阿里云
E-MapReduce

开源组件介绍

文档版本：20200709

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。
5. 阿里云文档中所有内容，包括但不限于图片、架构设计、页面布局、文字描述，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。
6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **禁止：** 重置操作将丢失用户配置数据。
	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **警告：** 重启操作将导致业务中断，恢复业务时间约十分钟。
	用于警示信息、补充说明等，是用户必须了解的内容。	 **注意：** 权重设置为0，该服务器不会再接受新请求。
	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	 **说明：** 您也可以通过按Ctrl + A选中全部文件。
>	多级菜单递进。	单击**设置** > **网络** > **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令。	执行cd /d C:/window命令，进入Windows系统文件夹。
斜体	表示参数、变量。	bae log list --instanceid Instance_ID
[]或者[a|b]	表示可选项，至多选择一个。	ipconfig [-all|-t]
{}或者{a|b}	表示必选项，至多选择一个。	switch {active|stand}

目录

1 Kudu使用说明

本文主要介绍Kudu的一些典型应用场景及其架构等。

概述

Kudu产生主要是为了填补Hadoop生态圈的功能空白，用户可能存在以下的应用场景：

- 需要利用HBase的快速写入和随机读取来导入数据，HBase也允许用户进行数据的修改。
- 使用HDFS/Parquet + hive/spark/impala来进行超大规模数据集的查询以及分析，在这种情况下Parquet列式存储有很大的优势。

Kudu可以同时支持以上两种不同场景，避免用户同时部署HDFS/Parquet以及HBase的集群，减少客户的运维的成本。

典型的应用场景

- 近实时计算场景

  流式计算场景通常会有持续不断的、大量写入数据的操作，与此同时这些数据还要支持近乎实时的读、写以及更新的操作。Kudu的设计能够很好的处理此场景。
- 时间序列数据的场景

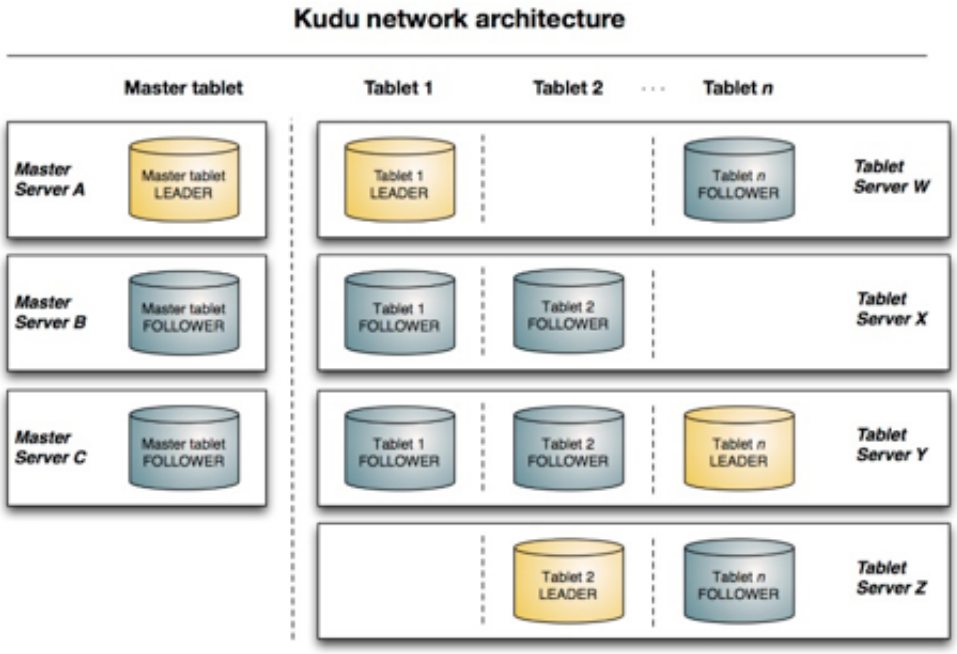
  时间序列数据是随着时间产生的，用户可以根据这些数据调查性能指标以及基于时间序列数据进行预测。例如用户以时间序列的方式存储客户的购买点击历史，用来预计未来的购买，或者这部分数据有客户代表使用，用于分析用户的购买行为，在分析的同时，客户可能会产生更多的数据或者修改已有的数据，而这些数据可以很快为分析所用， Kudu可以用可扩展的和高效的方式同时处理这些数据访问模式。
- 预测建模

  数据科学家经常需要通过大量数据建立预测模型，但是模型以及模型所依赖的数据处于经常的变化当中，如果数据或者模型存储在HDFS， 频繁的更新HDFS是非常耗时的， 而Kudu的设计可以很好满足这样的场景，数据科学家只要更改数据，重新运行查询，就可以在秒级或者分钟级别更新模型，而不是要花费几个小时或者一天的时间去得到更新的模型。
- 与存量数据共存

  用户生产环境中往往有大量的存量数据。可能存储在HDF、RDBMS或者Kudu中，Impala可以同时支持HDFS、Kudu等多个底层存储引擎，这个特性使得在使用的Kudu的同时，不必把所有的数据都迁移到Kudu中。

基本架构

Kudu有两种类型的组件，Master Server和Tablet Server。Master Server负责管理元数据，这些元数据包括Tablet Server的服务器的信息以及Tablet的信息，Master Server通过Raft协议提供高可用性；Tablet Server用来存储tablets，每个tablet存在多个副本存放在不同的tablet server中， 副本之间通过Raft协议提供高可用性，下图为Kudu集群的基本架构的示意图。



创建集群

E-MapReduce从E-MapReduce-3.22.0版本开始支持Kudu集群的创建，主要作为Hadoop集群类型的一个服务组件，用户在创建Hadoop集群时勾选Kudu组件就会创建Kudu集群，默认情况下Kudu集群包含3个Kudu Master服务并提供HA支持。

Impala集成

Impala集成Kudu不需要做特别的配置，可以在Impala配置文件中设置kudu_master_hosts = [master1][:port],[master2][:port],[master3][:port]；或者在SQL中通过TBLPROPERTIES语句指定kudu.master_addresses来指定Kudu集群。下面示例展示的是在SQL语句中添加kudu.master_addresses来指定Kudu的地址。

```
CREATE TABLE my_first_table
  (
    id BIGINT,
    name STRING,
    PRIMARY KEY(id)
  )
  PARTITION BY HASH PARTITIONS 16
  STORED AS KUDU
  TBLPROPERTIES(
    'kudu.master_addresses' = 'master1:7051,master2:7051,master3:7051');
```

常用命令

- 检查集群健康

  ```
  kudu cluster ksck <master_addresses>
  ```

- 检查集群Metrics

  ```
  kudu-tserver --dump_metrics_json
  ```

```
kudu-master --dump_metrics_json
```

- 获取tables

```
kudu table list <master_addresses>
```

- Dump出cfile文件内容

```
kudu fs dump cfile <block_id>
```

- Dump出kudu文件系统的tree

```
kudu fs dump tree [-fs_wal_dir=<dir>] [-fs_data_dirs=<dirs>]
```

2 Oozie使用说明

本文介绍如何在E-MapReduce上使用Oozie。

> **说明:**
> 阿里云E-MapReduce在2.0.0及之后的版本中提供了对Oozie的支持，如果需要在集群中使用Oozie，请确认集群的版本不低于2.0.0。

准备工作

在集群建立出来之后，需要打通SSH隧道，详细步骤请参见#unique_5。

这里以MAC环境为例，使用Chrome浏览器实现端口转发（假设集群Master节点公网IP为**xx.xx.xx.xx**）：

1. 登录到Master节点。

```
ssh root@xx.xx.xx.xx
```

2. 输入密码。
3. 查看本机的**id_rsa.pub**。

```
cat ~/.ssh/id_rsa.pub
```

4. 将本机的**id_rsa.pub**内容写入到远程Master节点的~/.ssh/authorized_keys中。

```
mkdir ~/.ssh/
vim ~/.ssh/authorized_keys
```

5. 然后将步骤 3中看到的内容粘贴进来，现在应该可以直接使用ssh root@xx.xx.xx.xx免密登录Master节点了。
6. 在本机执行以下命令进行端口转发。

```
ssh -i ~/.ssh/id_rsa -ND 8157 root@xx.xx.xx.xx
```

7. 启动Chrome（在本机新开terminal执行）。

```
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --proxy-server
="socks5://localhost:8157" --host-resolver-rules="MAP * 0.0.0.0 , EXCLUDE localhost"
 --user-data-dir=/tmp
```

访问Oozie UI页面

在进行端口转发的Chrome浏览器中访问：xx.xx.xx.xx:11000/oozie、localhost:11000/oozie或者内网ip:11000/oozie。

提交Workflow作业

运行Oozie需要先安装Oozie的shareLib。

在E-MapReduce集群中，默认给Oozie用户安装了sharelib，即如果使用Oozie用户来提交workflow作业，则不需要再进行sharelib的安装。

由于开启HA的集群和没有开启HA的集群，访问NameNode和ResourceManager的方式不同，在提交oozie workflow job的时候，job.properties文件中需要指定不同的NameNode和JobTracker（ResourceManager）。具体如下：

- 非HA集群

```
nameNode=hdfs://emr-header-1:9000
jobTracker=emr-header-1:8032
```

- HA集群

```
nameNode=hdfs://emr-cluster
jobTracker=rm1,rm2
```

下面操作示例中，已经针对是否是HA集群配置好了，即样例代码不需要任何修改即可以直接运行。

关于workflow文件的具体格式，请参见 Oozie官方文档。

- 在非HA集群上提交workflow作业

1. 登录集群的主Master节点。

```
ssh root@master公网Ip
```

2. 下载示例代码。

```
[root@emr-header-1 ~]# su oozie
[oozie@emr-header-1 root]$ cd /tmp
[oozie@emr-header-1 tmp]$ wget http://emr-sample-projects.oss-cn-hangzhou.
aliyuncs.com/oozie-examples/oozie-examples.zip
[oozie@emr-header-1 tmp]$ unzip oozie-examples.zip
```

3. 将Oozie workflow代码同步到HDFS上。

```
[oozie@emr-header-1 tmp]$ hadoop fs -copyFromLocal examples/ /user/oozie/
examples
```

4. 提交Oozie workflow样例作业。

```
[oozie@emr-header-1 tmp]$ $OOZIE_HOME/bin/oozie job -config examples/apps/
map-reduce/job.properties -run
```

执行成功之后，会返回一个jobId，类似如下信息。

```
job: 0000000-160627195651086-oozie-oozi-W
```

5. 访问Oozie UI页面，可以看到刚刚提交的Oozie workflow job。

- 在HA集群上提交workflow作业

  1. 登录集群的主Master节点。

     ```
     ssh root@主master公网Ip
     ```

     可以通过是否能访问Oozie UI来判断哪个Master节点是当前的主Master节点， Oozie server服务默认是启动在主Master节点xx.xx.xx.xx:11000/oozie。

  2. 下载HA集群的示例代码。

     ```
     [root@emr-header-1 ~]# su oozie
     [oozie@emr-header-1 root]$ cd /tmp
     [oozie@emr-header-1 tmp]$ wget http://emr-sample-projects.oss-cn-hangzhou.aliyuncs.com/oozie-examples/oozie-examples-ha.zip
     [oozie@emr-header-1 tmp]$ unzip oozie-examples-ha.zip
     ```

  3. 将Oozie workflow代码同步到HDFS上。

     ```
     [oozie@emr-header-1 tmp]$ hadoop fs -copyFromLocal examples/ /user/oozie/examples
     ```

  4. 提交Oozie workflow样例作业。

     ```
     [oozie@emr-header-1 tmp]$ $OOZIE_HOME/bin/oozie job -config examples/apps/map-reduce/job.properties -run
     ```

     执行成功之后，会返回一个 jobId，类似如下信息。

     ```
     job: 0000000-160627195651086-oozie-oozi-W
     ```

  5. 访问Oozie UI页面，可以看到刚刚提交的Oozie workflow job。

3 Zeppelin使用说明

本节介绍阿里云E-MapReduce如何访问Zeppelin，通过访问Zeppelin进行大数据可视化分析。

前提条件

1. 在集群#unique_7中设置安全组规则，打开8080端口，详情请参见#unique_9。
2. 在Knox中，添加访问用户名和密码。详情请参见Knox使用说明，设置Knox用户。用户名和密码仅用于登录Knox的各项服务，与阿里云RAM的用户名无关。

访问Zeppelin

1. 已通过主账号登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 在左侧导航栏中单击**访问链接与端口**。
6. 单击Zeppelin所在行的链接，可直接访问Web UI页面。

问题反馈

如果您在使用阿里云E-MapReduce过程中有任何疑问，欢迎您扫描下面的二维码加入钉钉群进行反馈。

4 ZooKeeper使用说明

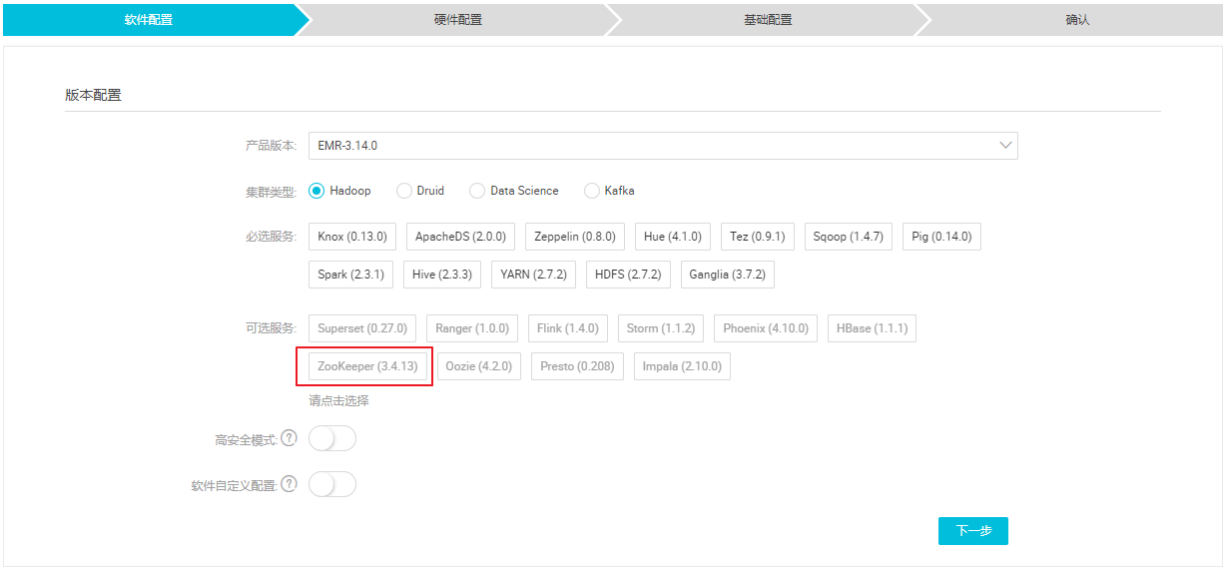
目前E-MapReduce集群中默认启动了ZooKeeper服务。

背景信息

目前无论集群内有多少台机器，ZooKeeper只会有3个节点。目前还不支持更多的节点。

创建集群

E-MapReduce创建集群的软件配置页面，可选择ZooKeeper服务 。具体创建集群请参见#unique_12。



查看节点信息

1. 已通过主账号登录阿里云E-MapReduce控制台。
2. 单击上方的**集群管理**页签。
3. 在**集群管理**页面，单击集群右侧的**详情**。

> **说明:**
> 待创建集群的**状态**为**空闲**时，才可以单击集群所在行**详情**。

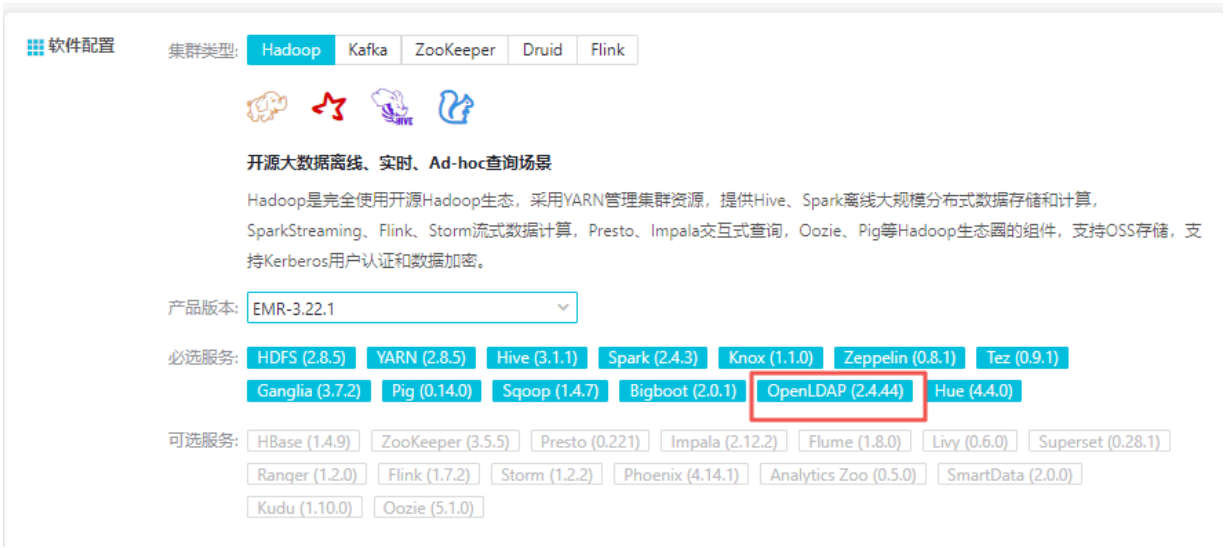
4. 单击左侧导航栏的**集群服务** > **ZooKeeper**。
5. 单击**部署拓扑**，可以查看ZooKeeper的节点信息，E-MapReduce会启动3个ZooKeeper节点。

5 OpenLDAP使用说明

从EMR-3.22.0版本开始，E-MapReduce（简称EMR）集群中默认启动了OpenLDAP服务。默认的，Knox与OpenLDAP集成，可以通过OpenLDAP管理Knox用户信息。从EMR-3.24.0版本开始，高安全集群Has服务以OpenLDAP作为后端，可以通过OpenLDAP管理HAS的principal。

创建集群

E-MapReduce创建集群的软件配置页面，OpenLDAP作为必选服务。创建集群的详细步骤，请参见#unique_12。



查看节点信息

1. 已通过主账号登录阿里云E-MapReduce控制台。
2. 单击上方的**集群管理**。
3. 在**集群管理**页面，单击集群右侧的**详情**。

> **说明:**
> 待创建集群的**状态**为**空闲**时，才可以单击集群所在行**详情**。

4. 单击左侧导航栏的**集群服务** > **OpenLDAP**。
5. 单击**部署拓扑**。

   可查看OpenLDAP的节点信息，OpenLDAP部署在Master节点。如果是高可用集群，OpenLDAP部署在两个Master节点，具备高可用性。

修改OpenLDAP信息

- 方式一：在E-MapReduce控制台修改集群中的OpenLDAP信息。

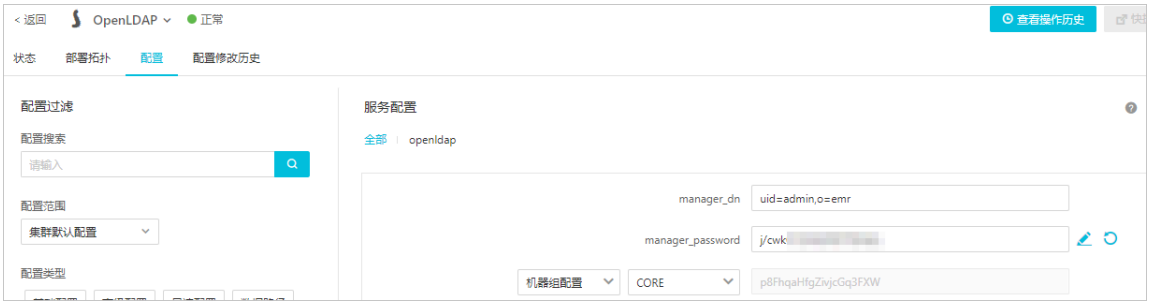
  在**用户管理**页面，通过设置Knox账户，添加或删除集群中的LDAP信息，详细信息请参见#unique_14。

  > **说明：**
  > 若集群没有**用户管理**功能，请参见方式二修改OpenLDAP信息。

- 方式二：通过ldap命令修改集群中的OpenLDAP信息。

  例如，添加uid为arch，密码为12345678的LDAP信息。

  1. 在E-MapReduce控制台上，通过OpenLDAP的服务配置获取root dn和密码。

  2. 编辑arch.ldif文件。

     添加LDAP信息。

     ```
     dn: uid=arch,ou=people,o=emr
     cn: arch
     sn: arch
     objectClass: inetOrgPerson
     userPassword: 12345678
     uid: arch
     ```

  3. 登录集群的Master节点，执行如下命令即可。

     ```
     ldapadd -H ldap://emr-header-1:10389 -f arch.ldif -D uid=admin,o=emr -w ${
     rootDnPW}
     ```

     > **说明：**
     > - ${rootDnPW}为root dn的密码。

- 10389为OpenLDAP服务的监听端口。

4. 执行成功后，可以查看到该LDAP信息。

```
ldapsearch -w ${rootDnPW}  -D "uid=admin,o=emr" -H ldap://emr-header-1:10389
 -b uid=arch,ou=people,o=emr
```

如果要删除添加的LDAP信息时，请执行以下命令。

```
ldapdelete -x -D "uid=admin,o=emr" -w ${rootDnPW} -r uid=arch,ou=people,o=emr
 -H ldap://emr-header-1:10389
```

6 Sqoop使用说明

Sqoop是一款用来在不同数据存储软件之间进行数据传输的开源软件，它支持多种类型的数据储存软件。

安装Sqoop

> **注意：**
> 目前，E-MapReduce从版本1.3开始都会默认支持Sqoop组件，您无需再自行安装，可以跳过本节。

若您使用的E-MapReduce的版本低于1.3，则还没有集成Sqoop，如果需要使用请参考如下的安装方式。

1. 从官网下载Sqoop 1.4.6版本。

   若下载的sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz无法打开，也可以使用镜像地址。http://mirror.bit.edu.cn/apache/sqoop/1.4.6/sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz进行下载，请执行如下命令。

   ```
   wget http://mirror.bit.edu.cn/apache/sqoop/1.4.6/sqoop-1.4.6.bin__ha doop-2.0.4-alpha.tar.gz
   ```

2. 执行如下命令，将下载下来的sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz解压到Master节点上。

   ```
   tar zxf sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz
   ```

3. 要从MySQL导数据需要安装MySQL driver。请从官网下载最新版本，或执行如下命令进行下载（此处以版本5.1.38为例）。

   ```
   wget https://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-5.1.38.tar.gz
   ```

4. 解压后将jar包放到Sqoop目录下的lib目录下。

数据传输

常见的使用场景如下所示：

- MySQL -> HDFS
- HDFS -> MySQL
- MySQL -> Hive
- Hive -> MySQL

- 使用SQL作为导入条件

> **注意:**
> 在执行如下的命令前，请先切换您的用户为Hadoop。

```
su hadoop
```

- 从MySQL到HDFS。

  在集群的Master节点上执行如下命令。

```
sqoop import --connect jdbc:mysql://<dburi>/<dbname> --username <username>
 --password <password> --table <tablename> --check-column <col> --incremental <
mode> --last-value <value> --target-dir <hdfs-dir>
```

  参数说明：

  - dburi：数据库的访问链接，例如： jdbc:mysql://192.168.1.124:3306/ 如果您的访问链接中含有参数，那么请用单引号将整个链接包裹住，例如 ： jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true。
  - dbname：数据库的名字，例如：user。
  - username：数据库登录用户名。
  - password：用户对应的密码。
  - tablename：MySQL表的名字。
  - col：要检查的列的名称。
  - mode：该模式决定Sqoop如何定义哪些行为新的行。取值为append或lastmodified。
  - value：前一个导入中检查列的最大值。
  - hdfs-dir：HDFS 的写入目录，例如：/user/hive/result。

  更加详细的参数使用请参见Sqoop Import。

- 从HDFS到MySQL。

  需要先创建好对应HDFS中的数据结构的MySQL表，然后在集群的Master节点上执行如下命令，指定要导的数据文件的路径。

  ```
  sqoop export --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --table <tablename> --export-dir <hdfs-dir>
  ```

  参数说明：

  - dburi：数据库的访问链接，例如： jdbc:mysql://192.168.1.124:3306/。如果您的访问链接中含有参数，那么请用单引号将整个链接包裹住，例如：jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true。
  - dbname：数据库的名字，例如：user。
  - username：数据库登录用户名。
  - password：用户对应的密码。
  - tablename：MySQL的表的名字。
  - hdfs-dir：要导到MySQL去的HDFS的数据目录，例如：/user/hive/result。

  更加详细的参数使用请参见Sqoop Export。

- 从MySQL到Hive。

  在集群的Master节点上执行如下命令后，从MySQL数据库导入数据的同时，也会新建一个Hive表。

  ```
  sqoop import --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --table <tablename> --check-column <col> --incremental <
  ```

```
mode> --last-value <value> --fields-terminated-by "\t" --lines-terminated-by "\n" --
hive-import --target-dir <hdfs-dir> --hive-table <hive-tablename>
```

参数说明：

- dburi：数据库的访问链接，例如： jdbc:mysql://192.168.1.124:3306/ 如果您的访问链接中含有参数，那么请用单引号将整个链接包裹住，例如：jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true。
- dbname：数据库的名字，例如：user。
- username：据库登录用户名。
- password：用户对应的密码。
- tablename：MySQL的表的名字。
- col：要检查的列的名称。
- mode：该模式决定Sqoop如何定义哪些行为新的行。取值为append或lastmodified。由Sqoop导入数据到Hive不支持append模式。
- value：前一个导入中检查列的最大值。
- hdfs-dir：要导到MySQL去的HDFS的数据目录，例如：/user/hive/result。
- hive-tablename：对应的Hive中的表名，可以是xxx.yyy。

更加详细的参数使用请参见Sqoop Import。

- 从Hive到MySQL

  请参考上面的从HDFS到MySQL的命令，只需要指定Hive表对应的HDFS路径就可以了。

- 从MySQL到OSS

  类似从MySQL到HDFS，只是—target-dir不同。在集群的Master节点上执行如下命令：

```
sqoop import --connect jdbc:mysql://<dburi>/<dbname> --username <username>
 --password <password> --table <tablename> --check-column <col> --incremental <
mode> --last-value <value> --target-dir <oss-dir> --temporary-rootdir <oss-tmpdir>
```

注意：

- OSS地址中的host有内网地址、外网地址和VPC网络地址之分。如果用经典网络，需要指定内网地址，杭州是oss-cn-hangzhou-internal.aliyuncs.com，VPC要指定VPC内网，杭州是vpc100-oss-cn-hangzhou.aliyuncs.com。

- 目前同步到OSS不支持—delete-target-dir，用这个参数会报错Wrong FS。如果要覆盖以前目录的数据，可以在调用sqoop前用hadoop fs -rm -r osspath先把原来的OSS目录删了。

```
sqoop import --connect jdbc:mysql://<dburi>/<dbname> --username <username>
 --password <password> --table <tablename> --check-column <col> --incremental <
mode> --last-value <value> --target-dir <oss-dir> --temporary-rootdir <oss-tmpdir>
```

参数说明：

- dburi：数据库的访问链接，例如： jdbc:mysql://192.168.1.124:3306/如果您的访问链接中含有参数，那么请用单引号将整个链接包裹住，例如：jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true。
- dbname：数据库的名字，例如：user。
- username：数据库登录用户名。
- password：用户对应的密码。
- tablename：MySQL表的名字。
- col：要检查的列的名称。
- mode：该模式决定Sqoop如何定义哪些行为新的行。取值为append或lastmodified。
- value：前一个导入中检查列的最大值。
- oss-dir：OSS的写入目录，例如：oss://<accessid>:<accesskey>@<bucketname>.oss-cn-hangzhou-internal.aliyuncs.com/result。
- oss-tmpdir：临时写入目录。指定append模式的同时，需要指定该参数。如果目标目录已经存在于HDFS中，则Sqoop将拒绝导入并覆盖该目录的内容。采用append模式后，Sqoop会将数据导入临时目录，然后将文件重命名为正常目标目录。

更加详细的参数使用请参见 Sqoop Import。

- 从OSS到MySQL

  类似MySQL到HDFS，只是—export-dir 不同。需要创建好对应OSS中的数据结构的MySQL表。

  然后在集群的Master节点上执行如下：指定要导的数据文件的路径。

  ```
  sqoop export --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --table <tablename> --export-dir <oss-dir>
  ```

  参数说明：

  - dburi：数据库的访问链接，例如： jdbc:mysql://192.168.1.124:3306/如果您的访问链接中含有参数，那么请用单引号将整个链接包裹住，例如：jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true。
  - dbname：数据库的名字，例如：user。
  - username：数据库登录用户名。
  - password：用户对应的密码。
  - tablename：MySQL表的名字。
  - oss-dir：OSS的写入目录，例如：oss://<accessid>:<accesskey>@<bucketname>.oss-cn-hangzhou-internal.aliyuncs.com/result。
  - oss-tmpdir：临时写入目录。指定append模式的同时，需要指定该参数。如果目标目录已经存在于HDFS中，则Sqoop将拒绝导入并覆盖该目录的内容。采用append模式后，Sqoop会将数据导入临时目录，然后将文件重命名为正常目标目录。

  说明：
  OSS地址host有内网地址，外网地址，VPC网络地址之分。如果用经典网络，需要指定内网地址，杭州是oss-cn-hangzhou-internal.aliyuncs.com，VPC需要指定VPC内网，杭州是vpc100-oss-cn-hangzhou.aliyuncs.com。

  更加详细的参数使用请参见Sqoop Export。

- 使用SQL作为导入条件

  除了指定MySQL的全表导入，还可以写SQL来指定导入的数据，如下所示。

  ```
  sqoop import --connect jdbc:mysql://<dburi>/<dbname> --username <username> --password <password> --query <query-sql> --split-by <sp-column> --hive-import --hive-table <hive-tablename> --target-dir <hdfs-dir>
  ```

  参数说明：

  - dburi：数据库的访问链接，例如： jdbc:mysql://192.168.1.124:3306/如果您的访问链接中含有参数，那么请用单引号将整个链接包裹住，例如：jdbc:mysql://192.168.1.124:3306/mydatabase?useUnicode=true。
  - dbname：数据库的名字，例如：user。
  - username：数据库登录用户名。
  - password：用户对应的密码。
  - query-sql：使用的查询语句，例如：`SELECT * FROM profile WHERE id>1 AND \$CONDITIONS`。记得要用引号包围，最后一定要带上 `AND \$CONDITIONS`。
  - sp-column：进行切分的条件，一般跟MySQL表的主键有关。
  - hdfs-dir：要导到MySQL去的HDFS的数据目录，例如：/user/hive/result。
  - hive-tablename：对应的Hive中的表名，可以是xxx.yyy。

  更加详细的参数使用请参见Sqoop Query Import。

  集群和其他数据库的网络配置请参见#unique_16。

7 Knox使用说明

本文介绍如何在E-MapReduce上使用Knox。

背景信息

目前E-MapReduce中支持了Apache Knox，选择支持Knox的镜像创建集群，完成以下准备工作后，即可在公网直接访问YARN、HDFS、Spark History Server等服务的Web UI。

准备工作

- 开启Knox公网IP访问

  1. E-MapReduce上Knox的服务端口是8443，在集群基础信息中找到集群所在的ECS安全组。
  2. 在ECS控制台修改对应的安全组，在**入方向**添加一条规则，打开8443端口。

  > **注意:**
  > - 为了安全原因，这里设置的授权对象必须是您的一个有限的IP段范围，禁止使用0.0.0.0/0。
  > - 打开安全组的8443端口之后，该安全组内的所有机器均会打开公网入方向的8443端口，包括非E-MapReduce的ECS机器。
  > - 若集群在创建时，没有挂载公网IP。可以在ECS控制台为该ECS实例添加公网IP。添加成功后，返回EMR控制台，在集群管理下的**主机列表**页面，单击**同步主机信息**可以立即同步。
  > - Knox节点新挂载公网IP后，需要通过提交工单联系E-MapReduce产品团队，进行域名和公网IP的绑定操作。

- 设置Knox用户

  访问Knox时需要对身份进行验证，会要求您输入用户名和密码。Knox的用户身份验证基于LDAP，您可以使用自有LDAP服务，也可以使用集群中的Apache Directory Server的LDAP服务。

  - 使用集群中的LDAP服务

    方式一（推荐）

    在用户管理中直接添加Knox访问账号。

    方式二

    1. SSH登录到集群上，详细步骤请参见#unique_17。
    2. 准备您的用户数据，如：Tom。将文件中所有的**emr-guest**替换为Tom，将**cn:EMR GUEST**替换为cn:Tom，设置**userPassword**的值为您自己的密码。

    ```
    su knox
    cd /usr/lib/knox-current/templates
    ```

```
vi users.ldif
```

注意：

出于安全原因，导入前务必修改users.ldif的用户密码，即：设置**userPassword**的值为您自己的用户密码。

3. 导入到LDAP。

```
su knox
cd /usr/lib/knox-current/templates
sh ldap-sample-users.sh
```

- 使用自有LDAP服务的情况

1. 在集群配置管理中找到KNOX的配置管理，在cluster-topo配置中设置两个属性：**main.ldapRealm.userDnTemplate**与**main.ldapRealm.contextFactory.url**。**main.ldapRealm.userDnTemplate**设置为自己的用户DN模板、**main.ldapRealm.contextFactory.url**设置为自己的LDAP服务器域名和端口。设置完成后保存并重启Knox。

2. 一般自己的LDAP服务不在集群上运行，所以需要开启Knox访问公网LDAP服务的端口，如：10389。参见8443端口的开启步骤，选择**出方向**。

注意：

为了安全原因，这里设置的授权对象必须是您的Knox所在集群的公网IP，禁止使用0.0.0.0/0。

开始访问Knox

- 使用E-MapReduce快捷链接访问

  1. 登录阿里云 E-MapReduce 控制台。
  2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
  3. 单击上方的**集群管理**页签。
  4. 在**集群管理**页面，单击相应集群所在行的**详情**。
  5. 在左侧导航栏，单击**集群服务** > **HDFS**。

     选择相应的服务，如：HDFS、YARN等，本文以HDFS为例。
  6. 单击右上角HDFS的快捷链接。

- 使用集群公网IP地址访问

  1. 通过集群详情查看公网IP。
  2. 在浏览器中访问相应服务的 URL。
     - HDFS UI：https://{集群公网ip}:8443/gateway/cluster-topo/hdfs/。
     - Yarn UI：https://{集群公网ip}:8443/gateway/cluster-topo/yarn/。
     - SparkHistory UI：https://{集群公网ip}:8443/gateway/cluster-topo/sparkhistory/。
     - Ganglia UI：https://{集群公网ip}:8443/gateway/cluster-topo/ganglia/。
     - Storm UI：https://{集群公网ip}:8443/gateway/cluster-topo/storm/。
     - Oozie UI：https://{集群公网ip}:8443/gateway/cluster-topo/oozie/。
  3. 浏览器显示**您的链接不是私密链接**，是因为Knox服务使用了自签名证书，请再次确认访问的是自己集群的 IP、且端口为8443。单击**高级** > **继续前往**。
  4. 在登录框中输入您在LDAP中设置的用户名和密码。

用户权限管理（ACLs）

Knox提供服务级别的权限管理，可以限制特定的用户，特定的用户组和特定的IP地址访问特定的服务，可以参见Apache Knox授权。

示例：

- 场景：Yarn UI只允许用户Tom访问。

- 步骤：在集群配置管理中找到KNOX的配置管理，找到cluster-topo配置，在cluter-topo配置的<gateway>...</gateway>标签之间添加ACLs代码：

```
<provider>
    <role>authorization</role>
    <name>AclsAuthz</name>
    <enabled>true</enabled>
    <param>
      <name>YARNUI.acl</name>
      <value>Tom;*;*</value>
    </param>
</provider>
```

警告：

Knox会开放相应服务的REST API，用户可以通过各服务的REST API操作服务，如：HDFS的文件添加、删除等。出于安全原因，请务必确保在ECS控制台开启安全组Knox端口8443时，授权对象必须是您的一个有效IP地址段，禁止使用0.0.0.0/0；请勿使用Knox安装目录下的LDAP用户名和密码作为Knox的访问用户。

8 TensorFlow使用说明

E-MapReduce 3.13.0以后版本支持TensorFlow。您可以在软件配置可选服务中添加TensorFlow组件。当您使用E-MapReduce TensorFlow运行高性能计算时，可通过YARN对集群中的CPU和GPU进行调度。

准备工作

- 软件层面，E-MapReduce集群安装TensorFlow和TensorFlow on YARN组件。
- 硬件层面，E-MapReduce支持使用CPU和GPU两种资源进行计算。如您需使用GPU进行计算，可在 core 或 task 节点中选择ECS异构计算GPU计算型，如ecs.gn5、ecs.gn6机型。选定GPU机型后，选择您所需的CUDA和CuDNN版本。

提交TensorFlow作业

用户可以登录E-MapReduce主节点，以命令行的方式提交Tensorflow作业，例如：

```
el_submit [-h] [-t APP_TYPE] [-a APP_NAME] [-m MODE] [-m_arg MODE_ARG]
[-interact INTERACT] [-x EXIT]
[-enable_tensorboard ENABLE_TENSORBOARD]
[-log_tensorboard LOG_TENSORBOARD] [-conf CONF] [-f FILES]
[-pn PS_NUM] [-pc PS_CPU] [-pm PS_MEMORY] [-wn WORKER_NUM]
[-wc WORKER_CPU] [-wg WORKER_GPU] [-wm WORKER_MEMORY]
[-wnpg WNPG] [-ppn PPN] [-c COMMAND [COMMAND ...]]
```

基础参数说明：

参数	说明
-t APP_TYPE	提交的任务类型，支持三种类型的任务类型tensorflow-ps、 tensorflow-mpi、standalone三种类型要配合后面运行模式使用。 • tensorflow-ps模式，采用Parameter Server方式通信，该方式为原生TensorFlow PS模式。 • tensorflow-mpi模式，采用的是UBER开源的MPI架构horovod进行通信。 • standalone模式，是用户将任务调度到YARN集群中启动单机任务，类似于单机运行。

参数	说明
-a APP_NAME	指提交TensorFlow提交的作业名称，用户可以根据需要自行命名。
-m MODE	指TensorFlow作业提交的运行时环境，E-MapReduce支持三种类型运行环境local、virtual-env和docker。 • local模式，使用的是emr-worker上面的python运行环境，所以如果要使用一些第三方Python包需要手动在所有机器上进行安装。 • docker模式， 使用的是emr-worker上面的docker运行的，tensorflow运行在docker container内。 • virtual-env模式，用户上传的Python环境，可以在Python环境中安装一些不同于worker节点的Python库。
-m_arg MODE_ARG	提交运行模式的补充参数，和MODE配合使用，如果运行时是docker，则设置为docker镜像名称，如果是virtual-env，则指定Python环境文件名称，tar.gz 打包。
-x Exit	分布式TensorFlow有些API需要用户手动退出PS，在这种情况下用户可以指定-x选项，当所有worker完成训练并成功后，PS节点自动退出。
-enable_tensorboard	是否在启动训练任务的同时启动TensorBoard。
-log_tensorboard	指定HDFS中TensorBoard日志的位置。如果训练同时启动 TensorBoard，需要指定TensorBoard日志位置。
-conf CONF Hadoop conf	位置，默认可以不设置 ，使用E-MapReduce默认配置即可。
-f FILES	运行TensorFlow所有依赖的文件和文件夹，包含执行脚本，如果设置 virtual-env执行的virtual-env文件，用户可以将所有依赖放置到一个文件夹中，脚本会自动将文件夹按照文件夹层次关系上传到HDFS中。
-pn TensorFlow	启动的参数服务器个数。
-pc	每个参数服务器申请的CPU核数。
-pm	每个参数服务器申请的内存大小。
-wn TensorFlow	启动的Worker节点个数。

参数	说明
-wc	每个Worker申请的CPU核数。
-wg	每个Worker申请的GPU核数。
-wm	每个Worker申请的内存个数。
-c COMMAND	执行的命令，如pythoncensus.py。

进阶选项，用户需要谨慎使用进阶选项，可能造成任务失败。

参数	说明
-wnpg	每个GPU核上同时跑的Worker数量（针对tensorflow-ps）
-ppn	每个GPU核上同时跑的Worker数量（针对horovod） 以上两个选项指的是单卡多进程操作，由于共用一张显卡，需要在程序上进行一定限制，否则会造成显卡OOM。

9 Kafka

9.1 Kafka 快速入门

从 E-MapReduce 3.4.0 版本将开始支持 Kafka 服务。

创建Kafka集群

在 E-MapReduce 控制台创建集群时，选择集群类型为 Kafka，则会创建一个默认只包含 Kafka 组件的集群，除了基础组件外包括 Zookeeper、Kafka 和 KafkaManager 三个组件。每个节点只部署一个 Kafka broker。我们建议您的Kafka 集群是一个专用集群，不要和 Hadoop 相关服务混部在一起。

本地盘Kafka集群

为了更好地降低单位成本以及应对更大的存储需求，E-MapReduce 将在 EMR-3.5.1 版本开始支持基于本地盘（D1类簇机型，详情请参见#unique_21介绍文档）的 Kafka 集群。相比较云盘，本地盘 Kafka 集群有如下特点：

- 实例配备大容量、高吞吐 SATA HDD 本地盘，单磁盘 190 MB/s 顺序读写性能，单实例最大 5 GB/s 存储吞吐能力。
- 本地存储价格比 SSD 云盘降低 97%。
- 更高网络性能，最大 17 Gbit/s实例间网络带宽，满足业务高峰期实例间数据交互需求。

> **注意：**
>
> - 当宿主机宕机或者磁盘损坏时，磁盘中的数据将会丢失。
> - 请勿在本地磁盘上存储需要长期保存的业务数据，并及时做好数据备份和采用高可用架构。如需长期保存，建议将数据存储在云盘上。

为了能够在本地盘上部署 Kafka 服务，E-MapReduce 默认以下要求：

1. **default.replication.factor** = 3，即要求 topic 的分区副本数至少为3。如果设置更小副本数，则会增加数据丢失风险。
2. **min.insync.replicas** = 2，即要求当 producer 设定 acks 为 all(-1)时，每次至少写入两个副本才算写入成功。

当出现本地盘损坏时，E-MapReduce 会进行：

1. 将坏盘从 Broker 的配置中剔除，重启 Broker，在其他正常可用的本地盘上恢复坏盘丢失的数据。根据坏盘上已经写入的数据量不等，恢复的总时间也不等。

2. 当机器磁盘损坏数目过多（超过 20%）时，E-MapReduce 将主动进行机器迁移，恢复异常的磁盘。
3. 如果当前机器上可用剩余磁盘空间不足以恢复坏盘丢失数据时，Broker 将异常 Down 掉。这种情况，您可以选择清理一些数据，腾出磁盘空间并重启 Broker 服务；也可以联系 E-MapReduce 进行机器迁移，恢复异常的磁盘。

参数说明

您可以在 E-MapReduce 的集群配置管理中查看 Kafka 的软件配置，当前主要有：

配置项	说明
zookeeper.connect	Kafka 集群的 Zookeeper 连接地址
kafka.heap.opts	Kafka broker 的堆内存大小
num.io.threads	Kafka broker的 IO 线程数，默认为机器 CPU 核数目的 2 倍
num.network.threads	Kafka broker 的网络线程数，默认为机器的 CPU 核数目

9.2 Kafka 跨集群访问

通常，我们会单独部署一个Kafka集群来提供服务，所以经常需要跨集群访问Kafka服务。

Kafka 跨集群访问说明

跨集群访问Kafka场景分为两种：

- 阿里云内网环境中访问E-MapReduce Kafka集群。
- 公网环境访问E-MapReduce Kafka集群。

> **说明：**
> 经典网络的E-MapReduce Kafka集群暂不支持公网访问。

针对不同E-MapReduce版本，我们提供了不同的解决方案。

EMR-3.11.x及之后版本

- 阿里云内网中访问Kafka

  直接使用Kafka集群节点的内网 IP 访问即可，内网访问Kafka请使用 9092 端口。

  访问Kafka前请保证网络是互通的，VPC访问VPC的配置请参见配置 VPC 到 VPC 连接。

- 公网环境访问 Kafka

  Kafka 集群的 Core 节点默认无法通过公网访问，所以如果您需要公网环境访问 Kafka 集群，可以参考以下步骤完成：

  1. 使 Kafka 集群和公网主机网络互通。

     Kafka 集群部署在 VPC 网络环境，有两种方式：

     - 集群Core节点挂载弹性公网IP，以下操作步骤使用此方式。
     - 部署高速通道打通内网和公网网络，请参见高速通道文档。

  2. 在 E-MapReduce集群管理页面，单击对应集群操作栏中的**详情**，进入**集群基础信息**页面。
  3. 单击右上角的**网络管理** > **挂载公网**。
  4. 配置 Kafka 集群安全组规则来限制公网可访问 Kafka 集群的 IP 等，目的是提高 Kafka 集群暴露在公网中的安全性。您可以在 E-MapReduce 控制台查看到集群所属的安全组，根据安全组 ID 去查找并配置安全组规则，具体请参见#unique_8。
  5. 在 **集群基础信息**页面，单击页面右上角的**实例状态管理** > **同步主机信息**。
  6. 单击左侧导航栏的**集群服务** > **Kafka**，单击**配置**，在**服务配置**中找到 **kafka.public-access.enable**，修改为**true**。
  7. 重启 Kafka服务。
  8. 公网环境使用 Kafka 集群节点的 EIP 访问 9093 端口。

EMR-3.11.x 以前版本

- 阿里云内网中访问 Kafka

  我们需要在主机上配置 Kafka 集群节点的 host 信息。注意，这里我们需要在 client 端的主机上配置 Kafka 集群节点的**长域名**，否则会出现访问不到 Kafka 服务的问题。示例如下：

```
/etc/hosts
# kafka cluster
10.0.1.23 emr-header-1.cluster-48742
10.0.1.24 emr-worker-1.cluster-48742
10.0.1.25 emr-worker-2.cluster-48742
```

```
10.0.1.26 emr-worker-3.cluster-48742
```

- 公网环境访问 Kafka

  Kafka 集群的 Core 节点默认无法通过公网访问，所以如果您需要在公网环境访问 Kafka 集群，可以参考以下步骤完成：

  1. 使 Kafka 集群和公网主机网络互通。

     Kafka 集群部署在 VPC 网络环境，有两种方式：

     - 集群Core节点挂载弹性公网 IP，以下步骤使用此方式。
     - 部署高速通道打通内网和公网网络，参见高速通道文档。

  2. 在VPC 控制台申请 EIP，根据您 Kafka 集群 Core 节点个数购买相应的 EIP。
  3. 配置 Kafka 集群安全组规则来限制公网可访问 Kafka 集群的 IP 等，目的是提高 Kafka 集群暴露在公网中的安全性。您可以在 EMR 控制台查看到集群所属的安全组，根据安全组 ID 去查找并配置安全组规则，请参见#unique_8。
  4. 修改 Kafka 集群的软件配置 **listeners.address.principal** 为 HOST，并重启 Kafka 集群。
  5. 配置本地客户端主机的 hosts 文件。

9.3 Kafka Ranger 使用说明

前面简介中介绍了 E-MapReduce 中创建启动 Ranger 服务的集群，以及一些准备工作，本节介绍 Kafka 集成 Ranger 的一些步骤流程。

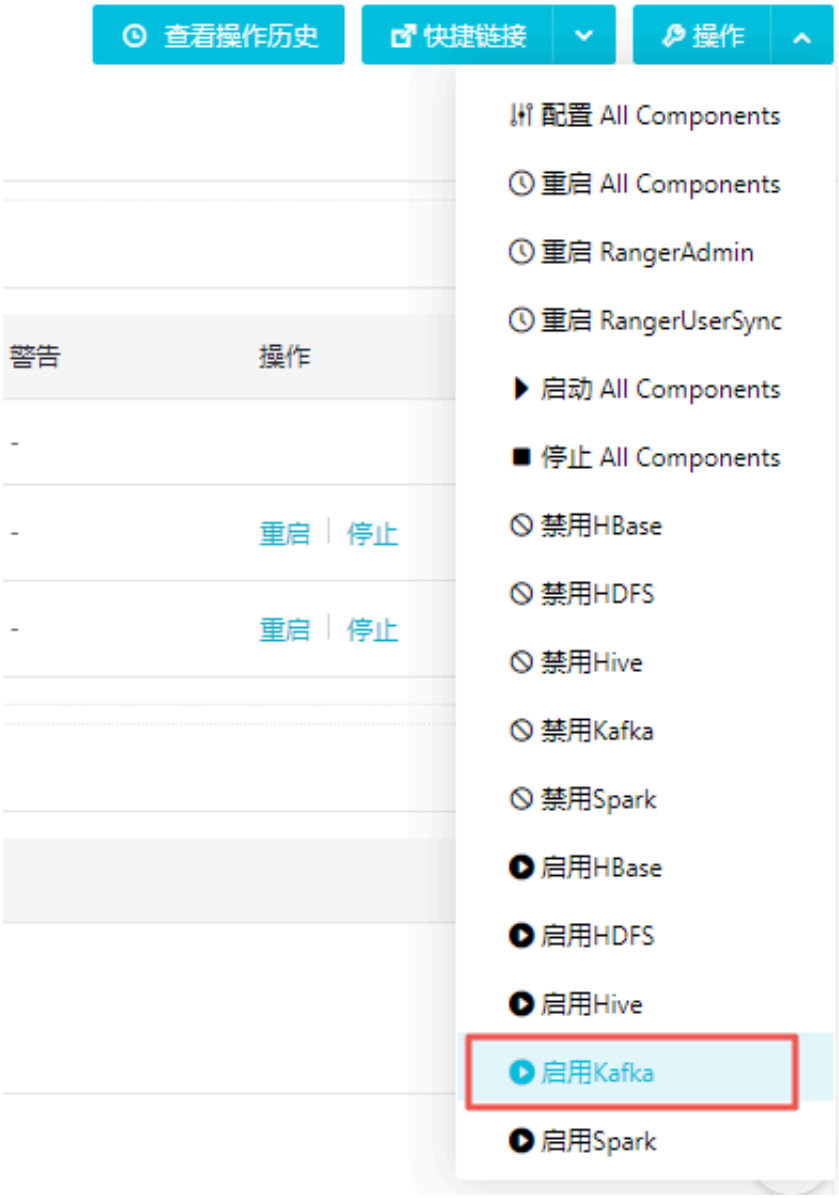
Kafka 集成 Ranger

从 E-MapReduce-3.12.0 版本开始，E-MapReduce Kafka 支持用 Ranger 进行权限配置，步骤如下：

- 通过主账号登录阿里云 E-MapReduce 控制台。
- 单击上方的**集群管理**页签。
- 在**集群管理**页面，单击相应集群所在行的**详情**。

- 启用Kafka

  1. 左侧导航栏单击**集群服务** > **RANGER**。
  2. 单击右侧的**操作**下拉菜单，选择 **启用Kafka**。

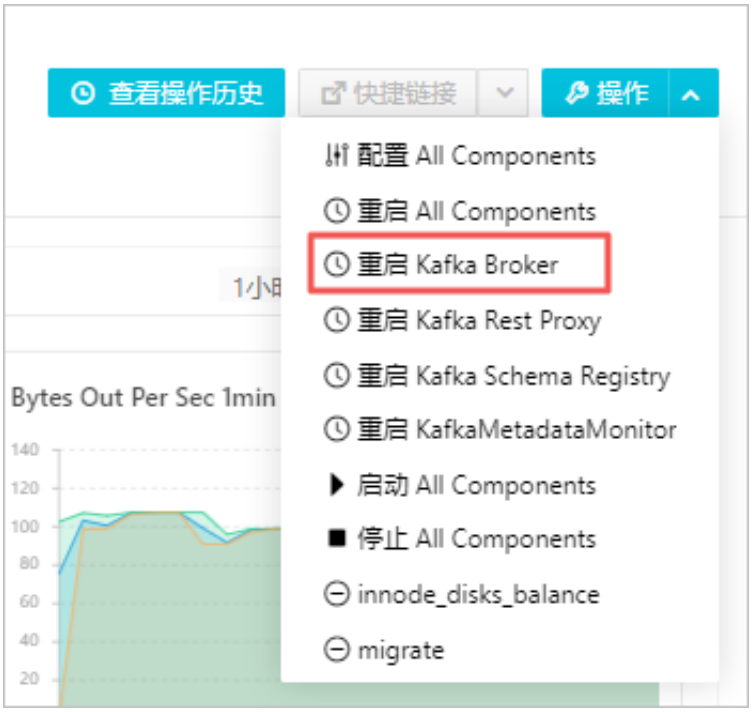


  3. 单击右上角的**查看操作历史**查看任务进度，等待状态为成功且进度为100%时，表示已完成启用kafka。

- 重启 Kafka broker

  上述任务完成后，需要重启 broker 才能生效。

  1. 左侧导航栏单击**集群服务** > **Kafka**。
  2. 单击右侧的**操作**下拉菜单，选择 **重启 Kafka Broker**。

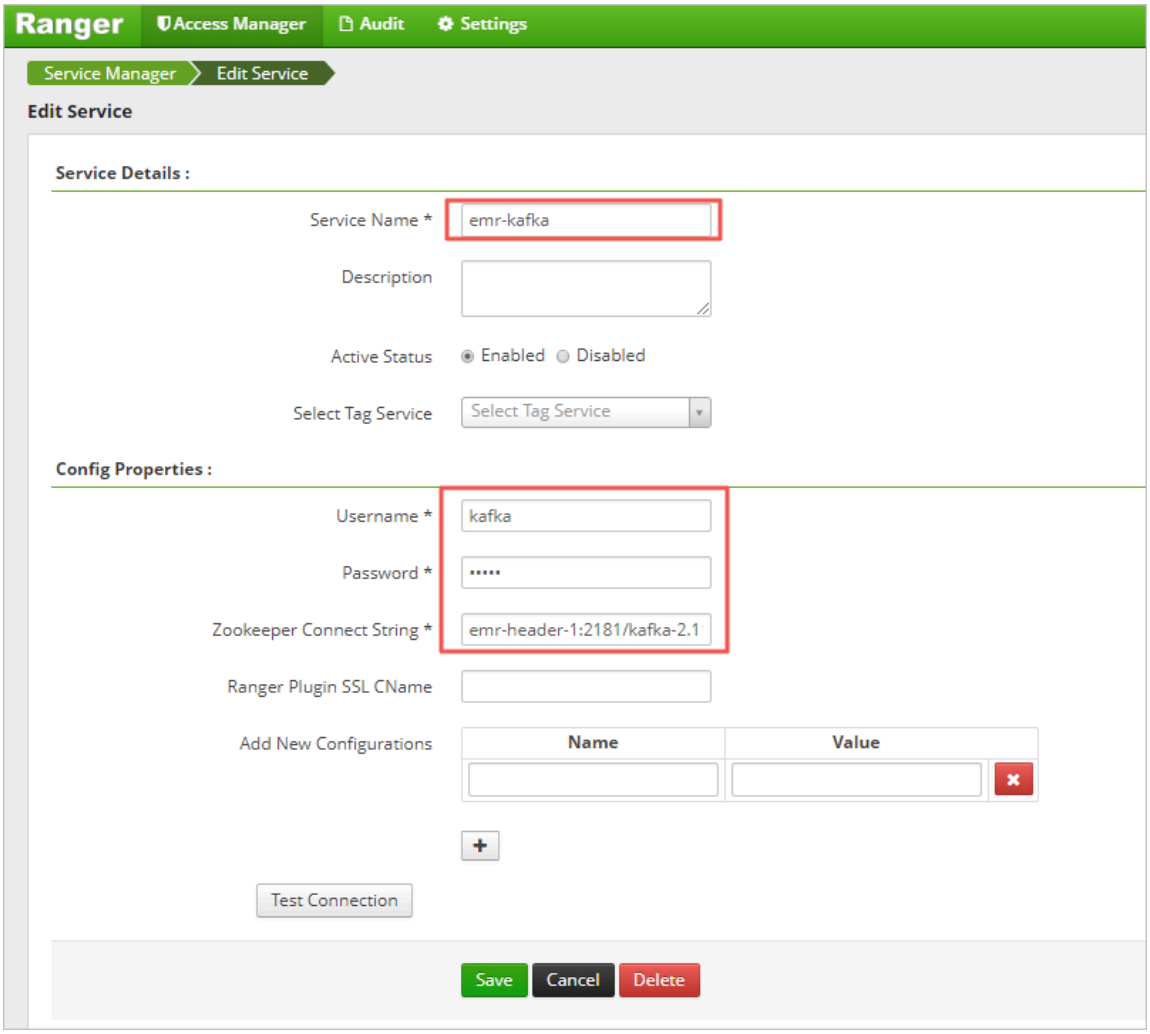


  3. 单击右上角**查看操作历史**，查看任务进度，等待重启任务完成。

- Ranger UI 页面添加 Kafka Service

  1. 进入 Ranger UI 页面，详情请参见Ranger简介。
  2. 在 **Ranger** 的 UI 页面添加 Kafka Service：

  3. 配置 Kafka Service



参数	说明
Service Name	固定填写emr-kafka。
Username	固定填写kafka。
Password	可任意填写。

参数	说明
Zookeeper Connect String	填写格式emr-header-1:2181/kafka-x.xx。 **说明：** 其中kafka-x.xx根据kafka实际版本填写。

权限配置示例

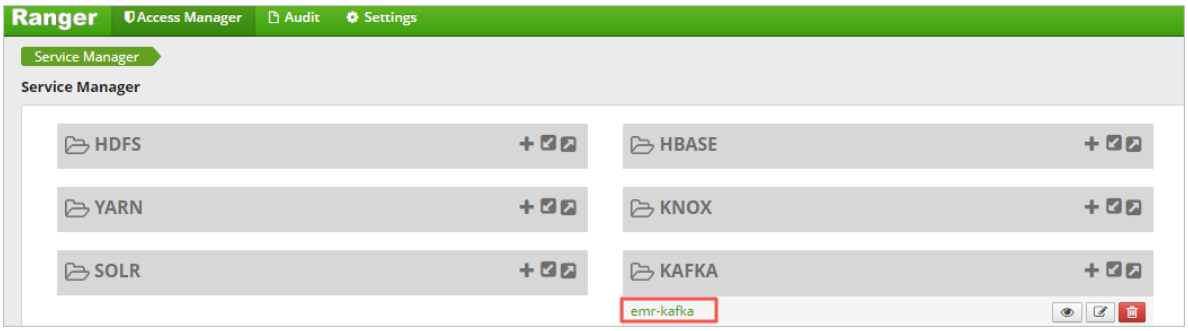
上面一节中已经将 Ranger 集成到 Kafka，现在可以进行相关的权限设置。

注意：

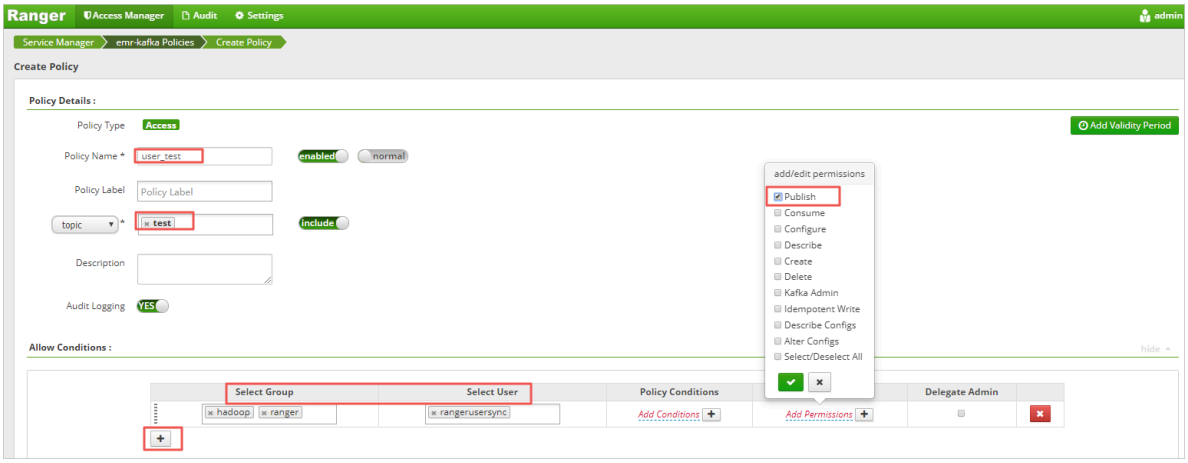
标准集群中，在添加了 Kafka Service 后，ranger 会默认生成规则 all - topic，不作任何权限限制（即允许所有用户进行所有操作），此时ranger无法通过用户进行权限识别。

以 test 用户为例，添加 Publish 权限。

1. 单击配置好的emr-kafka。



2. 单击右上角的**Add New Policy**。
3. 填写相关的参数。



参数	说明
Policy Name	可自定义。

参数	说明
topic	自定义。可填写多个，填写一个需按一次Enter键。
Select Group	Group和User会自动从集群同步过来，可以提前在集群添加好，同步过来需要一分钟左右。
Select User	
Permissions	单击 ，选择**Publish**。

单击**Select Group**下方的 ，可对多个Group进行授权。

4. 单击**add**。添加一个 Policy 后，就实现了对 **test** 的授权，然后用户 **test** 就可以对 **test** 的topic进行写入操作。

> **说明:**
> Policy 添加后需要 1 分钟左右才会生效。

9.4 Kafka SSL使用说明

E-MapReduce Kafka从EMR-3.12.0版本开始支持SSL功能。

前提条件

已创建Kafka类型的集群。详情请参见#unique_12。

开启SSL服务

Kafka集群默认没有开启SSL功能，您可以在Kafka服务的配置页面开启SSL。

1. 登录阿里云 E-MapReduce 控制台。
2. 单击上方的**集群管理**页签。
3. 在**集群管理**页面，单击相应集群所在行的**详情**。
4. 在左侧导航栏单击**集群服务** > **kafka**。

5. 单击**配置**页签，修改服务配置。

   a. 将**kafka.ssl.enable**修改为**true**。

   b. 单击**保存**。

   c. 在**确认修改**页面，输入**执行原因**，开启**自动更新配置**。

   d. 单击**确定**。

6. 单击**查看操作历史**，等待**状态**为**成功**时，表示配置完成。

7. 单击右上角的**操作** > **重启 All Componens**。

   a. 在**执行集群操作**页面，选择**执行范围**、**是否滚动执行**和**失败处理策略**，输入**执行原因**。

   b. 单击**确定**。

8. 单击**查看操作历史**，等待**状态**为**成功**时，表示开启SSL服务成功。

客户端访问Kafka

客户端通过SSL访问Kafka时需要设置 **security.protocol**、**truststore** 和 **keystore** 的相关配置。以非安全集群为例，如果是在 Kafka 集群运行作业，可以配置如下：

```
security.protocol=SSL
ssl.truststore.location=/etc/ecm/kafka-conf/truststore
ssl.truststore.password=${password}
ssl.keystore.location=/etc/ecm/kafka-conf/keystore
ssl.keystore.password=${password}
```

如果是在Kafka集群以外的环境运行作业，可将Kafka集群中的truststore和keystore文件（位于集群任意一个节点的 /etc/ecm/kafka-conf/目录中）拷贝至运行环境作相应配置。

以Kafka自带的producer和consumer程序，在Kafka集群运行为例：

1. 创建配置文件ssl.properties， 添加配置项。

   ```
   security.protocol=SSL
    ssl.truststore.location=/etc/ecm/kafka-conf/truststore
    ssl.truststore.password=${password}
    ssl.keystore.location=/etc/ecm/kafka-conf/keystore
    ssl.keystore.password=${password}
   ```

2. 创建topic。

   ```
   kafka-topics.sh --zookeeper emr-header-1:2181/kafka-1.0.1 --replication-factor 2 --
   ```

```
partitions 100 --topic test --create
```

3. 使用SSL配置文件产生数据。

```
kafka-producer-perf-test.sh --topic test --num-records 123456 --throughput 10000 --
record-size 1024 --producer-props bootstrap.servers=emr-worker-1:9092 --producer.
config ssl.properties
```

4. 使用SSL配置文件消费数据。

```
kafka-consumer-perf-test.sh --broker-list emr-worker-1:9092 --messages 100000000
 --topic test --consumer.config ssl.properties
```

9.5 Kafka Manager使用说明

E-MapReduce支持通过Kafka Manager服务对Kafka集群进行管理。

前提条件

> **注意:**
> 创建**Kafka**集群时默认安装Kafka Manager软件服务，并开启Kafka Manager的认证功能。

已创建**Kafka**类型的集群。

操作步骤

1. 使用SSH隧道方式访问Web页面，详情请参见#unique_5。

> **说明:**
> - 建议您首次使用Kafka Manager时修改默认密码。
> - 为了防止8085端口暴露，建议使用SSH隧道方式来访问Web界面；如果使用http://localhost:8085方式访问Web界面，请做好IP白名单保护，避免数据泄漏。

2. 在登录页面，输入**用户名**和**密码**。

   详细信息可通过以下步骤获取。

   a. 通过主账号登录阿里云 E-MapReduce 控制台。

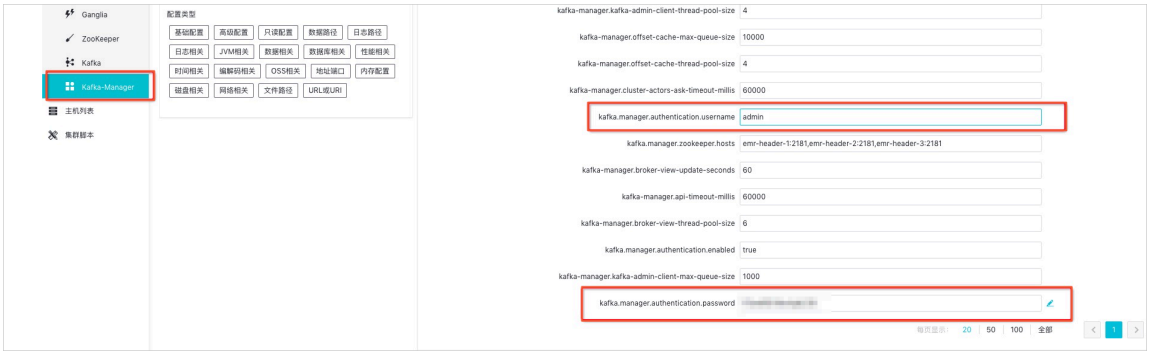
   b. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。

   c. 单击上方的**集群管理**页签。

   d. 在**集群管理**页面的集群列表中，单击对应集群后面的**详情**。

   e. 在左侧导航栏，选择**集群服务** > **Kafka-Manager**。

   f. 单击**配置**，在**服务配置**区域，用户名请查看**kafka.manager.authentication.username**、密码请查看**kafka.manager.authentication.password**。



   g. Kafka集群的Zookeeper地址，请查看**kafka.manager.zookeeper.hosts**。

3. 添加一个创建好的Kafka集群。

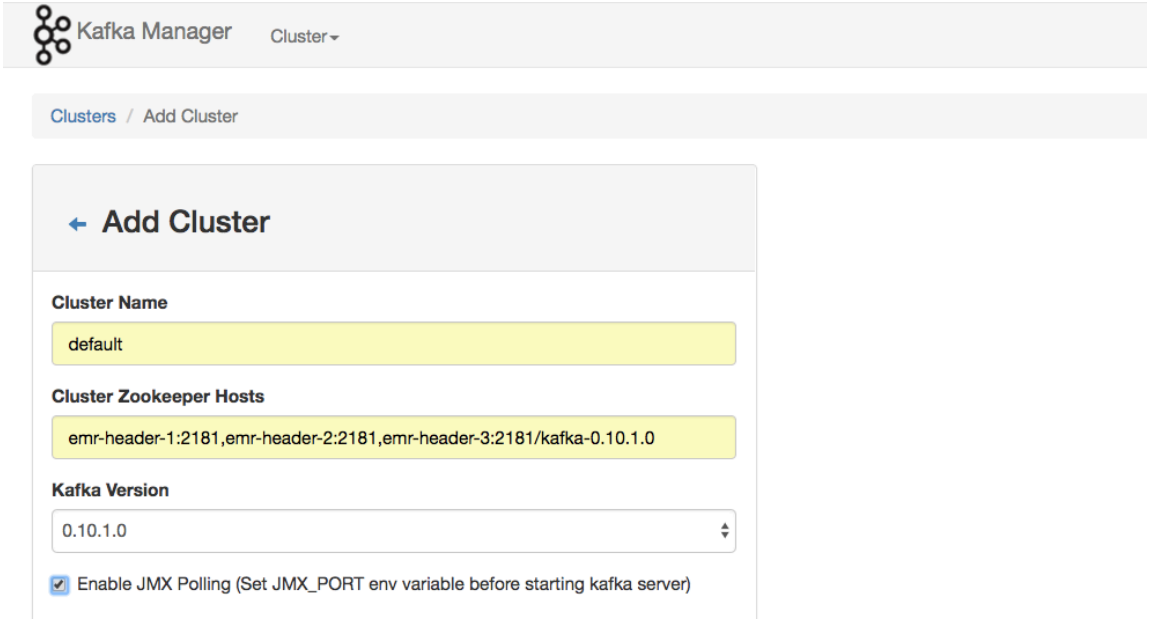
   a. 输入集群名称。

   b. 配置Kafka集群的Zookeeper地址。

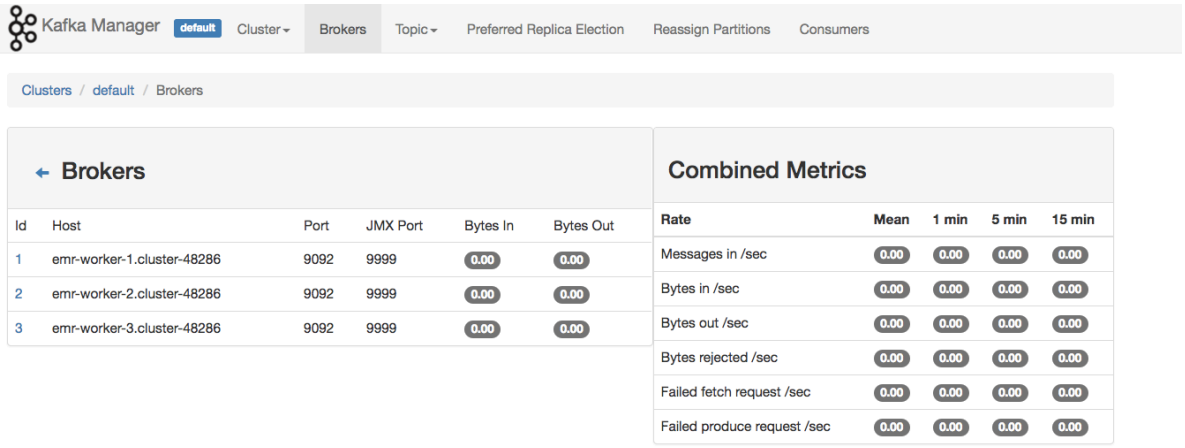
      填写在步骤2中获取**kafka.manager.zookeeper.hosts**的值。

   c. 选择对应的Kafka版本。

   d. 建议打开JMX功能。



创建好之后即可使用一些常见的Kafka功能。



9.6 Kafka 常见问题

本文介绍 Kafka 常见问题的一些问题以及解决方法。

- Error while executing topic command : Replication factor: 1 larger than available brokers : 0.

  常见原因：

  - Kafka 服务异常，集群 Broker 进程都退出了，需要结合日志排查问题。
  - Kafka 服务的 Zookeeper 地址使用错误，请注意查看并使用集群配置管理中 Kafka 组件的 Zookeeper 连接地址。

- java.net.BindException: Address already in use (Bind failed)

  当您使用 Kafka 命令行工具时，有时会碰到 java.net.BindException: Address already in use (Bind failed) 异常，这一般是由 JMX 端口被占用导致，您可以在命令行前手动指定一个 JMX 端口即可。例如：

  ```
  JMX_PORT=10101 kafka-topics --zookeeper emr-header-1:2181/kafka-1.0.0 --list
  ```

10 Druid

10.1 Druid 简介

Apache Druid 是一个分布式内存实时分析系统，用于解决如何在大规模数据集下进行快速的、交互式的查询和分析的问题。Apache Druid 由 Metamarkets 公司（一家为在线媒体或广告公司提供数据分析服务的公司）开发，在2019年春季被捐献给 Apache 软件基金会。

基本特点

Apache Druid 具有以下特点：

- 亚秒级 OLAP 查询，包括多维过滤、Ad-hoc 的属性分组、快速聚合数据等等。
- 实时的数据消费，真正做到数据摄入实时、查询结果实时。
- 高效的多租户能力，最高可以做到几千用户同时在线查询。
- 扩展性强，支持 PB 级数据、千亿级事件快速处理，支持每秒数千查询并发。
- 极高的高可用保障，支持滚动升级。

应用场景

实时数据分析是 Apache Druid 最典型的使用场景。该场景涵盖的面很广，例如：

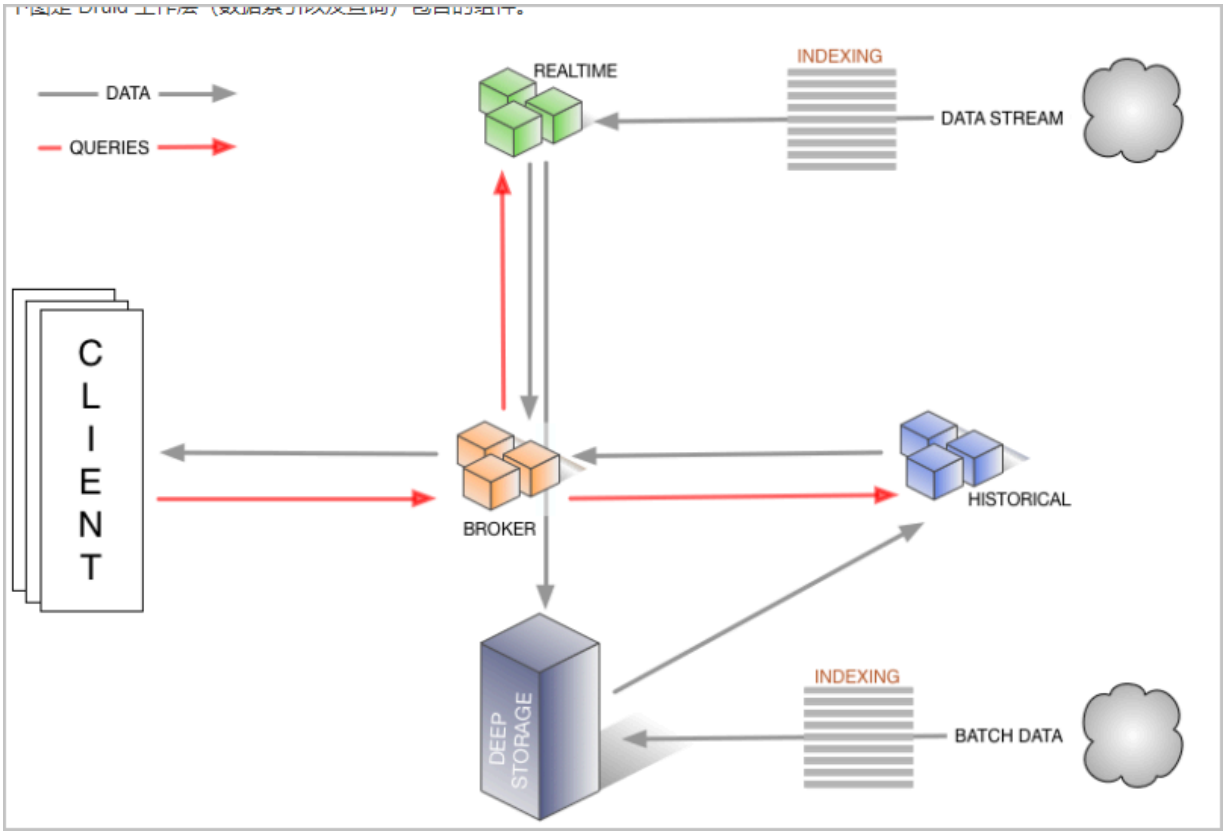
- 实时指标监控
- 推荐模型
- 广告平台
- 搜索模型

这些场景的特点都是拥有大量的数据，且对数据查询的时延要求非常高。在实时指标监控中，系统问题需要在出现的一刻被检测到并被及时给出报警。在推荐模型中，用户行为数据需要实时采集，并及时反馈到推荐系统中。用户几次点击之后系统就能够识别其搜索意图，并在之后的搜索中推荐更合理的结果。

Apache Druid 架构

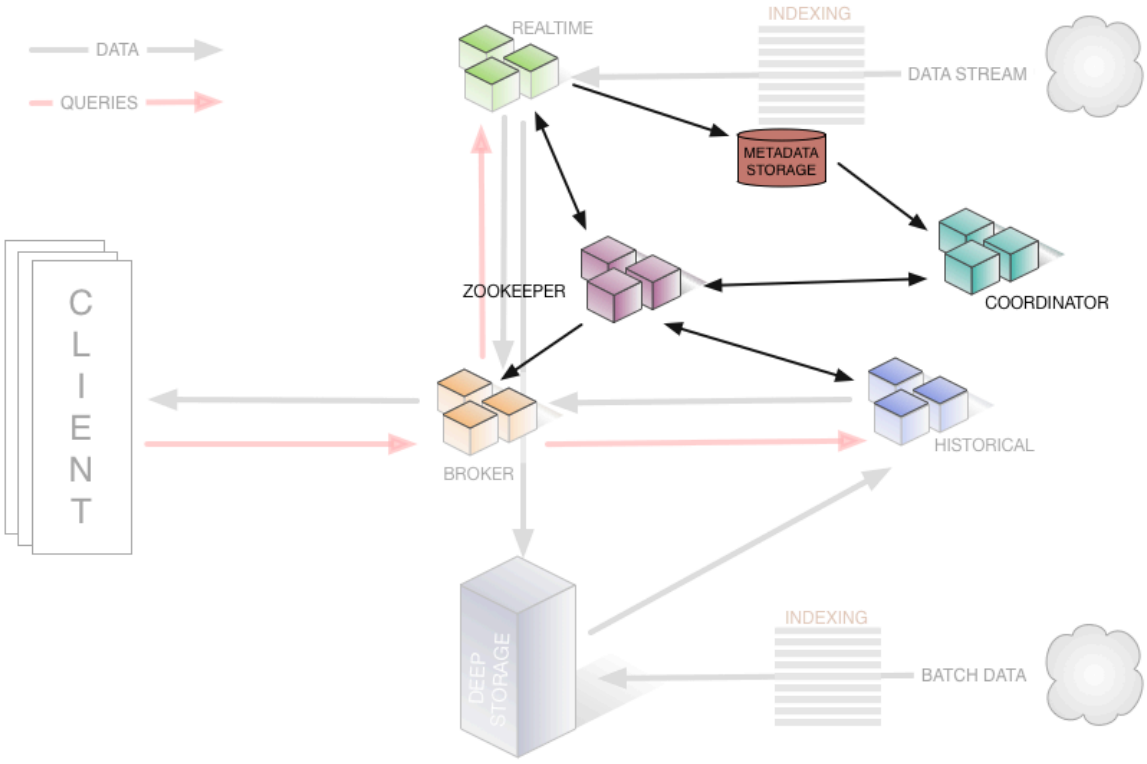
Apache Druid 拥有优秀的架构设计，多个组件协同工作，共同完成数据从摄取到索引、存储、查询等一系列流程。

下图是 Druid 工作层（数据索引以及查询）包含的组件。



- Realtime 组件负责数据的实时摄入。
- Broker 阶段负责查询任务的分发以及查询结果的汇总，并将结果返回给用户。
- Historical 节点负责索引后的历史数据的存储，数据存储在 deep storage。Deep storage 可以是本地，也可以是HDFS 等分布式文件系统。
- Indexing service 包含两个组件（图中未画出）。
  - Overlord 组件负责索引任务的管理、分发。
  - MiddleManager 负责索引任务的具体执行。

下图是 Druid segments（Druid 索引文件）管理层所涉及的组件。



- Zookeeper 负责存储集群的状态以及作为服务发现组件，例如集群的拓扑信息、overlord leader 的选举、indexing task 的管理等等。
- Coordinator 负责 segments 的管理，如 segments 下载、删除以及如何在 historical 之间做均衡等等。
- Metadata storage 负责存储 segments 的元信息，以及管理集群各种各样的持久化或临时性数据，例如配置信息、审计信息等等。

E-MapReduce 增强型 Druid

E-MapReduce Druid 基于Apache Druid 做了大量的改进，包括与E-MapReduce和阿里云周边生态的集成、方便的监控与运维支持、易用的产品接口等等，真正做到了即买即用和 7*24 免运维。

E-MapReduce Druid 目前支持的特性如下所示：

- 支持以 OSS 作为 deep storage。
- 支持将 OSS 文件作为批量索引的数据来源。
- 支持从日志服务（Log Service）流式地索引数据（类似于 Kafka），并提供高可靠保证和 exactly-once 语义。
- 支持将元数据存储到 RDS。
- 集成了 Superset 工具。
- 方便地扩容、缩容（缩容针对 task 节点）。

- 丰富的监控指标和告警规则。
- 坏节点迁移。
- 具有高安全性。
- 支持 HA。

10.2 快速入门

从EMR-3.11.0版本开始，E-MapReduce支持将E-MapReduce Druid作为E-MapReduce的一个集群类型。

背景信息

将E-MapReduce Druid作为一种单独的集群类型，而不再是在Hadoop集群中增加Druid组件，主要基于以下几方面的考虑：

- E-MapReduce Druid可以完全脱离Hadoop来使用。
- 大数据量情况下，E-MapReduce Druid对内存要求比较高，尤其是Broker节点Historical节点。E-MapReduce Druid本身不受YARN管控 ，在多服务运行时容易发生资源抢夺。
- Hadoop作为基础设施，其规模可以比较大，而E-MapReduce Druid集群可以比较小，两者配合起来工作灵活性更高。

创建Druid集群

在创建集群时选择Druid集群类型即可，具体创建集群操作请参见#unique_12。

> **说明：**
> 您在创建E-MapReduce Druid集群时可以勾选YARN和Superset服务，E-MapReduce Druid集群自带的HDFS和YARN仅供测试使用，原因如背景信息所述。对于生产环境，我们强烈建议您采用专门的Hadoop集群。

配置集群

- 配置使用HDFS作为E-MapReduce Druid的deep storage。

  对于独立的E-MapReduce Druid集群，如果您需要将索引数据存放在另外一个Hadoop集群的HDFS上，则您首先需要设置两个集群的连通性（请参见下文的与Hadoop集群交互），然后

在E-MapReduce Druid配置页面，配置以下两个选项并重启服务即可（配置项位于配置页面的common.runtime）。

- **druid.storage.type**设置为**hdfs**。
- **druid.storage.storageDirectory**：HDFS目录，强烈建议填写完整目录，例如：hdfs://emr-header-1.cluster-xxxxxxxx:9000/druid/segments。

说明：

如果Hadoop集群为HA集群，emr-header-1.cluster-xxxxx:9000需要改成emr-cluster，或者把端口9000改成8020。

- 配置使用OSS作为 E-MapReduce Druid的deep storage。

  E-MapReduce Druid支持以OSS作为deep storage，借助于E-MapReduce的免AccessKey能力，E-MapReduce Druid不用做AccessKey配置即可访问OSS。由于OSS的访问能力是借助于HDFS的OSS功能实现的，因此在配置时，**druid.storage.type**需要仍然配置为HDFS。

  - **druid.storage.type**设置为**hdfs**。
  - **druid.storage.storageDirectory:** （如oss://emr-druid-cn-hangzhou/segments）

  由于OSS访问借助了HDFS，因此您需要选择以下两种方案之一：

  - 创建集群的时候选择安装HDFS，系统自动安装好HDFS。
  - 在E-MapReduce Druid的配置目录/etc/ecm/druid-conf/druid/_common/下新建hdfs-site.xml，内容如下，然后将该文件拷贝至所有节点的相同目录下。

    ```
    <?xml version="1.0"?>
     <configuration>
      <property>
       <name>fs.oss.impl</name>
       <value>com.aliyun.fs.oss.nat.NativeOssFileSystem</value>
      </property>
      <property>
       <name>fs.oss.buffer.dirs</name>
       <value>file:///mnt/disk1/data,...</value>
      </property>
      <property>
       <name>fs.oss.impl.disable.cache</name>
       <value>true</value>
      </property>
     </configuration>
    ```

    其中**fs.oss.buffer.dirs**可以设置多个路径。

- 配置使用RDS作为E-MapReduce Druid的元数据存储。

  默认情况下E-MapReduce Druid利用header-1节点上的本地MySQL数据库作为元数据存储。您也可以配置使用阿里云RDS作为元数据存储。

  下面以RDS MySQL版为例演示配置。在具体配置之前，请先确保：

  - 已创建RDS MySQL实例。
  - 为E-MapReduce Druid访问RDS MySQL创建了单独的账户（不推荐使用root），假设账户名为druid，密码为druidpw。
  - 为E-MapReduce Druid元数据创建单独的MySQL数据库，假设数据库名为druiddb。
  - 确保账户druid有权限访问druiddb。

  在E-MapReduce管理控制台，进入E-MapReduce Druid集群，单击Druid组件，选择**配置**选项卡，找到common.runtime配置文件。单击**自定义配置**，添加如下三个配置项：

  - druid.metadata.storage.connector.connectURI，值为jdbc:mysql://rm-xxxxx.mysql.rds.aliyuncs.com:3306/druiddb。
  - druid.metadata.storage.connector.user，值为druid。
  - druid.metadata.storage.connector.password，值为druidpw。

  依次单击右上角的**保存**、**部署配置文件到主机**、**重启所有组件**，配置即可生效。

  登录RDS管理控制台，查看druiddb创建表的情况，如果正常，您将会看到一些druid自动创建的表。

- 配置组件内存。

  E-MapReduce Druid组件内存设置主要包括两方面：堆内存（通过jvm.config配置）和direct内存（通过jvm.config和runtime.properteis配置）。在创建集群时，E-MapReduce会自动生成一套配置，不过在某些情况下您仍然可能需要自己调整内存配置。

  要调整组件内存配置，您可以通过E-MapReduce控制台进入到集群组件，在页面上进行操作。

  > **说明：**
  > 对于 direct 内存，调整时请确保：
  >
  > -XX:MaxDirectMemorySize >= druid.processing.buffer.sizeBytes * (druid.processing.numMergeBuffers + druid.processing.numThreads + 1)

访问Druid web页面

E-MapReduce Druid自带三个Web页面：

- Overlord：http://emr-header-1.cluster-1234:18090，用于查看task运行情况。

- Coordinator：http://emr-header-1.cluster-1234:18081，用于查看segments存储情况，并设置rule加载和丢弃segments。
- Router（EMR-3.23.0及以上版本）：http://emr-header-1.cluster-1234:18888，也称之为console，是新版Druid的统一入口。

E-MapReduce提供三种方式访问E-MapReduce Druid的Web页面：

- 在集群管理页面，单击**访问链接与端口**，找到Druid overlord或Druid coordinator链接，单击链接进入。

  说明：
  您可以使用Knox账号访问Druid Web页面，Knox账号创建请参见#unique_14，Knox使用请参见Knox 使用说明。
- 通过建立SSH隧道，开启代理浏览器访问。具体操作步骤请参见#unique_5。
- 通过公网IP+端口访问，如http://123.123.123.123:18090（不推荐，请通过安全组设置合理控制公网访问集群权限）。

批量索引

- 与Hadoop集群交互

  您在创建E-MapReduce Druid集群时如果勾选了HDFS和YARN（自带Hadoop集群），那么系统将会自动为您配置好与HDFS和YARN的交互，您无需做额外操作。下面的介绍是配置独立E-MapReduce Druid集群与独立Hadoop集群之间交互，这里假设E-MapReduce Druid集群cluster id 为1234，Hadoop集群cluster id为5678。另外请严格按照指导进行操作，如果操作不当，集群可能就不会按照预期工作。

  对于与非安全独立Hadoop集群交互，请按照如下操作进行：

  1. 确保集群间能够通信（两个集群在一个安全组下，或两个集群在不同安全组，但两个安全组之间配置了访问规则）。
  2. 在E-MapReduce Druid集群的每个节点的指定路径下，放置一份Hadoop集群中/etc/ecm/hadoop-conf路径下的core-site.xml、hdfs-site.xml、yarn-site.xml、 mapred-site.xml文件。 这些文件在E-MapReduce Druid集群节点上放置的路径与E-MapReduce集群的版本有关，详情说明如下：
     - EMR-3.23.0及以上版本：/etc/ecm/druid-conf/druid/cluster/_common
     - EMR-3.23.0以下版本：/etc/ecm/druid-conf/druid/_common

     说明：

如果创建集群时选了自带Hadoop，则在上述目录下会有几个软链接指向自带Hadoop的配置，请先移除这些软链接。

3. 将Hadoop集群的hosts写入到E-MapReduce Druid集群的hosts列表中，注意Hadoop集群的hostname应采用长名形式，如emr-header-1.cluster-xxxxxxxx，且最好将Hadoop的hosts放在本集群hosts之后，例如：

```
...
10.157.*.*    emr-as.cn-hangzhou.aliyuncs.com
10.157.*.*    eas.cn-hangzhou.emr.aliyuncs.com
192.168.*.*   emr-worker-1.cluster-1234 emr-worker-1 emr-header-2.cluster-1234 emr-header-2 iZbp1h9g7boqo9x23qb****
192.168.*.*   emr-worker-2.cluster-1234 emr-worker-2 emr-header-3.cluster-1234 emr-header-3 iZbp1eaa5819tkjx55y****
192.168.*.*   emr-header-1.cluster-1234 emr-header-1 iZbp1e3zwuvnmakmsje****
--以下为hadoop集群的hosts信息
192.168.*.*   emr-worker-1.cluster-5678 emr-header-2.cluster-5678 iZbp195rj7zvx8qar4f****
192.168.*.*   emr-worker-2.cluster-5678 emr-header-3.cluster-5678 iZbp15vy2rsxoegki4q****
192.168.*.*   emr-header-1.cluster-5678 iZbp10tx4egw3wfnh5o****
```

对于安全Hadoop集群，请按如下操作进行：

1. 确保集群间能够通信（两个集群在一个安全组下，或两个集群在不同安全组，但两个安全组之间配置了访问规则）。
2. 在E-MapReduce Druid集群的每个节点的指定路径下，放置一份Hadoop集群/etc/ecm/hadoop-conf路径下的core-site.xml、hdfs-site.xml、yarn-site.xml、 mapred-site.xml文件，并修改core-site.xml中**hadoop.security.authentication.use.has** 为 false。

   其中，core-site.xml、hdfs-site.xml、yarn-site.xml、 mapred-site.xml文件在E-MapReduce Druid集群节点上放置的路径与E-MapReduce集群的版本有关，详情说明如下：

   - EMR-3.23.0 及以上版本：/etc/ecm/druid-conf/druid/cluster/_common
   - EMR-3.23.0 以下版本：/etc/ecm/druid-conf/druid/_common

   说明：

如果创建集群时选了自带Hadoop，则在上述目录下会有几个软链接指向自带Hadoop的配置，请先移除这些软链接。

其中，**hadoop.security.authentication.use.has**是一个客户端配置，目的是让用户能够使用AccessKey进行认证。如果使用Kerberos认证方式，则需要disable该配置。

3. 将Hadoop集群的hosts写入到E-MapReduce Druid集群每个节点的hosts列表中，注意Hadoop集群的hostname应采用长名形式，如emr-header-1.cluster-xxxxxxxx，且最好将Hadoop的hosts放在本集群hosts之后。
4. 设置两个集群间的Kerberos跨域互信，详情请参见跨域互信。
5. 在Hadoop集群的所有节点下都创建一个本地druid账户（`useradd -m -g hadoop druid`），或者设置 druid.auth.authenticator.kerberos.authToLocal（具体预发规则请参见Druid-Kerberos）创建Kerberos账户到本地账户的映射规则。推荐第一种做法，操作简便不易出错。

   说明:
   默认在安全Hadoop集群中，所有Hadoop命令必须运行在一个本地的账户中，该本地账户需要与principal的name部分同名。YARN也支持将一个principal映射至本地一个账户，即上文第二种做法。
6. 重启Druid服务。

- 使用Hadoop对批量数据创建索引

  E-MapReduce Druid自带了一个名为wikiticker的例子，位于${DRUID_HOME}/quickstart/tutorial下面（${DRUID_HOME}默认为/usr/lib/druid-current）。wikiticker文件（wikiticker-2015-09-12-sampled.json.gz）的每一行是一条记录，每条记录是个json对象。其格式如下所示：

  ```json
  {
    "time": "2015-09-12T00:46:58.771Z",
    "channel": "#en.wikipedia",
    "cityName": null,
    "comment": "added project",
    "countryIsoCode": null,
    "countryName": null,
    "isAnonymous": false,
    "isMinor": false,
    "isNew": false,
    "isRobot": false,
    "isUnpatrolled": false,
    "metroCode": null,
    "namespace": "Talk",
    "page": "Talk:Oswald Tilghman",
    "regionIsoCode": null,
    "regionName": null,
    "user": "GELongstreet",
    "delta": 36,
  ```

```
  "added": 36,
  "deleted": 0
}
```

使用Hadoop对批量数据创建索引，请按照如下步骤进行操作：

1. 将该压缩文件解压，并放置于HDFS的一个目录下（如 hdfs://emr-header-1.cluster-5678:9000/druid）。 在Hadoop集群上执行如下命令。

```
### 如果是在独立Hadoop集群上进行操作，做好两个集群互信之后需要拷贝一个 druid.keytab到Hadoop集群再kinit。
 kinit -kt /etc/ecm/druid-conf/druid.keytab druid
 ###
 hdfs dfs -mkdir hdfs://emr-header-1.cluster-5678:9000/druid
 hdfs dfs -put ${DRUID_HOME}/quickstart/tutorial/wikiticker-2015-09-12-sampled.json hdfs://emr-header-1.cluster-5678:9000/druid
```

说明：

- 对于安全集群执行 HDFS 命令前先修改/etc/ecm/hadoop-conf/core-site.xml中**hadoop.security.authentication.use.has**为false。
- 请确保已经在Hadoop集群每个节点上创建名为druid的Linux账户。

2. 准备一个数据索引任务文件${DRUID_HOME}/quickstart/tutorial/wikiticker-index.json，如下所示：

```
{
    "type" : "index_hadoop",
    "spec" : {
      "ioConfig" : {
        "type" : "hadoop",
        "inputSpec" : {
          "type" : "static",
          "paths" : "hdfs://emr-header-1.cluster-5678:9000/druid/wikiticker-2015-09-12-sampled.json"
        }
      },
      "dataSchema" : {
        "dataSource" : "wikiticker",
        "granularitySpec" : {
          "type" : "uniform",
          "segmentGranularity" : "day",
          "queryGranularity" : "none",
          "intervals" : ["2015-09-12/2015-09-13"]
        },
        "parser" : {
          "type" : "hadoopyString",
          "parseSpec" : {
            "format" : "json",
            "dimensionsSpec" : {
              "dimensions" : [
                "channel",
                "cityName",
                "comment",
                "countryIsoCode",
                "countryName",
```

```
                    "isAnonymous",
                    "isMinor",
                    "isNew",
                    "isRobot",
                    "isUnpatrolled",
                    "metroCode",
                    "namespace",
                    "page",
                    "regionIsoCode",
                    "regionName",
                    "user"
                  ]
                },
                "timestampSpec" : {
                  "format" : "auto",
                  "column" : "time"
                }
              }
            },
            "metricsSpec" : [
              {
                "name" : "count",
                "type" : "count"
              },
              {
                "name" : "added",
                "type" : "longSum",
                "fieldName" : "added"
              },
              {
                "name" : "deleted",
                "type" : "longSum",
                "fieldName" : "deleted"
              },
              {
                "name" : "delta",
                "type" : "longSum",
                "fieldName" : "delta"
              },
              {
                "name" : "user_unique",
                "type" : "hyperUnique",
                "fieldName" : "user"
              }
            ]
          },
          "tuningConfig" : {
            "type" : "hadoop",
            "partitionsSpec" : {
              "type" : "hashed",
              "targetPartitionSize" : 5000000
            },
            "jobProperties" : {
              "mapreduce.job.classloader": "true"
            }
          }
        },
        "hadoopDependencyCoordinates": ["org.apache.hadoop:hadoop-client:2.8.5"]
}
```

说明：

- **spec.ioConfig.type**设置为hadoop。
- **spec.ioConfig.inputSpec.paths**为输入文件路径。
- **tuningConfig.type**为hadoop。
- **tuningConfig.jobProperties**设置了mapreduce job的classloader。
- **hadoopDependencyCoordinates** 制定了hadoop client的版本。

3. 在E-MapReduce Druid集群上运行批量索引命令。

```
cd ${DRUID_HOME}
 curl --negotiate -u:druid -b ~/cookies -c ~/cookies -XPOST -H 'Content-Type:
application/json' -d @quickstart/tutorial/wikiticker-index.json http://emr-header-1
.cluster-1234:18090/druid/indexer/v1/task
```

其中`- -negotiate`、`-u`、`-b`、`-c`等选项是针对安全E-MapReduce Druid集群的。Overlord的端口默认为18090。

4. 查看作业运行情况。

在浏览器访问http://emr-header-1.cluster-1234:18090/console.html查看作业运行情况。

5. 根据Druid语法查询数据。

Druid有自己的查询语法。请准备一个描述您如何查询json格式的查询文件，如下所示为对wikiticker数据的一个top N查询（${DRUID_HOME}/quickstart/tutorial/wikiticker-top-pages.json）：

```
{
   "queryType" : "topN",
   "dataSource" : "wikiticker",
   "intervals" : ["2015-09-12/2015-09-13"],
   "granularity" : "all",
   "dimension" : "page",
   "metric" : "edits",
   "threshold" : 25,
   "aggregations" : [
     {
        "type" : "longSum",
        "name" : "edits",
        "fieldName" : "count"
     }
   ]
 }
```

在命令行界面运行下面的命令即可看到查询结果。

```
cd ${DRUID_HOME}
 curl --negotiate -u:druid -b ~/cookies -c ~/cookies -XPOST -H 'Content-Type:
application/json' -d @quickstart/tutorial/wikiticker-top-pages.json 'http://emr-
header-1.cluster-1234:18082/druid/v2/?pretty'
```

其中`- -negotiate`、`-u`、`-b`、`-c` 等选项是针对安全E-MapReduce Druid集群的。如果一切正常，您将能看到具体的查询结果。

实时索引

对于数据从Kafka集群实时到E-MapReduce Druid集群进行索引，我们推荐使用Kafka Indexing Service扩展，提供了高可靠保证，支持exactly-once语义。关于Druid Kafka Indexing Service 实时消费Kafka数据具体步骤，请参见Kafka Indexing Service。

如果您的数据实时打到了阿里云日志服务（SLS），并想用E-MapReduce Druid实时索引这部分数据，我们提供了SLS Indexing Service扩展。使用SLS Indexing Service避免了您额外建立并维护Kafka集群的开销。SLS Indexing Service的作用与Kafka Indexing Service相同，也提供高可靠保证和 Exactly-Once语义。在这里，您完全可以把SLS当成一个Kafka来使用。详情请参见 SLS-Indexing-Service。

Kafka Indexing Service和SLS Indexing Service是类似的，都使用拉的方式从数据源拉取数据到E-MapReduce Druid集群，并提供高可靠保证和 exactly-once语义。

索引失败问题分析思路

当发现索引失败时，一般遵循如下排错思路：

- 对于批量索引
  1. 如果curl直接返回错误，或者不返回，检查一下输入文件格式，或者curl加上 **-v** 参数，观察REST API的返回情况。
  2. 在Overlord页面观察作业执行情况，如果失败，查看页面上的logs。
  3. 在很多情况下并没有生成logs，如果是Hadoop作业，打开YARN页面查看是否有索引作业生成，并查看作业执行log。
  4. 如果上述情况都没有定位到错误，需要登录到E-MapReduce Druid集群，查看Overlord的执行日志（位于/mnt/disk1/log/druid/overlord-emr-header-1.cluster-xxxx.log），如果是HA集群，查看您提交作业的那个Overlord。
  5. 如果作业已经被提交到Middlemanager，但是从Middlemanager返回了失败，则需要从Overlord中查看作业提交到了那个worker，并登录到相应的worker，查看Middlemanager的日志（位于/mnt/disk1/log/druid/middleManager-emr-header-1.cluster-xxxx.log）。
- 对于Kafka Indexing Service和SLS Indexing Service
  1. 首先查看Overlord的Web页面：http://emr-header-1:18090， 查看Supervisor的运行状态，检查payload是否合理。
  2. 查看失败task的log。
  3. 如果不能从task log定位出失败原因，则需要从Overlord log排查问题。

10.3 数据格式描述文件

本文介绍索引数据的描述文件（Ingestion Spec文件）。

Ingestion Spec（数据格式描述）是Druid对要索引数据的格式以及如何索引该数据格式的一个统一描述，它是一个JSON文件，一般由三部分组成。

```
{
   "dataSchema" : {...},
   "ioConfig" : {...},
   "tuningConfig" : {...}
}
```

键	格式	描述	是否必须
dataSchema	JSON对象	描述所要消费数据的schema信息。dataSchema 是固定的，不随数据消费方式改变。	是
ioConfig	JSON对象	描述所要消费数据的来源和消费去向。数据消费方式不同，ioConfig也不相同。	是
tuningConfig	JSON对象	调节数据消费时的一些参数。数据消费方式不同，可调节的参数也不相同。	否

DataSchema

第一部分的dataSchema描述了数据的格式，如何解析该数据，典型结构如下。

```
{
   "dataSource": <name_of_dataSource>,
   "parser": {
      "type": <>,
      "parseSpec": {
         "format": <>,
         "timestampSpec": {},
         "dimensionsSpec": {}
      }
   },
   "metricsSpec": {},
   "granularitySpec": {}
}
```

键	格式	描述	是否必须
dataSource	字符串	数据源的名称。	是
parser	JSON 对象	数据的解析方式。	是
metricsSpec	JSON 对象数组	聚合器（aggregator）列表。	是

键	格式	描述	是否必须
granularitySpec	JSON 对象	数据聚合设置，如创建segments、聚合粒度等。	是

- parser

  parser部分决定了您的数据如何被正确地解析，metricsSpec定义了数据如何被聚集计算，granularitySpec定义了数据分片的粒度、查询的粒度。

  对于parser，type有两个选项：string和hadoopString，后者用于Hadoop索引的job。parseSpec是数据格式解析的具体定义。

键	格式	描述	是否必须
type	字符串	数据格式，可以是 “json” 、 “jsonLowercase” 、 “csv” 和 “tsv” 几种格式。	是
timestampSpec	JSON对象	时间戳和时间戳类型。	是
dimensionsSpec	JSON对象	数据的维度（包含哪些列）。	是

  对于不同的数据格式，可能还有额外的parseSpec选项。

  timestampSpec表的描述如下。

键	格式	描述	是否必须
column	字符串	时间戳对应的列。	是
format	字符串	时间戳类型，可选” iso” 、 “millis” 、 “posix” 、 “auto” 和joda time支持的类型。	是

  dimensionsSpec表的描述如下。

键	格式	描述	是否必须
dimensions	JSON数组	描述数据包含哪些维度。每个维度可以只是个字符串，或者可以额外指明维度的属性，例如 “dimensions” : [“dimenssion1” , “dimenssion2” , “{ “type” : “long” , “name” : “dimenssion3” }]，默认是string类型。	是

键	格式	描述	是否必须
dimensionExclusions	JSON字符串数组	数据消费时要剔除的维度。	否
spatialDimensions	JSON对象数组	空间维度。	否

- metricsSpec

  metricsSpec是一个JSON对象数组，定义了一些聚合器（aggregators）。聚合器通常有如下的结构。

  ```json
  {
    "type": <type>,
    "name": <output_name>,
    "fieldName": <metric_name>
  }
  ```

  官方提供了以下常用的聚合器。

类型	type 可选
count	count
sum	longSum、doubleSum、floatSum
min/max	longMin/longMax、doubleMin/doubleMax、floatMin/floatMax
first/last	longFirst/longLast、doubleFirst/doubleLast、floatFirst/floatLast
javascript	javascript
cardinality	cardinality
hyperUnique	hyperUnique

说明：

后三个属于高级聚合器，请参见Apache Druid 官方文档学习如何使用。

- granularitySpec

  聚合支持两种聚合方式：uniform和arbitrary，前者以一个固定的时间间隔聚合数据，后者尽量保证每个segments大小一致，时间间隔是不固定的。目前uniform是默认选项。

键	格式	描述	是否必须
segmentGranularity	字符串	segments粒度。uniform 方式使用。默认为”DAY”。	否
queryGranularity	字符串	可供查询的最小数据聚合粒度，默认值为”true”。	否
rollup	bool值	是否聚合。	否
intervals	字符串	数据消费时间间隔。	- batch：是 - realtime：否

ioConfig

第二部分ioConfig描述了数据来源。以下是一个Hadoop索引的例子。

```
{
  "type": "hadoop",
  "inputSpec": {
    "type": "static",
    "paths": "hdfs://emr-header-1.cluster-6789:9000/druid/quickstart/wikiticker-2015-09-16-sampled.json"
  }
}
```

说明：

对于通过Tranquility处理的流式数据，这部分是不需要的。

Tunning Config

Tuning Config是指一些额外的设置。例如，Hadoop对批量数据创建索引，可以在这里指定一些MapReduce参数。Tunning Config的内容依赖于您的数据来源可能有不同的内容。

10.4 Kafka Indexing Service

本文将介绍在 E-MapReduce 中如何使用 Apache Druid Kafka Indexing Service 实时消费 Kafka 数据。

Kafka Indexing Service 是 Apache Druid 推出的使用 Apache Druid 的 Indexing Service 服务实时消费 Kafka 数据的插件。该插件会在 Overlord 中启动一个 supervisor，supervisor 启动之后会在 Middlemanager 中启动一些 indexing task，这些 task 会连接到 Kafka 集群消费

topic 数据，并完成索引创建。您只需要准备一个数据消费格式文件，之后通过 REST API 手动启动 supervisor。

与 Kafka 集群交互

E-MapReduce Druid 集群与 Kafka 集群交互的配置方式与 Hadoop 集群类似，均需要设置连通性、hosts 等。

对于非安全 Kafka 集群，请按照以下步骤操作：

1. 确保集群间能够通信（两个集群在一个安全组下，或两个集群在不同安全组，但两个安全组之间配置了访问规则）。
2. 将 Kafka 集群的 hosts 写入到 E-MapReduce Druid 集群每一个节点的 hosts 列表中。

> **注意：**
> Kafka 集群的 hostname 应采用长名形式，例如 emr-header-1.cluster-xxxxxxxx。

对于安全 Kafka 集群，您需要执行下列操作（前两步与非安全 Kafka 集群相同）：

1. 确保集群间能够通信（两个集群在一个安全组下，或两个集群在不同安全组，但两个安全组之间配置了访问规则）。
2. 将 Kafka 集群的 hosts 写入到 E-MapReduce Druid 集群每一个节点的 hosts 列表中。

> **注意：**
> Kafka 集群的 hostname 应采用长名形式，如 emr-header-1.cluster-xxxxxxxx。

3. 设置两个集群间的 Kerberos 跨域互信（详情请参见跨域互信），推荐做双向互信。
4. 准备一个客户端安全配置文件，文件内容格式如下。

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/ecm/druid-conf/druid.keytab"
    principal="druid@EMR.1234.COM";
 };
```

文件准备好后，将该配置文件同步到 E-MapReduce Druid 集群的所有节点上，放置于某一个目录下面（例如/tmp/kafka/kafka_client_jaas.conf）。

5. 在 E-MapReduce Druid 配置页面的 overlord.jvm 中新增如下选项。

```
Djava.security.auth.login.config=/tmp/kafka/kafka_client_jaas.conf
```

6. 在 E-MapReduce Druid 配置页面的 middleManager.runtime 中配置druid.indexer.runner.javaOpts=-Djava.security.auth.login.confi=/tmp/kafka/kafka_client_jaas.conf和其他 JVM 启动参数。

7. 重启 Druid 服务。

使用 Apache Druid Kafka Indexing Service 实时消费 Kafka 数据

1. 在 Kafka 集群（或 Gateway）上执行以下命令创建一个名称为 metrics 的 topic。

```
-- 如果开启了 Kafka 高安全：
 export KAFKA_OPTS="-Djava.security.auth.login.config=/etc/ecm/kafka-conf/
kafka_client_jaas.conf"
 --
 kafka-topics.sh --create --zookeeper emr-header-1:2181,emr-header-2,emr-header-
3/kafka-1.0.0 --partitions 1 --replication-factor 1 --topic metrics
```

实际创建 topic 时，您需要根据您的环境配置来替换上述命令中的各个参数。其中，**--zookeeper** 参数中 /kafka-1.0.0 是一个路径，该路径的获取方法是：登录阿里云 E-MapReduce 控制台> 进入 Kafka 集群的 Kafka 服务**配置**页面 > 查看 zookeeper.connect 配置项的值。如果您的 Kafka 集群是自建集群，则您需要根据集群的实际配置来替换 **--zookeeper** 参数。

2. 定义数据源的数据格式描述文件（名称命名为 metrics-kafka.json），并放置在当前目录下（或放置在其他您指定的目录上）。

```
{
    "type": "kafka",
    "dataSchema": {
      "dataSource": "metrics-kafka",
      "parser": {
        "type": "string",
        "parseSpec": {
          "timestampSpec": {
            "column": "time",
            "format": "auto"
          },
          "dimensionsSpec": {
            "dimensions": ["url", "user"]
          },
          "format": "json"
        }
      },
      "granularitySpec": {
        "type": "uniform",
        "segmentGranularity": "hour",
        "queryGranularity": "none"
      },
      "metricsSpec": [{
          "type": "count",
          "name": "views"
        },
        {
          "name": "latencyMs",
          "type": "doubleSum",
          "fieldName": "latencyMs"
        }
      ]
    },
    "ioConfig": {
      "topic": "metrics",
```

```
      "consumerProperties": {
        "bootstrap.servers": "emr-worker-1.cluster-xxxxxxxx:9092(您 Kafka 集群的
bootstrap.servers)",
        "group.id": "kafka-indexing-service",
        "security.protocol": "SASL_PLAINTEXT",
        "sasl.mechanism": "GSSAPI"
      },
      "taskCount": 1,
      "replicas": 1,
      "taskDuration": "PT1H"
    },
    "tuningConfig": {
      "type": "kafka",
      "maxRowsInMemory": "100000"
    }
}
```

说明：

ioConfig.consumerProperties.security.protocol 和 **ioConfig.consumerProperties.sasl.mechanism** 为安全相关选项（非安全 Kafka 集群不需要）。

3. 执行下述命令添加 Kafka supervisor。

```
curl --negotiate -u:druid -b ~/cookies -c ~/cookies -XPOST -H 'Content-Type:
application/json' -d @metrics-kafka.json http://emr-header-1.cluster-1234:18090/
druid/indexer/v1/supervisor
```

其中 `--negotiate`、`-u`、`-b`、`-c` 等是针对安全 E-MapReduce Druid 集群的选项。

4. 在 Kafka 集群上开启一个 console producer。

```
-- 如果开启了 Kafka 高安全：
 export KAFKA_OPTS="-Djava.security.auth.login.config=/etc/ecm/kafka-conf/
kafka_client_jaas.conf"
 echo -e "security.protocol=SASL_PLAINTEXT\nsasl.mechanism=GSSAPI" > /tmp/Kafka
/producer.conf
 --
 Kafka-console-producer.sh --producer.config /tmp/kafka/producer.conf --broker-list
 emr-worker-1:9092，emr-worker-2:9092，emr-worker-3:9092 --topic metrics
 >
```

其中 --producer.config /tmp/Kafka/producer.conf 是针对安全 Kafka 集群的选项。

5. 在 kafka_console_producer 的命令提示符下输入一些数据。

```
{"time": "2018-03-06T09:57:58Z", "url": "/foo/bar", "user": "alice", "latencyMs": 32}
{"time": "2018-03-06T09:57:59Z", "url": "/", "user": "bob", "latencyMs": 11}
{"time": "2018-03-06T09:58:00Z", "url": "/foo/bar", "user": "bob", "latencyMs": 45}
```

其中时间戳可用如下 python 命令生成：

```
python -c 'import datetime; print(datetime.datetime.utcnow().strftime("%Y-%m-%dT%
H:%M:%SZ"))'
```

6. 准备一个查询文件，命名为 metrics-search.json。

```
{
```

```
    "queryType" : "search",
    "dataSource" : "metrics-kafka",
    "intervals" : ["2018-03-02T00:00:00.000/2018-03-08T00:00:00.000"],
    "granularity" : "all",
    "searchDimensions": [
      "url",
      "user"
    ],
    "query": {
      "type": "insensitive_contains",
      "value": "bob"
    }
}
```

7. 在 E-MapReduce Druid 集群 Master 上执行查询。

```
curl --negotiate -u:Druid -b ~/cookies -c ~/cookies -XPOST -H 'Content-Type:
application/json' -d @metrics-search.json http://emr-header-1.cluster-1234:18082/
druid/v2/?pretty
```

其中--negotiate、-u、-b、-c 等是针对安全 E-MapReduce Druid 集群的选项。

正常返回结果示例：

```
[ {
  "timestamp" : "2018-03-06T09:00:00.000Z",
  "result" : [ {
    "dimension" : "user",
    "value" : "bob",
    "count" : 2
  } ]
} ]
```

10.5 SLS Indexing Service

SLS Indexing Service 是 E-MapReduce 推出的一个 Druid 插件，用于从 SLS 消费数据。

背景介绍

SLS Indexing Service 消费原理与 Kafka Indexing Service 类似，因此也支持 Kafka Indexing Service 一样的 Exactly-Once 语义。其综合了 SLS 与 Kafka Indexing Service 两个服务的优点：

- 极为便捷的数据采集，可以利用 SLS 的多种数据采集方式实时将数据导入 SLS。
- 不用额外维护一个 Kafka 集群，省去了数据流的一个环节。
- 支持 Exactly-Once 语义。
- 消费作业高可靠保证，作业失败重试，集群重启/升级业务无感知等。

准备工作

- 如果您还没有开通 SLS 服务，请先开通 SLS 服务，并配置好相应的 Project 和 Logstore。

- 准备好以下配置项内容：
  - SLS 服务的 endpoint（注意要用内网服务入口）
  - 可访问 SLS 服务的 AccessKeyId 和对应的 AccessKeySecret

使用 SLS Indexing Service

1. 准备数据格式描述文件

   如果您熟悉 Kafka Indexing Service，那么 SLS Indexing Service 会非常简单。具体请参见Kafka Indexing Service的介绍，我们用同样的数据进行索引，那么数据源的数据格式描述文件如下（将其保存为 metrics-sls.json）：

```
{
  "type": "sls",
  "dataSchema": {
    "dataSource": "metrics-sls",
    "parser": {
      "type": "string",
      "parseSpec": {
        "timestampSpec": {
          "column": "time",
          "format": "auto"
        },
        "dimensionsSpec": {
          "dimensions": ["url", "user"]
        },
        "format": "json"
      }
    },
    "granularitySpec": {
      "type": "uniform",
      "segmentGranularity": "hour",
      "queryGranularity": "none"
    },
    "metricsSpec": [{
        "type": "count",
        "name": "views"
      },
      {
        "name": "latencyMs",
        "type": "doubleSum",
        "fieldName": "latencyMs"
      }
    ]
  },
  "ioConfig": {
    "project": <your_project>,
    "logstore": <your_logstore>,
    "consumerProperties": {
      "endpoint": "cn-hangzhou-intranet.log.aliyuncs.com", (以杭州为例，注意使用内网服务入口)
      "access-key-id": <your_access_key_id>,
      "access-key-secret": <your_access_key_secret>,
      "logtail.collection-mode": "simple"/"other"
    },
    "taskCount": 1,
    "replicas": 1,
    "taskDuration": "PT1H"
```

```
  },
  "tuningConfig": {
     "type": "sls",
     "maxRowsInMemory": "100000"
  }
}
```

对比 Kafka Indexing Service 一节中的介绍，我们发现两者基本上是一样的。这里简要列一下需要注意的字段：

- type: sls。
- dataSchema.parser.parseSpec.format：与 ioConfig.consumerProperties.logtail.collection-mode 有关，也就是与 SLS 日志的收集模式有关。如果是极简模式（simple）收集，那么该处原本文件是什么格式，就填什么格式。如果是非极简模式（other）收集，那么此处取值为 json。
- ioConfig.project：您要收集的日志的 project。
- ioConfig.logstore： 您要收集的日志的 logstore。
- ioConfig.consumerProperties.endpoint： SLS 内网服务地址，例如杭州对应 cn-hangzhou-intranet.log.aliyuncs.com。
- ioConfig.consumerProperties.access-key-id：账户的 AccessKeyID。
- ioConfig.consumerProperties.access-key-secret： 账户的 AccessKeySecret。
- ioConfig.consumerProperties.logtail.collection-mode： SLS 日志收集模式，极简模式填 simple，其他情况填 other。

 **注意：**

上述配置文件中的 ioConfig 配置格式仅适用于 EMR-3.20.0 及之前版本。自 EMR-3.21.0 开始，ioConfig 配置变更如下：

```
"ioConfig": {
      "project": <your_project>,
      "logstore": <your_logstore>,
      "endpoint": "cn-hangzhou-intranet.log.aliyuncs.com", (以杭州为例，注意使用内网服务入口)
      "accessKeyId": <your_access_key_id>,
      "accessKeySec": <your_access_key_secret>,
      "collectMode": "simple"/"other"
      "taskCount": 1,
      "replicas": 1,
      "taskDuration": "PT1H"
  },
```

即，取消了 **consumerProperties** 层级、**access-key-id**、**access-key-secret**，**logtail.collection-mode** 变更为 **accessKeyIdaccessKeySeccollectMode** 。

2. 执行下述命令添加 SLS supervisor。

```
curl --negotiate -u:druid -b ~/cookies -c ~/cookies -XPOST -H 'Content-Type: application/json' -d @metrics-sls.json http://emr-header-1.cluster-1234:18090/druid/indexer/v1/supervisor
```

> **注意:**
> 其中 --negotiate、-u、-b、-c 等选项是针对安全 Druid 集群。

3. 向 SLS 中导入数据。

   您可以采用多种方式向 SLS 中导入数据。具体请参见 SLS 文档。

4. 在 Druid 端进行相关查询。

10.6 Superset

E-MapReduce Druid集群集成了Superset工具。Superset对E-MapReduce Druid做了深度集成，同时也支持多种关系型数据库。由于E-MapReduce Druid也支持SQL，所以可以通过Superset以两种方式访问E-MapReduce Druid，即Apache Druid原生查询语言或者SQL。

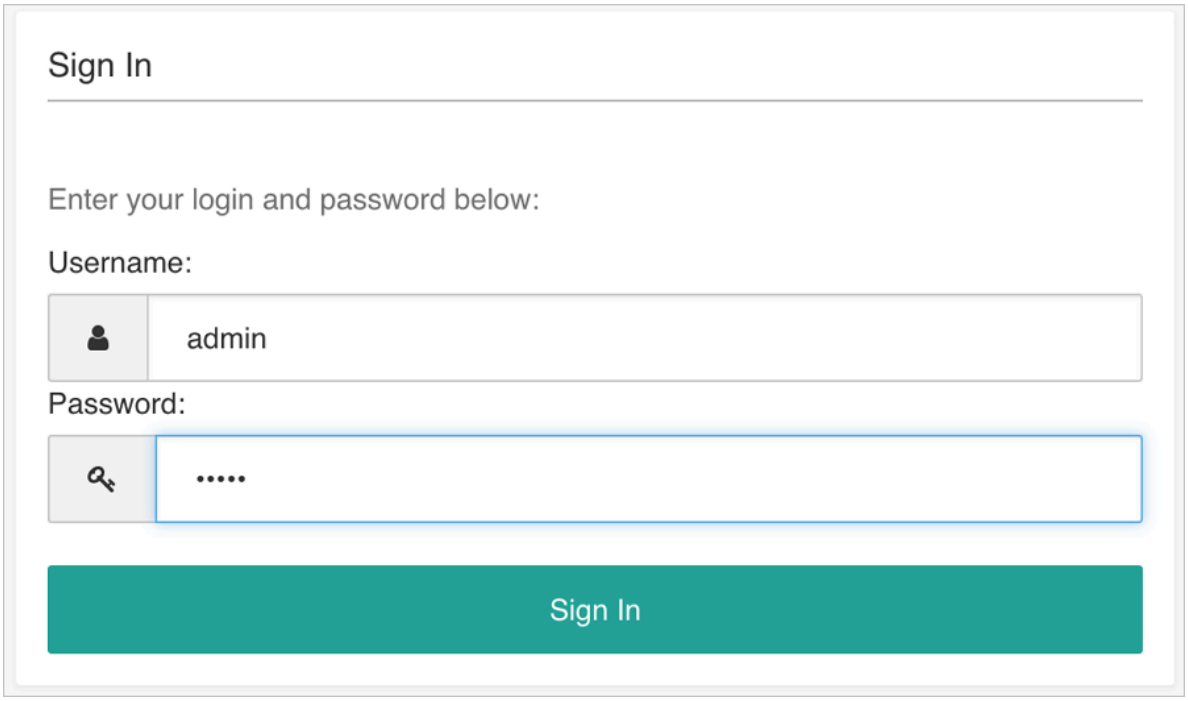
前提条件

Superset默认安装在emr-header-1节点，目前还不支持HA。在使用该工具前，确保您的主机能够正常访问emr-header-1，具体步骤请参见#unique_17。

使用Superset

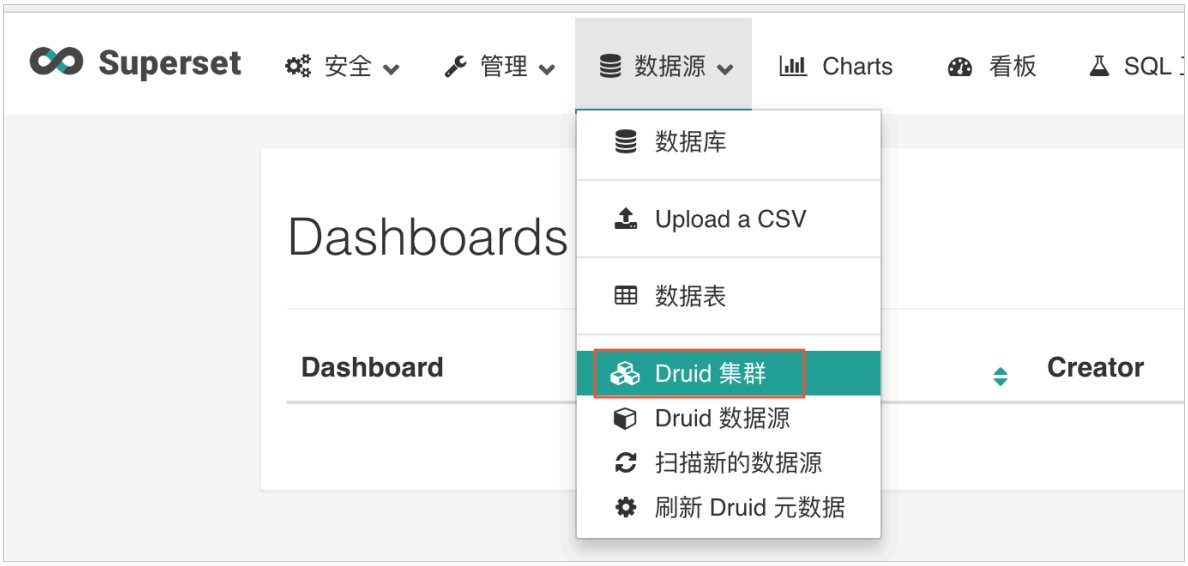
1. 登录Superset。

   在浏览器地址栏中输入http://emr-header-1:18088，按回车，打开Superset登录界面，默认用户名和密码均为admin，请您登录后及时修改密码。

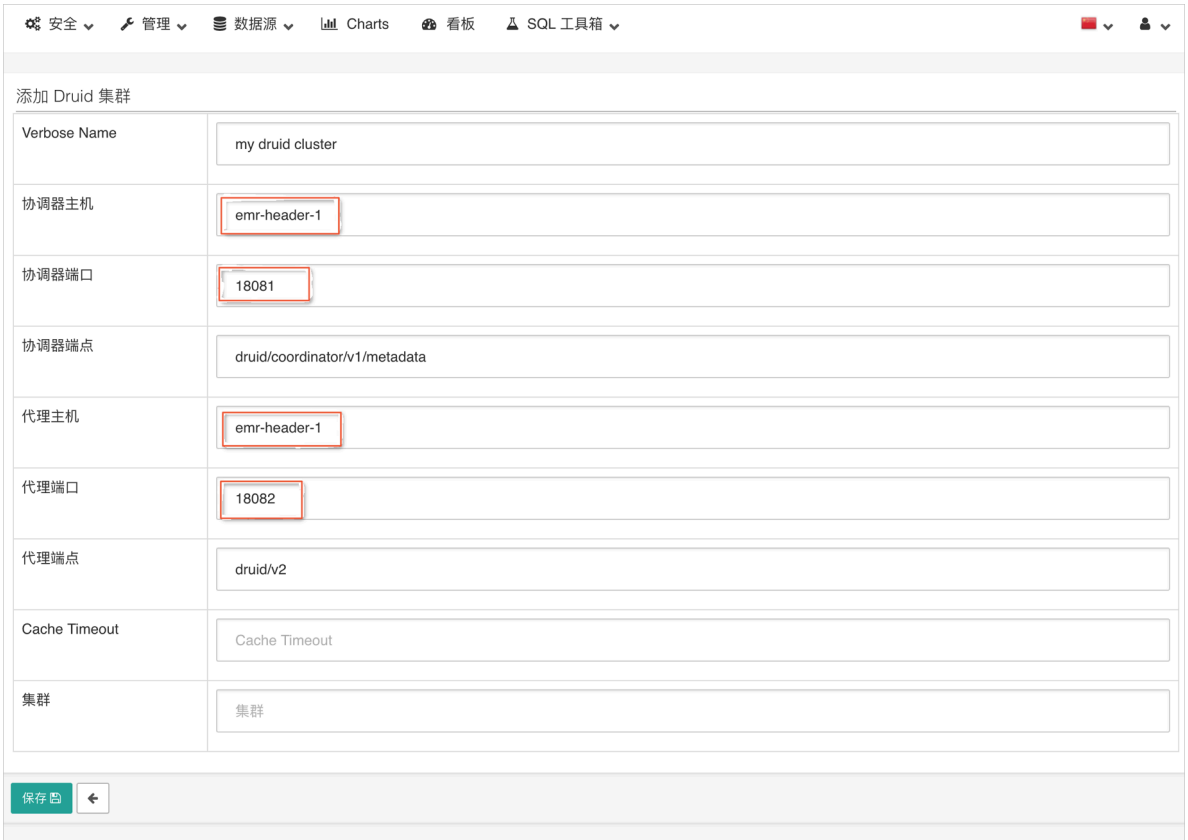


2. 添加E-MapReduce Druid集群。

登录后默认为英文界面，可单击右上角的国旗图标选择合适的语言。接下来在上方菜单栏中依次选择**数据源** > **Druid 集群**来添加一个E-MapReduce Druid集群。



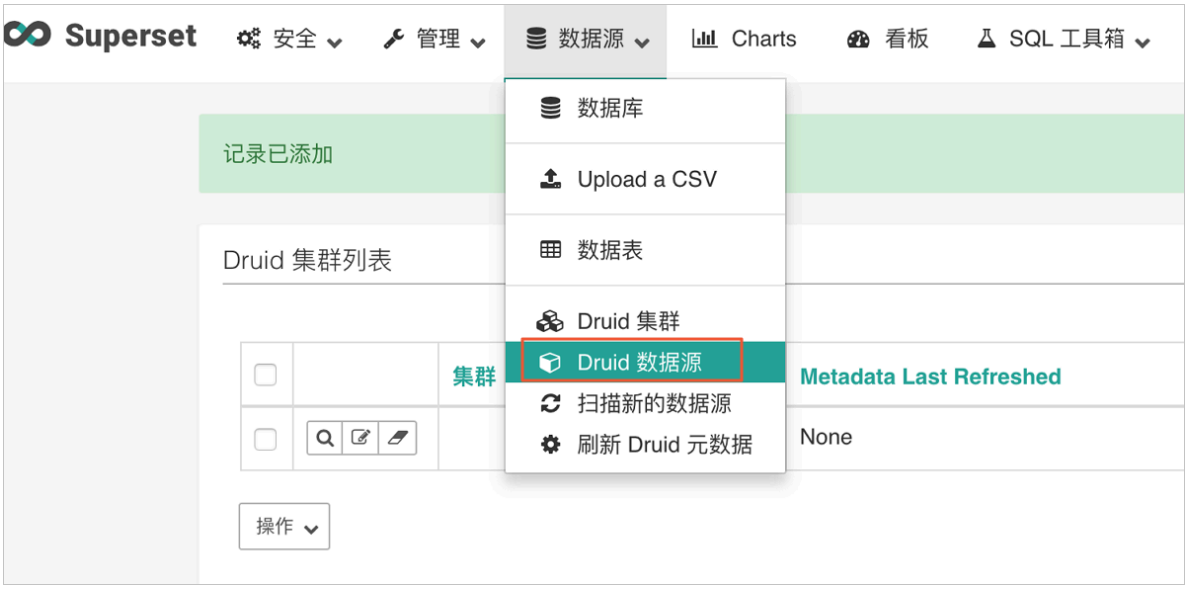
配置好协调机（Coordinator）和代理机（Broker）的地址，注意E-MapReduce中默认端口均为相应的开源端口前加数字1，例如开源Broker 端口为8082，E-MapReduce中为18082。



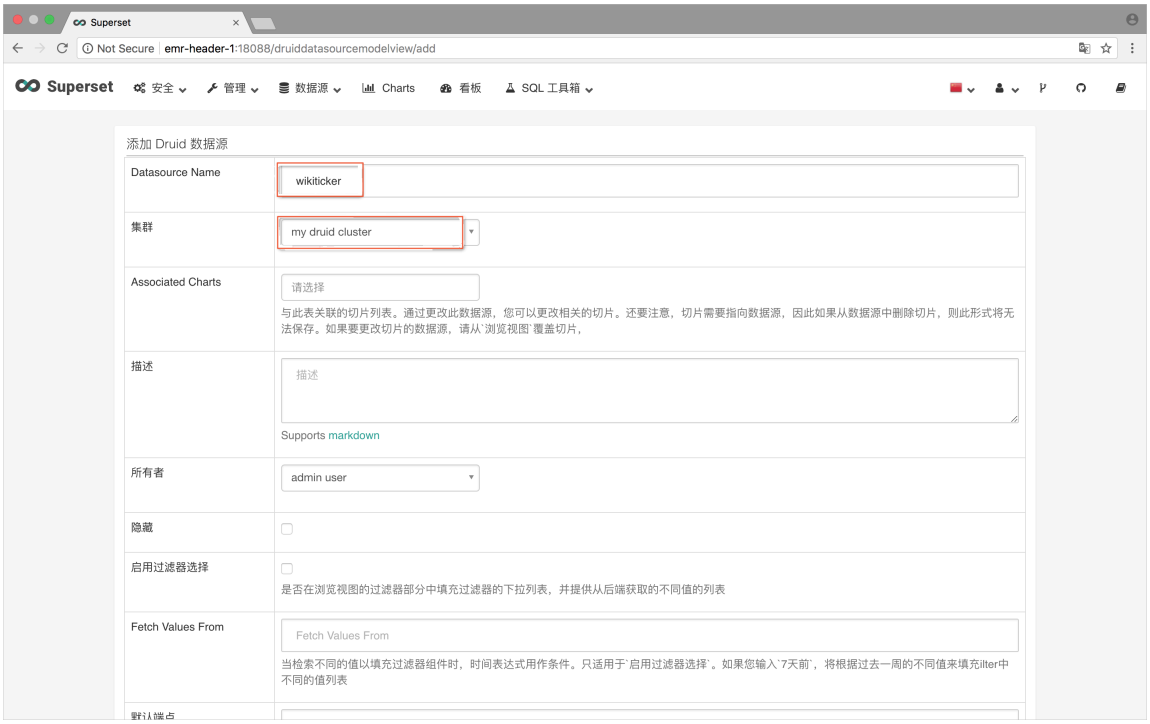
3. 刷新或者添加新数据源。

   添加好E-MapReduce Druid集群之后，您可以单击**数据源** > **扫描新的数据源**，这时E-MapReduce Druid集群上的数据源（datasource）就可以自动被加载进来。

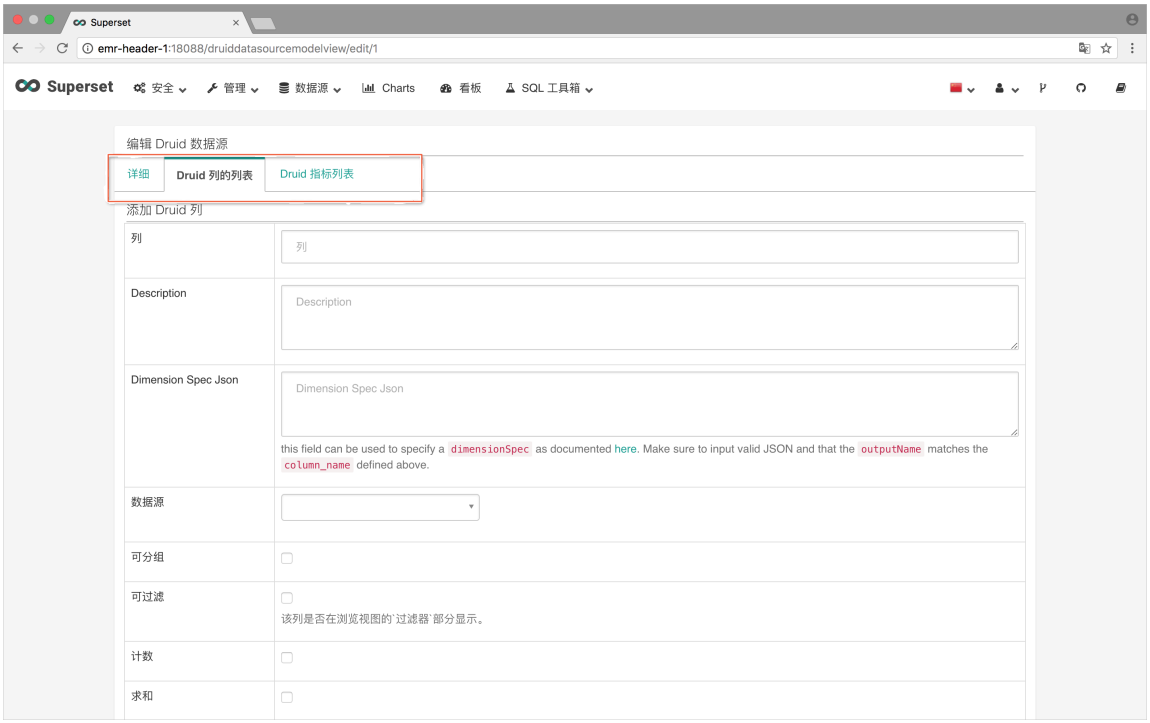
   您也可以在界面上单击**数据源** > **Druid 数据源**自定义新的数据源（其操作等同于写一个data source ingestion的json文件），步骤如下。



   a. 自定义数据源时需要填写必要的信息，然后保存。

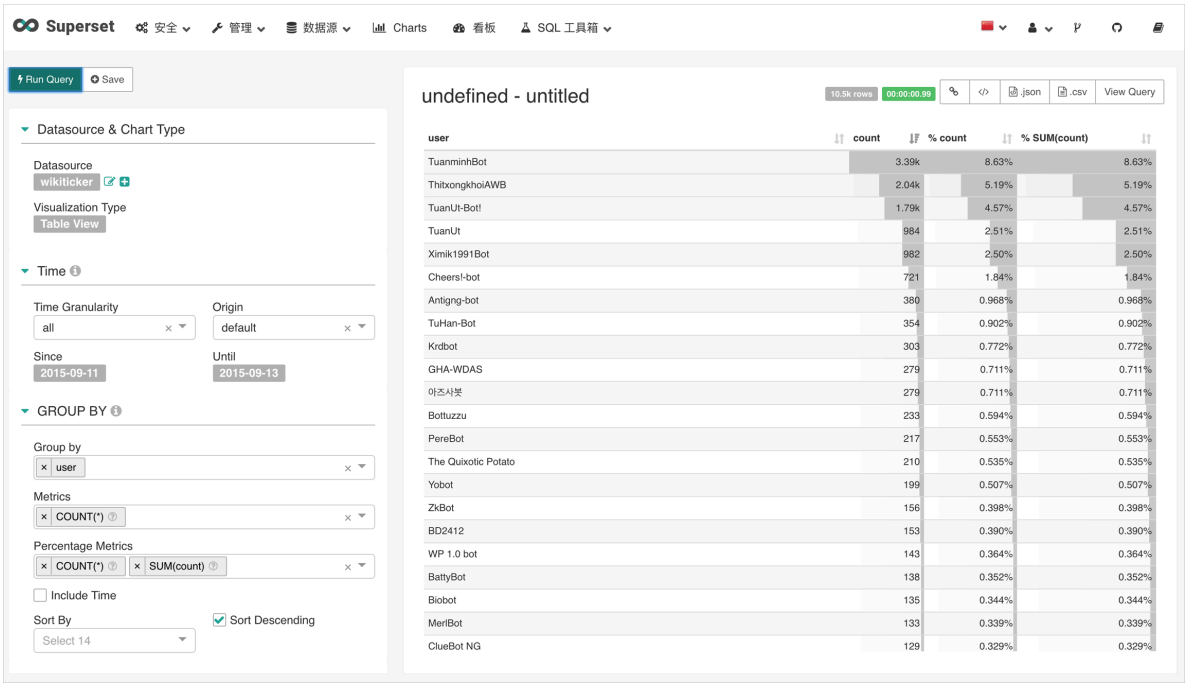


   b. 保存之后单击左侧  ，编辑该数据源，填写相应的维度列与指标列等信息。



4. 查询E-MapReduce Druid。

数据源添加成功后，单击数据源名称，进入查询页面进行查询。

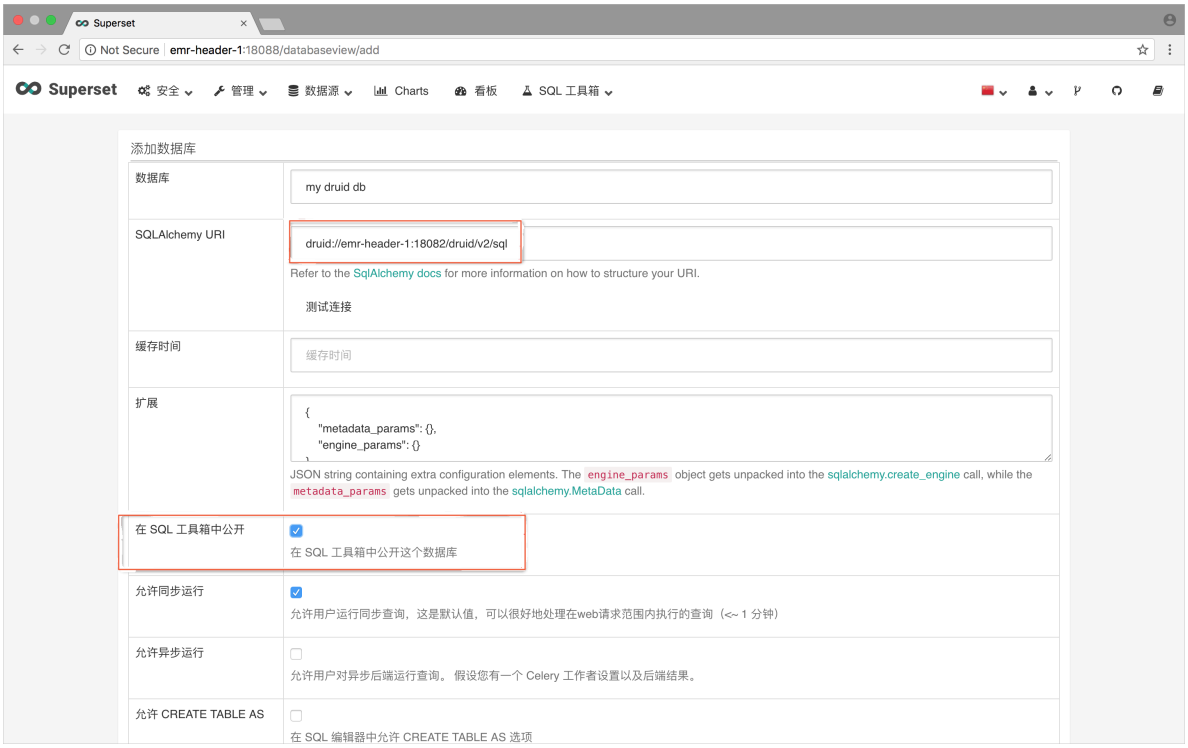


5. （可选）将E-MapReduce Druid作为E-MapReduce Druid数据库使用。

   Superset提供了SQLAlchemy以多种方言支持各种各样的数据库，其支持的数据库类型如下表所示。

database	pypi package	SQLAlchemy URI prefix
MySQL	`pip install mysqlclient`	`mysql://`
Postgres	`pip install psycopg2`	`postgresql+psycopg2://`
Presto	`pip install pyhive`	`presto://`
Oracle	`pip install cx_Oracle`	`oracle://`
sqlite		`sqlite://`
Redshift	`pip install sqlalchemy-redshift`	`redshift+psycopg2://`
MSSQL	`pip install pymssql`	`mssql://`
Impala	`pip install impyla`	`impala://`
SparkSQL	`pip install pyhive`	`jdbc+hive://`
Greenplum	`pip install psycopg2`	`postgresql+psycopg2://`
Athena	`pip install "PyAthenaJDBC>1.0.9"`	`awsathena+jdbc://`
Vertica	`pip install sqlalchemy-vertica-python`	`vertica+vertica_python://`
ClickHouse	`pip install sqlalchemy-clickhouse`	`clickhouse://`
Kylin	`pip install kylinpy`	`kylin://`

   Superset亦支持该方式访问E-MapReduce Druid，E-MapReduce Druid对应的 SQLAlchemy URI为druid://emr-header-1:18082/druid/v2/sql，如下图所示，将E-MapReduce Druid作为一个数据库添加。



接下来就可以在SQL工具箱里用SQL进行查询了。

10.7 常见问题

本文介绍E-MapReduce Druid使用过程中遇到的一些常见问题以及解决方法。

索引失败问题分析思路

当发现索引失败时，一般遵循如下排错思路：

- 对于批量索引

1. 如果 curl 直接返回错误，或者不返回，检查一下输入文件格式。或者 curl 加上 **-v** 参数，观察 REST API 的返回情况。
2. 在 Overlord 页面观察作业执行情况，如果失败，查看页面上的 logs。
3. 在很多情况下并没有生成 logs。如果是 Hadoop 作业，打开 YARN 页面查看是否有索引作业生成，并查看作业执行 log。
4. 如果上述情况都没有定位到错误，需要登录到 E-MapReduce Druid 集群，查看 overlord 的执行日志（位于 /mnt/disk1/log/druid/overlord—emr-header-1.cluster-xxxx.log），如果是 HA 集群，查看您提交作业的那个 Overlord。
5. 如果作业已经被提交到 Middlemanager，但是从 Middlemanager 返回了失败，则需要从 Overlord 中查看作业提交到了哪个 worker，并登录到相应的 worker，查看

Middlemanager 的日志（位于 /mnt/disk1/log/druid/middleManager-emr-header-1.cluster-xxxx.log）。

- 对于 Tranquility 实时索引

  查看 Tranquility log，查看消息是否被接收到了或者是否被丢弃（drop）掉了。

  其余的排查步骤同批量索引的 2 ~ 5。

  错误多数情况为集群配置问题和作业问题。集群配置问题包括：内存参数是否合理、跨集群联通性是否正确、安全集群访问是否通过、principal 是否正确等等，作业问题包括作业描述文件格式是否正确、输入数据是否能够正常被解析，以及一些其他的作业相关的配置（如 ioConfig 等）。

常见问题

- 组件启动失败

  此类问题多数是由于组件 JVM 运行参数配置问题，例如机器可能没有很大的内存，而配置了较大的 JVM 内存或者较多的线程数量。

  解决方法：查看组件日志并调整相关参数即可解决。JVM 内存涉及堆内存和直接内存。具体可参见Apache Druid 官方文档。

- 索引时 YARN task 执行失败，显示诸如 `Error: class com.fasterxml.jackson.datatype.guava.deser.HostAndPortDeserializer overrides final method deserialize.(Lcom/fasterxml/jackson/core/JsonParser;Lcom/fasterxml/jackson/databind/DeserializationContext;)Ljava/lang/Object;` 之类的 jar 包冲突错误。

  解决方法：在 indexing 的作业配置文件中加入如下配置。

  ```
  "tuningConfig" : {
     ...
     "jobProperties" : {
        "mapreduce.job.classloader": "true"
        或者
        "mapreduce.job.user.classpath.first": "true"
     }
     ...
  }
  ```

  其中参数 **mapreduce.job.classloader** 让 MR job 用独立的 classloader，**mapreduce.job.user.classpath.first** 是让 MapReduce 优先使用用户的 jar 包，两个配置项配置一个即可。 可参见 Apache Druid 官方文档。

- indexing 作业的日志中报 reduce 无法创建 segments 目录。

  解决方法：

  - 注意检查 deep storage 的设置，包括 type 和 directory。当 type 为 local 时，注意 directory 的权限设置。当 type 为 HDFS 时，directory 尽量用完整的 HDFS 路径写法，如

hdfs://:9000/。hdfs_master 最好用 IP，如果用域名，要用完整域名，如 emr-header-1.cluster-xxxxxxxx，而不是 emr-header-1。

- 用 Hadoop 批量索引时，要将 segments 的 deep storage 设置为 “hdfs”，“local” 的方式会导致 MR 作业处于 UNDEFINED 状态，这是因为远程的 YARN 集群无法在 reduce task 下创建 local 的 segments 目录。（此针对独立 E-MapReduce Druid 集群）。

- Failed to create directory within 10000 attempts...

  此问题一般为 JVM 配置文件中 java.io.tmp 设置的路径不存在的问题。设置该路径并确保 E-MapReduce Druid 账户有权限访问即可。

- com.twitter.finagle.NoBrokersAvailableException: No hosts are available for disco!firehose:druid:overlord

  此问题一般是 ZooKeeper 的连接问题。确保 E-MapReduce Druid 与 Tranquility 对于 ZooKeeper 有相同的连接字符串。注意：E-MapReduce Druid 默认的ZooKeeper路径为 /druid，因此确保 Tranquility 设置中 zookeeper.connect 包含路径 /druid。（另注意 Tranquility Kafka 设置中有两个 ZooKeeper 的设置，一个为 zookeeper.connect，连接E-MapReduce Druid 集群的 ZooKeeper，一个为 kafka.zookeeper.connect，连接Kafka集群的 ZooKeeper。这两个 ZooKeeper 可能不是一个 ZooKeeper 集群）。

- 索引时 MiddleManager 报找不到类 com.hadoop.compression.lzo.LzoCodec。

  这是因为 EMR 的 Hadoop 集群配置了 lzo 压缩。

  解决方法：拷贝 EMR HADOOP_HOME/lib 下的 jar 包和 native 文件夹到 E-MapReduce Druid 的 druid.extensions.hadoopDependenciesDir（默认为 DRUID_HOME/hadoop-dependencies）。

- 索引时报如下错误：

```
2018-02-01T09:00:32,647 ERROR [task-runner-0-priority-0] com.hadoop.compression.lzo.GPLNativeCodeLoader - could not unpack the binaries
 java.io.IOException: No such file or directory
        at java.io.UnixFileSystem.createFileExclusively(Native Method) ~[?:1.8.0_151]
        at java.io.File.createTempFile(File.java:2024) ~[?:1.8.0_151]
        at java.io.File.createTempFile(File.java:2070) ~[?:1.8.0_151]
        at com.hadoop.compression.lzo.GPLNativeCodeLoader.unpackBinaries(GPLNativeCodeLoader.java:115) [hadoop-lzo-0.4.21-SNAPSHOT.jar:?]
```

  这个问题还是因为 java.io.tmp 路径不存在的问题。设置该路径并确保 E-MapReduce Druid 账户有权限访问。

11 Delta

11.1 EMR Delta简介

Delta Lake是DataBricks公司推出的一种数据湖方案，Delta为该方案核心组件。Delta以数据为中心，围绕数据流走向（数据从流入数据湖，数据组织管理，数据查询到流出数据湖）推出了一系列功能特性，协助您搭配第三方上下游工具，搭建快捷、易用、安全的数据湖。

背景信息

通常的数据湖方案是选取大数据存储引擎构建数据湖（如阿里云OSS等对象存储产品或云下HDFS），然后将产生的各种类型数据存储在该存储引擎中。在使用数据时，通过Spark或Presto对接数据分析引擎并进行数据解析。但该套方案存在如下问题：

- 数据导入可能会失败，失败后脏数据清理和作业恢复困难。
- 方案中没有ETL（Extract Transform Load）过程，缺少必要的数据质量监管。
- 方案中没有事务将读和写隔离，致使流式和批式读写无法相互隔离。

Delta完美地解决了该问题：

- 在大数据存储层之上提供了数据管理层，该数据管理层等同于数据库中的元数据管理，其元数据随着数据一起存放并对用户可见（如图 11-1: 数据仓库与数据湖所示）。
- Delta基于元数据管理引入了ACID，解决了因数据导入失败而产生脏数据和数据导入时的读写隔离问题。
- 元数据存储了数据的字段信息，Delta提供了数据导入时数据校验功能，保证数据质量。
- 事务功能使得批式读写和流式读写能够互相隔离。

说明：

ACID指数据库事务正确执行的四个基本要素的缩写。包含：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。

图 11-1: 数据仓库与数据湖

Data Warehouse、Data Lake和Delta Lak对比如下所示。

对比项	Data Warehouse	Data Lake	Delta Lake
架构	计算存储一体或分离	计算存储分离	计算存储分离
存储管理	严格、非通用	原生格式	通用格式、轻量级
场景	报表、分析	报表、分析、数据科学	报表、分析、数据科学
灵活性	低	高	较高
数据质量和可靠性	很高	低	较高
事务性	支持	不支持	支持
性能	高	低	较高
扩展性	依赖于具体实现	高	高
面向人员	管理人员	管理人员、数据科学家	管理人员、数据科学家
成本	高	低	低

应用场景

Delta适用于云上数据湖数据管理解决方案。如果您存在以下场景，可使用Delta：

- 实时查询：数据实时从上游流入Delta，查询侧可即查询该数以立据。同时，由于有ACID的支持，保证了数据的流入和查询的隔离性，不产生脏读数据。
- 删除或更新，GDPR（General Data Protection Regulation）：通常数据湖方案不支持数据的删除或更新。如果需要删除或更新数据，则需要把原始数据清理掉，然后把更新后的数据写入存储。而Delta支持数据的删除或更新。

- 数据实时同步，CDC（Change Data Capture）：使用Delta merge功能，启动流作业，实时将上游的数据通过merge更新到Delta Lake。
- 数据质量控制：借助于Delta Schema校验功能，在数据导入时剔除异常数据，或者对异常数据做进一步处理。
- 数据演化：数据的Schema并非固定不变，Delta支持通过API 方式改变数据的Schema。
- 实时机器学习：在机器学习场景中，通常需要花费大量的时间用于处理数据，如数据清洗、转换、提取特征等等。同时，您还需要对历史和实时数据分别处理。而Delta简化了工作流程，整条数据处理过程是一条完整的、可靠的实时流，其数据的清洗、转换、特征化等操作都是流上的节点动作，无需对历史和实时数据分别处理。

EMR-Delta

EMR-Delta丰富了开源Delta的特性，例如对SQL和Optimize的支持等。下表列出了Delta Lake的基本特性，并将EMR-Delta与开源Delta（0.5.0）做了对比。

特性	EMR-Delta	开源Delta
SQL	• ALTER • CONVERT • CREATE • CTAS • DELETE • DESC HISTORY • INSERT • MERGE • OPTIMIZE • UPDATE • VACUUM	• CREATE **说明：** 建表示例：CREATE TABLE <tbl> USING delta LOCATION <delta_table_path> - 仅支持基于已有的Delta目录建表。 - 建表时请不要指定Schema。 • CONVERT • DESC HISTORY • VACUUM
API	• batch read/write • streaming read/write • optimize • delete • update • merge • convert • history • vacuum	• batch read/write • streaming read/write • delete • update • merge • convert • history • vacuum
Hive connector	支持	支持

特性	EMR-Delta	开源Delta
Presto connector	支持	支持
Parquet	支持	支持
ORC	不支持	不支持
文本格式	不支持	不支持
Data Skipping	不支持	不支持
ZOrder	支持	不支持
Native DeltaLog	支持	不支持

11.2 快速入门

本文介绍Delta Lake一些基础的使用示例。

建表并写入数据

- Scala

```
val data = spark.range(0, 5)
data.write.format("delta").save("/tmp/delta_table")
```

- SQL

```
CREATE TABLE delta_table (id INT) USING delta LOCATION "/tmp/delta_table";
INSERT INTO delta_table VALUES 0,1,2,3,4;
```

读表

- Scala

```
val df = spark.read.format("delta").load("/tmp/delta_table")
df.show()
```

- SQL

```
SELECT * FROM delta_table;
```

覆盖写数据

- Scala

```
val data1 = spark.range(5, 10)
data1.write.format("delta").mode("overwrite").save("/tmp/delta_table")
df.show()
```

- SQL

```
INSERT OVERWRITE TABLE delta_table VALUES 5,6,7,8,9;
```

```
SELECT * FROM delta_table;
```

DELETE/UPDATE/MERGE

- Scala

```
import io.delta.tables._
import org.apache.spark.sql.functions._

val deltaTable = DeltaTable.forPath("/tmp/delta_table")

// Update every even value by adding 100 to it
deltaTable.update(
  condition = expr("id % 2 == 0"),
  set = Map("id" -> expr("id + 100")))

deltaTable.toDF.show()

// Delete every even value
deltaTable.delete(condition = expr("id % 2 == 0"))

deltaTable.toDF.show()

// Upsert (merge) new data
val newData = spark.range(0, 10).toDF

deltaTable.as("oldData")
  .merge(
    newData.as("newData"),
    "oldData.id = newData.id")
  .whenMatched
  .updateExpr(Map("id" -> "newData.id + 100"))
  .whenNotMatched
  .insertExpr(Map("id" -> "newData.id"))
  .execute()

deltaTable.toDF.show()
```

- SQL

```
UPDATE delta_table SET id = id + 100 WHERE mod(id,2) = 0;
SELECT * FROM delta_table;

DELETE FROM delta_table WHERE mod(id,2) = 0;
SELECT * FROM delta_table;

CREATE TABLE newData(id INT) USING delta LOCATION "/tmp/newData";
INSERT INTO newData VALUES 0,1,2,3,4,5,6,7,8,9;

MERGE INTO delta_table AS target
  USING newData AS source
  ON target.id = source.id
  WHEN MATCHED THEN UPDATE SET target.id = source.id + 100
  WHEN NOT MATCHED THEN INSERT *;
```

```
SELECT * FROM delta_table;
```

流式读

- Scala

```
val stream1 = spark.readStream.format("delta").load("/tmp/delta_table2").writeStrea
m.format("console").start()
```

- SQL

```
CREATE SCAN stream_delta_table on delta_table USING STREAM;

CREATE STREAM job
INSERT INTO stream_debug_table
SELECT *
FROM stream_delta_table;
```

流式写

- Scala

```
val streamingDf = spark.readStream.format("kafka")
  .option(kafka.bootstrap.servers="${BOOTSTRAP_SERVERS}", subscribe ="${
TOPIC_NAME}").load()
val stream = streamingDf.select(s"CAST(value AS STRING)" as "id")
  .writeStream.format("delta").option("checkpointLocation", "/tmp/checkpoint").start
("/tmp/delta_table")
```

- SQL

```
CREATE TABLE IF NOT EXISTS kafka_topic
USING kafka
OPTIONS (
kafka.bootstrap.servers = "${BOOTSTRAP_SERVERS}",
subscribe = "${TOPIC_NAME}"
);

CREATE SCAN stream_kafka_topic on kafka_topic USING STREAM;

CREATE STREAM job
OPTIONS(
checkpointLocation='/tmp/'
)
INSERT INTO delta_table
```

```
SELECT CAST(value AS STRING) AS id FROM stream_kafka_topic;
```

11.3 应用场景

11.3.1 场景一：流式入库

目前支持流式入库的系统都基本遵循了一个思路，流式数据按照小批量数据写小文件到存储系统，然后定时对这些文件进行合并，从HIVE到Delta Lake无一例外（Kudu也可以做到流式入库，但是Kudu的存储是自己设计的，不属于前述的基于大数据存储系统之上的解决方案）。

流式入库演变

阶段	详细情况
以前	以前针对流式入库的需求，一般都是自己动手，事实表按照时间划分Partition，粒度比较细。例如，五分钟一个Partition，每当一个 Partition运行完成，触发一个INSERT OVERWRITE动作，将该Partition内的文件合并重新写入分区。但是这么做有以下几个问题： • 缺少读写隔离，易造成读端失败或者产生数据准确性问题。 • 流式作业没有Exactly-Once保证，入库作业失败后需要人工介入，确保数据不会写重或者写漏（如果是SparkStreaming，有At-Least-Once保证）。 HIVE从0.13版本提供了事务支持，并且从2.0版本开始提供了HIVE Streaming功能来实现流式入库的支持。但是在实际使用HIVE Streaming功能的案例并不多见。其主要原因如下： • HIVE事务的实现修改了底层文件，导致公共的存储格式等仅能够被HIVE读取，导致很多使用SparkSQL、Presto等进行数据分析的用户无法使用该功能。 • HIVE事务目前仅支持ORC。 • HIVE的模式为Merge-on-read，需要对小文件进行Sort-Merge。小文件数量增多之后读性能急剧下降，所以用户需要及时进行小文件的合并。而小文件的合并作业经常失败，影响用户业务效率。 • HIVE这种模式无法拓展到Data Lake场景，仅仅停留在Data Warehouse场景。在Data Lake场景中，数据来源以及数据需求都是多样性的。

阶段	详细情况
现在	有了Delta，可以很方便地应对流式入库的场景。只需要以下四个动作： **1.** 建表。 **2.** 启动Spark Streaming任务写入数据。 **3.** 定时Optimize（例如：每个Partition写入完成）。 **4.** 定时Vacuum（例如：每天）。

Delta实例展示

从上游Kafka中读取数据，写入Delta表。上游Kafka准备一个Python脚本，不断向Kafka内发送数据。

```
#! /usr/bin/env python3

import json
import time

from kafka import KafkaProducer
from kafka.errors import KafkaError

bootstrap = ['emr-header-1:9092']
topic = 'delta_stream_sample'

def gnerator():
    id = 0
    line = {}
    while True:
        line['id'] = id
        line['date'] = '2019-11-11'
        line['name'] = 'Robert'
        line['sales'] = 123
        yield line
        id = id + 1

def sendToKafka():
    producer = KafkaProducer(bootstrap_servers=bootstrap)

    for line in gnerator():
        data = json.dumps(line).encode('utf-8')

        # Asynchronous by default
        future = producer.send(topic, data)

        # Block for 'synchronous' sends
        try:
            record_metadata = future.get(timeout=10)
        except KafkaError as e:
            # Decide what to do if produce request failed
            pass
        time.sleep(0.1)
```

```
sendToKafka()
```

为了方便，数据只有id不一样。

```
{"id": 0, "date": "2019-11-11", "name": "Robert", "sales": 123}
{"id": 1, "date": "2019-11-11", "name": "Robert", "sales": 123}
{"id": 2, "date": "2019-11-11", "name": "Robert", "sales": 123}
{"id": 3, "date": "2019-11-11", "name": "Robert", "sales": 123}
{"id": 4, "date": "2019-11-11", "name": "Robert", "sales": 123}
{"id": 5, "date": "2019-11-11", "name": "Robert", "sales": 123}
```

启动一个Spark Streaming作业，从Kafka读数据，写入Delta表。

- Scala
  - bash

    ```
    spark-shell --master local --use-emr-datasource
    ```

  - scala

    ```
    import org.apache.spark.sql.{functions, SparkSession}
    import org.apache.spark.sql.types.DataTypes
    import org.apache.spark.sql.types.StructField

    val targetDir = "/tmp/delta_table"
    val checkpointLocation = "/tmp/delta_table_checkpoint"
    val bootstrapServers = "192.168.XX.XX:9092"
    val topic = "delta_stream_sample"

    val schema = DataTypes.createStructType(Array[StructField](
      DataTypes.createStructField("id", DataTypes.LongType, false),
      DataTypes.createStructField("date", DataTypes.DateType, false),
      DataTypes.createStructField("name", DataTypes.StringType, false),
      DataTypes.createStructField("sales", DataTypes.StringType, false)))

    val lines = spark
        .readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", bootstrapServers)
        .option("subscribe", topic)
        .option("maxOffsetsPerTrigger", 1000)
        .option("startingOffsets", "earliest")
        .option("failOnDataLoss", value = false)
        .load()
        .select(functions.from_json(functions.col("value").cast("string"), schema).as("
    json"))
        .select("json.*")

    val query = lines.writeStream
        .outputMode("append")
        .format("delta")
        .option("checkpointLocation", checkpointLocation)
        .start(targetDir)
    ```

```
query.awaitTermination()
```

- SQL
  - bash

```
streaming-sql --master local --use-emr-datasource
```

  - SQL

```
CREATE TABLE IF NOT EXISTS kafka_table
USING kafka
OPTIONS(
kafka.bootstrap.servers='192.168.XX.XX:9092',
subscribe='delta_stream_sample'
);

CREATE TABLE IF NOT EXISTS delta_table (id LONG, `date` DATE, name STRING, sales
STRING)
USING delta
LOCATION '/tmp/delta_table';

CREATE SCAN stream_kafka_table on kafka_table USING STREAM
OPTIONS(
maxOffsetsPerTrigger='1000',
startingOffsets='earliest',
failOnDataLoss=false
);

CREATE STREAM job
OPTIONS(
checkpointLocation='/tmp/delta_table_checkpoint'
)
INSERT INTO delta_table
SELECT
   content.id as id,
   content.date as date,
   content.name as name,
   content.sales as sales
FROM (
   SELECT from_json(CAST(value as STRING), 'id LONG, `date` DATE, name STRING,
sales STRING') as content
   FROM stream_kafka_table
);
```

另新建一个spark-shell，确认已经读到数据。

- Scala

```
val df = spark.read.format("delta").load("/tmp/delta_table")
df.select("*").orderBy("id").show(10000)
```

- SQL

```
SELECT * FROM delta_table ORDER BY id LIMIT 10000;
```

现在已经写入了2285条数据。

```
|2295|2019-11-11|Robert|  123|
```

```
|2296|2019-11-11|Robert|  123|
|2297|2019-11-11|Robert|  123|
|2275|2019-11-11|Robert|  123|
|2276|2019-11-11|Robert|  123|
|2277|2019-11-11|Robert|  123|
|2278|2019-11-11|Robert|  123|
|2279|2019-11-11|Robert|  123|
|2280|2019-11-11|Robert|  123|
|2281|2019-11-11|Robert|  123|
|2282|2019-11-11|Robert|  123|
|2283|2019-11-11|Robert|  123|
|2284|2019-11-11|Robert|  123|
|2285|2019-11-11|Robert|  123|
+----+----------+------+-----+
```

Exactly-Once测试

将Spark Streaming作业停掉，再重新启动。重新读一下表，读数据正常的话，数据能够从上次断掉的地方衔接上。

- Scala

```
df.select("*").orderBy("id").show(10000)
```

- SQL

```
SELECT * FROM delta_table ORDER BY id LIMIT 10000;
```

```
|2878|2019-11-11|Robert|  123|
|2879|2019-11-11|Robert|  123|
|2880|2019-11-11|Robert|  123|
|2881|2019-11-11|Robert|  123|
|2882|2019-11-11|Robert|  123|
|2883|2019-11-11|Robert|  123|
|2884|2019-11-11|Robert|  123|
|2885|2019-11-11|Robert|  123|
|2886|2019-11-11|Robert|  123|
|2887|2019-11-11|Robert|  123|
|2888|2019-11-11|Robert|  123|
|2889|2019-11-11|Robert|  123|
|2890|2019-11-11|Robert|  123|
|2891|2019-11-11|Robert|  123|
```

11.3.2 场景二：数据同步

数据同步是指数仓或者数据湖内的数据与上游业务库内的数据保持同步的状态。当上游业务库内的数据发生变更之后，下游的数仓/数据湖立即感知到数据变化，并将数据变化同步过来。在数据库中，这类场景称为Change Data Capture（CDC）场景。

背景信息

CDC的实现方案比较多，但是大多是在数据库领域，相应的工具也比较多。在大数据领域，这方面的实践较少，也缺乏相应的标准和技术实现。通常您需要选择已有的引擎，利用它们的能力自己搭建一套CDC方案。常见的方案大概分为下面两类：

- 定期批量Merge方式：上游原始表捕获增量更新，将更新的数据输出到一个新的表中，下游仓库利用MERGE或UPSERT语法将增量表与已有表进行合并。这种方式要求表具有主键或者联合主键，且实时性也较差。另外，这种方法一般不能处理DELETE的数据，实际上用删除原表重新写入的方式支持了DELETE，但是相当于每次都重新写一次全量表，性能不可取，还需要有一个特殊字段来标记数据是否属于增量更新数据。
- 上游源表输出binlog（这里我们指广义的binlog，不限于MySQL），下游仓库进行binlog的回放。这种方案一般需要下游仓库能够具有实时回放的能力。但是可以将row的变化作为binlog输出，这样，只要下游具备INSERT、UPDATE、DELETE的能力就可以了。不同于第一种方案，这种方案可以和流式系统结合起来。binlog可以实时地流入注入Kafka的消息分发系统，下游仓库订阅相应的Topic，实时拉取并进行回放。

批量更新方式

此方案适用于没有Delete且实时性要求不那么高的场景。

1. 建立一张MySQL表，插入一部分数据。

```
CREATE TABLE sales(id LONG, date DATE, name VARCHAR(32), sales DOUBLE, modified DATETIME);

INSERT INTO sales VALUES (1, '2019-11-11', 'Robert', 323.00, '2019-11-11 12:00:05'), (2, '2019-11-11', 'Lee', 500.00, '2019-11-11 16:11:46'), (3, '2019-11-12', 'Robert', 136.00, '2019-11-12 10:23:54'), (4, '2019-11-13', 'Lee', 211.00, '2019-11-13 11:33:27');

SELECT * FROM sales;
```

```
+------+------------+--------+-------+---------------------+
| id   | date       | name   | sales | modified            |
+------+------------+--------+-------+---------------------+
| 1    | 2019-11-11 | Robert |   323 | 2019-11-11 12:00:05 |
| 2    | 2019-11-11 | Lee    |   500 | 2019-11-11 16:11:46 |
| 3    | 2019-11-12 | Robert |   136 | 2019-11-12 10:23:54 |
| 4    | 2019-11-13 | Lee    |   211 | 2019-11-13 11:33:27 |
+------+------------+--------+-------+---------------------+
```

说明：

modified就是我们上文提到的用于标识数据是否属于增量更新数据的字段。

2. 将MySQL表的内容全量导出到HDFS。

```
sqoop import --connect jdbc:mysql://emr-header-1:3306/test --username root --password EMRroot1234 -table sales -m1 --target-dir /tmp/cdc/staging_sales

hdfs dfs -ls /tmp/cdc/staging_sales
Found 2 items
-rw-r-----   2 hadoop hadoop        0 2019-11-26 10:58 /tmp/cdc/staging_sales/_SUCCESS
```

```
-rw-r-----   2 hadoop hadoop       186 2019-11-26 10:58 /tmp/cdc/staging_sales/part
-m-00000
```

3. 建立delta表，并导入MySQL表的全量数据。

```
-- `LOAD DATA INPATH`语法对delta table不可用，先建立一个临时外部表。
CREATE TABLE staging_sales (id LONG, date STRING, name STRING, sales DOUBLE,
modified STRING) USING csv LOCATION '/tmp/cdc/staging_sales/';

CREATE TABLE sales USING delta LOCATION '/user/hive/warehouse/test.db/test'
SELECT * FROM staging_sales;

SELECT * FROM sales;

1   2019-11-11   Robert   323.0   2019-11-11 12:00:05.0
2   2019-11-11   Lee   500.0   2019-11-11 16:11:46.0
3   2019-11-12   Robert   136.0   2019-11-12 10:23:54.0
4   2019-11-13   Lee   211.0   2019-11-13 11:33:27.0

--删除临时表。
DROP TABLE staging_sales;
```

切换到命令行删除临时目录。

```
hdfs dfs -rm -r -skipTrash /tmp/cdc/staging_sales/  # 删除临时目录。
```

4. 在原MySQL表做一些操作，插入更新部分数据。

```
-- 注意DELETE的数据无法被Sqoop导出，因而没办法合并到目标表中
-- DELETE FROM sales WHERE id = 1;
UPDATE sales SET name='Robert',modified=now() WHERE id = 2;
INSERT INTO sales VALUES (5, '2019-11-14', 'Lee', 500.00, now());

SELECT * FROM sales;
```

```
+------+------------+--------+-------+--------------------+
|id  |date      |name  |sales|modified          |
+------+------------+--------+-------+--------------------+
|1   |2019-11-11|Robert|  323|2019-11-11 12:00:05|
|2   |2019-11-11|Robert|  500|2019-11-26 11:08:34|
|3   |2019-11-12|Robert|  136|2019-11-12 10:23:54|
|4   |2019-11-13|Lee   |  211|2019-11-13 11:33:27|
|5   |2019-11-14|Lee   |  500|2019-11-26 11:08:38|
+------+------------+--------+-------+--------------------+
```

5. sqoop导出更新数据。

```
sqoop import --connect jdbc:mysql://emr-header-1:3306/test --username root --
password EMRroot1234 -table sales -m1 --target-dir /tmp/cdc/staging_sales --
incremental lastmodified --check-column modified --last-value "2019-11-20 00:00:00
"


hdfs dfs -ls /tmp/cdc/staging_sales/
Found 2 items
-rw-r-----   2 hadoop hadoop         0 2019-11-26 11:11 /tmp/cdc/staging_sales/
_SUCCESS
```

```
-rw-r-----   2 hadoop hadoop       93 2019-11-26 11:11 /tmp/cdc/staging_sales/part-m-00000
```

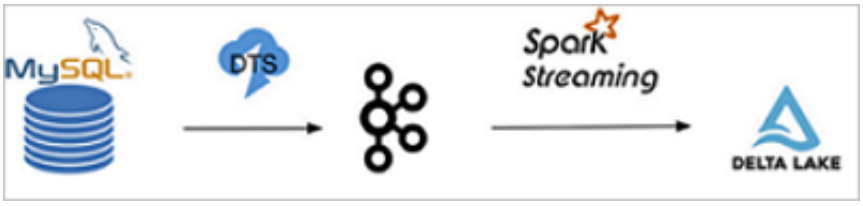
6. 为更新数据建立临时表，然后MERGE到目标表。

```
CREATE TABLE staging_sales (id LONG, date STRING, name STRING, sales DOUBLE, modified STRING) USING csv LOCATION '/tmp/cdc/staging_sales/';

MERGE INTO sales AS target
USING staging_sales AS source
ON target.id = source.id
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *;

SELECT * FROM sales;

1   2019-11-11   Robert   323.0   2019-11-11 12:00:05.0
3   2019-11-12   Robert   136.0   2019-11-12 10:23:54.0
2   2019-11-11   Robert   500.0   2019-11-26 11:08:34.0
5   2019-11-14   Lee   500.0   2019-11-26 11:08:38.0
4   2019-11-13   Lee   211.0   2019-11-13 11:33:27.0
```

实时同步

实时同步的方式对场景的限制没有第一种方式多，例如，DELETE数据也能处理，不需要修改业务模型增加一个额外字段。但是这种方式实现较为复杂，如果binlog的输出不标准的话，您还需要写专门的UDF来处理binlog数据。例如RDS MySQL输出的binlog，以及Log Service输出的binlog格式上就不相同。

在这个例子中，我们使用阿里云RDS MySQL版作为源库，使用阿里云DTS服务将源库的binlog数据实时导出到Kafka集群，您也可以选择开源的Maxwell或Canal等。 之后我们定期从Kafka读取binlog并存放到OSS或HDFS，然后用Spark读取该binlog并解析出Insert、Update、Delete的数据，最后用Delta的Merge API将源表的变动更新到Delta表，其链路如下图所示。



1. 首先开通RDS MySQL服务，设置好相应的用户、Database和权限（RDS的具体使用请参见#unique_45）。建立一张表并插入一些数据。

```
-- 该建表动作可以在RDS控制台页面方便地完成，这里展示最后的建表语句。
CREATE TABLE `sales` (
  `id` bigint(20) NOT NULL,
  `date` date DEFAULT NULL,
  `name` varchar(32) DEFAULT NULL,
  `sales` double DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8

-- 插入部分数据，并确认。
```

```
INSERT INTO sales VALUES (1, '2019-11-11', 'Robert', 323.00), (2, '2019-11-11', 'Lee', 500.00), (3, '2019-11-12', 'Robert', 136.00), (4, '2019-11-13', 'Lee', 211.00);

SELECT * FROM `sales` ;
```


	id	date	name	sales
1	1	2019-11-11	Robert	323
2	2	2019-11-11	Lee	500
3	3	2019-11-12	Robert	136
4	4	2019-11-13	Lee	211

2. 建立一个EMR Kafka集群（如果已有EMR Kafka集群的话请跳过），并在Kafka集群上创建一个名为sales的topic：

   bash

   ```
   kafka-topics.sh --create --zookeeper emr-header-1:2181,emr-header-2:2181,emr-header-3:2181 --partitions 1 --replication-factor 1 --topic sales
   ```

3. 开通DTS服务（如果未开通的话），并创建一个同步作业，源实例选择RDS MySQL，目标实例选择Kafka。
4. 配置DTS的同步链路，将RDS的sales table同步至Kafka，目标topic选择sales。正常的话，可以在Kafka的机器上看到数据。
5. 编写Spark Streaming作业，从Kafka中解析binlog，利用Delta的MERGE API将binlog数据实时回放到目标Delta表。DTS导入到Kafka的binlog数据的样子如下，其每一条记录都表示了一条数据库数据的变更。 详情请参见附录：Kafka内binlog格式窥探。

```
|字段名称|值|
|:--|:--|
|recordid|1|
|source|{"sourceType": "MySQL", "version": "0.0.0.0"}|
|dbtable|delta_cdc.sales|
|recordtype|INIT|
|recordtimestamp|1970-01-01 08:00:00|
|extratags|{}|
|fields|["id","date","name","sales"]|
|beforeimages|{}|
|afterimages|{"sales":"323.0","date":"2019-11-11","name":"Robert","id":"1"}|
```

说明：

这里最重要的字段是recordtype、beforeimages、afterimages。其中recordtype是该行记录对应的动作，包含INIT、UPDATE、DELETE、INSERT几种。beforeimages为该动作执行前的内容，afterimages为动作执行后的内容。

- Scala
  - bash

```
spark-shell --master yarn --use-emr-datasource
```

  - scala

```
import io.delta.tables._
import org.apache.spark.internal.Logging
import org.apache.spark.sql.{AnalysisException, SparkSession}
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types.{DataTypes, StructField}

val schema = DataTypes.createStructType(Array[StructField](
  DataTypes.createStructField("id", DataTypes.StringType, false),
  DataTypes.createStructField("date", DataTypes.StringType, true),
  DataTypes.createStructField("name", DataTypes.StringType, true),
  DataTypes.createStructField("sales", DataTypes.StringType, true)
))

//初始化delta表中INIT类型的数据。
def initDeltaTable(): Unit = {
  spark.read
      .format("kafka")
      .option("kafka.bootstrap.servers", "192.168.XX.XX:9092")
      .option("subscribe", "sales")
      .option("failOnDataLoss", value = false)
      .load()
      .createTempView("initData")

  // 对于DTS同步到Kafka的数据，需要avro解码，EMR提供了dts_binlog_parser的UDF来处理此问题。
  val dataBatch = spark.sql(
      """
        |SELECT dts_binlog_parser(value)
        |AS (recordID, source, dbTable, recordType, recordTimestamp, extraTags, fields, beforeImages, afterImages)
        |FROM initData
      """.stripMargin)

  // 选择INIT类型的数据作为初始数据。
  dataBatch.select(from_json(col("afterImages").cast("string"), schema).as("jsonData"))
      .where("recordType = 'INIT'")
      .select(
        col("jsonData.id").cast("long").as("id"),
        col("jsonData.date").as("date"),
        col("jsonData.name").as("name"),
        col("jsonData.sales").cast("decimal(7,2)")).as("sales")
      .write.format("delta").mode("append").save("/delta/sales")
}

try {
  DeltaTable.forPath("/delta/sales")
```

```
} catch {
  case e: AnalysisException if e.getMessage().contains("is not a Delta table") =>
    initDeltaTable()
}

spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "192.168.XX.XX:9092")
    .option("subscribe", "sales")
    .option("startingOffsets", "earliest")
    .option("maxOffsetsPerTrigger", 1000)
    .option("failOnDataLoss", value = false)
    .load()
    .createTempView("incremental")

// 对于DTS同步到Kafka的数据，需要avro解码，EMR提供了dts_binlog_parser的UDF来处理此问题。
val dataStream = spark.sql(
  """
    |SELECT dts_binlog_parser(value)
    |AS (recordID, source, dbTable, recordType, recordTimestamp, extraTags, fields, beforeImages, afterImages)
    |FROM incremental
  """.stripMargin)

val task = dataStream.writeStream
    .option("checkpointLocation", "/delta/sales_checkpoint")
    .foreachBatch(
      (ops, id) => {
        // 该window function用于提取针对某一记录的最新一条修改。
        val windowSpec = Window
            .partitionBy(coalesce(col("before_id"), col("id")))
            .orderBy(col("recordId").desc)
        // 从binlog中解析出recordType, beforeImages.id, afterImages.id, afterImages.date, afterImages.name, afterImages.sales。
        val mergeDf = ops
            .select(
              col("recordId"),
              col("recordType"),
              from_json(col("beforeImages").cast("string"), schema).as("before"),
              from_json(col("afterImages").cast("string"), schema).as("after"))
            .where("recordType != 'INIT'")
            .select(
              col("recordId"),
              col("recordType"),
              when(col("recordType") === "INSERT", col("after.id")).otherwise(col("before.id")).cast("long").as("before_id"),
              when(col("recordType") === "DELETE", col("before.id")).otherwise(col("after.id")).cast("long").as("id"),
              when(col("recordType") === "DELETE", col("before.date")).otherwise(col("after.date")).as("date"),
              when(col("recordType") === "DELETE", col("before.name")).otherwise(col("after.name")).as("name"),
              when(col("recordType") === "DELETE", col("before.sales")).otherwise(col("after.sales")).cast("decimal(7,2)").as("sales")
            )
            .select(
              dense_rank().over(windowSpec).as("rk"),
              col("recordType"),
              col("before_id"),
              col("id"),
              col("date"),
              col("name"),
              col("sales")
```

```
        )
        .where("rk = 1")

      //merge条件，用于将incremental数据和delta表数据做合并。
      val mergeCond = "target.id = source.before_id"

      DeltaTable.forPath(spark, "/delta/sales").as("target")
        .merge(mergeDf.as("source"), mergeCond)
        .whenMatched("source.recordType='UPDATE'")
        .updateExpr(Map(
          "id" -> "source.id",
          "date" -> "source.date",
          "name" -> "source.name",
          "sales" -> "source.sales"))
        .whenMatched("source.recordType='DELETE'")
        .delete()
        .whenNotMatched("source.recordType='INSERT' OR source.recordType='
UPDATE'")
        .insertExpr(Map(
          "id" -> "source.id",
          "date" -> "source.date",
          "name" -> "source.name",
          "sales" -> "source.sales"))
        .execute()
    }
  ).start()

task.awaitTermination()
```

- SQL
  - bash

    ```
    streaming-sql --master yarn --use-emr-datasource
    ```

  - SQL

    ```
    CREATE TABLE kafka_sales
    USING kafka
    OPTIONS(
    kafka.bootstrap.servers='192.168.XX.XX:9092',
    subscribe='sales'
    );

    CREATE TABLE delta_sales(id long, date string,  name string, sales decimal(7, 2))
    USING delta
    LOCATION '/delta/sales';

    INSERT INTO delta_sales
    SELECT CAST(jsonData.id AS LONG), jsonData.date, jsonData.name, jsonData.
    sales
    FROM (
      SELECT from_json(CAST(afterImages as STRING), 'id STRING, date DATE, name
    STRING, sales STRING') as jsonData
      FROM (
        SELECT dts_binlog_parser(value) AS (recordID, source, dbTable, recordType,
    recordTimestamp, extraTags, fields, beforeImages, afterImages)
        FROM kafka_sales
      ) binlog WHERE recordType='INIT'
    ) binlog_wo_init;

    CREATE SCAN incremental on kafka_sales USING STREAM
    ```

```
OPTIONS(
startingOffsets='earliest',
maxOffsetsPerTrigger='1000',
failOnDataLoss=false
);


CREATE STREAM job
OPTIONS(
checkpointLocation='/delta/sales_checkpoint'
)
MERGE INTO delta_sales as target
USING (
  SELECT
  recordId,
  recordType,
  before_id,
  id,
  date,
  name,
  sales
  FROM(
    SELECT
    recordId,
    recordType,
    CASE WHEN recordType = "INSERT" then after.id else before.id end as before_id
,
    CASE WHEN recordType = "DELETE" then CAST(before.id as LONG) else CAST(
after.id as LONG) end as id,
    CASE WHEN recordType = "DELETE" then before.date else after.date end as
date,
    CASE WHEN recordType = "DELETE" then before.name else after.name end as
name,
    CASE WHEN recordType = "DELETE" then CAST(before.sales as DECIMAL(7, 2))
else CAST(after.sales as DECIMAL(7, 2)) end as sales,
    dense_rank() OVER (PARTITION BY coalesce(before.id,after.id) ORDER BY
recordId DESC) as rank
    FROM (
      SELECT
      recordId,
      recordType,
      from_json(CAST(beforeImages as STRING), 'id STRING, date STRING, name
STRING, sales STRING') as before,
      from_json(CAST(afterImages as STRING), 'id STRING, date STRING, name
STRING, sales STRING') as after
      FROM (
        select dts_binlog_parser(value) as (recordID, source, dbTable, recordType
, recordTimestamp, extraTags, fields, beforeImages, afterImages) from
incremental
      ) binlog WHERE recordType != 'INIT'
    ) binlog_wo_init
  ) binlog_extract WHERE rank=1
) as source
ON target.id = source.before_id
WHEN MATCHED AND source.recordType='UPDATE' THEN
UPDATE SET id=source.id, date=source.date, name=source.name, sales=source.
sales
WHEN MATCHED AND source.recordType='DELETE' THEN
DELETE
WHEN NOT MATCHED AND (source.recordType='INSERT' OR source.recordType='
UPDATE') THEN
```

```
INSERT (id, date, name, sales) values (source.id, source.date, source.name,
source.sales);
```

6. 待上一步骤中的Spark Streaming作业启动后，我们尝试读一下这个Delta Table。

   scala

```
spark.read.format("delta").load("/delta/sales").show

+---+----------+------+------+
| id|      date|  name| sales|
+---+----------+------+------+
|  1|2019-11-11|Robert|323.00|
|  2|2019-11-11|   Lee|500.00|
|  3|2019-11-12|Robert|136.00|
|  4|2019-11-13|   Lee|211.00|
+---+----------+------+------+
```

   在RDS控制台执行下列四条命令并确认结果，注意我们对于id = 2的记录update了两次，理论上最终结果应当为最新一次修改。

```
DELETE FROM sales WHERE id = 1;
UPDATE sales SET sales = 150 WHERE id = 2;
UPDATE sales SET sales = 175 WHERE id = 2;
INSERT INTO sales VALUES (5, '2019-11-14', 'Robert', 233);

SELECT * FROM sales;
```

	id	date	name	sales
1	2	2019-11-11	Lee	150
2	3	2019-11-12	Robert	136
3	4	2019-11-13	Lee	211
4	5	2019-11-14	Robert	233

   重新读一下Delta表，发现数据已经更新了，且id=2的结果为最后一次的修改：

```
spark.read.format("delta").load("/delta/sales").show

+---+----------+------+------+
| id|      date|  name| sales|
+---+----------+------+------+
|  5|2019-11-14|Robert|233.00|
|  3|2019-11-12|Robert|136.00|
|  4|2019-11-13|   Lee|211.00|
|  2|2019-11-11|   Lee|175.00|
+---+----------+------+------+
```

最佳实践

随着数据实时流入，Delta内的小文件会迅速增多。针对这种情况，有两种解决方案：

- 对表进行分区。一方面，写入多数情况下是针对最近的分区，历史分区修改往往频次不是很高，这个时候对历史分区进行compaction操作，compaction因事务冲突失败的可能性较低。另一方面，带有分区谓词的查询效率较不分区的情况会高很多。
- 在流式写入的过程中，定期进行compation操作。例如，每过10个mini batch进行一次compaction。这种方式不存在compaction由于事务冲突失败的问题，但是由于compaction可能会影响到后续mini batch的时效性，因此采用这种方式要注意控制compaction的频次。

对于时效性要求不是那么高的场景，又担心compation因事务冲突失败，可以采用如下所示处理。在这种方式中，binlog的数据被定期收集到oss上（可以通过dts到kafka然后借助kafka-connect-oss将binlog定期收集到oss，也可以采用其他工具），然后启动spark批作业读取oss上的binlog，一次性的将binlog合并到Delta Lake。其流程图如下所示。

说明：

虚线部分可替换为其他可能方案。

附录：Kafka内binlog格式窥探

DTS同步到Kafka的binlog是avro编码的。如果要探查其文本形式，我们需要借助EMR提供的一个avro解析的UDF：dts_binlog_parser。

- Scala
  - bash

    ```
    spark-shell --master local --use-emr-datasource
    ```

  - scala

    在启动的spark-shell中执行以下命令。

    ```
    spark.read
       .format("kafka")
       .option("kafka.bootstrap.servers", "192.168.XX.XX:9092")
       .option("subscribe", "sales")
       .option("maxOffsetsPerTrigger", 1000)
       .load()
       .createTempView("kafkaData")

    val kafkaDF = spark.sql("SELECT dts_binlog_parser(value) FROM kafkaData")
    ```

```
kafkaDF.show(false)
```

- SQL
  - bash

```
streaming-sql --master local --use-emr-datasource
```

  - SQL

```
CREATE TABLE kafkaData
USING kafka
OPTIONS(
kafka.bootstrap.servers='192.168.XX.XX:9092',
subscribe='sales'
);

SELECT dts_binlog_parser(value) FROM kafkaData;
```

最终显示结果如下所示。

```
+--------+----------------------------------------------+---------------+----------
+------------------+---------+--------------------------+------------
+-------------------------------------------------------------+
|recordid|source                                |dbtable       |recordtype|recordtimestamp    |
extratags|fields                  |beforeimages|afterimages                                  |
+--------+----------------------------------------------+---------------+----------
+------------------+---------+--------------------------+------------
+-------------------------------------------------------------+
|1       |{"sourceType": "MySQL", "version": "0.0.0.0"}|delta_cdc.sales|INIT      |1970-01-01 08:
00:00|{}       |["id","date","name","sales"]|{}         |{"sales":"323.0","date":"2019-11-11","name
":"Robert","id":"1"}|
|2       |{"sourceType": "MySQL", "version": "0.0.0.0"}|delta_cdc.sales|INIT      |1970-01-01 08:
00:00|{}       |["id","date","name","sales"]|{}         |{"sales":"500.0","date":"2019-11-11","name
":"Lee","id":"2"}   |
|3       |{"sourceType": "MySQL", "version": "0.0.0.0"}|delta_cdc.sales|INIT      |1970-01-01 08:
00:00|{}       |["id","date","name","sales"]|{}         |{"sales":"136.0","date":"2019-11-12","name
":"Robert","id":"3"}|
|4       |{"sourceType": "MySQL", "version": "0.0.0.0"}|delta_cdc.sales|INIT      |1970-01-01 08:
00:00|{}       |["id","date","name","sales"]|{}         |{"sales":"211.0","date":"2019-11-13","name
":"Lee","id":"4"}   |
+--------+----------------------------------------------+---------------+----------
+------------------+---------+--------------------------+------------
+-------------------------------------------------------------+
```

11.3.3 场景三：冷热分层

本文首先介绍了冷数据的特点和适应场景，接着对海量结构化数据的冷热分层进行了方案和架构的设计，最后通过Tablestore结合Delta Lake对冷热分层进行了生动的实战示例演示。通过冷热分层可以让计算和存储的资源得到充分利用，进而让业务能够用更低的成本承载更优质的服务。

背景信息

在海量大数据场景下，随着业务和数据量的不断增长，性能和成本的权衡变成了大数据系统设计面临的关键挑战。所以在架构设计之初，我们就需要把整套架构的成本考虑进来，这对应的就是数据的分层存储和存储计算引擎的选择。

Delta Lake是DataBricks公司推出的一种新型数据湖方案，围绕数据流入、数据组织管理、数据查询和数据流出，推出了一系列功能特性，同时提供了数据操作的ACID和CRUD。通过结合Delta Lake和上下游组件，可以搭建出一个便捷、易用、安全的数据湖架构。在数据湖架构设计中，通常我们会应用HTAP（Hybrid Transaction and Analytical Process）体系结构，通过合理的选择分层存储组件和计算引擎，既能支持海量数据分析和快速的事务更新写入，又能有效的进行冷热数据的分离进而降低成本。

更多介绍可参见结构化大数据分析平台设计、面向海量数据的极致成本优化-云HBase的一体化冷热分离和云上如何做冷热数据分离。

冷数据介绍

数据按照实际访问的频率可以区分为热数据、温数据和冷数据，其中冷数据的数据量较大，很少被访问，甚至整个生命周期都可能不会被访问，只是为了满足业务合规或者特定场景需求在一定时间内保存。通常我们有两个方式来区分冷热数据：

- 按照数据的创建时间：常见于交易类数据、时序监控、IM聊天等场景，通常数据写入初期用户的关注度较高且访问频繁，但随着时间的推移，旧的数据访问频率会越来越低，仅存在少量查询甚至完全不查询。
- 按照访问热度：有些数据的访问频率并非按时间，例如某些大V的旧博客由于某些原因突然大量被访问到，这样的冷数据也会变成热数据。这个时候就不应该再按时间区分，需要根据具体的业务和数据分布规律来区分冷热。

本文主要讨论按数据的创建时间的冷热数据分层，而对于按访问热度的数据分层，通常可以采用业务打标或系统自识别等手段。

冷数据特点

从冷热数据的区分，可以看出，冷数据具备以下一些特点：

- 数据量大：不同于热数据，冷数据通常需要保存较长时间甚至是所有时间的数据。
- 成本敏感：数据量大且访问频率较低，不宜投入过多的成本。
- 性能要求不高：这里是相对的概念，相比于一般的TP请求不需要查询在毫秒级别返回，冷数据的查询可以容忍到数十秒甚至更长时间才出结果，或者可以进行异步处理。
- 业务场景较简单：对于冷数据基本都是批量的写入和删除，一般没有更新操作。在查询时，一般只需要读取指定条件的数据，且查询条件不会过于复杂。

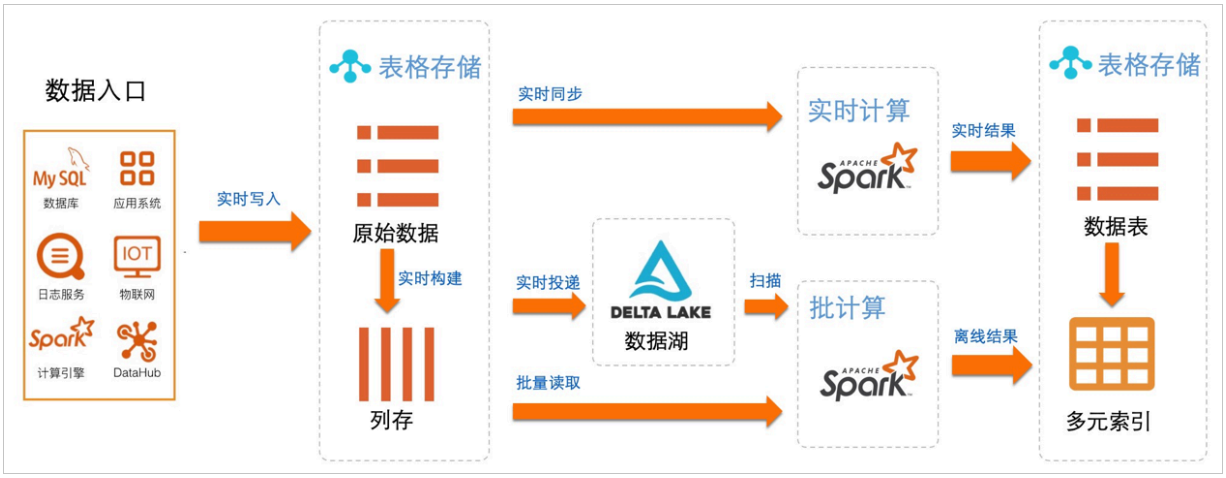
冷热分层适用场景

针对冷数据的特点，挖掘出一些冷热分层适用的场景：

- 时序类数据场景：时序类数据天然具备时间属性，数据量大，且几乎只做追加操作。时序数据无处不在，常见于监控类数据、交易类数据、物联网数据、环境监测等场景。
  - IM场景：如钉钉，用户一般会查阅最近的若干条聊天记录，历史的数据一般只有在有特殊需求的时候才会去查询。
  - 监控场景：如云监控，用户通常只会查看近期的监控，历史数据一般只有在调查问题或者制定报表时才会去查询。
  - 账单场景：如支付宝，我们通常只会查询最近几天或者一个月内的账单，超过一年以上的账单基本不会去查询。
  - 物联网场景：如IOT，通常设备近期上报的数据是热点数据，会经常被分析，而历史数据的分析频率都较低。
- 归档类场景：对于重写轻读的数据，可以将数据定期归档到成本更低的存储组件或更高压缩比的存储介质中，达到降低成本的目的。

海量结构化数据Delta Lake架构

针对结构化冷热分层数据场景，表格存储（Tablestore）联合EMR团队推出了一个海量结构化数据的Delta Lake架构。针对冷热数据方案设计的几个问题，都进行了很好的解决。基于表格存储的通道服务，可以将原始数据利用CDC技术派生到多种存储组件中，例如，将原始数据派生到Delta Lake和Tablestore引擎自身的列存中，进而完成冷热数据的分离和异构。同时表格存储提供灵活的上游数据入口和TTL功能，您可以定制热数据的生命周期，将冷数据不断的实时投递到Delta Lake和列存中，达到降成本的目的。最后对于计算和查询层，Tablestore结合Spark可以完成对冷热数据的全增量一体的定制化计算，并可以最终将计算结果存入Tablestore的索引引擎中进行统一的查询。

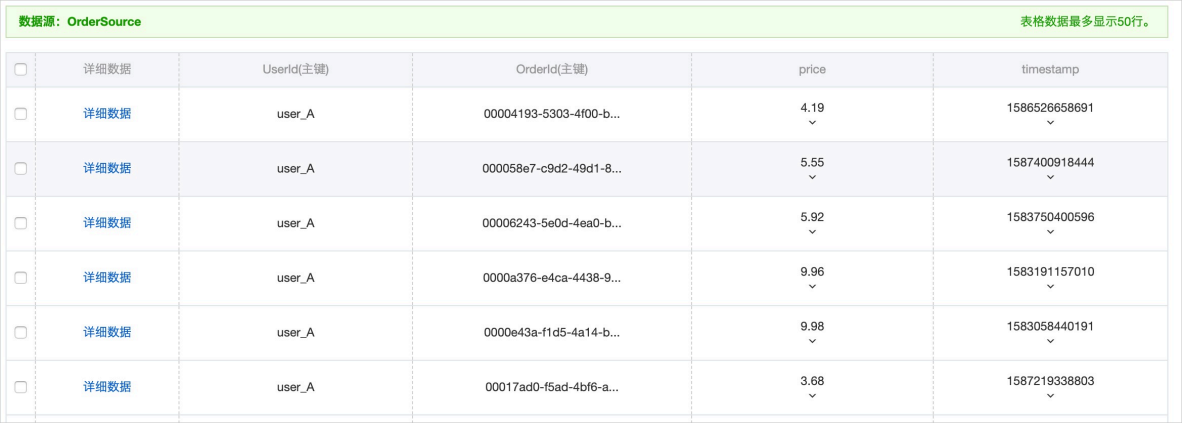


冷热分层示例

本示例结合Tablestore和Delta Lake进行数据湖冷热分层。

1. 数据源说明。

   数据源是一张简单的原始订单表OrderSource，表有两个主键UserId（用户ID）和OrderId（订单ID），两个属性列price（价格）和timestamp（订单时间）。使用Tablestore SDK的BatchWrite接口进行订单数据的写入，订单的时间戳的时间范围为最近的90天（本文的模拟时间范围为2020-02-26~2020-05-26），共计写入3112400条。

   数据源：OrderSource　　表格数据最多显示50行。

详细数据	UserId(主键)	OrderId(主键)	price	timestamp
详细数据	user_A	00004193-5303-4f00-b...	4.19	1586526658691
详细数据	user_A	000058e7-c9d2-49d1-8...	5.55	1587400918444
详细数据	user_A	00006243-5e0d-4ea0-b...	5.92	1583750400596
详细数据	user_A	0000a376-e4ca-4438-9...	9.96	1583191157010
详细数据	user_A	0000e43a-f1d5-4a14-b...	9.98	1583058440191
详细数据	user_A	00017ad0-f5ad-4bf6-a...	3.68	1587219338803

   在模拟订单写入时，对应Tablestore表中的属性列的版本号也会被设置为相应的时间戳，这样通过配置表上的TTL属性，当写入数据的保留时长超过设置的TTL后，系统会自动清理对应版本号的数据。

2. 实时流式投递。

a) 创建数据源表，同时在表格存储控制台上创建增量通道，利用通道提供的CDC（日志变更捕获）技术将新增的主表数据不断同步到Delta中，创建得到的通道ID将会用于后续的SQL配置。

b) 在EMR集群的Header机器上启动streaming-sql交互式命令行。

```
streaming-sql --master yarn --use-emr-datasource --num-executors 16 --executor
-memory 4g --executor-cores 4
```

```
// 源表和目的表
// 1. 创建源表
DROP TABLE IF EXISTS order_source;
CREATE TABLE order_source
USING tablestore
OPTIONS(
endpoint="http://vehicle-test.cn-hangzhou.vpc.tablestore.aliyuncs.com",
access.key.id="",
access.key.secret="",
instance.name="vehicle-test",
table.name="OrderSource",
catalog='{"columns": {"UserId": {"col": "UserId", "type": "string"}, "OrderId": {"col": "
OrderId", "type": "string"},"price": {"col": "price", "type": "double"}, "timestamp": {"
col": "timestamp", "type": "long"}}}',
);


// 2. 创建Delta Lake Sink: delta_orders
DROP TABLE IF EXISTS delta_orders;
CREATE TABLE delta_orders(
UserId string,
OrderId string,
price double,
timestamp long
)
USING delta
LOCATION '/delta/orders';

// 3. 在源表上创建增量SCAN视图
CREATE SCAN incremental_orders ON order_source USING STREAM
OPTIONS(
```

```
tunnel.id="324c6bee-b10d-4265-9858-b829a1b71b4b",
maxoffsetsperchannel="10000");

// 4. 启动Stream作业，将Tablestore CDC数据实时同步到Delta Lake中。
CREATE STREAM orders_job
OPTIONS (
checkpointLocation='/delta/orders_checkpoint',
triggerIntervalMs='3000'
)
MERGE INTO delta_orders
USING incremental_orders AS delta_source
ON delta_orders.UserId=delta_source.UserId AND delta_orders.OrderId=delta_sour
ce.OrderId
WHEN MATCHED AND delta_source.__ots_record_type__='DELETE' THEN
DELETE
WHEN MATCHED AND delta_source.__ots_record_type__='UPDATE' THEN
UPDATE SET UserId=delta_source.UserId, OrderId=delta_source.OrderId, price=
delta_source.price, timestamp=delta_source.timestamp
WHEN NOT MATCHED AND delta_source.__ots_record_type__='PUT' THEN
INSERT (UserId, OrderId, price, timestamp) values (delta_source.UserId, delta_sour
ce.OrderId, delta_source.price, delta_source.timestamp);
```

- Tablestore源表创建：创建order_source源表，其中OPTIONS参数中catalog为表字段的Schema定义（本例中对应UserId、OrderId、price和timestamp四列）。
- Delta Lake Sink表创建：创建写完Delta的delta_orders目的表，LOCATION中指定的是Delta文件存储的位置。
- Tablestore源表上创建增量SCAN视图：本例创建incremental_orders的流式视图，其中OPTIONS参数中tunnel.id为第1步创建的增量通道ID，maxoffsetsperchannel为通道每个分区（每个Spark微批）写的最大数据量。
- 启动Stream作业进行实时投递：本例中会根据Tablestore的主键列（UserId和OrderId）主键进行聚合，同时根据CDC日志的操作类型（PUT，UPDATE，DELETE）转化为对应的Delta操作。特别说明的是 __ots_record_type__ 是Tablestore流式Source提供的预定义列，表示的是行操作类型。

3. 查询冷热数据。

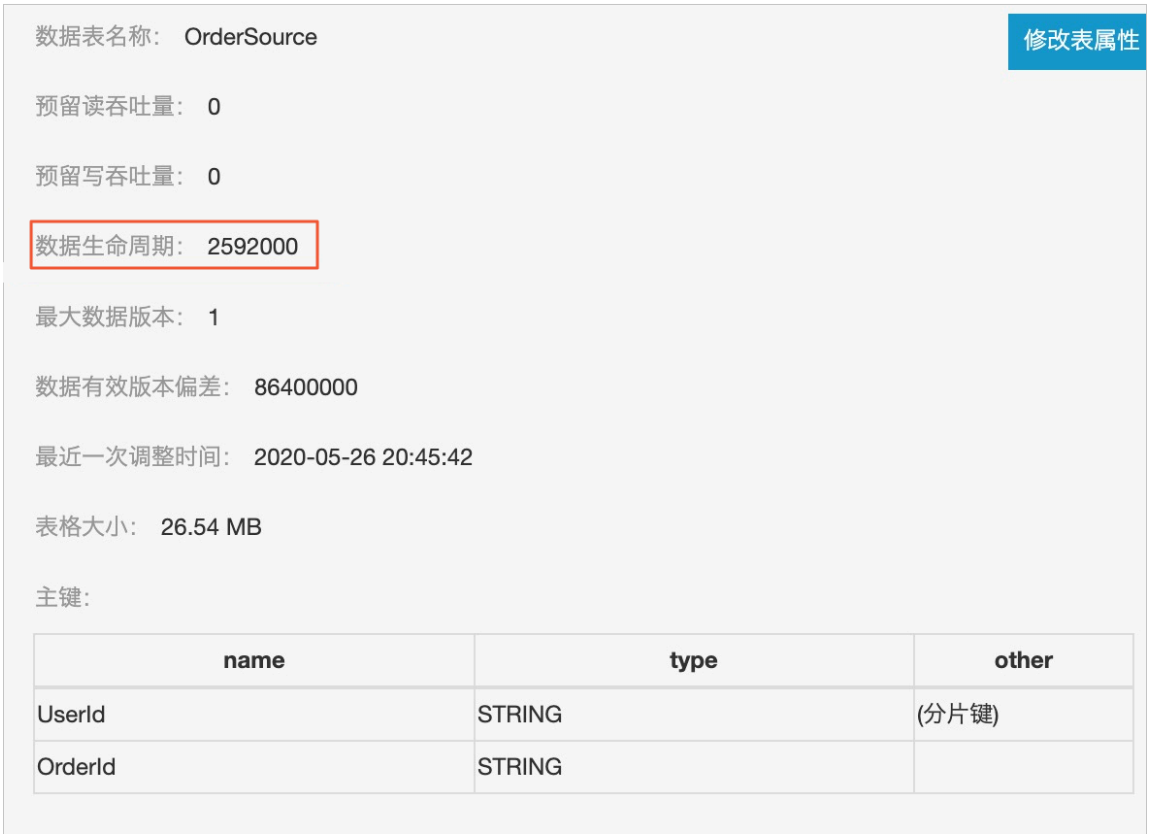
在实际的设计中，我们一般会把热数据保存在Tablestore表中进行高效的TP查询，冷数据或全量数据保存在Delta中。通过配置Tablestore表的生命周期（TTL），我们可以灵活的对热数据量进行控制。

a) 在配置主表的TTL之前，对源表（order_source）和目的表（delta_orders）进行一些查询，此时两边的查询结果保持一致。

```
spark-sql> SELECT COUNT(*) FROM order_source;
3112400
Time taken: 7.85 seconds, Fetched 1 row(s)
spark-sql> SELECT COUNT(*) FROM delta_orders;
3112400
Time taken: 5.004 seconds, Fetched 1 row(s)
spark-sql> SELECT COUNT(*) FROM order_source WHERE price > 5;
1554430
Time taken: 9.51 seconds, Fetched 1 row(s)
spark-sql> SELECT COUNT(*) FROM delta_orders WHERE price > 5;
1554430
Time taken: 6.45 seconds, Fetched 1 row(s)
spark-sql>
```

b) 配置Tablestore的TTL为最近30天。

这样Tablestore中的热数据只有最近30天的数据，而Delta中依旧保留的是全量数据，进而达到冷热分层的目的。



c) 展示一些分层之后的简单查询，具体的查询路由需要结合业务逻辑进行一些路由选择。分层之后热数据总条数为1017004条，冷数据（全量数据）保持不变依旧为3112400条。

```
spark-sql> SELECT COUNT(*) FROM delta_orders;
3112400
Time taken: 5.823 seconds, Fetched 1 row(s)
spark-sql> SELECT COUNT(*) FROM order_source;
1016912
Time taken: 4.083 seconds, Fetched 1 row(s)
spark-sql> SELECT COUNT(*) FROM order_source where timestamp < 1587868748000;
0
Time taken: 5.316 seconds, Fetched 1 row(s)
spark-sql> SELECT COUNT(*) FROM delta_orders where timestamp < 1587868748000;
2060732
Time taken: 8.846 seconds, Fetched 1 row(s)
spark-sql>
```

11.4 基本操作

11.4.1 批式读写

本文介绍Delta Lake如何批式读写数据。

建表并写入数据

- Scala

```
// 非分区表
data.write.format("delta").save("/tmp/delta_table")
// 分区表
data.write.format("delta").partitionedBy("date").save("/tmp/delta_table")
```

- SQL

```
-- 非分区表
CREATE TABLE delta_table (id INT) USING delta LOCATION "/tmp/delta_table";
INSERT INTO delta_table VALUES 0,1,2,3,4;
-- 分区表
CREATE TABLE delta_table (


id INT, date STRING) USING delta PARTITIONED BY (date) LOCATION "/tmp/delta_table";
INSERT INTO delta_table PARTITION (date='2019-11-11') VALUES 0,1,2,3,4;
-- 或者使用动态分区写入
INSERT INTO delta_table PARTITION (date) VALUES (0,'2019-11-01'),(1,'2019-11-02'),(2,'2019-11-05'),(3,'2019-11-08'),(4,'2019-11-11');
```

追加数据

- Scala

```
// 非分区表
data.write.format("delta").mode("append").save("/tmp/delta_table")
// 分区表
```

```
data.write.format("delta").mode("append").save("/tmp/delta_table")
```

- SQL

```
-- 非分区表
INSERT INTO delta_table VALUES 0,1,2,3,4;
-- 分区表
INSERT INTO delta_table PARTITION (date='2019-11-11') VALUES 0,1,2,3,4;
-- 或者使用动态分区写入
INSERT INTO delta_table PARTITION (date) VALUES (0,'2019-11-01'),(1,'2019-11-02'),(2,'2019-11-05'),(3,'2019-11-08'),(4,'2019-11-11');
```

覆盖数据

- Scala

```
// 非分区表
data.write.format("delta").mode("overwrite").save("/tmp/delta_table")
// 分区表
data.write.format("delta").mode("overwrite").option("replaceWhere", "date >= '2019-11-01' AND date <= '2019-11-11'").save("/tmp/delta_table")
```

- SQL

```
INSERT OVERWRITE TABLE delta_table VALUES 0,1,2,3,4;
-- 分区表
INSERT OVERWRITE delta_table PARTITION (date='2019-11-11') VALUES 0,1,2,3,4;
-- 或者使用动态分区写入
INSERT OVERWRITE delta_table PARTITION (date) VALUES (0,'2019-11-01'),(1,'2019-11-02'),(2,'2019-11-05'),(3,'2019-11-08'),(4,'2019-11-11');
```

读表

- Scala

```
spark.read.format("delta").load("/tmp/delta_table")
```

- SQL

```
SELECT * FROM delta_table;
```

历史版本访问

- Scala

```
df1 = spark.read.format("delta").option("timestampAsOf", timestamp_string).load("/tmp/delta_table")
df2 = spark.read.format("delta").option("versionAsOf", version).load("/tmp/delta_table")
```

- SQL

  不支持。

11.4.2 流式读写

本文介绍Delta Lake作为数据源和数据接收端如何流式读写数据。

Delta Table作为数据源（Source）

```
spark.readStream
  .format("delta")
  .option("maxFilesPerTrigger", 1000)
  .load("/tmp/delta_table")
```

maxFilesPerTrigger指定了一个批次最多处理的文件数量，默认值为1000。

通常作为数据源的组件，数据一旦产生就会被下游消费，数据不会发生变化。但是Delta还兼顾了数据湖的角色，数据可能会被删除、更新，或者合并。目前Delta提供了两个选项来应对这种情况：

- **ignoreDeletes**：设置该选项为true后，对分区的删除动作不会有任何影响。
- **ignoreChanges**：设置该选项为true后，删除、更新或合并动作不会被特殊处理，但是这些动作产生的新文件会被当成新数据发送到下游。例如，某一个文件包含10000条数据，更新其中一条数据后，新文件有9999条旧数据和1条新数据。这9999条旧数据和1条新数据会被发送到下游。

Delta Table作为数据接收端（Sink）

- Append模式：该模式是Spark Streaming的默认工作模式。

  ```
  df.writeStream
    .format("delta")
    .outputMode("append")
    .option("checkpointLocation", "/tmp/delta_table/_checkpoints")
    .start("/tmp/delta_table")
  ```

- Complete模式：在该模式下每一次batch的执行都会以全表覆盖的形式写目标表。例如，对于（id LONG, date DATE, name STRING, sales DOUBLE）这张表，您可以统计每个人的总销售额，将统计结果写入目标表，每个批次更新一次。

  ```
  spark.readStream
    .format("delta")
    .load("/tmp/delta_table")
    .select("name","sales")
    .groupBy("name")
    .agg(sum("sales"))
    .writeStream
    .format("delta")
    .outputMode("complete")
    .option("checkpointLocation", "/tmp/delta_table_summary/_checkpoints")
  ```

```
  .start("/tmp/delta_table_summary")
```

11.4.3 管理数据

本文介绍Delta Lake如何删除、更新与合并数据。

删除数据

- Scala

```
import io.delta.tables.
val deltaTable = DeltaTable.forPath(spark, "/tmp/delta_table")
deltaTable.delete("date < '2019-11-11'")
import org.apache.spark.sql.functions.
import spark.implicits.
deltaTable.delete(col("date") < "2019-11-11")
```

- SQL

```
DELETE FROM delta_table [AS t] [WHERE t.date < '2019-11-11'];
```

相关介绍如下：

- 暂不支持带有子查询的WHERE条件。但如果子查询为标量子查询且使用SQL，可以设置spark.sql.uncorrelated.scalar.subquery.preexecution.enabled为true后进行查询，例如：

  ```
  DELETE FROM delta_table WHERE t.date < (SELECT date FROM ref_table WHERE ....)
  ```

- 如果您需要根据另一张表对目标表的匹配行进行删除，请使用Merge语法。

  例如：DELETE FROM target WHERE target.col = ref.col ...。

说明：

使用DELETE命令，如果没有条件限制，则会删除所有数据。

更新数据

- Scala

```
import io.delta.tables._

val deltaTable = DeltaTable.forPath(spark, "/tmp/delta_table")

deltaTable.updateExpr(          //使用SQL字符串。
  "name = 'Robet'",
  Map("name" -> "'Robert'")

import org.apache.spark.sql.functions._
import spark.implicits._

deltaTable.update(              //使用SQL函数和隐式转换。
  col("name") === "Robet"),
```

```
  Map("name" -> lit("Robert"));
```

- SQL

```
UPDATE delta_table [AS t] SET t.id = t.id + 1 [WHERE t.date < '2019-11-11'];
```

相关介绍如下：

- 暂不支持带有子查询的WHERE条件。但如果子查询为标量子查询且使用SQL，可以设置spark.sql.uncorrelated.scalar.subquery.preexecution.enabled为true后进行查询，例如：

  ```
  UPDATE delta_table SET t.id = t.id + 1 WHERE t.date < (SELECT date FROM ref_table WHERE ....)
  ```

- 如果要根据另一张表对目标表的匹配行进行删除，请使用Merge语法。

  例如，UPDATE target SET target.col = ref.col ... 或 WHERE target.col = ref.col ...。

合并数据

- Scala

```
import io.delta.tables._
import org.apache.spark.sql.functions._

val updatesDF = ...  // define the updates DataFrame[date, id, name]

DeltaTable.forPath(spark, "/tmp/delta_table")
  .as("target")
  .merge(updatesDF.as("source"), "target.id = source.id")
  .whenMatched("target.name = 'should_update'")
  .updateExpr(Map("target.name" -> "source.name"))
  .whenMatched("target.name = 'should_delete'")
  .delete()
  .whenNotMatched("source.name = 'shoulde_insert'")
  .insertExpr(
    Map(
      "date" -> "updates.date",
      "eventId" -> "updates.eventId",
      "data" -> "updates.data"))
  .execute()
```

- SQL

```
MERGE INTO target AS t
USING source AS s
ON t.date = s.date
WHEN MATCHED [AND t.name = 'should_update'] THEN UPDATE SET target.name = source.name
WHEN MATCHED [AND t.name = 'should_delete'] THEN DELETE
```

```
WHEN NOT MATCHED [AND s.name = 'should_insert'] THEN INSERT (t.date, t.name, t.id)
VALUES (s.date, s.name.s.id)
```

相关介绍如下：

- UPDATE子句和INSERT子句支持*语法：如果设置为UPDATE SETE *或者INSERT *，则会更新或插入所有字段。
- 暂不支持带有子查询的ON条件，但如果子查询为标量子查询的形式且使用SQL，可以设置spark.sql.uncorrelated.scalar.subquery.preexecution.enabled为true后进行查询。

11.4.4 优化表

本文介绍如何通过Optimize和Vacuum命令优化表。

Optimize

- Scala

```
import io.delta.tables._

val deltaTable = DeltaTable.forPath(spark, "/tmp/delta_table")

deltaTable.optimizeExpr("date < '2019-11-11'", Seq("date", "id"))                    // 使用
SQL字符串。

import org.apache.spark.sql.functions._
import spark.implicits._

deltaTable.optimize(col("date < '2019-11-11'"), Seq(col("date"), col("id")))          // 使用
SQL函数和隐式转换。
```

- SQL

```
OPTIMIZE delta_table [WHERE t.date < '2019-11-11'] [ZORDER BY t.date,t.id];
```

当deltaTable.optimize()不指定参数时，表示对全表进行优化。

说明：

从EMR-3.27.0版本开始支持ZOrder功能。

Vacuum

通过Optimize后，小文件会被合并为大文件，并且小文件被标记为墓碑文件。因为Delta有访问历史的功能，所以原有小文件不会被删除，当访问历史版本时，仍然需要读取小文件，影响优化表的执行效率。

Vacuum用于清理历史文件。如果墓碑文件涉及的数据过时很久，可以用Vacuum命令将其物理删除。默认情况下，墓碑文件只有经过一个安全期才能被删除，如果删除一个尚在安全期内的文件，将会抛出异常。

如果您要删除近期的墓碑文件，例如1天前的墓碑文件，可通过以下两种方式：

- 设置合理的安全期参数（推荐）：spark.databricks.delta.properties.defaults.deletedFileRetentionDuration、interval 1 day、2 weeks、365 days等。不支持设置Month和Year。这些参数为表的静态属性，无法通过命令行或者设置--conf来实现。其设置方法为：

  1. 在spark-default.conf中配置全局属性。
  2. 通过ALTER TABLE <tbl> SET PROPERTIES (key=value)更改此表属性。

- 通过关闭安全期检查（不推荐）：设置set spark.databricks.delta.retentionDurationCheck.enabled = flase关闭后，即可删除近期合并过的小文件。

注意：

- 不建议关闭安全期检查。
- 不建议安全期设置过小。

  过小的安全期可能导致正在读取某一历史版本的作业失败。甚至导致当前执行作业产生的临时文件被清理掉，从而导致作业失败。

- Scala

```
import io.delta.tables._
val deltaTable = DeltaTable.forPath(spark, pathToTable)

deltaTable.vacuum()       // 物理删除超出安全期的墓碑文件。
deltaTable.vacuum(100)    // 物理删除100小时前的墓碑文件。
```

- SQL

```
VACUUM table_name [RETAIN num HOURS] [DRY RUN]
```

相关介绍如下：

- 通过RETAIN子句指定删除超出设置时间间隔的墓碑文件。RETAIN num HOURS的默认值为spark.databricks.delta.properties.defaults.deletedFileRetentionDuration指定的值。
- 如果指定了DRY RUN，被删除文件不会被实际删除。

11.4.5 转换表

本文介绍如何将Hive或SparkSQL的Parquet表转为Delta表。

使用示例如下：

- Scala

```
import io.delta.tables._
// 非分区表
val deltaTable = DeltaTable.convertToDelta(spark, "parquet.`/path/to/table`")
```

```
// 分区表
val partitionedDeltaTable = DeltaTable.convertToDelta(spark, "parquet.`/path/to/table`", "date string")
```

- SQL

```
-- 非分区表
CONVERT TO DELTA parquet_table;
-- 分区表
CONVERT TO DELTA parquet_table PARTITIONED BY (date STRING);
```

11.4.6 修改表

EMR-3.27.0及以后版本增加了对ALTER TABLE的部分支持。

增加列

- Scala

  不支持。

- SQL

```
ALTER TABLE table_name ADD COLUMNS (col_name data_type [COMMENT col_comment], ...)
```

修改列

- Scala

  不支持。

- SQL

```
ALTER TABLE table_name CHANGE [COLUMN] col_name col_name data_type [COMMENT col_comment]
```

> **说明:**
> 目前仅支持修改列的comments。

设置表属性

- Scala

  不支持。

- SQL

```
ALTER [TABLE] table_name SET TBLPROPERTIES (key1=val1, key2=val2, ...)
ALTER [TABLE] table_name UNSET TBLPROPERTIES [IF EXISTS] (key1, key2, ...)
```

> **说明:**

UNSET语句中，若没有指定IF EXISTS，key不存在时会抛出异常。

11.4.7 数据质量与Schema演化

本文介绍Schema的校验、合并以及重建。

Schema校验

Schema校验默认开启。当所写的数据没有包含表定义的字段时，该字段会设置为null；当所写数据的字段在表中没有定义时，抛出异常，并提示Schema不匹配。

Delta定义了三种Schema校验规则：

- 数据字段中包含表中未定义的字段。
- 数据字段类型与表中该字段的类型定义不同。
- 数据中包含不区分大小写的同名字段，例如：Foo和foo。

```
import scala.collection.JavaConverters._
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._

val schema=new StructTupe().add("f1", LongType)
val df = spark.createDataFrame(List(Row(1L)).asJava, schema)
df.write.format("delta").mode("append").save("/tmp/delta_table")

org.apache.spark.sql.AnalysisException: A schema mismatch detected when writing to the Delta table.
To enable schema migration, please set:
'.option("mergeSchema", "true")'.

Table schema:
root
-- id: long (nullable = true)
-- date: date (nullable = true)
-- name: string (nullable = true)
-- sales: string (nullable = true)


Data schema:
root
-- f1: long (nullable = true)
```

Schema合并

如果需要将数据写入目标表，同时更新Schema，则开启mergeSchema功能。

开启mergeSchema后，如果数据和表的Schema不一致，且满足自动合并Schema的条件，将被自动合并。

- Scala

  ```
  df.write.option("mergeSchema", "true")
  ```

- SQL

  不支持。

自动合并Schema的条件如下：

- 增加列。
- 数据类型的兼容式转换：
  - NullType -> 其他任何类型。
  - ByteType -> ShortType -> IntegerType。

  > **说明：**
  > 当新写入数据的Schema变更了分区，请重建Schema。

Schema重建

当Schema变动不属于Schema合并的范围，则需要重建Schema。例如：删除某个列或者将列从一个类型转换为另一个不兼容的类型。

- Scala

  ```
  df.write.option("overwriteSchema", "true")
  ```

  重建Schema也需要对数据进行重写，以防数据和元数据Schema不一致。

  如果需要重建Schema，请使用df.write.mode("overwrite").option("overwriteSchema", "true")对原有数据和Schema进行重写。其中.option("overwriteSchema", "true")是必配项。

- SQL

  不支持。

11.4.8 访问历史信息

本文介绍如何以Scala或SQL方式访问Delta表的历史信息。

- Scala

  ```
  import io.delta.tables._
  val deltaTable = DeltaTable.forPath(spark, pathToTable)
  val fullHistoryDF = deltaTable.history()    // 获取table的全部历史。
  ```

```
val lastOperationDF = deltaTable.history(1) // 获取最近一个版本历史。
```

- SQL

```
DESCRIBE HISTORY table [LIMIT 1]
```

11.5 使用Hive读Delta table

E-MapReduce支持Hive读取Delta table，提供DeltaInputFormat和SparkSQL两种读取方式，其中DeltaInputFormat为E-MapReduce独有的方式。本文介绍如何使用Hive读Delta table。

使用DeltaInputFormat读Delta table（仅限EMR）

1. 使用Hive客户端，在Hive Metastore中创建指向Delta目录的外表。

```
CREATE EXTERNAL TABLE delta_tbl(id bigint, `date` string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT 'io.delta.hive.DeltaInputFormat'
OUTPUTFORMAT 'io.delta.hive.DeltaOutputFormat'
LOCATION '/tmp/delta_table';
```

说明：

- 如果Delta表是分区表，请执行partitioned by命令在Hive中创建对应的外表。
- 如果Delta表中存在新增分区，请执行msck repair命令同步分区信息到Hive外表。

2. 启动Hive客户端读取数据。

```
SET hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;---EMR-3.25.0（不含）之前版本必须设置。
SELECT * FROM delta_tbl LIMIT 10;
```

在SparkSQL中查询Hive建立的Delta表

使用DeltaInputFormat读Delta table（仅限EMR）创建的Delta表，由于表中缺少了Spark取表需要的必要信息，导致SparkSQL无法正常访问。如需正常访问，请在Hive建表语句中补全信息。

```
CREATE EXTERNAL TABLE delta_tbl(id bigint, `date` string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
WITH SERDEPROPERTIES("path" = "/tmp/delta_table")
STORED AS INPUTFORMAT 'io.delta.hive.DeltaInputFormat'
OUTPUTFORMAT 'io.delta.hive.DeltaOutputFormat'
LOCATION '/tmp/delta_table'
TBLPROPERTIES("spark.sql.sources.provider" = "delta");
```

说明：

目前Hive尚不兼容SparkSQL Using语法建立的Delta表。

11.6 使用Presto读Delta table

E-MapReduce支持Presto读取Delta table，提供DeltaInputFormat和SymlinkTextInputFormat两种读取方式，其中DeltaInputFormat为E-MapReduce独有的方式。本文介绍如何使用Presto读Delta table。

使用DeltaInputFormat读Delta table（仅限EMR）

1. 使用Hive客户端，在Hive Metastore中创建一张指向Delta目录的外表。

```
CREATE EXTERNAL TABLE delta_tbl(id bigint, `date` string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT 'io.delta.hive.DeltaInputFormat'
OUTPUTFORMAT 'io.delta.hive.DeltaOutputFormat'
LOCATION '/tmp/delta_table';
```

说明：

- 如果Delta表是分区表，请执行partitioned by命令在Hive中创建对应的外表。
- 如果Delta表中存在新增分区，请执行msck repair命令同步分区信息到Hive外表。

2. 启动Presto客户端读取Delta table。

```
SELECT * FROM delta_tbl LIMIT 10;
```

使用SymlinkTextInputFormat读取Delta table

1. 使用SparkSQL为目标Delta表创建Symlink文件。

```
GENERATE symlink_format_manifest FOR TABLE delta.`/delta_test/order`
```

说明：

Delta表每次更新后，请执行 GENERATE，确保Presto读取的是Delta表中最新数据。

2. 使用Hive客户端在Hive Metastore中创建一张指向Delta目录的外表。

```
CREATE EXTERNAL TABLE delta_tbl(id bigint, `date` string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.SymlinkTextInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION '/tmp/delta_table/_symlink_format_manifest/';
```

3. 启动Presto客户端读取Delta table。

```
SELECT * FROM delta_tbl LIMIT 10;
```

说明：

目前Presto尚不兼容SparkSQL Using语法建立的Delta表。

11.7 附录

11.7.1 附录1 重要参数介绍

本文介绍一些Delta中比较重要的参数。

Delta的设置参数分为三类：

- Spark SQL设置，即设置SQL运行时的参数。
- 运行时参数，即可以在Session中动态设置的参数，以 spark.databricks.delta.前缀开头。
- 非运行时参数，只能够在Spark的配置文件中配置为全局参数，或者建表时在TBLPROPERTIES中指定为表参数。表参数的优先级高于全局参数。当设置为全局参数时，配置前缀为spark.databricks.delta.properties.defaults.，当设置为 TBLPROPERTIES时，前缀为 delta.。

参数	描述
spark.databricks.delta.snapshotPartitions	默认值为10。 此参数为delta log元数据的partition数量。当delta log特别大时，需要增大此值，反之减小此值的设置。该值的大小对于delta table的解析性能影响较大。
spark.databricks.delta.retentionDurationCheck.enabled	默认值为true。 清理墓碑文件时是否进行安全期检查。 **警告：** 如果您想删除近期合并过的小文件，可以设置此参数为false，来关闭此安全检查，但不建议您关闭此检查，这样可能会删除近期的数据而造成数据读写失败。

参数	描述
spark.databricks.delta.schema.autoMerge.enabled	默认值为false。 Delta有校验写入数据是否符合表定义Schema的功能，用于保证写入数据的正确性。当您数据的Schema发生变更后，需要在写入数据时在option中显示指定mergeSchema为true。如果您期望当数据Schema发生变化自动进行Schema的合并，请设置该值为true。但是我们仍然建议您使用显示指定的方式，而不是让它自动合并Schema。
spark.databricks.delta.properties.defaults.deletedFileRetentionDuration或delta.deletedFileRetentionDuration	默认值为interval 1 week。 Delta墓碑文件的安全期。清空未超过安全期内的墓碑文件将会抛出异常（如果spark.databricks.delta.retentionDurationCheck.enabled为true的话）。 **说明：** 此值应当大于等于1个小时。
spark.databricks.delta.properties.defaults.logRetentionDuration或delta.logRetentionDuration	默认值为interval 30 days。 Delta log文件过期时间。Delta log过期被定义为： • 该log文件对应的数据文件已经做过了compaction。 • 该log文件超过了上述文件过期时间。每当Delta log进行checkpoint动作时，会检查是否有需要删除的过期文件，如果有，则删除这些过期文件以防Delta log文件无限增长。

参数	描述
spark.sql.sources.parallelPartitionDiscovery.parallelism	默认值为1000。 此参数为Delta扫描文件时所用的并行度。如果文件数量较少，则减小此值。目前仅使用在Vacuum中。如果设置不当，影响Vacuum扫描文件的效率。 **说明：** 此参数为Spark SQL参数。

11.7.2 附录2 优化建议

本文介绍关于事实表分区、保留时间和batch大小设置的优化建议。

优化建议如下：

- 建议事实表进行基于时间的分区，并定时运行Optimize动作。Optimize建议运行在已经完成的分区上，避开当前正在写入的分区（否则当前分区写完后仍然需要再进行一次针对当前分区的Optimize）。
- 定期运行Vacuum动作，设置合理的保留时间，默认保留时间为7天。
- 流式入库时，设置合理的batch大小。如果batch size较小，实时性会好一些，但是写入吞吐会下降。在上游数据压力较大时，适当提高batch size有助于提升写入吞吐，例如，在实时要求不是很高的场景，可以将batch size设置为1000或者10000。

11.7.3 附录3 常见问题

本文介绍Delta使用过程中遇到的一些常见问题以及解决方法。

为什么建表失败？

Delta建表需要制定LOCATION，这种表在Spark中为外表。建表时，如果目标目录不存在，即创建一张全新的表，理论上不会出现这种情况。如果LOCATION已经存在，那么基于此LOCATION建表应当确保，建表语句的Schema与LOCATION内Delta log中定义的Schema相同。

流式写入Delta时产生了很多的小文件怎么办？

用Spark Streaming写数据到Delta，本质上是执行一系列的mini batch，一个batch会产生一个或者多个文件。由于batch size通常较小，因此Spark Streaming连续运行会产生相当数量的小文件。解决方法有两种：

- 如果实时性要求不高，建议增大mini batch的trigger size。
- 定期运行Optimize，对表进行合并小文件的操作。

Optimize执行时间很长是什么原因？

如果长时间没有进行Optimize操作，Delta内可能会累积相当数量的小文件，此时运行Optimize可能执行时间会比较长。因此建议设置定时任务来定期触发Optimize动作。

为什么Optimize失败了？应该如何处理？

Optimize会有删除历史数据和写新数据的动作。由于Delta采用的乐观锁机制，写事务在提交的时候，其中一个写事务会失败。尤其是一个流式作业在不断地更新Delta内的数据（例如：CDC场景），此时Optimize失败的概率会更大（注意：如果流式作业仅仅是新增数据而不涉及删除或者更新，Optimize不会失败）。建议用户对表进行基于时间的分区，每当一个分区完成，对该分区进行Optimize操作。

执行了Optimize，为什么还有很多小文件？

Optimize是合并小文件，但是被合并的小文件不会被立即删除。因为Delta有访问历史的功能，因此如果要访问合并之前的历史版本，这些小文件会被用到。如果要删除这些小文件，请使用Vacuum命令。

执行了Vacuum，为什么还有很多小文件？

Vacuum动作是清理已经合并过的且已经超出了安全期的小文件。默认安全期为7天。如果小文件没有被合并过，或者合并过的小文件尚在安全期之内，Vacuum不会将之删除。

如果想删除最近产生的小文件（这些小文件已经被合并），应该如何处理？

不建议删除时间过近的小文件，因为Delta的历史访问功能可能会用到这些小文件。如果确实要这么做，有两种做法：

- 关闭安全期检查：`spark.databricks.delta.retentionDurationCheck.enabled=false`，这个设置可以在启动spark任务时作为参数传入。
- 修改全局的安全期为一个较小的值：例如在`spark-defaults.conf` 中设置`spark.databricks.delta.properties.defaults.deletedFileRetentionDuration interval 1 hour`。

执行了Vacuum，为什么还有很多的Delta log文件？

Vacuum动作是合并数据文件，并非合并Delta log文件。Delta log文件的合并和清理是Delta自动做的，每经历10个提交，会自动触发一次Delta log的合并，合并之后同时检查超出安全期的log文件，如果超出，则删除。默认Delta log的安全期为30天。

有没有自动触发Optimize或Vacuum的机制？

Delta仅仅是一个库，而非运行时，因此尚没有自动化的机制，但可以设置定时任务定期来触发Optimize或Vacuum的机制。

12 Presto

12.1 简介

Presto是一个开源的分布式SQL查询引擎，适用于交互式分析查询。

基本特性

Presto使用Java语言进行开发，具备易用、高性能和强扩展能力等特点，具体如下：

- 完全支持ANSI SQL。
- 支持丰富的数据源：
  - 与Hive数仓操作
  - Cassandra
  - Kafka
  - MongoDB
  - MySQL
  - PostgreSQL
  - SQL Server
  - Redis
  - Redshift
  - 本地文件
- 支持高级数据结构：
  - 支持数组和Map数据。
  - 支持JSON数据。
  - 支持GIS数据。
  - 支持颜色数据。
- 功能扩展能力强，Presto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

  您可以根据自身业务特点扩展相应的模块，实现高效的业务处理。
- 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WebUI，可视化的呈现查询任务执行过程。
  - 支持JMX协议。

应用场景

Presto是定位在数据仓库和数据分析业务的分布式SQL引擎，适合以下应用场景：

- ETL
- Ad-Hoc查询
- 海量结构化数据或半结构化数据分析
- 海量多维数据聚合或报表分析

注意：

Presto是一个数仓类产品，因为其对事务支持有限，所以不适合在线业务场景。

产品优势

EMR Presto产品除了开源Presto本身具有的优点外，还具备如下优势：

- 即买即用，快速完成上百节点的Presto集群搭建。
- 弹性扩容简单操作。
- 与EMR软件栈完美结合，支持处理存储在OSS的数据。
- 免运维7*24一站式服务。

更多参考

根据集群版本，获取Presto组件的版本号，详情请参见#unique_64。

请根据Presto组件的版本号，查看开源Presto文档：

- 当Presto版本是3XX时，修改prestosql.io/docs/3XX/中的版本号，在浏览器访问该链接。

  例如，当Presto版本是331时，访问prestosql.io/docs/331/，详情请参见Presto 331 Documentation。
- 当Presto版本是0.2XX时，修改prestodb.io/docs/0.2XX/中的版本号，在浏览器访问该链接。

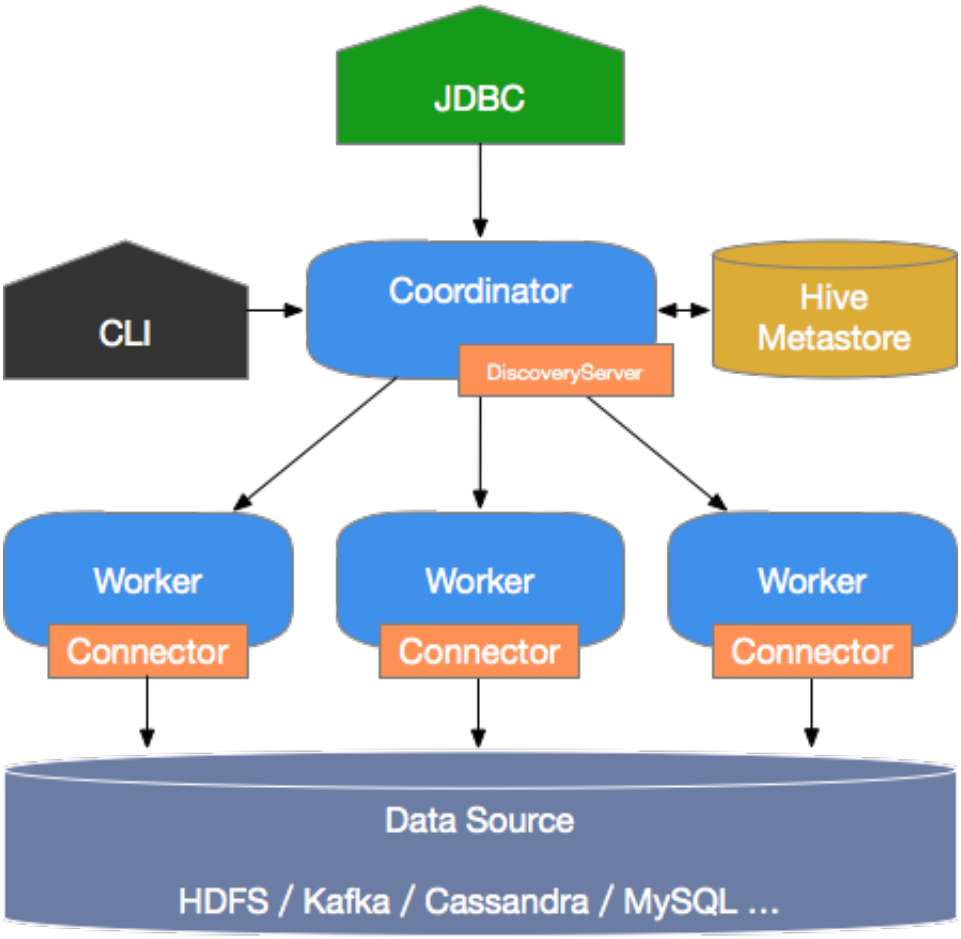
  例如，当Presto版本是0.228时，访问prestodb.io/docs/0.228/，详情请参见Presto 0.228 Documentation。

12.2 快速入门

12.2.1 系统组成

本节介绍Presto的系统组成。

Presto的系统组成如下图所示。



Presto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节点负责执行下发到任务，通过连接器读取外部存储系统到数据，进行处理，并将处理结果发送给Coordinator节点。

12.2.2 基本概念

本节介绍Presto中的基本概念，以便更好的理解Presto的工作机制。

数据模型

数据模型即数据的组织形式。Presto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。

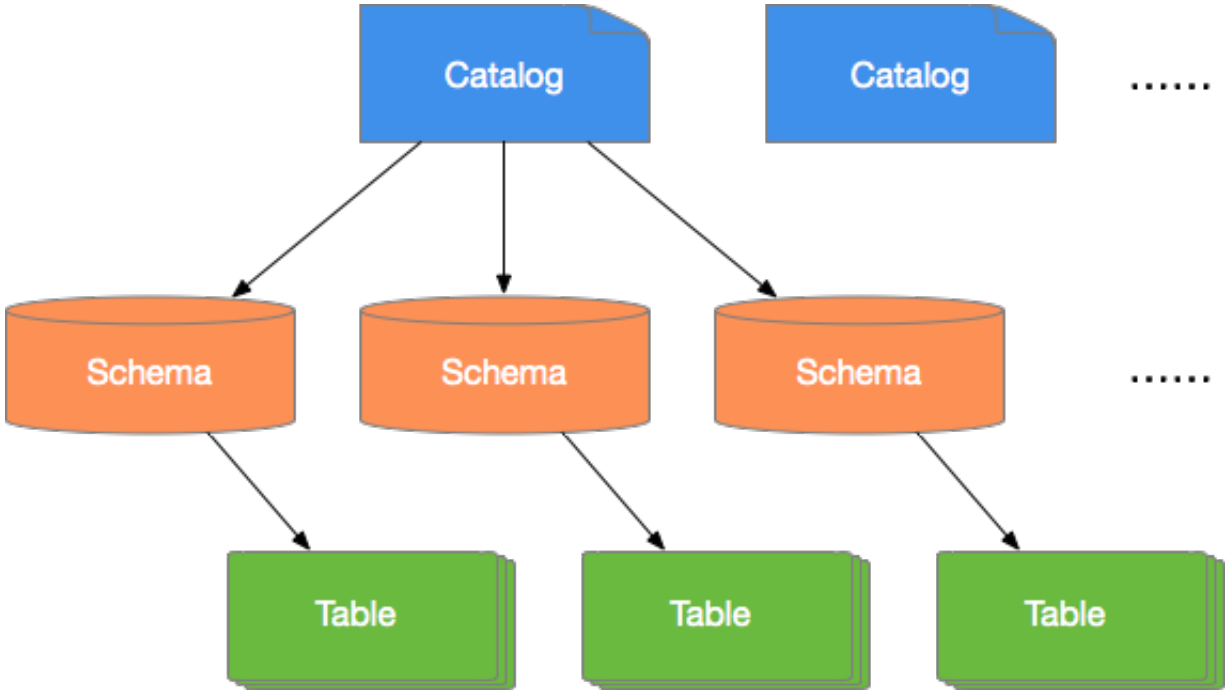
- Schema

  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。

Catalog、Schema和Table之间的关系如下图所示。



Connector

Presto通过各种Connector来接入多种外部数据源。Presto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一般会绑定一种类型的Connector（在Catalog的Properties文件中设置）。Presto内置了多种Connector。

12.2.3 使用命令行工具

本文介绍如何使用命令行工具操作Presto控制台。

操作步骤

1. 通过SSH方式登录集群，执行下面命令进入Presto控制台。

   登录详情请参见#unique_17。

   ```
   $ presto --server emr-header-1:9090 --catalog hive --schema default --user hadoop
   ```

   高安全集群使用如下命令形式。

   ```
   $ presto  --server https://emr-header-1:7778  \
         --enable-authentication \
         --krb5-config-path /etc/krb5.conf \
         --krb5-keytab-path  /etc/ecm/presto-conf/presto.keytab \
         --krb5-remote-service-name presto \
         --keystore-path /etc/ecm/presto-conf/keystore \
         --keystore-password 81ba14ce6084 \
         --catalog hive --schema default \
         --krb5-principal  presto/emr-header-1.cluster-XXXX@EMR.XXXX.COM
   ```

   说明：

   - XXXX为集群的ecm id，为一串数字，可以通过cat /etc/hosts获取。
   - 81ba14ce6084为/etc/ecm/presto-conf/keystore的默认密码，建议部署后替换为自己的keystore。

2. 执行以下命令，查看当前Catalog下的Schema。

   ```
   presto:default> show schemas;
     Schema
   --------------------
   default
   hive
   information_schema
   tpch_100gb_orc
   tpch_10gb_orc
   tpch_10tb_orc
   tpch_1tb_orc
   (7 rows)
   ```

   执行presto --help命令，可以获取控制台的帮助，各个参数解释如下所示。

   ```
   --server <server>                  # 指定Coordinator的URI
   --user <user>                     # 设置用户名
   --catalog <catalog>                  # 指定默认的Catalog
   --schema <schema>                    # 指定默认的Schema
   --execute <execute>                  # 执行一条语句，然后退出
   -f <file>, --file <file>            # 执行一个SQL文件，然后退出
   --debug                           # 显示调试信息
   --client-request-timeout <timeout>     # 指定客户端超时时间，默认为2m
   --enable-authentication              # 启用客户端认证
   --keystore-password <keystore password> # KeyStore密码
   --keystore-path <keystore path>        # KeyStore路径
   --krb5-config-path <krb5 config path>   # Kerberos配置文件路径（默认为/etc/krb5.conf
   )
   --krb5-credential-cache-path <path>    # Kerberos凭据缓存路径
   --krb5-keytab-path <krb5 keytab path>   # Kerberos Key table路径
   --krb5-principal <krb5 principal>      # 要使用的Kerberos principal
   ```

```
--krb5-remote-service-name <name>      # 远程Kerberos节点名称
--log-levels-file <log levels>         # 调试日志配置文件路径
--output-format <output-format>        # 批量导出的数据格式，默认为 CSV
--session <session>                    # 指定回话属性，格式如下 key=value
--socks-proxy <socks-proxy>            # 设置代理服务器
--source <source>                      # 设置查询的Source
--version                              # 显示版本信息
-h, --help                             # 显示帮助信息
```

12.2.4 使用 JDBC

Java 应用可以使用 Presto 提供的 JDBC driver 连接数据库，使用方式与一般 RDBMS 数据库差别不大。

在 Maven 中引入

可以在 pom 文件中加入如下配置引入 Presto JDBC driver：

```
<dependency>
  <groupId>com.facebook.presto</groupId>
  <artifactId>presto-jdbc</artifactId>
  <version>0.187</version>
</dependency>
```

Driver 类名

Presto JDBC driver 类为`com.facebook.presto.jdbc.PrestoDriver`。

连接字串

可以使用如下格式的连接字串：

```
jdbc:presto://<COORDINATOR>:<PORT>/[CATALOG]/[SCHEMA]
```

例如：

```
jdbc:presto://emr-header-1:9090            # 连接数据库，使用Catalog和Schema
jdbc:presto://emr-header-1:9090/hive       # 连接数据库，使用Catalog(hive)和Schema
jdbc:presto://emr-header-1:9090/hive/default # 连接数据库，使用Catalog(hive)和
Schema(default)
```

连接参数

Presto JDBC driver 支持很多参数，这些参数既可以通过 **Properties** 对象传入，也可以通过 URL 传入参数，这两种方式是等价的。

通过 **Properties** 对象传入示例：

```
String url = "jdbc:presto://emr-header-1:9090/hive/default";
Properties properties = new Properties();
properties.setProperty("user", "hadoop");
Connection connection = DriverManager.getConnection(url, properties);
```

```
......
```

通过 URL 传入参数示例：

```
String url = "jdbc:presto://emr-header-1:9090/hive/default?user=hadoop";
Connection connection = DriverManager.getConnection(url);
......
```

各参数说明如下：

参数名称	格式	参数说明
user	STRING	用户名
password	STRING	密码
socksProxy	\:\	SOCKS 代理服务器地址，如 localhost:1080
httpProxy	\:\	HTTP 代理服务器地址，如 localhost:8888
SSL	true\	是否使用 HTTPS 连接，默认为 false
SSLTrustStorePath	STRING	Java TrustStore 文件路径
SSLTrustStorePassword	STRING	Java TrustStore 密码
KerberosRemoteServiceName	STRING	Kerberos 服务名称
KerberosPrincipal	STRING	Kerberos principal
KerberosUseCanonicalHostname	true\	是否使用规范化主机名，默认为 false
KerberosConfigPath	STRING	Kerberos 配置文件路径
KerberosKeytabPath	STRING	Kerberos KeyTab 文件路径
KerberosCredentialCachePath	STRING	Kerberos credential 缓存路径

Java 示例

下面给出一个 Java 使用 Presto JDBC driver 的例子。

```
.....
// 加载JDBC Driver类
try {
    Class.forName("com.facebook.presto.jdbc.PrestoDriver");
} catch(ClassNotFoundException e) {
    LOG.ERROR("Failed to load presto jdbc driver.", e);
    System.exit(-1);
}
Connection connection = null;
Statement statement = null;
try {
    String url = "jdbc:presto://emr-header-1:9090/hive/default";
    Properties properties = new Properties();
```

```
    properties.setProperty("user", "hadoop");
    // 创建连接对象
    connection = DriverManager.getConnection(url, properties);
    // 创建Statement对象
    statement = connection.createStatement();
    // 执行查询
    ResultSet rs = statement.executeQuery("select * from t1");
    // 获取结果
    int columnNum = rs.getMetaData().getColumnCount();
    int rowIndex = 0;
    while (rs.next()) {
        rowIndex++;
        for(int i = 1; i <= columnNum; i++) {
            System.out.println("Row " + rowIndex + ", Column " + i + ": " + rs.getInt(i));
        }
    }
} catch(SQLException e) {
    LOG.ERROR("Exception thrown.", e);
} finally {
  // 销毁Statement对象
  if (statement != null) {
      try {
        statement.close();
    } catch(Throwable t) {
        // No-ops
    }
  }
  // 关闭连接
  if (connection != null) {
      try {
        connection.close();
    } catch(Throwable t) {
        // No-ops
    }
  }
}
```

使用反向代理

可以使用 HAProxy 反向代理 Coodinator，实现通过代理服务访问 Presto 服务。

非安全集群配置集群代理步骤如下：

1. 在代理节点安装 HAProxy
2. 修改 HAProxy 配置（/etc/haproxy/haproxy.cfg），添加如下内容：

```
......

listen prestojdbc :9090
    mode tcp
    option tcplog
    balance source
    server presto-coodinator-1 emr-header-1:9090
```

3. 重启 HAProxy 服务

至此，可以使用代理服务器来访问 Presto 了，只需要将连接的服务器 IP 改为代理服务的 IP 即可。

12.2.5 通过 Gateway 访问

本节介绍使用 HAProxy 反向代理实现通过 Gateway 节点访问 Presto 服务的方法。该方法也很容易扩展到其他组件，如 Impala 等。

Gateway 是与 EMR 集群处于同一个内网中的 ECS 服务器，可以使用 Gateway 实现负载均衡和安全隔离。您可以通过**控制台页面** > **概览** > **创建 Gateway**，也可以通过**控制台页面** > **集群管理** > **创建 Gateway**来创建对应集群的 Gateway 节点。

说明：

Gateway 节点默认已经安装了 HAProxy 服务，但没有启动。

普通集群

普通集群配置 Gateway 代理比较简单，只需要配置 HAProxy 反向代理，对 EMR 集群上Header 节点的 Presto Coodrinator 的 9090 端口实现反向代理即可。配置步骤如下：

1. 配置 HAProxy

   通过 SSH 登入Gateway节点，修改 HAProxy 的配置文件/etc/haproxy/haproxy.cfg。添加如下内容：

```
#---------------------------------------------------------------------
# Global settings
#---------------------------------------------------------------------
global
......
## 配置代理，将Gateway的9090端口映射到emr-header-1.cluster-xxxx的9090端口
listen prestojdbc :9090
   mode tcp
   option tcplog
   balance source
   server presto-coodinator-1 emr-header-1.cluster-xxxx:9090
```

2. 保存退出，使用如下命令重启 HAProxy 服务：

```
$> service haproxy restart
```

3. 配置安全组

   需要配置的规则如下：

方向	配置规则	说明
公网入	自定义 TCP，开放 9090 端口	该端口用于 HAProxy 代理 Header 节点 Coodinator 端口

至此就可以通过 ECS 控制台删除 Header 节点的公网 IP，在自己的客户机上通过 Gateway 访问 Presto 服务了。

- 命令行使用 Presto 的示例请参见使用命令行工具。

- JDBC 访问 Presto 的示例请参见使用 JDBC。

高安全集群

EMR 高安全集群中的 Presto 服务使用 Kerberos 服务进行认证，其中 Kerberos KDC 服务位于 emr-header-1 上，端口为 88，同时支持 TCP/UDP 协议。 使用 Gateway 访问高安全集群中的 Presto 服务，需要同时对 Presto Coodinator 服务端口和 Kerberos KDC 实现代理。另外，EMR Presto Coodinator 集群默认使用 keystore 配置的 CN 为 emr-header-1，只能在内网使用，因此需要重新生成 CN=emr-header-1.cluster-xxx 的 keystore。

- HTTPS 认证相关

1. 创建服务端 CN=emr-header-1.cluster-xxx的keystore。

```
[root@emr-header-1 presto-conf]# keytool -genkey -dname "CN=emr-header-1.
cluster-xxx,OU=Alibaba,O=Alibaba,L=HZ, ST=zhejiang, C=CN" -alias server -keyalg
RSA -keystore keystore -keypass 81ba14ce6084 -storepass 81ba14ce6084 -validity
36500
Warning:
JKS 密钥库使用专用格式。建议使用 "keytool -importkeystore -srckeystore keystore -
destkeystore keystore -deststoretype pkcs12" 迁移到行业标准格式 PKCS12。
```

2. 导出证书。

```
[root@emr-header-1 presto-conf]# keytool -export -alias server -file server.cer -
keystore keystore -storepass 81ba14ce6084
存储在文件 <server.cer> 中的证书
Warning:
JKS 密钥库使用专用格式。建议使用 "keytool -importkeystore -srckeystore keystore -
destkeystore keystore -deststoretype pkcs12" 迁移到行业标准格式 PKCS12。
```

3. 制作客户端 keystore。

```
[root@emr-header-1 presto-conf]# keytool -genkey -dname "CN=myhost,OU=
Alibaba,O=Alibaba,L=HZ, ST=zhejiang, C=CN" -alias client -keyalg RSA -keystore
client.keystore -keypass 123456 -storepass 123456 -validity 36500
Warning:
JKS 密钥库使用专用格式。建议使用 "keytool -importkeystore -srckeystore client.
keystore -destkeystore client.keystore -deststoretype pkcs12" 迁移到行业标准格式
PKCS12。
```

4. 将证书导入到客户端 keystore 中。

```
[root@emr-header-1 presto-conf]# keytool -import -alias server -keystore client.
keystore -file server.cer -storepass 123456
所有者: CN=emr-header-2.cluster-xxx, OU=Alibaba, O=Alibaba, L=HZ, ST=zhejiang,
C=CN
发布者: CN=emr-header-2.cluster-xxx, OU=Alibaba, O=Alibaba, L=HZ, ST=zhejiang,
C=CN
序列号: 4247108
有效期为 Thu Mar 01 09:11:31 CST 2018 至 Sat Feb 05 09:11:31 CST 2118
证书指纹:
    MD5:  75:2A:AA:40:01:5B:3F:86:8F:9A:DB:B1:85:BD:44:8A
    SHA1: C7:25:B9:AD:5F:FE:FC:05:8E:A0:24:4A:1C:AA:6A:8D:6C:39:28:16
    SHA256: DB:86:69:65:73:D5:C6:E2:98:7C:4A:3B:31:EF:70:80:F0:3C:3B:0C:14:94:
37:9F:9C:22:47:EA:7E:1E:DE:8C
签名算法名称: SHA256withRSA
```

```
主体公共密钥算法: 2048 位 RSA 密钥
版本: 3
扩展:
#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 45 1D A9 C7 D5 4E BB CF   BD CE B4 5E E2 16 FB 2F  E....N.....^.../
0010: E9 5D 4A B6                                        .]J.
]
]
是否信任此证书? [否]:  是
证书已添加到密钥库中
Warning:
JKS 密钥库使用专用格式。建议使用 "keytool -importkeystore -srckeystore client.
keystore -destkeystore client.keystore -deststoretype pkcs12" 迁移到行业标准格式
PKCS12。
```

5. 将生成的文件拷贝到客户端。

```
$> scp root@xxx.xxx.xxx.xxx:/etc/ecm/presto-conf/client.keystore ./
```

- Kerberos 认证相关

1. 添加客户端用户 principal。

```
[root@emr-header-1 presto-conf]# sh /usr/lib/has-current/bin/hadmin-local.sh /
etc/ecm/has-conf -k /etc/ecm/has-conf/admin.keytab
[INFO] conf_dir=/etc/ecm/has-conf
Debug is  true storeKey true useTicketCache false useKeyTab true doNotPromp
t true ticketCache is null isInitiator true KeyTab is /etc/ecm/has-conf/admin.
keytab refreshKrb5Config is true principal is kadmin/EMR.xxx.COM@EMR.xxx.COM
tryFirstPass is false useFirstPass is false storePass is false clearPass is false
Refreshing Kerberos configuration
principal is kadmin/EMR.xxx.COM@EMR.xxx.COM
Will use keytab
Commit Succeeded
Login successful for user: kadmin/EMR.xxx.COM@EMR.xxx.COM
enter "cmd" to see legal commands.
HadminLocalTool.local: addprinc -pw 123456 clientuser
Success to add principal :clientuser
HadminLocalTool.local: ktadd -k /root/clientuser.keytab clientuser
Principal export to keytab file : /root/clientuser.keytab successful .
HadminLocalTool.local: exit
```

2. 将生成的文件拷贝到客户端。

```
$> scp root@xxx.xxx.xxx.xxx:/root/clientuser.keytab ./
$> scp root@xxx.xxx.xxx.xxx:/etc/krb5.conf ./
```

3. 修改拷贝到客户端的 krb5.conf 文件，修改如下两处。

```
[libdefaults]
   kdc_realm = EMR.xxx.COM
   default_realm = EMR.xxx.COM
   # 修改参数为1，使客户端使用TCP协议与KDC通信（因为HAProxy不支持UDP协议）
   udp_preference_limit = 1
   kdc_tcp_port = 88
   kdc_udp_port = 88
   dns_lookup_kdc = false
[realms]
   EMR.xxx.COM = {
```

```
    # 设置为Gateway的外网IP
    kdc = xxx.xxx.xxx.xxx:88
  }
```

4. 修改客户端主机的 hosts 文件，添加如下内容。

```
#  gateway ip
xxx.xxx.xxx.xxx emr-header-1.cluster-xxx
```

- 配置 Gateway HAProxy。

1. 通过 SSH 登入到 Gateway 节点，修改/etc/haproxy/haproxy.cfg。添加如下内容。

```
#---------------------------------------------------------------------
# Global settings
#---------------------------------------------------------------------
global
......
listen prestojdbc :7778
   mode tcp
   option tcplog
   balance source
   server presto-coodinator-1 emr-header-1.cluster-xxx:7778
listen kdc :88
   mode tcp
   option tcplog
   balance source
   server emr-kdc emr-header-1:88
```

2. 保存退出，使用如下命令重启HAProxy服务。

```
$> service haproxy restart
```

3. 配置安全组规则

需要配置的规则如下：

方向	配置规则	说明
公网入	自定义 UDP，开放 88 端口	该端口用于 HAProxy 代理 Header 节点上的 KDC
公网入	自定义 TCP，开放 88 端口	该端口用于 HAProxy 代理Header节点上的 KDC
公网入	自定义 TCP，开放 7778 端口	该端口用于 HAProxy 代理 Header 节点 Coodinator 端口

至此，就可以通过 ECS 控制台删除 Header 节点的公网 IP，在自己的客户机上通过 Gateway 访问 Presto 服务了。

- 使用 JDBC 访问 Presto 示例

代码如下。

```
try {
   Class.forName("com.facebook.presto.jdbc.PrestoDriver");
```

```
} catch(ClassNotFoundException e) {
    LOG.error("Failed to load presto jdbc driver.", e);
    System.exit(-1);
}
Connection connection = null;
Statement statement = null;
try {
    String url = "jdbc:presto://emr-header-1.cluster-59824:7778/hive/default";
    Properties properties = new Properties();
    properties.setProperty("user", "hadoop");
    // https相关配置
    properties.setProperty("SSL", "true");
    properties.setProperty("SSLTrustStorePath", "resources/59824/client.keystore");
    properties.setProperty("SSLTrustStorePassword", "123456");
    // Kerberos相关配置
    properties.setProperty("KerberosRemoteServiceName", "presto");
    properties.setProperty("KerberosPrincipal", "clientuser@EMR.59824.COM");
    properties.setProperty("KerberosConfigPath", "resources/59824/krb5.conf");
    properties.setProperty("KerberosKeytabPath", "resources/59824/clientuser.keytab
");
    // 创建连接对象
    connection = DriverManager.getConnection(url, properties);
    // 创建Statement对象
    statement = connection.createStatement();
    // 执行查询
    ResultSet rs = statement.executeQuery("select * from table1");
    // 获取结果
    int columnNum = rs.getMetaData().getColumnCount();
    int rowIndex = 0;
    while (rs.next()) {
        rowIndex++;
        for(int i = 1; i <= columnNum; i++) {
            System.out.println("Row " + rowIndex + ", Column " + i + ": " + rs.getString(i));
        }
    }
} catch(SQLException e) {
    LOG.error("Exception thrown.", e);
} finally {
    // 销毁Statement对象
    if (statement != null) {
        try {
            statement.close();
            } catch(Throwable t) {
              // No-ops
            }
    }
   // 关闭连接
   if (connection != null) {
         try {
          connection.close();
       } catch(Throwable t) {
         // No-ops
       }
    }
```

```
}
```

12.3 数据类型

Presto 默认支持多种常见的数据类型，包括布尔类型、整型、浮点型、字符串型、日期型等。同时，用户可以通过插件等方式增加自定义的数据类型。并且，自定义的 Presto 连接器不需要支持所有数据类型。

数值类型

Presto 内置支持如下几种数值类型：

- BOOLEAN

  表示一个二值选项，值为TRUE或FALSE。
- TINYINT

  表示一个 8 位有符号整型，二进制补码形式存储。
- SMALLINT

  表示一个 16 位有符号整型，二进制补码形式存储。
- INTEGER

  表示一个 32 位有符号整型，二进制补码形式存储。
- BIGINT

  表示一个 64 位有符号整型，二进制补码形式存储。
- REAL

  一个 32 位多精度的二进制浮点数值类型。
- DOUBLE

  一个 64 位多精度的二进制浮点数值类型。
- DECIMAL

  一个固定精度的数值类型，最大可支持 38 位有效数字，有效数字在 17 位以下性能最好。定义DECIMAL类型字段时需要确定两个字面参数：

  1. 精度（precision）数值总的位数，不包括符号位。

  2. 范围（scale）小数位数，可选参数，默认为 0。

  示例：DECIMAL '-10.7'该值可用DECIMAL(3，1)类型表示。

下表说明整型数值类型的位宽和取值范围：

数值类型	位宽	最小值	最大值
TINYINT	8 bit	-2^7	2^7 - 1
SMALLINT	16 bit	2^15	2^15 - 1
INTEGER	32 bit	-2^31	-2^31 - 1
BIGINT	64 bit	-2^63	-2^63 - 1

字符类型

Presto 内置支持如下几种字符类型：

- VARCHAR

  表示一个可变长度的字符串类型，可以设置最大长度。

  示例：VARCHAR， VARCHAR(10)

- CHAR

  表示一个固定长度的字符串类型，使用时可以指定字符串的长度，不指定则默认为1。

  示例：CHAR， CHAR(10)

  注意：
  指定了长度的字符串总是包含与该长度相同的字符，如果字符串字面长度小于指定长度，则不足部分会用不可见字符填充，填充部分也会参与字符比较，因此，两个字面量一样的字段，如果它们的长度定义不一样，就永远不可能相等。

- VARBINARY 表示一块可变长度的二进制数据。

日期和时间

Presto 内置支持如下几种时间和日期类型：

- DATE

  表示一个日期类型的字段，日期包括年、月、日，但是不包括时间。

  示例： DATE '1988-01-30'

- TIME

  表示一个时间类型，包括时、分、秒、毫秒。时间类型可以加时区进行修饰。

  示例：

  - TIME '18:01:02.345'， 无时区定义，使用系统时区进行解析。
  - TIME '18:01:02.345 Asia/Shanghai'， 有时区定义，使用定义的时区进行解析。

- TIMESTAMP

  表示一个时间戳类型的字段，时间戳包含了日期和时间两个部分的信息，取值范围为'1970-01-01 00:00:01' UTC到'2038-01-19 03:14:07' UTC，支持使用时区进行修饰。

  示例：TIMESTAMP '1988-01-30 01:02:03.321'， TIMESTAMP '1988-01-30 01:02:03.321 Asia/Shanghai'

- INTERVAL

  主要用于时间计算表达式中，表示一个间隔，单位可以是如下几个：

  - YEAR - 年
  - QUARTER - 季度
  - MONTH - 月
  - DAY - 天
  - HOUR - 小时
  - MINUTE - 分钟
  - SECOND - 秒
  - MILLISECOND - 毫秒

  示例： DATE '2012-08-08' + INTERVAL '2' DAY

复杂类型

Presto内置支持多种复杂的数据类型，以便支持更加复杂的业务场景，这些类型包括：

- JSON

  表示字段为一个 JSON 字符串，包括 JSON 对象、JSON 数组、JSON 单值（数值或字符串），还包括布尔类型的true、false以及表示空的null。

  示例：

  - JSON '[1， null， 1988]'
  - JSON '{"k1":1， "k2": "abc"}'

- ARRAY

  表示一个数组，数组中各个元素的类型必须一致。

  示例：ARRAY[1， 2， 3]

- MAP

  表示一个映射关系，由键数组和值数组组成。

  示例：MAP(ARRAY['foo'， 'bar']， ARRAY[1， 2])

- ROW

  表示一行数据，行中每个列都有列名对应，可以使用.运算符+列名的方式来访问数据列。

  示例：CAST(ROW(1988， 1.0， 30) AS ROW(y BIGINT， m DOUBLE， d TINYINT))

- IPADDRESS

  表示一个 IPv4 或 IPv6 地址。内部实现上，将 IPv4 处理成 IPv6 地址使用（IPv4 到 IPv6 映射表）。

  示例：IPADDRESS '0.0.0.0'， IPADDRESS '2001:db8::1'

12.4 常用连接器

12.4.1 Kafka 连接器

本连接器将Kafka上的topic映射为Presto中的表。Kafka中的每条记录都映射为Presto表中的消息。

> **注意：**
>
> 由于Kafka中数据是动态变化的，因此，在使用Presto进行多次查询时，可能会出现数据不一致的情形。目前，Presto还没有处理这些情况的能力。

配置

创建etc/catalog/kafka.properties文件，添加如下内容，启用 Kafka 连接器。

```
connector.name=kafka
kafka.table-names=table1,table2
kafka.nodes=host1:port,host2:port
```

> **说明：**
>
> Presto 可以同时连接多个 Kafka 集群，只需要在配置目录中添加新的 properties 文件即可，文件名将会被映射为 Presto 的 catalog。如新增一个orders.properties配置文件，Presto 会创建一个新的名为ordersorders的 catalog。

```
## orders.properties
connector.name=kafka  # 这个表示连接器类型，不能变
kafka.table-names=tableA,tableB
kafka.nodes=host1:port,host2:port
```

Kafka 连接器提供了如下几个属性：

参数	是否必选	描述	说明
kafka.table-names	是	定义本连接器支持的表格列表	这里的文件名可以使用 Schema 名称进行修饰，形式如{schema_name}.{table_name}。也可以不使用 Schema 名称修饰，此时，表格将被映射到kafka.default-schema中定义的schema中。
kafka.default-schema	-	默认的Schema名称，默认值为default	-
kafka.nodes	是	Kafka集群中的节点列表	配置格式形如hostname:port[,hostname:port...]。此处可以只配置部分 Kafka 节点，但是，Presto 必须能够连接到 Kafka 集群中的所有节点。否则，可能拿不到部分数据。
kafka.connect-timeout	否	连接器与 Kafka 集群的超时时间，默认为 10 秒	如果 Kafka 集群压力比较大，创建连接可能需要相当长的时间，从而导致 Presto 在执行查询时出现超时的情况。此时，增加当前的配置值是一个不错的选择。
kafka.buffer-size	否	读缓冲区大小，默认为 64 kb	用于设置从 Kafka 读取数据的内部数据缓冲区的大小。 数据缓冲区必须至少大于一条消息的大小。 每个 Worker 和数据节点都会分配一个数据缓冲区。
kafka.table-description-dir	否	Topic（表）描述文件目录，默认为etc/kafka	该目录下保存着 JSON 格式的数据表定义文件（必须使用.json作为后缀）。

参数	是否必选	描述	说明
kafka.hide-internal-columns	否	需要隐藏的预置列清单，默认为true	除了在表格描述文件中定义的数据列之外，连接器还为每个表格维护了许多额外的列。本属性用于控制这些列在是否会在DESCRIBE <table-name>和SELECT *语句的执行结果中显示。无论本配置设置是什么，这些列都可以参与查询过程中。

Kafka连接器提供的内部列如下表所示：

列名	类型	说明
_partition_id	BIGINT	本行记录所在的 Kafka分区 ID。
_partition_offset	BIGINT	本行记录在 Kafka 分区中的偏移量。
_segment_start	BIGINT	包含此行的数据段的最低偏移量。此偏移量是针对每个分区的。
_segment_end	BIGINT	包含此行的数据段的最大偏移量（为下一个段的起始偏移量）。此偏移量是针对每个分区的。
_segment_count	BIGINT	当前行在数据段中的序号，对于没有压缩的topic, _segment_start+ _segment_count = _partition_offset。
_message_corrupt	BOOLEAN	如果解码器无法解码本行记录，本字段将会被设为TRUE。
_message	VARCHAR	将消息字节作为 UTF-8 编码的字符串。当 topic 的消息为文本类型时，这个字段比较有用。
_message_length	BIGINT	本行消息的字节长度。
_key_corrupt	BOOLEAN	如果键解码器无法解码本行记录，该字段将会被设为TRUE。

列名	类型	说明
_key	VARCHAR	将键字节作为 UTF-8 编码的字符串。当 topic 的消息为文本类型时，这个字段比较有用。
_key_length	BIGINT	消息中键的字节长度。

说明：

对于那些没有定义文件的表，_key_corrupt和_message_corrupt默认为FALSE。

表格定义文件

Kafka 本身是 Schema-Less 的消息系统，消息的格式需要生产者和消费者自己来定义。而 Presto 要求数据必须可以被映射成表的形式。因此，通常需要用户根据消息的实际情况，提供对应的表格定义文件。对于 JSON 格式的消息，如果没有提供定义文件，也可以在查询时使用Presto的JSON函数进行解析。这种处理方式虽然比较灵活，但是会增加 SQL 语句的编写难度。

一个表格描述文件使用 JSON 来定义一个表，文件名可以任意，但是必须以.json作为扩展名。

```
{
    "tableName": ...,
    "schemaName": ...,
    "topicName": ...,
    "key": {
        "dataFormat": ...,
        "fields": [
            ...
        ]
    },
    "message": {
        "dataFormat": ...,
        "fields": [
            ...
      ]
    }
}
```

字段	可选性	类型	说明
tableName	必选	string	Presto 表名。
schemaName	可选	string	表所在的 Schema 名称。
topicName	必选	string	Kafka topic 名称。
key	可选	JSON object	消息键到列的映射规则。
message	可选	JSON object	消息到列的映射规则。

其中，键和消息的映射规则使用如下几个字段来描述。

字段	可选性	类型	说明
dataFormat	必选	string	设置一组列的解码器。
fields	必选	JSON 数组	列定义列表。

这里的fields是一个 JSON 数组，每一个元素为如下的 JOSN 对象：

```
{
    "name": ...,
    "type": ...,
    "dataFormat": ...,
    "mapping": ...,
    "formatHint": ...,
    "hidden": ...,
    "comment": ...
}
```

字段	可选性	类型	说明
name	必选	string	列名
type	必选	string	列的数据类型
dataFormat	可选	string	列数据解码器
mapping	可选	string	解码器参数
formatHint	可选	string	设置在该列上的提示，可以被解码器使用
hidden	可选	boolean	是否隐藏
comment	可选	string	列的描述

解码器

解码器的功能是将 Kafka 的消息（key+message）映射到数据列表中。如果没有表的定义文件，Presto 将使用dummy解码器。

Kafka 连接器提供了如下 3 中解码器：

- raw - 不做转换，直接使用原始的 bytes。
- csv - 将消息作为 CSV 格式的字符串进行处理。
- json - 将消息作为 JSON 进行处理。

12.4.2 JMX 连接器

可以通过 JMX 连接器查询 Presto 集群中所有节点的 JMX 信息。本连接器通常用于系统监控和调试。

通过修改本连接器的配置，可以实现 JMX 信息定期转储的功能。

配置

创建etc/catalog/jmx.properties文件，添加如下内容，启用 JMX 连接器。

```
connector.name=jmx
```

如果希望定期转储 JMX 数据，可以在配置文件中添加如下内容：

```
connector.name=jmx
jmx.dump-tables=java.lang:type=Runtime,com.facebook.presto.execution.scheduler:
name=NodeScheduler
jmx.dump-period=10s
jmx.max-entries=86400
```

其中：

- dump-tables是用逗号隔开的 MBean（Managed Beans）列表。该配置项指定了每个采样周期哪些 MBean 指标会被采样并存储到内存中；
- dump-period用于设置采样周期，默认为 10 s；
- max-entries用于设置历史记录的最大长度，默认为 86400 条。

如果指标项的名称中包含逗号，则需要使用\\,进行转义，如下所示：

```
connector.name=jmx
jmx.dump-tables=com.facebook.presto.memory:type=memorypool\\,name=general,\
  com.facebook.presto.memory:type=memorypool\\,name=system,\
  com.facebook.presto.memory:type=memorypool\\,name=reserved
```

数据表

JMX 连接器提供了两个 schemas，分别为current和history。其中：

current中包含了 Presto 集群中每个节点当前的 MBean，MBean的名称即为current中的表名（如果 bean 的名称中包含非标准字符，则需在查询时用双引号将表名扩起来），可以通过如下语句获取：

```
SHOW TABLES FROM jmx.current;
```

示例：

```
--- 获取每个节点的jvm信息
SELECT node, vmname, vmversion
FROM jmx.current."java.lang:type=runtime";
```

```
    node    |           vmname            | vmversion
------------+-------------------------------------+-----------
 ddc4df17-xxx | Java HotSpot(TM) 64-Bit Server VM | 24.60-b09
(1 row)
```

```
--- 获取每个节点最大和最小的文件描述符个数指标
SELECT openfiledescriptorcount, maxfiledescriptorcount
FROM jmx.current."java.lang:type=operatingsystem";

 openfiledescriptorcount | maxfiledescriptorcount
-------------------------+------------------------
                     329 |                  10240
(1 row)
```

history中包含了配置文件中配置的需要转储的指标对应的数据表。可以通过如下语句进行查询：

```
SELECT "timestamp", "uptime" FROM jmx.history."java.lang:type=runtime";

       timestamp       | uptime
-----------------------+--------
 2016-01-28 10:18:50.000 |  11420
 2016-01-28 10:19:00.000 |  21422
 2016-01-28 10:19:10.000 |  31412
(3 rows)
```

12.4.3 系统连接器

通过本连接器可以使用 SQL 查询 Presto 集群的基本信息和度量。

配置

无需配置，所有信息都可以通过名为system的 catalog 获取。

示例如下。

```
--- 列出所有支持的数据项目
SHOW SCHEMAS FROM system;

--- 列出运行时项目中的所有数据项
SHOW TABLES FROM system.runtime;

--- 获取节点状态
SELECT * FROM system.runtime.nodes;

    node_id |        http_uri        | node_version | coordinator | state
------------+------------------------+--------------+-------------+--------
 3d7e8095-...| http://192.168.1.100:9090| 0.188        | false       | active
 7868d742-...| http://192.168.1.101:9090| 0.188        | false       | active
 7c51b0c1-...| http://192.168.1.102:9090| 0.188        | true        | active

--- 强制取消一个查询
CALL system.runtime.kill_query('20151207_215727_00146_tx3nr');
```

数据表

本连接器提供了如下几个数据表。

TABLE	SCHEMA	说明
catalogs	metadata	该表包含了本连接器支持的所有 catalog 列表。
schema_properties	metadata	本表包含创建新 Schema 时可以设置的可用属性列表。
table_properties	metadata	本表包含创建新表时可以设置的可用属性列表。
nodes	runtime	本表包含了 Presto 集群中所有可见节点及其状态列表。
queries	runtime	查询表包含了 Presto 群集上当前和最近运行的查询的信息，包括原始查询文本（SQL），运行查询的用户的身份以及有关查询的性能信息，如查询排队和分析的时间等。
tasks	runtime	任务表包含了关于 Presto 查询中涉及的任务的信息，包括它们的执行位置以及每个任务处理的行数和字节数。
transactions	runtime	本表包含当前打开的事务和相关元数据的列表。这包括创建时间，空闲时间，初始化参数和访问目录等信息。

存储过程

本连接器支持如下存储过程。

runtime.kill_query(id)

取消给定 id 的查询。

12.5 常用函数和操作符

12.5.1 逻辑运算符

本文为您介绍Presto支持的逻辑操作符及其计算规则。

Presto支持与、或、非三种逻辑操作符，并支持 NULL参与逻辑运算，如下所示。

```
SELECT CAST(null as boolean) AND true; --- null
SELECT CAST(null AS boolean) AND false; -- false
SELECT CAST(null AS boolean) AND CAST(null AS boolean); -- null
SELECT NOT CAST(null AS boolean); -- null
```

完整的运算真值表如下所示。

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	TRUE	NULL	TRUE
NULL	FALSE	FALSE	NULL
NULL	NULL	NULL	NULL

另外， NOT NULL的结果为NULL。

12.5.2 比较函数和运算符

比较操作符

Presto 支持的比较操作如下所示：

操作符	说明
<	小于。
>	大于。
<=	小于等于。
>=	大于等于。
=	等于。
<>/!=	不等于。
[NOT] BETWEEN	值 X [不]介于 min 和 max 之间。

操作符	说明
IS [NOT] NULL	判断是否为 NULL。
IS [NOT] DISTINCT FROM	用于比较两个值是否一致。一般情况下，NULL作为未定义数据存在，任何有NULL 参与的比较都会返回NULL。但是IS [NOT] DISTINCT FROM将NULL作为值处理，返回TRUE或FALSE。

比较函数

Presto 提供了如下几个比较操作相关的函数：

- GREATEST

  返回最大值。

  示例：GREATEST(1, 2)
- LEAST

  返回最小值。

  示例：LEAST(1, 2)

比较相关的修饰谓语

Presto 还提供了几个比较相关的修饰谓语，可以增强比较语义的表达能力。使用方式如下。

```
<EXPRESSION><OPERATOR><QUANTIFIER> (<SUBQUERY>)
```

例如：

```
SELECT 'hello' = ANY (VALUES 'hello', 'world'); -- true
SELECT 21 < ALL (VALUES 19, 20, 21); -- false
SELECT 42 >= SOME (SELECT 41 UNION ALL SELECT 42 UNION ALL SELECT 43); -- true
```

其中，ANY、ALL、SOME就是比较修饰谓语。

- A = ALL (...) A 和所有值相等，则返回 TRUE。
- A <> ALL (...) A 和所有值不相等，则返回 TRUE。
- A < ALL (...) A 小于所有值，则返回 TRUE。
- A = ANY (...) A 只要等于其中一个值，则返回 TRUE，等价于A IN (...)。
- A <> ANY (...) A 只要和其中一个值不相等，则返回 TRUE，等价于A IN (...)。
- A < ANY (...) A 只要小于其中一个值，则返回 TRUE。

ANY和SOME含义相同，使用时可以互换。

12.5.3 条件表达式

条件表达式主要用于表达分支逻辑。本文为您介绍如何使用Presto的条件表达式。

CASE

在标准SQL中，CASE表达式如下所示。

```
CASE expression
   WHEN <value|condition> THEN result
   [ WHEN ... ]
   [ ELSE result]
END
```

CASE语句将比较expression和value|condition中的值或条件，如果符合（值相等或条件匹配）则返回结果。

- 示例一

```
SELECT a,
     CASE a
        WHEN 1 THEN 'one'
        WHEN 2 THEN 'two'
        ELSE 'many'
     END
```

- 示例二

```
SELECT a, b,
     CASE
        WHEN a = 1 THEN 'aaa'
        WHEN b = 2 THEN 'bbb'
        ELSE 'ccc'
     END
```

IF

IF是比较函数，用于简化两值比较逻辑的写法。表达形式如下。

```
IF(condition, true_value, [false_value])
```

如果condition返回TRUE，则函数返回true_value，否则返回false_value。其中false_value可选，不设置则返回NULL。

COALESCE

函数COALESCE返回的是它第一个不是空值的参数。只有在所有参数都为空值的情况下，才会返回空值。表达形式如下。

```
COALESCE(value1, value2[, ...])
```

NULLIF

函数NULLIF在value1与value2相等时，返回NULL，否则返回value1。表达形式如下。

```
NULLIF(value1, value2)
```

TRY

函数TRY会捕获expression计算过程中抛出的异常，并返回NULL。TRY捕获的异常包括如下几类：

- 除零异常，如x/0。
- 类型转换错误。
- 数值越界。

通常和COALESCE一起使用，以便在出错时，返回默认值。表达形式如下。

```
TRY(expression)
```

示例：

```
SELECT COALESCE(TRY(total_cost / packages), 0) AS per_package FROM shipping;
```

COALESCE和TRY搭配使用，当packages=0抛出异常时，返回默认值（0）。

```
per_package
-------------
          4
         14
          0
         19
(4 rows)
```

12.5.4 转换函数

Presto提供给了如下几个显式类型转换函数：

- CAST

  显式的进行类型转换，只是在出现转换错误的时候抛出异常。使用方法如下：

  ```
  CAST(value AS type) -> value1:type
  ```

- TRY_CAST

  与CAST功能相同，只是在出现转换错误的时候，返回NULL。使用方法如下：

  ```
  TRY_CAST(value AS TYPE) -> value1:TYPE | NULL
  ```

- TYPEOF

  获取参数或表达式值的类型字符串，使用方法如下：

  ```
  TYPEOF(expression) -> type:VARCHAR
  ```

  示例：

  ```
  SELECT TYPEOF(123); -- integer
  SELECT TYPEOF('cat'); -- varchar(3)
  SELECT TYPEOF(cos(2) + 1.5); -- double
  ```

12.5.5 数学函数与运算符

本文为您介绍Presto 支持的数学操作符及其数学函数。

数学操作符

Presto 支持的数学操作符如下表所示。

操作符	说明
+	加
-	减
*	乘
/	除（整数相除会截断）
%	取余

数学函数

Presto 提供了十分丰富的数学函数，如下表所示。

函数	语法	说明
abs	abs(x) →	x
cbrt	cbrt(x) → double	返回立方根

函数	语法	说明
ceil	ceil(x)	返回比 x 大的最小整数，是ceiling的别名
ceiling	ceiling(x)	返回比 x 大的最小整数
cosine_similarity	cosine_similarity(x, y) → double	返回两个稀疏向量的余弦相似度
degrees	degress(x) -> double	弧度换算成角度
e	e()->double	获取欧拉常数
exp	exp(x)->double	指数函数
floor	floor(x)	获取小于 x 的最大整数
from_base	from_base(string, radix) → bigint	获取字面量的值，该值的基数为 radix
inverse_normal_cdf	inverse_normal_cdf(mean,sd,p)->double	计算给定正态分布（均值、标准差和累计概率）的累计分布函数的倒数
ln	ln(x)->double	返回自然对数值
log2	log2(x)->double	返回以 2 为底的对数
log10	log10(x)->double	返回以 10 为底的对数
log	log(x,b) -> double	返回以 b 为底数的对数
mod	mod(n,m)	返回 n/m 的余数
pi	pi()->double	返回常数 Pi
pow	pow(x,p)->double	计算 x^p 的值，power的别名
power	power(x,p)->double	计算 x^p 的值
radians	radians(x)->double	将角度换算成弧度
rand	rand()->double	获取一个为随机数，返回值范围为[0.0,1.0)。random的别名
random	random()->double	获取一个为随机数，返回值范围为[0.0,1.0)
random	random(n)	获取一个为随机数，返回值范围为[0.0,n)
round	round(x)	返回与 x 最接近的整数
round	round(x, d)	返回与 x 最接近的数，精确到小数后 d 位

函数	语法	说明
sign	sign(x)	符号函数，x 如果是整数，则当x=0，返回0; x>0，返回1；x<0，返回-1。x 如果是浮点数，则当 x 为 NaN 时，返回Nan；当 x 为 +∞ 时，返回1；当x为-∞时，返回-1
sqrt	sqrt(x)->double	平方根函数
to_base	to_base(x, radix)->varchar	返回 x 以 radix 为基数的字面量
truncate	truncate(x) → double	截取整数部分
width_bucket	width_bucket(x, bound1, bound2, n) → bigint	获取 x 在[bound1, bound2]范围内 n 等的分直方图中的 bin 数
width_bucket	width_bucket(x, bins)	获取 x 在给定分布的直方图中的 bin 数
acos	acos(x)->double	获取 x 的反余弦值，x 为弧度
asin	asin(x)->double	获取 x 的反正弦值，x 为弧度
atan	atan(x)->double	获取 x 的反正切值，x 为弧度
atan2	atan2(y,x)->double	获取 y/x 的反正切值，x 为弧度
cos	cos(x)->double	获取 x 的余弦值，x为弧度
cosh	cosh(x)->double	获取 x 的双曲余弦值，x 为弧度
sin	sin(x)->double	获取 x 的正弦值，x为弧度
tan	tan(x)->double	获取 x 的正切值，x为弧度
tanh	tanh(x)->double	获取 x 的双曲正切值，x 为弧度
infinity	infinity() → double	获取正无穷常数
is_finite	is_finite(x) → boolean	判断 x 是否为有限数值
is_infinite	is_infinite(x) → boolean	判断 x 是否为无穷数值
is_nan	is_nan(x) → boolean	判断 x 是否不是一个数值类型的变量
nan	nan()	获取一个表示NAN（not-a-number）的常数

12.5.6 位运算函数

Presto 提供了如下几种位运算函数：

函数	语法	说明
bit_count	bit_count(x, bits) → bigint	返回 x 的补码中置 1 的位数
bitwise_and	bitwise_and(x, y) → bigint	位与函数
bitwise_not	bitwise_not(x) → bigint	取非操作
bitwise_or	bitwise_or(x, y) → bigint	位或函数
bitwise_xor	bitwise_xor(x, y) → bigint	抑或函数
bitwise_and_agg	bitwise_and_agg(x) → bigint	返回 x 中所有值的与操作结果，x 为数组
bitwise_or_agg	bitwise_or_agg(x) → bigint	返回 x 中所有值的或操作结果，x 位数组

示例：

```
SELECT bit_count(9, 64); -- 2
SELECT bit_count(9, 8); -- 2
SELECT bit_count(-7, 64); -- 62
SELECT bit_count(-7, 8); -- 6
```

12.5.7 Decimal 函数

字面量

使用如下形式来表示一个类型为DECIMAL的值的字面量：

```
DECIMAL 'xxxx.yyyyy'
```

DECIMAL 字面量的precision和其字面数字个数相同（包括前导的0），而scale和其小数位相同（包括后置的 0），示例如下：

字面量	数据类型
DECIMAL ‘0’	DECIMAL(1)
DECIMAL ‘12345’	DECIMAL(5)
DECIMAL ‘0000012345.1234500000’	DECIMAL(20, 10)

运算符

- 算术运算符

  假设有如下两个DECIMAL类型的变量 x， y：

  - x : DECIMAL(xp,xs)
  - y : DECIMAL(yp,ps)

  两个变量在参与算术运算时，遵循如下规则：

  - x + y或x - y
    - precision = min(38, 1 + min(xs, ys) + min(xp-xs, yp-ys))
    - scale = max(xs, ys)
  - x * y
    - precision = min(38, xp + yp)
    - scale = xs + ys
  - x / y
    - precision = min(38, xp + ys + max(0, ys-xs))
    - scale = max(xs, ys)
  - x % y
    - precision = min(xp - xs, yp - ys) + max(xs, bs)
    - scale = max(xs, ys)
- 比较运算符

  DECIMAL可以使用标准比较运算符和BETWEEN进行比较运算。
- 一元运算符

  DECIMAL可以使用一元运算符-取负数。

12.5.8 字符函数

拼接运算符

使用||运算符实现字符串的拼接。

字符函数

下表列出了 Presto 支持的字符函数：

函数名	语法	说明
chr	chr(n) → varchar	返回 UCP（Unicode code point）n作为一个字符返回。
codepoint	codepoint(string) → integer	返回字符string的 UCP 值。
concat	concat(string1, ..., stringN) → varchar	连接字符串，功能同||运算符。
hamming_distance	hamming_distance(string1, string2) → bigint	返回两个字符串对应位置的不同字符的个数。 **说明：** 两个字符串的长度应该相等。
length	length(string) → bigint	返回字符串的长度。
levenshtein_distance	levenshtein_distance(string1, string2) → bigint	返回两个字符串之间的编辑距离。
lower	lower(string) → varchar	转成小写。
upper	upper(string) → varchar	转成大写。
replace	replace(string, search) → varchar	用空字符来替换字符串string中和search相同的子串。
replace	replace(string, search, replace) → varchar	用replace来替换字符串string中和search相同的子串。
reverse	reverse(string) → varchar	反转字符串。
lpad	lpad(string, size, padstring) → varchar	用padstring从左边开始填充字符串string，填充成长度为size的字符串。padstring不能为空，size不能为0。
rpad	rpad(string, size, padstring) → varchar	用padstring从右边开始填充字符串string，填充成长度为size的字符串。padstring不能为空，size不能为0。
ltrim	ltrim(string) → varchar	删除前置空白符。
rtrim	rtrim(string) → varchar	删除后置空白符。
split	split(string, delimiter) → array	拆分字符串。
split	split(string, delimiter, limit) → array	拆分字符串，对数组大小有限制。

函数名	语法	说明
split_part	split_part(string, delimiter, index) → varchar	从index处开始拆分字符串，index 从 1 开始。
split_to_map	split_to_map(string, entryDelimiter, keyValueDelimiter) → map<varchar, varchar>	将字符串拆成一个map。
strpos	strpos(string, substring) → bigint	返回第一个符合子串的位置序号，序号从1开始，没找到则返回 0。
position	position(substring IN string) → bigint	返回子串在给定字符串中的起始位置。
substr	substr(string, start, [length]) → varchar	截取从位置start位置开始的子串，位置序号从1开始。长度参数可选。

Unicode 相关的函数

- normalize(string) → varchar

  用NFC规范化形式转换字符串。

- normalize(string, form) → varchar

  按给定的格式归一化字符串，form可选如下：

  - NFD：Canonical Decomposition
  - NFC：Canonical Decomposition, followed by Canonical Composition
  - NFKD：Compatibility Decomposition
  - NFKC：Compatibility Decomposition, followed by Canonical Composition

- to_utf8(string) → varbinary

  转换成 UTF-8 格式的字符串。

- from_utf8(binary, [replace]) → varchar

  将二进制数据解析成 UTF-8 字符串。非法序列会用replace替代，该项参数可选，默认为 Unicode 字符U+FFFD。需要注意的是，replace必须是单个字符，可以为空字符。

12.5.9 正则表达式

Presto使用Java Pattern的正则表达式语法。但也有几个例外，如下所示：

- 多行模式
  - 使用?m开启多行模式。
  - 行终止符为\n。
  - 不支持?d选项。
- 大小写敏感模式
  - 使用?i开启。
  - 不支持?u选项。
  - 不支持上下文敏感匹配。
  - 不支持局部敏感匹配。
- 不支持 Surrogate pairs

  如 Unicode U+10000必须使用\x{10000}来表示，而不能用\uD800\uDC00来表示。
- 如果模式字符串中不包含基本字符，并且没有间隔，那么，使用边界字符\b会出错。
- 不支持在字符类（如[A-Z123]）中使用\Q和\E。
- 在Unicode字符类\p{prop}的支持上存在如下不同：
  - 不支持下划线，如使用OldItalic代替Old_Italic。
  - 不支持使用Is，script=，sc=来指定脚本，取而代之的是直接使用脚本名。如\p{Hiragana}，而不是\p{script=Hiragana}。
  - 不支持使用block=，blk=来表示区块，只能使用In。如\p{InMongolia}。
  - 不支持使用Is， general_category=，gc=来指定分类，直接使用类别名称。如\p{L}。
  - 直接使用二进制属性，如使用\p{NoncharacterCodePoint}而不是\p{IsNoncharacterCodePoint}。

下面介绍 Presto 提供的正则表达式函数：

- regexp_extract_all(string, pattern, [group]) → array<varchar>

  提取字符串string中所有与模式pattern匹配的子串。pattern中如果使用了分组的功能，则可以通过设置group参数，用于说明匹配哪个捕获组。

  示例

```
SELECT regexp_extract_all('1a 2b 14m', '\d+'); -- [1, 2, 14]
SELECT regexp_extract_all('1a 2b 14m', '(\d+)([a-z]+)', 2); -- ['a', 'b', 'm']
```

```
```

- regexp_extract(string, pattern, [group]) → varchar

  功能和用法与regexp_extract_all类似，只是本函数只提取第一个匹配的结果。

  示例

  ```
  SELECT regexp_extract('1a 2b 14m', '\d+'); -- 1
  SELECT regexp_extract('1a 2b 14m', '(\d+)([a-z]+)', 2); -- 'a'
  ```

- regexp_like(string, pattern) → boolean

  判断字符串string中是否包含符合pattern模式的子串，包含返回TRUE，否则返回FALSE。本函数的功能与 SQL 中的LIKE语句功能相似，不同的是LIKE需要匹配整个模式字串，而本函数只需要字串中包含与模式字串相匹配子串即返回TRUE。

  示例

  ```
  SELECT regexp_like('1a 2b 14m', '\d+b'); -- true
  ```

- regexp_replace(string, pattern, [replacement]) → varchar

  用replacement替换字符串string中所有符合模式pattern的子串。replacement可选，不设置将使用空字符''替换（即删除匹配的子串）。

  可以通过在replacement字串中使用$g（g为捕获组的序号，从1开始）或${组名称}来设置捕获组。美元符号$在replacement字串中需要使用\$进行转义。

  示例

  ```
  SELECT regexp_replace('1a 2b 14m', '\d+[ab] '); -- '14m'
  SELECT regexp_replace('1a 2b 14m', '(\d+)([ab]) ', '3c$2 '); -- '3ca 3cb 14m'
  ```

- regexp_split(string, pattern) → array<varchar>

  使用模式字串pattern拆分字符串，保留尾部的空字符串。

  示例

  ```
  SELECT regexp_split('1a 2b 14m', '\s*[a-z]+\s*'); -- ['1', '2', '14', ''] 4个元素
                                                    -- 最后一个为空字符
  ```

12.5.10 二进制函数

拼接运算符

使用||运算符实现二进制串的拼接。

二进制函数

函数	语法	说明
length	length(binary) → bigint	返回二进制块的字节长度
concat	concat(binary1, ..., binaryN) → varbinary	将多个二进制块拼接在一起
to_base64	to_base64(binary) → varchar	获取二进制块的base64 编码
from_base64	from_base64(string) → varbinary	base64 解码
to_base64url	to_base64url(binary) → varchar	使用 URL 安全字符进行 base64 编码
from_base64url	from_base64url(string) → varbinary	使用 URL 安全字符进行 base64 解码
to_hex	to_hex(binary) → varchar	将二进制块编码为 16 进制字符串
from_hex	from_hex(string) → varbinary	将 16 进制编码的字符串解码成二进制块
to_big_endian_64	to_big_endian_64(bigint) → varbinary	将 bigint 编码为64位大端补码格式
from_big_endian_64	from_big_endian_64(binary) → bigint	64 位大端补码格式的二进制解码位 bigint 类型的数字
to_ieee754_32	to_ieee754_32(real) → varbinary	根据IEEE 754算法，将单精度浮点数编码为一个 32 位大端字节序的二进制块
to_ieee754_64	to_ieee754_64(double) → varbinary	根据IEEE 754算法，将双精度浮点数编码为一个 64 位大端字节序的二进制块
crc32	crc32(binary) → bigint	计算二进制块的CRC 32 值
md5	md5(binary) → varbinary	计算二进制块的MD 5 哈希值
sha1	sha1(binary) → varbinary	计算二进制块的SHA 1 哈希值
sha256	sha256(binary) → varbinary	计算二进制块的SHA 256 哈希值
sha512	sha512(binary) → varbinary	计算二进制块的SHA 512 哈希值
xxhash64	xxhash64(binary) → varbinary	计算二进制块的XXHASH 64 值

12.5.11 日期时间处理函数

本文介绍日期时间函数的语法规则，包括参数解释和函数示例等。

日期和时间操作符

Presto支持两种日期时间操作符+和-。

具体示例如下。

运算符	示例	结果	
+	date '2012-08-08' + interval '2' day	2012-08-10	
+	time '01:00' + interval '3' hour	04:00:00.000	
+	timestamp '2012-08-08 01:00' + interval '29' hour	2012-08-09 06:00:00.000	
+	timestamp '2012-10-31 01:00' + interval '1' month	2012-11-30 01:00:00.000	
+	interval '2' day + interval '3' hour	2 03:00:00.000	
+	interval '3' year + interval '5' month	3-5	
-	date '2012-08-08' - interval '2' day	2012-08-06	
-	time '01:00' - interval '3' hour	22:00:00.000	
-	timestamp '2012-08-08 01:00' - interval '29' hour	2012-08-06 20:00:00.000	
-	timestamp '2012-10-31 01:00' - interval '1' month	2012-09-30 01:00:00.000	
-	interval '2' day - interval '3' hour	1 21:00:00.000	
-	interval '3' year - interval '5' month	2-7	

时区转换

使用AT TIME ZONE操作符，设置一个时间戳的时区。

AT TIME ZONE示例如下。

```
SELECT timestamp '2012-10-31 01:00 UTC';
2012-10-31 01:00:00.000 UTC
SELECT timestamp '2012-10-31 01:00 UTC' AT TIME ZONE 'America/Los_Angeles';
```

```
2012-10-30 18:00:00.000 America/Los_Angeles
```

时间和日期函数

- 基本函数

函数	语法	说明
current_date	current_date -> date	返回查询开始时的当前日期。
current_time	current_time -> time with time zone	返回查询开始时的当前时间。
current_timestamp	current_timestamp -> timestamp with time zone	返回查询开始时的当前时间戳。
current_timezone	current_timezone() -> varchar	以IANA（例如，America或Los_Angeles）定义的格式返回当前时区，或以UTC的固定偏移量（例如+08：35）返回当前时区。
date	date(x) -> date	将日期字面量转换成日期类型的变量。
from_iso8601_timestamp	from_iso8601_timestamp(string) -> timestamp with time zone	将ISO 8601格式化的字符串解析为具有时区的时间戳。
from_iso8601_date	from_iso8601_date(string) -> date	将ISO 8601格式的字符串解析为日期。
from_unixtime	from_unixtime(unixtime) → timestamp	返回unixtime时间戳。
	from_unixtime(unixtime, string) → timestamp with time zone	返回指定时区的unixtime时间戳。
	from_unixtime(unixtime , hours, minutes) → timestamp with time zone	返回为hours和minutes对应时区的unixtime时间戳
localtime	localtime -> time	返回查询开始时的当前时间。
localtimestamp	localtimestamp -> timestamp	返回查询开始时的当前时间戳。
now	now() -> timestamp with time zone	这是current_timestam的另一种表达。

函数	语法	说明
to_iso8601	to_iso8601(x) -> varchar	将x格式化为ISO 8601 字符串。x可以是date、timestamp或带时区的timestamp。
to_milliseconds	to_milliseconds(interval) -> bigint	获取当前距当天零时已经过去的毫秒数。
to_unixtime	to_unixtime(timestamp) -> double	将时间戳转换成UNIX时间。

- 截取函数

  截取函数将时间日期变量按给定的单位进行截取，返回该单位的时间日期值。使用方法如下。

单位	示例
second	2001-08-22 03:04:05.000
minute	2001-08-22 03:04:00.000
hour	2001-08-22 03:00:00.000
day	2001-08-22 00:00:00.000
week	2001-08-20 00:00:00.000
month	2001-08-01 00:00:00.000
quarter	2001-07-01 00:00:00.000
year	2001-01-01 00:00:00.000

- 间隔函数

  本文介绍如下两个间隔函数：

  - date_add(unit, value, timestamp) → [same as input]

    在timestamp的基础上加上value个unit。如果想要执行相减的操作，可以通过将value赋值为负数来完成。

  - date_diff(unit, timestamp1, timestamp2) → bigint

    返回timestamp2 - timestamp1之后的值，该值的表示单位是unit。

  间隔函数支持如下所示单位。

单位	描述
millisecond	Milliseconds
second	Seconds

单位	描述
minute	Minutes
hour	Hours
day	Days
week	Weeks
month	Months
quarter	Quarters of a year
year	Years

- 抽取函数

  抽取函数支持的数据类型取决于需要抽取的域。大多数域都支持日期和时间类型。

  extract(field FROM x) → bigint

  其中，x为要提取的日期时间变量，field为要提取的域。

域	描述
YEAR	year()
QUARTER	quarter()
MONTH	month()
WEEK	week()
DAY	day()
DAY_OF_MONTH	day()
DAY_OF_WEEK	day_of_week()
DOW	day_of_week()
DAY_OF_YEAR	day_of_year()
DOY	day_of_year()
YEAR_OF_WEEK	year_of_week()
YOW	year_of_week()
HOUR	hour()
MINUTE	minute()
SECOND	second()
TIMEZONE_HOUR	timezone_hour()
TIMEZONE_MINUTE	timezone_minute()

- 便利的抽取函数

语法	说明
day(x) -> bigint	返回指定日期在当月的天数。
day_of_month(x) -> bigint	day()的另一种表达。
day_of_week(x) -> bigint	返回指定日期对应的星期值，取值范围1~7。
day_of_year(x) -> bigint	返回指定日期对应一年中的第几天，取值范围1~366。
dow(x) -> bigint	day_of_week()的另一种表达。
doy(x) -> bigint	day_of_year()的另一种表达。
hour(x) -> bigint	返回指定日期对应的小时，取值范围为0~23。
minute(x) -> bigint	返回指定日期对应的分钟数。
month(x) -> bigint	返回指定日期对应的月份。
quarter(x) -> bigint	返回指定日期对应的季度。
second(x) -> bigint	返回指定日期对应的秒数。
timezone_hour(timestamp) -> bigint	返回从指定时间戳对应时区偏移的小时数。
timezone_minute(timestamp) -> bigint	返回从指定时间戳对应时区偏移的分钟数。
week(x) -> bigint	返回指定日期对应一年中的ISO week，取值范围1~53。
week_of_year(x) -> bigint	week()的另一种表达。
year(x) -> bigint	返回指定日期对应的年份。
year_of_week(x) -> bigint	返回指定日期对应的ISO week的年份。
yow(x) -> bigint	year_of_week()的另一种表达。

- MySQL日期函数

  Presto提供了两个日期解析相关的函数，用于兼容MySQL的日期函数date_parse和str_to_date，它们分别是：

  - date_format(timestamp, format) → varchar

    使用format格式化timestamp。

  - date_parse(string, format) → timestamp

    按format格式解析日期字面量。

  Presto支持的MySQL格式符号如下表所示。

符号	说明
%a	星期简写 （Sun~Sat）。
%b	月份简写 （Jan~Dec）。
%c	月份，数字（1~12）。
%d	月中天数，数字（1~31）。
%e	月中天数，数字 （1~31）。
%f	秒数 （打印6位数字：000000~999000；解析1~9位数字：0~999999999）。
%H	小时（00~23）。
%h	小时（01~12）。
%I	小时（01~12）。
%i	分钟（00~59）。
%j	一年中的第几天 （001~366）。
%k	小时（0~23）。
%l	小时（1~12）。
%M	月份名称 （January~December）。
%m	月份，数字 （01~12）。
%p	AM或PM。
%r	时间，12小时 （hh:mm:ss AM/PM）。
%S	秒（00~59）。
%s	秒 （00~59）。
%T	时间，24小时 （hh:mm:ss）。

符号	说明
%v	星期 （01~53），第一条为星期一，与%X配合使用。
%W	星期名称 （Sunday~Saturday）。
%x	年份，数字，4位，第一天为星期一。
%Y	年份，数字，4位。
%y	年份，数字，2位，表示年份范围为[1970, 2069]。
%%	表示字符'%'。

说明:

Presto目前不支持的符号有： %D、 %U、 %u、 %V、 %w、 %X。

- Java日期函数

  下列函数用于兼容Java的日期格式（JodaTime Pattern）：

  - format_datetime(timestamp, format) → varchar，格式化时间戳。
  - parse_datetime(string, format) → timestamp with time zone，解析时间戳字符串。

12.5.12 聚合函数

聚合函数具有如下特点：

- 输入一个数据集
- 输出一个单一的计算结果

绝大部分聚合函数都会在计算时忽略null值，并且在输入为空或均为null时，返回null。但也有例外，如下几个聚合函数：

- count
- count_if
- max_by
- min_by
- approx_distinct

基本聚合函数

函数	语法	说明
arbitrary	arbitrary(x) → [same as input]	随机返回 x 中的一个非 null 值。

函数	语法	说明
array_agg	array_agg(x) → array<[same as input]>	从输入的元素中创建数组。
avg	avg(x) → double	求算术平均值。
avg	avg(time interval type) → time interval type	计算输入时间序列的平均时间间隔。
bool_and	bool_and(boolean) → boolean	如果所有输入的值都为 TRUE，则返回 TRUE，否则返回 FALSE。
bool_or	bool_or(boolean) → boolean	如果输入的序列中有一个为 True，则返回 True，否则返回 False。
checksum	checksum(x) → varbinary	返回 x 的校验和（顺序不敏感）。
count	count(*) → bigint	返回行数。
count	count(x) → bigint	返回非 null 元素的个数。
count_if	count_if(x) → bigint	返回 x 中元素为True 的个数，等同于count(CASE WHEN x THEN 1 END)。
every	every(boolean) → boolean	同bool_and。
geometric_mean	geometric_mean(x) → double	返回 x 的几何平均值。
max_by	max_by(x, y) → [same as x]	返回与 y 的最大值相关的 x 值。
max_by	max_by(x, y, n) → array<[same as x]>	返回与 y 的前 n 个最大值相关的 x 值的数组。
min_by	min_by(x, y) → [same as x]	返回与 y 的最小值相关的 x 值。
min_by	min_by(x, y, n) → array<[same as x]>	返回与 y 的前 n 个最小值相关的 x 值的数组。
max	max(x) → [same as input]	返回最大值。
max	max(x, n) → array<[same as x]>	返回前 n 个最大值列表。
min	min(x) → [same as input]	返回最小值。
min	min(x, n) → array<[same as x]>	返回前 n 个最小值列表。

函数	语法	说明
sum	sum(x) → [same as input]	求和。

位聚合函数

位聚合函数参见位运算函数中介绍的bitwise_and_agg和bitwise_or_agg函数。

Map 聚合函数

函数	语法	说明
histogram	histogram(x) → map<K, bigint>	统计直方图。
map_agg	map_agg(key, value) → map<K,V>	创建一个MAP类型的变量。
map_union	map_union(x<K, V>) → map<K,V>	返回输入map列表的 Union 结果，如果有多个 map 对象包含相同的key，最终的结果中，对于 key 的 value 随机的从输入的 map 中选取。
multimap_agg	multimap_agg(key, value) → map<K,array>	创建一个多重映射的MAP变量。

近似聚合函数

函数	语法	说明
approx_distinct	approx_distinct(x, [e]) → bigint	返回输入列表中不重复的值的个数。本函数返回的是count(DISTINCT x)的近似值，如果所有值都是 null，则返回 0。e为期望标准差的上界，可选，默认为 2.3%，当前的实现方式对e的取值范围有约束，要求在[0.01150, 0.26000]之间。对于特定对输入，不保证误差上界。
approx_percentile	approx_percentile(x, percentage) → [same as x]	估计序列 x 中位于第百分之 percentage 位的数值。
approx_percentile	approx_percentile(x, percentages) → array<[same as x]>	类似上面，percentages 为数组，返回值与之一一对应。

函数	语法	说明
approx_percentile	approx_percentile(x, w, percentage) → [same as x]	类似上面，w为x的权值。
approx_percentile	approx_percentile(x, w, percentage, accuracy) → [same as x]	类似上面，accuracy为预估精度的上线，取值范围为[0, 1]。
approx_percentile	approx_percentile(x, w, percentages) → array<[same as x]>	类似上面，percentages为数组，返回值与之一一对应。
numeric_histogram	numeric_histogram(buckets, value, [weight]) → map<double, double>	按给定的桶数计算数值直方图。buckets必须是BIGINT类型，value和weight 必须是数值类型。权重列表weight可选，默认为1。

统计聚合函数

函数	语法	说明
corr	corr(y, x) → double	计算相关系数。
covar_pop	covar_pop(y, x) → double	计算总体协方差。
covar_samp	covar_samp(y, x) → double	计算样本协方差。
kurtosis	kurtosis(x) → double	计算超值峰度. 使用下列表达式进行无偏估计： `kurtosis(x) = n(n+1)/((n-1)(n-2)(n-3))sum[(x_i-mean)^4]/sttdev(x)^4-3(n-1)^2/((n-2)(n-3))`
regr_intercept	regr_intercept(y, x) → double	计算线性回归截距. y为相关变量. x为独立变量.
regr_slope	regr_slope(y, x) → double	计算线性回归斜率。y为相关变量. x为独立变量。
skewness	skewness(x) → double	计算偏度。
sttdev_pop	sttdev_pop(x) → double	计算总体标准差。
sttdev_samp	sttdev_samp(x) → double	计算样本标准差。
sttdev	sttdev(x) → double	计算标准差，同sttdev_samp。
var_pop	var_pop(x) → double	计算总体方差。

函数	语法	说明
var_samp	var_samp(x) → double	计算样本方差。
variance	variance(x) → double	同var_samp。

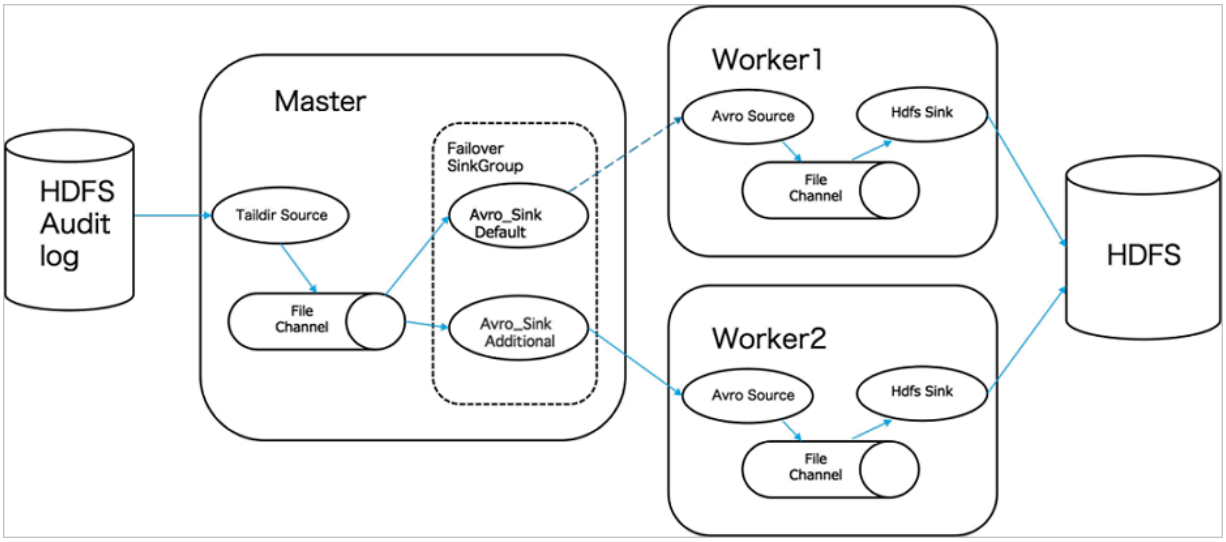
13 Flume

13.1 Flume 使用说明

本文以E-MapReduce-Flume实时同步HDFS audit日志至HDFS为例，介绍Flume的使用。

背景信息

E-MapReduce从3.19.0版本开始对EMR-Flume提供集群管理的功能。通过集群管理功能，可以在Web页面方便的配置和管理Flume Agent。示例中，在master实例启动Flume agent，收集本地磁盘中的audit日志通过Avro协议发送数据至core实例，在core实例配置并启动failover sink processor，接收master实例发送的数据并sink到HDFS中。Flume Agent拓扑结构如下图所示。



注意：

您可以根据实际情况设置Flume Agent的拓扑结构。

Flume其他使用场景的配置，请参见Flume配置说明。

准备工作

创建E-MapReduce Hadoop集群，在可选服务中选择Flume。详情请参见#unique_12。

操作步骤

1. Core实例配置并启动Flume Agent。

   例如在emr-worker-1节点进行操作。

   如下图所示。

   a. 登录阿里云 E-MapReduce 控制台。

   b. 单击上方的**集群管理**页签。

   c. 在**集群管理**页面，单击集群右侧的**详情**。

   d. 在左侧导航栏中，选择**集群服务** > **FLUME**。

   e. 单击**配置**页签，在配置页面设置如下 。

default-agent.sinks.default-sink.type	hdfs
default-agent.channels.default-channel.type	file
default-agent.sources.default-source.type	avro
deploy_node_hostname	emr-worker-1

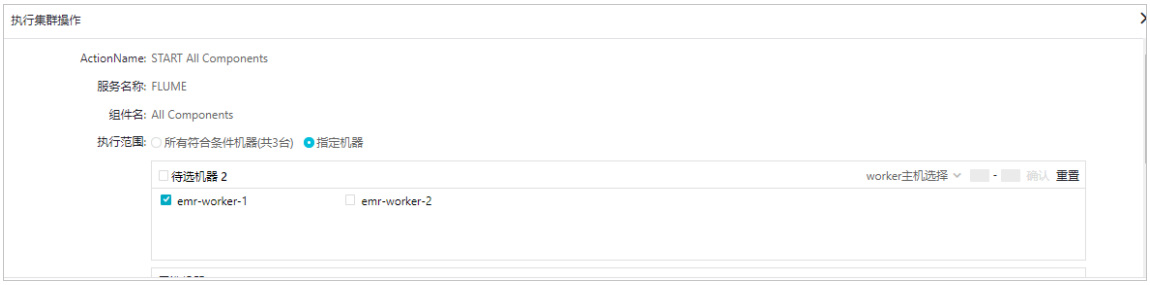
   f. 在配置页面通过自定义配置添加如下配置：

default-agent.sinks.default-sink.hdfs.path	对于高可用集群，使用 hdfs://emr-cluster/path 形式的地址
default-agent.sinks.default-sink.hdfs.fileType	DataStream

default-agent.sinks.default-sink.hdfs.rollSize	0
default-agent.sinks.default-sink.hdfs.rollCount	0
default-agent.sinks.default-sink.hdfs.rollInterval	86400
default-agent.sinks.default-sink.hdfs.batchSize	51200
default-agent.sources.default-source.bind	0.0.0.0
default-agent.sources.default-source.port	根据实际设置
default-agent.channels.default-channel.transactionCapacity	51200
default-agent.channels.default-channel.dataDirs	channel 存储 event 数据的路径
default-agent.channels.default-channel.checkpointDir	存储 checkpoint 的路径

default-agent.channels.default-channel.capacity	根据 hdfs roll 进行设置

g. 保存配置后启动Flume agent。



h. 单击**查看操作历史**，显示操作成功后，**部署拓扑**页面可以看到emr-worker-1节点的flume已经是started状态。

emr-worker-1节点启动成功后，开始启动第二个worker节点。 同样的方式，例如在worker-2节点启动flume，修改配置项。

deploy_node_hostname	节点的hostname
default-agent.sinks.default-sink.hdfs.path	对于高可用集群，使用 hdfs://emr-cluster/path 形式的地址

i. 保存配置后，启动All Components，指定机器为emr-worker-2。

2. Master实例配置并启动Flume Agent。

例如在emr-header-1节点进行操作。

配置agent如下：

additional_sinks	k1
deploy_node_hostname	emr-header-1
default-agent.sources.default-source.type	taildir
default-agent.sinks.default-sink.type	avro

default-agent.channels.default-channel.type	file

新增配置如下：

配置项	值
default-agent.sources.default-source.filegroups	f1
default-agent.sources.default-source.filegroups.f1	/mnt/disk1/log/hadoop-hdfs/hdfs-audit.log.*
default-agent.sources.default-source.positionFile	存储 position file 的路径
default-agent.channels.default-channel.checkpointDir	存储 checkpoint 的路径
default-agent.channels.default-channel.dataDirs	存储 event 数据的路径
default-agent.channels.default-channel.capacity	根据 hdfs roll 进行设置
default-agent.sources.default-source.batchSize	2000
default-agent.channels.default-channel.transactionCapacity	2000
default-agent.sources.default-source.ignoreRenameWhenMultiMatching	true
default-agent.sinkgroups	g1
default-agent.sinkgroups.g1.sinks	default-sink k1
default-agent.sinkgroups.g1.processor.type	failover
default-agent.sinkgroups.g1.processor.priority.default-sink	10
default-agent.sinkgroups.g1.processor.priority.k1	5
default-agent.sinks.default-sink.hostname	emr-worker-1 节点的IP
default-agent.sinks.default-sink.port	emr-worker-1 节点 Flume Agent 的 port
default-agent.sinks.k1.hostname	emr-worker-2 节点的 IP
default-agent.sinks.k1.port	emr-worker-2 节点 Flume Agent 的 port

配置项	值
default-agent.sinks.default-sink.batch-size	2000
default-agent.sinks.k1.batch-size	2000
default-agent.sinks.k1.type	avro
default-agent.sinks.k1.channel	default-channel

查看同步结果

使用HDFS命令，可以看到同步的数据被写入FlumeData.${timestamp}形式的文件中，其中timestamp为文件创建的时间戳。

```
[root@emr-header-1 ~]# hdfs dfs -ls /tmp/emr-worker-1/data
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/apps/ecm/service/hadoop/2.7.2-1.2.8/package/hadoop-2.7.2-1.2.8/share/hadoop/common/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/apps/ecm/service/tez/0.8.4/package/tez-0.8.4/lib/slf4j-log4j12-1.7.10.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]
Found 174 items
-rw-r--r--   3 root hadoop   34641327 2019-03-27 10:49 /tmp/emr-worker-1/data/FlumeData.1553654928352
-rw-r--r--   3 root hadoop   30963637 2019-03-27 10:49 /tmp/emr-worker-1/data/FlumeData.1553654958983
-rw-r--r--   3 root hadoop   29282243 2019-03-27 10:50 /tmp/emr-worker-1/data/FlumeData.1553654989023
-rw-r--r--   3 root hadoop   31138131 2019-03-27 10:50 /tmp/emr-worker-1/data/FlumeData.1553655019874
-rw-r--r--   3 root hadoop   29995885 2019-03-27 10:51 /tmp/emr-worker-1/data/FlumeData.1553655049922
-rw-r--r--   3 root hadoop   30238902 2019-03-27 10:51 /tmp/emr-worker-1/data/FlumeData.1553655079981
-rw-r--r--   3 root hadoop   31127038 2019-03-27 10:52 /tmp/emr-worker-1/data/FlumeData.1553655110987
-rw-r--r--   3 root hadoop   30121183 2019-03-27 10:52 /tmp/emr-worker-1/data/FlumeData.1553655141136
-rw-r--r--   3 root hadoop   30124697 2019-03-27 10:53 /tmp/emr-worker-1/data/FlumeData.1553655171175
-rw-r--r--   3 root hadoop   30624989 2019-03-27 10:53 /tmp/emr-worker-1/data/FlumeData.1553655202156
-rw-r--r--   3 root hadoop   30122940 2019-03-27 10:54 /tmp/emr-worker-1/data/FlumeData.1553655232325
-rw-r--r--   3 root hadoop   30122940 2019-03-27 10:54 /tmp/emr-worker-1/data/FlumeData.1553655262387
```

查看日志

Flume agent日志的存放路径为/mnt/disk1/log/flume/default-agent/flume.log。

查看监控信息

集群与服务管理页面提供了Flume agent的监控信息。通过在集群与服务管理页面单击**Flume**服务进行访问。

13.2 Flume配置说明

E-MapReduce（以下简称EMR） 从EMR-3.16.0版本开始支持Apache Flume。本文介绍如何使用Flume将数据从EMR Kafka集群同步至EMR Hadoop集群的HDFS、Hive、HBase以及阿里云OSS。

准备工作

- 已登录阿里云 E-MapReduce 控制台。
- 创建Hadoop集群时，在**可选服务**中选择Flume。

> **说明：**
> Flume软件安装目录在/usr/lib/flume-current下，其他常用文件路径获取方式请参见#unique_92。

- 创建Kafka集群，并创建名称为flume-test的Topic，用于生成数据。

> **说明：**
> - 如果创建的是Hadoop高安全集群，消费标准Kafka集群的数据，在Hadoop集群配置Kerberos认证，详情请参见兼容MIT Kerberos认证。
> - 如果创建的是Kafka高安全集群，通过Flume将数据写入标准Hadoop集群，请参见 Kerberos Kafka Source。
> - 如果创建的Hadoop集群和Kafka集群都是高安全集群，需配置跨域互信，详情请参见跨域互信，其它配置请参见跨域互信使用Flume。

Kafka->HDFS

1. 配置Flume。

   创建配置文件flume.properties，添加如下配置：

   - **a1.sources.source1.kafka.bootstrap.servers**：Kafka集群Broker的Host和端口号。
   - **a1.sources.source1.kafka.topics**：Flume消费Kafka数据的Topic。
   - **a1.sinks.k1.hdfs.path**：Flume向HDFS写入数据的路径。

```
a1.sources = source1
a1.sinks = k1
a1.channels = c1

a1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
a1.sources.source1.channels = c1
a1.sources.source1.kafka.bootstrap.servers = kafka-host1:port1,kafka-host2:port2...
a1.sources.source1.kafka.topics = flume-test
a1.sources.source1.kafka.consumer.group.id = flume-test-group
```

```
# Describe the sink
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path = /tmp/flume/test-data
a1.sinks.k1.hdfs.fileType=DataStream

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 100
a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.source1.channels = c1
a1.sinks.k1.channel = c1
```

如果配置项**a1.sinks.k1.hdfs.path**使用URL的形式，针对不同集群代码示例如下：

- 高可用集群

  ```
  a1.sinks.k1.hdfs.path = hdfs://emr-cluster/tmp/flume/test-data
  ```

- 标准集群

  ```
  a1.sinks.k1.hdfs.path = hdfs://emr-header-1:9000/tmp/flume/test-data
  ```

2. 启动服务。

   Flume默认配置文件存储在/etc/ecm/flume-conf下，使用该配置启动Flume Agent。

   ```
   flume-ng agent --name a1 --conf /etc/ecm/flume-conf  --conf-file flume.properties
   ```

   启动Agent时，因为使用了/etc/ecm/flume-conf下的log4j.properties，所以会在当前路径下生成日志logs/flume.log，您可根据实际情况对log4j.properties进行配置。

3. 测试数据写入情况。

   在Kafka集群使用kafka-console-producer.sh输入测试数据abc。

   ```
   [root@emr-header-1 ~]# kafka-console-producer.sh --topic flume-test --broker-list emr-header-1:9092
   >abc
   >
   ```

   Flume会在HDFS中以当前时间的（毫秒）时间戳生成文件FlumeData.xxxx。文件内容是在Kafka中输入的数据。

   ```
   [root@emr-header-1 ~]# hdfs dfs -cat /tmp/flume/test-data/FlumeData.1543386053173
   abc
   [root@emr-header-1 ~]#
   ```

Kafka->Hive

1. 创建Hive表。

   Flume使用事务操作将数据写入Hive，需要在创建Hive表时设置**transactional**属性，例如创建flume_test表。

   ```
   create table flume_test (id int, content string)
   clustered by (id) into 2 buckets
   stored as orc  TBLPROPERTIES('transactional'='true');
   ```

2. 配置Flume。

   创建配置文件flume.properties，添加如下配置：

   - **a1.sources.source1.kafka.bootstrap.servers**：Kafka集群Broker的Host和端口号。
   - **a1.sinks.k1.hive.metastore**：Hive metastore 的URI，配置为hive-site.xml中配置项 **hive.metastore.uris**的值。

   ```
   a1.sources = source1
   a1.sinks = k1
   a1.channels = c1

   a1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
   a1.sources.source1.channels = c1
   a1.sources.source1.kafka.bootstrap.servers = kafka-host1:port1,kafka-host2:port2...
   a1.sources.source1.kafka.topics = flume-test
   a1.sources.source1.kafka.consumer.group.id = flume-test-group

   # Describe the sink
   a1.sinks.k1.type = hive
   a1.sinks.k1.hive.metastore = thrift://xxxx:9083
   a1.sinks.k1.hive.database = default
   a1.sinks.k1.hive.table = flume_test
   a1.sinks.k1.serializer = DELIMITED
   a1.sinks.k1.serializer.delimiter = ","
   a1.sinks.k1.serializer.serdeSeparator = ','
   a1.sinks.k1.serializer.fieldnames =id,content

   a1.channels.c1.type = memory
   a1.channels.c1.capacity = 100
   a1.channels.c1.transactionCapacity = 100

   a1.sources.source1.channels = c1
   a1.sinks.k1.channel = c1
   ```

3. 启动Flume。

   ```
   flume-ng agent --name a1 --conf /etc/ecm/flume-conf  --conf-file flume.properties
   ```

4. 生成数据。

   在Kafka集群中使用kafka-console-producer.sh，以逗号（,）为分隔符输入测试数据`1,a`。

5. 检测数据写入情况。

   查询Hive事务表并在客户端进行配置。

   ```
   hive.support.concurrency - true
   hive.exec.dynamic.partition.mode - nonstrict
   hive.txn.manager - org.apache.hadoop.hive.ql.lockmgr.DbTxnManager
   ```

   配置好后查询flume_test表中的数据。

   ```
   hive> set hive.support.concurrency=true;
   hive> set hive.exec.dynamic.partition.mode=nonstrict;
   hive> set hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;
   hive> select * from flume_test;
   OK
   1       a
   Time taken: 0.149 seconds, Fetched: 1 row(s)
   ```

Kafka->HBase

1. 创建HBase表。

   创建HBase表flume_test及列簇column。

   ```
   hbase(main):001:0> create 'flume_test', 'column'
   0 row(s) in 1.3940 seconds

   => Hbase::Table - flume_test
   ```

2. 配置Flume。

   创建配置文件flume.properties，添加如下配置：

   - **a1.sources.source1.kafka.bootstrap.servers**：Kafka集群Broker的Host和端口号。
   - **a1.sinks.k1.table**：HBase表名。
   - **a1.sinks.k1.columnFamily**：列簇名。

   ```
   a1.sources = source1
   a1.sinks = k1
   a1.channels = c1

   a1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
   a1.sources.source1.channels = c1
   a1.sources.source1.kafka.bootstrap.servers = kafka-host1:port1,kafka-host2:port2...
   a1.sources.source1.kafka.topics = flume-test
   a1.sources.source1.kafka.consumer.group.id = flume-test-group

   a1.sinks.k1.type = hbase
   a1.sinks.k1.table = flume_test
   a1.sinks.k1.columnFamily = column


   # Use a channel which buffers events in memory
   a1.channels.c1.type = memory
   a1.channels.c1.capacity = 1000
   a1.channels.c1.transactionCapacity = 100
   ```

```
# Bind the source and sink to the channel
a1.sources.source1.channels = c1
a1.sinks.k1.channel = c1
```

3. 启动服务。

```
flume-ng agent --name a1 --conf /etc/ecm/flume-conf  --conf-file flume.properties
```

4. 测试数据写入情况。

在Kafka集群使用kafka-console-producer.sh生成数据后，在HBase查到数据。

```
=> ["flume_test"]
hbase(main):003:0> scan 'flume_test'
ROW                    COLUMN+CELL
 defaultf2add0ee-5040-4 column=column:pCol, timestamp=1543493834351, value=data
 7dc-b002-f269b679977b
 incRow                column=column:iCol, timestamp=1543493834373, value=\x00\x00\x00\
                       x00\x00\x00\x00\x01
2 row(s) in 0.0310 seconds
```

Kafka->OSS

1. 创建OSS路径。

创建OSS Bucket及目录，例如oss://flume-test/result。

2. 配置Flume。

Flume向OSS写入数据时，因为需要占用较大的JVM内存，所以可以改小OSS缓存或者增大Flume Agent的Xmx。

- 修改OSS缓存大小。

  将hdfs-site.xml配置文件从/etc/ecm/hadoop-conf拷贝至/etc/ecm/flume-conf，改小配置项**smartdata.cache.buffer.size**的值，例如修改为1048576。

- 修改Xmx。

  在Flume的配置路径/etc/ecm/flume-conf下，复制配置文件flume-env.sh.template并重命名为flume-env.sh，设置Xmx的值，例如设置为**1g**。

  ```
  export JAVA_OPTS="-Xmx1g"
  ```

创建配置文件flume.properties，添加如下配置：

- **a1.sources.source1.kafka.bootstrap.servers**：Kafka集群Broker的Host和端口号。
- **a1.sinks.k1.hdfs.path**：OSS路径。

```
a1.sources = source1
a1.sinks = k1
a1.channels = c1

a1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
a1.sources.source1.channels = c1
```

```
a1.sources.source1.kafka.bootstrap.servers = kafka-host1:port1,kafka-host2:port2...
a1.sources.source1.kafka.topics = flume-test
a1.sources.source1.kafka.consumer.group.id = flume-test-group

a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path = oss://flume-test/result
a1.sinks.k1.hdfs.fileType=DataStream

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 100
a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.source1.channels = c1
a1.sinks.k1.channel = c1
```

3. 启动Flume。

   如果配置Flume时修改了OSS缓存大小，需要使用--classpath参数传入OSS相关依赖和配置。

   ```
   flume-ng agent --name a1 --conf /etc/ecm/flume-conf  --conf-file flume.properties
    --classpath "/opt/apps/extra-jars/*:/etc/ecm/flume-conf/hdfs-site.xml"
   ```

   如果修改了Flume Agent的Xmx，只需要传入OSS相关依赖。

   ```
   flume-ng agent --name a1 --conf /etc/ecm/flume-conf  --conf-file flume.properties
    --classpath "/opt/apps/extra-jars/*"
   ```

4. 测试数据写入情况。

   在Kafka集群使用kafka-console-producer.sh生成数据后，在OSS的oss://flume-test/result路径下会以当前时间的（毫秒）时间戳为后缀生成文件FlumeData.xxxx。

Kerberos Kafka source

消费高安全Kafka集群的数据时，需要完成额外的配置：

- 在Kafka集群配置Kerberos认证，将生成的keytab文件test.keytab拷贝至Hadoop集群的/etc/ecm/flume-conf路径下，详情请参见兼容MIT Kerberos认证；将Kafka集群的 /etc/ecm/has-conf/krb5.conf文件拷贝至Hadoop集群的/etc/ecm/flume-conf路径下。
- 配置flume.properties。

  在flume.properties中添加如下配置。

  ```
  a1.sources.source1.kafka.consumer.security.protocol = SASL_PLAINTEXT
  a1.sources.source1.kafka.consumer.sasl.mechanism = GSSAPI
  a1.sources.source1.kafka.consumer.sasl.kerberos.service.name = kafka
  ```

- 配置Kafka client。
  - 在/etc/ecm/flume-conf下创建文件flume_jaas.conf，内容如下。

    ```
    KafkaClient {
     com.sun.security.auth.module.Krb5LoginModule required
    ```

```
  useKeyTab=true
  storeKey=true
  keyTab="/etc/ecm/flume-conf/test.keytab"
  serviceName="kafka"
  principal="test@EMR.${realm}.COM";
};
```

其中，**${realm}** 替换为Kafka集群的Kerberos realm。获取方式如下。

在Kafka集群执行命令**hostname**，得到形式为emr-header-1.cluster-xxx的主机名，例如emr-header-1.cluster-123456，最后的数字串123456即为realm。

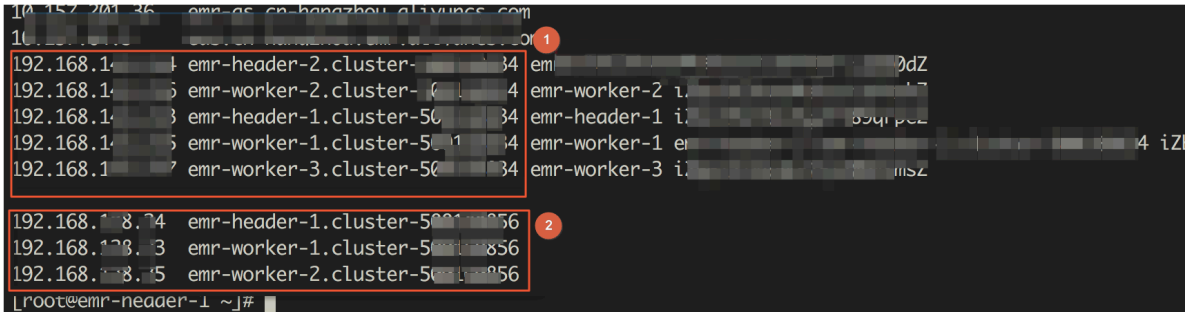
- 修改/etc/ecm/flume-conf/flume-env.sh。

  初始情况下，/etc/ecm/flume-conf/下没有flume-env.sh 文件，需要拷贝flume-env.sh.template并重命名为flume-env.sh。添加如下内容。

```
export JAVA_OPTS="$JAVA_OPTS -Djava.security.krb5.conf=/etc/ecm/flume-conf/krb5.conf"
export JAVA_OPTS="$JAVA_OPTS -Djava.security.auth.login.config=/etc/ecm/flume-conf/flume_jaas.conf"
```

- 设置域名。

  将Kafka集群各节点的长域名和IP的绑定信息添加到Hadoop集群的/etc/hosts。长域名的形式例如emr-header-1.cluster-123456。



说明：

图中标注①表示的是本集群（即Hadoop集群）域名；图中标注②表示新增加的Kafka集群域名。

跨域互信使用Flume

在配置了跨域互信后，其他配置如下：

- 参见兼容MIT Kerberos认证在Kafka集群配置Kerberos认证，将生成的keytab文件test.keytab拷贝至Hadoop集群的/etc/ecm/flume-conf路径下。

- 配置flume.properties

  在flume.properties中添加如下配置。

  ```
  a1.sources.source1.kafka.consumer.security.protocol = SASL_PLAINTEXT
  a1.sources.source1.kafka.consumer.sasl.mechanism = GSSAPI
  a1.sources.source1.kafka.consumer.sasl.kerberos.service.name = kafka
  ```

- 配置Kafka client。

  - 在/etc/ecm/flume-conf下创建文件flume_jaas.conf，内容如下。

    ```
    KafkaClient {
      com.sun.security.auth.module.Krb5LoginModule required
      useKeyTab=true
      storeKey=true
      keyTab="/etc/ecm/flume-conf/test.keytab"
      serviceName="kafka"
      principal="test@EMR.${realm}.COM";
    };
    ```

    其中，**${realm}**替换为Kafka集群的Kerberos realm。获取方式如下。

    在Kafka集群执行命令**hostname**，得到形式为emr-header-1.cluster-xxx的主机名，如emr-header-1.cluster-123456，最后的数字串123456即为realm。

  - 修改/etc/ecm/flume-conf/flume-env.sh。

    初始情况下，/etc/ecm/flume-conf/下没有flume-env.sh文件，需要拷贝flume-env.sh.template并重命名为flume-env.sh。添加如下内容。

    ```
    export JAVA_OPTS="$JAVA_OPTS -Djava.security.auth.login.config=/etc/ecm/flume-conf/flume_jaas.conf""
    ```

13.3 使用 LogHub Source 将非 E-MapReduce 集群的数据同步至 E-MapReduce 集群的 HDFS

本文介绍使用 EMR-Flume 实时同步 Log Service 的数据至 EMR 集群的 HDFS，并根据数据记录的时间戳将数据存入 HDFS 相应的分区中。

背景信息

E-MapReduce 从 3.20.0 版本开始对 EMR-Flume 加入了 Log Service Source。借助 Log Service 的 Logtail 等工具，可以将需要同步的数据实时采集并上传到 LogHub，再使用 EMR-Flume 将 LogHub 的数据同步至 EMR 集群的 HDFS。 有关采集数据到 Log Service 的 LogHub 的详细方法及步骤参见#unique_95。

准备工作

创建 Hadoop 集群，在可选软件中选择 Flume，详细步骤请参见#unique_12。

配置 Flume

- 配置 Source

配置项	值	说明
type	org.apache.flume.source.loghub.LogHubSource	-
endpoint	LogHub 的 Endpoint	如果使用 VPC/经典网络的 endpoint，要保证与 EMR 集群在同一个地区；如果使用公网 endpoint，要保证运行 Flume agent 的节点有公网 IP。
project	LogHub 的 project	-
logstore	LogHub 的 logstore	-
accessKeyId	阿里云的 AccessKey ID	-
accessKey	阿里云的 AccessKey	-
useRecordTime	true	默认值为 false。如果 header 中没有 timestamp 属性，接收 event 的时间戳会被加入到 header 中； 但是在Flume Agent 启停或者同步滞后等情况下，会将数据放入错误的时间分区中。为避免这种情况，可以将该值设置为true，使用数据收集到 LogHub 的时间作为 timestamp。
consumerGroup	consumer_1	消费组名称，默认值为consumer_1。

其他配置项说明如下：

- **consumerPosition**

  消费组在第一次消费 LogHub 数据时的位置，默认值为 end，即从最近的数据开始消费；可以设置的其他值为 begin 或 special。begin 表示从最早的数据开始消费；special 表示从指定的时间点开始消费。在配置为 special 时，需要配置 startTime 为开始消费的时间点，单位为秒。 首次运行后 LogHub 服务端会记录消费组的消费点，此时如果想更改

consumerPosition，可以清除 LogHub 的消费组状态，参见#unique_96；或者更改配置consumerGroup 为新的消费组。

- **heartbeatInterval** 和 **fetchInOrder**

  heartbeatInterval 表示消费组与服务端维持心跳的间隔，单位是毫秒，默认为30000毫秒；fetchInOrder 表示相同 key 的数据是否按序消费，默认值为 false。

- batchSize 和 batchDurationMillis

  通用的 source batch 配置，表示触发 event 写入 channel 的阈值。

- backoffSleepIncrement 和 maxBackoffSleep

  通用的 source sleep 配置，表示 LogHub 没有数据时触发 sleep 的时间和增量。

- 配置 channel 和 sink

  此处使用 memory channel 和 HDFS sink。

  - HDFS Sink 配置如下：

配置项	值
hdfs.path	/tmp/flume-data/loghub/datetime=%y%m%d/hour=%H
hdfs.fileType	DataStream
hdfs.rollInterval	3600
hdfs.round	true
hdfs.roundValue	60
hdfs.roundUnit	minute
hdfs.rollSize	0
hdfs.rollCount	0

  - Memory channel 配置如下：

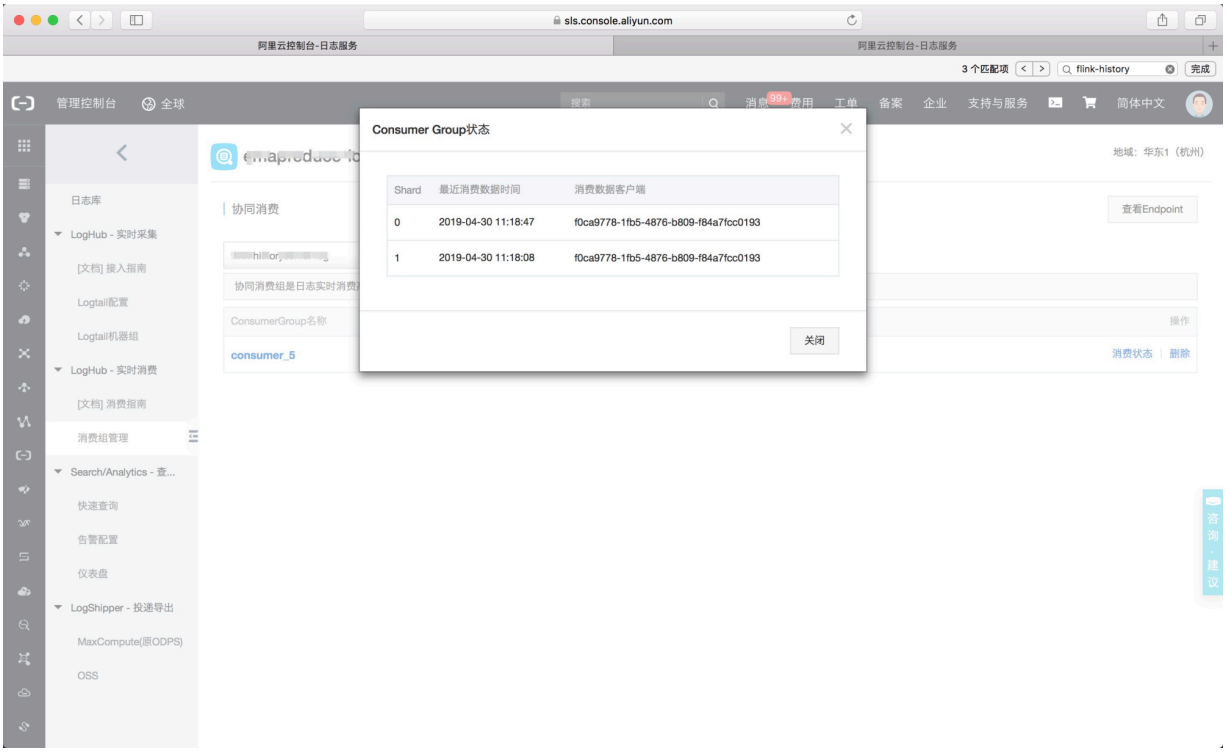
配置项	值
capacity	2000
transactionCapacity	2000

运行 Flume agent

在阿里云 E-Mapreduce 控制台页面启动 Flume agent 的具体操作参见 Flume 使用说明。成功启动后，可以看到配置的 HDFS 路径下按照 record timestamp 存储的日志数据。

```
[root@emr-worker-3 ~]# hdfs dfs -ls /tmp/flume-data/loghub/datetime=190430/hour=11
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/apps/ecm/service/hadoop/2.7.2-1.2.8/package/hadoop-2.7.2-1.2.8/share/hadoop/common/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/apps/ecm/service/tez/0.8.4/package/tez-0.8.4/lib/slf4j-log4j12-1.7.10.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]
Found 1 items
-rw-r--r--   3 root hadoop       1344 2019-04-30 11:18 /tmp/flume-data/loghub/datetime=190430/hour=11/FlumeData.1556594288836.tmp
[root@emr-worker-3 ~]#
```

查看 Log Service 上的消费组状态，详细步骤请参见#unique_96。



14 Hue

14.1 Hue使用说明

本文介绍如何在E-MapReduce上配置及访问Hue，通过使用Hue可以在浏览器端与Hadoop集群进行交互来分析处理数据。

前提条件

在集群#unique_7中设置安全组访问，打开8888端口，详情请参见#unique_9。

> **注意：**
> 设置安全组规则时要针对有限的IP范围。禁止在配置的时候对0.0.0.0/0开放规则。

查看初始密码

Hue服务默认在第一次运行时，如果未设置管理员则将第一个登录用户设置为管理员。因此出于安全考虑，E-MapReduce将默认为Hue服务创建一个名为admin的管理员账号，并为其设置一个随机的初始密码。您可以通过以下方式查看该管理员账号的初始密码：

1. 已通过主账号登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 单击左侧导航栏中的**集群服务**，在集群服务列表中，选择**Hue**。
6. 单击**配置**页签，找到**admin_pwd**参数，该参数对应的值就是随机密码。

> **注意：**
> **admin_pwd**仅为admin账号的初始密码，在E-MapReduce控制台上改变该密码不会同步到Hue中。如果需要改变admin账号在Hue中的登录密码，请使用该初始密码登录Hue，然后在Hue的用户管理模块中进行修改。

访问Hue

在E-MapReduce控制台中提供了快速访问集群中Hue服务的链接入口，您可通过以下方式访问Hue服务：

1. 已通过主账号登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。

3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 在页面左侧导航栏中，单击**访问链接与端口**。
6. 单击Hue服务所在行的链接。
7. 输入Hue账号和对应的密码。

创建Hue用户账号

如果您忘记了自己的Hue账号所对应的密码，可以通过以下方式重新创建一个账号：

1. 已通过主账号登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 在**主实例组**区域获取Master节点的公网IP。
6. 登录Master节点，具体步骤请参见#unique_17。
7. 执行以下命令，创建新账号。

```
/opt/apps/hue/build/env/bin/hue createsuperuser
```

8. 输入新用户名、电子邮件，然后输入密码，再次输入密码后，按**Enter**键。

   如果提示**Superuser created successfully**，则说明新账号创建成功，稍后用新账号登录Hue即可。

添加配置

您可以通过自定义配置添加相关配置：

1. 已通过主账号登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 在页面左侧导航栏中，单击**集群服务** > **Hue**。
6. 单击**配置**页签。
7. 在**服务配置**区域，单击**hue**。

8. 单击右上角的**自定义配置**，添加配置的key和Value值，其中key需要遵循下面规范。

```
$section_path.$real_key
```

> **说明：**
>
> - **$real_key**即为需要添加的实际的key，如hive_server_host。
> - **$section_path**可以通过hue.ini文件进行查看，例如：
>
> hive_server_host，通过hue.ini文件可以看出它属于**[beeswax]**这个section下，则**$section_path**为beeswax。
> - 综上，添加的key为beeswax.hive_server_host。
> - 同理，如需修改hue.ini文件中的多级section**[desktop] -> [[ldap]] -> [[[ldap_servers]]] -> [[[[users]]]] ->user_name_attr**的值，则需要配置的key为desktop.ldap.ldap_servers.users.user_name_attr。

调整YARN队列

HUE进行SQL交互查询时，需要向YARN申请资源进行计算，如果需要对计算资源进行管理和隔离，则需要配置HiveSQL和SparkSQL的对应队列。

1. 已通过主账号登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 修改或添加自定义配置。
   - HiveSQL需要根据不同引擎设置HiveServer2。

     > **注意：**
     > 本文的QUEUENAME为需要配置的队列名称。

     a. 单击左侧导航栏的**集群服务** > **Hive**。
     b. 单击**配置**页签。
     c. 单击**服务配置**区域的**hiveserver2-site**页签。
     d. 单击右上角的**自定义配置**添加相应如下配置：

引擎	配置项	说明
Hive on MR	mapreduce.job.queuename	QUEUENAME

引擎	配置项	说明
Hive on Tez	tez.queue.name	
Hive on Spark	spark.yarn.queue	

说明:
若需修改配置，可直接在**服务配置**页面修改配置项的值。

- SparkSQL使用SparkThriftServer，在Spark组件上修改spark-thriftServer配置或添加自定义配置：

  a. 单击左侧导航栏的**集群服务** > **Spark**。

  b. 单击**配置**页签。

  c. 单击**服务配置**区域的**spark-thriftServer**页签。

  d. 单击右上角的**自定义配置**添加相应如下配置：

  spark.yarn.queue：QUEUENAME

6. 重启Hue所在集群的HiveServer2和Spark的ThriftServer。

a) 在**集群管理**页面，单击**集群服务** > **Hive**。

b) 在**组件列表**区域，单击**HiveServer2**所在行的**重启**。

输入相关信息，单击**确定**。

c) 在**集群管理**页面，单击**集群服务** > **Spark**。

d) 在**组件列表**区域，单击**ThriftServer**所在行的**重启**。

输入相关信息，单击**确定**。

14.2 Hue对接LDAP

当您使用LDAP管理用户账号，访问Hue时需进行LDAP相关的配置。本文以Hue对接E-MapReduce自带的OpenLDAP为例，介绍如何配置Hue后端对接LDAP，并通过LDAP进行身份验证，自建的LDAP请根据实际情况进行修改。

操作步骤

1. 进入服务配置。

   a) 登录阿里云E-MapReduce控制台。

   b) 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。

   c) 单击上方的**集群管理**页签。

   d) 在**集群管理**页面，单击相应集群所在行的**详情**。

   e) 在左侧导航栏中，单击**集群服务** > **Hue**。

   f) 单击**配置**页签。

   g) 在**服务配置**区域，单击**hue**。

2. 修改**backend**的值为**desktop.auth.backend.LdapBackend**。

3. 增加自定义配置。

   a) 单击右上角的**自定义配置**，添加如下配置项。

配置项	说明	示例
desktop.ldap.ldap_url	LDAP服务器的URL。	ldap://emr-header-1:10389
desktop.ldap.bind_dn	绑定的用户dn，该dn用于连接到LDAP或AD以搜索用户和用户组信息。如果LDAP服务器支持匿名绑定，则此项可不设置。	uid=admin,o=emr
desktop.ldap.bind_password	绑定的用户dn的密码。	[password]
desktop.ldap.ldap_username_pattern	LDAP用户名dn匹配模式，描述了username如何对应到LDAP中的dn。必须包含特殊的**<username>**字符串才能在身份验证期间用于替换。	uid=<username>,ou=people,o=emr
desktop.ldap.base_dn	用于搜索LDAP用户名及用户组的base dn。	ou=people,o=emr

配置项	说明	示例
desktop.ldap.search_bind_authentication	是否使用**desktop.ldap.bind_dn**和**desktop.ldap.bind_password**配置中提供的凭据，搜索绑定身份验证连接到LDAP服务器。	false
desktop.ldap.use_start_tls	是否尝试与用ldap://指定的LDAP服务器建立TLS连接。	false
desktop.ldap.create_users_on_login	用户尝试以LDAP凭据登录后，是否在Hue中创建用户。	true

b) 单击**确定**。

4. 保存配置。

a) 单击右上角的**保存**。

b) 开启**自动更新配置**并设置相关信息。

c) 单击**确定**。

5. 部署配置。

a) 单击右上角的**部署客户端配置**。

b) 设置相关信息。

c) 单击**确定**。

6. 单击右上角的**操作** > **重启Hue**。

后续步骤

注意:

对接LDAP之后，原有的管理员账号admin已经不能登录，新的管理员用户为对接LDAP之后第一个登录的用户。

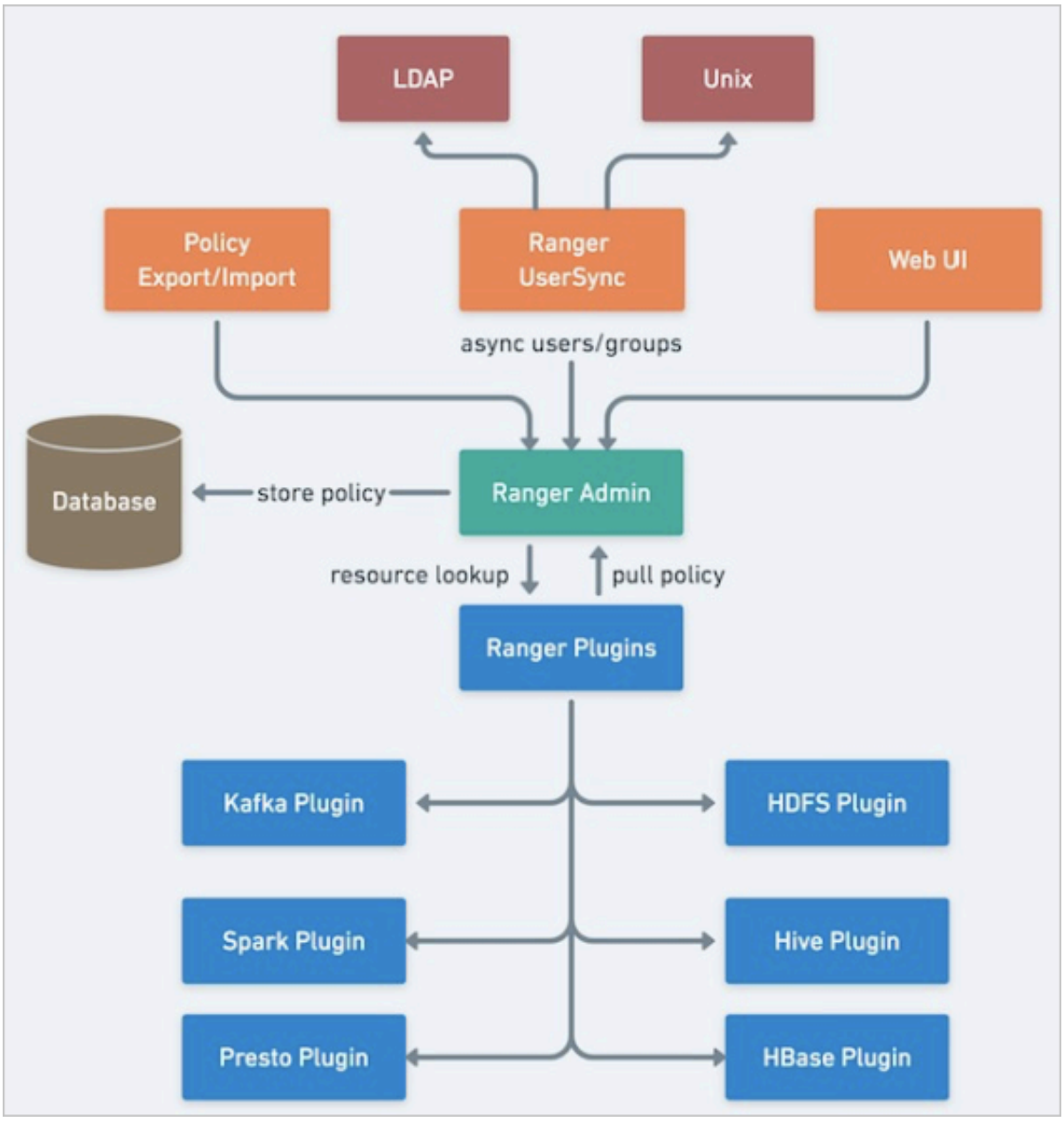
访问Hue，请参见Hue使用说明。

15 Ranger

15.1 Ranger简介

Apache Ranger提供集中式的权限管理框架，可以对Hadoop生态中的HDFS、Hive、YARN、Kafka、Storm和Solr等组件进行细粒度的权限访问控制，并且提供了Web UI方便管理员进行操作。

组件介绍



Ranger主要由三个组件组成：

- Ranger Admin

  用户可以创建、更新安全访问策略，这些策略被存储在数据库中。各个组件的Plugin定期对这些策略进行轮询。

- Ranger Plugins

  Plugin嵌入在各个集群组件的进程里，是一个轻量级的Java程序。例如，Ranger对Hive的组件，就被嵌入在Hiveserver2里。这些Plugin从Ranger Admin服务端拉取策略，并把它们存储在本地文件中。当接收到来自组件的用户请求时，对应组件的plugin会拦截该请求，并根据安全策略对其进行评估。

- Ranger UserSync

  Ranger提供了一个用户同步工具，可以从Unix或者LDAP中拉取用户和用户组的信息。这些用户和用户组的信息被存储在Ranger Admin的数据库中，可以在定义策略时使用。

组件安装

- 新建集群时，如果在E-MapReduce控制台创建EMR-2.9.2或EMR-3.9.0及以上的集群，直接勾选Ranger组件即可。

- 已有EMR-2.9.2或EMR-3.9.0及以上的集群，可以在**集群管理**页面添加Ranger服务。

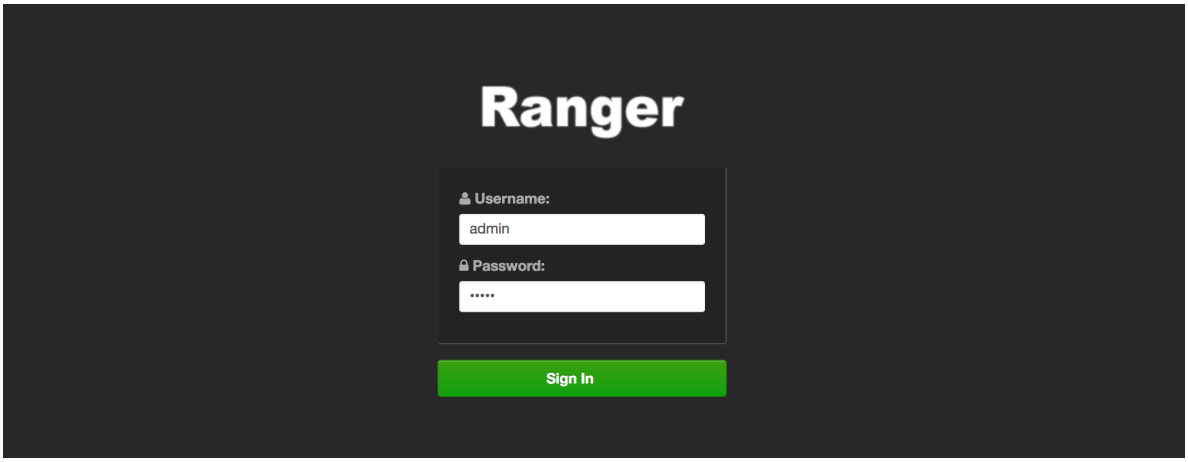
说明:

- 开启Ranger后，设置安全控制策略之前，不会对应用程序产生影响和限制。
- Ranger中设置的用户策略为集群Hadoop账号。

访问Ranger UI

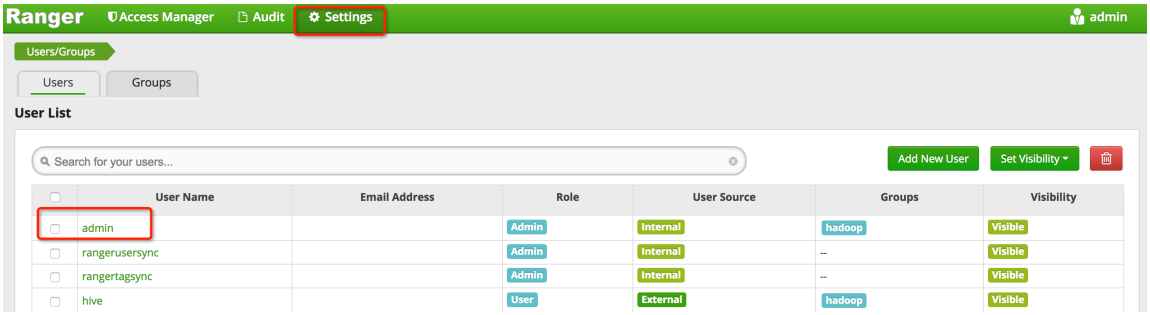
在访问Ranger UI之前，需要确认已设置安全组，即Hadoop集群允许您通过当前网络访问该集群。具体详情请参见#unique_9。

1. 登录阿里云E-MapReduce控制台。
2. 单击上方的**集群管理**页签。
3. 在**集群管理**页面，单击相应集群所在行的**详情**。
4. 在左侧导航栏中，选择**访问链接与端口**，可以通过链接访问Ranger UI。
5. 单击**RANGER UI**所在行的链接。
6. 在Ranger UI登录界面，输入用户名和密码（默认的用户名和密码均为admin）。

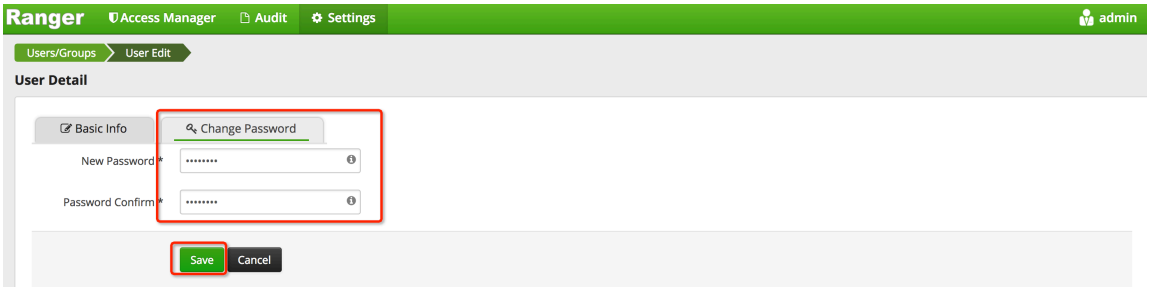


7. 修改密码。

a) 管理员首次登录后，需单击上方的**Settings**。



b) 修改**admin**的密码。



c) 单击右上角的**admin** > **Log Out**。

使用新的密码登录即可。

组件集成Ranger

通过插件的方式，Ranger与集群中的开源组件进行集成。通过Ranger可以对组件进行细粒度的访问权限控制。目前已经支持的组件和EMR版本如下表所示。

组件名	EMR版本
HDFS	EMR-2.9.2 /EMR-3.9.0 及以上
Hive	EMR-2.9.2 /EMR-3.9.0 及以上
HBase	EMR-2.9.2 /EMR-3.9.0 及以上
Yarn	EMR-2.9.2 /EMR-3.9.0 及以上

组件名	EMR版本	
Kafka	EMR-3.12.0及以上	
Spark	EMR-3.24.0及以上	
Presto	EMR-3.25.0及以上	

可以将集群中的相关组件集成到Ranger，进行相关权限的控制，详情请参见以下配置。

- HDFS配置
- Hive配置
- Spark配置
- Presto配置
- Kafka配置
- HBase配置

用户管理

用户可以使用Ranger对用户或用户组进行权限管理。用户或用户组可以来自本地Unix系统或者LDAP，推荐使用LDAP。

- Ranger admin与LDAP集成

  详情请参见Ranger admin与LDAP集成。
- Ranger usersync与LDAP集成

  详情请参见Ranger usersync对接LDAP。

15.2 组件集成

15.2.1 HDFS配置

本文介绍HDFS如何集成到Ranger，以及如何配置权限。

HDFS集成Ranger

1. Ranger启用HDFS。

   a) 登录阿里云E-MapReduce控制台。

   b) 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。

   c) 单击上方的**集群管理**页签。

   d) 在**集群管理**页面，单击相应集群所在行的**详情**。

   e) 在左侧导航栏单击**集群服务** > **RANGER**。

   f) 单击右侧的**操作**下拉菜单，选择**启用HDFS**。

   g) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

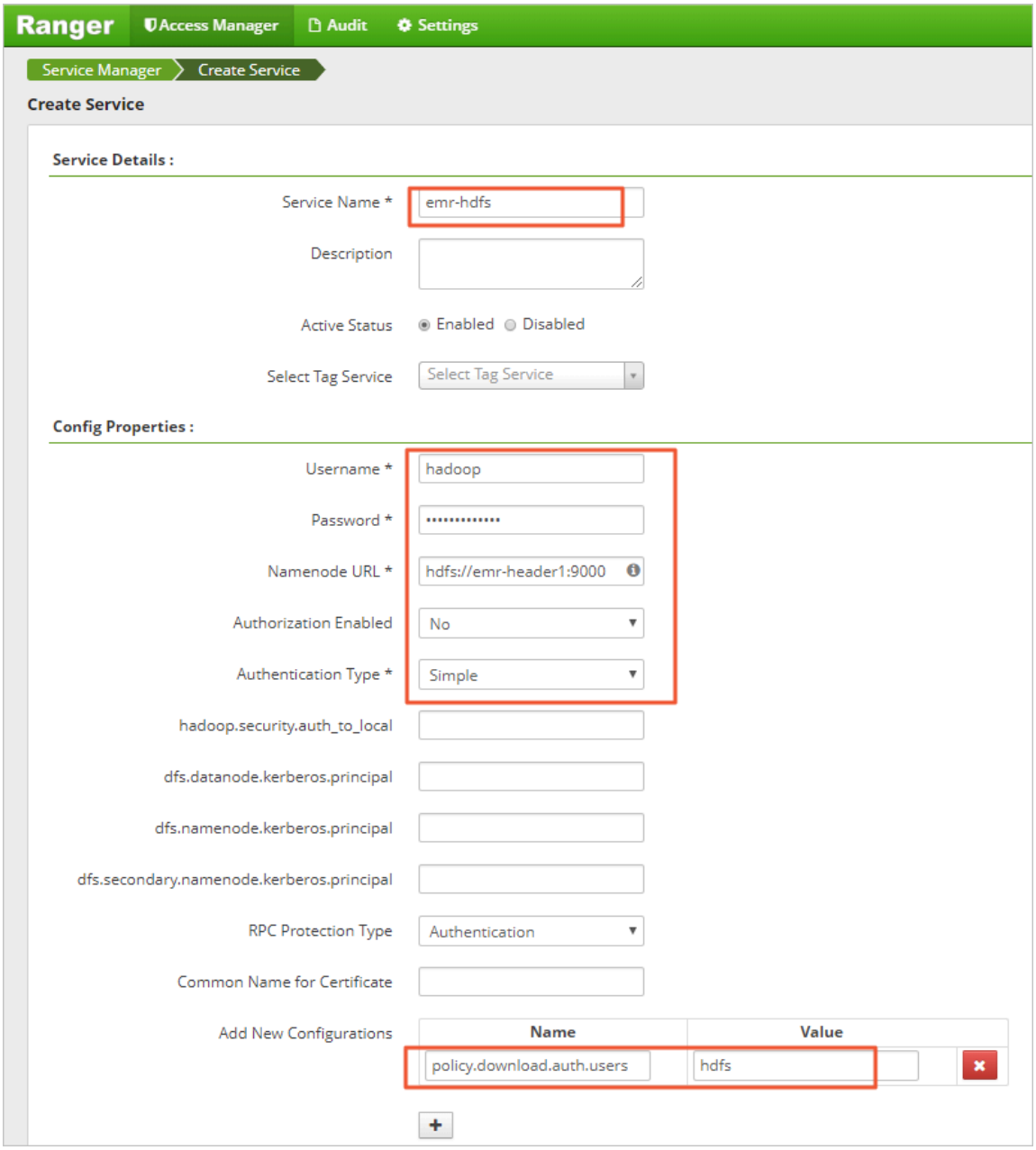
   单击右上角**查看操作历史**查看任务进度，等待任务完成。

2. Ranger UI添加HDFS Service。

   a) 进入Ranger UI页面，详情请参见Ranger简介。

   b) 在Ranger UI页面添加HDFS Service。

   c) 配置相关参数。



参数	说明
Service Name	固定值emr-hdfs。

参数	说明
Username	固定值hadoop。
Password	自定义。
Namenode URL	• 标准集群：hdfs://emr-header1:9000。 • 高安全集群：hdfs://emr-header1:8020。
Authorization Enabled	标准集群选择**No**；高安全集群选择**Yes**。
Authentication Type	• Simple：表示标准集群。 • Kerberos：表示高安全集群。
dfs.datanode.kerberos.principal	标准集群时不填写；高安全集群时填写**hdfs/_HOST@EMR.${id}.com**。 **说明：** ${id}可登录机器执行hostname命令，hostname中的数字即为${id}的值。
dfs.namenode.kerberos.principal	
dfs.secondary.namenode.kerberos.princip	
Add New Configurations	填写如下： • **Name**：固定值**policy.download.auth.users**。 • **Value**：固定值**hdfs**。

3. 重启HDFS。

   a) 左侧导航栏单击**集群服务** > **HDFS**。

   b) 单击右上角**操作**下拉菜单，选择**重启 All Components**。

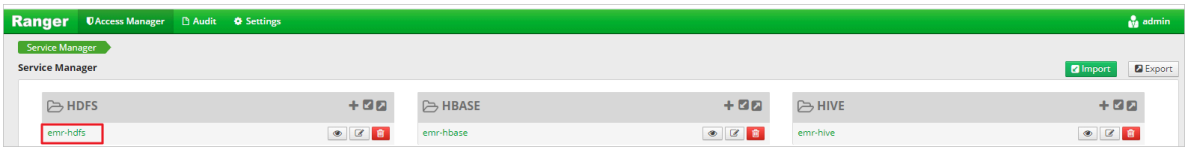
   c) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

   单击右上角**查看操作历史**查看任务进度，等待任务完成。

权限配置示例

上面一节中已经将Ranger集成到HDFS，现在可以进行相关的权限设置。例如给test用户授予/user/foo路径的写或执行权限：

1. 单击配置好的**emr-hdfs**。



2. 单击右上角的**Add New Policy**。

3. 配置相关参数。

参数	说明
Policy Name	策略名称，可以自定义。
Resoure Path	资源路径。
Recursive	子目录或文件是否集成权限。
Select Group	指定添加此策略的用户组。
Select User	指定添加此策略的用户。
Permissions	选择授予的权限。

4. 单击**add**。

添加Policy后，实现了对test用户的授权。test用户即可访问/user/foo的HDFS路径。

说明:

添加、删除或修改Policy后，需要等待一分钟左右授权才能生效。

15.2.2 HBase配置

本文介绍HBase如何集成到Ranger，以及如何配置权限。

HBase集成Ranger

1. Ranger启用HBase。

   a) 登录阿里云E-MapReduce控制台。

   b) 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。

   c) 单击上方的**集群管理**页签。

   d) 在**集群管理**页面，单击相应集群所在行的**详情**。

   e) 在左侧导航栏单击**集群服务** > **RANGER**。

   f) 单击右侧的**操作**下拉菜单，选择**启用HBase**。

   g) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

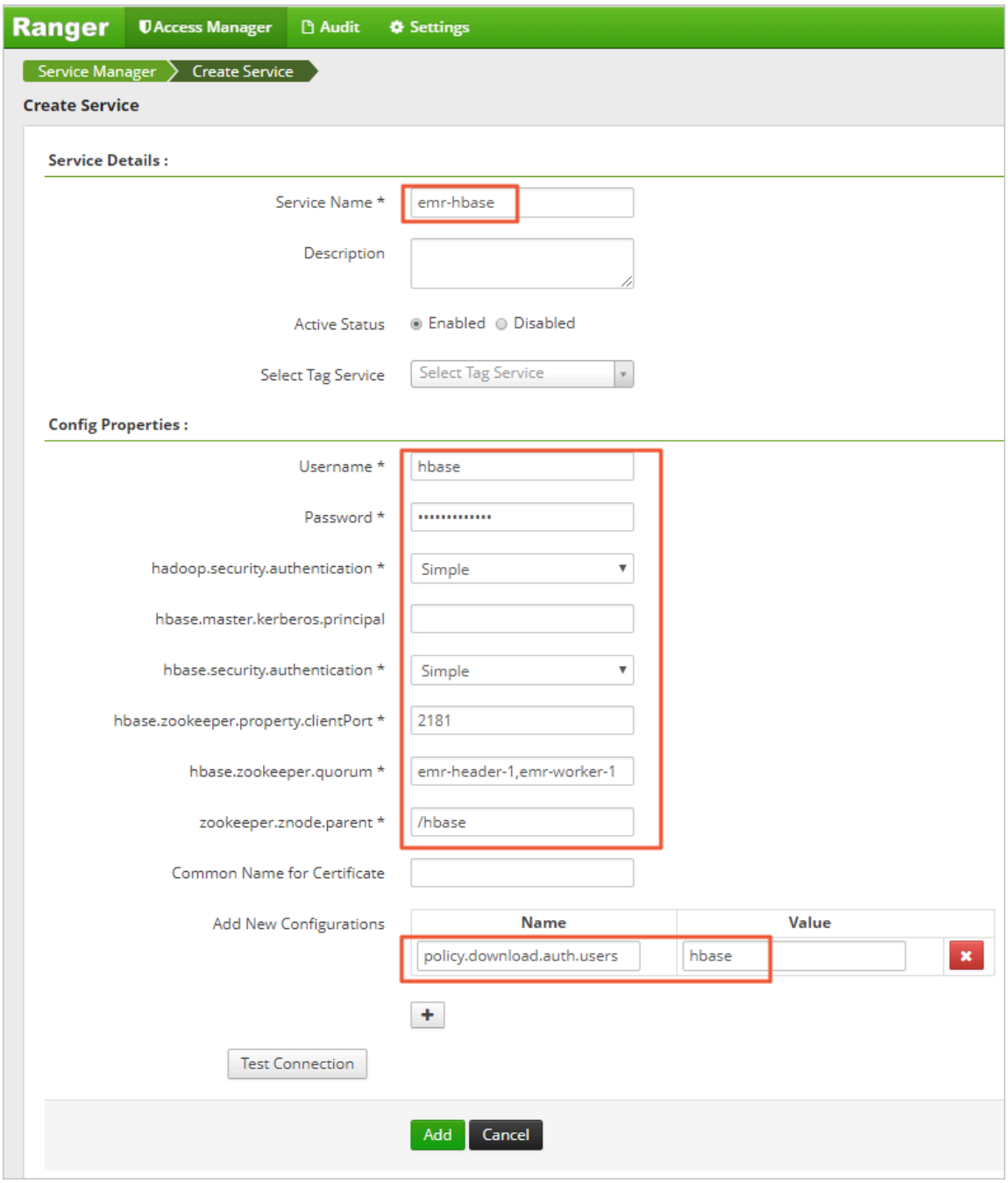
   单击右上角**查看操作历史**查看任务进度，等待任务完成。

2. Ranger UI添加HBase Service。

   a) 进入Ranger UI页面，详情请参见Ranger简介。

   b) 在Ranger UI页面，添加HBase Service。

   c) 配置相关参数。



参数	说明
Service Name	固定值**emr-hbase**。

参数	说明
Username	固定填写**hbase**。
Password	自定义。
hadoop.security.authentication	• 标准集群（非高安全集群）：选择**Simple**。 • 高安全集群：选择**Kerberos**。
hbase.master.kerberos.principal	标准集群时不填写；高安全集群时填写**hbase/_HOST@EMR.${id}.COM**。 **说明：** ${id}可登录机器执行hostname命令，hostname中的数字即为${id}的值。
hbase.security.authentication	• 标准集群（非高安全集群）：选择**Simple**。 • 高安全集群：选择**Kerberos**。
hbase.zookeeper.property.clientPort	固定值**2181**。
hbase.zookeeper.quorum	固定值**emr-header-1,emr-worker-1**。
zookeeper.znode.parent	固定值**/hbase**。
Add New Configurations	• **Name**：固定值**policy.download.auth.users**。 • **Value**：固定值**hbase**。

d) 单击**Add**。

3. 重启HBase。

a) 左侧导航栏单击**集群服务** > **HBase**。

b) 单击右上角**操作**下拉菜单，选择**重启 All Components**。

c) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

单击右上角**查看操作历史**查看任务进度，等待任务完成。

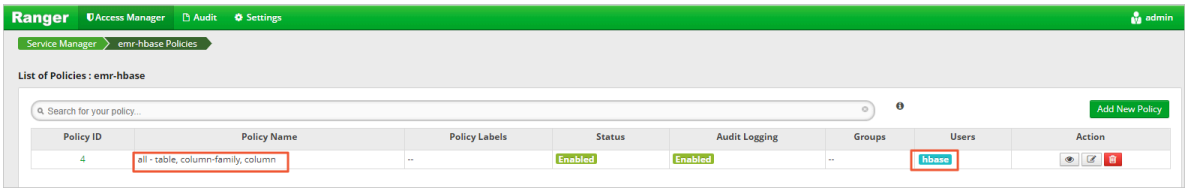
设置管理员账号

1. 在**Service Manager**页面，单击创建好的**emr-hbase**。

2. 设置管理员账号的权限（admin权限）。

   用于执行管理命令，例如balance、compaction、flush或split等。

   因为当前服务已经存在权限策略，所以您只需要单击右侧的  图标，在User中添加需要设置的账号即可。另外也可以修改其中的权限（例如只保留admin权限）。HBase账号必须默认设置为管理员账号。



   如果使用Phoenix，则需在Ranger的HBase中新增如下策略。

参数	说明
HBase Column-family	*
HBase Column	*
Select Group	**public**
Permissions	**Read**、**Write**、**Create**和**Admin**,

权限配置示例

上面一节中已经将Ranger集成到HBase，可以进行权限设置。例如给test用户授予表foo_ns:test的Create/Write/Read权限。

1. 单击配置好的**emr-hbase**。

2. 单击右上角的**Add New Policy**。
3. 配置相关权限。

参数	说明
Policy Name	策略名称，可以自定义。

参数	说明
HBase Table	表对象，格式为**${namespace}:${tablename}**。可输入多个，填写一个需按一次Enter键。 如果是default的namespace，不需要加default。支持通配符星号（*），例如，**foo_ns:***表示foo_ns下的所有表。 **说明:** 目前不支持**default:***。
HBase Column-family	列簇。
HBase Column	列名。
Select Group	指定添加此策略的用户组。
Select User	指定添加此策略的用户。
Permissions	选择授予的权限。

4. 单击**add**。

   添加Policy后，实现对test用户的授权。test用户即可以对foo_ns:test表进行访问。

   说明:
   添加、删除或修改Policy后，需要等待一分钟左右授权才能生效。

15.2.3 Hive配置

本文介绍HDFS如何集成到Hive，以及如何配置权限。

Hive访问模型

访问Hive数据，包括HiveServer2、Hive Client和HDFS三种方式：

- HiveServer2方式
  - 场景： 您可以通过HiveServer2访问Hive数据。
  - 方式：使用Beeline客户端或者JDBC代码通过HiveServer2执行Hive脚本。
  - 权限设置：

    Hive官方自带的Hive授权就是针对HiveServer2使用场景进行权限控制的。

    Ranger中对Hive的表或列级别的权限控制也是针对HiveServer2的使用场景。如果您还可以通过Hive Client或者HDFS访问Hive数据，仅仅对表或列层面做权限控制还不够，需要下面两种方式设置以进一步控制权限。
- Hive Client方式
  - 场景： 您可以通过Hive Client访问Hive数据。
  - 方式： 使用Hive Client访问。
  - 权限设置：

    Hive Client会请求HiveMetaStore进行一些DDL操作（例如Alter Table Add Columns等），也会通过提交MapReduce作业读取HDFS中的数据进行处理。

    Hive官方自带的Hive授权只针对Hive Client使用场景进行的权限控制，它会根据SQL中表的HDFS路径的读写权限，来决定该您是否可以进行DDL或DML操作，例如`ALTER TABLE test ADD COLUMNS(b STRING)`

    Ranger中可以对Hive表的HDFS路径进行权限控制，加上HiveMetaStore配置Storage Based Authorization，从而可以实现对Hive Client访问场景的权限控制。

    > **说明：**
    > Hive Client场景的DDL操作权限实际也是通过底层HDFS权限控制，所以如果您有HDFS权限，则对应也会有表的DDL操作权限（例如Drop Table或Alter Table等）。
- HDFS方式
  - 场景： 您可以通过HDFS访问数据。
  - 方式： HDFS客户端或代码等。
  - 权限设置：

    您可以直接访问HDFS，需要对Hive表的底层HDFS数据增加HDFS的权限控制。

    通过Ranger对Hive表底层的HDFS路径进行权限控制，详情请参见权限配置示例。

Hive集成Ranger

1. Ranger启用Hive。

   a) 登录阿里云E-MapReduce控制台。

   b) 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。

   c) 单击上方的**集群管理**页签。

   d) 在**集群管理**页面，单击相应集群所在行的**详情**。

   e) 在左侧导航栏单击**集群服务** > **RANGER**。

   f) 单击右侧的**操作**下拉菜单，选择**启用Hive**。

   g) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

   单击右上角**查看操作历史**查看任务进度，等待任务完成。

说明：

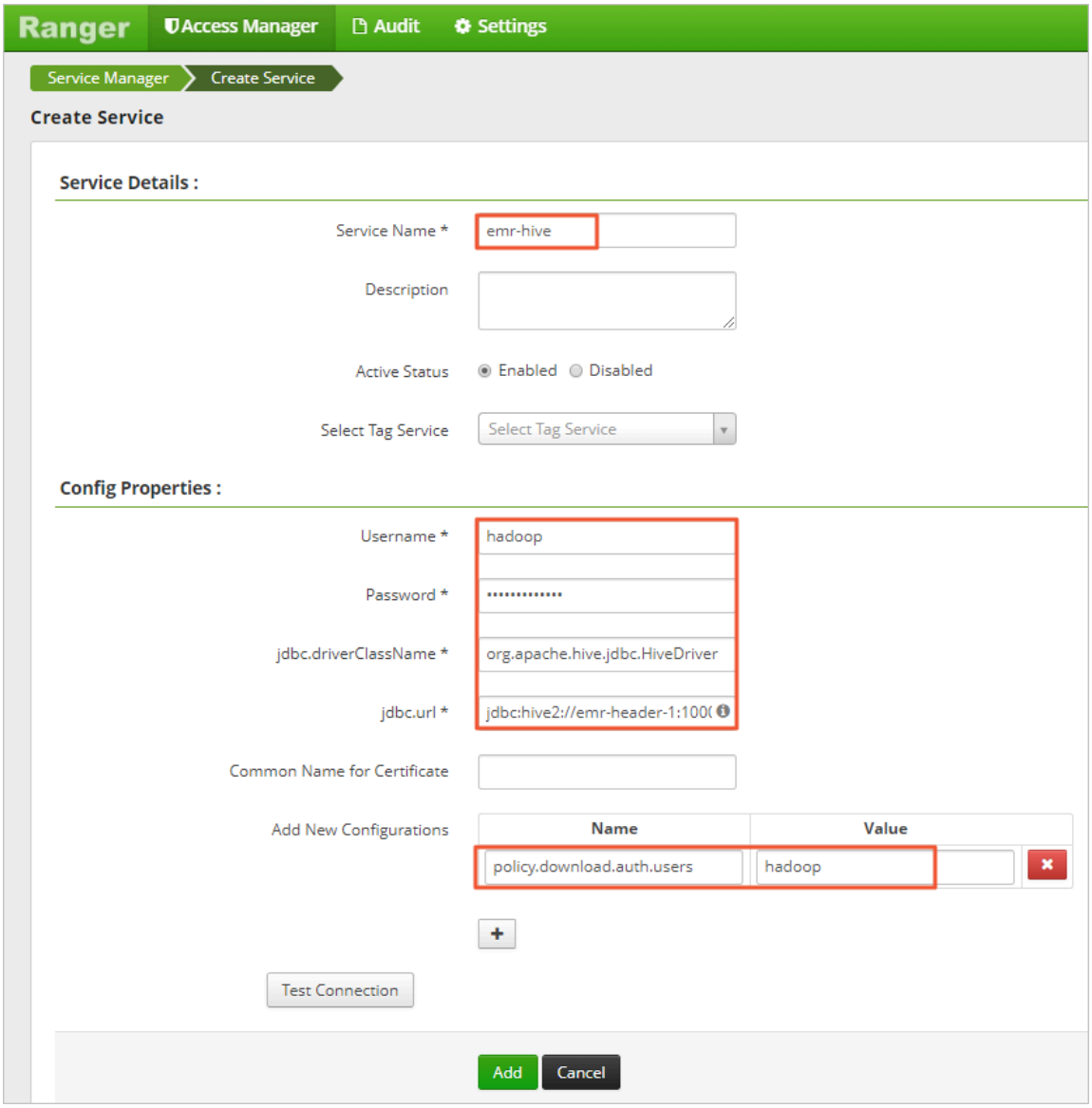
Enable Hive Plugin并重启Hive后，HiveServer2场景或HiveClient场景（Storage Based Authorization）配置参数，HiveServer2场景使用Ranger进行权限配置，HiveClient场景需要借助HDFS权限进行控制，HDFS的权限可参见HDFS配置进行开启。

2. Ranger UI添加Hive Service。

   a) 进入Ranger UI页面，详情请参见Ranger简介。

   b) 在Ranger UI页面，添加Hive Service。

   c) 配置参数。



参数	说明
Service Name	固定值**emr-hive**。
Username	固定值**hadoop**。
Password	自定义。

参数	说明
jdbc.driverClassName	默认值**org.apache.hive.jdbc.HiveDriver**。无需修改。
jdbc.url	• 标准集群： **jdbc:hive2://emr-header-1:10000/** • 高安全集群：**jdbc:hive2://${master1_fullhost}:10000/;principal=hive/${master1_fullhost}@EMR.$id.COM** **说明:** ${master1_fullhost}为master 1的长域名，可登录master 1执行hostname命令获取，${master1_fullhost} 中的数字即为$id的值。
Add New Configurations	• **Name**：固定值**policy.download.auth.users**。 • **Value**：hadoop（标准集群）、hive（高安全集群）。

说明:
测试连接失败时，可忽略。

d) 单击**Add**。

3. 重启Hive。

上述任务完成后，需要重启Hive才生效。

a) 左侧导航栏单击**集群服务** > **Hive**。

b) 单击右上角**操作**下拉菜单，选择**重启 All Components**。

c) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

单击右上角**查看操作历史**查看任务进度，等待任务完成。

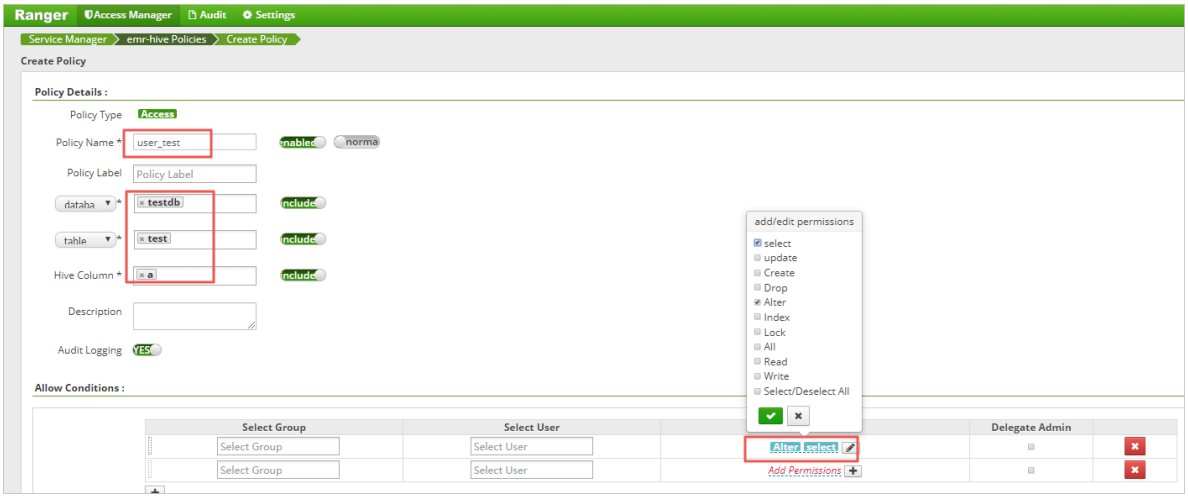
权限配置示例

上面一节中已经将Ranger集成到Hive，现在可以设置权限。例如给foo用户授予表testdb.test的a列Select权限。

1. 单击配置好的**emr-hive**。

2. 单击右上角的**Add New Policy**。

3. 配置权限。



参数	说明
Policy Name	策略名称，可以自定义。
database	添加Hive中的数据库，例如testdb。
table	添加表，例如test。
Hive Column	可添加列名。填写*时表示所有列。
Select Group	指定添加此策略的用户组。
Select User	指定添加此策略的用户。
Permissions	选择授予的权限。

4. 单击**add**。

添加Policy后，实现对foo的授权。foo用户即可以访问testdb.test表。

> **说明:**
> 添加、删除或修改Policy后，需要等待一分钟左右授权才能生效。

15.2.4 Spark配置

本文介绍Spark如何集成到Ranger，以及相关的权限配置。

前提条件

已创建集群，详情请参见#unique_12。

> **说明：**
> 创建集群时，产品版本需高于EMR-3.26.0。

背景信息

Spark集成Ranger进行权限控制，仅适用于通过Spark ThriftServer执行Spark SQL作业，例如，使用Spark的beeline客户端或JDBC通过Spark ThriftServer去提交Spark SQL作业。

Spark SQL集成Ranger

1. 配置Hive集成Ranger。

   在Ranger中，Spark SQL与Hive共享权限配置，想要通过Ranger控制Spark SQL的权限，首先需要将Hive集成Ranger，详细信息请参见Hive配置。

2. Ranger启用Spark。

   a) 在**集群管理**页面，单击相应集群所在行的**详情**。

   b) 在左侧导航栏单击**集群服务** > **RANGER**。

   c) 单击右侧的**操作**下拉菜单，选择**启用Spark**。

   d) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

   单击右上角**查看操作历史**查看任务进度，等待任务完成。

3. 重启Spark ThriftServer。

   a) 在左侧导航栏单击**集群服务** > **Spark**。

   b) 在Spark**集群服务**页面，单击右侧的**操作** > **重启ThriftServer**。

   c) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

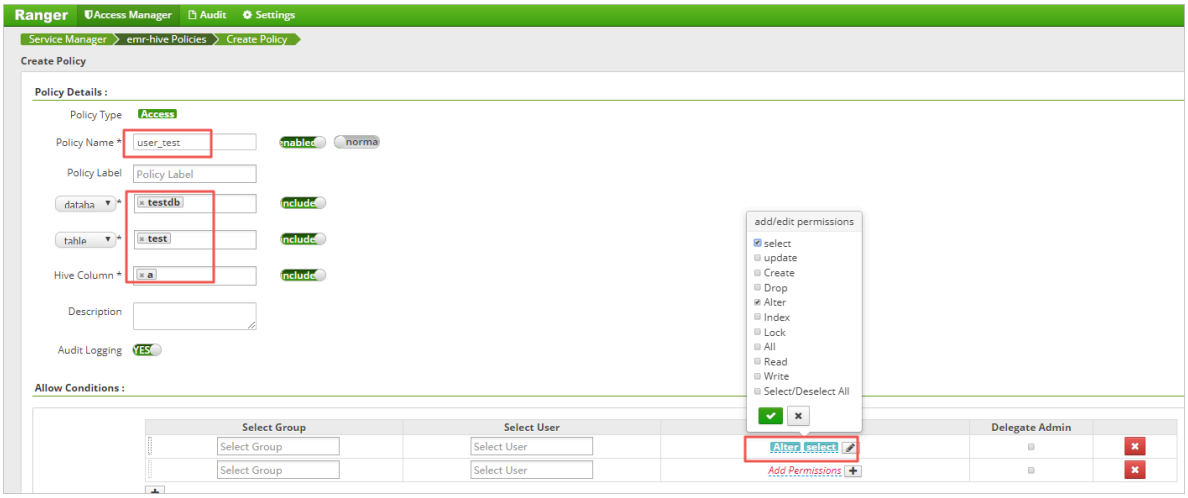
   单击右上角**查看操作历史**查看任务进度，等待任务完成。

权限配置示例（Ranger UI配置相关权限）

例如：给用户foo授予表testdb.test的a列Select权限。

1. 单击配置好的**emr-hive**。

2. 单击右上角的**Add New Policy**。
3. 配置权限。



参数	说明
Policy Name	策略名称，可以自定义。
database	添加Hive中的数据库，例如testdb。
table	添加表，例如test。
Hive Column	可添加列名。填写*时表示所有列。
Select Group	指定添加此策略的用户组。
Select User	指定添加此策略的用户。
Permissions	选择授予的权限。

4. 单击**add**。

添加Policy后，实现对foo的授权。foo用户即可以访问testdb.test表。

> **说明:**
> 添加、删除或修改Policy后，需要等待一分钟左右授权才能生效。

15.2.5 Kafka配置

本文介绍HBase如何集成到Kafka，以及如何配置权限。

Kafka集成Ranger

1. Ranger启用Kafka。

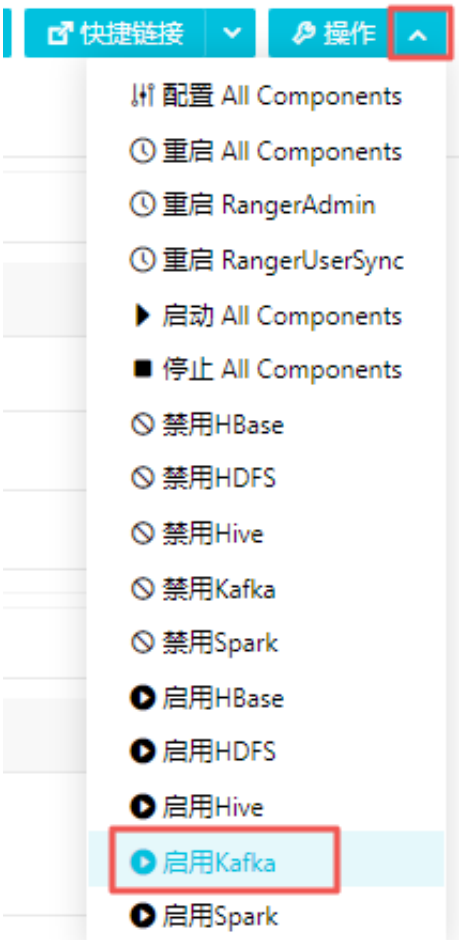
   a) 登录阿里云E-MapReduce控制台。

   b) 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。

   c) 单击上方的**集群管理**页签。

   d) 在**集群管理**页面，单击相应集群所在行的**详情**。

   e) 单击右侧的**操作**，选择**启用Kafka**。



   f) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

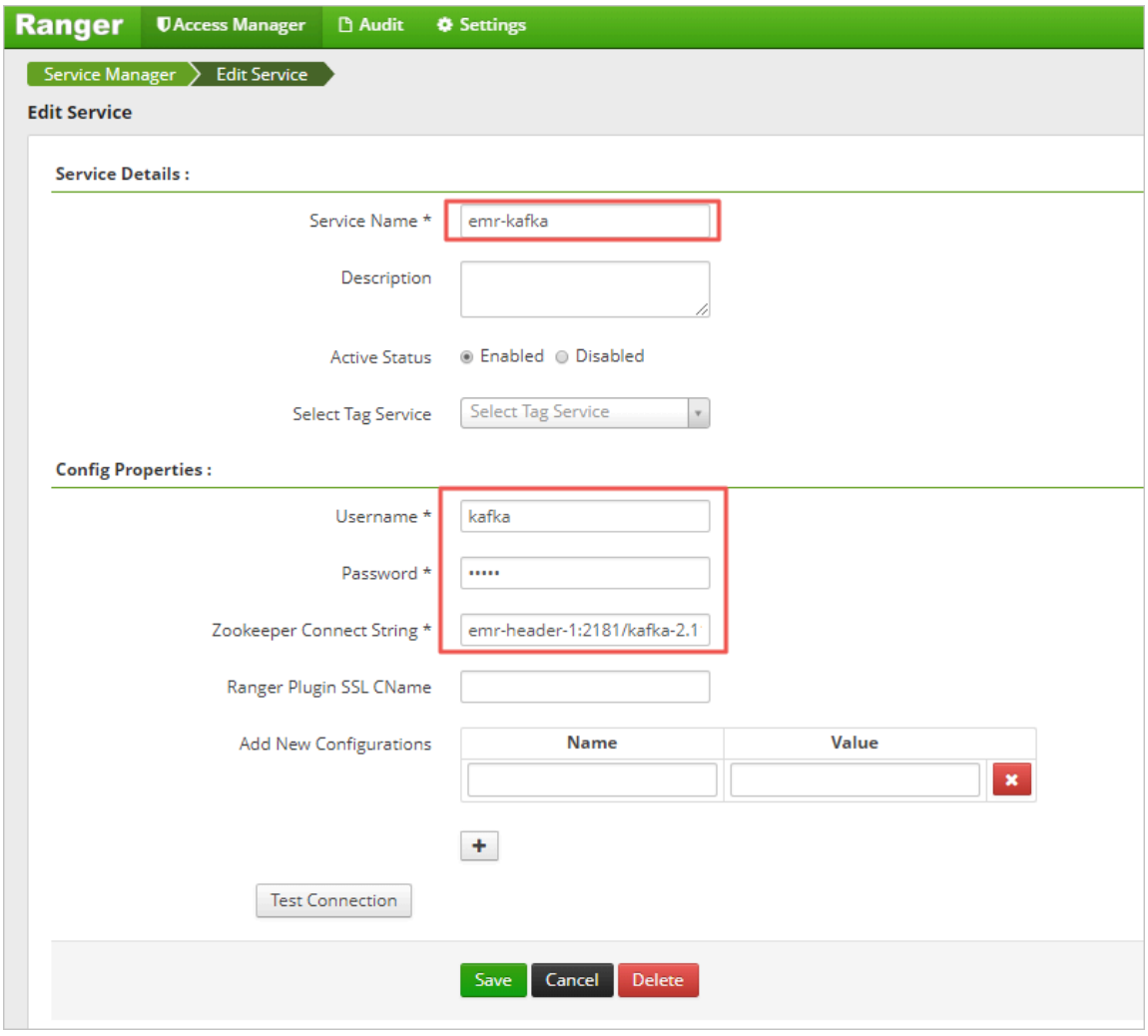
   单击右上角**查看操作历史**查看任务进度，等待任务完成。

2. Ranger UI页面添加Kafka Service。

a) 进入Ranger UI页面，详情请参见Ranger简介。

b) 在Ranger UI页面，添加Kafka Service。

c) 配置Kafka Service。



参数	说明
Service Name	固定值**emr-kafka**。
Username	固定值**kafka**。
Password	可自定义。

参数	说明
Zookeeper Connect String	填写格式**emr-header-1:2181/kafka-x.xx**。 **说明：** 其中kafka-x.xx根据Kafka实际版本填写。

d) 单击**add**。

3. 重启Kafka Broker。

a) 左侧导航栏单击**集群服务** > **Kafka**。

b) 单击右上角**操作**下拉菜单，选择 **重启 Kafka Broker**。

c) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

单击右上角**查看操作历史**查看任务进度，等待任务完成。

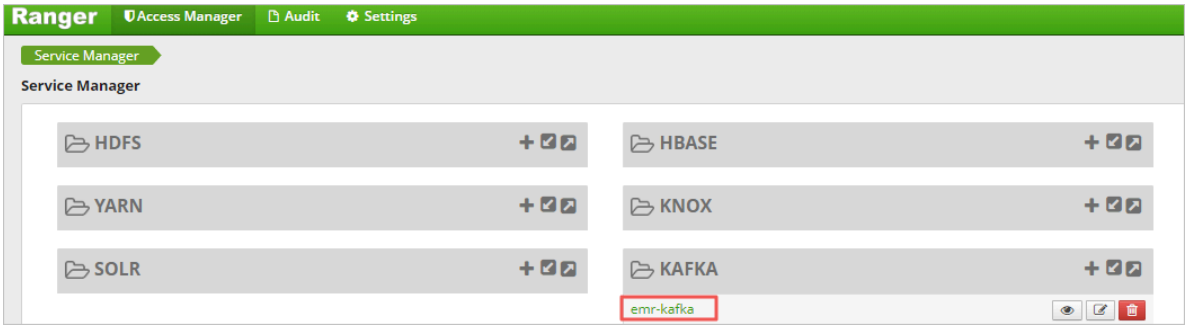
权限配置示例

上面一节中已经将Ranger集成到Kafka，现在可以设置权限。

注意：
标准集群中，在添加了Kafka Service后，Ranger会默认生成规则all - topic，不作任何权限限制（即允许所有用户进行所有操作），此时Ranger无法通过用户进行权限识别。

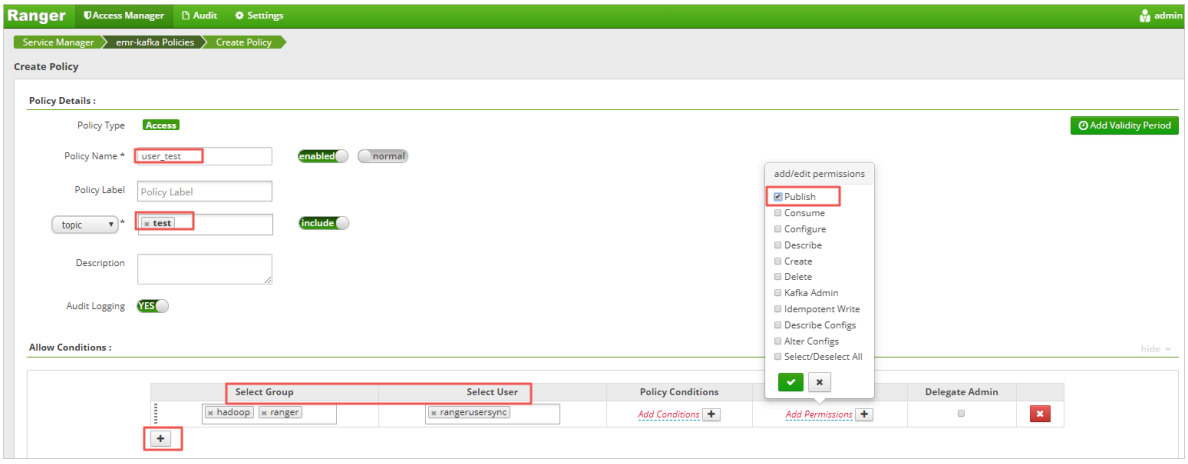
以test用户为例，添加Publish权限。

1. 单击配置好的**emr-kafka**。



2. 单击右上角的**Add New Policy**。

3. 填写参数。



参数	说明
Policy Name	策略名称，可以自定义。
topic	自定义。可填写多个，填写一个需按一次Enter键。
Select Group	指定添加此策略的用户组。
Select User	指定添加此策略的用户。
Permissions	单击 ，选择**Publish**。

单击**Select Group**下方的 ，可对多个Group进行授权。

4. 单击**add**。

添加Policy后，实现对**test**的授权。test用户即可以对**test**的topic进行写入操作。

说明：

添加、删除或修改Policy后，需要等待一分钟左右授权才能生效。

15.2.6 Presto配置

本文介绍Presto如何集成到Ranger，以及如何配置权限。

Presto集成Ranger

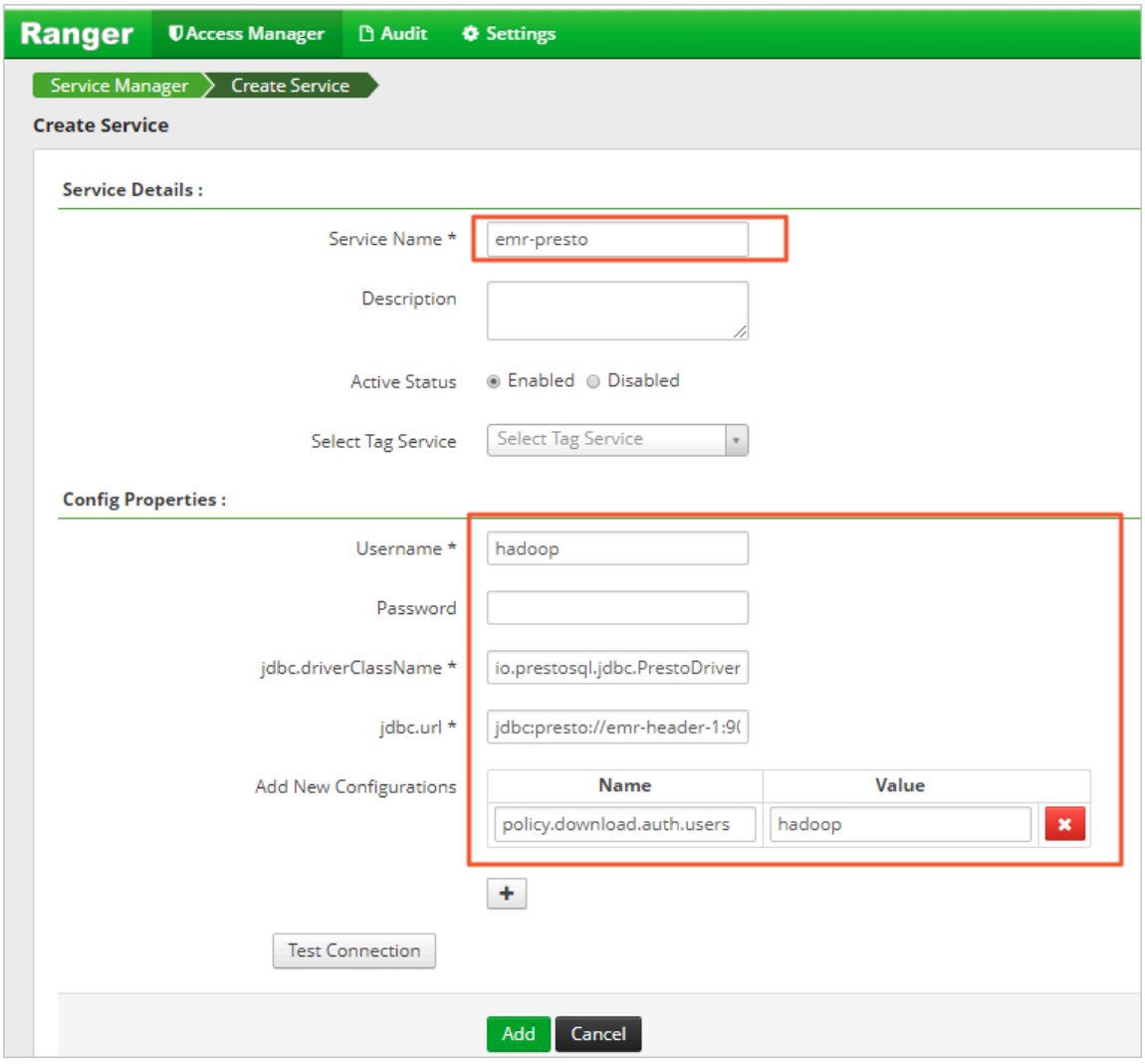
1. Ranger启用Presto。
   a) 登录阿里云E-MapReduce控制台。
   b) 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
   c) 单击上方的**集群管理**页签。
   d) 在**集群管理**页面，单击相应集群所在行的**详情**。
   e) 在左侧导航栏单击**集群服务** > **RANGER**。
   f) 单击右侧的**操作**下拉菜单，选择**启用Presto**。
   g) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。
   单击右上角**查看操作历史**查看任务进度，等待任务完成。

2. Ranger UI添加Presto Service。

   a) 进入Ranger UI页面，详情请参见Ranger简介。

   b) 在Ranger UI页面添加Presto Service。

   c) 配置参数。



参数	说明
Service Name	固定填写**emr-presto**。
Username	固定填写**hadoop**。

参数	说明
Password	不需要填写。
jdbc.driverClassName	固定填写**io.prestosql.jdbc.PrestoDriver**。
jdbc.url	固定填写**jdbc:presto://emr-header-1:9090**。
Add New Configurations	• **Name**：固定值**policy.download.auth.users**。 • **Value**：固定值**hadoop**。

d) 单击**Add**。

3. 重启Presto Master。

a) 左侧导航栏单击**集群服务** > **Presto**。

b) 单击右上角**操作**下拉菜单，选择**重启PrestoMaster**。

c) 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

单击右上角**查看操作历史**查看任务进度，等待任务完成。

权限配置示例

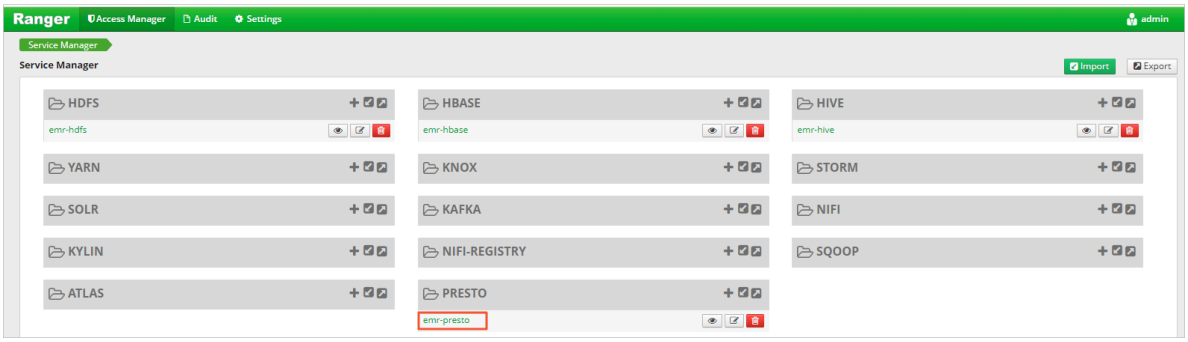
Ranger Presto权限控制与Ranger Hive、Ranger Hbase等权限控制不同，Ranger Presto采用的是权限分层次控制的策略，需要特别注意权限配置的方法。

注意：

- 配置的权限应与所属的层次保持一致。如果配置的权限与所属的层次不相符，则该权限配置将不起作用。
- Presto在对某个用户进行权限检查时，会进行两次权限检查，首先检查该用户是否有访问Catalog的权限，再检查本次访问所涉及到的权限。

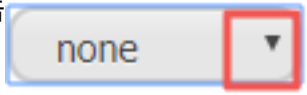
示例一：给用户liu授予访问hive表testdb.test的a列的Select权限。

1. 单击配置好的**emr-presto**。

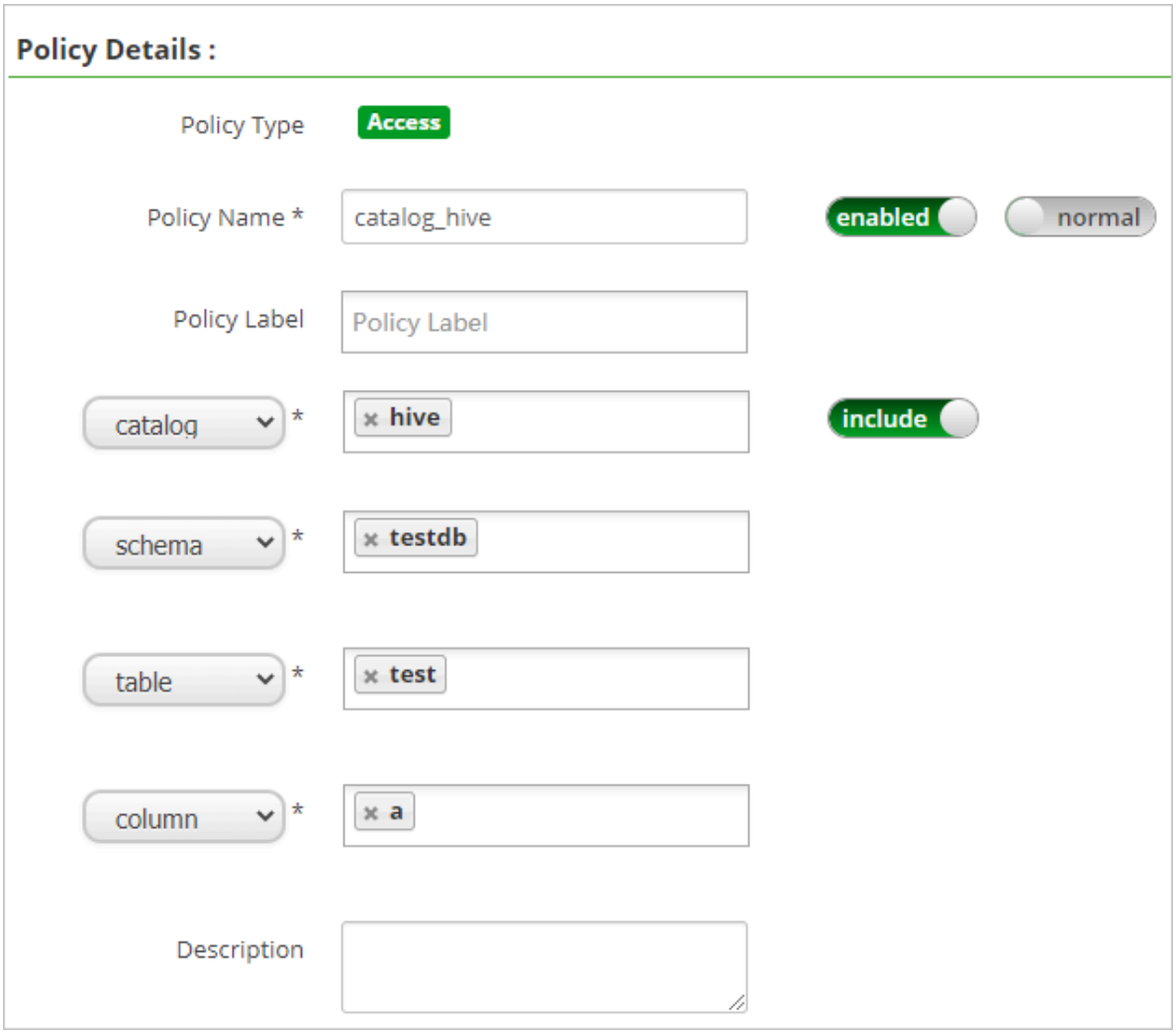


2. 单击右上角的**Add New Policy**。

3. 添加Policy对catalog的访问权限进行控制。

   配置liu用户对表testdb.test的a列的Select权限。由于表的Select权限属于column层次，因此需要单击 

图标，依次添加schema、table、column层次的内容。



参数	说明
Policy Name	策略名称，可自定义。例如catalog_hive。
Presto Catalog	Catalog的名称，可自定义。例如hive。
schema	添加schema的名称，例如testdb。星号（*）表示所有schema。
table	添加table的名称，例如test。星号（*）表示所有table。
column	添加column的名称，例如a。星号（*）表示所有column。

参数	说明
Select User	指定添加此策略的用户，例如liu。
Permissions	选择授予的权限，例如Select。

4. 单击**Add**。

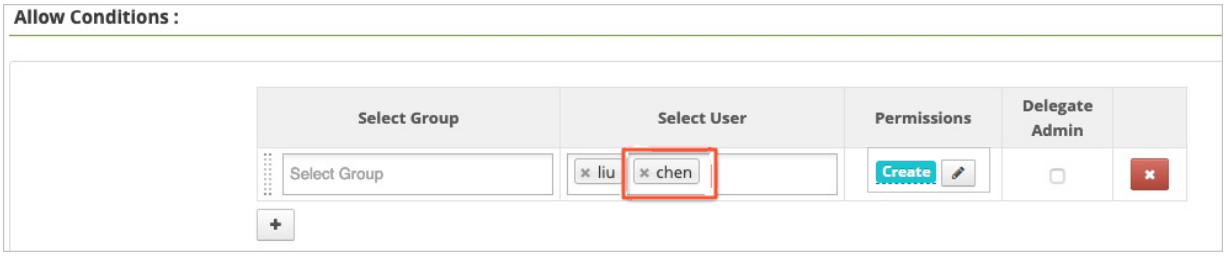
   添加Policy后，实现对liu用户的授权。liu用户即可以对testdb.test表的a列进行访问。

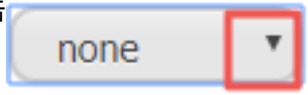
   > **说明:**
   > 添加、删除或修改Policy后，需要等待一分钟左右授权才能生效。

示例二：给用户chen添加创建hive表testdb.test的Create权限。

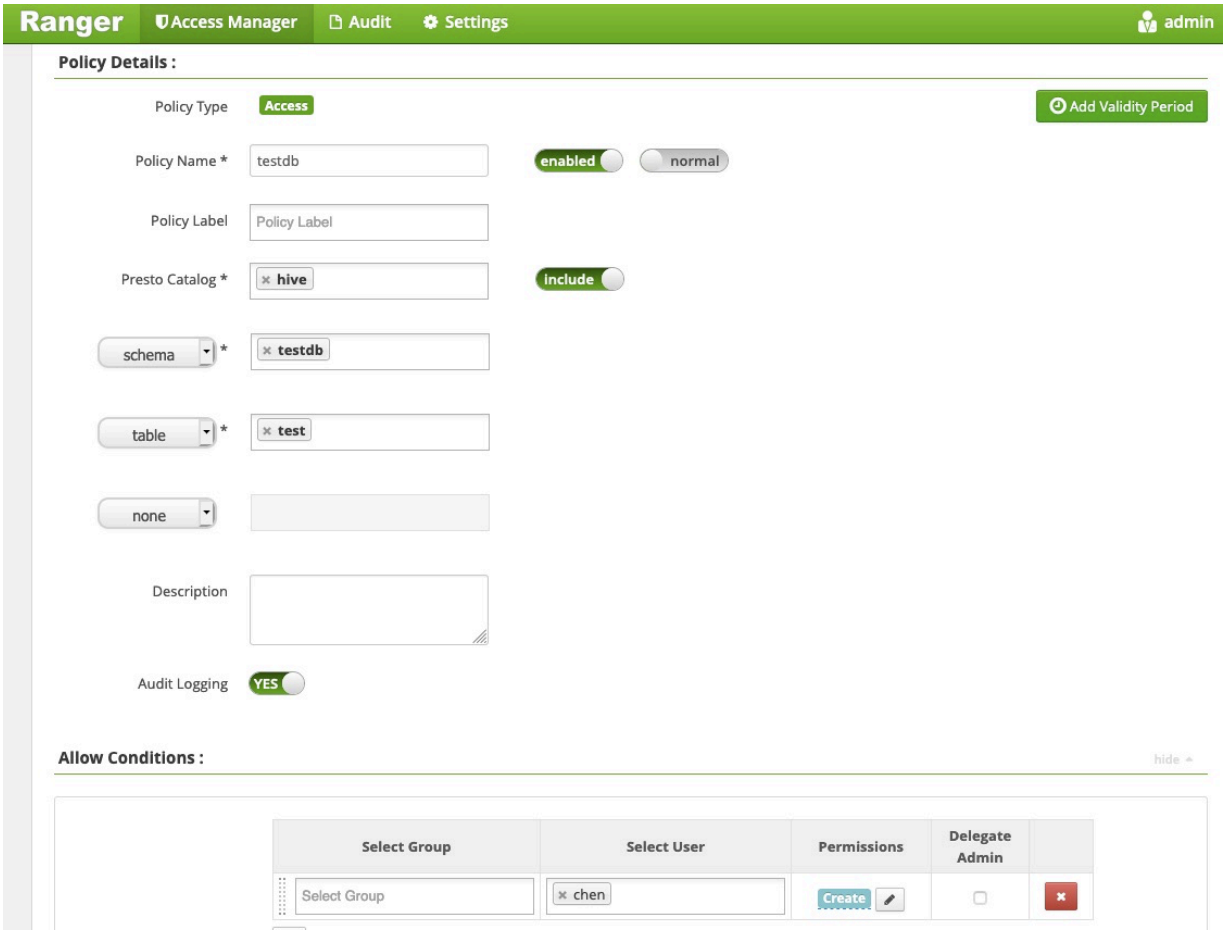
首先需要给用户chen添加Catalog的访问权限（此例中Catalog=hive），分为以下两种情况：

- 若已经添加过，则直接对该Policy进行编辑，在Select User中增加用户chen。
- 若还未对Catalog=hive添加过访问权限，则按照示例一中的方法进行添加，并给用户chen添加Create权限。



由于表的Create权限属于table层次，因此您需要配置权限到table层次。单击 

按钮，依次添加schema和table层次的内容。



Ranger Presto与Ranger Hive共享权限配置

上文介绍了Presto集成Ranger的方法以及Ranger Presto权限配置的方法，您可以独立使用Ranger Presto对Presto的权限进行控制。但是在部分场景下，Presto和Hive中配置的权限是一致的，如果Presto中也配置相同的权限会非常麻烦，因此EMR的Ranger提供了可以让Ranger Presto和Ranger Hive共享权限的方案，只需要在Ranger的Hive service中配置相关权限，Ranger Presto可以直接使用该权限配置进行用户的权限检查。

配置前需要关注以下事项：

- 已正确配置了Ranger Hive，详情请参见Hive配置。
- 使用Ranger Presto与Ranger Hive共享权限也需要在Ranger UI中添加Ranger Presto service。
- 与Ranger Presto一样，若需使用show schemas、show tables等命令，也需要在Ranger Hive service中配置用户对database=information_schema,table=*,column=*的Select权限。

使用及配置方法如下：

1. 登录集群header-1节点，修改配置文件/etc/ecm/presto-conf/ranger-presto-security.xml，将ranger.plugin.hive.authorization.enable修改为true。
2. 在左侧导航栏中，单击**集群服务** > **Presto**。

3. 单击右上角的**操作** > **重启 PrestoMaster**。

   PrestoMaster重启完毕之后，Ranger Presto便能够使用Ranger Hive service中设置的权限。

15.3 Ranger对接LDAP

15.3.1 Ranger usersync对接LDAP

如果您希望在配置Ranger的Policy时，能够对LDAP中的用户控制组件进行授权，可以使用usersync对接LDAP。

背景信息

Ranger usersync对接LDAP之后，在配置service的Policy时，可以选择的用户包含了LDAP中的用户（UNIX用户将看不到了，只能生效其中一个）。

配置方法

1. 打开install.properties文件。

```
cd /usr/lib/ranger-usersync-current
vim install.properties
```

2. 修改该文件中SYNC_SOURCE = ldap，之后修改该文件中如下几个配置项。

```
SYNC_LDAP_URL = ldap://emr-header-1:10389
SYNC_LDAP_BIND_DN = uid=admin,o=emr
SYNC_LDAP_BIND_PASSWORD = [password]
SYNC_LDAP_USER_SEARCH_BASE = ou=people,o=emr
```

   示例所给出的配置值均为对接EMR OpenLDAP，若要对接自建的LDAP，按照下文对各个参数的解释修改为自建LDAP对应值即可，有关各个配置项的详细解释，请参见Ranger usersync官方安装教程。

配置项	说明
SYNC_LDAP_URL	LDAP服务的地址。例如：ldap://ldap.example.com:389。
SYNC_LDAP_BIND_DN	连接LDAP进行用户和用户组查询的dn。例如：cn=ldapadmin,ou=users,dc=example,dc=com。
SYNC_LDAP_BIND_PASSWORD	用于连接的dn对应的密码。
EARCH_BASE	LDAP中用户搜索域。例如：ou=users,dc=example,dc=com。

进阶配置

经过如上配置之后，Ranger已经将LDAP中的用户同步过来了，但是这些用户只同步了用户名信息，并没有同步LDAP中的用户组信息。如果用户希望在Ranger中设置Policy对组件进行授权的时候，也能对LDAP中的用户组进行权限统一设置，则需要同步LDAP中用户组中的信息。Ranger同步LDAP用户组的配置较为复杂，并且不是所有的LDAP用户组信息都能进行同步（目前EMR的OpenLDAP不支持同步用户组信息，需要您手动同步），主要修改的配置如下所示。

```
SYNC_LDAP_USER_GROUP_NAME_ATTRIBUTE = gitNumber
SYNC_GROUP_SEARCH_ENABLED = true
SYNC_GROUP_USER_MAP_SYNC_ENABLED = true
SYNC_GROUP_SEARCH_BASE = ou=group,o=emr
SYNC_GROUP_OBJECT_CLASS = posixGroup
SYNC_GROUP_NAME_ATTRIBUTE = cn
SYNC_GROUP_MEMBER_ATTRIBUTE_NAME = memberUid
```

配置项	说明
SYNC_LDAP_USER_GROUP_NAME_ATTRIBUTE	用户entry中表示用户组的attribute的名称。例如：gitNumber(user objectClass=posixAccount)。
SYNC_GROUP_SEARCH_ENABLED	是否仅根据用户entry中记录的用户组attribute来确定用户组还是直接通过LDAP搜索所有用户组信息。true。
SYNC_GROUP_USER_MAP_SYNC_ENABLED	用户与用户组之间的映射关系是否通过LDAP搜索进行确定。例如：true。
SYNC_GROUP_SEARCH_BASE	LDAP中用户搜索域。例如：ou=groups,dc=example,dc=com。
SYNC_GROUP_OBJECT_CLASS	用户组的objectClass类型。例如：posixGroup。
SYNC_GROUP_NAME_ATTRIBUTE	用户组entry中用户组名的标识。例如：cn。
SYNC_GROUP_MEMBER_ATTRIBUTE_NAME	用户组entry中标识用户组成员的attribute名称。例如：memberUid。

生效配置

1. 配置好以上内容之后，需要在emr-header-1节点的/usr/lib/ranger-usersync-current路径下执行setup.sh。

   ```
   cd /usr/lib/ranger-usersync-current
   ```

```
sh setup.sh
```

2. 在EMR控制台Ranger集群服务中，重启RangerUserSync使得配置生效。

   a) 登录阿里云E-MapReduce控制台。

   b) 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。

   c) 单击上方的**集群管理**页签。

   d) 在左侧导航栏单击**集群服务** > **RANGER**。

   e) 在**组件列表**区域，单击**RangerUserSync**所在行的**重启**，使得配置生效。

15.3.2 Ranger admin与LDAP集成

如果您希望使用LDAP中的用户登录Ranger WebUI时，可以使用Ranger admin对接LDAP。

背景信息

Ranger的用户可以分为internal user和external user，像LDAP、UNIX这种用户都属于external user，这些用户不会影响internal user，只有管理员用户才可以添加service、修改Policy等操作，普通用户只能进行查看。已有的管理员用户admin，可以在Setting菜单中对用户权限进行配置，将普通用户升级成管理员用户。

配置方法

1. 打开install.properties文件。

```
cd /usr/lib/ranger-admin-current
vim install.properties
```

2. 修改authentication_method = LDAP，其中几处关键配置如下所示。

```
xa_ldap_url = ldap://emr-header-1:10389
xa_ldap_userDNpattern = uid={0},ou=people,o=emr
xa_ldap_base_dn = ou=people,o=emr
xa_ldap_bind_dn = uid=admin,o=emr
xa_ldap_bind_password = [password]
```

示例给出的配置值均为对接EMR OpenLDAP，若要对接自建的LDAP，按照下文将各个参数的解释修改为自建LDAP对应的值即可，有关各个配置项的详细解释，请参见Ranger admin官方安装教程。

配置项	说明
xa_ldap_url	LDAP服务的地址。例如：ldap://127.0.0.1:389。

配置项	说明
xa_ldap_userDNpattern	登录用户匹配LDAP dn的pattern。例如uid={0},ou=users,dc=example,dc=com表示当用户hadoop在WebUI登录时，其对应检查的LDAP dn为uid=hadoop,ou=users,dc=example,dc=com。
xa_ldap_base_dn	LDAP中用户搜索域。例如：dc=example,dc=com。
xa_ldap_bind_dn	连接LDAP并且进行用户和用户组查询的dn。例如：cn=ldapadmin,ou=users,dc=example,dc=com。
xa_ldap_bind_password	用于连接的dn对应的密码。

生效配置

1. 配置好以上内容之后，需要在emr-header-1节点的/usr/lib/ranger-admin-current路径下执行setup.sh。

```
cd /usr/lib/ranger-admin-current
sh setup.sh
```

2. 在EMR控制台Ranger集群服务中，重启RangerAdmin使得配置生效。

   a) 登录阿里云E-MapReduce控制台。

   b) 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。

   c) 单击上方的**集群管理**页签。

   d) 在**集群管理**页面，单击相应集群所在行的**详情**。

   e) 在左侧导航栏单击**集群服务** > **RANGER**。

   f) 在**组件列表**区域，单击**RangerAdmin**所在行的**重启**，使得配置生效。

15.4 Hive数据脱敏

Ranger支持对Hive数据的脱敏处理（Data Masking），它对select的返回结果进行脱敏处理，对用户屏蔽敏感信息。

背景信息

该功能只针对HiveServer2的场景（例如，beeline/jdbc/Hue等途径执行的select），对于使用Hive Client（例如hive -e 'select xxxx'）不支持。

Hive组件配置Ranger

请参见文档： Hive配置。

配置Data Mask Policy

在Ranger UI的emr-hive的service页面可以对用户访问Hive数据进行脱敏处理：

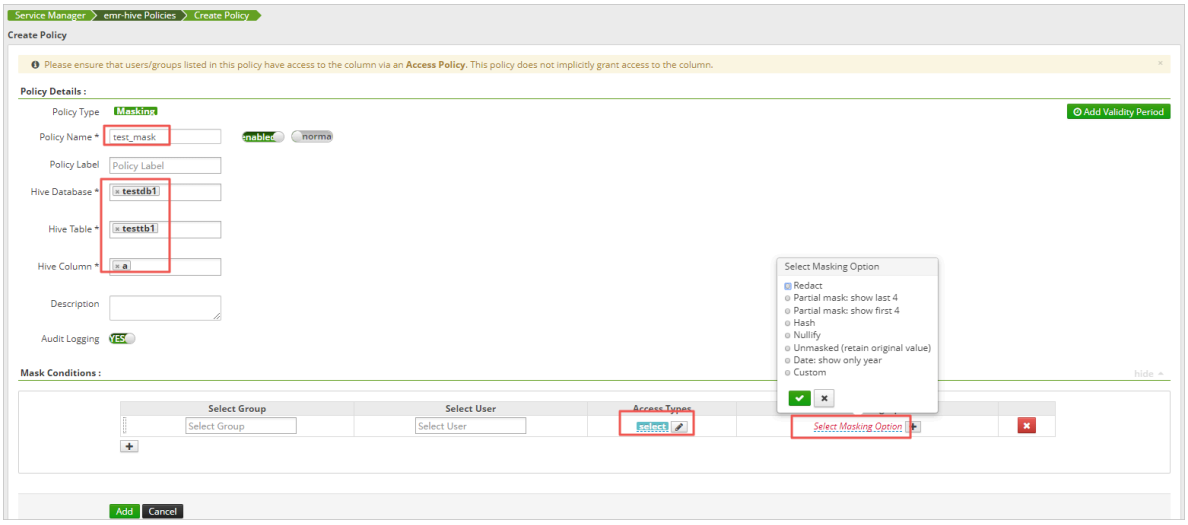
- 支持多种脱敏处理方式，例如显示开始的4个字符/显示最后的4个字符/Hash处理等。
- 配置Mask Policy时不支持通配符，例如policy中table/column不能配置*。
- 每个policy只能配置一个列的mask策略，多个列需要配置各自的mask policy。

配置Policy流程如下所示：

1. 单击已创建的**emr-hive**。

2. 单击上方的**Masking**。

3. 单击右上角的**Add New Policy**。
4. 配置相关参数。



参数	说明
Policy Name	Policy的名称。可自定义，例如test_mask。
database	添加Hive中的数据库，例如：testdb1。
table	添加表，例如：testtb1。
Hive Column	可添加列名。例如a。

参数	说明
Access Types	选择授予的权限。
Select Masking Option	选择脱敏方式。

5. 单击**add**。

测试数据脱敏

- 场景：

  用户 test在select表testdb1.testtbl中列a的数据时，只显示最开始的4个字符。

- 流程：

  1. 配置policy

     在上节的最后一个截图，其实就是配置了该场景的一个policy，可参考上图（其中脱敏方式选择了show first 4）。

  2. 脱敏验证

     test用户使用beeline连接HiveServer2，执行`select a from testdb1.testtbl`。

  如上图所示，test用户执行select命令后，列a显示的数据只有前面4个字符是正常显示，后面字符全部用x来脱敏处理。

16 组件授权

16.1 HDFS授权

HDFS开启了权限控制后，用户访问HDFS需要有合法的权限才能正常操作HDFS，如读取数据和创建文件夹等。

进入配置页面

1. 登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 在左侧导航栏单击**集群服务** > **HDFS**。
6. 单击**配置**页签。

> **说明:**
>
> - 对于Kerberos安全集群，已经默认设置了HDFS的权限（umask 设置为 027），无需配置和重启服务。
> - 对于非Kerberos安全集群需要添加配置并重启服务。

添加配置

HDFS权限相关的配置如下：

- **dfs.permissions.enabled**

  开启权限检查，即使该值为**false**， chmod/chgrp/chown/setfacl操作还是会进行权限检查。
- **dfs.datanode.data.dir.perm**

  datanode使用的本地文件夹路径的权限，默认**755**。

- **fs.permissions.umask-mode**
  - 权限掩码，在新建文件/文件夹的时候的默认权限设置。
  - 新建文件：0666 & ^umask。
  - 新建文件夹：0777 & ^umask。
  - 默认umask值为022，即新建文件权限为644（666&^022=644），新建文件夹权限为755（777&^022=755）。
  - EMR 的 Kerberos 安全集群默认设置为027，对应新建文件权限为640，新建文件夹权限为750。
- **dfs.namenode.acls.enabled**
  - 打开 ACL 控制，打开后除了可以对owner/group进行权限控制外，还可以对其它用户进行设置。
  - 设置 ACL 相关命令：

```
hadoop fs -getfacl [-R] <path>
hadoop fs -setfacl [-R] [-b |-k -m |-x <acl_spec> <path>] |[--set <acl_spec>   <path>]
```

如：

```
su test
 #test用户创建文件夹
 hadoop fs -mkdir /tmp/test
 #查看创建的文件夹的权限
 hadoop fs -ls /tmp
 drwxr-x---   - test   hadoop         0 2017-11-26 21:18 /tmp/test
 #设置acl，授权给foo用户rwx
 hadoop fs -setfacl -m user:foo:rwx /tmp/test
 #查看文件权限(+号表示设置了ACL)
 hadoop fs -ls  /tmp/
 drwxrwx---+  - test   hadoop         0 2017-11-26 21:18 /tmp/test
 #查看acl
  hadoop fs -getfacl  /tmp/test
  # file: /tmp/test
# owner: test
# group: hadoop
user::rwx
user:foo:rwx
group::r-x
mask::rwx
other::---
```

- **dfs.permissions.superusergroup**

  超级用户组，属于该组的用户都具有超级用户的权限。

重启HDFS服务

1. 在**集群服务** > **HDFS**页面，单击右上角的**操作** > **重启 All Components**。

2. 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

   单击右上角**查看操作历史**查看任务进度，等待任务完成。

其它

- umask值可以根据需求自行修改。
- HDFS是一个基础的服务，Hive/HBase等都是基于HDFS，所以在配置其它上层服务时，需要提前配置好HDFS的权限控制。
- 在HDFS开启权限后，需要设置好服务的日志路径（如Spark 的 /spark-history、YARN 的 /tmp/$user/ 等）。
- sticky bit

  针对文件夹可设置sticky bit，可以防止除了superuser/file owner/dir owner之外的其它用户删除该文件夹中的文件/文件夹（即使其它用户对该文件夹有rwx 权限）。

```
#即在第一位添加数字1
hadoop fs -chmod 1777 /tmp
hadoop fs -chmod 1777 /spark-history
hadoop fs -chmod 1777 /user/hive/warehouse
```

16.2 YARN授权

YARN的授权根据授权实体，可以分为服务级别的授权、队列级别的授权。

进入配置页面

1. 登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 在左侧导航栏单击**集群服务** > **YARN**。
6. 单击**配置**页签。

服务级别的授权

详见Hadoop官方文档。

- 控制特定用户访问集群服务，例如提交作业。
- 配置在hadoop-policy.xml。
- 服务级别的权限校验在其他权限校验之前（如 HDFS 的 permission 检查 /yarn 提交作业到队列控制）。

说明：

一般设置了 HDFS permission 检查 /yarn 队列资源控制，可以不设置服务级别的授权控制，用户可以根据自己需求进行相关配置。

队列级别的授权

YARN 可以通过队列对资源进行授权管理，有Capacity Scheduler和Fair Scheduler两种队列调度。

这里以Capacity Scheduler为例。

- 添加配置

  队列也有两个级别的授权，一个是提交作业到队列的授权，一个是管理队列的授权。

  说明：

  - 队列的ACL的控制对象为user/group，设置相关参数时，user和group可同时设置，中间用空格分开，user/group内部可用逗号分开，只有一个空格表示任何人都没有权限。
  - 队列ACL继承：如果一个user/group可以向某个队列中提交应用程序，则它可以向它的所有子队列中提交应用程序，同理管理队列的 ACL也具有继承性。所以如果要防止某个user/group提交作业到某个队列，则需要设置该队列以及该队列的所有父队列的ACL来限制该user/group的提交作业的权限。

  - **yarn.acl.enable**

    ACL开关，设置为**true**。

  - yarn.admin.acl

    ■ yarn的管理员设置，如可执行yarn rmadmin/yarn kill等命令，该值必须配置，否则后续的队列相关的acl管理员设置无法生效。

    ■ 如上备注，配置值时可以设置user/group。

    ```
    user1,user2 group1,group2 #user和group用空格隔开
     group1,group2 #只有group情况下，必须在最前面加上空格
    ```

    EMR集群中需将has配置为admin的ACL权限。

  - yarn.scheduler.capacity.${queue-name}.acl_submit_applications

    ■ 设置能够向该队列提交的user/group。

    ■ 其中 ${queue-name} 为队列的名称，可以是多级队列，注意多级情况下的ACL继承机制。

    ```
    #queue-name=root
     <property>
       <name>yarn.scheduler.capacity.root.acl_submit_applications</name>
       <value> </value> #空格表示任何人都无法往root队列提交作业
     </property>
    ```

```
#queue-name=root.testqueue
<property>
  <name>yarn.scheduler.capacity.root.testqueue.acl_submit_applications</
name>
    <value>test testgrp</value> #testqueue只允许test用户/testgrp组提交作业
 </property>
```

- yarn.scheduler.capacity.${queue-name}.acl_administer_queue
  - 设置某些user/group管理队列，例如kill队列中作业等。
  - queue-name可以是多级，注意多级情况下的ACL继承机制。

```
#queue-name=root
 <property>
    <name>yarn.scheduler.capacity.root.acl_administer_queue</name>
    <value> </value>
 </property>
#queue-name=root.testqueue
<property>
  <name>yarn.scheduler.capacity.root.testqueue.acl_administer_queue</name
>
    <value>test testgrp</value>
 </property>
```

- 重启YARN服务
  - 对于Kerberos安全集群已经默认开启ACL，用户可以根据自己需求配置队列的相关ACL权限控制。
  - 对于非Kerberos安全集群根据上述开启ACL并配置好队列的权限控制，重启YARN服务。

    1. 在**集群服务** > **YARN**页面，单击右上角的**操作** > **重启 All Components**。
    2. 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

       单击右上角**查看操作历史**查看任务进度，等待任务完成。
- 配置示例
  - yarn-site.xml

Key	Value
yarn.acl.enable	**true**
yarn.admin.acl	**has**

  - capacity-scheduler.xml
  - default队列：禁用default队列，不允许任何用户提交或管理。
  - q1队列：只允许test用户提交作业以及管理队列（如 kill）。
  - q2队列：只允许foo用户提交作业以及管理队列。

```
<configuration>
  <property>
    <name>yarn.scheduler.capacity.maximum-applications</name>
    <value>10000</value>
```

```
    <description>Maximum number of applications that can be pending and running
.</description>
  </property>
  <property>
    <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>
    <value>0.25</value>
    <description>Maximum percent of resources in the cluster which can be used to
run application masters i.e.
      controls number of concurrent running applications.
    </description>
  </property>
  <property>
    <name>yarn.scheduler.capacity.resource-calculator</name>
    <value>org.apache.hadoop.yarn.util.resource.DefaultResourceCalculator</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>default,q1,q2</value>
    <!-- 3个队列-->
    <description>The queues at the this level (root is the root queue).</description>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.default.capacity</name>
    <value>0</value>
    <description>Default queue target capacity.</description>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.default.user-limit-factor</name>
    <value>1</value>
    <description>Default queue user limit a percentage from 0.0 to 1.0.</description>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.default.maximum-capacity</name>
    <value>100</value>
    <description>The maximum capacity of the default queue.</description>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.default.state</name>
    <value>STOPPED</value>
    <!-- default队列状态设置为STOPPED-->
    <description>The state of the default queue. State can be one of RUNNING or
STOPPED.</description>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.default.acl_submit_applications</name>
    <value> </value>
    <!-- default队列禁止提交作业-->
    <description>The ACL of who can submit jobs to the default queue.</description>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.default.acl_administer_queue</name>
    <value> </value>
    <!-- 禁止管理default队列-->
    <description>The ACL of who can administer jobs on the default queue.</
description>
  </property>
  <property>
    <name>yarn.scheduler.capacity.node-locality-delay</name>
    <value>40</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.queue-mappings</name>
    <value>u:test:q1,u:foo:q2</value>
    <!-- 队列映射，test用户自动映射到q1队列-->
```

```
      <description>A list of mappings that will be used to assign jobs to queues. The
syntax for this list is
        [u|g]:[name]:[queue_name][,next mapping]* Typically this list will be used to
map users to queues,for
        example, u:%user:%user maps all users to queues with the same name as the
user.
      </description>
    </property>
    <property>
      <name>yarn.scheduler.capacity.queue-mappings-override.enable</name>
      <value>true</value>
      <!-- 上述queue-mappings设置的映射，是否覆盖客户端设置的队列参数-->
      <description>If a queue mapping is present, will it override the value specified by
the user? This can be used
        by administrators to place jobs in queues that are different than the one
specified by the user. The default
        is false.
      </description>
    </property>
    <property>
      <name>yarn.scheduler.capacity.root.acl_submit_applications</name>
      <value> </value>
      <!-- ACL继承性，父队列需控制住权限-->
      <description>
        The ACL of who can submit jobs to the root queue.
      </description>
    </property>
    <property>
      <name>yarn.scheduler.capacity.root.q1.acl_submit_applications</name>
      <value>test</value>
      <!-- q1只允许test用户提交作业-->
    </property>
    <property>
      <name>yarn.scheduler.capacity.root.q2.acl_submit_applications</name>
      <value>foo</value>
      <!-- q2只允许foo用户提交作业-->
    </property>
    <property>
      <name>yarn.scheduler.capacity.root.q1.maximum-capacity</name>
      <value>100</value>
    </property>
    <property>
      <name>yarn.scheduler.capacity.root.q2.maximum-capacity</name>
      <value>100</value>
    </property>
    <property>
      <name>yarn.scheduler.capacity.root.q1.capacity</name>
      <value>50</value>
    </property>
    <property>
      <name>yarn.scheduler.capacity.root.q2.capacity</name>
      <value>50</value>
    </property>
    <property>
      <name>yarn.scheduler.capacity.root.acl_administer_queue</name>
      <value> </value>
      <!-- ACL继承性，父队列需控制住权限-->
    </property>
    <property>
      <name>yarn.scheduler.capacity.root.q1.acl_administer_queue</name>
      <value>test</value>
      <!-- q1队列只允许test用户管理，如kill作业-->
    </property>
    <property>
```

```
    <name>yarn.scheduler.capacity.root.q2.acl_administer_queue</name>
    <value>foo</value>
    <!-- q2队列只允许foo用户管理，如kill作业-->
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.q1.state</name>
    <value>RUNNING</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.q2.state</name>
    <value>RUNNING</value>
  </property>
</configuration>
```

16.3 Hive授权

Hive内置有基于底层HDFS的权限（Storage Based Authorization）和基于标准SQL的grant等命令（SQL Standards Based Authorization）两种授权机制。

背景信息

场景：

如果集群的用户直接通过HDFS/Hive Client访问Hive的数据，需要对Hive在HDFS中的数据进行相关的权限控制，通过HDFS权限控制，进而可以控制Hive SQL相关的操作权限。详见Hive官方文档。

说明：

两种授权机制可以同时配置，不冲突。

Storage Based Authorization （针对HiveMetaStore）。

进入配置页面

1. 登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 在左侧导航栏单击**集群服务** > **Hive**。
6. 单击**配置**页签。

添加配置

1. 在**服务配置**区域，单击**hive-site**。

2. 单击右侧的**自定义配置**，设置以下配置。

Key	Value
hive.metastore.pre.event.listeners	org.apache.hadoop.hive.ql.security.authorization.Aut
hive.security.metastore.authorization.manager	org.apache.hadoop.hive.ql.security.authorization.Sto
hive.security.metastore.authenticator.manager	org.apache.hadoop.hive.ql.security.HadoopDefaultMe

重启Hive MetaStore

1. 在**集群服务** > **Hive**页面，单击右上角的**操作** > **重启 Hive MetaStore**。
2. 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

   单击右上角**查看操作历史**查看任务进度，等待任务完成。

HDFS权限控制

EMR的Kerberos安全集群已经设置了Hive的 warehouse的HDFS相关权限。

对于非Kerberos安全集群，用户需要做如下步骤设置hive基本的HDFS权限：

- 打开HDFS的权限。
- 配置Hive的warehouse权限。

```
hadoop fs -chmod 1771 /user/hive/warehouse
也可以设置成，1 表示 stick bit（不能删除别人创建的文件/文件夹）
hadoop fs -chmod 1777 /user/hive/warehouse
```

有了上述设置基础权限后，可以通过对warehouse文件夹授权，让相关用户/用户组能够正常创建表/读写表。

```
sudo su has
    #授予test对warehouse文件夹rwx权限
    hadoop fs -setfacl -m user:test:rwx /user/hive/warehouse
    #授予hivegrp对warehouse文件夹rwx权限
    hadoo fs -setfacl -m group:hivegrp:rwx /user/hive/warehouse
```

经过HDFS的授权后，让相关用户/用户组能够正常创建表/读写表等，而且不同的账号创建的hive表在HDFS中的数据只能自己的账号才能访问。

验证

- test用户建表testtbl。

```
hive> create table testtbl(a string);
FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask
. MetaException(message:Got exception: org.apache.hadoop.security.AccessCont
rolException Permission denied: user=test, access=WRITE, inode="/user/hive/
warehouse/testtbl":hadoop:hadoop:drwxrwx--t
at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.check(FSPermissi
onChecker.java:320)
```

```
at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.check(FSPermissi
onChecker.java:292)
```

上面显示错误没有权限，需要给test用户添加权限。

```
#从root账号切到has账号
su has
#给test账号添加acl，对warehouse目录的rwx权限
 hadoop fs -setfacl -m user:test:rwx /user/hive/warehouse
```

test账号再创建database。

```
hive> create table testtbl(a string);
OK
Time taken: 1.371 seconds
#查看hdfs中testtbl的目录，从权限可以看出test用户创建的表数据只有test和hadoop组可以
读取，其他用户没有任何权限
hadoop fs -ls /user/hive/warehouse
drwxr-x---   - test hadoop          0 2017-11-25 14:51 /user/hive/warehouse/testtbl
#插入一条数据
hive>insert into table testtbl select "hz"
```

- foo用户访问testtbl。

```
#drop table
hive> drop table testtbl;
FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask
. MetaException(message:Permission denied: user=foo, access=READ, inode="/user/
hive/warehouse/testtbl":test:hadoop:drwxr-x---
    at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.check(
FSPermissionChecker.java:320)
    at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.checkPermi
ssion(FSPermissionChecker.java:219)
    at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.checkPermi
ssion(FSPermissionChecker.java:190)
#alter table
hive> alter table testtbl add columns(b string);
FAILED: SemanticException Unable to fetch table testtbl. java.security.AccessCont
rolException: Permission denied: user=foo, access=READ, inode="/user/hive/
warehouse/testtbl":test:hadoop:drwxr-x---
    at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.check(
FSPermissionChecker.java:320)
    at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.checkPermi
ssion(FSPermissionChecker.java:219)
    at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.checkPermi
ssion(FSPermissionChecker.java:190)
    at org.apache.hadoop.hdfs.server.namenode.FSDirectory.checkPermission(
FSDirectory.java:1720)
#select
hive> select * from testtbl;
FAILED: SemanticException Unable to fetch table testtbl. java.security.AccessCont
rolException: Permission denied: user=foo, access=READ, inode="/user/hive/
warehouse/testtbl":test:hadoop:drwxr-x---
    at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.check(
FSPermissionChecker.java:320)
```

```
    at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.checkPermi
ssion(FSPermissionChecker.java:219)
```

可见foo用户不能对test用户创建的表做任何的操作，如果想要授权给foo，需要通过HDFS的授权来实现。

```
su has
#只授权读的权限，也可以根据情况授权写权限（比如alter）
#备注: -R 将testtbl文件夹下的文件也设置可读
hadoop fs -setfacl -R -m user:foo:r-x /user/hive/warehouse/testtbl
#可以select成功
hive> select * from testtbl;
OK
hz
Time taken: 2.134 seconds, Fetched: 1 row(s)
```

说明:

一般可以根据需求新建一个hive用户的group，然后通过给group授权，后续将新用户添加到group中，同一个group的数据权限都可以访问。

SQL Standards Based Authorization

- 场景

  如果集群的用户不能直接通过hdfs/hive client访问，只能通过 HiveServer（beeline/jdbc） 等来执行hive相关的命令，可以使用SQL Standards Based Authorization的授权方式。

  如果用户能够使用hive shell等方式，即使做下面的一些设置操作，只要用户客户端的hive-site.xml没有相关配置，都可以正常访问hive。

  详见Hive文档。

- 添加配置

  配置是提供给HiveServer。

  1. 在**服务配置**区域，单击**hive-site**。
  2. 单击右侧的**自定义配置**，设置以下配置。

Key	Value
hive.security.authorization.enabled	**true**
hive.users.in.admin.role	**hive**
hive.security.authorization.createtable.owner.grants	**ALL**

- 重启HiveServer2

  1. 在**集群服务** > **Hive**页面，单击**操作** > **重启 HiveServer2**。
  2. 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

     单击右上角**查看操作历史**查看任务进度，等待任务完成。

- 权限操作命令

  具体命令操作详见Hive 文档。

- 验证

  - 用户foo通过beeline访问test用户的testtbl表。

    ```
    2: jdbc:hive2://emr-header-1.cluster-xxx:10> select * from testtbl;
    Error: Error while compiling statement: FAILED: HiveAccessControlException
    Permission denied: Principal [name=foo, type=USER] does not have following
    privileges for operation QUERY [[SELECT] on Object [type=TABLE_OR_VIEW, name=
    default.testtbl]] (state=42000,code=40000)
    ```

  - grant权限。

    ```
    切到 test 账号执行 grant 给 foo 授权 select 操作
    hive> grant select on table testtbl to user foo;
     OK
     Time taken: 1.205 seconds
    ```

  - foo可以正常select。

    ```
    0: jdbc:hive2://emr-header-1.cluster-xxxxx:10> select * from testtbl;
    INFO  : OK
    +------------+--+
    | testtbl.a  |
    +------------+--+
    | hz         |
    +------------+--+
    1 row selected (0.787 seconds)
    ```

  - 回收权限。

    ```
    切换到 test 账号，回收权限 foo 的 select 权限
      hive> revoke select from user foo;
        OK
        Time taken: 1.094 seconds
    ```

  - foo无法select testtbl的数据。

    ```
    0: jdbc:hive2://emr-header-1.cluster-xxxxx:10> select * from testtbl;
    Error: Error while compiling statement: FAILED: HiveAccessControlException
    Permission denied: Principal [name=foo, type=USER] does not have following
    ```

```
privileges for operation QUERY [[SELECT] on Object [type=TABLE_OR_VIEW, name=
default.testtbl]] (state=42000,code=40000)
```

16.4 HBase授权

HBase在不开启授权的情况下，任何账号对HBase集群可以进行任何操作，例如disable table、drop table、major compact等。

背景信息

对于没有Kerberos认证的集群，即使开启了HBase授权，用户也可以伪造身份访问集群服务。所以建议创建高安全模式（即支持 Kerberos）的集群，详情请参见Kerberos简介。

进入配置页面

1. 登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 在左侧导航栏单击**集群服务** > **HBase**。
6. 单击**配置**页签。

添加配置

1. 在**服务配置**区域，单击**hbase-site**。
2. 单击右侧的**自定义配置**，设置以下配置。

Key	Value
hbase.security.authorization	true
hbase.coprocessor.master.classes	org.apache.hadoop.hbase.security.access.AccessCont
hbase.coprocessor.region.classes	org.apache.hadoop.hbase.security.token.TokenProvid
hbase.coprocessor.regionserver.classes	org.apache.hadoop.hbase.security.access.AccessCont

重启HBase集群

1. 在**集群服务** > **HBase**页面，单击右上角的**操作** > **重启 All Components**。
2. 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

   单击右上角**查看操作历史**查看任务进度，等待任务完成。

授权（ACL）

- 基本概念

  授权就是将对 [某个范围的资源] 的 [操作权限] 授予[某个实体]。

  在 HBase 中，上述对应的三个概念分别为：

  - 某个范围（Scope）的资源

名称	描述
Superuser	超级账号可以进行任何操作，运行HBase服务的账号默认是 Superuser。也可以通过在hbase-site.xml中配置hbase.superuser的值可以添加超级账号。
Global	Global Scope拥有集群所有table的Admin权限。
Namespace	在Namespace Scope进行相关权限控制。
Table	在Table Scope进行相关权限控制。
ColumnFamily	在ColumnFamily Scope进行相关权限控制。
Cell	在Cell Scope进行相关权限控制。

  - 操作权限

名称	描述
Read（R）	读取某个Scope资源的数据。
Write （W）	写数据到某个Scope的资源。
Execute （X）	在某个Scope执行协处理器。
Create （C）	在某个Scope创建或删除表等操作。
Admin（A）	在某个Scope进行集群相关操作，如balance、assign等。

  - 某个实体

名称	描述
User	对某个用户授权。
Group	对某个用户组授权。

- 授权命令
  - grant授权

    ```
    grant <user> <permissions> [<@namespace> [<table> [<column family> [<column qualifier>]]]
    ```

    ■ user和group的授权方式一样，但group需要加一个前缀@

    ```
    grant 'test','R','tbl1'  #给用户test授予表tbl1的读权限。
    grant '@testgrp','R','tbl1' #给用户组testgrp授予表tbl1的读权限。
    ```

    ■ namespace需要加一个前缀@

    ```
    grant 'test','C','@ns_1'  #给用户test授予namespace ns_1的CREATE权限。
    ```

  - revoke回收

    ```
    revoke 'trafodion'  #回收trafodion用户的所有权限。
    ```

  - user_permission查看权限

    ```
    user_permission 'TABLE_A'  #查看TABLE_A表的所有权限。
    ```

16.5 Kafka授权

如果没有开启Kafka认证（如 Kerberos 认证或者简单的用户名密码），即使开启了Kafka授权，用户也可以伪造身份访问服务。所以建议创建高安全模式，即支持Kerberos的Kafka集群。

背景信息

本文的权限配置只针对E-MapReduce的高安全模式集群，即Kafka以Kerberos的方式启动，详见Kerberos简介。

进入配置页面

1. 登录阿里云E-MapReduce控制台。
2. 在顶部菜单栏处，根据实际情况选择地域（Region）和资源组。
3. 单击上方的**集群管理**页签。
4. 在**集群管理**页面，单击相应集群所在行的**详情**。
5. 在左侧导航栏单击**集群服务** > **Kafka**。
6. 单击**配置**页签。

添加配置

1. 在**服务配置**区域，单击**server.properties**。

2. 单击**自定义配置**，添加如下几个参数。

key	value	备注
authorizer.class.name	**kafka.security.auth.SimpleAclAuthorizer**	固定。
super.users	**User:kafka**	User:kafka是必须的，可添加其它用户用分号（;）隔开。

说明：

zookeeper.set.acl用来设置Kafka在ZooKeeper中数据的权限，E-MapReduce集群中已经设置为true，所以此处不需要再添加该配置。该配置设置为true后，在Kerberos环境中，只有用户名称为Kafka且通过Kerberos 认证后才能执行kafka-topics.sh命令（kafka-topics.sh会直接读写/修改ZooKeeper中的数据）。

重启Kafka服务

1. 在**集群服务** > **Kafka**页面，单击右上角的**操作** > **重启 All Components**。
2. 在**执行集群操作**对话框设置相关参数，然后单击**确定**。

   单击右上角**查看操作历史**查看任务进度，等待任务完成。

授权（ACL）

- 基本概念

  Kafka官方文档定义。

  > Kafka acls are defined in the general format of "Principal P is [Allowed/Denied] Operation O From Host H On Resource R"

  即ACL过程涉及Principal、Allowed/Denied、Operation、Host 和 Resource。

  - Principal：用户名

安全协议	value
PLAINTEXT	ANONYMOUS
SSL	ANONYMOUS
SASL_PLAINTEXT	mechanism 为 PLAIN 时，用户名是 client_jaas.conf 指定的用户名，mechanism 为 GSSAPI 时，用户名为 client_jaas.conf 指定的 principal

安全协议	value
SASL_SSL	-

- Allowed/Denied：允许/拒绝。
- Operation： 操作，包括 Read、Write、Create、DeleteAlter、Describe、ClusterAction、AlterConfigs、DescribeConfigs、IdempotentWrite和All。
- Host： 针对的机器。
- Resource： 权限作用的资源对象，包括 Topic、Group、Cluster 和 TransactionalId。

Operation/Resource的一些详细对应关系，如哪些Resource支持哪些Operation的授权，详见KIP-11 - Authorization Interface。

- 授权命令

  我们使用脚本kafka-acls.sh （/usr/lib/kafka-current/bin/kafka-acls.sh） 进行Kafka授权，您可以直接执行 kafka-acls.sh --help查看如何使用该命令。

操作示例

在已经创建的E-MapReduce高安全Kafka集群的master节点上进行相关示例操作。

1. 新建用户test，执行以下命令。

   useradd test

2. 创建topic。

   第一节添加配置的备注中提到zookeeper.set.acl=true，kafka-topics.sh需要在kafka账号下执行，而且kafka账号下要通过Kerberos认证。

   ```
   # kafka_client_jaas.conf中已经设置了kafka的Kerberos认证相关信息
   export KAFKA_HEAP_OPTS="-Djava.security.auth.login.config=/etc/ecm/kafka-conf/kafka_client_jaas.conf"
   # zookeeper地址改成自己集群的对应地址（执行`hostnamed`后即可获取）
   kafka-topics.sh --create --zookeeper emr-header-1:2181/kafka-1.0.0 --replication-factor 3 --partitions 1 --topic test
   ```

3. test用户执行kafka-console-producer.sh。

   a) 创建test用户的keytab 文件，用于 zookeeper/kafka的认证。

   ```
   su root
   sh /usr/lib/has-current/bin/hadmin-local.sh /etc/ecm/has-conf -k /etc/ecm/has-conf/admin.keytab
   HadminLocalTool.local: #直接按回车可以看到一些命令的用法
   HadminLocalTool.local: addprinc #输入命令按回车可以看到具体命令的用法
   HadminLocalTool.local: addprinc -pw 123456 test #添加test的principal，密码设置为123456
   ```

```
HadminLocalTool.local: ktadd -k /home/test/test.keytab test #导出keytab文件，后续可使用该文件
```

b) 添加kafka_client_test.conf。

如文件放到 /home/test/kafka_client_test.conf，内容如下。

```
KafkaClient {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
storeKey=true
serviceName="kafka"
keyTab="/home/test/test.keytab"
principal="test";
};
// Zookeeper client authentication
Client {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
useTicketCache=false
serviceName="zookeeper"
keyTab="/home/test/test.keytab"
principal="test";
};
```

c) 添加producer.conf。

如文件放到`/home/test/producer.conf`，内容如下。

```
security.protocol=SASL_PLAINTEXT
sasl.mechanism=GSSAPI
```

d) 执行kafka-console-producer.sh。

```
su test
export KAFKA_HEAP_OPTS="-Djava.security.auth.login.config=/home/test/kafka_client_test.conf"
kafka-console-producer.sh --producer.config /home/test/producer.conf --topic test --broker-list emr-worker-1:9092
```

由于没有设置ACL，所以上述会报错。

```
org.apache.kafka.common.errors.TopicAuthorizationException: Not authorized to access topics: [test]
```

e) 设置 ACL。

同样`kafka-acls.sh`也需要kafka账号执行。

```
su kafka
export KAFKA_HEAP_OPTS="-Djava.security.auth.login.config=/etc/ecm/kafka-conf/kafka_client_jaas.conf"
kafka-acls.sh --authorizer-properties zookeeper.connect=emr-header-1:2181/kafka-1.0.0 --add --allow-principal User:test --operation Write --topic test
```

f) 再执行kafka-console-producer.sh。

```
su test
```

```
export KAFKA_HEAP_OPTS="-Djava.security.auth.login.config=/home/test/
kafka_client_test.conf"
kafka-console-producer.sh --producer.config /home/test/producer.conf --topic
test --broker-list emr-worker-1:9092
```

正常情况下显示如下。

```
[2018-02-28 22:25:36,178] INFO Kafka commitId : aaa7af6d4a11b29d (org.apache.
kafka.common.utils.AppInfoParser)
>alibaba
>E-MapReduce
>
```

4. est用户执行kafka-console-consumer.sh。

上面成功执行kafka-console-producer.sh，并往 topic 里面写入一些数据后，就可以执行kafka-console-consumer.sh进行消费测试。

a) 添加consumer.conf。

如文件放到/home/test/consumer.conf，内容如下。

```
security.protocol=SASL_PLAINTEXT
sasl.mechanism=GSSAPI
```

b) 执行kafka-console-consumer.sh。

```
su test
#kafka_client_test.conf跟上面producer使用的是一样的
export KAFKA_HEAP_OPTS="-Djava.security.auth.login.config=/home/test/
kafka_client_test.conf"
kafka-console-consumer.sh --consumer.config consumer.conf  --topic test --
bootstrap-server emr-worker-1:9092 --group test-group  --from-beginning
```

由于未设置权限，会报以下错误。

```
org.apache.kafka.common.errors.GroupAuthorizationException: Not authorized to
access group: test-group
```

c) 设置ACL。

```
su kafka
export KAFKA_HEAP_OPTS="-Djava.security.auth.login.config=/etc/ecm/kafka-conf
/kafka_client_jaas.conf"
# test-group权限
kafka-acls.sh --authorizer-properties zookeeper.connect=emr-header-1:2181/
kafka-1.0.0 --add --allow-principal User:test --operation Read --group test-group
# topic权限
kafka-acls.sh --authorizer-properties zookeeper.connect=emr-header-1:2181/
kafka-1.0.0 --add --allow-principal User:test --operation Read --topic test
```

d) 再执行kafka-console-consumer.sh。

```
su test
# kafka_client_test.conf跟上面producer使用的是一样的
export KAFKA_HEAP_OPTS="-Djava.security.auth.login.config=/home/test/
kafka_client_test.conf"
```

```
kafka-console-consumer.sh --consumer.config consumer.conf  --topic test --
bootstrap-server emr-worker-1:9092 --group test-group  --from-beginning
```

正常输出。

```
alibaba
E-MapReduce
```

17 Kerberos认证

17.1 Kerberos简介

E-MapReduce从2.7.x/3.5.x版本开始支持创建安全类型的集群，支持的集群类型包括Hadoop、Kafka、Druid和Flink。高安全类型集群中的开源组件以Kerberos的安全模式启动，在这种安全环境下只有经过认证的客户端（Client）才能访问集群的服务（Service，如 HDFS）。

背景信息

对于客户端而言，集群开启Kerberos之后，可以对可信任的客户端提供认证，使得可信任客户端能够正确提交作业，恶意用户无法伪装成其他用户侵入到集群当中，能够有效防止恶意冒充客户端提交作业的情况。对于服务端而言，集群开启Kerberos之后，集群中的服务都是可以信任的，集群服务之间使用密钥进行通信，避免了冒充服务的情况。

开启Kerberos能够提升集群的安全性，但是也会提升用户使用集群的复杂度，提交作业的方式与没有开启Kerberos前会有一些区别，需要对作业进行改造，增加Kerberos认证的相关内容。开启Kerberos前需要用户对Kerberos的原理、使用有一些了解，才能更好地使用Kerberos。此外，由于集群服务间的通信加入了Kerberos认证机制，认证过程会有一些轻微的时间消耗，相同作业相较于没有开启Kerberos的同规格集群执行速度会有一些下降。

开启Kerberos

创建安全集群：需要在集群创建时，在**软件配置**页面的**高级设置**中打开**Kerberos集群模式**即可。

Kerberos组件列表

目前E-MapReduce版本中支持的Kerberos的组件列表如下所示。

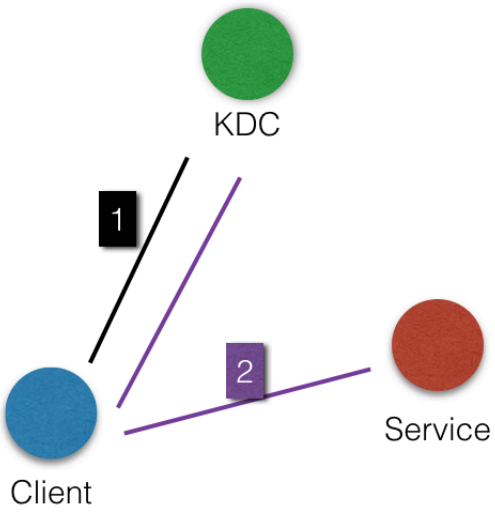
组件名称	组件版本
HDFS	2.8.5及以上
YARN	2.8.5及以上
Spark	2.4.3

组件名称	组件版本
Hive	2.3.5及以上
Tez	0.9.1及以上
Zookeeper	3.5.5及以上
Hue	4.4.0
Zeppelin	0.8.1
Oozie	5.1.0
Sqoop	1.4.7
HBase	1.4.9及以上
Phoenix	4.14.1及以上
Druid	0.13.0及以上
Flink	1.5.6及以上
Impala	2.12.2及以上
Kafka	2.11/1.1.1及以上
Presto	prestodb 0.213及以上/prestosql 310
Ranger	1.0.0及以上
Storm	1.2.2

Kerberos身份认证原理

Kerberos是一种基于对称密钥技术的身份认证协议，它作为一个独立的第三方的身份认证服务，可以为其它服务提供身份认证功能，且支持SSO（即客户端身份认证后，可以访问多个服务如HBase/HDFS等）。

Kerberos协议过程主要有两个阶段，第一个阶段是KDC对Client身份认证，第二个阶段是Service对Client身份认证。



- KDC：Kerberos的服务端程序。
- Client：需要访问服务的用户（principal），KDC和Service会对用户的身份进行认证。
- Service：集成了Kerberos的服务，如HDFS/YARN/HBase等。
- KDC对Client身份认证

  当客户端用户（principal）访问一个集成了Kerberos的服务之前，需要先通过KDC的身份认证。

  若身份认证通过，则客户端会获取到一个TGT（Ticket Granting Ticket），后续就可以使用该TGT去访问集成了Kerberos的服务。
- Service对Client身份认证

  当用户获取TGT后，就可以继续访问Service服务。它会使用TGT以及需要访问的服务名称（如HDFS）去KDC获取SGT（Service Granting Ticket），然后使用SGT去访问 Service，Service会利用相关信息对Client进行身份认证，认证通过后就可以正常访问Service服务。

Kerberos实践

E-MapReduce的Kerberos安全集群中的服务，在创建集群的时候会以Kerberos安全模式启动。

- Kerberos服务端程序为HASServer
  - 在集群管理页面，单击待操作集群所在行的**详情**，然后在**集群服务** > **Has**页面，可以执行查看/修改配置/重启等操作。
  - 非HA集群部署在emr-header-1，HA集群部署在emr-header-1/emr-header-2两个节点。

- 支持四种身份认证方式

  HASServer可同时支持以下4种身份认证方式，客户端可以通过配置相关参数来指定HASServer使用哪种方式进行身份认证。

  - 兼容MIT Kerberos的身份认证方式

    客户端配置：

    ■ 如果在集群的某个节点上执行客户端命令，则需要将/etc/ecm/hadoop-conf/core-site.xml中的**hadoop.security.authentication.use.has**设置为**false**。

    ■ 如果通过控制台的执行计划运行作业，则不能修改master节点上面/etc/ecm/hadoop-conf/core-site.xml中的值，否则执行计划的作业认证就会失败，可以使用下面的方式 `export HADOOP_CONF_DIR=/etc/has/hadoop-conf`临时export环境变量，将该路径下的**hadoop.security.authentication.use.has**设置为**false**。

    访问方式：Service的客户端包完全可使用开源的，如HDFS客户端等。详情请参见兼容 MIT Kerberos 认证。

  - RAM身份认证

    客户端配置：

    ■ 如果在集群的某个节点上执行客户端命令，则需要将/etc/ecm/hadoop-conf/core-site.xml中的**hadoop.security.authentication.use.has**设置为**false**，/etc/has/has-client.conf中的**auth_type**设置为**RAM**。

    ■ 如果有通过控制台的执行计划运行作业，则不能修 master 节点上面/etc/ecm/hadoop-conf/core-site.xml以及/etc/has/has-client.conf中的值，否则执行计划的作业认证就会失败，可以使用下面的方式`export HADOOP_CONF_DIR=/etc/has/hadoop-conf`；`export HAS_CONF_DIR=/path/to/has-client.conf`临时export环境变量，其中HAS_CONF_DIR文件夹下的has-client.conf的**auth_type**设置为**RAM**。

    访问方式： 客户端需要使用集群中的软件包（如Hadoop/HBase等），详情请参见RAM认证。

  - LDAP身份认证

    客户端配置：

    ■ 如果在集群的某个节点上执行客户端命令，则需要将/etc/ecm/hadoop-conf/core-site.xml中**hadoop.security.authentication.use.has**设置为**true**，/etc/has/has-client.conf中**auth_type**设置为**LDAP**。

    ■ 如果有通过控制台的执行计划运行作业，则不能修改master节点上面/etc/ecm/hadoop-conf/core-site.xml以及 /etc/has/has-client.conf中的值，否则执行计划的作业认

证就会失败，可以使用下面的方式export HADOOP_CONF_DIR=/etc/has/hadoop-conf；export HAS_CONF_DIR=/path/to/has-client.conf临时export环境变量，其中HAS_CONF_DIR文件夹下的has-client.conf的**auth_type**设置为**LDAP**。

访问方式：客户端需要使用集群中的软件包（如Hadoop/HBase等），详情请参见LDAP 认证。

- 执行计划认证

  如果用户有使用E-MapReduce控制台的执行计划提交作业，则emr-header-1节点的配置必须不能被修改（默认配置）。

  客户端配置：

  emr-header-1上面的/etc/ecm/hadoop-conf/core-site.xml中**hadoop.security.authentication.use.has**设置为**true**，/etc/has/has-client.conf中**auth_type**设置为**EMR**。

  访问方式：跟非Kerberos安全集群使用方式一致，详情请参见执行计划认证。

- 其他

  登录master节点访问集群

  集群管理员也可以登录master节点访问集群服务，登录master节点切换到has账号（默认使用兼容MIT Kerberos的方式）即可访问集群服务，方便做一些排查问题或者运维等。

```
>sudo su has
>hadoop fs -ls /
```

> **说明：**
> 也可以登录其他账号操作集群，前提是该账号可以通过Kerberos认证。另外，如果在master节点上需要使用兼容MITKerberos的方式，需要在该账号下先export一个环境变量export HADOOP_CONF_DIR=/etc/has/hadoop-conf/。

17.2 兼容 MIT Kerberos 认证

本文将通过 HDFS 服务介绍兼容 MIT Kerberos 认证流程。

兼容 MIT Kerberos 的身份认证方式

EMR 集群中 Kerberos 服务端在 master 节点上启动，涉及一些管理操作需在 master 节点（emr-header-1）的 root 账号执行。

下面以 test 用户访问 HDFS 服务为例介绍相关流程。

- Gateway 上执行hadoop fs -ls /
  - 配置 krb5.conf

```
Gateway 上面使用root账号
 scp root@emr-header-1:/etc/krb5.conf /etc/
```

  - 添加 principal
    - 登录集群 emr-header-1 节点（必须是 header-1，HA 不能在 header-2 上操作），切换到 root 账号。
    - 进入 Kerberos 的 admin 工具。

```
 sh /usr/lib/has-current/bin/hadmin-local.sh /etc/ecm/has-conf -k /etc/ecm/has-conf/admin.keytab
 HadminLocalTool.local: #直接按回车可以看到一些命令的用法
 HadminLocalTool.local: addprinc #输入命令按回车可以看到具体命令的用法
 HadminLocalTool.local: addprinc -pw 123456 test #添加test的princippal，密码设置为123456
```

  - 导出 keytab 文件

    登录集群 emr-header-1（必须是 header-1，HA 不能在header-2操作），导入 keytab 文件。

    使用 Kerberos 的 admin 工具可以导出 principal 对应的 keytab 文件。

```
HadminLocalTool.local: ktadd -k /root/test.keytab test #导出keytab文件，后续可使用该文件
```

  - kinit 获取 Ticket

    在执行 hdfs 命令的客户端机器上面，如 Gateway。
    - 添加 linux 账号 test

```
useradd test
```

    - 安装 MITKerberos 客户端工具。

      可以使用 MITKerberos tools 进行相关操作（如 kinit/klist 等），详情请参见 MITKerberos 文档。

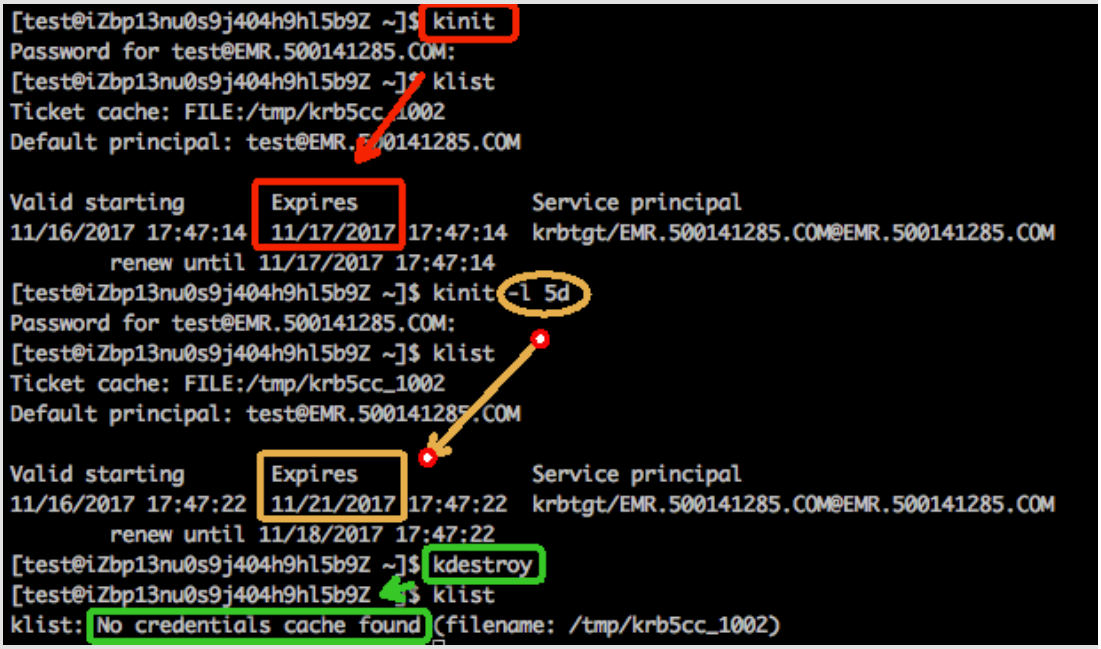
      yum install krb5-libs krb5-workstation -y
    - 切到 test 账号执行 kinit

```
su test
#如果没有 keytab文件，则执行
kinit #直接回车
Password for test: 123456 #即可
#如有有keytab文件，也可执行
kinit -kt test.keytab test
#查看ticket
```

```
klist
```

说明：

MITKerberos 工具使用实例



- 导入环境变量生效，在Gateway节点上执行以下命令。

```
export HADOOP_CONF_DIR=/etc/has/hadoop-conf
```

- 执行 hdfs 命令

  获取到 Ticket 后，就可以正常执行 hdfs 命令了。

```
hadoop fs -ls /
    Found 5 items
    drwxr-xr-x    - hadoop hadoop        0 2017-11-12 14:23 /apps
    drwx------    - hbase  hadoop        0 2017-11-15 19:40 /hbase
    drwxrwx--t+   - hadoop hadoop        0 2017-11-15 17:51 /spark-history
    drwxrwxrwt    - hadoop hadoop        0 2017-11-13 23:25 /tmp
    drwxr-x--t    - hadoop hadoop        0 2017-11-13 16:12 /user
```

说明：

跑 yarn 作业，需要提前在集群中所有节点添加对应的 linux 账号（详情请参见EMR集群添加test账号）。

- java 代码访问 HDFS

  - 使用本地 ticket cache

说明：

需要提前执行 kinit 获取 ticket，且 ticket 过期后程序会访问异常。

```
public static void main(String[] args) throws IOException {
  Configuration conf = new Configuration();
  //加载hdfs的配置，配置从emr集群上复制一份
  conf.addResource(new Path("/etc/ecm/hadoop-conf/hdfs-site.xml"));
  conf.addResource(new Path("/etc/ecm/hadoop-conf/core-site.xml"));
  //需要在程序所在linux账号下，提前kinit获取ticket
  UserGroupInformation.setConfiguration(conf);
  UserGroupInformation.loginUserFromSubject(null);
  FileSystem fs = FileSystem.get(conf);
  FileStatus[] fsStatus = fs.listStatus(new Path("/"));
  for(int i = 0; i < fsStatus.length; i++){
    System.out.println(fsStatus[i].getPath().toString());
  }
}
```

- 使用 keytab 文件（推荐）

说明：

keytab 长期有效，跟本地 ticket 无关。

```
public static void main(String[] args) throws IOException {
 String keytab = args[0];
 String principal = args[1];
 Configuration conf = new Configuration();
 //加载 hdfs 的配置，配置从 emr 集群上复制一份
 conf.addResource(new Path("/etc/ecm/hadoop-conf/hdfs-site.xml"));
 conf.addResource(new Path("/etc/ecm/hadoop-conf/core-site.xml"));
 //直接使用 keytab 文件，该文件从 emr 集群 master-1 上面执行相关命令获取[文档前面有介绍命令]
 UserGroupInformation.setConfiguration(conf);
 UserGroupInformation.loginUserFromKeytab(principal, keytab);
 FileSystem fs = FileSystem.get(conf);
 FileStatus[] fsStatus = fs.listStatus(new Path("/"));
 for(int i = 0; i < fsStatus.length; i++){
   System.out.println(fsStatus[i].getPath().toString());
 }
 }
```

附 pom 依赖：

```
<dependencies>
        <dependency>
          <groupId>org.apache.hadoop</groupId>
          <artifactId>hadoop-common</artifactId>
          <version>2.7.2</version>
        </dependency>
        <dependency>
          <groupId>org.apache.hadoop</groupId>
          <artifactId>hadoop-hdfs</artifactId>
          <version>2.7.2</version>
        </dependency>
```

```
    </dependencies>
```

17.3 RAM认证

E-MapReduce（以下简称EMR）集群中的Kerberos服务端除了支持第一种兼容MIT Kerberos的使用方式，也支持Kerberos客户端使用RAM作为身份信息进行身份认证。

RAM身份认证

RAM产品可以创建或管理子账号，通过子账号实现对云上各个资源的访问控制。

主账号的管理员可以在RAM的用户管理界面创建一个子账号（子账户名称必须符合linux用户的规范），然后将子账号的AccessKey下载下来提供给该子账号对应的开发人员，后续开发人员可以配置AccessKey，从而通过Kerberos认证访问集群服务。

使用RAM身份认证不需要像第一部分MIT Kerberos使用方式一样，提前在Kerberos服务端添加principle等操作。

下面以已经创建的子账号test在Gateway访问为例：

- EMR集群添加test账号。

  EMR的安全集群的yarn使用了LinuxContainerExecutor，在集群上跑yarn作业必须要在集群所有节点上面添加跑作业的用户账号，LinuxContainerExecutor执行程序过程中会根据用户账号进行相关的权限校验。

  EMR 集群管理员在EMR集群的master节点上执行：

  ```
  sudo su hadoop
  sh adduser.sh test 1 2
  ```

  附：adduser.sh代码。

  ```
  #添加的账户名称
   user_name=$1
   #集群master节点个数，如HA集群有2个master
   master_cnt=$2
   #集群worker节点个数
   worker_cnt=$3
   for((i=1;i<=$master_cnt;i++))
   do
    ssh -o StrictHostKeyChecking=no emr-header-$i sudo useradd $user_name
   done
   for((i=1;i<=$worker_cnt;i++))
   do
    ssh -o StrictHostKeyChecking=no emr-worker-$i sudo useradd $user_name
  ```

```
done
```

- Gateway管理员在Gateway机器上添加test用户。

```
useradd test
```

- Gateway管理员配置Kerberos基础环境。

```
sudo su root
 sh config_gateway_kerberos.sh 10.27.230.10 /pathto/emrheader1_pwd_file
 #确保Gateway上面/etc/ecm/hadoop-conf/core-site.xml中值为true
 <property>
   <name>hadoop.security.authentication.use.has</name>
   <value>true</value>
 </property>
```

附： config_gateway_kerberos.sh脚本代码。

```
#EMR集群的emr-header-1的ip
 masterip=$1
 #保存了masterip对应的root登录密码文件
 masterpwdfile=$2
 if ! type sshpass >/dev/null 2>&1; then
   yum install -y sshpass
fi
  ## Kerberos conf
 sshpass -f $masterpwdfile scp root@$masterip:/etc/krb5.conf /etc/
 mkdir /etc/has
 sshpass -f $masterpwdfile scp root@$masterip:/etc/has/has-client.conf /etc/has
 sshpass -f $masterpwdfile scp root@$masterip:/etc/has/truststore /etc/has/
 sshpass -f $masterpwdfile scp root@$masterip:/etc/has/ssl-client.conf /etc/has/
 #修改Kerberos客户端配置，将默认的auth_type从EMR改为RAM
 #也可以手工修改该文件
 sed -i 's/EMR/RAM/g' /etc/has/has-client.conf
```

- test用户登录Gateway配置AccessKey。

```
#登录Gateway的test账号
#执行脚本
sh add_accesskey.sh test
```

附： add_accesskey.sh 脚本（修改一下AccessKey）。

```
user=$1
 if [[ `cat /home/$user/.bashrc | grep 'export AccessKey'` == "" ]];then
 echo "
 #修改为test用户的AccessKeyId/AccessKeySecret
 export AccessKeyId=YOUR_AccessKeyId
 export AccessKeySecret=YOUR_AccessKeySecret
 " >>~/.bashrc
 else
   echo $user AccessKey has been added to .bashrc
```

```
fi
```

- test用户执行命令

  经过以上步骤，test用户可以执行相关命令访问集群服务了。

  1. 执行hdfs命令。

  ```
  [test@gateway ~]$ hadoop fs -ls /
   17/11/19 12:32:15 INFO client.HasClient: The plugin type is: RAM
   Found 4 items
   drwxr-x---   - has    hadoop          0 2017-11-18 21:12 /apps
   drwxrwxrwt   - hadoop hadoop          0 2017-11-19 12:32 /spark-history
   drwxrwxrwt   - hadoop hadoop          0 2017-11-18 21:16 /tmp
   drwxrwxrwt   - hadoop hadoop          0 2017-11-18 21:16 /user
  ```

  2. 运行hadoop作业。

  ```
  [test@gateway ~]$ hadoop jar /usr/lib/hadoop-current/share/hadoop/mapreduce
  /hadoop-mapreduce-examples-2.7.2.jar pi 10 1
  ```

  3. 运行Spark作业。

  ```
  [test@gateway ~]$ spark-submit --conf spark.ui.view.acls=* --class org.apache
  .spark.examples.SparkPi --master yarn-client --driver-memory 512m --num-
  executors 1 --executor-memory 1g --executor-cores 2 /usr/lib/spark-current/
  examples/jars/spark-examples_2.11-2.1.1.jar 10
  ```

17.4 LDAP 认证

E-MapReduce 集群还支持基于 LDAP 的身份认证，通过 LDAP 来管理账号体系，Kerberos 客户端使用 LDAP 中的账号信息作为身份信息进行身份认证。

LDAP 身份认证

LDAP 账号可以和其它服务共用，例如 Hue 等，只需要在 Kerberos 服务端进行相关配置即可。用户可以使用 EMR 集群中已经配置好的 LDAP 服务（ApacheDS），也可以使用已经存在的 LDAP 服务并在 Kerberos 服务端进行相关配置即可。

下面以集群中已经默认启动的 LDAP 服务（ApacheDS）为例。

- Gateway 管理对基础环境进行配置（跟第二部分RAM中的一致，如果已经配置可以跳过）

  区别的地方只是 /etc/has/has-client.conf 中的 auth_type 需要改为 LDAP

  也可以不修改 /etc/has/has-client.conf，用户 test 在自己的账号下拷贝一份该文件进行修改 auth_type，然后通过环境变量指定路径。

  `export HAS_CONF_DIR=/home/test/has-conf`

- EMR 控制台配置 LDAP 管理员用户名/密码到 Kerberos 服务端（HAS）

  进入 EMR 控制台集群的配置管理-HAS软件下，将 LDAP 的管理员用户名和密码配置到对应的 bind_dn 和 bind_password 字段，然后重启 HAS 服务。

  此例中，LDAP 服务即 EMR 集群中的 ApacheDS 服务，相关字段可以从 ApacheDS 中获取。

- EMR 集群管理员在 LDAP 中添加用户信息

  - 获取 ApacheDS 的 LDAP 服务的管理员用户名和密码在 EMR 控制台集群的配置管理 / ApacheDS 的配置中可以查看 manager_dn 和 manager_password。
  - 在 ApacheDS 中添加 test 用户名和密码。

    ```
    登录集群emr-header-1节点root账号
     新建test.ldif文件，内容如下：
     dn: cn=test,ou=people,o=emr
     objectclass: inetOrgPerson
     objectclass: organizationalPerson
     objectclass: person
     objectclass: top
     cn: test
     sn: test
     mail: test@example.com
     userpassword: test1234
     #添加到LDAP，其中-w 对应修改成密码manager_password
     ldapmodify -x -h localhost -p 10389 -D "uid=admin,ou=system" -w "Ns1aSe" -a -f
    test.ldif
     #删除test.ldif
     rm test.ldif
    ```

    将添加的用户名/密码提供给 test 使用。

- 用户 test 配置 LDAP 信息。

  ```
  登录Gateway的test账号
   #执行脚本
   sh add_ldap.sh test
  ```

  附： add_ldap.sh 脚本

  ```
  user=$1
   if [[ `cat /home/$user/.bashrc | grep 'export LDAP_'` == "" ]];then
   echo "
   #修改为test用户的LDAP_USER/LDAP_PWD
   export LDAP_USER=YOUR_LDAP_USER
   export LDAP_PWD=YOUR_LDAP_USER
   " >>~/.bashrc
   else
     echo $user LDAP user info has been added to .bashrc
   fi
  ```

- 用户 test 访问集群服务

  执行 hdfs 命令。

  ```
  [test@iZbp1cyio18s5ymggr7yhrZ ~]$ hadoop fs -ls /
  ```

```
17/11/19 13:33:33 INFO client.HasClient: The plugin type is: LDAP
Found 4 items
drwxr-x---   - has    hadoop        0 2017-11-18 21:12 /apps
drwxrwxrwt   - hadoop hadoop        0 2017-11-19 13:33 /spark-history
drwxrwxrwt   - hadoop hadoop        0 2017-11-19 12:41 /tmp
drwxrwxrwt   - hadoop hadoop        0 2017-11-19 12:41 /user
```

执行Hadoop/Spark 作业等。

17.5 执行计划认证

E-MapReduce集群支持执行计划认证，您可以通过主账号授权子账号进行执行计划的访问。

主账号访问

在主账号登录E-MapReduce控制台的情况下，在执行计划页面运行相关执行计划，将作业提交到安全集群上面执行，以hadoop用户访问作业中涉及的相关开源组件服务。

子账号访问

在RAM子账号登录E-MapReduce控制台的情况下，在执行计划页面运行相关执行计划，将作业提交到安全集群上面执行，以RAM子账号对应的用户名访问作业中涉及的相关开源组件服务。

示例

通过主账号授权子账号的执行计划示例如下：

1. 主账号管理员根据需求创建多个子账号（如A、B和C）。

   在RAM控制台页面给子账号授予AliyunEMRFullAccess和AliyunEMRDevelopAccess的权限后，子账号就能正常登录并使用E-MapReduce控制台上的相关功能。授权详情请参见#unique_123。
2. 主账号管理员将子账号提供给相关开发人员。
3. 开发人员完成作业的创建或执行计划的创建，然后启动运行执行计划提交作业到集群运行，最终在集群上面以该子账号对应的用户名（如A、B和C）去访问相关的组件服务。

   说明:
   周期调度的执行计划目前统一以hadoop账号执行。
4. 组件服务使用子账户的用户名进行相关的权限控制，如子账户A是否有权限访问hdfs中的某个文件等。

17.6 跨域互信

E-MapReduce中的Kerberos支持跨域访问（cross-realm），即不同的Kerberos集群之间可以互相访问。

下面以Cluster-A跨域去访问Cluster-B中的服务为例：

- Cluster-A的emr-header-1的hostname -> emr-header-1.cluster-1234；realm -> EMR.1234.COM。
- Cluster-B的emr-header-1的hostname -> emr-header-1.cluster-6789；realm -> EMR.6789.COM。
- **说明：**
  - hostname可以在emr-header-1上面执行命令hostname获取。
  - realm可以在emr-header-1上面的/etc/krb5.conf获取。

添加 principal

Cluster-A和Cluster-B两个集群的emr-header-1节点分别执行以下完全一样的命令。

```
# root账号
    sh /usr/lib/has-current/bin/hadmin-local.sh /etc/ecm/has-conf -k /etc/ecm/has-conf/admin.keytab
    HadminLocalTool.local: addprinc -pw 123456 krbtgt/EMR.6789.COM@EMR.1234.COM
```

说明：

- 123456是初始密码，可自行修改。
- EMR.6789.COM是Cluster-B的realm，即被访问的集群的realm。
- EMR.1234.COM是Cluster-A的realm，即发起访问的集群realm。

配置 Cluster-A 的 /etc/krb5.conf

在Cluster-A集群上配置 [realms]/[domain_realm]/[capaths]，如下所示。

```
[libdefaults]
    kdc_realm = EMR.1234.COM
    default_realm = EMR.1234.COM
    udp_preference_limit = 4096
    kdc_tcp_port = 88
    kdc_udp_port = 88
    dns_lookup_kdc = false
[realms]
    EMR.1234.COM = {
            kdc = 10.81.49.3:88
    }
    EMR.6789.COM = {
            kdc = 10.81.49.7:88
    }
```

```
[domain_realm]
   .cluster-1234 = EMR.1234.COM
   .cluster-6789 = EMR.6789.COM
[capaths]
   EMR.1234.COM = {
     EMR.6789.COM = .
   }
   EMR.6789.COM = {
     EMR.1234.COM = .
   }
```

将上述/etc/krb5.conf同步到Cluster-A所有节点。

将Cluster-B节点的/etc/hosts文件中绑定信息（只需要长域名emr-xxx-x.cluster-xxx）拷贝到Cluster-A的所有节点 /etc/hosts。

```
10.81.45.89  emr-worker-1.cluster-xxx
 10.81.46.222  emr-worker-2.cluster-xx
 10.81.44.177  emr-header-1.cluster-xxx
```

说明：

- Cluster-A上面如果要跑作业访问Cluster-B，需要先重启yarn。
- Cluster-A的所有节点配置Cluster-B的host绑定信息。

访问Cluster-B服务

在Cluster-A上面可以用Cluster-A的Kerberos的keytab文件/ticket缓存，去访问Cluster-B的服务。

如访问Cluster-B的hdfs服务。

```
su has;
hadoop fs -ls hdfs://emr-header-1.cluster-6789:9000/
Found 4 items
-rw-r-----   2 has    hadoop         34 2017-12-05 18:15 hdfs://emr-header-1.cluster-6789:9000/abc
drwxrwxrwt   - hadoop hadoop          0 2017-12-05 18:32 hdfs://emr-header-1.cluster-6789:9000/spark-history
drwxrwxrwt   - hadoop hadoop          0 2017-12-05 17:53 hdfs://emr-header-1.cluster-6789:9000/tmp
drwxrwxrwt   - hadoop hadoop          0 2017-12-05 18:24 hdfs://emr-header-1.cluster-6789:9000/user
```