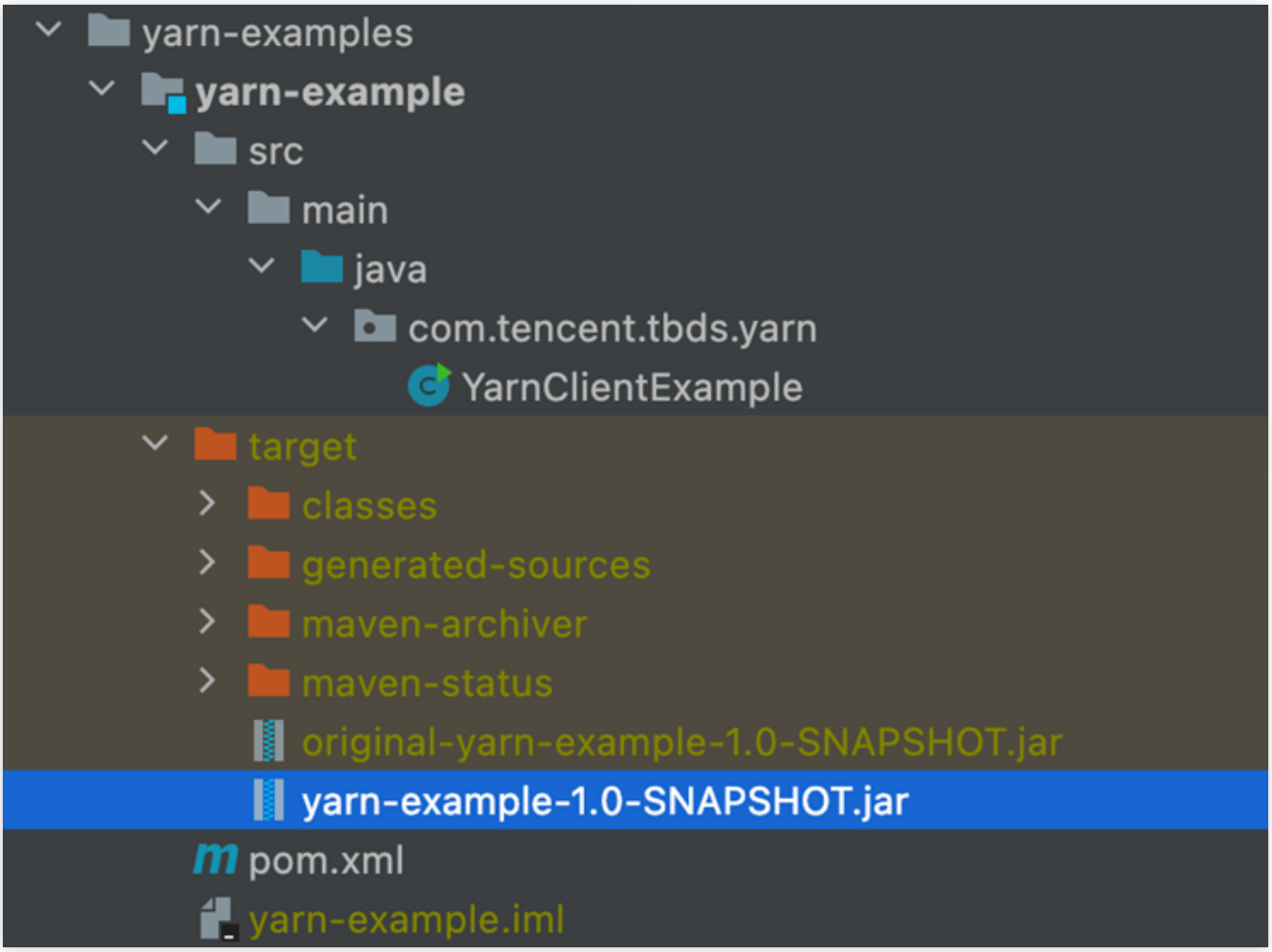
```
        } catch (YarnException| IOExceptione) {
            e.printStackTrace();
        }
        yarnClient.stop();
    }
}
```

代码调试

1. IntelliJ IDEA调试，参考Spark代码调试。

2. Linux环境编译参考：

```
 //编译
/usr/local/jdk/bin/javac -classpath hadoop-common-3.2.2-${TBDS_VERSION}.jar:hadoop-yarn-ap
i-3.2.2-${TBDS_VERSION}.jar:hadoop-yarn-client-3.2.2-${TBDS_VERSION}.jar:security-common-1.1.9.1.j
ar YarnClientExample.java
//打包
/usr/local/jdk/bin/jar -cvf YarnClientExample.jar YarnClientExample.class

//建议使用idea编译打包fat jar
//参考示例工程 yarn-examples/yarn-example 打包出 com.tencent.tbds.yarn.YarnClientExample
mvn clean package
```



3. 测试运行。

```
 //运行
/usr/local/jdk/bin/java -classpath yarn-example-1.0-SNAPSHOT.jar com.tencent.tbds.yarn.YarnClientExa
mple
```

```
ApplicationId =============> application_1700480662840_0001
name =============> HIVE-6ee69a9f-a03c-4219-bfde-5c7b0a29e567
queue =============> default
user =============> hadoop
type =============> TEZ
status =============> SUCCEEDED

ApplicationId =============> application_1699884535150_0495
name =============> HIVE-d1bdc243-613e-4979-b670-9bee0f2e800d
queue =============> default
user =============> hadoop
type =============> TEZ
status =============> SUCCEEDED

ApplicationId =============> application_1699884535150_0496
name =============> insert into hive_default v...'hive_default') (Stage-1)
queue =============> default
user =============> hadoop
type =============> MAPREDUCE
status =============> SUCCEEDED

ApplicationId =============> application_1699884535150_0486
name =============> insert into hive_openx_json values(0, ...18) (Stage-1)
queue =============> default
user =============> hadoop
type =============> MAPREDUCE
status =============> SUCCEEDED

ApplicationId =============> application_1699884535150_0487
name =============> select * from iceberg_db.trino_001_ice...ASC (Stage-1)
queue =============> default
user =============> hadoop
type =============> MAPREDUCE
status =============> SUCCEEDED
```

示例说明

```
# kerberos认证
klist -kt /var/krb5kdc/emr.keytab
kinit -kt /var/krb5kdc/emr.keytab xxxx/x.x.x.x@TBDS-XXXXXX
# 采用 Quasi-Monte Carlo 算法来估算PI的值
yarn jar /usr/local/service/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.2.2-TBDS-5.3.1.jar pi -D mapreduce.job.queuename=default 16 1000
```

api接口

java api

官方文档：Apache Hadoop Main 3.2.2 API。
常用的java api接口是YarnClient，用于Client与ResourceManager之间，常用的接口有：

```
/*
 * 提交app
 */
public abstract YarnClientApplication createApplication()
    throws YarnException, IOException;
public abstract ApplicationId submitApplication(
    ApplicationSubmissionContext appContext) throws YarnException,
    IOException;


/*
 * fail app
 */
public abstract void failApplicationAttempt(
    ApplicationAttemptId applicationAttemptId) throws YarnException,
    IOException;


/*
 * kill app
 */
public abstract void killApplication(ApplicationId applicationId) throws YarnException,
    IOException;


/*
 * 获取app信息
 */
public abstract ApplicationReport getApplicationReport(ApplicationId appId)
    throws YarnException, IOException;


/*
 * 获取app列表
 */
public abstract List<ApplicationReport> getApplications()
    throws YarnException, IOException;
public abstract List<ApplicationReport> getApplications(
```

```
    Set<String> applicationTypes) throws YarnException, IOException;
public abstract List<ApplicationReport>
  getApplications(EnumSet<YarnApplicationState> applicationStates)
      throws YarnException, IOException;
public abstract List<ApplicationReport> getApplications(
    Set<String> applicationTypes,
    EnumSet<YarnApplicationState> applicationStates) throws YarnException,
    IOException;
public abstract List<ApplicationReport> getApplications(
    Set<String> applicationTypes,
    EnumSet<YarnApplicationState> applicationStates,
    Set<String> applicationTags) throws YarnException,
    IOException;
public abstract List<ApplicationReport> getApplications(Set<String> queues,
    Set<String> users, Set<String> applicationTypes,
    EnumSet<YarnApplicationState> applicationStates) throws YarnException,
    IOException;


/*
 * 获取集群列表
 */
public abstract YarnClusterMetrics getYarnClusterMetrics() throws YarnException,
    IOException;


/*
 * 获取节点列表
 */
public abstract List<NodeReport> getNodeReports(NodeState... states)
    throws YarnException, IOException;


/*
 * 获取队列列表
 */
public abstract QueueInfo getQueueInfo(String queueName) throws YarnException,
    IOException;
public abstract List<QueueInfo> getAllQueues() throws YarnException, IOException;
```

rest api

官方文档：Apache Hadoop 3.2.2 – Hadoop YARN - Introduction to the web services REST APIs。
常用的接口有：

```
/*
 * 指标信息
 */
curl -v -k "https://${ip}:5005/ws/v1/cluster/metrics"

/*
 * app列表
 */
curl -v -k "https://${ip}:5005/ws/v1/cluster/apps?states=FAILED&queue=default"

/*
 * 统计信息
 */
curl -v -k "https://${ip}:5005/ws/v1/cluster/appstatistics?states=accepted,running,finished&application
Types=tez"

/*
 * 单个app信息
 */
curl -v -k "https://${ip}:5005/xxx/${application_id}"

/*
 * 节点信息
 */
curl -v -k "https://${ip}:5005/ws/v1/cluster/nodes"

/*
 * 集群信息
 */
curl -v -k "https://${ip}:5005/ws/v1/cluster"

/*
 * 调度信息
 */
curl -v -k "https://${ip}:5005/ws/v1/cluster/scheduler"

/*
 * 修改app状态
 */
curl -v -k -XPUT -H "Content-Type: application/json" -d '{"state":"KILLED"}'
https://${ip}:5005/ws/v1/cluster/apps/${application_id}/state

/*
 * 提交app
 */
curl -v -k -XPOST "https://${ip}:5005/ws/v1/cluster/apps/new-application"
```

```
curl -v -k -XPOST "https://${ip}:5005/ws/v1/cluster/apps" -H "Content-Type: application/json" -d '....'
```

常用命令

```
# 列出所有Application：
yarn application -list

# 根据Application状态过滤：
yarn application -list -appStates <ALL,NEW,NEW_SAVING,SUBMITTED,ACCEPTED,RUNNING,FINISHED,FAILED,KILLED>

# Kill掉Application：
yarn application -kill <Application-Id>

# 查询Application日志：
yarn logs -applicationId <ApplicationId>

# 查询Container日志：
yarn logs -applicationId <ApplicationId> -containerId <ContainerId>

# yarn node查看节点状态 列出所有节点：
yarn node -list -all

# yarn rmadmin 更新配置 加载队列配置：
yarn rmadmin -refreshQueues

# yarn queue 查看队列 打印队列信息：
yarn queue -status <QueueName>
```

常见问题

节点配置调优

1. 操作场景
   合理配置大数据集群的调度器后，还可通过调节每个节点的可用内存、CPU资源及本地磁盘的配置进行性能调优。
   具体包括以下配置项：可用内存、CPU虚拟核数、物理CPU使用百分比、内存和CPU资源的协调、本地磁盘。
2. 操作步骤

- 可用内存
  除了分配给操作系统、其他服务的内存外，剩余的资源应尽量分配给YARN。通过如下配置参数进行调整。
  例如，如果一个container默认使用512M，则内存使用的计算公式为：512M*container数。
  默认情况下，Map或Reduce container会使用1个虚拟CPU内核和1024MB内存，ApplicationMaster使用1536MB内存。

参数	描述	默认值
yarn.nodemanager.resource.memory-mb	设置可分配给容器的物理内存数量。单位：MB，取值范围大于0。 建议配置成节点物理内存总量的80%。若该节点有其他业务的常驻进程，请降低此参数值给该进程预留足够运行资源。	10240

- CPU虚拟核数
  建议将此配置设定在逻辑核数的1.5～2倍之间。如果上层计算应用对CPU的计算能力要求不高，可以配置为2倍的逻辑CPU。

参数	描述	默认值
yarn.nodemanager.resource.cpu-vcores	目前推荐将该值设置为逻辑CPU核数的1.5～2倍之间	10

- 物理CPU使用百分比
  建议预留适量的CPU给操作系统和其他进程（数据库、HBase等）外，剩余的CPU核都分配给YARN。可以通过如下配置参数进行调整。

参数	描述	默

		认值
yarn.nodemanager.resource.percentage-physical-cpu-limit	表示该节点上YARN可使用的物理CPU百分比。默认是80，即不进行CPU控制，YARN可以使用节点全部CPU。该参数只支持查看，可通过调整YARN的RES_CPUSET_PERCENTAGE参数来修改本参数值。注意，目前推荐将该值设为可供YARN集群使用的CPU百分数。	80

- 本地磁盘
  由于本地磁盘会提供给MapReduce写job执行的中间结果，数据量大。因此配置的原则是磁盘尽量多，且磁盘空间尽量大，单个达到百GB以上规模最好。简单的做法是配置和DataNode相同的磁盘，只在最下一级目录上不同即可。

参数	描述	默认值
yarn.nodemanager.log-dirs	日志存放地址	{{nm_log_dirs}}
yarn.nodemanager.local-dirs	本地化后的文件的存储位置	{{nm_local_dirs}}

任务优先级

1. 集群的资源竞争场景如下：
   提交两个低优先级的应用Job 1和Job 2。
   正在运行中的Job 1和Job 2有部分task处于running状态，但由于集群或队列资源容量有限，仍有部分task未得到资源而处于pending状态。
   提交一个较高优先级的应用Job 3，此时会出现如下资源分配情况：当Job 1和Job 2中running状态的task运行结束并释放资源后，Job 3中处于pending状态的task将优先得到这部分新释放的资源。
   Job 3完成后，资源释放给Job 1、Job 2继续执行。
   用户可以在YARN中配置任务的优先级。任务优先级是通过ResourceManager的调度器实现的。
2. 操作步骤
   设置参数“mapreduce.job.priority”，使用命令行接口或API接口设置任务优先级。
   命令行：提交任务时，添加“-Dmapreduce.job.priority=”参数。
   可以设置为：VERY_HIGH、HIGH、NORMAL、LOW、VERY_LOW。

RM处于Standby状态，无法自动恢复Active状态，该如何处理

1. 检查支持自动恢复的必选配置项是否配置正确。

参数	描述
yarn.resourcemanager.ha.enabled	需要配置为true
yarn.resourcemanager.ha.automatic-failover.enabled	需要配置为true

2. 修改为上述配置后，问题还未解决，您可以通过以下方式排查问题：

- 检查ZooKeeper服务是否正常。
- 检查ZooKeeper客户端（RM）读取数据是否超出其Buffer上限。
  - 问题现象：RM日志内存在异常，提示Zookeeper error len*** is out of range!或Unreasonable length = ***。
  - 处理方法：在TM服务管理的YARN服务的配置页面，单击yarn-env页签，更新参数yarn_resourcemanager_opts的参数值为-Djute.maxbuffer=值，此处的值需与zookeeper服务的zookeeper-env中的Djute.maxbuffer参数值保持一致，然后重启RM。
- ZooKeeper服务端写入数据是否超出其Buffer上限。
  - 问题现象：ZooKeeper日志内存在异常，提示Exception causing close of session 0x1000004d5701b6a: Len error ***。
  - 处理方法：ZooKeeper服务各节点新增或更新配置项-Djute.maxbuffer= (单位：bytes) ，将Buffer上限调大超过异常显示的长度。

NM组件OOM如何处理

1. Java heap space或GC overhead limit exceeded或频繁FullGC。

- 直接原因：JVM堆空间不足，NM进程内部对象无法获取到足够的资源，并且在抛出OOM之前已经进行过一轮或多轮Full GC回收了失效的对象，仍无法获取足够的堆空间用于分配。
- 原因分析：NM进程中的常驻对象不多，一般只包含当前节点、运行应用、执行任务容器（Container）等基本信息，这部分占用空间不会很大。可能占用空间较大的是External Shuffle Service的Cache和Buffer，这部分受Shuffle Service相关配置（例如Spark：spark.shuffle.service.index.cache.size或spark.shuffle.file.buffer，MapReduce：mapreduce.shuffle.ssl.file.buffer.size或mapreduce.shuffle.transfer.buffer.size等）影响，并且与使用了Shuffle Service的运行应用数（或执行任务容器数）成正比。这些信息占用的堆空间随着使用Shuffle Service应用数或任务容器数的增大而增大，因此对于规格较大可运行任务较多的节点需要保证NM进程的内存配置也较大（建议最小不低于1 GB）。
- 解决方案：如果节点资源足够，建议适当增大NM堆内存（yarn-env.sh配置项YARN_NODEMANAGER_HEAPSIZE）。检查Shuffle Service相关配置是否合理，例如Spark Cache配置不应该占用大部分堆内存。

2. Direct buffer memory

- 直接原因：堆外内存溢出导致的OOM，通常与External Shuffle Service有关，例如有的Shuffle Service服务的RPC调用使用了NIO DirectByteBuffer，就会用到堆外内存。
- 原因分析：堆外内存跟使用ShuffleService的应用数或任务容器数成正比。对于使用Shuffle Service的任务较多的节点，需要确认NM进程的堆外内存配置是否过小。
- 解决方案：检查堆外内存配置-XX:MaxDirectMemorySize（yarn-env.sh配置项YARN_NODEMANAGER_OPTS）是否合理。无此配置时默认与堆内存大小相同，不合理时适当调大堆外内存空间。

权限策略配置

配置步骤

1. 安装 Ranger Plugin：

   i. 下载并安装 Ranger YARN 插件。

2. 配置 YARN：

   i. 修改 yarn-site.xml 文件，添加以下配置：

```
<property>
  <name>yarn.resourcemanager.acl.enable</name>
  <value>true</value>
</property>
<property>
  <name>yarn.resourcemanager.authorization-provider</name>
  <value>org.apache.ranger.authorization.yarn.authorizer.RangerYarnAuthorizer</value>
</property>
```

3. 配置 Ranger：

   i. 登录TBDS Manager界面，在经典集群模块选择hadoop集群进入集群详情页，选择“集群服务->Yarn”。

   ii. 进入“配置管理”页，搜索参数“yarn.acl.enable”，修改参数值为“true”。如果该参数值已经为“true”，则无需处理。

   iii. 使用Ranger管理员用户rangeradmin登录Ranger管理页面。

   iv. 在首页中单击“YARN”区域的组件插件名称，例如“yarn”。

   v. 单击“Add New Policy”，添加Yarn权限控制策略。

   vi. 根据业务需求配置相关参数。

   Yarn权限参数如下：

参数名称	描述
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。

Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持"*"通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。
Policy Label	为当前策略指定一个标签，可以根据这些标签搜索报告和筛选策略。
Queue	队列名称，支持通配符"*"。 如需子队列继承上级队列权限，可打开递归开关按钮。 ● Non-recursive：关闭递归 ● Recursive：打开递归
Description	策略描述信息。
Audit Logging	是否审计此策略。
Allow Conditions	策略允许条件，配置本策略内允许的权限及例外。 在"Select Role"、"Select Group"、"Select User"列选择已创建好的需要授予权限的Role、用户组或用户，单击"Add Conditions"，添加策略适用的IP地址范围，单击"Add Permissions"，添加对应权限。 ● submit-app：提交队列任务权限 ● admin-queue：管理队列任务权限 ● Select/Deselect All：全选/取消全选 如需让当前条件中的用户或用户组管理本条策略，可勾选"Delegate Admin"使这些用户成为受委托的管理员。被委托的管理员可以更新、删除本策略，它还可以基于原始策略创建子策略。 如需添加多条权限控制规则，可单击 按钮添加。如需删除权限控制规则，可单击 按钮删除。 Exclude from Allow Conditions：配置策略例外条件。
Deny All Other Accesses	是否拒绝其他所有访问。 ● True：拒绝其他所有访问。 ● False：设置为False，可配置Deny Conditions。
Deny Conditions	策略拒绝条件，配置本策略内拒绝的权限及例外，配置方法与"Allow Conditions"类似。拒绝条件的优先级高于"Allow Conditions"中配置的允许条件。 Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

设置权限场景如下：

任务场景	角色授权操作
设置Yarn管理员权限	1. 在首页中单击"YARN"区域的组件插件名称，例如"Yarn"。 2. 选择"Policy Name"为"all - queue"的策略，单击 按钮编辑策略。 3. 在"Allow Conditions"区域，单击"Select User"下选择框选择用户。
设置用户在指定Yarn队列提交任务的权限	1. 在"Queue"配置队列名。 2. 在"Allow Conditions"区域，单击"Select User"下选择框选择用户。

	3. 单击“Add Permissions”，勾选“submit-app”。
设置用户在指定Yarn队列管理任务的权限	1. 在“Queue”配置队列名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“admin-queue”。

7. （可选）添加策略有效期。在页面右上角单击“Add Validity period”，设置“Start Time”和“End Time”，选择“Time Zone”。单击“Save”保存。如需添加多条策略有效期，可单击 按钮添加。如需删除策略有效期，可单击 按钮删除。
8. 单击“Add”，在策略列表可查看策略的基本信息。等待策略生效后，验证相关权限是否正常。如需禁用某条策略，可单击 按钮编辑策略，设置策略开关为“Disabled”。

如果不再使用策略，可单击 按钮删除策略。

HBase开发
概述

HBase 是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，是 Google BigTable 的开源实现。HBase 利用 Hadoop HDFS 作为其文件存储系统；Hadoop MapReduce 来处理 HBase 中的海量数据；ZooKeeper 来做协同服务。

HBase 主要由 ZooKeeper、HMaster 和 HRegionServer 组成。其中：

1. ZooKeeper 可避免 HMaster 的单点故障，其 Master 选举机制可保证一个 Master 提供服务。
2. HMaster 管理用户对表的增删改查操作，管理 HRegionServer 的负载均衡。并可调整 Region 的分布，在 HRegionServer 退出时迁移其内的 HRegion 到其他 HRegionServer 上。
3. HRegionServer 是 HBase 中最核心的模块，其主要负责响应用户的 I/O 请求，向 HDFS 文件系统中读写数据。HRegionServer 内部管理了一系列 HRegion 对象，每个 HRegion 对应一个 Region，HRegion 中由多个 Store 组成。每个 Store 对应了 Column Family 的存储。

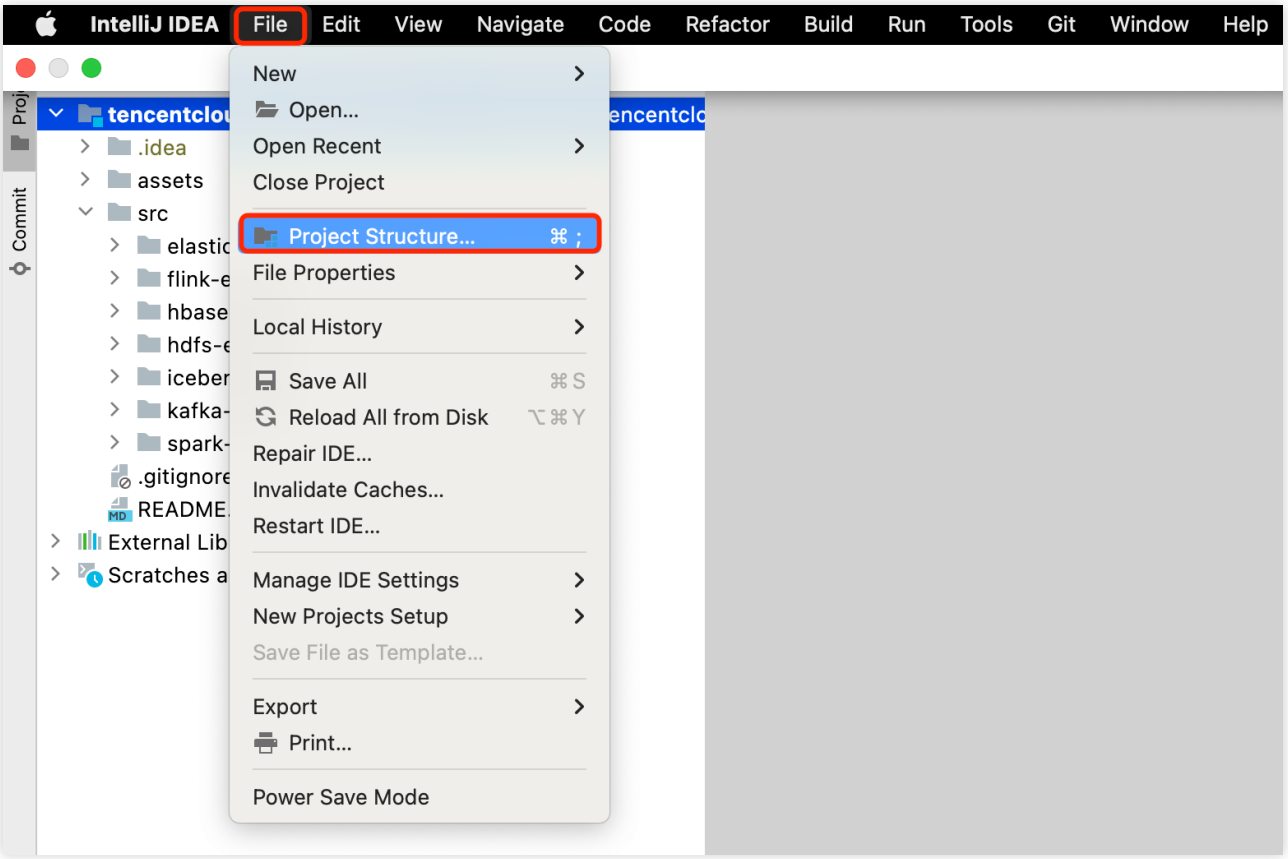
示例工程开发

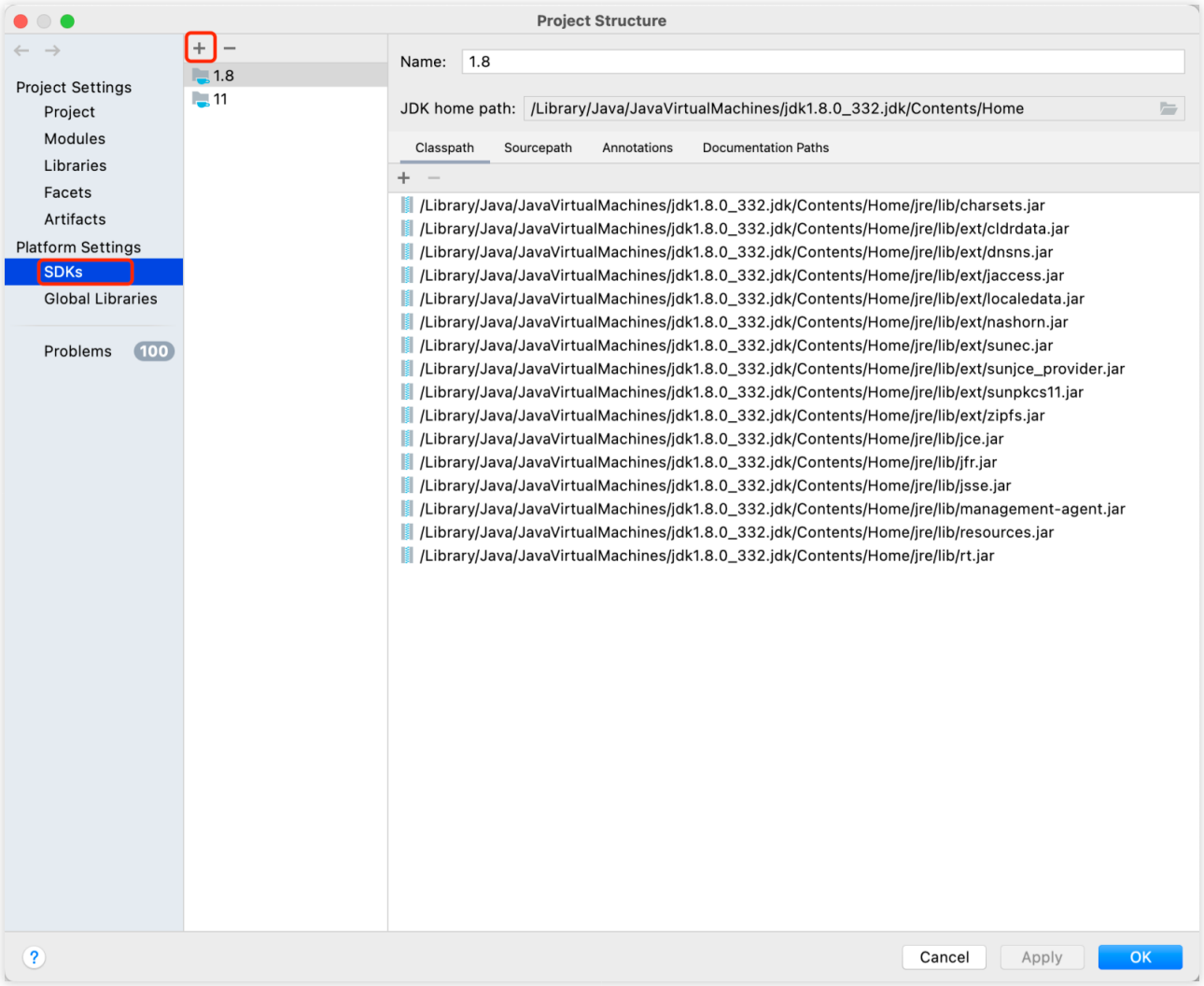
环境准备

1. 开发环境准备（Java/Scala）

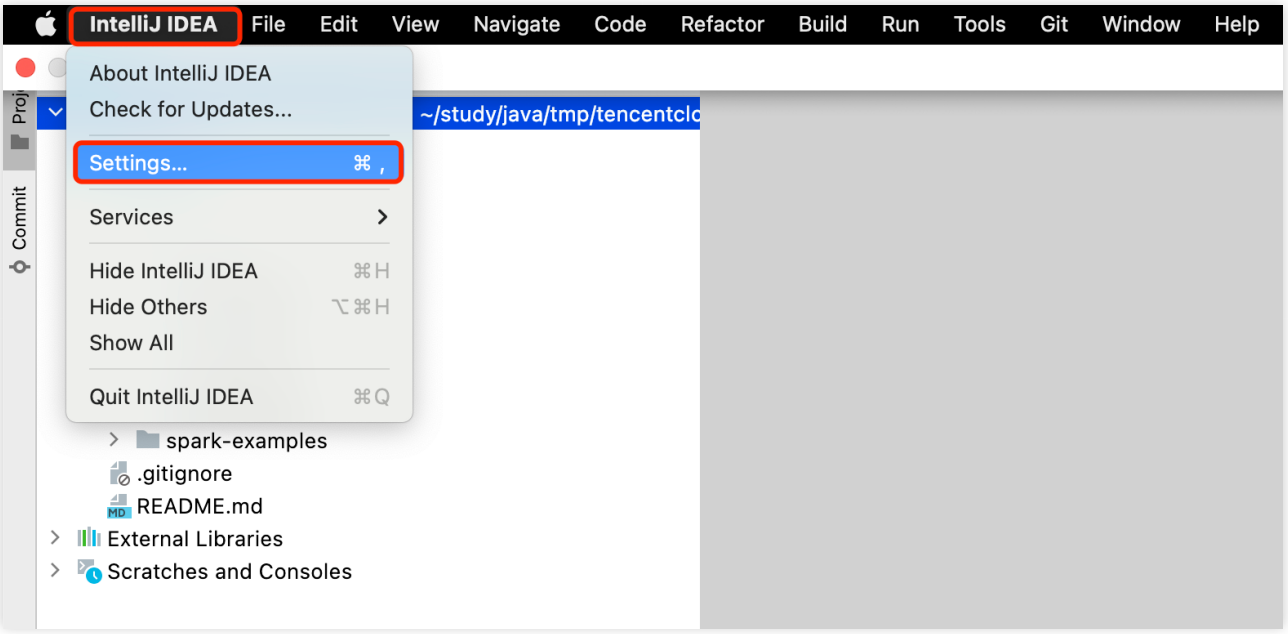
准备项	说明
安装JDK	JDK 8 ，推荐使用 konaJDK，下载地址
安装 Scala	Scala 2.12，下载地址
安装和配置 IDE	按需选择，比如 IntelliJ IDEA 或 Eclipse
安装 Maven	开发环境基础配置，负责构建 Java 应用程序
Maven 配置准备	如果需要本地调试，需要参考开发环境准备，配置 Maven pom.xml，推荐 Maven 3.6.3，下载地址

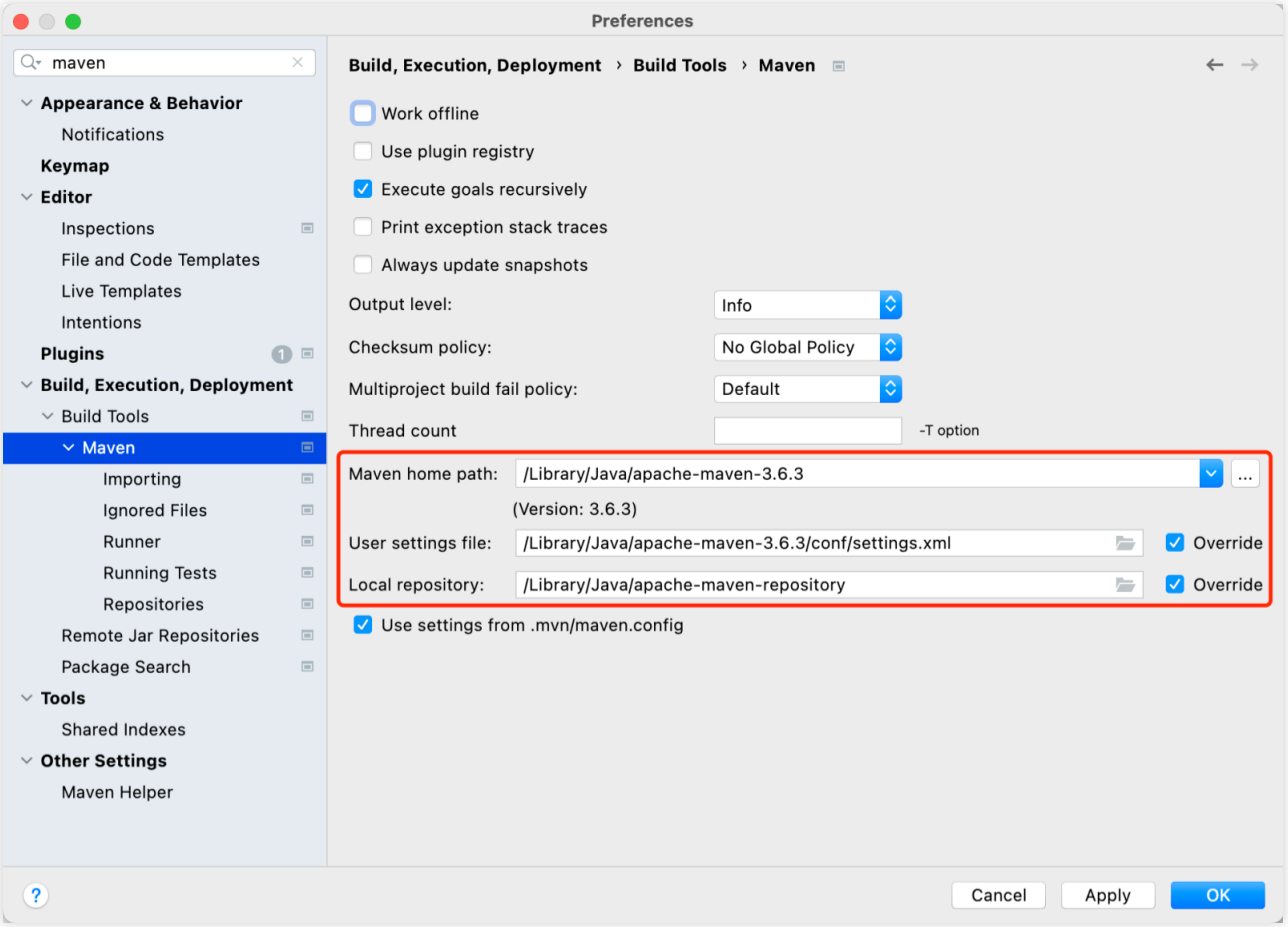
2. 远程调试环境准备
   以下使用 Linux 环境作为开发机进行应用调试说明。
   i. 准备开发机（可选）：建议使用 Linux 操作系统
   ii. 部署客户端：参考部署TBDS客户端在开发机上执行客户端部署
3. 导入示例工程代码
   以下以 IntelliJ IDEA 举例，将示例工程代码导入进行说明。
   i. 下载样例代码：git clone https://e.coding.net/g-necm8077/tencentcloud-tbds-examples/tbds-examples.git
   ii. 导入项目：安装完 IntelliJ IDEA 和 JDK 工具后，导入样例工程到 IntelliJ IDEA 开发环境。点击 Open 后，选择上一步下载的项目地址打开。
   iii. IDEA 配置 JDK：首先确保在本地安装了 JDK 1.8，并配置好了环境变量。IDEA 选择 File 下的 Project Structure，点击 SDKs，选择 JDK 1.8，点击 Apply，再点击 OK。若没有 JDK 1.8，则点击 + 号进行添加，点击 Add JDK 后选择 JDK 1.8 安装目录，然后点击 OK 即可。


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
New
Open...
Open Recent
Close Project
Project Structure... ⌘ ;
File Properties
Local History
Save All ⌘S
Reload All from Disk ⌥⌘Y
Repair IDE...
Invalidate Caches...
Restart IDE...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
Commit
.idea
assets
src
External Lib
Scratches a



iv. IDEA 配置 Maven：首先确保在本地安装了 Maven，并配置好了环境变量和 settings.xml 文件。IDEA 点击 Settings 进入配置页面，左上角输入 maven 进行搜索，点击 Build Tools 下的 Maven 配置项，修改 “Maven home path” 为本地 Maven 的安装目录，修改 “User settings file” 为本地 Maven settings.xml 配置文件的文件路径，并勾选 Override，此时 “Local repository” 将自动设置为 settings.xml 文件中配置的本地 Maven 仓库的目录。最后点击 Apply，再点击 OK。





代码逻辑说明

1. 功能说明

   本教程演示的是：HBase创建集群连接、创建表、读写、删除表的基本功能操作示例。

2. 代码逻辑说明

初始化HBase Configuration，这里会从java程序的classpath中尝试加载hbase-site.xml，推荐使用这种方式初始化：

```
conf = HBaseConfiguration.create();
```

如果不方便在classpath设置hbase-site.xml的路径，可以手动在代码中设置。

```
conf.addResource(new Path( "/etc/hbase/conf/hbase-site.xml"));
```

用户认证相关初始化，从conf中加载认证类型：hadoop.security.authentication和hbase.security.authentication

```
UserGroupInformation.setConfiguration(conf);
```

创建HBase集群连接。

```
conn = ConnectionFactory.createConnection(conf);
```

创建表： 先通过connection拿到HAdmin对象，再调用其createTable方法。

```
public void createTable() throws IOException {
    // Specify the table descriptor.
    TableDescriptorBuilder tableBuilder = TableDescriptorBuilder.newBuilder(tableName);

    // Set the column family name to info
    ColumnFamilyDescriptorBuilder cfBuilder = ColumnFamilyDescriptorBuilder.newBuilder(Bytes.toBytes("info"));

    // Set data encoding methods. HBase provides DIFF,FAST_DIFF,PREFIX
    cfBuilder.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);

    // Set compression methods
    cfBuilder.setCompressionType(Compression.Algorithm.SNAPPY);

    ColumnFamilyDescriptor cf = cfBuilder.build();

    tableBuilder.setColumnFamily(cf);

    Admin admin = this.conn.getAdmin();

    admin.createTable(tableBuilder.build());
}
```

写入数据：先通过connection获得HTable对象，再调用其put方法。 参数是一个Put对象需要通过组装获取。

```
    Table table = conn.getTable(tableName);
    table.put(puts);
```

组装Put对象

```
private static void putData(List<Put> puts, String rowKey, List<String> data) {
    Put put = new Put(Bytes.toBytes(rowKey));
```

```
        put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"), Bytes.toBytes(data.get(0)));
        put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("age"), Bytes.toBytes(data.get(1)));
        put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("gender"), Bytes.toBytes(data.get(2)));
        put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("score"), Bytes.toBytes(data.get(3)));
        puts.add(put);
    }
```

读取数据：先通过connection获得HTable对象，再调用其get方法。

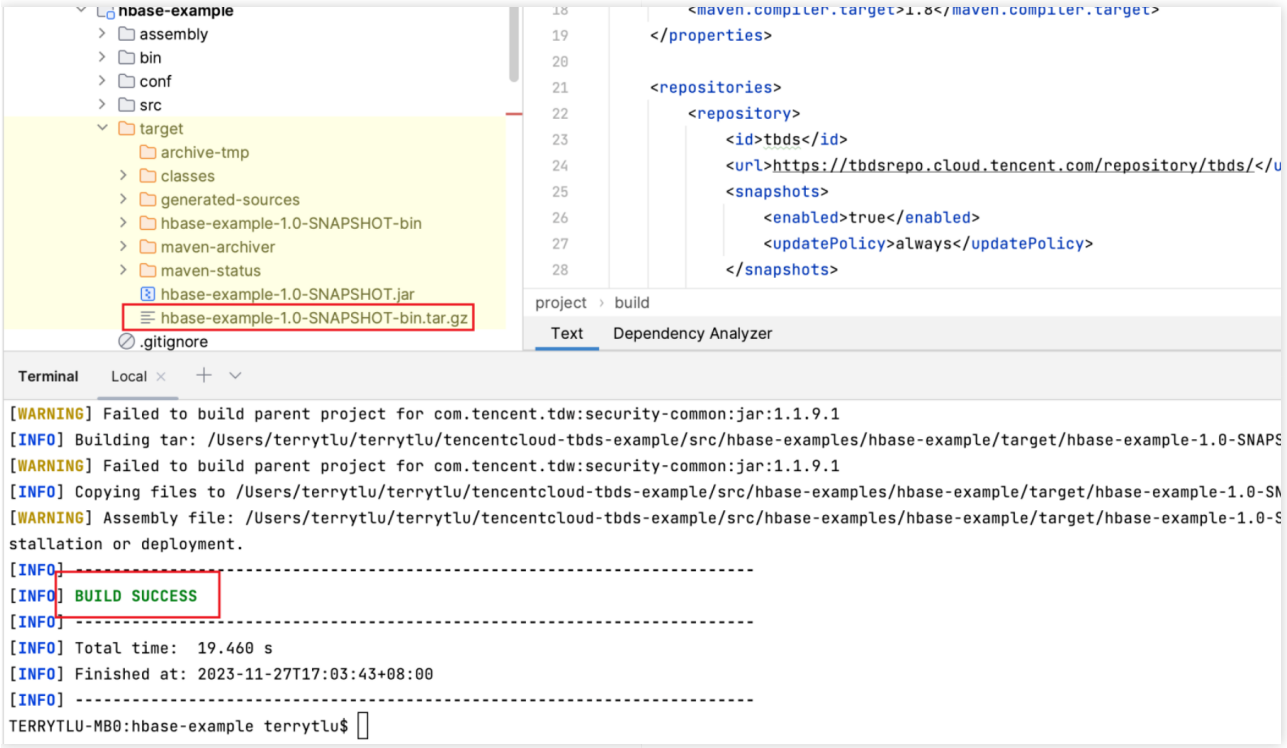
```
        Get get = new Get(Bytes.toBytes("row1"));

        // Submit a get request.
        Table table = conn.getTable(tableName);
        Result result = table.get(get);
              // Print query results.
        for (Cell cell : result.rawCells()) {
            LOG.info("{}:{},{},{}", Bytes.toString(CellUtil.cloneRow(cell)),
                    Bytes.toString(CellUtil.cloneFamily(cell)), Bytes.toString(CellUtil.cloneQualifier(cell)),
                    Bytes.toString(CellUtil.cloneValue(cell)));
        }
```

打包发布

1. 打包
   以下使用 IntelliJ IDEA 说明示例工程代码编译过程。点击 IDEA 下方 Terminal 打开终端，切换到示例工程的 HBase 工程目录下，然后使用命令 mvn clean package 对工程进行打包，运行过程中可能还需要下载一些文件，直到出现 BUILD SUCCESS 表示打包成功。

```
cd src/hbase-examples/hbase-example/
mvn clean package
```

通过上述编译打包后，将在工程目录下 target 文件夹中看到打好的 tar.gz 包，如图示中的 hbase-example-1.0-SNAPSHOT-bin.tar.gz，这里面包含了样例程序以及运行过程中所需的所有依赖，大小在60MB左右。



2. 开发机运行

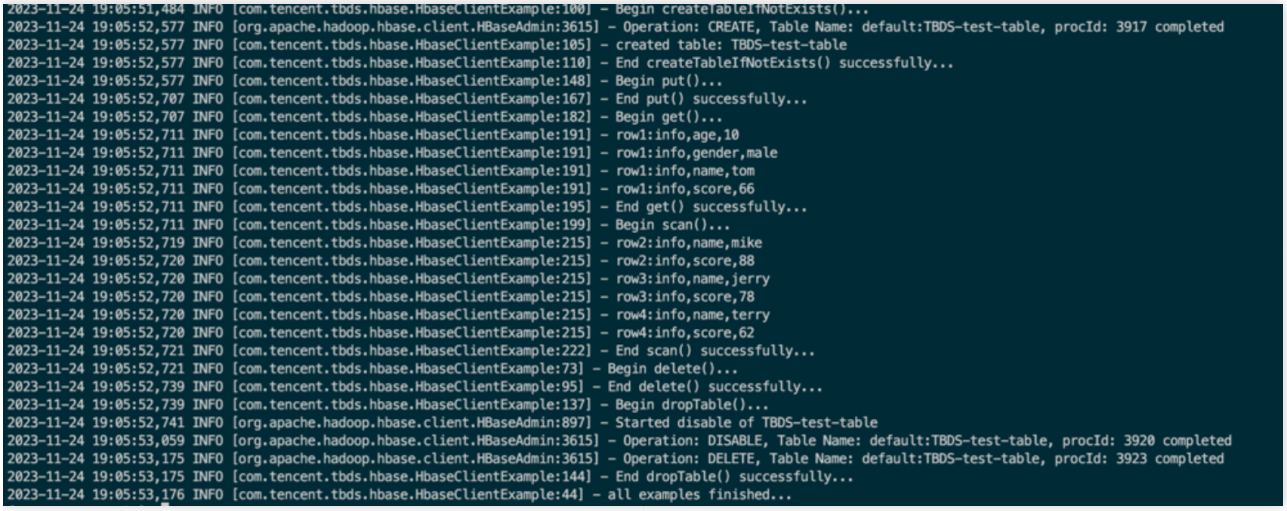
1. 准备用户：参考获取用户认证 获取认证信息。若是 Kerberos 环境，需要将用户的 keytab 文件下载到本地，然后上传至开发机。这里将 test 用户的 test.keytab 文件上传至开发机的 /tmp 目录。
2. 将集群的配置文件hbase-site.xml放在解压后的conf目录下，如果是集群内的节点，直接执行：cp /usr/local/service/hbase/conf/hbase-site.xml conf/ 即可。
3. 安全认证：将principal和keytab信息放入hbase-site.xml中，增加这3个对应配置，hbase客户端会读取对应配置自动完成认证处理：

```
 <property>
  <name>hadoop.security.authentication</name>
  <value>kerberos</value>
</property>
<property>
  <name>hbase.client.keytab.principal</name>
  <value>test@TBDS-XXXXX</value>
</property>
<property>
  <name>hbase.client.keytab.file</name>
  <value>/tmp/test.keytab</value>
</property>
```

4. Ranger 授权：参考Ranger授权（经典集群），确保 test 用户具有创建/写入对应表的权限。
5. 运行样例
    i. 设置要运行的类： export MAIN_CLASS=com.tencent.tbds.hbase.HBaseClientExample
    ii. 执行测试: sh start.sh

程序运维监控

控制台以及logs/client.log都会打印运行期间的日志。



示例说明

通过如下命令您可以进入 HBase Shell：

```
[hadoop@10hbase]$ hbase shell
```

常用操作

通用命令

- status: 提供HBase的状态，例如，服务器的数量
- version: 提供正在使用HBase版本
- table_help: 表引用命令提供帮助
- whoami: 提供有关用户的信息

数据定义语言

- create: 创建一个表
- list: 列出HBase的所有表
- disable: 禁用表
- is_disabled: 验证表是否被禁用
- enable: 启用一个表
- is_enabled: 验证表是否已启用
- describe: 提供了一个表的描述
- alter: 修改一个表
- exists: 验证表是否存在
- drop: 从HBase中删除表

数据操纵语言

- put: 把指定列放入指定的行中单元格的值在一个特定的表
- get: 获取行或单元格的内容
- delete: 删除表中的单元格值
- deleteall: 删除给定行的所有单元格
- scan: 扫描并返回表数据
- count: 计数并返回表中的行的数目
- truncate: 清空一个指定的表的数据
- truncate_preserve

示例

1. 在 hbase shell 下输入 help 可以查看基本的使用信息和示例的指令。接下来我们使用以下指令建立一个新表：

```
hbase(main):001:0> create 'test', 'cf'
```

2. 表格建立后，可以使用 list指令来查看您建立的表是否已经存在：

```
hbase(main):002:0> list 'test'
TABLE
test
1 row(s) in 0.0030 seconds

=> ["test"]
```

3. 使用put指令来为您创建的表添加元素：

```
hbase(main):003:0> put 'test', 'row1', 'cf:a', 'value1'
0 row(s) in 0.0850 seconds

hbase(main):004:0> put 'test', 'row2', 'cf:b', 'value2'
0 row(s) in 0.0110 seconds

hbase(main):005:0> put 'test', 'row3', 'cf:c', 'value3'
0 row(s) in 0.0100 seconds
```

4. 使用scan指令来遍历整个表：

```
hbase(main):006:0> scan 'test'
ROW  COLUMN+CELL
row1   column=cf:a, timestamp=1530276759697, value=value1
row2   column=cf:b, timestamp=1530276777806, value=value2
row3   column=cf:c, timestamp=1530276792839, value=value3
3 row(s) in 0.2110 seconds
```

5. 使用get指令来获取表中指定行的值：

```
hbase(main):007:0> get 'test', 'row1'
COLUMN  CELL
 cf:a       timestamp=1530276759697, value=value
1 row(s) in 0.0790 seconds
```

6. 使用drop指令来删除一个表，在删除表之前需要先使用disable指令来禁用一个表：

```
hbase(main):010:0> disable 'test'
hbase(main):011:0> drop 'test'
```

api接口

TBDS 大数据平台上的 HBase 访问接口与开源兼容，可以参考 Apache HBase 2.4.0 API。

常用命令

操作	命令	描述
进入hbase shell客户端	hbase shell	hbase shell中用户可以交互式访问/操作HBase表
检测HBase集群元数据一致性	hbase hbck	扫描所有表的状态和对应HDFS目录、zookeeper上的状态进行对比，得出集群元数据是否一致。
检测HBase集群读写链路是否正常	hbase canary -writeSniffing hbase:canary	测试读写hbase:canary表，来健康检测所有RegionServer的读写链路是否正常

HBase shell 常用命令

操作	命令	描述
帮助	help	显示HBase shell的帮助信息。
列出所有表	list	列出当前HBase中的所有表。
创建表	create 'table_name', 'cf'	创建一个名为table_name的表，并包含一个列族cf。
描述表	describe 'table_name'	显示表table_name的结构信息。
禁用表	disable 'table_name'	禁用表table_name，在删除或修改表结构之前需要先禁用表。
启用表	enable 'table_name'	启用表table_name。
删除表	drop 'table_name'	删除表table_name，表必须先被禁用。
插入数据	put 'table_name', 'row', 'cf:col', 'value'	向表table_name的row行的cf:col列插入值value。
获取数据	get 'table_name', 'row'	获取表table_name中row行的数据。
扫描表	scan 'table_name'	扫描表table_name中的所有数据。
删除数据	delete 'table_name', 'row', 'cf:col'	删除表table_name中row行的cf:col列的数据。
计数行数	count 'table_name'	计算表table_name中的行数。

操作	命令	描述
修改表	alter 'table_name', {NAME => 'cf', VERSIONS => 5}	修改表table_name的列族cf的属性，例如设置版本数为5。
显示表状态	is_enabled 'table_name'	检查表table_name是否启用。
显示表状态	is_disabled 'table_name'	检查表table_name是否禁用。
列出命名空间	list_namespace	列出所有命名空间。
创建命名空间	create_namespace 'namespace'	创建一个新的命名空间namespace。
删除命名空间	drop_namespace 'namespace'	删除命名空间namespace，命名空间必须为空。
列出命名空间中的表	list_namespace_tables 'namespace'	列出命名空间namespace中的所有表。
退出HBase shell	exit	退出HBase shell。

权限策略配置

配置步骤

1. 安装 Ranger Plugin：

   i. 下载并安装 Ranger HBase插件。

2. 配置HBase：

   i. 修改hbase-site.xml 文件，添加以下配置：

```
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.ranger.authorization.hbase.RangerAuthorizationCoprocessor</value>
</property>
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>org.apache.ranger.authorization.hbase.RangerAuthorizationCoprocessor</value>
</property>
```

3. 配置 Ranger：

   i. 使用Ranger管理员用户rangeradmin登录Ranger管理页面。

   ii. 在首页中单击“HBASE”区域的组件插件名称，例如“HBase”。

   iii. 单击“Add New Policy”，添加HBase权限控制策略。

   iv. 根据业务需求配置相关参数。

   HBase权限参数如下：

参数名称	描述
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。
Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持“*”通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。
Policy Label	为当前策略指定一个标签，可以根据这些标签搜索报告和筛选策略。

HBase Table	将适用该策略的表。 可支持通配符“*”，例如“table1:*”表示table1下的所有表。 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。 说明： Ranger界面上HBase服务插件的“hbase.rpc.protection”参数值必须和HBase服务端的“hbase.rpc.protection”参数值保持一致。
HBase Column-family	将适用该策略的列族。 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
HBase Column	将适用该策略的列。 “Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
Description	策略描述信息。
Audit Logging	是否审计此策略。
Allow Conditions	策略允许条件，配置本策略内允许的权限及例外。 在“Select Role”、“Select Group”、“Select User”列选择已创建好的需要授予权限的Role、用户组或用户，单击“Add Conditions”，添加策略适用的IP地址范围，单击“Add Permissions”，添加对应权限。 ● Read：读权限 ● Write：写权限 ● Create：创建权限 ● Admin：管理权限 ● Select/Deselect All：全选/取消全选 如需让当前条件中的用户或用户组管理本条策略，可勾选“Delegate Admin”使这些用户或用户组成为受委托的管理员。被委托的管理员可以更新、删除本策略，还可以基于原始策略创建子策略。 如需添加多条权限控制规则，可单击 按钮添加。如需删除权限控制规则， 可单击 按钮删除。 Exclude from Allow Conditions：配置策略例外条件。
Deny All Other Accesses	是否拒绝其他所有访问。 ● True：拒绝其他所有访问 False：设置为False，可配置Deny Conditions。
Deny Conditions	策略拒绝条件，配置本策略内拒绝的权限及例外，配置方法与“Allow Conditions”类似。 拒绝条件的优先级高于“Allow Conditions”中配置的允许条件。

	Exclude from Deny Conditions：配置排除在拒绝条件之外的例外规则。

设置权限场景如下：

任务场景	角色授权操作
设置HBase管理员权限	1. 在首页中单击“HBase”区域的组件插件名称，例如“HBase”。 2. 选择“Policy Name”为“all - table, column-family, column”的策略，单击 按钮编辑策略。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。
设置用户创建表的权限	1. 在“HBase Table”配置表名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Create”。 4. 该用户具有以下操作权限：create table drop table truncate table alter table enable table flush table flush region compact disable enable desc
设置用户写入数据的权限	1. 在“HBase Table”配置表名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Write”。 4. 该用户具有put，delete，append，incr等操作权限。
设置用户读取数据的权限	1. 在“HBase Table”配置表名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Read”。 4. 该用户具有get，scan操作权限。
设置用户管理命名空间或表的权限	1. 在“HBase Table”配置表名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Admin”。

	4. 该用户具有rsgroup，peer，assign，balance等操作权限。
设置列的读取或写入权限	1. 在“HBase Table”配置表名。 2. 在“HBase Column-family”配置列族名。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Read”或者“Write”。

> **说明：**
>
> 如果用户在hbase shell中执行desc操作，需要同时给该用户赋予hbase:quota表的读权限。
>

5. （可选）添加策略有效期。在页面右上角单击“Add Validity period”，设置“Start Time”和“End Time”，选择“Time Zone”。单击“Save”保存。如需添加多条策略有效期，可单击 按钮添加。如需删除策略有效期，可单击 按钮删除。

6. 单击“Add”，在策略列表可查看策略的基本信息。等待策略生效后，验证相关权限是否正常。

   如需禁用某条策略，可单击 按钮编辑策略，设置策略开关为“Disabled”。

   如果不再使用策略，可单击 按钮删除策略。

7. 重启 HBase：

   i. 重启 HBase 服务以应用配置。

开发规范

总则

概述

为规范TBDS HBase数据库开发工作，特制定本《TBDS HBase数据库开发规范》。

适用范围

本规范适用于 TBDS HBase 组件的开发。

基础开发指南

HBase介绍

TBDS HBase是一个分布式的、面向列的开源数据库。HBase不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库。另一个不同的是HBase基于列的而不是基于行的模式。
TBDS HBase – Hadoop Database，是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用HBase技术可在廉价PC Server上搭建起大规模结构化存储集群。HBase是Google Bigtable的开源实现，类似Google Bigtable利用GFS作为其文件存储系统，HBase利用Hadoop HDFS作为其文件存储系统；Google运行MapReduce来处理Bigtable中的海量数据，HBase同样利用Hadoop MapReduce来处理HBase中的海量数据；Google Bigtable利用 Chubby作为协同服务，HBase利用Zookeeper作为对应。其提供多种的数据访问方式：
Native Java API：最常规和高效的访问方式，适合Hadoop MapReduce Job并行批处理HBase表数据。
HBase Shell：HBase的命令行工具，最简单的接口，适合HBase管理使用。
Thrift Gateway：利用Thrift序列化技术，支持C++，PHP，Python等多种语言，适合其他异构系统在线访问HBase表数据。
REST Gateway：支持REST 风格的Http API访问HBase, 解除了语言限制。
Pig：可以使用Pig Latin流式编程语言来操作HBase中的数据，和Hive类似，本质最终也是编译成MapReduce Job来处理HBase表数据，适合做数据统计。

命名规范

数据库命名规范

采用26个英文字母(区分大小写)和0-9的自然数(经常不需要)加上下划线组成，命名简洁明确，多个单词用下划线'_'分隔，一个业务项目一个数据库，多个业务项目慎用同一个数据库。

数据库表命名规范

数据表命名规范

（1）采用26个英文字母(区分大小写)和0-9的自然数(经常不需要)加上下划线'*'组成，命名简洁明确，多个单词用下划线*"分隔。

（2）全部小写命名，禁止出现大写。

（3）禁止使用数据库关键字，如：name，time，datetime，password等。

（4）表名称不应该取得太长（一般不超过三个英文单词）。

（5）表的名称一般使用名词或者动宾短语。

（6）用单数形式表示名称，例如，使用employee，而不是employees。

（7）明细表的名称为：主表的名称+字符dtl（detail缩写）例如：采购订单的名称为：po_order，则采购订单的明细表为：po_orderdtl。

（8）表必须填写描述信息。

字段命名规范

（1）采用26个英文字母(区分大小写)和0-9的自然数(经常不需要)加上下划线'*'组成，命名简洁明确，多个单词用下划线*"分隔。

（2）全部小写命名，禁止出现大写。

（3）字段必须填写描述信息。

（4）禁止使用数据库关键字，如：name，time，datetime, password等。

（5）字段名称一般采用名词或动宾短语。

（6）采用字段的名称必须是易于理解，一般不超过三个英文单词。

（7）在命名表的列时，不要重复表的名称。

（8）不要在列的名称中包含数据类型。

（9）字段命名使用完整名称，禁止缩写。

HBase表设计规范

命名空间

采用英文单词、阿拉伯数字的组合形式，其中，单词必须大写，并且首字符必须为英文字符，不能是数字。不建议用连接符（下划线）拼接多个单词，简单语义的可采用单个单词，复杂语义的可采用多个单词的首字母拼接。长度尽量限制在4~8字符之间。命名空间一般可与项目名称、组织机构名称等保持一致。根据项目名称构建命名空间：DLQX

（电力气象首字母拼接形式），简短明了。不建议过长的命名空间名称，譬如不推荐采用以下形式：USER_INFO_MANAGE等。

表名称

采用英文单词、阿拉伯数字、连接符（_）的组合形式，其中，单词必须大写，并且首字符必须为英文字符，不能是数字，可用连接符拼接多个单词。长度尽量限制在8~16字符之间。尽量采用具有明确意义的英文单词，而不建议采用汉字的拼音字母或者拼音首字母组合。符合规范的表名称：USER_INFO_MANAGE、WEATHER_DATA、T_ELECTRIC_GATHER等。

列族名称

采用英文单词、阿拉伯数字的组合形式，其中，单词必须大写，并且首字符必须为英文字符，不能是数字。长度尽量限制在1~6字符之间，过长的列族名称将占用更多的存储空间。符合规范的列族名称：D1、D2、DATA等。不推荐的列族名称：USER_INFO、D_1等。

列名称

采用英文单词、阿拉伯数字、连接符（_）的组合形式，其中，单词必须大写，并且首字符必须为英文字符，不能是数字，可用连接符拼接多个单词。长度尽量限制在1~16字符之间。尽量采用具有明确意义的英文单词，而不建议采用汉字的拼音字母或者拼音首字母组合。符合规范的列名称：USER_ID、DATA_1、REMARK等。不推荐的列名称：UserID、1_DATA等。

内部存储机制

HBase中的所有数据文件都存储在Hadoop HDFS文件系统上，主要包括上述提出的两种文件类型：
HFile， HBase中KeyValue数据的存储格式，HFile是Hadoop的二进制格式文件，实际上StoreFile就是对HFile做了轻量级包装，即StoreFile底层就是HFile。
HLog File，HBase中WAL（Write Ahead Log）的存储格式，物理上是Hadoop的Sequence File。

HFile

首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。正如图中所示的，Trailer中有指针指向其他数据块的起始点。File Info中记录了文件的一些Meta信息，例如：AVG_KEY_LEN, AVG_VALUE_LEN, LAST_KEY, COMPARATOR, MAX_SEQ_ID_KEY等。Data Index和Meta Index块记录了每个Data块和Meta块的起始点。
Data Block是HBase I/O的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定，大号的Block有利于顺序Scan，小号Block利于随机查询。每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成, Magic内容就是一些随机数字，目的是防止数据损坏。后面会详细介绍每个KeyValue对的内部构造。
HFile里面的每个KeyValue对就是一个简单的byte数组。但是这个byte数组里面包含了很多项，并且有固定的结构。开始是两个固定长度的数值，分别表示Key的长度和Value的长度。紧接着是Key，开始是固定长度的数值，表示

RowKey的长度，紧接着是RowKey，然后是固定长度的数值，表示Family的长度，然后是Family，接着是Qualifier，然后是两个固定长度的数值，表示Time Stamp和Key Type（Put/Delete）。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。

HLogFile

HLog文件的结构，其实HLog文件就是一个普通的Hadoop Sequence File，Sequence File 的Key是HLogKey对象，HLogKey中记录了写入数据的归属信息，除了table和region名字外，同时还包括 sequence number和timestamp，timestamp是"写入时间"，sequence number的起始值为0，或者是最近一次存入文件系统中sequence number。
HLog Sequence File的Value是HBase的KeyValue对象，即对应HFile中的KeyValue。

HBase Phoenix SQL语法

Phoenix将HBase的数据模型映射到关系型世界，支持所有标准SQL查询构造，包括SELECT，FROM，WHERE，GROUP BY，HAVING，ORDER BY等。
它还支持一整套DML命令以及通过DDL命令创建表和版本化增量更改。Phoenix的目标：通过定义明确的行业标准API，成为Hadoop的OLTP和运营分析的可信数据平台。

数据类型

Apache Phoenix SQL支持如下这些数据类型：
整数：integer
小数：float, double, decimal, decimal(10,2)
字符串：char(10), varchar, varchar(255)，值必须用单引号，不能用双引号。
布尔：boolean
时间：time, date, timestamp。
字节数组：binary, varbinary。
数组：varchar array, char(10) array[5], integer[] integer[100]

Phoenix SQL语法

DDL支持：CREATE TABLE，DROP TABLE，ALTER TABLE。
DML支持：UPSERT VALUES用于逐行插入，UPSERT SELECT用于在相同或不同的表之间传输大量数据，DELETE用于删除行。
Join连接并不完全受支持。不支持FULL OUTER JOIN和CROSS JOIN。
Phoenix SQL在CRUD操作语法解释如下。

CREATE

创建一个简单的user表，id为主键，d为列族(如果不指定列族，内部映射到'0'列族)。可定义DDL属性'DEFAULT_COLUMN_FAMILY=列族名'来覆盖默认的列族名。

```
 create table user(
        id integer not null primary key,
        d.username varchar,
        d.address varchar
);
```

UPSERT

注意，Phoenix没有insert，其insert与update合起来叫做：upsert。

现在更新/插入两行。这里，显式地为id列设置值。在内部，这个SQL调用被转换为一个HBase put。

```
upsert into user values(1,'张三','城市1');
upsert into user values(2,'李四','城市2');
```

SELECT

查询指定的用户

```
select username from user;
```

ALTER

修改表结构。

```
alter table user add zipcode integer;
```

DELETE

删除行。

```
delete from user where id=1;
```

下面演示常见的Phoenix SQL语法的使用。

```
--创建电商表
CREATE TABLE t1(
pkey  integer NOT NULL PRIMARY KEY,
p.name varchar,
p.price double,
c.name varchar,
c.address varchar
);
-- 插入数据
-- 1) 商品数据
```

```
UPSERT INTO t1(pkey, p.name, p.price) values(1, '衬衣',168.50);
UPSERT INTO t1(pkey, p.name, p.price) values(2, '电脑',5168.50);
UPSERT INTO t1(pkey, c.name, c.address) values(3, '张三','城市1');
UPSERT INTO t1(pkey, c.name, c.address) values(4, '李四','城市2');
UPSERT INTO t1 values(5,'图书', 68.80, '王老五','城市3');
UPSERT INTO t1(pkey, p.name, p.price) values(3,'手机',3568.00);
UPSERT INTO t1(pkey, p.price) values(3,2538.00);
-- 动态修改表结构
ALTER TABLE t1 ADD c.phone varchar;
UPSERT INTO t1(pkey, c.name, c.address, c.phone) values(5, '赵小六','城市4','13566668888');
-- 删除
DELETE FROM t1 WHERE pkey=2;
DELETE FROM t1 WHERE p.name='手机';
DELETE FROM t1;
-- 查询
SELECT * FROM t1;
SELECT pkey, p.name,c.name FROM t1;
-- 删除表
DROP TABLE t1;
```

执行批处理

我们也可以创建自己的SQL脚本并使用命令行工具执行它们。可以通过Phoenix的bin目录下的psql.py脚本加载CSV数据或者执行包含sql脚本的文件。

现在我们来看一个例子。导航到Phoenix安装位置的bin/目录。

1）首先，创建一个us_population.sql文件，包含一个表定义：

```
CREATE TABLE IF NOT EXISTS us_population (
    state CHAR(2) NOT NULL,
    city VARCHAR NOT NULL,
    population BIGINT
    CONSTRAINT my_pk PRIMARY KEY (state, city));
```

2）然后创建一个us_population.csv文件，其中包含要放入该表的一些数据:

```
NY,New York,8143197
CA,Los Angeles,3844829
IL,Chicago,2842518
TX,Houston,2016582
PA,Philadelphia,1463281
AZ,Phoenix,1461575
TX,San Antonio,1256509
CA,San Diego,1255540
TX,Dallas,1213825
CA,San Jose,912332
```

3）最后，创建一个us_population_queries.sql文件，包含希望在该数据上运行的查询。

```
SELECT state as "State",count(city) as "City Count",sum(population) as "Population Sum"
FROM us_population
GROUP BY state
ORDER BY sum(population) DESC;
```

4）在命令行终端，执行如下的命令：

```
$ ./psql.py localhost:2181 us_population.sql us_population.csv us_population_queries.sql
```

这条命令同时做了三件事：创建表、插入数据、查询结果。典型的upsert速率是每秒20K - 50K行(取决于行有多宽)。

开发规范

Configuration实例的创建

该类应该通过调用HBaseConfiguration的create()方法来实例化。否则，将无法正确加载HBase中的相关配置项。
正确示例：

```
//该部分，应该是在类成员变量的声明区域声明
private Configuration hbaseConfig = null;
//最好在类的构造函数中，或者初始化方法中实例化该类
hbaseConfig = HBaseConfiguration.create();
```

错误示例：

```
hbaseConfig = new Configuration();
```

共享Configuration实例

HBase客户端代码通过创建一个与Zookeeper之间的HConnection，来获取与一个HBase集群进行交互的权限。一个Zookeeper的HConnection连接，对应着一个Configuration实例，已经创建的HConnection实例，会被缓存起来。也就是说，如果客户端需要与HBase集群进行交互的时候，会传递一个Configuration实例过去，HBase Client部分通过已缓存的HConnection实例，来判断属于这个Configuration实例的HConnection实例是否存在，如果不存在，就会创建一个新的，如果存在，就会直接返回相应的实例。
因此，如果频频地创建Configuration实例，会导致创建很多不必要的HConnection实例，很容易达到Zookeeper的

连接数上限。

建议在整个客户端代码范围内，都共用同一个Configuration对象实例。

Table实例的创建

```
public abstract class TableOperationImpl {
    private static Configuration conf = null;
    private static Connection connection = null;
    private static Table table = null;
    private static TableName tableName = TableName.valueOf("sample_table");
    public TableOperationImpl() {
        init();
    }
    public void init() {
        conf = ConfigurationSample.getConfiguration();
        try {
            connection = ConnectionFactory.createConnection(conf);
            table = conn.getTable(tableName);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void close() {
        if (table != null) {
            try {
                table.close();
            } catch (IOException e) {
                System.out.println("Can not close table.");
            } finally {
                table = null;
            }
        }
        if (connection != null) {
            try {
                connection.close();
            } catch (IOException e) {
                System.out.println("Can not close connection.");
            } finally {
                connection = null;
            }
        }
    }
    public void operate() {
        init();
        process();
        close();
    }
```

```
}
```

不允许多个线程在同一时间共用同一个Table实例

Table是一个非线程安全类，因此，同一个Table实例，不应该被多个线程同时使用，否则可能会带来并发问题。

Table实例缓存

如果一个Table实例可能会被长时间且被同一个线程固定且频繁地用到，例如，通过一个线程不断的往一个表内写入数据，那么这个Table在实例化后，就需要缓存下来，而不是每一次插入操作，都要实例化一个Table对象（尽管提倡实例缓存，但也不是在一个线程中一直沿用一个实例，个别场景下依然需要重构，可参见下一条规则）。
正确示例：
说明：
注意该实例中提供的以Map形式缓存Table实例的方法，未必通用。这与多线程多Table实例的设计方案有关。如果确定一个Table实例仅仅可能会被用于一个线程，而且该线程也仅有一个Table实例的话，就无须使用Map。这里提供的思路仅供参考。

```
//该Map中以TableName为Key值，缓存所有已经实例化的Table
private Map < String, Table > demoTables = new HashMap < String, Table > ();
//所有的Table实例，都将共享这个Configuration实例
private Configuration demoConf = null;
/**
<初始化一个HTable类>
<功能详细描述>
@param tableName
@return
@throws IOException
@see [类、类#方法、类#成员]
*/
private Table initNewTable(String tableName) throws IOException {
    try (Connection conn = ConnectionFactory.createConnection(demoConf)) {
        return conn.getTable(tableName);
    }
}
/**
<获取Table实例>
<功能详细描述>
@see [类、类#方法、类#成员]
*/
private Table getTable(String tableName) {
    if (demoTables.containsKey(tableName)) {
```

```
            return demoTables.get(tableName);
        } else {
            Table table = null;
            try {
                table = initNewTable(tableName);
                demoTables.put(tableName, table);
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            return table;
        }
    }
    /**
    * 写数据
    */
    public void putData(List < Put > dataList, String tableName) {
        Table table = getTable(tableName);
        //关于这里的同步：如果在采用的设计方案中，不存在多线程共用同一个Table实例
        //的可能的话，就无须同步了。这里需要注意的一点，就是Table实例是非线程安全的
        synchronized(table) {
            try {
                table.put(dataList);
                table.notifyAll();
            } catch (IOException e) {
                // 在捕获到IOE时，需要将缓存的实例重构。
                try {
                    // 关闭之前的Connection.
                    table.close();
                    // 重新创建这个实例.
                    table = initNewTable(tableName);
                } catch (IOException e1) {
                    // TODO
                }
            }
        }
    }
    /**
    * 错误示例：
    */
    public void putDataIncorrect(List < Put > dataList, String tableName) {
        Table table = null;
        try {
            //每次写数据，都创建一个HTable实例
            table = initNewTable(tableName);
            table.put(dataList);
        } catch (IOException e1) {
```

```
            // TODO Auto-generated catch block
            e1.printStackTrace();
        } finally {
            table.close();
        }
    }
```

Table实例写数据的异常处理

尽管在前一条规则中提到了提倡Table实例的重构，但是，并非提倡一个线程自始至终要沿用同一个Table实例，当捕获到IOException时，依然需要重构Table实例。示例代码可参考上一个规则的示例。
另外，勿轻易调用如下两个方法：
Configuration#clear：
这个方法，会清理掉所有的已经加载的属性，那么，对于已经在使用这个Configuration的类或线程而言，可能会带来潜在的问题（例如，假如Table还在使用这个Configuration，那么，调用这个方法后，Table中的这个Configuration的所有的参数，都被清理掉了），也就是说：只要还有对象或者线程在使用这个Configuration，就不应该调用这个clear方法，除非所有的类或线程，都已经确定不用这个Configuration了。那么，这个操作，可以在所有的线程要退出的时候来做，而不是每一次。
因此，这个方法，应该要放在进程要退出的地方去做。而不是每一次Table要重构的时候做。
HConnectionManager#deleteAllConnections:
这个可能会导致现有的正在使用的连接被从连接集合中清理掉，同时，因为在HTable中保存了原有连接的引用，可能会导致这个连接无法关闭，进而可能会造成泄漏。因此，这个方法不建议使用。

写入失败的数据要做相应的处理

在写数据的过程中，如果进程异常或一些其它的短暂的异常，可能会导致一些写入操作失败。因此，对于操作的数据，需要将其记录下来。在集群恢复正常后，重新将其写入到HBase数据表中。
另外，有一点需要注意：HBase Client返回写入失败的数据，是不会自动重试的，仅仅会告诉接口调用者哪些数据写入失败了。对于写入失败的数据，一定要做一些安全的处理，例如可以考虑将这些失败的数据，暂时写在文件中，或者，直接缓存在内存中。
正确示例：

```
private List<Row> errorList = new ArrayList<Row>();
/**
<采用PutList的模式插入数据>
<如果不是多线程调用该方法，可不采用同步>
*
@param put 一条数据记录
@throws IOException
@see [类、类#方法、类#成员]
*/
```

```
public synchronized void putData(Put put)
{
  // 暂时将数据缓存在该List中
  dataList.add(put);
  // 当dataList的大小达到PUT_LIST_SIZE之后，就执行一次Put操作
  if (dataList.size() >= PUT_LIST_SIZE)
  {
     try
     {
       demoTable.put(dataList);
     } catch (IOException e)
     {
       // 如果是RetriesExhaustedWithDetailsException类型的异常，
       // 说明这些数据中有部分是写入失败的这通常都是因为
       // HBase集群的进程异常引起，当然有时也会因为有大量
       // 的Region正在被转移，导致尝试一定的次数后失败
       if (e instanceof RetriesExhaustedWithDetailsException)
       {
          RetriesExhaustedWithDetailsException ree =
                    (RetriesExhaustedWithDetailsException) e;
          int failures = ree.getNumExceptions();
          for (int i = 0; i < failures; i++)
          {
             errorList.add(ree.getRow(i));
          }
        }
     }
     dataList.clear();
  }
}
```

资源释放

关于ResultScanner和Table实例，在用完之后，需要调用它们的close方法，将资源释放掉。Close方法，要放在finally块中，来确保一定会被调用到。

正确示例：

```
ResultScanner scanner = null;
try
{
    scanner = demoTable.getScanner(s);
    //Do Something here.
}
finally
{
    scanner.close();
}
```

错误示例：
在代码中未调用scanner.close()方法释放相关资源。
scanner.close()方法未放置在finally块中。

```
ResultScanner scanner = null;
scanner = demoTable.getScanner(s);
//Do Something here.
scanner.close();
```

Scan时的容错处理

Scan时不排除会遇到异常，例如，租约过期。在遇到异常时，建议Scan应该有重试的操作。事实上，重试在各类异常的容错处理中，都是一种优秀的实践，这一点，可以应用在各类与HBase操作相关的接口方法的容错处理过程中。
不用Admin时，要及时关闭，Admin实例不应常驻内存。
Admin的实例应尽量遵循 “用时创建，用完关闭”的原则。不应该长时间缓存同一个Admin实例。

hbase shell操作

DDL操作

```
#开启hbase shell
hbase shell
#查看hbase状态
status
#查看hbase版本
version
#创建命名空间
create_namespace '命名空间名'
#显示所有命名空间
list_namespace
#删除命名空间, 在删除一个命名空间时，该命名空间不能包含任何的表，否则会报错
drop_namespace '命名空间名'
```

```
#创建默认命名空间的表
create '表名称', '列族名称1','列族名称2','列族名称N'
#创建带有命名空间的表
create '命名空间:表名称', '列族名称1','列族名称2','列族名称N'
#列出所有表
list
#获得表的描述
describe '表名'
#删除table 表的 列族名称1 列族
alter 'table',{NAME=>'列族名称1',METHOD=>'delete'}
#删除多个列族
alter 'table', {NAME => '列族名称1', METHOD => 'delete'},{NAME => '列族名称2', METHOD => 'delete'}
#先把表下线
disable '表名'
#再drop表
drop '表名'。
```

DML操作

```
#添加数据
# 语法：put <table>,<rowkey>,<family:column>,<value>,[<timestamp>]
#如果不写timestamp，则系统默认
put 'table','id01', 'c_f1:name','111'
#获取数据
#get： 获取表中一行数据，不能扫描全表
# 语法：get <table>,<rowkey>,[<family:column>,....]
get 'table','id01'
#更新数据
#语法：重新put，put时会覆盖原来的数据
put 'table','id01', 'c_f1:name','222'
#scan扫描
# 语法：scan <table> ,{COLUMNS => [ <family:column>,.... ], LIMIT => num}
#扫描全表，大表操作不可取
scan 'table'
#获取表中前两行
scan 'table', {LIMIT => 2}
#扫描表中指定列族数据
scan 'table', {COLUMNS => 'c_f1'}
#扫描表中执行列族中列的数据
scan 'table', {COLUMNS => 'c_f2:cert_no'}
#扫描表中值=222 的数据
scan 'table', FILTER=>"ValueFilter(=,'name:222')"
# 筛选行，按照rowkey的范围[STARTROW,STOPROW)
scan 'table', {STARTROW =>'id01' , STOPROW => 'id03'}
#删除行中某列数据
# 语法：delete <table>, <rowkey>, <family:column>
```

```
# 必须指定列名
# 会删除执行列的所有版本数据
delete 'table', 'id04',  'c_f2:name'
#删除整行
# 语法：deleteall <table>, <rowkey>
deleteall 'table', 'id05'
#清空表数据
# 语法： truncate <table>
truncate 'table'
#查询表中有多少行
# 语法：count <table>, {INTERVAL => intervalNum, CACHE => cacheNum}
# INTERVAL设置多少行显示一次及对应的rowkey，默认1000；
# CACHE每次去取的缓存区大小，默认是10，调整该参数可提高查询速度
#查询表中数据行数
count 'table'
#按照2行显示一次，查询
count 'table', {INTERVAL => 2}。
```

hbase BulkLoad写海量数据

HBase写数据方式通常使用 HBase 提供的 API 方法，实现了接口调用，但对于海量的数据，接口的调用引起 CPU、内存占用过高，影响正常业务使用。
使用 Bulk Load 方式由于利用了 HBase 的数据信息是按照特定格式存储在 HDFS 里的这一特性，直接在 HDFS 中生成持久化的 HFile 数据格式文件，然后完成巨量数据快速入库的操作，配合 MapReduce 完成这样的操作，不占用 Region 资源，不会产生巨量的写入 I/O，所以需要较少的 CPU 和网络资源。
通过MapReduce 生成 HFile具体的代码实现：

1. 作业的配置。

```
 /* Job Configure */
Job job = Job.getInstance(conf,"HFileProducerETL");
job.setJarByClass(HFileProducerETL2.class);
TableMapReduceUtil.addDependencyJars(job);
job.setMapperClass(BulkLoadMapper.class);
job.setReducerClass(KeyValueSortReducer.class);
job.setMapOutputKeyClass(ImmutableBytesWritable.class);
job.setOutputValueClass(KeyValue.class);

if (fs.exists(outputPath)) {
    fs.delete(outputPath, true);
}
FileInputFormat.setInputPaths(job,inputPath);
FileOutputFormat.setOutputPath(job,outputPath);
```

```
HTable htable =new HTable(conf, tablename);
HFileOutputFormat2.configureIncrementalLoad(job,htable,htable);

return job.waitForCompletion(false) ? 0 : 1;
```

2. Map类实现。

key 类：ImmutableBytesWritable

value 类：KeyValue

```
public void (LongWritable key, Text value, Context context) {
    try {
        String line = value.toString();
        String[] all_column_values = line.split(context.getConfiguration().get(FILE_DELIMITER_ST
R),-1);
        String rowKeyString = all_column_values[rowKeyIndex];
        byte[] rowKey=Bytes.toBytes(rowKeyString);

        // rowKey 赋值
        ImmutableBytesWritable rowKeyWritable=new ImmutableBytesWritable(rowKey);

        if (all_column_name.length!=all_column_values.length){
            return;
        }
        for (int i = 0; i < all_column_name.length; i++) {
            if (export_column_index[i] == EXPORT_FLAG
                    && !all_column_values[i].trim().equals(HBaseConstant.HiveNULL)
                    && !all_column_values[i].trim().equals(HBaseConstant.HBaseNull)
                    && !all_column_values[i].trim().equals(HBaseConstant.HEmpty)
            ){
                KeyValue kv ;
                // 如果是 HFILE_CLEAR_FLAG 则 清空
                if (all_column_values[i].equals(HBaseConstant.HFILE_CLEAR_FLAG)){
                    all_column_values[i]=HBaseConstant.HEmpty;
                }
                if (timestampIndex == TIMESTAMP_COLUMN_DEFAULT_INDEX){
                    kv = new KeyValue(rowKey, CF, Bytes.toBytes(all_column_name[i]),Bytes.toBytes(all_co
lumn_values[i]));
                }
                else{
                    Long timestamp_value = Long.parseLong(all_column_values[timestampIndex]);
                    kv = new KeyValue(rowKey, CF, Bytes.toBytes(all_column_name[i]),timestamp_value,By
tes.toBytes(all_column_values[i]));
                }
                context.write(rowKeyWritable,kv);
            }
        }
```

```
        }catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

3. 生成 HFile 之后，与HBase映射实现，代码如下：

```
Configuration conf = DefConfiguration.GetConfiguration(hbaseEnv);
conf.setInt("hbase.mapreduce.bulkload.max.hfiles.perRegion.perFamily",5000)
HTable htable =new HTable(conf, tableName);
LoadIncrementalHFiles loader = new LoadIncrementalHFiles(conf);
loader.doBulkLoad(hFilePath, htable);
```

性能优化

HBase表设计建议及优化

业务表设计建议

(1)、预分region，使region分布均匀，提高并发。
(2)、避免过多的热点region。根据应用场景，可考虑将时间因素引入Rowkey。
(3)、同时访问的数据，应设计Rowkey使其在物理上相邻存储，以便通过一次范围扫描高效获取，避免多次单点查询带来的性能开销。
(4)、数据存放在同一行；同时读取的数据存放在同一cell。
(5)、查询频繁属性放在Rowkey前面部分。Rowkey的设计在排序上必须与主要的查询条件契合。
(6)、离散度较好的属性作为RowKey组成部分。分析数据离散度特点以及查询场景，综合各种场景进行设计。
(7)、存储冗余信息，提高检索性能。使用二级索引，适应更多查询场景。
(8)、利用过期时间、版本个数设置等操作，让表能自动清除过期数据。
说明：在HBase中，一直在繁忙写数据的region被称为热点region。

Pre-Creating Regions

默认情况下，在创建HBase表的时候会自动创建一个region分区，当导入数据的时候，所有的HBase客户端都向这一个region写数据，直到这个region足够大了才进行切分。一种可以加快批量写入速度的方法是通过预先创建一些空的regions，这样当数据写入HBase时，会按照region分区情况，在集群内做数据的负载均衡。有关预分区，详情参见：Table Creation: Pre-Creating Regions，下面是一个例子：

```
public static boolean createTable(HBaseAdmin admin, HTableDescriptor table, byte[][] splits)throws IOE
xception {
  try {
    admin.createTable(table, splits);
    return true;
  } catch (TableExistsException e) {
    logger.info("table " + table.getNameAsString() + " already exists");
    // the table already exists...
    return false;
  }
}
public static byte[][] getHexSplits(String startKey, String endKey, int numRegions) {
  //start:001,endkey:100,10region [001,010][011,020]
  byte[][] splits = new byte[numRegions-1][];
  BigInteger lowestKey = new BigInteger(startKey, 16);
  BigInteger highestKey = new BigInteger(endKey, 16);
  BigInteger range = highestKey.subtract(lowestKey);
  BigInteger regionIncrement = range.divide(BigInteger.valueOf(numRegions));
  lowestKey = lowestKey.add(regionIncrement);
  for(int i=0; i < numRegions-1;i++) {
    BigInteger key = lowestKey.add(regionIncrement.multiply(BigInteger.valueOf(i)));
    byte[] b = String.format("%016x", key).getBytes();
    splits[i] = b;
  }
  return splits;
}
```

Row Key

HBase中row key用来检索表中的记录，支持以下三种方式：
通过单个row key访问：即按照某个row key键值进行get操作；
通过row key的range进行scan：即通过设置startRowKey和endRowKey，在这个范围内进行扫描；
全表扫描：即直接扫描整张表中所有行记录。
在HBase中，row key可以是任意字符串，最大长度64KB，实际应用中一般为10~100bytes，存为byte[]字节数组，一般设计成定长的。
row key是按照字典序存储，因此，设计row key时，要充分利用这个排序特点，将经常一起读取的数据存储到一块，将最近可能会被访问的数据放在一块。
举个例子：如果最近写入HBase表中的数据是最可能被访问的，可以考虑将时间戳作为row key的一部分，由于是字典序排序，所以可以使用Long.MAX_VALUE - timestamp作为row key，这样能保证新写入的数据在读取时可以被快速命中。

Column Family

不要在一张表里定义太多的column family。目前Hbase并不能很好的处理超过2~3个column family的表。因为某个

column family在flush的时候，它邻近的column family也会因关联效应被触发flush，最终导致系统产生更多的I/O。感兴趣的同学可以对自己的HBase集群进行实际测试，从得到的测试结果数据验证一下。

In Memory

创建表的时候，可以通过HColumnDescriptor.setInMemory(true)将表放到RegionServer的缓存中，保证在读取的时候被cache命中（小表使用，大表不建议使用）。

Max Version

创建表的时候，可以通过HColumnDescriptor.setMaxVersions(int maxVersions)设置表中数据的最大版本，如果只需要保存最新版本的数据，那么可以设置setMaxVersions(1)。

Time To Live

创建表的时候，可以通过HColumnDescriptor.setTimeToLive(int timeToLive)设置表中数据的存储生命期，过期数据将自动被删除，例如如果只需要存储最近两天的数据，那么可以设置setTimeToLive(2 * 24 * 60 * 60)。

Compact & Split

在HBase中，数据在更新时首先写入WAL 日志(HLog)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。与此同时， 系统会在zookeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了(minor compact)。
StoreFile是只读的，一旦创建后就不可以再修改。因此Hbase的更新其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并(major compact)，将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行分割(split)，等分为两个StoreFile。
由于对表的更新是不断追加的，处理读请求时，需要访问Store中全部的StoreFile和MemStore，将它们按照row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，通常合并过程还是比较快的。
实际应用中，可以考虑必要时手动进行major compaction，将同一个row key的修改进行合并形成一个大的StoreFile。同时，可以将StoreFile设置大些，减少split的发生。
HBase为了防止小文件（被刷到磁盘的MemStore）过多，以保证查询效率，hbase需要在必要的时候将这些小的store file合并成相对较大的store file，这个过程就称之为compaction。在hbase中，主要存在两种类型的compaction：minor compaction和major compaction。
minor compaction:是较小、很少文件的合并。
major compaction的功能是将所有的store file合并成一个，触发major compaction的可能条件有：major_compact 命令、majorCompact() API、region server自动运行（相关参数：hbase.hregion.majorcompaction 默认为24 小时、hbase.hregion.majorcompaction.jitter 默认值为0.2 防止region server 在同一时间进行major compaction）。hbase.hregion.majorcompaction.jjitter数的作用是：对参数hbase.hregion.majorcompaction 规定的值起到浮动的作用，假如两个参数都为默认值24和0,2，那么major compact最终使用的数值为：19.2~28.8 这个范围。
1、关闭自动major compaction。

2、手动编程major compaction。
Timer类，minor compaction的运行机制要复杂一些，它由以下几个参数共同决定：
hbase.hstore.compaction.min :默认值为 3，表示至少需要三个满足条件的store file时，minor compaction才会启动。
hbase.hstore.compaction.max 默认值为10，表示一次minor compaction中最多选取10个store file。
hbase.hstore.compaction.min.size 表示文件大小小于该值的store file 一定会加入到minor compaction的store file中
hbase.hstore.compaction.max.size 表示文件大小大于该值的store file 一定会被minor compaction排除
hbase.hstore.compaction.ratio 将store file 按照文件年龄排序（older to younger），minor compaction总是从older store file开始选择。

容量限制

官方建议每台HBase RegionServer的region数量不超过200个，实际使用经验建议不超过500个。

写表操作

多HTable并发写

创建多个HTable客户端用于写操作，提高写数据的吞吐量，一个例子：

```
static final Configuration conf = HBaseConfiguration.create();
static final String table_log_name = “user_log”;
wTableLog = new HTable[tableN];
for (int i = 0; i < tableN; i++) {
    wTableLog[i] = new HTable(conf, table_log_name);
    wTableLog[i].setWriteBufferSize(5 * 1024 * 1024); //5MB
    wTableLog[i].setAutoFlush(false);
}
```

HTable参数设置

Auto Flush

通过调用HTable.setAutoFlush(false)方法可以将HTable写客户端的自动flush关闭，这样可以批量写入数据到HBase，而不是有一条put就执行一次更新，只有当put填满客户端写缓存时，才实际向HBase服务端发起写请求。默认情况下autoFlush是开启的。

Write Buffer

通过调用HTable.setWriteBufferSize(writeBufferSize)方法可以设置HTable客户端的写buffer大小，如果新设置的buffer小于当前写buffer中的数据时，buffer将会被flush到服务端。其中，writeBufferSize的单位是byte字节数，可以根据实际写入数据量的多少来设置该值。

WAL Flag

在HBase中，客户端向集群中的RegionServer提交数据时（Put/Delete操作），会先写WAL（Write Ahead Log）日志（即HLog，一个RegionServer上的所有Region共享一个HLog），只有当WAL日志写成功后，再接着写MemStore，然后客户端被通知提交数据成功；如果写WAL日志失败，客户端则被通知提交失败。这样做的好处是可以做到RegionServer宕机后的数据恢复。
因此，对于相对不太重要的数据，可以在Put/Delete操作时，通过调用Put.setWriteToWAL(false)或Delete.setWriteToWAL(false)函数，放弃写WAL日志，从而提高数据写入的性能。
值得注意的是：谨慎选择关闭WAL日志，因为这样的话，一旦RegionServer宕机，Put/Delete的数据将会无法根据WAL日志进行恢复。

批量写

通过调用HTable.put(Put)方法可以将一个指定的row key记录写入HBase，同样HBase提供了另一个方法：通过调用HTable.put(List)方法可以将指定的row key列表，批量写入多行记录，这样做的好处是批量执行，只需要一次网络I/O开销，这对于对数据实时性要求高，网络传输RTT高的情景下可能带来明显的性能提升。

多线程并发写

在客户端开启多个HTable写线程，每个写线程负责一个HTable对象的flush操作，这样结合定时flush和写buffer（writeBufferSize），可以既保证在数据量小的时候，数据可以在较短时间内被flush（如1秒内），同时又保证在数据量大的时候，写buffer一满就及时进行flush。下面给个具体的例子：

```
for (int i = 0; i < threadN; i++) {
    Thread th = new Thread() {
        public void run() {
            while (true) {
                try {
                    sleep(1000); //1 second
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
synchronized (wTableLog[i]) {
                    try {
                        wTableLog[i].flushCommits();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
}
    };
    th.setDaemon(true);
    th.start();
}。
```

HBase BulkLoad写数据

HBase底层存储是以HFile文件存储在磁盘上。

1、根据HDFS的数据或者外部的数据生成HBase底层的数据格式HFile文件。

2、根据生成目标HFile，利用HBase提供的Bulkload工具将HFile移动（或加载）到HBase目录下，bulkLoad主要是将数据编写成HFile的形式，批量加载到HBase中，具有优于其他数据提取机制的优点，此操作完全绕过写入路径。

优缺点：

① 数据可以立即被HBase使用，但是会对集群造成额外的负载和延迟。

② BulkLoad操作不会预写日志（WALs），因此不会引起过量的flush和split。

③ BulkLoad操作不会引起过多的垃圾回收（GC）

BulkLoad使用流程

(1)、从现有的数据源中提取数据，可以从关系型数据库中导出数据，如果是TSV/CSV形式的数据参见http://hbase.apache.org/book.html#importtsv.

(2)、将已导出的数据，处理成为HFile数据格式(HFile格式介绍:http://hbase.apache.org/book.html#_hfile_format_2)，需要使用MR任务进行转换，并且数据是唯一的，通常需要我们自己编写Mapper。作业必须输出行键Key，以及一个KeyValue（一个put，delete作为value），Reducer由HBase进行处理，如果需要我们自己对数据进行处理，可以在MR任务中，以此流程Mapper->Reducer->Mapper编写，最终Reducer交由hbase处理：

1、检查表的配置信息，以及总的分区器排序。

2、将分区文件上传到集群中，并添加到 DistributedCache中。

3、设置一定数量的reduce任务数，匹配当前region数量。

4、设置输出的key/value类，匹配HFileOutputFormat。

5、设置Reducer适当的排序（ KeyValueSortReducer或者 PutSortReducer）

(3)、在输出的文件夹中每个region创建一个HFile，输入的数据会被完全覆盖，一般情况下需要准备的磁盘大小至少是原始数据集大小的两倍，比如mysqldump出100G的数据，在以HFile上传到HDFS后，需要准备至少200G的存储空间。

(4)、将文件加载至HBase中。使用LoadIncrementalHFiles命令，并向HDFS传入定位文件的URL，每个文件都加载到该区域的RegionServer上相关的region当中，可以通过传递限制输入的版本数量–version = N选项，在N需要包含最大版本数量从旧到新（最大时间戳到最小时间戳），如果在创建文件后，分割了一个region，则该工具会自动根据新边界进行切分HFile，效率相对较低。

读表操作

多HTable并发读

创建多个HTable客户端用于读操作，提高读数据的吞吐量，一个例子：

```
static final Configuration conf = HBaseConfiguration.create();
static final String table_log_name = “user_log”;
rTableLog = new HTable[tableN];
```

```
for (int i = 0; i < tableN; i++) {
rTableLog[i] = new HTable(conf, table_log_name);
rTableLog[i].setScannerCaching(50);
```

HTable参数设置

Scanner Caching

hbase.client.scanner.caching配置项可以设置HBase scanner一次从服务端抓取的数据条数，默认情况下一次一条。通过将其设置成一个合理的值，可以减少scan过程中next()的时间开销，代价是scanner需要通过客户端的内存来维持这些被cache的行记录。
有三个地方可以进行配置：1）在HBase的conf配置文件中进行配置；2）通过调用HTable.setScannerCaching(int scannerCaching)进行配置；3）通过调用Scan.setCaching(int caching)进行配置。三者的优先级越来越高。

Scan Attribute Selection

scan时指定需要的Column Family，可以减少网络传输数据量，否则默认scan操作会返回整行所有Column Family的数据。

Close ResultScanner

通过scan取完数据后，记得要关闭ResultScanner，否则RegionServer可能会出现问题（对应的Server资源无法释放）。

批量读

通过调用HTable.get(Get)方法可以根据一个指定的row key获取一行记录，同样HBase提供了另一个方法：通过调用HTable.get(List)方法可以根据一个指定的row key列表，批量获取多行记录，这样做的好处是批量执行，只需要一次网络I/O开销，这对于对数据实时性要求高而且网络传输RTT高的情景下可能带来明显的性能提升。

多线程并发读

在客户端开启多个HTable读线程，每个读线程负责通过HTable对象进行get操作。下面是一个多线程并发读取HBase，获取店铺一天内各分钟PV值的例子：

```
public class DataReaderServer {
    //获取店铺一天内各分钟PV值的入口函数
    public static ConcurrentHashMap<String, String> getUnitMinutePV(long uid, long startStamp, long
endStamp){
        long min = startStamp;
        int count = (int)((endStamp - startStamp) / (60*1000));
        List<String> lst = new ArrayList<String>();
        for (int i = 0; i <= count; i++) {
          min = startStamp + i * 60 * 1000;
          lst.add(uid + "_" + min);
        }
        return parallelBatchMinutePV(lst);
```

```
    }
     //多线程并发查询，获取分钟PV值
private static ConcurrentHashMap<String, String> parallelBatchMinutePV(List<String> lstKeys){
        ConcurrentHashMap<String, String> hashRet = new ConcurrentHashMap<String, String>();
        int parallel = 3;
        List<List<String>> lstBatchKeys  = null;
        if (lstKeys.size() < parallel ){
            lstBatchKeys  = new ArrayList<List<String>>(1);
            lstBatchKeys.add(lstKeys);
        }
        else{
            lstBatchKeys  = new ArrayList<List<String>>(parallel);
            for(int i = 0; i < parallel; i++  ){
                List<String> lst = new ArrayList<String>();
                lstBatchKeys.add(lst);
            }

            for(int i = 0 ; i < lstKeys.size() ; i ++ ){
                lstBatchKeys.get(i%parallel).add(lstKeys.get(i));
            }
        }

        List<Future< ConcurrentHashMap<String, String> >> futures = new ArrayList<Future< Concurre
ntHashMap<String, String> >>(5);

        ThreadFactoryBuilder builder = new ThreadFactoryBuilder();
        builder.setNameFormat("ParallelBatchQuery");
        ThreadFactory factory = builder.build();
        ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(lstBatchKey
s.size(), factory);

        for(List<String> keys : lstBatchKeys){
            Callable< ConcurrentHashMap<String, String> > callable = new BatchMinutePVCallable(keys);
            FutureTask< ConcurrentHashMap<String, String> > future = (FutureTask< ConcurrentHashMap
<String, String> >) executor.submit(callable);
            futures.add(future);
        }
        executor.shutdown();

        // Wait for all the tasks to finish
        try {
          boolean stillRunning = !executor.awaitTermination(
              5000000, TimeUnit.MILLISECONDS);
          if (stillRunning) {
            try {
                executor.shutdownNow();
            } catch (Exception e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
          }
        }
      } catch (InterruptedException e) {
        try {
          Thread.currentThread().interrupt();
        } catch (Exception e1) {
          // TODO Auto-generated catch block
          e1.printStackTrace();
        }
      }
      // Look for any exception
      for (Future f : futures) {
        try {
          if(f.get() != null)
          {
            hashRet.putAll((ConcurrentHashMap<String, String>)f.get());
          }
        } catch (InterruptedException e) {
          try {
            Thread.currentThread().interrupt();
          } catch (Exception e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
          }
        } catch (ExecutionException e) {
          e.printStackTrace();
        }
      }
      return hashRet;
  }
   //一个线程批量查询，获取分钟PV值
  protected static ConcurrentHashMap<String, String> getBatchMinutePV(List<String> lstKeys){
      ConcurrentHashMap<String, String> hashRet = null;
      List<Get> lstGet = new ArrayList<Get>();
      String[] splitValue = null;
      for (String s : lstKeys) {
          splitValue = s.split("_");
          long uid = Long.parseLong(splitValue[0]);
          long min = Long.parseLong(splitValue[1]);
          byte[] key = new byte[16];
          Bytes.putLong(key, 0, uid);
          Bytes.putLong(key, 8, min);
          Get g = new Get(key);
          g.addFamily(fp);
          lstGet.add(g);
```

```
        }
        Result[] res = null;
        try {
            res = tableMinutePV[rand.nextInt(tableN)].get(lstGet);
        } catch (IOException e1) {
            logger.error("tableMinutePV exception, e=" + e1.getStackTrace());
        }

        if (res != null && res.length > 0) {
            hashRet = new ConcurrentHashMap<String, String>(res.length);
            for (Result re : res) {
                if (re != null && !re.isEmpty()) {
                    try {
                        byte[] key = re.getRow();
                        byte[] value = re.getValue(fp, cp);
                        if (key != null && value != null) {
                            hashRet.put(String.valueOf(Bytes.toLong(key,
                                    Bytes.SIZEOF_LONG)), String.valueOf(Bytes
                                    .toLong(value)));
                        }
                    } catch (Exception e2) {
                        logger.error(e2.getStackTrace());
                    }
                }
            }
        }

        return hashRet;
    }
}
//调用接口类，实现Callable接口
class BatchMinutePVCallable implements Callable<ConcurrentHashMap<String, String>>{
    private List<String> keys;

    public BatchMinutePVCallable(List<String> lstKeys ) {
        this.keys = lstKeys;
    }

    public ConcurrentHashMap<String, String> call() throws Exception {
        return DataReadServer.getBatchMinutePV(keys);
    }
}
```

缓存查询结果

对于频繁查询HBase的应用场景，可以考虑在应用程序中做缓存，当有新的查询请求时，首先在缓存中查找，如果存

在则直接返回，不再查询HBase；否则对HBase发起读请求查询，然后在应用程序中将查询结果缓存起来。至于缓存的替换策略，可以考虑LRU等常用的策略。

Blockcache

HBase上RegionServer的内存分为两个部分，一部分作为Memstore，主要用来写；另外一部分作为BlockCache，主要用于读。
写请求会先写入MemStore，RegionServer会给每个region提供一个Memstore，当Memstore满64MB以后，会启动 flush刷新到磁盘。当Memstore的总大小超过限制时（heapsize * hbase.regionserver.global.memstore.upperLimit * 0.9），会强行启动flush进程，从最大的Memstore开始flush直到低于限制。
读请求先到MemStore中查数据，查不到就到BlockCache中查，再查不到就会到磁盘上读，并把读的结果放入BlockCache。由于BlockCache采用的是LRU策略，因此BlockCache达到上限(heapsize * hfile.block.cache.size * 0.85)后，会启动淘汰机制，淘汰掉最老的一批数据。
一个RegionServer上有一个BlockCache和N个MemStore，它们的大小之和不能大于等于heapsize * 0.8，否则HBase不能启动。默认BlockCache为0.3，而Memstore为0.4。对于注重读响应时间的系统，可以将 BlockCache设大些，比如设置BlockCache=0.4，Memstore=0.39，以加大缓存的命中率。
有关BlockCache机制，请参考这里：HBase的BlockCache，HBase的BlockCache机制，HBase中的缓存的计算与使用。
HTable和HTablePool使用注意事项
HTable和HTablePool都是HBase客户端API的一部分，可以使用它们对HBase表进行CRUD操作。下面结合在项目中的应用情况，对二者使用过程中的注意事项做一下概括总结。

```
Configuration conf = HBaseConfiguration.create();
try (Connection connection = ConnectionFactory.createConnection(conf)) {
  try (Table table = connection.getTable(TableName.valueOf(tablename)) {
    // use table as needed, the table returned is lightweight
  }
}
```

最佳实践

HBase Row Key 设计原则

HBase中RowKey用来检索表中的记录，支持以下三种方式：

1. 通过单个RowKey访问：即按照某个row key键值进行get操作；
2. 通过RowKey的range进行scan：即通过设置startRowKey和endRowKey，在这个范围内进行扫描；
3. 全表扫描：即直接扫描整张表中所有行记录。

在HBase中，RowKey可以是任意字符串，最大长度64KB，实际应用中一般为10~100bytes，存为byte[]字节数组，一般设计成定长的。
RowKey是按照字典序存储，因此，设计row key时，要充分利用这个排序特点，将经常一起读取的数据存储到一块，将最近可能会被访问的数据放在一块。
举个例子：如果最近写入HBase表中的数据是最可能被访问的，可以考虑将时间戳作为RowKey的一部分，由于是字典序排序，所以可以使用Long.MAX_VALUE - timestamp作为row key，这样能保证新写入的数据在读取时可以被快速命中。

RowKey规则

RowKey要尽可能的短，越小越好

当HBase的RowKey过长时，会对性能产生影响，因为HBase是基于Key-Value存储的数据库，RowKey越长，需要扫描的字节数就越多，查询和写入的效率就会降低。因此，我们需要考虑如何处理过长的RowKey。

以下是一些处理过长RowKey的方法：

1. 缩短RowKey的长度：如果RowKey包含不必要的信息，可以考虑删除或缩短这些信息，以减少RowKey的长度。
2. 使用Hash算法：可以使用Hash算法将RowKey转换为一个较短的值，例如，使用MD5、SHA-1等算法将RowKey转换到固定长度的值。
3. 分解RowKey：如果RowKey包含多个部分，可以将其拆分成多个部分，以降低每个部分的长度。例如，将时间戳和设备ID作为两个部分，分别存储在RowKey的不同位置。
4. 压缩RowKey：可以使用压缩算法，例如，LZ4、Snappy等算法，将RowKey压缩后存储，以减少存储空间和读写时间。

以上是处理过长RowKey的一些方法，具体方法需要根据实际情况进行选择。

RowKey要和业务场景相贴合，避免造成热点写入/读取问题

一般有下面这几种方案：

1. 均匀分区：将数据均匀的分配到不同的分区中，这种方式适用于数据分布比较均匀的场景。
2. 前缀分区：根据数据的前缀信息进行分区，这种方式适用于数据的前缀信息比较规律的场景。
3. 范围分区：根据数据的范围信息进行分区，将数据按照一定的范围进行分区，这种方式适用于数据分布比较集中的场景。
4. hash分区：根据数据的hash值进行分区，将数据均匀地分配到不同的分区中，这种方式适用于数据分布不均匀的场景。
5. 自定义分区：根据业务需求自定义分区策略，根据不同的需求采用不同的分区方式，这种方式适用于特定的业务场景。
   以上是避免热点RowKey的一些方法，具体方法需要根据实际情况进行选择。

如何应对HBase集群状态异常

查看HBase Master UI，观察以下指标。

每台RegionServer的平均Region数量

1. 数量过多(官方建议100，实际可以到500-800，超过1000就容易出问题)则需要进行扩容；
2. 当Region过多时，可以调整以下参数进行优化：
   - hbase.regionserver.region.split.policy=org.apache.hadoop.hbase.regionserver.IncreasingToUpperBoundRegionSplitPolicy
   - hbase.increasing.policy.initial.size=5368709120 -- Region到达这个大小才会开始分裂
   - hbase.hregion.max.filesize=10737418240 -- Region最大值

是否存在一直无法上线的RIT Region

可尝试清理Master WALs和Zookeeper上的临时数据，再重启Master尝试：

1. 停止所有Master服务；
2. 删除HDFS上的/apps/hbase/data/MasterProcWALs目录；
3. 进入zkcli，删除Zookeeper上/hbase-unsecure/region-in-transition；
4. 启动所有Master服务；

Region分布是否均衡

每台RegionServer的Region数量不均衡，后台进入hbase shell执行均衡命令balancer_enabled值为false，需要手动开启：

1. balancer_switch true
2. balancer_enabled
3. balancer

4. 观察一段时间，如果还是不均衡，可考虑修改按Region数量进行均衡，然后重启Master服务：
    - hbase.master.loadbalance.bytable=true
    - hbase.master.loadbalancer.class=org.apache.hadoop.hbase.master.balancer.SimpleLoadBalancer

是否存在Region热点问题

某台RegionServer请求数较大，怀疑Region热点问题：

1. 分析RegionServer具体哪个组件或者表的Region读写较多，比如OpenTSDB;
2. 对于历史数据导入HBase操作，建议使用bulkload方式进行导入，增量导入可加大客户端缓存，减少RPC请求，降低HBase集群压力。

性能调优

组件运行参数调优

服务	配置文件	参数	推荐值	默认值（仅参考）	调优说明
HBase	hbase-env	HeapSize	--	1024	堆内内存，但具体大小应根据硬件配置和工作负载来调整。
		HBASE_REGIONSERVER_OFFHEAPSIZE	--	-1(JVM参数默认值)	堆外内存 JVM参数-XX:MaxDirectMemorySize的值 若出现 java.lang.OutOfMemoryError: Direct buffer memory异常可以上调这个值。
	hbase-site	hbase.regionserver.handler.count	256	30	总的处理线程数，可以根据客户端的请求数进行调整，读写请求较多时，增加此值。
		hbase.ipc.server.callqueue.handler.factor	0.1	0.1	请求量大时，减少队列锁竞争。 建议保持默认值。
		hbase.ipc.server.callqueue.type	fifo	fifo	fifo类型时，可以使用fastpath，请求不进入队列，直接由handler处理。 建议保持默认值。
		hbase.regionserver.memstore.class	org.apache.hadoop.hbase.regionserver.skiplist.hbase.CCSMapMemStore	org.apache.hadoop.hbase.regionserver.DefaultMemStore	开启CCSMapMemstore特性
		hbase.regionserver.global.memstore.size	0.4	0.4	在RegionServer中，MemStoreFlusher线程负责执行flush操作。该线程会定期检查写操作内存的使用情况，当写操作占用的内存总量达到设定的阈值时，MemStoreFlusher将启动flush操作。它会按照从大到小的顺序，flush部分相对较大的memstore，直到所占用的内存量低于阈值。 建议保持默认值。
		hbase.client.ipc.pool.size	1	1	客户端和某个RS的socket数量。 建议保持默认值。
		hbase.hstore.compaction.throughput.lower.bound	209715200 (200MB)	52428800（50MB）	compaction速度最高上限设置，主要是方便验证时改表结构做major_compact快一些
		hbase.hstore.compaction.throughput.higher.bound	524288000 (500MB)	104857600（100MB）	
HDFS	hdfs-site	dfs.client.read.shortcircuit	true	false	HDFS的配置，短路读特性开启
		dfs.client.read.shortcircuit.streams.cache.size	256	256	short-circuit最多持有的文件描述符数量。 建议保持默认值。
		dfs.client.hedged.read.threadpool.size	0	0	关掉hedged read，防止重复请求，资源浪费。 建议保持默认值。

表属性设置优化

设置方法	说明
alter 'usertable',{NAME=>'0',COMPRESSION => 'SNAPPY'} major_compact 'usertable'	压缩，取值有NONE/SNAPPY 主要影响读性能，一般取SNAPPY较优
alter 'usertable',{NAME=>'0', DATA_BLOCK_ENCODING=>'ROW_INDEX_V1'} major_compact 'usertable'	编码,取值有FAST_DIFF和ROW_INDEX_V1 主要影响读性能，一般取ROW_INDEX_V1较优
alter 'usertable',{NAME=>'0',IN_MEMORY => true, PREFETCH_BLOCKS_ON_OPEN =>true} major_compact 'usertable'	如果L2缓存足够大，可以缓存下表所有内容，考虑修改表属性，在open的时候就缓存表数据到内存，避免需要提前预热，如需去除相关属性：
alter 'mock_table_prod',METHOD => 'table_att_unset', NAME => 'PREFETCH_BLOCKS_ON_OPEN'	
alter 'usertable',{NAME=>'0', BLOCKSIZE => 16384} major_compact 'usertable'	BLOCKSIZE默认值是64K，该参数一般值较小适用于get查询，较大适用于scan查询。 get场景可以设置为16K(16384)，观察是否比默认值有增加

常见问题

客户端访问hbase报错Connection reset by peer

Connection reset by peer是服务端主动关闭了连接。这种情况一般有以下几个原因：

1. 客户端传递的认证有问题，服务端未校验通过认证信息关闭的连接，建议检查下认证信息是否设置正确。
2. 客户端hbase-site.xml配置文件未正确加载。 kerberos认证过程中需要的配置项比较多，这些配置项一般在集群的 $HBASE_HOME/conf/hbase-site.xml 中定义，可以尝试让应用加载到该配置文件。如果是spark应用，可以把 hbase-site.xml 拷贝到所有节点 $SPARK_HOME/conf 目录下面。

HMaster/RegionServer挂掉，频繁重启

- 可能原因
  HMaster/RegionServer 分配堆内存太小，导致OOM。
- 处理方式
  调大 hbase-env.sh 中的 Heapsize 和 HMHeapsize

如何更改hbase表名

原生没有这个命令，可以通过快照方式绕行，这个过程中只有文件软链的生成，不会发生真实文件拷贝。

1. 停止表继续插入

```
hbase shell>disable 'tableName'
```

2. 制作快照

```
hbase shell> snapshot 'tableName', 'tableSnapshot'
```

3. 克隆快照为新的名字

```
hbase shell> clone_snapshot 'tableSnapshot', 'newTableName'
```

4. 删除快照

```
hbase shell> delete_snapshot 'tableSnapshot'
```

5. 删除原来表

```
hbase shell> drop 'tableName'
```

list后即可查看到newTableName表

hbase表未上线

大量表无法上线通用处理方法

1. 停掉HBase服务
2. 然后移走MasterData目录 (这里不会涉及数据丢失)

```
hdfs dfs -mv /hbase/MasterData /hbase/MasterData2
```

3. 启动HBase服务
4. 执行 hbase hbck > hbck.txt 执行健康检查命令，检查表是否都正常上线。

meta 表无法上线

- 现象
  执行hbck报错， Master is initializing ， 查看active master日志，有打印 hbase:meta is NOT online
- 处理方式
  上线hbase:meta

```
hbase hbck -j $HBASE_HOME/hbase-hbck2/hbase-hbck2-1.2.0.jar assigns -o 1588230740
```

namespace表未上线

- 现象
  执行hbck报错， Master is initializing ， 查看active master日志，有打印 hbase:namespace xxx is NOT online
- 处理方式
  上线 hbase:namespace

```
hbase hbck -j $HBASE_HOME/hbase-hbck2/hbase-hbck2-1.2.0.jar assigns -o ${namespace的region id}
```

普通表region未上线如何处理

- 现象
  执行hbck后，打印输出中有一些ERROR报错信息： region not deployed on any region server
- 处理方式
  单个region：

```
hbase hbck -j /usr/local/service/hbase/hbase-hbck2/hbase-hbck2-*.jar assigns -o ${namespace的region id}
```

如果有问题region数量较多，批量处理方式：

```
hbase hbck > hbck.txt
###下面命令有个jar包路径，可能需要根据实际路径替换
cat hbck.txt | grep Region | awk -F '\\.,' '{print $1}' | awk -F '\\.' '{print "hbase hbck -j /usr/local/service/hbase/
hbase-hbck2/hbase-hbck2-*.jar assigns -o "$NF}' | grep -v ERROR > script.sh
###执行下script.sh之前检查一下再执行
chmod +x script.sh
sh script.sh
```

assign命令执行后仍无法上线region

- 现象
  assign 命令后仍然无法上线region， hbck输出还是 not deployed
  查看hbase UI上，对应的region处于 RIT CLOSING 状态。
  查看Procedure & Locks页面，看到对应region有其他procedure，导致assign没有获取到锁，因此没有成功上线。
- 处理方式
  需要执行以下命令将procedure lock释放，再重新执行assign命令。

```
hbase hbck -j $HBASE_HOME/hbase-hbck2/hbase-hbck2-1.2.0.jar bypass -or pid
```

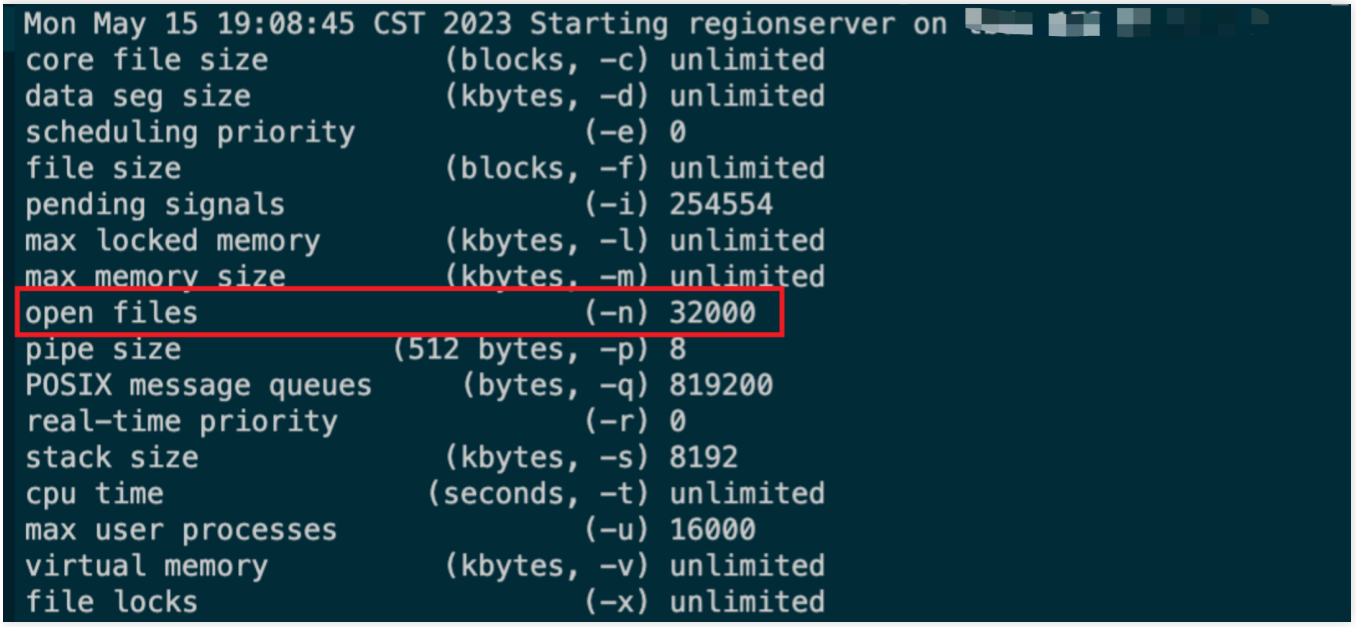
访问开启kerberos认证的hbase报错GSS initiate

failed

可能原因：
hbase.master.kerberos.principal 和 hbase.regionserver.kerberos.principal 配置时候不能直接配置主机名字，需要用 _HOST 代替，这里的 principal 是客户端和服务端协商使用。HMaster会主备切换，主机名不固定。

hbase支持的最大连接数是多少？

hbase不同于HiveServer，没有session的概念，所以没有固定的长连接。Hbase有并行处理线程数设置，
由 hbase.regionserver.handler.count (默认值64)参数控制，意味着每个RegionServer可以同时处理64个客户端请求，多余的请求会暂时被积压到callqueue队列中等待，没有直接的限制连接数量的参数，客户端连接一直增多可能会触发进程的句柄数量限制。

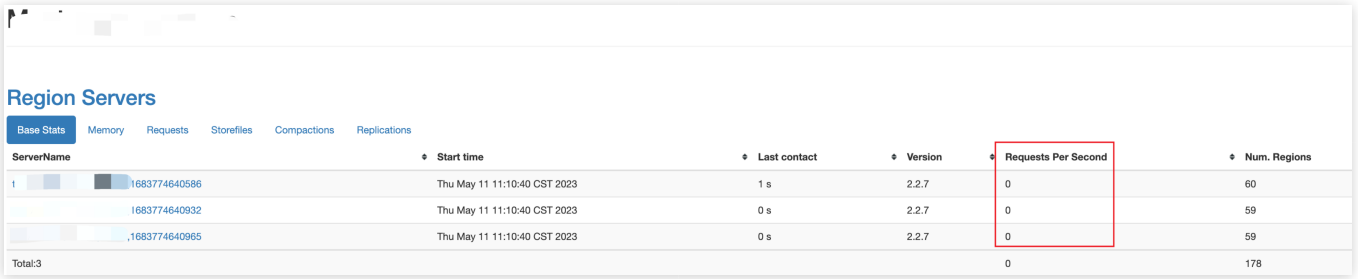


建议测试Regionserver每秒处理请求数比较有参考意义，测试方式：

1. 设置参数，处理get请求的handler数量为300*0.5*(1-0.2)=120个

hbase.regionserver.handler.count=300，
hbase.ipc.server.callqueue.read.ratio=0.5
hbase.ipc.server.callqueue.scan.ratio=0.2
hbase.ipc.server.callqueue.handler.factor=0.1
hbase.ipc.server.max.callqueue.length=1000

2. 停掉集群至只有1台regionserver
3. 并发get查询某测试表的1条数据，查看该regionserver的Requests Per Second值。



Regionserver宕机后，重启后(运行一段时间后)自动退出

File does not exist

- 现象
  存在某个(parent)Region java.io.FileNotFoundException: File does not exist 异常：
- 解决建议
  执行 hbase hbck 检查，看看是否存在 Found lingering reference file 错误，有则执行 hbase hbck -fixReferenceFiles 修复；

存在Full GC耗时很长

- 现象
  分析regionserver的GC日志(与RegionServer日志同一目录，默认在 /data/emr/hbase/logs)，存在Full GC耗时很长时(可能导致读写变慢)，如图：

```
terrytlu@TERRYTLU-MB0 20230526 % grep "pause of approximately" xxx
2023-05-22 13:05:23,362 INFO  [JvmPauseMonitor] util.JvmPauseMonitor: Detected pause in JVM or host machine (eg GC): pause of approximately 1091ms
2023-05-22 13:30:23,341 INFO  [JvmPauseMonitor] util.JvmPauseMonitor: Detected pause in JVM or host machine (eg GC): pause of approximately 1017ms
2023-05-22 13:31:14,071 WARN  [JvmPauseMonitor] util.JvmPauseMonitor: Detected pause in JVM or host machine (eg GC): pause of approximately 43783ms
2023-05-22 13:43:08,481 WARN  [JvmPauseMonitor] util.JvmPauseMonitor: Detected pause in JVM or host machine (eg GC): pause of approximately 45092ms
2023-05-22 13:44:30,190 WARN  [JvmPauseMonitor] util.JvmPauseMonitor: Detected pause in JVM or host machine (eg GC): pause of approximately 42190ms
2023-05-22 14:30:23,345 INFO  [JvmPauseMonitor] util.JvmPauseMonitor: Detected pause in JVM or host machine (eg GC): pause of approximately 1096ms
2023-05-22 14:43:17,788 WARN  [JvmPauseMonitor] util.JvmPauseMonitor: Detected pause in JVM or host machine (eg GC): pause of approximately 46864ms
2023-05-22 14:44:32,837 WARN  [JvmPauseMonitor] util.JvmPauseMonitor: Detected pause in JVM or host machine (eg GC): pause of approximately 46048ms
```

```
   [Eden: 0.0B(38.4G)->0.0B(38.4G) Survivors: 0.0B->0.0B Heap: 95.8G(96.0G)->95.8G(96.0G)]
 [Times: user=19.81 sys=2.78, real=5.01 secs]
2021-05-07T16:07:33.244+0800: 2231.235: [Full GC (Allocation Failure)  95G->31G(96G), 43.6958421 secs]
   [Eden: 0.0B(38.4G)->0.0B(38.4G) Survivors: 0.0B->0.0B Heap: 95.8G(96.0G)->31.3G(96.0G)], [Metaspace: 65727K->65727K(67584K)]
```

- 解决建议
  修改RegionServer内存，hbase-env.sh 的 Heapsize 的大小。

启动报某个协处理器coprocessor类找不到时，RegionServer会退出

1. 分析看看是否是业务自行添加的
2. 修改或添加hbase-site参数跳过coprocessor异常： hbase.coprocessor.abortonerror=false
3. 启动RegionServer

存在 wal.FSHLog: Slow sync cost 告警，发现写WALs慢，可以配置MultiWAL特性增加并行度

1. 修改hbase-site中配置 hbase.wal.provider=multiwal
2. 重启RegionServer服务

kerberos hbase客户端调用需要的最小配置集合是什么

需要以下配置项，具体值可以在HMaster节点的$HBASE_HOME/conf/hbase-site.xml中获取。

"hbase.zookeeper.quorum": "xx1.com,xx2.com,xx3.com",
"zookeeper.znode.parent": "/hbase-secure",
"hadoop.security.authentication":"kerberos",
"hbase.security.authentication" :"kerberos",
"hbase.master.kerberos.principal": "xx",
"hbase.regionserver.kerberos.principal": "xx",
选加：
hbase.client.keytab.principal
hbase.client.keytab.file

hbase shell 如何手动覆盖某个配置

可以通过传递或覆盖 hbase-*.xml 中指定的hbase配置。 在命令行上以-D为前缀的键/值，如下所示：

hbase shell -Dhbase.xxx=xxxx -Dhbase.xxx2=xxxx2

进入shell之后获取配置

```
@shell.hbase.configuration.get("hbase.zookeeper.quorum")
```

如何清空HBase 集群的数据

如果新部署未投产的集群发生了故障，这种时候可以通过清空数据来快速解决掉故障。

1. 停掉HBase服务
2. 删除 /hbase hdfs dfs -mv /hbase /tmp
3. zkcli进入zk客户端之后，删除znode deleteall /hbase-{id}
4. 启动HBase集群
   其他：
   如果开启了kerberos的集群，连接zk时需要认证。
   ${KEYTAB_PATH} 是服务器keytab的存储路径
   ${PRINCIPAL} 是使用klist列出的principal

```
klist -kt ${KEYTAB_PAH}
kinit -kt ${KEYTAB_PAH} ${PRINCIPAL}
```

连接zk服务端：

```
${ZOOKEEPER_HOME}/bin/zkCli.sh -server ${zk ip}:2181
```

snapshot报错 Protocol message was too large

- 现象
  报错：Protocol message was too large. May be malicious. Use CodedInputStream.setSizeLimit() to increase the size limit.
- 可能原因
  表数据量很大，HFile很多导致生成的快照信息过大，在校验snapshot信息的时候超过了Protobuf默认的大小64MB。
- 解决方案

1. 建议在业务空闲时间段，通过major_compact命令分批合并表的region，文件数量减少后该问题就可以解决。
2. 直接报错可以通过在hbase-site中增加参数 snapshot.manifest.size.limit=268435456 (256MB)， 然后重启HMaster解决
3. 然后可能会遇到snapshot超时的报错，需要增加参数，然后重启HMaster解决 hbase.snapshot.master.timeout.millis=1800000（30分钟）

如何查看major compaction进度

如果有个超大的region，做了major_compact ${region id}之后不知道进展。

解决方案：

可以在hbase master ui上面看到对应的表的compaction状态是Major。

1. 然后在后台可以看到对应region的.tmp目录大小一直在增长，compaction合并之后的HFile放在这里了，可以根据增长速度估计进度。
2. RegionServer有个接口可以获取要处理多少KV，已经处理了多少KV，写程序获取。

```
@Override
public List<RegionMetrics> getRegionMetrics(ServerName serverName, TableName tableName)
    throws IOException {
  AdminService.BlockingInterface admin = this.connection.getAdmin(serverName);
  HBaseRpcController controller = rpcControllerFactory.newController();
  AdminProtos.GetRegionLoadRequest request =
    RequestConverter.buildGetRegionLoadRequest(tableName);
  try {
    return admin.getRegionLoad(controller, request).getRegionLoadsList().stream()
      .map(RegionMetricsBuilder::toRegionMetrics).collect(Collectors.toList());
  } catch (ServiceException se) {
    throw ProtobufUtil.getRemoteException(se);
  }
}
```

Phoenix开发
概述

Phoenix 查询引擎支持使用 SQL 进行 HBase 数据的查询，会将 SQL 查询转换为一个或多个 HBase API，协同处理器与自定义过滤器的实现，并编排执行。使用 Phoenix 进行简单查询，其性能量级是毫秒，对于百万级别的行数来说，其性能量级是秒。TBDS 中包含 HBase 组件的集群，默认集成 Phoenix引擎客户端。

示例工程开发

环境准备

开发环境准备（Java/Scala）

准备项	说明
安装JDK	JDK 8 ，推荐使用 konaJDK，下载地址
安装 Scala	Scala 2.12，下载地址
安装和配置 IDE	按需选择，比如 IntelliJ IDEA 或 Eclipse
安装 Maven	开发环境基础配置，负责构建 Java 应用程序
Maven 配置准备	如果需要本地调试，需要参考开发环境准备，配置 Maven pom.xml，推荐 Maven 3.6.3，下载地址

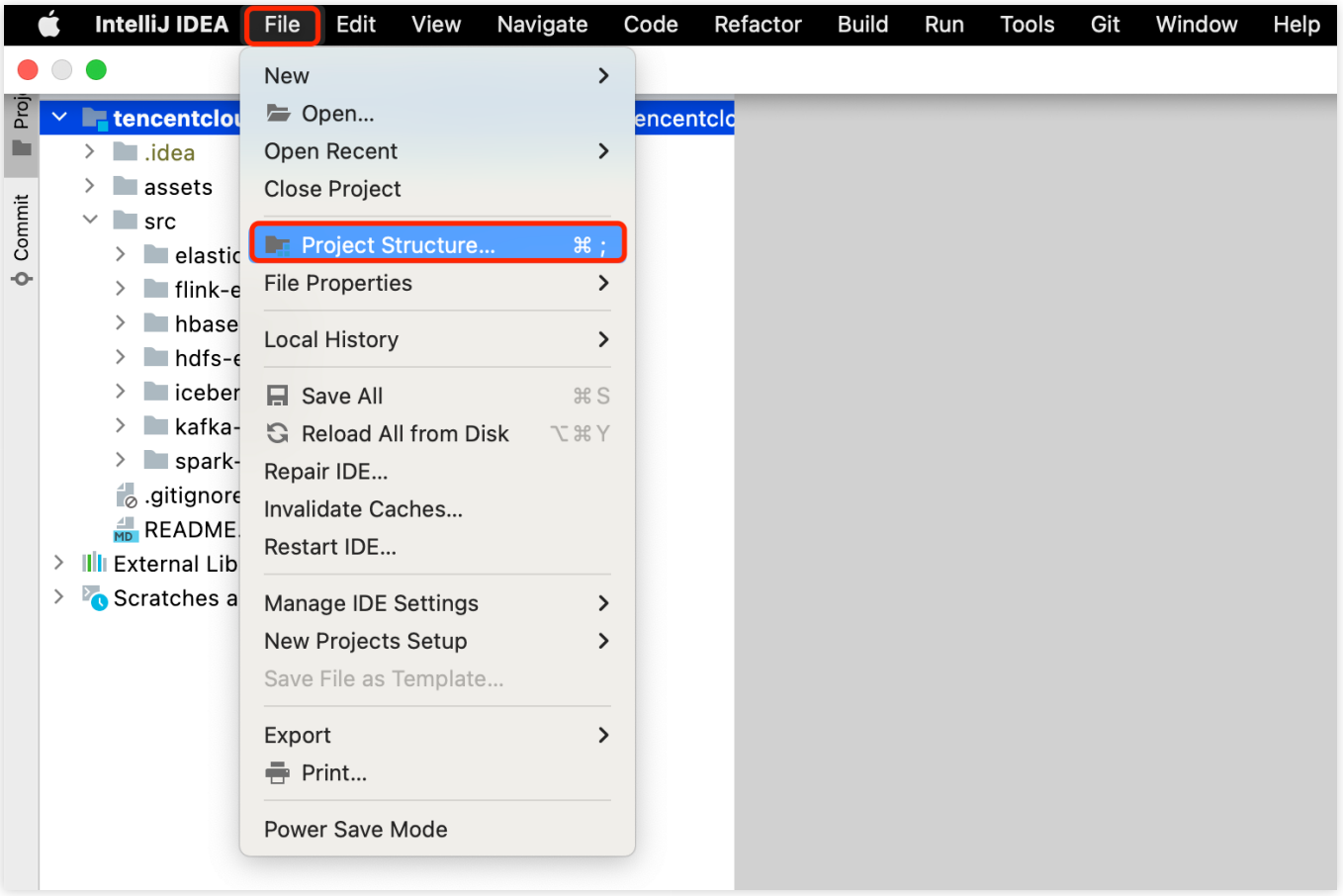
远程调试环境准备

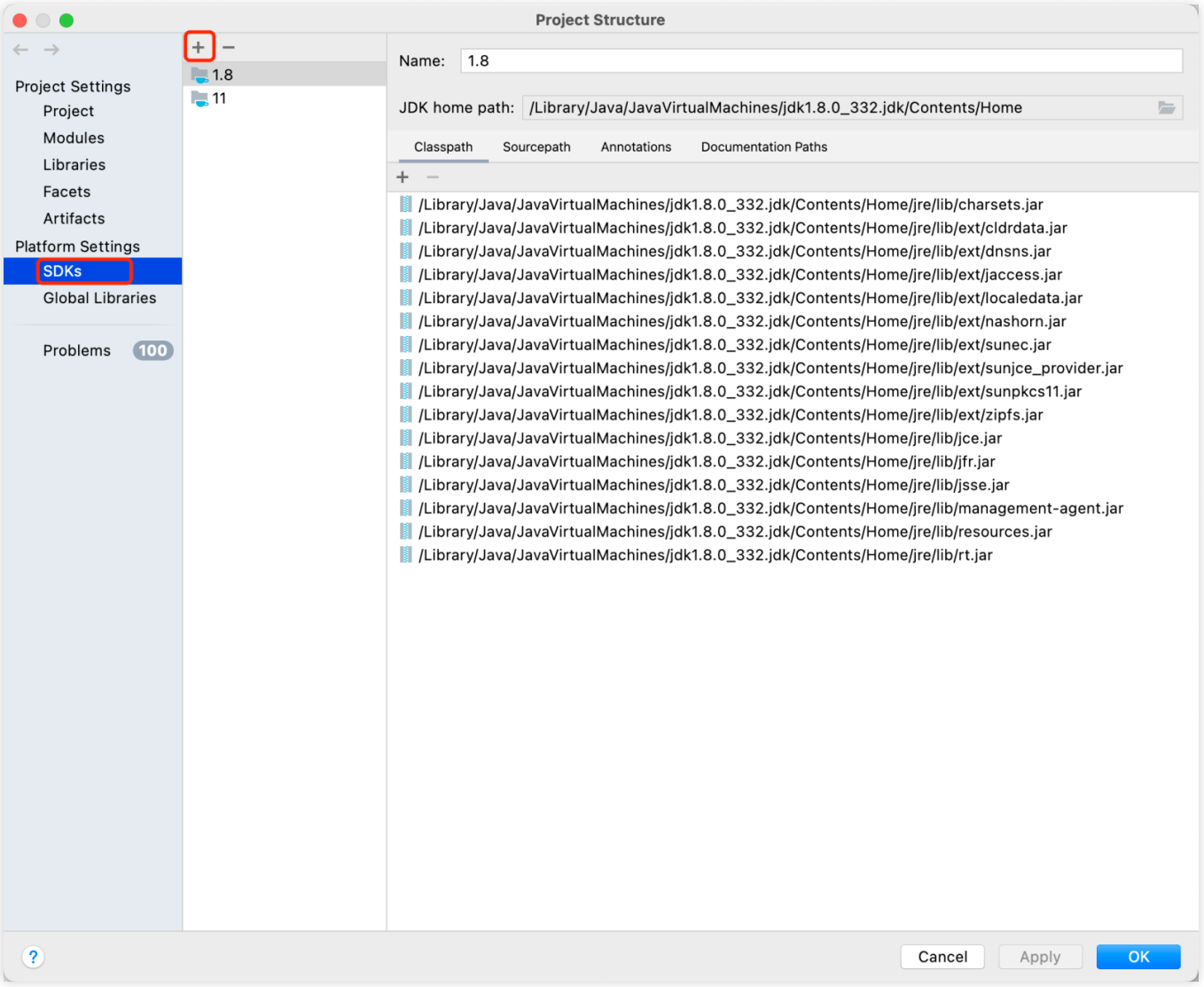
1. 以下使用 Linux 环境作为开发机进行应用调试说明。
2. 准备开发机（可选）：建议使用 Linux 操作系统
3. 部署客户端：部署TBDS客户端，在开发机上执行客户端部署。

导入示例工程代码

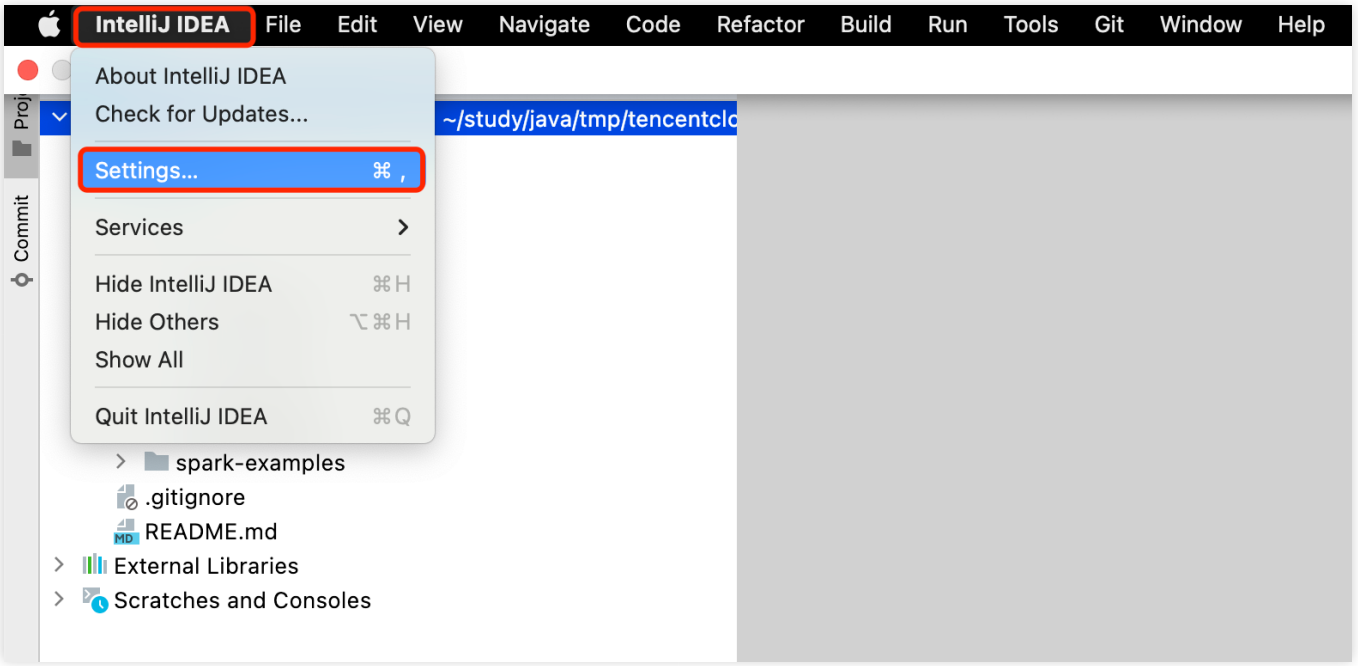
以下以 IntelliJ IDEA 举例，将示例工程代码导入进行说明。

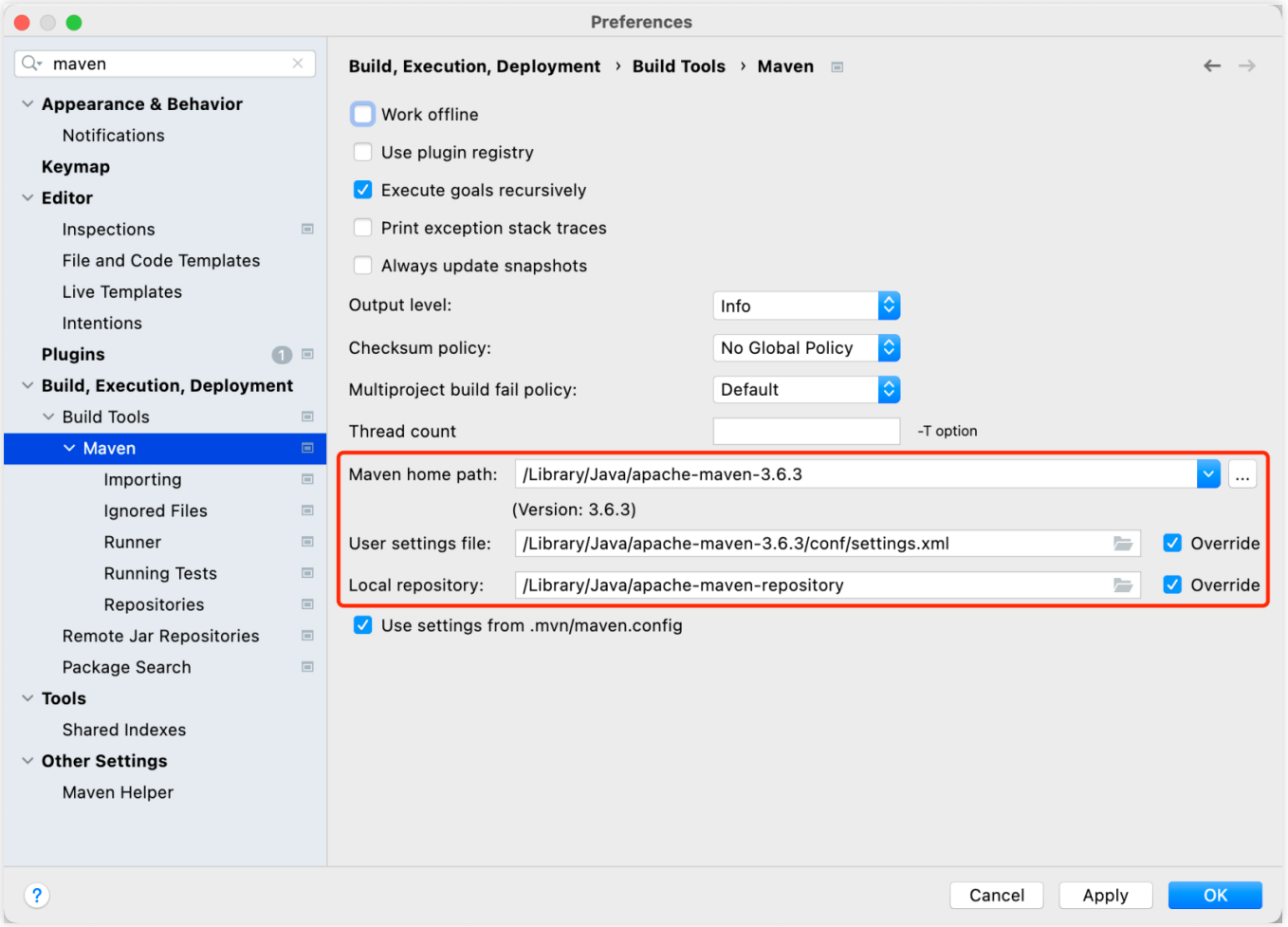
1. 下载样例代码：git clone https://e.coding.net/g-necm8077/tencentcloud-tbds-examples/tbds-examples.git
2. 导入项目：安装完 IntelliJ IDEA 和 JDK 工具后，导入样例工程到 IntelliJ IDEA 开发环境。点击 Open 后，选择上一步下载的项目地址打开。
3. IDEA 配置 JDK：首先确保在本地安装了 JDK 1.8，并配置好了环境变量。IDEA 选择 File 下的 Project Structure，点击 SDKs，选择 JDK 1.8，点击 Apply，再点击 OK。若没有 JDK 1.8，则点击 + 号进行添加，点击 Add JDK 后选择 JDK 1.8 安装目录，然后点击 OK 即可。


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
New
Open...
Open Recent
Close Project
Project Structure... ⌘ ;
File Properties
Local History
Save All ⌘S
Reload All from Disk ⌥⌘Y
Repair IDE...
Invalidate Caches...
Restart IDE...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
Commit
.idea
assets
src
.gitignore
README
External Lib
Scratches a



4. IDEA 配置 Maven：首先确保在本地安装了 Maven，并配置好了环境变量和 settings.xml 文件。IDEA 点击 Settings 进入配置页面，左上角输入 Maven 进行搜索，点击 Build Tools 下的 Maven 配置项，修改 “Maven home path” 为本地 Maven 的安装目录，修改 “User settings file” 为本地 Maven settings.xml 配置文件的文件路径，并勾选 Override，此时 “Local repository” 将自动设置为 settings.xml 文件中配置的本地 Maven 仓库的目录。最后点击 Apply，再点击 OK。


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
About IntelliJ IDEA
Check for Updates...
Settings... ⌘ ,
Services
Hide IntelliJ IDEA ⌘H
Hide Others ⌥⌘H
Show All
Quit IntelliJ IDEA ⌘Q
~/study/java/tmp/tencentclo
Proj
Commit
spark-examples
.gitignore
README.md
External Libraries
Scratches and Consoles


Preferences
maven
Appearance & Behavior
Notifications
Keymap
Editor
Inspections
File and Code Templates
Live Templates
Intentions
Plugins
Build, Execution, Deployment
Build Tools
Maven
Importing
Ignored Files
Runner
Running Tests
Repositories
Remote Jar Repositories
Package Search
Tools
Shared Indexes
Other Settings
Maven Helper
Build, Execution, Deployment › Build Tools › Maven
Work offline
Use plugin registry
Execute goals recursively
Print exception stack traces
Always update snapshots
Output level: Info
Checksum policy: No Global Policy
Multiproject build fail policy: Default
Thread count
-T option
Maven home path: /Library/Java/apache-maven-3.6.3
(Version: 3.6.3)
User settings file: /Library/Java/apache-maven-3.6.3/conf/settings.xml
Override
Local repository: /Library/Java/apache-maven-repository
Override
Use settings from .mvn/maven.config
Cancel
Apply
OK

代码逻辑说明

功能说明

本教程演示的是：开发Phoenix SQL查询应用示例，并将查询结果打印出来。

代码逻辑说明

创建Phoenix Driver连接，如果url为jdbc:pheonix，就会从hbase-site.xml中查找相关的连接配置比如zk地址、znode路径、kerberos认证信息等。

```
String url = "jdbc:phoenix";
Connection conn = DriverManager.getConnection(url);
```

hbase-site.xml必须在Java进程的classpath中，TBDS提供的样例代码中start.sh启动时会设置classpath，如果是自行搭建的工程需要注意一下。

执行SQL语句

```
// 这里执行SQL语句
String sql = args[0];
Statement statement = conn.createStatement();
ResultSet rs = statement.executeQuery(sql);
// 这里打印查询到的数据
printResultSet(rs);
```

遍历打印查询到的数据

```
ResultSetMetaData md = rs.getMetaData();//获取键名
int columnCount = md.getColumnCount();//获取行的数量
while (rs.next()) {
    StringBuilder line = new StringBuilder();
    for (int i = 1; i <= columnCount; i++) {
        line.append(" ").append(md.getColumnName(i)).append("=").append(rs.getObject(i));
    }
    LOG.info(line.());//获取键名及值
}
```

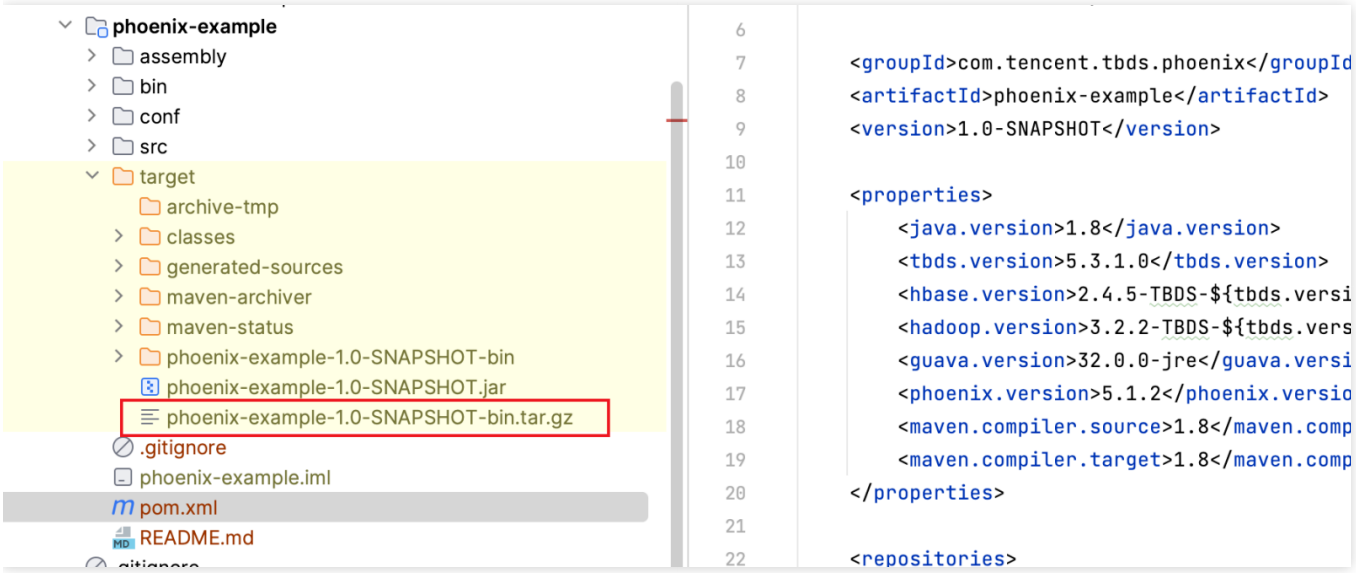
打包发布

打包

以下使用 IntelliJ IDEA 说明示例工程代码编译过程。点击 IDEA 下方 Terminal 打开终端，切换到示例工程的 HBase 工程目录下，然后使用命令 mvn clean package 对工程进行打包，运行过程中可能还需要下载一些文件，直到出现 BUILD SUCCESS 表示打包成功。

```
cd src/phoenix-examples/phoenix-example/
mvn clean package
```

通过上述编译打包后，将在工程目录下 target 文件夹中看到打好的 tar.gz 包，如图示中的 hbase-example-1.0-SNAPSHOT-bin.tar.gz，这里面包含了样例程序以及运行过程中所需的所有依赖，大小在140MB左右。



开发机运行

1. 准备用户：参考获取用户认证。若是 Kerberos 环境，需要将用户的 keytab 文件下载到本地，然后上传至开发机。这里将 test 用户的 test.keytab 文件上传至开发机的 /tmp 目录。
2. 将集群的配置文件hbase-site.xml放在解压后的conf目录下，如果是集群内的节点，直接执行：cp /usr/local/service/hbase/conf/hbase-site.xml conf/ 即可。
3. 安全认证：将principal和keytab信息放入hbase-site.xml中，增加这3个对应配置，phoenix客户端会读取对应配置自动完成认证处理（注意这里principal和keytab配置项名称和hbase-example中的不一致）

```
 <property>
   <name>hadoop.security.authentication</name>
   <value>kerberos</value>
</property>
<property>
   <name>hbase.myclient.principal</name>
   <value>*******</value>
```

```
</property>
<property>
  <name>hbase.myclient.keytab</name>
  <value>/var/krb5kdc/emr.keytab</value>
</property>
```

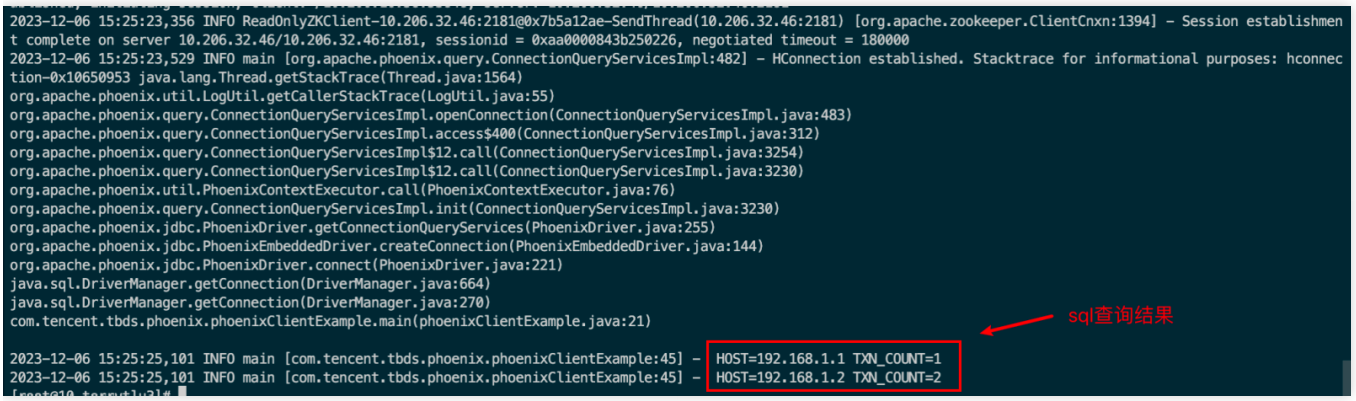
4. Ranger 授权：参考Ranger授权（经典集群），确保 test 用户具有创建/写入对应表的权限，phoenix表和hbase表一一对应，这里需要授权对应hbase表的权限。
5. 运行样例

   设置要运行的类变量： export MAIN_CLASS=com.tencent.tbds.phoenix.phoenixClientExample

   执行测试: sh start.sh "select * from TEST limit 10"

程序运维监控

控制台以及logs/client.log都会打印运行期间的日志



示例说明

- 启动客户端
  切换成 hadoop 用户，进入/usr/local/service/hbase/phoenix-client/bin 目录，使用 Phoenix 的 Python 命令行工具：

./sqlline.py

执行成功后显示：

```
[hadoop@172 /usr/local/service/hbase/phoenix-client/bin]$ ./sqlline.py
Setting property: [incremental, false]
Setting property: [isolation, TRANSACTION_READ_COMMITTED]
issuing: !connect -p driver org.apache.phoenix.jdbc.PhoenixDriver -p user "none" -p password "none" "jdbc:phoenix:"
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/local/service/hbase/phoenix-client/phoenix-client-hbase-2.4-5.1.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/service/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Connecting to jdbc:phoenix:
Connected to: Phoenix (version 5.1)
Driver: PhoenixEmbeddedDriver (version 5.1)
Autocommit status: true
Transaction isolation: TRANSACTION_READ_COMMITTED
sqlline version 1.9.0
0: jdbc:phoenix:>
```

Phoenix 引擎支持使用 SQL 进行数据查询，一些常见操作如下：

- 创建表

```
0: jdbc:phoenix:> CREATE TABLE IF NOT EXISTS TEST (
 host char(50) not null,
 txn_count bigint
 CONSTRAINT pk PRIMARY KEY (host)
);
```

- 插入数据

```
0: jdbc:phoenix:>UPSERT INTO TEST(host,txn_count) VALUES('192.168.1.1',1);
0: jdbc:phoenix:>UPSERT INTO TEST(host,txn_count) VALUES('192.168.1.2',2);
```

- 查询数据

```
0: jdbc:phoenix:>SELECT * FROM TEST;
```

- 删除数据表

```
0: jdbc:phoenix:>DROP TABLE IF EXISTS TEST;
```

api接口

sql语法官方链接：Grammar | Apache Phoenix

常用命令

- help命令，查看命令列表。

!help

- 查看当前库中存在的表

!tables

注意：

- 如果不加双引号，会自动将小写转为大写。
- phoenix表名区分大小写
- 创建表

CREATE TABLE IF NOT EXISTS 表名(ID varchar not null primary key, NAME VARCHAR);

- 查看表结构

!describe 表名

注意：phoenix/hbase对表名、字段名都是大小写敏感，如果直接写小写字母，不加双引号，则默认会被转换成大写字母。

- 修改表配置

ALTER TABLE 表名 SET UPDATE_CACHE_FREQUENCY=60000,COMPRESSION='SNAPPY';

- 添加二级索引

create index if not exists 索引(表)名 on 表名(NAME) IMMUTABLE_ROWS=false, VERSIONS=1, DATA_BLOCK_ENCODING='FAST_DIFF',COMPRESSION='GZ',bloomfilter='ROW';
demo:
create index if not exists test_table_index on test_table(NAME) IMMUTABLE_ROWS=false, VERSIONS=1, DATA_BLOCK_ENCODING='FAST_DIFF',COMPRESSION='GZ',bloomfilter='ROW';

- 删除索引

drop index 索引(表)名 on 表名;

- 插入、更新。
  Phoenix中不存在update的语法关键字，而是upsert ，功能上替代了Insert+update

upsert into 表名(id,name) values('1','test');

- 查询

select count(1) from 表名 where NAME like '1%';

- 删除表

drop table 表名;

- 退出

!quit

常见问题

phoenix索引状态变DISABLE修复

1. 修复处于DISABLED状态的索引：
   在phoenix客户端将索引状态置为UNUSABLE，这样索引挂掉之前的数据可以跳过，减少修复数据量:

```
ALTER INDEX IF EXISTS METADATA_AE_HIVE_COLUMN_QN_INDEX ON METADATA.AE_HIVE_COLUMN UNUSABLE;
```

如果上一步报错了，某些情况下不允许设置为UNUSABLE，那需要设置为REBUILD：

```
ALTER INDEX IF EXISTS METADATA_AE_HIVE_COLUMN_QN_INDEX ON METADATA.AE_HIVE_COLUMN REBUILD;
```

但是如果大表的话，REBUILD会超时，不过没关系，执行下一步通过MR任务生成索引。

2. 通过MR任务生成索引数据并导入：
   执行认证，tbds认证或者kerberos认证等，可以用hdfs做认证，不会有权限问题，然后执行下面命令，用后台挂起方式进行，因为运行时间比较长，终端非常可能会断掉引起异常：

```
nohup hbase org.apache.phoenix.mapreduce.index.IndexTool -Dmapreduce.reduce.java.opts="-Xmx8192m" -Dmapreduce.reduce.memory.mb=8500 --schema METADATA --data-table AE_HIVE_COLUMN --index-table METADATA_AE_HIVE_COLUMN_QN_INDEX --output-path /tmp/hbase/indexes/ > log 2>&1 &
```

Phoenix的!tables命令看不到索引表状态

使用!tables命令看不到INDEX STATE这一列。

原因分析：
sqlline中的outputformat默认为table类型，这种比较占用横向空间，如果列数太多有些就会隐藏。
处理方式1：
设置宽度!set maxwidth 3000，一行数据会变成两行显示。
处理方式2：
更改outputformat为csv类型看到，命令：!outputformat csv。
之后再执行!tables就可以看到很多列了，只不过没有table类型美观。

outputformat可选类型：

outputformat <table/vertical/xmlattr/xmlelements/csv>

【作用】设置数据展示格式，默认为table格式。

【例子】

table	0: jdbc:phoenix> !outputformat table 0: jdbc:phoenix> select * from tab2; +----+-----+ | K | C1 | +----+-----+ | 1 | 2 | +----+-----+ 1 row selected (0.029 seconds)
vertical	0: jdbc:phoenix> !outputformat vertical 0: jdbc:phoenix> select * from tab2; K 1 C1 2 1 row selected (0.016 seconds)
xmlattr	0: jdbc:phoenix> !outputformat xmlattr 0: jdbc:phoenix> select * from tab2; <resultset> <result K="1" C1="2"/> </resultset> 1 row selected (0.017 seconds)
xmlelements	0: jdbc:phoenix> !outputformat xmlelements 0: jdbc:phoenix> select * from tab2; <resultset> <result> <K>1</K> <C1>2</C1> </result> </resultset> 1 row selected (0.02 seconds)
csv	0: jdbc:phoenix> !outputformat csv 0: jdbc:phoenix> select * from tab2; 'K','C1' '1','2' 1 row selected (0.02 seconds)

phoenix索引表的region分区不正常

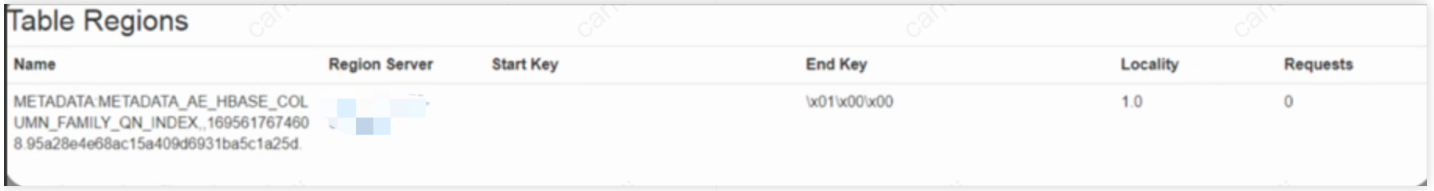


Table Regions

Name	Region Server	Start Key	End Key	Locality	Requests
METADATA.METADATA_AE_HBASE_COLUMN_FAMILY_QN_INDEX,,1695617674608.95a28e4e68ac15a409d6931ba5c1a25d.	[redacted]		\x01\x00\x00	1.0	0

因为索引表是没有数据的，truncate '表名'清空之后，可以通过修复索引再恢复数据。

Kafka开发

概述

Kafka是一个分布式的消息发布-订阅系统。它采用独特的设计提供了类似JMS的特性，主要用于处理活跃的流式数据。
Kafka有很多适用的场景：消息队列、行为跟踪、运维数据监控、日志收集、流处理、事件溯源、持久化日志等。

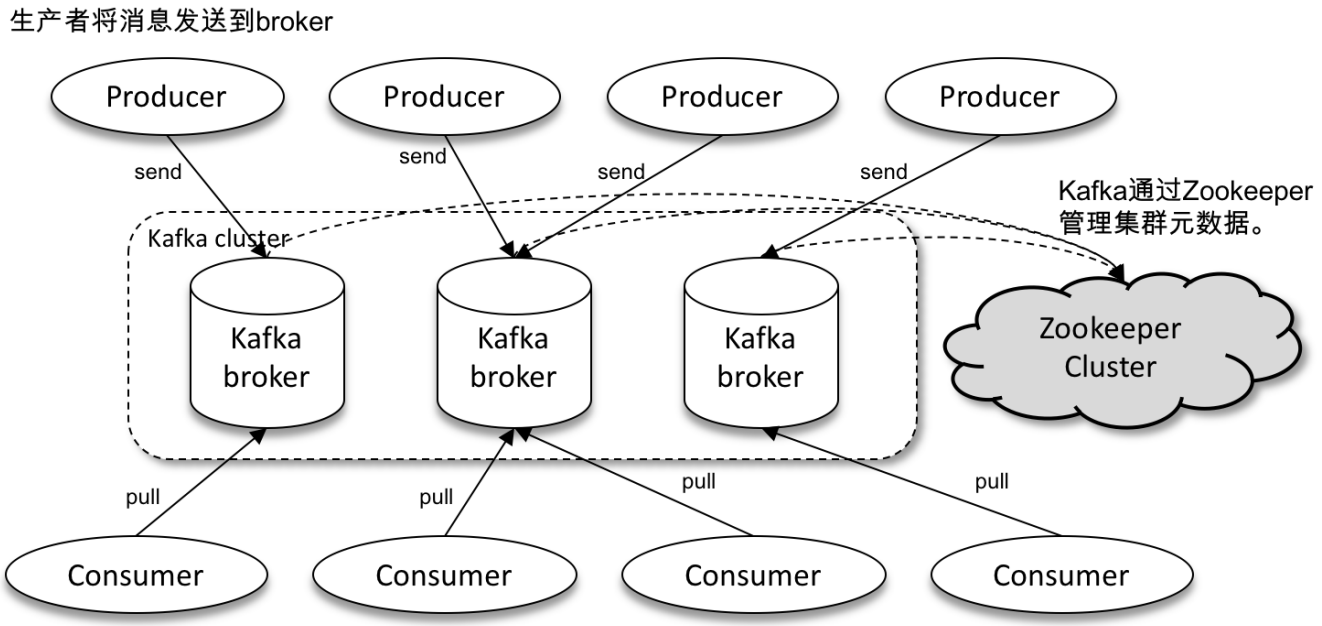
Kafka特点

Kafka有如下几个特点：

- 高吞吐量
- 消息持久化到磁盘
- 分布式系统易扩展
- 容错性好
- 支持online和offline场景

Kafka架构

kafka集群架构如下图所示：



共包含四个角色：

（1）Producer：发送消息的Kafka客户端

（2）Broker：kafka服务端

（3）Consumer：消费消息的Kafka客户端

（4）Zookeeper：Kafka服务元数据管理器

Kafka基本概念

主要有以下基本概念：

（1）Topic：Kafka维护的同一类的消息称为一个Topic。

（2）Partition：每一个Topic可以被分为多个Partition，每个Partition对应一个可持续追加的、有序不可变的log文件。

示例工程开发

环境准备

参考 spark开发->示例工程开发配置 环境

> 禁止：
>
> 严格禁止使用低版本kafka操作高版本的kafka的topic，如修改分区数和增加副本等操作。由此不合规操作导致的集群组件异常等后果，不属于TBDS责任范围（参考产品SLA），需自行承担相关责任！

代码逻辑说明

1. 功能说明。
   本示例演示创建和删除topic等基本操作

2. POM中添加依赖。

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-clients</artifactId>
   <version>2.8.2</version>
</dependency>
```

3. 核心代码示例。

```
import org.apache.kafka.clients.admin.*;
import org.apache.kafka.common.Node;
import org.apache.kafka.common.TopicPartitionInfo;

import java.util.*;
import java.util.concurrent.ExecutionException;

public class KafkaClientExample {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        System.setProperty("java.security.auth.login.config", "/usr/local/service/kafka/config/kafka-gssap
```

```
i-jaas.conf");
        System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
        Properties props = new Properties();
        //kafka集群broker list
        props.put("bootstrap.servers", "x.x.x.x:9092");
        props.put("acks", "all");
        // kerberos认证
        props.put("security.protocol", "SASL_PLAINTEXT");
        props.put("sasl.mechanism", "GSSAPI");
        props.put("sasl.kerberos.service.name", "hadoop");
        props.put("sasl.jaas.config", "com.sun.security.auth.module.Krb5LoginModule required useKeyTab
=true storeKey=true keyTab=\"/var/krb5kdc/emr.keytab\" principal=\"xxxx/x.x.x.x@TBDS-XXXX\";");
        // 实例化adminClient
        AdminClient adminClient = AdminClient.create(props);
        // 创建topic
        NewTopic newTopic = new NewTopic("test_topic_001",4, (short) 1);
        Collection<NewTopic> newTopicList = new ArrayList<>();
        newTopicList.add(newTopic);
        adminClient.createTopics(newTopicList);
        // topic list
        System.out.println();
        ListTopicsResult result = adminClient.listTopics();
        Collection<TopicListing> topics = result.listings().get();
        topics.forEach(each -> System.out.println(each.name()));
        // describe topic
        Collection<String> topicNames = new ArrayList<>();
        topicNames.add("test_topic_001");
        DescribeTopicsResult describeTopicsResult = adminClient.describeTopics(topicNames);
        Map<String, TopicDescription> map = describeTopicsResult.all().get();
        for (Map.Entry<String, TopicDescription> entry : map.entrySet()) {
            System.out.println("topicName:" + entry.getValue().name()); //当前topic的名字
            System.out.println("partition num:" + entry.getValue().partitions().size()); //当前topic的partition
数量
            System.out.println("listp:");
            List<TopicPartitionInfo> listp = entry.getValue().partitions(); //拿到topic的partitions相关信息
            for (TopicPartitionInfo info : listp) {
                System.out.println("--------------------------------");
                System.out.println("info.partition():" + info.partition());
                System.out.println("info.leader().id():" + info.leader().id()); //领导者所在机器id，也就是机器编号
配置文件中的service id
                System.out.println("info.leader().host():" + info.leader().host()); //领导者所在机器host ip
                System.out.println("info.leader().port():" + info.leader().port()); //领导者所在机器port
                System.out.println("info.replicas():" + info.replicas());  //副本的信息，有多少会拿到多少
                List<Node> listInfo = info.replicas();
                //输出node信息
                for (Node n : listInfo) {
                    System.out.println("info.id():" + n.id()); //副本所在的node id
```

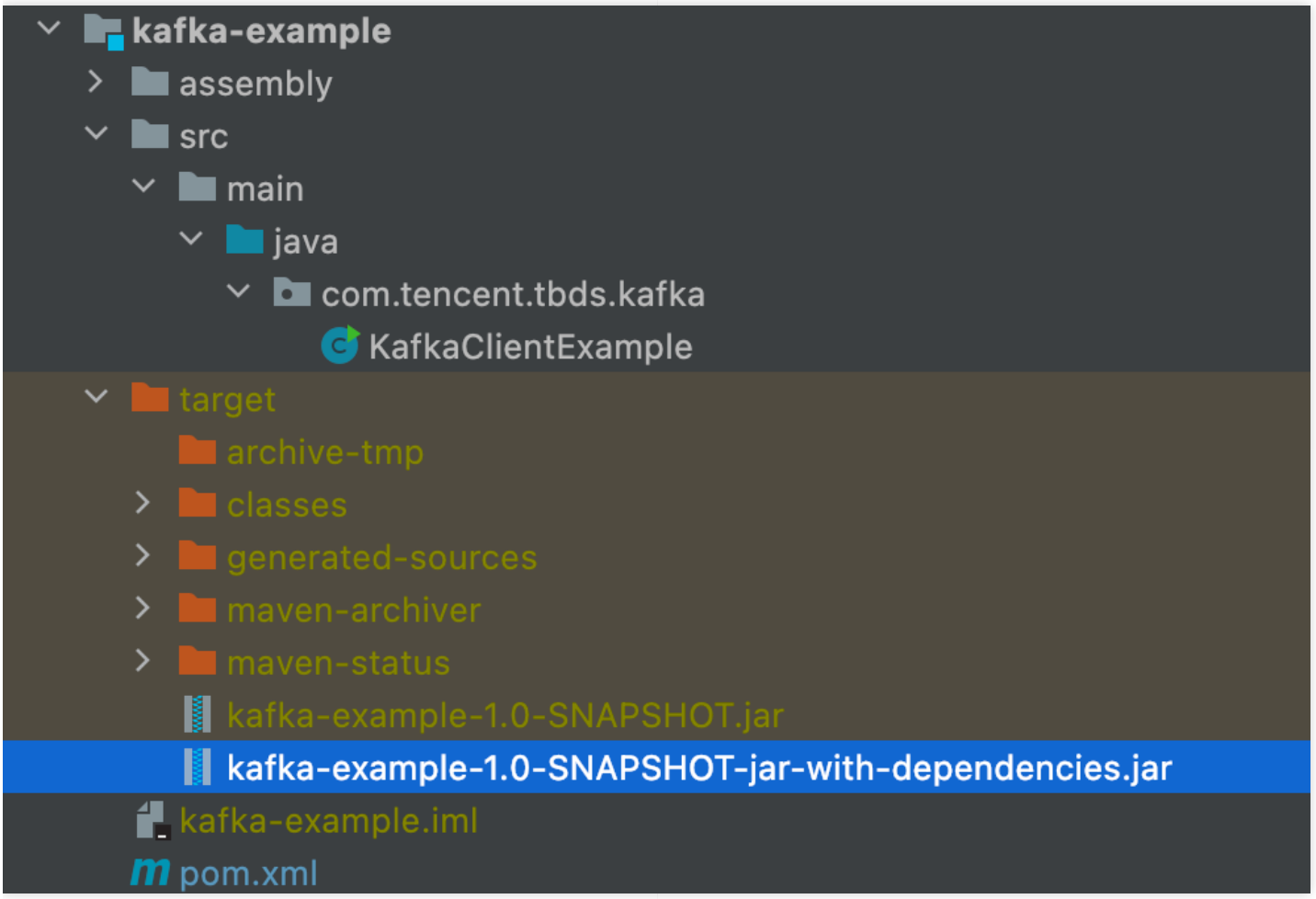
```
                System.out.println("info.host():" + n.host()); //副本所在node的host ip
                System.out.println("info.port():" + n.port()); //副本所在node的port
            }
        }
    }
    // delete topic
    adminClient.deleteTopics(topicNames);
    // topic list
    System.out.println();
    result = adminClient.listTopics();
    topics = result.listings().get();
    topics.forEach(each -> System.out.println(each.name()));
    // close
    adminClient.close();
  }
}
```

示例说明

示例编译运行

1. Linux编译参考。

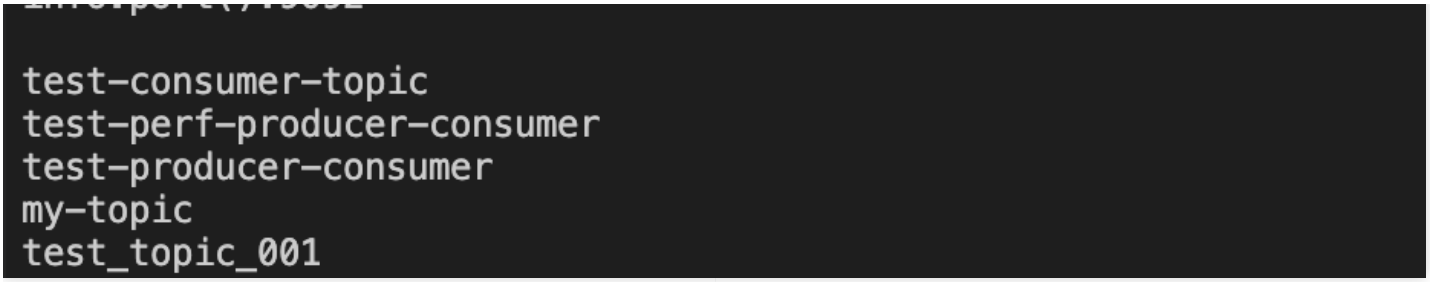
```
//编译
/usr/local/jdk/bin/javac -classpath kafka-clients-2.8.2.jar KafkaClientExample.java
//打包
/usr/local/jdk/bin/jar -cvf KafkaClientExample.jar KafkaClientExample.class

//建议使用idea编译打包fat jar
//参考示例工程 kafka-examples/kafka-example 打包出 kafka-example-1.0-SNAPSHOT-jar-with-dependencies.jar
mvn clean package
```



2. 测试运行。

//运行
/usr/local/jdk/bin/java -classpath kafka-example-1.0-SNAPSHOT-jar-with-dependencies.jar com.tencent.tbds.kafka.KafkaClientExample

```
test-consumer-topic
test-perf-producer-consumer
test_topic_001001
test-producer-consumer
my-topic
test_topic_001
topicName:test_topic_001001
partition num:4
listp:
---------------------------------
info.partition():0
info.leader().id():1
info.leader().host():10.206.17.14
info.leader().port():9092
info.replicas():[10.206.17.14:9092 (id: 1 rack: null)]
info.id():1
info.host():10.206.17.14
info.port():9092
---------------------------------
info.partition():1
info.leader().id():2
info.leader().host():10.206.16.170
info.leader().port():9092
info.replicas():[10.206.16.170:9092 (id: 2 rack: null)]
info.id():2
info.host():10.206.16.170
info.port():9092
---------------------------------
info.partition():2
info.leader().id():3
info.leader().host():10.206.17.144
info.leader().port():9092
info.replicas():[10.206.17.144:9092 (id: 3 rack: null)]
info.id():3
info.host():10.206.17.144
info.port():9092
---------------------------------
info.partition():3
info.leader().id():1
info.leader().host():10.206.17.14
info.leader().port():9092
info.replicas():[10.206.17.14:9092 (id: 1 rack: null)]
info.id():1
info.host():10.206.17.14
info.port():9092
```


test-consumer-topic
test-perf-producer-consumer
test-producer-consumer
my-topic
test_topic_001

api接口

api接口主要有：AdminClient (kafka 2.8.1 API)、KafkaProducer (kafka 2.8.1 API)、KafkaConsumer (kafka 2.8.1 API)

java api

生产者java api使用示例

```
//生产
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class KafkaJavaApiClientExample {

    public static void main(String[] args) {
        System.setProperty("java.security.auth.login.config", "/usr/local/service/kafka/config/kafka-gssapi-jaas.conf");
        System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
        Properties props = new Properties();
        props.put("bootstrap.servers", "x.x.x.x:9092");//kafka集群，broker-list
        props.put("acks", "all");
        props.put("retries", 1);//重试次数
        props.put("batch.size", 16384);//批次大小
        props.put("linger.ms", 1);//等待时间
        props.put("buffer.memory", 33554432);//RecordAccumulator缓冲区大小
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        // kerberos认证
        props.put("security.protocol", "SASL_PLAINTEXT");
        props.put("sasl.mechanism", "GSSAPI");
        props.put("sasl.kerberos.service.name", "hadoop");
        props.put("sasl.jaas.config", "com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true storeKey=true keyTab=\"/var/krb5kdc/emr.keytab\" principal=\"xxxx/x.x.x.x@TBDS-XXXX\";");

        // 创建KafkaProducer客户端
        KafkaProducer<String, String> producer = new KafkaProducer<>(props);
        for (int i = 0; i < 10 ; i++) {
            producer.send(new ProducerRecord<>("test_topic_001","ImKey-"+i,"ImValue-"+i));
        }
```

```
        // 关闭资源
        producer.close();
    }
}
```

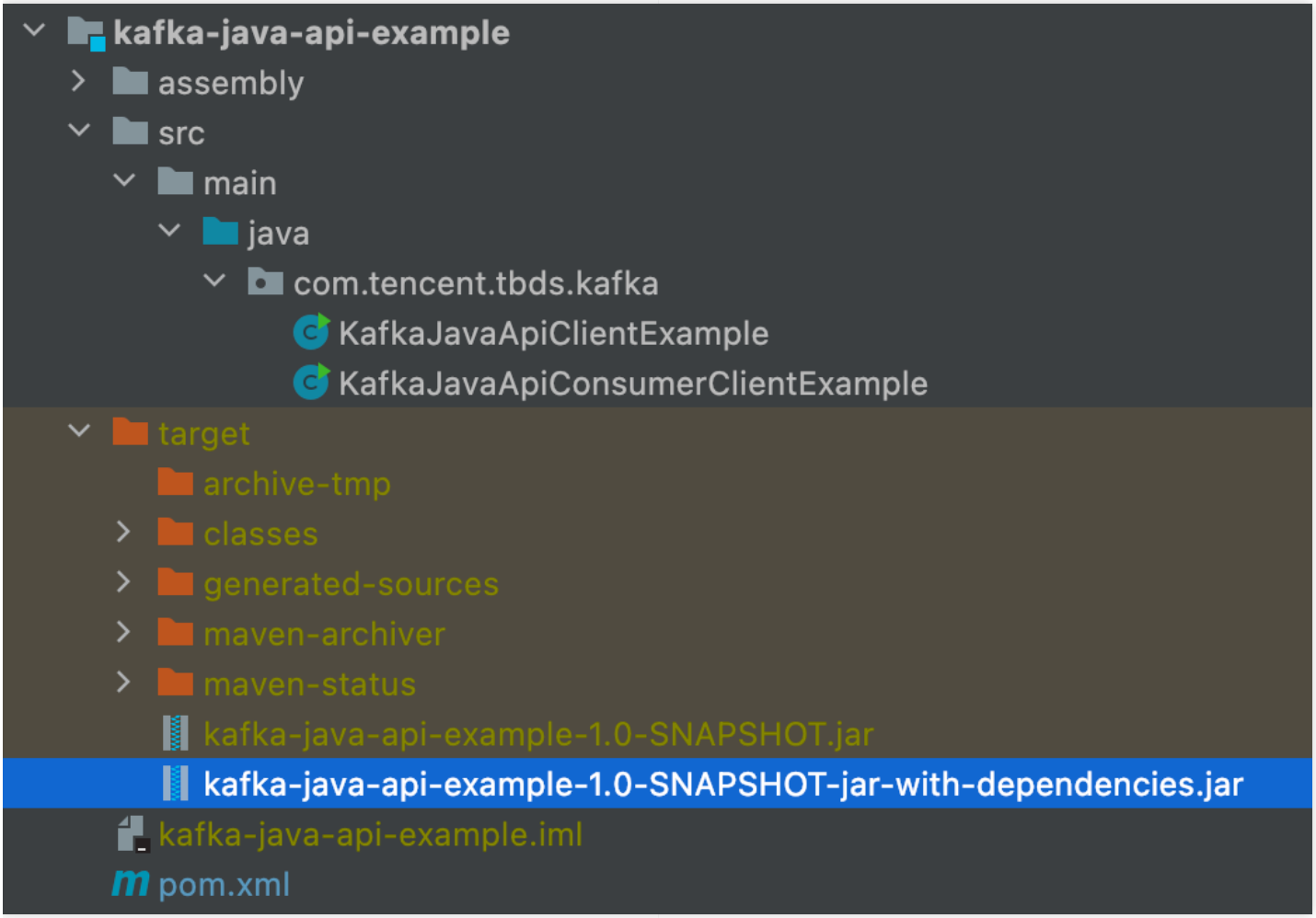
```
//编译
/usr/local/jdk/bin/javac -classpath kafka-clients-2.8.2.jar KafkaJavaApiClientExample.java
//打包
/usr/local/jdk/bin/jar -cvf KafkaJavaApiClientExample.jar KafkaJavaApiClientExample.class

//建议使用idea编译打包fat jar
//参考示例工程 kafka-examples/kafka-java-api-example 打包出 kafka-java-api-example-1.0-SNAPSHOT-jar-with-dependencies.jar
mvn clean package

//运行
/usr/local/jdk/bin/java -classpath kafka-java-api-example-1.0-SNAPSHOT-jar-with-dependencies.jar com.tencent.tbds.kafka.KafkaJavaApiClientExample
```

消费者java api使用示例

```
//消费
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.Arrays;
import java.util.Properties;

public class KafkaJavaApiConsumerClientExample {

    public static void main(String[] args) throws InterruptedException {
        System.setProperty("java.security.auth.login.config", "/usr/local/service/kafka/config/kafka-gssapi-jaas.conf");
        System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
        Properties props = new Properties();
        props.put("bootstrap.servers", "x.x.x.x:9092");//kafka集群，broker-list
        props.put("group.id", "test-consumer-group111");//消费者组，只要group.id相同，就属于同一个消费者组
        props.put("enable.auto.commit", "true");//自动提交offset
        props.put("auto.commit.interval.ms", "1000"); // 自动提交时间间隔
        props.put("max.poll.records", "1000"); // 拉取的数据条数
        props.put("session.timeout.ms", "10000"); // 维持session的时间。超过这个时间没有心跳 就会剔出消费者组
```

```
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("auto.offset.reset", "earliest");
        // kerberos认证
        props.put("security.protocol", "SASL_PLAINTEXT");
        props.put("sasl.mechanism", "GSSAPI");
        props.put("sasl.kerberos.service.name", "hadoop");
        props.put("sasl.jaas.config", "com.sun.security.auth.module.Krb5LoginModule required useKeyTab
=true storeKey=true keyTab=\"/var/krb5kdc/emr.keytab\" principal=\"xxxx/x.x.x.x@TBDS-XXXX\";");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        // 可以写多个topic
        consumer.subscribe(Arrays.asList("test_topic_001"));

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(5000);
            if (records.isEmpty()) {
                System.out.println("没有需要消费的数据了！");
                break;
            }
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), recor
d.value());
            }
            Thread.sleep(5000L);
            System.out.println("处理了一批数据！");
        }
    }
}
```

```
//编译
/usr/local/jdk/bin/javac -classpath kafka-clients-2.8.2.jar KafkaJavaApiConsumerClientExample.java
//打包
/usr/local/jdk/bin/jar -cvf KafkaJavaApiConsumerClientExample.jar KafkaJavaApiConsumerClientExamp
le.class

//建议使用idea编译打包fat jar
//参考示例工程 kafka-examples/kafka-java-api-example 打包出 kafka-java-api-example-1.0-SNAPSHOT-
jar-with-dependencies.jar
mvn clean package

//运行
/usr/local/jdk/bin/java -classpath kafka-java-api-example-1.0-SNAPSHOT-jar-with-dependencies.jar co
m.tencent.tbds.kafka.KafkaJavaApiConsumerClientExample
```



Scala api

生产者Scala api使用示例

```
//生产
import java.util.Properties
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}

object KafkaScalaApiClientExample {
  def main(args: Array[String]): Unit = {
    System.setProperty("java.security.auth.login.config", "/usr/local/service/kafka/config/kafka-gssapi-ja
as.conf");
    System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
    val props:Properties = new Properties()
    props.put("bootstrap.servers", "x.x.x.x:9092");//kafka集群，broker-list
    props.put("acks","all")
```

```
    props.put("acks", "all");
    props.put("retries", "1");//重试次数
    props.put("batch.size", "16384");//批次大小
    props.put("linger.ms", "1");//等待时间
    props.put("buffer.memory", "33554432");//RecordAccumulator缓冲区大小
    props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    // kerberos认证
    props.put("security.protocol", "SASL_PLAINTEXT");
    props.put("sasl.mechanism", "GSSAPI");
    props.put("sasl.kerberos.service.name", "hadoop");
    props.put("sasl.jaas.config", "com.sun.security.auth.module.Krb5LoginModule required useKeyTab=tr
ue storeKey=true keyTab=\"/var/krb5kdc/emr.keytab\" principal=\"xxxx/x.x.x.x@TBDS-XXXX\";");

    val producer = new KafkaProducer[String, String](props)
    val topic = "test_topic_001"

    try {
      for (i <- 0 to 10) {
        val record = new ProducerRecord[String, String](topic, "00"+i, "11"+i)
        val metadata = producer.send(record)
        printf(s"sent record(key=%s value=%s) " +
          "meta(partition=%d, offset=%d)\n",
          record.key(), record.value(),
          metadata.get().partition(),
          metadata.get().offset())
      }
    }catch{
      case e:Exception => e.printStackTrace()
    }finally {
      producer.close()
    }
  }
}



//编译
scalac -classpath kafka-clients-2.8.2.jar:slf4j-api-2.0.9.jar KafkaScalaApiClientExample.scala

//建议使用idea编译打包fat jar
//参考示例工程 kafka-examples/kafka-scala-api-example 打包出 kafka-scala-api-example-1.0-SNAPSHO
T-jar-with-dependencies.jar
mvn clean package

//运行
scala -cp .:kafka-scala-api-example-1.0-SNAPSHOT-jar-with-dependencies.jar com.tencent.tbds.kafka.K
```

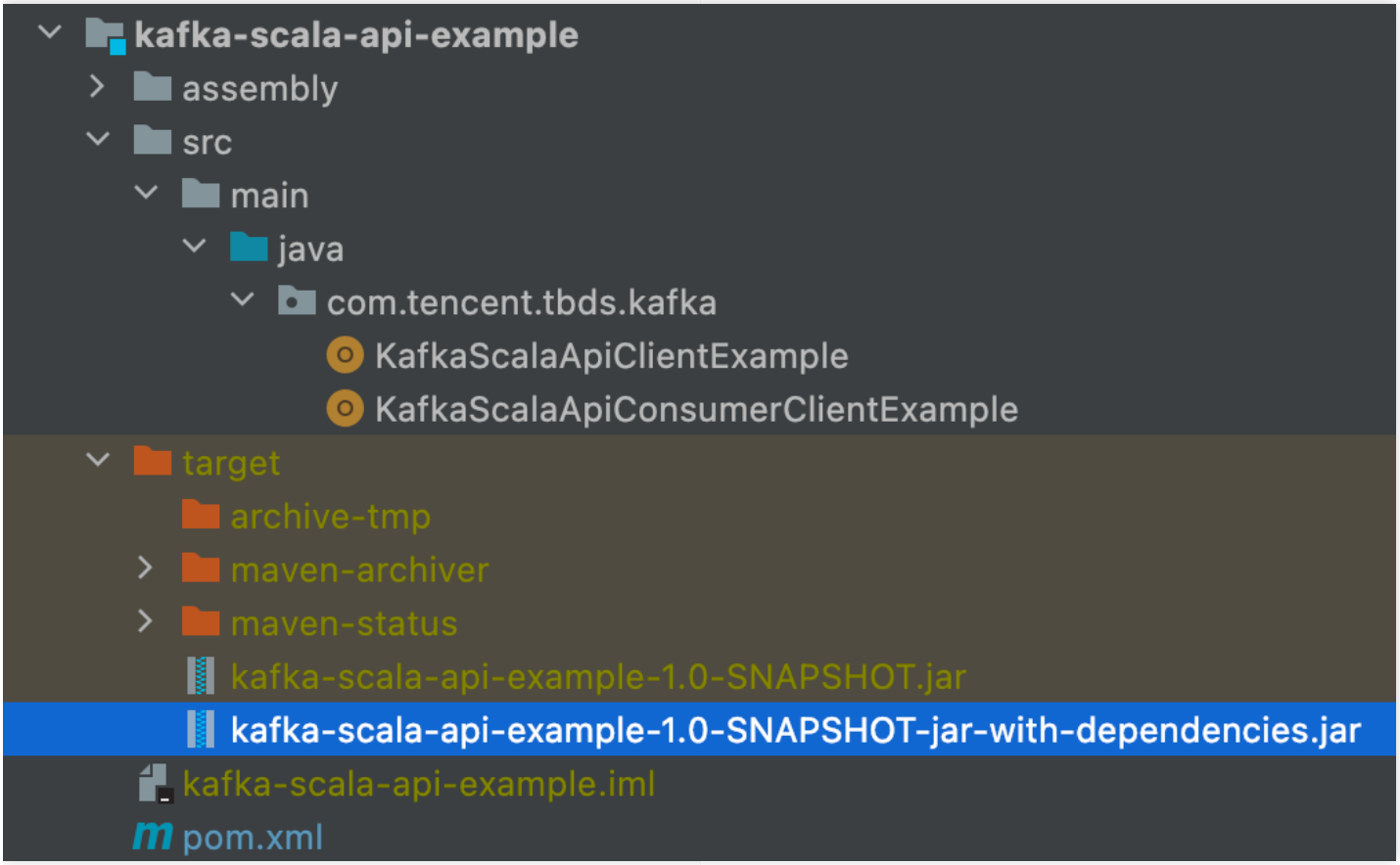
afkaScalaApiClientExample

消费者java api使用示例

```
//消费
import java.util.Properties
import scala.collection.JavaConverters._
import org.apache.kafka.clients.consumer.KafkaConsumer

object KafkaScalaApiConsumerClientExample {

  def main(args: Array[String]): Unit = {
    System.setProperty("java.security.auth.login.config", "/usr/local/service/kafka/config/kafka-gssapi-jaas.conf");
    System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
    val props:Properties = new Properties()
    props.put("bootstrap.servers", "x.x.x.x:9092");//kafka集群，broker-list
    props.put("group.id", "test-consumer-group111");//消费者组，只要group.id相同，就属于同一个消费者组
    props.put("enable.auto.commit", "true");//自动提交offset
    props.put("auto.commit.interval.ms", "1000"); // 自动提交时间间隔
    props.put("max.poll.records", "1000"); // 拉取的数据条数
    props.put("session.timeout.ms", "10000"); // 维持session的时间。超过这个时间没有心跳 就会剔出消费者组
    props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("auto.offset.reset", "earliest");
    // kerberos认证
    props.put("security.protocol", "SASL_PLAINTEXT");
    props.put("sasl.mechanism", "GSSAPI");
    props.put("sasl.kerberos.service.name", "hadoop");
    props.put("sasl.jaas.config", "com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true storeKey=true keyTab=\"/var/krb5kdc/emr.keytab\" principal=\"xxxx/x.x.x.x@TBDS-XXXX\";");

    val consumer = new KafkaConsumer(props)
    val topics = List("test_topic_001")

    try {
      consumer.subscribe(topics.asJava)   // 订阅主题
      while (true) {
        val records = consumer.poll(10)   // 轮询
        for (record <- records.asScala) {
          println("Topic: " + record.topic() +
            ",Key: " + record.key() +
            ",Value: " + record.value() +
            ", Offset: " + record.offset() +
```

```
            ", Partition: " + record.partition())
        }
      }
    }catch{
      case e:Exception => e.printStackTrace()
    }finally {
      consumer.close()
    }
  }
}
```

```
//编译
scalac -classpath kafka-clients-2.8.2.jar:slf4j-api-2.0.9.jar KafkaScalaApiConsumerClientExample.scala

//建议使用idea编译打包fat jar
//参考示例工程 kafka-examples/kafka-scala-api-example 打包出 kafka-scala-api-example-1.0-SNAPSHOT-jar-with-dependencies.jar
mvn clean package

//运行
scala -cp .:kafka-scala-api-example-1.0-SNAPSHOT-jar-with-dependencies.jar com.tencent.tbds.kafka.KafkaScalaApiConsumerClientExample
```


kafka-scala-api-example
assembly
src
main
java
com.tencent.tbds.kafka
KafkaScalaApiClientExample
KafkaScalaApiConsumerClientExample
target
archive-tmp
maven-archiver
maven-status
kafka-scala-api-example-1.0-SNAPSHOT.jar
kafka-scala-api-example-1.0-SNAPSHOT-jar-with-dependencies.jar
kafka-scala-api-example.iml
pom.xml

常用命令

在尝试kafka命令之前，有以下两项工作需要在client侧完成。

- 准备JAAS配置文件：暂定文件名为/tmp/kafka-jaas-config.properties。
- 准备sasl认证配置文件：暂定文件名为/tmp/kafka-sasl-config.properties。

准备JAAS配置文件

文件名为/tmp/kafka-jaas-config.properties，名称可随意取。不同的认证方式，文件内容不同，下面举例说明了两种常见的认证方式配置：

```
//SASL PLAIN认证方式下，文件内容如下：
KafkaClient {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="xxxx"
    password="xxxx@Tbds.com";
};

//SASL GSSAPI（Kerberos）认证方式下，文件内容如下：
//注意替换${keytab path}和"${principal name}
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="${keytab path}"
    principal="${principal name}";
};
```

准备sasl认证配置文件

文件名为/tmp/kafka-sasl-config.properties，名称可随意取。不同的认证方式，文件内容不同，下面举例说明了两种常见的认证方式配置：

```
//SASL PLAIN认证方式下，文件内容如下：
security.protocol=SASL_PLAINTEXT
sasl.mechanism=PLAIN
```

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="xxxx" password="xxxx@Tbds.com";

//SASL GSSAPI认证（即Kerberos认证）方式下，文件内容如下：
//注意替换${keytab path}和"${principal name}
security.protocol=SASL_PLAINTEXT
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=hadoop
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true storeKey=true keyTab="${keytab path}" principal="${principal name}";
```

创建topic

```
#如果kafka-sasl-config.properties中配置了sasl.jaas.config属性，则无须export KAFKA_OPTS
export KAFKA_OPTS='-Djava.security.auth.login.config=/tmp/kafka-jaas-config.properties';
/usr/local/service/kafka/bin/kafka-topics.sh --bootstrap-server 10.0.1.153:9092 --create --topic test_lucien_topic_002 --partitions 5 --replication-factor 2 --command-config /tmp/kafka-sasl-config.properties
```

查看topic

```
#如果kafka-sasl-config.properties中配置了sasl.jaas.config属性，则无须export KAFKA_OPTS
export KAFKA_OPTS='-Djava.security.auth.login.config=/tmp/kafka-jaas-config.properties';
/usr/local/service/kafka/bin/kafka-topics.sh --bootstrap-server 10.0.1.153:9092 --describe --topic test_lucien_topic_001 --command-config /tmp/kafka-sasl-config.properties
```

更新topic

```
#如果kafka-sasl-config.properties中配置了sasl.jaas.config属性，则无须export KAFKA_OPTS
export KAFKA_OPTS='-Djava.security.auth.login.config=/tmp/kafka-jaas-config.properties';
#修改分区数
/usr/local/service/kafka/bin/kafka-topics.sh --bootstrap-server  ${brokerIP}:9092 --alter --topic test_topic_001 --partitions 30 --command-config /tmp/kafka-sasl-config.properties
```

#修改配置
/usr/local/service/kafka/bin/kafka-configs.sh --bootstrap-server ${brokerIP}:9092 --entity-type topics --entity-name test_topic_001 --alter --add-config retention.ms=604800000 --command-config /tmp/kafka-sasl-config.properties

#删除配置
/usr/local/service/kafka/bin/kafka-configs.sh --bootstrap-server ${brokerIP}:9092 --entity-type topics --entity-name test_topic_001 --alter --delete-config retention.ms --command-config /tmp/kafka-sasl-config.properties

删除topic

#如果kafka-sasl-config.properties中配置了sasl.jaas.config属性，则无须export KAFKA_OPTS
export KAFKA_OPTS='-Djava.security.auth.login.config=/tmp/kafka-jaas-config.properties';
/usr/local/service/kafka/bin/kafka-topics.sh --bootstrap-server 10.0.1.153:9092 --delete --topic test_lucien_topic_001 --command-config /tmp/kafka-sasl-config.properties

生产数据

#如果kafka-sasl-config.properties中配置了sasl.jaas.config属性，则无须export KAFKA_OPTS
export KAFKA_OPTS='-Djava.security.auth.login.config=/tmp/kafka-jaas-config.properties';
/usr/local/service/kafka/bin/kafka-console-producer.sh --topic test_lucien_topic_001 --bootstrap-server 10.0.1.153:9092 --producer.config /tmp/kafka-sasl-config.properties

{"id":6,"name":"fengfeng6"}

消费数据

```
#如果kafka-sasl-config.properties中配置了sasl.jaas.config属性，则无须export KAFKA_OPTS
export KAFKA_OPTS='-Djava.security.auth.login.config=/tmp/kafka-jaas-config.properties';
/usr/local/service/kafka/bin/kafka-console-consumer.sh --topic test_lucien_topic_001 --from-beginnin
g --bootstrap-server 10.0.1.153:9092 --consumer.config /tmp/kafka-sasl-config.properties
```

常见问题

客户在使用kafka时，可能会出现以下问题。

Kafka客户端认证信息如何正确配置

目前TBDS Kafka集群有两种认证方式，一种是PLAIN认证，一种是Kerberos认证。
PLAIN认证对应的认证配置文件内容如下：

```
security.protocol=SASL_PLAINTEXT
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="{usern
ame}" password="{password}";
```

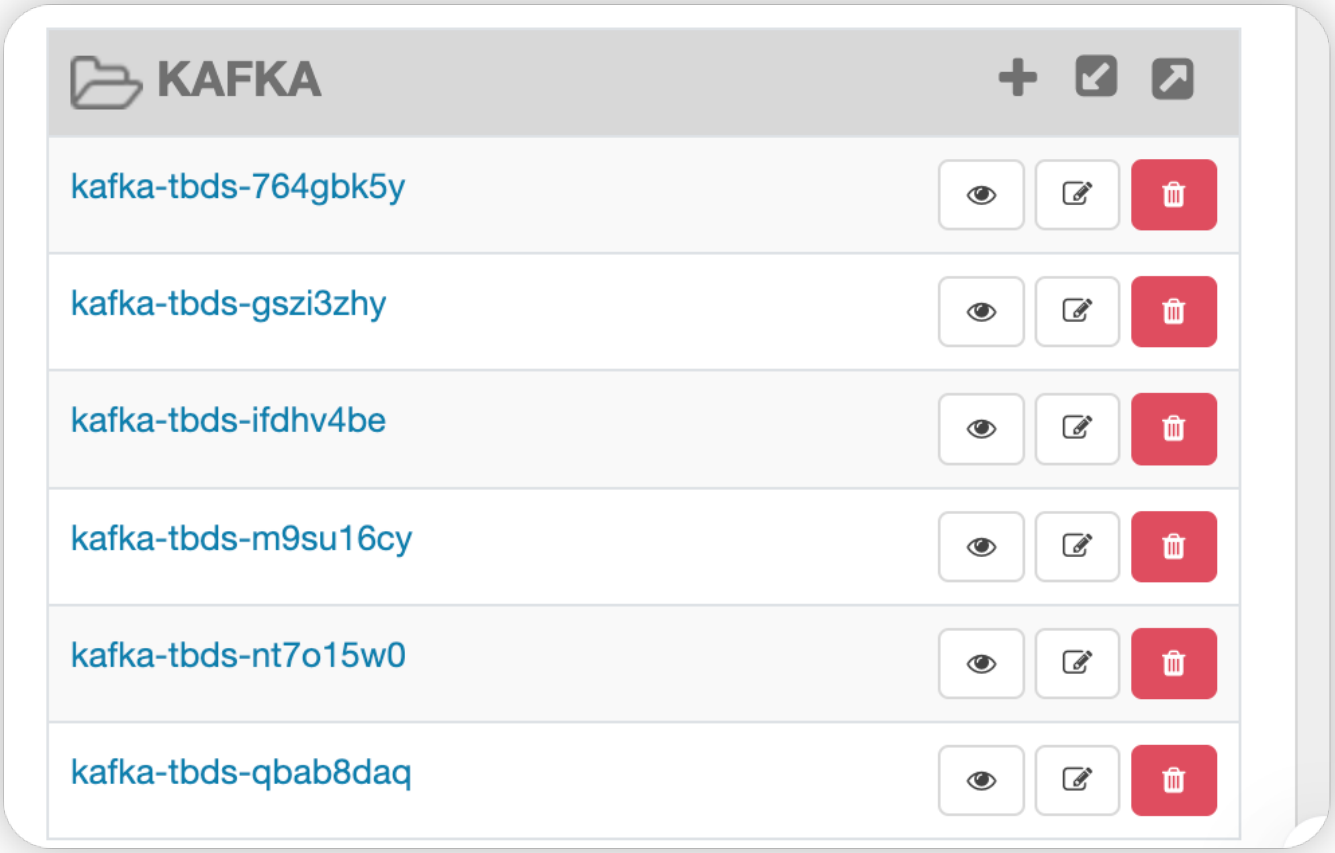
其中username和password是Kafka服务端/usr/local/service/kafka/config/kafka-plain-jaas.conf文件配置的用户名和密码。
Kerberos认证对应的认证配置文件内容如下：

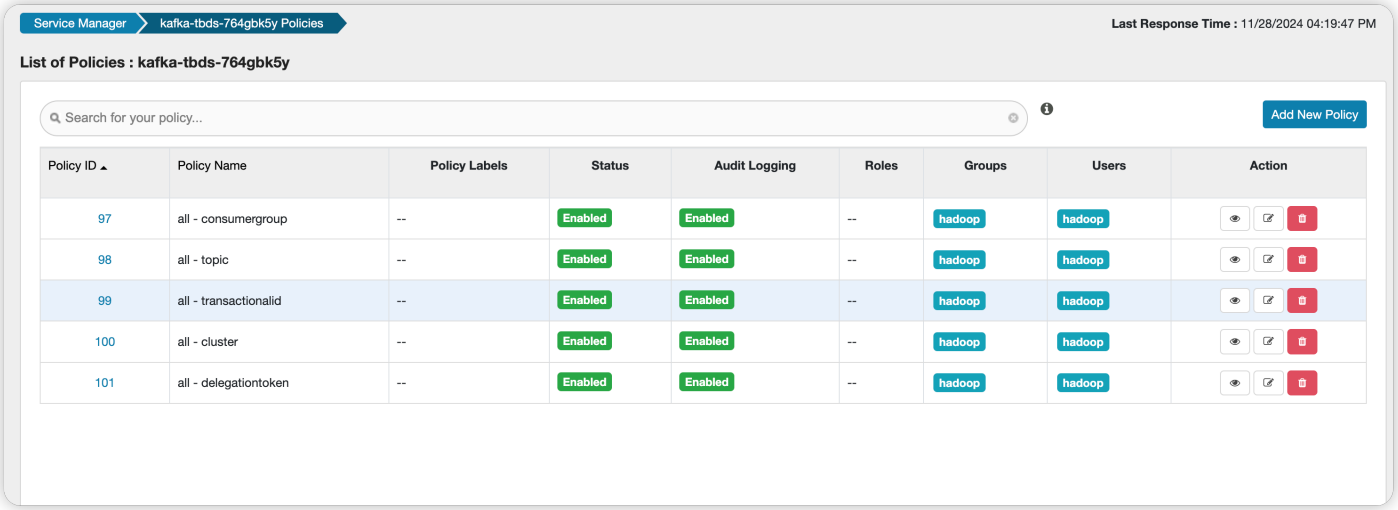
```
security.protocol=SASL_PLAINTEXT
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=hadoop
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true storeKey=tr
ue keyTab="${keytab path}" principal="${principal name}";
```

keytab path和principal name是kafka客户端使用的keytab和principal，可自行设定。

如何修改Kafka对应的Ranger策略

TBDS集群的kafka已经默认集成Ranger插件，基于Ranger鉴权插件，可进行细粒度的Kafka ACL管理。
登录RangerUI可以管理Kafka组件的ACL，新增/修改/删除相关的Kafka topic权限信息（具体使用方式请参考Ranger授权）：


KAFKA
kafka-tbds-764gbk5y
kafka-tbds-gszi3zhy
kafka-tbds-ifdhv4be
kafka-tbds-m9su16cy
kafka-tbds-nt7o15w0
kafka-tbds-qbab8daq


Service Manager
kafka-tbds-764gbk5y Policies
Last Response Time : 11/28/2024 04:19:47 PM
List of Policies : kafka-tbds-764gbk5y
Search for your policy...
Add New Policy
Policy ID
Policy Name
Policy Labels
Status
Audit Logging
Roles
Groups
Users
Action
97 all - consumergroup -- Enabled Enabled -- hadoop hadoop
98 all - topic -- Enabled Enabled -- hadoop hadoop
99 all - transactionalid -- Enabled Enabled -- hadoop hadoop
100 all - cluster -- Enabled Enabled -- hadoop hadoop
101 all - delegationtoken -- Enabled Enabled -- hadoop hadoop

权限策略配置

配置步骤

1. 安装 Ranger Plugin：

   i. 下载并安装 Ranger Kafka插件。

2. 配置Kafka：

   i. 修改server.properties 文件，添加以下配置：

   authorizer.class.name=org.apache.ranger.authorization.kafka.authorizer.RangerKafkaAuthorizer

3. 配置 Ranger：

   i. 使用Ranger管理员用户rangeradmin登录Ranger管理页面。
   ii. 在首页中单击“KAFKA”区域的组件插件名称，例如“kafka”。
   iii. 单击“Add New Policy”，添加Kafka权限控制策略。
   iv. 根据业务需求配置相关参数。

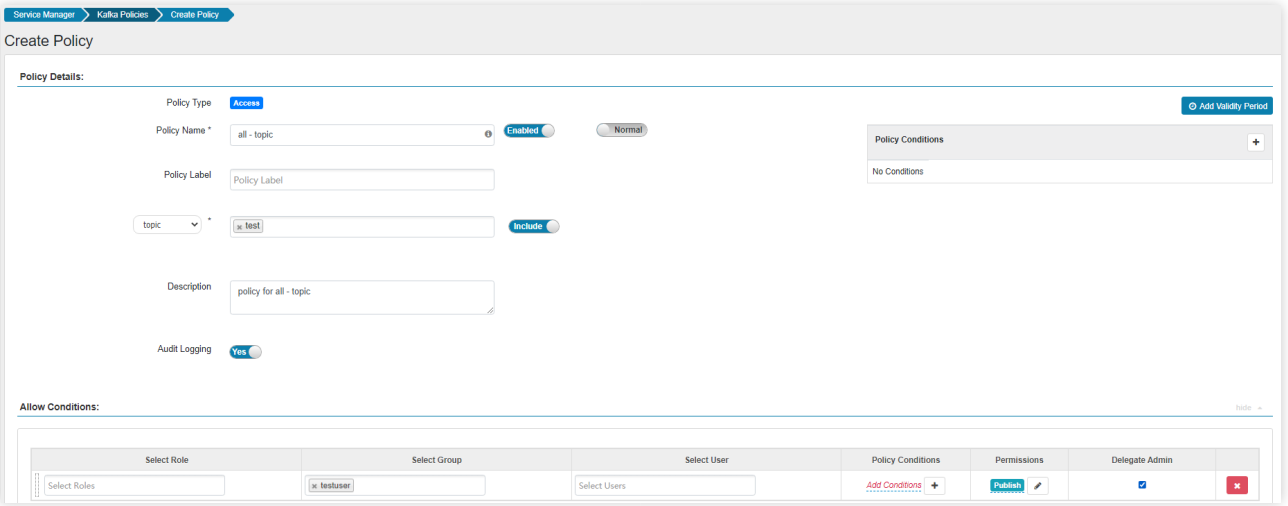
   Kafka权限参数如下：

参数名称	描述
Policy Type	Access。
Policy Conditions	IP过滤策略，可自定义，配置当前策略适用的主机节点，可填写一个或多个IP或IP段，并且IP填写支持“*”通配符，例如：172.138.1.10,172.138.1.20或者172.138.1.*。
Policy Name	策略名称，可自定义，不能与本服务内其他策略名称重复。
Policy Label	为当前策略指定一个标签，您可以根据这些标签搜索报告和筛选策略。
topic	配置当前策略适用的topic名，可以填写多个值。这里支持通配符，例如：test、test*、*。“Include”策略适用于当前输入的对象，“Exclude”表示策略适用于除去当前输入内容之外的其他对象。
Description	策略描述信息。
Audit Logging	是否审计此策略。
Allow Conditions	策略允许条件，配置本策略内允许的权限及例外，例外条件优先级高于正常条件。 在“Select Role”、“Select Group”、“Select User”列选择已创建好的需要授予权限的Role、用户组或用户。 单击“Add Conditions”，添加策略适用的IP地址范围，单击“Add Permissions”，添加对应权限。 ● Publish：生产权限。 ● Consume：消费权限。

	● Describe：查询权限。 ● Create： 创建主题权限。 ● Delete： 删除主题权限。 ● Describe Configs：查询配置权限。 ● Alter：修改topic的partition数量的权限。 ● Alter Configs：修改配置权限。 ● Select/Deselect All：全选/取消全选。 如需添加多条权限控制规则，可单击 按钮添加。 如需当前条件中的用户或用户组管理本条策略，可勾选“Delegate Admin”，这些用户将成为受委托的管理员。被委托的管理员可以更新、删除本策略，它还可以基于原始策略创建子策略。
Deny Conditions	策略拒绝条件，配置本策略内拒绝的权限及例外，配置方法与“Allow Conditions”类型，拒绝条件的优先级高于“Allow Conditions”中配置的允许条件。

例如为用户“testuser”添加“test”主题的生产权限，配置如下：

Kafka权限参数：



设置权限场景如下：

任务场景	角色授权操作
设置Kafka管理员权限	1. 在首页中单击“KAFKA”区域的组件插件名称，例如“Kafka”。 2. 选择“Policy Name”为“all - topic”的策略，单击 按钮编辑策略。 3. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 4. 单击“Add Permissions”，勾选“Select/Deselect All”。

设置用户对Topic的创建权限	1. 在“topic”配置Topic名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Create”。 说明： 目前Kafka内核支持"--zookeeper"和"--bootstrap-server"两种方式创建Topic，社区将会在后续的版本中删掉对"--zookeeper"的支持，所以建议用户使用"--bootstrap-server"的方式创建Topic。 注意：目前Kafka只支持"--bootstrap-server"方式创建Topic行为的鉴权，不支持对"--zookeeper"方式的鉴权。
设置用户对Topic的删除权限	1. 在“topic”配置Topic名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Delete”。 说明： 目前Kafka内核支持"--zookeeper"和"--bootstrap-server"两种方式创建Topic，社区将会在后续的版本中删掉对"--zookeeper"的支持，所以建议用户使用"--bootstrap-server"的方式创建Topic。 注意：目前Kafka只支持"--bootstrap-server"方式创建Topic行为的鉴权，不支持对"--zookeeper"方式的鉴权。
设置用户对Topic的查询权限	1. 在“topic”配置Topic名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Describe”和“Describe Configs”。 说明： 目前Kafka内核支持"--zookeeper"和"--bootstrap-server"两种方式创建Topic，社区将会在后续的版本中删掉对"--zookeeper"的支持，所以建议用户使用"--bootstrap-server"的方式创建Topic。 注意：目前Kafka只支持"--bootstrap-server"方式创建Topic行为的鉴权，不支持对"--zookeeper"方式的鉴权。
设置用户对Topic的生产权限	1. 在“topic”配置Topic名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Publish”。
设置用户对Topic的消费权限	1. 在“topic”配置Topic名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Consume”。 说明： 因为消费Topic时，涉及到Offset的管理操作，必须同时开启ConsumerGroup的“Consume”权限，详见“设置用户对ConsumerGroup Offsets的提交权限”。
设置用户对Topic的扩容权限（增加分区）	1. 在“topic”配置Topic名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。

	3. 单击“Add Permissions”，勾选“Alter”。
设置用户对Topic的配置修改权限	当前Kafka内核暂不支持基于“--bootstrap-server”的Topic参数修改行为，故当前Ranger不支持对此行为的鉴权操作。
设置用户对Cluster的所有管理权限	1. 在“cluster”右侧输入并选择集群名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Kafka Admin”。
设置用户对Cluster的创建权限	1. 在首页中单击“KAFKA”区域的组件插件名称，例如“Kafka”。 2. 选择“Policy Name”为“all - cluster”的策略，单击 按钮编辑策略。 3. 在“cluster”右侧输入并选择集群名。 4. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 5. 单击“Add Permissions”，勾选“Create”。 说明： 对于Cluster的Create操作鉴权主要涉及以下两个场景： 1.集群开启了“auto.create.topics.enable”参数后，客户端向服务的还未创建的Topic发送数据的场景，此时会判断用户是否有集群的Create权限。 2.对于用户创建大量Topic的场景，如果授予用户Cluster Create权限，那么该用户可以在集群内部创建任意Topic。
设置用户对Cluster的配置修改权限	1. 在“cluster”右侧输入并选择集群名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Alter Configs”。 说明： 此处的配置修改权限，指的是Broker、Broker Logger的配置权限。 当授予用户配置修改权限后，即使不授予配置查询权限也可查询配置详情（配置修改权限高于且包含配置查询权限）。
设置用户对Cluster的配置查询权限	1. 在“cluster”右侧输入并选择集群名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Describe”和 “Describe Configs”。 说明： 此处查询指的是查询集群内的Broker、Broker Logger信息。该查询不涉及Topic。
设置用户对Cluster的Idempotent Write权限	1. 在“cluster”右侧输入并选择集群名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Idempotent Write”。 说明： 此权限会对用户客户端的Idempotent Produce行为进行鉴权。

设置用户对Cluster的分区迁移权限管理	1. 在“cluster”右侧输入并选择集群名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Alter”。 说明： Cluster的Alter权限可以对以下三种场景进行权限控制： 1. Partition Reassign场景下，迁移副本的存储目录。 2. 集群里各分区内部leader选举。 3. Acl管理（添加或删除）。 其中1和2都是集群内部Controller与Broker间、Broker与Broker间的操作，创建集群时，默认授予内置kafka用户此权限，普通用户授予此权限没有意义。 3涉及Acl的管理，Acl设计的就是用于鉴权，由于目前kafka鉴权已全部托管给Ranger，所以这个场景也基本不涉及（配置后亦不生效）。
设置用户对Cluster的Cluster Action权限	1. 在“cluster”右侧输入并选择集群名。 2. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 3. 单击“Add Permissions”，勾选“Cluster Action”。 说明： 此权限主要对集群内部副本主从同步、节点间通信进行控制，在集群创建时已经授权给内置kafka用户，普通用户授予此权限没有意义。
设置用户对TransactionalId的权限	1. 在首页中单击“KAFKA”区域的组件插件名称，例如“Kafka”。 2. 选择“Policy Name”为“all - TransactionalId”的策略，单击 按钮编辑策略。 3. 在“transactionalid”配置事务ID。 4. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 5. 单击“Add Permissions”，勾选“Publish”和 "Describe"。 说明： “Publish”权限主要对用户开启了事务特性的客户端请求进行鉴权，例如事务开启、结束、提交offset、事务性数据生产等行为。 “Describe”权限主要对于开启事务特性的客户端与Coordinator的请求进行鉴权。 建议在开启事务特性的场景下，给用户同时授予“Publish”和“Describe”权限。
设置用户对DelegationToken的权限	1. 在首页中单击“KAFKA”区域的组件插件名称，例如“Kafka”。 2. 选择“Policy Name”为“all - delegationtoken”的策略，单击 按钮编辑策略。 3. 在“DelegationToken”配置delegationtoken。 4. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 5. 单击“Add Permissions”，勾选“ Describe”。 说明：

	当前Ranger对DelegationToken的鉴权控制仅限于对查询的权限控制，不支持对DelegationToken的create、renew、expire操作的权限控制。
设置用户对ConsumerGroup Offsets的查询权限	1. 在首页中单击“KAFKA”区域的组件插件名称，例如“Kafka”。 2. 选择“Policy Name”为“all - consumergroup”的策略，单击 按钮编辑策略。 3. 在“consumergroup”配置需要管理的consumergroup。 4. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 5. 单击“Add Permissions”，勾选“Describe”。
设置用户对ConsumerGroup Offsets的提交权限	1. 在首页中单击“KAFKA”区域的组件插件名称，例如“Kafka”。 2. 选择“Policy Name”为“all - consumergroup”的策略，单击 按钮编辑策略。 3. 在“consumergroup”配置需要管理的consumergroup。 4. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 5. 单击“Add Permissions”，勾选“Consume”。 说明： 当给用户授予了ConsumerGroup的“Consume”权限后，用户会同时被授予“Describe”权限。
设置用户对ConsumerGroup Offsets的删除权限	1. 在首页中单击“KAFKA”区域的组件插件名称，例如“Kafka”。 2. 选择“Policy Name”为“all - consumergroup”的策略，单击 按钮编辑策略。 3. 在“consumergroup”配置需要管理的consumergroup。 4. 在“Allow Conditions”区域，单击“Select User”下选择框选择用户。 5. 单击“Add Permissions”，勾选“Delete”。 说明： 当给用户授予了ConsumerGroup的“Delete”权限后，用户会同时被授予“Describe”权限。

5. （可选）添加策略有效期。在页面右上角单击“Add Validity period”，设置“Start Time”和“End Time”，选择“Time Zone”。单击“Save”保存。如需添加多条策略有效期，可单击 按钮添加。如需删除策略有效期，可单击 按钮删除。
6. 单击“Add”，在策略列表可查看策略的基本信息。等待策略生效后，验证相关权限是否正常。如需禁用某条策略，可单击 按钮编辑策略，设置策略开关为“Disabled”。

   如果不再使用策略，可单击 按钮删除策略。
7. 重启 Kafka：

i. 重启 Kafka 服务以应用配置。

Kafka机架配置

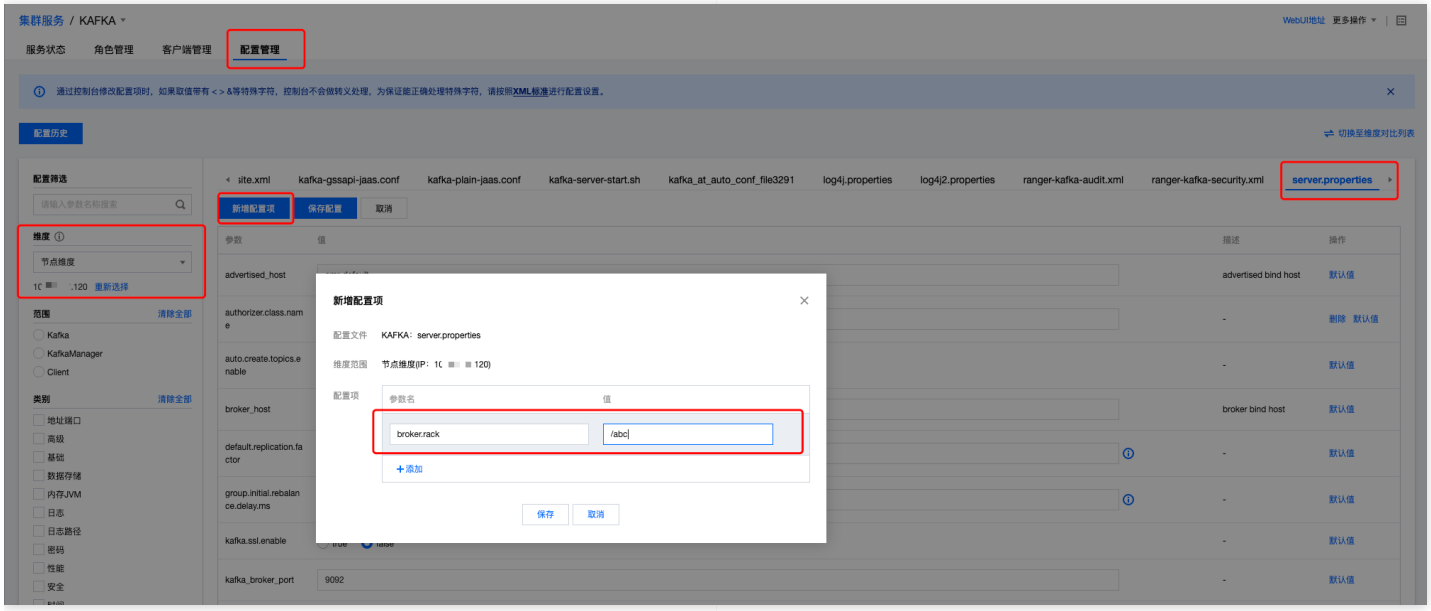
机架使用场景

Kafka Broker提供机架配置的能力，在副本分配阶段根据机架将副本分配到不同机架的Kafka Broker中，以提升副本的容错能力。
常见使用场景：

- 集群内首次配置机架感知能力，提升副本容错性。
- 集群已开启机架后，扩容新的kafka broker节点，则需要为新扩容的节点配置机架信息。

机架配置方法

Kafka机架配置可直接在TM上进行操作，需要按节点维度将所有的broker配置broker.rack属性，操作路径为“集群服务--->Kafka--->配置管理--->维度--->选择节点维度--->选择节点信息--->选择server.properties--->新增配置项--->新增broker.rack属性--->保存--->下发”，如下图所示。



所有节点都配置并下发完成后，将kafka集群重启即可。
注意，需要对所有的节点配置不同的机架信息，如果有节点配置了机架，而有些节点未配置机架，则在副本分配阶段会报错Not all brokers have rack information for replica rack aware assignment，如下图所示：

```
tbds-10-0-1-19@TBDS-D5ZMZHAP; authorizationID=hadoop/tbds-10-0-1-19@TBDS-D5ZMZHAP.
[2025-05-20 10:41:17,962] [ERROR] [data-plane-kafka-request-handler-5:43369867] [kafka.server.ZkAdminManager] [Logging.scala:76] - [Admin Manager on B
roker 1003]: Error processing create topic request CreatableTopic(name='test-consumer-topic1747708178', numPartitions=1, replicationFactor=2, assignme
nts=[], configs=[])
kafka.admin.AdminOperationException: Not all brokers have rack information for replica rack aware assignment.
        at kafka.admin.AdminUtils$.assignReplicasToBrokers(AdminUtils.scala:119) ~[kafka_2.13-2.8.2.jar:?]
        at kafka.server.ZkAdminManager.$anonfun$createTopics$2(ZkAdminManager.scala:177) ~[kafka_2.13-2.8.2.jar:?]
        at scala.collection.Iterator$$anon$9.next(Iterator.scala:575) ~[scala-library-2.13.5.jar:?]
        at scala.collection.immutable.List.prependedAll(List.scala:153) ~[scala-library-2.13.5.jar:?]
        at scala.collection.immutable.List$.from(List.scala:651) ~[scala-library-2.13.5.jar:?]
        at scala.collection.immutable.List$.from(List.scala:648) ~[scala-library-2.13.5.jar:?]
```

验证机架配置是否生效

机架的是否生效体现在以下两点：

1. 所有的Kafka Broker节点的server.properties配置了broker.rack属性，执行以下命令。

cat /usr/local/service/kafka/config/server.properties | grep "broker.rack"

结果例如下图所示：

```
#
#
# cat /usr/local/service/kafka/config/server.properties | grep "broker.rack"
broker.rack=/rack1
#
```

2. 创建一个新的topic（创建topic和查看topic的方法参考常用命令）。如果副本数不大于机架数，该topic的每个分区的不同副本必分配在不同的机架对应的kafka broker上。如果副本数大于机架数，则每个机架上必定存在该topic的同一个分区的一个或多个副本。
举例来说，1001的broker.rack是/rack2，1002和1003的broker.rack是/rack1，创建一个五分区两副本的topic，其副本分布如下图所示：

```
[2025-06-20 10:58:13,791] [INFO ] [main:898] [org.apache.kafka.common.utils.AppInfoParser] [AppInfoParser.java:120] - Kafka com
[2025-06-20 10:58:13,791] [INFO ] [main:898] [org.apache.kafka.common.utils.AppInfoParser] [AppInfoParser.java:121] - Kafka sta
Topic: test-topic-002   TopicId: CrinNyOKTuWFLVgQWyumpQ PartitionCount: 5       ReplicationFactor: 2    Configs: min.insync.rep
        Topic: test-topic-002   Partition: 0    Leader: 1001    Replicas: 1001,1003     Isr: 1001,1003
        Topic: test-topic-002   Partition: 1    Leader: 1003    Replicas: 1003,1001     Isr: 1003,1001
        Topic: test-topic-002   Partition: 2    Leader: 1002    Replicas: 1002,1001     Isr: 1002,1001
        Topic: test-topic-002   Partition: 3    Leader: 1001    Replicas: 1001,1003     Isr: 1001,1003
        Topic: test-topic-002   Partition: 4    Leader: 1003    Replicas: 1003,1001     Isr: 1003,1001
[2025-06-20 10:58:14,106] [INFO ] [kafka-admin-client-thread | adminclient-1:1213] [org.apache.kafka.common.utils.AppInfoParser
```

Kyuubi开发
概述

Apache Kyuubi 是一个 Thrift JDBC/ODBC 服务，对接了 Apache Spark 计算框架以及Trino，支持多租户和分布式等特性，可以满足企业内诸如 ETL、BI 报表等多种大数据场景的应用。
Kyuubi架构大体可分为4个层次：客户端、服务发现、KyuubiServer、Engine。

- 客户端：包括JDBC客户端、Http客户端、beeline客户端以及Thrift客户端。
- 服务发现层：目前最新版本中Kyuubi采用了ZK和etcd作为服务发现层，服务发现层除了作为KyuubiServer到客户端，Engine到KyuubiServer的服务发现外，也承载着Kyuubi内元数据信息的存储，如：会话信息等。
- KyuubiServer: 负责接收客户端的连接请求、将客户端的任务生成Engine或者转发到对应的Engine中去执行，以及连接会话和Engine的生命周期的管理等功能。
- Engine: 客户端任务的执行器，目前主要有Spark SQL、Trino、JDBC、HiveJDBC。

示例工程开发

Kyuubi目前支持beeline,jdbc,Rest以及Thrift四种接入方式。在1.6.0版本中Rest方式只支持spark-jar包任务的提交、日志获取、状态查询和任务中断功能。针对于beeline和jdbc的接入方式,在Kyuubi不同隔离级别下：任务的中断功能当前还无法支持。比如在K8s环境，非CONNECTION隔离级别下，无法直接通过Kyuubi关闭任务，需要通过调用K8s的相关接口直接关闭容器的方式关闭任务，这种方式对于共享Engine下的使用方式也不友好。在升级到1.7.1+的版本后，Kyuubi提供了通过Rest接口可以实现sql级别的任务控制。
主流的Kyuubi使用方式有三种：Beeline,JDBC,Rest批任务。
Beeline

1. 客户端需通过kinit进行kerberos认证

```
>klist -kt /var/krb5kdc/xxx.keytab       -- 用户kerberos认证keytab
>kinit -kt /var/krb5kdc/xxx.keytab  xxx@xxx     -- 用户的principal
>klist
```

1. 通过Kyuubi提供的beeline与kyuubi建立连接，支持通过zk方式和指定KyuubiServer的方式，下面是通过指定kyuubiserver的方式

```
并通过kyuubi.engine.share.level=CONNECTION指定了隔离级别为CONNECTION
>cd /usr/local/service/kyuubi
>bin/beeline -u "jdbc:hive2://172.16.12.14:10009/default;principal=hadoop/tbds-4ewgkc73@TBDS-MD3Y2ZV9(需要使用KyuubiServer的
kerberos);#kyuubi.engine.type=SPARK_SQL;kyuubi.engine.share.level=CONNECTION;"

3.使用zookeeper 方式：
/usr/local/service/kyuubi/bin/beeline -u "jdbc:hive2://{zookeeper-ip}:2181/
default;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=kyuubi;#kyuubi.engine.type=SPARK_SQL;kyuubi.engine.share.level=CONNECTION;"
```

JDBC
JDBC的方式，目前Kyuubi支持使用Hive-jdbc SDK，也可用通过Kyuubi提供的jdbc SDK与Kyuubi建立连接。

1. pom引入依赖

```
<dependency>
  <groupId>org.apache.kyuubi</groupId>
   <art
   ifactId>kyuubi-hive-jdbc-shaded</artifactId>
   <version>1.8.0-incubating</version>
</dependency>
或者
<dependency>
   <groupId>org.apache.hive</groupId>
  <artifactId>hive-jdbc</artifactId>
   <version>3.1.3</version>
</dependency>
```

2. 参考demo编写程序通过JDBC与Kyuubi建立连接。

```
package com.tencent.kyuubi.demo.jdbc.hive;

import java.sql.*;
import java.util.List;
import java.util.concurrent.TimeUnit;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.security.UserGroupInformation;
import org.apache.kyuubi.jdbc.KyuubiHiveDriver;
import org.apache.kyuubi.jdbc.hive.KyuubiQueryResultSet;
import org.apache.kyuubi.jdbc.hive.KyuubiStatement;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class KyuubiHiveJDBCTest {
   public static Logger LOG = LoggerFactory.getLogger(KyuubiHiveJDBCTest.class);
   public static String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";
```

```
    public static String KYUUBI_DRIVER = KyuubiHiveDriver.class.getName();

    public static KyuubiStatement stmt;
    public static void main(String[] args)
            throws SQLException, InterruptedException {
        String url = args[0];
        String userPrincipal = args[1];
        String userKeytab = args[2];
        String krb5File = args[3];
        String user = args[4];
        LOG.warn("## parameter url:{}",url);
        LOG.warn("## userPrincipal :{}",userPrincipal);
        LOG.warn("## userKeytab :{}",userKeytab);
        LOG.warn("## krb5.conf :{}",krb5File);
       loadingHiveKrbConfig(userPrincipal,userKeytab,krb5File);
        try {
            Class.forName(KYUUBI_DRIVER);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }
        try {
            Connection con = DriverManager.getConnection(url, user, "");
            stmt = (KyuubiStatement) con.createStatement();
            stmt.execute("select 1;");

            try {
                LOG.info("## statement handle {}","null");
                KyuubiQueryResultSet resultSet = (KyuubiQueryResultSet)stmt.executeQuery("select 1");
                while (resultSet.next()) {
                    String id = resultSet.getString("1");
                    LOG.warn("## job id:{}",id);
                    LOG.warn("## "+resultSet.toString());
                }
            }catch (Exception e) {
                e.printStackTrace();
            }

            String tableName = "KyuubiTestByJava";
            try {
                stmt.execute("drop table if exists " + tableName);

} catch (Exception e) {
                e.printStackTrace();
            }

            List<String> execLog = stmt.getExecLog();
            for(String log:execLog) {
                LOG.warn("## drop table sql exec log: {}",log);
            }

            try {
                stmt.execute("create table " + tableName +
                        " (key int, value string)");
                LOG.warn("## Create table success!");
            } catch (Exception e) {
                e.printStackTrace();
            }
            execLog = stmt.getExecLog();
            for(String log:execLog) {
                LOG.warn("## create table sql exec log: {}",log);
            }

            // show tables
            String sql = "show tables '" + tableName + "'";
            LOG.warn("## Running: {}" , sql);
            ResultSet res = stmt.executeQuery(sql);
            if (res.next()) {
```

```
            LOG.warn(res.getString(1));
        }
        execLog = stmt.getExecLog();
        for(String log:execLog) {
            LOG.warn("##show table sql exec log: {}",log);
        }

        // describe table
        sql = "describe " + tableName;
        LOG.warn("## Running: {}" , sql);
        res = stmt.executeQuery(sql);
        while (res.next()) {
            LOG.warn(res.getString(1) + "\t" + res.getString(2));
        }
        execLog = stmt.getExecLog();
        for(String log:execLog) {
            LOG.warn("##desc table sql exec log: {}",log);
        }

        sql = "insert into " + tableName + " values (42,\"hello\"),(48,\"world\")";
        stmt.execute(sql);

        execLog = stmt.getExecLog();
        for(String log:execLog) {
            LOG.warn("## insert table sql exec log: {}",log);
        }

        sql = "select * from " + tableName;
        LOG.warn("## Running: {}" , sql);
        res = stmt.executeQuery(sql);
        while (res.next()) {

LOG.warn(String.valueOf(res.getInt(1)) + "\t"
                    + res.getString(2));
        }
        execLog = stmt.getExecLog();
        for(String log:execLog) {
            LOG.warn("##select table sql exec log: {}",log);
        }

        sql = "select count(1) from " + tableName;
        LOG.warn("## Running: {}" , sql);
        res = stmt.executeQuery(sql);
        while (res.next()) {
            LOG.warn(res.getString(1));
        }
        execLog = stmt.getExecLog();
        for(String log:execLog) {
            LOG.warn("##select count sql exec log: {}",log);
        }

    } catch (Exception e) {
        e.printStackTrace();
    }

    Thread.sleep(10*60*1000);

}
/**
 * krb 配置
 */
public static void loadingHiveKrbConfig(String userPrincipal,String userKeyTab,String krb5File) {
    try {
        LOG.warn(" krb login");
       System.setProperty("java.security.krb5.conf", krb5File/*"/etc/krb5.conf"*/);
        Configuration configuration = new Configuration();
       configuration.setBoolean("hadoop.security.authorization", true);
       configuration.set("hadoop.security.authentication", "Kerberos");
```

```
            UserGroupInformation.setConfiguration(configuration);
            if (UserGroupInformation.isLoginKeytabBased()) {
                LOG.warn("UserGroupInformation.isLoginKeytabBased() is ture");
                UserGroupInformation.getLoginUser().reloginFromKeytab();
            } else {
                LOG.warn("UserGroupInformation.isLoginKeytabBased() is false");
                UserGroupInformation.loginUserFromKeytab(userPrincipal, userKeyTab);
            }
            LOG.warn("ticketCache=====>" + UserGroupInformation.isLoginTicketBased());
        } catch (Exception e) {
            LOG.error("krb 配置 失败：", e);
            e.printStackTrace();
        }
    }
}
```

REST批任务

1. 引入Kyuubi Rest SDK。

```
<dependency>
  <groupId>org.apache.kyuubi</groupId>
  <artifactId>kyuubi-rest-client</artifactId>
  <version>1.9.0</version>
</dependency>
```

2. 参考demo编写逻辑。

```
package com.tencent.kyuubi.demo.http;

import com.typesafe.config.Config;
import com.typesafe.config.ConfigFactory;
import com.typesafe.config.ConfigValue;
import org.apache.hadoop.security.UserGroupInformation;
import org.apache.kyuubi.client.BatchRestApi;
import org.apache.kyuubi.client.KyuubiRestClient;
import org.apache.kyuubi.client.api.v1.dto.Batch;
import org.apache.kyuubi.client.api.v1.dto.BatchRequest;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class KyuubiBatchJobSubmit {

    public static Logger LOG = LoggerFactory.getLogger(KyuubiBatchJobSubmit.class);
    static Config appConf = ConfigFactory.load();

    public static void main(String[] args) throws Exception {
        String kyuubiServer = appConf.getString("kyuubi.server");
        String user = appConf.getString("kyuubi.user");
        String passwd = appConf.getString("kyuubi.passwd");

        // 如果需要kerberos认证则执行下面的逻辑, you need kerberos authentication firstly
        //  option 1: run kinit to generate TGT cache before running this program
        //  option 2: set client.principal and client.keytab in application.conf
        if (appConf.hasPath("client.principal") && appConf.hasPath("client.keytab")) {
            String principal = appConf.getString("client.principal");
            String keytab = appConf.getString("client.keytab");
            LOG.info("login using principal: {} and keytab: {}", principal, keytab);
```

```
            UserGroupInformation.loginUserFromKeytab(principal, keytab);
        } else {
            LOG.info("login using TGT cache");
        }


        String kyuubiPrincipal = appConf.getString("kyuubi.principal");
        String[] parts = splitPrincipal(kyuubiPrincipal);
        if (parts.length != 3) {
            throw new IllegalArgumentException("Invalid kyuubi.principal: " + kyuubiPrincipal +
                    ", should be 3 parts like HTTP/hostname@REALM");
        }


        /***** spnegoClient ********/
        KyuubiRestClient spnegoClient = KyuubiRestClient.builder(kyuubiServer)

                .authHeaderMethod(KyuubiRestClient.AuthHeaderMethod.BASIC)

                .username(user)

                .password(passwd)
                .spnegoHost("tbds-172-16-16-4")
                .maxAttempts(2)
                .build();
        String appStatus = submitBatchJob(basicClient);
        LOG.info("Final Application Status: {}", appStatus);

    }
    public static String[] splitPrincipal(String principal) {
        return principal.split("[/@]");
    }

    /**
     * 提交任务并轮询获取任务状态
     * @param client
     * @return
     * @throws Exception
     */
    public static String submitBatchJob(KyuubiRestClient client) throws Exception {
        BatchRestApi api = new BatchRestApi(client);
        BatchRequest req = buildBatchReq();
        Batch batch = api.createBatch(req);
        String batchId = batch.getId();
        String appId = batch.getAppId();
        String appStatus = batch.getAppState();
        LOG.info("batchId: {}, appId: {}, appStatus: {}", batchId, appId, appStatus);

        // PENDING, RUNNING, FINISHED, KILLED, FAILED, ZOMBIE, NOT_FOUND, UNKNOWN
        while (!isAppTerminated(appStatus)) {
            Thread.sleep(5000);
            batch = api.getBatchById(batchId);
            appId = batch.getAppId();
            appStatus = batch.getAppState();
            LOG.info("batchId: {}, appId: {}, appStatus: {}", batchId, appId, appStatus);
        }

        // INITIALIZED, PENDING, RUNNING, COMPILED, FINISHED, TIMEOUT, CANCELED, CLOSED, ERROR, UNKNOWN
        LOG.info("Batch status: {}", batch.getState());
        return appStatus;
    }

    public static boolean isAppTerminated(String appState) {
        if (appState == null) return false;

        switch (appState) {
            case "FAILED":
            case "KILLED":
```

```
        case "FINISHED":
            return true;
        default:
            return false;
    }
}

/**
 * 请求对象封装
 * @return {@link BatchRequest}
 */
public static BatchRequest buildBatchReq() {
    Map<String, String> batchConf = new HashMap<>();
    for (Map.Entry<String, ConfigValue> kv : appConf.getConfig("batch.conf").entrySet()) {
        batchConf.put(kv.getKey(), kv.getValue().unwrapped().toString());
    }
    String appName = appConf.getString("batch.appName");    public static String[] splitPrincipal(String principal) {
    return principal.split("[/@]");
}

/**
 * 提交任务并轮询获取任务状态
 * @param client
 * @return
 * @throws Exception
 */
public static String submitBatchJob(KyuubiRestClient client) throws Exception {
    BatchRestApi api = new BatchRestApi(client);
    BatchRequest req = buildBatchReq();
    Batch batch = api.createBatch(req);
    String batchId = batch.getId();
    String appId = batch.getAppId();
    String appStatus = batch.getAppState();
    LOG.info("batchId: {}, appId: {}, appStatus: {}", batchId, appId, appStatus);

    // PENDING, RUNNING, FINISHED, KILLED, FAILED, ZOMBIE, NOT_FOUND, UNKNOWN
    while (!isAppTerminated(appStatus)) {
        Thread.sleep(5000);
        batch = api.getBatchById(batchId);
        appId = batch.getAppId();
        appStatus = batch.getAppState();
        LOG.info("batchId: {}, appId: {}, appStatus: {}", batchId, appId, appStatus);
    }

    // INITIALIZED, PENDING, RUNNING, COMPILED, FINISHED, TIMEOUT, CANCELED, CLOSED, ERROR, UNKNOWN
    LOG.info("Batch status: {}", batch.getState());
    return appStatus;
}

public static boolean isAppTerminated(String appState) {
    if (appState == null) return false;

    switch (appState) {
        case "FAILED":
        case "KILLED":
        case "FINISHED":
            return true;
        default:
            return false;
    }
}

/**
 * 请求对象封装
 * @return {@link BatchRequest}
 */
public static BatchRequest buildBatchReq() {
    Map<String, String> batchConf = new HashMap<>();
```

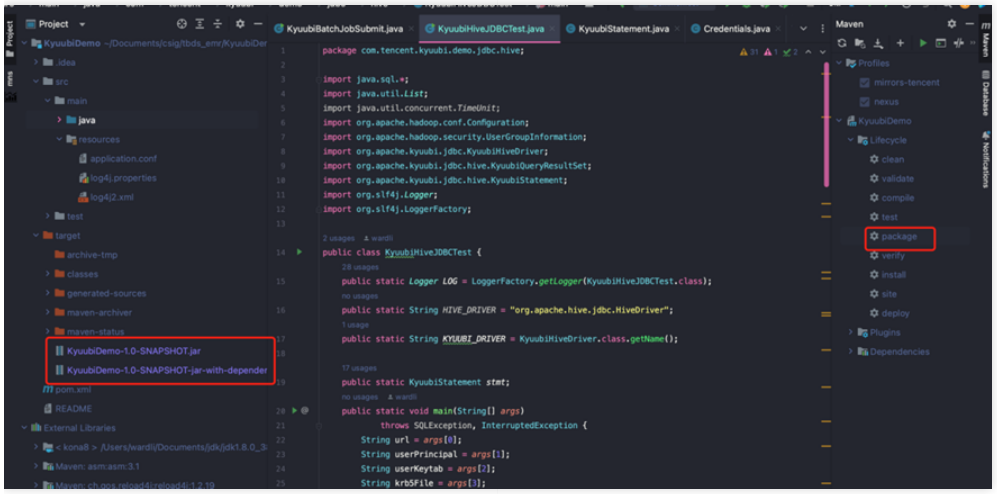
```
        for (Map.Entry<String, ConfigValue> kv : appConf.getConfig("batch.conf").entrySet()) {
            batchConf.put(kv.getKey(), kv.getValue().unwrapped().toString());
        }
        String appName = appConf.getString("batch.appName");
        String resource = appConf.getString("batch.resource");
        String mainClass = appConf.getString("batch.mainClass");
        List<String> args = appConf.getStringList("batch.args");
        return new BatchRequest("SPARK", resource, mainClass, appName, batchConf, args);
    }
}
```

打包发布

1. 打包
   以下使用 IntelliJ IDEA 说明示例工程代码编译过程。点击 IDEA 下方 Terminal 打开终端，切换到示例工程的 Spark 工程目录下，然后使用命令 mvn clean install 对工程进行打包，运行过程中可能还需要下载一些文件，直到出现 build success 表示打包成功。

通过打包
mvn clean package
或者idea上通过maven插件package进行打包

通过上述编译打包后，将在工程目录下 target 文件夹中看到打好的 jar 包，如图示中KyuubiDemo-1.0-SNAPSHOT-jar-with-dependencies.jar。



2. 开发机运行

1. 准备用户：参考获取用户认证。若是 Kerberos 环境，需要将用户的 keytab 文件下载到本地，然后上传至开发机。这里将 test 用户的 test.keytab 文件上传至开发机的 /tmp 目录。
2. 测试运行。

```
 # 1. Kerberos认证（Simple认证跳过该步）
[test@10 ~]$ klist -kt /tmp/test.keytab
Keytab name: FILE:/tmp/test.keytab
KVNO Timestamp           Principal
---- ------------------- ------------------------------------------------------
   1 11/22/2023 17:00:59 test@TBDS-CURPL8E5
   1 11/22/2023 17:00:59 test@TBDS-CURPL8E5
[test@10 ~]$ kinit -kt /tmp/test.keytab test@TBDS-CURPL8E5

# 2. 准备启动运行
[test@10 ~]$ java -jar KyuubiDemo-1.0-SNAPSHOT-jar-with-dependencies.jar "jdbc:hive2://172.16.12.14:10009/default;principal=hadoop/tbds-4ewgk
c73@TBDS-MD3Y2ZV9;#kyuubi.engine.type=SPARK_SQL;kyuubi.engine.share.level=CONNECTION"  "test@TBDS-CURPL8E5" "/tmp/test.keytab" "/et
```

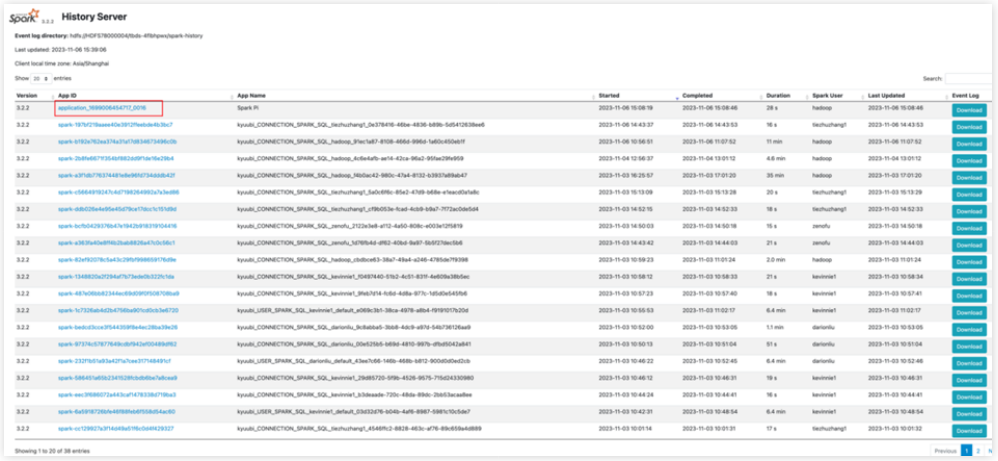
c/krb5.conf" "test"

3. Ranger授权：参考Ranger授权，确保 test 用户具有提交到 yarn default 队列的权限。

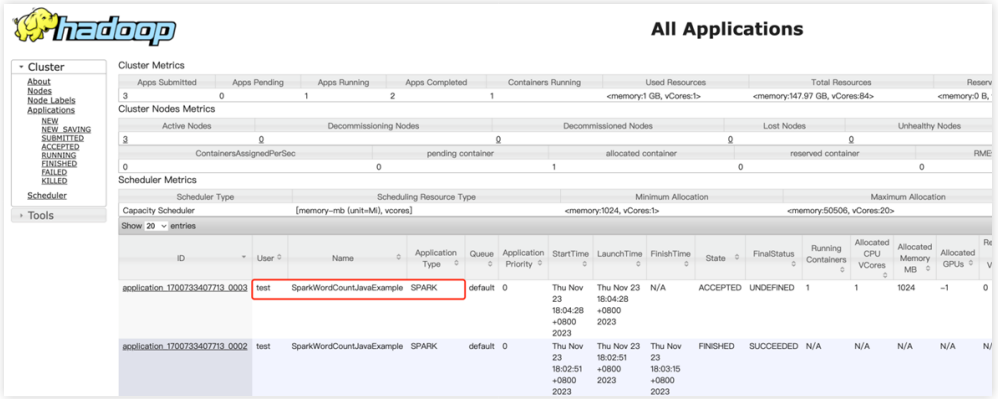
程序运维监控

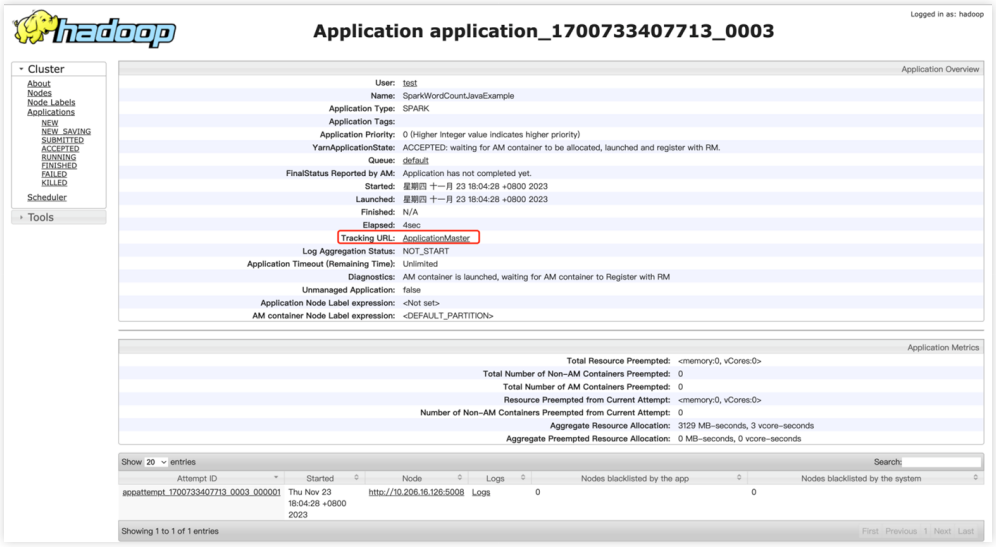
1. 监控
   登录 TBDS Manager 管控平台，点击"集群列表"，选择 Spark 程序运行所对应的集群。点击"集群服务"，点击 WebUI 地址跳转至 SparkHistory UI 界面。

换成对应外网IP：

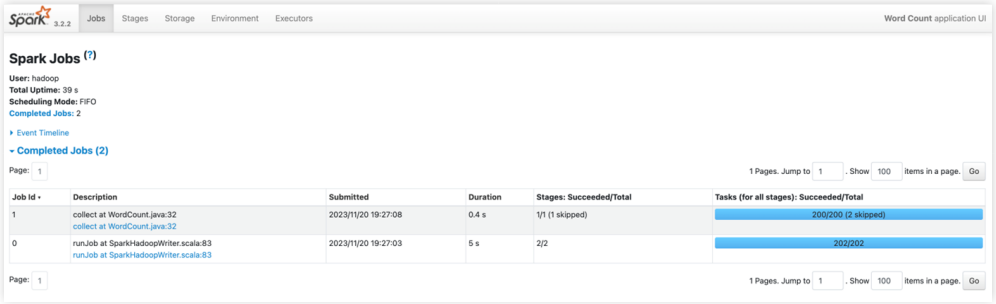


也可以YARN UI 页面根据 ApplicationID、User、Name 等信息，找到对应的Spark 任务。由于 Spark 任务的ApplicationID 在日志级别为 INFO 及以下才会输出至控制台，而 Spark 默认的日志级别为 WARN，可以根据任务启动后的日志等信息进行查找对应的 Spark 任务，点击 applicationID 链接，然后点击 Tracking URL 链接，跳转 Spark 作业监控页面。



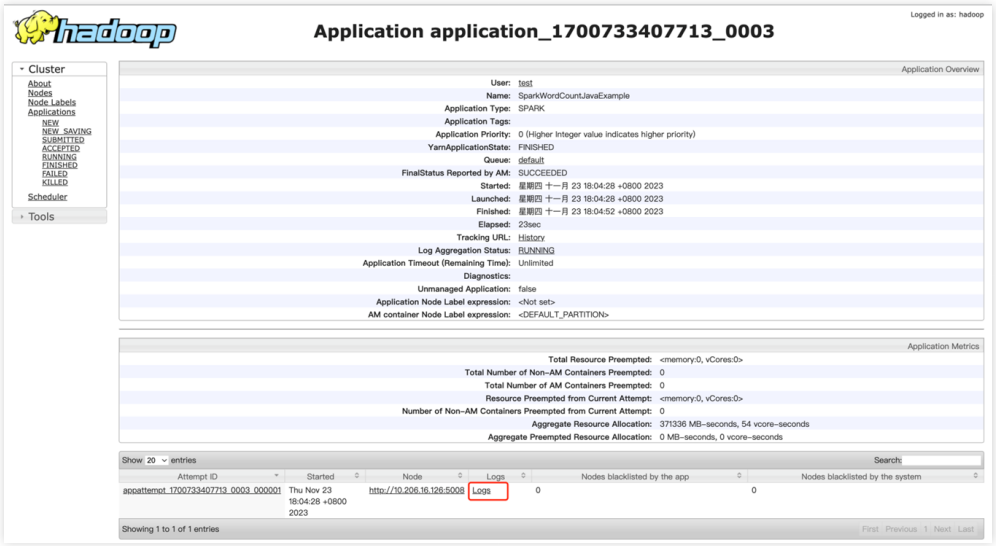


Spark 监控页面大致如下，更多介绍详见 Spark 官网。



2. 日志查看

若任务以 yarn client 模式提交，则日志将直接输出至控制台。若任务以 yarn cluster 模式提交，则需要到 YARN UI 页面根据ApplicationID、Name 等信息，找到对应的Spark 任务，点击 Logs 查看日志。



3. 作业操作

与其它提交至 YARN 的任务类似，可以通过 YARN 相关命令查看、停止任务，参考常用命令。如通过命令 yarn application -kill 停止 Spark 应用；通过命令 yarn logs -applicationId 查看 Spark 任务日志（cluster 模式）。

示例说明

```
# 1. Kerberos认证
//kerberos认证
klist -kt /etc/xxx.keytab
kinit -kt /etc/xxx.keytab  xx@TBDS-MD3Y2ZV9
klist


# 2. 创建连接
bin/beeline -u "jdbc:hive2://172.16.12.14:10009/default;principal=hadoop/tbds-4ewgkc73@TBDS-MD3Y2ZV9;#kyuubi.engine.type=SPARK_SQL;kyuubi.engine.share.level=CONNECTION"

# 2.1 创建TEMPORARY VIEW
CREATE TEMPORARY VIEW temp_view AS SELECT array(1, 2, 3) as numbers;
+---------+
| Result  |
+---------+
+---------+

# 2.2 下面使用explode函数将数组展开为多行 这将返回三行数据，每行包含一个数字
SELECT explode(numbers) as number FROM temp_view;
+---------+
| number  |
+---------+
| 1       |
| 2       |
| 3       |
+---------+

# 3. 使用collect_list函数将这些数字重新组合成一个数组 将返回一个包含一个数组的行，这个数组包含我们原来的数字
SELECT collect_list(number) as numbers FROM (SELECT explode(numbers) as number FROM temp_view);
+-----------+
| numbers   |
+-----------+
| [1,2,3]   |
+-----------+
```

api接口

TBDS 大数据平台上的 Kyuubi 访问接口与开源兼容，可参考REST API。

常用命令

命令说明

操作	命令	描述
服务启停	/usr/local/service/kyuubi/bin/kyuubi start /usr/local/service/kyuubi/bin/kyuubi stop	start : 启动Kyuubi Server服务 stop: 停止Kyuubi Server服务
交互式命令	/usr/local/service/kyuubi/bin/beeline	• -u <database url> e.g.: jdbc:hive2://172.16.12.14:10009/default;principal=hadoop/xx@xx • -n <username> kerberos环境不用传递这个参数 • -p <password> ldap认证模式或者kyuubi配置为用户名密码认证模式下使用 • -e <query> 可通过-e 指定查询一个sql ，则不进入交互式命令窗口 • -f <exec file> 可通过-f 指定查询一个sql文件，文件中写要执行的sql • --hiveconf property=value • --hivevar name=value
	/usr/local/service/kyuubi/bin/kyuubi-ctl	• create --hostUrl <value> --username <value> --password <value> 可以用于创建一个batch任务 • get -u <user> -et <engine-type> -es <engine-subdomain> -esl <engine-share-level> • delete batch|server|engine -u <user> -et <engine-type> -es <engine-subdomain> -esl <engine-share-level> • list batch|server|engine --batchType <value> --batchUser <value> --batchState <value> --createTime <yyyyMMddHHmmss> --endTime <yyyyMMddHHmmss> --from <value> --size <value> • log [batch] [options] [<batchID>] --from <value> --size <value> • submit batch -f <filename>

命令示例

beeline
bin/beeline -u "jdbc:hive2://172.16.12.14:10009/default;principal=hadoop/tbds-4ewgkc73@TBDS-MD3Y2ZV9;#kyuubi.engine.type=SPARK_SQL;kyuubi.engine.share.level=CONNECTION"

kyuubi-ctl
先通过list查看当前engine信息
./kyuubi-ctl list engine -et SPARK_SQL -zk tbds-etrmnatt-zookeeper-quorumpeermain-inner:2181 -u hadoop --conf kyuubi.engine.type=SPARK_SQL --conf kyuubi.engine.share.level=CONNECTION
因为上面beeline 链接的时候指定了参数，所致这里需要加上 --conf kyuubi.engine.type=SPARK_SQL --conf kyuubi.engine.share.level=CONNECTION

然后通过delete删除引擎
./kyuubi-ctl delete engine -et SPARK_SQL -zk tbds-etrmnatt-zookeeper-quorumpeermain-inner:2181 -u hadoop -s 172.16.9.31 -p 33091

常见问题

Kyuubi 连接执行报错

问题

TBDS执行SparkSQL-Kyuubi报错

原因

一体化/云原生 kyuubi 连接串错误，例如：

- principal 写在 # 号后面，# 号前是 beeline 使用参数，如认证。

. 号后是 kyuubi server 使用参数

- 云原生场景下，多加了不必要的 yarn 参数。

解决方案

参考文档：示例工程开发

引擎共享等级

根据 kyuubi-defaults 配置文件中提供的 kyuubi.engine.share.level 配置，您可以定义 Kyuubi 服务的引擎共享级别。
支持的共享等级与相关含义说明如下所示：

共享级别	含义	备注
CONNECTION	每个 Session 独占一个引擎	每个连接首次建立时，带来较大的引擎创建开销。
USER	每个用户独占一个引擎	开箱配置。
GROUP	每个资源组使用一个引擎	--
SERVER	每个集群使用一个引擎	隔离级别最低，需要管理员对集群安全性有足够把握。

租户级别引擎配置

在 Kyuubi 中，为了支持更细粒度租户级别的引擎配置，在 Kyuubi-defaults.conf 配置文件中，支持以 ___{username}__.*{config_key}形式定义此类配置，在用户名的两侧需要有连续三个下划线*，并以.符号将具体config_key与前缀分隔开。
这种形式配置生效的时机，只发生在引擎创建时。它的优先级，高于Spark-defaults与Kyuubi-defaults，但会被JDBC Connection URL与Set Command提供的配置覆盖。
下面提供部分配置示例。

```
# For system defaults
spark.master=local
spark.sql.adaptive.enabled=true

# For a user named kent
___kent___.spark.master=yarn
___kent___.spark.sql.adaptive.enabled=false

# For a user named bob
___bob___.spark.master=spark://master:7077
___bob___.spark.executor.memory=8g
```

引擎复用与 Subdomain 定义

以下场景基于 USER 共享级别。
Kyuubi 提交的 Spark 引擎，会在作业完成之后存在一段时间，在引擎存活的期间，继续提交的作业可以直接复用原引擎，省去了引擎提交等待创建的开销，可以有效提升作业性能。该存活时间由配置kyuubi.session.engine.idle.timeout决定，默认时间是PT30M（即半小时）。
对于同一个用户，在实际的生产环境中，可能希望根据不同的作业类型，在不同的引擎中进行作业运行。在 Kyuubi 中，这样的需求可以通过配置kyuubi.engine.share.level.subdomain满足。
这是一个连接级别的配置，在对应连接配置参数中直接指定，就可以将本次连接的作业提交到不同的 subdomain 当中。

```
beeline -n user1 -p user1_password -u "jdbc:hive2://master-1-1:10009/biz1?kyuubi.engine.share.level.subdomain=biz1" \
-f query1.sql

beeline -n user1 -p user1_password -u "jdbc:hive2://master-1-1:10009/biz2?kyuubi.engine.share.level.subdomain=biz2" \
-f query2.sql

beeline -n user1 -p user1_password -u "jdbc:hive2://master-1-1:10009/biz3?kyuubi.engine.share.level.subdomain=biz3" \
-f query3.sql
```

Curl API接口出现问题

```
[root@test ~]# curl --location --request POST 'http://[redacted]:10099/api/v1/batches' --negotiate -u :  --form 'batchRequest="{\"batchType\":\"pyspark\",\"name\":\"Spark Pi\"}";type=application/json' --form 'resourceFile=@"/usr/local/service/spark/examples/src/main/python/pi.py"'
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=ISO-8859-1"/>
<title>Error 401 No auth scheme matched for null</title>
</head>
<body><h2>HTTP ERROR 401 No auth scheme matched for null</h2>
<table>
<tr><th>URI:</th><td>/api/v1/batches</td></tr>
<tr><th>STATUS:</th><td>401</td></tr>
<tr><th>MESSAGE:</th><td>No auth scheme matched for null</td></tr>
<tr><th>SERVLET:</th><td>org.glassfish.jersey.servlet.ServletContainer-6be8ce1b</td></tr>
</table>
<hr/><a href="https://jetty.org/">Powered by Jetty:// 9.4.57.v20241219</a><hr/>

</body>
</html>
```

上述问题主要在Curl版本过低的情况下低版本Curl 不支持 SPNEGO，建议升级Curl的版本，测试环境下Curl版本>=7.71.1。

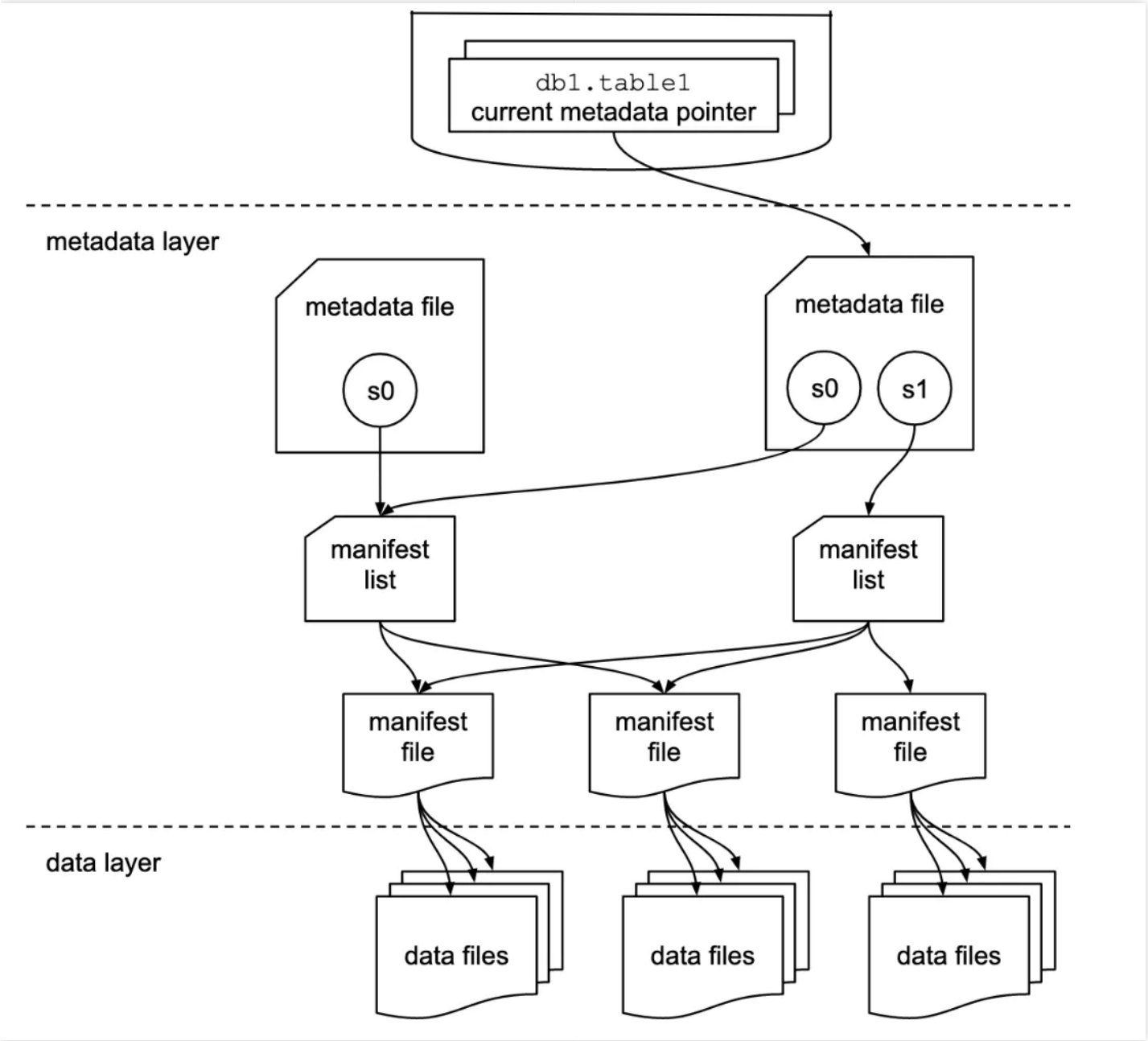
Iceberg开发
概述

简介

Apache Iceberg 是一种用于大型分析数据集的开放表格式，是实现数据湖架构的核心基础之一，支持 ACID 事务、时间旅行、模式演变、分区演变等功能。Iceberg 使用一种与 SQL 表类似的高性能表格式将表添加到计算引擎（包括 Spark、Trino、PrestoDB、Flink、Hive 和 Impala）。Iceberg 可以帮助消费者在使用数据湖时获得更好的性能和灵活性，简化数据架构，并可以更快地向消费者提供数据，同时降低计算和存储成本。

架构

Apache Iceberg 的架构如下图所示。



Apache Iceberg 对元数据进行了分层，这是 Iceberg 能提供高速的读取、写入和查询的关键。Iceberg 通过元数据层实现智能修剪，它首先可以有效地跟踪并判断清除哪些文件和目录，然后扫描文件统计信息以确定是否需要读取该文件以进行特定查询。Iceberg 的架构自上而下可分为三个层次：catalog层、元数据层、数据层。

1. catalog层
   catalog 层主要用于跟踪当前元数据文件的引用/指针。这通常是一些可以提供事务保证的存储，如关系数据库（PostgreSQL 等）或元存储（Hive、Project Nessie等）。
2. 元数据层（Metadata Layer）
   Iceberg 使用三层元数据文件，涵盖三个不同的范围。
   清单文件（Manifest files）——快照的子集。这些文件跟踪该子集中数据层中的各个文件以及元数据，以便进一步的修剪等工作。
   清单列表（Manifest lists）——定义表的快照并列出组成该快照的所有清单文件以及这些清单文件上的元数据

以供修剪。

元数据文件（Metadata files）——定义表并跟踪清单列表、当前和以前的快照、模式和分区方案。

3. 数据层（Data Layer）

数据层是Iceberg的最底层，主要用于保存表中的实际数据，由两种类型的文件组成：

数据文件（Data files）——以 Parquet、ORC 或 Avro 等文件格式存储的数据文件。

删除文件（Delete files）——跟踪数据文件中仍存在但应被视为已删除的记录。

快速使用

对于 Spark、Flink、Hive 引擎， TBDS 默认集成了标准 iceberg 1.6.1 版本。Trino 自带了 iceberg connector，原生支持 iceberg。

Spark

在 spark-default.conf 文件中，已经默认设置了以下三种 iceberg catalog，用户可以使用这些 catalog 对 iceberg 进行操作。

Name	Type	Implement
local	hadoop	org.apache.iceberg.spark.SparkCatalog
spark_catalog	hive	org.apache.iceberg.spark.SparkSessionCatalog
hive_prod	hive	org.apache.iceberg.spark.SparkSessionCatalog

kerberos认证

klist -kt /path/to/xxx.keytab
kinit -kt /path/to/xxx.keytab hadoop/xxx@TBDS-{ClusterId}

JAR包下载

- maven 仓库下载对应spark版本的JAR包，上传集群spark安装目录 /jars 下

```
[root@10 jars]# pwd
/usr/local/service/spark/jars
[root@10 jars]# ll | grep iceberg
-rwxr-xr-x 1 root   root    27781776 Jul 18 21:32 iceberg-spark-runtime-3.2_2.12-1.2.1.jar
[root@10 jars]#
```

- spark-defaults.conf配置
  使用Hive catalog

spark.sql.catalog.hive_prod=org.apache.iceberg.spark.SparkCatalog
spark.sql.catalog.hive_prod.type=hive
spark.sql.catalog.hive_prod.uri=thrift://xx.xx.xx.xx:xx

使用Hadoop catalog

```
spark.sql.catalog.hadoop_prod=org.apache.iceberg.spark.SparkCatalog
spark.sql.catalog.hadoop_prod.type=hadoop
spark.sql.catalog.hadoop_prod.warehouse=hdfs://xx.xx.x.xx:xx/warehouse/spark-iceberg
```

添加扩展参数

```
spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions
```

DDL

- 创建普通分区表

```
CREATE TABLE hive_prod.default.spark_iceberg1 (
   id bigint,
   data string,
   category string)
USING iceberg
PARTITIONED BY (category);

#插入
insert into spark_iceberg1 values (1,"TOM","a");
insert into spark_iceberg1 values (1,"jelly","b");
insert into spark_iceberg1 values (1,"Danny","c");
```

插入数据后分区结果如下：

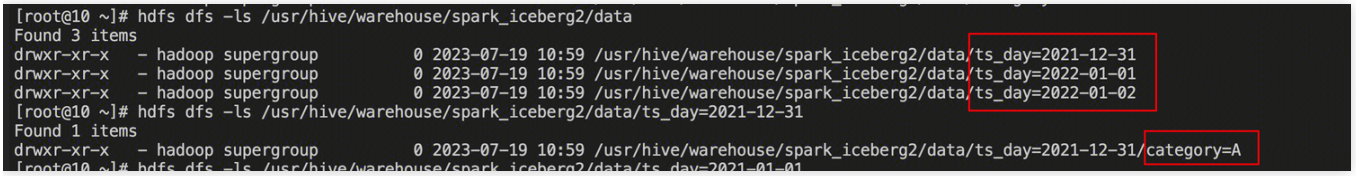
```
[root@10 ~]# hdfs dfs -ls /usr/hive/warehouse/spark_iceberg1/data
Found 3 items
drwxr-xr-x   - hadoop supergroup          0 2023-07-19 10:27 /usr/hive/warehouse/spark_iceberg1/data/category=a
drwxr-xr-x   - hadoop supergroup          0 2023-07-19 10:27 /usr/hive/warehouse/spark_iceberg1/data/category=b
drwxr-xr-x   - hadoop supergroup          0 2023-07-19 10:27 /usr/hive/warehouse/spark_iceberg1/data/category=c
[root@10 ~]#
```

- 创建隐藏分区表

```
CREATE TABLE hive_prod.default.spark_iceberg2 (
  id bigint,
  data string,
  category string,
  ts timestamp)
USING iceberg
PARTITIONED BY (days(ts), category);

#插入数据
INSERT INTO hive_prod.default.spark_iceberg2 (id, data, category, ts)
```

```
VALUES
  (1, 'value1', 'A', CAST('2022-01-01 00:00:00' AS TIMESTAMP)),
  (2, 'value2', 'B', CAST('2022-01-02 00:00:00' AS TIMESTAMP)),
  (3, 'value3', 'A', CAST('2022-01-03 00:00:00' AS TIMESTAMP));
```

插入数据后分区结果如下：



- 查询

```
CREATE TABLE hive_prod.default.spark_iceberg3 (
  id INT,
  data STRING,
  properties MAP<STRING, STRING>)
USING iceberg;

INSERT INTO spark_iceberg3 VALUES
  (1, 'value1', map('key1', 'valueA', 'key2', 'valueB')),
  (2, 'value2', map('key1', 'valueC', 'key2', 'valueD')),
  (3, 'value3', map('key1', 'valueA', 'key2', 'valueE'));

SELECT * FROM spark_iceberg3 WHERE properties['key1'] = 'valueA'
```

- 其他SQL

```
# merge into
/usr/local/service/spark/bin/spark-sql -e "
MERGE INTO default.test_target_table t
USING (SELECT * FROM default.test_source_table) s
ON t.id = s.id
WHEN MATCHED AND s.category = 'delete' THEN DELETE
WHEN MATCHED AND s.category = 'update' THEN UPDATE SET t.data = s.data, t.ts = s.ts
WHEN NOT MATCHED THEN INSERT *;
"
# 删除 iceberg 表
/usr/local/service/spark/bin/spark-sql -e "
DROP TABLE default.test_source_table;
"
# 删除 iceberg 表 并且删除数据
/usr/local/service/spark/bin/spark-sql -e "
DROP TABLE default.test_target_table PURGE;
```

Flink

Kerberos认证 (SIMPLE 认证可跳过)

```
//kerberos认证
klist -kt /path/to/xxx.keytab
kinit -kt /path/to/xxx.keytab  hadoop/xxx@TBDS-{ClusterId}
```

JAR包准备以及Flink启动

maven 仓库下载对应flink版本的JAR包：flink-sql-connector-hive-3.1.2_2.12-1.14.0.jar 和 iceberg-flink-runtime-1.14-1.2.1.jar
将JAR包移至/path/to/flink/libs（注意，当前TBDS-flink下的两个jar包在flink/opt/connect目录下，测试完后注意及时删除/flink/libs目录下的jar包避免产生冲突）。
启动命令如下

```
/path/to/flink/bin/sql-client.sh embedded
```

创建及使用 iceberg catalog

```
CREATE CATALOG iceberg_catalog WITH (
  'type'='iceberg',
  'catalog-type'='hive',
  'uri'='thrift://xx.xx.xx.xx:xx',
  'clients'='5',
  'property-version'='1',
  'warehouse'='hdfs://xx.xx.xx.xx:xx/warehouse/iceberg-hive'
);

USE CATALOG iceberg_catalog;
# 创建表
CREATE TABLE `default`.`sample` (
    id BIGINT COMMENT 'unique id',
    data STRING
);

# 写入表
INSERT INTO `default`.`sample` VALUES (1, 'a');

# 查询表
SET execution.runtime-mode = batch;
SELECT FROM `default`.`sample`;
```

Hive

kerberos认证

```
klist -kt /path/to/xxx.keytab
kinit -kt /path/to/xxx.keytab  hadoop/xxx@TBDS-{ClusterId}
```

JAR包下载

maven 仓库下载对应hive版本的JAR包，上传集群hive安装目录 /lib 下
Hive中Iceberg格式表插入数据时需要“libfb303-0.9.3.jar”包，将此包也上传到Hive服务端和客户端对应的lib目录下。
Hive加载第三方JAR包直接丢到${HIVE_HOME}/lib 下不生效
推荐方式，将JAR包上传到${HIVE_HOME}/auxlib 下，启动hive时指定目录加载jar包。

hive-site.xml配置与连接hive

```
#开启hive全局支持
iceberg.engine.hive.enabled=true
#jar包加载路径
hive.aux.jars.path=/usr/local/service/hive/auxlib
#Hive 3.1.2 或更高版本上使用 Tez 引擎，Tez 需要升级到 >= 0.10.1，并加如下配置
tez.mrreader.config.update.properties=hive.io.file.readcolumn.names,hive.io.file.readcolumn.ids
#Iceberg 开始0.11.0，当将 Hive 与 Tez 一起使用时，还必须禁用矢量化
hive.vectorized.execution.enabled=false
```

- catalog使用 hive

```
#设置catalog类型及catalog名
set iceberg.catalog.iceberg_hive.type=hive
#设置元数据地址
set iceberg.catalog.iceberg_hive.uri=thrift://xx.xx.xx.xx:xx
```

- catalog使用 hadoop

```
#设置catalog类型及catalog名
set iceberg.catalog.iceberg_hadoop.type=hadoop;
#catalog存储路径
set iceberg.catalog.iceberg_hadoop.warehouse=hdfs://xx.xx.xx.xx:xx/warehouse/iceberg-hadoop;
```

- 连接Hive

```
# 通过beeline连接 hiveserver
[hadoop@172 ~] beeline -u 'jdbc:hive2://xx.xx.xx.xx:xx/;principal=xx@TBDS-{ClusterId}'
```

DDL

- Catalog: Iceberg_hive下创建表：

```
CREATE TABLE iceberg_test1(id int, name string)
stored by 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'
TBLPROPERTIES('iceberg.catalog'='iceberg_hive');
insert into iceberg_test1 values (1, "Tom");
select * from iceberg_test1;
```

- Catalog: Iceberg_hadoop下创建表：

```
CREATE TABLE iceberg_test2(id int, name string)
stored by 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'
LOCATION 'hdfs://xx.xx.xx.xx:xx/warehouse/iceberg-hadoop/default/iceberg_test2'
TBLPROPERTIES('iceberg.catalog'='iceberg_hadoop');
```

```
insert into iceberg_test2 values (1 ,"TOM");
select * from iceberg_test2;
```

Trino

kerberos认证

```
klist -kt /path/to/xxx.keytab
kinit -kt /path/to/xxx.keytab  hadoop/xxx@TBDS-{ClusterId}
```

环境确认

- 在路径：/usr/local/service/trino/etc/catalog 下，添加 iceberg.properties 文件并重启trino，文件内容如下：

```
connector.name=iceberg
hive.hdfs.impersonation.enabled=true
hive.metastore.thrift.impersonation.enabled=true

hive.metastore.uri=thrift://xx.xx.xx.xx:xx
hive.config.resources=/usr/local/service/hadoop/etc/hadoop/core-site.xml,/usr/local/service/hadoop/etc/hadoop/hdfs-site.xml
hive.hdfs.authentication.type=KERBEROS
hive.hdfs.trino.keytab=/path/to/xxx.keytab
hive.hdfs.trino.principal=hadoop/xxx@TBDS-{ClusterId}
hive.metastore.authentication.type=KERBEROS
hive.metastore.client.principal=hadoop/xxx@TBDS-{ClusterId}
hive.metastore.service.principal=hadoop/xxx@TBDS-{ClusterId}
hive.metastore.client.keytab=/path/to/xxx.keytab
```

- trino-cli

  确认是否存在cd /usr/local/service/trino/client

  如果不存在做如下操作

  i. 按照如下链接下载客户端

  https://repo1.maven.org/maven2/io/trino/trino-cli/389/trino-cli-389-executable.jar

  ii. 将步骤1所在下载的JAR放在集群的任一服务器，并将其改名为trino

```
# 将下载的客户端改名
```

```
mv trino-cli-389-executable.jar trino
# 查看版本信息
./trino --version
# 通过客户端连接访问trino，把${ip}更换成trino coordinator节点ip
./trino --server ${ip}:9000 --catalog hive --user hadoop --debug
```

登录trino-cli

```
[hadoop@10 ~]# /usr/local/service/trino/client/trino-cli --server http://{coordinator_ip}:9000 --user hadoop --catalog iceberg --schema default
```

DDL

```
#一些操作
show catalogs;

use iceberg.iceberg_auto_functional_spark_test;

show tables;

select * from iceberg_01_upper;

DESCRIBE iceberg_01_upper;
```

更多使用样例可参考官方文档：

Spark：Getting Started：https://iceberg.apache.org/docs/1.4.3/spark-getting-started/

Flink：Flink Getting Started：https://iceberg.apache.org/docs/1.4.3/flink/

Hive：Hive：https://iceberg.apache.org/docs/1.4.3/hive/

Trino：Trino：https://trino.io/docs/current/connector/iceberg.html

性能优化

Apache Iceberg 是一种表格格式，旨在简化数据湖管理并增强工作负载性能。不同的用例可能会优先考虑不同的方面，例如成本、读取性能、写入性能或数据保留，因此 Iceberg 提供了配置选项来管理这些权衡。本指南为优化和微调 Iceberg 工作负载以满足您的需求提供了深入的见解。

主题

- 一般最佳实践
- 优化读取性能
- 优化写入性能
- 优化存储
- 数据湖表维护

一般最佳实践

无论您的用例如何，当您在 TBDS 上使用 Apache Iceberg 时，我们建议您遵循以下一般的最佳实践。

使用 Iceberg 格式版本 2

默认情况下，TBDS 使用 Iceberg 格式版本 2（Iceberg Format Version 2）。当您在 TBDS 上使用 Spark 创建 Iceberg 表时，请按照 Iceberg 文档中所述指定格式版本。

使用 Zstandard (ZSTD) 压缩

Iceberg 的默认压缩编解码器是 gzip（GZIP），可以通过表属性进行修改。我们建议使用 ZSTD 压缩编解码器，因为它在 GZIP 和 Snappy 之间取得了平衡，并且在不影响压缩比的情况下提供了良好的读写性能。此外，ZSTD 允许根据需要调整压缩级别。
Snappy 可能提供最佳的整体读取和写入性能，但其压缩率低于 GZIP 和 ZSTD。如果您优先考虑性能，即使这意味着在存储集群中存储更大的数据量，Snappy 也是一个不错的选择。

优化读取性能

本节讨论了可以独立于引擎调整的表属性，以优化读取性能。

分区

与 Hive 表类似，Iceberg 使用分区作为索引的主要层，以避免读取不必要的元数据文件和数据文件。列统计数据也被视为索引的辅助层，以进一步改进查询计划，从而缩短总体执行时间。

对您的数据进行分区

要减少查询 Iceberg 表时扫描的数据量，请选择与预期读取模式一致的平衡分区策略：

- 确定查询中经常使用的列：这些是理想的分区候选对象。例如，如果您通常查询特定日期的数据，分区列的自然示例就是日期列。
- 选择低基数分区列：避免创建过多的分区。过多的分区会增加表中的文件数量，从而对查询性能产生负面影响。根据经验，"分区过多"可以定义为大多数分区中的数据大小小于设定值的 2-5 倍的情况。
  注意：如果您通常通过对高基数列（例如，可以包含数千个值的列）使用过滤器进行查询，请使用带有存储桶转换的 Iceberg 隐藏分区功能，如下一节所述。

使用隐藏分区

如果您的查询通常根据表列的派生项进行筛选，请使用隐藏分区，而不是显式创建新列作为分区。隐藏分区允许您通过转换函数即时创建分区，从而避免在表中增加额外的列。
例如，在具有时间戳列的数据集中（例如，2023-01-01 09:00:00），可以使用分区转换从时间戳中提取日期部分并即时创建这些分区，而不是使用解析后的日期创建新列（例如，2023-01-01）。
隐藏分区最常见的用例包括：

- 当数据包含时间戳列时，按日期或时间进行分区。Iceberg 提供多种变换来提取时间戳的日期或时间部分。
- 对高基数列进行分区时，使用 Iceberg 的存储桶转换，通过在分区列上使用哈希函数，将多个分区值组合成更少的隐藏（存储桶）分区。
  有关所有可用分区转换的概述，请参阅 Iceberg 文档中的分区转换部分。

使用分区演变

当现有的分区策略不是最佳时，请使用 Iceberg 的分区演变（partition evolution）。例如，如果您选择的每小时分区结果太小（每个分区只有几兆字节），可以考虑切换到每日或每月分区。
分区演变的另一个有效用途是，当数据量发生变化并且当前的分区策略随着时间的推移而变得不那么有效时。
有关如何演变分区的说明，请参阅 Iceberg 文档中的 ALTER TABLE SQL 扩展。

调整文件大小

优化查询性能包括最大限度地减少表中的小文件数量。为了获得良好的查询性能，我们通常建议保留大于 100 MB 的 Parquet 和 ORC 文件。
文件大小也会影响 Iceberg 表的查询计划。随着表中文件数量的增加，元数据文件的大小也随之增加。较大的元数据文件可能会导致查询计划变慢。因此，当表大小增加时，请增加文件大小以控制元数据的指数级扩展。

设置目标文件和行组的大小

Iceberg 提供了以下用于调整数据文件布局的关键配置参数。我们建议使用这些参数来设置目标文件大小和行组（row group）或行标（stripe）大小。

参数	默认值	评论
write.target-filesize-bytes	512MB	该参数指定 Iceberg 将创建的最大文件大小。但是，写入某些文件的大小可能小于此限制。

参数	默认值	评论
write.parquet.rowgroup-size-bytes	128MB	Parquet 和 ORC 都将数据分块存储，因此引擎可以避免在某些操作中读取整个文件。
write.orc.stripe-size-bytes	64MB	
write.distribution-mode	无，适用于 Iceberg 1.1 及更低版本	Iceberg 要求 Spark 在写入存储之前在其任务之间对数据进行排序。

根据表格大小调整参数

根据您的预期表格大小，请遵循以下一般准则：

- 小表（最多几千兆字节）：将目标文件大小减至 128 MB。还可以减小行组或条带大小（例如，减至 8 MB 或 16 MB）。
- 中型到大型表（从几千兆字节到几百千兆字节不等）：默认值是这些表的良好起点。如果查询选择性很强，请调整行组或条带大小（例如，调整到 16 MB）。
- 非常大的表（数百 GB 或 TB）：将目标文件大小增加到 1024 MB 或更多。如果查询通常会提取大量数据，请考虑增加行组或条带大小。
  为确保写入 Iceberg 表的 Spark 应用程序创建大小合适的文件，请将 write.distribution-mode 属性设置为 hash 或 range。有关这些模式之间差异的详细说明，请参阅 Iceberg 文档中的写入分发模式部分。
  这些是一般准则。我们建议您运行测试以确定最适合您的特定表和工作负载的值。

定期进行维护

上述配置设置了写入任务可以创建的最大文件大小，但不能保证文件会达到该大小。为确保文件大小合适，请定期运行维护，将小文件合并成较大的文件。有关运行维护的详细指导，请参阅本指南后面的 Iceberg 维护部分。

优化写入性能

本节讨论了可以调整的表属性，以优化 Iceberg 表的写入性能，独立于引擎。

设置表格分配模式

Iceberg 提供多种写入分配模式，这些模式定义了写入数据在 Spark 任务中的分布方式。有关可用模式的概述，请参阅 Iceberg 文档中的写入分发模式部分。
对于优先考虑写入速度的用例，尤其是在流式工作负载中，建议将 write.distribution-mode 设置为 none。这样可以确保 Iceberg 不会请求额外的 Spark 洗牌（shuffle），并且可以在 Spark 任务中可用时写入数据。
注意：将写入分配模式设置为 none 往往会生成大量小文件，从而降低读取性能。我们建议定期进行压缩，将这些小文件整合到大小合适的文件中，以提高查询性能。

选择正确的更新策略

如果您的用例可以接受较慢的写入操作，可以使用 copy-on-write 策略来优化读取性能。这是 Iceberg 的默认策略。
Copy-on-write 可以提高读取性能，因为文件是以读取优化的方式直接写入存储的。但是，与 merge-on-read 相比，每次写入操作花费的时间更长，消耗的计算资源也更多。这在读取和写入延迟之间进行了典型的权衡。通常，copy-on-write 非常适合大多数更新并集中在同一个表分区中的用例（例如，用于每日批量加载）。
Copy-on-write 配置（write.update.mode、write.delete.mode 和 write.merge.mode）可以在表级别进行设置，也可以在应用程序端独立设置。

使用 Merge-on-read 策略

当您的用例可以接受对最新数据执行较慢的读取操作时，请使用 merge-on-read 策略来优化写入性能。在使用 merge-on-read 策略时，Iceberg 会将更新和删除内容作为单独的小文件写入存储器。读取表格时，读取器必须将这些更改与基础文件合并，以返回最新的数据视图。这会导致读取操作的性能下降，但会加快更新和删除的写入速度。通常，merge-on-read 适用于具有更新的流式处理工作负载或更新较少且分布在多个表分区中的任务。
使用 merge-on-read 需要定期进行维护，以防止读取性能随着时间的推移而下降。 数据湖维护可将最新和删除与现有数据文件进行协调，以创建一组新的数据文件，从而消除读取性能的损失。

选择正确的文件格式

Iceberg 支持以 Parquet、ORC 和 Avro 格式写入数据。默认格式是 Parquet（Parquet）。Parquet 和 ORC 是列式格式，提供卓越的读取性能，但通常写入速度较慢。这代表了读取和写入性能之间的典型权衡。
如果写入速度对您的用例很重要，例如在流媒体工作负载中，建议在编写器的选项中设置为 Avro 格式，以Avro 格式写入。由于 Avro 是一种基于行的格式，因此它以较慢的读取性能为代价提供更快的写入速度。
要提高读取性能，请定期进行数据湖维护，将小 Avro 文件合并并转换为较大的 Parquet 文件。维护过程的结果取决于 write.format.default 表格设置。Iceberg 的默认格式是 Parquet，因此，如果您用 Avro 编写然后运行维护，Iceberg 会将 Avro 文件转换为 Parquet 文件。

示例配置

```
CREATE TABLE IF NOT EXISTS glue_catalog.<DB_NAME>.<TABLE_NAME> (
  Col_1 float,
  -- 其他列 ...
  ts timestamp
)
USING iceberg
PARTITIONED BY (days(ts))
OPTIONS (
  'format-version'='2',
  'write.format.default'='parquet'
)
query = df \
  .writeStream \
  .format("iceberg") \
  .option("write-format", "avro") \
```

```
.outputMode("append") \
.trigger(processingTime='60 seconds') \
.option("path", f"glue_catalog.{DB_NAME}.{TABLE_NAME}") \
.option("checkpointLocation", f"s3://{BUCKET_NAME}/checkpoints/iceberg/") \
.start()
```

优化存储

更新或删除 Iceberg 表中的数据会增加数据的副本数量，如下图所示。运行压缩也是如此：它增加了存储中的数据副本数量。这是因为 Iceberg 将所有表的底层文件视为不可变的。
按照本节中的最佳实践来管理存储成本。

存档或删除历史快照

对于向 Iceberg 表提交的每笔事务（插入、更新、合并、压缩），都会创建该表的新版本或快照。随着时间的推移，存储集群中的版本数量和元数据文件数量会不断增加。
快照隔离、表回滚和时空旅行查询等功能需要保留表的快照。但是，存储成本会随着您保留的版本数量的增加而增加。

设计模式

解决方案	使用案例
删除旧快照	使用 TBDS 中的 Amoro 删除旧的快照。这种方法会删除不再需要的快照以降低存储成本。您可以根据数据保留要求配置应保留多少快照或保留多长时间。
设置保留策略	在 Iceberg 中使用标签标记特定的快照并定义保留策略。例如，您可以每月保留一个快照，为期一年，然后在 TBDS 上删除剩余的未加标签的中间快照。这有助于遵守业务或法律要求，同时降低存储成本。

删除孤立文件

在某些情况下，Iceberg 应用程序可能会在提交事务之前失败。这会将数据文件留在存储集群中。由于没有提交，这些文件不会与任何表相关联，因此需要异步清理。
您可以在 TBDS 上使用 Spark 运行 remove_orphan_files 程序。此操作会产生计算成本，并且必须单独进行计划。

数据湖表维护

Iceberg 包含的功能允许您在向表写入数据后执行表维护操作。一些维护操作侧重于优化元数据文件，而另一些维护操作则增强了数据在文件中的聚集方式，以便查询引擎可以有效地找到响应用户请求所需的信息。本节重点介绍与压缩相关的优化。

Iceberg 表维护

在 TBDS 中，您可以使用 Amoro 来执行以下任务：

- 将小文件合并成大文件：通常超过 100 MB。这种技术被称为文件重写。
- 将删除文件与数据文件合并：删除文件由使用 merge-on-read 方法的更新或删除生成。
- （重新）根据查询模式对数据进行排序：可以在没有任何排序顺序的情况下写入数据，也可以使用适合写入和更新的排序顺序写入数据。
- 使用空间填充曲线对数据进行聚类：优化不同的查询模式，尤其是 Z 顺序排序。

调整压缩行为

以下关键属性控制 Amoro 如何选择数据文件进行压缩：

属性	默认值	描述
self-optimizing.enabled	true	是否启用自动压缩优化
self-optimizing.target-size	134217728(128MB)	压缩后目标文件大小
self-optimizing.max-file-count	10000	单次压缩处理的最大文件数量
self-optimizing.fragment-ratio	8	碎片文件大小阈值比例。目标大小除以此比例得到实际碎片文件大小阈值
self-optimizing.minor.trigger.file-count	12	触发小型压缩的最小文件数量(碎片文件数量和等值删除文件数量之和)
self-optimizing.minor.trigger.interval	3600000(1小时)	触发小型压缩的时间间隔(毫秒)
self-optimizing.major.trigger.duplicate-ratio	0.1	触发大型压缩的段文件重复数据比例
self-optimizing.min-target-size-ratio	0.75	小于目标大小的段文件阈值比例。低于此阈值的段文件将被考虑重写
self-optimizing.quota	0.1	压缩可使用的 CPU 资源配额
self-optimizing.execute.num-retries	5	压缩失败后的重试次数

调整 Spark 集群大小以运行压缩

以下示例使用 TBDS Spark 引擎

示例 Spark 配置

spark.dynamicAllocation.enabled = FALSE
spark.executor.memory = 20 GB
spark.executor.instances = 5
要加快压缩速度，可以通过增加并行压缩的文件组数量来横向扩展。例如：

- 手动缩放（例如，按系数 4）
- 设置 MAX_CONCURRENT_FILE_GROUP_REWRITES=4
- 设置 spark.executor.instances=20
- 设置 spark.dynamicAllocation.enabled = FALSE
  或者，使用动态缩放：
- 设置 spark.dynamicAllocation.enabled=TRUE

数据湖维护的建议

使用案例	建议
按计划运行维护	可以使用 TBDS 内置的 Amoro 组件按计划执行数据湖维护
压缩大量小文件	如果您希望压缩大量小文件，请使用 TBDS Spark 引擎执行
对数据进行排序的维护	使用 TBDS Spark 引擎执行，因为排序是一项昂贵的操作，可能需要将数据重新读写。
使用 Z 顺序排序对数据进行聚类的维护	使用 TBDS Spark 引擎执行，因为 Z 顺序排序是一项非常昂贵的操作，可能需要将数据泄漏到磁盘。

总结

通过遵循本指南中的最佳实践，您可以有效地优化 Apache Iceberg 在 TBDS 上的工作负载，提升性能并管理存储成本。无论是优化读取性能、写入性能还是存储管理，Iceberg 提供了灵活的配置选项来满足不同的业务需求。定期进行维护操作，确保您的数据湖保持高效和可扩展。

常用参数

读取属性

属性名称	默认值	描述
read.split.target-size	134217728（128MB）	设置数据被拆分后的目标大小。
read.split.metadata-target-size	33554432（32MB）	设置元数据被拆分后的目标大小。
read.split.planning-lookback	10	设置拆分数据时的 bin 的数量。
read.split.open-file-cost	4194304（4MB）	打开文件的预计成本，用来作为合并拆分时的最小权重。
read.parquet.vectorization.enabled	true	控制是否使用 Parquet 矢量化读取。
read.parquet.vectorization.batch-size	5000	Parquet 矢量化读取的批处理大小。
read.orc.vectorization.enabled	false	用于控制是否使用 orc 矢量化读取。
read.orc.vectorization.batch-size	5000	orc 矢量化读取的批处理大小。

写入属性

属性名称	默认值	描述
write.format.default	parquet	表的默认文件格式；parquet、Avro 或 orc
write.delete.format.default	data file format	表的默认删除文件格式；parquet、Avro 或 orc
write.parquet.row-group-size-bytes	134217728 (128 MB)	Parquet 行组大小
write.parquet.page-size-bytes	1048576 (1 MB)	Parquet 页面大小
write.parquet.page-row-limit	20000	Parquet 页面行限制
write.parquet.dict-size-bytes	2097152 (2 MB)	Parquet 字典页面大小

属性名称	默认值	描述
write.parquet.compression-codec	zstd	Parquet 压缩编解码器：zstd、brotli、lz4、gzip、snappy、uncompressed
write.parquet.compression-level	null	Parquet 压缩编解码器级别
write.parquet.bloom-filter-enabled.column.col1	(not set)	提示 parquet 为列 col1 写一个布隆过滤器
write.parquet.bloom-filter-max-bytes	1048576 (1 MB)	布隆过滤器位集的最大字节数
write.parquet.bloom-filter-fpp.column.col1	0.01	应用于 col1 的布隆过滤器的假阳性概率（必须 > 0.0 且 < 1.0）
write.Avro.compression-codec	gzip	Avro 压缩编解码器：gzip（9 级 deflate）、zstd、snappy、uncompressed
write.avro.compression-level	null	Avro 压缩级别
write.orc.stripe-size-bytes	67108864 (64 MB)	定义默认 ORC 条带大小（以字节为单位）
write.orc.block-size-bytes	268435456 (256 MB)	定义 ORC 文件的默认文件系统块大小
write.orc.compression-codec	zlib	ORC 压缩编解码器：zstd、lz4、lzo、zlib、snappy、none
write.orc.compression-strategy	speed	ORC压缩策略：速度、压缩
write.orc.bloom.filter.columns	(not set)	必须为其创建布隆过滤器的列名的逗号分隔列表
write.orc.bloom.filter.fpp	0.05	布隆过滤器的假阳性概率（必须 > 0.0 且 < 1.0）
write.location-provider.impl	null	LocationProvider 的可选自定义实现
write.metadata.compression-codec	none	元数据压缩编解码器；none 或 gzip
write.metadata.metrics.max-inferred-column-defaults	100	定义收集指标的顶级列的最大数量。对于具有嵌套字段的表，存储的指标数量可以高于此限制
write.metadata.metrics.default	truncate(16)	表中所有列的默认度量模式：none、counts、truncate(length)、full

属性名称	默认值	描述
write.metadata.metrics.column.col1	(not set)	列“col1”的度量模式允许对每个列进行调整：none、counts、truncate(length)、full
write.target-file-size-bytes	536870912 (512 MB)	控制生成的文件的大小，以值大小为目标
write.delete.target-file-size-bytes	67108864 (64 MB)	控制生成的删除文件的大小，以值大小为目标
write.distribution-mode	none, see engines for specific defaults	定义写入数据的分布： none：不随机排列行； hash：按分区键进行哈希分布； range：按分区键或排序键（如果表有SortOrder）进行范围分布
write.delete.distribution-mode	hash	定义写入删除数据的分布
write.update.distribution-mode	hash	定义写入更新数据的分布
write.merge.distribution-mode	none	定义写入合并数据的分布
write.wap.enabled	false	启用写入-审计-发布写入
write.summary.partition-limit	0	如果更改的分区数小于此限制，则在快照摘要中包含分区级摘要统计信息
write.metadata.delete-after-commit.enabled	false	控制提交后是否删除最老的跟踪版本元数据文件
write.metadata.previous-versions-max	100	提交后删除之前保留的先前版本元数据文件的最大数量
write.spark.fanout.enabled	false	启用 Spark 中的 fanout writer，不需要对数据进行聚类；使用更多内存
write.object-storage.enabled	false	启用对象存储位置提供程序，将哈希组件添加到文件路径
write.object-storage.partitioned-paths	true	在文件路径中包含分区值
write.data.path	table location + /data	数据文件的基本位置
write.metadata.path	table location + /metadata	元数据文件的基本位置

属性名称	默认值	描述
write.delete.mode	copy-on-write	用于删除命令的模式：copy-on-write or merge-on-read (v2 only)
write.delete.isolation-level	serializable	删除命令的隔离级别：serializable or snapshot
write.update.mode	copy-on-write	用于更新命令的模式：copy-on-write or merge-on-read (v2 only)
write.update.isolation-level	serializable	更新命令的隔离级别：serializable or snapshot
write.merge.mode	copy-on-write	合并命令使用的模式：copy-on-write or merge-on-read (v2 only)
write.merge.isolation-level	serializable	合并命令的隔离级别：serializable or snapshot

表行为属性

属性名称	默认值	描述
commit.retry.num-retries	4	失败前重试提交的次数
commit.retry.min-wait-ms	100	重试提交之前等待的最短时间（以毫秒为单位）
commit.retry.max-wait-ms	60000 (1 min)	重试提交之前等待的最长时间（以毫秒为单位）
commit.retry.total-timeout-ms	1800000 (30 min)	提交的总重试超时时间（以毫秒为单位）
commit.status-check.num-retries	3	在由于未知提交状态而失败之前，在连接丢失后检查提交是否成功的次数
commit.status-check.min-wait-ms	1000 (1s)	重试状态检查前等待的最短时间（以毫秒为单位）
commit.status-check.max-wait-ms	60000 (1 min)	重试状态检查之前等待的最长时间（以毫秒为单位）
commit.status-check.total-timeout-ms	1800000 (30 min)	提交状态检查必须成功的总超时时间（以毫秒为单位）

属性名称	默认值	描述
commit.manifest.target-size-bytes	8388608 (8 MB)	合并清单文件时的目标大小
commit.manifest.min-count-to-merge	100	合并前需累积的最少清单数量
commit.manifest-merge.enabled	true	控制是否在写入时自动合并清单
history.expire.max-snapshot-age-ms	432000000 (5 days)	快照过期时保留在表及其所有分支上的快照的默认最大期限
history.expire.min-snapshots-to-keep	1	快照过期时保留在表及其所有分支上的默认最小快照数
history.expire.max-ref-age-ms	Long.MAX_VALUE (forever)	对于主分支以外的快照引用，快照过期时保留的快照引用的默认最大期限。 主分支永不过期。

更多参数详见Iceberg官方文档：

Iceberg Configuration：https://iceberg.apache.org/docs/nightly/configuration/

最佳实践

基于Spark SQL Iceberg表结构，上层sql无需变动直接使用

以Spark SQL 3.x 读写Iceberg操作

```
# 启动spark-sql 客户端：
cd /usr/hdp/2.2.0.0-2041/spark
export SPARK_HOME=/path/to/spark/home/
export HADOOP_CONF_DIR=/path/to/hdfs/conf/
export SPARK_CONF_DIR=/path/to/spark/conf/
export HIVE_CONF_DIR=/path/to/hive/conf/
export HADOOP_USER_NAME=tbds
./bin/spark-sql --conf spark.sql.adaptive.enabled=true \
--conf spark.sql.catalog.local=org.apache.iceberg.spark.SparkCatalog \
--conf spark.sql.catalog.local.type=hadoop \
--conf spark.sql.catalog.local.warehouse=/tmp/hive/warehouse \
--conf spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCatalog \
--conf spark.sql.catalog.spark_catalog.type=hive \
--conf spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions \
--conf spark.sql.warehouse.dir=/apps/hive/warehouse


# 建库语句：
CREATE DATABASE if not exists iceberg_db_check ;

# 建表语句：
drop table if exists iceberg_db_check.tb_spark_check1;
CREATE TABLE iceberg_db_check.tb_spark_check1 (
    id bigint,
    data string,
    category string)
USING iceberg;

#插入数据
INSERT INTO iceberg_db_check.tb_spark_check1 VALUES (1,'1','1'),(2,'2','2');
#修改字段为分区
ALTER TABLE iceberg_db_check.tb_spark_check1 ADD PARTITION FIELD category;
#插入数据
INSERT INTO iceberg_db_check.tb_spark_check1 VALUES (3,'3','3'),(4,'4','4');
#删除字段分区
```

```
ALTER TABLE iceberg_db_check.tb_spark_check1 DROP PARTITION FIELD category;
#插入数据
INSERT INTO iceberg_db_check.tb_spark_check1 VALUES (5,'5','5'),(6,'6','6');

#查询
select * from iceberg_db_check.tb_spark_check1;
```

Spark Structured Streaming实时写入Iceberg

使用 Structured Streaming 从 Kafka 中实时读取数据，然后将结果实时写入到 Iceberg 中（注：Structured Streaming 只支持实时向 Iceberg 中写入数据，不支持实时从 Iceberg 中读取数据）。

创建Kafka topic

启动Kafka集群，创建“kafka-iceberg-topic”

```
./kafka-topics.sh  --zookeeper node3:2181,node4:2181,node5:2181
--create  --topic kafka-iceberg-topic  --partitions 3 --replication-factor 3
```

编写向Kafka生产数据代码

```
/**
  * 向Kafka中写入数据
  */
object WriteDataToKafka {
  def main(args: Array[String]): Unit = {
    val props = new Properties()
    props.put("bootstrap.servers", "node1:9092,node2:9092,node3:9092")
    props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
    props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

    val producer = new KafkaProducer[String,String](props)
    var counter = 0
    var keyFlag = 0
    while(true){
      counter +=1
      keyFlag +=1
      val content: String = userlogs()
```

```
      producer.send(new ProducerRecord[String, String]("kafka-iceberg-topic", content))
      //producer.send(new ProducerRecord[String, String]("kafka-iceberg-topic", s"key-$keyFlag", conten
t))
      if(0 == counter%100){
        counter = 0
        Thread.sleep(5000)
      }
    }
    producer.close()
  }

  def userlogs()={
    val userLogBuffer = new StringBuffer("")
    val timestamp = new Date().getTime();
    var userID = 0L
    var pageID = 0L

    //随机生成的用户ID
    userID = Random.nextInt(2000)

    //随机生成的页面ID
    pageID =  Random.nextInt(2000);

    //随机生成Channel
    val channelNames = Array[String]("Spark","Scala","Kafka","Flink","Hadoop","Storm","Hive","Impal
a","HBase","ML")
    val channel = channelNames(Random.nextInt(10))

    val actionNames = Array[String]("View", "Register")
    //随机生成action行为
    val action = actionNames(Random.nextInt(2))

    val dateToday = new SimpleDateFormat("yyyy-MM-dd").format(new Date())
    userLogBuffer.append(dateToday)
      .append("\t")
      .append(timestamp)
      .append("\t")
      .append(userID)
      .append("\t")
      .append(pageID)
      .append("\t")
      .append(channel)
      .append("\t")
      .append(action)
    System.out.println(userLogBuffer.toString())
    userLogBuffer.toString()
  }
```

```
}
```

编写 Structured Streaming 读取 Kafka 数据实时写入 Iceberg

```
object StructuredStreamingSinkIceberg {
  def main(args: Array[String]): Unit = {
    //1.准备对象
    val spark: SparkSession = SparkSession.builder().master("local").appName("StructuredSinkIceberg")
      //指定hadoop catalog，catalog名称为hadoop_prod
      .config("spark.sql.catalog.hadoop_prod", "org.apache.iceberg.spark.SparkCatalog")
      .config("spark.sql.catalog.hadoop_prod.type", "hadoop")
      .config("spark.sql.catalog.hadoop_prod.warehouse", "hdfs://hdfsCluster/structuredstreaming")
      .getOrCreate()
//    spark.sparkContext.setLogLevel("Error")

    //2.创建Iceberg 表
    spark.sql(
      """
        |create table if not exists hadoop_prod.iceberg_db.iceberg_table (
        | current_day string,
        | user_id string,
        | page_id string,
        | channel string,
        | action string
        |) using iceberg
      """.stripMargin)

    val checkpointPath = "hdfs://hdfsCluster/iceberg_table_checkpoint"
    val bootstrapServers = "node1:9092,node2:9092,node3:9092"
    //多个topic 逗号分开
    val topic = "kafka-iceberg-topic"

    //3.读取Kafka读取数据
    val df = spark.readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", bootstrapServers)
      .option("auto.offset.reset", "latest")
      .option("group.id", "iceberg-kafka")
      .option("subscribe", topic)
      .load()

    import spark.implicits._
    import org.apache.spark.sql.functions._
```

```
    val resDF = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
      .as[(String, String)].toDF("id", "data")

    val transDF: DataFrame = resDF.withColumn("current_day", split(col("data"), "\t")(0))
      .withColumn("ts", split(col("data"), "\t")(1))
      .withColumn("user_id", split(col("data"), "\t")(2))
      .withColumn("page_id", split(col("data"), "\t")(3))
      .withColumn("channel", split(col("data"), "\t")(4))
      .withColumn("action", split(col("data"), "\t")(5))
      .select("current_day", "user_id", "page_id", "channel", "action")

    //结果打印到控制台,Default trigger (runs micro-batch as soon as it can)
//    val query: StreamingQuery = transDF.writeStream
//      .outputMode("append")
//      .format("console")
//      .start()

    //4.流式写入Iceberg表
    val query = transDF.writeStream
      .format("iceberg")
      .outputMode("append")
      //每分钟触发一次Trigger.ProcessingTime(1, TimeUnit.MINUTES)
      //每10s 触发一次 Trigger.ProcessingTime(1, TimeUnit.MINUTES)
      .trigger(Trigger.ProcessingTime(10, TimeUnit.SECONDS))
      .option("path", "hadoop_prod.iceberg_db.iceberg_table")
      .option("fanout-enabled", "true")
      .option("checkpointLocation", checkpointPath)
      .start()
    query.awaitTermination()
  }
}
```

Structured Streaming 向 Iceberg 实时写入数据有以下几个注意点：

- 写 Iceberg 表写出数据支持两种模式：append 和 complete，append 是将每个微批数据行追加到表中。complete 是替换每个微批数据内容。
- 向 Iceberg 中写出数据时指定的 path 可以是 HDFS 路径，可以是 Iceberg 表名，如果是表名，要预先创建好 Iceberg 表。
- 写出参数 fanout-enabled 指的是如果 Iceberg 写出的表是分区表，在向表中写数据之前要求 Spark 每个分区的数据必须排序，但这样会带来数据延迟，为了避免这个延迟，可以设置“fanout-enabled”参数为 true，可以针对每个 Spark 分区打开一个文件，直到当前 task 批次数据写完，这个文件再关闭。
- 实时向 Iceberg 表中写数据时，建议 trigger 设置至少为1分钟提交一次，因为每次提交都会产生一个新的数据文件和元数据文件，这样可以减少一些小文件。为了进一步减少数据文件，建议定期合并“data files”和删除旧的快照。

查看 Iceberg 中数据结果

启动向 Kafka 生产数据代码，启动向 Iceberg 中写入数据的 Structured Streaming 程序，执行以下代码来查看对应的 Iceberg 结果：

```
//1.准备对象
val spark: SparkSession = SparkSession.builder().master("local").appName("StructuredSinkIceberg")
  //指定hadoop catalog，catalog名称为hadoop_prod
  .config("spark.sql.catalog.hadoop_prod", "org.apache.iceberg.spark.SparkCatalog")
  .config("spark.sql.catalog.hadoop_prod.type", "hadoop")
  .config("spark.sql.catalog.hadoop_prod.warehouse", "hdfs://hdfsCluster/structuredstreaming")
  .getOrCreate()

//2.读取Iceberg 表中的数据结果
spark.sql(
  """
    |select * from hadoop_prod.iceberg_db.iceberg_table
  """.stripMargin).show()
```

flink 结合 kafka 实时写到 iceberg 表

使用 HDFS 的一个路径作为 iceberg 的结果表，使用 Flink 实时消费 kafka 中的数据并写入 iceberg 表，并且使用 Hive 作为客户端实时读取。
因为 iceberg 强大的读写分离特性，新写入的数据几乎可以实时读取。

创建 Hadoop Catalog 的 Iceberg 表

```
// create hadoop catalog
tenv.executeSql("CREATE CATALOG hadoop_catalog WITH (\n" +
        "  'type'='iceberg',\n" +
        "  'catalog-type'='hadoop',\n" +
        "  'warehouse'='hdfs://hdfsCluster/tmp',\n" +
        "  'property-version'='1'\n" +
        ")");

// change catalog
tenv.useCatalog("hadoop_catalog");
tenv.executeSql("CREATE DATABASE if not exists iceberg_hadoop_db");
tenv.useDatabase("iceberg_hadoop_db");
// create iceberg result table
```

```
tenv.executeSql("drop table hadoop_catalog.iceberg_hadoop_db.iceberg_002");
tenv.executeSql("CREATE TABLE  hadoop_catalog.iceberg_hadoop_db.iceberg_002 (\n" +
            "    user_id STRING COMMENT 'user_id',\n" +
            "    order_amount DOUBLE COMMENT 'order_amount',\n" +
            "    log_ts STRING\n" +
            ")");
```

使用Hive Catalog创建Kafka流表

```
String HIVE_CATALOG = "myhive";
String DEFAULT_DATABASE = "tmp";
String HIVE_CONF_DIR = "/data/bigdata/tbds/etc/hive/conf/";
Catalog catalog = new HiveCatalog(HIVE_CATALOG, DEFAULT_DATABASE, HIVE_CONF_DIR);
tenv.registerCatalog(HIVE_CATALOG, catalog);
tenv.useCatalog("myhive");
// create kafka stream table
tenv.executeSql("DROP TABLE IF EXISTS ods_k_2_iceberg");
tenv.executeSql("CREATE TABLE ods_k_2_iceberg (\n" +
            " user_id STRING,\n" +
            " order_amount DOUBLE,\n" +
            " log_ts TIMESTAMP(3),\n" +
            " WATERMARK FOR log_ts AS log_ts - INTERVAL '5' SECOND\n" +
            ") WITH (\n" +
            "  'connector'='kafka',\n" +
            "  'topic'='t_kafka_03',\n" +
            "  'scan.startup.mode'='latest-offset',\n" +
            "  'properties.bootstrap.servers'='xx:9092',\n" +
            "  'properties.group.id' = 'testGroup_01',\n" +
            "  'format'='json'\n" +
            ")");
```

使用SQL连接kafka流表和iceberg 目标表

```
System.out.println("---> 3. insert into iceberg  table from kafka stream table .... ");
tenv.executeSql("INSERT INTO  hadoop_catalog.iceberg_hadoop_db.iceberg_002 " +
    " SELECT user_id, order_amount, DATE_FORMAT(log_ts, 'yyyy-MM-dd') FROM myhive.tmp.ods_k_2_i
ceberg");
```

数据验证

```
bin/kafka-console-producer.sh --broker-list xx:9092 --topic t_kafka_03
{"user_id":"a1111","order_amount":11.0,"log_ts":"2020-06-29 12:12:12"}
{"user_id":"a1111","order_amount":11.0,"log_ts":"2020-06-29 12:15:00"}
{"user_id":"a1111","order_amount":11.0,"log_ts":"2020-06-29 12:20:00"}
{"user_id":"a1111","order_amount":11.0,"log_ts":"2020-06-29 12:30:00"}
{"user_id":"a1111","order_amount":13.0,"log_ts":"2020-06-29 12:32:00"}
{"user_id":"a1112","order_amount":15.0,"log_ts":"2020-11-26 12:12:12"}


hive> add jar /data/bigdata/tbds/usr/local/cluster-shim/lite/500/lib/hive/lib/iceberg-hive-runtime-0.14.1.jar;
hive> CREATE EXTERNAL TABLE tmp.iceberg_002(user_id STRING,order_amount DOUBLE,log_ts STRING)
STORED BY 'org.apache.iceberg.mr.hive.HiveIcebergStorageHandler'
LOCATION '/tmp/iceberg_hadoop_db/iceberg_002';
hive> select * from tmp.iceberg_002  limit 5;
a1111    11.0    2020-06-29
a1111    11.0    2020-06-29
a1111    11.0    2020-06-29
a1111    11.0    2020-06-29
a1111    13.0    2020-06-29
Time taken: 0.108 seconds, Fetched: 5 row(s)
```

使用Flink Table API 消费kafka并实时写入基于HDFS Hadoop Catalog的iceberg 结果表中

trino查看数据历史版本以及回滚到某个历史版本

```
# 启动trino客户端（先进入镜像,PRESTO服务PRESTO_COORDINATOR节点）：
docker ps -a
docker exec -it presto-server bash
/root/trino-cli-360-executable.jar --server localhost:18089 --catalog iceberg --user hive --debug


#建库
create schema if not exists iceberg.iceberg_db_check;
#建表
DROP TABLE if exists iceberg.iceberg_db_check.tb_trino_check1;
CREATE TABLE iceberg.iceberg_db_check.tb_trino_check1 (
   c1 integer,
   c2 varchar,
   c3 varchar)
WITH (
   format = 'PARQUET');
```

```
#分批次插入数据
insert into iceberg.iceberg_db_check.tb_trino_check1 values (1,'1','1'),(2,'2','2');
insert into iceberg.iceberg_db_check.tb_trino_check1 values (3,'3','3'),(4,'4','4');
insert into iceberg.iceberg_db_check.tb_trino_check1 values (5,'5','5'),(6,'6','6');

#查询历史版本commit的一些记录
SELECT * FROM iceberg.iceberg_db_check."tb_trino_check1$snapshots" ORDER BY committed_at DESC;

#选取上面查询的历史中的某个commit的 snapshot_id 回滚 eg:snapshot_id=4304372289675937535
CALL iceberg.system.rollback_to_snapshot('iceberg_db_check', 'tb_trino_check1', 430437228967593753
5);

#在查询，就是对应历史版本的数据了
select * from iceberg.iceberg_db_check.tb_trino_check1;
```

关键功能

Apache Iceberg 具备以下能力：
● 模式演进（Schema evolution）：支持 Add（添加）、Drop（删除）、Update（更新）、Rename（重命名）和 Reorder（重排）表格式定义。
● 分区布局演进（Partition layout evolution）：可以随着数据量或查询模式的变化而更新表的布局。
● 隐式分区（Hidden partitioning）：查询不再取决于表的物理布局。通过物理和逻辑之间的分隔，Iceberg 表可以随着数据量的变化和时间的推移发展分区方案。错误配置的表可以得到修复，无需进行昂贵的迁移。
● 时光穿梭（Time travel）：支持用户使用完全相同的快照进行重复查询，或者使用户轻松检查更改。
备注：支持在 DataFrame 中使用 Time Travel 能力。
● 版本回滚（Version rollback）：使用户可以通过将表重置为良好状态来快速纠正问题。
在可靠性与性能方面，Iceberg 可在生产中应用到数十 PB 的数据表，即使没有分布式 SQL 引擎，也可以读取这些巨大规模的表：
● 扫描速度快，无需使用分布式 SQL 引擎即可读取表或查找文件。
● 高级过滤，基于表元数据，使用分区和列级统计信息对数据文件进行裁剪。
Iceberg 被设计用来解决最终一致的云对象存储中的正确性问题：
● 可与任何云存储一起使用，并且通过避免调用 list 和 rename 来减少 HDFS 的 NameNode 拥塞。
● 可序列化的隔离，表更改是原子性的，用户永远不会看到部分更改或未提交的更改。
● 多个并发写入使用乐观锁机制进行并发控制，即使写入冲突，也会重试以确保兼容更新成功。
Iceberg 设计为以快照（Snapshot）的形式来管理表的各个历史版本数据。快照代表一张表在某个时刻的状态。每个快照中会列出表在某个时刻的所有数据文件列表。Data 文件存储在不同的 Manifest 文件中，Manifest 文件存储在一个 Manifest List 文件中，Manifest 文件可以在不同的 Manifest List 文件间共享，一个 Manifest List 文件代表一个快照。
● Manifest List 文件是元数据文件，其中存储的是 Manifest 文件的列表，每个 Manifest 文件占据一行。
● Manifest 文件是元数据文件，其中列出了组成某个快照的数据文件列表。每行都是每个数据文件的详细描述，包括数据文件的状态、文件路径、分区信息、列级别的统计信息（例如每列的最大最小值、空值数等）、文件的大小以及文件中数据的行数等信息。
● Data 文件是 Iceberg 表真实存储数据的文件，一般是在表的数据存储目录的 data 目录下。

5.3.1.3使用 Iceberg 1.6.1 版本，以下为该版本的功能受限内容：

- Avro 格式文件不支持默认值填充。
- 支持 timestamp、timestamptz，但不支持 timestamp_ns、timestamptz_ns。
- 不支持 spark 3.4.4；不支持 spark 3.5.2 及以后版本。

常见问题

Spark Sql 设置 IDENTIFIER FIELDS

在使用 Spark SQL 设置 IDENTIFIER FIELDS ，被设置的字段必须是 NOT NULL 的。

```
// 建表
CREATE TABLE sample (
  id bigint NOT NULL,
  data string NOT NULL,
  category string,
  ts timestamp)
USING iceberg;

// 使用 SET IDENTIFIER FIELDS 语法
// 只有 id 和 data 才能被设置成为 IDENTIFIER FIELDS 因为它们是 NOT NULL
alter table sample  SET IDENTIFIER FIELDS  id;
alter table sample  SET IDENTIFIER FIELDS  data;
```

设置 engine.hive.enabled

任何非 hive 引擎创建表后，如果需要该表对 hive 可见，需要使用 spark 设置表属性 engine.hive.enabled'='true' 。

源表-目标表 schema 非空字段对齐

在使用 Spark SQL 对源表和目标表之间进行 insert into... insert overwrite... merge into... 操作时，要注意源表和目标表的非空字段是对齐的。如果源表的字段允许非空，而目标表的字段不允许非空，那么 spark 将会校验不通过。在源表是 hive，目标表是 iceberg 时尤其需要注意。

spark-shell 操作 Iceberg API

iceberg 提供了 Java API，可以直接操作 iceberg 表。其核心接口为：org.apache.iceberg.Table (Table)。在 spark-shell 环境中使用可以参考以下脚本：

```
import org.apache.iceberg.relocated.com.google.common.collect.Maps;
import org.apache.iceberg.hive.HiveCatalog;
```

```
import org.apache.iceberg.catalog.TableIdentifier
import org.apache.iceberg.Table

var catalog = new HiveCatalog();
catalog.initialize("hive_prod", Maps.newHashMap());
catalog.setConf(spark.sparkContext.hadoopConfiguration);

var name = TableIdentifier.of("iceberg_db", "iceberg_table");
var table = catalog.loadTable(name);
```

主键约束

flink 建表中使用的主键，只对 flink 引擎生效。其他的引擎依然可以使用相同的主键插入数据，形成相同主键的多条数据。在这样的场景下，flink 依然可以正常查询，但是在做 upsert 操作时，会删除其他主键相同的数据，只留下一条记录。

spark-iceberg timestamp timezone 不一致

iceberg 支持不带 timezone 的 timestamp 但是 spark 不支持不带 timezone 的 timestamp。所以如果其他引擎创建的 iceberg 表中带有 timestamp without timezone 字段，那么使用 spark 读会存在潜在的时区转换问题。在 spark 引擎下需要增加相关配置来规避：

```
# Controls whether reading/writing timestamps without timezones is allowed
set spark.sql.iceberg.handle-timestamp-without-timezone=true

# Controls whether timestamp types for new tables should be stored with timezone info
set spark.sql.iceberg.use-timestamp-without-timezone-in-new-tables=true
# 以上用法不推荐，建议在建表时就对齐字段类型！
```

Trino查询Iceberg表速度慢

trino查询iceberg耗时14秒，分析是由于数据源比较小(37MB)，扫描数据和预汇聚都是单并发，可以通过调整参数set session prefer_partial_aggregation=false;取消预汇聚，后续步骤可以并发提升性能，和配置测试最新耗时在5-6秒。

ElasticSearch开发
概述

Elasticsearch是一个基于Apache Lucene的开源搜索引擎。Elasticsearch 可用于搜索任何类型的文档。它提供可扩展的搜索，具有近乎实时的搜索，并支持多租户。

以下是Elasticsearch的一些关键特点：

1. 分布式和扩展性：Elasticsearch自动处理数据分片和复制，以支持大规模数据集和高并发请求。
2. 全文搜索：Elasticsearch基于Lucene，提供了强大的全文搜索能力，支持复杂的查询和过滤。
3. 实时分析：Elasticsearch可以在大规模数据上进行实时的聚合分析，支持各种类型的统计和分析需求。
4. 易用性：Elasticsearch提供了简单的RESTful API，支持各种语言的客户端库，使得开发者可以很容易地使用它。
5. 集成：Elasticsearch是Elastic Stack（也被称为ELK Stack）的一部分，与Logstash和Kibana等工具紧密集成，提供了日志收集、搜索和可视化的完整解决方案。

以下是一些Elasticsearch的关键概念：

1. 文档：在Elasticsearch中，文档是可被索引的基本单位，通常以JSON格式表示。
2. 索引：索引是一种存储文档的容器，类似于传统数据库中的“数据库”，每个索引都有一个名字，可以用来对文档进行读写。
3. 节点：一个节点就是运行着Elasticsearch的一个服务器实例。
4. 集群：一个集群是由一个或多个节点组成，它们共同存储数据，并提供联合的索引和搜索能力。
5. 分片：Elasticsearch将每个索引分成多个片段，这些片段可以分布在集群的多个节点上，以支持大规模数据和高并发请求。
6. 复制：Elasticsearch自动复制分片，以防止数据丢失，并提高查询的并发能力。
7. 映射：映射是定义文档和它们字段如何存储和索引的过程。每个索引都有一个或多个映射，这些映射类似于传统数据库中的“表结构”。

ES 用户体系介绍

简要介绍

ES的用户体系包括如下两个方面：

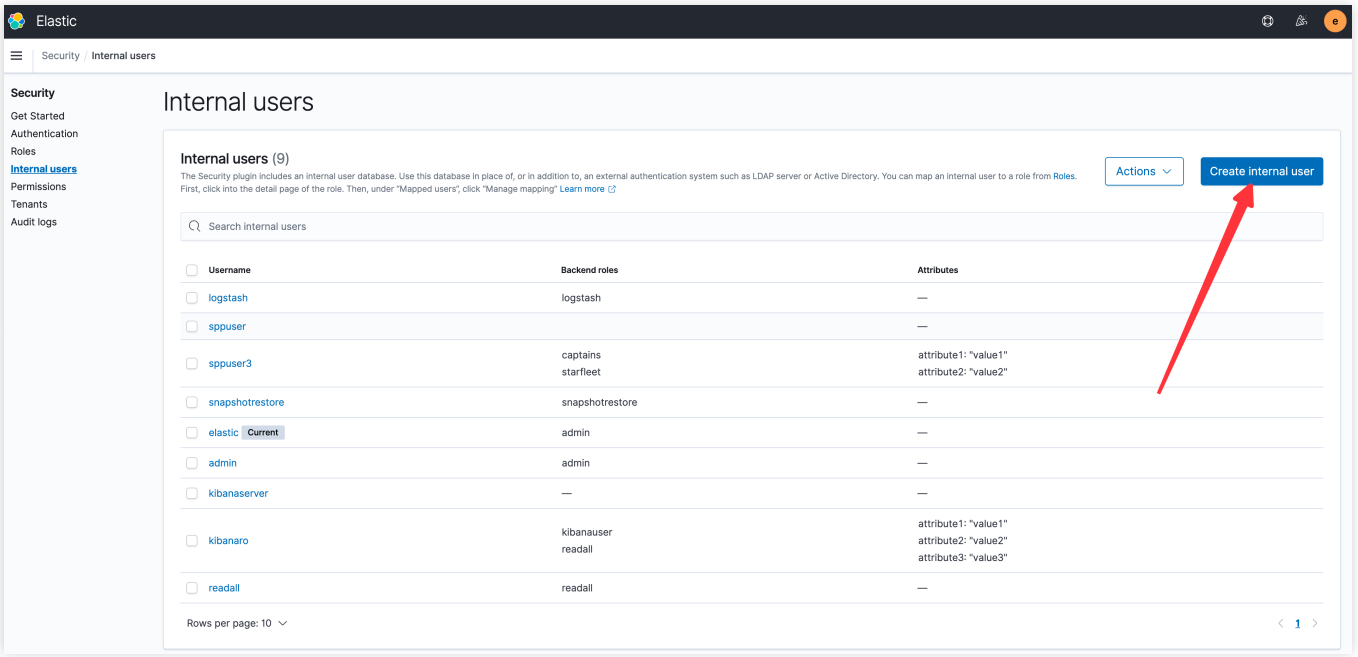
- TBDS平台用户：TBDS平台用户，通过 TBDS 平台用户体系管理，面向集群运维管控、平台常用操作，同时 TBDS 会默认将该用户名同步到 ES 内核（默认同步过来的用户均无权限，需要二次授权）。
- ES 内核用户：ES 内核 security 体系来管理，可以通过TBDS平台创建并同步过来，也可以通过调用 ES API创建（Kibana Web UI亦可）
  在 ES 工程开发过程中，两种用户都可以使用，如下介绍用户的创建、授权方式。

创建用户

- TBDS 平台用户创建
  参见 添加用户
  安全认证同时支持 Kerberos 认证和 Simple ，如果用户在安装 ES 集群时未开启 Kerberos 认证，则默认使用Simple认证方式。在这种情况下,用户只需要使用创建的用户名和密码即可，无需keytab等凭证。更多参考：身份认证和授权

  > 说明：
  >
  > Kibana Web UI 的登录需要经过认证代理：
  >
  > 1、代理页面的登录：root/{默认为公共集群密码}【代理页面访问用户名为root，初始密码为创建公共集群时设置的密码】
  >
  > 2、Kibana 页面的登录：elastic/Elasticsearch@123【ES 集群的内核用户，默认 admin 权限。可以用来调用 REST 接口、登录 Kibana 页面等】

- ES 内核用户创建
  方式一：通过API创建，参考：API
  方式二：通过Kibana页面创建



用户授权

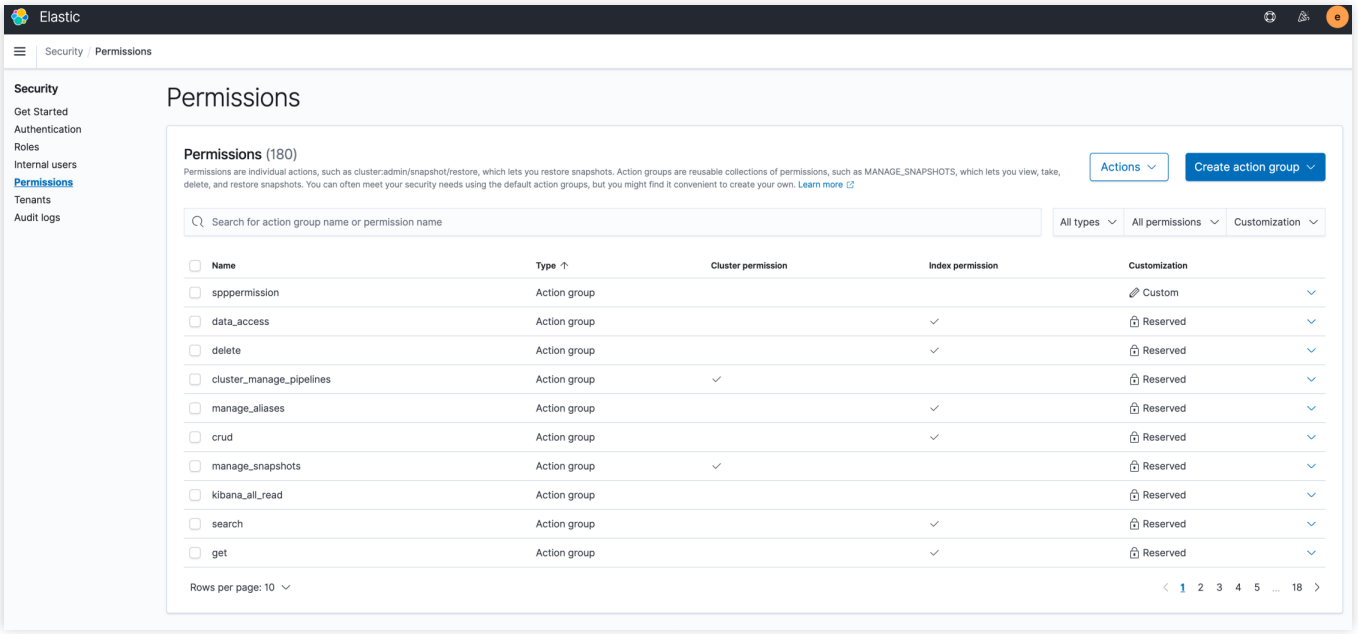
- TBDS 平台用户授权

  i. 平台用户，可以先为其授权平台管理范畴的权限/角色，参见 为用户授权

  ii. 创建角色并授权，这一步可以调用 API 接口，也可以在 Kibana Web UI 页面操作。

- ES 内核用户授权

  创建好的 ES 内核用户无法用于登录 TBDS 管控平台，主要面向 ES 开发场景，授权方式支持调用 API 接口，也可以在 Kibana Web UI 页面操作。


Elastic
Security / Permissions
Security
Get Started
Authentication
Roles
Internal users
Permissions
Tenants
Audit logs
Permissions
Permissions (180)
Permissions are individual actions, such as cluster:admin/snapshot/restore, which lets you restore snapshots. Action groups are reusable collections of permissions, such as MANAGE_SNAPSHOTS, which lets you view, take, delete, and restore snapshots. You can often meet your security needs using the default action groups, but you might find it convenient to create your own. Learn more
Actions
Create action group
Search for action group name or permission name
All types
All permissions
Customization
Name
Type
Cluster permission
Index permission
Customization
spppermission Action group Custom
data_access Action group Reserved
delete Action group Reserved
cluster_manage_pipelines Action group Reserved
manage_aliases Action group Reserved
crud Action group Reserved
manage_snapshots Action group Reserved
kibana_all_read Action group Reserved
search Action group Reserved
get Action group Reserved
Rows per page: 10
1 2 3 4 5 ... 18

示例工程开发

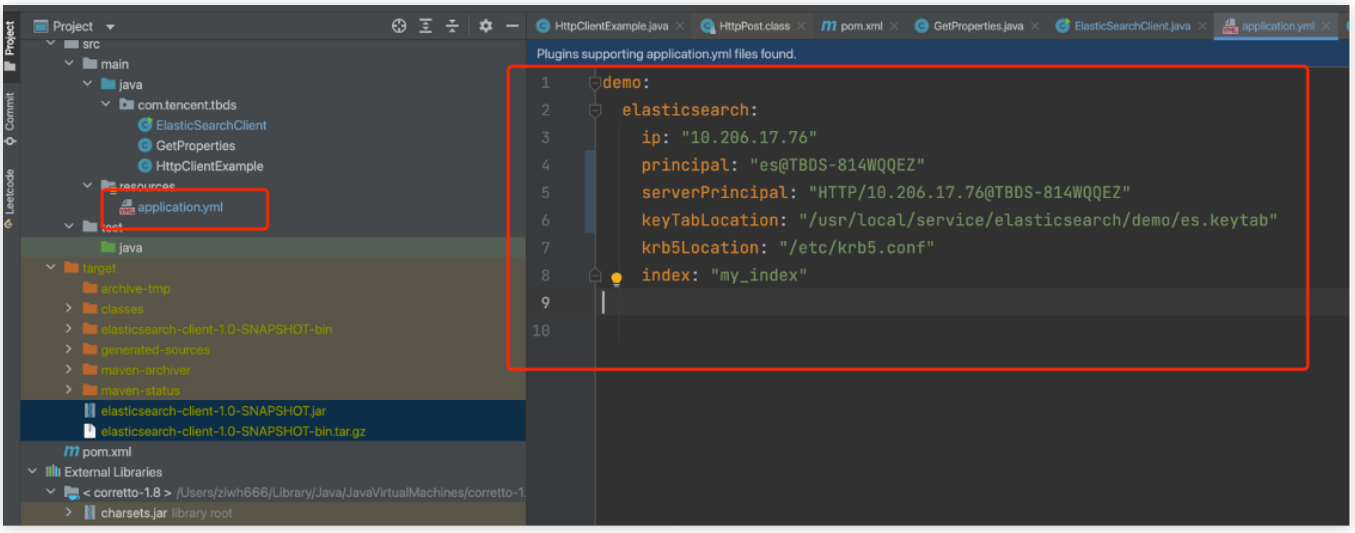
环境准备

1. 开发环境准备

准备项	说明
安装JDK	JDK8，推荐使用KonaJDK，下载地址
安装和配置IDE	按需选择，比如IntelliJ IDEA或Eclipse
安装Maven	开发环境基础配置，负责构建Java应用程序
Maven配置准备	如果需要本地调试，需要参考 开发环境准备 配置Maven pom.xml，推荐Maven 3.6.3，下载地址

2. 运行环境准备
   以下使用Linux环境作为开发机进行应用调试说明。

- 准备开发机（可选）：建议使用Linux操作系统，
- 部署客户端：参考在开发机上执行客户端部署，Elasticsearch开发涉及的配置在resources下的application.yml中：



keyTabLocation 需要管理员提供的keytab文件，需要上传到jar包所在环境，然后修改配置文件中对应的路径

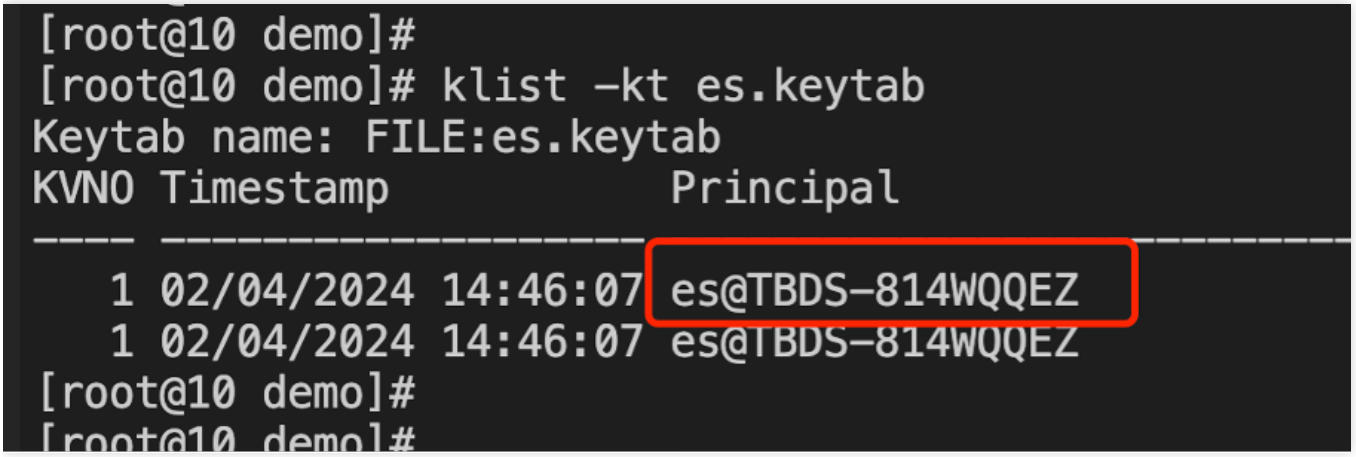
principal 是指对应keytab文件对应的kerberos用户, 可以通过klist -kt {keytab_file}查看

serverPrincipal 是指ES所在节点与kerberos通信的serverPrincipal 可通过 cat /usr/local/service/elasticsearch/config/elasticsearch.yml | grep principal 查看到

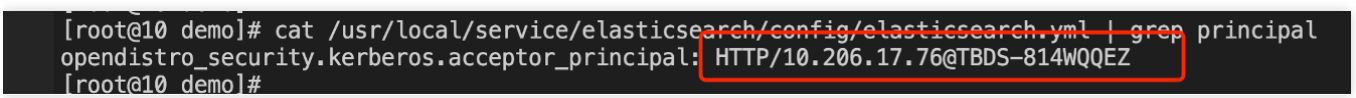
krb5Location 默认是/etc/krb5.conf, 一般不用调整

ip是指ES节点的ip, 一般将jar包上传到哪个节点就配置哪个节点的ip

上传了keytab之后, principal如何查看: klist -kt es.keytab



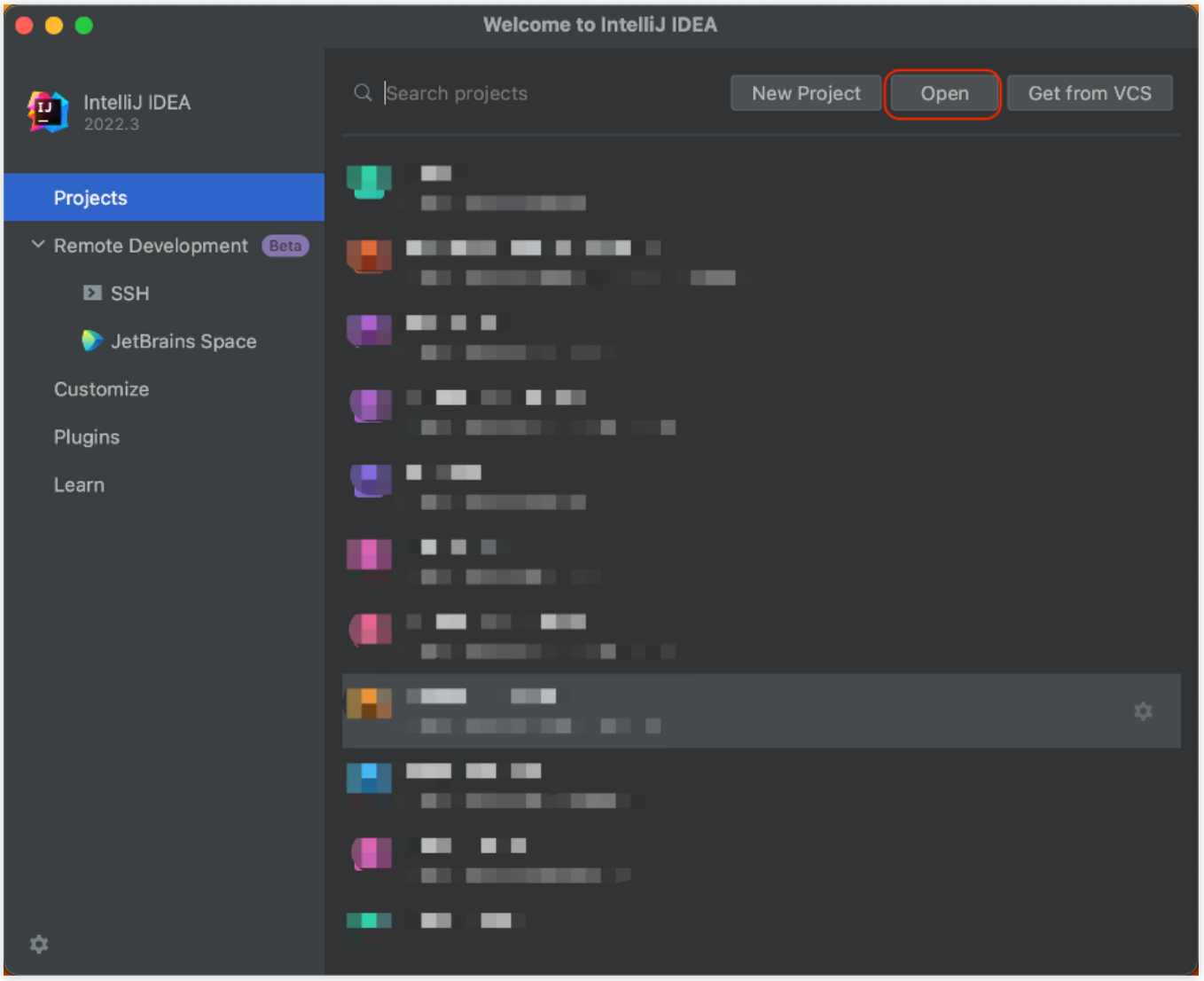
serverPrincipal如何查看: cat /usr/local/service/elasticsearch/config/elasticsearch.yml | grep principal



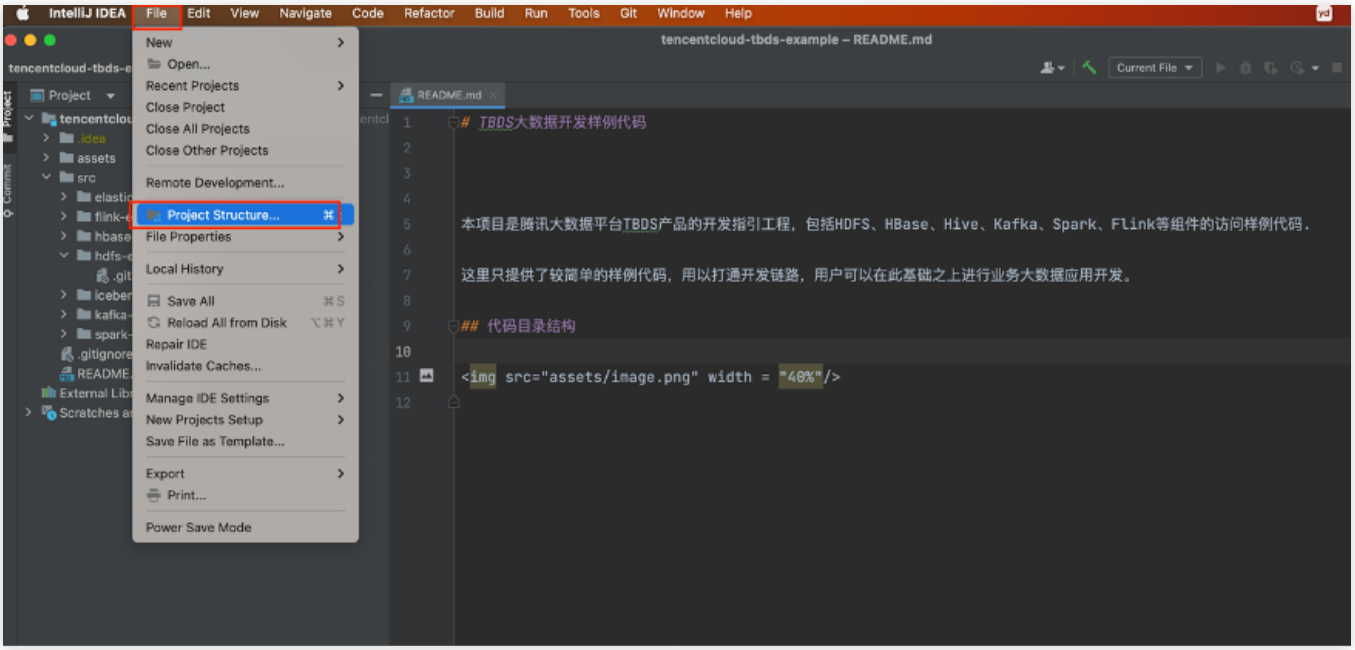
3. 导入示例工程代码

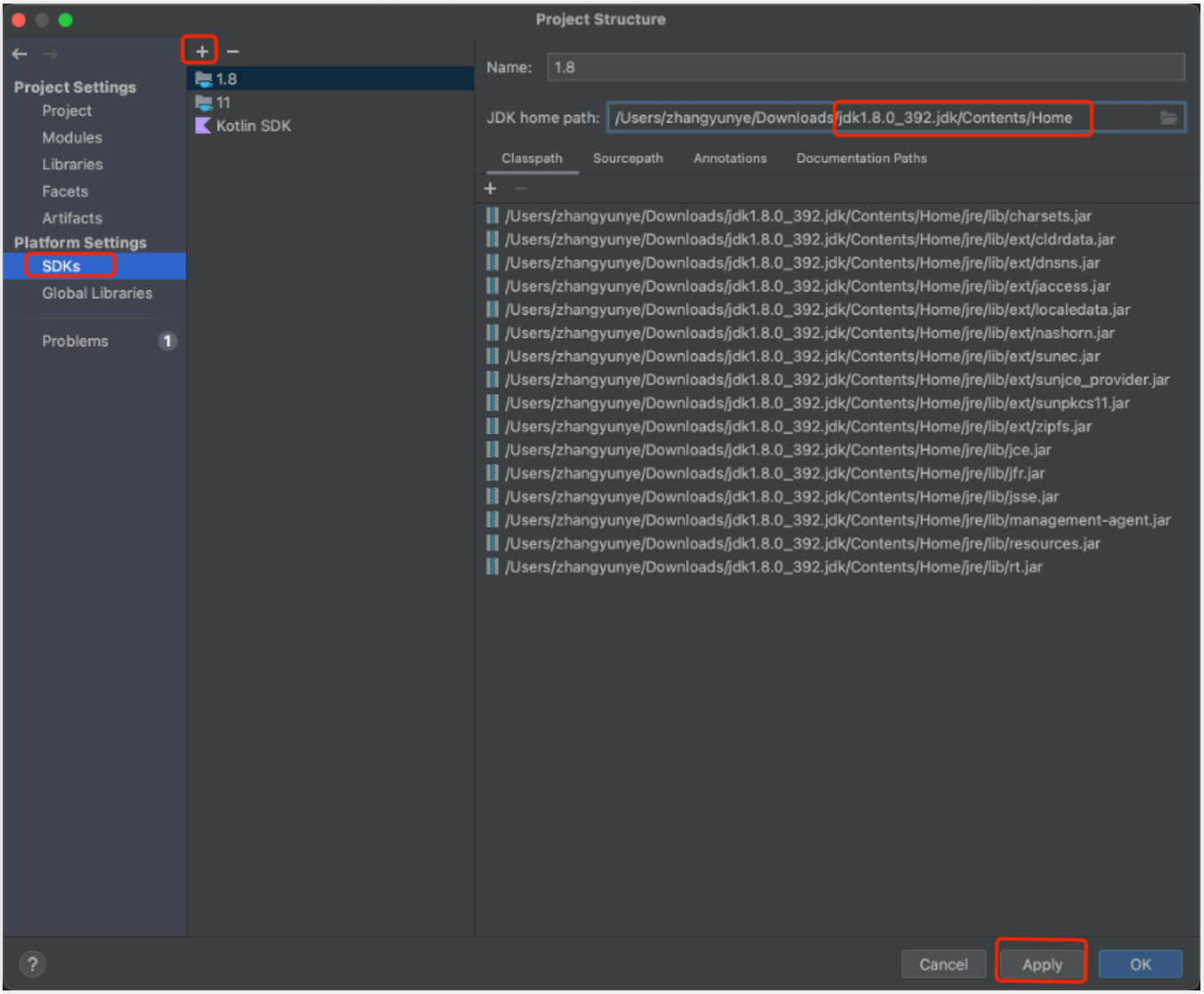
以下以IntelliJ IDEA举例，将ElasticSearch示例工程代码导入进行说明。

- 在GitHub获取样例代码：仓库地址：https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master
- 导入样例工程到IntelliJ IDEA开发环境

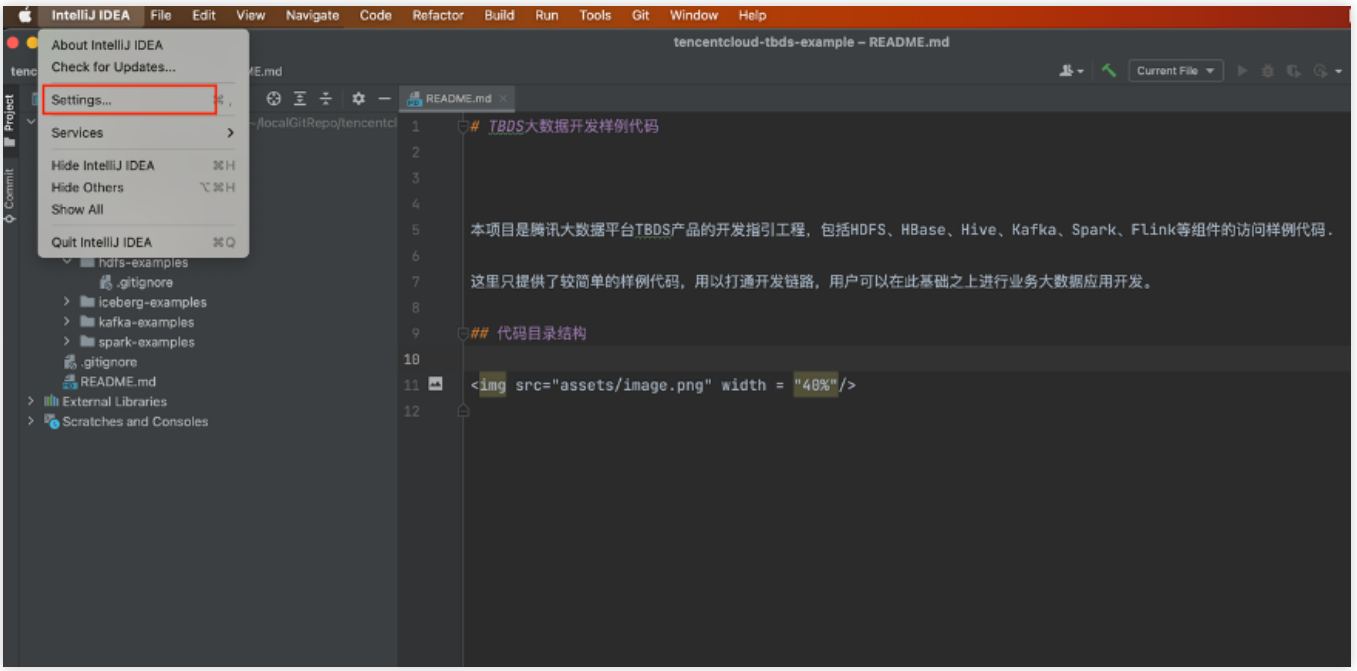


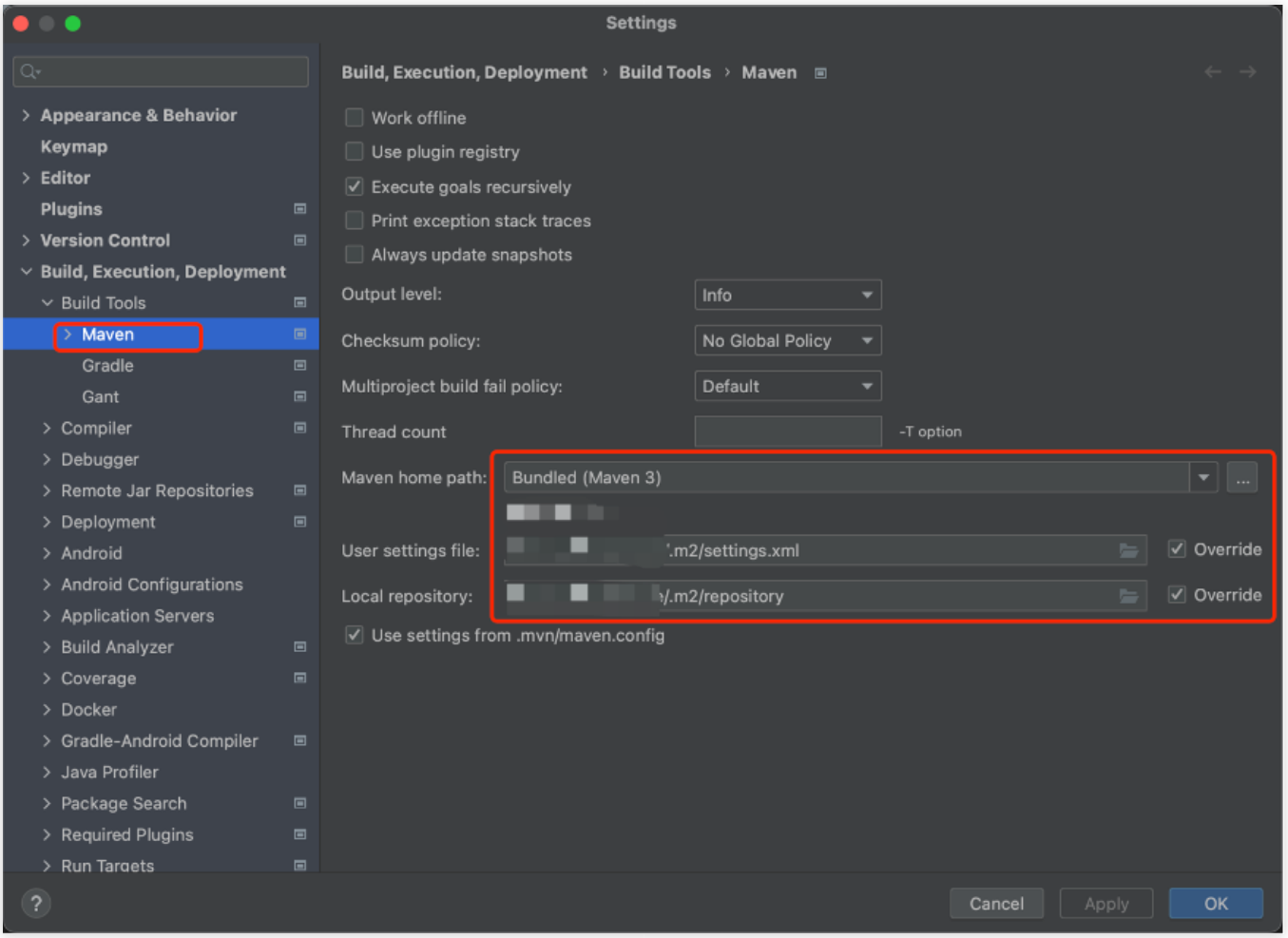
- 安装完IntelliJ IDEA和JDK工具后,需要在IntelliJ IDEA中配置JDK。选择 File 下的 Project Structure，点击 SDKs，选择 JDK 1.8，点击 Apply，再点击 OK。若没有 JDK 1.8，则点击 + 号进行添加，点击 Add JDK 后选择 JDK 1.8 安装目录，然后点击 OK 即可。


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
New
Open...
Recent Projects
Close Project
Close All Projects
Close Other Projects
Remote Development...
Project Structure...
File Properties
Local History
Save All
Reload All from Disk
Repair IDE
Invalidate Caches...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
tencentcloud-tbds-example – README.md
README.md
TBDS大数据开发样例代码
本项目是腾讯大数据平台TBDS产品的开发指引工程，包括HDFS、HBase、Hive、Kafka、Spark、Flink等组件的访问样例代码。
这里只提供了较简单的样例代码，用以打通开发链路，用户可以在此基础之上进行业务大数据应用开发。
代码目录结构


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
1.8
11
Kotlin SDK
Name: 1.8
JDK home path: /Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home
Classpath
Sourcepath
Annotations
Documentation Paths
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/charsets.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/ext/cldrdata.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/ext/dnsns.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/ext/jaccess.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/ext/localedata.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/ext/nashorn.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/ext/sunec.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/ext/sunjce_provider.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/ext/sunpkcs11.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/ext/zipfs.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/jce.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/jfr.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/jsse.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/management-agent.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/resources.jar
/Users/zhangyunye/Downloads/jdk1.8.0_392.jdk/Contents/Home/jre/lib/rt.jar
Cancel
Apply
OK

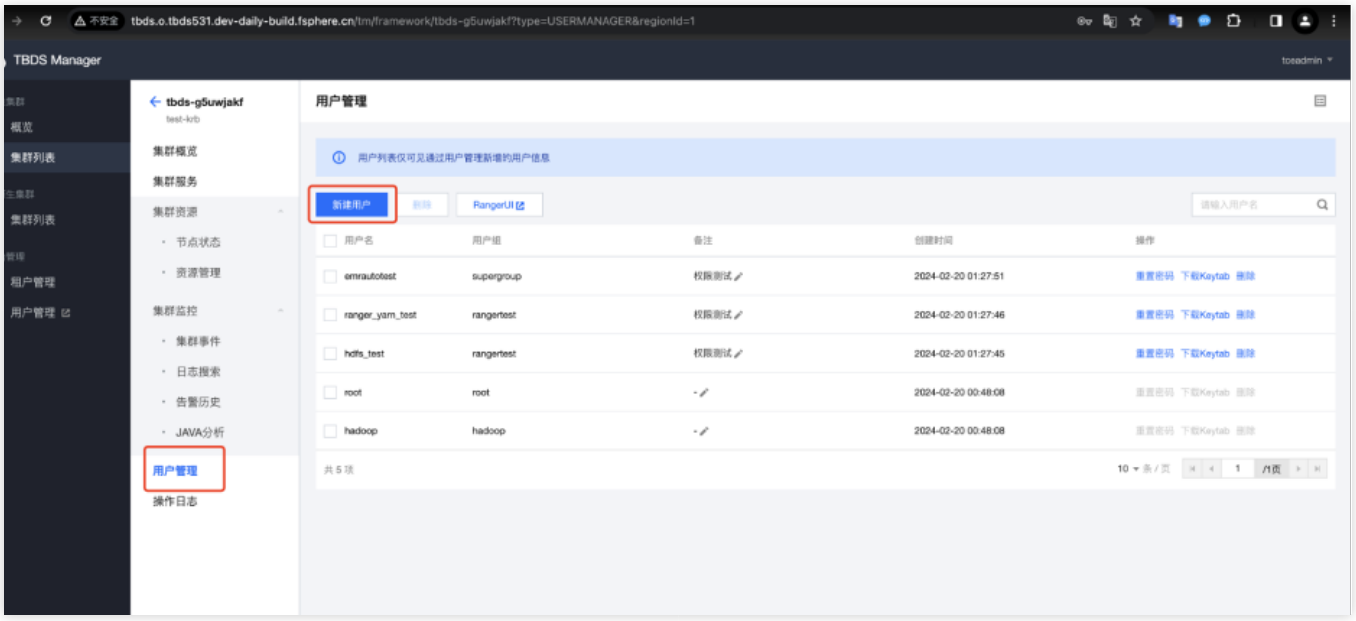
- 将工程依赖的jar包添加到类路径。需确保在本地安装了 Maven，并配置好了环境变量和 settings.xml 文件。IDEA 点击 Settings 进入配置页面，左上角输入 Maven 进行搜索，点击 Build Tools 下的 Maven 配置项，修改 “Maven home path” 为本地 Maven 的安装目录，修改 “User settings file” 为本地 Maven settings.xml 配置文件的文件路径，并勾选 Override，此时 “Local repository” 将自动设置为settings.xml 文件中配置的本地 Maven 仓库的目录。最后点击 Apply，再点击 OK。





用户权限配置

- 到安全集群页面创建用户(管理员需要做的)
  这块ES支持LDAP认证和Kerberos认证, ldap认证直接在请求时加上用户和密码, kerberos认证可参考此处的用例.



- 参考https://opendistro.github.io/for-elasticsearch-docs/docs/security/access-control/api/#roles文档在Kibana页面或者es节点后台用hadoop用户 为kerberos用户赋予相应的权限 (管理员需要做的). 下面给出了一个用例（注意将${password},${ip:port}替换为实际密码和ip:port）：

```
# 用hadoop用户创建一个es_role角色, 给该角色分配拥有哪些索引的权限
curl -u 'hadoop:${password}' -X PUT "${ip:port}/_opendistro/_security/api/roles/es_role" -H 'Content-Type: application/json' -d'
{
  "cluster_permissions": [
    "cluster_composite_ops",
    "indices_monitor"
  ],
  "index_permissions": [{
    "index_patterns": [
      "*"
    ],
    "dls": "",
    "fls": [],
    "masked_fields": [],
    "allowed_actions": [
      "read"
    ]
  }],
  "tenant_permissions": [{
    "tenant_patterns": [
      "human_resources"
    ],
    "allowed_actions": [
      "kibana_all_read"
    ]
```

```
  }]
}'
# 然后将该es_role角色的权限分配给es用户
curl -u 'hadoop:${ip:port}' -X PUT "${ip:port}/_opendistro/_security/api/rolesmapping/es_role" -H 'Content-Type: application/json' -d'
{
  "backend_roles" : [],
  "hosts" : [],
  "users" : ["es"]
}'

#
```

之后就可以用该用户和密码或
kerberos
认证来进行请求了

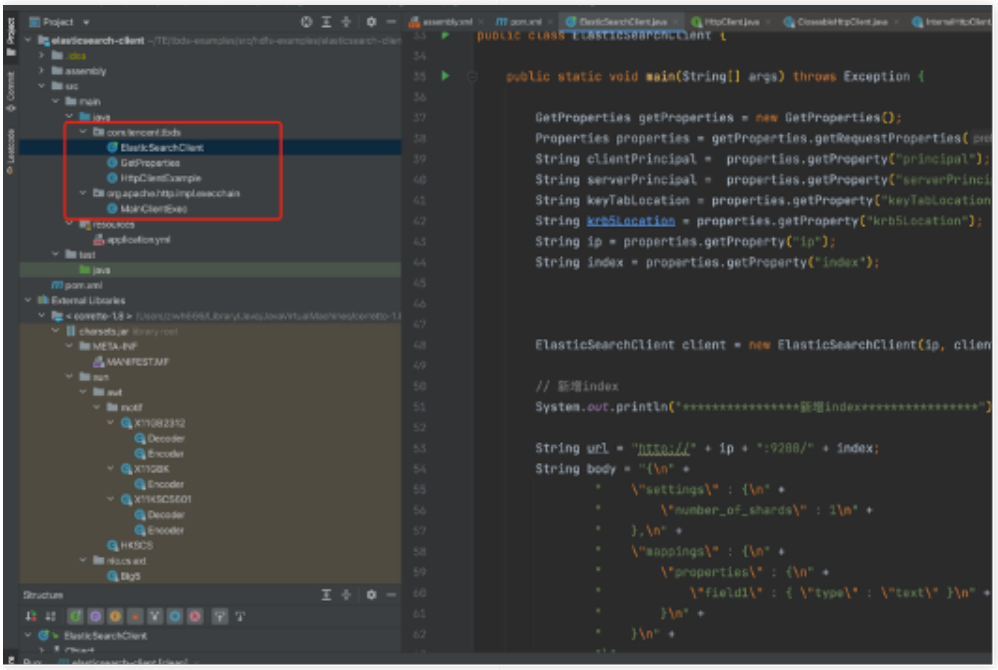
代码逻辑说明

功能说明

Elasticsearch demo的主要逻辑如下：

- 加载application.yml配置文件的内容。
- 进行Kerberos认证, 拿到认证的subject。
- 通过subject调用封装好的http请求, 拿到返回的body数据

代码逻辑说明



代码主要包含ElasticsearchClient.java(进行kerberos认证主函数), GetProperties.java(获取配置), HttpClientExample.java(构建http请求), org/apache/http/impl/execchain/MainClientExec.java(为了适配ES的kerberos认证, 重写的HttpClient jar包中的MainClientExec.java方法, 打包时会用该类替换httpclientjar包中的MainClientExec.java类)

```
/**
 * 获取初始化配置
 */
public class GetProperties {
 private Properties properties;
 private Logger LOG = LoggerFactory.getLogger(GetProperties.class);
 public Properties getRequestProperties(String prefix) throws Exception {
     YamlPropertiesFactoryBean yaml = new YamlPropertiesFactoryBean();
     yaml.setResources(new FileSystemResource("application.yml"));
     Properties properties = yaml.getObject();
     Properties requestProperties = new Properties();
     for (Map.Entry<Object, Object> entry : properties.entrySet()) {
         String propKey = entry.getKey().toString();
         if (!propKey.startsWith(prefix)) {
             continue;
         }
         requestProperties.put(StringUtils.substringAfter(propKey, prefix), entry.getValue());
     }
     LOG.info("finish get request info of properties={}", requestProperties);
     return requestProperties;
 }
}
/**
 *Http请求实现Kerberos安全认证
```

```
 */
public class ElasticSearchClient {


private final String ip;
    private final String principal;
    private final String keyTabLocation;
    private final String serverPrincipal;

    public ElasticSearchClient(String ip, String principal, String keyTabLocation, String serverPrincipal, String kr
b5Location, boolean isDebug) {
        this.ip = ip;
        this.principal = principal;
        this.keyTabLocation = keyTabLocation;
        this.serverPrincipal = serverPrincipal;
        System.setProperty("java.security.krb5.conf", krb5Location);
        System.setProperty("javax.security.auth.useSubjectCredsOnly", "false");
        if (isDebug) {
            System.setProperty("sun.security.spnego.debug", "true");
            System.setProperty("sun.security.krb5.debug", "true");
        }
    }

    private static HttpClient buildSpengoHttpClient() {
        HttpClientBuilder builder = HttpClientBuilder.create();
        Lookup<AuthSchemeProvider> authSchemeRegistry = RegistryBuilder.<AuthSchemeProvider>create().
                register(AuthSchemes.SPNEGO, new SPNegoSchemeFactory(true)).build();
        builder.setDefaultAuthSchemeRegistry(authSchemeRegistry);
        BasicCredentialsProvider credentialsProvider = new BasicCredentialsProvider();
        credentialsProvider.setCredentials(new AuthScope(null, -1, null), new Credentials() {
            @Override
            public Principal getUserPrincipal() {
                return null;
            }
            @Override
            public String getPassword() {
                return null;
            }
        });
        builder.setDefaultCredentialsProvider(credentialsProvider);
        CloseableHttpClient httpClient = builder.build();
        return httpClient;
    }

    public HttpResponse request(HttpUriRequest request) {
//        System.out.println(String.format("Calling KerberosHttpClient %s %s %s", this.principal, this.keyTabLoca
tion, request.toString()));
        Configuration config = new Configuration() {
            @SuppressWarnings("serial")
```

```
        @Override
        public AppConfigurationEntry[] getAppConfigurationEntry(String name) {
            return new AppConfigurationEntry[]{new AppConfigurationEntry("com.sun.security.auth.module.Krb5LoginModule",
                    AppConfigurationEntry.LoginModuleControlFlag.REQUIRED, new HashMap<String, Object>() {
                {
                    //Krb5 in GSS API needs to be refreshed so it does not throw the error
                    //Specified version of key is not available
                    put("useTicketCache", "false");
                    put("useKeyTab", "true");
                    put("keyTab", keyTabLocation);
                    put("refreshKrb5Config", "true");
                    put("principal", principal);
                    put("storeKey", "true");
                    put("doNotPrompt", "true");
                    put("isInitiator", "true");
                }
            })};
        }
    };
    Set<Principal> princ = new HashSet<Principal>(1);
    princ.add(new KerberosPrincipal(serverPrincipal));
    Subject sub = new Subject(false, princ, new HashSet<Object>(), new HashSet<Object>());
    try {
        //指定认证模块为Krb5Login
        LoginContext lc = new LoginContext("Krb5Login", sub, null, config);
        //请求kdc进行client身份认证，如果能通过的话，则可以从TGT获取ticket作为后面二次访问时Authorization: Negotiate的基础
        lc.login();
        //无报错则表示client身份认证通过，此时可以在Subject对象中看到已经获取了kerberos的ticket
        Subject serviceSubject = lc.getSubject();
        return Subject.doAs(sub, new PrivilegedAction<HttpResponse>() {
            HttpResponse httpResponse = null;
            @Override
            public HttpResponse run() {
                try {
//                    HttpUriRequest request = new HttpGet(url);
                    //根据刚刚获取的kerberos的ticket构建Spengo请求，会经历一次401后再带上Negotiate二次请求
                    HttpClient spnegoHttpClient = buildSpengoHttpClient();
                    //返回的为二次响应后的结果
                    httpResponse = spnegoHttpClient.execute(request);
                    return httpResponse;
                } catch (IOException ioe) {
                    ioe.printStackTrace();
                }
                return httpResponse;
            }
        });
```

```
        } catch (Exception le) {
            le.printStackTrace();
        }
        return null;
    }

} /**
    *封装请求类
    */
public class HttpClientExample {

    private String url;
    private String json;
    private String method;

    public HttpClientExample(String url, String method, String json) {
        this.url = url;
        this.json = json;
        this.method = method;
    }

    public HttpClientExample(String url, String method) {
        this.url = url;
        this.method = method;
    }


    @Override
    public String toString() {
        return "HttpClientExample{" +
                "url='" + url + '\'' +
                ", json='" + json + '\'' +
                ", method='" + method + '\'' +
                '}';
    }

    public HttpUriRequest execute() throws Exception {
        switch (method.toUpperCase()) {
            case "GET":
                return sendGet();
            case "POST":
                return sendPost();
            case "PUT":
                return sendPut();
            case "DELETE":
                return sendDelete();
        }
        return null;
    }
```

```
    public HttpUriRequest sendGet() throws Exception {
        return new HttpGet(url);
    }

    public HttpUriRequest sendPost() throws Exception {
        HttpPost request = new HttpPost(url);
        StringEntity entity = new StringEntity(json);
        request.addHeader("content-type", "application/json");
        request.setEntity(entity);
        return request;
    }

    public HttpUriRequest sendPut() throws Exception {
        HttpPut request = new HttpPut(url);
        StringEntity entity = new StringEntity(json);
        request.addHeader("content-type", "application/json");
        request.setEntity(entity);
        return request;
    }

    public HttpUriRequest sendDelete() throws Exception {
        return new HttpDelete(url);
    }
}

   /**
    *创建用例
    */
   public static void main(String[] args) throws Exception {

        GetProperties getProperties = new GetProperties();
        Properties properties = getProperties.getRequestProperties("demo.elasticsearch.");
        String clientPrincipal =  properties.getProperty("principal");
        String serverPrincipal =  properties.getProperty("serverPrincipal");
        String keyTabLocation = properties.getProperty("keyTabLocation");
        String krb5Location = properties.getProperty("krb5Location");
        String ip = properties.getProperty("ip");
        String index = properties.getProperty("index");


  ElasticSearchClient client = new ElasticSearchClient(ip, clientPrincipal, keyTabLocation, serverPrincipal, krb5L
ocation, false);

        // 新增index
        System.out.println("***************新增index***************");

        String url = "http://" + ip + ":${port}/" + index;
        String body = "{\n" +
                "    \"settings\" : {\n" +
```

```
            "       \"number_of_shards\" : 1\n" +
            "    },\n" +
            "    \"mappings\" : {\n" +
            "       \"properties\" : {\n" +
            "          \"field1\" : { \"type\" : \"text\" }\n" +
            "       }\n" +
            "    }\n" +
            "}";
        HttpClientExample httpClient = new HttpClientExample(url, "PUT", body);
        HttpUriRequest request = httpClient.execute();
        HttpResponse response = client.request(request);
        System.out.println(httpResponseToString(response));

        // 查询index
        System.out.println("****************查询index****************");

        url = "http://" + ip + ":${port}/" + index;
        httpClient = new HttpClientExample(url, "GET");
        request = httpClient.execute();
        response = client.request(request);
        System.out.println(httpResponseToString(response));


        // 删除index
        System.out.println("****************删除index****************");
        url = "http://" + ip + ":${port}/" + index;
        httpClient = new HttpClientExample(url, "DELETE");
        request = httpClient.execute();
        response = client.request(request);
        System.out.println(httpResponseToString(response));

    }
```

代码调试

以下使用IntelliJ IDEA说明示例工程代码调试过程。

- 编译代码

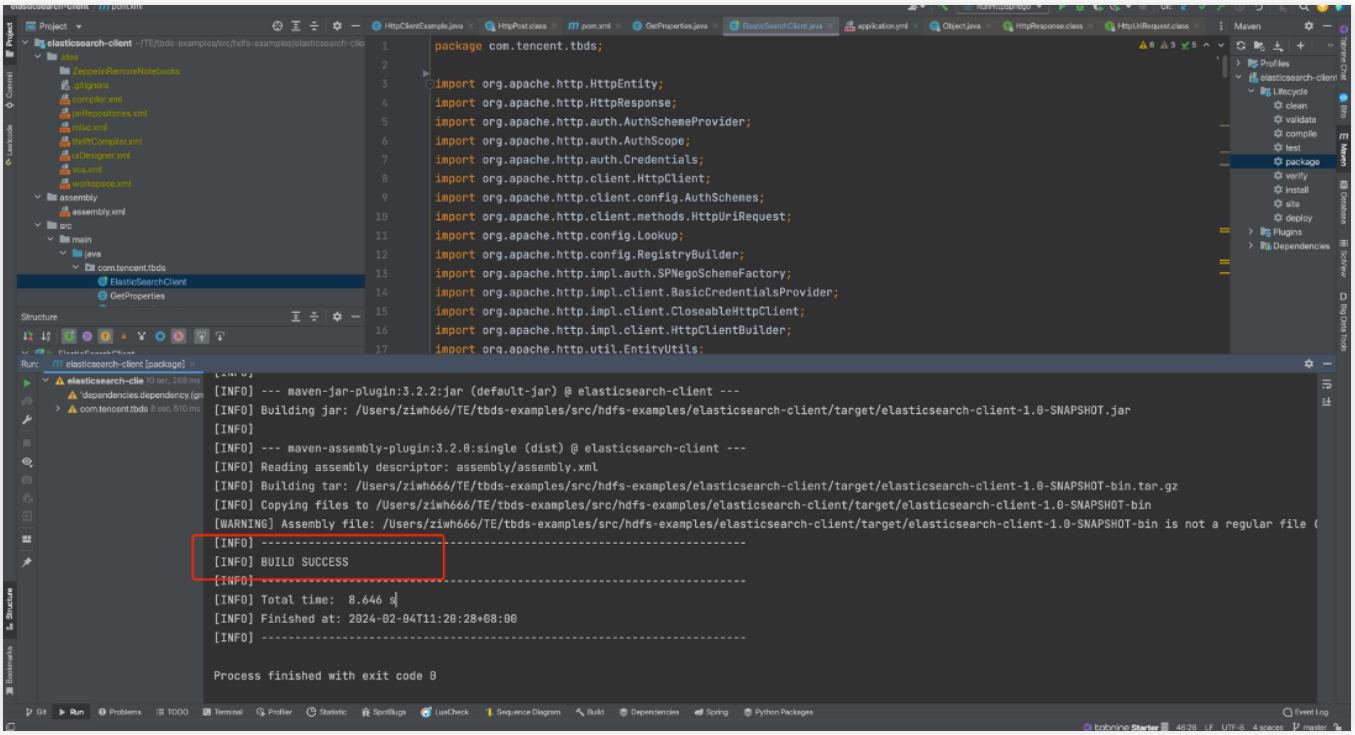
  打开IntelliJ IDEA终端，进入elasticsearch-examples目录

  cd src/elasticsearch-examples/

  使用maven命令打包

  mvn install clean package

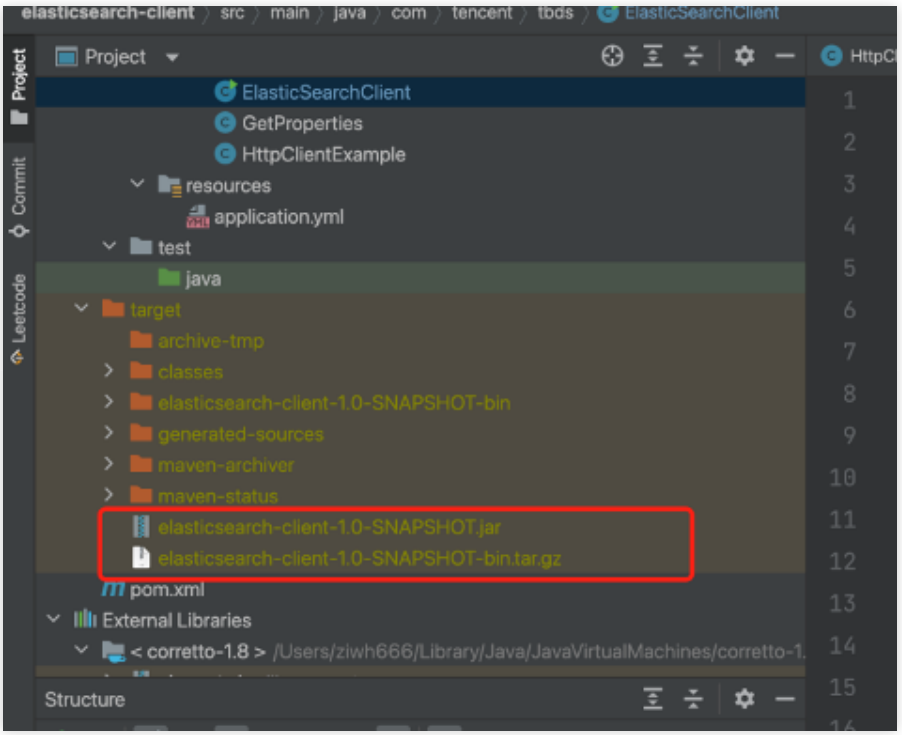
  编译成功界面如下：



编译过程中遇到报错需要根据错误信息进行排查和修改，打印“BUILD SUCCESS”为编译成功。

- 开发机调试

  编译成功后在target目录下得到elasticsearch-client-1.0-SNAPSHOT.jar和elasticsearch-client-1.0-SNAPSHOT-bin.tar.gz文件，上传到开发机新建的/data/demo目录进行调试运行。



发布

通过上述编译后得到elasticsearch-client-1.0-SNAPSHOT.jar和elasticsearch-client-1.0-SNAPSHOT-bin.tar.gz。其elasticsearch-client-1.0-SNAPSHOT-bin.tar.gz包含运行过程中依赖的目录和文件；elasticsearch-client-1.0-SNAPSHOT.jar为elasticsearch demo的java执行文件。将两个文件上传到开发机，并解压tar包。
将keytab文件也放到对应位置, 修改application.yml的内容。然后运行相应的jar包
命令如下：

```
#开发机中新建目录
mkdir -p /data/demo
#本地上传压缩文件和jar包
scp elasticsearch-client-1.0-SNAPSHOT.jar elasticsearch-client-1.0-SNAPSHOT-bin.tar.gz root@xx:/data/demo
#开发机上解压压缩文件
cd /data/demo
tar zxf elasticsearch-client-1.0-SNAPSHOT-bin.tar.gz
mv conf/application.yml ./
```

- 运行样例：在开发机上进入/data/demo目录，执行命令运行jar文件，更多参数说明参考 https://www.elastic.co/guide/en/elasticsearch/reference/7.10/rest-apis.html。该demo演示了对Elasticsearch JAVA API的基本实现，包括：创建索引, 查询索引, 删除索引等功能。后续要调用其他接口, 将对应的URL, method, body参数传入即可。
  java -jar elasticsearch-client-1.0-SNAPSHOT.jar

Logstash 开发说明

简要介绍

TBDS-ES 平台同时内置了开源版 Logstash ，用户可以基于 TBDS 平台进行安装部署、运维、使用。它是一个服务器端的数据处理管道，支持动态的从不同来源采集和转换数据，并将数据标准化到目标位置。Logstash 常和 Elasticsearch 配合，通过输入、过滤和输出插件，加工和转换任何类型的事件，将数据加载到 Elasticsearch。
Logstash 主要分为 input、filter、output 三种工作方式：

- 数据输入：支持多样的数据来源，通过输入插件方便的采集日志、指标、Web 应用、数据库、消息队列、传感器等来源数据。
- 数据过滤：通过过滤插件清理和转换数据，如将非结构化数据解析导出结构、解析 IP 地址、标准化日期、通过编解码器简化常见格式等。
- 数据输出：通过输出插件将数据传输到需要的地方，例如 Elasticsearch、数据库等，以便对数据做进一步的分析和处理。
  更多参考：Logstash Reference [7.10]

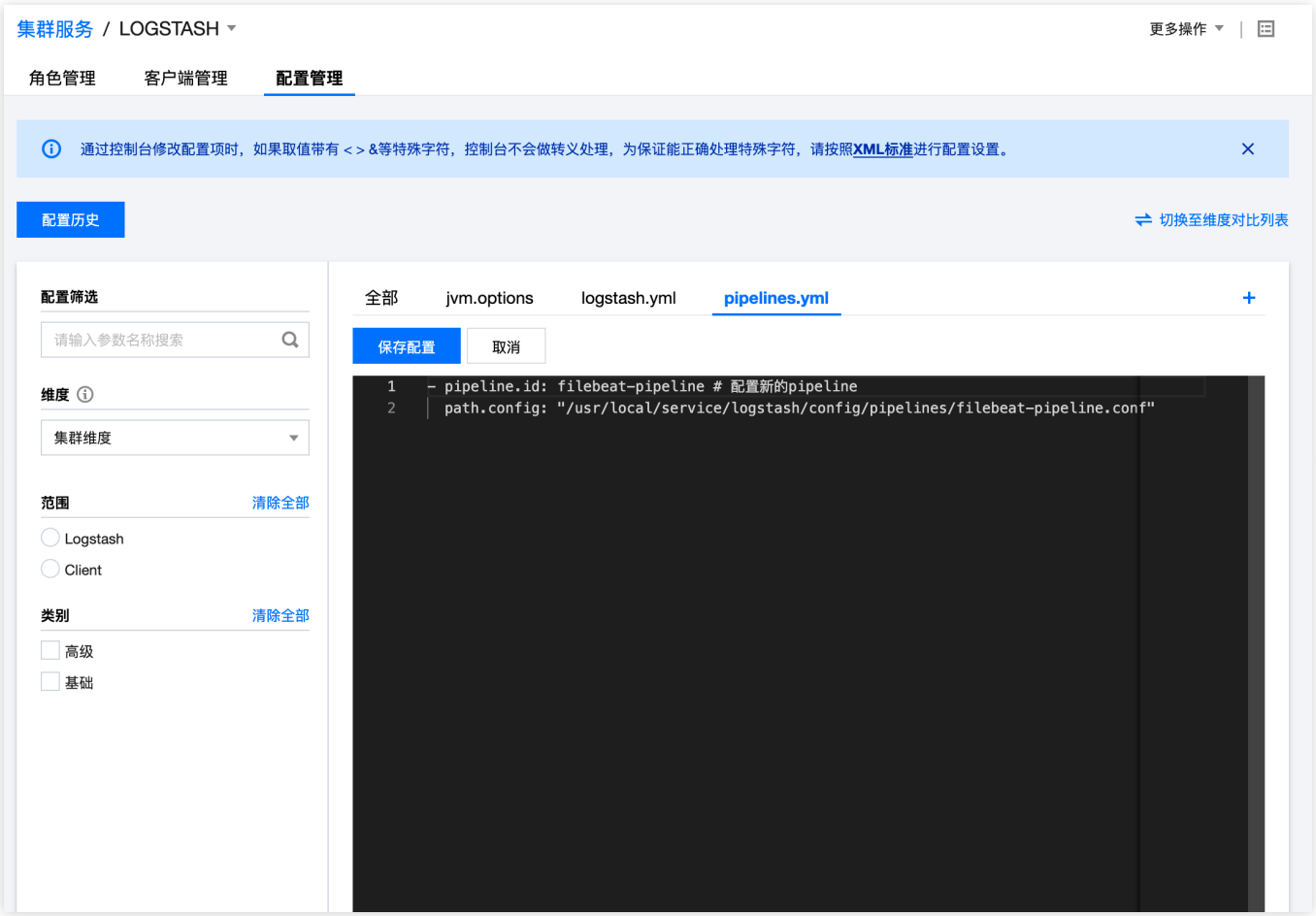
开发使用说明

Logstash 的部署、配置说明：

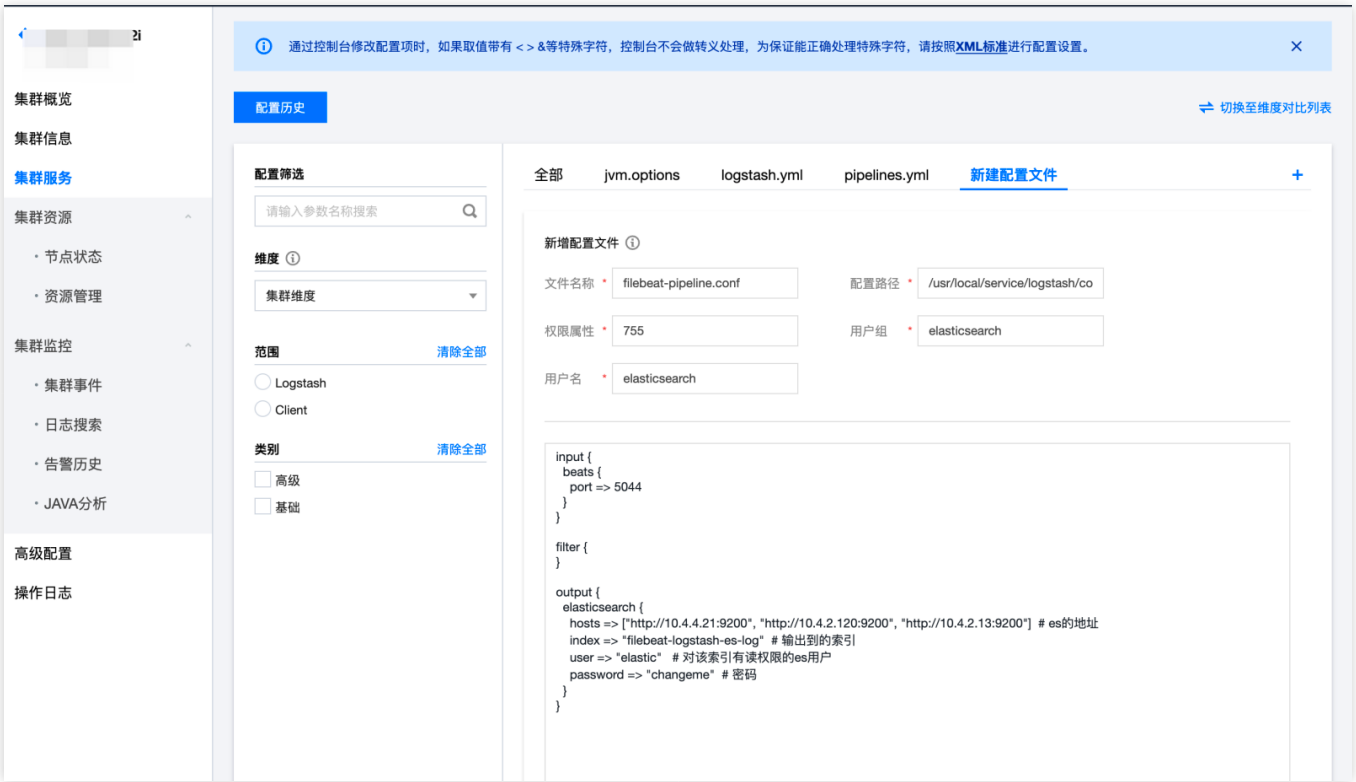
- 支持和 ES 节点混合部署，也支持独立节点部署，用户在集群的安装&扩容时可以进行选择。
- 实例配置可以在 TBDS 平台进行配置，包括 JVM、YAML文件配置。
- Logstash 通过管道来实现数据的采集处理，它包含必选的 input 和 output 插件，以及可选的 filter 插件，并支持多管道并行运行。用户可以通过 TBDS 平台进行Logstash 管道文件的配置管理，包括创建管道、修改管道、和删除管道以及版本管理。

操作如下：

1. 配置pipelines.yml 文件，指向具体的路径文件。



2. 利用 TBDS 平台的“新增配置文件”功能，将上一步具体指向的文件进行填写，系统会下发到服务器生效。



ELK 链路 Demo 可参考：Beats 下载

Kibana 开发说明

简要介绍

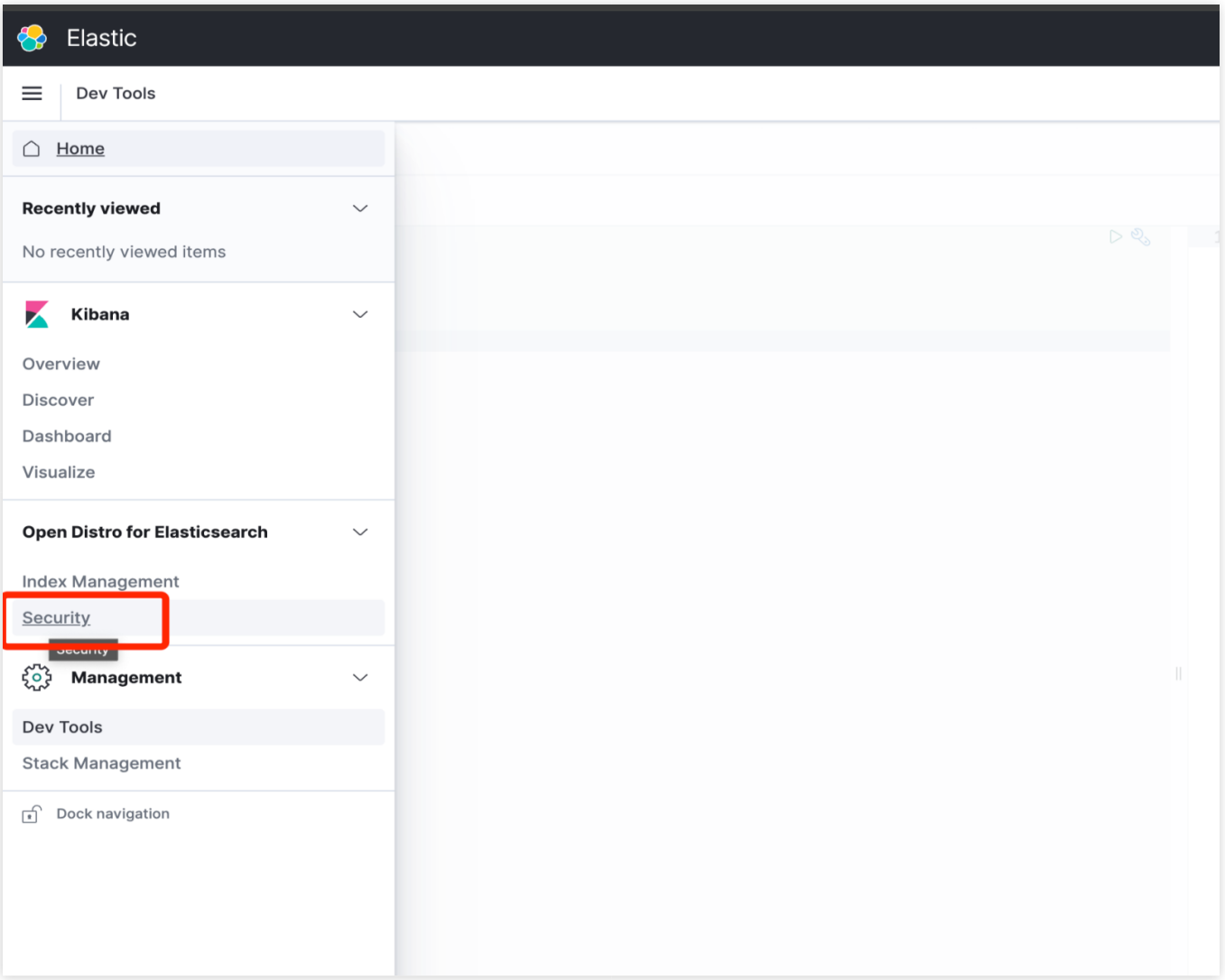
TBDS-ES 平台为用户提供开源版的 Kibana，包括常用的 Discover、Dashboard、Dev Tools等，不包括 X-Pack 相关能力。
为了更方便辅助用户在 ELK 生态开发过程中，能够更好的使用 Kibana 的可视化能力，TBDS 平台为 Kibana 集成了Open Distro for ES 的 Index、Security 等相关插件。
更多参考：Kibana

Kibana 的登录和使用

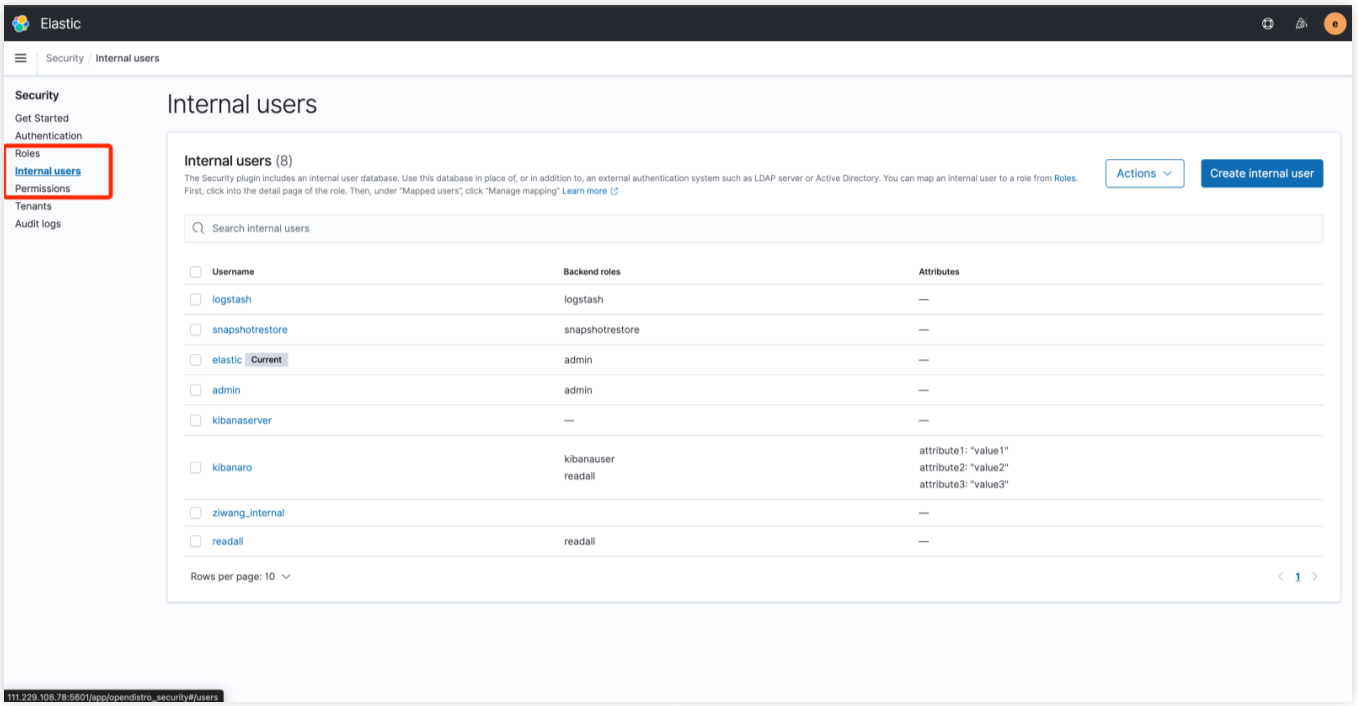
Kibana 的登录

在 Kibana 上创建用户和授权

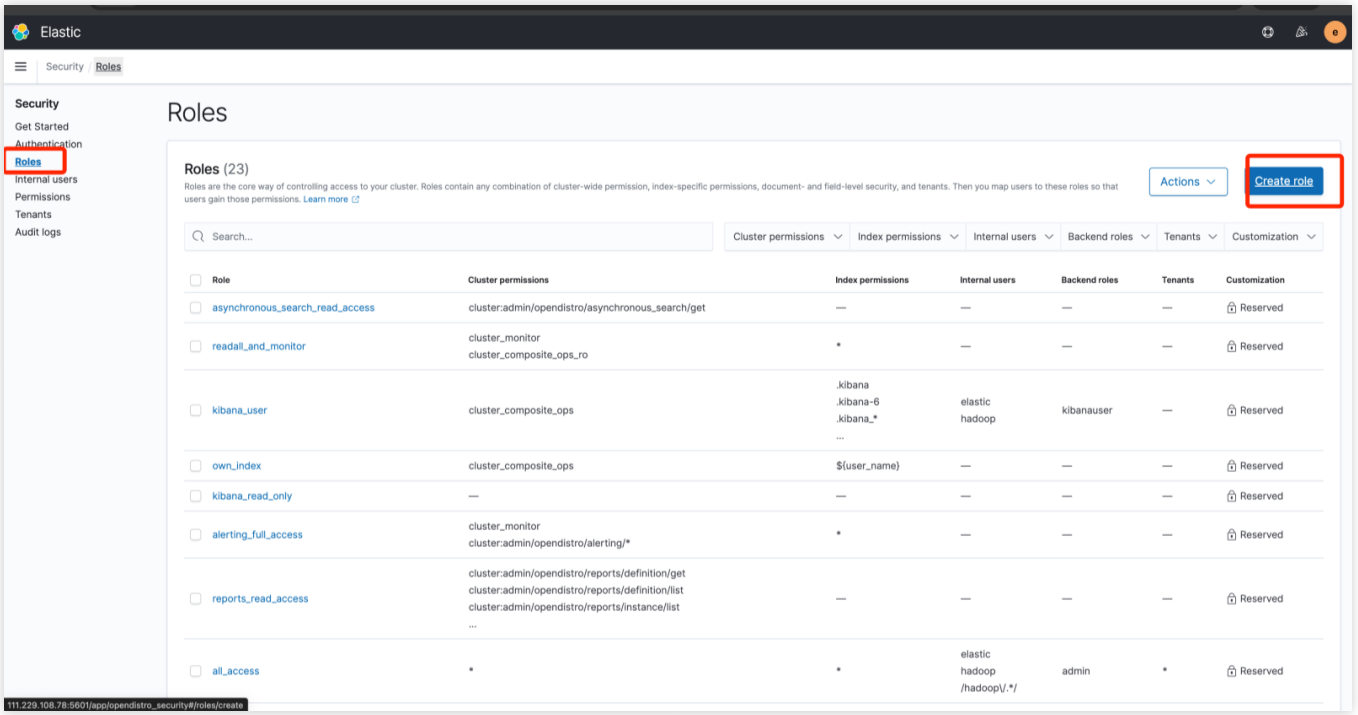
1. 用超管用户登录 Kibana Web UI，单击导航栏“Security”。

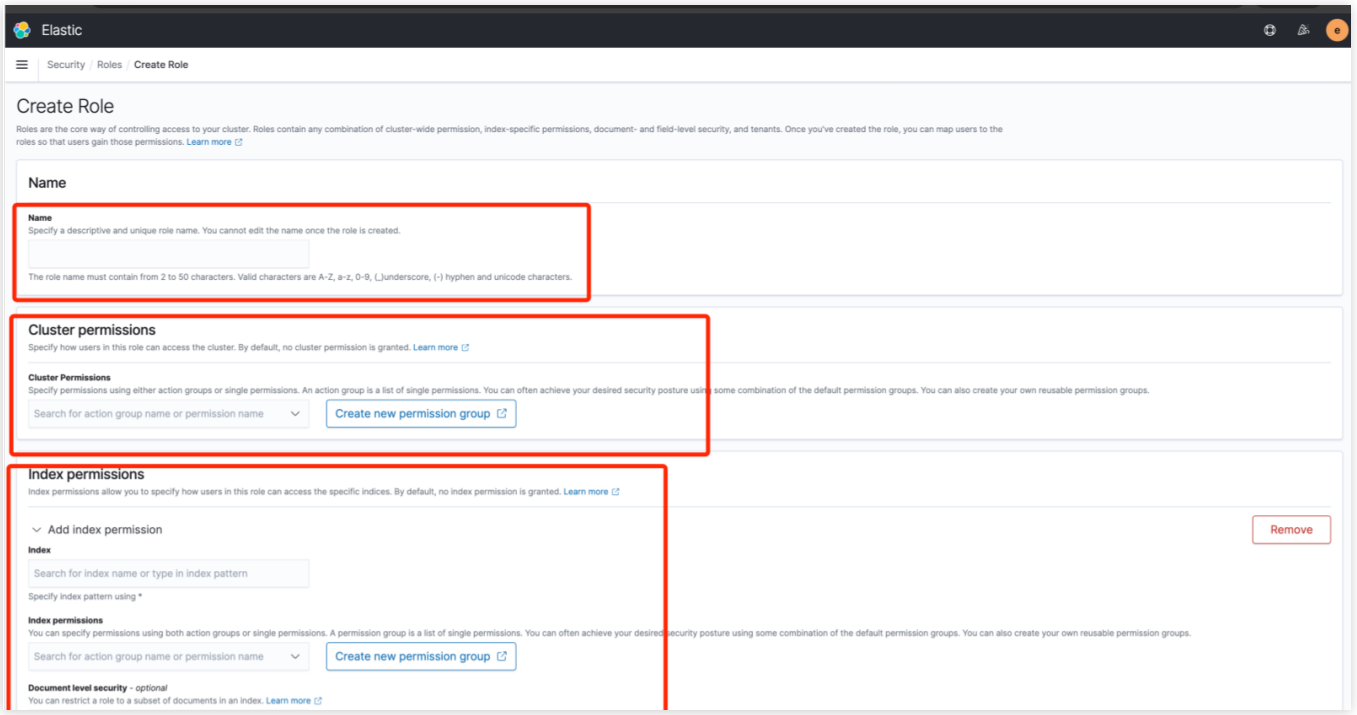


2. 单击Internal Users可以创建新的用户，并且可以对用户的权限进行配置, 单击 Roles 可以对角色的权限进行配置，单击Permissions可以对权限进行配置。

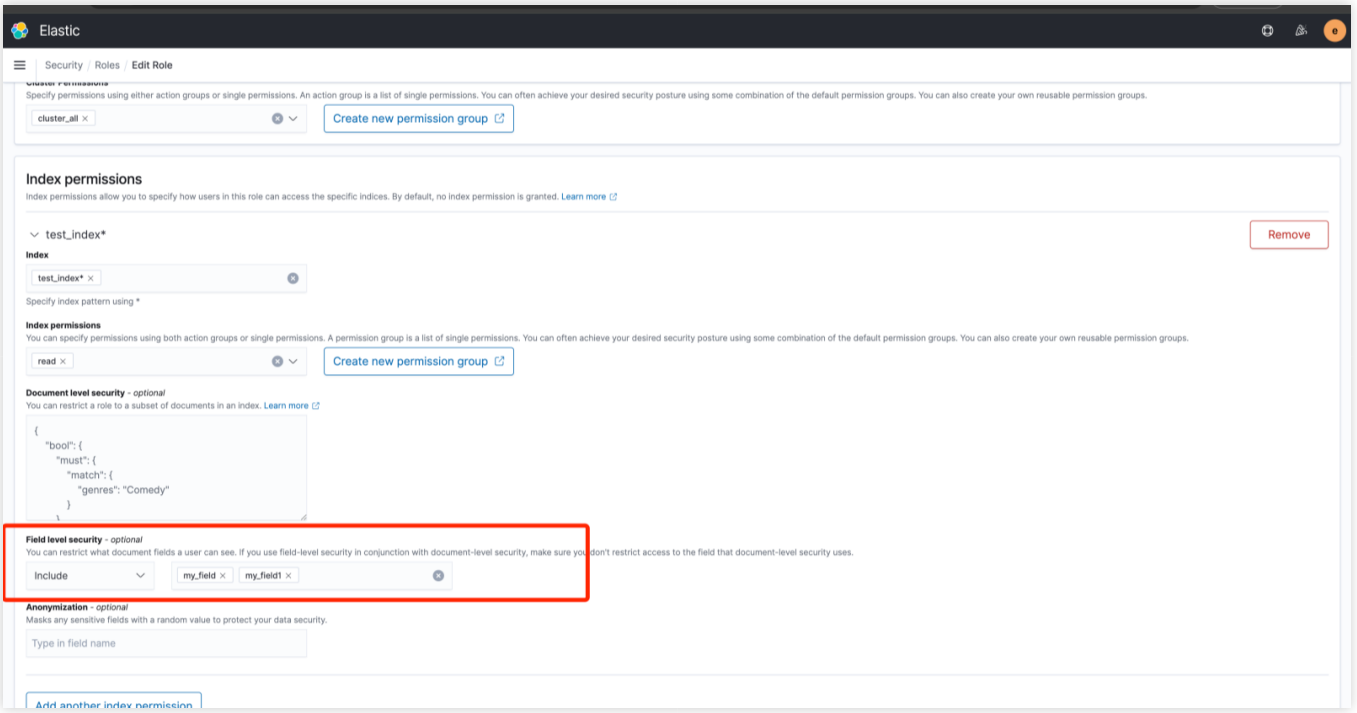


3. 配置角色，分别为角色创建对应的名称, 分配集群权限, 分配索引权限。



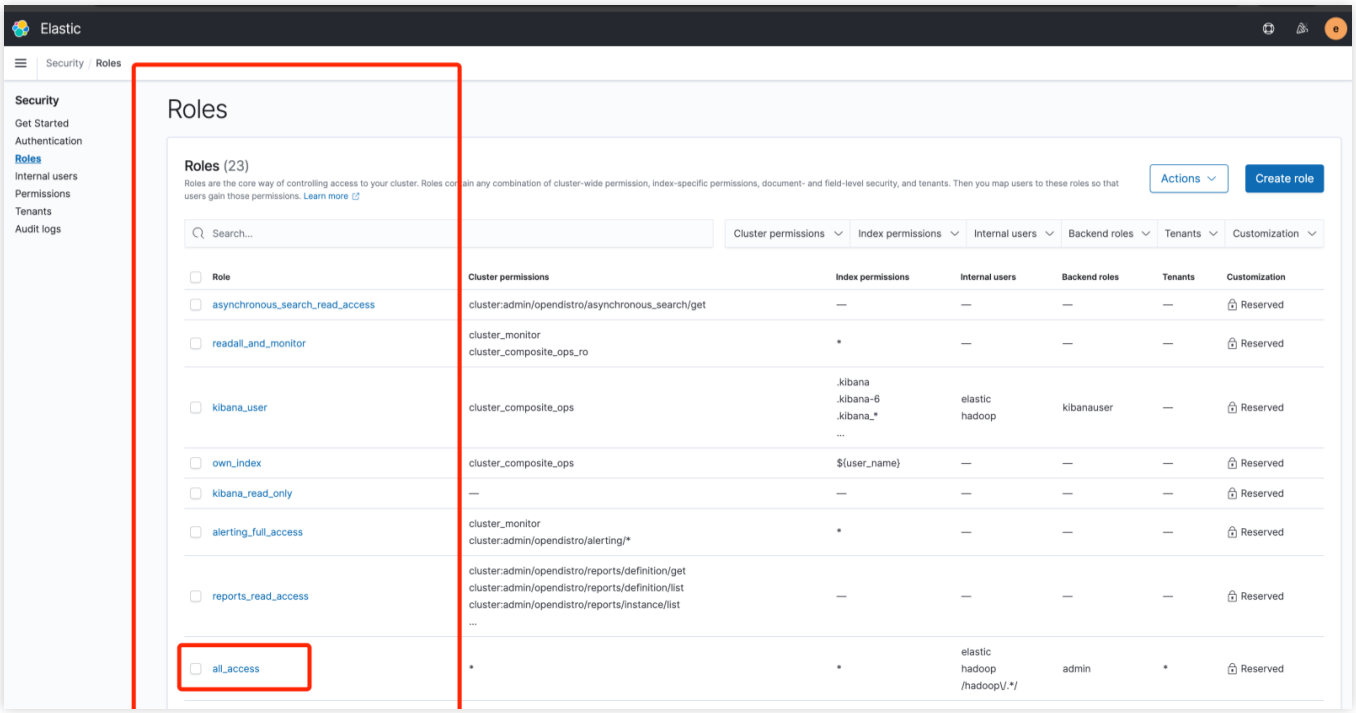


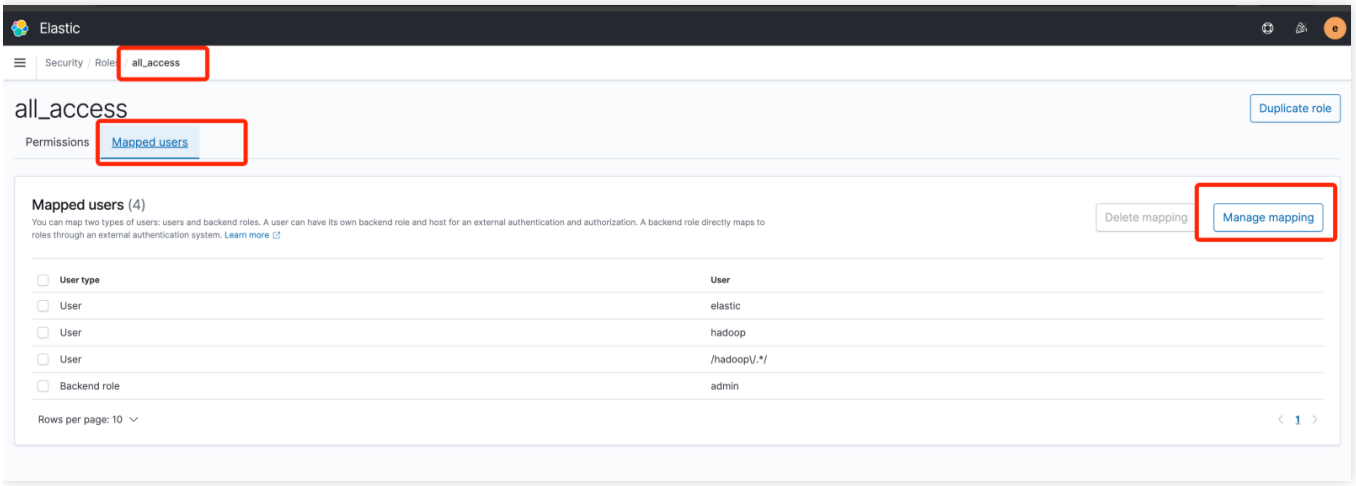
若要为该角色设置索引中字段的权限, 可以在Field Level Security中填写允许的字段。



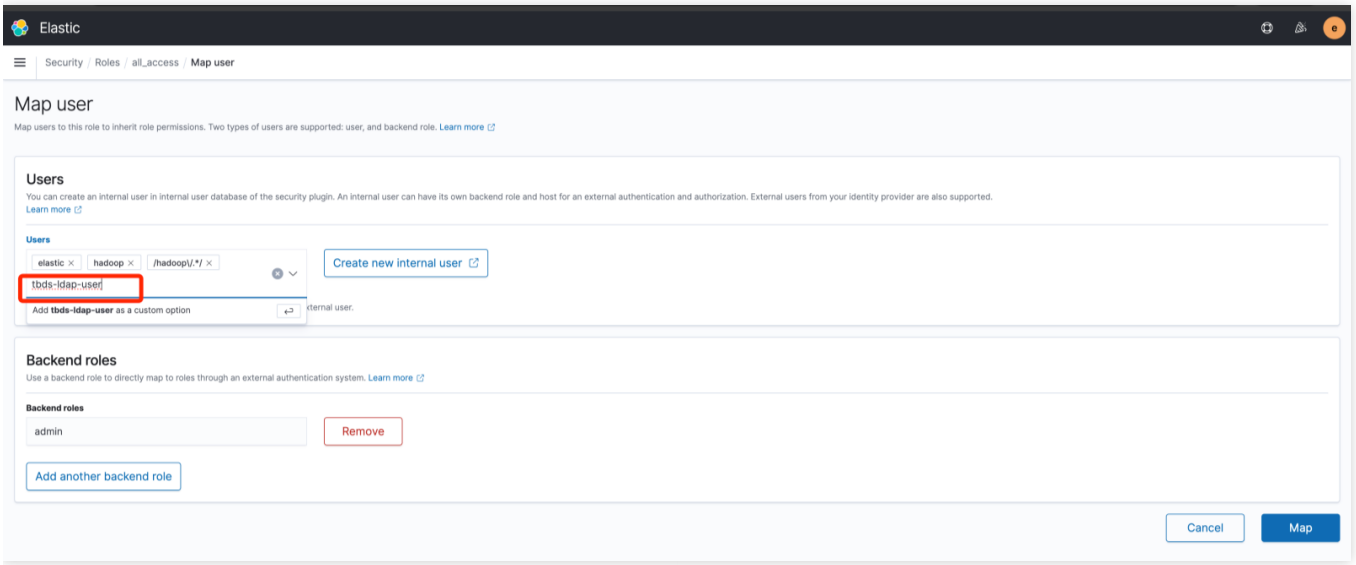
4. 为用户分配角色

   在页面创建用户后, 或者在 TBDS 平台中创建的用户默认在 ES 集群中是没有权限的, 需要参考如下步骤配置用户的角色。在 Roles 页面, 选择对应的角色赋予给用户。



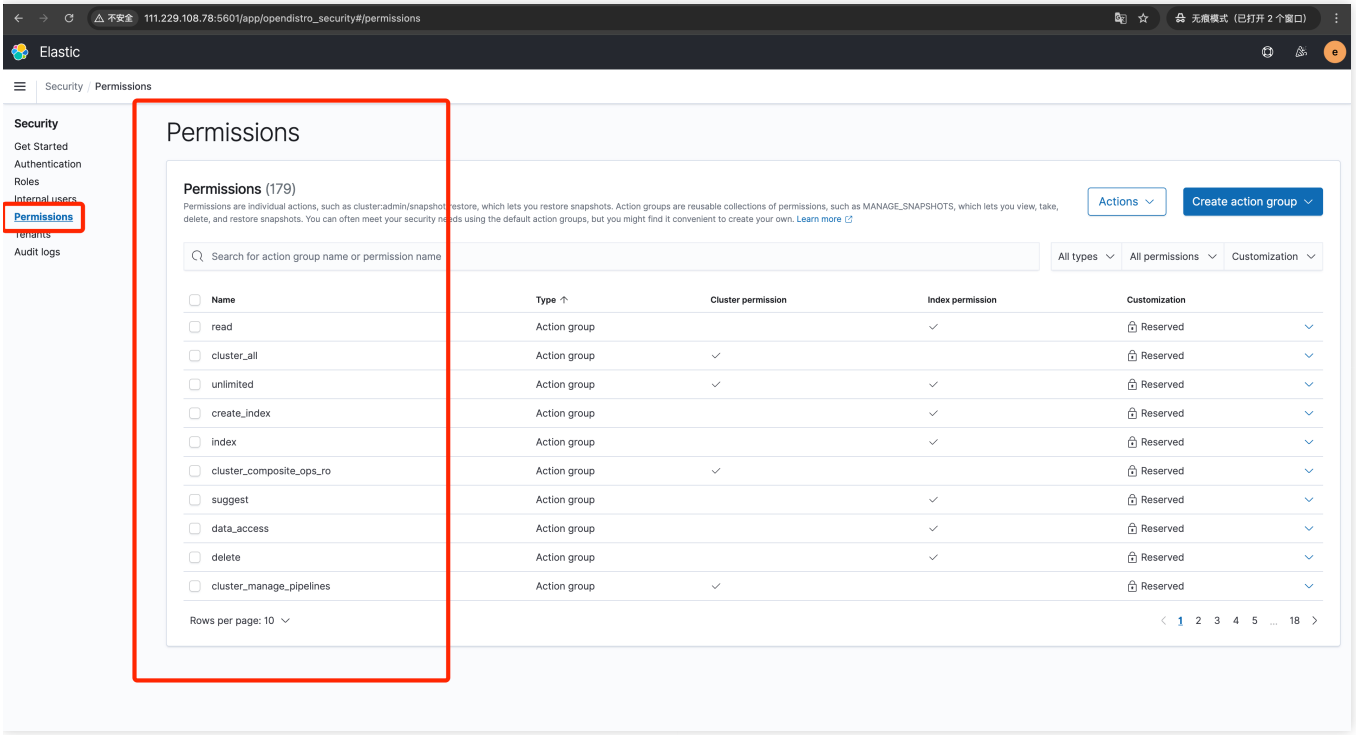


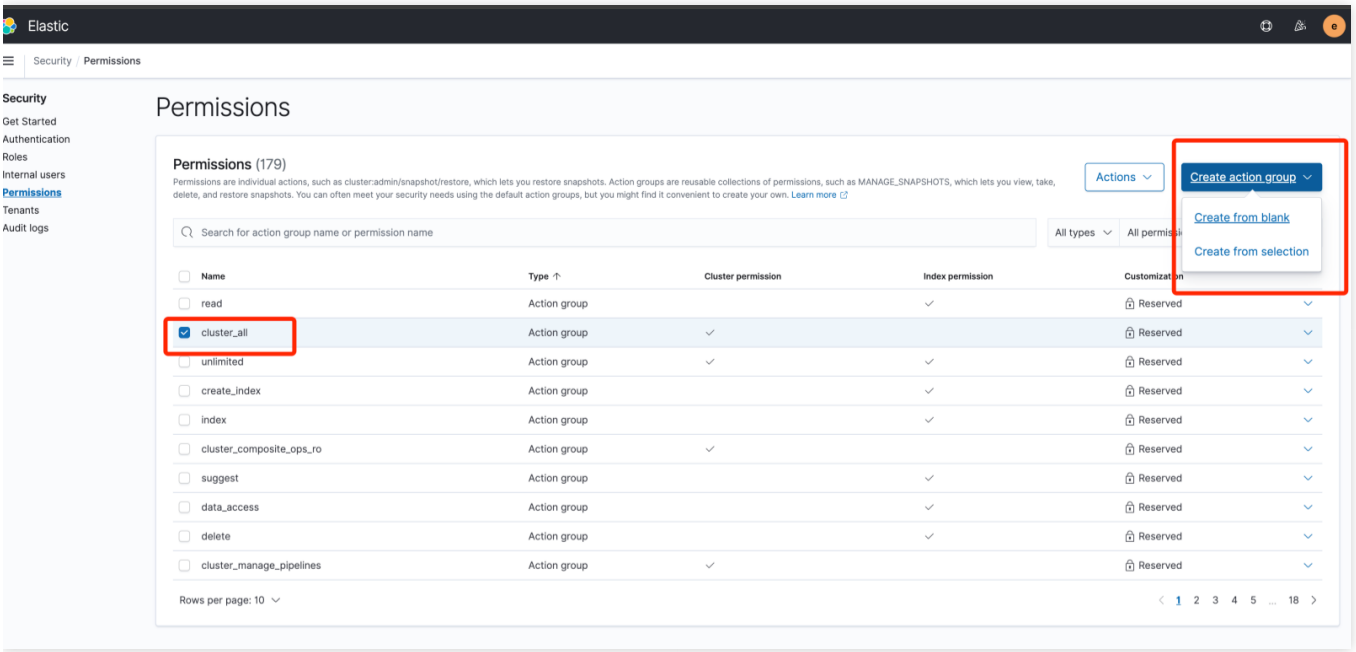
选择internal_users中的用户名或者 TBDS 中的用户名, 单击右下角的Map键, 即可为用户赋予这个角色的权限。



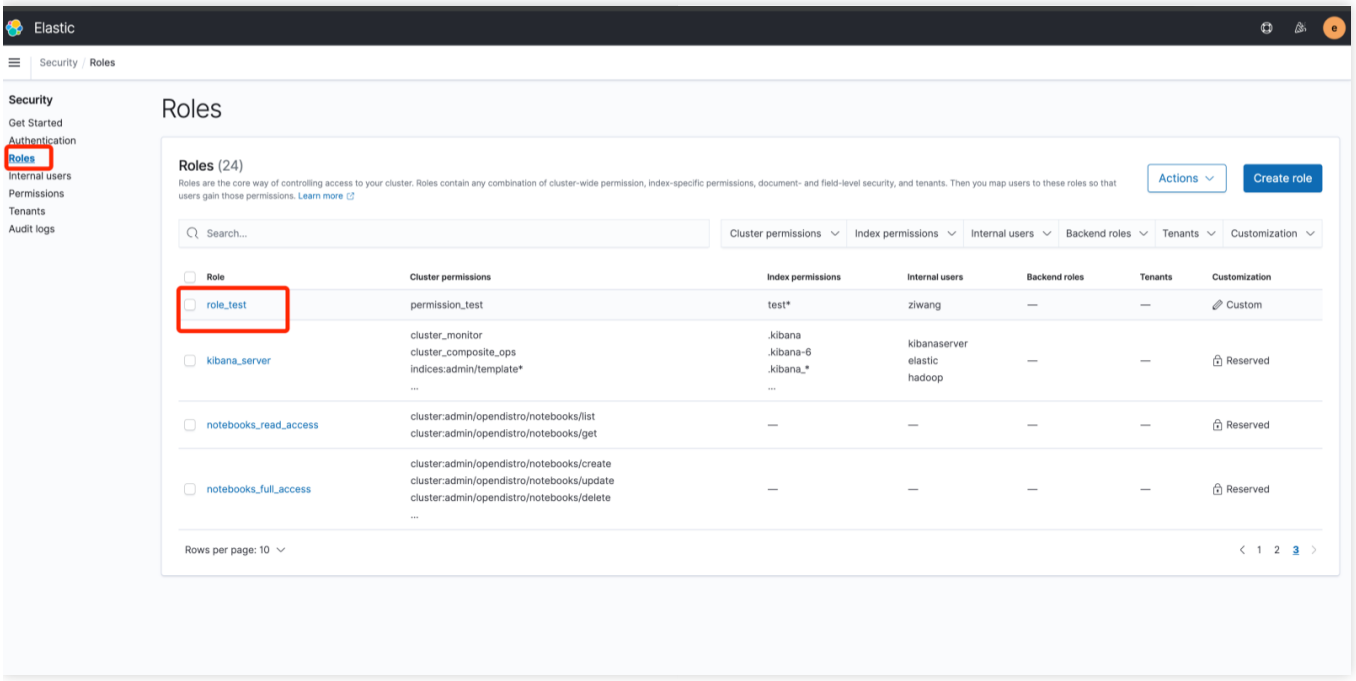
5. 配置权限

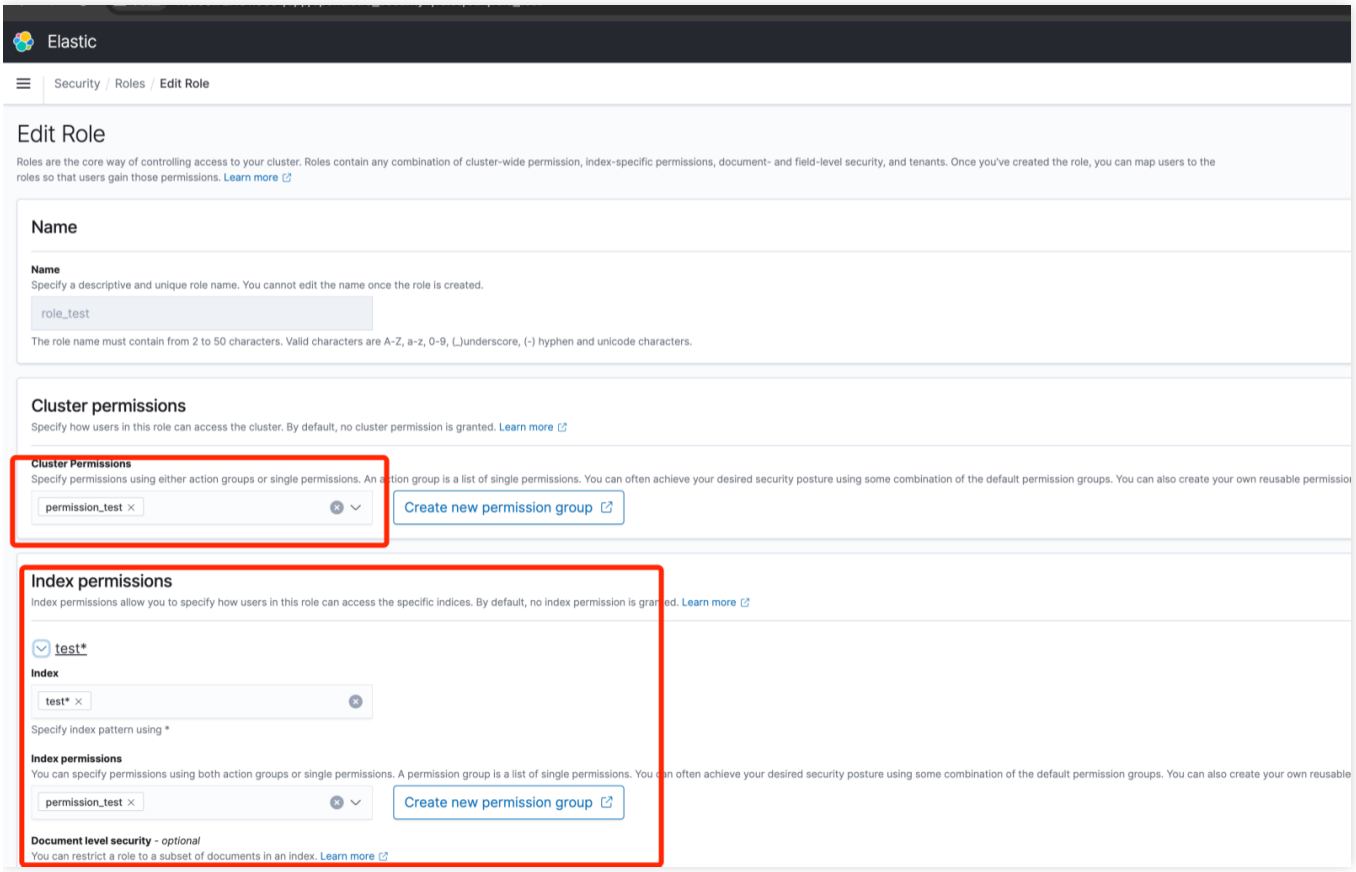
Permissions页面已经配置了各种各样的权限了，基本可以满足大部分需求了，不过若是还想创建权限组，可以单击下图右上角创建新的权限。





6. 为角色配置所创建的权限，单击已有的 Role 角色，并进行编辑添加。





在 Kibana 上使用 Index 管理

TBDS-ES的索引生命周期管理是基于 ISM 插件实现。支持通过索引使用期限、索引大小或文档数等信息的变化来自动触发周期性的管理操作。通过ISM插件可以自定义索引策略，实现自动处理索引的滚动或删除，以优化集群搜索性能或降低存储成本。

ISM 的使用流程包括：

1. 创建生命周期策略
2. 索引关联生命周期策略
3. 管理索引策略

   通过 ISM，业务上可以实现时间序列数据滚动索引等需求场景，更多参考：ISM。

高级功能说明
Index 管理

简要介绍

TBDS-ES的索引生命周期管理是基于 ISM 插件实现。支持通过索引使用期限、索引大小或文档数等信息的变化来自动触发周期性的管理操作。通过ISM插件可以自定义索引策略，实现自动处理索引的滚动或删除，以优化集群搜索性能或降低存储成本。
ISM 的使用流程包括：

1. 创建生命周期策略
2. 索引关联生命周期策略
3. 管理索引策略
   通过 ISM，业务上可以实现时间序列数据滚动索引等需求场景，更多参考：ISM

参考实践

通过索引生命周期实现时间序列数据滚动索引。

背景：对于时间序列数据，随着时间推移数据持续写入，索引会越来越大，通过生命周期管理来定期将数据滚动到新索引，并将历史老索引删除，实现自动滚动索引。
本案例通过配置生命周期策略，当索引的大小达到1TB或索引创建超过1天时【注：索引的滚动条件1天是以索引的创建时间来计算的，并不是完整自然日区分的。】，自动滚动生成新索引；当索引创建7天后，关闭数据副本；当索引创建30天后，删除该索引。

1. 创建索引和索引别名
2. 批量插入数据
3. 执行滚动索引
4. 创建索引滚动策略
5. 创建规则
6. 测试验证

具体执行步骤：

1. 创建索引和索引别名

```
PUT /%3Cexpample_%7Bnow%2Fd%7D-1%3E
{
  "settings": {
    "number_of_shards": 1,
    "analysis": {
      "analyzer": {
        "my_tokenizer": {
          "type": "pattern",
          "pattern": ","
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "field1": {
        "type": "text",
        "analyzer": "my_tokenizer",
        "search_analyzer": "my_tokenizer"
      }
    }
  },
  "aliases": {
    "expample_aliase": {}
  }
}
```

2. 批量插入数据

##批量创建文档

```
POST expample_tiem_aliase/_bulk
{"index":{"_id":1}}
{"id":1, "field1": "1,dog"}
{"index":{"_id":2}}
{"id":2, "field1": "2,dog"}
{"index":{"_id":3}}
{"id":3, "field1": "3,dog"}
{"index":{"_id":4}}
{"id":4, "field1": "4,dog"}
{"index":{"_id":5}}
{"id":5, "field1": "5,dog"}
{"index":{"_id":6}}
{"id":6, "field1": "6,dog"}
```

3. 执行滚动索引

当索引满足1天或者2个文档或者1GB文档大小条件时触发滚动

```
POST /expample_tiem_aliase/_rollover
{
  "conditions": {
    "max_age":   "1d",
    "max_docs":  2,
    "max_size":  "1gb"
  }
}
```

自动滚动索引测试

1. 创建索引滚动策略

```
PUT _opendistro/_ism/policies/rollover_policy?if_seq_no=71&if_primary_term=1
{
  "policy": {
    "description": "Example rollover policy.",
    "default_state": "rollover",
    "states": [
      {
        "name": "rollover",
        "actions": [
          {
            "rollover": {
              "min_doc_count": 30,
              "min_index_age": "1d"
            }
          }
        ],
        "transitions": []
      }
    ],
    "ism_template": {
      "index_patterns": ["log*"],
      "priority": 100
    }
  }
}
```

2. 查询

```
GET _opendistro/_ism/policies/rollover_policy
```

取别名

```
PUT _index_template/ism_rollover
{
  "index_patterns": ["log*"],
  "template": {
    "settings": {
      "opendistro.index_state_management.rollover_alias": "log"
    }
  }
}
```

结果说明：每天会生成log开头的索引别名指向所有log*

自动生成带时间戳的索引测试

1. 创建索引滚动策略

```
 PUT /%3Cexpample_%7Bnow%2Fd%7D-1%3E
{
  "settings": {
    "number_of_shards": 1,
    "analysis": {
      "analyzer": {
        "my_tokenizer": {
          "type": "pattern",
          "pattern": ","
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "field1": {
        "type": "text",
        "analyzer": "my_tokenizer",
        "search_analyzer": "my_tokenizer"
      }
    }
  },
  "aliases": {
    "expample_tiem_aliase": {}
  }
}
```

2. 创建规则

```
 POST /expample_tiem_aliase/_rollover
{
  "conditions": {
    "max_age":   "1d"

  }
}
```

3. 创建定时任务

```
  vim /root/index_rollover.sh
curl  -X  POST -u "{用户名}"{ip:port}/{索引别名}/_rollover  -H "Content-Type: application/json" -d '{"condi
tions": {"max_age": "1d"}}'
crontab –e 0 1 * * * /root/index_rollover.sh > /dev/null 2>&1 &
```

CCR 跨集群数据复制

在 TBDS-ES中，CCR（Cross-cluster replication，跨集群数据复制）是一种插件能力，允许在不同的 Elasticsearch 集群之间进行数据复制。
其他参考：https://github.com/opendistro-for-elasticsearch/cross-cluster-replication

主要功能特点

1. 数据同步
   CCR 能够实现从一个源集群（leader 集群）到一个或多个目标集群（follower 集群）的实时数据复制。这确保了多个集群之间的数据一致性，对于需要在不同地理位置或不同环境中访问相同数据的场景非常有用。
   例如，一个企业可能有一个主要的数据中心作为源集群，同时在其他地区设置了备份集群作为 follower 集群，以提高数据的可用性和灾难恢复能力。
2. 自动故障转移

- 在配置了 CCR 的情况下，如果源集群出现故障，目标集群可以自动提升为新的 leader 集群，继续提供服务。这提供了一种高可用性的解决方案，确保业务的连续性。
  例如，当源集群由于硬件故障或网络问题不可用时，follower 集群可以迅速接管，用户可以继续查询和写入数据到新的 leader 集群。
  灵活的配置
- 用户可以根据自己的需求配置 CCR 的复制策略。可以选择复制特定的索引或整个集群的数据，可以设置复制的频率和延迟等参数。
  例如，可以只复制一部分关键业务数据到目标集群，以降低网络带宽和存储成本。同时，可以根据业务需求调整复制的频率，以平衡数据同步的及时性和系统资源的消耗。
  增量复制
- CCR 采用增量复制的方式，只复制自上次复制以来发生变化的数据。这大大减少了网络带宽的使用和复制时间，提高了复制效率。
  例如，当源集群中的数据发生少量变化时，只需要复制这些变化的数据到目标集群，而不是整个数据集。

应用场景

1. 高可用性和灾难恢复
   通过在不同地理位置或数据中心设置多个集群，并使用 CCR 进行数据复制，可以在源集群出现故障时快速切换到目标集群，保证业务的连续性。
   例如，金融机构可以使用 CCR 来确保在发生灾难事件时，客户数据仍然可用，并且可以继续进行交易和查询。

2. 多数据中心部署

对于具有多个数据中心的企业，可以使用 CCR 在不同数据中心之间复制数据，以提高数据的可用性和访问速度。用户可以根据地理位置就近访问数据，减少网络延迟。

例如，跨国企业可以在不同国家的数据中心之间复制数据，以便当地用户能够快速访问和查询数据。

3. 数据分析和报告

可以将生产数据复制到一个专门用于数据分析和报告的集群中，而不会影响生产集群的性能。这样可以在不干扰生产环境的情况下进行大规模的数据分析和报告生成。

例如，企业可以将每天的交易数据复制到一个分析集群中，进行数据挖掘和趋势分析，以支持业务决策。

参考实践

> （示例为固定索引的配置 demo，如果需要配置映射到索引模板，可以参考auto-follow API来实现。）

1. 设置跨集群连接

设置灾备集群到主集群的远程连接，进入灾备集群Kibana Dev Tools执行：

```
PUT /_cluster/settings?pretty
{
  "persistent": {
    "cluster": {
      "remote": {
        "leader-cluster": {
          "seeds": [
            "100.103.0.5:9300"
          ]
        }
      }
    }
  }
}
```

2. 在主集群创建样本数据，创建索引。进入主集群Kibana Dev Tools，执行：

```
  PUT /ccr_leader
{
  "settings": {
    "number_of_shards": 2,
    "number_of_replicas": 1
  },
  "mappings": {
```

```
    "properties": {
      "field1": { "type": "text" }
    }
  }
}
```

批量写入样本数据：

```
POST ccr_leader/_bulk
{"index":{"_id":1}}
{"id":1, "field1": "1,dog"}
{"index":{"_id":2}}
{"id":2, "field1": "2,dog"}
{"index":{"_id":3}}
{"id":3, "field1": "3,dog"}
```

3. 在灾备集群开启复制。进入灾备集群Kibana Dev Tools，执行：

```
PUT _opendistro/_replication/ccr_follower/_start?pretty
{
  "remote_cluster": "leader-cluster", "remote_index": "ccr_leader"
}
```

4. 进入灾备集群Kibana Dev Tools，查看数据，执行：

```
GET ccr_follower/_search
```

5. 对比主备集群数据
   查看主备集群数据对比。
6. 停止数据复制，进入灾备集群Kibana Dev Tools，执行：

```
POST _opendistro/_replication/ccr_follower/_stop?pretty
```

【非标】kNN（k-Nearest neighbor）search

简要介绍

TBDS-ES 内置了向量检索插件——OpenDistro for Elasticsearch k-NN，它是一个基于 Elasticsearch 的开源插件，旨在提供高效的 k-近邻（k-Nearest Neighbors, k-NN）搜索功能。该插件利用了 Elasticsearch 的分布式特性，使得在大规模数据集上进行向量检索成为可能。k-NN 搜索在图像搜索、推荐系统、图像识别、自然语言处理等领域有着广泛的应用，能够帮助用户快速实现向量检索场景的需求。参考：OpenDistro for Elasticsearch k-NN

参考实践

k-NN 插件安装说明

1. 通过 elasticsearch-plugin 安装使用命令如下，然后重启ES 即可用。

./bin/elasticsearch-plugin install file:///data1/knn-1.0.0.1.zip

2. 或者通过TBDS 平台的插件管理进行自定义插件安装（WebUI 化）

创建 k-NN 索引

创建 k-NN 索引首先需要设置 index.knn 为 true，则插件会为索引创建HNSW图。其次，如果打算使用近似 k-NN，需要通过 index.knn.space_type 指定空间类型。
然后再添加一个或多个knn_vector数据类型的字段。 knn_vector 数据类型支持 dimension <=10000。NMSLIB是支持点积的，但是目前插件还未支持点积。

```
PUT my-knn-index-1
{
  "settings": {
    "index": {
      "knn": true,
      "knn.space_type": "cosinesimil"
```

```
    }
   },
   "mappings": {
    "properties": {
     "my_vector1": {
      "type": "knn_vector",
      "dimension": 2
     },
     "my_vector2": {
      "type": "knn_vector",
      "dimension": 4
     }
    }
   }
}
```

插入一些数据:

```
POST _bulk
{ "index": { "_index": "my-knn-index-1", "_id": "1" } }
{ "my_vector1": [1.5, 2.5], "price": 12.2 }
{ "index": { "_index": "my-knn-index-1", "_id": "2" } }
{ "my_vector1": [2.5, 3.5], "price": 7.1 }
{ "index": { "_index": "my-knn-index-1", "_id": "3" } }
{ "my_vector1": [3.5, 4.5], "price": 12.9 }
{ "index": { "_index": "my-knn-index-1", "_id": "4" } }
{ "my_vector1": [5.5, 6.5], "price": 1.2 }
{ "index": { "_index": "my-knn-index-1", "_id": "5" } }
{ "my_vector1": [4.5, 5.5], "price": 3.7 }
{ "index": { "_index": "my-knn-index-1", "_id": "6" } }
{ "my_vector2": [1.5, 5.5, 4.5, 6.4], "price": 10.3 }
{ "index": { "_index": "my-knn-index-1", "_id": "7" } }
{ "my_vector2": [2.5, 3.5, 5.6, 6.7], "price": 5.5 }
{ "index": { "_index": "my-knn-index-1", "_id": "8" } }
{ "my_vector2": [4.5, 5.5, 6.7, 3.7], "price": 4.4 }
{ "index": { "_index": "my-knn-index-1", "_id": "9" } }
{ "my_vector2": [1.5, 5.5, 4.5, 6.4], "price": 8.9 }
```

近似最近邻搜索

query 类型使用 k-NN，样例如下：

样例1: returns k amount of results for each shard (and each segment) and size amount of results for th e entire query. The plugin supports a maximum k value of 10,000.

```
{
  "size": 2,
  "query": {
    "knn": {
      "my_vector2": {
        "vector": [
          2,
          3,
          5,
          6
        ],
        "k": 2
      }
    }
  }
}

// 返回如下
{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2,
      "relation" : "eq"
    },
    "max_score" : 0.99958557,
    "hits" : [
      {
        "_index" : "my-knn-index-1",
        "_type" : "_doc",
        "_id" : "7",
        "_score" : 0.99958557,
        "_source" : {
          "my_vector2" : [
            2.5,
            3.5,
            5.6,
```

```
          6.7
        ],
        "price" : 5.5
      }
    },
    {
      "_index" : "my-knn-index-1",
      "_type" : "_doc",
      "_id" : "6",
      "_score" : 0.96664,
      "_source" : {
        "my_vector2" : [
          1.5,
          5.5,
          4.5,
          6.4
        ],
        "price" : 10.3
      }
    }
  ]
 }
}

// 样例2：同时使用 knn 跟 filter 过滤
GET my-knn-index-1/_search
{
 "size": 2,
 "query": {
  "knn": {
   "my_vector2": {
    "vector": [
     2,
     3,
     5,
     6
    ],
    "k": 2
   }
  }
 },
 "post_filter": {
  "range": {
   "price": {
    "gte": 5,
    "lte": 10
```

```
      }
    }
  }
}

// 返回如下
{
  "took" : 11,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 0.99958557,
    "hits" : [
      {
        "_index" : "my-knn-index-1",
        "_type" : "_doc",
        "_id" : "7",
        "_score" : 0.99958557,
        "_source" : {
          "my_vector2" : [
            2.5,
            3.5,
            5.6,
            6.7
          ],
          "price" : 5.5
        }
      }
    ]
  }
}
```

请求k-NN统计数据API

```
GET /_knn/stats?pretty
```

```
{
  "_nodes": {
    "total": 1,
    "successful": 1,
    "failed": 0
  },
  "cluster_name": "_run",
  "circuit_breaker_triggered": false,
  "nodes": {
    "HYMrXXsBSamUkcAjhjeN0w": {
      "eviction_count": 0,
      "miss_count": 1,
      "graph_memory_usage": 1,
      "graph_memory_usage_percentage": 3.68,
      "graph_index_requests": 7,
      "graph_index_errors": 1,
      "knn_query_requests": 4,
      "graph_query_requests": 30,
      "graph_query_errors": 15,
      "indices_in_cache": {
        "myindex": {
          "graph_memory_usage": 2,
          "graph_memory_usage_percentage": 3.68,
          "graph_count": 2
        }
      },
      "cache_capacity_reached": false,
      "load_exception_count": 0,
      "hit_count": 0,
      "load_success_count": 1,
      "total_load_time": 2878745,
      "script_compilations": 1,
      "script_compilation_errors": 0,
      "script_query_requests": 534,
      "script_query_errors": 0
    }
  }
}

# 返回如下
{
  "_nodes" : {
    "total" : 3,
    "successful" : 3,
    "failed" : 0
  },
  "cluster_name" : "tbds-es",
```

```
"circuit_breaker_triggered" : false,
"nodes" : {
 "7QDJxgZyTci9172CWn1sQA" : {
  "miss_count" : 1,
  "graph_memory_usage_percentage" : 3.3725044E-6,
  "graph_query_requests" : 1,
  "graph_memory_usage" : 1,
  "cache_capacity_reached" : false,
  "graph_index_requests" : 2,
  "load_exception_count" : 0,
  "load_success_count" : 1,
  "eviction_count" : 0,
  "indices_in_cache" : {
   "my-knn-index-1" : {
    "graph_memory_usage" : 1,
    "graph_memory_usage_percentage" : 3.3725044E-6,
    "graph_count" : 1
   }
  },
  "script_query_errors" : 0,
  "script_compilations" : 0,
  "script_query_requests" : 0,
  "graph_query_errors" : 0,
  "hit_count" : 0,
  "graph_index_errors" : 0,
  "knn_query_requests" : 0,
  "total_load_time" : 1656803,
  "script_compilation_errors" : 0
 },
 "Gjd1qGPuQ0WzvuKSQ00gvg" : {
  "miss_count" : 0,
  "graph_memory_usage_percentage" : 0.0,
  "graph_query_requests" : 0,
  "graph_memory_usage" : 0,
  "cache_capacity_reached" : false,
  "graph_index_requests" : 0,
  "load_exception_count" : 0,
  "load_success_count" : 0,
  "eviction_count" : 0,
  "indices_in_cache" : { },
  "script_query_errors" : 0,
  "script_compilations" : 0,
  "script_query_requests" : 0,
  "graph_query_errors" : 0,
  "hit_count" : 0,
  "graph_index_errors" : 0,
  "knn_query_requests" : 3,
```

```
      "total_load_time" : 0,
      "script_compilation_errors" : 0
    },
    "GHFpATcmQqqRIL7pTrZWiA" : {
      "miss_count" : 1,
      "graph_memory_usage_percentage" : 3.3725044E-6,
      "graph_query_requests" : 2,
      "graph_memory_usage" : 1,
      "cache_capacity_reached" : false,
      "graph_index_requests" : 2,
      "load_exception_count" : 0,
      "load_success_count" : 1,
      "eviction_count" : 0,
      "indices_in_cache" : {
        "my-knn-index-1" : {
          "graph_memory_usage" : 1,
          "graph_memory_usage_percentage" : 3.3725044E-6,
          "graph_count" : 1
        }
      },
      "script_query_errors" : 0,
      "script_compilations" : 0,
      "script_query_requests" : 0,
      "graph_query_errors" : 0,
      "hit_count" : 1,
      "graph_index_errors" : 0,
      "knn_query_requests" : 0,
      "total_load_time" : 1876648,
      "script_compilation_errors" : 0
    }
  }
}
```

API接口说明

TBDS大数据平台上的ElasticSearch访问接口与开源兼容，可以参考https://www.elastic.co/guide/en/elasticsearch/reference/7.10/rest-apis.html。以下给出部分常用的API样例。
创建索引：这个命令用于创建一个名为index_name的索引。

```
PUT /index_name
```

删除索引：这个命令用于删除一个名为index_name的索引。

```
DELETE /index_name
```

插入文档：这个命令用于在index_name索引中插入一个新的文档。文档的内容是JSON格式的，包含了一个或多个字段。

```
POST /index_name/_doc
{
  "field1": "value1",
  "field2": "value2"
}
```

查询文档：这个命令用于查询index_name索引中字段field的值为value的文档。

```
GET /index_name/_search?q=field:value
```

更新文档：这个命令用于更新index_name索引中ID为doc_id的文档，将字段field的值更新为new_value。

```
POST /index_name/_update/doc_id
{
  "doc": {
    "field": "new_value"
  }
}
```

删除文档：这个命令用于删除index_name索引中ID为doc_id的文档。

```
DELETE /index_name/_doc/doc_id
```

常见问题处理

TBDS提供标准的ElasticSearch 7.10.1版本，ES是一个强大的搜索和分析引擎，但在使用过程中可能会遇到一些常见的问题。以下是一些常见问题及其解决方案：

- 集群健康状态为红色或黄色：这可能是由于某些分片无法分配。你可以使用_cat/shards API查看无法分配的分片，然后根据具体情况进行处理，比如增加节点，调整分片和副本的数量等。
- 搜索性能低下：这可能是由于查询过于复杂，或者数据量过大。你可以尝试优化你的查询，比如使用更精确的查询而不是全文搜索，避免使用高开销的特性如脚本和深度分页等。另外，你也可以考虑使用更强大的硬件，或者增加节点来提高性能。
- 数据丢失：这可能是由于节点故障或者误操作导致的。你应该定期备份你的数据，以防止数据丢失。如果数据已经丢失，你可以尝试从备份中恢复。
- 内存不足：Elasticsearch是一个内存密集型的应用，如果内存不足可能会导致性能下降或者节点崩溃。你应该监控你的内存使用情况，并根据需要调整JVM的堆大小或者增加节点。
- 磁盘空间不足：Elasticsearch需要足够的磁盘空间来存储数据和日志。如果磁盘空间不足，可能会导致数据丢失或者节点崩溃。你应该监控你的磁盘使用情况，并根据需要增加磁盘空间或者清理不需要的数据。
  如有其他问题可参考官方文档指南https://www.elastic.co/guide/en/elasticsearch/reference/7.10/index.html, 也可到ES节点的/data/emr/es/log/tbds-es.log查看es的日志内容。

StarRocks开发

概述

StarRocks是新一代极速全场景MPP(Massively Parallel Processing)数据库。StarRocks的愿景是能够让用户的数据分析变得更加简单和敏捷。用户无需经过复杂的预处理，就可以用 StarRocks 来支持多种数据分析场景的极速分析。以下是StarRocks的一些关键特点和概念：

- StarRocks 架构简洁，采用了全面向量化引擎，并配备全新设计的 CBO (Cost Based Optimizer) 优化器，查询速度（尤其是多表关联查询）远超同类产品。
- StarRocks 能很好地支持实时数据分析，并能实现对实时更新数据的高效查询。StarRocks 还支持现代化物化视图，进一步加速查询。
- 使用 StarRocks，用户可以灵活构建包括大宽表、星型模型、雪花模型在内的各类模型。
- StarRocks 兼容 MySQL 协议，支持标准 SQL 语法，易于对接使用，全系统无外部依赖，高可用，易于运维管理。StarRocks 还兼容多种主流 BI 产品，包括 Tableau、Power BI、FineBI 和 Smartbi。

">适用场景

StarRocks 可以满足企业级用户的多种分析需求，包括 OLAP (Online Analytical Processing) 多维分析、定制报表、实时数据分析和 ad hoc 数据分析等。

">OLAP 多维分析

利用 StarRocks 的 MPP 框架和向量化执行引擎，用户可以灵活的选择雪花模型，星型模型，宽表模型或者预聚合模型。适用于灵活配置的多维分析报表，业务场景包括：

- 用户行为分析
- 用户画像、标签分析、圈人
- 高维业务指标报表
- 自助式报表平台
- 业务问题探查分析
- 跨主题业务分析
- 财务报表
- 系统监控分析

">实时数据仓库

StarRocks 设计和实现了主键表，能够实时更新数据并极速查询，可以秒级同步 TP (Transaction Processing) 数据库的变化，构建实时数仓，业务场景包括：

- 电商大促数据分析
- 物流行业的运单分析
- 金融行业绩效分析、指标计算
- 直播质量分析
- 广告投放分析
- 管理驾驶舱
- 探针分析APM（Application Performance Management）

">高并发查询

StarRocks 通过良好的数据分布特性，灵活的索引以及物化视图等特性，可以支持面向用户侧的分析场景，业务场景包括：

- 广告主报表分析
- 零售行业渠道人员分析
- SaaS 行业面向用户分析报表
- Dashboard 多页面分析

">统一分析

- 通过使用一套系统支持多维分析、高并发查询、预计算、实时分析查询等场景，降低系统复杂度和多技术栈开发与维护成本。
- 使用 StarRocks 统一管理数据湖和数据仓库，将高并发和实时性要求很高的业务放在 StarRocks 中分析，也可以使用 External Catalog 和外部表进行数据湖上的分析。

快速使用
表，分区，分桶，索引
CREATE INDEX

">功能

创建索引。仅支持使用该语句创建 Bitmap 索引。
提示

- 该操作需要对应表的 ALTER 权限。请参考 GRANT 为用户赋权。
- 一列只能创建一个 Bitmap index。如果某列已有 index，再次创建会返回失败。

">语法

```
CREATE INDEX index_name ON table_name (column_name) [USING BITMAP] [COMMENT'']
```

参数说明

参数	必选	说明
index_name	是	索引名称，命名要求参见系统限制。在同一张表中不能创建名称相同的索引。
table_name	是	表名。
column_name	是	创建索引的列名。执行一次该语句只能为某一列创建索引，且同一列只能创建一个索引。
COMMENT	否	索引备注。

">示例

例如有一张表 sales_records，其建表语句如下：

```
CREATE TABLE sales_records
(
    record_id int,
    seller_id int,
    item_id int
)
DISTRIBUTED BY hash(record_id)
PROPERTIES (
    "replication_num" = "3"
);
```

为表 sales_records 中的 item_id 列创建 Bitmap 索引，索引名称为 index3。

```
CREATE INDEX index3 ON sales_records (item_id) USING BITMAP COMMENT '';
```

或

```
CREATE INDEX index3 ON sales_records (item_id);
```

CREATE TABLE

">功能

该语句用于创建表。

语法

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [database.]table_name
(column_definition1[, column_definition2, ...]
[, index_definition1[, index_definition2, ...]])
[ENGINE = [olap|mysql|elasticsearch|hive|iceberg|hudi|jdbc]]
[key_desc]
[COMMENT "table comment"]
[partition_desc]
[distribution_desc]
[rollup_index]
[ORDER BY (column_definition1,...)]
[PROPERTIES ("key"="value", ...)]
[BROKER PROPERTIES ("key"="value", ...)]
```

参数说明

">column_definition

语法：

```
col_name col_type [agg_type] [NULL | NOT NULL] [DEFAULT "default_value"] [AUTO_INCREMENT] [AS generation_expr]
```

说明：

col_name：列名称。

注意，在一般情况下，不能直接创建以 __op 或 __row 开头命名的列，因为此类列名被 StarRocks 保留用于特殊目的，创建这样的列可能导致未知行为。如需创建这样的列，必须将 FE 动态参数 allow_system_reserved_names 设

置为 TRUE。

col_type：列数据类型
支持的列类型以及取值范围等信息如下：

- TINYINT（1字节） 范围：-2^7 + 1 ~ 2^7 - 1
- SMALLINT（2字节） 范围：-2^15 + 1 ~ 2^15 - 1
- INT（4字节） 范围：-2^31 + 1 ~ 2^31 - 1
- BIGINT（8字节） 范围：-2^63 + 1 ~ 2^63 - 1
- LARGEINT（16字节） 范围：-2^127 + 1 ~ 2^127 - 1
- FLOAT（4字节） 支持科学计数法。
- DOUBLE（8字节） 支持科学计数法。
- DECIMAL[(precision, scale)] (16字节) 保证精度的小数类型。默认是 DECIMAL(10, 0) ，precision: 1 ~ 38；scale: 0 ~ precision，其中整数部分为：precision - scale 不支持科学计数法。
- DATE（3字节） 范围：0000-01-01 ~ 9999-12-31
- DATETIME（8字节） 范围：0000-01-01 00:00:00 ~ 9999-12-31 23:59:59
- CHAR[(length)]定长字符串。长度范围：1 ~ 255。默认为 1。
- VARCHAR[(length)]变长字符串。单位：字节，默认取值为 1。
- StarRocks 2.1.0 之前的版本，length 的取值范围为 1~65533。
- 【公测中】自 StarRocks 2.1.0 版本开始，length 的取值范围为 1~1048576。
- HLL (1~16385个字节)HLL 列类型，不需要指定长度和默认值，长度根据数据的聚合程度系统内控制，并且 HLL 列只能通过配套的 hll_union_agg、hll_cardinality、hll_hash进行查询或使用。
- BITMAP BITMAP 列类型，不需要指定长度和默认值。表示整型的集合，元素个数最大支持到 2^64 - 1。
- ARRAY 支持在一个数组中嵌套子数组，最多可嵌套 14 层。您必须使用尖括号（ < 和 > ）来声明 ARRAY 的元素类型，如 ARRAY < INT >。目前不支持将数组中的元素声明为 Fast Decimal 类型。

agg_type：聚合类型，如果不指定，则该列为 key 列。否则，该列为 value 列。
支持的聚合类型如下：

- SUM、MAX、MIN、REPLACE
- HLL_UNION（仅用于 HLL列，为 HLL 独有的聚合方式）。
- BITMAP_UNION（仅用于 BITMAP 列，为 BITMAP 独有的聚合方式）。
- REPLACE_IF_NOT_NULL：这个聚合类型的含义是当且仅当新导入数据是非 NULL 值时会发生替换行为。如果新导入的数据是 NULL，那么 StarRocks 仍然会保留原值。
  注意：

1. BITMAP_UNION 聚合类型列在导入时的原始数据类型必须是 TINYINT, SMALLINT, INT, BIGINT。
2. 如果在建表时 REPLACE_IF_NOT_NULL 列指定了 NOT NULL，那么 StarRocks 仍然会将其设为 NULL，不会向用户报错。用户可以借助这个类型完成「部分列导入」的功能。 该类型只对聚合表有用 (key_desc 的 type 为 AGGREGATE KEY)。自 3.1.9 起，REPLACE_IF_NOT_NULL 新增支持 BITMAP 类型的列。

NULL | NOT NULL：列数据是否允许为 NULL。其中明细表、聚合表和更新表中所有列都默认指定 NULL。主键表的指标列默认指定 NULL，维度列默认指定 NOT NULL。如源数据文件中存在 NULL 值，可以用 \N 来表示，导入时 StarRocks 会将其解析为 NULL。

DEFAULT "default_value"：列数据的默认值。导入数据时，如果该列对应的源数据文件中的字段为空，则自动填充 DEFAULT 关键字中指定的默认值。支持以下三种指定方式：

- DEFAULT current_timestamp：默认值为当前时间。
- DEFAULT <默认值>：默认值为指定类型的值。例如，列类型为 VARCHAR，即可指定默认值为 DEFAULT "city"。当前不支持指定 ARRAY、BITMAP、JSON、HLL 和 BOOLEAN 类型为默认值。
- DEFAULT (<表达式>)：默认值为指定函数返回的结果。目前仅支持 UUID() 和 uuid_numeric() 表达式。

AUTO_INCREMENT：指定自增列。自增列的数据类型只支持 BIGINT，自增 ID 从 1 开始增加，自增步长为 1。自 v3.0，StarRocks 支持该功能。

AS generation_expr：指定生成列和其使用的表达式。生成列用于预先计算并存储表达式的结果，可以加速包含复杂表达式的查询。自 v3.1，StarRocks 支持该功能。

">index_definition

创建 BITMAP 索引的语法如下：
INDEX index_name (col_name[, col_name, ...]) [USING BITMAP] [COMMENT '']

ENGINE 类型

默认为 OLAP，表示创建的是 StarRocks 内部表。
可选值：mysql、elasticsearch、hive、JDBC (2.3 及以后)、iceberg、hudi（2.2 及以后）。如果指定了可选值，则创建的是对应类型的外部表 (external table)，在建表时需要使用 CREATE EXTERNAL TABLE。
从 3.0 版本起，对于查询 Hive、Iceberg的场景，推荐使用 Catalog 直接查询，不再推荐外部表的方式。
从 3.1 版本起，支持直接在 Iceberg catalog 内创建表（当前仅支持 Parquet 格式的表），您可以通过 INSERT INTO 把数据插入到 Iceberg 表中。
从 3.2 版本起，支持直接在 Hive Catalog 内创建 Parquet 格式的表，并支持通过 INSERT INTO 把数据插入到 Parquet 格式的 Hive 表中。从 3.3 版本起，支持直接在 Hive Catalog 中创建 ORC 及 Textfile 格式的表，并支持通过 INSERT INTO 把数据插入到 ORC 及 Textfile 格式的 Hive 表中。

1. 如果是 mysql，则需要在 properties 提供以下信息：

```
PROPERTIES (
    "host" = "mysql_server_host",
    "port" = "mysql_server_port",
```

```
    "user" = "your_user_name",
    "password" = "your_password",
    "database" = "database_name",
    "table" = "table_name"
)
```

注意： "table" 条目中的 "table_name" 是 MySQL 中的真实表名。 而 CREATE TABLE 语句中的 table_name 是该 MySQL 表在 StarRocks 中的名字，可以不同。
在 StarRocks 创建 MySQL 表的目的是可以通过 StarRocks 访问 MySQL 数据库。 而 StarRocks 本身并不维护、存储任何 MySQL 数据。

2. 如果是 elasticsearch，则需要在 properties 提供以下信息：

```
PROPERTIES (
    "hosts" = "http://192.168.0.1:8200,http://192.168.0.2:8200",
    "user" = "root",
    "password" = "root",
    "index" = "tindex",
    "type" = "doc"
)
```

其中 hosts 为 Elasticsearch 集群连接地址，可指定一个或者多个，user 和 password 为开启 basic 认证的 Elasticsearch 集群的用户名/密码，index 是 StarRocks 中的表对应的 Elasticsearch 的 index 名字，可以是 alias，type 指定 index 的类型，默认是 doc。

3. 如果是 hive，则需要在 properties 提供以下信息：

```
PROPERTIES (
    "database" = "hive_db_name",
    "table" = "hive_table_name",
    "hive.metastore.uris" = "thrift://xx.xx.xx.xx:9083"
)
```

其中 database 是 Hive 表对应的库名字，table 是 hive 表的名字，hive.metastore.uris 是 Hive metastore 服务地址。

4. 如果是 JDBC，则需要在 properties 提供以下信息：

```
PROPERTIES (
    "resource"="jdbc0",
    "table"="dest_tbl"
)
```

其中 resource 是所使用 JDBC 资源的名称。table 是目标数据库表名。

5. 如果是 iceberg，则需要在 properties 提供以下信息：

```
PROPERTIES (
    "resource" = "iceberg0",
    "database" = "iceberg",
    "table" = "iceberg_table"
)
```

其中 resource 是引用的 Iceberg 资源的名称。database 是 Iceberg 表所属的数据库名称。table Iceberg 表名称。

6. 如果是 hudi，则需要在 properties 提供以下信息：

```
PROPERTIES (
    "resource" = "hudi0",
    "database" = "hudi",
    "table" = "hudi_table"
)
```

key_desc

语法：

```
`key_type(k1[,k2 ...])`
```

说明：
数据按照指定的 key 列进行排序，且根据不同的 key_type 具有不同特性。 key_type 支持以下类型：

- AGGREGATE KEY: key列中相同的记录将根据指定的聚合类型聚合到value列，适合报表、多维分析等业务场景。
- UNIQUE KEY/PRIMARY KEY: 当一条新记录被导入时，如果它的 Key 列的值与表中已存在的某条记录的 Key 列的值完全相同，那么这条新记录将会覆盖那条旧记录，适合按 key 列进行增删改查的点查询 (point query) 业务。
- DUPLICATE KEY: key 列相同的记录，同时存在于 StarRocks 中，适合存储明细数据或者数据无聚合特性的业务场景。
  默认为 DUPLICATE KEY，数据按 key 列做排序。
  除 AGGREGATE KEY 外，其他 key_type 在建表时，value 列不需要指定聚合类型 (agg_type)。

COMMENT

表的注释，可选。注意建表时 COMMENT 必须在 key_desc 之后，否则建表失败。
如果后续想修改表的注释，可以使用 ALTER TABLE COMMENT = "new table comment"（3.1 版本开始支持）。

partition_desc

支持三种分区方式，表达式分区（推荐）、Range 分区 和 List 分区。
使用 Range 分区时，提供三种创建方式，其语法、说明和示例如下：

- 动态创建分区
  动态分区提供了分区生命周期管理（TTL）。StarRocks 会自动提前创建新的分区，并删除过期的分区，以确保数据时效性。要启用这个功能，您可以在创建表时配置与动态分区相关的属性。
- 手动创建分区
  - 仅指定各个分区的上界
    语法：

```
PARTITION BY RANGE ( <partitioning_column1> [, <partitioning_column2>, ... ] )
PARTITION <partition1_name> VALUES LESS THAN ("<upper_bound_for_partitioning_column1>" [ , "<
upper_bound_for_partitioning_column2>", ... ] )
[ ,
PARTITION <partition2_name> VALUES LESS THAN ("<upper_bound_for_partitioning_column1>" [ , "<
upper_bound_for_partitioning_column2>", ... ] )
, ... ]
)
```

说明：
使用指定的 key 列和指定的数值范围进行分区。
3.3.0 之前，仅支持以下类型的列作为 Range 分区列：TINYINT, SMALLINT, INT, BIGINT, LARGEINT, DATE, DATETIME。自 3.3.0 起，支持三个特定时间函数为 Range 分区列。
分区为左闭右开区间，首个分区的左边界为最小值。
NULL 值只会存放在包含 最小值 的分区中。当包含最小值的分区被删除后，NULL 值将无法导入。
可以指定一列或多列作为分区列。如果分区值缺省，则会默认填充最小值。
当只指定一个列作为分区列时，您可以设置最后一个分区的分区列的上界为 MAXVALUE。
注意：

1. 分区一般用于时间维度的数据管理。
2. 有数据回溯需求的，可以考虑首个分区为空分区，以便后续增加分区。
   示例：
   a. 分区列 pay_dt 为 DATE 类型，并且按天分区。

```
PARTITION BY RANGE(pay_dt)
```

```
(
    PARTITION p1 VALUES LESS THAN ("2021-01-02"),
    PARTITION p2 VALUES LESS THAN ("2021-01-03"),
    PARTITION p3 VALUES LESS THAN ("2021-01-04")
)
```

b. 分区列 pay_dt 为 INT 类型，并且按天分区。

```
  PARTITION BY RANGE(pay_dt)
(
    PARTITION p1 VALUES LESS THAN ("20210102"),
    PARTITION p2 VALUES LESS THAN ("20210103"),
    PARTITION p3 VALUES LESS THAN ("20210104")
)
```

c. 分区列 pay_dt 为 INT 类型，并且按天分区，最后一个分区没有上界。

```
PARTITION BY RANGE(pay_dt)
(
    PARTITION p1 VALUES LESS THAN ("20210102"),
    PARTITION p2 VALUES LESS THAN ("20210103"),
    PARTITION p3 VALUES LESS THAN MAXVALUE
)
```

- 指定各个分区的上界和下界

  语法：

```
PARTITION BY RANGE ( <partitioning_column1> [, <partitioning_column2>, ... ] )
(
    PARTITION <partition_name1> VALUES [( "<lower_bound_for_partitioning_column1>" [ , "<lower_bound_for_partitioning_column2>", ... ] ), ( "<upper_bound_for_partitioning_column1>" [ , "<upper_bound_for_partitioning_column2>", ... ] ) )
    [,
    PARTITION <partition_name2> VALUES [( "<lower_bound_for_partitioning_column1>" [ , "<lower_bound_for_partitioning_column2>", ... ] ), ( "<upper_bound_for_partitioning_column1>" [ , "<upper_bound_for_partitioning_column2>", ... ] ) )
    , ...]
)
```

说明：

与仅指定分区下界相比，指定各个分区的上界和下界相对灵活，并且您可以自定义左右区间。

其他与 LESS THAN 保持同步。

当只指定一个列作为分区列时，您可以设置最后一个分区的分区列的上界为 MAXVALUE。

示例：

a. 分区列 pay_dt 为 DATE 类型，并且按月分区。

```
PARTITION BY RANGE (pay_dt)
(
   PARTITION p202101 VALUES [("2021-01-01"), ("2021-02-01")),
   PARTITION p202102 VALUES [("2021-02-01"), ("2021-03-01")),
   PARTITION p202103 VALUES [("2021-03-01"), ("2021-04-01"))
)
```

b. 分区列 pay_dt 为 INT 类型，并且按月分区。

```
PARTITION BY RANGE (pay_dt)
(
   PARTITION p202101 VALUES [("20210101"), ("20210201")),
   PARTITION p202102 VALUES [("20210201"), ("20210301")),
   PARTITION p202103 VALUES [("20210301"), ("20210401"))
)
```

c. 分区列 pay_dt 为 INT 类型，并且按月分区，最后一个分区没有上界。

```
PARTITION BY RANGE (pay_dt)
(
   PARTITION p202101 VALUES [("20210101"), ("20210201")),
   PARTITION p202102 VALUES [("20210201"), ("20210301")),
   PARTITION p202103 VALUES [("20210301"), (MAXVALUE))
)
```

- 批量创建分区

  语法：

  - 如果分区列为时间类型

```
PARTITION BY RANGE (<partitioning_column>) (
   START ("<start_date>") END ("<end_date>") EVERY (INTERVAL <N> <time_unit>)
)
```

- 如果分区列为整数类型

```
PARTITION BY RANGE (<partitioning_column>) (
   START ("<start_integer>") END ("<end_integer>") EVERY (<partitioning_granularity>)
)
```

说明： 用户可以通过给出一个 START 值、一个 END 值以及一个定义分区增量值的 EVERY 子句批量创建分区。

3.3.0 之前，仅支持以下类型的列作为 Range 分区列：TINYINT, SMALLINT, INT, BIGINT, LARGEINT, DATE,

DATETIME。自 3.3.0 起，支持三个特定时间函数为 Range 分区列。
当分区列为日期类型时，需要指定 INTERVAL 关键字来表示日期间隔。目前日期间隔支持 hour (v3.0)、day、week、month、year，分区的命名规则同动态分区一样。
当分区列为整数类型时，START 值、END 值仍需要用双引号包裹。
仅支持指定一列作为分区列。
示例：
a. 分区列 pay_dt 为 DATE 类型，并且按年分区。

```
PARTITION BY RANGE (pay_dt) (
   START ("2018-01-01") END ("2023-01-01") EVERY (INTERVAL 1 YEAR)
)
```

b. 分区列 pay_dt 为 INT 类型，并且按年分区。

```
PARTITION BY RANGE (pay_dt) (
   START ("2018") END ("2023") EVERY (1)
)
```

distribution_desc

支持随机分桶（Random bucketing）和哈希分桶（Hash bucketing）。如果不指定分桶信息，则 StarRocks 默认使用随机分桶且自动设置分桶数量。

- 随机分桶（自 v3.1）
  对每个分区的数据，StarRocks 将数据随机地分布在所有分桶中，而不受到特定列值的影响。并且如果选择由系统设置分桶数量，则您无需设置分桶信息。如果选择手动指定分桶数量，则语法如下：

```
DISTRIBUTED BY RANDOM BUCKETS <num>
```

不过值得注意的是，如果查询海量数据且查询时经常使用一些列会作为条件列，随机分桶提供的查询性能可能不够理想。在该场景下建议您使用哈希分桶，当查询时经常使用这些列作为条件列时，只需要扫描和计算查询命中的少量分桶，则可以显著提高查询性能。
注意事项

- 不支持主键表、更新表和聚合表。
- 不支持指定 Colocation Group。
- 不支持 Spark Load。

- 自 2.5.7 版本起，建表时无需手动指定分桶数量，StarRocks 自动设置分桶数量。

- 哈希分桶
  语法：

```
DISTRIBUTED BY HASH (k1[,k2 ...]) [BUCKETS num]
```

对每个分区的数据，StarRocks 会根据分桶键和分桶数量进行哈希分桶。

对于分桶键的选择，如果列是高基数且经常作为查询条件，则优先选择其为分桶键，进行哈希分桶。 如果不存在同时满足两个条件的列，则需要根据查询进行判断。

- 如果查询比较复杂，则建议选择高基数的列为分桶键，保证数据在各个分桶中尽量均衡，提高集群资源利用率。
- 如果查询比较简单，则建议选择经常作为查询条件的列为分桶键，提高查询效率。 并且，如果数据倾斜情况严重，您还可以使用多个列作为数据的分桶键，但是建议不超过 3 个列。
  注意事项
- 建表时，必须指定分桶键。
- 作为分桶键的列，该列的值不支持更新。
- 分桶键指定后不支持修改。
- 自 2.5.7 版本起，建表时无需手动指定分桶数量，StarRocks 自动设置分桶数量。

ORDER BY

自 3.0 版本起，主键表支持使用 ORDER BY 定义排序键，自 3.3 版本起，明细表、聚合表和更新表支持使用 ORDER BY 定义排序键。

TEMPORARY

创建临时表。从 v3.3.1 版本开始，StarRocks 支持在 Default Catalog 中创建临时表。
备注
创建临时表时，必须将 ENGINE 设置为 olap。

PROPERTIES

设置数据的初始存储介质、自动降冷时间和副本数。
如果 ENGINE 类型为 OLAP，可以在属性 properties 中设置该表数据的初始存储介质（storage_medium）、自动降冷时间（storage_cooldown_time）或者时间间隔（storage_cooldown_ttl）和副本数（replication_num）。

属性生效范围：当表为单分区表时，以上属性为表的属性。当表划分成多个分区时，以上属性属于每一个分区。并且如果希望不同分区有不同属性，则建表后可以执行 ALTER TABLE ... ADD PARTITION 或 ALTER TABLE ... MODIFY PARTITION。

设置数据的初始存储介质、自动降冷时间。

```
PROPERTIES (
    "storage_medium" = "[SSD|HDD]",
    { "storage_cooldown_ttl" = "<num> { YEAR | MONTH | DAY | HOUR } "
    | "storage_cooldown_time" = "yyyy-MM-dd HH:mm:ss" }
)
```

storage_medium：数据初始存储介质，取值为 SSD 或 HDD。显式指定该参数时，请确保该值与 BE 静态参数 storage_root_path 中指定的集群存储介质相匹配。
当 FE 配置项 enable_strict_storage_medium_check 为 true 时，表示在建表时会严格校验 BE 上的存储介质。如果建表语句中的存储介质和 BE 的存储类型不一致，建表语句会报错 Failed to find enough hosts with storage medium [SSD|HDD] at all backends...。当 enable_strict_storage_medium_check 为 false 时，可以忽略该报错强行建表，但是后续可能会导致集群磁盘空间分布出现不均衡。
从 2.3.6，2.4.2，2.5.1，3.0 及以上版本开始，支持在未显式指定该参数的情况下，由系统自动推导存储介质。

- 在以下场景中，系统推导该参数为 SSD：
  - BE 上报的存储路径 (storage_root_path) 都是 SSD。
  - BE 上报的存储路径包含 SSD 和 HDD。并且从 2.3.10，2.4.5，2.5.4，3.0 及以上版本开始，如果 BE 上报的存储路径两者都有且 FE 配置文件中设置了 storage_cooldown_second，将会自动启用冷热数据分层存储策略。
- 在以下场景中，系统推导该参数为 HDD：
  - BE 上报的存储路径都是 HDD。
  - 从 2.3.10，2.4.5，2.5.4，3.0 及以上版本开始，如果 BE 上报的存储路径两者都有且 FE 配置文件中未设置 storage_cooldown_second，将不会启用自动的冷热数据分层迁移功能。

storage_cooldown_ttl 或 storage_cooldown_time：数据自动降冷的时间间隔或者时间点。数据自动降冷，即数据自动从 SSD 介质迁移到 HDD 介质。只在数据初始存储介质为 SSD 时生效。

参数说明：
storage_cooldown_ttl：该表分区自动降冷时间间隔。如果您需要保留最近几个分区在 SSD，其它较早的分区经过一定时间间隔自动降冷至 HDD，则您可以使用该参数，各个分区的自动降冷时间点为该参数值 + 该分区的时间上界。取值为 YEAR， MONTH， DAY 或 HOUR。 为非负整数。默认值为空，表示该表分区不进行自动降冷。

例如建表时指定 "storage_cooldown_ttl"="1 DAY"，建表后存在分区 p20230801 ，其范围为 [2023-08-01 00:00:00,2023-08-02 00:00:00)，则该分区的自动降冷时间点是 2023-08-03 00:00:00，即 2023-08-02 00:00:00 + 1 DAY。如果建表时指定 "storage_cooldown_ttl"="0 DAY"，则该分区自动降冷时间点是 2023-08-02 00:00:00。

storage_cooldown_time：该表自动降冷时间点（绝对时间）。数据在该时间点之后从 SSD 自动降冷到 HDD，设置的时间必须大于当前时间。取值格式为："yyyy-MM-dd HH:mm:ss"。如果需要不同分区具有不同自动降冷时间点，则需要执行 ALTER TABLE ... ADD PARTITION 或 ALTER TABLE ... MODIFY PARTITION 手动指定。

使用说明

- 目前 StarRocks 提供如下数据自动降冷的相关参数，对比如下：
  storage_cooldown_ttl：表的属性，指定该表中分区自动降冷时间间隔，由系统自动降冷表中到达时间点（时间间隔+分区时间上界）的分区。并且表按照分区粒度自动降冷，更加灵活。
  storage_cooldown_time：表的属性，指定该表的自动降冷时间点（绝对时间）。建表后也可以为不同分区配置不同时间点。
  storage_cooldown_second：FE 静态参数，指定集群范围内所有表的自动降冷时延。
- 表属性 storage_cooldown_ttl 或 storage_cooldown_time 比 FE 静态参数 storage_cooldown_second 优先级高。
- 配置以上参数时，必须指定"storage_medium" = "SSD"。
- 不配置以上参数时，则不进行自动降冷。
- 执行 SHOW PARTITIONS FROM 查看各个分区的自动冷却时间点。

限制

- 不支持表达式分区和 List 分区。
- 不支持分区列为非日期类型。
- 不支持多个分区列。
- 不支持主键表。

设置分区 Tablet 副本数
replication_num：分区 Tablet 副本数。默认为 3。

```
PROPERTIES (
    "replication_num" = "<num>"
)
```

创建表时为列添加 Bloom filter 索引
如果 Engine 类型为 olap, 可以指定某列使用 Bloom filter 索引。Bloom filter 索引使用时有如下限制：

- 主键表和明细表中所有列都可以创建 Bloom filter 索引；聚合表和更新表中，只有维度列（即 Key 列）支持创建 Bloom filter 索引。
- 不支持为 TINYINT、FLOAT、DOUBLE 和 DECIMAL 类型的列创建 Bloom filter 索引。
- Bloom filter 索引只能提高查询条件为 in 和 = 的查询效率，值越分散效果越好。

```
PROPERTIES (
    "bloom_filter_columns" = "k1, k2, k3"
```

```
)
```

添加属性支持 Colocate Join
如果希望使用 Colocate Join 特性，需要在 properties 中指定：

```
PROPERTIES (
    "colocate_with" = "table1"
)
```

详细的 Colocate Join 使用方法及应用场景请参考 Colocate Join 章节。

设置动态分区
如果希望使用动态分区特性，需要在 properties 中指定如下参数：

```
PROPERTIES (
    "dynamic_partition.enable" = "true|false",
    "dynamic_partition.time_unit" = "DAY|WEEK|MONTH",
    "dynamic_partition.start" = "${integer_value}",
    "dynamic_partition.end" = "${integer_value}",
    "dynamic_partition.prefix" = "${string_value}",
    "dynamic_partition.buckets" = "${integer_value}"
)
```

参数	是否必填	说明
dynamic_partition.enable	否	开启动态分区特性，取值为 TRUE（默认）或 FALSE。
dynamic_partition.time_unit	是	动态分区的时间粒度，取值为 DAY、WEEK 或 MONTH。时间粒度会决定动态创建的分区名后缀格式。 取值为 DAY 时，动态创建的分区名后缀格式为 yyyyMMdd，例如 20200321。 取值为 WEEK 时，动态创建的分区名后缀格式为 yyyy_ww，例如 2020_13 代表 2020 年第 13 周。 取值为 MONTH 时，动态创建的分区名后缀格式为 yyyyMM，例如 202003。
dynamic_partition.start：	否	保留的动态分区的起始偏移，取值范围为负整数。根据 dynamic_partition.time_unit 属性的不同，以当天（周/月）为基准，分区范围在此偏移之前的分区将会被删除。比如设置为-3，并且 dynamic_partition.time_unit为DAY，则表示 3 天前的分区会被删掉。 如果不填写，则默认为 Integer.MIN_VALUE，即 -2147483648，表示不删除历史分区。

dynamic_partition.end	是	提前创建的分区数量，取值范围为正整数。根据 dynamic_partition.time_unit 属性的不同，以当天（周/月）为基准，提前创建对应范围的分区。
dynamic_partition.prefix	否	动态分区的前缀名，默认值为 p。
dynamic_partition.buckets	否	动态分区的分桶数量。默认与 BUCKETS 保留字指定的分桶数量、或者 StarRocks 自动设置的分桶数量保持一致。

设置随机分桶表中分桶大小 (bucket_size)
自 3.2 版本起，对于随机分桶的表，您可以在建表时在 PROPERTIES 中设置 bucket_size 参数来指定分桶大小，启用按需动态增加分桶数量。单位为 B。

```
PROPERTIES (
    "bucket_size" = "1073741824"
)
```

设置数据压缩算法
您可以在建表时通过增加属性 compression 为该表指定数据压缩算法。
compression 有效值包括：

- LZ4：LZ4 算法。
- ZSTD：Zstandard 算法。
- ZLIB：zlib 算法。
- SNAPPY：Snappy 算法。
  如不指定数据压缩算法，StarRocks 默认使用 LZ4。
  自 v3.3.2 起，StarRocks 支持在建表时指定 ZSTD 压缩格式的压缩级别。
  语法：

```
PROPERTIES ("compression" = "zstd(<compression_level>)")
```

compression_level：ZSTD 压缩格式的压缩级别。类型：Integer。范围：[1,22]。默认值：3（推荐）。数字越大，压缩率越高。压缩级别越高，压缩和解压的耗时越长。
示例：

```
PROPERTIES ("compression" = "zstd(3)")
```

设置数据导入安全等级
如果您的 StarRocks 集群有多数据副本，可以在建表时在 PROPERTIES 中设置 write_quorum 参数来指定数据导入安全等级，即设置需要多少数据副本导入成功后 StarRocks 可返回导入成功。该属性从 2.5 版本开始支持。
write_quorum 的取值及其对应描述如下：

- MAJORITY：默认值。当多数数据副本导入成功时，StarRocks 返回导入成功，否则返回失败。
- ONE：当一个数据副本导入成功时，StarRocks 返回导入成功，否则返回失败。
- ALL：当所有数据副本导入成功时，StarRocks 返回导入成功，否则返回失败。
  注意
- 设置较低的导入数据安全等级会增加数据不可访问甚至丢失的风险。例如，在 StarRocks 集群有两个数据副本的情况下设置 write_quorum 为 ONE，如果某个 Tablet 实际只成功导入了一个副本，而此副本所在机器后续下线，则会导致该 Tablet 中的数据因为没有存活副本而无法访问。如果服务器磁盘受损，则会导致该 Tablet 中的数据丢失。
- 仅当所有数据副本返回导入状态后，StarRocks 才会返回导入任务状态。当有副本导入状态未知时，StarRocks 不会返回导入任务状态。导入超时的副本亦会被标记为失败。

指定数据在多副本间的写入和同步方式
如果您的 StarRocks 集群有多数据副本，可以在建表时在 PROPERTIES 中设置 replicated_storage 参数来指定数据在多副本间的写入和同步方式。

- 设置为 true（3.0 及后续版本的默认值）表示 single leader replication，即数据只写入到主副本 (primary replica)，由主副本同步数据到从副本 (secondary replica)。该模式能有效降低多副本写入带来的 CPU 成本。该模式从 2.5 版本开始支持。
- 设置为 false（2.5 版本的默认值）表示 leaderless replication，即数据直接写入到多个副本，不区分主从副本。该模式 CPU 成本比较高。
  默认配置在绝大部分场景下能获得更好的写入性能，如果要修改已有表的多副本写入和同步方式，可执行 ALTER TABLE 命令，举例：

```
ALTER TABLE example_db.my_table
SET ("replicated_storage" = "true");
```

建表时批量创建多个 Rollup
语法：

```
ROLLUP (rollup_name (column_name1, column_name2, ...)
[FROM from_index_name]
[PROPERTIES ("key" = "value", ...)],...)
```

为 View Delta Join 查询改写定义 Unique Key 和外键约束
要在 View Delta Join 场景中启用查询重写，您必须为 Delta Join 中的表定义 Unique Key 约束 unique_constraints 和外键约束 foreign_key_constraints。详细信息，请参阅 异步物化视图 - 基于 View Delta Join 场景改写查询。

```
PROPERTIES (
    "unique_constraints" = "<unique_key>[, ...]",
    "foreign_key_constraints" = "
```

```
    (<child_column>[, ...])
    REFERENCES
    [catalog_name].[database_name].<parent_table_name>(<parent_column>[, ...])
    [;...]
    "
)
```

child_column：当前表中的外键列。 您可以定义多个 child_column。
catalog_name：待 Join 表所在的数据目录名。未指定此参数时使用默认目录。
database_name：待 Join 表所在的数据库名。未指定此参数时使用当前数据库。
parent_table_name：待 Join 表名。
parent_column：待 Join 列名，必须为相应表的 Primary Key 或 Unique Key。

注意：

- unique_constraints 约束和 foreign_key_constraints 约束仅用于查询重写。导入数据时，不保证进行外键约束校验。您必须确保导入的数据满足约束条件。
- 主键表的 Primary Key 或更新表的 Unique Key 默认是其 unique_constraints，您无需手动设置。
- foreign_key_constraints 中的 child_column 必须对应另一个表的 unique_constraints 中的 unique_key。
- child_column 和 parent_column 的数量必须一致。
- child_column 和对应的 parent_column 的数据类型必须匹配。

为 StarRocks 存算分离集群创建云原生表
为了使用 StarRocks 存算分离集群，您需要通过以下 PROPERTIES 创建云原生表：

```
PROPERTIES (
    "storage_volume" = "<storage_volume_name>",
    "datacache.enable" = "{ true | false }",
    "datacache.partition_duration" = "<string_value>"
)
```

storage_volume：建表使用的 Storage Volume 名称。该属性自 v3.1 版本起支持。如果未指定该属性，则使用默认 Storage Volume。示例："storage_volume" = "def_volume"。

datacache.enable：是否启用本地磁盘缓存。默认值：true。
当该属性设置为 true 时，数据会同时导入对象存储（或 HDFS）和本地磁盘（作为查询加速的缓存）。
当该属性设置为 false 时，数据仅导入到对象存储中。
说明：如需启用本地磁盘缓存，必须在 BE 配置项 storage_root_path 中指定磁盘目录。

datacache.partition_duration：热数据的有效期。当启用本地磁盘缓存时，所有数据都会导入至本地磁盘缓存中。当缓存满时，StarRocks 会从缓存中删除最近较少使用（Least recently used）的数据。当有查询需要扫描已删除的数据时，StarRocks 会检查该数据是否在有效期内。如果数据在有效期内，StarRocks 会再次将数据导入至缓存中。如果数据不在有效期内，StarRocks 不会将其导入至缓存中。该属性为字符串，您可以使用以下单位指定：YEAR、MONTH、DAY 和 HOUR，例如，7 DAY 和 12 HOUR。如果不指定，StarRocks 将所有数据都作为热数据进行缓

存。
说明：仅当 datacache.enable 设置为 true 时，此属性可用。

设置 fast schema evolution
fast_schema_evolution: 是否开启该表的 fast schema evolution，取值：TRUE 或 FALSE（默认）。开启后增删列时可以提高 schema change 速度并降低资源使用。目前仅支持在建表时开启该属性，建表后不支持通过 ALTER TABLE 修改该属性。
说明：
仅 StarRocks 存算一体集群支持该参数，自 v3.2.0 版本起。
如果您需要为存算分离表开启 fast schema evolution，则必须在集群范围内启用。需要通过 FE 动态参数 enable_fast_schema_evolution 设置。

禁止 Base Compaction
base_compaction_forbidden_time_ranges：禁止对表进行 Base Compaction 的时间范围。设置该属性后，系统将仅在指定时间范围之外对符合条件的 Tablet 执行 Base Compaction。该属性 v3.2.13 起支持。
说明：请确保在禁止 Base Compaction 期间，对于该表的导入次数不超过 500 次。
base_compaction_forbidden_time_ranges 的值遵循 Quartz cron 语法，且只支持以下字段： ，其中 必须为 *。

```
crontab_param_value ::= [ "" | crontab ]
crontab ::= * <hour> <day-of-the-month> <month> <day-of-the-week>
```

如果未设置此属性或设置为 ""（空字符串），则在任何时候都不禁止 Base Compaction。
当此属性设置为 * * * * * 时，Base Compaction 始终被禁止。
其他值遵循 Quartz cron 语法。

- 独立数值表示字段的单位时间。例如， 字段中的 8 表示 8:00-8:59。
- 数值范围表示字段的时间范围。例如， 字段中的 8-9 表示 8:00-9:59。
- 用逗号分隔的多个数值范围表示字段的多个时间范围。
- 的起始值为 1 表示周日，7 表示周六。
  示例：

```
-- 每天 8AM ~ 9PM 禁止执行 Base Compaction。
'base_compaction_forbidden_time_ranges' = '* 8-20 * * *'

-- 每天 0-5，21-23 点禁止执行 Base Compaction。
'base_compaction_forbidden_time_ranges' = '* 0-4,21-22 * * *'

-- 每周一到周五禁止执行 Base Compaction（也即是在每周六/周日的全天允许执行）。
'base_compaction_forbidden_time_ranges' = '* * * * 2-6'

-- 每个工作日（周一到周五）的 8AM ~ 9PM 禁止执行 Base Compaction。
'base_compaction_forbidden_time_ranges' = '* 8-20 * * 2-6'
```

示例

">创建 Hash 分桶表并根据 key 列对数据进行聚合

创建一个 olap 表，使用 Hash 分桶，使用列存，相同 key 的记录进行聚合。

```
CREATE TABLE example_db.table_hash
(
    k1 TINYINT,
    k2 DECIMAL(10, 2) DEFAULT "10.5",
    v1 CHAR(10) REPLACE,
    v2 INT SUM
)
ENGINE = olap
AGGREGATE KEY(k1, k2)
COMMENT "my first starrocks table"
DISTRIBUTED BY HASH(k1)
PROPERTIES ("storage_type" = "column");
```

">创建表并设置存储介质和数据自动冷却时间

创建一个 olap 表，使用 Hash 分桶，使用列存，相同 key 的记录进行覆盖，设置初始存储介质和冷却时间。

```
CREATE TABLE example_db.table_hash
(
    k1 BIGINT,
    k2 LARGEINT,
    v1 VARCHAR(2048) REPLACE,
    v2 SMALLINT DEFAULT "10"
)
ENGINE = olap
UNIQUE KEY(k1, k2)
DISTRIBUTED BY HASH (k1, k2)
PROPERTIES(
    "storage_type" = "column",
    "storage_medium" = "SSD",
    "storage_cooldown_time" = "2025-06-04 00:00:00"
);
```

或

```
CREATE TABLE example_db.table_hash
(
    k1 BIGINT,
    k2 LARGEINT,
    v1 VARCHAR(2048) REPLACE,
    v2 SMALLINT DEFAULT "10"
)
ENGINE = olap
PRIMARY KEY(k1, k2)
DISTRIBUTED BY HASH (k1, k2)
PROPERTIES(
    "storage_type" = "column",
    "storage_medium" = "SSD",
    "storage_cooldown_time" = "2025-06-04 00:00:00"
);
```

">创建分区表

创建一个 olap 表，使用 Range 分区，使用 Hash 分桶，默认使用列存，相同 key 的记录同时存在，设置初始存储介质和冷却时间。
使用 LESS THAN 方式按照范围划分分区：

```
CREATE TABLE example_db.table_range
(
    k1 DATE,
    k2 INT,
    k3 SMALLINT,
    v1 VARCHAR(2048),
    v2 DATETIME DEFAULT "2014-02-04 15:36:00"
)
ENGINE = olap
DUPLICATE KEY(k1, k2, k3)
PARTITION BY RANGE (k1)
(
    PARTITION p1 VALUES LESS THAN ("2014-01-01"),
    PARTITION p2 VALUES LESS THAN ("2014-06-01"),
    PARTITION p3 VALUES LESS THAN ("2014-12-01")
)
DISTRIBUTED BY HASH(k2)
PROPERTIES(
    "storage_medium" = "SSD",
    "storage_cooldown_time" = "2025-06-04 00:00:00"
);
```

说明： 这个语句会将数据划分成如下 3 个分区：

```
( {    MIN     },   {"2014-01-01"} )
[ {"2014-01-01"},   {"2014-06-01"} )
[ {"2014-06-01"},   {"2014-12-01"} )
```

不在这些分区范围内的数据将视为非法数据被过滤。

使用 Fixed Range 方式创建分区：

```
CREATE TABLE table_range
(
    k1 DATE,
    k2 INT,
    k3 SMALLINT,
    v1 VARCHAR(2048),
    v2 DATETIME DEFAULT "2014-02-04 15:36:00"
)
ENGINE = olap
DUPLICATE KEY(k1, k2, k3)
PARTITION BY RANGE (k1, k2, k3)
(
    PARTITION p1 VALUES [("2014-01-01", "10", "200"), ("2014-01-01", "20", "300")),
    PARTITION p2 VALUES [("2014-06-01", "100", "200"), ("2014-07-01", "100", "300"))
```

```
)
DISTRIBUTED BY HASH(k2)
PROPERTIES(
    "storage_medium" = "SSD"
);
```

创建一个 MySQL 外表

```
CREATE EXTERNAL TABLE example_db.table_mysql
(
    k1 DATE,
    k2 INT,
    k3 SMALLINT,
    k4 VARCHAR(2048),
    k5 DATETIME
)
ENGINE = mysql
PROPERTIES
(
    "host" = "127.0.0.1",
    "port" = "8239",
    "user" = "mysql_user",
    "password" = "mysql_passwd",
    "database" = "mysql_db_test",
    "table" = "mysql_table_test"
);
```

创建一张含有 HLL 列的表

```
CREATE TABLE example_db.example_table
(
    k1 TINYINT,
    k2 DECIMAL(10, 2) DEFAULT "10.5",
    v1 HLL HLL_UNION,
    v2 HLL HLL_UNION
)
ENGINE = olap
AGGREGATE KEY(k1, k2)
DISTRIBUTED BY HASH(k1)
PROPERTIES ("storage_type" = "column");
```

创建一张含有 BITMAP_UNION 聚合类型的表

v1 和 v2 列的原始数据类型必须是 TINYINT, SMALLINT, INT。

```
CREATE TABLE example_db.example_table
(
    k1 TINYINT,
    k2 DECIMAL(10, 2) DEFAULT "10.5",
    v1 BITMAP BITMAP_UNION,
    v2 BITMAP BITMAP_UNION
)
ENGINE = olap
AGGREGATE KEY(k1, k2)
DISTRIBUTED BY HASH(k1)
PROPERTIES ("storage_type" = "column");
```

创建两张支持 Colocate Join 的表

创建 t1 和 t2 两个表，两表可进行 Colocate Join。两表属性中的 colocate_with 属性的值需保持一致。

```
CREATE TABLE `t1` (
    `id` int(11) COMMENT "",
    `value` varchar(8) COMMENT ""
) ENGINE = OLAP
DUPLICATE KEY(`id`)
DISTRIBUTED BY HASH(`id`)
PROPERTIES (
    "colocate_with" = "t1"
);

CREATE TABLE `t2` (
    `id` int(11) COMMENT "",
    `value` varchar(8) COMMENT ""
) ENGINE = OLAP
DUPLICATE KEY(`id`)
DISTRIBUTED BY HASH(`id`)
PROPERTIES (
    "colocate_with" = "t1"
);
```

创建一个带有 bitmap 索引的表

```
CREATE TABLE example_db.table_hash
(
    k1 TINYINT,
    k2 DECIMAL(10, 2) DEFAULT "10.5",
    v1 CHAR(10) REPLACE,
    v2 INT SUM,
    INDEX k1_idx (k1) USING BITMAP COMMENT 'xxxxxx'
)
ENGINE = olap
AGGREGATE KEY(k1, k2)
COMMENT "my first starrocks table"
DISTRIBUTED BY HASH(k1)
PROPERTIES ("storage_type" = "column");
```

">创建动态分区表

创建动态分区表需要在 FE 配置中开启 动态分区 功能（"dynamic_partition.enable" = "true"），参数可参见本文设置动态分区。

该表每天提前创建 3 天的分区，并删除 3 天前的分区。例如今天为 2020-01-08，则会创建分区名为 p20200108，p20200109，p20200110，p20200111 的分区. 分区范围分别为:

```
[types: [DATE]; keys: [2020-01-08]; ..types: [DATE]; keys: [2020-01-09]; )
[types: [DATE]; keys: [2020-01-09]; ..types: [DATE]; keys: [2020-01-10]; )
[types: [DATE]; keys: [2020-01-10]; ..types: [DATE]; keys: [2020-01-11]; )
[types: [DATE]; keys: [2020-01-11]; ..types: [DATE]; keys: [2020-01-12]; )
```

```
CREATE TABLE example_db.dynamic_partition
(
    k1 DATE,
    k2 INT,
    k3 SMALLINT,
    v1 VARCHAR(2048),
    v2 DATETIME DEFAULT "2014-02-04 15:36:00"
)
ENGINE = olap
DUPLICATE KEY(k1, k2, k3)
PARTITION BY RANGE (k1)
(
    PARTITION p1 VALUES LESS THAN ("2014-01-01"),
    PARTITION p2 VALUES LESS THAN ("2014-06-01"),
    PARTITION p3 VALUES LESS THAN ("2014-12-01")
)
```

```
DISTRIBUTED BY HASH(k2)
PROPERTIES(
    "storage_medium" = "SSD",
    "dynamic_partition.time_unit" = "DAY",
    "dynamic_partition.start" = "-3",
    "dynamic_partition.end" = "3",
    "dynamic_partition.prefix" = "p"
);
```

创建一个 Hive 外部表

```
CREATE EXTERNAL TABLE example_db.table_hive
(
    k1 TINYINT,
    k2 VARCHAR(50),
    v INT
)
ENGINE = hive
PROPERTIES
(
    "resource" = "hive0",
    "database" = "hive_db_name",
    "table" = "hive_table_name"
);
```

创建一张主键表并且指定排序键

假设需要按地域、最近活跃时间实时分析用户情况，则可以将表示用户 ID 的 user_id 列作为主键，表示地域的 address 列和表示最近活跃时间的 last_active 列作为排序键。建表语句如下：

```
create table users (
    user_id bigint NOT NULL,
    name string NOT NULL,
    email string NULL,
    address string NULL,
    age tinyint NULL,
    sex tinyint NULL,
    last_active datetime,
    property0 tinyint NOT NULL,
    property1 tinyint NOT NULL,
    property2 tinyint NOT NULL,
```

```
    property3 tinyint NOT NULL
)
PRIMARY KEY (`user_id`)
DISTRIBUTED BY HASH(`user_id`)
ORDER BY(`address`,`last_active`)
PROPERTIES(
    "replication_num" = "3",
    "enable_persistent_index" = "true"
);
```

DELETE

该语句用于从表中删除数据行。表可以是分区表或者非分区表。

对于明细表、聚合表，以及更新表，支持删除表中指定分区的数据。从 2.3 版本开始，主键表支持完整的 DELETE...WHERE 语义，即支持按主键、任意列、以及子查询结果删除数据。从 3.0 版本开始，主键类型表丰富了 DELETE...WHERE 语义，支持使用多表关联和公用表表达式（CTE）。

注意事项

执行 DELETE 操作需要有对应表的 DELETE 权限。
不建议您执行高频的 DELETE 操作。如需要，请在业务低峰期进行。
DELETE 删除的是表中数据，表依然存在。如果要删除表，参见 DROP TABLE。
为防止误删整表，必须在 DELETE 语句中指定 WHERE 子句。
删除的数据会标记为“Deleted”，暂时保留在 Segment 中，不会立即进行物理删除。Compaction（数据版本合并）完成之后会被回收。
该操作会同时删除和表相关的物化视图的数据。

DELETE 与明细类型、聚合类型和更新类型表

语法

```
DELETE FROM [<db_name>.]<table_name> [PARTITION <partition_name>]
WHERE
column_name1 op { value | value_list } [ AND column_name2 op { value | value_list } ...]
```

参数说明

参数	必选	描述

参数	必选	描述
db_name	No	要操作的表所在的数据库。如果不指定，默认为当前数据库。
table_name	Yes	要操作的表名。
partition_name	No	要操作的分区名。
column_name	Yes	作为删除条件的列，可以指定一个或多个列。
op	Yes	指定的算子。支持 =、>、<、>=、<=、!=、IN 和 NOT IN。

使用限制

- 聚合类型和更新类型表仅支持 Key 列作为删除条件。明细类型表支持任意列作为删除条件。
- 条件之间只能是“AND”关系。若希望达成“OR”的关系，需要将条件分别写在两个 DELETE 语句中。
- 明细类型、聚合类型和更新表类型下，DELETE 语句目前不支持以子查询结果作为删除条件。

影响

执行 DELETE 语句后，可能会导致接下来一段时间内（Compaction 完成之前）的查询效率降低。影响程度取决于语句中指定的删除条件的数量。指定的条件越多，影响越大。

示例

创建表并插入数据
以下示例创建一张明细类型的分区表。

```
CREATE TABLE `my_table` (
    `date` date NOT NULL,
    `k1` int(11) NOT NULL COMMENT "",
    `k2` varchar(65533) NULL DEFAULT "" COMMENT "")
DUPLICATE KEY(`date`)
PARTITION BY RANGE(`date`)
(
    PARTITION p1 VALUES [('2022-03-11'), ('2022-03-12')),
    PARTITION p2 VALUES [('2022-03-12'), ('2022-03-13'))
```

```
)
DISTRIBUTED BY HASH(`date`)
PROPERTIES
("replication_num" = "3");

INSERT INTO `my_table` VALUES
('2022-03-11', 3, 'abc'),
('2022-03-11', 2, 'acb'),
('2022-03-11', 4, 'abc'),
('2022-03-12', 2, 'bca'),
('2022-03-12', 4, 'cba'),
('2022-03-12', 5, 'cba');
```

查询表数据

```
select * from my_table order by date;
+------------+------+------+
| date       | k1   | k2   |
+------------+------+------+
| 2022-03-11 |    3 | abc  |
| 2022-03-11 |    2 | acb  |
| 2022-03-11 |    4 | abc  |
| 2022-03-12 |    2 | bca  |
| 2022-03-12 |    4 | cba  |
| 2022-03-12 |    5 | cba  |
+------------+------+------+
```

删除数据

删除指定分区中的行

删除 my_table 表 p1 分区中 k1 列值为 3 的数据行。

```
DELETE FROM my_table PARTITION p1
WHERE k1 = 3;

-- 可以看到 p1 分区中 k1 为 3 的那行数据被删除。
select * from my_table partition (p1);
+------------+------+------+
| date       | k1   | k2   |
+------------+------+------+
| 2022-03-11 |    2 | acb  |
| 2022-03-11 |    4 | abc  |
+------------+------+------+
```

删除指定分区中满足 AND 条件的行

删除 my_table 表 p1 分区中 k1 列值大于等于 3 且 k2 列值为 "abc" 的数据行。

```
DELETE FROM my_table PARTITION p1
WHERE k1 >= 3 AND k2 = "abc";

select * from my_table partition (p1);
+------------+------+------+
| date       | k1   | k2   |
+------------+------+------+
| 2022-03-11 |    2 | acb  |
+------------+------+------+
```

删除所有分区中满足条件的行
删除 my_table 表所有分区中 k2 列值为 "abc" 或 "cba" 的数据行。

```
DELETE FROM my_table
WHERE k2 in ("abc", "cba");

select * from my_table order by date;
+------------+------+------+
| date       | k1   | k2   |
+------------+------+------+
| 2022-03-11 |    2 | acb  |
| 2022-03-12 |    2 | bca  |
+------------+------+------+
```

DELETE 与主键类型表

从 2.3 版本开始，主键类型表支持完整的 DELETE...WHERE 语义。支持按主键、任意列、以及子查询结果删除数据。从 3.0 版本开始，StarRocks 丰富了 DELETE...WHERE 语义，支持使用多表关联和公用表表达式（CTE）。如果需要将待操作的表与数据库中其他表关联，则可以在 USING 子句或 CTE 中引用其他的表。

语法

```
[ WITH <with_query> [, ...] ]
DELETE FROM <table_name>
[ USING <from_item> [, ...] ]
[ WHERE <where_condition> ]
```

参数说明

参数	是否必选	描述
with_query	No	一个或多个可以在 DELETE 语句中通过名字引用的 CTE。CTE 是一个临时结果集，可以提高复杂语句的易读性。
table_name	Yes	要操作的表名。
from_item	No	引用数据库中一个或者多个其他的表。该表与待操作的表进行连接，WHERE 子句指定连接条件，最终 StarRocks 基于连接查询的结果集删除待操作表中的匹配行。 例如 USING 子句为 USING t1 WHERE t0.pk = t1.pk;，StarRocks 实际执行 DELETE 语句时会将该 USING 子句的表表达式转换为 t0 JOIN t1 ON t0.pk=t1.pk;。
where_condition	No	只有满足 WHERE 条件的行才会被删除。该参数为必选，防止误删整张表。如需删除表中的所有行，请使用 WHERE true。

注意事项

- 主键类型表目前还不支持删除指定分区中的数据，例如 DELETE FROM PARTITION WHERE 。
- 支持如下比较运算符：=、>、<、>=、<=、!=、IN 和 NOT IN。
- 支持如下逻辑运算符：AND 和 OR。
- 不支持并发删除或导入数据时删除，因为可能无法保证导入的事务性。

">示例

创建表并插入数据

在数据库 test 中创建一张名为 score_board 的主键类型表：

```
CREATE TABLE `score_board` (
    `id` int(11) NOT NULL COMMENT "",
    `name` varchar(65533) NULL DEFAULT "" COMMENT "",
    `score` int(11) NOT NULL DEFAULT "0" COMMENT "")
ENGINE=OLAP
PRIMARY KEY(`id`)
COMMENT "OLAP"
DISTRIBUTED BY HASH(`id`)
PROPERTIES
(
    "replication_num" = "3",
    "storage_format" = "DEFAULT",
    "enable_persistent_index" = "false"
```

```
);

INSERT INTO score_board VALUES
(0, 'Jack', 21),
(1, 'Bob', 21),
(2, 'Stan', 21),
(3, 'Sam', 22);
```

查询表数据

```
select * from score_board;
+------+------+------+
| id   | name | score|
+------+------+------+
|    0 | Jack |   21 |
|    1 | Bob  |   21 |
|    2 | Stan |   21 |
|    3 | Sam  |   22 |
+------+------+------+
4 rows in set (0.00 sec)
```

删除数据

按主键删除数据

通过指定主键，可以避免全表扫描。

删除 score_board 表中 id 列值为 0 的数据行。

```
DELETE FROM score_board WHERE id = 0;

select * from score_board;
+------+------+------+
| id   | name | score|
+------+------+------+
|    1 | Bob  |   21 |
|    2 | Stan |   21 |
|    3 | Sam  |   22 |
+------+------+------+
3 rows in set (0.01 sec)
```

按条件删除数据

条件中的列，可以为任意列。

示例一：删除 score_board 表中 score 列值等于 22 的所有数据行。

```
DELETE FROM score_board WHERE score = 22;

select * from score_board;
```

```
+------+------+------+
| id   | name | score|
+------+------+------+
|    0 | Jack |   21 |
|    1 | Bob  |   21 |
|    2 | Stan |   21 |
+------+------+------+
3 rows in set (0.01 sec)
```

示例二：删除 score_board 表中 score 列值小于 22 的所有数据行。

```
DELETE FROM score_board WHERE score < 22;

select * from score_board;
+------+------+------+
| id   | name | score|
+------+------+------+
|    3 | Sam  |   22 |
+------+------+------+
1 row in set (0.01 sec)
```

示例三：删除 score_board 表中 score 列值小于 22、且 name 列值不为 Bob 的所有数据行。

```
DELETE FROM score_board WHERE score < 22 and name != "Bob";

select * from score_board;
+------+------+------+
| id   | name | score|
+------+------+------+
|    1 | Bob  |   21 |
|    3 | Sam  |   22 |
+------+------+------+
2 rows in set (0.00 sec)
```

按子查询结果删除数据

可以在 DELETE 语句中嵌套一个或多个子查询，并使用子查询结果作为删除条件。

开始删除操作之前，先在数据库 test 中再创建一张名为 users 的主键类型表：

```
CREATE TABLE
    `users` (`uid` int(11) NOT NULL COMMENT "",
    `name` varchar(65533) NOT NULL COMMENT "",
    `country` varchar(65533) NULL COMMENT "")
ENGINE=OLAP
PRIMARY KEY(`uid`)COMMENT "OLAP"
DISTRIBUTED BY HASH(`uid`)
PROPERTIES
```

```
(
    "replication_num" = "3",
    "storage_format" = "DEFAULT",
    "enable_persistent_index" = "false"
);
```

向 users 表中插入数据：

```
INSERT INTO users VALUES
(0, "Jack", "France"),
(2, "Stan", "USA"),
(1, "Bob", "France"),
(3, "Sam", "USA");
select * from users;
+------+------+---------+
| uid  | name | country |
+------+------+---------+
|    0 | Jack | France  |
|    1 | Bob  | France  |
|    2 | Stan | USA     |
|    3 | Sam  | USA     |
+------+------+---------+
4 rows in set (0.00 sec)
```

查询 users 表中 country 列值为 France 的数据行，然后删除 score_board 表中与 users 表中查询到的数据行具有相同 name 列值的所有数据行。有两种实现方法：

- 方法 1

```
DELETE FROM score_board
WHERE name IN (select name from users where country = "France");
select * from score_board;
+------+------+------+
| id   | name | score|
+------+------+------+
|    2 | Stan |   21 |
|    3 | Sam  |   22 |
+------+------+------+
2 rows in set (0.00 sec)
```

- 方法 2

```
DELETE FROM score_board
WHERE EXISTS (select name from users
            where score_board.name = users.name and country = "France");
```

```
select * from score_board;
+------+------+-------+
| id   | name | score |
+------+------+-------+
|    2 | Stan |    21 |
|    3 | Sam  |    22 |
+------+------+-------+
2 rows in set (0.00 sec)
```

按多表关联或者 CTE 的结果集删除数据

删除制片人 foo 制作的所有电影，则可以执行以下语句：

```
DELETE FROM films USING producers
WHERE producer_id = producers.id
    AND producers.name = 'foo';
```

也可以使用 CTE 改写上述语句，增加易读性。

```
WITH foo_producers as (
    SELECT * from producers
    where producers.name = 'foo'
)
DELETE FROM films USING foo_producers
WHERE producer_id = foo_producers.id;
```

DROP TABLE

">功能

该语句用于删除表。
语法

```
DROP [TEMPORARY] TABLE [IF EXISTS] [db_name.]table_name [FORCE]
```

说明：

- 执行 DROP TABLE 后一段时间内（默认 1 天），可以通过 RECOVER 语句恢复被删除的表。
- 如果执行 DROP TABLE FORCE，则系统不会检查该表是否存在未完成的事务，表将直接被删除并且不能被恢复，一般不建议执行此操作。
- 临时表删除后不可以通过 RECOVER 语句恢复。

">示例

1. 删除一个 TABLE。

```
DROP TABLE my_table;
```

2. 如果存在，删除指定 DATABASE 的 TABLE。

```
DROP TABLE IF EXISTS example_db.my_table;
```

3. 强制删除表，并清理磁盘文件。

```
DROP TABLE my_table FORCE;
```

DESC分区键 (Partition Key)

功能

可以使用该语句进行如下操作：
查看 StarRocks 表结构、排序键 (Sort Key) 类型和物化视图。
查看外部数据源（如 Apache Hive™）中的表结构。

语法

```
DESC[RIBE] { [[<catalog_name>.]<db_name>.]<table_name> [ALL] | FILES(files_loading_properties) }
```

参数说明

参数	必选	说明
catalog_name	否	Internal catalog 或 external catalog 的名称。 - 如指定 internal catalog 名称，即 default_catalog，则查看当前 StarRocks 集群的指定表结构。 - 如指定 external catalog 名称，则查看外部数据源的指定表结构。
db_name	否	数据库名称。
table_name	是	表名称。
ALL	否	- 如要查看 StarRocks 表的排序键类型和物化视图，则指定该关键字；如只查看 StarRocks 表结构，则可以不指定。 - 如查看外部数据源表结构，不能指定该关键词。
FILES	否	FILES() 表函数。自 v3.3.4 起，您可以使用 DESC 和 FILES() 查看远端存储中文件的 Schema 信息。

返回信息说明

```
+-----------+--------------+-------+------+------+-----+---------+-------+
| IndexName | IndexKeysType | Field | Type | Null | Key | Default | Extra |
+-----------+--------------+-------+------+------+-----+---------+-------+
```

返回信息中的参数说明：

参数	说明
IndexName	表名。如查看外部数据源表结构，则不返回该参数。
IndexKeysType	表的排序键类型。如查看外部数据源表结构，则不返回该参数。
Field	列名。
Type	列的数据类型。
Null	是否允许为 NULL。 - yes: 表示允许为 NULL。 - no：表示不允许为 NULL。
Key	是否为排序键。 - true: 表示为排序键。 - false：表示不为排序键。
Default	数据类型的默认值。如该数据类型没有默认值，则返回 NULL。
Extra	- 如果是查看 StarRocks 表结构，该参数会根据情况返回以下信息： - 该列使用了哪种聚合函数，如 SUM 和 MIN。 - 该列是否创建了 Bloom filter 索引。如创建，则追加显示 BLOOM_FILTER。 - 如果是查看外部数据源表结构，该参数会显示该列是否为分区键 (Partition Key)。如是，则显示 Partition Key。

">示例

示例一：查看 StarRocks 的 example_table 表结构信息。

```
DESC example_table;
```

或

```
DESC default_catalog.example_db.example_table;
```

返回信息如下：

```
+-------+--------------+------+-------+---------+-------+
| Field | Type         | Null | Key   | Default | Extra |
+-------+--------------+------+-------+---------+-------+
| k1    | TINYINT      | Yes  | true  | NULL    |       |
| k2    | DECIMAL(10,2) | Yes  | true  | 10.5    |       |
| k3    | CHAR(10)     | Yes  | false | NULL    |       |
| v1    | INT          | Yes  | false | NULL    |       |
+-------+--------------+------+-------+---------+-------+
```

示例二：查看 StarRocks 的 sales_records 表结构、排序键类型和物化视图。如下所示， sales_records 表只有一张物化视图 store_amt。

```
DESC db1.sales_records ALL;
```

```
+--------------+--------------+-----------+--------+------+-------+---------+-------+
| IndexName    | IndexKeysType | Field     | Type   | Null | Key   | Default | Extra |
+--------------+--------------+-----------+--------+------+-------+---------+-------+
| sales_records | DUP_KEYS     | record_id | INT    | Yes  | true  | NULL    |       |
|              |              | seller_id | INT    | Yes  | true  | NULL    |       |
|              |              | store_id  | INT    | Yes  | true  | NULL    |       |
|              |              | sale_date | DATE   | Yes  | false | NULL    | NONE  |
|              |              | sale_amt  | BIGINT | Yes  | false | NULL    | NONE  |
|              |              |           |        |      |       |         |       |
| store_amt    | AGG_KEYS     | store_id  | INT    | Yes  | true  | NULL    |       |
|              |              | sale_amt  | BIGINT | Yes  | false | NULL    | SUM   |
+--------------+--------------+-----------+--------+------+-------+---------+-------+
```

示例三：查看 Hive 中 hive_table 表结构。

```
DESC hive_catalog.hive_db.hive_table;
```

```
+-------+---------------+------+-------+---------+--------------+
| Field | Type          | Null | Key   | Default | Extra        |
+-------+---------------+------+-------+---------+--------------+
| id    | INT           | Yes  | false | NULL    |              |
| name  | VARCHAR(65533) | Yes  | false | NULL    |              |
| date  | DATE          | Yes  | false | NULL    | partition key |
+-------+---------------+------+-------+---------+--------------+
```

DROP INDEX

">功能

删除指定表的某个 bitmap 索引。创建 bitmap 索引会占用额外的存储空间，所以不用的索引建议删除。删除索引后，存储空间会立即释放。

">语法

```
DROP INDEX index_name ON [db_name.]table_name
```

参数说明

参数	必选	说明
index_name	是	要删除的索引名称。
table_name	是	创建索引的表。
db_name	否	表所属的数据库。

">示例

例如为表 sales_records 中的 item_id 列创建位图索引，索引名称为 index3。

```
CREATE INDEX index3 ON sales_records (item_id);
```

删除表sales_records中的索引index3。

```
DROP INDEX index3 ON sales_records;
```

REFRESH EXTERNAL TABLE

">功能

该语句用于更新缓存在 StarRocks 中的数据湖上的元数据，主要有以下两个使用场景：

- 外部表：使用 Hive 外部表查询 Hive 数据时，可使用该语句更新缓存的 Hive 元数据。
- External catalog：使用 Hive catalog查询对应数据源数据时，可使用该语句更新缓存的元数据。
  注意：
  只有拥有对应外表 ALTER 权限的用户才可以执行该操作。

">基本概念

- Hive 外部表：在 StarRocks 中创建并保存的表，用于查询 Hive 集群中的数据。
- Hive 表：在 Hive 中创建并保存的表。

">语法和参数说明

在不同的使用场景下，对应语法和参数说明如下：

- 外部表

```
REFRESH EXTERNAL TABLE <table_name>
[PARTITION ('partition_name', ...)]
```

参数	必选	说明
table_name	是	Hive 外部表名。
partition_name	否	Hive 表中的分区名。如指定，则更新缓存的 Hive 表指定分区的元数据。

- External catalog

```
REFRESH EXTERNAL TABLE [external_catalog.][db_name.]<table_name>
[PARTITION ('partition_name', ...)]
```

参数	必选	说明
external_catalog	否	外部数据目录名称，支持 Hive等。
db_name	否	表所在的数据库名。
table_name	是	表名。
partition_name	否	表中的分区名。如指定，则更新缓存的表中指定分区的元数据。

">示例

在不同使用场景下， 对应的示例如下。
外部表
示例：更新外部表 hive1 对应的 Hive 表元数据。

```
REFRESH EXTERNAL TABLE hive1;
```

External catalog
示例一：更新缓存的 hive_table 表的元数据。

```
REFRESH EXTERNAL TABLE hive_catalog.hive_db.hive_table;
```

或

```
USE hive_catalog.hive_db;
REFRESH EXTERNAL TABLE hive_table;
```

示例二：更新缓存的 hive_table 表的二级分区 p2 的元数据。

```
USE hive_catalog.hive_db;
REFRESH EXTERNAL TABLE hive_table PARTITION ('p1=${date}/p2=${hour}');
```

SELECT

">功能

SELECT 语句用于从单个或多个表，视图，物化视图中读取数据。
SELECT 可以作为独立的语句也可以作为其他语句的子句，其查询结果可以作为另一个语句的输入值。
StarRocks 的查询语句基本符合 SQL-92 标准。下面介绍支持的 SELECT 用法。
说明：如果要查询 StarRocks 表、视图、或物化视图内的数据，需要有对应对象的 SELECT 权限。如果要查询 External Catalog 里的数据，需要有对应 Catalog 的 USAGE 权限。

">WITH

可以在 SELECT 语句之前添加的子句，用于定义在 SELECT 内部多次引用的复杂表达式的别名。
与 CREATE VIEW 类似，但在子句中定义的表和列名在查询结束后不会持久，也不会与实际表或 VIEW 中的名称冲突。
使用 WITH 子句的好处：

- 方便和易于维护，减少查询内部的重复。
- 通过将查询中最复杂的部分抽象成单独的块，更易于阅读和理解 SQL 代码。
  示例：

```
-- Define one subquery at the outer level, and another at the inner level as part of the
-- initial stage of the UNION ALL query.

with t1 as (select 1),t2 as (select 2)
select * from t1 union all select * from t2;
```

">连接 (Join)

连接操作合并 2 个或多个表的数据，然后返回某些表中某些列的结果集。
目前 StarRocks 支持 Self Join、Cross Join、Inner Join、Outer Join、Semi Join 和 Anti Join。其中，Outer Join 包括 Left Join、Right Join 和 Full Join。
Join 的语法定义如下：

```
SELECT select_list FROM
table_or_subquery1 [INNER] JOIN table_or_subquery2 |
```

```
table_or_subquery1 {LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]} JOIN table_or_subquery2 |
table_or_subquery1 {LEFT | RIGHT} SEMI JOIN table_or_subquery2 |
table_or_subquery1 {LEFT | RIGHT} ANTI JOIN table_or_subquery2 |
[ ON col1 = col2 [AND col3 = col4 ...] |
USING (col1 [, col2 ...]) ]
[other_join_clause ...]
[ WHERE where_clauses ]


SELECT select_list FROM
table_or_subquery1, table_or_subquery2 [, table_or_subquery3 ...]
[other_join_clause ...]
WHERE
col1 = col2 [AND col3 = col4 ...]


SELECT select_list FROM
table_or_subquery1 CROSS JOIN table_or_subquery2
[other_join_clause ...]
[ WHERE where_clauses ]
```

Self Join

StarRocks 支持 Self Join，即自己和自己 Join。例如同一张表的不同列进行 Join。

实际上没有特殊的语法标识 Self Join。Self Join 中 Join 两边的条件都来自同一张表，

需要给他们分配不同的别名。

例如：

```
SELECT lhs.id, rhs.parent, lhs.c1, rhs.c2 FROM tree_data lhs, tree_data rhs WHERE lhs.id = rhs.parent;
```

笛卡尔积 (Cross Join)

Cross Join 会产生大量的结果，须慎用 Cross Join。

即使需要使用 Cross Join 时也需要使用过滤条件并且确保返回结果数较少。例如：

```
SELECT * FROM t1, t2;
SELECT * FROM t1 CROSS JOIN t2;
```

Inner Join

Inner Join 是大家最熟知，最常用的 Join。返回的结果来自相近的两张表所请求的列，Join 的条件为两个表的列包含有相同的值。

如果两个表的某个列名相同，我们需要使用全名（table_name.column_name 形式）或者给列名起别名。

例如：

下列 3 个查询是等价的。

```
SELECT t1.id, c1, c2 FROM t1, t2 WHERE t1.id = t2.id;
SELECT t1.id, c1, c2 FROM t1 JOIN t2 ON t1.id = t2.id;
```

```
SELECT t1.id, c1, c2 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

Outer Join

Outer Join 返回左表或者右表或者两者所有的行。如果在另一张表中没有匹配的数据，则将其设置为 NULL。例如：

```
SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 RIGHT OUTER JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 FULL OUTER JOIN t2 ON t1.id = t2.id;
```

Semi Join

Left Semi Join 只返回左表中能匹配右表数据的行，不管能匹配右表多少行数据，
左表的该行最多只返回一次。Right Semi Join 原理相似，只是返回的数据是右表的。
例如：

```
SELECT t1.c1, t1.c2, t1.c2 FROM t1 LEFT SEMI JOIN t2 ON t1.id = t2.id;
```

Anti Join

Left Anti Join 只返回左表中不能匹配右表的行。
Right Anti Join 反转了这个比较，只返回右表中不能匹配左表的行。例如：

```
SELECT t1.c1, t1.c2, t1.c2 FROM t1 LEFT ANTI JOIN t2 ON t1.id = t2.id;
```

等值 Join 和非等值 Join

根据 Join 条件的不同，StarRocks 支持的上述各种类型的 Join 可以分为等值 Join 和非等值 Join，如下表所示：

等值 Join	Self Join、Cross Join、Inner Join、Outer Join、Semi Join 和 Anti Join
非等值 Join	Cross Join、Inner Join，LEFT SEMI JOIN， LEFT ANTI JOIN 和 Outer Join

- 等值 Join
  等值 Join 使用等值条件作为连接条件，例如 a JOIN b ON a.id = b.id。

- 非等值 Join
  非等值 Join 不使用等值条件，使用 <、<=、>、>=、<> 等比较操作符，例如 a JOIN b ON a.id < b.id。与等值 Join 相比，非等值 Join 目前效率较低，建议谨慎使用。

以下为非等值 Join 的两个使用示例：

```
SELECT t1.id, c1, c2
FROM t1
INNER JOIN t2 ON t1.id < t2.id;
```

```
SELECT t1.id, c1, c2
```

```
FROM t1
LEFT JOIN t2 ON t1.id > t2.id;
```

">ORDER BY

ORDER BY 通过比较一列或者多列的大小来对结果集进行排序。
ORDER BY 是比较耗时耗资源的操作，因为所有数据都需要发送到 1 个节点后才能排序，需要更多的内存。
如果需要返回前 N 个排序结果，需要使用 LIMIT 子句；为了限制内存的使用，如果您没有指定 LIMIT 子句，则默认返回前 65535 个排序结果。
ORDER BY 语法定义如下：

```
ORDER BY col [ASC | DESC]
[ NULLS FIRST | NULLS LAST ]
```

默认排序顺序是 ASC（升序）。示例：

```
select * from big_table order by tiny_column, short_column desc;
```

StarRocks 支持在 ORDER BY 后声明 NULL 值排在最前面还是最后面，语法为 order by <> [NULLS FIRST | NULLS LAST]。NULLS FIRST 表示 null 值的记录将排在最前面，NULLS LAST 表示 null 值的记录将排在最后面。
示例：将 NULL 值始终排在最前面。

```
select  *  from  sales_record  order by  employee_id  nulls first;
```

">GROUP BY

GROUP BY 子句通常和聚合函数（例如 COUNT(), SUM(), AVG(), MIN()和 MAX()）一起使用。
GROUP BY 指定的列不会参加聚合操作。GROUP BY 子句可以加入 HAVING 子句来过滤聚合函数产出的结果。例如：

```
select tiny_column, sum(short_column)
from small_table
group by tiny_column;
```

```
+-------------+---------------------+
| tiny_column |  sum('short_column')|
+-------------+---------------------+
|    1    |     2        |
|    2    |     1        |
```

```
+------------+--------------------+
```

语法

```
SELECT ...
FROM ...
[ ... ]
GROUP BY [
    , ... |
    GROUPING SETS [, ...] (  groupSet [ , groupSet [ , ... ] ] ) |
    ROLLUP(expr  [ , expr [ , ... ] ]) |
    CUBE(expr  [ , expr [ , ... ] ])
    ]
[ ... ]
```

">Parameters

- groupSet 表示 select list 中的列，别名或者表达式组成的集合 groupSet ::= { (expr [, expr [, ...]])}。
- expr 表示 select list 中的列，别名或者表达式。

">Note

StarRocks 支持类似 PostgreSQL 语法，语法实例如下：

```
SELECT a, b, SUM( c ) FROM tab1 GROUP BY GROUPING SETS ( (a, b), (a), (b), ( ) );
SELECT a, b,c, SUM( d ) FROM tab1 GROUP BY ROLLUP(a,b,c)
SELECT a, b,c, SUM( d ) FROM tab1 GROUP BY CUBE(a,b,c)
```

ROLLUP(a,b,c) 等价于如下 GROUPING SETS 语句。

```
GROUPING SETS (
(a,b,c),
(a,b  ),
(a    ),
(     )
)
```

CUBE (a, b, c) 等价于如下 GROUPING SETS 语句。

```
GROUPING SETS (
( a, b, c ),
( a, b    ),
( a,    c ),
( a       ),
(    b, c ),
(    b    ),
(       c ),
(         )
)
```

">示例

下面是一个实际数据的例子:

```
SELECT * FROM t;
+------+------+------+
| k1   | k2   | k3   |
+------+------+------+
| a    | A    |    1 |
| a    | A    |    2 |
| a    | B    |    1 |
| a    | B    |    3 |
| b    | A    |    1 |
| b    | A    |    4 |
| b    | B    |    1 |
| b    | B    |    5 |
+------+------+------+
8 rows in set (0.01 sec)

SELECT k1, k2, SUM(k3) FROM t
GROUP BY GROUPING SETS ( (k1, k2), (k2), (k1), ( ) );
+------+------+-----------+
| k1   | k2   | sum(`k3`) |
+------+------+-----------+
| b    | B    |         6 |
| a    | B    |         4 |
| a    | A    |         3 |
| b    | A    |         5 |
| NULL | B    |        10 |
| NULL | A    |         8 |
| a    | NULL |         7 |
```

```
| b    | NULL |       11 |
| NULL | NULL |       18 |
+------+------+----------+
9 rows in set (0.06 sec)

SELECT k1, k2, GROUPING_ID(k1,k2), SUM(k3) FROM t
GROUP BY GROUPING SETS ((k1, k2), (k1), (k2), ());
+------+------+---------------+----------------+
| k1   | k2   | grouping_id(k1,k2) | sum(`k3`) |
+------+------+---------------+----------------+
| a    | A    |             0 |              3 |
| a    | B    |             0 |              4 |
| a    | NULL |             1 |              7 |
| b    | A    |             0 |              5 |
| b    | B    |             0 |              6 |
| b    | NULL |             1 |             11 |
| NULL | A    |             2 |              8 |
| NULL | B    |             2 |             10 |
| NULL | NULL |             3 |             18 |
+------+------+---------------+----------------+
9 rows in set (0.02 sec)
```

GROUP BY GROUPING SETS | CUBE | ROLLUP 是对 GROUP BY 子句的扩展，它能够在一个 GROUP BY 子句中实现多个集合的分组的聚合。其结果等价于将多个相应 GROUP BY 子句进行 UNION 操作。
GROUP BY 子句是只含有一个元素的 GROUP BY GROUPING SETS 的特例。 例如，GROUPING SETS 语句：

```
SELECT a, b, SUM( c ) FROM tab1 GROUP BY GROUPING SETS ( (a, b), (a), (b), ( ) );
```

其查询结果等价于：

```
SELECT a, b, SUM( c ) FROM tab1 GROUP BY a, b
UNION
SELECT a, null, SUM( c ) FROM tab1 GROUP BY a
UNION
SELECT null, b, SUM( c ) FROM tab1 GROUP BY b
UNION
SELECT null, null, SUM( c ) FROM tab1
```

GROUPING(expr) 指示一个列是否为聚合列，如果是聚合列为 0，否则为 1。
GROUPING_ID(expr [, expr [, ...]]) 与 GROUPING 类似，GROUPING_ID 根据指定的 column 顺序，计算出一个列列表的 bitmap 值，每一位为 GROUPING 的值. GROUPING_ID()函数返回位向量的十进制值。

">HAVING

HAVING 子句不过滤表中的行数据，而是过滤聚合函数产出的结果。
通常来说 HAVING 要和聚合函数（例如 COUNT(), SUM(), AVG(), MIN(), MAX()）以及 group by 子句一起使用。
示例：

```
select tiny_column, sum(short_column)
from small_table
group by tiny_column
having sum(short_column) = 1;


+-------------+--------------------+
|tiny_column  | sum('short_column') |
+-------------+--------------------+
|       2   |      1         |
+-------------+--------------------+

1 row in set (0.07 sec)


select tiny_column, sum(short_column)
from small_table
group by tiny_column
having tiny_column > 1;


+-------------+--------------------+
|tiny_column  | sum('short_column') |
+-------------+--------------------+
|     2     |        1        |
+-------------+--------------------+

1 row in set (0.07 sec)
```

">LIMIT

LIMIT 子句用于限制返回结果的最大行数。设置返回结果的最大行数可以帮助 StarRocks 优化内存的使用。
该子句主要应用如下场景：

1. 返回 top-N 的查询结果。
2. 简单查看表中包含的内容。
3. 表中数据量大，或者 WHERE 子句没有过滤太多的数据，需要限制查询结果集的大小。
   使用说明：LIMIT 子句的值必须是数字型字面常量。
   示例：

```
mysql> select tiny_column from small_table limit 1;

+-------------+
|tiny_column  |
+-------------+
|    1      |
+-------------+

1 row in set (0.02 sec)

mysql> select tiny_column from small_table limit 10000;

+-------------+
|tiny_column  |
+-------------+
|     1     |
|     2     |
+-------------+

2 rows in set (0.01 sec)
```

">OFFSET

OFFSET 子句用于跳过结果集的前若干行结果，直接返回后续的结果。
默认从第 0 行开始，因此 OFFSET 0 和不带 OFFSET 返回相同的结果。
通常来说，OFFSET 子句需要与 ORDER BY 子句和 LIMIT 子句一起使用才有效。
示例：

```
mysql> select varchar_column from big_table order by varchar_column limit 3;

+----------------+
| varchar_column |
+----------------+
|    city1     |
|    city2   |
|    city3     |
+----------------+

3 rows in set (0.02 sec)

mysql> select varchar_column from big_table order by varchar_column limit 1 offset 0;
```

```
+----------------+
|varchar_column  |
+----------------+
|    city1    |
+----------------+

1 row in set (0.01 sec)

mysql> select varchar_column from big_table order by varchar_column limit 1 offset 1;

+----------------+
|varchar_column  |
+----------------+
|    city2    |
+----------------+

1 row in set (0.01 sec)

mysql> select varchar_column from big_table order by varchar_column limit 1 offset 2;

+----------------+
|varchar_column  |
+----------------+
|    city3    |
+----------------+

1 row in set (0.02 sec)
```

注意：在没有 ORDER BY 的情况下使用 OFFSET 语法是允许的，但是此时 OFFSET 无意义。这种情况只取 LIMIT 的值，忽略 OFFSET 的值。因此在没有 ORDER BY 的情况下，OFFSET 超过结果集的最大行数依然是有结果的。建议使用 OFFSET 时一定带上 ORDER BY。

UNION
UNION 子句用于合并多个查询的结果，即获取并集。
语法如下：

```
query_1 UNION [DISTINCT | ALL] query_2
```

- DISTINCT：默认值，返回不重复的结果。UNION 和 UNION DISTINCT 效果相同（见第二个示例）。
- ALL: 返回所有结果的集合，不进行去重。由于去重工作比较耗费内存，因此使用 UNION ALL 查询速度会快一些，内存消耗也会少一些（见第一个示例）。
  说明：每条 SELECT 查询返回的列数必须相同，且列类型必须能够兼容。

示例：

以表 select1 和 select2 示例说明。

```
CREATE TABLE select1(
    id          INT,
    price       INT
    )
DISTRIBUTED BY HASH(id);

INSERT INTO select1 VALUES
    (1,2),
    (1,2),
    (2,3),
    (5,6),
    (5,6);

CREATE TABLE select2(
    id          INT,
    price       INT
    )
DISTRIBUTED BY HASH(id);

INSERT INTO select2 VALUES
    (2,3),
    (3,4),
    (5,6),
    (7,8);
```

示例一：返回两张表中所有 id 的并集，不进行去重。

```
mysql> (select id from select1) union all (select id from select2) order by id;

+------+
| id   |
+------+
|    1 |
|    1 |
|    2 |
|    2 |
|    3 |
|    5 |
|    5 |
|    5 |
|    7 |
+------+
11 rows in set (0.02 sec)
```

示例二：返回两张表中 id 的并集，进行去重。下面两条查询在功能上对等。

```
mysql> (select id from select1) union (select id from select2) order by id;

+------+
| id   |
+------+
|    1 |
|    2 |
|    3 |
|    5 |
|    7 |
+------+
6 rows in set (0.01 sec)

mysql> (select id from select1) union distinct (select id from select2) order by id;
+------+
| id   |
+------+
|    1 |
|    2 |
|    3 |
|    5 |
|    7 |
+------+
5 rows in set (0.02 sec)
```

示例三：返回两张表中所有 id 的并集，进行去重，只返回 3 行结果。下面两条查询在功能上对等。

```
mysql> (select id from select1) union distinct (select id from select2)
order by id
limit 3;
++------+
| id   |
+------+
|    1 |
|    2 |
|    3 |
+------+
4 rows in set (0.11 sec)

mysql> select * from (select id from select1 union distinct select id from select2) as t1
order by id
limit 3;
+------+
| id   |
```

```
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.01 sec)
```

">INTERSECT

INTERSECT 子句用于返回多个查询结果之间的交集，即每个结果中都有的数据，并对结果集进行去重。
语法如下：

```
query_1 INTERSECT [DISTINCT] query_2
```

说明

- INTERSECT 效果等同于 INTERSECT DISTINCT。不支持 ALL 关键字。
- 每条 SELECT 查询返回的列数必须相同，且列类型能够兼容。
  示例：
  继续使用 UNION 子句里的两张表。返回两张表中 id 和 price 组合的交集。下面两条查询在功能上对等。

```
mysql> (select id, price from select1) intersect (select id, price from select2)
order by id;

+------+-------+
| id   | price |
+------+-------+
|    2 |     3 |
|    5 |     6 |
+------+-------+

mysql> (select id, price from select1) intersect distinct (select id, price from select2)
order by id;

+------+-------+
| id   | price |
+------+-------+
|    2 |     3 |
|    5 |     6 |
+------+-------+
```

EXCEPT/MINUS
EXCEPT/MINUS 子句用于返回多个查询结果之间的补集，即返回左侧查询中在右侧查询中不存在的数据，并对结果

集去重。EXCEPT 和 MINUS 功能对等。
语法如下：

```
query_1 {EXCEPT | MINUS} [DISTINCT] query_2
```

说明

- EXCEPT 效果等同于 EXCEPT DISTINCT。不支持 ALL 关键字。
- 每条 SELECT 查询返回的列数必须相同，且列类型能够兼容。
  示例：
  继续使用 UNION 子句里的两张表。返回表 select1 中不存在于表 select2 的 (id,price) 组合。
  可以看到在结果中对组合 (1,2) 进行了去重。

```
mysql> (select id, price from select1) except (select id, price from select2)
order by id;
+------+-------+
| id   | price |
+------+-------+
|    1 |     2 |
+------+-------+

mysql> (select id, price from select1) minus (select id, price from select2)
order by id;
+------+-------+
| id   | price |
+------+-------+
|    1 |     2 |
+------+-------+
```

">DISTINCT

DISTINCT 关键字对结果集进行去重。示例：

```
-- 返回一列中去重后的值。
select distinct tiny_column from big_table limit 2;

-- 返回多列中去重后的组合。
select distinct tiny_column, int_column from big_table limit 2;
```

DISTINCT 可以和聚合函数 (通常是 COUNT) 一同使用，COUNT(distinct) 用于计算出一个列或多个列上包含多少不同的组合。

```
-- 计算一列中不重复值的个数。
select count(distinct tiny_column) from small_table;
```

```
+------------------------------+
| count(DISTINCT 'tiny_column') |
+------------------------------+
|              2               |
+------------------------------+
1 row in set (0.06 sec)
```

```
-- 计算多列中不重复组合的个数。
select count(distinct tiny_column, int_column) from big_table limit 2;
```

StarRocks 支持多个聚合函数同时使用 DISTINCT。

```
-- 单独返回多个聚合函数去重后的个数。
select count(distinct tiny_column, int_column), count(distinct varchar_column) from big_table;
```

">子查询

子查询按相关性可以分为不相关子查询和相关子查询。

- 不相关子查询（简单查询）不依赖外层查询的结果。
- 相关子查询需要依赖外层查询的结果才能执行。

不相关子查询

不相关子查询支持 [NOT] IN 和 EXISTS。

示例：

```
SELECT x FROM t1 WHERE x [NOT] IN (SELECT y FROM t2);
SELECT * FROM t1 WHERE (x,y) [NOT] IN (SELECT x,y FROM t2 LIMIT 2);
SELECT x FROM t1 WHERE EXISTS (SELECT y FROM t2 WHERE y = 1);
```

从 3.0 版本开始，SELECT... FROM... WHERE... [NOT] IN 支持在 WHERE 中指定多个字段进行比较，即上面第二个示例中的 WHERE (x,y) 用法。

相关子查询

相关子查询支持 [NOT] IN 和 [NOT] EXISTS。

示例：

```
SELECT * FROM t1 WHERE x [NOT] IN (SELECT a FROM t2 WHERE t1.y = t2.b);
```

SELECT * FROM t1 WHERE [NOT] EXISTS (SELECT a FROM t2 WHERE t1.y = t2.b);

子查询还支持标量子查询。分为不相关标量子查询、相关标量子查询和标量子查询作为普通函数的参数。

示例：

1、 不相关标量子查询，谓词为 = 号。例如输出最高工资的人的信息。

SELECT name FROM table WHERE salary = (SELECT MAX(salary) FROM table);

2、不相关标量子查询，谓词为 >,< 等。例如输出比平均工资高的人的信息。

SELECT name FROM table WHERE salary > (SELECT AVG(salary) FROM table);

3、相关标量子查询。例如输出各个部门工资最高的信息。

SELECT name FROM table a WHERE salary = （SELECT MAX(salary) FROM table b WHERE b.部门= a.部门）;

4、标量子查询作为普通函数的参数。

SELECT name FROM table WHERE salary = abs((SELECT MAX(salary) FROM table));

">WHERE 与操作符

WHERE 与操作符

SQL 操作符是一系列用于比较的函数，这些操作符广泛地用于 SELECT 语句的 WHERE 子句中。

算术操作符

算术操作符通常出现在包含左操作数，操作符和右操作数（大部分情况下）组成的表达式中。

+和-：分别代表着加法和减法，可以作为单元或 2 元操作符。当其作为单元操作符时，如+1, -2.5 或者-col_name，表达的意思是该值乘以+1 或者-1。

因此单元操作符+返回的是未发生变化的值，单元操作符-改变了该值的符号位。

用户可以将两个单元操作符叠加起来，比如+ +5(返回的是正值)，-+2 或者+ -2（这两种情况返回的是负值），但是用户不能使用连续的两个-号。

因为--被解释为后面的语句是注释（用户也是可以使用两个-号的，此时需要在两个-号之间加上空格或圆括号，如-(-2)或者- -2，这两种写法实际表达的结果都是+2）。

+或者-作为 2 元操作符时，例如 2+2，3-1.5 或者 col1 + col2，表达的含义是左值加或者减去右值。左值和右值必须都是数字类型。

*和/：分别代表着乘法和除法操作符。两侧的操作数必须都是数字类型。

当两个数相乘时，类型较小的操作数在需要的情况下类型可能会提升（比如 SMALLINT 提升到 INT 或者 BIGINT

等），表达式的结果被提升到下一个较大的类型，
比如 TINYINT 乘以 INT 产生的结果的类型会是 BIGINT。当两个数相乘时，为了避免精度丢失，操作数和表达式结果都会被解释成 DOUBLE 类型。
如果用户想把表达式结果转换成其他类型，需要用 CAST 函数转换。

%：取模操作符。返回左操作数除以右操作数的余数。左操作数和右操作数都必须是整型。

&，|和^：按位操作符返回对两个操作数进行按位与，按位或，按位异或操作的结果。两个操作数都要求是一种整型类型。
如果按位操作符的两个操作数的类型不一致，则类型小的操作数会被提升到类型较大的操作数，然后再做相应的按位操作。
在 1 个表达式中可以出现多个算术操作符，用户可以用小括号将相应的算术表达式括起来。算术操作符通常没有对应的数学函数来表达和算术操作符相同的功能。
比如我们没有 MOD()函数来表示%操作符的功能。反过来，数学函数也没有对应的算术操作符。比如幂函数 POW()并没有相应的 **求幂操作符。

BETWEEN 操作符
在 WHERE 子句中，表达式可能同时与上界和下界比较。如果表达式大于等于下界，同时小于等于上界，比较的结果是 true。语法定义如下：

```
expression BETWEEN lower_bound AND upper_bound
```

数据类型：通常表达式（expression）的计算结果都是数字类型，该操作符也支持其他数据类型。如果必须要确保下界和上界都是可比较的数据类型，可以使用 CAST()函数。
使用说明：如果操作数是 string 类型应注意，起始部分为上界的长字符串将不会匹配上界，该字符串比上界要大。例如：‘MJ’比‘M’大，所以 BETWEEN 'A' and 'M' 不会匹配‘MJ’。
如果需要确保表达式能够正常工作，可以使用一些函数，如 UPPER(), LOWER(), SUBSTR(), TRIM()。
示例：

```
select c1 from t1 where month between 1 and 6;
```

比较操作符
比较操作符用来判断列和列是否相等或者对列进行排序。=, !=, <=, >= 可以适用所有数据类型。
其中 <> 符号是不等于的意思，与 != 的功能一致。IN 和 BETWEEN 操作符提供更简短的表达来描述相等、小于、大于等关系的比较。

IN 操作符
IN 操作符会和 VALUE 集合进行比较，如果可以匹配该集合中任何一元素，则返回 TRUE。
参数和 VALUE 集合必须是可比较的。所有使用 IN 操作符的表达式都可以写成用 OR 连接的等值比较，但是 IN 的语法更简单，更精准，更容易让 StarRocks 进行优化。
示例：

```
select * from small_table where tiny_column in (1,2);
```

LIKE 操作符
该操作符用于和字符串进行比较。"_"用来匹配单个字符，"%"用来匹配多个字符。参数必须要匹配完整的字符串。通常，把"%" 放在字符串的尾部更加符合实际用法。
示例：

```
mysql> select varchar_column from small_table where varchar_column like 'm%';

+----------------+
|varchar_column  |
+----------------+
|    milan       |
+----------------+

1 row in set (0.02 sec)

mysql> select varchar_column from small_table where varchar_column like 'm____';

+----------------+
| varchar_column |
+----------------+
|    milan       |
+----------------+

1 row in set (0.01 sec)
```

逻辑操作符
逻辑操作符返回一个 BOOL 值，逻辑操作符包括单元操作符和多元操作符，每个操作符处理的参数都是返回值为 BOOL 值的表达式。支持的操作符有：

- AND: 2 元操作符，如果左侧和右侧的参数的计算结果都是 TRUE，则 AND 操作符返回 TRUE。
- OR: 2 元操作符，如果左侧和右侧的参数的计算结果有一个为 TRUE，则 OR 操作符返回 TRUE。如果两个参数都是 FALSE，则 OR 操作符返回 FALSE。
- NOT: 单元操作符，反转表达式的结果。如果参数为 TRUE，则该操作符返回 FALSE；如果参数为 FALSE，则该操作符返回 TRUE。
  示例：

```
mysql> select true and true;

+------------------+
| (TRUE) AND (TRUE) |
+------------------+
```

```
|       1       |
+-------------------+

1 row in set (0.00 sec)

mysql> select true and false;

+--------------------+
| (TRUE) AND (FALSE) |
+--------------------+
|       0        |
+--------------------+

1 row in set (0.01 sec)

mysql> select true or false;

+-------------------+
| (TRUE) OR (FALSE) |
+-------------------+
|       1       |
+-------------------+

1 row in set (0.01 sec)

mysql> select not true;

+----------+
| NOT TRUE |
+----------+
|    0   |
+----------+

1 row in set (0.01 sec)
```

正则表达式操作符

判断是否匹配正则表达式，使用 POSIX 标准的正则表达式。

"^" 用来匹配字符串的首部，"$" 用来匹配字符串的尾部，"." 匹配任何一个单字符，"*"匹配 0 个或多个选项，"+"匹配 1个或多个选项，"?" 匹配 0 个或1个等等。正则表达式需要匹配完整的值，并不是仅仅匹配字符串的部分内容。

如果想匹配中间的部分，正则表达式的前面部分可以写成 "^." 或者"."。"^" 和 "$" 通常是可以省略的。RLIKE 操作符和 REGEXP 操作符是同义词。

"|" 操作符是个可选操作符，"|" 两侧的正则表达式只需满足 1 侧条件即可，"|" 操作符和两侧的正则表达式通常需要用()括起来。

示例：

```
mysql> select varchar_column from small_table where varchar_column regexp '(mi|MI).*';

+----------------+
| varchar_column |
+----------------+
|     milan      |
+----------------+

1 row in set (0.01 sec)


mysql> select varchar_column from small_table where varchar_column regexp 'm.*';

+----------------+
| varchar_column |
+----------------+
|     milan      |
+----------------+

1 row in set (0.01 sec)
```

别名 (alias)

在查询中书写表名、列名，或者包含列的表达式的名字时，可以通过 AS 给它们分配一个别名。

当需要使用表名、列名时，可以使用别名来访问。别名通常相对原名来说更简短更容易记忆。当需要新建一个别名时，只需在 SELECT list 或者 FROM list 中的表、列、表达式名称后面加上 AS alias 子句即可。AS 关键词是可选的，用户可以直接在原名后面指定别名。如果别名或者其他标识符和 StarRocks 内部保留关键字同名时，需要在该名称加上反引号，比如 rank。别名对大小写敏感，但是列别名和表达式别名对大小写不敏感。

示例：

```
select tiny_column as name, int_column as sex from big_table;

select sum(tiny_column) as total_count from big_table;

select one.tiny_column, two.int_column from small_table one, big_table two where one.tiny_column = t
wo.tiny_column;
```

PIVOT

该函数从 3.3 版本开始支持。

PIVOT操作符是SQL中的一个高级特性，它允许你将表中的行转换为列，通常用于数据透视表的创建。这在处理数据库报表或分析时非常有用，特别是当你需要对数据进行汇总或分类展示时。

实际上，PIVOT 是一种语法糖，它可以简化像 sum(case when ... then ... end) 这样的查询语句的编写。

语法

```
pivot:
SELECT ...
FROM ...
PIVOT (
  aggregate_function(<expr>) [[AS] alias] [, aggregate_function(<expr>) [[AS] alias] ...]
  FOR <pivot_column>
  IN (<pivot_value>)
)

pivot_column:
<column_name>
| (<column_name> [, <column_name> ...])

pivot_value:
<literal> [, <literal> ...]
| (<literal>, <literal> ...) [, (<literal>, <literal> ...)]
```

参数

在PIVOT操作中，你需要指定以下几个关键部分：

- aggregate_function()：聚合函数，如SUM、AVG、COUNT等，用于对数据进行汇总。
- alias：为聚合结果指定的别名，使得结果更易于理解。
- FOR pivot_column：指定要进行行转列操作的列名。
- IN (pivot_value)：指定pivot_column列中要转换为列的具体值。

  示例

```
create table t1 (c0 int, c1 int, c2 int, c3 int);
SELECT * FROM t1 PIVOT (SUM(c1) AS sum_c1, AVG(c2) AS avg_c2 FOR c3 IN (1, 2, 3, 4, 5));
-- 结果等同于以下查询：
SELECT SUM(CASE WHEN c3 = 1 THEN c1 ELSE NULL END) AS sum_c1_1,
       AVG(CASE WHEN c3 = 1 THEN c2 ELSE NULL END) AS avg_c2_1,
       SUM(CASE WHEN c3 = 2 THEN c1 ELSE NULL END) AS sum_c1_2,
       AVG(CASE WHEN c3 = 2 THEN c2 ELSE NULL END) AS avg_c2_2,
       SUM(CASE WHEN c3 = 3 THEN c1 ELSE NULL END) AS sum_c1_3,
       AVG(CASE WHEN c3 = 3 THEN c2 ELSE NULL END) AS avg_c2_3,
       SUM(CASE WHEN c3 = 4 THEN c1 ELSE NULL END) AS sum_c1_4,
       AVG(CASE WHEN c3 = 4 THEN c2 ELSE NULL END) AS avg_c2_4,
       SUM(CASE WHEN c3 = 5 THEN c1 ELSE NULL END) AS sum_c1_5,
```

```
    AVG(CASE WHEN c3 = 5 THEN c2 ELSE NULL END) AS avg_c2_5
FROM t1
GROUP BY c0;
```

UPDATE

该语句用于更新一张主键表中的数据行。
2.3 ~ 2.5 版本，StarRocks 提供 UPDATE 语句，并且仅支持单表 UPDATE 且不支持公用表表达式（CTE）。从 3.0 版本开始，StarRocks 丰富了 UPDATE 语法，支持使用多表关联和 CTE。如果需要将待更新的表与数据库中其他表关联，则可以在 FROM 子句或 CTE 中引用其他的表。从 3.1 版本开始，支持列模式的部分更新，适用于涉及少数列但是大量行的场景，加快更新速度。

使用说明

如果为多表 UPDATE，则您需要确保 UPDATE 语句中 FROM 子句的表表达式可以转换成等价的 JOIN 查询语句。因为 StarRocks 实际执行 UPDATE 语句时，内部会进行这样的转换。假设 UPDATE 语句为 UPDATE t0 SET v1=t1.v1 FROM t1 WHERE t0.pk = t1.pk;，该 FROM 子句的表表达式可以转换为 t0 JOIN t1 ON t0.pk=t1.pk;。并且 StarRocks 根据 JOIN 查询的结果集，匹配待更新表的数据行，更新其指定列的值。如果结果集中存在多行数据和待更新表的某一行数据相匹配，则待更新表中这行数据指定列更新后的值，是结果集多行数据中随机一行的指定列的值。

语法

单表 UPDATE

如果待更新表的数据行满足 WHERE 条件，则对该数据行的指定列赋予新值。

```
[ WITH <with_query> [, ...] ]
UPDATE <table_name>
    SET <column_name> = <expression> [, ...]
    WHERE <where_condition>
```

多表 UPDATE

基于多表关联查询的结果集与待更新的表进行匹配，如果待更新表的数据行匹配结果集并且满足 WHERE 条件，则对该数据行的指定列赋予新值。

```
[ WITH <with_query> [, ...] ]
UPDATE <table_name>
    SET <column_name> = <expression> [, ...]
    [ FROM <from_item> [, ...] ]
    WHERE <where_condition>
```

">参数说明

with_query：
一个或多个可以在 UPDATE 语句中通过名字引用的 CTE。CTE 是一个临时结果集，可以提高复杂语句的易读性。

table_name：
待更新的表的名称。

column_name：
待更新的列的名称。不需要包含表名，例如 UPDATE t1 SET t1.col = 1 是不合法的。

expression：给列赋值的表达式。

from_item：引用数据库中一个或者多个其他的表。该表与待更新的表进行连接，WHERE 子句指定连接条件，最终基于连接查询的结果集给待更新的表中匹配行的列赋值。 例如 FROM 子句为 FROM t1 WHERE t0.pk = t1.pk;，StarRocks 实际执行 UPDATE 语句时会将该 FROM 子句的表表达式转换为 t0 JOIN t1 ON t0.pk=t1.pk;。

where_condition：只有满足 WHERE 条件的行才会被更新。该参数为必选，防止误更新整张表。如需更新整张表，请使用 WHERE true。如果使用列模式的部分更新，则该参数不是必选。

列模式的部分更新

列模式的部分更新适用于涉及少数列并且大量行的场景。在该场景，开启列模式，更新速度更快。例如，在一个包含 100 列的表中，每次更新 10 列（占比 10%）并更新所有行，则开启列模式，更新性能将提高 10 倍。
系统变量 partial_update_mode 用于控制部分更新的模式，支持取值为：

- auto（默认值），表示由系统通过分析更新语句以及其涉及的列，自动判断执行部分更新时使用的模式。如果满足如下标准，则系统自动使用列模式：
    - 更新的列数占所有列数的百分比小于 30%，并且更新的列数少于 4 个。
    - 更新语句中没有使用 WHERE 条件。

    反之，则系统不会使用列模式。
- column，指定使用列模式执行部分更新，比较适用于涉及少数列并且大量行的部分列更新场景。
  可以使用 EXPLAIN UPDATE xxx 查看执行部分列更新的模式。

">示例

">单表 UPDATE

创建表 Employees 来记录雇员信息，向表中插入五行数据。

```
CREATE TABLE Employees (
    EmployeeID INT,
    Name VARCHAR(50),
    Salary DECIMAL(10, 2)
)
PRIMARY KEY (EmployeeID)
DISTRIBUTED BY HASH (EmployeeID)
PROPERTIES ("replication_num" = "3");

INSERT INTO Employees VALUES
    (1, 'John Doe', 5000),
    (2, 'Jane Smith', 6000),
    (3, 'Robert Johnson', 5500),
    (4, 'Emily Williams', 4500),
    (5, 'Michael Brown', 7000);
```

如果需要对所有员工加薪 10%，则可以执行如下语句：

```
UPDATE Employees
SET Salary = Salary * 1.1  -- 将薪水增加10%
WHERE true;
```

如果需要对薪水低于平均薪水的员工加薪 10%，则可以执行如下语句，

```
UPDATE Employees
SET Salary = Salary * 1.1  -- 将薪水增加10%
WHERE Salary < (SELECT AVG(Salary) FROM Employees);
```

也可以使用 CTE 改写上述语句，增加易读性。

```
WITH AvgSalary AS (
    SELECT AVG(Salary) AS AverageSalary
    FROM Employees
)
UPDATE Employees
SET Salary = Salary * 1.1  -- 将薪水增加10%
FROM AvgSalary
```

```
WHERE Employees.Salary < AvgSalary.AverageSalary;
```

">多表 UPDATE

创建表 Accounts 来记录账户信息，向表中插入三行数据。

```
CREATE TABLE Accounts (
    Accounts_id BIGINT NOT NULL,
    Name VARCHAR(26) NOT NULL,
    Sales_person VARCHAR(50) NOT NULL
)
PRIMARY KEY (Accounts_id)
DISTRIBUTED BY HASH (Accounts_id)
PROPERTIES ("replication_num" = "3");

INSERT INTO Accounts VALUES
    (1,'Acme Corporation','John Doe'),
    (2,'Acme Corporation','Robert Johnson'),
    (3,'Acme Corporation','Lily Swift');
```

如果需要给表 Employees 中 Acme Corporation 公司管理帐户的员工加薪 10%，则可以执行如下语句：

```
UPDATE Employees
SET Salary = Salary * 1.1  -- 将薪水增加10%
FROM Accounts
WHERE Accounts.name = 'Acme Corporation'
    AND Employees.Name = Accounts.Sales_person;
```

也可以使用 CTE 改写上述语句，增加易读性。

```
WITH Acme_Accounts as (
    SELECT * from Accounts
    WHERE Accounts.name = 'Acme Corporation'
)
UPDATE Employees SET Salary = Salary * 1.1 -- 将薪水增加10%
FROM Acme_Accounts
WHERE Employees.Name = Acme_Accounts.Sales_person;
```

视图
ALTER VIEW

功能

该语句用于修改一个逻辑视图的定义。

语法

```
ALTER VIEW
[db_name.]view_name
(column1[ COMMENT "col comment"][, column2, ...])
AS query_stmt
```

说明：

1. 逻辑视图中的数据不会存储在物理介质上，在查询时，逻辑视图将作为语句中的子查询，因此，修改逻辑视图的定义等价于修改 query_stmt。
2. query_stmt 为任意支持的 SQL。

示例

修改example_db上的逻辑视图example_view。

```
ALTER VIEW example_db.example_view
(
   c1 COMMENT "column 1",
   c2 COMMENT "column 2",
   c3 COMMENT "column 3"
)
AS SELECT k1, k2, SUM(v1)
FROM example_table
GROUP BY k1, k2;
```

CREATE VIEW

">功能

创建一个视图。
视图（或逻辑视图）是一种虚拟表，其中的数据来自于对其他现有实体表的查询结果。因此，视图无需占用物理存储空间。所有针对视图的查询相当于该视图对应查询语句之上的子查询。
注意：
该操作需要有指定数据库的 CREATE VIEW 权限。

">语法

```
CREATE [OR REPLACE] VIEW [IF NOT EXISTS]
[<database>.]<view_name>
(
    <column_name>[ COMMENT 'column comment']
    [, <column_name>[ COMMENT 'column comment'], ...]
)
[COMMENT 'view comment']
AS <query_statement>
```

参数说明

参数	说明
OR REPLACE	替换已有视图。
database	视图所属的数据库名。
view_name	视图名。
column_name	视图中的列名。请注意，视图中的列和 query_statement 中查询的列的数量必须一致。
COMMENT	视图中的列或视图本身的注释。
query_statement	用于创建视图的查询语句。可以为 StarRocks 支持的任意查询语句。

">使用说明

- 查询视图需要该视图的 SELECT 权限和其对应基表的 SELECT 权限。
- 如果基表变更导致创建视图的查询语句无法执行，则查询该视图时报错。
  示例
  示例一：通过基于表 example_table 的聚合查询在数据库 example_db 中创建名为 example_view 的视图。

```
CREATE VIEW example_db.example_view (k1, k2, k3, v1)
AS
SELECT c1 as k1, k2, k3, SUM(v1) FROM example_table
WHERE k1 = 20160112 GROUP BY k1,k2,k3;
```

示例二：通过基于表 example_table 的聚合查询在数据库 example_db 中创建名为 example_view 的视图，并为视图和其中的列设置注释。

```
CREATE VIEW example_db.example_view
(
   k1 COMMENT 'first key',
   k2 COMMENT 'second key',
   k3 COMMENT 'third key',
   v1 COMMENT 'first value'
)
COMMENT 'my first view'
AS
SELECT c1 as k1, k2, k3, SUM(v1) FROM example_table
WHERE k1 = 20160112 GROUP BY k1,k2,k3;
```

DROP VIEW

">功能

该语句用于删除一个逻辑视图。

">语法

```
DROP VIEW [IF EXISTS] [db_name.]view_name
```

">示例

如果存在，删除 example_db 上的逻辑视图 example_view。

```
DROP VIEW IF EXISTS example_db.example_view;
```

SHOW CREATE VIEW

">功能

查看指定逻辑视图的创建语句 CREATE VIEW。视图创建语句可以帮助您理解视图定义，作为后续修改视图或重建视图的参考。
注意：只有拥有该视图和视图对应基表的 SELECT 权限的用户才可以查看。
从 2.5.4 版本开始，为了兼容 MySQL 标准语法，支持使用 SHOW CREATE VIEW 来查看异步物化视图的创建语句。该语句将物化视图当做普通视图来处理。

">语法

```
SHOW CREATE VIEW [<db_name>.]<view_name>
```

参数说明

参数	必选	说明
db_name	否	数据库名称。如不指定，则默认查看当前数据库中指定视图的创建语句。
view_name	是	视图名称。

返回结果说明

```
+---------+--------------+----------------------+----------------------+
| View    | Create View  | character_set_client | collation_connection |
+---------+--------------+----------------------+----------------------+
```

返回结果中的参数说明如下：

参数	说明

参数	说明
View	视图名称。
CREATE View	视图的创建语句。
character_set_client	客户端连接 StarRocks 服务端使用的字符集。
collation_connection	字符集的校对规则。

">示例

">查看逻辑视图创建语句

创建表 base。

```
CREATE TABLE base (
    k1 date,
    k2 int,
    v1 int sum)
PARTITION BY RANGE(k1)
(
    PARTITION p1 values less than('2020-02-01'),
    PARTITION p2 values less than('2020-03-01')
)
DISTRIBUTED BY HASH(k2)
PROPERTIES( "replication_num"  = "3");
```

在表 base 上创建视图 example_view。

```
CREATE VIEW example_view (k1, k2, v1)
AS SELECT k1, k2, v1 FROM base;
```

查看视图 example_view 的创建语句。

```
SHOW CREATE VIEW example_view;

MySQL [yn_db]> SHOW CREATE VIEW example_view;
+--------------
+----------------------------------------------------------------------------------------------------------------------
+---------------------+---------------------+
| View         | Create View
```

```
| character_set_client | collation_connection |
+--------------
+-----------------------------------------------------------------------------------------------------------------
+---------------------+---------------------+
| example_view | CREATE VIEW `example_view` (k1, k2, v1) COMMENT "VIEW" AS SELECT `yn_db`.`base`.
`k1`, `yn_db`.`base`.`k2`, `yn_db`.`base`.`v1`
FROM `yn_db`.`base`; | utf8                | utf8_general_ci     |
+--------------
+-----------------------------------------------------------------------------------------------------------------
+---------------------+---------------------+
```

">查看物化视图创建语句

在表 base 上创建物化视图 example_mv。

```
CREATE MATERIALIZED VIEW example_mv distributed by hash(k1)
AS SELECT k1 FROM base;
```

查看物化视图 example_mv 的创建语句。

```
SHOW CREATE VIEW example_mv;
+------------+-------------------------------------------------------------------------
+---------------------+---------------------+
| View       | Create View                                                | character_set_client | collation_connec
tion |
+------------+-------------------------------------------------------------------------
+---------------------+---------------------+
| example_mv | CREATE VIEW `example_mv` AS SELECT `yn_db`.`base`.`k1`
FROM `yn_db`.`base` | utf8                | utf8_general_ci     |
+------------+-----------------------------------------------
```

物化视图
ALTER MATERIALIZED VIEW

">功能

此 SQL 语句可以：

- 变更异步物化视图的名称
- 变更异步物化视图的刷新策略
- 变更异步物化视图的状态为 Active 或 Inactive
- 原子替换异步物化视图
- 变更异步物化视图的属性您可以使用此 SQL 语句变更异步物化视图的以下属性：
- partition_ttl_number
- partition_refresh_number
- resource_group
- auto_refresh_partitions_limit
- excluded_trigger_tables
- mv_rewrite_staleness_second
- unique_constraints
- foreign_key_constraints
- colocate_with
- excluded_refresh_tables
- 所有 Session 变量属性。

语法

```
ALTER MATERIALIZED VIEW [db_name.]<mv_name>
  { RENAME [db_name.]<new_mv_name>
  | REFRESH <new_refresh_scheme_desc>
  | ACTIVE | INACTIVE
  | SWAP WITH [db_name.]<mv2_name>
  | SET ( "<key>" = "<value>"[,...]) }
```

参数

参数	必选	说明
mv_name	是	待变更的物化视图的名称。
new_refresh_scheme_desc	否	新的刷新机制，详细信息请见CREATE MATERIALIZED VIEW-参数。
new_mv_name	否	新的物化视图的名称。
ACTIVE	否	将物化视图的状态设置为 Active。如果物化视图的基表发生更改，例如被删除后重新创建，StarRocks 会自动将该物化视图的状态设置为 Inactive，以避免原始元数据与更改后的基表不匹配的情况。状态为 Inactive 的物化视图无法用于查询加速或改写。更改基表后，您可以使用此 SQL 将该物化视图的状态设置为 Active。
INACTIVE	否	将物化视图的状态设置为 Inactive。Inactive 状态的物化视图无法被刷新，但您仍然可以将其作为表直接查询。
SWAP WITH	否	同另一物化视图进行原子替换。替换前，StarRocks 会进行必要的一致性检查。
key	否	待变更的属性的名称，详细信息请见CREATE MATERIALIZED VIEW-参数。 说明 如需更改物化视图的 Session 变量属性，则必须为 Session 属性添加 session. 前缀，例如，session.query_timeout。您无需为非 Session 属性指定前缀，例如，mv_rewrite_staleness_second。
value	否	待变更的属性的值。

示例

示例一：修改物化视图名称

```
ALTER MATERIALIZED VIEW lo_mv1 RENAME lo_mv1_new_name;
```

示例二：修改物化视图刷新间隔

```
ALTER MATERIALIZED VIEW lo_mv2 REFRESH ASYNC EVERY(INTERVAL 1 DAY);
```

示例三：修改物化视图属性，调整物化视图刷新 Timeout 为一小时（默认）。

```
ALTER MATERIALIZED VIEW mv1 SET ("session.query_timeout" = "3600");
```

示例四：修改物化视图状态为 Active。

```
ALTER MATERIALIZED VIEW order_mv ACTIVE;
```

示例五：原子替换物化视图 order_mv 和 order_mv1。

```
ALTER MATERIALIZED VIEW order_mv SWAP WITH order_mv1;
```

示例六：为物化视图刷新开启 Profile（默认开启）。

```
ALTER MATERIALIZED VIEW mv1 SET ("session.enable_profile" = "true");
```

示例七：为物化视图刷新开启中间结果落盘（自 v3.1 起默认开启），并设置落盘模式为 force。

```
-- 为物化视图刷新开启中间结果落盘。
ALTER MATERIALIZED VIEW mv1 SET ("session.enable_spill" = "true");
-- 设置落盘模式为 `force`。
ALTER MATERIALIZED VIEW mv1 SET ("session.spill_mode" = "force");
```

示例八：调整 Optimizer Timeout 为 30 秒（自 v3.3 起为默认值），适用于物化视图查询包含外表或多个 Join。

```
ALTER MATERIALIZED VIEW mv1 SET ("session.new_planner_optimize_timeout" = "30000");
```

示例九：调整物化视图查询改写 Staleness 时间为 600 秒。

```
ALTER MATERIALIZED VIEW mv1 SET ("mv_rewrite_staleness_second" = "600");
```

CANCEL REFRESH MATERIALIZED VIEW

功能

取消异步物化视图的刷新任务。
提示：该操作需要对应物化视图的 REFRESH 权限。

语法

```
CANCEL REFRESH MATERIALIZED VIEW [database_name.]materialized_view_name [FORCE]
```

参数说明

参数	必选	说明
database_name	否	待取消刷新的物化视图所属数据库名称。如果不指定该参数，则默认使用当前数据库。
materialized_view_name	是	待取消刷新的物化视图名称。
FORCE	否	强制取消物化视图刷新任务。

">示例

示例一：取消异步物化视图 lo_mv1 的刷新任务。

```
CANCEL REFRESH MATERIALIZED VIEW lo_mv1;
CANCEL REFRESH MATERIALIZED VIEW lo_mv1 FORCE;
```

CREATE MATERIALIZED VIEW

功能

创建物化视图。关于物化视图适用的场景请参考同步物化视图和异步物化视图。

创建物化视图是一个异步的操作。该命令执行成功即代表创建物化视图的任务提交成功。您可以通过 SHOW ALTER MATERIALIZED VIEW 命令查看当前数据库中同步物化视图的构建状态，或通过查询 INFORMATION_SCHEMA 中的 tasks 和 task_runs 来查看异步物化视图的构建状态。

注意：只有拥有基表所在数据库的 CREATE MATERIALIZED VIEW 权限的用户才可以创建物化视图。

StarRocks 自 v2.4 起支持异步物化视图。异步物化视图与先前版本中的同步物化视图区别主要体现在以下方面：

物化视图	单表聚合	多表关联	查询改写	刷新策略	基表
异步物化视图	是	是	是	- 异步刷新 - 手动刷新	支持多表构建。基表可以来自： - Default Catalog - External Catalog（v2.5） - 已有异步物化视图（v2.5） - 已有视图（v3.1）
同步物化视图（Rollup）	仅部分聚合函数	否	是	导入同步刷新	仅支持基于 Default Catalog 的单表构建

同步物化视图

语法

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] [database.]<mv_name>
[COMMENT ""]
[PROPERTIES ("key"="value", ...)]
AS
<query_statement>
```

参数

mv_name（必填）

物化视图的名称。命名要求如下：

- 必须由字母（a-z 或 A-Z）、数字（0-9）或下划线（_）组成，且只能以字母开头。
- 总长度不能超过 64 个字符。
- 视图名大小写敏感。

COMMENT（选填）
物化视图的注释。注意建立物化视图时 COMMENT 必须在 mv_name 之后，否则创建失败。

query_statement（必填）
创建物化视图的查询语句，其结果即为物化视图中的数据。语法如下：

```
SELECT select_expr[, select_expr ...]
[WHERE where_expr]
[GROUP BY column_name[, column_name ...]]
[ORDER BY column_name[, column_name ...]]
```

- select_expr（必填）
  构建同步物化视图的查询语句。
  - 单列或聚合列：形如 SELECT a, b, c FROM table_a，其中 a、b 和 c 为基表的列名。如果您没有为物化视图指定列名，那么 StarRocks 自动为这些列命名。
  - 表达式：形如 SELECT a+1 AS x, b+2 AS y, c*c AS z FROM table_a，其中 a+1、b+2 和 c*c* 为包含基表列名的表达式，x、y 和 z 为物化视图的列名。
    说明
  - 该参数至少需包含一个单列。
  - 使用聚合函数创建同步物化视图时，必须指定 GROUP BY 子句，并在 select_expr 中指定至少一个 GROUP BY 列。
  - 同步物化视图不支持 JOIN、以及 GROUP BY 的 HAVING 子句。
  - 从 v3.1 开始，每个同步物化视图支持为基表的每一列使用多个聚合函数，支持形如 select b, sum(a), min(a) from table group by b 形式的查询语句。
  - 从 v3.1 开始，同步物化视图支持 SELECT 和聚合函数的复杂表达式，即形如 select b, sum(a + 1) as sum_a1, min(cast (a as bigint)) as min_a from table group by b 或 select abs(b) as col1, a + 1 as col2, cast(a as bigint) as col3 from table 的查询语句。同步物化视图的复杂表达式有以下限制：
    每个复杂表达式必须有一个列名，并且基表所有同步物化视图中的不同复杂表达式的别名必须不同。例如，查询语句 select b, sum(a + 1) as sum_a from table group by b 和select b, sum(a) as sum_a from table group by b 不能同时用于为相同的基表创建同步物化视图，你可以为同一复杂表达式设置多个不同别名。
    可以通过执行 EXPLAIN 来查看您的查询是否被使用复杂表达式创建的同步物化视图改写。更多信息请参见查询分析。

WHERE （选填）

自 v3.1.8 起，同步物化视图支持通过 WHERE 子句筛选数据。

GROUP BY（选填）
构建物化视图查询语句的分组列。如不指定该参数，则默认不对数据进行分组。

ORDER BY（选填）
构建物化视图查询语句的排序列。

- 排序列的声明顺序必须和 select_expr 中列声明顺序一致。
- 如果查询语句中包含分组列，则排序列必须和分组列一致。
- 如果不指定排序列，则系统根据以下规则自动补充排序列：
  如果物化视图是聚合类型，则所有的分组列自动补充为排序列。
  如果物化视图是非聚合类型，则系统根据前缀列自动选择排序列。

查询同步物化视图

因为同步物化视图本质上是基表的索引而不是物理表，所以您只能使用 Hint [*SYNC_MV*] 查询同步物化视图：

```
-- 请勿省略 Hint 中的括号[]。
SELECT * FROM <mv_name> [_SYNC_MV_];
```

注意
目前，StarRocks 会自动为同步物化视图中的列生成名称。您为同步物化视图中的列指定的 Alias 将无法生效。

">同步物化视图查询自动改写

使用同步物化视图查询时，原始查询语句将会被自动改写并用于查询物化视图中保存的中间结果。
下表展示了原始查询聚合函数和构建同步物化视图用到的聚合函数的匹配关系。您可以根据业务场景选择对应的聚合函数构建同步物化视图。

原始查询聚合函数	物化视图构建聚合函数
sum	sum
min	min
max	max
count	count
bitmap_union, bitmap_union_count, count(distinct)	bitmap_union
hll_raw_agg, hll_union_agg, ndv, approx_count_distinct	hll_union

">异步物化视图

语法

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] [database.]<mv_name>
[COMMENT ""]
-- 必须至少指定 `distribution_desc` 和 `refresh_scheme` 其中之一。
-- distribution_desc
[DISTRIBUTED BY HASH(<bucket_key>[,<bucket_key2> ...]) [BUCKETS <bucket_number>]]
-- refresh_desc
[REFRESH
-- refresh_moment
    [IMMEDIATE | DEFERRED]
-- refresh_scheme
    [ASYNC | ASYNC [START (<start_time>)] EVERY (INTERVAL <refresh_interval>) | MANUAL]
]
-- partition_expression
[PARTITION BY
    {<date_column> | date_trunc(fmt, <date_column>)}
]
-- order_by_expression
[ORDER BY (<sort_key>)]
[PROPERTIES ("key"="value", ...)]
AS
<query_statement>
```

参数

mv_name（必填）

物化视图的名称。命名要求如下：

- 必须由字母（a-z 或 A-Z）、数字（0-9）或下划线（_）组成，且只能以字母开头。
- 总长度不能超过 64 个字符。
- 视图名大小写敏感。

  注意：同一张基表可以创建多个异步物化视图，但同一数据库内的异步物化视图名称不可重复。

COMMENT（选填）

物化视图的注释。注意建立物化视图时 COMMENT 必须在 mv_name 之后，否则创建失败。

distribution_desc（选填）

异步物化视图的分桶方式，包括哈希分桶和随机分桶（自 3.1 版本起）。如不指定该参数，StarRocks 使用随机分桶方式，并自动设置分桶数量。

说明：创建异步物化视图时必须至少指定 distribution_desc 和 refresh_scheme 其中之一。

- 哈希分桶：
  语法

```
DISTRIBUTED BY HASH (<bucket_key1>[,<bucket_key2> ...]) [BUCKETS <bucket_number>]
```

说明：自 2.5.7 版本起，StarRocks 支持在建表和新增分区时自动设置分桶数量 (BUCKETS)，您无需手动设置分桶数量。更多信息，请参见 设置分桶数量。

- 随机分桶：
  如果选择随机分桶方式，并且自动设置分桶数量，则无需指定 distribution_desc。如果您需要手动设置分桶数，请使用以下语法：

```
DISTRIBUTED BY RANDOM BUCKETS <bucket_number>
```

注意：采用随机分桶方式的异步物化视图不支持设置 Colocation Group。

refresh_moment（选填）
物化视图的刷新时刻。默认值：IMMEDIATE。有效值：

- IMMEDIATE：异步物化视图创建成功后立即刷新。
- DEFERRED：异步物化视图创建成功后不进行刷新。您可以通过手动调用或创建定时任务触发刷新。

refresh_scheme（选填）
说明：创建异步物化视图时必须至少指定 distribution_desc 和 refresh_scheme 其中之一。

物化视图的刷新方式。该参数支持如下值：

- ASYNC: 自动刷新模式。每当基表数据发生变化时，物化视图会自动刷新。
- ASYNC [START ()] EVERY(INTERVAL): 定时刷新模式。物化视图将按照定义的间隔定时刷新。您可以使用 DAY（天）、HOUR（小时）、MINUTE（分钟）和 SECOND（秒）作为单位指定间隔，格式为 EVERY (interval n day/hour/minute/second)。默认值为 10 MINUTE（10 分钟）。您还可以进一步指定刷新起始时间，格式为 START('yyyy-MM-dd hh:mm:ss')。如未指定起始时间，默认使用当前时间。示例：ASYNC START ('2023-09-12 16:30:25') EVERY (INTERVAL 5 MINUTE)。
- MANUAL: 手动刷新模式。除非手动触发刷新任务，否则物化视图不会刷新。
  如果不指定该参数，则默认使用 MANUAL 方式。

partition_expression（选填）
异步物化视图的分区表达式。目前仅支持在创建异步物化视图时使用一个分区表达式。
注意：自 v3.3.3 起，StarRocks 支持创建基于 List 分区策略的异步物化视图。

- 可以基于使用 List 分区或表达式分区策略创建的表来创建 List 分区的物化视图。
- 目前，当使用 List 分区策略创建物化视图时，您只能指定一个分区键。如果基表有多个分区键，您只能选择其

中一个分区键。
- 使用 List 分区策略的物化视图的刷新行为和查询改写逻辑与使用 Range 分区策略的物化视图一致。
该参数支持如下值：
- date_column：用于分区的列的名称。形如 PARTITION BY dt，表示按照 dt 列进行分区。
- date_trunc 函数：形如 PARTITION BY date_trunc("MONTH", dt)，表示将 dt 列截断至以月为单位进行分区。date_trunc 函数支持截断的单位包括 YEAR、MONTH、DAY、HOUR 以及 MINUTE。
- str2date 函数：用于将基表的 STRING 类型分区键转化为物化视图的分区键所需的日期类型。PARTITION BY str2date(dt, "%Y%m%d") 表示 dt 列是一个 STRING 类型日期，其日期格式为 "%Y%m%d"。str2date 函数支持多种日期格式。更多信息，参考str2date。自 v3.1.4 起支持。
- time_slice 函数：从 v3.1 开始，您可以进一步使用 time_slice 函数根据指定的时间粒度周期，将给定的时间转化到其所在的时间粒度周期的起始或结束时刻，例如 PARTITION BY date_trunc("MONTH", time_slice(dt, INTERVAL 7 DAY))，其中 time_slice 的时间粒度必须比 date_trunc 的时间粒度更细。你可以使用它们来指定一个比分区键更细时间粒度的 GROUP BY 列，例如，GROUP BY time_slice(dt, INTERVAL 1 MINUTE) PARTITION BY date_trunc('DAY', ts)。
如不指定该参数，则默认物化视图为无分区。

order_by_expression（选填）
异步物化视图的排序键。如不指定该参数，StarRocks 从 SELECT 列中选择部分前缀作为排序键，例如：SELECT a, b, c, d 中, 排序列可能为 a 和 b。此参数自 StarRocks 3.0 起支持。

PROPERTIES（选填）
异步物化视图的属性。可以使用 ALTER MATERIALIZED VIEW 修改已有异步物化视图的属性。

- session.: 如果您想要更改与物化视图相关的 Session 变量属性，必须在属性前添加 session. 前缀，例如，session.query_timeout。对于非 Session 属性，例如，mv_rewrite_staleness_second，则无需指定前缀。
- replication_num：设置物化视图副本数量。
- storage_medium：存储介质类型。有效值：HDD 和 SSD。
- storage_cooldown_time: 当设置存储介质为 SSD 时，指定该分区在该时间点之后从 SSD 降级到 HDD，设置的时间必须大于当前时间。如不指定该属性，默认不进行自动降冷。取值格式为："yyyy-MM-dd HH:mm:ss"。
- partition_ttl: 物化视图分区的生存时间 (TTL)。数据在指定的时间范围内的分区将被保留，过期的分区将被自动删除。单位：YEAR、MONTH、DAY、HOUR 和 MINUTE。例如，您可以将此属性设置为 2 MONTH（2 个月）。建议您使用此属性，不推荐使用 partition_ttl_number。该属性自 v3.1.5 起支持。
- partition_ttl_number：需要保留的最近的物化视图分区数量。对于分区开始时间小于当前时间的分区，当数量超过该值之后，多余的分区将会被删除。StarRocks 将根据 FE 配置项

dynamic_partition_check_interval_seconds 中的时间间隔定期检查物化视图分区，并自动删除过期分区。在动态分区场景下，提前创建的未来分区将不会被纳入 TTL 考虑。默认值：-1。当值为 -1 时，将保留物化视图所有分区。

- partition_refresh_number：单次刷新中，最多刷新的分区数量。如果需要刷新的分区数量超过该值，StarRocks 将拆分这次刷新任务，并分批完成。仅当前一批分区刷新成功时，StarRocks 会继续刷新下一批分区，直至所有分区刷新完成。如果其中有分区刷新失败，将不会产生后续的刷新任务。当值为 -1 时，将不会拆分刷新任务。自 v3.3 起，默认值由 -1 变为 1，表示 StarRocks 每次只刷新一个分区。

- excluded_trigger_tables：在此项属性中列出的基表，其数据产生变化时不会触发对应物化视图自动刷新。该参数仅针对导入触发式刷新，通常需要与属性 auto_refresh_partitions_limit 搭配使用。形式：[db_name.]table_name。默认值为空字符串。当值为空字符串时，任意的基表数据变化都将触发对应物化视图刷新。

- auto_refresh_partitions_limit：当触发物化视图刷新时，需要刷新的最近的物化视图分区数量。您可以通过该属性限制刷新的范围，降低刷新代价，但因为仅有部分分区刷新，有可能导致物化视图数据与基表无法保持一致。默认值：-1。当参数值为 -1 时，StarRocks 将刷新所有分区。当参数值为正整数 N 时，StarRocks 会将已存在的分区按时间先后排序，并刷新当前分区和 N-1 个历史分区。如果分区数不足 N，则刷新所有已存在的分区。如果物化视图存在提前创建的未来分区，将会刷新所有提前创建的分区。

- mv_rewrite_staleness_second：如果当前物化视图的上一次刷新在此属性指定的时间间隔内，则此物化视图可直接用于查询改写，无论基表数据是否更新。如果上一次刷新时间早于此属性指定的时间间隔，StarRocks 通过检查基表数据是否变更决定该物化视图能否用于查询改写。单位：秒。该属性自 v3.0 起支持。

- colocate_with：异步物化视图的 Colocation Group。更多信息请参阅 Colocate Join。该属性自 v3.0 起支持。

- unique_constraints 和 foreign_key_constraints：创建 View Delta Join 查询改写的异步物化视图时的 Unique Key 约束和外键约束。更多信息请参阅 异步物化视图 - 基于 View Delta Join 场景改写查询。该属性自 v3.0 起支持。

- excluded_refresh_tables：在此项属性中列出的基表，其数据产生变化时不会触发该表的数据刷新到物化视图。通常需要与属性 excluded_trigger_tables 搭配使用。形式：[db_name.]table_name。默认值为空字符串。当值为空字符串时，任意的基表数据变化都将触发对应物化视图刷新。
  注意：Unique Key 约束和外键约束仅用于查询改写。导入数据时，不保证进行外键约束校验。您必须确保导入的数据满足约束条件。

- resource_group: 为物化视图刷新任务设置资源组。默认值为 default_mv_wg，即一个系统定义的，专门用于物化视图刷新的资源组。该资源组的 cpu_core_limit 为 1，mem_limit 为 0.8。更多关于资源组信息，请参考

资源隔离。

- query_rewrite_consistency: 指定当前异步物化视图的查询改写规则。该属性自 v3.2 起支持。有效值：
  - disable：禁用基于该异步物化视图进行自动查询改写。
  - checked（默认值）：仅在物化视图满足时效性要求时启用自动查询改写，即：
    如果未指定 mv_rewrite_staleness_second，则只有当物化视图的数据与所有基表中的数据一致时，才可以将其用于查询改写。
    如果指定了 mv_rewrite_staleness_second，则只有在其最后刷新在 staleness 时间间隔内时，才可以将物化视图用于查询改写。
  - loose：直接启用自动查询改写，无需进行一致性检查。

- storage_volume：如果您使用存算分离集群，则需要指定创建物化视图的 Storage Volume 名称。该属性自 v3.1 版本起支持。如果未指定该属性，则使用默认 Storage Volume。示例："storage_volume" = "def_volume"。

- force_external_table_query_rewrite: 是否启用基于 External Catalog 的物化视图的查询改写。该属性自 v3.2 起支持。有效值：
  - true（自 v3.3 变为默认值）：启用基于 External Catalog 的物化视图的查询改写。
  - false：禁用基于 External Catalog 的物化视图的查询改写。
    由于无法保证基表和基于 External Catalog 的物化视图之间的数据强一致，因此默认情况下禁用此功能。启用此功能时，物化视图将根据 query_rewrite_consistency 中指定的规则改写查询。

- enable_query_rewrite：是否使用物化视图进行查询改写。当存在大量物化视图时，基于物化视图的查询改写可能会影响优化器的耗时。通过此属性，您可以控制是否允许使用物化视图进行查询改写。该功能自 v3.3.0 起支持。有效值：
  - default（默认）：系统将不会针对物化视图执行语义检查，但只有 SPJG 类型的物化视图可以用于查询改写。请注意，如果启用了基于文本的查询改写，非 SPJG 类型的物化视图也可以用于查询改写。
  - true：系统将在创建或修改物化视图时执行语义检查。如果物化视图不符合查询改写的条件（即，物化视图的定义不是 SPJG 类型的查询），则会返回失败信息。
  - false：物化视图将不会用于查询改写。

- [Preview] transparent_mv_rewrite_mode：为 直接针对物化视图的查询 指定透明改写模式。此功能从 v3.3.0 版本开始支持。有效值如下：
  - false（默认，与早期版本行为兼容）：直接针对物化视图的查询不会被改写，仅返回物化视图中现有的数据。根据物化视图的刷新状态（数据一致性），其结果可能与直接执行物化视图定义查询的结果不同。
  - true：直接针对物化视图的查询将被改写，并返回最新数据，结果与物化视图定义查询的一致。请注

意，当物化视图处于失效（Inactive）状态或不支持透明查询改写时，这些查询将路由至物化视图定义查询执行。
    - transparent_or_error：直接针对物化视图的查询将在符合条件时可以被改写。如果物化视图处于失效（Inactive）状态或不支持透明查询改写，将返回错误。
    - transparent_or_default：直接针对物化视图的查询将在符合条件时可以被改写。如果物化视图处于失效（Inactive）状态或不支持透明查询改写，将返回物化视图中现有的数据。

query_statement（必填）
创建异步物化视图的查询语句，其结果即为异步物化视图中的数据。从 v3.1.6 版本开始，StarRocks 支持使用 Common Table Expression (CTE) 创建异步物化视图。

">查询异步物化视图

异步物化视图本质是一张实体表。您可以将其作为普通表进行任何除直接导入数据以外的操作。

支持数据类型

- 基于 StarRocks 内部数据目录（Default Catalog）创建的异步物化视图支持以下数据类型：
    - 日期类型：DATE、DATETIME
    - 字符串类型：CHAR、VARCHAR
    - 数值类型：BOOLEAN、TINYINT、SMALLINT、INT、BIGINT、LARGEINT、FLOAT、DOUBLE、DECIMAL、PERCENTILE
    - 半结构化类型：ARRAY、JSON、MAP（自 v3.1 起）、STRUCT（自 v3.1 起）
    - 其他类型：BITMAP、HLL
      说明：自 v2.4.5 起支持 BITMAP、HLL 以及 PERCENTILE。
- 基于 StarRocks 外部数据目录（External Catalog）创建的异步物化视图支持以下数据类型：
- Hive Catalog
    - 数值类型：INT/INTEGER、BIGINT、DOUBLE、FLOAT、DECIMAL
    - 日期类型：TIMESTAMP
    - 字符串类型：STRING、VARCHAR、CHAR
    - 半结构化类型：ARRAY
- Iceberg Catalog
    - 数值类型：BOOLEAN、INT、LONG、FLOAT、DOUBLE、DECIMAL(P, S)
    - 日期类型：DATE、TIME、TIMESTAMP
    - 字符串类型：STRING、UUID、FIXED(L)、BINARY
    - 半结构化类型：LIST

">注意事项

- 版本暂时不支持同时创建多个物化视图。仅当当前创建任务完成时，方可执行下一个创建任务。
- 关于同步物化视图：
  - 同步物化视图仅支持单列聚合函数，不支持形如 sum(a+b) 的查询语句。
  - 同步物化视图仅支持对同一列数据使用一种聚合函数，不支持形如 SELECT sum(a), min(a) from table 的查询语句。
  - 同步物化视图中使用聚合函数需要与 GROUP BY 语句一起使用，且 SELECT 的列中至少包含一个分组列。
  - 同步物化视图创建语句不支持 JOIN 以及 GROUP BY 的 HAVING 子句。
  - 使用 ALTER TABLE DROP COLUMN 删除基表中特定列时，需要保证该基表所有同步物化视图中不包含被删除列，否则无法进行删除操作。如果必须删除该列，则需要将所有包含该列的同步物化视图删除，然后进行删除列操作。
  - 为一张表创建过多的同步物化视图会影响导入的效率。导入数据时，同步物化视图和基表数据将同步更新，如果一张基表包含 n 个物化视图，向基表导入数据时，其导入效率大约等同于导入 n 张表，数据导入的速度会变慢。
- 关于嵌套异步物化视图：
  - 每个异步物化视图的刷新方式仅影响当前物化视图。
  - 当前 StarRocks 不对嵌套层数进行限制。生产环境中建议嵌套层数不超过三层。
- 关于外部数据目录异步物化视图：
  - 外部数据目录物化视图仅支持异步定时刷新和手动刷新。
  - 物化视图中的数据不保证与外部数据目录的数据强一致。
  - 目前暂不支持基于资源（Resource）构建物化视图。
  - StarRocks 目前无法感知外部数据目录基表数据是否发生变动，所以每次刷新会默认刷新所有分区。您可以通过手动刷新方式指定刷新部分分区。

">示例

同步物化视图示例

基表结构为：

```
mysql> desc duplicate_table;
+-------+--------+------+------+---------+-------+
| Field | Type   | Null | Key  | Default | Extra |
+-------+--------+------+------+---------+-------+
| k1    | INT    | Yes  | true | N/A     |       |
| k2    | INT    | Yes  | true | N/A     |       |
| k3    | BIGINT | Yes  | true | N/A     |       |
```

```
| k4    | BIGINT | Yes  | true | N/A     |       |
+-------+--------+------+------+---------+-------+
```

1. 创建一个仅包含原始表（k1，k2）列的物化视图。

```
create materialized view k1_k2 as
select k1, k2 from duplicate_table;
```

物化视图的 schema 如下图，物化视图仅包含两列 k1、k2 且不带任何聚合。

```
+-----------------+-------+--------+------+------+---------+-------+
| IndexName       | Field | Type   | Null | Key  | Default | Extra |
+-----------------+-------+--------+------+------+---------+-------+
| k1_k2           | k1    | INT    | Yes  | true | N/A     |       |
|                 | k2    | INT    | Yes  | true | N/A     |       |
+-----------------+-------+--------+------+------+---------+-------+
```

2. 创建一个以 k2 为排序列的物化视图。

```
create materialized view k2_order as
select k2, k1 from duplicate_table order by k2;
```

物化视图的 schema 如下图，物化视图仅包含两列 k2、k1，其中 k2 列为排序列，不带任何聚合。

```
+-----------------+-------+--------+------+-------+---------+-------+
| IndexName       | Field | Type   | Null | Key   | Default | Extra |
+-----------------+-------+--------+------+-------+---------+-------+
| k2_order        | k2    | INT    | Yes  | true  | N/A     |       |
|                 | k1    | INT    | Yes  | false | N/A     | NONE  |
+-----------------+-------+--------+------+-------+---------+-------+
```

3. 创建一个以 k1，k2 分组，k3 列为 SUM 聚合的物化视图。

```
create materialized view k1_k2_sumk3 as
select k1, k2, sum(k3) from duplicate_table group by k1, k2;
```

物化视图的 schema 如下图，物化视图包含两列 k1、k2，sum(k3) 其中 k1、k2 为分组列，sum(k3) 为根据 k1、k2 分组后的 k3 列的求和值。
由于物化视图没有声明排序列，且物化视图带聚合数据，系统默认补充分组列 k1、k2 为排序列。

```
+-----------------+-------+--------+------+-------+---------+-------+
| IndexName       | Field | Type   | Null | Key   | Default | Extra |
+-----------------+-------+--------+------+-------+---------+-------+
```

```
| k1_k2_sumk3    | k1    | INT    | Yes  | true  | N/A     |       |
|                | k2    | INT    | Yes  | true  | N/A     |       |
|                | k3    | BIGINT | Yes  | false | N/A     | SUM   |
+----------------+-------+--------+------+-------+---------+-------+
```

4. 创建一个去除重复行的物化视图。

```
create materialized view deduplicate as
select k1, k2, k3, k4 from duplicate_table group by k1, k2, k3, k4;
```

物化视图 schema 如下图，物化视图包含 k1、k2、k3、k4 列，且不存在重复行。

```
+----------------+-------+--------+------+-------+---------+-------+
| IndexName      | Field | Type   | Null | Key   | Default | Extra |
+----------------+-------+--------+------+-------+---------+-------+
| deduplicate    | k1    | INT    | Yes  | true  | N/A     |       |
|                | k2    | INT    | Yes  | true  | N/A     |       |
|                | k3    | BIGINT | Yes  | true  | N/A     |       |
|                | k4    | BIGINT | Yes  | true  | N/A     |       |
+----------------+-------+--------+------+-------+---------+-------+
```

5. 创建一个不声明排序列的非聚合型物化视图。
   all_type_table 的 schema 如下：

```
+-------+--------------+------+-------+---------+-------+
| Field | Type         | Null | Key   | Default | Extra |
+-------+--------------+------+-------+---------+-------+
| k1    | TINYINT      | Yes  | true  | N/A     |       |
| k2    | SMALLINT     | Yes  | true  | N/A     |       |
| k3    | INT          | Yes  | true  | N/A     |       |
| k4    | BIGINT       | Yes  | true  | N/A     |       |
| k5    | DECIMAL(9,0) | Yes  | true  | N/A     |       |
| k6    | DOUBLE       | Yes  | false | N/A     | NONE  |
| k7    | VARCHAR(20)  | Yes  | false | N/A     | NONE  |
+-------+--------------+------+-------+---------+-------+
```

物化视图包含 k3、k4、k5、k6、k7 列，且不声明排序列，则创建语句如下：

```
create materialized view mv_1 as
select k3, k4, k5, k6, k7 from all_type_table;
```

系统默认补充的排序列为 k3、k4、k5 三列。这三列类型的字节数之和为 4(INT) + 8(BIGINT) + 16(DECIMAL) = 28 < 36。所以补充的是这三列作为排序列。 物化视图的 schema 如下，可以看到其中 k3、k4、k5 列的 key 字段为 true，也就是排序列。k6、k7 列的 key 字段为 false，也就是非排序列。

```
+---------------+-------+-------------+------+-------+---------+-------+
| IndexName     | Field | Type        | Null | Key   | Default | Extra |
+---------------+-------+-------------+------+-------+---------+-------+
| mv_1          | k3    | INT         | Yes  | true  | N/A     |       |
|               | k4    | BIGINT      | Yes  | true  | N/A     |       |
|               | k5    | DECIMAL(9,0) | Yes  | true  | N/A     |       |
|               | k6    | DOUBLE      | Yes  | false | N/A     | NONE  |
|               | k7    | VARCHAR(20)  | Yes  | false | N/A     | NONE  |
+---------------+-------+-------------+------+-------+---------+-------+
```

6. 使用 WHERE 子句和复杂表达式创建同步物化视图。

```
-- 创建基表 user_event
CREATE TABLE user_event (
    ds date   NOT NULL,
    id  varchar(256)    NOT NULL,
    user_id int DEFAULT NULL,
    user_id1    varchar(256)    DEFAULT NULL,
    user_id2    varchar(256)    DEFAULT NULL,
    column_01   int DEFAULT NULL,
    column_02   int DEFAULT NULL,
    column_03   int DEFAULT NULL,
    column_04   int DEFAULT NULL,
    column_05   int DEFAULT NULL,
    column_06   DECIMAL(12,2)   DEFAULT NULL,
    column_07   DECIMAL(12,3)   DEFAULT NULL,
    column_08   JSON   DEFAULT NULL,
    column_09   DATETIME    DEFAULT NULL,
    column_10   DATETIME    DEFAULT NULL,
    column_11   DATE    DEFAULT NULL,
    column_12   varchar(256)    DEFAULT NULL,
    column_13   varchar(256)    DEFAULT NULL,
    column_14   varchar(256)    DEFAULT NULL,
    column_15   varchar(256)    DEFAULT NULL,
    column_16   varchar(256)    DEFAULT NULL,
    column_17   varchar(256)    DEFAULT NULL,
    column_18   varchar(256)    DEFAULT NULL,
    column_19   varchar(256)    DEFAULT NULL,
    column_20   varchar(256)    DEFAULT NULL,
    column_21   varchar(256)    DEFAULT NULL,
    column_22   varchar(256)    DEFAULT NULL,
    column_23   varchar(256)    DEFAULT NULL,
    column_24   varchar(256)    DEFAULT NULL,
    column_25   varchar(256)    DEFAULT NULL,
    column_26   varchar(256)    DEFAULT NULL,
```

```
    column_27   varchar(256)    DEFAULT NULL,
    column_28   varchar(256)    DEFAULT NULL,
    column_29   varchar(256)    DEFAULT NULL,
    column_30   varchar(256)    DEFAULT NULL,
    column_31   varchar(256)    DEFAULT NULL,
    column_32   varchar(256)    DEFAULT NULL,
    column_33   varchar(256)    DEFAULT NULL,
    column_34   varchar(256)    DEFAULT NULL,
    column_35   varchar(256)    DEFAULT NULL,
    column_36   varchar(256)    DEFAULT NULL,
    column_37   varchar(256)    DEFAULT NULL
)
PARTITION BY date_trunc("day", ds)
DISTRIBUTED BY hash(id);

-- 使用 WHERE 子句和复杂表达式创建同步物化视图
CREATE MATERIALIZED VIEW test_mv1
AS
SELECT
ds,
column_19,
column_36,
sum(column_01) as column_01_sum,
bitmap_union(to_bitmap( user_id)) as user_id_dist_cnt,
bitmap_union(to_bitmap(case when column_01 > 1 and column_34 IN ('1','34')   then user_id2 else null
end)) as filter_dist_cnt_1,
bitmap_union(to_bitmap( case when column_02 > 60 and column_35 IN ('11','13') then  user_id2 else n
ull end)) as filter_dist_cnt_2,
bitmap_union(to_bitmap(case when column_03 > 70 and column_36 IN ('21','23') then  user_id2 else n
ull end)) as filter_dist_cnt_3,
bitmap_union(to_bitmap(case when column_04 > 20 and column_27 IN ('31','27') then  user_id2 else n
ull end)) as filter_dist_cnt_4,
bitmap_union(to_bitmap( case when column_05 > 90 and column_28 IN ('41','43') then  user_id2 else n
ull end)) as filter_dist_cnt_5
FROM user_event
WHERE ds >= '2023-11-02'
GROUP BY
ds,
column_19,
column_36;
```

">异步物化视图示例

以下示例基于下列基表。

```
CREATE TABLE `lineorder` (
  `lo_orderkey` int(11) NOT NULL COMMENT "",
  `lo_linenumber` int(11) NOT NULL COMMENT "",
  `lo_custkey` int(11) NOT NULL COMMENT "",
  `lo_partkey` int(11) NOT NULL COMMENT "",
  `lo_suppkey` int(11) NOT NULL COMMENT "",
  `lo_orderdate` int(11) NOT NULL COMMENT "",
  `lo_orderpriority` varchar(16) NOT NULL COMMENT "",
  `lo_shippriority` int(11) NOT NULL COMMENT "",
  `lo_quantity` int(11) NOT NULL COMMENT "",
  `lo_extendedprice` int(11) NOT NULL COMMENT "",
  `lo_ordtotalprice` int(11) NOT NULL COMMENT "",
  `lo_discount` int(11) NOT NULL COMMENT "",
  `lo_revenue` int(11) NOT NULL COMMENT "",
  `lo_supplycost` int(11) NOT NULL COMMENT "",
  `lo_tax` int(11) NOT NULL COMMENT "",
  `lo_commitdate` int(11) NOT NULL COMMENT "",
  `lo_shipmode` varchar(11) NOT NULL COMMENT ""
) ENGINE=OLAP
DUPLICATE KEY(`lo_orderkey`)
COMMENT "OLAP"
PARTITION BY RANGE(`lo_orderdate`)
(PARTITION p1 VALUES [("-2147483648"), ("19930101")),
PARTITION p2 VALUES [("19930101"), ("19940101")),
PARTITION p3 VALUES [("19940101"), ("19950101")),
PARTITION p4 VALUES [("19950101"), ("19960101")),
PARTITION p5 VALUES [("19960101"), ("19970101")),
PARTITION p6 VALUES [("19970101"), ("19980101")),
PARTITION p7 VALUES [("19980101"), ("19990101")))
DISTRIBUTED BY HASH(`lo_orderkey`);

CREATE TABLE IF NOT EXISTS `customer` (
  `c_custkey` int(11) NOT NULL COMMENT "",
  `c_name` varchar(26) NOT NULL COMMENT "",
  `c_address` varchar(41) NOT NULL COMMENT "",
  `c_city` varchar(11) NOT NULL COMMENT "",
  `c_nation` varchar(16) NOT NULL COMMENT "",
  `c_region` varchar(13) NOT NULL COMMENT "",
  `c_phone` varchar(16) NOT NULL COMMENT "",
  `c_mktsegment` varchar(11) NOT NULL COMMENT ""
) ENGINE=OLAP
DUPLICATE KEY(`c_custkey`)
COMMENT "OLAP"
DISTRIBUTED BY HASH(`c_custkey`);

CREATE TABLE IF NOT EXISTS `dates` (
```

```
  `d_datekey` int(11) NOT NULL COMMENT "",
  `d_date` varchar(20) NOT NULL COMMENT "",
  `d_dayofweek` varchar(10) NOT NULL COMMENT "",
  `d_month` varchar(11) NOT NULL COMMENT "",
  `d_year` int(11) NOT NULL COMMENT "",
  `d_yearmonthnum` int(11) NOT NULL COMMENT "",
  `d_yearmonth` varchar(9) NOT NULL COMMENT "",
  `d_daynuminweek` int(11) NOT NULL COMMENT "",
  `d_daynuminmonth` int(11) NOT NULL COMMENT "",
  `d_daynuminyear` int(11) NOT NULL COMMENT "",
  `d_monthnuminyear` int(11) NOT NULL COMMENT "",
  `d_weeknuminyear` int(11) NOT NULL COMMENT "",
  `d_sellingseason` varchar(14) NOT NULL COMMENT "",
  `d_lastdayinweekfl` int(11) NOT NULL COMMENT "",
  `d_lastdayinmonthfl` int(11) NOT NULL COMMENT "",
  `d_holidayfl` int(11) NOT NULL COMMENT "",
  `d_weekdayfl` int(11) NOT NULL COMMENT ""
) ENGINE=OLAP
DUPLICATE KEY(`d_datekey`)
COMMENT "OLAP"
DISTRIBUTED BY HASH(`d_datekey`);

CREATE TABLE IF NOT EXISTS `supplier` (
  `s_suppkey` int(11) NOT NULL COMMENT "",
  `s_name` varchar(26) NOT NULL COMMENT "",
  `s_address` varchar(26) NOT NULL COMMENT "",
  `s_city` varchar(11) NOT NULL COMMENT "",
  `s_nation` varchar(16) NOT NULL COMMENT "",
  `s_region` varchar(13) NOT NULL COMMENT "",
  `s_phone` varchar(16) NOT NULL COMMENT ""
) ENGINE=OLAP
DUPLICATE KEY(`s_suppkey`)
COMMENT "OLAP"
DISTRIBUTED BY HASH(`s_suppkey`);

CREATE TABLE IF NOT EXISTS `part` (
  `p_partkey` int(11) NOT NULL COMMENT "",
  `p_name` varchar(23) NOT NULL COMMENT "",
  `p_mfgr` varchar(7) NOT NULL COMMENT "",
  `p_category` varchar(8) NOT NULL COMMENT "",
  `p_brand` varchar(10) NOT NULL COMMENT "",
  `p_color` varchar(12) NOT NULL COMMENT "",
  `p_type` varchar(26) NOT NULL COMMENT "",
  `p_size` int(11) NOT NULL COMMENT "",
  `p_container` varchar(11) NOT NULL COMMENT ""
) ENGINE=OLAP
DUPLICATE KEY(`p_partkey`)
```

```
COMMENT "OLAP"
DISTRIBUTED BY HASH(`p_partkey`);

create table orders (
    dt date NOT NULL,
    order_id bigint NOT NULL,
    user_id int NOT NULL,
    merchant_id int NOT NULL,
    good_id int NOT NULL,
    good_name string NOT NULL,
    price int NOT NULL,
    cnt int NOT NULL,
    revenue int NOT NULL,
    state tinyint NOT NULL
)
PRIMARY KEY (dt, order_id)
PARTITION BY RANGE(`dt`)
( PARTITION p20210820 VALUES [('2021-08-20'), ('2021-08-21')),
PARTITION p20210821 VALUES [('2021-08-21'), ('2021-08-22')) )
DISTRIBUTED BY HASH(order_id)
PROPERTIES (
    "replication_num" = "3",
    "enable_persistent_index" = "true"
);
```

示例一：从源表创建非分区物化视图

```
CREATE MATERIALIZED VIEW lo_mv1
DISTRIBUTED BY HASH(`lo_orderkey`)
REFRESH ASYNC
AS
select
    lo_orderkey,
    lo_custkey,
    sum(lo_quantity) as total_quantity,
    sum(lo_revenue) as total_revenue,
    count(lo_shipmode) as shipmode_count
from lineorder
group by lo_orderkey, lo_custkey
order by lo_orderkey;
```

示例二：从源表创建分区物化视图

```
CREATE MATERIALIZED VIEW lo_mv2
PARTITION BY `lo_orderdate`
DISTRIBUTED BY HASH(`lo_orderkey`)
```

```
REFRESH ASYNC START('2023-07-01 10:00:00') EVERY (interval 1 day)
AS
select
    lo_orderkey,
    lo_orderdate,
    lo_custkey,
    sum(lo_quantity) as total_quantity,
    sum(lo_revenue) as total_revenue,
    count(lo_shipmode) as shipmode_count
from lineorder
group by lo_orderkey, lo_orderdate, lo_custkey
order by lo_orderkey;

# 使用 date_trunc 函数将 `dt` 列截断至以月为单位进行分区。
CREATE MATERIALIZED VIEW order_mv1
PARTITION BY date_trunc('month', `dt`)
DISTRIBUTED BY HASH(`order_id`)
REFRESH ASYNC START('2023-07-01 10:00:00') EVERY (interval 1 day)
AS
select
    dt,
    order_id,
    user_id,
    sum(cnt) as total_cnt,
    sum(revenue) as total_revenue,
    count(state) as state_count
from orders
group by dt, order_id, user_id;
```

示例三：创建异步物化视图

```
CREATE MATERIALIZED VIEW flat_lineorder
DISTRIBUTED BY HASH(`lo_orderkey`)
REFRESH MANUAL
AS
SELECT
    l.LO_ORDERKEY AS LO_ORDERKEY,
    l.LO_LINENUMBER AS LO_LINENUMBER,
    l.LO_CUSTKEY AS LO_CUSTKEY,
    l.LO_PARTKEY AS LO_PARTKEY,
    l.LO_SUPPKEY AS LO_SUPPKEY,
    l.LO_ORDERDATE AS LO_ORDERDATE,
    l.LO_ORDERPRIORITY AS LO_ORDERPRIORITY,
    l.LO_SHIPPRIORITY AS LO_SHIPPRIORITY,
    l.LO_QUANTITY AS LO_QUANTITY,
    l.LO_EXTENDEDPRICE AS LO_EXTENDEDPRICE,
    l.LO_ORDTOTALPRICE AS LO_ORDTOTALPRICE,
```

```
    l.LO_DISCOUNT AS LO_DISCOUNT,
    l.LO_REVENUE AS LO_REVENUE,
    l.LO_SUPPLYCOST AS LO_SUPPLYCOST,
    l.LO_TAX AS LO_TAX,
    l.LO_COMMITDATE AS LO_COMMITDATE,
    l.LO_SHIPMODE AS LO_SHIPMODE,
    c.C_NAME AS C_NAME,
    c.C_ADDRESS AS C_ADDRESS,
    c.C_CITY AS C_CITY,
    c.C_NATION AS C_NATION,
    c.C_REGION AS C_REGION,
    c.C_PHONE AS C_PHONE,
    c.C_MKTSEGMENT AS C_MKTSEGMENT,
    s.S_NAME AS S_NAME,
    s.S_ADDRESS AS S_ADDRESS,
    s.S_CITY AS S_CITY,
    s.S_NATION AS S_NATION,
    s.S_REGION AS S_REGION,
    s.S_PHONE AS S_PHONE,
    p.P_NAME AS P_NAME,
    p.P_MFGR AS P_MFGR,
    p.P_CATEGORY AS P_CATEGORY,
    p.P_BRAND AS P_BRAND,
    p.P_COLOR AS P_COLOR,
    p.P_TYPE AS P_TYPE,
    p.P_SIZE AS P_SIZE,
    p.P_CONTAINER AS P_CONTAINER FROM lineorder AS l
INNER JOIN customer AS c ON c.C_CUSTKEY = l.LO_CUSTKEY
INNER JOIN supplier AS s ON s.S_SUPPKEY = l.LO_SUPPKEY
INNER JOIN part AS p ON p.P_PARTKEY = l.LO_PARTKEY;
```

示例四：创建分区物化视图，并将基表 STRING 类型分区键转换为日期类型作为异步物化视图分区键。

```
-- 创建分区键为 STRING 类型的基表。
CREATE TABLE `part_dates` (
  `d_date` varchar(20) DEFAULT NULL,
  `d_dayofweek` varchar(10) DEFAULT NULL,
  `d_month` varchar(11) DEFAULT NULL,
  `d_year` int(11) DEFAULT NULL,
  `d_yearmonthnum` int(11) DEFAULT NULL,
  `d_yearmonth` varchar(9) DEFAULT NULL,
  `d_daynuminweek` int(11) DEFAULT NULL,
  `d_daynuminmonth` int(11) DEFAULT NULL,
  `d_daynuminyear` int(11) DEFAULT NULL,
  `d_monthnuminyear` int(11) DEFAULT NULL,
  `d_weeknuminyear` int(11) DEFAULT NULL,
  `d_sellingseason` varchar(14) DEFAULT NULL,
```

```
  `d_lastdayinweekfl` int(11) DEFAULT NULL,
  `d_lastdayinmonthfl` int(11) DEFAULT NULL,
  `d_holidayfl` int(11) DEFAULT NULL,
  `d_weekdayfl` int(11) DEFAULT NULL,
  `d_datekey` varchar(11) DEFAULT NULL
) partition by (d_datekey);

-- 使用 `str2date` 函数创建分区物化视图。
CREATE MATERIALIZED VIEW IF NOT EXISTS `test_mv`
PARTITION BY str2date(`d_datekey`,'%Y%m%d')
DISTRIBUTED BY HASH(`d_date`, `d_month`, `d_month`)
REFRESH MANUAL
AS
SELECT
  `d_date` ,
  `d_dayofweek`,
  `d_month` ,
  `d_yearmonthnum` ,
  `d_yearmonth` ,
  `d_daynuminweek`,
  `d_daynuminmonth`,
  `d_daynuminyear` ,
  `d_monthnuminyear` ,
  `d_weeknuminyear` ,
  `d_sellingseason`,
  `d_lastdayinweekfl`,
  `d_lastdayinmonthfl`,
  `d_holidayfl` ,
  `d_weekdayfl`,
  `d_datekey`
FROM
`hive_catalog`.`ssb_1g_orc`.`part_dates` ;
```

DROP MATERIALIZED VIEW

">功能

删除物化视图。
此命令无法用于删除正在创建中的同步物化视图。如要删除创建中的同步物化视图，您需要先终止其创建任务，然后再执行删除操作。
注意：只有拥有对应物化视图 DROP 权限的用户才可以删除物化视图。

">语法

```
DROP MATERIALIZED VIEW [IF EXISTS] [database.]mv_name
```

参数

参数	必选	说明
IF EXISTS	否	如果声明该参数，删除不存在的物化视图系统不会报错。如果不声明该参数，删除不存在的物化视图系统会报错。
mv_name	是	待删除的物化视图的名称。

">示例

示例一：删除存在的物化视图

1. 查看当前数据库中存在的物化视图。

```
MySQL > SHOW MATERIALIZED VIEWS\G
*************************** 1. row ***************************
          id: 470740
        name: order_mv1
database_name: default_cluster:sr_hub
```

text: SELECT `sr_hub`.`orders`.`dt` AS `dt`, `sr_hub`.`orders`.`order_id` AS `order_id`, `sr_hub`.`orders`.`user_id` AS `user_id`, sum(`sr_hub`.`orders`.`cnt`) AS `total_cnt`, sum(`sr_hub`.`orders`.`revenue`) AS `total_revenue`, count(`sr_hub`.`orders`.`state`) AS `state_count` FROM `sr_hub`.`orders` GROUP BY `sr_hub`.`orders`.`dt`, `sr_hub`.`orders`.`order_id`, `sr_hub`.`orders`.`user_id`
rows: 0
1 rows in set (0.00 sec)

2. 删除物化视图 order_mv1。

```
DROP MATERIALIZED VIEW order_mv1;
```

3. 删除后重新查看当前数据库中存在的物化视图将不会显示该物化视图。

```
MySQL > SHOW MATERIALIZED VIEWS;
Empty set (0.01 sec)
```

示例二：删除不存在的物化视图

- 当未声明 IF EXISTS 参数时，删除一个不属于当前数据库的物化视图 k1_k2 会报错。

```
MySQL > DROP MATERIALIZED VIEW k1_k2;
ERROR 1064 (HY000): Materialized view k1_k2 is not found
```

- 当声明 IF EXISTS 参数时，删除一个不属于当前数据库的物化视图 k1_k2 不会报错。

```
MySQL > DROP MATERIALIZED VIEW IF EXISTS k1_k2;
Query OK, 0 rows affected (0.00 sec)
```

REFRESH MATERIALIZED VIEW

">功能

手动刷新指定异步物化视图或其中部分分区。
注意：只能通过该命令手动刷新方式为 ASYNC 或 MANUAL 的异步物化视图。可以通过 SHOW MATERIALIZED VIEWS 查看物化视图的刷新方式。 该操作需要对应物化视图的 REFRESH 权限。

语法

```
REFRESH MATERIALIZED VIEW [database.]mv_name
[PARTITION START ("<partition_start_date>") END ("<partition_end_date>")]
[FORCE]
[WITH { SYNC | ASYNC } MODE]
```

参数

参数	必选	说明
mv_name	是	待手动刷新的异步物化视图名称。
PARTITION START () END ()	否	手动刷新该时间区间内的分区。
partition_start_date	否	待手动刷新的分区开始时间。
partition_end_date	否	待手动刷新的分区结束时间。
FORCE	否	如果指定该参数，StarRocks 将强制刷新相应的物化视图或分区。如果不指定该参数，StarRocks 会自动判断数据是否被更新过，只在需要时刷新分区。
WITH ... MODE	否	同步或异步调用刷新任务。SYNC 指同步调用刷新任务，执行 SQL 语句后，StarRocks 将在刷新任务成功或失败后返回结果。ASYNC 指异步调用刷新任务，执行 SQL 语句后，StarRocks 将在刷新任务提交后立即返回成功，实际刷新任务会异步在后台运行。您可以通过查询 StarRocks 的 INFORMATION_SCHEMA 中的 tasks 和 task_runs 元数据视图来查看异步物化视图的刷新状态。默认值：ASYNC。自 v2.5.8 和 v3.1.0 起支持。

注意：刷新基于外部数据目录（External Catalog）创建的异步物化视图时，StarRocks 会刷新所有分区。

">示例

示例一：异步调用任务手动刷新指定物化视图。

```
REFRESH MATERIALIZED VIEW lo_mv1;
REFRESH MATERIALIZED VIEW lo_mv1 WITH ASYNC MODE;
```

示例二：手动刷新物化视图指定分区。

```
REFRESH MATERIALIZED VIEW lo_mv1
PARTITION START ("2020-02-01") END ("2020-03-01");
```

示例三：强制手动刷新物化视图指定分区。

```
REFRESH MATERIALIZED VIEW lo_mv1
PARTITION START ("2020-02-01") END ("2020-03-01") FORCE;
```

示例四：同步调用任务手动刷新指定物化视图。

```
REFRESH MATERIALIZED VIEW lo_mv1 WITH SYNC MODE;
```

SHOW CREATE MATERIALIZED VIEW

">功能

查看指定异步物化视图的定义。
提示：该操作不需要权限。

">语法

```
SHOW CREATE MATERIALIZED VIEW [db_name.]mv_name
```

参数

参数	必选	说明
db_name	否	数据库名称。如不指定，则默认查看当前数据库中指定物化视图的定义。
mv_name	是	待查看定义的物化视图的名称。

返回

返回	说明
Materialized View	物化视图的名称。
Create Materialized View	物化视图的定义。

">示例

示例：查看指定物化视图定义

```
MySQL > SHOW CREATE MATERIALIZED VIEW lo_mv1\G
*************************** 1. row ***************************
       Materialized View: lo_mv1
Create Materialized View: CREATE MATERIALIZED VIEW `lo_mv1`
COMMENT "MATERIALIZED_VIEW"
DISTRIBUTED BY HASH(`lo_orderkey`) BUCKETS 10
REFRESH ASYNC
PROPERTIES (
"replication_num" = "3",
"storage_medium" = "HDD"
)
AS SELECT `wlc_test`.`lineorder`.`lo_orderkey` AS `lo_orderkey`, `wlc_test`.`lineorder`.`lo_custkey` AS `lo_custkey`, sum(`wlc_test`.`lineorder`.`lo_quantity`) AS `total_quantity`, sum(`wlc_test`.`lineorder`.`lo_revenue`) AS `total_revenue`, count(`wlc_test`.`lineorder`.`lo_shipmode`) AS `shipmode_count` FROM `wlc_test`.`lineorder` GROUP BY `wlc_test`.`lineorder`.`lo_orderkey`, `wlc_test`.`lineorder`.`lo_custkey` ORDER BY `wlc_test`.`lineorder`.`lo_orderkey` ASC ;
1 row in set (0.01 sec)
```

数据导入
Stream Load数据导入示例

在本地文件系统的 /data/emr/starrocks/files 目录下创建一个 CSV 格式的数据文件 example1.csv。文件一共包含三列，分别代表用户 ID、用户姓名和用户得分，如下所示：

```
1,Lily,23
2,Rose,23
3,Alice,24
4,Julia,25
```

使用mysql客户端连接StarRocks FE leader节点：

```
mysql -uroot -P9030 -h ${sr_ip} -p${cluster_password}
```

通过如下语句创建数据库、并切换至该数据库：

```
CREATE DATABASE IF NOT EXISTS mydatabase;
USE mydatabase;
```

通过如下语句手动创建主键表 table1，包含 id、name 和 score 三列，分别代表用户 ID、用户姓名和用户得分，主键为 id 列，如下所示：

```
CREATE TABLE `table1`
(
    `id` int(11) NOT NULL COMMENT "用户 ID",
    `name` varchar(65533) NULL COMMENT "用户姓名",
    `score` int(11) NOT NULL COMMENT "用户得分"
)
ENGINE=OLAP
PRIMARY KEY(`id`)
DISTRIBUTED BY HASH(`id`);
```

通过如下命令，把 example1.csv 文件中的数据导入到 table1 表中：

```
curl --location-trusted -u <username>:<password> -H "label:123" \
    -H "Expect:100-continue" \
    -H "column_separator:," \
    -H "columns: id, name, score" \
    -T example1.csv -XPUT \
    http://<fe_host>:<fe_http_port>/api/mydatabase/table1/_stream_load
```

example1.csv 文件中包含三列，跟 table1 表的 id、name、score 三列一一对应，并用逗号 (,) 作为列分隔符。因

此，需要通过 column_separator 参数指定列分隔符为逗号 (,)，并且在 columns 参数中按顺序把 example1.csv 文件中的三列临时命名为 id、name、score。columns 参数中声明的三列，按名称对应 table1 表中的三列。
导入完成后，您可以查询 table1 表，验证数据导入是否成功，如下所示：

```
SELECT * FROM table1;
+------+-------+-------+
| id   | name  | score |
+------+-------+-------+
|    1 | Lily  |    23 |
|    2 | Rose  |    23 |
|    3 | Alice |    24 |
|    4 | Julia |    25 |
+------+-------+-------+
4 rows in set (0.00 sec)
```

Broker Load导入

Broker Load导入本地数据示例

Broker Load 是一种异步导入方式。提交导入作业以后，StarRocks 会异步地执行导入作业，不会直接返回作业结果，您需要手动查询作业结果。参见查看 Broker Load 导入进度。
以 CSV 格式的数据为例，登录本地文件系统，在指定路径 /data/emr/starrocks/files下创建 CSV 格式的数据文件file1.CSV，数据文件都包含三列，分别代表用户 ID、用户姓名和用户得分，如下所示：

```
1,Lily,21
2,Rose,22
3,Alice,23
4,Julia,24
```

通过如下语句手动创建主键模型表 mytable，包含 id、name 和 score 三列，分别代表用户 ID、用户姓名和用户得分，主键为 id 列，如下所示：

```
DROP TABLE IF EXISTS `mytable`;
CREATE TABLE `mytable`
(
    `id` int(11) NOT NULL COMMENT "用户 ID",
    `name` varchar(65533) NULL DEFAULT "" COMMENT "用户姓名",
    `score` int(11) NOT NULL DEFAULT "0" COMMENT "用户得分"
)
ENGINE=OLAP
PRIMARY KEY(`id`)
DISTRIBUTED BY HASH(`id`)
PROPERTIES("replication_num"="1");
```

提交导入作业
通过如下语句，把本地文件系统的 /data/emr/starrocks/files 路径下的数据文件file1.CSV中的内容导入到目标表 mytable：

```
LOAD LABEL mydatabase.label_local
(
    DATA INFILE("file:///data/emr/starrocks/files/file1.csv")
    INTO TABLE mytable
    COLUMNS TERMINATED BY ","
    (id, name, score)
)
WITH BROKER "Broker_StarRocks"
```

```
PROPERTIES
(
    "timeout" = "3600"
);
```

> 注意：
>
> 所有测试文件必须放到存储目录 /data/emr/starrocks/files 才能实现导入；使用Broker Load进行本地导入时，测试文件需要放到所有alive的broker节点的此存储目录下。

使用以下命令查看导入命令的执行状态：

```
SELECT * FROM information_schema.loads WHERE LABEL = 'label_local';
```

确认导入作业完成后，您可以从表内查询数据，验证数据导入是否成功：

```
SELECT * FROM mytable;
```

Broker Load导入HDFS数据示例

以 CSV 格式的数据为例，在Hadoop集群创建file2.CSV文件。数据文件都包含三列，分别代表用户 ID、用户姓名和用户得分，如下所示：

```
5,Tony,25
6,Adam,26
7,Allen,27
8,Jacky,28
```

将数据文件上传到hdfs
如果Hadoop集群开启kerberos，需要先进行认证：

```
klist -kt /var/krb5kdc/emr.keytab
#以下命令中${hdfs_ip}是当前hdfs节点ip，${cluster_realm}为/etc/krb5.conf中default_realm的值
kinit -kt /var/krb5kdc/emr.keytab hadoop/${hdfs_ip}@${cluster_realm}
```

在hdfs创建任意路径，将数据文件上传：

```
hdfs dfs -mkdir /user/starrocks
hdfs dfs -put file2.csv /user/starrocks
```

建库建表
通过如下语句创建数据库、并切换至该数据库：

```
CREATE DATABASE IF NOT EXISTS mydatabase;
USE mydatabase;
```

通过如下语句手动创建主键模型表 mytable，包含 id、name 和 score 三列，分别代表用户 ID、用户姓名和用户得分，主键为 id 列，如下所示：

```
CREATE TABLE `table2`
(
  `id` int(11) NOT NULL COMMENT "用户 ID",
  `name` varchar(65533) NULL DEFAULT "" COMMENT "用户姓名",
  `score` int(11) NOT NULL DEFAULT "0" COMMENT "用户得分"
)
ENGINE=OLAP
PRIMARY KEY(`id`)
DISTRIBUTED BY HASH(`id`);
```

拷贝Hadoop集群的认证文件到SR集群的所有节点：

```
scp /etc/krb5.conf root@${sr_ip}:/etc/
scp /var/krb5kdc/emr.keytab root@${sr_ip}:/var/krb5kdc/
```

拷贝Hadoop集群的配置文件到SR集群所有节点的FE、BE、Broker配置目录下：

```
cd /usr/local/service/hadoop/etc/hadoop/
scp hdfs-site.xml core-site.xml root@${sr_ip}: /usr/local/service/starrocks/fe/conf/
scp hdfs-site.xml core-site.xml root@${sr_ip}: /usr/local/service/starrocks/be/conf/
scp hdfs-site.xml core-site.xml root@${sr_ip}: /usr/local/service/starrocks/apache_hdfs_broker/conf/
```

文件导入命令如下（具体的参数结合hdfs-site.xml和core-site.xml进行修改）：

```
LOAD LABEL mydatabase.label
(
    DATA INFILE("hdfs://HDFS78000028/user/starrocks/file2.csv")
    INTO TABLE table2
    COLUMNS TERMINATED BY ","
    (id, name, score)
)
WITH BROKER "Broker_StarRocks"
(
    "hadoop.security.authentication" = "kerberos",
    "kerberos_principal" = "hadoop/${hdfs_nn}@${cluster_realm}", --其中${cluster_realm}为/etc/krb5.con
f中default_realm的值。如果没有配置hostname，${hdfs_nn}为namenode节点ip；如果设置了hostnam
e，${hdfs_nn}为namenode节点的主机名
    "kerberos_keytab" = "/var/krb5kdc/emr.keytab",
    "dfs.nameservices" = "HDFS78000028",
```

```
    "dfs.ha.namenodes.HDFS78000028" = "nn1,nn2,nn3",
    "dfs.namenode.rpc-address.HDFS78000028.nn1" = "${hdfs_nn1_ip}:4010", --在hdfs-site.xml可以查到
    "dfs.namenode.rpc-address.HDFS78000028.nn2" = "${hdfs_nn2_ip}:4010",
    "dfs.namenode.rpc-address.HDFS78000028.nn3" = "${hdfs_nn3_ip}:4010",
    "dfs.client.failover.proxy.provider.HDFS78000028" = "org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider"
)
PROPERTIES
(
    "timeout" = "3600"
);
```

使用以下命令查看导入命令的执行状态：

```
SELECT * FROM information_schema.loads WHERE LABEL = 'label';
```

确认导入作业完成后，您可以从表内查询数据，验证数据导入是否成功：

```
SELECT * FROM table2;
```

INSERT数据导入示例

在 StarRocks 中创建数据库 load_test，并在其中创建导入目标表 insert_wiki_edit 以及数据源表 source_wiki_edit。

```
CREATE DATABASE IF NOT EXISTS load_test;
USE load_test;

CREATE TABLE insert_wiki_edit
(
    event_time      DATETIME,
    channel         VARCHAR(32)      DEFAULT '',
    user            VARCHAR(128)     DEFAULT '',
    is_anonymous    TINYINT          DEFAULT '0',
    is_minor        TINYINT          DEFAULT '0',
    is_new          TINYINT          DEFAULT '0',
    is_robot        TINYINT          DEFAULT '0',
    is_unpatrolled  TINYINT          DEFAULT '0',
    delta           INT              DEFAULT '0',
    added           INT              DEFAULT '0',
    deleted         INT              DEFAULT '0'
)
DUPLICATE KEY(
    event_time,
    channel,
    user,
    is_anonymous,
    is_minor,
    is_new,
    is_robot,
    is_unpatrolled
)
PARTITION BY RANGE(event_time)(
    PARTITION p06 VALUES LESS THAN ('2015-09-12 06:00:00'),
    PARTITION p12 VALUES LESS THAN ('2015-09-12 12:00:00'),
    PARTITION p18 VALUES LESS THAN ('2015-09-12 18:00:00'),
    PARTITION p24 VALUES LESS THAN ('2015-09-13 00:00:00')
)
DISTRIBUTED BY HASH(user);

CREATE TABLE source_wiki_edit
(
    event_time      DATETIME,
    channel         VARCHAR(32)      DEFAULT '',
    user            VARCHAR(128)     DEFAULT '',
```

```
    is_anonymous    TINYINT         DEFAULT '0',
    is_minor        TINYINT         DEFAULT '0',
    is_new          TINYINT         DEFAULT '0',
    is_robot        TINYINT         DEFAULT '0',
    is_unpatrolled  TINYINT         DEFAULT '0',
    delta           INT             DEFAULT '0',
    added           INT             DEFAULT '0',
    deleted         INT             DEFAULT '0'
)
DUPLICATE KEY(
    event_time,
    channel,user,
    is_anonymous,
    is_minor,
    is_new,
    is_robot,
    is_unpatrolled
)
PARTITION BY RANGE(event_time)(
    PARTITION p06 VALUES LESS THAN ('2015-09-12 06:00:00'),
    PARTITION p12 VALUES LESS THAN ('2015-09-12 12:00:00'),
    PARTITION p18 VALUES LESS THAN ('2015-09-12 18:00:00'),
    PARTITION p24 VALUES LESS THAN ('2015-09-13 00:00:00')
)
DISTRIBUTED BY HASH(user);
```

以 insert_load_wikipedia 为 Label 向源表 source_wiki_edit 中导入两条数据。Label 是导入作业的标识，数据库内唯一。

```
INSERT INTO source_wiki_edit
WITH LABEL insert_load_wikipedia
VALUES
    ("2015-09-12 00:00:00","#en.wikipedia","AustinFF",0,0,0,0,0,21,5,0),
    ("2015-09-12 00:00:00","#ca.wikipedia","helloSR",0,1,0,1,0,3,23,0);
```

查看数据导入情况：

```
SELECT * FROM information_schema.loads
WHERE database_name = 'load_test' and label = 'insert_load_wikipedia'\G
```

查看数据导入结果：

```
select * from source_wiki_edit where user="AustinFF";
```

Kafka connector导入数据到SR

完成基础配置

必须先完成《运维管理》中《SR跨集群基础配置》中"1. 配置域名映射"和"2. 准备Kerberos认证相关文件"操作。运维指南可以联系交付人员获取。

准备Kafka环境

在Kafka的broker节点配置Kafka connector。

下载starrocks-kafka-connector-1.0.3.tar.gz

下载并解压压缩包到/data路径下，命令为：

```
cd /data/
wget
tar zxf starrocks-kafka-connector-1.0.3.tar.gz
```

修改connect-standalone.properties文件

修改文件connect-standalone.properties：

```
vim /usr/local/service/kafka/config/connect-standalone.properties
```

修改内容：

```
plugin.path=/data/starrocks-kafka-connector-1.0.3 #填starrocks-kafka-connector-1.0.3的绝对路径
bootstrap.servers=${kafka_broker_ip}:9092
```

并且在此文件中添加认证配置：
如果是GSSAPI认证（修改为该Kafka broker节点对应的keytab和principal如果集群设置了hostname，需要使用带hostname的principal）：

```
consumer.security.protocol=SASL_PLAINTEXT
```

```
consumer.sasl.mechanism=GSSAPI
consumer.sasl.kerberos.service.name=hadoop
consumer.sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true st
oreKey=true keyTab="${keytab_path}" principal="hadoop/${kafka_broker_ip}@${cluster_realm}";
producer.security.protocol=SASL_PLAINTEXT
producer.sasl.mechanism=GSSAPI
producer.sasl.kerberos.service.name=hadoop
producer.sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true st
oreKey=true keyTab="${keytab_path}" principal="hadoop/${kafka_broker_ip}@${cluster_realm}";
security.protocol=SASL_PLAINTEXT
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=hadoop
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true storeKey=tr
ue keyTab="${keytab_path}" principal="hadoop/${kafka_broker_ip}@${cluster_realm}";
```

如果是PLAIN认证（修改对应的password）：

```
consumer.security.protocol=SASL_PLAINTEXT
consumer.sasl.mechanism=PLAIN
consumer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required userna
me="hadoop" password="${password}";
producer.security.protocol=SASL_PLAINTEXT
producer.sasl.mechanism=PLAIN
producer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required usernam
e="hadoop" password="${password}";
security.protocol=SASL_PLAINTEXT
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="hado
op" password="${password}";
```

创建connect-StarRocks-sink.properties

进入Kafka配置目录/usr/local/service/kafka/config/，新建文件connect-StarRocks-sink.properties。

内容：

```
name=starrocks-kafka-connector
connector.class=com.starrocks.connector.kafka.StarRocksSinkConnector
topics=dbserver4
starrocks.http.url=${sr_fe_ip1}:8030,${sr_fe_ip2}:8030,${sr_fe_ip3}:8030
starrocks.username=root
starrocks.password=${cluster_password}
starrocks.database.name=inventory
key.converter=org.apache.kafka.connect.json.JsonConverter
value.converter=org.apache.kafka.connect.json.JsonConverter
```

```
key.converter.schemas.enable=true
value.converter.schemas.enable=false
sink.properties.strip_outer_array=true
#其中：starrocks.database.name填将要导入sr的库名，topics填表名，其他的不要改
```

添加jar包

plugin.path下的starrocks-stream-load-sdk-1.0-20231130.060553-22-jar-with-dependencies.jar复制到kafka/libs路径下。
kafka/libs路径下的kafka-clients-2.8.2.jar复制到plugin.path下，plugin.path中的kafka-clients-3.x删掉。

在SR集群建库表

使用登录命令：
mysql -h${sr_ip} -uroot -P9030 -p${cluster_password}
创建库表，库表名称要与connect-StarRocks-sink.properties中保持一致：

```
create database inventory;
use inventory;
CREATE TABLE dbserver4
(
    id int(11) NOT NULL COMMENT "城市 ID",
    city varchar(65533) NULL COMMENT "城市名称"
)
ENGINE=OLAP
PRIMARY KEY(id)
DISTRIBUTED BY HASH(id);
```

启动Kafka connector

执行命令：

```
cd /usr/local/service/kafka/config
../bin/connect-standalone.sh connect-standalone.properties connect-StarRocks-sink.properties
```

调试没问题后可以放到后台执行

```
nohup ../bin/connect-standalone.sh connect-standalone.properties connect-StarRocks-sink.properties >> kafka_connector.log 2>&1 &
```

如果报错连不上localhost/127.0.0.1:9092，检查bootstrap.servers是否为${kafka_broker_ip}:9092。

创建topic

命令如下，其中${kafka_broker_ip}填当前Kafka broker节点的ip，topic名称跟上面新建表一致：

```
export KAFKA_OPTS='-Djava.security.auth.login.config=/tmp/kafka-jaas-config.properties'; /usr/local/service/kafka/bin/kafka-topics.sh --bootstrap-server ${kafka_broker_ip}:9092 --create --topic dbserver4 --partitions 5 --replication-factor 2 --command-config /tmp/kafka-sasl-config.properties
```

如果报错/tmp/kafka-sasl-config.properties或者/tmp/kafka-jaas-config.properties相关，参考上面《Kafka开发》->《常用命令》创建对应文件。

produce数据

命令如下：

```
export KAFKA_OPTS='-Djava.security.auth.login.config=/tmp/kafka-jaas-config.properties'; /usr/local/service/kafka/bin/kafka-console-producer.sh --topic dbserver4 --bootstrap-server  ${kafka_broker_ip}:9092 --producer.config /tmp/kafka-sasl-config.properties
```

输入JSON格式的数据

```
{"id":1, "city": "city1"}
```

生产数据效果如下：

```
        value.serializer = class org.apache.kafka.common.serialization.ByteArraySerializer

[2024-02-19 19:47:16,721] [INFO ] [main:672] [org.apache.kafka.common.security.authenticator.AbstractLogin] [AbstractLogin.java:61] - Successfully logged in.
[2024-02-19 19:47:16,759] [INFO ] [main:710] [org.apache.kafka.common.utils.AppInfoParser] [AppInfoParser.java:119] - Kafka version: 2.8.2
[2024-02-19 19:47:16,759] [INFO ] [main:710] [org.apache.kafka.common.utils.AppInfoParser] [AppInfoParser.java:120] - Kafka commitId: 957cfe23556e18de
[2024-02-19 19:47:16,759] [INFO ] [main:710] [org.apache.kafka.common.utils.AppInfoParser] [AppInfoParser.java:121] - Kafka startTimeMs: 1708343236746
>[2024-02-19 19:47:16,995] [INFO ] [kafka-producer-network-thread | console-producer:946] [org.apache.kafka.clients.Metadata] [Metadata.java:287] - [Producer clientId=console-producer] Cluster ID: SEgix1OkTZ6PI3dN1nHTLQ
{"id":1, "city": "beijing"}
[2024-02-19 19:47:36,088] [WARN ] [kafka-producer-network-thread | console-producer:20039] [org.apache.kafka.clients.NetworkClient] [NetworkClient.java:1100] - [Producer clientId=console-producer] Error while fetching metadata with correlation id 3 : {dbserver4=LEADER_NOT_AVAILABLE}
[2024-02-19 19:47:36,192] [INFO ] [kafka-producer-network-thread | console-producer:20143] [org.apache.kafka.clients.Metadata] [Metadata.java:401] - [Producer clientId=console-producer] Resetting the last seen epoch of partition dbserver4-0 to 0 since the associated topicId changed from null to nwGi3k7ASBKCqfaWUY79mg
>
```

在SR集群中查看导入数据

在SR集群查询对应库表的数据，可以看到Kafka的中数据已经同步到SR。

```
MySQL [inventory]> select * from dbserver3;
Empty set (0.006 sec)

MySQL [inventory]> CREATE TABLE `dbserver4`
    -> (
    ->     `id` int(11) NOT NULL COMMENT "城市 ID",
    ->     `city` varchar(65533) NULL COMMENT "城市名称"
    -> )
    -> ENGINE=OLAP
    -> PRIMARY KEY(`id`)
    -> DISTRIBUTED BY HASH(`id`);
Query OK, 0 rows affected (0.010 sec)

MySQL [inventory]> select * from dbserver4;
+------+---------+
| id   | city    |
+------+---------+
|    1 | beijing |
+------+---------+
1 row in set (0.016 sec)

MySQL [inventory]>
```

Kafka Routine Load导入

完成基础配置

必须先完成《运维手册》中《SR跨集群基础配置》中"1. 配置域名映射"和"2. 准备kerberos认证相关文件"操作。

创建目标数据库和表

```
create database example_db;
CREATE TABLE example_db.example_tbl1 (
    `order_id` bigint NOT NULL COMMENT "订单编号",
    `pay_dt` date NOT NULL COMMENT "支付日期",
    `customer_name` varchar(26) NULL COMMENT "顾客姓名",
    `nationality` varchar(26) NULL COMMENT "国籍",
    `price`double NULL COMMENT "支付金额"
)
ENGINE=OLAP
DUPLICATE KEY (order_id,pay_dt)
DISTRIBUTED BY HASH(`order_id`);
```

创建topic

```
/usr/local/service/kafka/bin/kafka-topics.sh --bootstrap-server ${kafka_broker_ip}:9092 --create --topic ordertest1 --partitions 5 --replication-factor 2 --command-config /tmp/kafka-sasl-config.properties
```

如果报错/tmp/kafka-sasl-config.properties或者/tmp/kafka-jaas-config.properties相关，参考上面《Kafka开发》->《常用命令》创建对应文件。

生成数据

命令如下：

```
/usr/local/service/kafka/bin/kafka-console-producer.sh --topic ordertest1 --bootstrap-server  ${kafka_
```

broker_ip}:9092 --producer.config /tmp/kafka-sasl-config.properties

数据用例：

2020050802,2020-05-08,Johann Georg Faust,Deutschland,male,895
2020050802,2020-05-08,Julien Sorel,France,male,893
2020050803,2020-05-08,Dorian Grey,UK,male,1262

创建导入作业

kerberos认证时的导入命令

```
CREATE ROUTINE LOAD example_db.example_tbl1_ordertest3 ON example_tbl1
COLUMNS TERMINATED BY ",",
COLUMNS (order_id, pay_dt, customer_name, nationality, temp_gender, price)
PROPERTIES
(
    "desired_concurrent_number" = "5"
)
FROM KAFKA
(
    "kafka_broker_list" = "${kafka_broker_ip}:9092",  -- kafka broker节点的ip和端口
    "kafka_topic" = "ordertest1",
    "kafka_partitions" = "0,1,2,3,4",
    "property.kafka_default_offsets" = "OFFSET_BEGINNING",
    "property.security.protocol" = "SASL_PLAINTEXT",
    "property.sasl.mechanism" = "GSSAPI", -- 指定 SASL 认证机制为 GSSAPI, 默认是 GSSAPI
    "property.sasl.kerberos.service.name" = "hadoop",
    "property.sasl.kerberos.keytab" = "/var/krb5kdc/emr.keytab",
    "property.sasl.kerberos.principal" = "hadoop/${sr_ip}@${cluster_realm}"  -- 如果集群配置了hostnam
e，中间的${sr_ip}改成${sr_hostname}
);
```

simple认证时的导入命令

其中/tmp/kafka-sasl-config.properties或者/tmp/kafka-jaas-config.properties要改为PLAIN 认证，参考上面《Kafka开发》->《常用命令》。具体的用户密码可以在kafka节点的/usr/local/service/kafka/config/kafka-plain-jaas.conf 里面找。

```
CREATE ROUTINE LOAD example_db.example_tbl1_ordertest3 ON example_tbl1
COLUMNS TERMINATED BY ",",
COLUMNS (order_id, pay_dt, customer_name, nationality, temp_gender, price)
PROPERTIES
(
    "desired_concurrent_number" = "5"
)
FROM KAFKA
(
    "kafka_broker_list" = "${kafka_broker_ip}:9092",
    "kafka_topic" = "ordertest1",
    "kafka_partitions" = "0,1,2,3,4",
    "property.kafka_default_offsets" = "OFFSET_BEGINNING",
    "property.security.protocol" = "SASL_PLAINTEXT",
    "property.sasl.mechanism" = "PLAIN",
    "property.sasl.username" = "hadoop", -- SASL 的用户
    "property.sasl.password" = "hadoop@Tbds.com" -- SASL 的密码
);
```

查看导入作业

查看导入情况：

```
SHOW ROUTINE LOAD FOR example_tbl1_ordertest3\G;
```

查看表格数据：

```
select * from example_tbl1;
```

```
MySQL [example_db]> SHOW ROUTINE LOAD FOR example_tbl1_ordertest3\G
*************************** 1. row ***************************
                  Id: 15146
                Name: example_tbl1_ordertest3
          CreateTime: 2024-07-10 12:09:31
           PauseTime: NULL
             EndTime: NULL
              DbName: example_db
           TableName: example_tbl1
               State: RUNNING
      DataSourceType: KAFKA
      CurrentTaskNum: 3
       JobProperties: {"partial_update_mode":"null","timezone":"Asia/Shanghai","columnSeparator":",","log_rejected_record_num":"0","taskTimeoutSecond":"
60","maxFilterRatio":"1.0","strict_mode":"false","jsonpaths":"","currentTaskConcurrentNum":"3","escape":"0","enclose":"0","partitions":"*","rowDelimiter
":"\n","partial_update":"false","trim_space":"false","columnToColumnExpr":"order_id,pay_dt,customer_name,nationality,temp_gender,price","maxBatchInterva
lS":"10","whereExpr":"*","format":"csv","json_root":"","taskConsumeSecond":"15","desireTaskConcurrentNum":"5","maxErrorNum":"0","strip_outer_array":"fal
se","maxBatchRows":"200000"}
DataSourceProperties: {"topic":"ordertest1","currentKafkaPartitions":"0,1,2,3,4","brokerList":"10.206.16.74:9092"}
    CustomProperties: {"security.protocol":"SASL_PLAINTEXT","sasl.mechanism":"GSSAPI","kafka_default_offsets":"OFFSET_BEGINNING","group.id":"example_tbl
1_ordertest3_8109584d-1b60-4741-a39f-cf15ecc3795f","sasl.kerberos.keytab":"/var/krb5kdc/emr.keytab","sasl.kerberos.principal":"hadoop/10.206.16.74@TBDS-
FY2W27ZK","sasl.kerberos.service.name":"hadoop"}
           Statistic: {"receivedBytes":157,"errorRows":1,"committedTaskNum":1,"loadedRows":1,"loadRowsRate":0,"abortedTaskNum":1,"totalRows":2,"unselect
edRows":0,"receivedBytesRate":0,"taskExecuteTimeMs":1082}
            Progress: {"0":"OFFSET_ZERO","1":"0","2":"OFFSET_ZERO","3":"0","4":"OFFSET_ZERO"}
   TimestampProgress: {"1":"1720584805496","3":"1720582159285"}
ReasonOfStateChanged:
        ErrorLogUrls: http://10.206.17.172:8040/api/_load_error_log?file=error_log_e91ae5519fcc46c3_a76e43ddc34e1cec
         TrackingSQL: select tracking_log from information_schema.load_tracking_logs where job_id=15146
           OtherMsg: [2024-07-10 12:16:20] [task id: 6138af09-68ad-49b1-b1ed-ea73258cfe05] [txn id: -1] there is no new data in kafka, wait for 10 seco
nds to schedule again
LatestSourcePosition: {"0":"0","1":"1","2":"0","3":"1","4":"0"}
1 row in set (0.005 sec)

MySQL [example_db]> select * from example_tbl1;
+------------+------------+-------------------+-------------+-------+
| order_id   | pay_dt     | customer_name     | nationality | price |
+------------+------------+-------------------+-------------+-------+
| 2020050901 | 2020-05-09 | Anna Karenina"    | Russia      |   175 |
| 2020050802 | 2020-05-08 | Johann Georg Faust | Deutschland |   895 |
| 2020050803 | 2020-05-08 | Dorian Grey       | UK          |  1262 |
+------------+------------+-------------------+-------------+-------+
3 rows in set (0.018 sec)

MySQL [example_db]>
```

常见问题

如果在使用show routine load for xxx时报错认证相关问题，检查《SR跨集群基础配置》中“1. 配置域名映射”和“2. 准备kerberos认证相关文件”是否完整执行，执行后SR是否重启。

数据湖
SR对接Hive Catalog

完成基础配置

必须先完成《运维手册》中《SR跨集群基础配置》的全部操作。可联系交付人员获取运维指南。

在SR集群的leader节点使用Hive catalog

登录命令：mysql -h${sr_ip} -uroot -P9030 -p${cluster_password}
创建hive catalog，填入对应的hive.metastore.uris参数：

```
CREATE EXTERNAL CATALOG hive_catalog_hms
PROPERTIES
(
"type" = "hive",
"hive.metastore.type" = "hive",
"hive.metastore.uris" = "thrift://xxxx:7004,thrift://xxxx:7004"
);
```

其中hive.metastore.uris可以在Hadoop集群进行查看：

```
grep -A 3 -B 3 "hive.metastore.uris" /usr/local/service/hive/conf/hive-site.xml
```

查看新创建的catalog：

```
show catalogs;
```

设置当前catalog：

```
set catalog hive_catalog_hms;
```

读取本hadoop集群上的hive库表：

```
select * from test_db.test_table;
```

查询效果展示：

```
MySQL [(none)]> CREATE EXTERNAL CATALOG hive_catalog_hms00
    -> PROPERTIES
    -> (
    ->     "type" = "hive",
    ->     "hive.metastore.type" = "hive",
    ->     "hive.metastore.uris" = "thrift://10.206.17.160:7004,thrift://10.206.16.201:7004"
    -> );
Query OK, 0 rows affected (1.100 sec)

MySQL [(none)]>
MySQL [(none)]> set catalog hive_catalog_hms00;
Query OK, 0 rows affected (0.001 sec)

MySQL [(none)]>
MySQL [(none)]> show databases;
+----------+
| Database |
+----------+
| default  |
| srdb     |
+----------+
2 rows in set (0.362 sec)

MySQL [(none)]>
MySQL [(none)]> show catalogs;
+--------------------+----------+------------------------------------------------------------------+
| Catalog            | Type     | Comment                                                          |
+--------------------+----------+------------------------------------------------------------------+
| default_catalog    | Internal | An internal catalog contains this cluster's self-managed tables. |
| hive_catalog_hms00 | Hive     | NULL                                                             |
+--------------------+----------+------------------------------------------------------------------+
2 rows in set (0.002 sec)

MySQL [(none)]> select * from srdb.hive_test;
Empty set (3.069 sec)

MySQL [(none)]> select * from srdb.hive_test;
+------+------+
| a    | b    |
+------+------+
|    2 | chf  |
+------+------+
1 row in set (1.579 sec)
```

常见问题

报错无法访问hive metastore，查看fe leader节点的日志：/data/emr/starrocks/fe/log/fe.log。

1. 如果是报错ldap相关，是因为配置路径下缺少对应hadoop集群的hive-site.xml配置。
2. 如果是报错认证相关，容器形态SR集群需要排查2个方面：
    i. 检查是否每个节点的sr-fe和sr-be容器里面都执行了kinit操作。
    ii. SR集群是否配置了hostname，如果配置了hostname，执行kinit操作时，需要用带本节点hostname的principal；如果没有配置，执行kinit操作时，需要用带本节点ip的principal。
3. 如果是报错认证相关，主机形态SR集群需要排查2个方面：
    i. 检查是否每个fe和be节点的hadoop用户下都执行了kinit操作。
    ii. SR集群是否配置了hostname，如果配置了hostname，执行kinit操作时，需要用带本节点hostname的principal；如果没有配置，执行kinit操作时，需要用带本节点ip的principal。
    iii. starrocks安装目录下的配置文件权限是否为777。

SR对接Iceberg Catalog

完成基础配置

必须先完成《运维手册》中《SR跨集群基础配置》的全部操作。可联系交付人员获取运维指南。

在SR集群的leader节点使用Iceberg catalog

mysql -h${sr_ip} -uroot -P9030 -p${cluster_password}

创建iceberg catalog，填入对应的hive.metastore.uris参数：

```
CREATE EXTERNAL CATALOG iceberg_catalog_hms
PROPERTIES
(
   "type" = "iceberg",
   "iceberg.catalog.type" = "hive",
   "hive.metastore.uris" = "thrift://xxxx:7004,thrift://xxxx:7004"
);
```

其中hive.metastore.uris可以在Hadoop集群进行查看：

```
grep -A 3 -B 3 "hive.metastore.uris" /usr/local/service/hive/conf/hive-site.xml
```

查看新创建的catalog：

```
show catalogs;
```

设置当前catalog：

```
set catalog iceberg_catalog_hms;
```

读取本hadoop集群上的hive库表：

```
select * from test_iceberg.example_table;
```

常见问题

报错无法访问hive metastore，查看fe leader节点的日志：/data/emr/starrocks/fe/log/fe.log。

1. 如果是报错ldap相关，是因为配置路径下缺少对应hadoop集群的hive-site.xml配置。
2. 如果是报错认证相关，容器形态SR集群需要排查2个方面：
    i. 检查是否每个节点的sr-fe和sr-be容器里面都执行了kinit操作。
    ii. SR集群是否配置了hostname，如果配置了hostname，执行kinit操作时，需要用带本节点hostname的principal；如果没有配置，执行kinit操作时，需要用带本节点ip的principal。
3. 如果是报错认证相关，主机形态SR集群需要排查2个方面：
    i. 检查是否每个fe和be节点的hadoop用户下都执行了kinit操作。
    ii. SR集群是否配置了hostname，如果配置了hostname，执行kinit操作时，需要用带本节点hostname的principal；如果没有配置，执行kinit操作时，需要用带本节点ip的principal。
    iii. starrocks安装目录下的配置文件权限是否为777

数据导出
Export导出示例

通过如下命令把table1表中的所有数据导出到 HDFS 集群的指定路径下（需要添加对应的认证信息和hdfs配置信息，并且把要导入的hdfs的core-site.xml、 hdfs-site.xml放到所有节点的fe/conf/、be/conf/、apache_hdfs_broker/conf/目录下，重启StarRocks）：

```
EXPORT TABLE table1
TO "hdfs://hdfs_namenode_ip:rpc_port/sr/"
WITH BROKER "Broker_StarRocks"
(
    "username"="root",
    "password"="${cluster_password}",
    "hadoop.security.authentication" = "kerberos",
    "kerberos_principal" = "hadoop/_HOST@${cluster_realm}", --${cluster_realm}为/etc/krb5.conf中default_realm的值
    "kerberos_keytab" = "/var/krb5kdc/emr.keytab"
);
```

```
MySQL [test_db]> EXPORT TABLE table1
    -> TO "hdfs://10.206.0.67:4007/sr/"
    -> WITH BROKER "Broker_StarRocks"
    -> (
    ->     "username"="root",
    ->     "password"="Tbds@2023",
    ->
    ->     "hadoop.security.authentication" = "kerberos",
    ->     "kerberos_principal" = "hadoop/_HOST@TBDS-JFIR9A45",
    ->     "kerberos_keytab" = "/var/krb5kdc/emr.keytab"
    -> );
Query OK, 0 rows affected (0.006 sec)

MySQL [test_db]> SELECT LAST_QUERY_ID()
    -> ;
+--------------------------------------+
| last_query_id()                      |
+--------------------------------------+
| ebbf08f0-aafc-11ee-948c-02426ffd59e4 |
+--------------------------------------+
1 row in set (0.005 sec)

MySQL [test_db]>  SHOW EXPORT WHERE queryid = "ebbf08f0-aafc-11ee-948c-02426ffd59e4" \G;
*************************** 1. row ***************************
     JobId: 10358
   QueryId: ebbf08f0-aafc-11ee-948c-02426ffd59e4
     State: FINISHED
  Progress: 100%
  TaskInfo: {"partitions":["*"],"column separator":"\t","columns":["*"],"tablet num":6,"broker":"Broker_StarRocks","coord num":1,"db":"test_db","tbl":"table1","row delimiter":"\n","mem limit":2147483648}
      Path: hdfs://10.206.0.67:4007/sr/
CreateTime: 2024-01-04 20:29:35
 StartTime: 2024-01-04 20:29:38
FinishTime: 2024-01-04 20:29:43
   Timeout: 7200
  ErrorMsg: NULL
1 row in set (0.001 sec)

ERROR: No query specified
```

```
[root@10 starrocks]# hdfs dfs -ls /sr
2024-01-04T20:19:26,598 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-01-04T20:19:26,791 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-01-04T20:19:26,833 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
Found 2 items
-rw-r--r--   3 hadoop supergroup         43 2024-01-04 15:18 /sr/file1.csv
-rw-r--r--   3 hadoop supergroup         42 2024-01-04 15:18 /sr/file2.csv
[root@10 starrocks]# hdfs dfs -ls /sr
2024-01-04T20:30:19,444 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-01-04T20:30:19,682 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-01-04T20:30:19,723 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
Found 5 items
-rw-r--r--   3 hadoop supergroup         45 2024-01-04 20:29 /sr/data_ebbf08f0-aafc-11ee-948c-02426ffd59e4_0_0_0.csv
-rw-r--r--   3 hadoop supergroup         22 2024-01-04 20:29 /sr/data_ebbf08f0-aafc-11ee-948c-02426ffd59e4_0_0_1_0.csv
-rw-r--r--   3 hadoop supergroup         21 2024-01-04 20:29 /sr/data_ebbf08f0-aafc-11ee-948c-02426ffd59e4_0_0_1_2_0.csv
-rw-r--r--   3 hadoop supergroup         43 2024-01-04 15:18 /sr/file1.csv
-rw-r--r--   3 hadoop supergroup         42 2024-01-04 15:18 /sr/file2.csv
[root@10 starrocks]# hdfs dfs -cat /sr/data_ebbf08f0-aafc-11ee-948c-02426ffd59e4_0_0_0.csv
2024-01-04T20:30:37,587 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-01-04T20:30:37,808 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-01-04T20:30:37,852 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
4       Julia   24
7        Allen  27
1       Lily    21
8        Jacky  28
[root@10 starrocks]# hdfs dfs -cat /sr/data_ebbf08f0-aafc-11ee-948c-02426ffd59e4_0_0_1_0.csv
2024-01-04T20:30:49,651 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-01-04T20:30:49,875 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-01-04T20:30:49,916 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
5        Tony   25
6        Adam   26
[root@10 starrocks]# hdfs dfs -cat /sr/data_ebbf08f0-aafc-11ee-948c-02426ffd59e4_0_0_1_2_0.csv
2024-01-04T20:31:00,709 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-01-04T20:31:00,928 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-01-04T20:31:00,969 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
3       Alice   23
2       Rose    22
```

Spark Connector导出示例

SR数据库中插入数据

进入 test_db 数据库，创建一张名为 score_board 的表：

```
USE test_db;
CREATE TABLE `score_board`
(
    `id` int(11) NOT NULL COMMENT "",
    `name` varchar(65533) NULL DEFAULT "" COMMENT "",
    `score` int(11) NOT NULL DEFAULT "0" COMMENT ""
)
ENGINE=OLAP
PRIMARY KEY(`id`)
COMMENT "OLAP"
DISTRIBUTED BY HASH(`id`)
PROPERTIES (
    "replication_num" = "3"
);
```

向 score_board 表中插入数据：

```
MySQL [test]> INSERT INTO score_board
VALUES
    (1, 'Bob', 21),
    (2, 'Stan', 21),
    (3, 'Sam', 22),
    (4, 'Tony', 22),
    (5, 'Alice', 22),
    (6, 'Lucy', 23),
    (7, 'Polly', 23),
    (8, 'Tom', 23),
    (9, 'Rose', 24),
    (10, 'Jerry', 24),
    (11, 'Jason', 24),
    (12, 'Lily', 25),
    (13, 'Stephen', 25),
    (14, 'David', 25),
    (15, 'Eddie', 26),
    (16, 'Kate', 27),
    (17, 'Cathy', 27),
    (18, 'Judy', 27),
```

```
    (19, 'Julia', 28),
    (20, 'Robert', 28),
    (21, 'Jack', 29);
```

下载Spark connector包（注意对齐spark版本）：https://repo1.maven.org/maven2/com/starrocks/starrocks-spark-connector-3.4_2.12/1.1.0/
下载Mysql 连接器相关jar包（这里建议使用mysql-connector-java-8.0.1.1.jar）MySQL site 或者 Maven Central 均可获取
将上述jar包放到spark/jars目录下。如果报错：出现 java.lang.ClassNotFoundException: com.starrocks.connector.spark.rdd.StarrocksPartition 错误，
手动把spark-defaults.conf中的这个参数注释掉，不用重启spark，但是spark-sql需要重新连接一下。因为现在使用内置的spark jars，如果要新增jar包，需要手动上传。

使用 Spark SQL 读取数据

进入 Spark 的可执行程序目录，运行如下命令：

```
sh spark-sql
```

在数据库 test 中的表 score_board 上创建一个名为 spark_starrocks 的临时视图：

```
CREATE TEMPORARY VIEW spark_starrocks
USING starrocks
OPTIONS
(
  "starrocks.table.identifier" = "test_db.score_board",
  "starrocks.fe.http.url" = "${sr_ip}:8030",
  "starrocks.fe.jdbc.url" = "jdbc:mysql://${sr_ip}:9030",
  "starrocks.user" = "root",
  "starrocks.password" = "${cluster_password}"
);
```

从临时视图中读取数据：

```
spark-sql> SELECT * FROM spark_starrocks;
```

```
spark-sql> CREATE TEMPORARY VIEW spark_starrocks
         > USING starrocks
         > OPTIONS
         > (
         >    "starrocks.table.identifier" = "test_db.score_board",
         >    "starrocks.fe.http.url" = "10.206.0.67:8030",
         >    "starrocks.fe.jdbc.url" = "jdbc:mysql://10.206.0.67:9030",
         >    "starrocks.user" = "root",
         >    "starrocks.password" = "Tbds@2023"
         > );
Time taken: 0.905 seconds
spark-sql> SELECT * FROM spark_starrocks;
5       Alice   22
6       Lucy    23
20      Robert  28
3       Sam     22
17      Cathy   27
2       Stan    21
11      Jason   24
21      Jack    29
9       Rose    24
10      Jerry   24
12      Lily    25
15      Eddie   26
18      Judy    27
4       Tony    22
7       Polly   23
14      David   25
16      Kate    27
19      Julia   28
1       Bob     21
8       Tom     23
13      Stephen 25
Time taken: 6.469 seconds, Fetched 21 row(s)
spark-sql>
```

Spark DataFrame读取数据

```
val starrocksSparkDF = spark.read.format("starrocks")
.option("starrocks.table.identifier", s"test_db.score_board")
.option("starrocks.fe.http.url", s"${sr_ip}:8030")
.option("starrocks.fe.jdbc.url", s"jdbc:mysql://${sr_ip}:9030")
.option("starrocks.user", s"root")
.option("starrocks.password", s"${cluster_password}").load()
starrocksSparkDF.show(10)
```

```
scala> val starrocksSparkDF = spark.read.format("starrocks").option("starrocks.table.identifier", s"test_db.score_board").option(
"starrocks.fe.http.url", s"10.206.0.67:8030").option("starrocks.fe.jdbc.url", s"jdbc:mysql://10.206.0.67:9030").option("starrocks
.user", s"root").option("starrocks.password", s"Tbds@2023").load()
starrocksSparkDF: org.apache.spark.sql.DataFrame = [id: int, name: string ... 1 more field]

scala> starrocksSparkDF.show(10)
+---+-------+-----+
| id|   name|score|
+---+-------+-----+
|  9|   Rose|   24|
| 10|  Jerry|   24|
| 12|   Lily|   25|
| 15|  Eddie|   26|
| 18|   Judy|   27|
|  1|    Bob|   21|
|  8|    Tom|   23|
| 13|Stephen|   25|
|  5|  Alice|   22|
|  6|   Lucy|   23|
+---+-------+-----+
only showing top 10 rows
```

Spark RDD读取数据

进入spark的可执行程序目录：

```
sh spark-shell
import com.starrocks.connector.spark._
val starrocksSparkRDD = sc.starrocksRDD(
tableIdentifier = Some("test_db.score_board"),
cfg = Some(Map(
  "starrocks.fenodes" -> "${sr_ip}:8030",
  "starrocks.request.auth.user" -> "root",
  "starrocks.request.auth.password" -> "${cluster_password}"
))
)
starrocksSparkRDD.take(10)
starrocksSparkRDD.collect()
```

```
scala> import com.starrocks.connector.spark._
import com.starrocks.connector.spark._

scala> val starrocksSparkRDD = sc.starrocksRDD(
     | tableIdentifier = Some("test_db.score_board"),
     | cfg = Some(Map(
     |    "starrocks.fenodes" -> "10.206.0.67:8030",
     |    "starrocks.request.auth.user" -> "root",
     |    "starrocks.request.auth.password" -> "Tbds@2023"
     | ))
     | )
starrocksSparkRDD: org.apache.spark.rdd.RDD[AnyRef] = ScalaStarrocksRDD[2] at RDD at AbstractStarrocksRDD.scala:34

scala> starrocksSparkRDD.take(10)
res17: Array[AnyRef] = Array([2, Stan, 21], [11, Jason, 24], [21, Jack, 29], [4, Tony, 22], [7, Polly, 23], [14, David, 25], [16,
 Kate, 27], [19, Julia, 28], [5, Alice, 22], [6, Lucy, 23])

scala> starrocksSparkRDD.collect()
res18: Array[AnyRef] = Array([2, Stan, 21], [11, Jason, 24], [21, Jack, 29], [4, Tony, 22], [7, Polly, 23], [14, David, 25], [16,
 Kate, 27], [19, Julia, 28], [5, Alice, 22], [6, Lucy, 23], [20, Robert, 28], [1, Bob, 21], [8, Tom, 23], [13, Stephen, 25], [9,
Rose, 24], [10, Jerry, 24], [12, Lily, 25], [15, Eddie, 26], [18, Judy, 27], [3, Sam, 22], [17, Cathy, 27])
```

Flink Connector读取数据

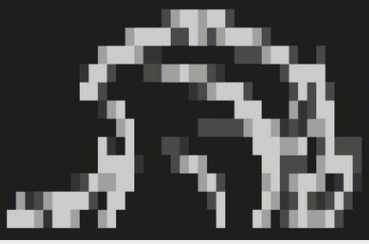
StarRocks 提供的 Apache Flink® Connector (StarRocks Connector for Apache Flink®)，支持通过 Flink 批量读取某个 StarRocks 集群中的数据。

Flink Connector 支持两种数据读取方式：Flink SQL 和 Flink DataStream。推荐使用 Flink SQL。

根据 Flink 的版本，选择和下载对应版本的 flink-connector-starrocks JAR 包，下载地址为：Tags · StarRocks/starrocks-connector-for-apache-flink · GitHub

参考上文《Flink开发》中的使用方式，启动Flink本地集群。

```
[root@172 flink]# mv flink-connector-starrocks-1.2.4_flink-1.14_2.12.jar  lib/
[root@172 flink]# ll lib/
total 219556
-rwxr-xr-x 1 root   root    14786229 Jan 13 11:48 flink-connector-starrocks-1.2.4_flink-1.14_2.12.jar
-rwxr-xr-x 1 hadoop hadoop     85583 Dec 14 11:28 flink-csv-1.14.5.jar
-rwxr-xr-x 1 hadoop hadoop 160083474 Dec 14 11:28 flink-dist_2.12-1.14.5.jar
-rwxr-xr-x 1 hadoop hadoop    153139 Dec 14 11:28 flink-json-1.14.5.jar
-rwxr-xr-x 1 hadoop hadoop   7709731 Dec 14 11:28 flink-shaded-zookeeper-3.4.14.jar
-rwxr-xr-x 1 hadoop hadoop  39666415 Dec 14 11:28 flink-table_2.12-1.14.5.jar
-rwxr-xr-x 1 hadoop hadoop    208006 Dec 14 11:28 log4j-1.2-api-2.17.1.jar
-rwxr-xr-x 1 hadoop hadoop    301872 Dec 14 11:28 log4j-api-2.17.1.jar
-rwxr-xr-x 1 hadoop hadoop   1790452 Dec 14 11:28 log4j-core-2.17.1.jar
-rwxr-xr-x 1 hadoop hadoop     24279 Dec 14 11:28 log4j-slf4j-impl-2.17.1.jar
[root@172 flink]# pwd
/usr/local/service/flink
[root@172 flink]# vi conf/flink-conf.yaml
[root@172 flink]# export HADOOP_CLASSPATH=`hadoop classpath`

[root@172 flink]# export HADOOP_CONF_DIR=/usr/local/service/hadoop/etc/hadoop
[root@172 flink]# export HIVE_CONF_DIR=/usr/local/service/hive/conf/
[root@172 flink]# ./bin/start-cluster.sh
Starting cluster.
[INFO] 1 instance(s) of standalonesession are already running on 172.16.48.66.
Starting standalonesession daemon on host 172.16.48.66.
[INFO] 1 instance(s) of taskexecutor are already running on 172.16.48.66.
Starting taskexecutor daemon on host 172.16.48.66.
[root@172 flink]#
[root@172 flink]# ./bin/sql-client.sh embedded -j lib/flink-connector-starrocks-1.2.4_flink-1.14_2.12.jar
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/local/service/flink/lib/log4j-slf4j-impl-2.17.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/service/hadoop/share/hadoop/common/lib/log4j-slf4j-impl-2.20.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
Command history file path: /root/.flink-sql-history
```



根据待导入数据的 StarRocks 表，在 Flink 中创建一张表，例如 flink_test，并配置读取任务属性，包括设置 Flink Connector 和库表的信息：

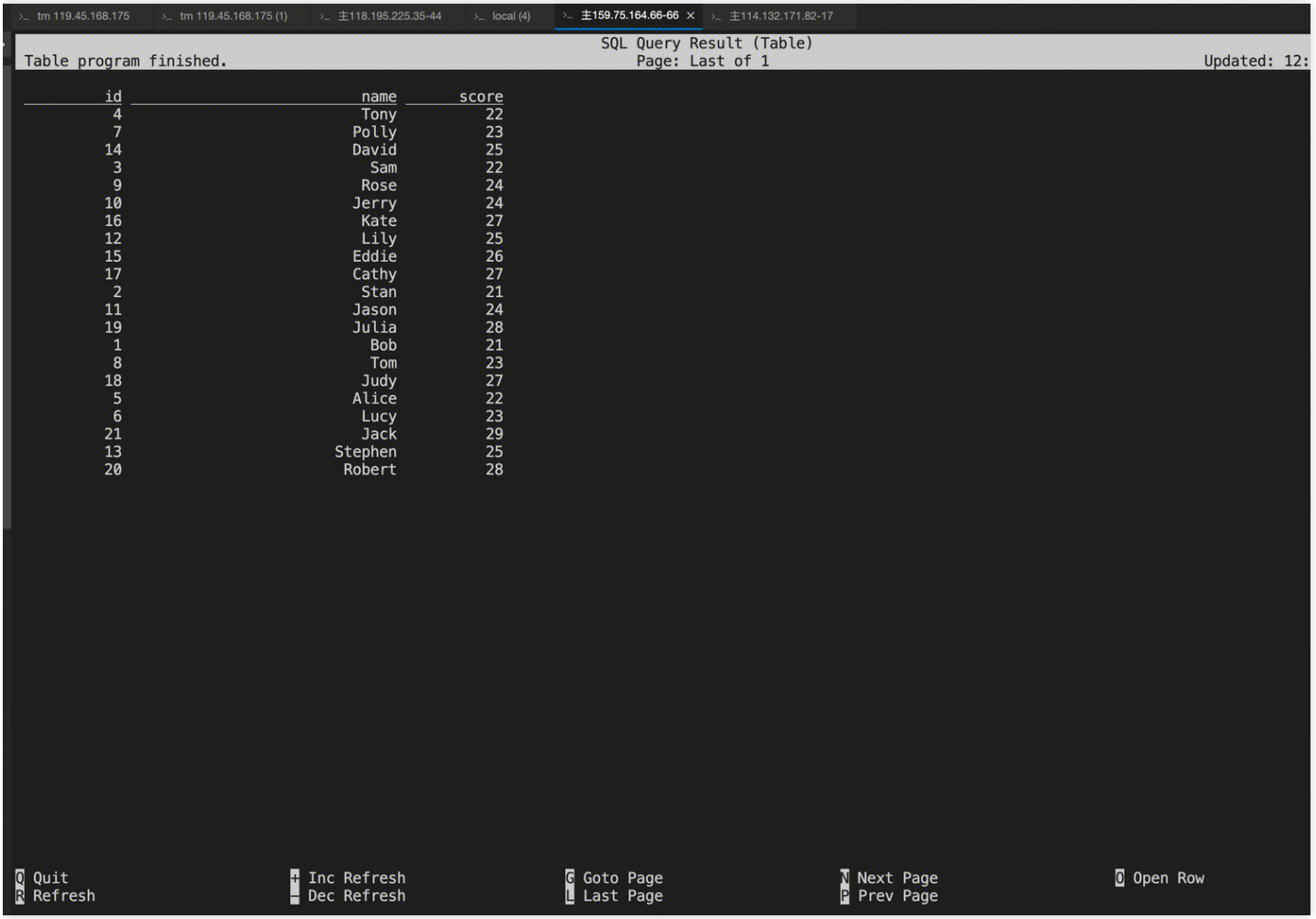
```
CREATE TABLE flink_test
(
    `id` INT,
    `name` STRING,
    `score` INT
)
WITH
(
    'connector'='starrocks',
```

```
    'scan-url'='${sr_ip}:8030',
    'jdbc-url'='jdbc:mysql://${sr_ip}:9030',
    'username'='root',
    'password'='${cluster_password}',
    'database-name'='test',
    'table-name'='score_board'
);
```

```
Flink SQL> CREATE TABLE flink_test
> (
>     id INT,
>     name STRING,
>     score INT
> )
> WITH
> (
>     'connector'='starrocks',
>     'scan-url'='10.206.16.44:8030',
>     'jdbc-url'='jdbc:mysql://10.206.16.44:9030',
>     'username'='root',
>     'password'='Tbds@2023',
>     'database-name'='test',
>     'table-name'='score_board'
> );
[INFO] Execute statement succeed.

Flink SQL> show tables;
+------------+
| table name |
+------------+
| flink_test |
+------------+
1 row in set

Flink SQL> select * from flink_test;
[INFO] Result retrieval cancelled.

Flink SQL>
```

使用 SQL 语句读取 StarRocks 的数据：

```
SELECT id, name FROM flink_test WHERE score > 20;
```


tm 119.45.168.175
tm 119.45.168.175 (1)
主118.195.225.35-44
local (4)
主159.75.164.66-66
主114.132.171.82-17
SQL Query Result (Table)
Table program finished.
Page: Last of 1
Updated: 12:
id name score
4 Tony 22
7 Polly 23
14 David 25
3 Sam 22
9 Rose 24
10 Jerry 24
16 Kate 27
12 Lily 25
15 Eddie 26
17 Cathy 27
2 Stan 21
11 Jason 24
19 Julia 28
1 Bob 21
8 Tom 23
18 Judy 27
5 Alice 22
6 Lucy 23
21 Jack 29
13 Stephen 25
20 Robert 28
Q Quit
R Refresh
+ Inc Refresh
- Dec Refresh
G Goto Page
L Last Page
N Next Page
P Prev Page
O Open Row

性能调优

类型	模块	参数	值	备注
公共参数	BE	storage_root_path	/data/emr/starrocks/be/storage,medium:hdd;/data1/emr/starrocks/be/storage,medium:hdd;/data2/emr/starrocks/be/storage,medium:hdd;/data3/emr/starrocks/be/storage,medium:hdd;/data4/emr/starrocks/be/storage,medium:hdd;	提升磁盘io吞吐量，利用多磁盘优势，数据会自动平均分配到多个磁盘，磁盘挂了会导致BE启动失败，需要摘除失败的磁盘，再启动BE
	BE	datacache_disk_path	/data/emr/starrocks/be/datacache/;/data1/emr/starrocks/be/datacache/;/data2/emr/starrocks/be/datacache/;/data3/emr/starrocks/be/datacache/;/data4/emr/starrocks/be/datacache/;	
	BE	disable_storage_page_cache	false	启动操作系统缓存，多次查询性能提升明显
	系统参数	query_mem_limit	xxx	如果查询结果很大，可以适当调大该值，但需要根据系统内存调整
	BE	storage_page_cache_limit	xxx	PageCache 的容量，可写为容量大小，也可以写为 PageCache 占系统内存的比例，例如，20%。该参数仅在 disable_storage_page_cache 为 false 时生效。
	FE	ignore_unknown_log_id	true	升级后如果碰到问题需要回滚，请在 fe.conf 文件中增加 ignore_unknown_log_id=true。这是因为新版本的元数据日志新增了类型，如果不加这个参数，则无法回滚。
	BE	datacache_disk_size	128849018880	主要用于存算分离，单个磁盘缓存数据量的上限，可设为比例上限（如 80%）或物理上限（如 2T, 500G 等）。举例：在 datacache_disk_path 中配置了 2 个磁盘，并设置 datacache_disk_size 参数值为 21474836480，即 20 GB，那么最多可缓存 40 GB 的磁盘数据。默认值为 0，即仅使用内存作为缓存介质，不使用磁盘
	BE	query_cache_capacity	4G	指定 Query Cache 的大小。默认为 512 MB。最小不低于 4 MB。如果当前的 BE 内存容量无法满足您期望的 Query Cache 大小，可以增加 BE 的内存容量，然后再设置合理的 Query Cache 大小。每个 BE 都有自己私有的 Query Cache 存储空间，BE 只 Populate 或 Probe 自己本地的 Query Cache 存储空间。
	BE	tablet_max_versions	2000	每个 Tablet 上允许的最大版本数。如果超过该值，新的写入请求会失败。
	FE	enable_collect_query_detail_info	true	是否收集查询的 Profile 信息。设置为 true 时，系统会收集查询的 Profile。设置为 false 时，系统不会收集查询的 profile。
	BE/FE	JAVA_OPTS	-Djava.security.krb5.conf=/etc/krb5.conf -Dlog4j2.formatMsgNoLookups=true -Xmx32768m -XX:+UseMembar -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=7 -XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+UseG1GC -verbose:gc -XX:-CMSParallelRemarkEnabled -	根据节点内存修改-Xmx

类型	模块	参数	值	备注
			XX:CMSInitiatingOccupancyFraction=80 -XX:SoftRefLRUPolicyMSPerMB=0 -Xloggc:${LOG_DIR}/gc_starrocks_fe.log.$DATE -XX:+PrintConcurrentLocks	
	BE/FE	JAVA_OPTS_FOR_JDK_11	-Dlog4j2.formatMsgNoLookups=true -Xmx32768m -XX:+UseG1GC -Xlog:gc*:${LOG_DIR}/fe.gc.log.$DATE:time	修改-Xmx 为32G
	建表语句	新增："replicated_storage" = "true"		创建具有自增列的表时，必须设置 'replicated_storage' = 'true'，以确保所有副本具有相同的自增 ID。
数据导入	系统参数	enable_query_queue_load	true	开启INSERT SELECT 导入队列，用于控制是否为导入任务启用查询队列。
	BE	streaming_load_max_mb	102400	流式导入单个文件大小的上限。自 3.0 版本起，默认值由 10240 变为 102400。
	BE	streaming_load_max_batch_size_mb	102400	流式导入json文件的最大值
	BE	flush_thread_num_per_store	8	每个盘的flush线程数，当用户盘比较少时可以设置较大，盘较多时设置较小，一般情况下 flush_thread_num_per_store * store_num < be_cpu_core_num / 2
		olap_table_sink_send_interval_ms	1	会影响Broker Load的速度，但影响有限
	BE	number_tablet_writer_threads	18	用于 Stream Load 的线程数。自 v3.1.7 起变为动态参数
	BE	load_process_max_memory_limit_percent	50	单节点上所有的导入线程占据的内存上限比例，取 mem_limit * load_process_max_memory_limit_percent / 100 和 load_process_max_memory_limit_bytes 中较小的值，导入内存到达限制，会触发刷盘和反压逻辑。
	BE	load_process_max_memory_limit_bytes	322122547200	单节点上所有的导入线程占据的内存上限，取 mem_limit * load_process_max_memory_limit_percent / 100 和 load_process_max_memory_limit_bytes 中较小的值。如导入内存到达限制，则会触发刷盘和反压逻辑。
	BE	enable_new_load_on_memory_limit_exceeded	true	默认true，开启该参数时，导入作业总内存达到阈值后，新增的导入作业会排入队列中
	BE	send_channel_buffer_limit	134217728	默认值64MB，当导入的数据列较多、单行数据较大时，可以适当调大
		broker_write_timeout_seconds	300	Broker写文件时flush操作的超时时间
	FE	stream_load_default_timeout_second	86400	Stream Load 的默认超时时间。
	FE	max_running_txn_num_per_db	1000	StarRocks 集群每个数据库中正在运行的导入相关事务的最大个数，默认值为 1000。自 3.1 版本起，默认值由 100 变为 1000。当数据库中正在运行的导入相关事务超过最大个数限制时，后续的导入不会执行。如果是同步的导入作业请求，作业会被拒绝；如果是异步的导入作业请求，作业会在队列中等待。不建议调大该值，会增加系统负载。
高并发查询	BE	fragment_pool_thread_num_max	8192	最大查询线程数。
	系统变量	enable_query_queue_select	true	默认值为false,在高并发场景下可以打开，启用查询队列（仅global级别）

类型	模块	参数	值	备注
	系统变量	query_timeout	7200	用于设置查询超时时间，单位为秒。该变量会作用于当前连接中所有的查询语句，以及 INSERT 语句。
	session级别	max_user_connections	5000	当前用户的最大连接数为 1000。SET PROPERTY FOR 'jack' 'max_user_connections' = '1000';
	FE	qe_max_connection	20000	FE 支持的最大连接数，包括所有用户发起的连接。
	系统变量	enable_query_cache	true	是否开启 Query Cache。取值范围：true 和 false。true 表示开启，false 表示关闭（默认值）。开启该功能后，只有当查询满足Query Cache 所述条件时，才会启用 Query Cache。 默认false,
	系统变量	cbo_enable_low_cardinality_optimize	true	是否开启低基数全局字典优化。开启后，查询 STRING 列时查询速度会有 3 倍左右提升 默认为true,但是可能会触发BE crash
实时写入	Flink	pipeline.object-reuse	true	flink 开启对象重用
	Flink	table.exec.sink.upsert-materialize	none	由于分布式系统中的 shuffle 会造成 changelog 数据的乱序，所以 sink 接收到的数据可能在全局的 upsert 中乱序，所以要在 upsert sink 之前添加一个 upsert 物化算子。该算子接收上游 changelog 数据，并且给下游生成一个 upsert 视图。 默认情况下，在唯一 key 遇到分布式乱序时，该物化算子会被添加，也可以选择不物化（NONE），或者是强制物化（FORCE）。 在测试吞吐量时设置为NONE
	Flink	sink.buffer-flush.max-bytes	300000000	积攒在内存的数据大小，达到该阈值后数据通过 Stream Load 一次性导入 StarRocks。取值范围：[64MB, 10GB]。将此参数设置为较大的值可以提高导入性能，但可能会增加导入延迟。 该参数只在 sink.semantic 为at-least-once才会生效。 测试吞吐时可加大
	Flink	sink.buffer-flush.max-rows	1000000	积攒在内存的数据条数，达到该阈值后数据通过 Stream Load 一次性导入 StarRocks。取值范围：[64000, 5000000]。该参数只在 sink.version 为 V1，sink.semantic 为 at-least-once 才会生效。 测试吞吐时可加大
	Flink	sink.buffer-flush.interval-ms	5000	数据发送的间隔，用于控制数据写入 StarRocks 的延迟，取值范围：[1000, 3600000]。该参数只在 sink.semantic 为 at-least-once才会生效。 测试吞吐时可加大
基准测试	系统变量	enable_query_tablet_affinity	true	基准测试由于只查询一次，需要精确利用上缓存，这个参数设置为true，相当于每次查询都查询固定的节点副本，从而使用Page Cache提速

开发规范

本文从部署、建模、导入、查询和监控五个模块介绍StarRocks的最佳使用方法。

部署

容量规划

【建议】做容量规划
为了实现集群高可用，建议集群最低3个节点，FE和BE分开部署也可以混合部署。
单节点配置要求：
BE推荐16核64GB内存以上，FE推荐8核16GB内存以上。
磁盘可以使用HDD或者SSD。
CPU必须支持AVX2指令集， cat /proc/cpuinfo |grep avx2 确认有输出即可，如果没有支持，建议更换机器，StarRocks的向量化技术需要CPU指令集支持才能发挥更好的效果。
网络需要万兆网卡和万兆交换机。
假定内存、磁盘都不会拖后腿的情况下，分析/查询的性能瓶颈在 CPU 的处理能力。所以通过对 CPU 的算力要求，来预估集群的数量。
集群需要的总 CPU 资源：e_core = scan_rows / cal_rows / e_rt * e_qps

变量名	变量含义	样例
e_core	预计要使用的CPU核数(vCPU)	
vCPU总数(逻辑处理器) = Socket数（CPU个数）x Core数（内核）x Thread数（超线程）	540c	
scan_rows	线上典型场景中的数据扫描量	3000万
e_qps	预期线上QPS	180qps
e_rt	预期线上响应时间	300ms
cal_rows	StarRocks针对SQL场景的计算能力	3000万/s

场景样例：

数据量：事实表一年 3.6亿行数据，大约 100万行/天；
典型查询场景：一个月的事实表数据（ 3000万 ）和比较小的维度表（万级别）做关联，再进行 group by、sum 等聚合计算；
期望：响应时间在 300ms 以内，业务的峰值 QPS 达 180 左右。
估算解释：
StarRocks 的处理能力在"单核 1000万~1亿/秒"，此场景有「多表 join」和「group by」以及一些表达式函数，相对复杂，所以按照「 3000万/s 的计算能力」估算，需要 3 个 vCPU：3000万 / 3000万/s / 300ms = 3c。
并发峰值为180qps，因此需要 3 * 180 = 540c，即总共需要 540 个 vCPU。按单台物理机48虚拟核(vCPU)算，理论计算大约需要12台物理机。
实际 POC 过程中，用3台物理机16虚拟核进行压力测试，能够在40qps下满足300-500ms的响应时间。最终，线上确定用7台48虚拟核的物理机。 所以，还是建议用户要根据实际的业务场景做一下POC测试。
综上：根据POC的测试结果，建议用户搭建3个FE节点每个节点16核64GB内存、7个BE节点每个节点48核152GB内存。
其他说明：
计算业务越复杂、处理中的一行的列数量越多越复杂，每秒能处理的行数就会越少；
计算中「条件过滤」的效果越好（能过滤掉很多数据），则能处理的行数就会越多（因为内部有一些索引结构，能更快地帮助处理数据）；
不同「表模型」会对处理能力有很大影响，上面是按照「明细模型」估算。其他模型，内部会有一些特殊处理，真实的数据量行数会和用户理解的数据量行数有一些差异；同时，分区/分桶，也会对查询性能有很大影响；（我们有其他相关文档来指导用户如何使用以达到最佳性能）
对于一些需要扫描大量数据的场景，磁盘的性能也会影响处理能力。需要时，可以使用SSD来加速。

基础环境配置

【必须】关注端口信息、swap关闭、overcommit设置为1、ulimit配置合理

- StarRocks 为不同的服务使用特定的端口，TBDS中默认使用以下端口
  FE的端口：
  8030：FE HTTP Server 端口（http_port）
  9020：FE Thrift Server 端口（rpc_port）
  9030：FE MySQL Server 端口（query_port）
  9010：FE 内部通讯端口（edit_log_port）
  6090：FE 云原生元数据服务 RPC 监听端口（cloud_native_meta_port）
  BE的端口：
  9060：BE Thrift Server 端口（be_port）
  8040：BE HTTP Server 端口（be_http_port）
  9050：BE 心跳服务端口（heartbeat_service_port）
  8060：BE bRPC 端口（brpc_port）

9070：BE 和 CN 的额外 Agent 服务端口。（starlet_port）
Broker的端口：
8000：Broker 上的 thrift server 端口，用于接收 FE 或 BE 的请求

- 内存设置
  Memory Overcommit
  Memory Overcommit 允许操作系统将额外的内存资源分配给进程。建议您启用 Memory Overcommit。

修改配置文件。

cat >> /etc/sysctl.conf << EOF
vm.overcommit_memory=1
EOF

使修改生效。

sysctl -p

- Swap Space
  建议您禁用 Swap Space。
  检查并禁用 Swap Space 操作步骤如下：

1. 关闭 Swap Space。
   swapoff /
   swapoff -a
2. 从/etc/fstab 文件中删除 Swap Space 信息。
   / swap swap defaults 0 0
3. 确认 Swap Space 已关闭。
   free -m

- ulimit 设置
  如果最大文件描述符和最大用户进程的值设置得过小，StarRocks 运行可能会出现问题。TBDS出厂配置已经将最大文件描述符和最大用户进程数调大，通过命令可以查看：
  cat >> /etc/security/limits.conf

机器配置

FE节点
【建议】 8C32GB
【必须】数据盘>=200GB，建议 SSD

BE节点
【建议】CPU:内存比，1:4，生产最小配置必须是 8C32GB+
【建议】单节点磁盘容量建议10TB，数据盘建议最大单盘2TB，建议SSD或者NVMe（如果是HDD，建议吞吐>150MB/s，IOPS>500）
【建议】集群中节点同构（机器规格一样，避免木桶效应）

部署方案

【必须】生产环境必须最小集群规模 3FE+3BE（建议FE和BE独立部署），如果混合部署， 必须配置be.conf 中的mem_limit 为减去其他服务后剩余内存量，例如机器内存40G，上面已经部署了FE，理论上限会用8G，那么配置下mem_limit=30G (40-8-2)，2g是给系统预留
【必须】生产必须 FE 高可用部署 ，1 Leader + 2 Follower，如果需要提高读并发，可以扩容Observer节点
【必须】生产必须使用负载均衡器连接集群进行读写，一般常用Nginx、Haproxy、F5等

建模

建表规范

- 仅支持UTF8编码
- 不支持修改表中的列名（即将支持）
- VARCHAR最大长度1048576
- KEY列不能使用FLOAT、DOUBLE类型
- 数据目录名、数据库名、表名、视图名、用户名、角色名 大小写敏感 ，列名和分区名 大小写不敏感
- 主键模型中，主键长度不超过128字节

模型选择

- 如果想要保留明细，建议使用明细模型
- 如果有明确主键，主键非空，写少读多，非主键列要利用索引，建议使用主键模型
- 如果有明确主键，主键可能为空，写多读少，建议使用更新模型
- 如果只想保留聚合数据，建议使用聚合模型

排序列和前缀索引选择

DUPLICATE KEY、AGGREGATE KEY、UNIQUE KEY中指定的列，3.0版本以前，主键模型中排序列通过PRIMARY KEY指定，3.0版本起，主键模型中排序列通过ORDER BY指定。
前缀索引是在排序列基础上引入的稀疏索引，进一步提升查询效率，全部加载在内存中

- 经常作为查询条件的列，建议选为排序列，例如经常用user_id过滤，where user_id=234，可以把user_id放在第一列
- 排序列建议选择3-5列，过多会增大排序开销，降低导入效率
- 前缀索引不超过36字节，不能超过3列，遇到varchar会截断，前缀索引中不能包含 float 或 double 类型的列因此可以结合实际业务查询场景，在确定 key 列以及字段顺序时，要充分考虑前缀索引带来的优势。尽可能将经常需要查询的key列字段，放置在前面，字段数据类型尽量选择 date 日期类型或者 int 等整数类型。
  举例：

```
CREATE TABLE site_access(
  site_id BIGINT DEFAULT '10',
  city_code INT,
  site_name VARCHAR(50),
  pv BIGINT DEFAULT '0'
)
DUPLICATE KEY(site_id,city_code,site_name)
DISTRIBUTED BY HASH(site_id);
```

在 site_access 表中，前缀索引为 site_id(8 Bytes) + city_code(4 Bytes) + site_name(前 24 Bytes)

- 如果查询条件只包含 site_id 和 city_code 两列，如下所示，则可以大幅减少查询过程中需要扫描的数据行：

```
select sum(pv) from site_access where site_id = 123 and city_code = 2;
```

- 如果查询条件只包含 site_id 一列，如下所示，可以定位到只包含 site_id 的数据行：

```
select sum(pv) from site_access where site_id = 123;
```

- 如果查询条件只包含 city_code 一列，如下所示，则需要扫描所有数据行，排序效果大打折扣：

```
select sum(pv) from site_access where city_code = 2;
```

- 如果 site_id和city_code联合查询和单独city_code的查询占比不相上下，可以考虑创建同步物化视图调整列顺序来达到查询性能提升，物化视图中的city_code放到第一列

```
create materialized view site_access_city_code_mv as
select city_code, site_id, site_name, pv
from site_access;
```

Bad case：

```
CREATE TABLE site_access_bad(
    site_name VARCHAR(20),
    site_id BIGINT DEFAULT '10',
    city_code INT,
    pv BIGINT DEFAULT '0'
)
PRIMARY KEY(site_id)
DISTRIBUTED BY HASH(site_id)
ORDER BY(site_id,city_code);
```

在 site_access_bad 表中，前缀索引只有 site_name

分区选择

【建议】值不会变化的时间列经常用于where过滤，使用该列创建分区
【建议】有数据淘汰需求的场景建议选择动态分区
【必须】数据更新有明显的冷热特征的，必须创建分区，例如经常更新最近一周的数据，可以按天分区
【必须】单个分区数据量不要超过100GB
【必须】超过50G或者5KW的表建议创建分区
【建议】按需创建分区，不要提前创建大量空分区，避免元数据太多占用fe的内存
当前支持时间类型（Range分区、表达式分区）、字符串（List分区）、数字（Range分区、List分区）
默认最大支持1024个分区，可以通过参数调整，不过一般情况下不需要调整

分桶选择

生产必须使用 3 副本
分桶个数判断
【必须】单个桶按照1GB预估，原始数据按照10GB（导入starrocks后，压缩比7:1～10:1）预估
当按照以上策略估算出来的分桶个数小于be个数的时候，最终分桶个数以be个数为准，例如6个be节点，按照1GB每个桶预估分桶个数为1，最终分桶个数取6
【必须】非分区表不要使用动态分桶，按照实际数据量估算分桶个数
【必须】如果分区表的各个分区的数据差异很大，建议不要使用动态分桶策略
分桶裁剪和数据倾斜如何抉择？

【建议】如果分桶列是where中经常用到的列，且分桶列的重复度比较低（例如用户id、事务id等），则可以利用该列作为分桶列
【建议】如果查询条件中有city_id和site_id，city_id取值只有几十，如果仅仅使用city_id分桶，则可能出现某些桶的数据量会比较大，出现数据倾斜，这个时候可以考虑使用city_id和site_id联合作为分桶字段，不过这样做的缺点是如果查询条件中只有city_id的时候，没办法利用分桶裁剪
【建议】如果没有合适的字段作为分桶字段打散数据，可以利用random分桶，不过这样的话没办法利用分桶裁剪的属性
【必须】2个或多个超过KW行以上的表join，建议使用colocate

字段类型

【建议】不要使用NULL属性
【必须】时间类型和数字类型的列选择正确的类型，否则计算的开销会比较大，例如时间类型的数据"2024-01-01 00:00:00"不要使用VARCHAR存储，这样没办法利用到starrocks内部的zonemap索引，没办法加速过滤

索引选择

bitmap索引

适合基数在10000-100000左右的列
适合等值条件 (=) 查询或 [NOT] IN 范围查询的列
不支持为 FLOAT、DOUBLE、BOOLEAN 和 DECIMAL 类型的列创建 Bitmap 索引
城市、性别这些基数在255以下的列不需要创建bitmap索引，因为starrocks内部有低基数字典，会针对这些case自动创建低基数字典用于加速
明细模型和主键模型，所有列可以创建bitmap索引，聚合模型和更新模型，只有Key列支持创建bitmap索引

bloomfilter索引

适合基数在100000+的列，列的重复度很低
适合 in 和 = 过滤条件的查询
不支持为 TINYINT、FLOAT、DOUBLE 和 DECIMAL 类型的列创建 Bloom filter 索引
主键模型和明细模型中所有列都可以创建 Bloom filter 索引；聚合模型和更新模型中，只有维度列（即 Key 列）支持创建 Bloom filter 索引

导入

使用建议

【必须】生产禁止使用insert into values() 导入数据
【必须】建议导入批次间隔5s+，也就是攒批写入，尤其是实时场景
【建议】主键模型更新场景，建议开启索引落盘，磁盘强制SSD、NVMe或者更高性能的磁盘。
【建议】比较多ETL（insert into select）的场景，建议开启spill落盘功能，避免内存超过限制

数据生命周期

【建议】使用truncate删除数据，不要使用delete
【必须】完整的UPDATE语法只能用于3.0版本以后的主键模型，禁止高并发 update，建议每次update操作需要间隔分钟以上
【必须】如果使用DELETE删除数据，需要带上where条件，并且禁止并发执行delete，例如要删除id=1，2，3，4，……1000，禁止delete xxx from tbl1 where id=1这样的语句执行1000条，建议delete xxx from tbl1 where id in (1,2,3…,1000)
【必须】drop操作默认会进入FE 回收站，默认保留86400（s），也就是1天（这个期间可以RECOVER恢复，避免误操作），受参数catalog_trash_expire_second控制，超过1天后会进入BE的trash目录，默认保留259200（s），也就是3天（2.5.17，3.0.9，3.1.6之后默认值改为了86400，也就是1天），受参数trash_file_expire_time_sec控制，如果drop后需要尽快释放磁盘，可以调小fe和be的trash保留时间

查询

高并发场景

【建议】尽可能利用分区分桶裁剪，具体参考上文的分区和分桶选择部分
【必须】调大客户的并发限制，可以设置为1000，默认100，SET PROPERTY FOR ‘jack’ ‘max_user_connections’ = ‘1000’;
【必须】开启page cache、query cache

数据精度

【必须】如果需要精确结果的，强制使用decimal类型，不要使用float、double类型

SQL查询

【必须】避免SELECT *，建议指定需要查询的列，例如select col0,col1 from tb1
【必须】避免全表扫描，建议增加过滤的谓词，例如SELECT col0,col1 from tb1 WHERE id=123，select col0,col1 from tb1 where dt>‘2024-01-01’
【必须】避免大数据量的下载，如果要使用，强制使用分页，SELECT col0,col1,col2,...,col50 from tb order by id limit 0,50000
【必须】分页操作需要加上ORDER BY，要不然是无序的
【建议】避免使用一些不必要的函数或者表达式，比如：

- 谓词中含CAST, 可以移除

```
-- bad case
select l_tax
from lineitem
where cast(l_shipdate as varchar) > substr('1990-01-02 12:30:31',1,10);
-- good case
select l_tax
from lineitem
where l_shipdate > '1990-01-02';
```

- 过度使用函数处理表达式

```
-- bad case
select count(1)
from lineitem
where l_shipdate >= regexp_extract("TIME:1996-01-02 20:00:00", "(\\d{4}-\\d{2}-\\d{2})", 1);
-- good case
select count(1)
from lineitem
where l_shipdate >= "1996-01-02"

-- bad case
select count(1)
from lineitem
where DATE_FORMAT(l_shipdate,'%Y-%m-%d') >= "1996-01-02"
-- good case
select count(1)
from lineitem
where l_shipdate >= "1996-01-02"
```

- JOIN
  【必须】关联的字段类型匹配，虽然starrocks已经在内部做了隐式转换来达到最优的性能，不过建议大家使用

类型一致的字段join，避免使用float、double类型join，可能会导致结果不准确
【必须】关联字段建议不要使用函数或者表达式，例如 JOIN on DATE_FORMAT(tb1.col1,'%Y-%m-%d')=DATE_FORMAT(tb2.col1,'%Y-%m-%d')
【必须】2个或多个KW行以上的表JOIN，推荐colocate join
【建议】避免笛卡尔积，查询多个表需要指定连接条件

```
-- bad case
SELECT * FROM table1, table2;
-- good case
SELECT * FROM table1, table2
ON table1.column1 = table2.column1;
```

- 正确关联子查询 * 在子查询中，确保外部查询和子查询之间的列有明确的关联

```
-- bad case
SELECT * FROM table1
WHERE column1 IN (SELECT column2 FROM table2);
-- good case
SELECT * FROM table1
WHERE column1 IN (SELECT column2 FROM table2 WHERE table1.column3 = table2.column3);
```

- 使用AND条件而不是OR

```
-- bad case
SELECT *
FROM table1
JOIN table2
WHERE (table1.column1 = table2.column1 OR table1.column2 = table2.column2);
-- good case
SELECT *
FROM table1
JOIN table2
ON table1.column1 = table2.column1 AND table1.column2 = table2.column2;
```

使用物化视图加速查询

- 精确去重
  以下示例基于一张广告业务相关的明细表 advertiser_view_record ，其中记录了点击日期 click_time 、广告代码 advertiser 、点击渠道 channel 以及点击用户 ID user_id 。

```
CREATE TABLE advertiser_view_record(
click_time DATE,
advertiser VARCHAR(10),
channel VARCHAR(10),
user_id INT
) distributed BY hash(click_time);
```

该场景需要频繁使用如下语句查询点击广告的 UV。

```
SELECT advertiser, channel, count(distinct user_id)
FROM advertiser_view_recordGROUP
BY advertiser, channel;
```

如需实现精确去重查询加速，您可以基于该明细表创建一张物化视图，并使用 bitmap_union() 函数预先聚合数据。

```
CREATE MATERIALIZED VIEW advertiser_uv ASSELECT advertiser, channel, bitmap_union(to_bitmap(use
r_id))
FROM advertiser_view_record
GROUP BY advertiser, channel;
```

物化视图创建完成后，后续查询语句中的子查询 count(distinct user_id) 会被自动改写为 bitmap_union_count (to_bitmap(user_id)) 以便查询命中物化视图。

- 异步物化视图最多支持3层嵌套

利用cache 加速查询

【建议】page cache，建议开启，可以加速数据扫描场景，如果内存有冗余，可以尽可能调大限制，默认是mem_limit*20%
【建议】query cache，建议开启，可以加速单表或多表JOIN的聚合场景
查询中不能包含 rand 、 random 、 uuid 和 sleep 等不确定性 (Nondeterministic) 函数
【建议】data cache，用于存算分离和湖分析场景，建议这两个场景下默认开启

元数据周期性后台刷新

元数据周期性后台刷新方案是 StarRocks 用于加速检索 Iceberg Catalog 中元数据的策略
【建议】可以通过系统变量 plan_mode 调整 Iceberg Catalog 元数据检索方案
元数据缓存周期性刷新与元数据自动异步更新策略配合使用，可以进一步加快数据访问速度，降低从外部数据源读取数据的压力，提升查询性能

监控

慢查询分析

【必须】通过审计插件把fe.audit.log的数据导入一个表方便进行分析慢查询。

监控告警

【建议】登录TM页面->集群服务可以查看监控信息。
原生监控指标也可以到以下页面查看：
FE监控指标页面：http://${fe_leader_ip}:8030/metrics
BE监控页面：http://${be_ip}:8040/metrics

大查询定位

大查询包括扫描大量数据或占用过多 CPU 和内存资源的查询。如果不施加限制，大查询很容易耗尽集群资源并导致系统过载。为了解决这个问题，StarRocks 提供了一系列措施来监控和管理大查询，防止大查询独占集群资源
处理 StarRocks 大查询的总体思路如下：

1. 通过资源组和查询队列对大查询设置自动预防措施
2. 实时监控大查询，并及时终止绕过预防措施的大查询
3. 分析审计日志和大查询日志，研究大查询的模式，优化先前设置的预防机制参数
   具体定位方法：
   查看当前FE上正在运行的查询
   SQL命令： show proc '/current_queries'
   返回结果包括以下几列：

- QueryId
- ConnectionId
- Database：当前查询的DB
- User：用户
- ScanBytes：当前已扫描的数据量，单位Bytes
- ProcessRow：当前已扫描的数据行数
- CPUCostSeconds：当前查询已使用的CPU时间，单位秒。此为多个线程累加的CPU时间，举个例子，如果有两个线程分别占用1秒和2秒的CPU时间，那么累加起来的CPU时间为3秒
- MemoryUsageBytes：当前占用的内存。如果查询涉及到多个BE节点，此值即为该查询在所有BE节点上占用

的内存之和

- ExecTime：查询从发起到现在的时长，单位为毫秒

```
mysql> show proc '/current_queries';
+--------------------------------------+-------------+-----------+-----+---------+----------------
+--------------+-----------------+----------+
| QueryId                              | ConnectionId| Database  | User| ScanBytes| ProcessRows    | CPUCostSeco
nds| MemoryUsageBytes |ExecTime  |
+--------------------------------------+-------------+-----------+-----+---------+----------------
+--------------+-----------------+----------+
| 7c56495f-ae8b-11ed-8ebf-00163e00accc| 4           | tpcds_100g | root| 37.88 MB | 1075769 Rows   | 11.
13 Seconds | 146.70 MB        | 3804     |
| 7d543160-ae8b-11ed-8ebf-00163e00accc| 6           | tpcds_100g | root| 13.02 GB | 487873176 Rows | 8
1.23 Seconds | 6.37 GB          | 2090     |
+--------------------------------------+-------------+-----------+-----+---------+----------------
+--------------+-----------------+----------+
2 rows in set (0.01 sec)
```

查看某个查询在每个BE节点上的资源消耗

SQL命令： show proc '/current_queries/${query_id}/hosts'

返回结果有多行，每行描述该查询在对应BE节点上的执行信息，包括以下几列：

- Host：BE节点信息
- ScanBytes：已经扫描的数据量，单位Bytes
- ScanRows：已经扫描的数据行数
- CPUCostSeconds：已使用的CPU时间
- MemUsageBytes：当前占用的内存

```
mysql> show proc '/current_queries/7c56495f-ae8b-11ed-8ebf-00163e00accc/hosts';
+------------------+----------+-----------+---------------+-------------+
| Host             | ScanBytes| ScanRows  | CpuCostSeconds| MemUsageBytes|
+------------------+----------+-----------+---------------+-------------+
| 172.26.34.185:8060| 11.61 MB | 356252 Rows| 52.93 Seconds | 51.14 MB    |
| 172.26.34.186:8060| 14.66 MB | 362646 Rows| 52.89 Seconds | 50.44 MB    |
| 172.26.34.187:8060| 11.60 MB | 356871 Rows| 52.91 Seconds | 48.95 MB    |
+------------------+----------+-----------+---------------+-------------+
3 rows in set (0.00 sec)
```

大查询业务重保

StarRocks 提供了两种预防工具处理大查询——资源组和查询队列。核心业务可以使用资源组来过滤并熔断大查询。

而查询队列可以帮助您在系统达到并发阈值或资源限制时对新查询请求进行排队，从而防止系统过载。

【建议】利用资源隔离大查询熔断，小查询保底

```
# shortquery_group 资源组用于核心业务重保
CREATE RESOURCE GROUP shortquery_group
TO
    (user='rg1_user1', role='rg1_role1', db='db1', query_type in ('select'), source_ip='192.168.x.x/24'),
WITH (
     'type' = 'short_query',
     'cpu_core_limit' = '10',
     'mem_limit' = '20%');

# bigquery_group 用于大查询熔断，避免大查询将集群资源打满
CREATE RESOURCE GROUP bigquery_group
TO
  (user='rg1_user2', role='rg1_role1', query_type in ('select')),
WITH (
   "type" = 'normal',
   'cpu_core_limit' = '10',
   'mem_limit' = '20%',
   'big_query_cpu_second_limit' = '100',
   'big_query_scan_rows_limit' = '100000',
   'big_query_mem_limit' = '1073741824');
```

常见问题

broker导入失败

使用命令查看导入失败的具体报错:
SELECT * FROM information_schema.loads ORDER BY CREATE_TIME DESC LIMIT 1 \G;

1. 如果是kerberos认证相关的，一定要使用hadoop集群hdfs namenode节点的keytab和principal；
2. 如果是HA相关的，检查mysql命令中hdfs的相关参数是否正确；如果是broker进程相关的，确认是否拉起对应进程并填写对应broker实例名称。

认证相关配置问题

参考下面《运维手册》中的《SR跨集群配置》

StarRocks 会缓存查询结果吗？

StarRocks 不直接缓存最终查询结果。从 2.5 版本开始，StarRocks 会将多阶段聚合查询的第一阶段聚合的中间结果缓存在 Query Cache 里，后续查询可以复用之前缓存的结果，加速计算。Query Cache 占用所在 BE 的内存。

当字段为NULL时，除了is null， 其他所有的计算结果都是false

标准 SQL 中 NULL 和其他表达式计算结果都是null。

StarRocks有DECODE函数吗？

StarRocks 不支持 Oracle 中的 DECODE 函数，StarRocks 语法兼容 MySQL，可以使用CASE WHEN。

StarRocks的主键覆盖是立刻生效的吗？还是说要等后台慢慢合并数据?

StarRocks 的后台合并参考 Google 的 MESA 模型，有两层 compaction，会后台策略触发合并。如果没有合并完成，查询时会合并，但是读出来只会有一个最新的版本，不存在「导入后数据读不到最新版本」的情况。

StarRocks 存储 utf8mb4 的字符，会不会被截断或者乱码？

MySQL的 utf8mb4 是标准的 UTF-8，StarRocks 可以完全兼容。

DELETE 中支持嵌套函数吗？

目前不支持类似如下的嵌套：DELETE from test_new WHERE TO_DAYS(NOW())-to_days(publish_time) >7;。这里'to_days(now())'属于嵌套。
如果一个数据库中有上百张表，USE database 会特别慢。
client连接的时候加上-A参数，比如 mysql -uroot -h127.0.0.1 -P8867 -A。-A不会预读数据库信息，切换database会很快。

VARCHAR 设置成最大值对存储有没有影响？

VARCHAR 是变长存储，存储跟数据实际长度有关，建表时指定不同的 VARCHAR 长度对同一数据的查询性能影响很小。

2021-10在StarRocks里是合法的日期格式吗？可以用作分区字段吗？

不是合法的日期格式，不可以用作分区字段，需要调整成 2021-10-01 再分区。

如何快速统计 StarRocks 库、表的大小，所占的磁盘资源？

库、表的存储大小可以用 SHOW DATA 命令查看。

SHOW DATA; 可以展示当前数据库下所有表的数据量和副本数。

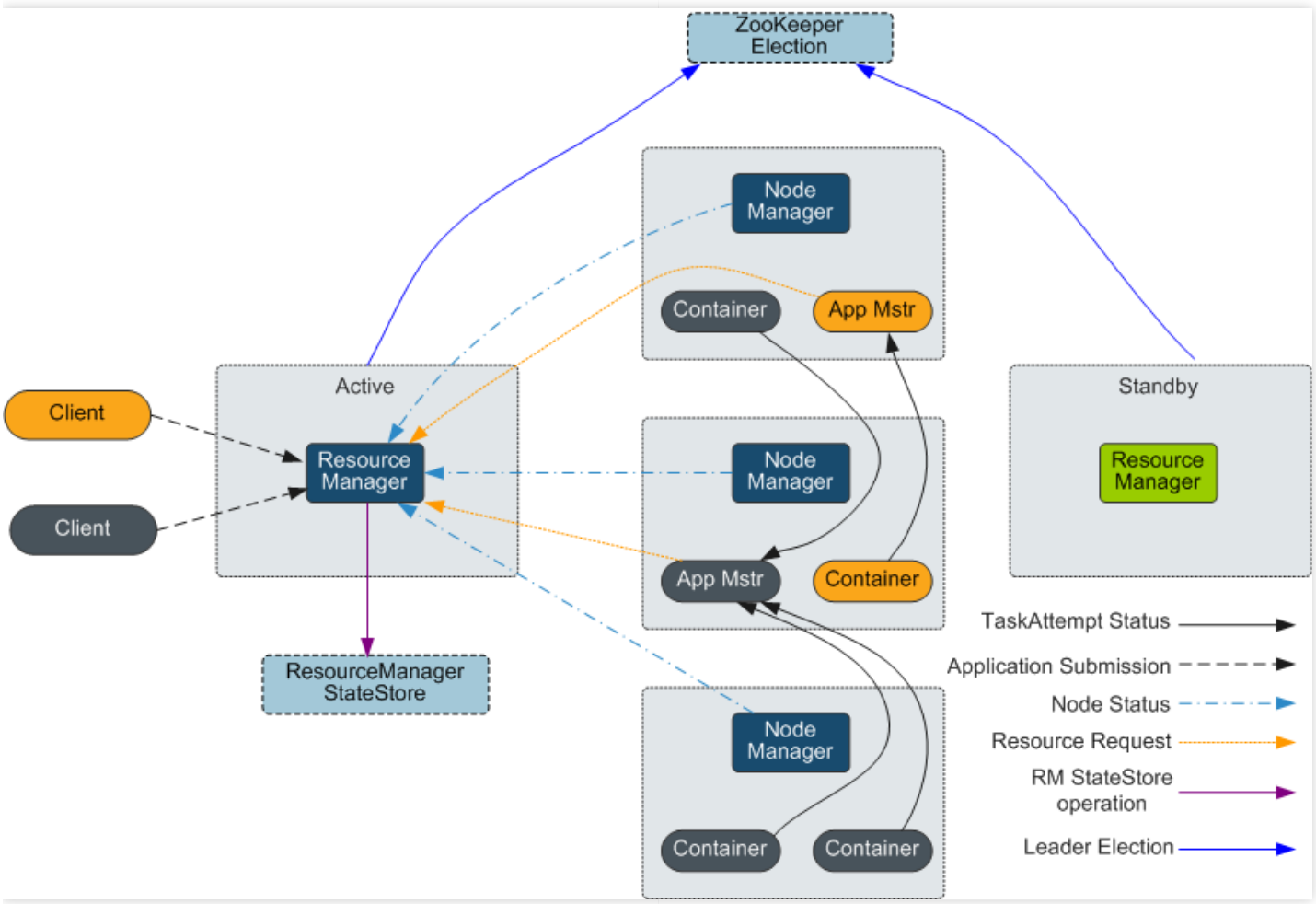
SHOW DATA FROM .; 可以展示指定数据库下某个表的数据量、副本数和统计行数。

MapReduce开发

概述

MapReduce是Hadoop的核心，是Google提出的一个软件架构，用于大规模数据集（大于1TB）的并行运算。概念"Map（映射）"和"Reduce（化简）"及其主要思想，均取自于函数式编程语言及矢量编程语言。
当前的软件实现是指定一个Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的Reduce（化简）函数，用来保证所有映射的键值对共享相同的键组。
MapReduce是用于并行处理大数据集的软件框架。MapReduce的根源是函数性编程中的Map和Reduce函数。Map函数接受一组数据并将其转换为一个键/值对列表，输入域中的每个元素对应一个键/值对。Reduce函数接受Map函数生成的列表，然后根据它们的键缩小键/值对列表。MapReduce具备将大事务分散到不同设备处理的能力，这样原来必须用单台较强服务器才能运行的任务，在分布式环境下也能完成。

MapReduce结构
MapReduce通过实现YARN的Client和ApplicationMaster接口集成到YARN中，利用YARN申请计算所需资源。



示例工程开发

环境准备

开发环境准备

准备项	说明
安装JDK	JDK 8、JDK11 ，推荐使用 konaJDK，下载地址
安装和配置 IDE	按需选择，比如 IntelliJ IDEA 或 Eclipse，示例使用IDEA
安装 Maven	开发环境基础配置，负责构建 Java 应用程序
Maven 配置准备	如果需要本地调试，需要参考 6.开发环境准备 配置 Maven settings.xml，推荐 Maven 3.6.3，下载地址

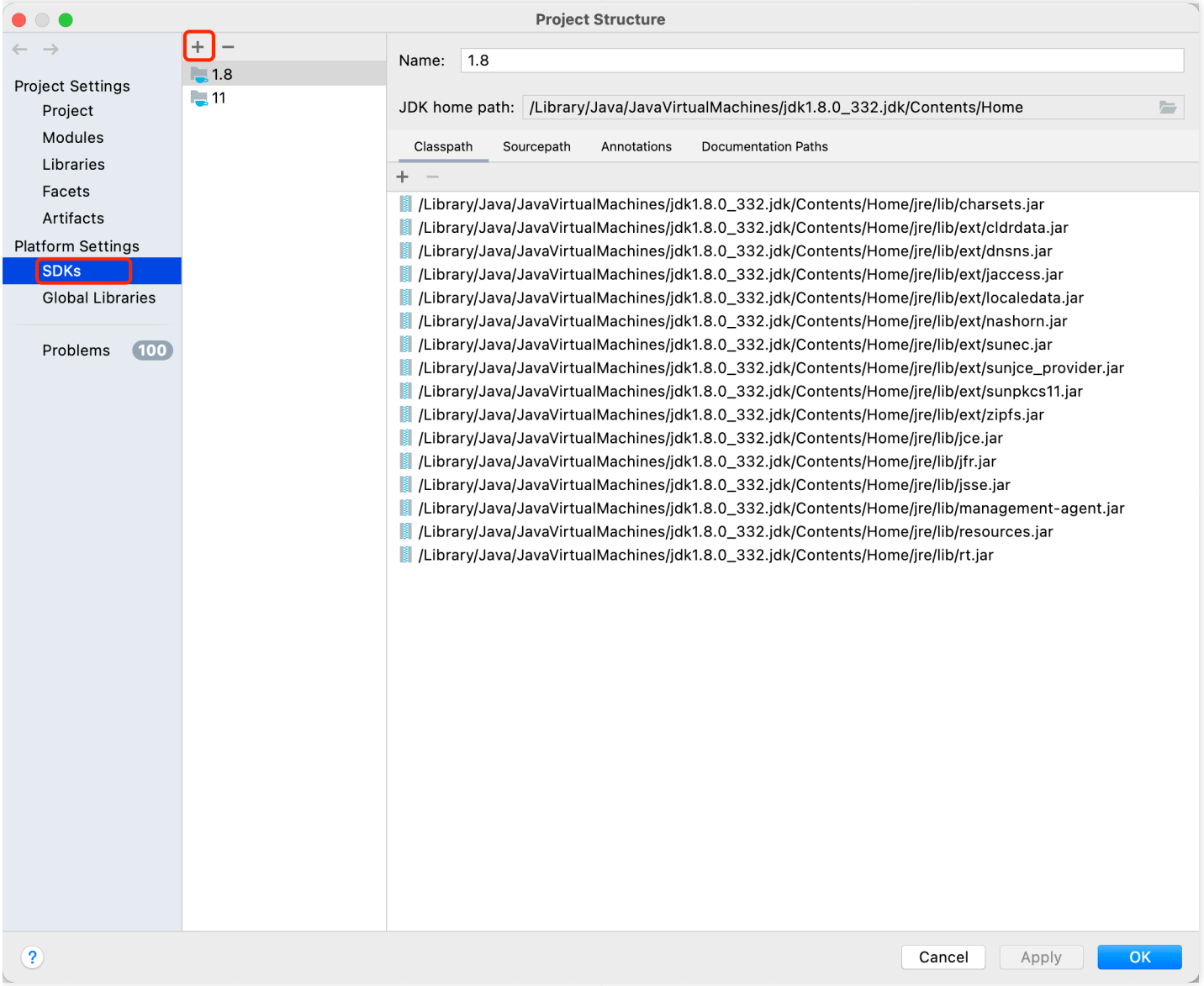
导入示例工程代码

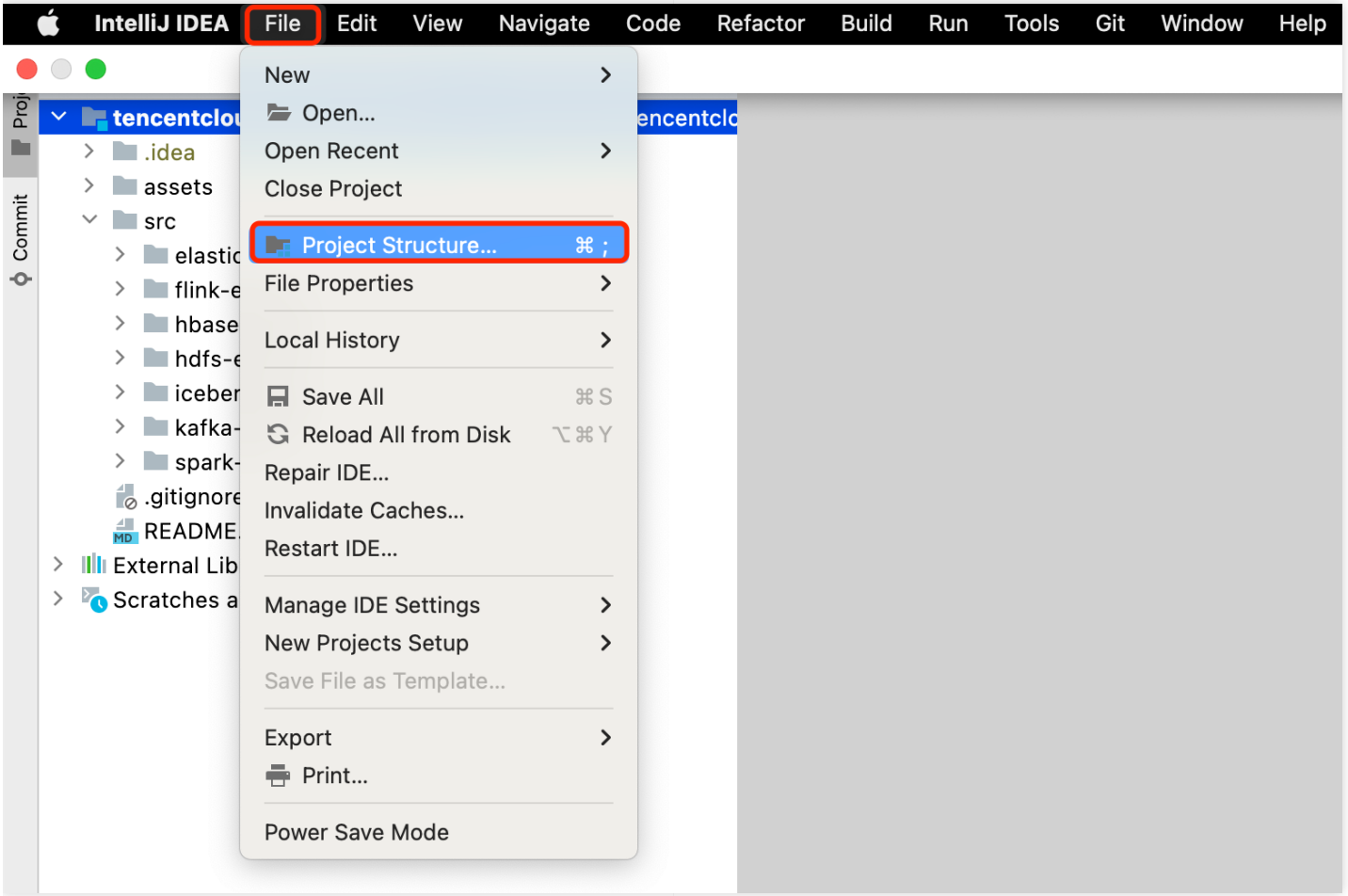
以下以 IntelliJ IDEA 举例，将示例工程代码导入进行说明。
2.1 下载样例代码：https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples/git/files/master

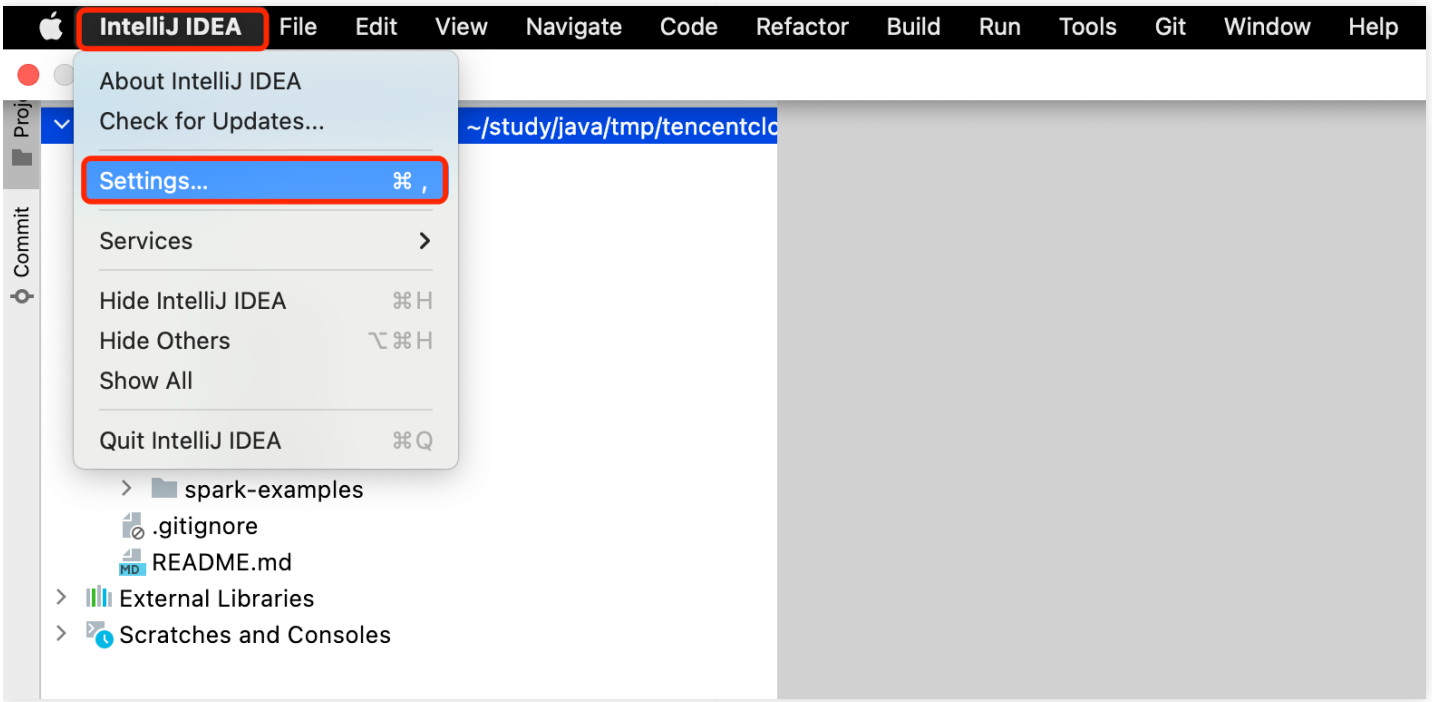
克隆或者直接下载master代码都可以

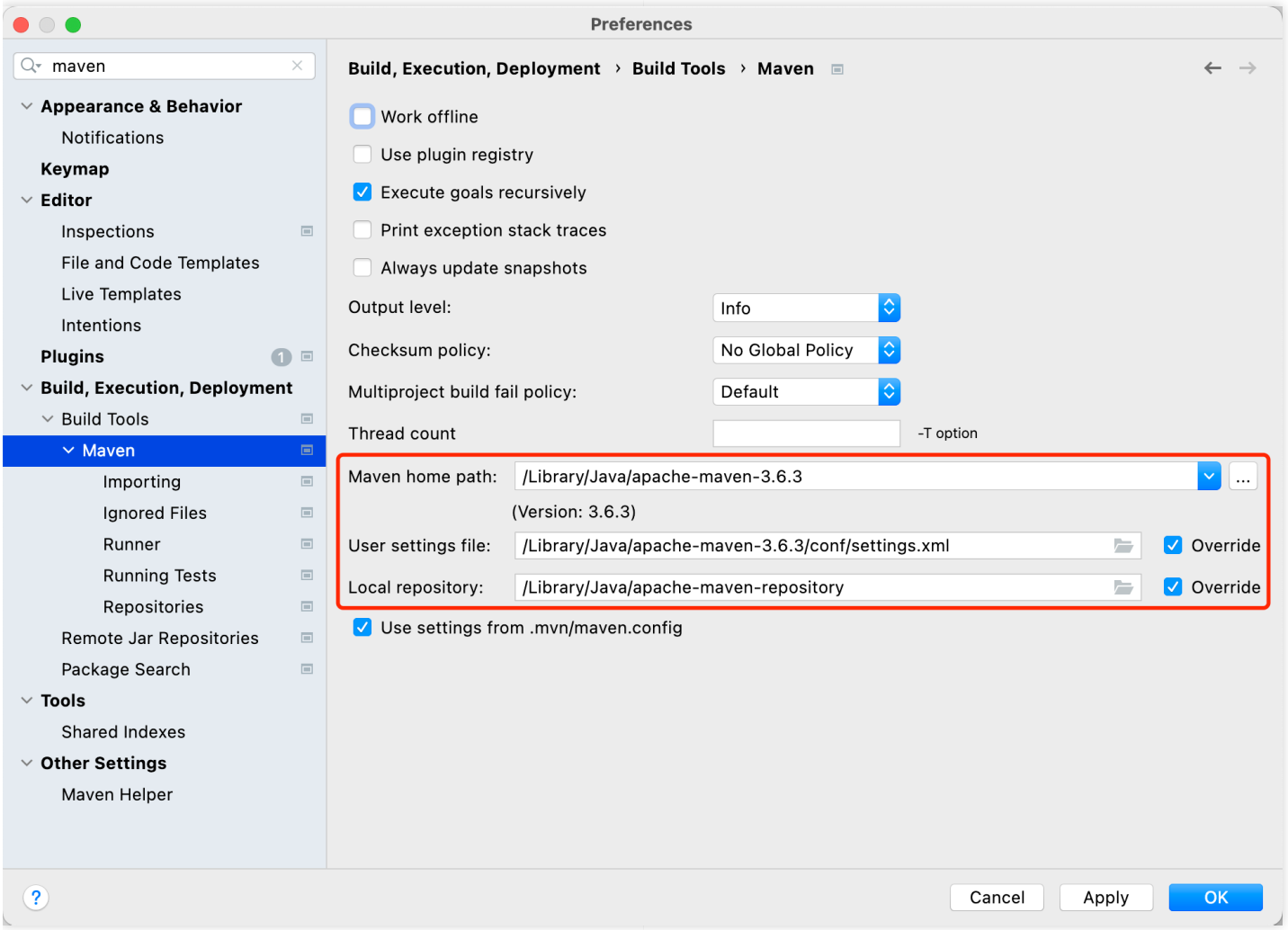
git clone https://g-necm8077.coding.net/public/tencentcloud-tbds-examples/tbds-examples

2.2 导入项目，然后选择JDK、MAVEN和settings文件。


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
100
1.8
11
Name:
1.8
JDK home path:
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home
Classpath
Sourcepath
Annotations
Documentation Paths
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/charsets.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/cldrdata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/dnsns.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/jaccess.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/localedata.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/nashorn.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunec.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunjce_provider.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/sunpkcs11.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/ext/zipfs.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jce.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jfr.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/jsse.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/management-agent.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/resources.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_332.jdk/Contents/Home/jre/lib/rt.jar
Cancel
Apply
OK


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
New
Open...
Open Recent
Close Project
Project Structure... ⌘ ;
File Properties
Local History
Save All ⌘S
Reload All from Disk ⌥⌘Y
Repair IDE...
Invalidate Caches...
Restart IDE...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
.idea
assets
src
.gitignore
README
External Lib
Scratches a
Proj
Commit


IntelliJ IDEA
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tools
Git
Window
Help
About IntelliJ IDEA
Check for Updates...
Settings... ⌘ ,
Services
Hide IntelliJ IDEA ⌘H
Hide Others ⌥⌘H
Show All
Quit IntelliJ IDEA ⌘Q
~/study/java/tmp/tencentclo
spark-examples
.gitignore
README.md
External Libraries
Scratches and Consoles
Proj
Commit



样例代码

POM文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>mapreduce-test</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
<properties>
  <hadoop.version>3.2.2-TBDS-5.3.1.3</hadoop.version>
  <hive.version>3.1.3-TBDS-5.3.1.3</hive.version>
</properties>

<dependencies>
  <!-- Hadoop Core Dependencies -->
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>${hadoop.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>${hadoop.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-common</artifactId>
    <version>${hadoop.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-jobclient</artifactId>
    <version>${hadoop.version}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs</artifactId>
    <version>${hadoop.version}</version>
  </dependency>

  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.32</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.17.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
```

```
        <version>2.17.2</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.17.2</version>
    </dependency>

</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.1.1</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <createDependencyReducedPom>false</createDependencyReducedPom>

                        <artifactSet>
                            <excludes>
                                <exclude>com.google.code.findbugs:jsr305</exclude>
                                <exclude>org.slf4j:*</exclude>
                                <exclude>log4j:*</exclude>
                            </excludes>
                        </artifactSet>
                        <filters>
                            <filter>
                                <!-- Do not copy the signatures in the META-INF folder.
                                Otherwise, this might cause SecurityExceptions when using the JAR. -->
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
                                    <exclude>META-INF/*.RSA</exclude>
                                </excludes>
                            </filter>
                        </filters>
                        <transformers>
                            <!-- 注意不要漏掉这个transformer，否则运行时会报类似"Could not find a suitable ta
ble factory
```

```
                    for 'org.apache.flink.table.delegation.ExecutorFactory" 的错误-->
                    <transformer
                        implementation="org.apache.maven.plugins.shade.resource.ServicesResource
Transformer"/>

                    <transformer
                        implementation="org.apache.maven.plugins.shade.resource.ManifestResourc
eTransformer">
                      <mainClass>com.tencent.tbds.WordCount</mainClass>
                    </transformer>
                  </transformers>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
    </build>
</project>
```

示例代码

```
public class WordCount {

    public static class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        private static final Logger logger = LoggerFactory.getLogger(WordCountMapper.class);

        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOException, Interrupte
dException {
            String line = value.toString();
            String[] parts = line.split(",");
            logger.info("Processing line: " + line);
            String[] words = line.split("\\s+");
            for (String w : words) {
                word.set(w);
                context.write(word, one);
                logger.debug("Emitting: (" + w + ", 1)");
            }
        }
    }

    public static class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
```

```
    private static final Logger logger = LoggerFactory.getLogger(WordCountReducer.class);

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOExceptio
n, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        String formattedOutput = String.format("%-30s %d", key.toString(), sum);
        context.write(new Text(formattedOutput), null);
        logger.info("Reducing: (" + key.toString() + ", " + sum + ")");
    }
  }

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: WordCount <input path> <output path>");
      System.exit(-1);
    }

    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Word Count");

    job.setJarByClass(WordCount.class);
    job.setMapperClass(WordCountMapper.class);
    job.setReducerClass(WordCountReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

示例说明

- 数据准备

hdfs dfs -mkdir -p /tmp/test/input/
hdfs dfs -put input.txt /tmp/test/input/

#input.txt示例
LiuYang,female,20
YuanJing,male,10
GuoYijun,male,20
CaiXuyu,female,20
Liyuan,male,20
FangBo,female,50
LiuYang,female,20
YuanJing,male,10
GuoYijun,male,20
CaiXuyu,female,50
FangBo,female,60

- 运行任务

hadoop jar mapreduce-test-1.0-SNAPSHOT.jar /tmp/test/input /tmp/test/output

- 查看结果

hdfs dfs -ls /tmp/test/output/
hdfs dfs -cat /tmp/test/output/part-r-00000

```
[root@tbds-10-4-4-19 ~]# hdfs dfs -cat /tmp/test/output/part-r-00000
2024-11-28T17:42:53,014 INFO security.UserGroupInformation: Hadoop UGI authentication : KERBEROS
2024-11-28T17:42:53,263 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
2024-11-28T17:42:53,316 INFO Configuration.deprecation: No unit for dfs.client.datanode-restart.timeout(30) assuming SECONDS
CaiXuyu,female,50               2
FangBo,female,50                1
FangBo,female,60                1
GuoYijun,male,5                 1
GuoYijun,male,50                1
LiuYang,female,20               2
Liyuan,male,20                  1
YuanJing,male,10                2
```

api接口

MapReduce Java API接口介绍

https://hadoop.apache.org/docs/r3.2.2/api/index.html

MapReduce REST API接口介绍

https://hadoop.apache.org/docs/r3.2.2/hadoop-mapreduce-client/hadoop-mapreduce-client-hs/HistoryServerRest.html

常用操作

常用命令

```
#列出所有作业
hadoop job -list

#查看作业状态
hadoop job -status <job-id>

#查看作业日志
hadoop job -logs <job-id>

#杀死作业
hadoop job -kill <job-id>

#提交 MapReduce 作业
hadoop jar <jar-file> <main-class> <input-path> <output-path>
```

配置MapReduce任务日志归档和清理机制

执行一个MapReduce应用会产生两种类型日志文件：作业日志和任务日志。

- 作业日志由MRApplicationMaster产生，详细记录了作业启动时间、运行时间，每个任务启动时间、运行时间、Counter值等信息。此日志内容被HistoryServer解析以后用于查看作业执行的详细信息。
- 任务日志记录了每个运行在Container中的任务输出的日志信息。默认情况下，任务日志只会存放在各NodeManager的本地磁盘上。打开日志聚合功能后，NodeManager会在作业运行完成后将本地的任务日志进行合并，写入到HDFS中。
  由于MapReduce的作业日志和任务日志（聚合功能开启的情况下）都保存在HDFS上。对于计算任务量大的集群，如果不进行合理的配置对日志文件进行定期归档和删除，日志文件将占用HDFS大量内存空间，增加集群负载。

作业日志参数

操作：TM-YARN-配置管理-mapred-site.xml

参数	描述	默认值
mapreduce.jobhistory.cleaner.enable	是否开启作业日志文件清理功能。	true
mapreduce.jobhistory.cleaner.interval-ms	作业日志文件清理启动周期。只有保留时间比“mapreduce.jobhistory.max-age-ms”更长的日志文件才会被清除。	604800000（7天）
mapreduce.jobhistory.max-age-ms	比此项设置的毫秒数保留时间更长的作业日志文件将被清理。	604800000（7天）

任务日志参数

参数	描述	默认值
yarn.log-aggregation-enable	是否启用日志聚合。日志聚合收集每个容器的日志，并在应用程序完成后将这些日志移动到文件系统（例如 HDFS）。	true
yarn.nodemanager.remote-app-log-dir	当应用程序运行结束后，日志被转移到的HDFS目录（启用日志聚集功能时有效）	/emr/logs
yarn.log-aggregation.retain-seconds	设置MapReduce任务日志在HDFS上的保留时间。设置为“-1”时日志文件永久保存。	604800（7天）
yarn.log-aggregation.retain-check-interval-seconds	设置MapReduce任务日志清理任务的检查周期（秒）。设置为“-1”时检查周期为日志保留时间的十分之一。	604800（7天）

常见问题

ResourceManager进行主备切换后，任务中断后运行时间过长

问题

在MapReduce任务运行过程中，ResourceManager发生主备切换，切换完成后，MapReduce任务继续执行，此时任务的运行时间过长。

解答

因为ResourceManager HA已启用，但是Work-preserving RM restart功能未启用。
如果Work-preserving RM restart功能未启用，ResourceManager切换时container会被kill，然后导致Application Master超时。
可以通过如下方式启用Work-preserving RM restart功能：
TM界面搜索框中“yarn.resourcemanager.work-preserving-recovery.enabled”，设置参数值为“true”。保存配置后，在业务低峰期重启Yarn配置过期的实例。

在缓存中找不到HDFS_DELEGATION_TOKEN如何处理

问题

安全模式下，在缓存中找不到HDFS_DELEGATION_TOKEN

解答

在MapReduce中，默认情况下，任务完成之后，HDFS_DELEGATION_TOKEN将会被删除。因此如果在下一个任务

中再次使用HDFS_DELEGATION_TOKEN，缓存中将会找不到HDFS_DELEGATION_TOKEN。
为了能够在随后的工作中再次使用同一个Token，为MapReduce任务配置参数。当参数为false时，用户能够再次使用同一个Token。

```
jobConf.setBoolean("mapreduce.job.complete.cancel.delegation.tokens", false);
```

Hudi开发

概述

Apache Hudi是一个开放数据湖平台，基于其高性能开放表格式，为数据湖带来数据库功能。Hudi通过强大的增量处理框架重新构想了缓慢的老式批量数据处理，用于低延迟分钟级分析。

快速使用

Spark SQL-Hudi基本使用

```
spark-sql --master yarn \
--deploy-mode client \
--num-executors 2 \
--executor-memory 1g \
--executor-cores 2 \
--jars /usr/local/service/spark/jars/hudi-spark3.4-bundle_2.12-*.jar \
--conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer' \
--conf 'spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension' \
--conf 'spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog'
```

建表

建表参数

参数名	默认值	说明
primaryKey	uuid	表的主键名，多个字段用逗号分隔。 同 hoodie.datasource.write.recordkey.field
preCombineField		表的预合并字段。 同 hoodie.datasource.write.precombine.field
type	COW	创建的表类型： type = 'COW' type = 'MOR' 同 hoodie.datasource.write.table.type

创建非分区表

```
-- 创建一个cow表，primaryKey 'id'，并提供preCombineField
create table hudi_cow_tbl (
  id int,
  name string,
  price double,
  ts bigint
) using hudi
tblproperties (
  type = 'cow',
  primaryKey = 'id',
```

```
  preCombineField = 'ts'
);

-- 创建一个mor非分区表
create table hudi_mor_tbl (
  id int,
  name string,
  price double,
  ts bigint
) using hudi
tblproperties (
  type = 'mor',
  primaryKey = 'id',
  preCombineField = 'ts'
);
```

创建分区表

```
-- 创建一个cow分区外部表，指定primaryKey和preCombineField
create table hudi_cow_pt_tbl (
  id bigint,
  name string,
  ts bigint,
  dt string,
  hh string
) using hudi
tblproperties (
  type = 'cow',
  primaryKey = 'id',
  preCombineField = 'ts'
 )
partitioned by (dt, hh)
location '/tmp/hudi/hudi_cow_pt_tbl';
```

通过CTAS (Create Table As Select)建表

通过CTAS从其他表加载数据

```
-- 创建内部表
create table parquet_mngd using parquet location 'hdfs:///tmp/parquet_dataset/*.parquet';

-- 通过CTAS加载数据
```

```
create table hudi_ctas_cow_pt_tbl2 using hudi location 'hdfs:/tmp/hudi/hudi_tbl/' options (
  type = 'cow',
  primaryKey = 'id',
  preCombineField = 'ts'
 )
partitioned by (datestr) as select * from parquet_mngd;
```

插入数据

默认情况下，如果提供了preCombineKey，则insert into的写操作类型为upsert，否则使用insert。

向非分区表插入数据

```
insert into hudi_cow_tbl select 1, 'a1', 20;insert into hudi_mor_tbl select 1, 'a1', 20, 1000;
```

向分区表动态分区插入数据

```
insert into hudi_cow_pt_tbl partition (dt, hh)select 1 as id, 'a1' as name, 1000 as ts, '2024-07-17' as dt, '
10' as hh;
```

向分区表静态分区插入数据

```
insert into hudi_cow_pt_tbl partition(dt = '2024-07-07', hh='11') select 2, 'a2', 1000;
```

使用bulk_insert插入数据

```
-- 向指定preCombineKey的表插入数据，则写操作为upsert
insert into hudi_mor_tbl select 1, 'a1_1', 20, 1001;
select id, name, price, ts from hudi_mor_tbl;
1   a1_1    20.0    1001

-- 向指定preCombineKey的表插入数据，指定写操作为bulk_insert
set hoodie.sql.bulk.insert.enable=true;
set hoodie.sql.insert.mode=non-strict;

insert into hudi_mor_tbl select 1, 'a1_2', 20, 1002;
select id, name, price, ts from hudi_mor_tbl;
1       a1_2    20.0    1002
1       a1_1    20.0    1001
```

查询数据

普通查询

```
select fare, begin_lon, begin_lat, ts from  hudi_trips_snapshot where fare > 20.0
```

时间旅行查询

```
-- 关闭前面开启的bulk_insert
set hoodie.sql.bulk.insert.enable=false;

create table hudi_cow_pt_tbl1 (
  id bigint,
  name string,
  ts bigint,
  dt string,
  hh string
) using hudi
tblproperties (
  type = 'cow',
  primaryKey = 'id',
  preCombineField = 'ts'
 )
partitioned by (dt, hh)
location '/tmp/hudi/hudi_cow_pt_tbl1';

-- 插入一条id为1的数据
insert into hudi_cow_pt_tbl1 select 1, 'a0', 1000, '2024-07-17', '10';
select * from hudi_cow_pt_tbl1;

-- 修改id为1的数据
insert into hudi_cow_pt_tbl1 select 1, 'a1', 1001, '2024-07-17', '10';
select * from hudi_cow_pt_tbl1;

-- 基于第一次提交时间进行时间旅行
select * from hudi_cow_pt_tbl1 timestamp as of '20240724154822829' where id = 1;

-- 其他时间格式的时间旅行写法
select * from hudi_cow_pt_tbl1 timestamp as of '2024-07-24 15:48:22.829' where id = 1;

select * from hudi_cow_pt_tbl1 timestamp as of '2024-07-24' where id = 1;
```

更新数据

update

```
-- 语法
UPDATE tableIdentifier SET column = EXPRESSION(,column = EXPRESSION) [ WHERE boolExpression]

-- 执行更新
update hudi_mor_tbl set price = price * 2, ts = 1111 where id = 1;

update hudi_cow_pt_tbl1 set name = 'a1_1', ts = 1001 where id = 1;

-- update using non-PK field
update hudi_cow_pt_tbl1 set ts = 1111 where name = 'a1_1';
```

MergeInto

```
-- 语法
MERGE INTO tableIdentifier AS target_alias
USING (sub_query | tableIdentifier) AS source_alias
ON <merge_condition>
[ WHEN MATCHED [ AND <condition> ] THEN <matched_action> ]
[ WHEN MATCHED [ AND <condition> ] THEN <matched_action> ]
[ WHEN NOT MATCHED [ AND <condition> ]  THEN <not_matched_action> ]

<merge_condition> =A equal bool condition
<matched_action>  =
 DELETE  |
 UPDATE SET *  |
 UPDATE SET column1 = expression1 [, column2 = expression2 ...]
<not_matched_action>  =
 INSERT *  |
 INSERT (column1 [, column2 ...]) VALUES (value1 [, value2 ...])

-- 执行案例
-- 1、准备source表：非分区的hudi表，插入数据
create table merge_source (id int, name string, price double, ts bigint) using hudi
tblproperties (primaryKey = 'id', preCombineField = 'ts');
insert into merge_source values (1, "old_a1", 22.22, 2900), (2, "new_a2", 33.33, 2000), (3, "new_a3", 44.4
4, 2000);

merge into hudi_mor_tbl as target
using merge_source as source
on target.id = source.id
when matched then update set *
when not matched then insert *
;
```

```
-- 2、准备source表：分区的parquet表，插入数据
create table merge_source2 (id int, name string, flag string, dt string, hh string) using parquet;
insert into merge_source2 values (1, "new_a1", 'update', '2024-07-17', '10'), (2, "new_a2", 'delete', '202
4-07-17', '11'), (3, "new_a3", 'insert', '2024-07-17', '12');

merge into hudi_cow_pt_tbl1 as target
using (
  select id, name, '2000' as ts, flag, dt, hh from merge_source2
) source
on target.id = source.id
when matched and flag != 'delete' then
 update set id = source.id, name = source.name, ts = source.ts, dt = source.dt, hh = source.hh
when matched and flag = 'delete' then delete
when not matched then
 insert (id, name, ts, dt, hh) values(source.id, source.name, source.ts, source.dt, source.hh)
;
```

删除数据

语法

```
DELETE FROM tableIdentifier [ WHERE BOOL_EXPRESSION]
```

用法

```
delete from hudi_cow_nonpcf_tbl where uuid = 1;

delete from hudi_mor_tbl where id % 2 = 0;

-- 使用非主键字段删除
delete from hudi_cow_pt_tbl1 where name = 'a1_1';
```

Flink使用Hudi

YARN-session模式

Yarn-Session模式
启动集群
bin/yarn-session.sh -s 2 -jm 4096 -tm 4096 -d

启动sql client
bin/sql-client.sh embedded -s yarn-session

建表及插入数据

```
set sql-client.execution.result-mode=tableau;
set execution.checkpointing.interval=1000;

-- 创建hudi表
CREATE TABLE t1(
  uuid VARCHAR(20) PRIMARY KEY NOT ENFORCED,
  name VARCHAR(10),
  age INT,
  ts TIMESTAMP(3),
  `partition` VARCHAR(20)
)
PARTITIONED BY (`partition`)
WITH (
  'connector' = 'hudi',
  'path' = 'hdfs://HDFS78000435/tmp/hudi_flink/t1',
  'table.type' = 'MERGE_ON_READ' –- 默认是COW
);
或如下写法
CREATE TABLE flink_hudi_mor(
  uuid VARCHAR(20),
  name VARCHAR(10),
  age INT,
  ts TIMESTAMP(3),
  `partition` VARCHAR(20),
  PRIMARY KEY(uuid) NOT ENFORCED
)
PARTITIONED BY (`partition`)
WITH (
  'connector' = 'hudi',
  'path' = 'hdfs://HDFS78000435/tmp/hudi_flink/flink_hudi_mor',
  'table.type' = 'MERGE_ON_READ'
);

-- 插入数据
```

```
INSERT INTO t1 VALUES
  ('id1','Danny',23,TIMESTAMP '1970-01-01 00:00:01','par1'),
  ('id2','Stephen',33,TIMESTAMP '1970-01-01 00:00:02','par1'),
  ('id3','Julian',53,TIMESTAMP '1970-01-01 00:00:03','par2'),
  ('id4','Fabian',31,TIMESTAMP '1970-01-01 00:00:04','par2'),
  ('id5','Sophia',18,TIMESTAMP '1970-01-01 00:00:05','par3'),
  ('id6','Emma',20,TIMESTAMP '1970-01-01 00:00:06','par3'),
  ('id7','Bob',44,TIMESTAMP '1970-01-01 00:00:07','par4'),
  ('id8','Han',56,TIMESTAMP '1970-01-01 00:00:08','par4');
```

Flink SQL通过Hudi HMS Catalog读写Hudi并同步Hive表(推荐这种方式)

它可以像Spark SQL创建表一样，直接将表建立在Hive中，并且表结构与Hive SQL和Spark SQL兼容，也就是Flink Hudi HMS Catalog中创建的表，可以同时使用Flink SQL、Hive SQL、Spark SQL、Trino查询，也可以同时使用Flink SQL、Spark SQL写Hudi。不像方式二中介绍的方式，Flink SQL写Hudi的表不能被Hive/Spark使用，只能通过同步表的方式。另外在Flink Hudi HMS Catalog中和Spark SQL一样默认开启同步Hive,也就是对于MOR表默认会同步创建对应的_ro表和_rt表，至于COW表因为同步的表名和创建的表名一样，所以读写是同一张表。总之和Spark SQL创建表、读写一致。
如果Hudi表类型是MERGE_ON_READ模式，那么映射的Hive表将会有2张，一张后缀为_rt(实时表)，另一张表后缀为_ro(读优化表)。后缀 _rt 对应的Hive表中存储的是Base文件Parquet格式数据+log Avro格式数据，也就是全量数据。后缀为 _ro Hive表中存储的是Base文件对应的数据。

```
set sql-client.execution.result-mode=tableau;
set execution.checkpointing.interval=1000;
-- 创建 catalog 信息
CREATE CATALOG hudi_catalog WITH (
    'type' = 'hudi',
    'mode' = 'hms',
    'default-database' = 'default',
    'hive.conf.dir' = '/usr/local/service/hive/conf',
    'table.external' = 'true'
);

use catalog hudi_catalog;
use hudi;

-- sets up the result mode to tableau to show the results directly in the CLI
set execution.result-mode=tableau;
```

```
CREATE TABLE test_hudi_flink_mor (
  id int PRIMARY KEY NOT ENFORCED,
  name VARCHAR(10),
  price int,
  ts int,
  dt VARCHAR(10)
)
PARTITIONED BY (dt)
WITH (
  'connector' = 'hudi',
  'path' = 'hdfs://HDFS78000007/tmp/hudi/test_hudi_flink_mor',
  'table.type' = 'MERGE_ON_READ',
  'hoodie.datasource.write.keygenerator.class' = 'org.apache.hudi.keygen.ComplexAvroKeyGenerator',
  'hoodie.datasource.write.recordkey.field' = 'id',
  'hoodie.datasource.write.hive_style_partitioning' = 'true',
  'hive_sync.conf.dir'='/usr/local/service/hive/conf',
  'spark.version'='spark3.4.2',
  'hoodie.datasource.write.precombine.field'='ts'
);

insert into test_hudi_flink_mor values (1,'hudi',10,100,'2024-07-17'),(2,'hudi',10,100,'2024-07-17');

CREATE TABLE test_hudi_flink_cow (
  id int PRIMARY KEY NOT ENFORCED,
  name VARCHAR(10),
  price int,
  ts int,
  dt VARCHAR(10)
)
PARTITIONED BY (dt)
WITH (
  'connector' = 'hudi',
  'hoodie.datasource.write.keygenerator.class' = 'org.apache.hudi.keygen.ComplexAvroKeyGenerator',
  'hoodie.datasource.write.recordkey.field' = 'id',
  'hoodie.datasource.write.hive_style_partitioning' = 'true',
  'hive_sync.conf.dir'='/usr/local/service/hive/conf',
  'hoodie.datasource.write.precombine.field'='ts'
);

insert into test_hudi_flink_cow values (1,'hudi',10,100,'2024-07-17'),(2,'hudi',10,100,'2024-07-17');
```

Trino使用Hudi

通过Trino-cli 连接trino

client/trino-cli --server https://10.206.16.246:9443 --catalog hudi --schema default --user hadoop --password --insecure
密码为hadoop对应的Ldap密码，默认为机器密码

```
trino:default> exit;
[hadoop@10 trino]$ client/trino-cli --server https://10.206.16.246:9443 --catalog hudi --schema default --user hadoop --password --insecure
Password:
trino:default> show tables;
       Table
---------------------
 h_1
 hive_test_111
 hudi_cow_nonpcf_tbl
 t1
 testlubao
(5 rows)

Query 20240718_074301_00001_6vgyw, FINISHED, 4 nodes
Splits: 29 total, 29 done (100.00%)
0.71 [5 rows, 131B] [7 rows/s, 186B/s]

trino:default> select * from hudi_cow_nonpcf_tbl;
 _hoodie_commit_time | _hoodie_commit_seqno  |  _hoodie_record_key   | _hoodie_partition_path |                          _hoodie_file_n>
---------------------+-----------------------+-----------------------+------------------------+---------------------------------------->
 20240716201035806   | 20240716201035806_0_0 | 20240716201035806_0_0 |                        | 6258e3cd-c590-4885-88e3-9e650067f627-0_0->
 20240716202247252   | 20240716202247252_0_1 | 20240716202247252_0_0 |                        | 6258e3cd-c590-4885-88e3-9e650067f627-0_0->
(2 rows)
```

Hive使用Hudi

beeline -u 'jdbc:hive2://x.x.x.x:2181,x.x.x.x:2181,x.x.x.x:2181/default;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2' -nhive -pxxx

注：hive不支持直接读取Flink SQL的MOR表，只支持读取同步到hive生成的_rt和_ro表，并且需要flink-sql mor表触发了文件合并生成.parquet文件后才能读取到数据，trino读flink hudi表也是一样的。

常用参数

本节介绍Hudi重要配置的详细信息，更多配置请参考hudi官网：https://hudi.apache.org/docs/configurations

写入操作配置

参数	描述	默认值
hoodie.datasource.write.table.name	指定写入的hudi表名。	无
hoodie.datasource.write.operation	写hudi表指定的操作类型，当前支持upsert、delete、insert、bulk_insert等方式。 upsert：更新插入混合操作 delete：删除操作 insert：插入操作 bulk_insert： 用于初始建表导入数据， 注意初始建表禁止使用upsert、insert方式 insert_overwrite：对静态分区执行insert overwrite insert_overwrite_table：动态分区执行insert overwrite，该操作并不会立刻删除全表做overwrite，会逻辑上重写hudi表的元数据，无用数据后续由hudi的clean机制清理。效率比bulk_insert + overwrite 高	upsert
hoodie.datasource.write.table.type	指定hudi表类型，一旦这个表类型被指定，后续禁止修改该参数，可选值MERGE_ON_READ。	COPY_ON_WRITE
hoodie.datasource.write.precombine.field	该值用于在写之前对具有相同的key的行进行合并去重。	ts
hoodie.datasource.write.payload.class	在更新过程中，该类用于提供方法将要更新的记录和更新的记录做合并，该实现可插拔，如要实现自己的合并逻辑，可自行编写。	org.apache.hudi.common.model.DefaultHoodieRecordPayload
hoodie.datasource.write.recordkey.field	用于指定hudi的主键，hudi表要求有唯一主键。	UUID
hoodie.datasource.write.partitionpath.field	用于指定分区键，该值配合hoodie.datasource.write.keygenerator.class使用可以满足不同的分区场景。	无
hoodie.datasource.write.hive_style_partitioning	用于指定分区方式是否和hive保持一致，建议该值设置为true。	true
hoodie.datasource.write.keygenerator.class	配合hoodie.datasource.write.partitionpath.field，hoodie.datasource.write.recordkey.field产生主键和分区方式。 说明：写入设置KeyGenerator与表保存的参数值不一致时将提示需要保持一致。	org.apache.hudi.keygen.ComplexKeyGenerator

同步Hive表配置

参数	描述	默认值
hoodie.datasource.hive_sync.enable	是否同步hudi表信息到hive metastore。 注意：建议该值设置为true，统一使用hive管理hudi表。	false
hoodie.datasource.hive_sync.database	要同步给hive的数据库名。	default
hoodie.datasource.hive_sync.table	要同步给hive的表名，建议这个值和hoodie.datasource.write.table.name保证一致。	unknown
hoodie.datasource.hive_sync.username	同步hive时，指定的用户名。	hive
hoodie.datasource.hive_sync.password	同步hive时，指定的密码。	hive
hoodie.datasource.hive_sync.jdbcurl	连接hive jdbc指定的连接。	""
hoodie.datasource.hive_sync.use_jdbc	是否使用hive jdbc方式连接hive同步hudi表信息。建议该值设置为false，设置为false后jdbc连接相关配置无效。	true
hoodie.datasource.hive_sync.partition_fields	用于决定hive分区列。	""
hoodie.datasource.hive_sync.partition_extractor_class	用于提取hudi分区列值，将其转换成hive分区列。	org.apache.hudi.hive.SlashEncodedDayPartitionValueExtractor
hoodie.datasource.hive_sync.support_timestamp	当hudi表存在timestamp类型字段时，需指定此参数为true，以实现同步timestamp类型到hive元数据中。该值默认为false，默认将timestamp类型同步为BIGINT，默认情况可能导致使用sql查询包含timestamp类型字段的hudi表出现错误。	true

Index相关配置

参数	描述	默认值
hoodie.index.class	用户自定义索引的全路径名，索引类必须为HoodieIndex的子类，当指定该配置时，其会优先于hoodie.index.type配置。	""
hoodie.index.type	使用的索引类型，默认为布隆过滤器。可能的选项是[BLOOM | HBASE | GLOBAL_BLOOM | SIMPLE | GLOBAL_SIMPLE]。 布隆过滤器消除了对外部系统的依赖，并存储在Parquet数据文件的页脚中。	BLOOM
hoodie.index.bloom.num_entries	存储在布隆过滤器中的条目数。 假设maxParquetFileSize为128MB，averageRecordSize为1024B，因此，一个文件中的记录总数约为130K。 默认值（60000）大约是此近似值的一半。 注意：将此值设置得太低，将产生很多误报，并且索引查找将必须扫描比其所需的更多的文件；如果将其设置得非常高，将线性增加每个数据文件的大小（每50000个条目大约4KB）。	60000
hoodie.index.bloom.fpp	根据条目数允许的错误率。 用于计算应为布隆过滤器分配多少位以及哈希函数的数量。通常将此值设置得很低（默认值：0.000000001），在磁盘空间上进行权衡以降低误报率。	0.000000001
hoodie.bloom.index.parallelism	索引查找的并行度，其中涉及Spark Shuffle。 默认情况下，根据输入的工作负载特征自动计算的。	0
hoodie.bloom.index.prune.by.ranges	为true时，从文件框定信息，可以加快索引查找的速度。 如果键具有单调递增的前缀，例如时间戳，则特别有用。	true

参数	描述	默认值
hoodie.bloom.index.use.caching	为true时，将通过减少用于计算并行度或受影响分区的IO来缓存输入的RDD以加快索引查找。	true
hoodie.bloom.index.use.treebased.filter	为true时，启用基于间隔树的文件过滤优化。与暴力模式相比，此模式可根据键范围加快文件过滤速度。	true
hoodie.bloom.index.bucketized.checking	为true时，启用了桶式布隆过滤。这减少了在基于排序的布隆索引查找中看到的偏差。	true
hoodie.bloom.index.keys.per.bucket	仅在启用BloomIndexBucketizedChecking并且索引类型为bloom的情况下适用。 此配置控制"存储桶"的大小，该大小可跟踪对单个文件进行的记录键检查的次数，并且是分配给执行布隆过滤器查找的每个分区的工作单位。 较高的值将分摊将布隆过滤器读取到内存的固定成本。	10000000
hoodie.bloom.index.update.partition.path	仅在索引类型为GLOBAL_BLOOM时适用。 为true时，当对一个已有记录执行包含分区路径的更新操作时，将会导致把新记录插入到新分区，而把原有记录从旧分区里删除。为false时，只对旧分区的原有记录进行更新。	true
hoodie.index.hbase.zkquorum	仅在索引类型为HBASE时适用，必填选项。要连接的HBase ZK Quorum URL。	无
hoodie.index.hbase.zkport	仅在索引类型为HBASE时适用，必填选项。要连接的HBase ZK Quorum端口。	无
hoodie.index.hbase.zknode.path	仅在索引类型为HBASE时适用，必填选项。这是根znode，它将包含HBase创建及使用的所有znode。	无
hoodie.index.hbase.table	仅在索引类型为HBASE时适用，必填选项。HBase表名称，用作索引。Hudi将row_key和[partition_path, fileID, commitTime]映射存储在表中。	无
hoodie.index.class	用户自定义索引的全路径名，索引类必须为HoodieIndex的子类，当指定该配置时，其会优先于hoodie.index.type配置。	""
hoodie.index.type	使用的索引类型，默认为布隆过滤器。可能的选项是[BLOOM | HBASE | GLOBAL_BLOOM | SIMPLE | GLOBAL_SIMPLE] 。 布隆过滤器消除了对外部系统的依赖，并存储在Parquet数据文件的页脚中。	BLOOM
hoodie.index.bloom.num_entries	存储在布隆过滤器中的条目数。 假设maxParquetFileSize为128MB，averageRecordSize为1024B，因此，一个文件中的记录总数约为130K。 默认值（60000）大约是此近似值的一半。 注意：将此值设置得太低，将产生很多误报，并且索引查找将必须扫描比其所需的更多的文件；如果将其设置得非常高，将线性增加每个数据文件的大小（每50000个条目大约4KB）。	60000
hoodie.index.bloom.fpp	根据条目数允许的错误率。 用于计算应为布隆过滤器分配多少位以及哈希函数的数量。通常将此值设置得很低（默认值：0.000000001），在磁盘空间上进行权衡以降低误报率。	0.000000001
hoodie.bloom.index.parallelism	索引查找的并行度，其中涉及Spark Shuffle。 默认情况下，根据输入的工作负载特征自动计算的。	0
hoodie.bloom.index.prune.by.ranges	为true时，从文件框定信息，可以加快索引查找的速度。 如果键具有单调递增的前缀，例如时间戳，则特别有用。	true
hoodie.bloom.index.use.caching	为true时，将通过减少用于计算并行度或受影响分区的IO来缓存输入的RDD以加快索引查找。	true
hoodie.bloom.index.use.treebased.filter	为true时，启用基于间隔树的文件过滤优化。与暴力模式相比，此模式可根据键范围加快文件过滤速度。	true
hoodie.bloom.index.bucketized.checking	为true时，启用了桶式布隆过滤。这减少了在基于排序的布隆索引查找中看到的偏差。	true
hoodie.bloom.index.keys.per.bucket	仅在启用BloomIndexBucketizedChecking并且索引类型为bloom的情况下适用。 此配置控制"存储桶"的大小，该大小可跟踪对单个文件进行的记录键检查的次数，并且是分配给执行布隆过滤器查找的每个分区的工作单位。 较高的值将分摊将布隆过滤器读取到内存的固定成本。	10000000
hoodie.bloom.index.update.partition.path	仅在索引类型为GLOBAL_BLOOM时适用。 为true时，当对一个已有记录执行包含分区路径的更新操作时，将会导致把新记录插入到新分区，而把原有记录从旧分区里删除。为false时，只对旧分区的原有记录进行更新。	true
hoodie.index.hbase.zkquorum	仅在索引类型为HBASE时适用，必填选项。要连接的HBase ZK Quorum URL。	无
hoodie.index.hbase.zkport	仅在索引类型为HBASE时适用，必填选项。要连接的HBase ZK Quorum端口。	无
hoodie.index.hbase.zknode.path	仅在索引类型为HBASE时适用，必填选项。这是根znode，它将包含HBase创建及使用的所有znode。	无
hoodie.index.hbase.table	仅在索引类型为HBASE时适用，必填选项。HBase表名称，用作索引。Hudi将row_key和[partition_path, fileID, commitTime]映射存储在表中。	无

存储信息配置

参数	描述	默认值
hoodie.parquet.max.file.size	Hudi写阶段生成的parquet文件的目标大小。对于DFS，这需要与基础文件系统块大小保持一致，以实现最佳性能。	120 * 1024 * 1024 byte
hoodie.parquet.block.size	parquet页面大小，页面是parquet文件中的读取单位，在一个块内，页面被分别压缩。	120 * 1024 * 1024 byte
hoodie.parquet.compression.ratio	当Hudi尝试调整新parquet文件的大小时，预期对parquet数据进行压缩的比例。 如果bulk_insert生成的文件小于预期大小，请增加此值。	0.1
hoodie.parquet.compression.codec	parquet压缩编解码方式名称，默认值为gzip。可能的选项是[gzip | snappy | uncompressed | lzo]	snappy
hoodie.logfile.max.size	LogFile的最大值。这是在将日志文件移到下一个版本之前允许的最大值。	1GB
hoodie.logfile.data.block.max.size	LogFile数据块的最大值。这是允许将单个数据块附加到日志文件的最大值。 这有助于确保附加到日志文件的数据被分解为可调整大小的块，以防止发生OOM错误。此大小应大于JVM内存。	256MB
hoodie.logfile.to.parquet.compression.ratio	随着记录从日志文件移动到parquet，预期会进行额外压缩的比例。 用于merge_on_read存储，以将插入内容发送到日志文件中并控制压缩parquet文件的大小。	0.35

Compaction 配置

参数	描述	默认值
hoodie.compact.inline	当设置为true时，紧接在插入或插入更新或批量插入的提交或增量提交操作之后由摄取本身触发压缩。	false
hoodie.compact.inline.max.delta.commits	触发内联压缩之前要保留的最大增量提交数。	5

参数	描述	默认值
hoodie.compaction.lazy.block.read	当CompactedLogScanner合并所有日志文件时，此配置有助于选择是否应延迟读取日志块。选择true以使用I/O密集型延迟块读取（低内存使用），或者为false来使用内存密集型立即块读取（高内存使用）。	true
hoodie.compaction.reverse.log.read	HoodieLogFormatReader会从pos=0到pos=file_length向前读取日志文件。如果此配置设置为true，则Reader会从pos=file_length到pos=0反向读取日志文件。	false
hoodie.compaction.strategy	用来决定在每次压缩运行期间选择要压缩的文件组的压缩策略。默认情况下，Hudi选择具有累积最多未合并数据的日志文件。	org.apache.hudi.table.action.compact.strategy.LogFileSizeBasedCompactionStrategy
hoodie.compaction.target.io	LogFileSizeBasedCompactionStrategy的压缩运行期间要花费的MB量。当压缩以内联模式运行时，此值有助于限制摄取延迟。	500 * 1024 MB
hoodie.compaction.daybased.target.partitions	由org.apache.hudi.io.compact.strategy.DayBasedCompactionStrategy使用，表示在压缩运行期间要压缩的最新分区数。	10
hoodie.parquet.small.file.limit	该值应小于maxFileSize，如果将其设置为0，会关闭此功能。由于批处理中分区中插入记录的数量众多，总会出现小文件。Hudi提供了一个选项，可以通过将对该分区中的插入作为对现有小文件的更新来解决小文件的问题。此处的大小是被视为"小文件大小"的最小文件大小。	104857600 byte
hoodie.copyonwrite.insert.split.size	插入写入并行度。为单个分区的总共插入次数。写出100MB的文件，至少1KB大小的记录，意味着每个文件有100K记录。默认值是超额配置为500K。 为了改善插入延迟，请对其进行调整以匹配单个文件中的记录数。将此值设置为较小的值将导致文件变小（尤其是当compactionSmallFileSize为0时）。	500000
hoodie.copyonwrite.insert.auto.split	Hudi是否应该基于最后24个提交的元数据动态计算insertSplitSize，默认关闭。	true
hoodie.copyonwrite.record.size.estimate	平均记录大小。如果指定，Hudi将使用它，并且不会基于最后24个提交的元数据动态地计算。 没有默认值设置。这对于计算插入并行度以及将插入打包到小文件中至关重要。	1024

Cleaning 配置

参数	描述	默认值
hoodie.clean.automatic	是否执行自动clean。	true
hoodie.clean.async	是否异步清理	false
hoodie.cleaner.policy	要使用的清理政策。Hudi将删除旧版本的parquet文件以回收空间。 任何引用此版本文件的查询和计算都将失败。建议确保数据保留的时间超过最大查询执行时间。	KEEP_LATEST_COMMITS
hoodie.cleaner.commits.retained	保留的提交数。因此，数据将保留为num_of_commits * time_between_commits（计划的），这也直接转化为逐步提取此数据集的数量。	10
hoodie.cleaner.parallelism	如果清理变慢，请增加此值。	200

Clustering 配置

参数	描述	默认值
hoodie.clustering.inline	是否同步执行clustering	false
hoodie.clustering.inline.max.commits	触发clustering的commit数	4
hoodie.clustering.plan.strategy.target.file.max.bytes	指定clustering后每个文件大小最大值	1024 * 1024 * 1024 byte
hoodie.clustering.plan.strategy.small.file.limit	小于该大小的文件会被clustering	300 * 1024 * 1024 byte
hoodie.clustering.plan.strategy.sort.columns	clustering用以排序的列	无
hoodie.layout.optimize.strategy	Clustering执行策略，可选linear、z-order、hilbert 三种排序方式	linear
hoodie.layout.optimize.enable	使用z-order、hilbert时需要开启	false
hoodie.clustering.plan.strategy.class	筛选FileGroup进行clustering的策略类，默认筛选小于hoodie.clustering.plan.strategy.small.file.limit阈值的文件	org.apache.hudi.client.clustering.plan.strategy.SparkSizeBasedClusteringPlanStrategy
hoodie.clustering.execution.strategy.class	执行clustering的策略类（RunClusteringStrategy的子类），用以定义群集计划的执行方式。 默认类按指定的列对计划中的文件组进行排序，同时满足配置的目标文件大小	org.apache.hudi.client.clustering.run.strategy.SparkSortAndSizeExecutionStrategy
hoodie.clustering.plan.strategy.max.num.groups	设置执行clustering时最多选择多少个FileGroup，该值越大并发度越大	30
hoodie.clustering.plan.strategy.max.bytes.per.group	设置执行clustering时每个FileGroup最多有多少数据参与clustering	2 * 1024 * 1024 * 1024 byte
hoodie.clustering.inline	是否同步执行clustering	false
hoodie.clustering.inline.max.commits	触发clustering的commit数	4

并发控制配置

参数	描述	默认值
hoodie.write.lock.provider	指定lock provider，不建议使用默认值，使用org.apache.hudi.hive.HiveMetastoreBasedLockProvider	org.apache.hudi.client.transaction.lock.ZookeeperBasedLockProvider

参数	描述	默认值
hoodie.write.lock.hivemetastore.database	Hive的database	无
hoodie.write.lock.hivemetastore.table	Hive的table name	无
hoodie.write.lock.client.num_retries	重试次数	10
hoodie.write.lock.client.wait_time_ms_between_retry	重试间隔	10000
hoodie.write.lock.conflict.resolution.strategy	lock provider类，必须是ConflictResolutionStrategy的子类	org.apache.hudi.client.transaction.SimpleConcurrentFileWritesConflictResolutionStrategy
hoodie.write.lock.zookeeper.base_path	存放ZNodes的路径，同一张表的并发写入需配置一致	无
hoodie.write.lock.zookeeper.lock_key	ZNode的名称，建议与Hudi表名相同	无
hoodie.write.lock.zookeeper.connection_timeout_ms	zk连接超时时间	15000
hoodie.write.lock.zookeeper.port	zk端口号	无
hoodie.write.lock.zookeeper.url	zk的url	无
hoodie.write.lock.zookeeper.session_timeout_ms	zk的session过期时间	60000
hoodie.write.lock.provider	指定lock provider，不建议使用默认值，使用org.apache.hudi.hive.HiveMetastoreBasedLockProvider	org.apache.hudi.client.transaction.lock.ZookeeperBasedLockProvider
hoodie.write.lock.hivemetastore.database	Hive的database	无
hoodie.write.lock.hivemetastore.table	Hive的table name	无

关键功能

该处的小文件合并，是一个广义的hudi数据表的重新组织优化，包括 MOR 表的行文件和基础base 文件的合并；基础base 文件的合并及数据重新排序；历史版本数据的清理等操作。hudi自身提供的上述相关操作同步和异步处理，但是在生产环境中会影响业务数据任务的正常运行，因此正常情况下使用异步的方式对hudi 数据表文件进行整理优化，提高数据访问性能。上述文件的优化合并及清理功能，理论上没有先后顺序。实际使用上建议的数据文件优化处理流程如下：

Compaction

该功能只对MOR 表有效（如果hudi 表类型是COW 表，该步骤省略），该功能是将mor 表的base file 和 log file 进行整理合并成一个新的base file，以提高查询执行效率，其异步执行参数和命令如下：

```
spark-submit  --class org.apache.hudi.utilities.HoodieCompactor \
/xxx/hudi-utilities-bundle_2.12-*.jar \
--spark-memory '4g' \
--mode 'scheduleAndExecute' \
--base-path "/user/hive/warehouse/hudi.db/small_file_hudi_mor" \
--table-name "small_file_hudi_mor" \
--hoodie-conf "hoodie.compact.schedule.inline=false" \
--hoodie-conf "hoodie.compact.inline=false" \
--hoodie-conf "hoodie.compact.inline.max.delta.commits=1"
```

核心参数说明：

配置项	设定值	说明
jar 路径及版本	/xxx/hudi-utilities-bundle_2.12-*.jar	jar 包路径信息，该路径为测试路径，实际中根据实际部署的路径进行设定。
--spark-memory	4g	spark 计算资源内存使用，根据实际数据情况调整计算资源量。
--mode	scheduleAndExecute	Compaction 执行模式设置参数，设置为排期并执行，在一个任务里面完成相关的合并操作。

配置项	设定值	说明
--base-path	/user/hive/warehouse/hudi.db/small_file_hudi_mor	进行 Compaction 的数据路径信息，设置为对应表的路径，进行整个表的数据合并。
--table-name	small_file_hudi_mor	要Compaction 的数据表名。
hoodie.compact.schedule.inline	false	hudi 配置项，是否进行异步调度排期。这里使用异步模式。设置为false
hoodie.compact.inline	false	hudi 配置项，是否内联同步合并。这里使用异步模式。设置为false
hoodie.compact.inline.max.delta.commits	1	触发Compaction 之前要保留的最大增量提交数，提交多少次commits 后，才触发合并。

其他参数详见：https://hudi.apache.org/docs/configurations#Compaction-Configs

Clustering

Clustering 是一种通用的数据布局优化手段，Spark SQL/Hive 中的 cluster by 和 Cassandra 中的 clustering key 都是 Clustering 思想的具体实现，只是 Hudi 的 Clustering 除了这一标准功能外还多了一项合并小文件的工作。该处主要进行小文件合并处理，如果业务上有需要，可以添加数据重排序处理，其异步执行参数和命令如下：

```
spark-submit --jars 'hudi-utilities-bundle_2.12-*.jar' \
--class 'org.apache.hudi.utilities.HoodieClusteringJob' \
/xxx/hudi-utilities-bundle_2.12-*.jar \
--spark-memory '4g' \
--mode 'scheduleAndExecute' \
--base-path "/user/hive/warehouse/hudi.db/small_file_hudi_cow/" \
--table-name "small_file_hudi_cow" \
--hoodie-conf "hoodie.clustering.async.enabled=true" \
--hoodie-conf "hoodie.clustering.async.max.commits=1" \
--hoodie-conf "hoodie.clustering.plan.strategy.small.file.limit=536870912" \
--hoodie-conf "hoodie.clustering.plan.strategy.target.file.max.bytes=1073741824"
```

核心参数说明：

配置项	设定值	说明
jar 路径及版本	/xxx/hudi-utilities-bundle_2.12-*.jar	jar 包路径信息，该路径为测试路径，实际中根据实际部署的路径进行设定。
--spark-memory	4g	spark 计算资源内存使用，根据实际数据情况调整计算资源量。
--mode	'scheduleAndExecute'	Clustering 执行模式设置参数，设置为排期并执行，在一个任务里面完成相关的小文件合并操作。
--base-path	/user/hive/warehouse/hudi.db/small_file_hudi_mor	进行 Clustering 的数据路径信息，设置为对应表的路径，进行整个表的数据小文件合并。
--table-name	small_file_hudi_mor	要Clustering 的数据表名。
hoodie.clustering.async.enabled	true	hudi 配置项，是否允许异步小文件合并。这里使用异步模式。设置为true
hoodie.clustering.async.max.commits	1	hudi 配置项，异步合并小文件执行的最大 commit 数。
hoodie.clustering.plan.strategy.small.file.limit	536870912	hudi 配置项，小于该值时，被认为小文件。然后对这些小文件进行合并。
hoodie.clustering.plan.strategy.target.file.max.bytes	1073741824	合并后的文件最大大小，
hoodie.clustering.plan.strategy.sort.columns	数据列	排序列，根据业务查询需要，可以在合并过程中进行数据排序。

其他参数详见：https://hudi.apache.org/docs/configurations#Clustering-Configs

Cleaning

对于每次更新hudi 表数据，就会生成一个新版本的数据文件用于保存更新后的记录(COPY_ON_WRITE) 或将这些增量更新写入日志文件以避免重写更新版本的数据文件 (MERGE_ON_READ)。在这种情况下，根据更新频率，文件版本数可能会无限增长，但大部分业务如果不需要保留无限的历史记录。因此就需要通过清理服务，来删除历史不需要的版本数据，其异步执行参数和命令如下：

```
spark-submit --class org.apache.hudi.utilities.HoodieCleaner /xxx/hudi-utilities-bundle_2.12-*.jar \
  --target-base-path "/user/hive/warehouse/hudi.db/small_file_hudi_cow/" \
  --hoodie-conf "hoodie.cleaner.policy=KEEP_LATEST_FILE_VERSIONS" \
  --hoodie-conf "hoodie.cleaner.fileversions.retained=1" \
  --hoodie-conf "hoodie.cleaner.parallelism=200" \
  --hoodie-conf "hoodie.clean.async=true" \
  --hoodie-conf "hoodie.clean.automatic=false"
```

核心参数说明：

配置项	设定值	说明
jar 路径及版本	/xxx/hudi-utilities-bundle_2.12-*.jar	jar 包路径信息，该路径为测试路径，实际中根据实际部署的路径进行设定。
hoodie.cleaner.policy	KEEP_LATEST_FILE_VERSIONS	hudi 配置项，清理策略。设置为保留N个版本数据。
hoodie.cleaner.fileversions.retained	1	hudi 配置项，保留多少个版本数据，根据业务情况进行设置。
hoodie.cleaner.parallelism	200	hudi 配置项，清理执行的并行度。
hoodie.clean.async	true	hudi 配置项，是否异步清理。设置为true。
hoodie.clean.automatic	false	hudi 配置项，是否自动内联清理，因为异步清理，设置为false。

其他参数详见：https://hudi.apache.org/docs/configurations#Clean-Configs

开发规范

Hudi表模型设计规范

基本规则

- Hudi表必须设置合理的主键。Hudi表提供了数据更新和幂等写入能力，该能力要求Hudi表必须设置主键，主键设置不合理会导致数据重复。主键可以为单一主键也可以为复合主键，两种主键类型均要求主键不能有null值和空值，可以参考以下示例设置主键：
  SparkSQL：

```
-- 通过primaryKey指定主键，如果是复合主键需要用逗号分隔。
create table hudi_table (
id1 int,
id2 int,
name string,
price double)
using hudi
options (
primaryKey = 'id1,id2',
preCombineField = 'price'
);
```

SparkDatasource：

```
--通过hoodie.datasource.write.recordkey.field指定主键。
df.write.format("hudi")
.option("hoodie.datasource.write.table.type", COPY_ON_WRITE)
.option("hoodie.datasource.write.precombine.field", "price")
.option("hoodie.datasource.write.recordkey.field", "id1,id2").
```

FlinkSQL：

```
--通过hoodie.datasource.write.recordkey.field指定主键。
create table hudi_table(
id1 int,
id2 int
,name string,
price double)
```

```
partitioned by (name)
with ('connector' = 'hudi','hoodie.datasource.write.recordkey.field' = 'id1,id2',
'write.precombine.field' = 'price'
)
```

Hudi表必须配置precombine字段：
在数据同步过程中不可避免会出现数据重复写入、数据乱序问题，例如：异常数据恢复、写入程序异常重启等场景。通过设置合理precombine字段值可以保证数据的准确性，老数据不会覆盖新数据，也就是幂等写入能力。该字段可用选择的类型包括：业务表中更新时间戳、数据库的提交时间戳等。precombine字段不能有null值和空值，可以参考以下示例设置precombine字段：
SparkSQL:

```
--通过preCombineField指定precombine字段。
create table hudi_table (
id1 int,
id2 int,
name string,
price double)
using hudi
options (primaryKey = 'id1,id2',
preCombineField = 'price'
);
```

SparkDatasource:

```
--通过hoodie.datasource.write.precombine.field指定precombine字段。
df.write.format("hudi")
.option("hoodie.datasource.write.table.type", COPY_ON_WRITE)
.option("hoodie.datasource.write.precombine.field", "price")
.option("hoodie.datasource.write.recordkey.field", "id1,id2")
```

Flink:

```
--通过write.precombine.field指定precombine字段。
create table hudi_table(
id1 int,
id2 int,
name string,
price double) partitioned by (name)
with ('connector' = 'hudi',
'hoodie.datasource.write.recordkey.field' = 'id1,id2',
'write.precombine.field' = 'price')
```

表类型选择

在建表阶段依据实际应用开发需求选择合适的表类型：MOR或者COW。
流式计算采用MOR表：
流式计算为低时延的实时计算，需要高性能的流式读写能力，在Hudi表中存在的MOR和COW两种模型中，MOR表的流式读写性能相对较好，因此在流式计算场景下采用MOR表模型。关于MOR表在读写性能的对比关系如下：

对比维度	MOR表	COW表
流式写	高	低
流式读	高	低
批量写	高	低
批量读	低	高

- 实时入湖，表模型采用MOR表。
  实时入湖一般的性能要求都在分钟内或者分钟级，结合Hudi两种表模型的对比，因此在实时入湖场景中需要选择MOR表模型。
- Hudi表名以及列名采用小写字母。
  多引擎读写同一张Hudi表时，为了规避引擎之间大小写的支持不同，统一采用小写字母。

建议

- Spark批处理场景，对写入时延要求不高的场景，采用COW表。COW表模型中，写入数据存在写放大问题，因此写入速度较慢；但COW具有非常好的读取性能力。而且批量计算对写入时延不是很敏感，因此可以采用COW表。
- Hudi表的写任务要开启Hive元数据同步功能。SparkSQL天然与Hive集成，无需考虑元数据问题。该条建议针对的是通过Spark Datasource API或者Flink写Hudi表的场景，通过这两种方式写Hudi时需要增加向Hive同步元数据的配置项；该配置的目的是将Hudi表的元数据统一托管到Hive元数据服务中，为后续的跨引擎操作数据以及数据管理提供便利。
- 针对Spark和Trino要求查询Hudi表数据一致性要求高的场景使用COW表类型。
  此外针对COW和MOR还有如下一些选择建议
  COW
- 写少读多场景。
- 写入频率低或者不需要实时写入的场景。
- 只有插入，无update操作的场景。
- 针对读取时延要求较高的场景。

MOR

- 写多读少的场景。
- 高频的写入和更新，并且对写入性能要求较高的场景。
- 需要流式写入的场景。

数据类型设计规范

- 对于整数，小于等于9位的使用int类型，超过9位的使用long类型。
- 小数均采用decimal(precision,scale)类型。
- 字符类型字段，统一使用string类型。
- 数值类型中，建议用0代表值为NULL的场景。

写入模式规范

hudi提供三种写入模式：bulk_insert、insert、upsert。

- 可容忍主键重复的记录，并且数据量较多的场景，进行初始化时建议使用bulk_insert模式。
- 可容忍主键重复的记录，日志或者流式追加写的场景。使用insert模式时，为防止造成小文件过多问题，需要设置clustering。
- MOR表的数据初始化、对已有表数据修正和更新、不容忍主键重复的记录，数据实时更新和插入的场景建议使用upsert模式。

Hudi表索引设计规范

规则

- 禁止修改表索引类型。Hudi表的索引会决定数据存储方式，随意修改索引类型会导致表中已有的存量数据与新增数据之间出现数据重复和数据准确性问题。常见的索引类型如下：
  - 布隆索引：Spark引擎独有索引，采用bloomfilter机制，将布隆索引内容写入到Parquet文件的footer中。
  - Bucket索引：在写入数据过程中，通过主键进行Hash计算，将数据进行分桶写入；该索引写入速度最快，但是需要合理配置分桶数目；Flink、Spark均支持该索引写入。
  - 状态索引：Flink引擎独有索引，是将行记录的存储位置记录到状态后端的一种索引形式，在作业冷启动过程中会遍历所有数据存储文件生成索引信息。
- 用Flink状态索引，Flink写入后，不支持Spark继续写入。Flink在写Hudi的MOR表只会生成log文件，后续通过compaction操作，将log文件转为parquet文件。Spark在更新Hudi表时严重依赖parquet文件是否存在，如果

当前Hudi表写的是log文件，采用Spark写入就会导致重复数据的产生。在批量初始化阶段 ，先采用Spark批量写入Hudi表，再用Flink基于Flink状态索引写入不会有问题，原因是Flink冷启动的时候会遍历所有的数据文件生成状态索引。

- 实时入湖场景中，Spark引擎采用Bucket索引，Flink引擎可以用Bucket索引或者状态索引。实时入湖都是需要分钟内或者分钟级的高性能入湖，索引的选择会影响到写Hudi表的性能。在性能方面各个索引的区别如下：
  - Bucket索引优点：写入过程中对主键进行hash分桶写入，性能比较高，不受表的数据量限制。Flink和Spark引擎都支持，Flink和Spark引擎可以实现交叉混写同一张表。缺点：Bucket个数不能动态调整，数据量波动和整表数据量持续上涨会导致单个Bucket数据量过大出现大数据文件。需要结合分区表来进行平衡改善。
  - Flink状态索引优点：主键的索引信息存在状态后端，数据更新只需要点查状态后端即可，速度较快；同时生成的数据文件大小稳定，不会产生小文件、超大文件问题。缺点：该索引为Flink特有索引。在表的总数据行数达到数亿级别，需要优化状态后端参数来保持写入的性能。使用该索引无法支持Flink和Spark交叉混写。
- 对于数据总量持续上涨的表，采用Bucket索引时，须使用时间分区，分区键采用数据创建时间。参照Flink状态索引的特点，Hudi表超过一定数据量后，Flink作业状态后端压力很大，需要优化状态后端参数才能维持性能；同时由于Flink冷启动的时候需要遍历全表数据，大数据量也会导致Flink作业启动缓慢。因此基于简化使用的角度，针对大数据量的表，可以通过采用Bucket索引来避免状态后端的复杂调优。如果Bucket索引+分区表的模式无法平衡Bucket桶过大的问题，还是可以继续采用Flink状态索引，按照规范去优化对应的配置参数即可。

建议

- 基于Flink的流式写入的表，在数据量超过2亿条记录，采用Bucket索引，2亿以内可以采用Flink状态索引。参照Flink状态索引的特点，Hudi表超过一定数据量后，Flink作业状态后端压力很大，需要优化状态后端参数才能维持性能；同时由于Flink冷启动的时候需要遍历全表数据，大数据量也会导致Flink作业启动缓慢。因此基于简化使用的角度，针对大数据量的表，可以通过采用Bucket索引来避免状态后端的复杂调优。如果Bucket索引+分区表的模式无法平衡Bucket。桶过大的问题，还是可以继续采用Flink状态索引，按照规范去优化对应的配置参数即可。
- 基于Bucket索引的表，按照单个Bucket 2GB数据量进行设计。
- 2GB的数据存储成列存Parquet文件后，大概的数据文件大小是150MB ~ 256MB左右。不同业务数据会有出入。而HDFS单个数据块一般会是128MB，这样可以有效地利用存储空间。
- 数据读写占用的内存空间都是原始数据大小（包括空值也是会占用内存的），2GB在大数据计算过程中，处于单task读写可接受范围之内。
- 如果是单个Bucket的数据量超过了该值范围，可能会有什么影响？
- 读写任务可能会出现OOM的问题，解决方法就是提升单个task的内存占比。
- 读写性能下降，因为单个task的处理的数据量变大，导致处理耗时变大。
  为什么建议是2GB？
  - 2GB的数据存储成列存Parquet文件后，大概的数据文件大小是150MB ~ 256MB左右。不同业务数据会有出入。而HDFS单个数据块一般会是128MB，这样可以有效地利用存储空间。

- 数据读写占用的内存空间都是原始数据大小（包括空值也是会占用内存的），2GB在大数据计算过程中，处于单task读写可接受范围之内。
  如果是单个Bucket的数据量超过了该值范围，可能会有什么影响？
- 读写任务可能会出现OOM的问题，解决方法就是提升单个task的内存占比。
- 读写性能下降，因为单个task的处理的数据量变大，导致处理耗时变大。

Hudi表分区设计规范

规则

分区键不可以被更新。
Hudi具有主键唯一性机制，但在分区表的场景下通常只能保证分区内主键唯一，因此如果分区键的值发生变更后，会导致相同主键的行记录出现多条的情况。在以日期分区的场景，可采用数据的创建时间为分区字段，切记不要采用数据更新时间做分区。

> 说明：
>
> 当指定Hudi的索引类型为Global索引类型时，Hudi支持跨分区进行数据更新，但Global索引性能较差一般不建议使用。

建议

- 事实表采用日期分区表，维度表采用非分区或者大颗粒度的日期分区。
  - 事实表：数据总量大，增量大，数据读取多以日期做切分，读取一定时间段的数据。
  - 维度表：总量相对小，增量小，多以更新操作为主，数据读取会是全表读取，或者按照对应业务ID过滤。
- 基于以上考虑，维度表采用天分区会导致文件数过多，而且是全表读取，会导致所需要的文件读取Task过多，采用大颗粒度的日期分区，例如年分区，可以有效降低分区个数和文件数量；对于增量不是很大的维度表，也可以采用非分区表。如果维度表的总数据量很大或者增量也很大，可以考虑采用某个业务ID进行分区，在大部分数据处理逻辑中针对大维度表，会有一定的业务条件进行过滤来提升处理性能，这类表要结合一定的业务场景来进行优化，无法从单纯的日期分区进行优化。事实表读取方式都会按照时间段切分，近一年、近一个月或者近一天，读取的文件数相对稳定可控，所以事实表优先考虑日期分区表。
- 分区采用日期字段，分区表粒度，要基于数据更新范围确定，不要过大也不要过小。分区粒度可以采用年、月、日，分区粒度的目标是减少同时写入的文件桶数，尤其是在有数据量更新，且更新数据有一定时间范围规律的，比如：近一个月的数据更新占比最大，可以按照月份创建分区；近一天内的数据更新占比大，可以按照天进行分区。采用Bucket索引，写入是通过主键Hash打散的，数据会均匀的写入到分区下每个桶。因为各个分

区的数据量是会有波动的，分区下桶的个数设计一般会按照最大分区数据量计算，这样会出现越细粒度的分区，桶的个数会冗余越多。例如：采用天级分区，平均的日增数据量是3GB，最多一天的日志是8GB，这个会采用Bucket桶数= 8GB/2GB = 4 来创建表；每天的更新数据占比较高，且主要分散到近一个月。这样会导致结果是，每天的数据会写入到全月的Bucket桶中，那就是4*30 = 120个桶。如果采用月分区，分区桶的个数= 3GB * 30 /2GB = 45个桶 ，这样写入的数据桶数减少到了45个桶。在有限的计算资源下，写入的桶数越少，性能越高。

- 分区层级不宜过多，最好不要超过3层。
- 建议hudi表与hive采用一致的分区风格，hoodie.datasource.write.hive_style_partitioning=true。
- 实时表通过记录创建日期分区，维度表创建为非分区，或者采用分区粒度较低的字段进行分区。

Hudi数据表Compaction规范

MOR表更新数据以行存log的形式写入，log读取时需要按主键合并，并且是行存的，导致log读取效率比parquet低很多。为了解决log读取的性能问题，Hudi通过compaction将log压缩成parquet文件，大幅提升读取性能。

规则

- 有数据持续写入的表，24小时内至少执行一次compaction。对于MOR表，不管是流式写入还是批量写入，需要保证每天至少完成1次Compaction操作。如果长时间不做compaction，Hudi表的log将会越来越大，这必将会出现以下问题：
  - Hudi表读取很慢，且需要很大的资源。 这是由于读MOR表涉及到log合并，大log合并需要消耗大量的资源并且速度很慢。
  - 长时间进行一次Compaction需要耗费很多资源才能完成，且容易出现OOM。
  - 阻塞Clean，如果没有Compaction操作来产生新版本的Parquet文件，那旧版本的文件就不能被Clean清理，增加存储压力。
- CPU与内存比例为1:~~41:8。Compaction作业是将存量的parquet文件内的数据与新增的log中的数据进行合并，需要消耗较高的内存资源，按照之前的表设计规范以及实际流量的波动结合考虑，建议Compaction作业CPU与内存的比例按照1:4~~1:8配置，保证Compaction作业稳定运行。当Compaction出现OOM问题，可以通过调大内存占比解决。

建议

- 通过增加并发数提升Compaction性能。CPU和内存比例配置合理会保证Compaction作业是稳定的，实现单个Compaction task的稳定运行。但是Compaction整体的运行时长取决于本次Compaction处理文件数以及分配的cpu核数（并发能力），因此可以通过增加Compaction作业的CPU核的个数来提升Compaction性能（注意增加cpu也要保证CPU与内存的比例）。

- Hudi表采用异步Compaction。为了保证流式入库作业的稳定运行，就需要保证流式作业不在实时入库的过程中做其它任务，比如Flink写Hudi的同时会做Compaction。这看似是一个不错的方案，即完成了入库又完成Compaction。但是Compaction操作是非常消耗内存和IO的，它会给流式入库作业带来以下影响：
  - 增加端到端时延：Compaction会放大写入时延，因为Compaction比入库更耗时。
  - 作业不稳定：Compaction会给入库作业带来更多的不稳定性，Compaction OOM将会导致整个作业直接失败。
- 建议2~4小时进行一次compaction。Compaction是MOR表非常重要且必须执行的维护手段，对于实时任务来说，要求Compaction执行合并的过程必须和实时任务解耦，通过周期调度Spark任务来完成异步Compaction，这个方案的关键之处在于如何合理的设置这个周期，周期如果太短意味着Spark任务可能会空跑，周期如果太长可能会积压太多的Compaction Plan没有去执行而导致Spark任务耗时长并且也会导致下游的读作业时延高。对此场景，在这里给出以下建议：按照集群资源使用情况，可以每2小时或每4个小时去调度执行一次异步Compaction作业，这是一个基本的维护MOR表的方案。
- 采用Spark异步执行Compaction，不采用Flink进行Compaction。Flink写hudi建议的方案是Flink只负责写数据和生成Compaction计划，由单独的Spark作业异步执行compaction、clean和archive。Compaction计划的生成是轻量级的对Flink写入作业影响可以忽略。上述方案落地的具体步骤参考如下：
  - Flink只负责写数据和生成Compaction计划Flink流任务建表语句中添加如下参数，控制Flink任务写Hudi时只会生成Compaction plan。

```
'compaction.async.enabled' = 'false'      -- 关闭Flink 执行Compaction任务
'compaction.schedule.enabled' = 'true'    -- 开启Compaction计划生成
'compaction.delta_commits' = '5'          -- MOR表默认5次checkpoint尝试生成compaction plan，该参数需要根据具体业务调整
'clean.async.enabled' = 'false'           -- 关闭Clean操作
'hoodie.archive.automatic' = 'false'      -- 关闭Archive操作
```

- Spark离线完成Compaction计划的执行，以及Clean和Archive操作。
  在调度平台（可以使用华为的DataArts）运行一个定时调度的离线任务来让Spark完成Hudi表的Compaction计划执行以及Clean和Archive操作。

```
set hoodie.archive.automatic = false;
set hoodie.clean.automatic = false;
set hoodie.compact.inline = true;
set hoodie.run.compact.only.inline=true;
set hoodie.cleaner.commits.retained = 500;  -- clean保留timeline上最新的500个deltacommit对应的数据文件，之前的deltacommit所对应的旧版本文件会被清理。该值需要大于compaction.delta_commits设置的值，需要根据具体业务调整。
set hoodie.keep.max.commits = 700;  -- timeline最多保留700个deltacommit
set hoodie.keep.min.commits = 501;  -- timeline最少保留500个deltacommit。该值需要大于hoodie.cleaner.commits.retained设置的值，需要根据具体业务调整。
run compaction on <database name>. <table name>;  -- 执行Compaction计划run clean on <database name>. <table name>;       -- 执行Clean操作
```

run archivelog on <database name>.<table name>; -- 执行Archive操作

- 异步Compaction可以将多个表串行到一个作业，资源配置相近的表放到一组，该组作业的资源配置为最大消耗资源的表所需的资源。
  对于在Hudi表采用异步Compaction和采用Spark异步执行Compaction，这里给出以下开发建议：
  - 不需要对每张Hudi表都开发异步Compaction任务，这样会导致作业开发成本高，集群作业爆炸，集群资源不能有效的利用和释放。
  - 异步Compaction任务可以通过执行SparkSQL来完成，多个Hudi表的Compaction、Clean和Archive可以放在同一个任务来执行，比如对table1和table2用同一个任务来执行异步维护操作：

```
set hoodie.clean.async = true;
set hoodie.clean.automatic = false;
set hoodie.compact.inline = true;
set hoodie.run.compact.only.inline=true;
set hoodie.cleaner.commits.retained = 500;
set hoodie.keep.min.commits = 501;
set hoodie.keep.max.commits = 700;
run compaction on <database name>. <table1>;
run clean on <database name>. <table1>;
run archivelog on <database name>.<table1>;
run compaction on <database name>.<table2>;
run clean on <database name>.<table2>;
run archivelog on <database name>.<table2>;
```

Hudi数据表Clean规范

Clean也是Hudi表的维护操作之一，该操作对于MOR表和COW表都需要执行。Clean操作的目的是为了清理旧版本文件（Hudi不再使用的数据文件），这不但可以节省Hudi表List过程的时间，也可以缓解存储压力。

规则

Hudi表必须执行Clean。
对于Hudi的MOR、COW表，都需要开启Clean。

- Hudi表在写入数据时会自动判断是否需要执行Clean，因为Clean的开关默认打开(hoodie.clean.automatic默认为true)。
- Clean操作并不是每次写数据时都会触发，至少需要满足两个条件：

1. Hudi表中需要有旧版本的文件。对于COW表来说，只要保证数据被更新过就一定存在旧版本的文件。对于

MOR表来说，要保证数据被更新过并且做过Compaction才能有旧版本的文件。

2. Hudi表满足hoodie.cleaner.commits.retained设置的阈值。如果是Flink写hudi，则至少提交的checkpoint要超过这个阈值；如果是批写Hudi，则批写次数要超过这个阈值。

建议

- MOR表下游采用批量读模式，采用clean的版本数为compaction版本数+1。MOR表一定要保证Compaction Plan能够被成功执行，Compaction Plan只是记录了Hudi表中哪些Log文件要和哪些Parquet文件合并，所以最重要的地方在于保证Compaction Plan在被执行的时候它需要合并的文件都存在。而Hudi表中只有Clean操作可以清理文件，所以建议Clean的触发阈值（hoodie.cleaner.commits.retained的值）至少要大于Compaction的触发阈值（对于Flink任务来说就是compaction.delta_commits的值）。
- MOR表下游采用流式计算，历史版本保留小时级。如果MOR表的下游是流式计算，例如Flink流读，可以按照业务需要保留小时级的历史版本，这样的话近几个小时之内的增量数据可以通过log文件读出，如果保留时长过短，下游flink作业在重启或者异常中断阻塞的情况下，上游增量数据已经Clean掉了，flink需要从parquet文件读增量数据，性能会有下降；如果保留时间过长，会导致log里面的历史数据冗余存储。具体可以按照下面的计算公式来保留2个小时的历史版本数据：版本数设置为3600*2/版本interval时间，版本interval时间来自于flink作业的checkpoint周期，或者上游批量写入的周期。
- COW表如果业务没有历史版本数据保留的特殊要求，保留版本数设置为1。COW表的每个版本都是表的全量数据，保留几个版本就会冗余多少个版本。因此如果业务无历史数据回溯的需求，保留版本数设置为1，也就是保留当前最新版本。
- clean作业每天至少执行一次，可以2~4小时执行一次。Hudi的MOR表和COW表都需要保证每天至少1次Clean，MOR表的Clean可以参考2.2.1.6小节和Compaction放在一起异步去执行。COW的Clean可以在写数据时自动判断是否执行。

Hudi数据表Archive规范

Archive（归档）是为了减轻Hudi读写元数据的压力，所有的元数据都存放在这个路径：Hudi表根目录/.hoodie目录，如果.hoodie目录下的文件数量超过10000就会发现Hudi表有非常明显的读写时延。

规则

Hudi表必须执行Archive。
对于Hudi的MOR类型和COW类型的表，都需要开启Archive。

- Hudi表在写入数据时会自动判断是否需要执行Archive，因为Archive的开关默认打开(hoodie.archive.automatic默认为true)。

- Archive操作并不是每次写数据时都会触发，至少需要满足以下两个条件：

1. Hudi表满足hoodie.keep.max.commits设置的阈值。如果是Flink写hudi至少提交的checkpoint要超过这个阈值；如果是Spark写hudi，写Hudi的次数要超过这个阈值。
2. Hudi表做过Clean，如果没有做过Clean就不会执行Archive。

建议

Archive作业每天至少执行一次，可以2~4小时执行一次。
Hudi的MOR表和COW表都需要保证每天至少1次Archive，MOR表的Archive可以参考2.2.1.6小节和Compaction放在一起异步去执行。COW的Archive可以在写数据时自动判断是否执行。

Spark on Hudi表数据维护规范

SparkSQL建表参数规范

规则

- 建表必须指定primaryKey和preCombineField。Hudi表提供了数据更新的能力和幂等写入的能力，该能力要求数据记录必须设置主键用来识别重复数据和更新操作。不指定主键会导致表丢失数据更新能力，不指定preCombineField会导致主键重复。

参数名称	参数描述	输入值	说明
primaryKey	hudi主键	按需	必须指定，可以是复合主键但是必须全局唯一。
preCombineField	预合并键，相同主键的多条数据按该字段进行合并	按需	必须指定，相同主键的数据会按该字段合并，不能指定多个字段。

- 禁止建表时将hoodie.datasource.hive_sync.enable指定为false。指定为false将导致新写入的分区无法同步到Hive Metastore中。由于缺失新写入的分区信息，查询引擎读取该时会丢数。
- 禁止指定Hudi的索引类型为INMEMORY类型。该索引仅是为了测试使用。生产环境上使用该索引将导致数据重复。

建表示例

```
create table data_partition(id int, comb int, col0 int,yy int, mm int, dd int)
 using hudi                          --指定hudi 数据源
 partitioned by(yy,mm,dd)            --指定分区， 支持多级分区
 location '/opt/log/data_partition'   --指定路径，如果不指定建表在hive warehouse里
 options(type='mor',                   --表类型 mor 或者
 cowprimaryKey='id',                 --主键，可以是复合主键但是必须全局唯一
 preCombineField='comb'               --预合并字段，相同主键的数据会按该字段合并，当前不能指定多个字段
 )
```

Spark增量读取Hudi参数规范

规则

增量查询之前必须指定当前表的查询为增量查询模式，并且查询后重写设置表的查询模式。

如果增量查询完，不重新将表查询模式设置回去，将影响后续的实时查询。

示例

```
set hoodie.tableName.consume.mode=INCREMENTAL;--必须设置当前表读取为增量读取模式。
set hoodie.tableName.consume.start.timestamp=20201227153030;--指定初始增量拉取commit。
set hoodie.tableName.consume.end.timestamp=20210308212318;  --指定增量拉取结束commit，如果不指定的话采用最新的commit。
select * from tableName where `_hoodie_commit_time`>'20201227153030' and `_hoodie_commit_time`<='20210308212318'; --结果必须根据start.timestamp和end.timestamp进行过滤，如果没有指定end.timestamp，则只需要根据start.timestamp进行过滤。
set hoodie.tableName.consume.mode=SNAPSHOT;  --使用完增量模式，必须把查询模式重新设置回来。
```

Spark异步任务执行表compaction参数设置规范

- 写作业未停止情况下，禁止手动执行run schedule命令生成compaction计划。
  错误示例：

```
run schedule on dsrTable
```

如果还有别的任务在写这张表，执行该操作会导致数据丢失。

- 执行run compaction命令时，禁止将hoodie.run.compact.only.inline设置成false，该值需要设置成true。
  错误示例：

```
set hoodie.run.compact.only.inline=false;
run compaction on dsrTable;
```

如果还有别的任务在写这张表，执行上述操作会导致数据丢失。

正确示例：异步Compaction

```
set hoodie.compact.inline = true;
set hoodie.run.compact.only.inline=true;
run compaction on dsrTable;
```

Spark on Hudi表数据维护规范

禁止通过Alter命令修改表关键属性信息：type/primaryKey/preCombineField/hoodie.index.type

错误示例，执行如下语句修改表关键属性：

```
alter table dsrTable set tblproperties('type'='xx');
alter table dsrTable set tblproperties('primaryKey'='xx');
alter table dsrTable set tblproperties('preCombineField'='xx');
alter table dsrTable set tblproperties('hoodie.index.type'='xx');
```

Hive/trino等引擎可以直接修改表属性，但是这种修改会导致整个Hudi表出现数据重复，甚至数据损坏；因此禁止修改上述属性。

Spark并发写Hudi建议

- 涉及到并发场景，推荐采用分区间并发写的方式：即不同的写入任务写不同的分区。
- 分区并发参数控制：
    - SQL方式：

```
set hoodie.support.partition.lock=true;
```

- DataSource API方式：

```
df.write.format("hudi")
  .options(xxx)
```

```
.option("hoodie.support.partition.lock", "true")
.mode(xxx)
.save("/tmp/tablePath")
```

> 说明：
>
> 所有参与分区间并发写入的任务，都必须配置上述参数。

- 不建议同分区内并发写，这种并发写入需要开启Hudi OCC方式并发写入，必须严格遵守并发参数配置，否则会出现表数据损坏的问题。
  并发OCC参数控制：
  - SQL方式：

```
--开启OCC。
set hoodie.write.concurrency.mode=optimistic_concurrency_control;
set hoodie.cleaner.policy.failed.writes=LAZY;--开启并发ZooKeeper锁。
set hoodie.write.lock.provider=org.apache.hudi.client.transaction.lock.ZookeeperBasedLockProvider;
--设置使用ZooKeeper锁。
set hoodie.write.lock.zookeeper.url=<zookeeper_url>;  --设置使用ZooKeeper地址。
set hoodie.write.lock.zookeeper.port=<zookeeper_port>; --设置使用ZooKeeper端口。
set hoodie.write.lock.zookeeper.lock_key=<table_name>;  --设置锁名称。
set hoodie.write.lock.zookeeper.base_path=<table_path>; --设置zk锁路径。
```

- DataSource API方式：

```
df.write
  .format("hudi")
  .options(xxx)
  .option("hoodie.write.concurrency.mode", "optimistic_concurrency_control")
  .option("hoodie.cleaner.policy.failed.writes", "LAZY")
  .option("hoodie.write.lock.zookeeper.url", "zookeeper_url")
  .option("hoodie.write.lock.zookeeper.port", "zookeeper_port")
  .option("hoodie.write.lock.zookeeper.lock_key", "table_name")
  .option("hoodie.write.lock.zookeeper.base_path", "table_path")
  .mode(xxx)
  .save("/tmp/tablePath")
```

> 说明：
>
> 1. 所有参与并发写入的任务，都必须配置上述参数。OCC不会保证所有参与并发写入的任务都执行成功;当出现多个写任务更新同一个文件时，只有一个任务可以成功，其余失败。
> 2. 并发场景下，需要设置cleaner policy为Lazy，因此无法自动清理垃圾文件。

Spark读写Hudi资源配置建议

- Spark读写Hudi任务资源配置规则，内存和CPU核心的比例2:1，堆外内存和CPU核心比例0.5:1；即一个核心，需要2G堆内存，0.5G堆外内存。

  > 说明：
  >
  > Spark初始化入库场景，由于处理的数据量比较大，上述资源配比需要调整，内存和Core的比例推荐4:1，堆外内存和Core的比例1:1。

示例：

```
spark-submit
--master yarn-cluster
--executor-cores  2                    --核心
--executor-memory 4g                   --堆内存
--conf spark.executor.memoryOverhead=1024    --堆外内存
```

- 基于Spark进行ETL计算，CPU核心 ：内存比例建议>1:2，推荐1：~~4~~1：~~8上一个规则是指纯读写的资源配比，如果Spark的作业除了读写还有业务逻辑计算，该过程会导致需要内存增加，因此建议CPU核心与内存的比例大于1：2，如果逻辑比较复杂适当调大内存，这要基于实际情况进行调整。一般默认推荐配置为1：~~4~~1：8。
- 针对bucket表的写入资源配置，建议给的CPU核心数量不小于桶数目（分区表每次可能写入多个分区，理想情况下建议给的CPU核心数量=写入分区*分桶数；实际配置的core小于这个值，写入性能线性下降）。示例：当前表bucket数为3，同时写入分区数为2， 建议入库Spark任务配置的core数量大于等于3*2。

```
spark-submit
--master yarn-cluster
--executor-cores 2
--executor-memory 4g
--excutor-num 3
```

以上配置代表executor-num*executor-cores=6 >=分区数*分桶数=6。

Spark On Hudi性能调优

优化Spark Shuffle参数提升Hudi写入效率

- 开启spark.shuffle.readHostLocalDisk=true，本地磁盘读取shuffle数据，减少网络传输的开销。
- 开启spark.io.encryption.enabled=false，关闭shuffle过程写加密磁盘，提升shuffle效率。
- 开启spark.shuffle.service.enabled=true，启动shuffle服务，提升任务shuffle的稳定性。

配置项	集群默认值	调整后

配置项	集群默认值	调整后
--conf spark.shuffle.readHostLocalDisk	false	true
--conf spark.io.encryption.enabled	true	false
--conf spark.shuffle.service.enabled	false	true

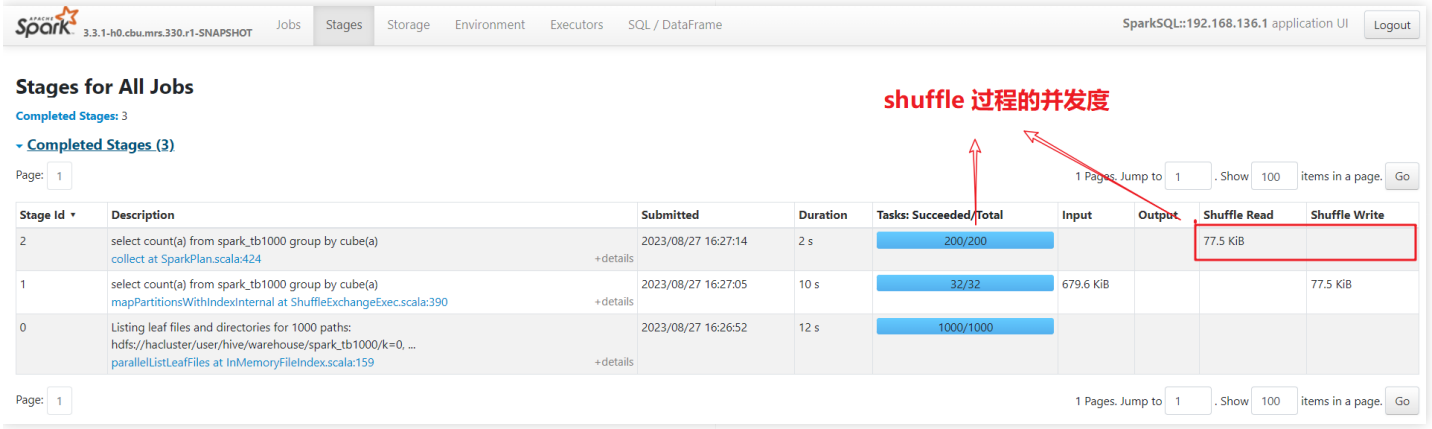
调整Spark调度参数优化OBS场景下Spark调度时延

- 开启对于OBS存储，可以关闭Spark的本地性进行优化，尽可能提升Spark调度效率。

配置项	集群默认值	调整后
--conf spark.locality.wait	3s	0s
--conf spark.locality.wait.process	3s	0s
--conf spark.locality.wait.node	3s	0s
--conf spark.locality.wait.rack	3s	0s

优化shuffle并行度，提升Spark加工效率

所谓的shuffle并发度如下图所示：



集群默认是200，作业可以单独设置。如果发现瓶颈stage（执行时间长），且分配给当前作业的核数大于当前的并发数，说明并发度不足。通过以下配置优化。

场景	配置项	集群默认值	调整后
Jar作业	spark.default.parallelism	200	按实际作业可用资源2倍设置
SQL作业	spark.sql.shuffle.partitions	200	按实际作业可用资源2倍设置

场景	配置项	集群默认值	调整后
hudi入库作业	hoodie.upsert.shuffle.parallelism	200	非bucket表使用，按实际作业可用资源2倍设置

> 注意：
>
> 动态资源调度情况下（spark.dynamicAllocation.enabled= true）时，资源按照spark.dynamicAllocation.maxExecutors评估。

Bucket表，可以开启桶裁剪提升主键点查效率

示例：
业务经常使用主键id作为查询条件，执行点查；比如select xxx where id = idx ... 。
建表时，可以加入如下属性，提升查询效率。默认配置下属性值等于primaryKey，即主键。

```
hoodie.bucket.index.hash.field=id
```

初始化Hudi表时，可以使用BulkInsert方式快速写入数据

示例：

```
set hoodie.combine.before.insert=true;              --入库前去重，如果数据没有重复 该参数无需设置。
set hoodie.datasource.write.operation = bulk_insert;  --指定写入方式为bulk insert方式。
set hoodie.bulkinsert.shuffle.parallelism = 4;       --指定bulk_insert写入时的并行度，等于写入完成后保存的分区parquet文件数。
insert into dsrTable select * from srcTabble
```

开启log列裁剪，提升MOR表查询效率

mor表读取的时候涉及到Log和Parquet的合并，性能不是很理想。可以开启log列裁剪减少合并时IO读取开销。
SparkSQL执行查询，先执行：

```
set hoodie.enable.log.column.prune=true;
```

Spark加工Hudi表时其他参数优化

- 设置spark.sql.enableToString=false，降低Spark解析复杂SQL时候内存使用，提升解析效率。
- 设置spark.speculation=false，关闭推测执行，开启该参数会带来额外的cpu消耗，同时Hudi不支持启动该参数，启用该参数写Hudi有概率导致文件损坏。

配置项	集群默认值	调整后

配置项	集群默认值	调整后
--conf spark.sql.enableToString	true	false
--conf spark.speculation	false	false

Bucket调优示例

创建Bucket索引表调优

Bucket索引常用设置参数：

- Spark：

```
hoodie.index.type=BUCKET
hoodie.bucket.index.num.buckets=5
```

- Flink

```
index.type=BUCKET
hoodie.bucket.index.num.buckets=5
```

判断使用分区表还是非分区表

根据表的使用场景一般将表分为事实表和维度表：

- 事实表通常整表数据规模较大，以新增数据为主，更新数据占比小，且更新数据大多落在近一段时间范围内（年或月或天），下游读取该表进行ETL计算时通常会使用时间范围进行裁剪（例如最近一天、一月、一年），这种表通常可以通过数据的创建时间来做分区以保证最佳读写性能。
- 维度表数据量一般整表数据规模较小，以更新数据为主，新增较少，表数据量比较稳定，且读取时通常需要全量读取做join之类的ETL计算，因此通常使用非分区表性能更好。
- 分区表的分区键不允许更新，否则会产生重复数据。
  例外场景：超大维度表和超小事实表
  特殊情况如存在持续大量新增数据的维度表（表数据量在200G以上或日增长量超过60M）或数据量非常小的事实表（表数据量小于10G且未来三至五年增长后也不会超过10G）需要针对具体场景来进行例外处理：
- 持续大量新增数据的维度表方法一：预留桶数，如使用非分区表则需通过预估较长一段时间内的数据增量来预先增加桶数，缺点是随着数据的增长，文件依然会持续膨胀；方法二：大粒度分区（推荐），如果使用分区表则需要根据数据增长情况来计算，例如使用年分区，这种方式相对麻烦些但是多年后表无需重新导入。方法

三：数据老化，按照业务逻辑分析大的维度表是否可以通过数据老化清理无效的维度数据从而降低数据规模。

- 数据量非常小的事实表这种可以在预估很长一段时间的数据增长量的前提下使用非分区表预留稍宽裕一些的桶数来提升读写性能。

确认表内桶数

Hudi表的桶数设置，关系到表的性能，需要格外引起注意。

以下几点，是设置桶数的关键信息，需要建表前确认。

- 非分区表

1. 单表数据总条数 = select count(1) from tablename（入湖时需提供）；
2. 单条数据大小 = 平均 1KB（华为建议通过select * from tablename limit 100将查询结果粘贴在notepad++中得出100条数据的大小再除以100得到单条平均大小）
3. 单表数据量大小(G) = 单表数据总条数*单表数据大小/1024/1024
4. 非分区表桶数 = MAX（单表数据量大小(G)/2G*2，再向上取整，4）

- 分区表

1. 最近一个月最大数据量分区数据总条数 = 入湖前咨询产品线
2. 单条数据大小 = 平均 1KB（华为建议通过select * from tablename limit 100将查询结果粘贴在notepad++中得出100条数据的大小再除以100得到单条平均大小）
3. 单分区数据量大小(G) = 最近一个月最大数据量分区数据总条数*单表数据大小/1024/1024
4. 分区表桶数 = MAX（单分区数据量大小(G)/2G，再后向上取整，1）

> 注意：
>
> i. 需要使用的是表的总数据大小，而不是压缩以后的文件大小。
> ii. 桶的设置以偶数最佳，非分区表最小桶数请设置4个，分区表最小桶数请设置1个。

最佳实践

hudi 数据重排序

在业务数据使用上，如果业务经常对表的某些字段进行过滤查询（排序字段可以有多个，但是如果查询过滤条件只使用第二排序字段的话，不会起到优化作用。如果同时使用第一、第二排序字段，会有优化作用），此时对该hudi 表数据进行数据的重排序，可以有效提高查询效率。

构造查询数据

创建hive 数据表

```
CREATE TABLE sample_data_partitioned (
  id INT,
  name STRING,
  age INT,
  city STRING,
  date_str STRING
)
PARTITIONED BY (event_date STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

导入hive 数据表

hive 表对应的数据文件为：

1. 将上述数据文件上传到hdfs集群上，路径为：/sample_data_partitioned.csv
2. 使用一下命令，将数据load hive 表中：LOAD DATA INPATH '/sample_data_partitioned.csv' INTO TABLE sample_data_partitioned PARTITION(event_date='2023-11-01');

创建hudi 数据表

```
create table small_file_hudi_cow (
  id int,
  name string,
  age int,
  city STRING,
  date_str STRING
```

```
) using hudi
tblproperties (
  type = 'cow',
  primaryKey = 'id',
  preCombineField = 'id'
)
partitioned by (date_str);
```

导入hudi 数据

1. 开启spark sql 客户端，加载hudi 相关包和配置：

```
spark-sql --master yarn \
--num-executors 2 \
--executor-memory 3g \
--executor-cores 2 \
--jars /usr/local/service/spark/jars/hudi-spark3.2-bundle_2.12-0.12.0.jar \
--conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer' \
--conf 'spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension' \
--conf 'spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog'
```

2. 执行查询导入hudi 数据

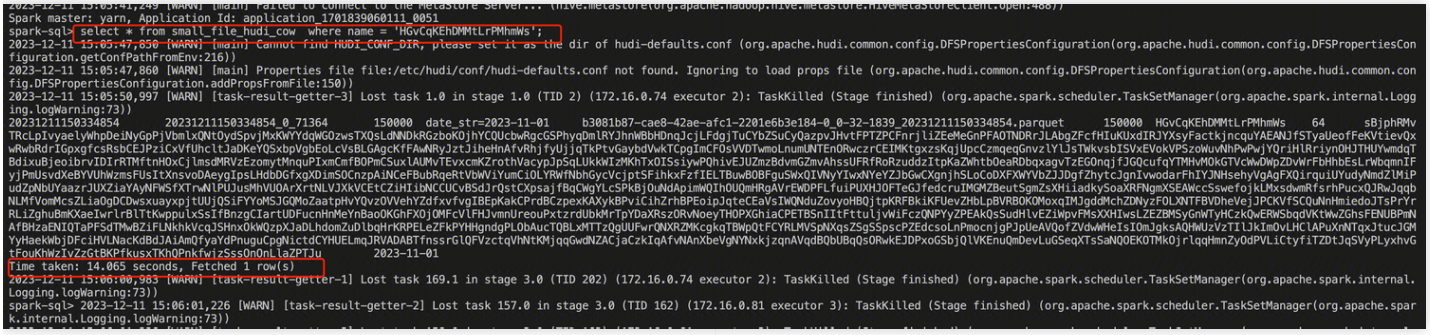
```
INSERT INTO small_file_hudi_cow SELECT id, name, age, city, event_date FROM sample_data_partitione
d where event_date='2023-11-01';
```

数据查询性能

1. 数据查询语句：select * from small_file_hudi_cow where name = 'HGvCqKEhDMMtLrPMhmWs';

2. 为了减少数据预加载影响，每次重新启动spark sql 客户端进行测试。

原始查询耗时



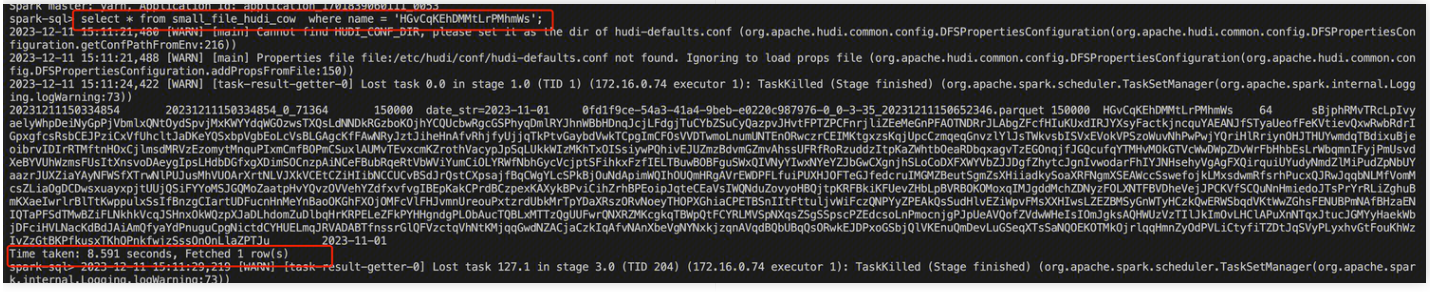
数据重排序查询耗时

对数据进行重排序

异步进行数据重排序和小文件合并处理：

```
spark-submit \
 --jars '/xxx/hudi-utilities-bundle_2.12-*.jar' \
 --class 'org.apache.hudi.utilities.HoodieClusteringJob' \
 /xxx/hudi-utilities-bundle_2.12-*.jar \
 --spark-memory '2g' \
 --mode 'scheduleAndExecute' \
 --base-path "/usr/hive/warehouse/small_file_hudi_cow/" \
 --table-name "small_file_hudi_cow" \
 --hoodie-conf "hoodie.clustering.async.enabled=true" \
 --hoodie-conf "hoodie.clustering.async.max.commits=1" \
 --hoodie-conf "hoodie.clustering.plan.strategy.sort.columns=name" \
 --hoodie-conf "hoodie.clustering.plan.strategy.small.file.limit=314572800" \
 --hoodie-conf "hoodie.clustering.plan.strategy.target.file.max.bytes=1073741824"
```

重排序后查询耗时



具体使用还可参考：

https://cloud.tencent.com/developer/article/1928686

https://hudi.apache.org/docs/clustering#execution-strategy

Hudi Savepoint操作说明

Savepoint用于保存并还原自定义的版本数据。（MoR表暂时不支持Savepoint）
Hudi提供的Savepoint就可以将不同的commit保存起来以便清理程序不会将其删除，后续可以使用Rollback进行恢复。
使用Spark SQL管理Savepoint。
示例如下：

- 创建Savepoint

```
call create_savepoint(table => 'hudi_cow_nonpcf_tbl', commit_time => '20240906173805505');
```

- 查看所有存在的Savepoint

```
call show_savepoints(table => 'hudi_cow_nonpcf_tbl');
```

- 回滚Savepoint

```
call rollback_to_savepoint(table => 'hudi_cow_nonpcf_tbl', instant_time => '20240906173805505');
```

hudi-utilities之HoodieDeltaStreamer

工具说明

HoodieDeltaStreamer工具 (hudi-utilities-bundle中的一部分) 提供了从DFS或Kafka等不同来源进行摄取的方式，并具有以下功能：

- 精准一次从Kafka采集新数据，从Sqoop、HiveIncrementalPuller的输出或DFS文件夹下的文件增量导入。
- 导入的数据支持json、Avro或自定义数据类型。
- 管理检查点，回滚和恢复。
- 利用 DFS 或 Confluent schema registry的 Avro Schema。
- 支持自定义转换操作。

```
[hadoop@10 spark]$ spark-submit \
> --class org.apache.hudi.utilities.deltastreamer.HoodieDeltaStreamer \
> --jars /usr/local/service/spark/jars/hudi-spark3.4-bundle_2.12-0.14.1.jar \
```

```
> /usr/local/service/spark/jars/hudi-utilities-slim-bundle_2.12-0.14.1.jar --help
Usage: <main class> [options]
...
```

注：默认使用的是hudi-utilities-slim-bundle工具，因此在使用这个工具时需要连同hudi-spark3.4-bundle的jar包一起带上。

SqlSource

创建Hive历史表

```
create database hive_test location '/hive_test';
create table hive_test.test_source (
  id int,
  name string,
  price double,
  dt string,
  ts bigint
);
insert into hive_test.test_source values (105,'hudi', 10.0,'2024-07-17',100);
```

Spark SQL创建Hudi目标表

```
create database hudi location '/hudi';
create table hudi.test_hudi_target (
  id int,
  name string,
  price double,
  ts long,
  dt string
) using hudi
partitioned by (dt)
options (
  primaryKey = 'id',
  preCombineField = 'ts',
  type = 'cow'
);
```

配置文件

common.properties

```
hoodie.datasource.write.hive_style_partitioning=true
```

```
hoodie.datasource.write.keygenerator.class=org.apache.hudi.keygen.SimpleKeyGenerator
hoodie.datasource.hive_sync.use_jdbc=false
hoodie.datasource.hive_sync.partition_extractor_class=org.apache.hudi.hive.MultiPartKeysValueExtract
or
```

sql_source.properties

```
include=common.properties
hoodie.datasource.write.recordkey.field=id
hoodie.datasource.write.partitionpath.field=dt
# 非分区表配置 hoodie.datasource.write.partitionpath.field=
hoodie.deltastreamer.source.sql.sql.query = select * from hive_test.test_source
# 和同步Hive相关的配置
hoodie.datasource.hive_sync.table=test_hudi_target
hoodie.datasource.hive_sync.database=hudi
## 非分区表可以不设置
hoodie.datasource.hive_sync.partition_fields=dt
## 内部表，默认外部表
hoodie.datasource.hive_sync.create_managed_table = true
hoodie.datasource.hive_sync.serde_properties = primaryKey=id
```

启动命令

```
spark-submit --conf "spark.sql.catalogImplementation=hive" \
--master yarn --deploy-mode client --executor-memory 2G --num-executors 3 --executor-cores 2 --dri
ver-memory 4G --driver-cores 2 \
--jars /usr/local/service/spark/jars/hudi-spark3.4-bundle_2.12-0.14.1.jar \
--principal hadoop/10.206.16.129@TBDS-MWY6XL24 --keytab /var/krb5kdc/emr.keytab \
--class org.apache.hudi.utilities.deltastreamer.HoodieDeltaStreamer /usr/local/service/spark/jars/hudi-
utilities-slim-bundle_2.12-0.14.1.jar \
--props file:///usr/local/service/spark/sql_source.properties  \
--target-base-path /hudi/test_hudi_target \
--target-table test_hudi_target \
--op BULK_INSERT \
--table-type COPY_ON_WRITE \
--source-ordering-field ts \
--source-class org.apache.hudi.utilities.sources.SqlSource \
--enable-sync  \
--checkpoint earliest \
--hoodie-conf 'hoodie.datasource.hive_sync.create_managed_table = true' \
--hoodie-conf 'hoodie.datasource.hive_sync.serde_properties = primaryKey=id'
```

注：如果运行中报如下错误：Caused by: org.apache.avro.SchemaParseException: Illegal character in: test_source_json.id

需要修改Hive配置（hive-site.xml）：hive.resultset.use.unique.column.names=false

KafkaSource

创建Kerberos环境Kafka客户端认证文件：kafka-client.properties

```
security.protocol=SASL_PLAINTEXT
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=hadoop
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true storeKey=tr
ue keyTab="./admin.keytab" principal="admin@TBDS-ME7KG01C";
```

创建kafka测试数据

```
# 创建topic
/usr/local/service/kafka/bin/kafka-topics.sh --bootstrap-server 10.206.16.81:9092 --create --topic hudi
_test --command-config /tmp/kafka-client.properties

# 造数据
/usr/local/service/kafka/bin/kafka-console-producer.sh --topic hudi_test --bootstrap-server  10.206.16.
81:9092 --producer.config /tmp/kafka-client.properties

{"id":1,"name":"hudi","price":11.0,"ts":100,"dt":"2024-07-17"}
{"id":2,"name":"hudi","price":12.0,"ts":100,"dt":"2024-07-17"}
{"id":3,"name":"hudi","price":13.0,"ts":100,"dt":"2024-07-17"}

# 测试消费 消费topic验证数据是否成功写到对应的topic
/usr/local/service/kafka/bin/kafka-console-consumer.sh --topic hudi_test --from-beginning --bootstra
p-server 10.206.16.81:9092 --group test_hudi_group  --consumer.config /tmp/kafka-client.properties

{"id":1,"name":"hudi","price":11.0,"ts":100,"dt":"2024-07-17"}
{"id":2,"name":"hudi","price":12.0,"ts":100,"dt":"2024-07-17"}
{"id":3,"name":"hudi","price":13.0,"ts":100,"dt":"2024-07-17"}
```

Hudi配置文件

kafka_source.properties

```
include=common.properties

hoodie.datasource.write.recordkey.field=id
hoodie.datasource.write.partitionpath.field=dt

hoodie.streamer.source.kafka.topic=hudi_test
```

```
bootstrap.servers=10.206.16.74:9092
auto.offset.reset=earliest
group.id=test_hudi_group_2
security.protocol=SASL_PLAINTEXT
sasl.mechanism=GSSAPI
sasl.kerberos.service.name=hadoop
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true storeKey=tr
ue keyTab="./admin.keytab" principal="admin@TBDS-MWY6XL24";

hoodie.deltastreamer.schemaprovider.source.schema.file=hdfs://HDFS78000026/tmp/hudi/spark/sourc
e-schema-json.avsc
hoodie.deltastreamer.schemaprovider.target.schema.file=hdfs://HDFS78000026/tmp/hudi/spark/targe
t-schema-json.avsc
```

source-schema-json.avsc

```
{
  "type": "record",
  "name": "Profiles",
  "fields": [
    {
      "name": "id",
      "type": [ "null", "int" ],
      "default": null
    },
    {
      "name": "name",
      "type": [ "null", "string" ],
      "default": null
    },
    {
      "name": "price",
      "type": [ "null", "double" ],
      "default": null
    },
    {
      "name": "ts",
      "type": [ "null", "long" ],
      "default": null
    },
    {
      "name": "dt",
      "type": [ "null", "string" ],
      "default": null
    }
  ]
}
```

```
# 创建target schema
cp source-schema-json.avsc target-schema-json.avsc
将两个文件上传到hdfs
```

启动命令

```
spark-submit --principal admin@TBDS-MWY6XL24 --keytab ./admin.keytab \
--deploy-mode client \
--files ./admin.keytab \
--jars /usr/local/service/spark/jars/hudi-spark3.4-bundle_2.12-0.14.1.jar \
--conf "spark.executor.extraJavaOptions=-Dsun.security.krb5.debug=true" \
--conf "spark.driver.extraJavaOptions=-Dsun.security.krb5.debug=true" \
--class org.apache.hudi.utilities.deltastreamer.HoodieDeltaStreamer  \
/usr/local/service/spark/jars/hudi-utilities-slim-bundle_2.12-0.14.1.jar  \
--props file:///usr/local/service/spark/kafka_source.properties \
--schemaprovider-class org.apache.hudi.utilities.schema.FilebasedSchemaProvider  \
--source-class org.apache.hudi.utilities.sources.JsonKafkaSource  \
--source-ordering-field id \
--target-base-path hdfs://HDFS78000026/tmp/hudi/hudi_test  \
--target-table hudi_test \
--op BULK_INSERT \
--table-type MERGE_ON_READ
```

上面的都是一次性读取转化，kafka也可以连续模式读取增量数据，通过参数--continuous,即：

```
spark-submit --principal admin@TBDS-MWY6XL24 --keytab ./admin.keytab \
--deploy-mode client \
--files ./admin.keytab \
--jars /usr/local/service/spark/jars/hudi-spark3.4-bundle_2.12-0.14.1.jar \
--conf "spark.executor.extraJavaOptions=-Dsun.security.krb5.debug=true" \
--conf "spark.driver.extraJavaOptions=-Dsun.security.krb5.debug=true" \
--class org.apache.hudi.utilities.deltastreamer.HoodieDeltaStreamer  \
/usr/local/service/spark/jars/hudi-utilities-slim-bundle_2.12-0.14.1.jar  \
--props file:///usr/local/service/spark/kafka_source.properties \
--schemaprovider-class org.apache.hudi.utilities.schema.FilebasedSchemaProvider  \
--source-class org.apache.hudi.utilities.sources.JsonKafkaSource  \
--source-ordering-field id \
--target-base-path hdfs://HDFS78000026/tmp/hudi/hudi_test  \
--target-table hudi_test \
--op BULK_INSERT \
--table-type MERGE_ON_READ \
--continuous
```

连续模式默认间隔0s即没有间隔连续性的读取checkpoint判断kafka（和offset对比）里是否有增量，可以通过参数--

min-sync-interval-seconds来修改间隔，比如 –min-sync-interval-seconds 60，设置60s读取一次。可以往kafka topic里再造几条JSON数据，进行验证，是否可以正常读取增量数据。
注：admin.keytab和admin@TBDS-MWY6XL24 为用户对应的kerberos认证信息，使用时需要在ranger上配置对应的yarn队列权限和kafka topic消费权限。
kafka_source.properties中sasl.jaas.config配置中的认证信息也要与其保持一致。

查看导入结果

启动Spark SQL

```
bin/spark-sql --master yarn --deploy-mode client --num-executors 2 --executor-memory 1g --executor-cores 2 --jars /usr/local/service/spark/jars/hudi-spark3.4-bundle_2.12-0.14.1.jar --conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer' --conf 'spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension' --conf 'spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog' --conf 'spark.kryo.registrator=org.apache.spark.HoodieSparkKryoRegistrar'
```

指定location创建hudi表

```
bin/spark-sql --master yarn --deploy-mode client --num-executors 2 --executor-memory 1g --executor-cores 2 --jars /usr/local/service/spark/jars/hudi-spark3.4-bundle_2.12-0.14.1.jar --conf 'spark.serializer=org.apache.spark.serializer.KryoSerializer' --conf 'spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension' --conf 'spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog' --conf 'spark.kryo.registrator=org.apache.spark.HoodieSparkKryoRegistrar'
```

查询hudi表

```
select * from hudi_test;
```

注：如果要通过Flink SQL插入或者更新spark-hudi表，则spark在创建表时需要配置hive的属性。

```
 'hive_sync.metastore.uris' = 'thrift://10.4.4.11:7004',
 'hive_sync.conf.dir'='/usr/local/service/hive/conf'
如下所示：
create table hudi_mor_tbl (
 id int,
 name string,
 price double,
 ts bigint
) using hudi
tblproperties (
 type = 'mor',
 primaryKey = 'id',
 preCombineField = 'ts',
```

```
  'hive_sync.enable' = 'true',
  'hive_sync.metastore.uris' = 'thrift://10.4.4.11:7004',
  'hive_sync.conf.dir'='/usr/local/service/hive/conf'
);
```

常见问题

Spark查询Hudi数据重复，如何处理？

问题原因：出现Spark查询hudi数据重复，通常是因为Hudi不支持Spark DataSource方式读取导致的。
解决方法：您需要在执行查询Hudi表的命令时，添加上spark.sql.hive.convertMetastoreParquet=false。

Hive查询Hudi数据重复，如何处理？

问题原因：Hive默认使用HiveCombineInputFormat不会调用表自定义的input format。
解决方法：您需要在执行查询Hudi表的命令时，添加上set hive.input.format = org.apache.hudi.hadoop.hive.HoodieCombineHiveInputFormat。

Spark查询Hudi表分区裁剪不生效？

问题原因：可能是在分区字段包含/（正斜线）的情况下，分区字段个数和实际分区目录级数不一致，导致Spark分区裁剪失效。
解决方法：您在使用Spark DataFrame API写Hudi表时，需要加上 hoodie.datasource.write.partitionpath.urlencode= true。

使用Spark的alter table语句时，报错xxx is only supported with v2 tables，如何处理？

问题原因：这是因为使用Hudi Spark的模式演变功能时，需要将Hudi的配置项hoodie.schema.on.read.enable设置为true。
解决方法：请在对Hudi表进行alter table操作时添加配置set hoodie.schema.on.read.enable=true。更多详细信息请参考Apache Hudi的Spark SQL Schema Evolution and Syntax Description。

写时复制（COW）与读时合并（MOR）存储类型之

间有什么区别？

写时复制（Copy On Write）：此存储类型使客户端能够以列式文件格式（当前为parquet）摄取数据。使用COW存储类型时，任何写入Hudi数据集的新数据都将写入新的parquet文件。更新现有的行将导致重写整个parquet文件（这些parquet文件包含要更新的受影响的行）。因此，所有对此类数据集的写入都受parquet写性能的限制，parquet文件越大，摄取数据所花费的时间就越长。
读时合并（Merge On Read）：此存储类型使客户端可以快速将数据摄取为基于行（如Avro）的数据格式。使用MOR存储类型时，任何写入Hudi数据集的新数据都将写入新的日志/增量文件，这些文件在内部将数据以avro进行编码。压缩（Compaction）过程（配置为嵌入式或异步）将日志文件格式转换为列式文件格式（parquet）。
两种不同的格式提供了两种不同视图（读优化视图和实时视图），读优化视图取决于列式parquet文件的读取性能，而实时视图取决于列式和/或日志文件的读取性能。
更新现有的行将导致：a）写入从以前通过压缩（Compaction）生成的基础parquet文件对应的日志/增量文件更新；或b）在未进行压缩的情况下写入日志/增量文件的更新。因此，对此类数据集的所有写入均受Avro /日志文件写入性能的限制，其速度比parquet快得多（写入时需要复制）。虽然，与列式（parquet）文件相比，读取日志/增量文件需要更高的成本（读取时需要合并）。

API文档
腾讯大数据套件（tbds）
版本（2020-01-16）
API 概览

API版本

V3

腾讯大数据套件

接口名称	接口功能
BmsCreateCluster	bms创建集群
BmsDescribeInstancesByName	根据实例名称获取bms实例信息
ClusterExists	判断集群名是否存在
CreateCluster	创建集群
DescribeClusterDetail	获取集群详情
DescribeClusterNodes	获取集群节点
DestroyCluster	销毁集群
GetClusters	获得集群列表
GetProgress	获得集群部署进度
TestApplyBms	bms创建实例接口测试

调用方式
接口签名v1

TCloudFinanceZone API 会对每个访问请求进行身份验证，即每个请求都需要在公共请求参数中包含签名信息（Signature）以验证请求者身份。
签名信息由安全凭证生成，安全凭证包括 SecretId 和 SecretKey；若用户还没有安全凭证，请前往云API密钥页面申请，否则无法调用云API接口。

1. 申请安全凭证

在第一次使用云API之前，请前往云API密钥页面申请安全凭证。
安全凭证包括 SecretId 和 SecretKey：

- SecretId 用于标识 API 调用者身份
- SecretKey 用于加密签名字符串和服务器端验证签名字符串的密钥。
- 用户必须严格保管安全凭证，避免泄露。

申请安全凭证的具体步骤如下：

1. 登录TCloudFinanceZone管理中心控制台。
2. 前往云API密钥的控制台页面
3. 在云API密钥页面，点击【新建】即可以创建一对SecretId/SecretKey

注意：开发商帐号最多可以拥有两对 SecretId / SecretKey。

2. 生成签名串

有了安全凭证SecretId 和 SecretKey后，就可以生成签名串了。以下是生成签名串的详细过程：

假设用户的 SecretId 和 SecretKey 分别是：

- SecretId: AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE
- SecretKey: Gu5t9xGARNpq86cd98joQYCN3EXAMPLE

注意：这里只是示例，请根据用户实际申请的 SecretId 和 SecretKey 进行后续操作！

以云服务器查看实例列表(DescribeInstances)请求为例，当用户调用这一接口时，其请求参数可能如下:

参数名称	中文	参数值

参数名称	中文	参数值
Action	方法名	DescribeInstances
SecretId	密钥Id	AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE
Timestamp	当前时间戳	1465185768
Nonce	随机正整数	11886
Region	实例所在区域	shjr
InstanceIds.0	待查询的实例ID	ins-09dx96dg
Offset	偏移量	0
Limit	最大允许输出	20
Version	接口版本号	2017-03-12

2.1. 对参数排序

首先对所有请求参数按参数名的字典序（ ASCII 码）升序排序。注意：1）只按参数名进行排序，参数值保持对应即可，不参与比大小；2）按 ASCII 码比大小，如 InstanceIds.2 要排在 InstanceIds.12 后面，不是按字母表，也不是按数值。用户可以借助编程语言中的相关排序函数来实现这一功能，如 php 中的 ksort 函数。上述示例参数的排序结果如下:

```
{
    'Action' : 'DescribeInstances',
    'InstanceIds.0' : 'ins-09dx96dg',
    'Limit' : 20,
    'Nonce' : 11886,
    'Offset' : 0,
    'Region' : 'shjr',
    'SecretId' : 'AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE',
    'Timestamp' : 1465185768,
    'Version': '2017-03-12',
}
```

使用其它程序设计语言开发时，可对上面示例中的参数进行排序，得到的结果一致即可。

2.2. 拼接请求字符串

此步骤生成请求字符串。
将把上一步排序好的请求参数格式化成“参数名称”=“参数值”的形式，如对 Action 参数，其参数名称为 "Action" ，参数值为 "DescribeInstances" ，因此格式化后就为 Action=DescribeInstances 。
注意：“参数值”为原始值而非url编码后的值。

然后将格式化后的各个参数用"&"拼接在一起，最终生成的请求字符串为:

```
Action=DescribeInstances&InstanceIds.0=ins-09dx96dg&Limit=20&Nonce=11886&Offset=0&Region=shjr&SecretId=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE&Timestamp=1465185768&Version=2017-03-12
```

2.3. 拼接签名原文字符串

此步骤生成签名原文字符串。
签名原文字符串由以下几个参数构成:

1. 请求方法: 支持 POST 和 GET 方式，这里使用 GET 请求，注意方法为全大写。
2. 请求主机:查看实例列表(DescribeInstances)的请求域名为：cvm.finance.cloud.tencent.com。实际的请求域名根据接口所属模块的不同而不同，详见各接口说明。
3. 请求路径: 当前版本云API的请求路径固定为 / 。
4. 请求字符串: 即上一步生成的请求字符串。

签名原文串的拼接规则为: 请求方法 + 请求主机 +请求路径 + ？+ 请求字符串

示例的拼接结果为：

```
GETcvm.finance.cloud.tencent.com/?Action=DescribeInstances&InstanceIds.0=ins-09dx96dg&Limit=20&Nonce=11886&Offset=0&Region=shjr&SecretId=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE&Timestamp=1465185768&Version=2017-03-12
```

2.4. 生成签名串

此步骤生成签名串。
首先使用 HMAC-SHA1 算法对上一步中获得的签名原文字符串进行签名，然后将生成的签名串使用 Base64 进行编码，即可获得最终的签名串。

具体代码如下，以 PHP 语言为例:

```
$secretKey = 'Gu5t9xGARNpq86cd98joQYCN3EXAMPLE';
$srcStr = 'GETcvm.finance.cloud.tencent.com/?Action=DescribeInstances&InstanceIds.0=ins-09dx96dg&Limit=20&Nonce=11886&Offset=0&Region=shjr&SecretId=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE&Timestamp=1465185768&Version=2017-03-12';
$signStr = base64_encode(hash_hmac('sha1', $srcStr, $secretKey, true));
echo $signStr;
```

最终得到的签名串为:

```
EliP9YW3pW28FpsEdkXt/+WcGeI=
```

使用其它程序设计语言开发时，可用上面示例中的原文进行签名验证，得到的签名串与例子中的一致即可。

3. 签名串编码

生成的签名串并不能直接作为请求参数，需要对其进行 URL 编码。

如上一步生成的签名串为 EliP9YW3pW28FpsEdkXt/+WcGeI= ，最终得到的签名串请求参数（Signature）为：EliP9YW3pW28FpsEdkXt%2f%2bWcGeI%3d，它将用于生成最终的请求 URL。

注意：如果用户的请求方法是 GET，或者请求方法为 POST 同时 Content-Type 为 application/x-www-form-urlencoded，则发送请求时所有请求参数的值均需要做 URL 编码，参数键和=符号不需要编码。非 ASCII 字符在 URL 编码前需要先以 UTF-8 进行编码。

注意：有些编程语言的 http 库会自动为所有参数进行 urlencode，在这种情况下，就不需要对签名串进行 URL 编码了，否则两次 URL 编码会导致签名失败。

注意：其他参数值也需要进行编码，编码采用 RFC 3986。使用 %XY 对特殊字符例如汉字进行百分比编码，其中“X”和“Y”为十六进制字符（0-9 和大写字母 A-F），使用小写将引发错误。

4. 签名失败

根据实际情况，存在以下签名失败的错误码，请根据实际情况处理

错误代码	错误描述
AuthFailure.SignatureExpire	签名过期
AuthFailure.SecretIdNotFound	密钥不存在
AuthFailure.SignatureFailure	签名错误
AuthFailure.TokenFailure	token 错误
AuthFailure.InvalidSecretId	密钥非法（不是云 API 密钥类型）

5. 签名演示

在实际调用 API 3.0 时，推荐使用配套的TCloudFinanceZone SDK 3.0 ，SDK 封装了签名的过程，开发时只关注产品提供的具体接口即可。详细信息参见 SDK 中心。当前支持的编程语言有：

- Python
- Java

- PHP
- Go
- Node

为了更清楚的解释签名过程，下面以实际编程语言为例，将上述的签名过程具体实现。请求的域名、调用的接口和参数的取值都以上述签名过程为准，代码只为解释签名过程，并不具备通用性，实际开发请尽量使用 SDK 。

最终输出的 url 可能为： https://cvm.finance.cloud.tencent.com/?Action=DescribeInstances&InstanceIds.0=ins-09dx96dg&Limit=20&Nonce=11886&Offset=0&Region=shjr&SecretId=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE&Signature=EliP9YW3pW28FpsEdkXt%2F%2BWcGeI%3D&Timestamp=1465185768&Version=2017-03-12

注意：由于示例中的密钥是虚构的，时间戳也不是系统当前时间，因此如果将此 url 在浏览器中打开或者用 curl 等命令调用时会返回鉴权错误：签名过期。为了得到一个可以正常返回的 url ，需要修改示例中的 SecretId 和 SecretKey 为真实的密钥，并使用系统当前时间戳作为 Timestamp 。

注意：在下面的示例中，不同编程语言，甚至同一语言每次执行得到的 url 可能都有所不同，表现为参数的顺序不同，但这并不影响正确性。只要所有参数都在，且签名计算正确即可。

注意：以下代码仅适用于 API 3.0，不能直接用于其他的签名流程，即使是旧版的 API ，由于存在细节差异也会导致签名计算错误，请以对应的实际文档为准。

Java

```
import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;
import java.util.Random;
import java.util.TreeMap;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import javax.xml.bind.DatatypeConverter;

public class CloudAPIDemo {
    private final static String CHARSET = "UTF-8";

    public static String sign(String s, String key, String method) throws Exception {
        Mac mac = Mac.getInstance(method);
        SecretKeySpec secretKeySpec = new SecretKeySpec(key.getBytes(CHARSET), mac.getAlgorithm());
        mac.init(secretKeySpec);
        byte[] hash = mac.doFinal(s.getBytes(CHARSET));
        return DatatypeConverter.printBase64Binary(hash);
    }

    public static String getStringToSign(TreeMap<String, Object> params) {
        StringBuilder s2s = new StringBuilder("GETcvm.finance.cloud.tencent.com/?");
```

```
        // 签名时要求对参数进行字典排序，此处用TreeMap保证顺序
        for (String k : params.keySet()) {
            s2s.append(k).append("=").append(params.get(k).toString()).append("&");
        }
        return s2s.toString().substring(0, s2s.length() - 1);
    }

    public static String getUrl(TreeMap<String, Object> params) throws UnsupportedEncodingException
{
        StringBuilder url = new StringBuilder("https://cvm.finance.cloud.tencent.com/?");
        // 实际请求的url中对参数顺序没有要求
        for (String k : params.keySet()) {
            // 需要对请求串进行urlencode，由于key都是英文字母，故此处仅对其value进行urlencode
            url.append(k).append("=").append(URLEncoder.encode(params.get(k).toString(), CHARSET)).app
end("&");
        }
        return url.toString().substring(0, url.length() - 1);
    }

    public static void main(String[] args) throws Exception {
        TreeMap<String, Object> params = new TreeMap<String, Object>(); // TreeMap可以自动排序
        // 实际调用时应当使用随机数，例如：params.put("Nonce", new Random().nextInt(java.lang.Intege
r.MAX_VALUE));
        params.put("Nonce", 11886); // 公共参数
        // 实际调用时应当使用系统当前时间，例如：  params.put("Timestamp", System.currentTimeMillis() /
1000);
        params.put("Timestamp", 1465185768); // 公共参数
        params.put("SecretId", "AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE"); // 公共参数
        params.put("Action", "DescribeInstances"); // 公共参数
        params.put("Version", "2017-03-12"); // 公共参数
        params.put("Region", "shjr"); // 公共参数
        params.put("Limit", 20); // 业务参数
        params.put("Offset", 0); // 业务参数
        params.put("InstanceIds.0", "ins-09dx96dg"); // 业务参数
        params.put("Signature", sign(getStringToSign(params), "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE
", "HmacSHA1")); // 公共参数
        System.out.println(getUrl(params));
    }
}
```

Python

注意：如果是在 Python 2 环境中运行，需要先安装 requests 依赖包： pip install requests 。

```
# -*- coding: utf8 -*-
import base64
```

```
import hashlib
import hmac
import time

import requests

secret_id = "AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE"
secret_key = "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE"

def get_string_to_sign(method, endpoint, params):
    s = method + endpoint + "/?"
    query_str = "&".join("%s=%s" % (k, params[k]) for k in sorted(params))
    return s + query_str

def sign_str(key, s, method):
    hmac_str = hmac.new(key.encode("utf8"), s.encode("utf8"), method).digest()
    return base64.b64encode(hmac_str)

if __name__ == '__main__':
    endpoint = "cvm.finance.cloud.tencent.com"
    data = {
        'Action' : 'DescribeInstances',
        'InstanceIds.0' : 'ins-09dx96dg',
        'Limit' : 20,
        'Nonce' : 11886,
        'Offset' : 0,
        'Region' : 'shjr',
        'SecretId' : secret_id,
        'Timestamp' : 1465185768, # int(time.time())
        'Version': '2017-03-12'
    }
    s = get_string_to_sign("GET", endpoint, data)
    data["Signature"] = sign_str(secret_key, s, hashlib.sha1)
    print(data["Signature"])
    # 此处会实际调用，成功后可能产生计费
    # resp = requests.get("https://" + endpoint, params=data)
    # print(resp.url)
```

接口签名v3

TCloudFinanceZone API 会对每个访问请求进行身份验证，即每个请求都需要在公共请求参数中包含签名信息（Signature）以验证请求者身份。
签名信息由安全凭证生成，安全凭证包括 SecretId 和 SecretKey；若用户还没有安全凭证，请前往云API密钥页面申请，否则无法调用云API接口。

1. 申请安全凭证

在第一次使用云API之前，请前往云API密钥页面申请安全凭证。
安全凭证包括 SecretId 和 SecretKey：

- SecretId 用于标识 API 调用者身份
- SecretKey 用于加密签名字符串和服务器端验证签名字符串的密钥。
- 用户必须严格保管安全凭证，避免泄露。

申请安全凭证的具体步骤如下：

1. 登录TCloudFinanceZone管理中心控制台。
2. 前往云API密钥的控制台页面
3. 在云API密钥页面，点击【新建】即可以创建一对SecretId/SecretKey

注意：开发商帐号最多可以拥有两对 SecretId / SecretKey。

2. TC3-HMAC-SHA256 签名方法

注意：对于GET方法，只支持 Content-Type: application/x-www-form-urlencoded 协议格式。对于POST方法，目前支持 Content-Type: application/json 以及 Content-Type: multipart/form-data 两种协议格式，json 格式默认所有业务接口均支持，multipart 格式只有特定业务接口支持，此时该接口不能使用 json 格式调用，参考具体业务接口文档说明。

下面以云服务器查询广州区实例列表作为例子，分步骤介绍签名的计算过程。我们仅用到了查询实例列表的两个参数：Limit 和 Offset，使用 GET 方法调用。

假设用户的 SecretId 和 SecretKey 分别是：AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE 和 Gu5t9xGARNpq86cd98joQYCN3EXAMPLE

2.1. 拼接规范请求串

按如下格式拼接规范请求串（CanonicalRequest）：

```
CanonicalRequest =
    HTTPRequestMethod + '\n' +
    CanonicalURI + '\n' +
    CanonicalQueryString + '\n' +
    CanonicalHeaders + '\n' +
    SignedHeaders + '\n' +
    HashedRequestPayload
```

- HTTPRequestMethod：HTTP 请求方法（GET、POST ），本示例中为 GET；
- CanonicalURI：URI 参数，API 3.0 固定为正斜杠（/）；
- CanonicalQueryString：发起 HTTP 请求 URL 中的查询字符串，对于 POST 请求，固定为空字符串，对于 GET 请求，则为 URL 中问号（?）后面的字符串内容，本示例取值为：Limit=10&Offset=0。注意：CanonicalQueryString 需要经过 URL 编码。
- CanonicalHeaders：参与签名的头部信息，至少包含 host 和 content-type 两个头部，也可加入自定义的头部参与签名以提高自身请求的唯一性和安全性。拼接规则：1）头部 key 和 value 统一转成小写，并去掉首尾空格，按照 key:value\n 格式拼接；2）多个头部，按照头部 key（小写）的字典排序进行拼接。此例中为： content-type:application/x-www-form-urlencoded\nhost:cvm.finance.cloud.tencent.com\n
- SignedHeaders：参与签名的头部信息，说明此次请求有哪些头部参与了签名，和 CanonicalHeaders 包含的头部内容是一一对应的。content-type 和 host 为必选头部。拼接规则：1）头部 key 统一转成小写；2）多个头部 key（小写）按照字典排序进行拼接，并且以分号（;）分隔。此例中为： content-type;host
- HashedRequestPayload：请求正文的哈希值，计算方法为 Lowercase(HexEncode(Hash.SHA256(RequestPayload)))，对 HTTP 请求整个正文 payload 做 SHA256 哈希，然后十六进制编码，最后编码串转换成小写字母。注意：对于 GET 请求，RequestPayload 固定为空字符串，对于 POST 请求，RequestPayload 即为 HTTP 请求正文 payload。

根据以上规则，示例中得到的规范请求串如下（为了展示清晰，\n 换行符通过另起打印新的一行替代）：

```
GET
/
Limit=10&Offset=0
content-type:application/x-www-form-urlencoded
host:cvm.finance.cloud.tencent.com

content-type;host
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
```

2.2. 拼接待签名字符串

按如下格式拼接待签名字符串：

```
StringToSign =
    Algorithm + \n +
```

```
RequestTimestamp + \n +
CredentialScope + \n +
HashedCanonicalRequest
```

- Algorithm：签名算法，目前固定为 TC3-HMAC-SHA256；
- RequestTimestamp：请求时间戳，即请求头部的 X-TC-Timestamp 取值，如上示例请求为 1539084154；
- CredentialScope：凭证范围，格式为 Date/service/tc3_request，包含日期、所请求的服务和终止字符串（tc3_request）。Date 为 UTC 标准时间的日期，取值需要和公共参数 X-TC-Timestamp 换算的 UTC 标准时间日期一致；service 为产品名，必须与调用的产品域名一致，例如 cvm。如上示例请求，取值为 2018-10-09/cvm/tc3_request；
- HashedCanonicalRequest：前述步骤拼接所得规范请求串的哈希值，计算方法为 Lowercase(HexEncode(Hash.SHA256(CanonicalRequest)))。

注意：

> 1. Date 必须从时间戳 X-TC-Timestamp 计算得到，且时区为 UTC+0。如果加入系统本地时区信息，例如东八区，将导致白天和晚上调用成功，但是凌晨时调用必定失败。假设时间戳为 1551113065，在东八区的时间是 2019-02-26 00:44:25，但是计算得到的 Date 取 UTC+0 的日期应为 2019-02-25，而不是 2019-02-26。
> 2. Timestamp 必须是当前系统时间，且需确保系统时间和标准时间是同步的，如果相差超过五分钟则必定失败。如果长时间不和标准时间同步，可能导致运行一段时间后，请求必定失败（返回签名过期错误）。

根据以上规则，示例中得到的待签名字符串如下（为了展示清晰，\n 换行符通过另起打印新的一行替代）：

```
TC3-HMAC-SHA256
1539084154
2018-10-09/cvm/tc3_request
91c9c192c14460df6c1ffc69e34e6c5e90708de2a6d282cccf957dbf1aa7f3a7
```

2.3. 计算签名

1）计算派生签名密钥，伪代码如下

```
SecretKey = "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE"
SecretDate = HMAC_SHA256("TC3" + SecretKey, Date)
SecretService = HMAC_SHA256(SecretDate, Service)
SecretSigning = HMAC_SHA256(SecretService, "tc3_request")
```

- SecretKey：原始的 SecretKey；
- Date：即 Credential 中的 Date 字段信息，如上示例，为2018-10-09；
- Service：即 Credential 中的 Service 字段信息，如上示例，为 cvm；

2）计算签名，伪代码如下

```
Signature = HexEncode(HMAC_SHA256(SecretSigning, StringToSign))
```

- SecretSigning：即以上计算得到的派生签名密钥；
- StringToSign：即步骤2计算得到的待签名字符串；

2.4. 拼接 Authorization

按如下格式拼接 Authorization：

```
Authorization =
    Algorithm + ' ' +
    'Credential=' + SecretId + '/' + CredentialScope + ', ' +
    'SignedHeaders=' + SignedHeaders + ', '
    'Signature=' + Signature
```

- Algorithm：签名方法，固定为 TC3-HMAC-SHA256；
- SecretId：密钥对中的 SecretId；
- CredentialScope：见上文，凭证范围；
- SignedHeaders：见上文，参与签名的头部信息；
- Signature：签名值

根据以上规则，示例中得到的值为：

```
TC3-HMAC-SHA256 Credential=AKIDEXAMPLE/Date/service/tc3_request, SignedHeaders=content-type;host, Signature=5da7a33f6993f0614b047e5df4582db9e9bf4672ba50567dba16c6ccf174c474
```

最终完整的调用信息如下：

```
https://cvm.finance.cloud.tencent.com/?Limit=10&Offset=0

Authorization: TC3-HMAC-SHA256 Credential=AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE/2018-10-09/cvm/tc3_request, SignedHeaders=content-type;host, Signature=5da7a33f6993f0614b047e5df4582db9e9bf4672ba50567dba16c6ccf174c474
Content-Type: application/x-www-form-urlencoded
Host: cvm.finance.cloud.tencent.com
X-TC-Action: DescribeInstances
X-TC-Version: 2017-03-12
X-TC-Timestamp: 1539084154
X-TC-Region: shjr
```

3. 签名失败

根据实际情况，存在以下签名失败的错误码，请根据实际情况处理

错误代码	错误描述
AuthFailure.SignatureExpire	签名过期
AuthFailure.SecretIdNotFound	密钥不存在
AuthFailure.SignatureFailure	签名错误
AuthFailure.TokenFailure	token 错误
AuthFailure.InvalidSecretId	密钥非法（不是云 API 密钥类型）

4. 签名演示

Java

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Map;
import java.util.TimeZone;
import java.util.TreeMap;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import javax.net.ssl.HttpsURLConnection;
import javax.xml.bind.DatatypeConverter;

import org.apache.commons.codec.digest.DigestUtils;

public class CloudAPITC3Demo {
    private final static String CHARSET = "UTF-8";
    private final static String ENDPOINT = "cvm.finance.cloud.tencent.com";
    private final static String PATH = "/";
    private final static String SECRET_ID = "AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE";
    private final static String SECRET_KEY = "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE";
    private final static String CT_X_WWW_FORM_URLENCODED = "application/x-www-form-urlencode
d";
    private final static String CT_JSON = "application/json";
```

```
    private final static String CT_FORM_DATA = "multipart/form-data";

    public static byte[] sign256(byte[] key, String msg) throws Exception {
        Mac mac = Mac.getInstance("HmacSHA256");
        SecretKeySpec secretKeySpec = new SecretKeySpec(key, mac.getAlgorithm());
        mac.init(secretKeySpec);
        return mac.doFinal(msg.getBytes(CHARSET));
    }

    public static void main(String[] args) throws Exception {
        String service = "cvm";
        String host = "cvm.finance.cloud.tencent.com";
        String region = "shjr";
        String action = "DescribeInstances";
        String version = "2017-03-12";
        String algorithm = "TC3-HMAC-SHA256";
        String timestamp = "1539084154";
        //String timestamp = String.valueOf(System.currentTimeMillis() / 1000);
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        // 注意时区，否则容易出错
        sdf.setTimeZone(TimeZone.getTimeZone("UTC"));
        String date = sdf.format(new Date(Long.valueOf(timestamp + "000")));

        // ************* 步骤 1：拼接规范请求串 *************
        String httpRequestMethod = "GET";
        String canonicalUri = "/";
        String canonicalQueryString = "Limit=10&Offset=0";
        String canonicalHeaders = "content-type:application/x-www-form-urlencoded\n" + "host:" + host
+ "\n";
        String signedHeaders = "content-type;host";
        String hashedRequestPayload = DigestUtils.sha256Hex("");
        String canonicalRequest = httpRequestMethod + "\n" + canonicalUri + "\n" + canonicalQueryStri
ng + "\n"
                + canonicalHeaders + "\n" + signedHeaders + "\n" + hashedRequestPayload;
        System.out.println(canonicalRequest);

        // ************* 步骤 2：拼接待签名字符串 *************
        String credentialScope = date + "/" + service + "/" + "tc3_request";
        String hashedCanonicalRequest = DigestUtils.sha256Hex(canonicalRequest.getBytes(CHARSET));
        String stringToSign = algorithm + "\n" + timestamp + "\n" + credentialScope + "\n" + hashedCan
onicalRequest;
        System.out.println(stringToSign);

        // ************* 步骤 3：计算签名 *************
        byte[] secretDate = sign256(("TC3" + SECRET_KEY).getBytes(CHARSET), date);
        byte[] secretService = sign256(secretDate, service);
        byte[] secretSigning = sign256(secretService, "tc3_request");
```

```
        String signature = DatatypeConverter.printHexBinary(sign256(secretSigning, stringToSign)).toLow
erCase();
        System.out.println(signature);

        // ************* 步骤 4：拼接 Authorization *************
        String authorization = algorithm + " " + "Credential=" + SECRET_ID + "/" + credentialScope + ", "
                + "SignedHeaders=" + signedHeaders + ", " + "Signature=" + signature;
        System.out.println(authorization);

        TreeMap<String, String> headers = new TreeMap<String, String>();
        headers.put("Authorization", authorization);
        headers.put("Host", host);
        headers.put("Content-Type", CT_X_WWW_FORM_URLENCODED);
        headers.put("X-TC-Action", action);
        headers.put("X-TC-Timestamp", timestamp);
        headers.put("X-TC-Version", version);
        headers.put("X-TC-Region", region);
    }
}
```

Python

```
# -*- coding: utf-8 -*-
import hashlib, hmac, json, os, sys, time
from datetime import datetime

# 密钥参数
secret_id = "AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE"
secret_key = "Gu5t9xGARNpq86cd98joQYCN3EXAMPLE"

service = "cvm"
host = "cvm.finance.cloud.tencent.com"
endpoint = "https://" + host
region = "shjr"
action = "DescribeInstances"
version = "2017-03-12"
algorithm = "TC3-HMAC-SHA256"
timestamp = 1539084154
date = datetime.utcfromtimestamp(timestamp).strftime("%Y-%m-%d")
params = {"Limit": 10, "Offset": 0}

# ************* 步骤 1：拼接规范请求串 *************
http_request_method = "GET"
canonical_uri = "/"
canonical_querystring = "Limit=10&Offset=0"
ct = "x-www-form-urlencoded"
```

```
payload = ""
if http_request_method == "POST":
    canonical_querystring = ""
    ct = "json"
    payload = json.dumps(params)
canonical_headers = "content-type:application/%s\nhost:%s\n" % (ct, host)
signed_headers = "content-type;host"
hashed_request_payload = hashlib.sha256(payload.encode("utf-8")).hexdigest()
canonical_request = (http_request_method + "\n" +
                     canonical_uri + "\n" +
                     canonical_querystring + "\n" +
                     canonical_headers + "\n" +
                     signed_headers + "\n" +
                     hashed_request_payload)
print(canonical_request)

# ************* 步骤 2：拼接待签名字符串 *************
credential_scope = date + "/" + service + "/" + "tc3_request"
hashed_canonical_request = hashlib.sha256(canonical_request.encode("utf-8")).hexdigest()
string_to_sign = (algorithm + "\n" +
                  str(timestamp) + "\n" +
                  credential_scope + "\n" +
                  hashed_canonical_request)
print(string_to_sign)

# ************* 步骤 3：计算签名 *************
# 计算签名摘要函数
def sign(key, msg):
    return hmac.new(key, msg.encode("utf-8"), hashlib.sha256).digest()
secret_date = sign(("TC3" + secret_key).encode("utf-8"), date)
secret_service = sign(secret_date, service)
secret_signing = sign(secret_service, "tc3_request")
signature = hmac.new(secret_signing, string_to_sign.encode("utf-8"), hashlib.sha256).hexdigest()
print(signature)

# ************* 步骤 4：拼接 Authorization *************
authorization = (algorithm + " " +
                 "Credential=" + secret_id + "/" + credential_scope + ", " +
                 "SignedHeaders=" + signed_headers + ", " +
                 "Signature=" + signature)
print(authorization)

# 公共参数添加到请求头部
headers = {
    "Authorization": authorization,
    "Host": host,
    "Content-Type": "application/%s" % ct,
```

```
    "X-TC-Action": action,
    "X-TC-Timestamp": str(timestamp),
    "X-TC-Version": version,
    "X-TC-Region": region,
}
```

请求结构

1. 服务地址

地域（Region）是指物理的数据中心的地理区域。TCloudFinanceZone交付验证不同地域之间完全隔离，保证不同地域间最大程度的稳定性和容错性。为了降低访问时延、提高下载速度，建议您选择最靠近您客户的地域。

您可以通过 API接口 查询地域列表 查看完成的地域列表。

2. 通信协议

TCloudFinanceZone API 的所有接口均通过 HTTPS 进行通信，提供高安全性的通信通道。

3. 请求方法

支持的 HTTP 请求方法:

- POST（推荐）
- GET

POST 请求支持的 Content-Type 类型：

- application/json（推荐），必须使用 TC3-HMAC-SHA256 签名方法。
- application/x-www-form-urlencoded，必须使用 HmacSHA1 或 HmacSHA256 签名方法。
- multipart/form-data（仅部分接口支持），必须使用 TC3-HMAC-SHA256 签名方法。

GET 请求的请求包大小不得超过 32 KB。POST 请求使用签名方法为 HmacSHA1、HmacSHA256 时不得超过 1 MB。POST 请求使用签名方法为 TC3-HMAC-SHA256 时支持 10 MB。

4. 字符编码

均使用UTF-8编码。

返回结果

正确返回结果

以云服务器的接口查看实例状态列表 (DescribeInstancesStatus) 2017-03-12 版本为例，若调用成功，其可能的返回如下为：

```
{
    "Response": {
        "TotalCount": 0,
        "InstanceStatusSet": [],
        "RequestId": "b5b41468-520d-4192-b42f-595cc34b6c1c"
    }
}
```

- Response 及其内部的 RequestId 是固定的字段，无论请求成功与否，只要 API 处理了，则必定会返回。
- RequestId 用于一个 API 请求的唯一标识，如果 API 出现异常，可以联系我们，并提供该 ID 来解决问题。
- 除了固定的字段外，其余均为具体接口定义的字段，不同的接口所返回的字段参见接口文档中的定义。此例中的 TotalCount 和 InstanceStatusSet 均为 DescribeInstancesStatus 接口定义的字段，由于调用请求的用户暂时还没有云服务器实例，因此 TotalCount 在此情况下的返回值为 0， InstanceStatusSet 列表为空。

错误返回结果

若调用失败，其返回值示例如下为：

```
{
    "Response": {
        "Error": {
            "Code": "AuthFailure.SignatureFailure",
            "Message": "The provided credentials could not be validated. Please check your signature is cor
rect."
        },
        "RequestId": "ed93f3cb-f35e-473f-b9f3-0d451b8b79c6"
    }
}
```

- Error 的出现代表着该请求调用失败。Error 字段连同其内部的 Code 和 Message 字段在调用失败时是必定返回的。
- Code 表示具体出错的错误码，当请求出错时可以先根据该错误码在公共错误码和当前接口对应的错误码列表里面查找对应原因和解决方案。

- Message 显示出了这个错误发生的具体原因，随着业务发展或体验优化，此文本可能会经常保持变更或更新，用户不应依赖这个返回值。
- RequestId 用于一个 API 请求的唯一标识，如果 API 出现异常，可以联系我们，并提供该 ID 来解决问题。

公共错误码

返回结果中如果存在 Error 字段，则表示调用 API 接口失败。 Error 中的 Code 字段表示错误码，所有业务都可能出现的错误码为公共错误码，下表列出了公共错误码。

错误码	错误描述
AuthFailure.InvalidSecretId	密钥非法（不是云 API 密钥类型）。
AuthFailure.MFAFailure	MFA 错误。
AuthFailure.SecretIdNotFound	密钥不存在。
AuthFailure.SignatureExpire	签名过期。
AuthFailure.SignatureFailure	签名错误。
AuthFailure.TokenFailure	token 错误。
AuthFailure.UnauthorizedOperation	请求未 CAM 授权。
DryRunOperation	DryRun 操作，代表请求将会是成功的，只是多传了 DryRun 参数。
FailedOperation	操作失败。
InternalError	内部错误。
InvalidAction	接口不存在。
InvalidParameter	参数错误。
InvalidParameterValue	参数取值错误。
LimitExceeded	超过配额限制。
MissingParameter	缺少参数错误。
NoSuchVersion	接口版本不存在。
RequestLimitExceeded	请求的次数超过了频率限制。
ResourceInUse	资源被占用。
ResourceInsufficient	资源不足。

错误码	错误描述
ResourceNotFound	资源不存在。
ResourceUnavailable	资源不可用。
UnauthorizedOperation	未授权操作。
UnknownParameter	未知参数错误。
UnsupportedOperation	操作不支持。
UnsupportedProtocol	http(s)请求协议错误，只支持 GET 和 POST 请求。
UnsupportedRegion	接口不支持所传地域。

公共参数

公共参数是用于标识用户和接口鉴权目的的参数，如非必要，在每个接口单独的接口文档中不再对这些参数进行说明，但每次请求均需要携带这些参数，才能正常发起请求。

签名方法 v3

使用 TC3-HMAC-SHA256 签名方法时，公共参数需要统一放到 HTTP Header 请求头部中，如下：

参数名称	类型	必选	描述
X-TC-Action	String	是	操作的接口名称。取值参考接口文档中输入参数公共参数 Action 的说明。例如云服务器的查询实例列表接口，取值为 DescribeInstances。
X-TC-Region	String	是	地域参数，用来标识希望操作哪个地域的数据。接口接受的地域取值参考接口文档中输入参数公共参数 Region 的说明。注意：某些接口不需要传递该参数，接口文档中会对此特别说明，此时即使传递该参数也不会生效。
X-TC-Timestamp	Integer	是	当前 UNIX 时间戳，可记录发起 API 请求的时间。例如 1529223702。注意：如果与服务器时间相差超过5分钟，会引起签名过期错误。
X-TC-Version	String	是	操作的 API 的版本。取值参考接口文档中入参公共参数 Version 的说明。例如云服务器的版本 2017-03-12。
Authorization	String	是	HTTP 标准身份认证头部字段，例如： TC3-HMAC-SHA256 Credential=AKIDEXAMPLE/Date/service/tc3_request, SignedHeaders=content-type;host, Signature=fe5f80f77d5fa3beca038a248ff027d0445342fe2855ddc963176630326f1024 其中， - TC3-HMAC-SHA256：签名方法，目前固定取该值； - Credential：签名凭证，AKIDEXAMPLE 是 SecretId；Date 是 UTC 标准时间的日期，取值需要和公共参数 X-TC-Timestamp 换算的 UTC 标准时间日期一致；service为产品名，必须与调用的产品域名一致，例如cvm； - SignedHeaders：参与签名计算的头部信息，content-type 和 host 为必选头部； - Signature：签名摘要。
X-TC-Token	String	否	临时证书所用的 Token ，需要结合临时密钥一起使用。临时密钥和 Token 需要到访问管理服务调用接口获取。长期密钥不需要 Token。

签名方法 v1

使用 HmacSHA1 和 HmacSHA256 签名方法时，公共参数需要统一放到请求串中，如下

参数名称	类型	必选	描述
Action	String	是	操作的接口名称。取值参考接口文档中输入参数公共参数 Action 的说明。例如云服务器的查询实例列表接口，取值为 DescribeInstances。
Region	String	是	地域参数，用来标识希望操作哪个地域的数据。接口接受的地域取值参考接口文档中输入参数公共参数 Region 的说明。注意：某些接口不需要传递该参数，接口文档中会对此特别说明，此时即使传递该参数也不会生效。

参数名称	类型	必选	描述
Timestamp	Integer	是	当前 UNIX 时间戳，可记录发起 API 请求的时间。例如1529223702，如果与当前时间相差过大，会引起签名过期错误。
Nonce	Integer	是	随机正整数，与 Timestamp 联合起来，用于防止重放攻击。
SecretId	String	是	在云API密钥上申请的标识身份的 SecretId，一个 SecretId 对应唯一的 SecretKey ，而 SecretKey 会用来生成请求签名 Signature。
Signature	String	是	请求签名，用来验证此次请求的合法性，需要用户根据实际的输入参数计算得出。具体计算方法参见接口鉴权文档。
Version	String	是	操作的 API 的版本。取值参考接口文档中入参公共参数 Version 的说明。例如云服务器的版本 2017-03-12。
SignatureMethod	String	否	签名方式，目前支持 HmacSHA256 和 HmacSHA1。只有指定此参数为 HmacSHA256 时，才使用 HmacSHA256 算法验证签名，其他情况均使用 HmacSHA1 验证签名。
Token	String	否	临时证书所用的 Token ，需要结合临时密钥一起使用。临时密钥和 Token 需要到访问管理服务调用接口获取。长期密钥不需要 Token 。

地域列表

地域（Region）是指物理的数据中心的地理区域。TCloudFinanceZone交付验证不同地域之间完全隔离，保证不同地域间最大程度的稳定性和容错性。为了降低访问时延、提高下载速度，建议您选择最靠近您客户的地域。

您可以通过 API接口 查询地域列表 查看完成的地域列表。

腾讯大数据套件
bms创建集群

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

创建TBDS集群，申请BMS机器，并在此机器上部署TBDS集群.

默认接口请求频率限制：20次/秒。

接口更新时间：2021-12-07 10:25:24。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：BmsCreateCluster
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
Name	是	否	String	集群的名称 示例值：
Creator	否	否	String	创建人 示例值：
CvmImageId	是	否	String	Docker镜像ID 示例值：
ZoneId	是	否	String	实例所属可用区ID 示例值：
VpcId	是	否	String	私有网络ID 示例值：

参数名称	必选	允许NULL	类型	描述
SubnetId	是	否	String	Bms子网ID 示例值：
Desc	是	否	String	集群描述 示例值：
MasterInstanceInfo	是	否	BmsInstanceInfo	master 节点参数 示例值：查看
PortalInstanceInfo	是	否	InstanceInfo	portal 节点参数 示例值：查看
WorkerInstanceInfo	是	否	BmsInstanceInfo	worker 节点参数 示例值：查看
CvmSubnetId	是	否	String	cvm的子网id 示例值：
OperatingSystem	是	否	String	操作系统名称，例如： Centos 7.5 for x86_64 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

根据实例名称获取bms实例信息

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

根据实例名称获取bms实例信息

默认接口请求频率限制：20次/秒。

接口更新时间：2021-11-15 16:12:56。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：BmsDescribeInstancesByName
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
InstanceNames	否	否	Array of String	实例名称数组 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

判断集群名是否存在

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

无

默认接口请求频率限制：20次/秒。

接口更新时间：2020-03-23 21:43:06。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：ClusterExists
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
Name	是	否	String	集群名称 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

创建集群

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

创建TBDS集群，申请CVM机器，并在此机器上部署TBDS集群

默认接口请求频率限制：20次/秒。

接口更新时间：2020-12-01 20:03:21。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：CreateCluster
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
Name	是	否	String	集群的名称 示例值：
Creator	否	否	String	创建人 示例值：
ImageId	是	否	String	Docker镜像ID 示例值：
ZoneId	是	否	String	实例所属可用区ID 示例值：
VpcId	是	否	String	私有网络ID 示例值：
SubnetId	是	否	String	子网ID 示例值：

参数名称	必选	允许NULL	类型	描述
Desc	是	否	String	集群描述 示例值：
MasterInstanceInfo	否	否	InstanceInfo	master 节点参数 示例值：查看
PortalInstanceInfo	是	否	InstanceInfo	portal 节点参数 示例值：查看
WorkerInstanceInfo	否	否	InstanceInfo	worker 节点参数 示例值：查看
DisasterRecoverGroupIds	否	否	Array of String	置放群组id，仅支持指定一个。 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

获取集群详情

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

获取集群详情

默认接口请求频率限制：20次/秒。

接口更新时间：2020-09-07 19:44:50。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：DescribeClusterDetail
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
ClusterId	是	否	String	集群id 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

获取集群节点

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

获取集群节点.

默认接口请求频率限制：20次/秒。

接口更新时间：2020-09-09 11:24:54。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：DescribeClusterNodes
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
ClusterId	是	否	String	集群id 示例值：
PageIndex	否	否	Uint64	从0开始的页面index 示例值：
PageSize	否	否	Uint64	页面大小 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

销毁集群

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

销毁集群

默认接口请求频率限制：20次/秒。

接口更新时间：2020-09-07 15:49:00。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：DestroyCluster
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
ClusterId	是	否	String	集群id 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

获得集群列表

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

无

默认接口请求频率限制：20次/秒。

接口更新时间：2021-12-08 10:41:45。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：GetClusters
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
PageIndex	否	否	Uint64	从0开始的页面index 示例值：
PageSize	否	否	Uint64	页面大小 示例值：
Keyword	否	否	String	关键字，支持
Type	否	否	String	类型，填充“cvm”或“bms” 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

获得集群部署进度

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

无

默认接口请求频率限制：20次/秒。

接口更新时间：2020-03-02 10:56:35。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：GetProgress
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
ClusterId	是	否	String	集群ID 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

bms创建实例接口测试

1. 接口描述

接口请求域名： tbds.api3.finance.cloud.tencent.com。

bms创建实例接口测试

默认接口请求频率限制：20次/秒。

接口更新时间：2021-11-23 14:35:40。

接口只验签名不鉴权。

2. 输入参数

以下请求参数列表仅列出了接口请求参数和部分公共参数，完整公共参数列表见公共请求参数。

参数名称	必选	允许NULL	类型	描述
Action	是	否	String	公共参数，本接口取值：TestApplyBms
Version	是	否	String	公共参数，本接口取值：2020-01-16
Region	是	否	String	公共参数，地域信息本接口不需要传递此参数。
Name	是	否	String	集群的名称 示例值：
Creator	否	否	String	创建人 示例值：
ZoneId	是	否	String	实例所属可用区ID 示例值：
VpcId	是	否	String	私有网络ID 示例值：
SubnetId	是	否	String	子网ID 示例值：

参数名称	必选	允许NULL	类型	描述
Desc	是	否	String	集群描述 示例值：
Password	否	否	String	主机密码 示例值：
InstanceName	否	否	String	实例名称 示例值：
HostName	否	否	String	主机名称 示例值：
PrivateIpAddresses	否	否	Array of String	服务器内网ip数组 示例值：
InternetAccessible	否	否	InternetAccessible	公网带宽信息 示例值：查看
InstanceCount	否	否	Uint64	实例个数 示例值：
FlavorId	否	否	String	套餐id 示例值：
OperatingSystemType	否	否	String	bms,系统类型，例如：Linux 示例值：
OperatingSystem	否	否	String	bms,系统名称 示例值：
EnhancedService	否	否	Bool	bms,开启主机安全服务 示例值：
RaidType	否	否	String	RAID类型存储类型 示例值：

3. 输出参数

参数名称	类型	描述
RequestId	String	唯一请求 ID，每次请求都会返回。定位问题时需要提供该次请求的 RequestId。

4. 错误码

该接口暂无业务逻辑相关的错误码，其他错误码详见公共错误码。

数据结构

BmsInstanceInfo

bms 实例信息

被如下接口引用：BmsCreateCluster

名称	必选	允许NULL	类型	描述
FlavorId	是	否	String	bms机型 示例值：
InstanceCount	是	否	Uint64	要申请创建的BMS实例个数 示例值：
RaidType	是	否	String	RAID类型存储类型 示例值：

InstanceInfo

CVM实例信息

被如下接口引用：BmsCreateCluster、CreateCluster

名称	必选	允许NULL	类型	描述
InstanceType	是	否	String	CVM实例类型 示例值：
InstanceCount	是	否	Uint64	要申请创建的CVM实例个数 示例值：
DiskType	是	否	String	系统盘类型 示例值：
SystemDiskSize	是	否	Uint64	系统盘大小，单位：GB。默认值为 50 示例值：
LocalStorageDiskCount	否	否	Uint64	本地磁盘的个数 示例值：

名称	必选	允许NULL	类型	描述
DataDisks	否	否	Array of DataDisk	数据盘 示例值：查看
DisasterRecoverGroupIds	否	否	Array of String	置放群组id，仅支持指定一个。 示例值：

DataDisk

数据盘对象

被如下接口引用：BmsCreateCluster、CreateCluster

名称	必选	允许NULL	类型	描述
DiskType	是	否	String	数据盘类型 示例值：
DiskSize	是	否	Uint64	数据盘大小，单位：GB。最小调整步长为10G，不同数据盘类型取值范围不同，具体限制详见：CVM实例配置。默认值为0，表示不购买数据盘。更多限制详见产品文档 示例值：
DeleteWithInstance	是	否	Bool	数据盘是否随子机销毁。TRUE：子机销毁时，销毁数据盘，只支持按小时后付费云盘。FALSE：子机销毁时，保留数据盘。默认取值：TRUE 示例值：

InternetAccessible

公网带宽信息

被如下接口引用：TestApplyBms

名称	必选	允许NULL	类型	描述
PublicIpAssigned	否	否	Bool	是否分配公网ip 示例值：
InternetMaxBandwidthOut	否	否	String	带宽大小 示例值：

名称	必选	允许NULL	类型	描述
InternetServiceProvider	否	否	String	外网供应商 示例值：

错误码

功能说明

如果返回结果中存在 Error 字段，则表示调用 API 接口失败。例如：

```
{
    "Response": {
        "Error": {
            "Code": "AuthFailure.SignatureFailure",
            "Message": "The provided credentials could not be validated. Please check your signature is cor
rect."
        },
        "RequestId": "ed93f3cb-f35e-473f-b9f3-0d451b8b79c6"
    }
}
```

Error 中的 Code 表示错误码，Message 表示该错误的具体信息。

错误码列表

公共错误码

错误码	说明
AuthFailure.InvalidSecretId	密钥非法（不是云 API 密钥类型）。
AuthFailure.MFAFailure	MFA 错误。
AuthFailure.SecretIdNotFound	密钥不存在。 请在控制台检查密钥是否已被删除或者禁用，如状态正常，请检查密钥是否填写正确，注意前后不得有空格。
AuthFailure.SignatureExpire	签名过期。Timestamp 和服务器时间相差不得超过五分钟，请检查本地时间是否和标准时间同步。
AuthFailure.SignatureFailure	签名错误。 签名计算错误，请对照调用方式中的接口鉴权文档检查签名计算过程。
AuthFailure.TokenFailure	token 错误。
AuthFailure.UnauthorizedOperation	请求未 CAM 授权。
DryRunOperation	DryRun 操作，代表请求将会是成功的，只是多传了 DryRun 参数。

错误码	说明
FailedOperation	操作失败。
InternalError	内部错误。
InvalidAction	接口不存在。
InvalidParameter	参数错误。
InvalidParameterValue	参数取值错误。
LimitExceeded	超过配额限制。
MissingParameter	缺少参数错误。
NoSuchVersion	接口版本不存在。
RequestLimitExceeded	请求的次数超过了频率限制。
ResourceInUse	资源被占用。
ResourceInsufficient	资源不足。
ResourceNotFound	资源不存在。
ResourceUnavailable	资源不可用。
UnauthorizedOperation	未授权操作。
UnknownParameter	未知参数错误。
UnsupportedOperation	操作不支持。
UnsupportedProtocol	http(s)请求协议错误，只支持 GET 和 POST 请求。
UnsupportedRegion	接口不支持所传地域。

业务错误码