表格存储服务

开发指南

文档版本 01

发布日期 2024-11-28

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 HBase 应用开发指导

1.1 开发流程

本文档主要介绍在CloudTable集群模式下如何调用HBase开源接口进行Java应用程序的开发。

开发流程中各阶段的说明如图1-1和表1-1所示。

图 1-1 应用程序开发流程

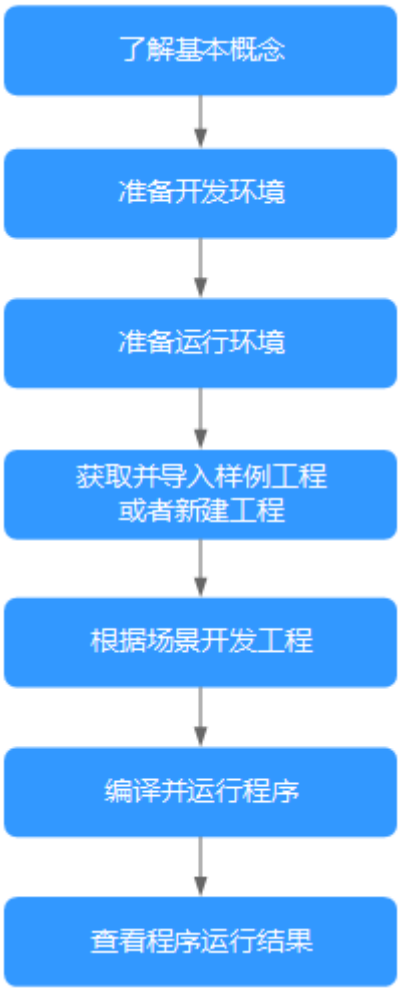


表 1-1 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解HBase的基本概念，了解场景需求，设计表等。	**HBase**
准备开发环境	HBase应用程序当前推荐使用Java语言进行开发。可使用Eclipse工具。	**开发环境简介**
准备运行环境	应用程序的运行环境即客户端环境，请根据指导完成客户端的安装和配置。	**准备Windows运行环境**
准备工程	CloudTable为用户提供了不同场景下的样例程序，您可以导入样例工程进行程序学习。或者您可以根据指导，新建一个工程。	**下载样例工程** **配置并导入工程**
根据场景开发工程	提供了Java语言的样例工程，包含从建表、写入到删除表全流程的样例工程。	**开发HBase应用**

阶段	说明	参考文档
编译并运行程序	指导用户将开发好的程序编译并提交运行。	**编译并运行程序** **安装客户端时编译并运行程序** 或 **未安装客户端时编译并运行程序**
查看程序运行结果	程序运行结果会写在用户指定的路径下。用户还可以通过UI查看应用运行情况。	• 在Windows环境中：**查看调测结果** • 在Linux环境中：**查看调测结果**

1.2 准备开发环境

1.2.1 开发环境简介

在进行二次开发时，要准备的开发环境如**表1-2**所示。

表 1-2 开发环境

准备项	说明
操作系统	Windows系统，推荐Windows 7及以上版本。
安装JDK	开发环境的基本配置。版本要求：1.7或者1.8。考虑到后续版本的兼容性，强烈推荐使用1.8。 **说明** 基于安全考虑，CloudTable服务只支持TLS 1.1和TLS 1.2加密协议，IBM JDK默认TLS只支持1.0，如果使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS1.0/1.1/1.2。详情请参见IBM官方网站的相关说明。
安装和配置Eclipse	用于开发CloudTable应用程序的工具。
网络	确保开发环境或客户端与表格存储服务主机在网络上互通。

1.2.2 准备运行环境

1.2.2.1 准备 Windows 运行环境

操作场景

CloudTable应用开发的运行环境可以部署在Windows环境下。按照如下操作完成运行环境准备。

操作步骤

步骤1 确认CloudTable集群已经安装，并正常运行。

步骤2 准备Windows弹性云服务器。

具体操作请参见**准备弹性云服务器**章节。

步骤3 请在Windows的弹性云服务器上安装JDK1.7及以上版本，强烈推荐使用JDK1.8及以上版本，并且安装Eclipse，Eclipse使用JDK1.7及以上的版本。

说明

- 如果使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
- 如果使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
- 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。

----结束

1.2.3 下载样例工程

前提条件

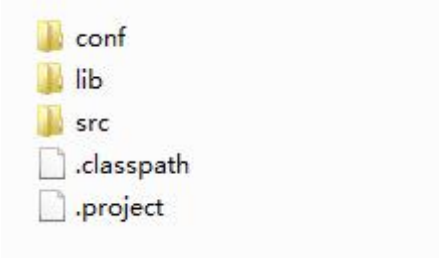
确认表格存储服务已经安装，并正常运行。

下载样例工程

步骤1 下载**样例代码**工程。

步骤2 下载完成后，将样例代码工程安装包解压到本地，得到一个Eclipse的JAVA工程。如**图1-2**所示。

图 1-2 样例代码工程目录结构



----结束

Maven 配置

样例工程中已经包含了hbase的客户端jar包，也可以替换成开源的HBase jar包访问表格存储服务，支持1.X.X版本以上的开源HBase API。如果需要在应用中引入表格存储服务的HBase jar包，可以在Maven中配置如下依赖。

```
<dependencies>
   <dependency>
      <groupId>org.apache.hbase</groupId>
      <artifactId>hbase-client</artifactId>
      <version>1.3.1.0305-cloudtable</version>
   </dependency>
```

```
    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-common</artifactId>
        <version>1.3.1.0305-cloudtable</version>
    </dependency>
</dependencies>
```

使用如下任意一种配置方法配置镜像仓地址（本文提供了如下两种配置方法）。

- **配置方法一**：

  在setting.xml配置文件的mirrors节点中添加开源镜像仓地址：

  ```
  <mirror>
      <id>repo2</id>
      <mirrorOf>central</mirrorOf>
      <url>https://repo1.maven.org/maven2/</url>
  </mirror>
  ```

  在setting.xml配置文件的profiles节点中添加如下镜像仓地址：

  ```
  <profile>
      <id>xxxcloudsdk</id>
      <repositories>
          <repository>
              <id>xxxcloudsdk</id>
              <url>https://repo.xxxcloud.com/repository/maven/xxxcloudsdk/</url>
              <releases><enabled>true</enabled></releases>
              <snapshots><enabled>true</enabled></snapshots>
          </repository>
      </repositories>
  </profile>
  ```

  在setting.xml配置文件的activeProfiles节点中添加如下镜像仓地址：

  ```
  <activeProfile>xxxcloudsdk</activeProfile>
  ```

  说明

  华为云开源镜像站不提供第三方开源jar包下载，请配置华为云开源镜像后，额外配置第三方Maven镜像仓库地址。

- **配置方法二**：

  在二次开发工程样例工程中的pom.xml文件添加如下镜像仓地址：

  ```
  <repositories>
      <repository>
          <id>xxxcloudsdk</id>
          <url>https://mirrors.xxxcloud.com/repository/maven/xxxcloudsdk/</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
      </repository>

      <repository>
          <id>central</id>
          <name>Mavn Centreal</name>
          <url>https://repo1.maven.org/maven2/</url>
      </repository>
  </repositories>
  ```

1.2.4 配置并导入工程

背景信息

将CloudTable样例代码工程导入到Eclipse，就可以开始CloudTable应用开发样例的学习。

前提条件

运行环境已经正确配置，请参见**准备Windows运行环境**。

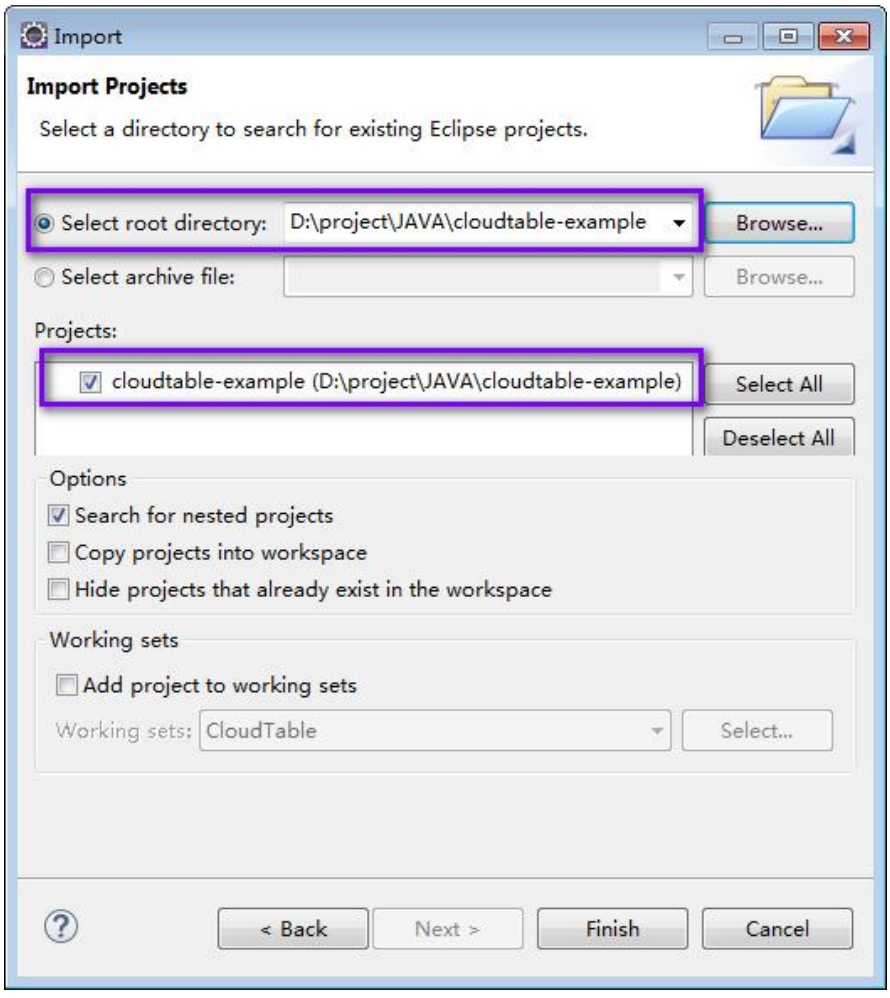
操作步骤

步骤1 把样例工程上传到Windows开发环境中。样例工程的获取方法请参见**下载样例工程**。

步骤2 在应用开发环境中，导入样例工程到Eclipse开发环境。

1. 选择“File > Import > General > Existing Projects into Workspace > Next > Browse”。
   显示“浏览文件夹”对话框。如**图1-3**所示。
2. 选择样例工程文件夹，单击“Finish”。

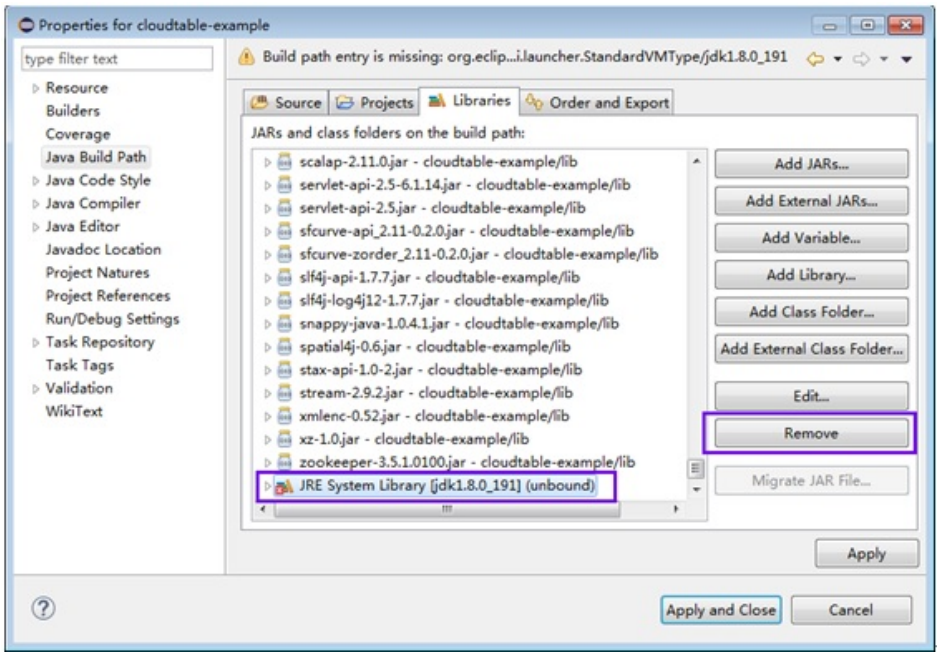
图 1-3 导入样例工程



步骤3 右键单击**cloudtable-example**工程，在弹出的右键菜单中单击“Properties”，弹出“Properties for cloudtable-example”窗口。

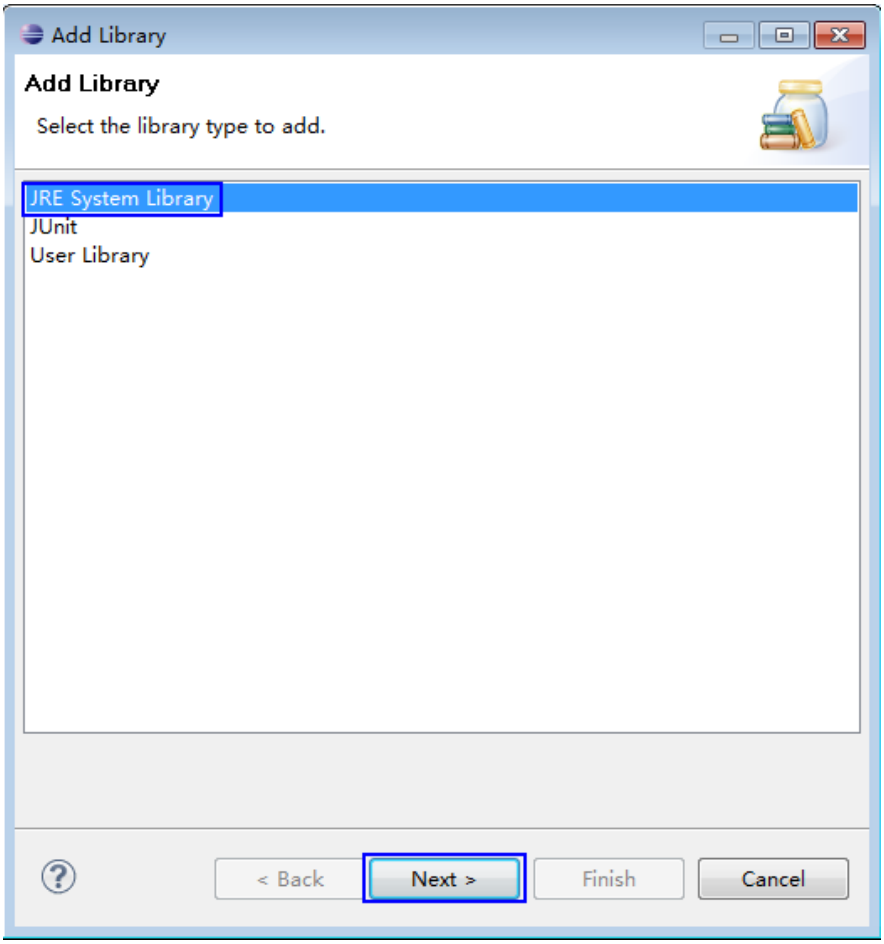
1. 在左边导航上选择“Java Build Path”，单击右侧“Libraries”标签页，按**图1-4**所示将报错的JDK选中后，单击“Remove”删除。

图 1-4 删除报错的 JDK



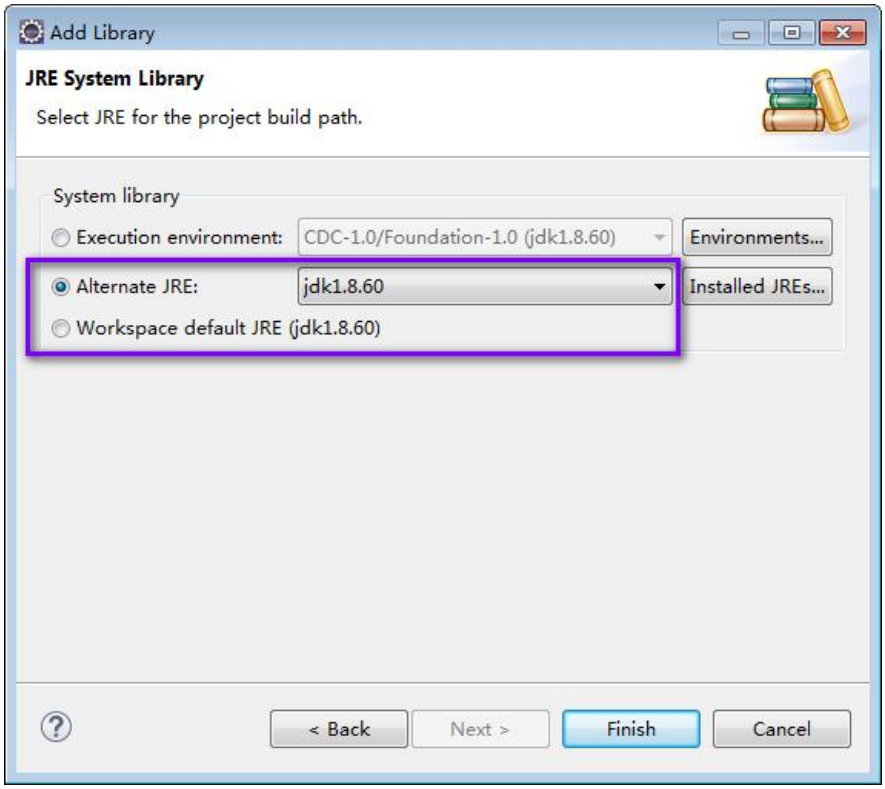
2. 单击“Add Library...”按钮，如图1-5所示，在弹出的窗口中选择“JRE System Library”。

图 1-5 选择增加的 library 类型



3. 在“Add Library”页面中，通过“Alternate JRE”或“Workspace default JRE”选项选择JDK版本。如图1-6所示，选中“Alternate JRE”后，选择JDK版本。

图 1-6 选择 JRE

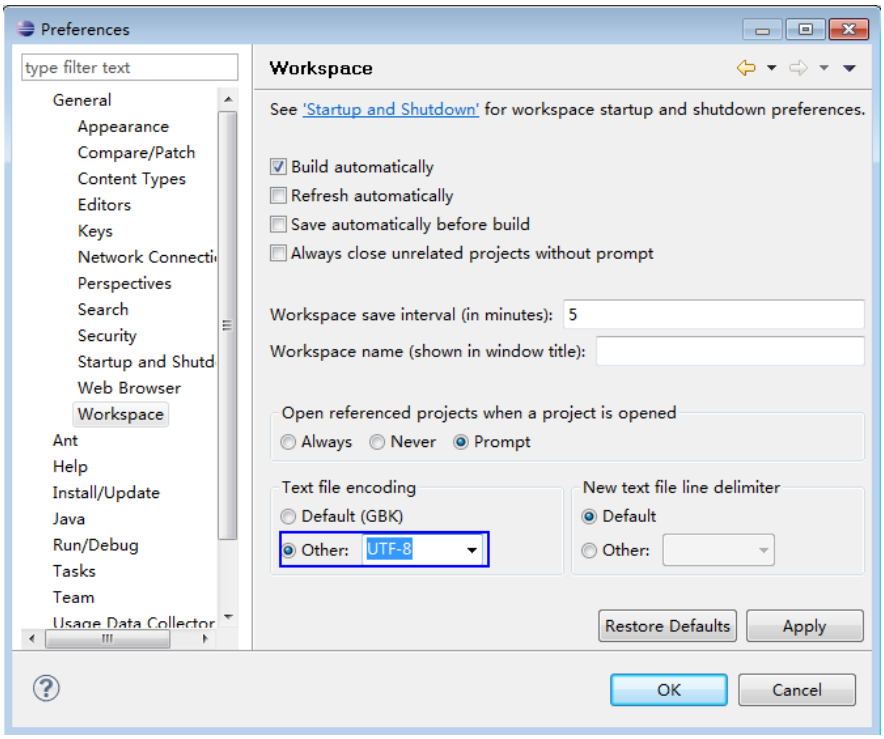


4. 单击“Finish”关闭窗口完成设置。

步骤4 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。
   弹出“Preferences”窗口。
2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图1-7所示。

图 1-7 设置 Eclipse 的编码格式



步骤5 打开样例工程中的“conf/hbase-site.xml”文件，修改“hbase.zookeeper.quorum”的值为正确的Zookeeper地址。

```
<property>
<name>hbase.zookeeper.quorum</name>
<value>xxx-zk1.cloudtable.com,xxx-zk2.cloudtable.com,xxx-zk3.cloudtable.com</value>
</property>
```

其中：*value*中的值为ZooKeeper集群的域名。登录表格存储服务管理控制台，在左侧导航树单击集群管理，然后在集群列表中找到所需要的集群，并获取相应的“ZK链接地址（内网）”。

----结束

1.3 开发 HBase 应用

1.3.1 典型场景说明

通过典型场景，我们可以快速学习和掌握HBase的开发过程，并且对关键的接口函数有所了解。

场景说明

假定用户开发一个应用程序，用于管理企业中的使用A业务的用户信息，如**表1-3**所示，A业务操作流程如下：

- 创建用户信息表。
- 在用户信息中新增用户的学历、职称等信息。
- 根据用户编号查询用户姓名和地址。

- 根据用户姓名进行查询。
- 查询年龄段在[20–29]之间的用户信息。
- 数据统计，统计用户信息表的人员数、年龄最大值、年龄最小值、平均年龄。
- 用户销户，删除用户信息表中该用户的数据。
- A业务结束后，删除用户信息表。

表 1-3 用户信息

编号	姓名	性别	年龄	地址
12005000201	A	Male	19	Shenzhen, Guangdong
12005000202	B	Female	23	Shijiazhuang, Hebei
12005000203	C	Male	26	Ningbo, Zhejiang
12005000204	D	Male	18	Xiangyang, Hubei
12005000205	E	Female	21	Shangrao, Jiangxi
12005000206	F	Male	32	Zhuzhou, Hunan
12005000207	G	Female	29	Nanyang, Henan
12005000208	H	Female	30	Kaixian, Chongqing
12005000209	I	Male	26	Weinan, Shaanxi
12005000210	J	Male	25	Dalian, Liaoning

数据规划

合理地设计表结构、行键、列名能充分利用HBase的优势。本样例工程以唯一编号作为RowKey，列都存储在info列族中。

1.3.2 开发思路

功能分解

根据上述的业务场景进行功能分解，需要开发的功能点如表1-4所示。

表 1-4 在 HBase 中开发的功能

序号	步骤	代码实现
1	根据**典型场景说明**中的信息创建表。	请参见**创建表**。
2	导入用户数据。	请参见**插入数据**。
3	增加“教育信息”列族，在用户信息中新增用户的学历、职称等信息。	请参见**修改表**。
4	根据用户编号查询用户姓名和地址。	请参见**使用Get读取数据**。
5	根据用户姓名进行查询。	请参见**使用过滤器Filter**。
6	用户销户，删除用户信息表中该用户的数据。	请参见**删除数据**。
7	A业务结束后，删除用户信息表。	请参见**删除表**。

关键设计原则

HBase是以RowKey为字典排序的分布式数据库系统，RowKey的设计对性能影响很大，具体的RowKey设计请考虑与业务结合。

1.3.3 样例代码说明

1.3.3.1 配置参数

步骤1 执行样例代码前，必须在hbase-site.xml配置文件中，配置正确的ZooKeeper集群的地址。

配置项如下：

```
<property>
<name>hbase.zookeeper.quorum</name>
<value>xxx-zk1.cloudtable.com,xxx-zk2.cloudtable.com,xxx-zk3.cloudtable.com</value>
</property>
```

其中：*value*中的值为ZooKeeper集群的域名。登录表格存储服务管理控制台，在左侧导航树单击集群管理，然后在集群列表中找到所需要的集群，并获取相应的“ZK链接地址（内网）”。

----结束

1.3.3.2 创建 Configuration

功能介绍

HBase通过加载配置文件来获取配置项。

说明

1. 加载配置文件是一个比较耗时的操作，如非必要，请尽量使用同一个Configuration对象。
2. 样例代码未考虑多线程同步的问题，如有需要，请自行增加。其它样例代码也一样，不再一一进行说明。

代码样例

下面代码片段在com.huaweicloudtable.hbase.examples包中。

```
private static void init() throws IOException {
  // Default load from conf directory
  conf = HBaseConfiguration.create(); // 注[1]
  String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
  Path hbaseSite = new Path(userdir + "hbase-site.xml");
  if (new File(hbaseSite.toString()).exists()) {
    conf.addResource(hbaseSite);
  }
}
```

注意事项

- 注[1] 如果配置文件目录conf已经加入classpath路径中，那么后面的加载指定配置文件的代码可以不执行。

1.3.3.3 创建 Connection

功能介绍

HBase通过ConnectionFactory.createConnection(configuration)方法创建Connection对象。传递的参数为上一步创建的Configuration。

Connection封装了底层与各实际服务器的连接以及与ZooKeeper的连接。Connection通过ConnectionFactory类实例化。创建Connection是重量级操作，而且Connection是线程安全的，因此，多个客户端线程可以共享一个Connection。

典型的用法，一个客户端程序共享一个单独的Connection，每一个线程获取自己的Admin或Table实例，然后调用Admin对象或Table对象提供的操作接口。不建议缓存或者池化Table、Admin。Connection的生命周期由调用者维护，调用者通过调用close()，释放资源。

说明

建议业务代码连接同一个CloudTable集群时，多线程创建并复用同一个Connection，不必每个线程都创建各自Connection。Connection是连接CloudTable集群的连接器，创建过多连接会加重Zookeeper负载，并损耗业务读写性能。

代码样例

以下代码片段是创建Connection对象的示例：

```
private TableName tableName = null;
private Connection conn = null;

public HBaseSample(Configuration conf) throws IOException {
  this.tableName = TableName.valueOf("hbase_sample_table");
  this.conn = ConnectionFactory.createConnection(conf);
}
```

1.3.3.4 创建表

功能简介

HBase通过org.apache.hadoop.hbase.client.Admin对象的createTable方法来创建表，并指定表名、列族名。创建表有两种方式（强烈建议采用预分Region建表方式）：

- 快速建表，即创建表后整张表只有一个Region，随着数据量的增加会自动分裂成多个Region。
- 预分Region建表，即创建表时预先分配多个Region，此种方法建表可以提高写入大量数据初期的数据写入速度。

说明

表名以及列族名不能包含特殊字符，可以由字母、数字以及下划线组成。

代码样例

```
public void testCreateTable() {
  LOG.info("Entering testCreateTable.");

  // Specify the table descriptor.
  HTableDescriptor htd = new HTableDescriptor(tableName); // (1)

  // Set the column family name to info.
  HColumnDescriptor hcd = new HColumnDescriptor("info"); // (2)

  // Set data encoding methods. HBase provides DIFF,FAST_DIFF,PREFIX
  // and PREFIX_TREE
  hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);  // 注[1]

  // Set compression methods, HBase provides two default compression
  // methods:GZ and SNAPPY
  // GZ has the highest compression rate,but low compression and
  // decompression efficiency,fit for cold data
  // SNAPPY has low compression rate, but high compression and
  // decompression efficiency,fit for hot data.
  // it is advised to use SANPPY
  hcd.setCompressionType(Compression.Algorithm.SNAPPY);
  htd.addFamily(hcd); // (3)

  Admin admin = null;
  try {
    // Instantiate an Admin object.
    admin = conn.getAdmin(); // (4)
    if (!admin.tableExists(tableName)) {
      LOG.info("Creating table...");
      admin.createTable(htd); // 注[2] (5)
      LOG.info(admin.getClusterStatus());
      LOG.info(admin.listNamespaceDescriptors());
      LOG.info("Table created successfully.");
    } else {
      LOG.warn("table already exists");
    }
  } catch (IOException e) {
    LOG.error("Create table failed.", e);
  } finally {
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Failed to close admin ", e);
      }
    }
```

```
  }
  LOG.info("Exiting testCreateTable.");
}
```

解释

（1）创建表描述符。

（2）创建列族描述符。

（3）添加列族描述符到表描述符中。

（4）获取Admin对象，Admin提供了建表、创建列族、检查表是否存在、修改表结构和列族结构以及删除表等功能。

（5）调用Admin的建表方法。

注意事项

- 注[1] 可以设置列族的压缩方式，代码片段如下：

```
//设置编码算法，HBase提供了DIFF，FAST_DIFF，PREFIX和PREFIX_TREE四种编码算法
 hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);
 //设置文件压缩方式，HBase默认提供了GZ和SNAPPY两种压缩算法
 //其中GZ的压缩率高，但压缩和解压性能低，适用于冷数据
 //SNAPPY压缩率低，但压缩解压性能高，适用于热数据
 //建议默认开启SNAPPY压缩
 hcd.setCompressionType(Compression.Algorithm.SNAPPY);
```

- 注[2] 可以通过指定起始和结束RowKey，或者通过RowKey数组预分Region两种方式建表，代码片段如下：

```
// 创建一个预划分region的表
 byte[][] splits = new byte[4][];
 splits[0] = Bytes.toBytes("A");
 splits[1] = Bytes.toBytes("H");
 splits[2] = Bytes.toBytes("O");
 splits[3] = Bytes.toBytes("U");
 admin.createTable(htd, splits);
```

1.3.3.5 删除表

功能简介

HBase通过org.apache.hadoop.hbase.client.Admin的deleteTable方法来删除表。

代码样例

```
public void dropTable() {
  LOG.info("Entering dropTable.");
  Admin admin = null;
  try {
    admin = conn.getAdmin();
    if (admin.tableExists(tableName)) {
      // Disable the table before deleting it.
      admin.disableTable(tableName);
      // Delete table.
      admin.deleteTable(tableName);//注[1]
    }
    LOG.info("Drop table successfully.");
  } catch (IOException e) {
    LOG.error("Drop table failed " ,e);
  } finally {
    if (admin != null) {
```



```
    try {
      // Close the Admin object.
      admin.close();
    } catch (IOException e) {
      LOG.error("Close admin failed " ,e);
    }
  }
 }
 LOG.info("Exiting dropTable.");
}
```

注意事项

注[1] 只有在调用disableTable接口后， 再调用deleteTable接口才能将表删除成功。因此，deleteTable常与disableTable，enableTable，tableExists，isTableEnabled，isTableDisabled结合在一起使用。

1.3.3.6 修改表

功能简介

HBase通过org.apache.hadoop.hbase.client.Admin的modifyTable方法修改表信息。

代码样例

```
public void testModifyTable() {
 LOG.info("Entering testModifyTable.");

 // Specify the column family name.
 byte[] familyName = Bytes.toBytes("education");
 Admin admin = null;
 try {
  // Instantiate an Admin object.
  admin = conn.getAdmin();
  // Obtain the table descriptor.
  HTableDescriptor htd = admin.getTableDescriptor(tableName);
  // Check whether the column family is specified before modification.
  if (!htd.hasFamily(familyName)) {
    // Create the column descriptor.
    HColumnDescriptor hcd = new HColumnDescriptor(familyName);
    htd.addFamily(hcd);
    // Disable the table to get the table offline before modifying
    // the table.
    admin.disableTable(tableName);
    // Submit a modifyTable request.
    admin.modifyTable(tableName, htd);  //注[1]
    // Enable the table to get the table online after modifying the
    // table.
    admin.enableTable(tableName);
  }
  LOG.info("Modify table successfully.");
 } catch (IOException e) {
  LOG.error("Modify table failed " ,e);
 } finally {
  if (admin != null) {
    try {
      // Close the Admin object.
      admin.close();
    } catch (IOException e) {
      LOG.error("Close admin failed " ,e);
    }
  }
 }
 LOG.info("Exiting testModifyTable.");
}
```


注意事项

注[1] 只有在调用disableTable接口后， 再调用modifyTable接口才能将表修改成功。之后，请调用enableTable接口重新启用表。

1.3.3.7 插入数据

功能简介

HBase是一个面向列的数据库，一行数据，可能对应多个列族，而一个列族又可以对应多个列。

通常，写入数据的时候，我们需要指定要写入的列（含列族名称和列名称）。

HBase通过HTable的put方法来Put数据，可以是一行数据也可以是数据集。

代码样例

```
public void testPut() {
  LOG.info("Entering testPut.");
  // Specify the column family name.
  byte[] familyName = Bytes.toBytes("info");
  // Specify the column name.
  byte[][] qualifiers = { Bytes.toBytes("name"), Bytes.toBytes("gender"),
      Bytes.toBytes("age"), Bytes.toBytes("address") };
  Table table = null;
  try {
    // Instantiate an HTable object.
    table = conn.getTable(tableName);
    List<Put> puts = new ArrayList<Put>();
    // Instantiate a Put object.
    Put put = new Put(Bytes.toBytes("012005000201"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("A"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("19"));
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Shenzhen, Guangdong"));
    puts.add(put);
    put = new Put(Bytes.toBytes("012005000202"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("B"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("23"));
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Shijiazhuang, Hebei"));
    puts.add(put);
    put = new Put(Bytes.toBytes("012005000203"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("C"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("26"));
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Ningbo, Zhejiang"));
    puts.add(put);
    put = new Put(Bytes.toBytes("012005000204"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("D"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("18"));
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Xiangyang, Hubei"));
    puts.add(put);
    put = new Put(Bytes.toBytes("012005000205"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("E"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("21"));
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Shangrao, Jiangxi"));
    puts.add(put);
    put = new Put(Bytes.toBytes("012005000206"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("F"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("32"));
```



```
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Zhuzhou, Hunan"));
    puts.add(put);
    put = new Put(Bytes.toBytes("012005000207"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("G"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("29"));
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Nanyang, Henan"));
    puts.add(put);
    put = new Put(Bytes.toBytes("012005000208"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("H"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("30"));
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Kaixian, Chongqing"));
    puts.add(put);
    put = new Put(Bytes.toBytes("012005000209"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("I"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("26"));
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Weinan, Shaanxi"));
    puts.add(put);
    put = new Put(Bytes.toBytes("012005000210"));
    put.addColumn(familyName, qualifiers[0], Bytes.toBytes("J"));
    put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
    put.addColumn(familyName, qualifiers[2], Bytes.toBytes("25"));
    put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Dalian, Liaoning"));
    puts.add(put);
    // Submit a put request.
    table.put(puts);

    LOG.info("Put successfully.");
  } catch (IOException e) {
    LOG.error("Put failed " ,e);
  } finally {
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testPut.");
}
```

注意事项

不允许多个线程在同一时间共用同一个HTable实例。HTable是一个非线程安全类，因此，同一个HTable实例，不应该被多个线程同时使用，否则可能会带来并发问题。

1.3.3.8 删除数据

功能简介

HBase通过Table实例的delete方法来Delete数据，可以是一行数据也可以是数据集。

具体删除方法根据用户使用场景选取。

代码样例

```
public void testDelete() {
  LOG.info("Entering testDelete.");

  byte[] rowKey = Bytes.toBytes("012005000201");
```



```
  Table table = null;
  try {
    // Instantiate an HTable object.
    table = conn.getTable(tableName);

    // Instantiate a Delete object.
    Delete delete = new Delete(rowKey);

    // Submit a delete request.
    table.delete(delete);

    LOG.info("Delete table successfully.");
  } catch (IOException e) {
    LOG.error("Delete table failed " ,e);
  } finally {
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testDelete.");
}
```

1.3.3.9 使用 Get 读取数据

功能简介

要从表中读取一条数据，首先需要实例化该表对应的Table实例，然后创建一个Get对象。

可以为Get对象设定参数值，如列族的名称和列的名称。

查询到的行数据存储在Result对象中，Result中可以存储多个Cell。

代码样例

```
public void testGet() {
  LOG.info("Entering testGet.");
  // Specify the column family name.
  byte[] familyName = Bytes.toBytes("info");
  // Specify the column name.
  byte[][] qualifier = { Bytes.toBytes("name"), Bytes.toBytes("address") };
  // Specify RowKey.
  byte[] rowKey = Bytes.toBytes("012005000201");
  Table table = null;
  try {
    // Create the Table instance.
    table = conn.getTable(tableName);
    // Instantiate a Get object.
    Get get = new Get(rowKey);
    // Set the column family name and column name.
    get.addColumn(familyName, qualifier[0]);
    get.addColumn(familyName, qualifier[1]);
    // Submit a get request.
    Result result = table.get(get);
    // Print query results.
    for (Cell cell : result.rawCells()) {
      LOG.info(Bytes.toString(CellUtil.cloneRow(cell)) + ":"
          + Bytes.toString(CellUtil.cloneFamily(cell)) + ","
          + Bytes.toString(CellUtil.cloneQualifier(cell)) + ","
          + Bytes.toString(CellUtil.cloneValue(cell)));
    }
```

```
    LOG.info("Get data successfully.");
  } catch (IOException e) {
    LOG.error("Get data failed " ,e);
  } finally {
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testGet.");
}
```

1.3.3.10 使用 Scan 读取数据

功能简介

要从表中读取数据，首先需要实例化该表对应的Table实例，然后创建一个Scan对象，并针对查询条件设置Scan对象的参数值，为了提高查询效率，最好指定StartRow和StopRow。查询结果的多行数据保存在ResultScanner对象中，每行数据以Result对象形式存储，Result中存储了多个Cell。

代码样例

```
public void testScanData() {
  LOG.info("Entering testScanData.");
  Table table = null;
  // Instantiate a ResultScanner object.
  ResultScanner rScanner = null;
  try {
    // Create the Configuration instance.
    table = conn.getTable(tableName);
    // Instantiate a Get object.
    Scan scan = new Scan();
    scan.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"));
    // Set the cache size.
    scan.setCaching(1000);
    // Submit a scan request.
    rScanner = table.getScanner(scan);
    // Print query results.
    for (Result r = rScanner.next(); r != null; r = rScanner.next()) {
      for (Cell cell : r.rawCells()) {
        LOG.info(Bytes.toString(CellUtil.cloneRow(cell)) + ":"
            + Bytes.toString(CellUtil.cloneFamily(cell)) + ","
            + Bytes.toString(CellUtil.cloneQualifier(cell)) + ","
            + Bytes.toString(CellUtil.cloneValue(cell)));
      }
    }
    LOG.info("Scan data successfully.");
  } catch (IOException e) {
    LOG.error("Scan data failed " ,e);
  } finally {
    if (rScanner != null) {
      // Close the scanner object.
      rScanner.close();
    }
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
```

```
    }
  }
  LOG.info("Exiting testScanData.");
}
```

注意事项

1. 建议Scan时指定StartRow和StopRow，一个有确切范围的Scan，性能会更好些。
2. 可以设置Batch和Caching关键参数。
   - Batch
     使用Scan调用next接口每次最大返回的记录数，与一次读取的列数有关。
   - Caching
     RPC请求返回next记录的最大数量，该参数与一次RPC获取的行数有关。

1.3.3.11 使用过滤器 Filter

功能简介

HBase Filter主要在Scan和Get过程中进行数据过滤，通过设置一些过滤条件来实现，如设置RowKey、列名或者列值的过滤条件。

具体过滤条件根据用户使用场景选取。

代码样例

```
public void testSingleColumnValueFilter() {
  LOG.info("Entering testSingleColumnValueFilter.");
  Table table = null;
  ResultScanner rScanner = null;

  try {
    table = conn.getTable(tableName);
    Scan scan = new Scan();
    scan.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"));
    // Set the filter criteria.
    SingleColumnValueFilter filter = new SingleColumnValueFilter(
        Bytes.toBytes("info"), Bytes.toBytes("name"), CompareOp.EQUAL,
        Bytes.toBytes("I"));
    scan.setFilter(filter);
    // Submit a scan request.
    rScanner = table.getScanner(scan);
    // Print query results.
    for (Result r = rScanner.next(); r != null; r = rScanner.next()) {
      for (Cell cell : r.rawCells()) {
        LOG.info(Bytes.toString(CellUtil.cloneRow(cell)) + ":"
            + Bytes.toString(CellUtil.cloneFamily(cell)) + ","
            + Bytes.toString(CellUtil.cloneQualifier(cell)) + ","
            + Bytes.toString(CellUtil.cloneValue(cell)));
      }
    }
    LOG.info("Single column value filter successfully.");
  } catch (IOException e) {
    LOG.error("Single column value filter failed " ,e);
  } finally {
      if (rScanner != null) {
        // Close the scanner object.
        rScanner.close();
      }
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
```

```
    } catch (IOException e) {
      LOG.error("Close table failed " ,e);
    }
   }
  }
  LOG.info("Exiting testSingleColumnValueFilter.");
}
```

1.4 访问 HBase ThriftServer 样例程序

1.4.1 访问 ThriftServer 操作表

操作场景

用户根据指定的host和port访问对应的ThriftServer实例，进行HBase表的创建，删除等操作。

前提条件

- 集群已启用ThriftServer并从集群详情页面获取到ThriftServer IP。
- 已下载Thrift安装包，安装包下载地址：**链接**。
- 已下载HBase Thrift定义文件，文件下载地址：**地址**。

操作步骤

步骤1 登录表格存储服务控制台。

步骤2 在页面左上角选择区域。

步骤3 单击“集群管理”，进入集群管理界面。

步骤4 单击HBase集群名称，进入集群详情页面查看ThriftServer的状态，如果ThriftServer为开启状态，无需开启操作；如果ThriftServer为关闭状态，则选返回集群管理界面，单击“更多 > 开启ThriftServer”，等待完成后，即可进行使用。

说明

- ThriftServer当前不具备负载均衡的能力，用户需要避免在代码里面同时访问同一个ThriftServer实例，避免单实例过载。
- 用户需要在应用代码里面增加重试机制，保证单ThriftServer实例故障或者重启时，可以重试其他ThriftServer实例。

步骤5 参考**Thrift官方指导**在客户端节点安装Thrift安装包。

步骤6 使用Thrift命令将HBase Thrift定义文件生成对应语言的接口文件，支持的语言有C++，Python等。参考命令如下：

```
thrift --gen <语言> hbase.thrift
```

说明

<语言>为要生成的目标语言，支持cpp(C++)、py(Python)等。

以Python为例，执行命令为：thrift --gen py hbase.thrif

----结束

C++代码样例

```
#include "THBaseService.h"
#include <config.h>
#include <vector>
#include <ostream>
#include <iostream>
#include "transport/TSocket.h"
#include <transport/TBufferTransports.h>
#include <protocol/TBinaryProtocol.h>
using namespace std;
using namespace apache::thrift;
using namespace apache::thrift::protocol;
using namespace apache::thrift::transport;
using namespace apache::hadoop::hbase::thrift2;
using boost::shared_ptr;
int main(int argc, char **argv) {
    // ThriftServer的ip和端口号
    std::string host = "x.x.x.x";
    int port = 9090;
    boost::shared_ptr<TSocket> socket(new TSocket(host, port));
    boost::shared_ptr<TTransport> transport(new TBufferedTransport(socket));
    boost::shared_ptr<TProtocol> protocol(new TBinaryProtocol(transport));
    // 设置表名
    std::string ns("default");
    std::string table("test");
    TTableName tableName;
    tableName.__set_ns(ns);
    tableName.__set_qualifier(table);
     try {
        // 创建连接
        transport->open();
        printf("Opened connection\n");
        // 初始化客户端接口
        THBaseServiceClient  client(protocol);

        // 建表
        TColumnFamilyDescriptor column;
        column.__set_name("f1");
        column.__set_maxVersions(10);
        std::vector<TColumnFamilyDescriptor> columns;
        columns.push_back(column);

        TTableDescriptor tableDescriptor;
        tableDescriptor.__set_tableName(tableName);
        tableDescriptor.__set_columns(columns);
        std::vector<std::string> splitKeys;
        splitKeys.push_back("row2");
        splitKeys.push_back("row4");
        splitKeys.push_back("row8");
        printf("Creating table: %s\n", table.c_str());
        try {
            client.createTable(tableDescriptor, splitKeys);
        } catch (const TException  &te) {
            std::cerr << "ERROR: " << te.what() << std::endl;
        }
        // Put写入单条数据
        TColumnValue columnValue;
        columnValue.__set_family("f1");
        columnValue.__set_qualifier("q1");
        columnValue.__set_value("val_001");
        std::vector<TColumnValue> columnValues;
        columnValues.push_back(columnValue);
        TPut put;
        put.__set_row("row1");
        put.__set_columnValues(columnValues);
        client.put(table, put);
        printf("Put single row success\n");
        // Put写入多条数据
        TColumnValue columnValue2;
```

```
    columnValue2.__set_family("f1");
    columnValue2.__set_qualifier("q1");
    columnValue2.__set_value("val_003");
    std::vector<TColumnValue> columnValues2;
    columnValues2.push_back(columnValue2);
    TPut put2;
    put2.__set_row("row3");
    put2.__set_columnValues(columnValues2);
    TColumnValue columnValue3;
    columnValue3.__set_family("f1");
    columnValue3.__set_qualifier("q1");
    columnValue3.__set_value("val_005");
    std::vector<TColumnValue> columnValues3;
    columnValues3.push_back(columnValue3);
    TPut put3;
    put3.__set_row("row5");
    put3.__set_columnValues(columnValues3);
    std::vector<TPut> puts;
    puts.push_back(put2);
    puts.push_back(put3);
    client.putMultiple(table, puts);
    printf("Put multiple rows success\n");
    // Get查询单条数据
    TResult result;
    TGet get;
    get.__set_row("row1");
    client.get(result, table, get);
    std::vector<TColumnValue> list=result.columnValues;
    std::vector<TColumnValue>::const_iterator iter;
    std::string row = result.row;
    for(iter=list.begin();iter!=list.end();iter++) {
        printf("%s=%s, %s,%s\n",row.c_str(),(*iter).family.c_str(),(*iter).qualifier.c_str(),(*iter).value.c_str());
    }
    printf("Get single row success.\n");
    // Get查询多条数据
    std::vector<TGet> multiGets;
    TGet get1;
    get1.__set_row("row1");
    multiGets.push_back(get1);
    TGet get2;
    get2.__set_row("row5");
    multiGets.push_back(get2);

    std::vector<TResult> multiRows;
    client.getMultiple(multiRows, table, multiGets);
    for(std::vector<TResult>::const_iterator iter1=multiRows.begin();iter1!=multiRows.end();iter1++) {
        std::vector<TColumnValue> list=(*iter1).columnValues;
        std::vector<TColumnValue>::const_iterator iter2;
        std::string row = (*iter1).row;
        for(iter2=list.begin();iter2!=list.end();iter2++) {
            printf("%s=%s, %s,%s\n",row.c_str(),(*iter2).family.c_str(),(*iter2).qualifier.c_str(),
(*iter2).value.c_str());
        }
    }
    printf("Get multiple rows success.\n");
    // Scan查询数据
    TScan scan;
    scan.__set_startRow("row1");
    scan.__set_stopRow("row7");
    int32_t nbRows = 2;
    std::vector<TResult> results;
    TResult* current = NULL;
    while (true) {
        client.getScannerResults(results, table, scan, nbRows);
        if (results.size() == 0) {
            printf("No more result.\n");
            break;
        }
        std::vector<TResult>::const_iterator itx;
```



```
            for(itx=results.begin();itx!=results.end();itx++) {
                current = (TResult*) &(*itx);
                if (current == NULL) {
                    break;
                } else {
                    std::vector<TColumnValue> values=(*current).columnValues;
                    std::vector<TColumnValue>::const_iterator iterator;
                    for(iterator=list.begin();iterator!=list.end();iterator++) {
                        printf("%s=%s, %s,%s\n",(*current).row.c_str(),(*iterator).family.c_str(),
(*iterator).qualifier.c_str(),(*iterator).value.c_str());
                    }
                }
            }
            if (current == NULL) {
                printf("Scan data done.\n");
                break;
            } else {
                scan.__set_startRow((*current).row + (char)0);
            }
        }
        //禁用和删除表
        client.disableTable(tableName);
        printf("Disabled %s\n", table.c_str());
        client.deleteTable(tableName);
        printf("Deleted %s\n", table.c_str());
        transport->close();
        printf("Closed connection\n");
    } catch (const TException &tx) {
        std::cerr << "ERROR(exception): " << tx.what() << std::endl;
    }
    return 0;
}
```

Python 代码样例

```
# -*- coding: utf-8  -*-

# 引入公共模块
import sys
import os

# 引入Thrift自带模块，如果不存在则需要执行pip install thrift安装
from thrift.transport import TTransport
from thrift.protocol import TBinaryProtocol
from thrift.transport import THttpClient
from thrift.transport import TSocket

# 引入通过hbase.thrift生成的模块
gen_py_path = os.path.abspath('gen-py')
sys.path.append(gen_py_path)
from hbase import THBaseService
from hbase.ttypes import TColumnValue, TColumn, TTableName, TTableDescriptor,
TColumnFamilyDescriptor, TGet, TPut, TScan
# 配置CloudTable HBase集群的ThriftServer的IP，可以通过集群的详情页面获取
host = "x.x.x.x"

socket = TSocket.TSocket(host, 9090)
transport = TTransport.TBufferedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(transport)
client = THBaseService.Client(protocol)
transport.open()

# 测试表名
tableNameInBytes = "test".encode("utf8")

tableName = TTableName(ns="default".encode("utf8"), qualifier=tableNameInBytes)
# 预分region的split key
splitKeys=[]
splitKeys.append("row3".encode("utf8"))
```

```
splitKeys.append("row5".encode("utf8"))
# 建表操作
client.createTable(TTableDescriptor(tableName=tableName,
columns=[TColumnFamilyDescriptor(name="cf1".encode("utf8"))]), splitKeys)
print("Create table %s success." % tableName)

# Put单条数据
put = TPut(row="row1".encode("utf8"), columnValues=[TColumnValue(family="cf1".encode("utf8"),
qualifier="q1".encode("utf8"), value="test_value1".encode("utf8"))])
client.put(tableNameInBytes, put)
print("Put single row success.")

# Put多条数据
puts = []
puts.append(TPut(row="row4".encode("utf8"), columnValues=[TColumnValue(family="cf1".encode("utf8"),
qualifier="q1".encode("utf8"), value="test_value1".encode("utf8"))]))
puts.append(TPut(row="row6".encode("utf8"), columnValues=[TColumnValue(family="cf1".encode("utf8"),
qualifier="q1".encode("utf8"), value="test_value1".encode("utf8"))]))
puts.append(TPut(row="row8".encode("utf8"), columnValues=[TColumnValue(family="cf1".encode("utf8"),
qualifier="q1".encode("utf8"), value="test_value1".encode("utf8"))]))
client.putMultiple(tableNameInBytes, puts)
print("Put rows success.")

# Get单条数据
get = TGet(row="row1".encode("utf8"))
result = client.get(tableNameInBytes, get)
print("Get Result: ", result)

# Get多条数据
gets = []
gets.append(TGet(row="row4".encode("utf8")))
gets.append(TGet(row="row8".encode("utf8")))
results = client.getMultiple(tableNameInBytes, gets)
print("Get multiple rows: ", results)

# Scan数据
startRow, stopRow = "row4".encode("utf8"), "row9".encode("utf8")
scan = TScan(startRow=startRow, stopRow=stopRow)
caching=1
results = []
while True:
    scannerResult = client.getScannerResults(tableNameInBytes, scan, caching)
    lastOne = None
    for result in scannerResult:
        results.append(result)
        print("Scan Result: ", result)
        lastOne = result
    # 没有更多数据，退出
    if lastOne is None:
        break
    else:
        # 重新生成下一次scan的startRow
        newStartRow = bytearray(lastOne.row)
        newStartRow.append(0x00)
        scan = TScan(startRow=newStartRow, stopRow=stopRow)


# 禁用和删除表
client.disableTable(tableName)
print("Disable table %s success." % tableName)
client.deleteTable(tableName)
print("Delete table %s success." % tableName)

# 所有操作都结束后，关闭连接
transport.close()
```

1.5 开发标签索引应用

1.5.1 应用背景

CloudTable作为大数据存储服务，提供高效的kv随机查询能力。在此基础上，CloudTable服务引入自研的**分布式多维标签索引**能力，存储格式与计算基于位图进行。用户可以根据自身业务需求来定义HBase表中的哪些字段需要构建标签索引，用户写入数据时将自动生成标签数据。同时，标签索引基于Lucene的语法，提供高效的多维标签查询接口。可应用于用户画像、推荐系统、人工智能、时空数据等场景。

CloudTable服务支持标签索引能力，您只需要创建CloudTable集群，就可以在弹性云服务器（ECS）上开发客户端应用进行多维标签查询。

1.5.2 典型场景说明

通过典型场景，我们可以快速学习和掌握标签索引的开发过程，并且对关键的接口函数有所了解。

场景说明

某在线付费学习APP给会员打上各种属性标签，以方便后续的资源投放和精准定位营销。例如，需要ms级统计拥有学士和硕士学位的用户数量是多少？以及是哪些用户？

用户信息表字段如下：

表 1-5 用户信息

字段名称	字段描述	是否需要标签索引
name	用户名	否
education	用户学历	是
otherInfo	用户其他信息	否

1.5.3 开发思路

表 1-6 开发思路

序号	步骤	代码实现
1	创建HBase表时开启标签索引	请参见**创建数据表开启标签索引**
2	HBase put写入数据	请参见**写入数据**
3	查询数据	请参见： • **普通查询** • **抽样查询** • **分页查询** • **统计查询**

1.5.4 样例代码说明

1.5.4.1 配置参数

步骤1 执行样例代码前，必须在hbase-site.xml配置文件中，配置正确的ZooKeeper集群的地址。

配置项如下：

```
<property>
<name>hbase.zookeeper.quorum</name>
<value>xxx-zk1.cloudtable.com,xxx-zk2.cloudtable.com,xxx-zk3.cloudtable.com</value>
</property>
```

其中：*value*中的值为ZooKeeper集群的域名。登录表格存储服务管理控制台，在左侧导航树单击“集群管理”，然后在集群列表中找到所需要的集群，并获取相应的“ZK链接地址（内网）”。

----结束

1.5.4.2 创建 Configuration

功能介绍

HBase通过加载配置文件来获取配置项。

说明

1. 加载配置文件是一个比较耗时的操作，如非必要，请尽量使用同一个Configuration对象。
2. 样例代码未考虑多线程同步的问题，如有需要，请自行增加。其它样例代码也一样，不再一一进行说明。

代码样例

下面代码片段在com.huawei.cloudtable.lemonIndex.examples.LemonIndexTestMain包中。

```
private static void init() throws IOException {
  // Default load from conf directory
  conf = HBaseConfiguration.create(); // 注[1]
  String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
  Path hbaseSite = new Path(userdir + "hbase-site.xml");
  if (new File(hbaseSite.toString()).exists()) {
    conf.addResource(hbaseSite);
  }
}
```

注意事项

- 注[1] 如果配置文件目录conf已经加入classpath路径中，那么后面的加载指定配置文件的代码可以不执行。

1.5.4.3 创建数据表开启标签索引

功能介绍

建表功能同**创建表**，在此基础上，表属性配置标签索引schema。

样例代码

```
public void testCreateTable() {
  LOG.info("Entering testCreateTable.");
  HTableDescriptor tableDesc = new HTableDescriptor(tableName);
  HColumnDescriptor cdm = new HColumnDescriptor(FAM_M);
  cdm.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);
  tableDesc.addFamily(cdm);
  HColumnDescriptor cdn = new HColumnDescriptor(FAM_N);
  cdn.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);
  tableDesc.addFamily(cdn);

  // Add bitmap index definitions.
  List<BitmapIndexDescriptor> bitmaps = new ArrayList<>();//(1)
  bitmaps.add(BitmapIndexDescriptor.builder()
    // Describe which column should be indexed.
    .setColumnName(FamilyOnlyName.valueOf(FAM_M))//(2)
    // Describe how to extract term(s) from KeyValue
    .setTermExtractor(TermExtractor.NAME_VALUE_EXTRACTOR)//(3)
    .build());
  // It will help to add several properties into HTableDescriptor.
  // SHARD_NUM should be less than the region number
  IndexHelper.enableAutoIndex(tableDesc, SHARD_NUM, bitmaps);//(4)

  List<byte[]> splitList = Arrays.stream(SPLIT.split(LemonConstants.COMMA))
    .map(s -> org.lemon.common.Bytes.toBytes(s.trim()))
    .collect(Collectors.toList());
  byte[][] splitArray = splitList.toArray(new byte[splitList.size()][]);

  Admin admin = null;
  try {
    // Instantiate an Admin object.
    admin = conn.getAdmin();
    if (!admin.tableExists(tableName)) {
      LOG.info("Creating table...");
      admin.createTable(tableDesc, splitArray);
      LOG.info(admin.getClusterStatus());
      LOG.info(admin.listNamespaceDescriptors());
      LOG.info("Table created successfully.");
    } else {
      LOG.warn("table already exists");
    }
  } catch (IOException e) {
    LOG.error("Create table failed.", e);
  } finally {
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Failed to close admin ", e);
      }
    }
  }
  LOG.info("Exiting testCreateTable.");
}
```

注意事项

- (1) BitmapIndexDescriptor描述哪些字段使用什么规则来抽取标签，数据表可以定义一个或多个BitmapIndexDescriptor。
- (2) 定义哪些列需要抽取标签。取值范围：
  - ExplicitColumnName：指定列。
  - FamilyOnlyName：某一ColumnFamily下的所有列。
  - PrefixColumnName：拥有某一前缀的列。

- (3) 定义列的抽取标签的规则，可选值如下：
  - QualifierExtractor：表示按照列名来抽取标签。

    例如，qualifier是Male，value是1，那么抽取的标签是Male。
  - QualifierValueExtractor：表示按照列名和value来抽取标签。

    例如，qualifier是education，value是master，那么抽取的标签是education:master。
  - QualifierArrayValueExtractor：可以抽取多个标签，value是json array格式。

    例如，qualifier是hobby，value 是 ["basketball","football","volleyball"]，抽取的标签如下：

    ```
    hobby:basketball
    hobby:football
    hobby:volleyball
    ```
  - QualifierMapValueExtractor：可以抽取多个标签，value是json map格式。

    例如，qualifier是hobby，value是{"basketball":"9","football":"8","volleyball":"7"}，抽取的标签如下：

    ```
    hobby:basketball
    hobby:football
    hobby:volleyball
    hobby:basketball_9
    hobby:football_8
    hobby:volleyball_7
    ```
- (4) 索引表的分区数量SHARD_NUM必须要小于或等于数据表。

1.5.4.4 写入数据

写入数据接口和HBase原生API一致。

可以参考以下样例代码。

样例代码

```
public void testPut() {
  LOG.info("Entering testPut.");
  try(Table table = conn.getTable(tableName)) {
    List<Put> puts = new ArrayList<>();
    Arrays.stream(SPLIT.split(LemonConstants.COMMA))
      .forEach(startkey -> {
        // Instantiate a Put object.
        Put put = new Put(Bytes.toBytes(startkey + "-rowkey001"));
        put.addColumn(FAM_M, QUA_M, Bytes.toBytes("bachelor"));
        put.addColumn(FAM_N, QUA_N, Bytes.toBytes("xiaowang"));
        puts.add(put);

        Put put1 = new Put(Bytes.toBytes(startkey + "-rowkey0012"));
        put1.addColumn(FAM_M, QUA_M, Bytes.toBytes("master"));
        put1.addColumn(FAM_N, QUA_N, Bytes.toBytes("xiaoming"));
        puts.add(put1);

        // Submit a put request.
        try {
          table.put(puts);
        } catch (IOException e) {
          LOG.info("put exception", e);
        }
      });
    LOG.info("Put successfully.");
  } catch (IOException e) {
    LOG.error("Put failed ", e);
  }
```

```
  LOG.info("Exiting testPut.");
}
```

1.5.4.5 普通查询

功能介绍

CloudTable标签索引基于Lucene语法，提供了自研的查询接口LemonTable.query(LemonQuery query)。

样例代码

```
public void testNormalQuery() {
  LOG.info("Entering testNormalQuery.");

  try (Table table = conn.getTable(tableName)) {
    // Using Table instance to create LemonTable.
    LemonTable lemonTable = new LemonTable(table);
    // Build LemonQuery.
    LemonQuery query = LemonQuery.builder()
      // Set ad-hoc query condition.
      .setQuery("education:bachelor OR education:master") //(1)
      // Set how many rows should be cached on client for the initial request.
      .setCaching(10) //(2)
      // Set return column family/columns.
      .addFamily(FAM_M)
      .addColumn(FAM_N, QUA_N) //(3)
      // Set return return result just contains rowkeys, no any qualifier
      // the CF of LemonConstants.EMPTY_COLUMN_RETURN can be a random existing CF
      //.addColumn(FAM_M, LemonConstants.EMPTY_COLUMN_RETURN)
      .build();
    ResultSet resultSet = lemonTable.query(query);
    // Read result rows.

    int count = resultSet.getCount();
    LOG.info("the entity count of query is " + count);

    List<EntityEntry> entries = resultSet.listRows();
    for (EntityEntry entry : entries) {
      Map<String, Map<String, String>> fams = entry.getColumns();
      for (Map.Entry<String, Map<String, String>> familyEntry : fams.entrySet()) {
        String family = familyEntry.getKey();
        Map<String, String> qualifiers = familyEntry.getValue();
        for (Map.Entry<String, String> qualifier : qualifiers.entrySet()) {
          String Qua = qualifier.getKey();
          String value = qualifier.getValue();
          LOG.info("rowkey is " + Bytes.toString(entry.getRow()) + ", qualifier is "
            + family + ":" + Qua + ", value is " + value);
        }
      }
    }
  } catch (IOException e) {
    LOG.error("testNormalQuery failed ", e);
  }

  LOG.info("Exiting testNormalQuery.");
}
```

注意事项

(1) 查询条件，遵循Lucene语法/BNF范式，例如：

```
"Male ANDMarried AND AGE:25-30 AND BLOOD_TYPE:A"
"Male ANDMarried AND (AGE:25-30 OR AGE:30-35)AND BLOOD_TYPE:A"
```

(2) 查询返回多少行数据。

(3) 每行数据返回哪些qualifier，如果只设置列族，则表示返回列族下的所有列。

1.5.4.6 抽样查询

功能介绍

在**普通查询**的基础上设置setSampling()，查询时从索引表中随机选择一个分片执行查询任务。

可以参考以下样例代码。

样例代码

```
public void testSamplingQuery() {
  LOG.info("Entering testSamplingQuery.");

  try (Table table = conn.getTable(tableName)) {
    // Using Table instance to create LemonTable.
    LemonTable lemonTable = new LemonTable(table);
    // Build LemonQuery.
    LemonQuery query = LemonQuery.builder()
      // Set ad-hoc query condition.
      .setQuery("education:bachelor OR education:master")
      // Set how many rows should be cached on client for the initial request.
      .setCaching(10)
      // sampling query will be select one random shard/region to query
      .setSampling()
      // Set return column family/columns.
      .addFamily(FAM_M)
      .addColumn(FAM_N, QUA_N)
      // Set return return result just contains rowkeys, no any qualifier
      // the CF of LemonConstants.EMPTY_COLUMN_RETURN can be a random existing CF
      //.addColumn(FAM_M, LemonConstants.EMPTY_COLUMN_RETURN)
      .build();
    ResultSet resultSet = lemonTable.query(query);
    // Read result rows.
    int count = resultSet.getCount();
    LOG.info("the entity count of query is " + count);

    List<EntityEntry> entries = resultSet.listRows();
    for (EntityEntry entry : entries) {
      Map<String, Map<String, String>> fams = entry.getColumns();
      for (Map.Entry<String, Map<String, String>> familyEntry : fams.entrySet()) {
        String family = familyEntry.getKey();
        Map<String, String> qualifiers = familyEntry.getValue();
        for (Map.Entry<String, String> qualifier : qualifiers.entrySet()) {
          String Qua = qualifier.getKey();
          String value = qualifier.getValue();
          LOG.info("rowkey is " + Bytes.toString(entry.getRow()) + ", qualifier is "
            + family + ":" + Qua + ", value is " + value);
        }
      }
    }
  } catch (IOException e) {
    LOG.error("testSamplingQuery failed ", e);
  }

  LOG.info("Exiting testSamplingQuery.");
  LOG.info("");
}
public void testSamplingQuery() {
  LOG.info("Entering testSamplingQuery.");

  try (Table table = conn.getTable(tableName)) {
    // Using Table instance to create LemonTable.
    LemonTable lemonTable = new LemonTable(table);
```

```
  // Build LemonQuery.
  LemonQuery query = LemonQuery.builder()
    // Set ad-hoc query condition.
    .setQuery("education:bachelor OR education:master")
    // Set how many rows should be cached on client for the initial request.
    .setCaching(10)
    // sampling query will be select one random shard/region to query
    .setSampling()
    // Set return column family/columns.
    .addFamily(FAM_M)
    .addColumn(FAM_N, QUA_N)
    // Set return return result just contains rowkeys, no any qualifier
    // the CF of LemonConstants.EMPTY_COLUMN_RETURN can be a random existing CF
    //.addColumn(FAM_M, LemonConstants.EMPTY_COLUMN_RETURN)
    .build();
  ResultSet resultSet = lemonTable.query(query);
  // Read result rows.
  int count = resultSet.getCount();
  LOG.info("the entity count of query is " + count);

  List<EntityEntry> entries = resultSet.listRows();
  for (EntityEntry entry : entries) {
    Map<String, Map<String, String>> fams = entry.getColumns();
    for (Map.Entry<String, Map<String, String>> familyEntry : fams.entrySet()) {
      String family = familyEntry.getKey();
      Map<String, String> qualifiers = familyEntry.getValue();
      for (Map.Entry<String, String> qualifier : qualifiers.entrySet()) {
        String Qua = qualifier.getKey();
        String value = qualifier.getValue();
        LOG.info("rowkey is " + Bytes.toString(entry.getRow()) + ", qualifier is "
          + family + ":" + Qua + ", value is " + value);
      }
    }
  }
 } catch (IOException e) {
  LOG.error("testSamplingQuery failed ", e);
 }

 LOG.info("Exiting testSamplingQuery.");
 LOG.info("");
}
```

1.5.4.7 分页查询

功能介绍

先执行query接口返回简要数据信息，而后调用listRows接口翻页。

可以参考以下样例代码。

样例代码

```
public void testPagingQuery() {
 LOG.info("Entering testPagingQuery.");

 try (Table table = conn.getTable(tableName)) {
  // Using Table instance to create LemonTable.
  LemonTable lemonTable = new LemonTable(table);
  // Build LemonQuery.
  LemonQuery query = LemonQuery.builder()
    // Set ad-hoc query condition.
    .setQuery("education:bachelor OR education:master")
    // Set how many rows should be cached on client for the initial request.
    .setCaching(10)
    // Set return column family/columns.
    .addFamily(FAM_M)
    .addColumn(FAM_N, QUA_N)
```

```
    // Set return return result just contains rowkeys, no any qualifier
    // the CF of LemonConstants.EMPTY_COLUMN_RETURN can be a random existing CF
    //.addColumn(FAM_M, LemonConstants.EMPTY_COLUMN_RETURN)
    .build();
  ResultSet resultSet = lemonTable.query(query);
  // Read result rows.
  int count = resultSet.getCount();
  LOG.info("the entity count of query is " + count);

  // Read result page by page, every page show 10 lines data
  int maxPage = 100;
  final int lineNumPerPage = 5;
  for (int i = 0; i < maxPage; i++) {
    int start = lineNumPerPage * i;
    List<EntityEntry> entries = resultSet.listRows(start, lineNumPerPage);
    if (entries == null || entries.size() == 0)
    {
      break;
    }

    LOG.info("page " + (i + 1) + " count is " + entries.size() + ", result is following:");
    for (EntityEntry entry : entries) {
      Map<String, Map<String, String>> fams = entry.getColumns();
      for (Map.Entry<String, Map<String, String>> familyEntry : fams.entrySet()) {
        String family = familyEntry.getKey();
        Map<String, String> qualifiers = familyEntry.getValue();
        for (Map.Entry<String, String> qualifier : qualifiers.entrySet()) {
          String Qua = qualifier.getKey();
          String value = qualifier.getValue();
          LOG.info("rowkey is " + Bytes.toString(entry.getRow()) + ", qualifier is "
            + family + ":" + Qua + ", value is " + value);
        }
      }
    }
  }
 } catch (IOException e) {
  LOG.error("testPagingQuery failed ", e);
 }

 LOG.info("Exiting testPagingQuery.");
}
```

1.5.4.8 统计查询

功能介绍

返回满足查询条件的实体总量，不返回数据的具体信息，代码中设置setCountOnly()。

可以参考以下样例代码。

样例代码

```
public void testCountOnlyQuery() {
 LOG.info("Entering testCountOnlyQuery.");

 try (Table table = conn.getTable(tableName)) {
  // Using Table instance to create LemonTable.
  LemonTable lemonTable = new LemonTable(table);
  // Build LemonQuery.
  LemonQuery query = LemonQuery.builder()
    // Set ad-hoc query condition.
    .setQuery("education:bachelor OR education:master")
    // just return how many entities meet the query condition, without any rowkey/column
    .setCountOnly()
    .build();
```

```
  ResultSet resultSet = lemonTable.query(query);
  // Read result rows.
  int count = resultSet.getCount();
  LOG.info("the entity count of query is " + count);
 } catch (IOException e) {
  LOG.error("testCountOnlyQuery failed ", e);
 }

 LOG.info("Exiting testCountOnlyQuery.");
}
```

1.6 调测程序

1.6.1 在 Windows 中调测程序

1.6.1.1 编译并运行程序

操作场景

在程序代码完成开发后，您可以在Windows开发环境中运行应用。

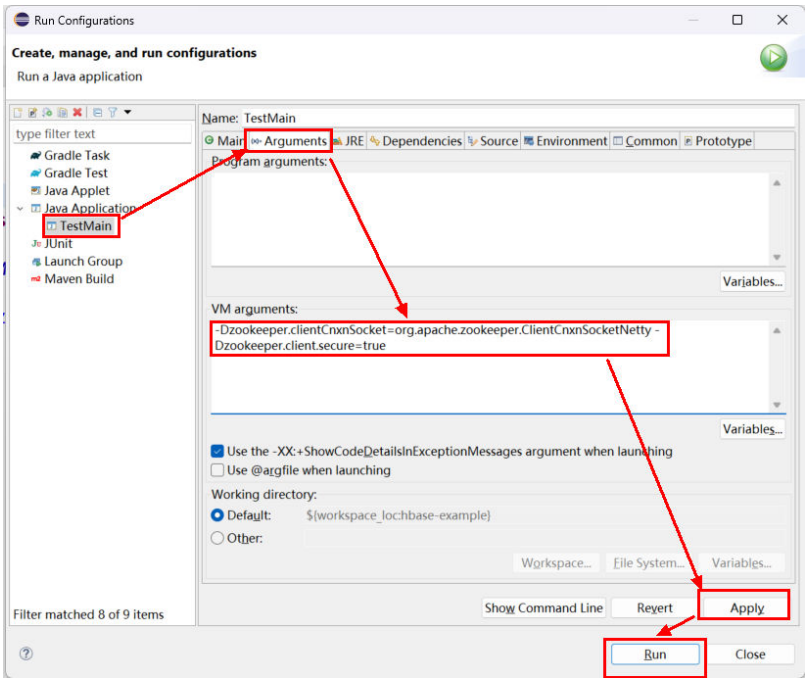
操作步骤

- **未开启加密通道的HBase集群**

  在开发环境中（例如Eclipse中），右击“TestMain.java”，单击“Run as > Java Application”运行对应的应用程序工程。

- **开启加密通道的HBase集群**

  在开发环境中（例如Eclipse中），右击“TestMain.java”，单击“Run as > Run Configurations”，参考如下截图添加环境变量"-Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty -Dzookeeper.client.secure=true"后，再运行对应的应用程序工程。



1.6.1.2 查看调测结果

运行结果中没有异常或失败信息即表明运行成功。

图 1-8 运行成功

```
2016-07-13 14:36:12,736 INFO  [main] basic.CreateTableSample: Create table sampleNameSpace:sampleTable successful!
2016-07-13 14:36:15,426 INFO  [main] basic.ModifyTableSample: Modify table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:16,708 INFO  [main] basic.MultiSplitSample: Mmulti split table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:17,299 INFO  [main] basic.PutDataSample: Successfully put 9 items data into sampleNameSpace:sampleTable.
2016-07-13 14:36:18,992 INFO  [main] basic.ScanSample: Scan data successfully.
2016-07-13 14:36:20,532 INFO  [main] basic.DeletaDataSample: Successfully delete data from table sampleNameSpace:sampleTable.
2016-07-13 14:36:21,006 INFO  [main] acl.AclSample: Grant ACL for table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:27,836 INFO  [main] index.CreateIndexSample: Successfully add index for table sampleNameSpace:sampleTable.
```

说明

在Windows环境运行样例代码时会出现下面的异常，但是不影响业务：

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

日志说明：

日志级别默认为INFO，可以通过调整日志打印级别（DEBUG，INFO，WARN，ERROR，FATAL）来显示更详细的信息。可以通过修改log4j.properties文件来实现，如：

```
hbase.root.logger=INFO,console
log4j.logger.org.apache.zookeeper=INFO
#log4j.logger.org.apache.hadoop.fs.FSNamesystem=DEBUG
log4j.logger.org.apache.hadoop.hbase=INFO
# Make these two classes DEBUG-level. Make them DEBUG to see more zk debug.
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZKUtil=INFO
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZooKeeperWatcher=INFO
```

1.6.2 在 Linux 中调测程序

1.6.2.1 安装客户端时编译并运行程序

操作场景

HBase应用程序支持在安装HBase客户端的Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至Linux环境中运行应用。

前提条件

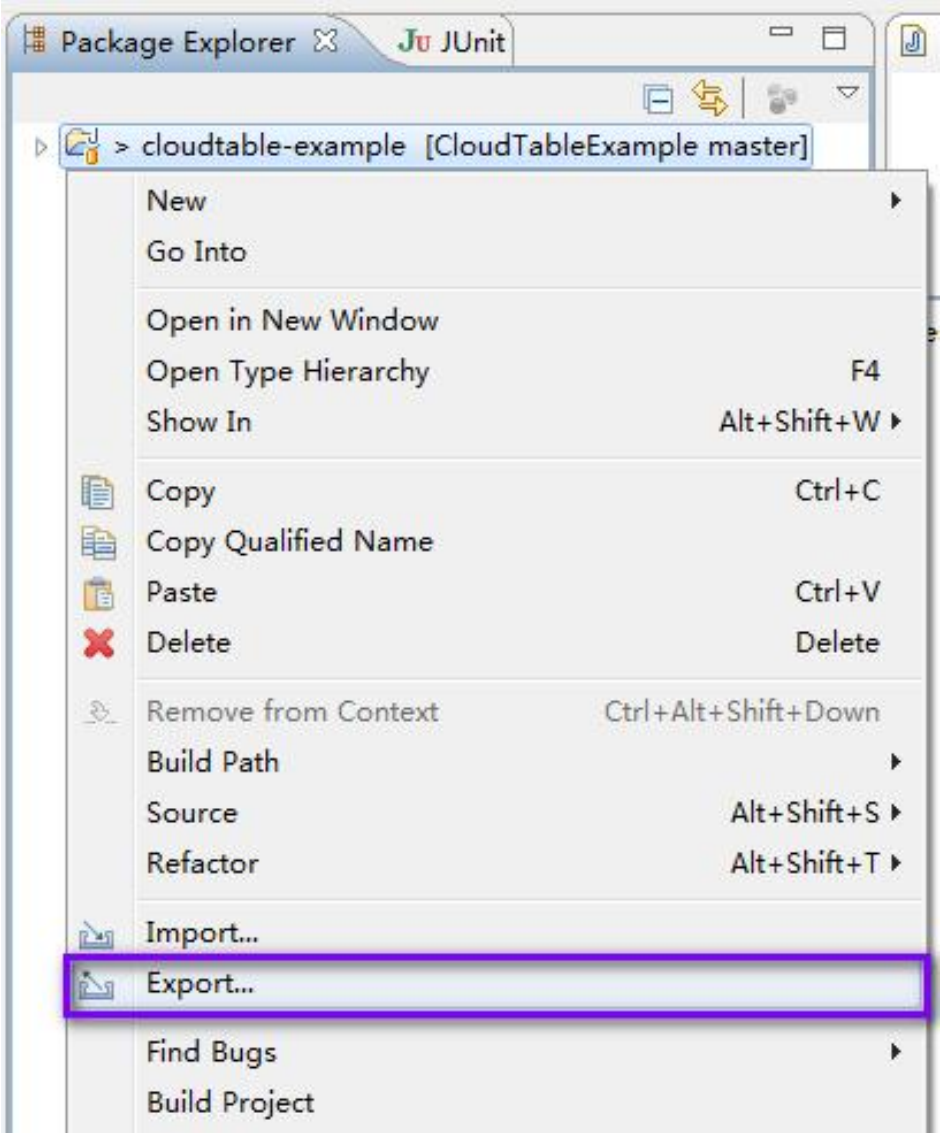
- 已安装HBase客户端。
- Linux环境已安装JDK，版本号需要和Eclipse导出Jar包使用的JDK版本一致。

操作步骤

步骤1 导出Jar包。

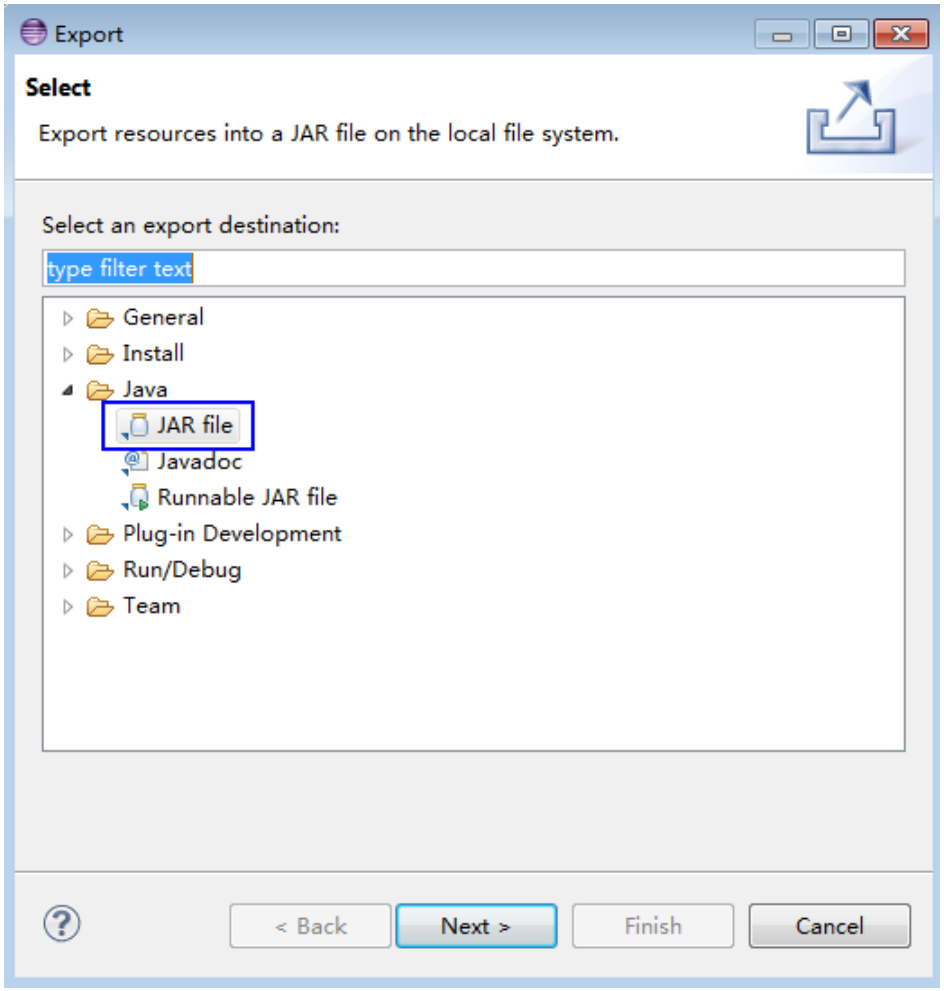
1. 右击样例工程，选择导出。

图 1-9 导出 Jar 包



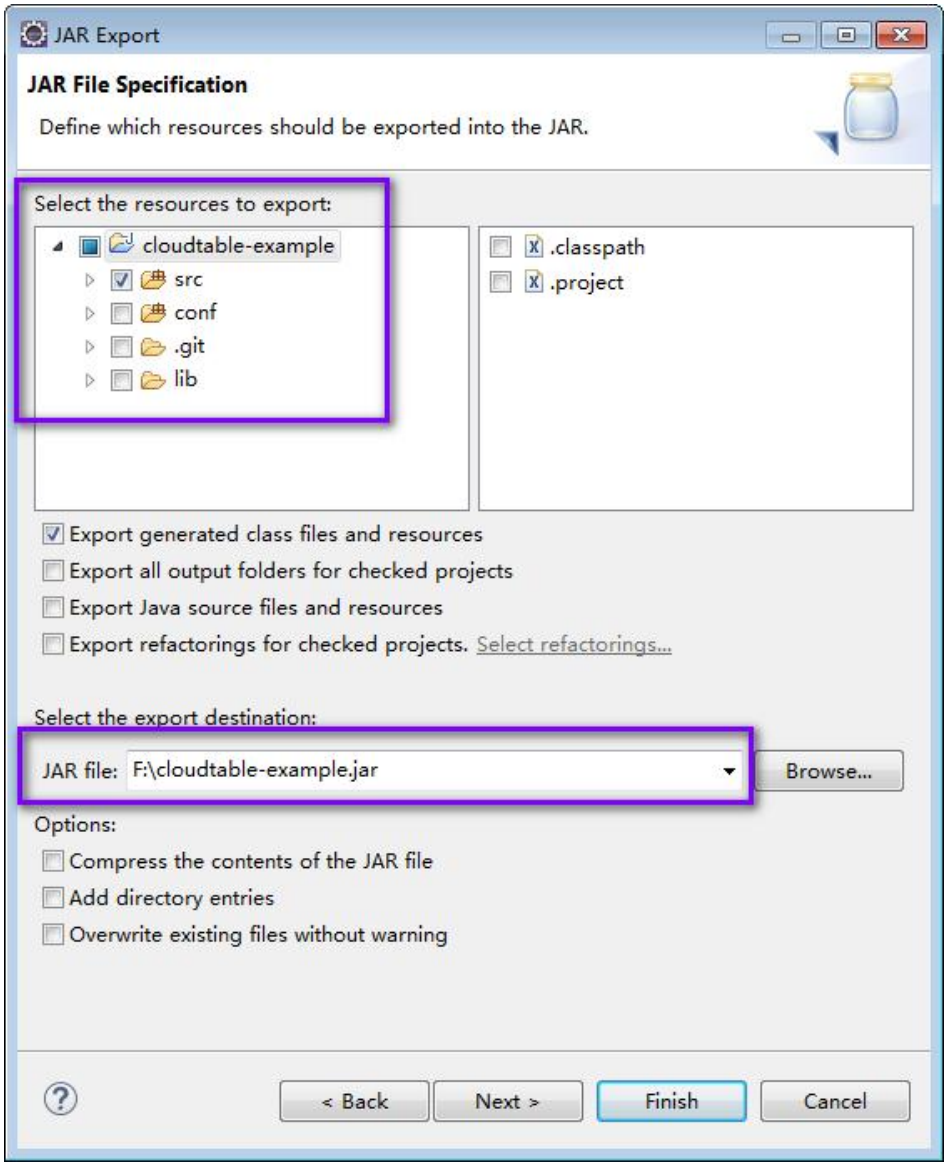
2. 选择JAR file，单击“Next”。

图 1-10 选择 JAR file



3. 勾选“src”和“conf”目录，导出Jar包到指定位置。单击两次“Next”。

图 1-11 选择导出路径



4. 单击“Finish”，完成导出Jar包。

步骤2 执行Jar包。

1. 在Linux客户端下执行Jar包的时候，先将应用开发环境中生成的Jar包拷贝上传至客户端安装目录的“lib”目录中，并确保Jar包的文件权限与其它文件相同。
2. 用安装用户切换到客户端目录的“bin”目录下，然后运行如下命令使Jar包执行：

```
[Ruby@cloudtable-08261700-hmaster-1-1 bin]# ./hbase
com.huawei.cloudtable.hbase.examples.TestMain
```

其中，*com.huawei.cloudtable.hbase.examples.TestMain*为举例，具体以实际样例代码为准。

----结束

1.6.2.2 未安装客户端时编译并运行程序

操作场景

HBase应用程序支持在未安装HBase客户端的Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至Linux环境中运行应用。

前提条件

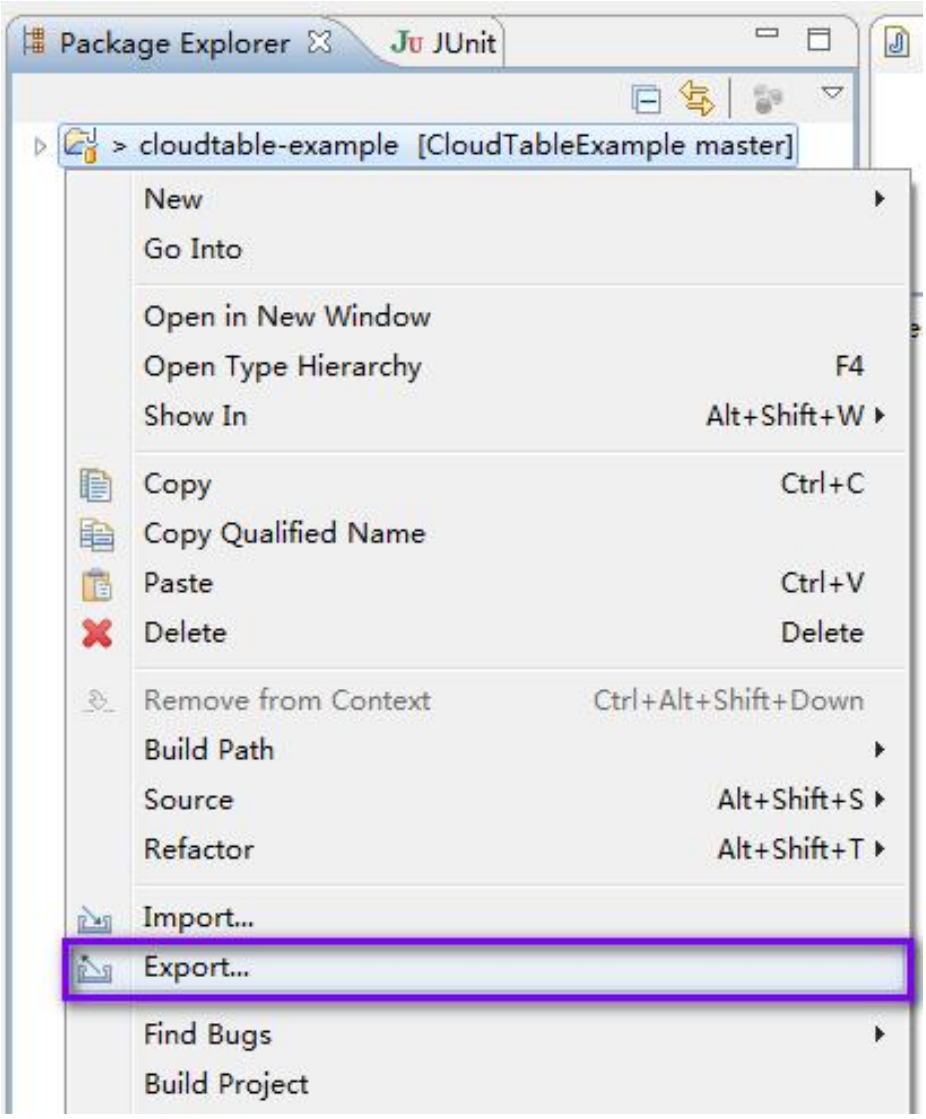
Linux环境已安装JDK，版本号需要和Eclipse导出Jar包使用的JDK版本一致。

操作步骤

步骤1 导出Jar包。

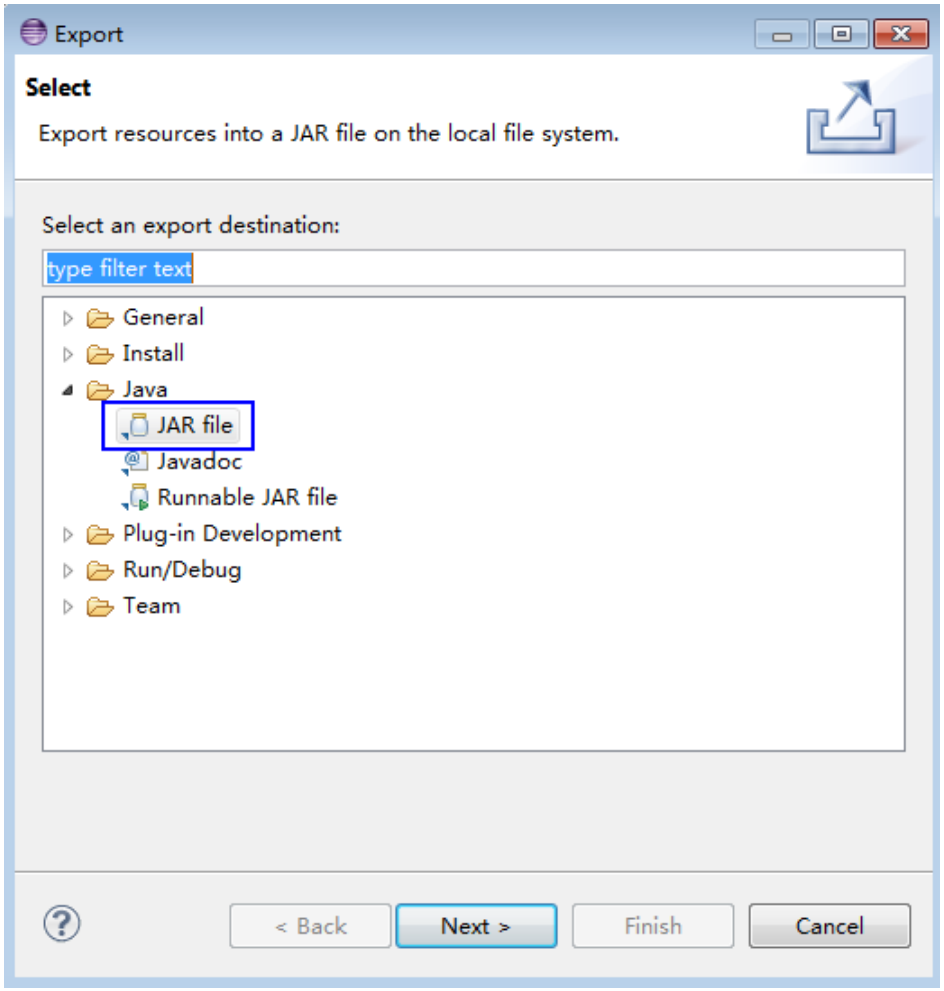
1. 右击样例工程，选择导出。

图 1-12 导出 Jar 包



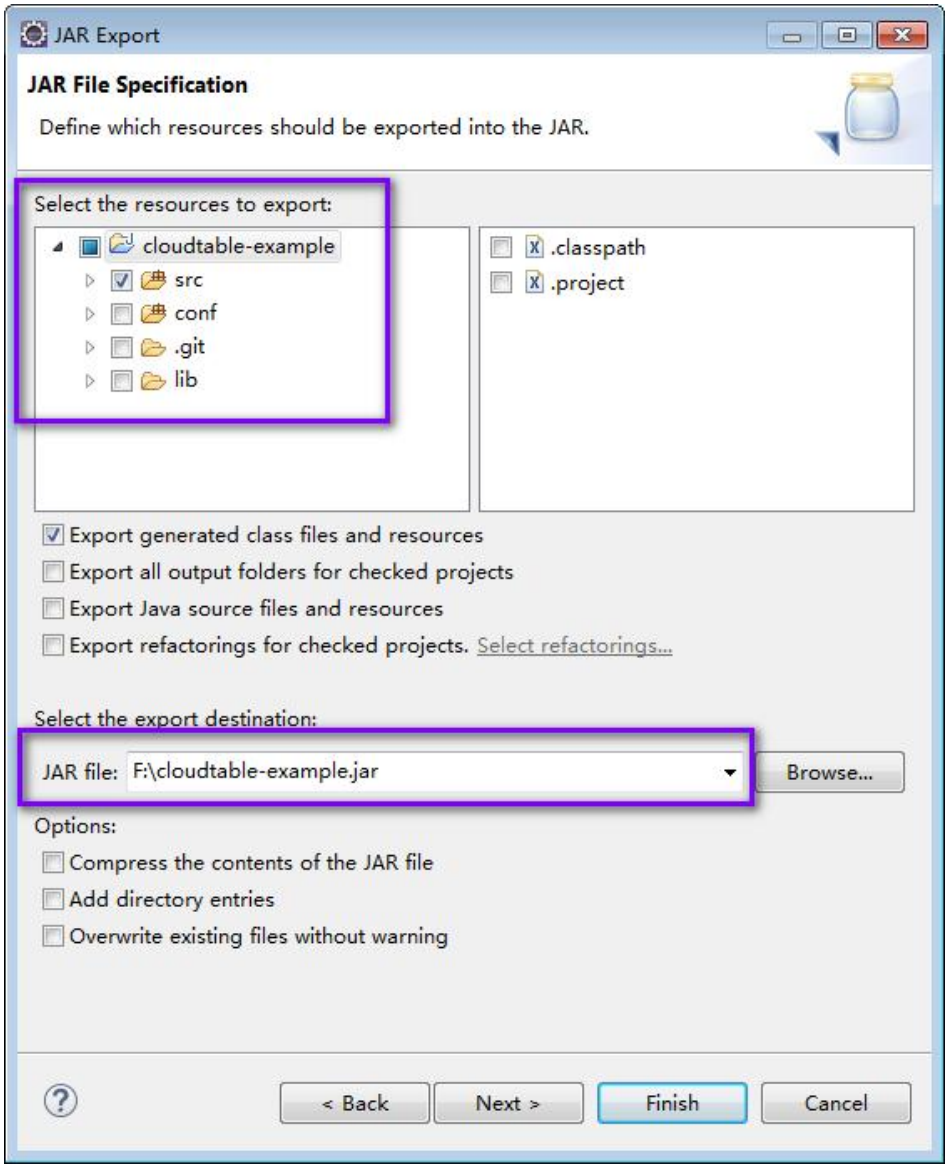
2. 选择JAR file，单击“Next”。

图 1-13 选择 JAR file



3. 勾选“src”目录，导出Jar包到指定位置。单击两次“Next”。

图 1-14 选择导出路径



4. 单击“Finish”，完成导出Jar包。

步骤2 准备依赖的Jar包和配置文件。

1. 在Linux环境新建目录，例如“/opt/test”，并创建子目录“lib”和“conf”。将样例工程中“lib”的Jar包，以及**步骤1**导出的Jar包，上传到Linux的“lib”目录。将样例工程中“conf”的配置文件上传到Linux中“conf”目录。
2. 在“/opt/test”根目录新建脚本“run.sh”，修改内容如下并保存：

```
#!/bin/sh
BASEDIR=`pwd`
SECURE=""
if [ $# -eq 1 ]; then
  SECURE="-Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty -
Dzookeeper.client.secure=true"
fi
cd ${BASEDIR}
for file in ${BASEDIR}/lib/*.jar
do
i_cp=$i_cp:$file
echo "$file"
```



```
done
for file in ${BASEDIR}/conf/*
do
i_cp=$i_cp:$file
done
java -cp .${i_cp} ${SECURE} com.huawei.cloudtable.hbase.examples.TestMain
```

步骤3 切换到“/opt/test”，执行以下命令，运行Jar包。

- **未开启加密通道的HBase集群**

  sh run.sh

- **开启加密通道的HBase集群**

  sh run.sh secure

  说明

  如果使用其他方式运行应用访问开启了加密通道的HBase集群，需要自行添加JVM参数："-Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty -Dzookeeper.client.secure=true

----结束

1.6.2.3 查看调测结果

运行结果中没有异常或失败信息即表明运行成功。

图 1-15 运行成功

```
2016-07-13 14:36:12,736 INFO  [main] basic.CreateTableSample: Create table sampleNameSpace:sampleTable successful!
2016-07-13 14:36:15,426 INFO  [main] basic.ModifyTableSample: Modify table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:16,708 INFO  [main] basic.MultiSplitSample: Mmulti split table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:17,299 INFO  [main] basic.PutDataSample: Successfully put 9 items data into sampleNameSpace:sampleTable.
2016-07-13 14:36:18,992 INFO  [main] basic.ScanSample: Scan data successfully.
2016-07-13 14:36:20,532 INFO  [main] basic.DeletaDataSample: Successfully delete data from table sampleNameSpace:sampleTable.
2016-07-13 14:36:21,006 INFO  [main] acl.AclSample: Grant ACL for table sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:27,836 INFO  [main] index.CreateIndexSample: Successfully add index for table sampleNameSpace:sampleTable.
```

日志说明：日志级别默认为INFO，可以通过调整日志打印级别（DEBUG，INFO，WARN，ERROR，FATAL）来显示更详细的信息。可以通过修改log4j.properties文件来实现，如：

```
hbase.root.logger=INFO,console
log4j.logger.org.apache.zookeeper=INFO
#log4j.logger.org.apache.hadoop.fs.FSNamesystem=DEBUG
log4j.logger.org.apache.hadoop.hbase=INFO
# Make these two classes DEBUG-level. Make them DEBUG to see more zk debug.
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZKUtil=INFO
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZooKeeperWatcher=INFO
```

1.7 对外接口

1.7.1 HBase Java API

HBase采用的接口与Apache HBase保持一致。

详细内容请参见 **https://hbase.apache.org/1.2/apidocs/index.html**

2 ClickHouse 应用开发指导

2.1 ClickHouse 表引擎概述

背景介绍

表引擎在ClickHouse中的作用十分关键，不同的表引擎决定了：

- 数据存储和读取的位置。
- 支持哪些查询方式。
- 能否并发式访问数据。
- 能否使用索引。
- 是否可以执行多线程请求。
- 数据复制使用的参数。

其中MergeTree和Distributed是ClickHouse表引擎中最重要，也是最常使用的两个引擎，本文将重点进行介绍。

概述

表引擎即表的类型，在云数据库ClickHouse中决定了如何存储和读取数据、是否支持索引、是否支持主备复制等。云数据库ClickHouse支持的表引擎，请参见下表。

表 2-1 表引擎

系列	描述	表引擎	特点
MergeTree	• MergeTree系列引擎适用于高负载任务，支持大数据量的快速写入并进行后续的数据处理，通用程度高且功能强大。 • 该系列引擎的共同特点是支持数据副本、分区、数据采样等特性。	MergeTree	• 基于分区键（partitioning key）的数据分区分块存储。 • 数据索引排序（基于primary key和order by）。 • 支持数据复制（带Replicated前缀的表引擎）。 • 支持数据抽样。 在写入数据时，该系列引擎表会按照分区键将数据分成不同的文件夹，文件夹内每列数据为不同的独立文件，以及创建数据的序列化索引排序记录文件。该结构使得数据读取时能够减少数据检索时的数据量，极大的提高查询效率。
		RelacingMergeTree	用于解决MergeTree表引擎相同主键无法去重的问题，可以删除主键值相同的重复项。
		CollapsingMergeTree	CollapsingMergeTree它通过定义一个sign标记位字段记录数据行的状态。如果sign标记为1，则表示这是一行有效的数据。如果sign标记为-1，则表示这行数据需要被删除。
		VersionedCollapsingMergeTree	在建表语句中新增Version列，用于解决CollapsingMergeTree表引擎乱序写入导致无法正常折叠（删除）的问题。
		SummigMergeTree	用于对主键列进行预先聚合，将所有相同主键的行合并为一行，从而大幅度降低存储空间占用，提升聚合计算性能。
		AggregatingMergeTree	AggregatingMergeTree是预先聚合引擎的一种，用于提升聚合计算的性能。AggregatingMergeTree引擎能够在合并分区时，按照预先定义的条件聚合数据，同时根据预先定义的聚合函数计算数据并通过二进制的格式存入表内。
		GraphiteMergeTree	用于存储Graphite数据并进行汇总，可以减少存储空间，提高Graphite数据的查询效率。

系列	描述	表引擎	特点
Replicated*MergeTree	ClickHouse中的所有MergeTree家族引擎前面加上Replicated就成了支持副本的合并树引擎。	Replicated*MergeTree系列	Replicated系列引擎借助ZooKeeper实现数据的同步，创建Replicated复制表时通过注册到ZooKeeper上的信息实现同一个分片的所有副本数据进行同步。
Distributed	-	Distributed	本身不存储数据，可以在多个服务器上进行分布式查询。

MergeTree

- 建表语法。

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse集群名]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1] [TTL expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2] [TTL expr2],
    ...
    INDEX index_name1 expr1 TYPE type1(...) GRANULARITY value1,
    INDEX index_name2 expr2 TYPE type2(...) GRANULARITY value2
) ENGINE = MergeTree()
ORDER BY expr
[PARTITION BY expr]
[PRIMARY KEY expr]
[SAMPLE BY expr]
[TTL expr [DELETE|TO DISK 'xxx'|TO VOLUME 'xxx'], ...]
[SETTINGS name=value, ...]
```

- 使用示例。

```
CREATE TABLE default.test (name1 DateTime,name2 String,name3 String,name4 String,name5 Date) ENGINE = MergeTree() PARTITION BY toYYYYMM(name5) ORDER BY (name1, name2) SETTINGS index_granularity = 8192;
```

示例参数说明：

表 2-2 参数说明

参数	说明
ENGINE = MergeTree()	MergeTree表引擎。
PARTITION BY toYYYYMM(name5)	分区，示例数据将以月份为分区，每个月份一个文件夹。
ORDER BY	排序字段，支持多字段的索引排序，第一个相同的时候按照第二个排序依次类推。
index_granularity = 8192	排序索引的颗粒度，每8192条数据记录一个排序索引值。

说明

如果被查询的数据存在于分区或排序字段中，能极大降低数据查找时间。

ReplacingMergeTree

为了解决MergeTree表引擎相同主键无法去重的问题，云数据库ClickHouse提供了ReplacingMergeTree表引擎，用于删除主键值相同的重复项。

- 建表语句。

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse集群名]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = ReplacingMergeTree([ver])
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

CollapsingMergeTree

CollapsingMergeTree表引擎用于消除ReplacingMergeTree表引擎的功能限制。该表引擎要求在建表语句中指定一个标记列Sign，按照Sign的值将行分为两类：Sign=1的行称为状态行，用于新增状态。Sign=-1的行称为取消行，用于删除状态。

- 建表语句。

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse集群名]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = CollapsingMergeTree(sign)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

- 使用示例。
  - 示例数据。

    例如：我们要计算用户在某个网站上访问了多少页面以及他们在那里的时间。 在某个时间点，我们用用户活动的状态写下面的行。

    表 2-3 示例数据

UserID	PageViews	Duration	Sign
4324182021466249494	5	146	1
4324182021466249494	5	146	-1
4324182021466249494	6	185	1

    - sign— 指定行类型的列名: 1 是一个 “state” 行, -1 是一个“cancel”行。
  - 建表Test。

```
CREATE TABLE Test(UserID UInt64,PageViews UInt8,Duration UInt8,Sign Int8)ENGINE =
CollapsingMergeTree(Sign) ORDER BY UserID;
```

- 插入数据。
  - 第一次插入数据。
    ```
    INSERT INTO Test VALUES (4324182021466249494, 5, 146, 1);
    ```
  - 第二次插入数据。
    ```
    INSERT INTO Test VALUES (4324182021466249494, 5, 146, -1),(4324182021466249494, 6,
    185, 1);
    ```
- 查看数据。
  ```
  SELECT * FROM Test;
  ```
  查询结果。
  ```
  ┌──────────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
  │ 4324182021466249494 │         5 │      146 │   -1 │
  │ 4324182021466249494 │         6 │      185 │    1 │
  └─────────────────────┴───────────┴──────────┴──────┘

  ┌──────────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
  │ 4324182021466249494 │         5 │      146 │    1 │
  └─────────────────────┴───────────┴──────────┴──────┘
  ```
- 对指定列进行数据聚合。
  ```
  SELECT UserID,sum(PageViews * Sign) AS PageViews,sum(Duration * Sign) AS Duration FROM
  Test GROUP BY UserID HAVING sum(Sign) > 0;
  ```
  查询结果如下所示。
  ```
  ┌──────────────────UserID─┬─PageViews─┬─Duration─┐
  │ 4324182021466249494 │         6 │      185 │
  └─────────────────────┴───────────┴──────────┘
  ```
- 强制折叠数据，用以下SQL命令。
  ```
  SELECT * FROM Test FINAL;
  ```
  查询结果如下所示。
  ```
  ┌──────────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┐
  │ 4324182021466249494 │         6 │      185 │    1 │
  └─────────────────────┴───────────┴──────────┴──────┘
  ```

VersionedCollapsingMergeTree

为了解决CollapsingMergeTree表引擎乱序写入导致无法正常折叠（删除）问题，云数据库ClickHouse提供了VersionedCollapsingMergeTree表引擎，在建表语句中新增一列Version，用于在乱序情况下记录状态行与取消行的对应关系。后台Compaction时会将主键相同、Version相同、Sign相反的行折叠（删除）。

- 建表语句。
  ```
  CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse集群名]
  (
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
  ) ENGINE = VersionedCollapsingMergeTree(sign, version)
  [PARTITION BY expr]
  [ORDER BY expr]
  [SAMPLE BY expr]
  [SETTINGS name=value, ...]
  ```
- 使用示例。
  - 示例数据。

    例如：我们要计算用户在某个网站上访问了多少页面以及他们在那里的时间。 在某个时间点，我们用用户活动的状态写下面的行。

表 2-4 示例数据

UserID	PageViews	Duration	Sign	Version
4324182021466249494	5	146	1	1
4324182021466249494	5	146	-1	1
4324182021466249494	6	185	1	2

  - sign— 指定行类型的列名: 1 是一个 “state” 行, -1 是一个“cancel”行。
  - version— 指定对象状态版本的列名。

- 创建表T。
  ```
  CREATE TABLE T(UserID UInt64,PageViews UInt8,Duration UInt8,Sign Int8,Version UInt8)ENGINE = VersionedCollapsingMergeTree(Sign, Version)ORDER BY UserID;
  ```
- 插入两部分不同的数据。
  ```
  INSERT INTO T VALUES (4324182021466249494, 5, 146, 1, 1);
  INSERT INTO T VALUES (4324182021466249494, 5, 146, -1, 1),(4324182021466249494, 6, 185, 1, 2);
  ```
- 查看数据。
  ```
  SELECT * FROM T;
  ```
- 对指定列进行数据聚合。
  ```
  SELECT UserID, sum(PageViews * Sign) AS PageViews,sum(Duration * Sign) AS Duration,Version FROM T GROUP BY UserID, Version HAVING sum(Sign) > 0;
  ```

  查询显示结果如下。

  ```
  ┌──────────────UserID─┬─PageViews─┬─Duration─┬─Version─┐
  │ 4324182021466249494 │         6 │      185 │       2 │
  └─────────────────────┴───────────┴──────────┴─────────┘
  ```
- 强制折叠数据，用以下SQL命令。
  ```
  SELECT * FROM T FINAL;
  ```

  查询显示结果如下。

  ```
  ┌──────────────UserID─┬─PageViews─┬─Duration─┬─Sign─┬─Version─┐
  │ 4324182021466249494 │         6 │      185 │    1 │       2 │
  └─────────────────────┴───────────┴──────────┴──────┴─────────┘
  ```

SummingMergeTree

SummingMergeTree表引擎用于对主键列进行预先聚合，将所有相同主键的行合并为一行，从而大幅度降低存储空间占用，提升聚合计算性能。

- 建表语句。
  ```
  CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse集群名]
  (
     name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
     name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
     ...
  ) ENGINE = SummingMergeTree([columns])
  [PARTITION BY expr]
  [ORDER BY expr]
  ```

```
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

- 使用示例。
  - 创建一个SummingMergeTree表testTable。
    ```
    CREATE TABLE testTable(id UInt32,value UInt32)ENGINE = SummingMergeTree() ORDER BY id;
    ```
  - testTable表中插入数据。
    ```
    INSERT INTO testTable Values(5,9),(5,3),(4,6),(1,2),(2,5),(1,4),(3,8);
    INSERT INTO testTable Values(88,5),(5,5),(3,7),(3,5),(1,6),(2,6),(4,7),(4,6),(43,5),(5,9),(3,6);
    ```
  - 在未合并parts查询所有数据。
    ```
    SELECT * FROM testTable;
    ```
    查询结果。
    ```
    ┌─id─┬─value─┐
    │  1 │     6 │
    │  2 │     5 │
    │  3 │     8 │
    │  4 │     6 │
    │  5 │    12 │
    └────┴───────┘
    ┌─id─┬─value─┐
    │  1 │     6 │
    │  2 │     6 │
    │  3 │    18 │
    │  4 │    13 │
    │  5 │    14 │
    │ 43 │     5 │
    │ 88 │     5 │
    └────┴───────┘
    ```
  - ClickHouse还没有汇总所有行，需要通过ID进行汇总聚合，需要用到sum和GROUP BY子句。
    ```
    SELECT id, sum(value) FROM testTable GROUP BY id;
    ```
    查询结果。
    ```
    ┌─id─┬─sum(value)─┐
    │  4 │         19 │
    │  3 │         26 │
    │ 88 │          5 │
    │  2 │         11 │
    │  5 │         26 │
    │  1 │         12 │
    │ 43 │          5 │
    └────┴────────────┘
    ```
  - 手工执行合并操作。
    ```
    OPTIMIZE TABLE testTable;
    ```
    查询表数据。
    ```
    SELECT * FROM testTable;
    ```
    查询结果。
    ```
    ┌─id─┬─value─┐
    │  1 │    12 │
    │  2 │    11 │
    │  3 │    26 │
    │  4 │    19 │
    │  5 │    26 │
    │ 43 │     5 │
    │ 88 │     5 │
    └────┴───────┘
    ```

说明

- SummingMergeTree根据ORDER BY排序键作为聚合数据的条件Key。即如果排序key是相同的，则会合并成一条数据，并对指定的合并字段进行聚合。
- 后台执行合并操作时才会进行数据的预先聚合，而合并操作的执行时机无法预测，所以可能存在部分数据已经被预先聚合、部分数据尚未被聚合的情况。因此，在执行聚合计算时，SQL中仍需要使用GROUP BY子句。

AggregatingMergeTree

AggregatingMergeTree表引擎也是预先聚合引擎的一种，用于提升聚合计算的性能。

- 建表语句。

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER ClickHouse集群名]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = AggregatingMergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[TTL expr]
[SETTINGS name=value, ...]
```

- 使用示例。

  AggregatingMergeTree无单独参数设置，在分区合并时，在每个数据分区内，会按照ORDER BY聚合，使用何种聚合函数，对哪些列字段计算，则是通过定义AggregateFunction函数类型实现。

  - 建表。

    ```
    create table test_table (name1 String,name2 String,name3 AggregateFunction(uniq,String),name4 AggregateFunction(sum,Int),name5 DateTime) ENGINE = AggregatingMergeTree() PARTITION BY toYYYYMM(name5) ORDER BY (name1,name2) PRIMARY KEY name1;
    ```

  AggregateFunction类型的数据在写入和查询时需要分别调用*state、*merge函数，*表示定义字段类型时使用的聚合函数。如上示例表test_table定义的name3、name4字段分别使用了uniq、sum函数，那么在写入数据时需要调用uniqState、sumState函数，并使用INSERT SELECT语法。

  - 插入数据。

    ```
    insert into test_table select '8','test1',uniqState('name1'),sumState(toInt32(100)),'2021-04-30 17:18:00';
    insert into test_table select '8','test1',uniqState('name1'),sumState(toInt32(200)),'2021-04-30 17:18:00';
    ```

  - 查询数据。

    ```
    select name1,name2,uniqMerge(name3),sumMerge(name4) from test_table group by name1,name2;
    ```

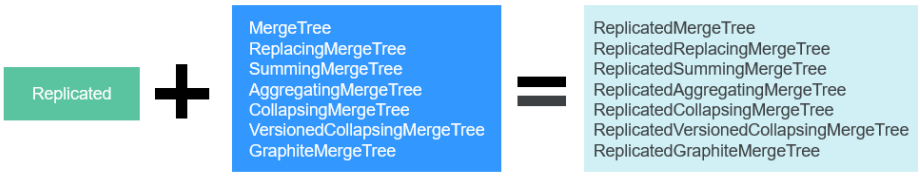
    查询结果。

    ```
    ┌─name1─┬─name2─┬─uniqMerge(name3)─┬─sumMerge(name4)─┐
    │ 8     │ test1 │                1 │             300 │
    └───────┴───────┴──────────────────┴─────────────────┘
    ```

Replicated*MergeTree 引擎

ClickHouse中的所有MergeTree家族引擎前面加上Replicated就成了支持副本的合并树引擎。

图 2-1 合并树引擎图



- Replicated表引擎的创建模板：
  ENGINE = Replicated*MergeTree('ZooKeeper存储路径','副本名称', ...)

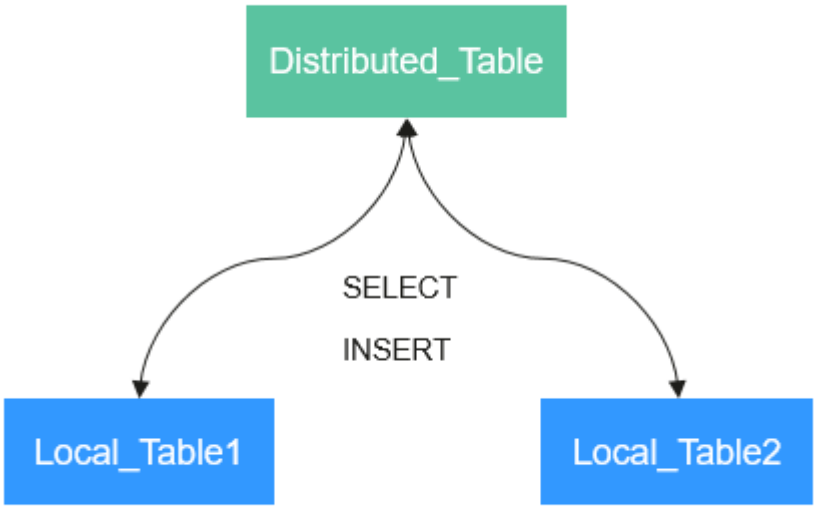
  表 2-5 参数表

参数	说明
ZooKeeper存储路径	ZooKeeper中该表相关数据的存储路径，建议规范化，如：/clickhouse/tables/{shard}/数据库名/表名。
副本名称	一般用{replica}即可。

Distributed 表引擎

Distributed表引擎本身不存储任何数据，而是作为数据分片的透明代理，能够自动路由数据到集群中的各个节点，分布式表需要和其他本地数据表一起协同工作。分布式表会将接收到的读写任务分发到各个本地表，而实际上数据的存储在各个节点的本地表中。

图 2-2 Distributed



- Distributed表引擎创建模板：
```
ENGINE = Distributed(cluster_name, database_name, table_name, [sharding_key])
```

表 2-6 Distributed 表参数说明

参数	说明
cluster_name	集群名称，在对分布式表执行读写的过程中，使用集群的配置信息查找对应的ClickHouse实例节点。
database_name	数据库名称。
table_name	数据库下对应的本地表名称，用于将分布式表映射到本地表上。
sharding_key	分片键（可选参数），分布式表会按照这个规则，将数据分发到各个本地表中。

- 使用示例。
  - 先创建一个表名为demo的ReplicatedMergeTree本地表。
```
CREATE TABLE default.demo ON CLUSTER default_cluster( `EventDate` DateTime, `id` UInt64)ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/default/demo', '{replica}') PARTITION BY toYYYYMM(EventDate) ORDER BY id;
```
  - 基于本地表demo创建表名为demo_all的Distributed表。
```
CREATE TABLE default.demo_all ON CLUSTER default_cluster( `EventDate` DateTime, `id` UInt64)ENGINE = Distributed(default_cluster, default, demo, rand());
```
- 分布式表创建规则。
  - 创建Distributed表时需加上on cluster cluster_name，这样建表语句在某一个ClickHouse实例上执行一次即可分发到集群中所有实例上执行。
  - 分布式表通常以本地表加“_all”命名。它与本地表形成一对多的映射关系，之后可以通过分布式表代理操作多张本地表。
  - 分布式表的表结构尽量和本地表的结构一致。如果不一致，在建表时不会报错，但在查询或者插入时可能会抛出异常。

2.2 SQL 语法参考

2.2.1 数据类型

此章节描述ClickHouse的数据类型。

数据类型表

表 2-7 数据类型表

分类	关键字	数据类型	描述
整数类型	Int8	Int8	取值范围：【-128，127】
	Int16	Int16	取值范围：【-32768，32767】

分类	关键字	数据类型	描述
	Int32	Int32	取值范围：【-2147483648，2147483647】
	Int64	Int64	取值范围：【-9223372036854775808，9223372036854775807】
浮点类型	Float32	单精度浮点数	同C语言Float类型，单精度浮点数在机内占4个字节，用32位二进制描述。
	Float64	双精度浮点数	同C语言Double类型，双精度浮点数在机内占8个字节，用64位二进制描述。
Decimal类型	Decimal	Decimal	有符号的定点数，可在加、减和乘法运算过程中保持精度。支持几种写法： ● Decimal(P, S) ● Decimal32(S) ● Decimal64(S) ● Decimal128(S) **说明** P：精度，有效范围：[1:38]，决定可以有多少个十进制数字（包括分数）。 S：规模，有效范围：[0：P]，决定数字的小数部分中包含的小数位数。
字符串类型	String	字符串	字符串可以是任意长度的。它可以包含任意的字节集，包含空字节。因此，字符串类型可以代替其他DBMSs中的VARCHAR、BLOB、CLOB等类型。
	FixedString	固定字符串	当数据的长度恰好为N个字节时，FixedString类型是高效的。 在其他情况下，这可能会降低效率。可以有效存储在FixedString类型的列中的值的示例： ● 二进制表示的IP地址（IPv6使用FixedString（16）） ● 语言代码（ru_RU, en_US …） ● 货币代码（USD, RUB …） ● 二进制表示的哈希值（MD5使用FixedString（16），SHA256使用FixedString（32））

分类	关键字	数据类型	描述
时间日期类型	Date	日期	用两个字节存储，表示从1970-01-01（无符号）到当前的日期值。日期中没有存储时区信息。
	DateTime	时间戳	用四个字节（无符号的）存储Unix时间戳。允许存储与日期类型相同的范围内的值。最小值为1970-01-01 00:00:00。时间戳类型值精确到秒（没有闰秒）。时区使用启动客户端或服务器时的系统时区。
	DateTime64	DateTime64	此类型允许以日期（date）加时间（time）的形式来存储一个时刻的时间值。
布尔型	Boolean	Boolean	ClickHouse没有单独的类型来存储布尔值。可以使用UInt8 类型，取值限制为0或1。
数组类型	Array	Array	Array(T)，由T类型元素组成的数组。T可以是任意类型，包含数组类型。但不推荐使用多维数组，ClickHouse对多维数组的支持有限。例如，不能在MergeTree表中存储多维数组。
元组类型	Tuple	Tuple	Tuple(T1, T2, ...)，元组，其中每个元素都有单独的类型，不能在表中存储元组（除了内存表）。它们可以用于临时列分组。在查询中，IN表达式和带特定参数的lambda函数可以来对临时列进行分组。
Domains数据类型	Domains	Domains	Domains类型是特定实现的类型： ● IPv4是与UInt32类型保持二进制兼容的Domains类型，用于存储IPv4地址的值。它提供了更为紧凑的二进制存储的同时支持识别可读性更加友好的输入输出格式。 ● IPv6是与FixedString（16）类型保持二进制兼容的Domain类型，用于存储IPv6地址的值。它提供了更为紧凑的二进制存储的同时支持识别可读性更加友好的输入输出格式。

分类	关键字	数据类型	描述
枚举类型	Enum8	Enum8	取值范围：【 -128，127 】 Enum保存 'string'= integer的对应关系，例如：Enum8('hello' = 1, 'world' = 2)
	Enum16	Enum16	取值范围：【 -32768，32767 】
可为空	Nullable	Nullable	除非在ClickHouse服务器配置中另有说明，否则NULL是任何Nullable类型的默认值。Nullable类型字段不能包含在表索引中。 可以与TypeName的正常值存放一起。例如，Nullable(Int8) 类型的列可以存储Int8类型值，而没有值的行将存储NULL。
嵌套类型	nested	nested	嵌套的数据结构就像单元格内的表格。嵌套数据结构的参数（列名和类型）的指定方式与CREATE TABLE查询中的指定方式相同。每个表行都可以对应于嵌套数据结构中的任意数量的行。 示例：Nested(Name1 Type1, Name2 Type2, …)

2.2.2 CREATE DATABASE

本章节介绍创建数据库的基本用法。

CREATE DATABASE

```
CREATE DATABASE [IF NOT EXISTS] db_name [ON CLUSTER ClickHouse集群名];
```

表 2-8 参数说明

参数	说明
db_name	数据库
IF NOT EXISTS	如果CREATE语句中存在IF NOT EXISTS关键字，则当数据库已经存在时，该语句不会创建数据库，且不会返回任何错误。
ON CLUSTER *ClickHouse集群名*	用于指定集群名称。

说明

集群名信息可以使用以下语句的cluster字段获取：

```
select cluster,shard_num,replica_num,host_name from system.clusters;
```

使用示例

- 创建数据库demo。

```
create database demo ON CLUSTER default_cluster;
```

- 查看新建的数据库。

```
host-172-16-30-9 :) show databases;
SHOW DATABASES
Query id: ced1af23-0286-40cc-9c7a-ccbca41178d8
┌─name──────────────┐
│ INFORMATION_SCHEMA │
│ default           │
│ demo              │
│ information_schema │
│ system            │
└───────────────────┘
5 rows in set. Elapsed: 0.002 sec.
```

2.2.3 CREATE TABLE

此章节介绍如何创建表。

创建本地表

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name [ON CLUSTER ClickHouse集群名]
(
name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
name2[type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
...
) ENGINE = engine_name()
[PARTITION BY expr_list]
[ORDER BY expr_list]
```

表 2-9 参数说明

参数	说明
database_name	数据库的名称，默认为当前选择的数据库。
table_name	本地表名。
ON CLUSTER*ClickHouse集群名*	在每一个节点上都创建一个本地表，固定为ON CLUSTER *ClickHouse集群名*。
name1,name2	列名。

参数	说明
ENGINE = engine_name()	表引擎类型。 双副本版集群建表时，需要使用MergeTree系列引擎中支持数据复制的Replicated*引擎，否则副本之间不进行数据复制，导致数据查询结果不一致。使用该引擎建表时，参数填写方式如下。 • ReplicatedMergeTree('/clickhouse/tables/{database}/{table}/{shard}', '{replica}')，固定配置，无需修改。 • ReplicatedMergeTree()，等同于ReplicatedMergeTree('/clickhouse/tables/{database}/{table}/{shard}', '{replica}')。
ORDER BY expr_list	排序键，必填项，可以是一组列的元组或任意表达式。
[PARTITION BY expr_list]	分区键。一般按照日期分区，也可以使用其他字段或字段表达式。

示例：

- 创建数据库。请参见**CREATE DATABASE**。
- 使用数据库。
  ```
  use demo;
  ```
- 创建名为demo.test表。
  ```
  CREATE TABLE demo.test ON CLUSTER default_cluster(`EventDate` DateTime, `id` UInt64)ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/default/test', '{replica}') PARTITION BY toYYYYMM(EventDate) ORDER BY id;
  ```

复制表结构创建表

可以通过复制表结构创建与源表具有相同结构的表。语法：

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name2 ON CLUSTER ClickHouse集群名
 AS [db.]table_name1 [ENGINE = engine_name];
```

表 2-10 参数说明

参数	说明
db	数据库的名称，默认为当前选择的数据库。
table_name1	被复制表结构的源表。
table_name2	新创建的表。
ON CLUSTER *ClickHouse集群名*	在每一个节点上都创建一个表，固定为ON CLUSTER *ClickHouse集群名*。
[ENGINE = engine_name]	表引擎类型。如果没有指定表引擎，默认与被复制表结构的表相同。

示例：

- 创建数据库。
  ```
  create database demo;
  ```
- 使用数据库。
  ```
  use demo;
  ```
- 创建数据表。
  ```
  create table demo_t(uid Int32,name String,age UInt32,gender String)engine = TinyLog;
  ```
- 复制表结构。
  ```
  create table demo_t2 as demo_t;
  ```
- **查看表结构**。

SELECT 语句创建

使用指定的表引擎创建一个与SELECT子句的结果具有相同结构的表，并使用SELECT子句的结果进行填充。

```
CREATE TABLE [IF NOT EXISTS] [database_name.]table_name ENGINE = engine_name AS SELECT ...
```

表 2-11 参数说明

参数	说明
database_name	数据库的名称，默认为当前选择的数据库。
table_name	通过SELECT语句创建的表。
ENGINE = engine_name()	表的引擎类型。
SELECT ...	SELECT子句。

示例：

- 创建表。
  ```
  CREATE TABLE default.demo1 ON CLUSTER default_cluster( `EventDate` DateTime, `id` UInt64)ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/default/demo1', '{replica}') PARTITION BY toYYYYMM(EventDate) ORDER BY id;
  ```
- 通过SELECT语句创建表。
  ```
  create table t3 ON CLUSTER default_cluster ENGINE =MergeTree() order by EventDate as select * from default.demo1;
  ```
- 查询demo1和t3表结构。
  ```
  desc demo1;
  ```
  查询结果显示，两张表结构一样。
  ```
  cloudtable-wlr-click-20230730-06-server-1-1 :) desc demo1;
  DESCRIBE TABLE demo1
  Query id: 712f6b91-668d-4f70-b160-aac8e52f63a4
  ┌─name──────┬─type─────┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─ttl_expression─┐
  │ EventDate │ DateTime │              │                    │         │                  │                │
  │ id        │ UInt64   │              │                    │         │                  │                │
  └───────────┴──────────┴──────────────┴────────────────────┴─────────┴──────────────────┴────────────────┘
  2 rows in set. Elapsed: 0.001 sec.
  cloudtable-wlr-click-20230730-06-server-1-1 :) desc t3;
  DESCRIBE TABLE t3
  Query id: 11b67532-26f0-49c5-b36d-439d45c279bf
  ┌─name──────┬─type─────┬─default_type─┬─default_expression─┬─comment─┬─
  ```

```
codec_expression─┬─ttl_expression─┐
│ EventDate │ DateTime │                │              │         │            │               │
│ id        │ UInt64   │                │              │         │            │               │
└───────────┴──────────┴────────────────┴──────────────┴─────────┴────────────┴───────────────┘
2 rows in set. Elapsed: 0.001 sec.
```

2.2.4 DESC 查询表结构

本章节主要介绍ClickHouse查询表结构的SQL基本语法和使用说明。

基本语法

```
DESC|DESCRIBE TABLE [database_name.]table [INTO OUTFILE filename] [FORMAT format]
```

接复制表结构创建表示例，查询demo_t和demo_2表结构：

```
cloudtable-wlr-click-20230730-06-server-1-1 :) desc demo_t;
DESCRIBE TABLE demo_t
Query id: 27a38d90-9459-430f-962e-881817789fc9
┌─name───┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─
ttl_expression─┐
│ uid    │ Int32  │              │                    │         │                  │                │
│ name   │ String │              │                    │         │                  │                │
│ age    │ UInt32 │              │                    │         │                  │                │
│ gender │ String │              │                    │         │                  │                │
└────────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┴────────────────┘
4 rows in set. Elapsed: 0.001 sec.
cloudtable-wlr-click-20230730-06-server-1-1 :) desc demo_t2;
DESCRIBE TABLE demo_t2
Query id: 60054fe3-794c-410a-be13-cd0b204a9129
┌─name───┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─
ttl_expression─┐
│ uid    │ Int32  │              │                    │         │                  │                │
│ name   │ String │              │                    │         │                  │                │
│ age    │ UInt32 │              │                    │         │                  │                │
│ gender │ String │              │                    │         │                  │                │
└────────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┴────────────────┘
4 rows in set. Elapsed: 0.001 sec.
```

2.2.5 CREATE VIEW

本章节介绍如何在ClickHouse中创建普通视图。

创建视图

```
CREATE VIEW [IF NOT EXISTS] [db.]view_name [ON CLUSTER ClickHouse集群名] AS SELECT ...
```

表 2-12 参数说明

参数	说明
db	数据库的名称，默认为当前选择的数据库。
view_name	视图名。
[ON CLUSTER ClickHouse集群名]	在每一个节点上都创建一个视图，固定为ON CLUSTER ClickHouse集群名。

参数	说明
SELECT ...	SELECT子句。当数据写入视图中SELECT子句所指定的源表时，插入的数据会通过SELECT子句查询进行转换并将最终结果插入到视图中。

示例：

1. 创建源表。
   ```
   create table DB.table1 ON CLUSTER default_cluster (id Int16,name String) ENGINE = MergeTree()
   ORDER BY (id);
   ```
2. 创建视图。
   ```
   CREATE VIEW test_view ON CLUSTER default_cluster AS SELECT * FROM DB.table1;
   ```
3. 插入数据到源表中。
   ```
   insert into DB.table1 values(1,'X'),(2,'Y'),(3,'Z');
   ```
4. 查询视图。
   ```
   SELECT * FROM test_view;
   ```
5. 删除视图。
   ```
   drop table test_view ON CLUSTER default_cluster;
   ```

   说明

   - 如果建表语句中包含了“ON CLUSTER ClickHouse集群名”，删除表命令：
     ```
     drop table 表名 ON CLUSTER default_cluster;
     ```
   - 如果建表语句不包含“ON CLUSTER ClickHouse集群名”，删除表命令：
     ```
     drop table 表名;
     ```

2.2.6 CREATE MATERIALIZED VIEW

本章节介绍如何在ClickHouse中创建物化视图。

创建物化视图

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] [db.]Materialized_name [TO[db.]name] [ON
CLUSTERClickHouse集群名]
ENGINE = engine_name()
ORDER BY expr
[POPULATE]
AS SELECT ...
```

表 2-13 参数说明

参数	说明
db	数据库的名称，默认为当前选择的数据库。
Materialized_name	物化视图名。
TO[db.]name	将物化视图的数据写入到新表中。
[ON CLUSTER*ClickHouse集群名*]	在每一个节点上都创建一个物化视图，固定为ON CLUSTER *ClickHouse集群名*。

参数	说明
ENGINE = engine_name()	表引擎类型。
[POPULATE]	POPULATE关键字。如果创建物化视图时指定了POPULATE关键字，则在创建时将SELECT子句所指定的源表数据插入到物化视图中。不指定POPULATE关键字时，物化视图只会包含在物化视图创建后新写入源表的数据。 **说明** 一般不推荐使用POPULATE关键字，因为在物化视图创建期间写入源表的数据将不会写入物化视图中。
SELECT ...	SELECT子句。当数据写入物化视图中SELECT子句所指定的源表时，插入的数据会通过SELECT子句查询进行转换并将最终结果插入到物化视图中。 **说明** SELECT查询可以包含DISTINCT、GROUP BY、ORDER BY和LIMIT等，但是相应的转换是在每个插入数据块上独立执行的。

示例：

1. 创建源表。
   ```
   create table DB.table1 ON CLUSTER default_cluster (id Int16,name String) ENGINE = MergeTree() ORDER BY (id);
   ```
2. 插入数据。
   ```
   insert into DB.table1 values(1,'X'),(2,'Y'),(3,'Z');
   ```
3. 创建基于源表的物化视图。
   ```
   CREATE MATERIALIZED VIEW demo_view ON CLUSTER default_cluster ENGINE = MergeTree() ORDER BY (id) AS SELECT * FROM DB.table1;
   ```
4. 查询物化视图。
   ```
   SELECT * FROM demo_view;
   ```

   说明

   查询数据为空，说明未指定POPULATE关键字时，查询不到物化视图创建前写入源表的数据。
5. DB.table1表中插如数据。
   ```
   insert into demo_view values(4,'x'),(5,'y'),(6,'z');
   ```
6. 查询物化视图。
   ```
   SELECT * FROM demo_view;
   ```

   查询结果。

   ```
   ┌─id─┬─name─┐
   │  4 │ x    │
   │  5 │ y    │
   │  6 │ z    │
   └────┴──────┘
   ```

2.2.7 INSERT INTO

本章节介绍如何插入数据。

基本语法

- 标准格式插入数据。
```
INSERT INTO [db.]table [(c1, c2, c3)] VALUES (v11, v12, v13), (v21, v22, v23), ...
```

> **说明**
>
> 对于存在于表结构中但不存在于插入列表中的列，它们将会按照如下方式填充数据：
> - 如果存在DEFAULT表达式，根据DEFAULT表达式计算被填充的值。
> - 如果没有定义DEFAULT表达式，则填充零或空字符串。

接**复制表结构创建表**示例，插入数据：
```
insert into demo_t values(1,'Candy','23','M'),(2,'cici','33','F');
```

- 使用SELECT的结果写入。
```
INSERT INTO [db.]table [(c1, c2, c3)] SELECT ...
```

> **说明**
>
> 写入的列与SELECT的列的对应关系是使用位置来进行对应的，它们在SELECT表达式与INSERT中的名称可以是不同的。需要对它们进行对应的类型转换。
>
> 除了VALUES格式之外，其他格式中的数据都不允许出现诸如now()，1+2等表达式。VALUES格式允许您有限度的使用这些表达式，但是不建议您这么做，因为执行这些表达式很低效。

2.2.8 SELETC

描述如何使用SELECT语句查询数据。

基本语法

```
SELECT [DISTINCT] expr_list
[FROM [database_name.]table | (subquery) | table_function] [FINAL]
[SAMPLE sample_coeff]
[ARRAY JOIN ...]
[GLOBAL] [ANY|ALL|ASOF] [INNER|LEFT|RIGHT|FULL|CROSS] [OUTER|SEMI|ANTI] JOIN (subquery)|table
(ON <expr_list>)|(USING <column_list>)
[PREWHERE expr]
[WHERE expr]
[GROUP BY expr_list] [WITH TOTALS]
[HAVING expr]
[ORDER BY expr_list] [WITH FILL] [FROM expr] [TO expr] [STEP expr]
[LIMIT [offset_value, ]n BY columns]
[LIMIT [n, ]m] [WITH TIES]
[UNION ALL ...]
[INTO OUTFILE filename]
[FORMAT format]
```

示例：

- 查看ClickHouse集群信息。
```
select * from system.clusters;
```
- 显示当前节点设置的宏。
```
select * from system.macros;
```
- 查看数据库容量。
```
select sum(rows) as "总行数", formatReadableSize(sum(data_uncompressed_bytes)) as "原始大小",
formatReadableSize(sum(data_compressed_bytes)) as "压缩大小",
round(sum(data_compressed_bytes) / sum(data_uncompressed_bytes) * 100, 0) "压缩率" from
system.parts;
```
- 查询test表容量。where条件根据实际情况添加修改。
```
select sum(rows) as "总行数", formatReadableSize(sum(data_uncompressed_bytes)) as "原始大小",
formatReadableSize(sum(data_compressed_bytes)) as "压缩大小",
```

```
round(sum(data_compressed_bytes) / sum(data_uncompressed_bytes) * 100, 0) "压缩率" from
system.parts where table in ('test') and partition like '2020-11-%' group by table;
```

2.2.9 ALTER TABLE 修改表结构

本章节主要介绍ClickHouse修改表结构的SQL基本语法和使用说明。

基本语法

```
ALTER TABLE [database_name].name [ON CLUSTER ClickHouse集群名] ADD|DROP|CLEAR|COMMENT|
MODIFY COLUMN ...
```

说明

ALTER仅支持 *MergeTree ， Merge以及Distributed等引擎表。

示例：

1. 创建表DB_table1。
   ```
   CREATE TABLE DB_table1 ON CLUSTER default_cluster(Year UInt16,Quarter UInt8,Month
   UInt8,DayofMonth UInt8,DayOfWeek UInt8,FlightDate Date,FlightNum String,Div5WheelsOff
   String,Div5TailNum String)ENGINE = MergeTree() PARTITION BY toYYYYMM(FlightDate) PRIMARY
   KEY (intHash32(FlightDate)) ORDER BY (intHash32(FlightDate),FlightNum) SAMPLE BY
   intHash32(FlightDate) SETTINGS index_granularity= 8192;
   ```
2. 给DB_table1增加列test。
   ```
   ALTER TABLE DB_table1 ADD COLUMN test String DEFAULT 'defaultvalue';
   ```
   查表。
   ```
   desc DB_tables;
   ```
3. 修改表DB_table1列Year类型为UInt8。
   ```
   ALTER TABLE DB_table1 MODIFY COLUMN Year UInt8;
   ```
   查表结构。
   ```
   desc DB_tables;
   ```
4. 删除表DB_table1列test。
   ```
   ALTER TABLE DB_table1 DROP COLUMN test;
   ```
   查表。
   ```
   desc DB_tables;
   ```
5. 修改表DB_table1列Month为Month_test。
   ```
   ALTER TABLE DB_table1 RENAME COLUMN Month to Month_test;
   ```
   查表。
   ```
   desc DB_tables;
   ```

2.2.10 DROP 删除表

此章节主要介绍ClickHouse删除表的SQL基本语法和使用说明。

基本语法

```
DROP [TEMPORARY] TABLE [IF EXISTS] [database_name.]name [ON CLUSTER cluster] [SYNC]
```

示例：

- 删除表t1。
  ```
  drop table t1 SYNC;
  ```

说明

- 在删除复制表时，因为复制表需要在Zookeeper上建立一个路径，存放相关数据。ClickHouse默认的库引擎是原子数据库引擎，删除Atomic数据库中的表后，它不会立即删除，而是会在24小时后删除。在删除表时，加上SYNC字段，即可解决该问题，例如：drop table t1 SYNC;
- 删除本地表和分布式表，则不会出现该问题，可不带SYNC字段，例如：drop table t1;
- 如果建表语句中包含了“ON CLUSTER ClickHouse集群名”，删除表命令：
  ```
  drop table 表名 ON CLUSTER default_cluster;
  ```
- 如果建表语句不包含“ON CLUSTER ClickHouse集群名”，删除表命令：
  ```
  drop table 表名;
  ```
- 删除数据表前，需确认此数据表是否应用中，以免引起不必要的麻烦。删除数据表后可在24小时内恢复，超过24小时无法恢复。恢复命令如下：
  ```
  set allow_experimental_undrop_table_query = 1;
  UNDROP TABLE 数据表名;
  ```

2.2.11 SHOW 显示数据库和表信息

此章节主要介绍ClickHouse显示数据库和表信息的SQL基本语法和使用说明。

基本语法

```
show databases;
show tables;
```

示例：

- 查询数据库。
  ```
  show databases;
  ```
- 查询表信息。
  ```
  show tables;
  ```

2.3 数据迁移同步

2.3.1 数据导入导出

本章节主要介绍使用ClickHouse客户端导入导出文件数据的基本语法和使用说明。

CSV 格式数据导入导出

- CSV格式数据导入。
  - 非安全集群
    ```
    cat csv_ssl | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 --query="INSERT INTO test145 FORMAT CSV"
    ```
  - 安全集群
    ```
    cat csv_no_ssl | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 --query="INSERT INTO test146 FORMAT CSV"
    ```

1. host：主机名/ClickHouse实例IP地址。
2. port：端口号（在集群详情页面查看）。
3. user：创建集群时创建的用户名。
4. database：数据库名。

5. password：创建集群时，创建的密码。
6. INSERT INTO：后面跟数据表。
7. cat文件路径：文件存放的路径，路径自定义。
8. config-file ./config.xml：指定配置文件，请参见**ClickHouse安全通道**章节。

- CSV格式数据导出。
  - 非安全集群
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password Password --database test010 -m --query="select * from test139 FORMAT CSV" > ./csv_no_ssl
    ```
  - 安全集群
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 -m --query="select * from test139 FORMAT CSV" > ./csv_no_ssl
    ```

1. host：主机名/ClickHouse实例IP地址。
2. port：端口号（在集群详情页面查看）。
3. user：创建集群时创建的用户名。
4. database：数据库名。
5. password：创建集群时，创建的密码。
6. SELECT * FROM：后面跟数据表。
7. ./csv_no_ssl：指文件存放路径，存放路径自定义。
8. config-file ./config.xml：指定配置文件，请参见**ClickHouse安全通道**章节。

parquet 格式数据导入导出

- parquet格式数据导入。
  - 非安全集群
    ```
    cat parquet_no_ssl.parquet | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 --query="INSERT INTO test145 FORMAT Parquet"
    ```
  - 安全集群
    ```
    cat parquet_no_ssl.parquet | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 --query="INSERT INTO test146 FORMAT Parquet"
    ```

1. parquet_no_ssl.parquet：表示格式文件存放路径，路径自定义。
2. host：主机名/ClickHouse实例IP地址。
3. port：端口号（在集群详情页面查看）。
4. user：创建集群时创建的用户名。
5. database：数据库名。
6. password：创建集群时，创建的密码。
7. INSERT INTO：后面跟数据表。
8. config-file ./config.xml：指定配置文件，请参见**ClickHouse安全通道**章节。

- parquet格式数据导出。
  - 非安全集群
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 -m --query="select * from test139 FORMAT Parquet" > ./parquet_no_ssl.parquet
    ```
  - 安全集群
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 -m --query="select * from test139 FORMAT Parquet" > ./parquet_ssl.parque
    ```

1. host：主机名/ClickHouse实例IP地址。
2. port：端口号（在集群详情页面查看）。
3. user：创建集群时创建的用户名。
4. database：数据库名。
5. password：创建集群时，创建的密码。
6. select * from：后面跟数据表。
7. ./parquet_no_ssl.parquet：代表parquet格式文件导出路径，路径自定义。
8. config-file ./config.xml：指定配置文件，请参见**ClickHouse安全通道**章节。

ORC 格式数据导入导出

- ORC格式数据导入。
  - 非安全集群
    ```
    cat orc_no_ssl.orc | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 --query="INSERT INTO test143 FORMAT ORC"
    ```
  - 安全集群
    ```
    cat orc_no_ssl.orc | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 --query="INSERT INTO test144 FORMAT ORC
    ```

1. cat orc_no_ssl.orc：orc格式文件存放路径，路径自定义。
2. host：主机名/ClickHouse实例IP地址。
3. port：端口号（在集群详情页面查看）。
4. user：创建集群时创建的用户名。
5. database：数据库名。
6. password：创建集群时，创建的密码。
7. INSERT INTO：后面跟数据表。
8. config-file ./config.xml：指定配置文件，请参见**ClickHouse安全通道**章节。

- ORC格式数据导出。
  - 安全集群
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 -m --query="select * from test139 FORMAT ORC" > ./orc_ssl.orc
    ```
  - 非安全集群
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 -m --query="select * from test139 FORMAT ORC" > ./orc_no_ssl.orc
    ```

1. host：主机名/ClickHouse实例IP地址。
2. port：端口号（在集群详情页面查看）。
3. user：创建集群时创建的用户名。
4. database：数据库名。
5. password：创建集群时，创建的密码。
6. config-file ./config.xml：指定配置文件，请参见**ClickHouse安全通道**章节。
7. select * from：后面跟数据表。
8. /opt/student.orc：导出的ORC格式文件路径，路径自定义。

JSON 格式数据导入导出

- JSON格式数据导入。
  - 非安全集群
    ```
    cat ./jsonnossl.json | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 --query="INSERT INTO test141 FORMAT JSON"
    ```
  - 安全集群
    ```
    cat ./jsonssl.json | ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 --query="INSERT INTO test142 FORMAT JSON"
    ```

1. cat文件路径：导入文件路径，路径自定义。
2. host：主机名/ClickHouse实例IP地址。
3. port：端口号（在集群详情页面查看）。
4. user：创建集群时创建的用户名。
5. database：数据库名。
6. password：创建集群时，创建的密码。
7. INSERT INTO：后面跟数据表。
8. config-file ./config.xml：指定配置文件，请参见**ClickHouse安全通道**章节。

- JSON格式数据导出。
  - 安全集群
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --database test010 -m --query="select * from test139 FORMAT JSON" > ./jsonnossl.json
    ```
  - 非安全集群
    ```
    ./clickhouse client --host 192.168.x.x --port port --user admin --password password --config-file ./config.xml --database test010 -m --query="select * from test139 FORMAT JSON" > ./jsonssl.json
    ```

1. host：主机名/ClickHouse实例IP地址。
2. port：端口号（在集群详情页面查看）。
3. user：创建集群时创建的用户名。
4. database：数据库名。
5. password：创建集群时，创建的密码。
6. SELECT * FROM：后面跟数据表。
7. ./jsonssl.json：文件导出路径，路径自定义。
8. config-file ./config.xml：指定配置文件，请参见**ClickHouse安全通道**章节。

2.3.2 ClickHouse 访问 RDS MySQL 服务

ClickHouse面向OLAP场景提供高效的数据分析能力，支持通过MySQL等数据库引擎将远程数据库服务器中的表映射到ClickHouse集群中，后续可以在ClickHouse中进行数据分析。以下操作通过ClickHouse集群和RDS服务下的MySQL数据库实例对接进行举例说明。

前提条件

- 已提前准备好对接的RDS数据库实例及数据库用户名、密码。详细操作可以参考**创建和连接RDS数据库实例**。
- 已成功创建ClickHouse集群且集群和实例状态正常。

约束限制

- RDS数据库实例和ClickHouse集群在相同的VPC和子网内。
- 在进行数据同步操作时需要评估对源数据库和目标数据库性能的影响，同时建议您在业务低峰期执行数据同步。
- 当前ClickHouse支持和RDS服务下的MySQL、PostgreSQL实例进行对接，不支持对接SQL Server实例。

ClickHouse 通过 MySQL 引擎对接 RDS 服务

MySQL引擎用于将远程的MySQL服务器中的表映射到ClickHouse中，并允许您对表进行INSERT和SELECT查询，以方便您在ClickHouse与MySQL之间进行数据交换。

- MySQL引擎使用语法：

```
CREATE DATABASE [IF NOT EXISTS] db_name [ON CLUSTER cluster]
ENGINE = MySQL('host:port', ['database' | database], 'user', 'password')
```

表 2-14 MySQL 数据库引擎参数说明

参数	描述
hostport	RDS服务MySQL数据库实例IP地址和端口。
database	RDS服务MySQL数据库名。
user	RDS服务MySQL数据库用户名。
password	RDS服务MySQL数据库用户密码。

MySQL引擎使用示例：

a. 连接到RDS服务的MySQL数据库。详细操作可以参考**RDS服务MySQL实例连接**。

b. 在MySQL数据库上创建表，并插入数据。

c. 使用客户端命令连接ClickHouse。

非安全集群连接命令

```
./clickhouse client --host 集群内网地址  --port 端口 --user admin --password password
```

安全集群连接命令，详细操作请参见**ClickHouse安全通道**章节。

```
./clickhouse client --host 集群内网地址  --port 端口 --user admin --password password --secure
--config-file /root/config.xml
```

说明

集群内网地址：集群详情页面中集群访问地址，这里替换成您自己购买的集群的访问地址。

d. 在ClickHouse中创建MySQL引擎的数据库，创建成功后自动与MySQL服务器交换数据。

```
CREATE DATABASE mysql_db ENGINE = MySQL('RDS服务MySQL数据库实例IP地址:MySQL数据库实例端口', 'MySQL数据库名', 'MySQL数据库用户名', 'MySQL数据库用户名密码');
```

e. 切换到新建的数据库mysql_db，并查询表数据。

```
USE mysql_db;
```

在ClickHouse中查询MySQL数据库表数据。

```
SELECT * FROM mysql_table;
┌─int_id─┬─float─┐
```

```
│     1 │     2 │
└───────┴───────┘
```

新增插入数据后也可以正常进行查询。

```
INSERT INTO mysql_table VALUES (3,4);
SELECT * FROM mysql_table;
┌─int_id─┬─float─┐
│     1 │     2 │
│     3 │     4 │
└───────┴───────┘
```

2.4 开发程序

2.4.1 典型场景说明

通过典型场景，用户可以快速学习和掌握ClickHouse的开发过程，并且对关键的接口函数有所了解。

场景说明

假定用户需要开发一个应用程序，用于存储或根据一定条件查询人员的姓名、年龄和入职日期。主要操作步骤：

1. 建立数据库的连接。
2. 建立一张人员信息表。
3. 插入数据（样例代码中数据为随机生成）。
4. 根据条件查询数据。

2.4.2 开发思路

ClickHouse作为一款独立的DBMS系统，使用SQL语言就可以进行常见的操作。开发程序示例中，全部通过clickhouse-jdbc API接口来进行描述。

- **设置属性**：设置连接ClickHouse服务实例的参数属性。
- **建立连接**：建立和ClickHouse服务实例的连接。
- **创建库**：创建ClickHouse数据库。
- **创建表**：创建ClickHouse数据库下的表。
- **插入数据**：插入数据到ClickHouse表中。
- **查询数据**：查询ClickHouse表数据。
- **删除表**：删除已创建的ClickHouse表。

2.4.3 准备开发和运行环境

准备开发环境

在进行应用开发时，要准备的开发和运行环境如表1所示。

表 2-15 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows7以上版本。 • 运行环境：Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	安装JDK，版本为1.8.0_272。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** • 如果使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 如果使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 如果使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
准备开发用户	准备用于应用开发的ClickHouse集群用户并授予相应权限。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

2.4.4 配置并导入样例工程

背景信息

获取ClickHouse开发样例工程，将工程导入到IntelliJ IDEA开始样例学习。

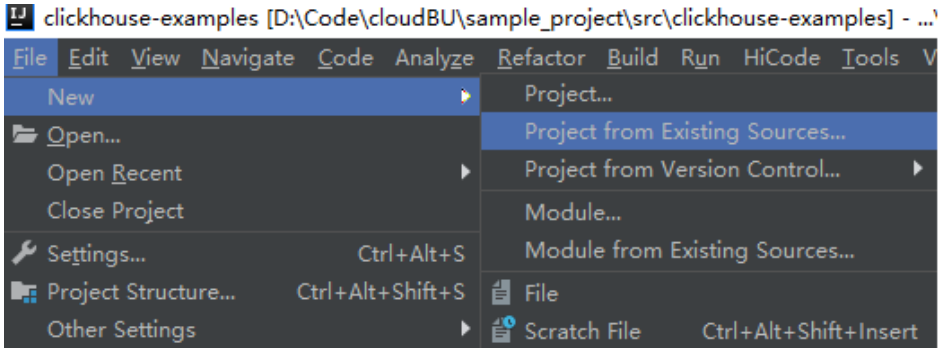
操作场景

ClickHouse针对多个场景提供样例工程，帮助客户快速学习ClickHouse工程。

操作步骤

步骤1 在应用开发环境中，导入代码样例工程到IntelliJ IDEA开发环境。

1. 在IDEA界面选择“File>New>Project from Existing Sources”。



2. 在显示的“Select File or Directory to Import”对话框中，选择“clickhouse-examples”文件夹中的“pom.xml”文件，单击“OK”。
3. 确认后续配置，单击“Next”，如无特殊需求，使用默认值即可。
4. 选择推荐的JDK版本，单击“Finish”完成导入。

步骤2 工程导入完成后，修改样例工程的“conf”目录下的“clickhouse-example.properties”文件，根据实际环境信息修改相关参数。

```
ipList=
sslUsed=false
httpPort=8123
httpsPort=
CLICKHOUSE_SECURITY_ENABLED=false
user=default
password=
clusterName=default_cluster
databaseName=testdb
tableName=testtb
batchRows=10000
batchNum=10
clickhouse_dataSource_ip_list=ip:8123,ip:8123
native_dataSource_ip_list=ip:9000,ip:9000
```

表 2-16 配置说明表

配置名称	默认值	含义
iPList	-	必填参数，配置为clickhouse节点的集群访问地址列表。 登录cloudtable控制台，单击集群名称，进入集群详情页，拿到集群访问地址。 多个地址使用逗号分隔，例如配置为“cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com,cloudtable-wlr-cli-server-2-1-iqVWp2Mo.mycloudtable.com”。
sslUsed	false	是否启用ssl加密，默认为“false”。
httpPort	8123	连接的HTTP端口，值为8123。
httpsPort	-	连接使用的HTTPS端口，值为8443。
CLICKHOUSE_SECURITY_ENABLED	false	ClickHouse安全模式开关，普通模式集群时该参数填写为false。
user	default	表1中已准备好的开发用户。
password	-	开发用户对应的密码。
clusterName	default_cluster	ClickHouse逻辑集群名称，保持默认值。
databaseName	testdb	样例代码工程中需要创建的数据库名称，可以根据实际情况修改。
tableName	testtb	样例代码工程中需要创建的表名称，可以根据实际情况修改。

配置名称	默认值	含义
batchRows	10000	一个批次写入数据的条数。
batchNum	10	写入数据的总批次。
clickhouse_dataSource_ip_list	-	clickhouse节点的ip和http端口集合，例如配置为：cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:8123,cloudtable-wlr-cli-server-2-1-iqVWp2Mo.mycloudtable.com:8123
native_dataSource_ip_list	-	clickhouse节点的ip和tcp端口集合，格式参考：cloudtable-wlr-cli-server-1-1-2lIWzDO9.mycloudtable.com:9000,cloudtable-wlr-cli-server-2-1-iqVWp2Mo.mycloudtable.com:9000

----结束

2.4.5 样例代码说明

2.4.5.1 设置属性

功能介绍

可以通过Properties设置连接属性。

如下样例代码设置socket超时时间为60s， 设置不使用SSL。

代码样例

```
Properties clickHouseProperties = new Properties();
clickHouseProperties.setProperty(ClickHouseClientOption.CONNECTION_TIMEOUT.getKey(),
Integer.toString(60000));
clickHouseProperties.setProperty(ClickHouseClientOption.SSL.getKey(), Boolean.toString(false));
clickHouseProperties.setProperty(ClickHouseClientOption.SSL_MODE.getKey(), "none");
```

2.4.5.2 建立连接

功能介绍

创建连接时使用ClickHouseDataSource配置连接使用的url和属性。

然后使用clickhouse-example.properties配置的user和password作为认证凭据，ClickHouse会带着用户名和密码在服务端进行安全认证。

样例代码

```
ClickHouseDataSource clickHouseDataSource =new ClickHouseDataSource(JDBC_PREFIX +
serverList.get(tries - 1), clickHouseProperties);
connection = clickHouseDataSource.getConnection(user, password);
```

说明

认证用的密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

2.4.5.3 创建库

功能介绍

如下示例中通过on cluster语句在集群的所有Server节点创建数据库。

其中数据库名定义在clickhouse-example.properties文件的databaseName字段。

样例代码

```
private void createDatabase(String databaseName, String clusterName) throws Exception  {
        String createDbSql = "create database if not exists " + databaseName + " on cluster " + clusterName;
        util.exeSql(createDbSql);
}
```

2.4.5.4 创建表

功能介绍

如下示例中通过on cluster语句在集群的所有Server节点创建分布式表和本地表。

createSql为本地表，createDisSql为基于本地表的分布式表。

样例代码

```
private void createTable(String databaseName, String tableName, String clusterName) throws Exception {
    String createSql = "create table " + databaseName + "." + tableName + " on cluster " + clusterName
            + " (name String, age UInt8, date Date)engine=ReplicatedMergeTree('/clickhouse/tables/{shard}/" +
databaseName
            + "." + tableName + "'," + "'{replica}') partition by toYYYYMM(date) order by age";
    String createDisSql = "create table " + databaseName + "." + tableName + "_all" + " on cluster " +
clusterName + " as "
            + databaseName + "." + tableName + " ENGINE = Distributed(default_cluster," + databaseName +
"," + tableName + ", rand());";
    ArrayList<String> sqlList = new ArrayList<String>();
    sqlList.add(createSql);
    sqlList.add(createDisSql);
    util.exeSql(sqlList);
}
```

2.4.5.5 插入数据

功能介绍

如下示例代码通过循环batchNum次，构造示例数据并通过PreparedStatement的executeBatch()方法批量插入数据。

其中数据类型为创建的表所指定的三个字段，分别是String、UInt8和Date类型。

样例代码

```
String insertSql = "insert into " + databaseName + "." + tableName + " values (?,?,?)";
PreparedStatement preparedStatement = connection.prepareStatement(insertSql);
long allBatchBegin = System.currentTimeMillis();
```

```
for (int j = 0; j < batchNum; j++) {
    for (int i = 0; i < batchRows; i++) {
        preparedStatement.setString(1, "xxx_" + (i + j * 10));
        preparedStatement.setInt(2, ((int) (Math.random() * 100)));
        preparedStatement.setDate(3, generateRandomDate("2018-01-01", "2021-12-31"));
        preparedStatement.addBatch();
    }
    long begin = System.currentTimeMillis();
    preparedStatement.executeBatch();
    long end = System.currentTimeMillis();
    log.info("Inert batch time is {} ms", end - begin);
}
long allBatchEnd = System.currentTimeMillis();
log.info("Inert all batch time is {} ms", allBatchEnd - allBatchBegin);
```

2.4.5.6 查询数据

功能介绍

查询语句1：querySql1查询创建表创建的tableName表中任意10条数据；

查询语句2：querySql2通过内置函数对创建表创建的tableName表中的日期字段取年月后进行聚合。

样例代码

```
private void queryData(String databaseName, String tableName) throws Exception {
    String querySql1 = "select * from " + databaseName + "." + tableName + "_all" + " order by age limit
10";
    String querySql2 = "select toYYYYMM(date),count(1) from " + databaseName + "." + tableName + "_all"
+ " group by toYYYYMM(date) order by count(1) DESC limit 10";
    ArrayList<String> sqlList = new ArrayList<String>();
    sqlList.add(querySql1);
    sqlList.add(querySql2);
    ArrayList<ArrayList<ArrayList<String>>> result = util.exeSql(sqlList);
    for (ArrayList<ArrayList<String>> singleResult : result) {
        for (ArrayList<String> strings : singleResult) {
            StringBuilder stringBuilder = new StringBuilder();
            for (String string : strings) {
                stringBuilder.append(string).append("\t");
            }
            log.info(stringBuilder.toString());
        }
    }
}
```

2.4.5.7 删除表

功能介绍

删除在创建表中创建的副本表和分布式表。

语句1：使用drop table将集群中的本地表删除。

语句2：使用drop table将集群中的分布式表删除。

样例代码

```
private void dropTable(String databaseName, String tableName, String clusterName) throws Exception {
    String dropLocalTableSql = "drop table if exists " + databaseName + "." + tableName + " on cluster " +
clusterName;
    String dropDisTableSql = "drop table if exists " + databaseName + "." + tableName + "_all" + " on
```

```
cluster " + clusterName;
    ArrayList<String> sqlList = new ArrayList<String>();
    sqlList.add(dropLocalTableSql);
    sqlList.add(dropDisTableSql);
    util.exeSql(sqlList);
}
```

2.5 调测程序

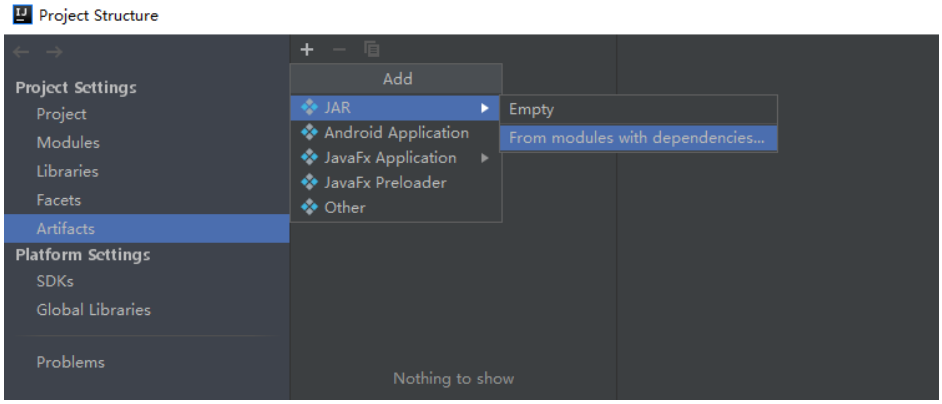
ClickHouse应用程序支持在Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至准备好的Linux运行环境中运行。该环境需要和clickhouse集群处于同一vpc和安全组，以保证网络连通。

前提条件

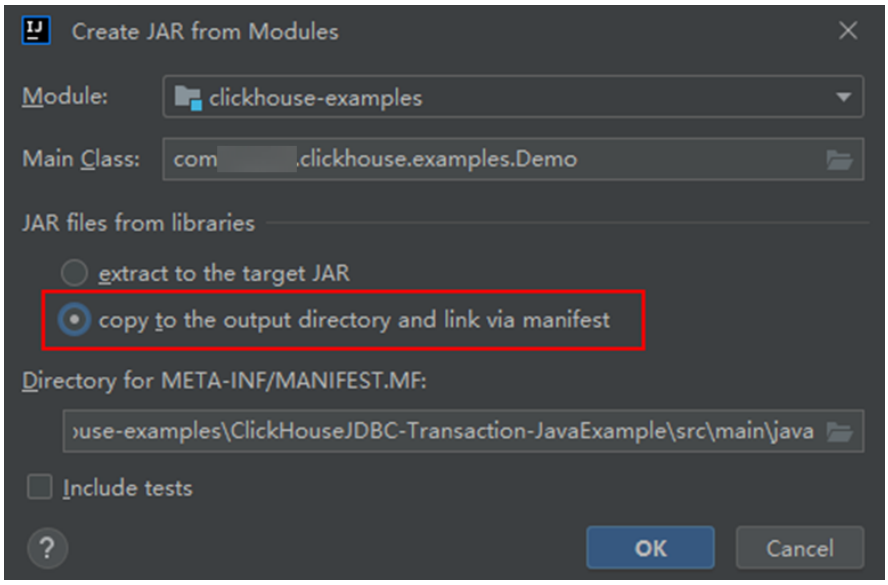
Linux环境已安装JDK，版本号需要和IntelliJ IDEA导出Jar包使用的JDK版本一致，并设置好Java环境变量。

编译并运行程序

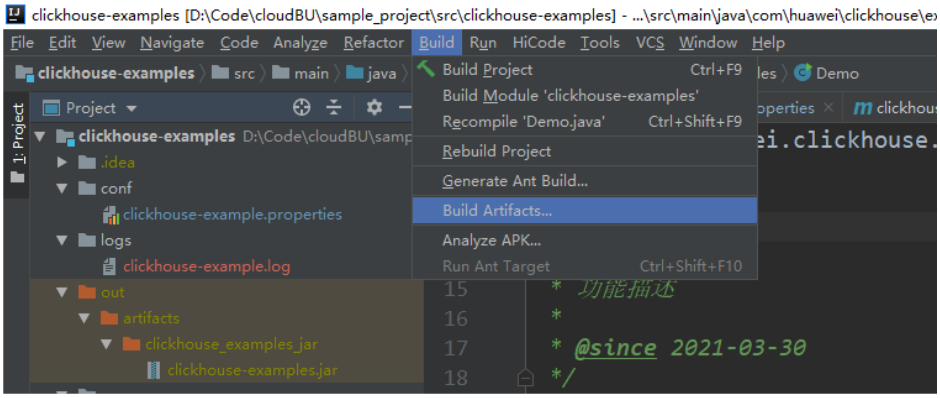
1. 导出jar包。
   a. 进入IntelliJ IDEA，选择“File > Project Structure > Artifacts”。
   b. 单击“加号”，选择“JAR > From modules with dependencies”。



   c. “Main Class”选择“com.xxx.clickhouse.examples.Demo”，单击OK。



d. 选择“Build> Build Artifacts”。编译成功后在“clickhouse-examples\out\artifacts\clickhouse_examples_jar”目录下查看并获取当前目录的所有jar文件。



2. 将“clickhouse-examples\out\artifacts\clickhouse_examples.jar”目录下的所有jar文件和“clickhouse-examples”目录下的“conf”文件夹拷贝到ECS的同一目录下。
3. 登录客户端节点，进入jar文件上传目录下，修改文件权限为700。

   chmod -R 700 clickhouse-examples.jar
4. 在“clickhouse_examples.jar”所在客户端目录下执行如下命令运行jar包：

   java -cp ./*:conf/clickhouse-example.properties com.xxx.clickhouse.examples.Demo

查看调测结果

运行结果中没有异常或失败信息即表明运行成功。

图 2-3 运行日志图