## 第二部分 机票分析平台数据存储

本章任务开始为机票分析平台准备职位数据，所以我们要把数据存储到集群的HDFS上，为接下来的大数据分析做好准备。因此要了解如何使用HDFS存储原始数据、修改文件存储格式。

### 任务1 使用HDFS存储原始数据

#### 1.1　HDFS架构及原理

##### 1.HDFS 是做什么的

HDFS （Hadoop Distributed File System）源于 Google 在2003年10月份发表的GFS（Google File System） 论文。 它其实就是 GFS 的一个克隆版本。是Hadoop项目的核心子项目，是分布式计算中数据存储管理的基础，是基于流数据模式访问和处理超大文件的需求而开发的，可以运行于廉价的商用服务器上。它所具有的高容错、高可靠性、高可扩展性、高获得性、高吞吐率等特征为海量数据提供了不怕故障的存储，为超大数据集（Large Data Set）的应用处理带来了很多便利。

##### 2.HDFS的架构和数据存储原理

HDFS是一个主/从（Mater/Slave）体系结构，从最终用户的角度来看，它就像传统的文件系统一样，可以通过目录路径对文件执行CRUD操作。但由于分布式存储的性质，HDFS集群拥有一个Master和多个Slave。Master上的NameNode管理文件系统的元数据，Slave上的DataNode存储实际的数据。客户端通过同Master和Slave的交互访问文件系统。客户端联系Master的NameNode以获取文件的元数据，而真正的文件I/O操作是直接和Slave的DataNode进行交互的。

##### 3.HDFS 的架构图

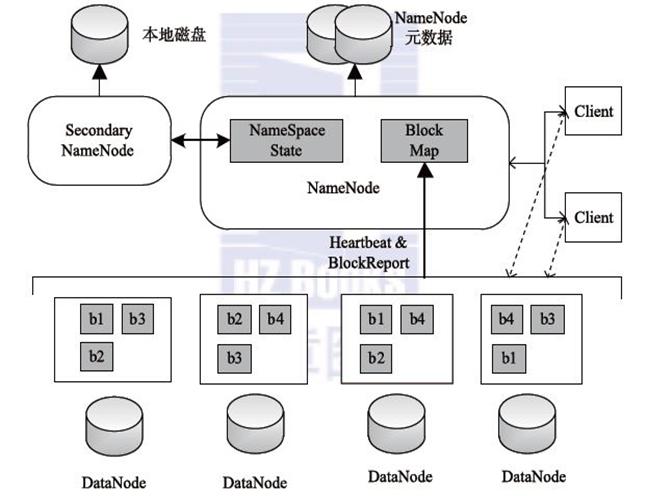
****

图4-1 HDFS的架构图

HDFS的架构图图4-1所示主要由HDFS Client、NameNode、DataNode和Secondary NameNode四个部分组成。下面分别介绍这四个组成部分。

1、Client：就是客户端。

1. 文件分割。文件上传 HDFS 的时候，Client 首先将文件分割成 一个一个的Block，然后进行存储。
2. 然后在与 NameNode进行交互，获取文件的位置信息。
3. 获取文件的位置信息后，再与 DataNode 交互，读取数据或者写入数据。
4. Client 提供一些命令来管理 HDFS、访问 HDFS，比如启动或者关闭HDFS。
5. NameNode：就是 master，它是一个集群管理者。
6. NameNode管理 HDFS 的名称空间
7. NameNode管理数据块映射信息
8. 配置副本策略
9. 处理Client读取数据或者写入数据请求。
10. DataNode：就是Slave。集群管理者(master)下达命令，Slave执行实际的操作。
11. DataNode是存储实际的数据块的。
12. DataNode实际执行数据块的读/写操作。
13. Secondary NameNode：并非 NameNode 的后备程序。当NameNode 挂掉后，它并不能立即替换 NameNode 并提供服务。
14. Secondary NameNode是辅助 NameNode，帮助NameNode分担其工作量。
15. Secondary NameNode定期合并 fsimage和fsedits，然后推送给NameNode。
16. 在紧急情况下，可辅助性恢复 NameNode。

##### 4.为什么选择 HDFS 存储数据

之所以选择 HDFS 存储数据是因为 HDFS 具有以下优点：

1、高容错性

数据自动保存了多个副本。HDFS通过增加副本的方式，大大的提高容错性。假如某一个副本丢失以后，它可以通过其他副本自动恢复。

2、适合批量处理

HDFS是通过移动计算而不是移动数据，HDFS会把数据位置提供给给计算框架，计算框架自己去获取数据。

3、适合大量数据处理

能够处理百万规模以上的文件数量，数量相当之大，甚至PB级别的数据。

4、流式访问文件

一次写入，能够多次读取。但是文件一旦写入就不能修改，只允许追加。这样能够保证数据的一致性，满足多次使用。

当然 HDFS 也有它的劣势，并不适合所有的场合：

1、低延时数据访问

比如毫秒级的来存储数据，这是不行的，它做不到。比如毫秒级以内读取数据，这样它是很难做到的。

2、小文件存储

存储大量小文件(小于HDFS系统的Block大小的文件（默认64M）)，它会占用 NameNode大量的内存来存储文件、目录和块信息。这样是不可取的，因为NameNode的内存总是有限的。小文件存储的获取元数据的时间会超过读取时间，它违反了HDFS的设计目标。

3、并发写入、文件随机修改

一个文件只能有一个写，不允许多个线程同时写。仅支持数据 append（追加），不支持文件的随机修改。

因此大数据项目都是对大量数据进行分析以及处理，所以大数据项目一般都会选用 HDFS 存储数据。

##### 5.HDFS 如何读取文件

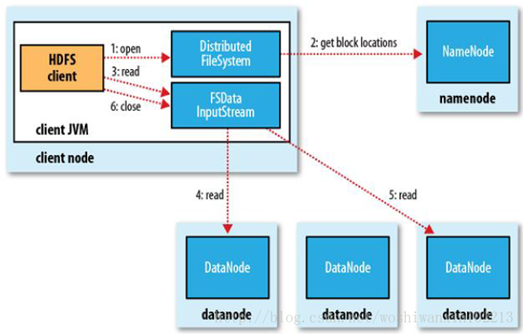
****

图4-2 HDFS 读取文件流程

HDFS的文件读取原理如图4-2，主要包括以下几个步骤：

1. 客户端（Client）调用FileSystem对象的open方法在分布式文件系统中打开要读取的文件。

2.分布式文件系统通过使用远程过程调用调用namenode，来确定文件起始块的位置信息。

3.前两步会返回一个FSDataInputStream对象，该对象会被封装成 DFSInputStream对象，DFSInputStream可以方便的管理datanode和namenode数据流。客户端调用read方法，DFSInputStream就会找出离客户端最近的datanode并连接datanode。数据从datanode源传送到客户端（Client）。

如果一个datanode的的数据读完了，就会关闭指向这个的datanode连接，DFSInputStream就再连接距离最近的datanode，通过反复调用read方法，将数据从datanode传输到客户端（Client）。

1. 客户端（Client）完成读取，对FSDataInputStream调用close()方法关闭连接。

##### 6.HDFS 如何写入文件

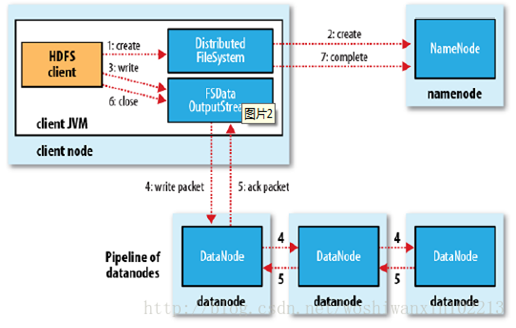
****

图4-3 HDFS 写入文件

HDFS的文件写入原理如图4-3，主要包括以下几个步骤：

1. 客户端（Client）通过调用 DistributedFileSystem 的create方法，创建一个新的文件。

2. DistributedFileSystem 通过 RPC（远程过程调用）调用 NameNode，在文件系统的命名空间中新建一个文件。

3. 前两步结束后会返回 FSDataOutputStream 的对象，FSDataOutputStream对象封装一个DFSoutPutstream对象，负责处理namenode和datanode之间的通信，客户端开始写入数据。

FSDataOutputStream将数据分成一个一个的Block数据包，写入内部队列“数据队列”中，DataStreamer负责将Block数据包依次流式传输到由一组namenode构成的队列中。

5. DFSOutputStream维护着确认“数据队列”来等待datanode收到确认回执，收到所有datanode确认后，数据包从确认“数据队列”删除。

6. 客户端（Client）完成写数据后，调用close方法关闭写入流。

7. DataStreamer 收到最后一个datanode确认回执后，通知 DataNode 把文件标示为已完成。

#### 1.2　使用HDFS-SHELL存储数据

##### 1.HDFS的Shell的基本概念

1. 调用文件系统(FS)Shell命令应使用 bin/hadoop fs 命令或 bin/hdfs dfs 命令的形式。

2. 所有的FS shell命令使用URI路径作为参数。

　   URI格式是scheme://authority/path。HDFS的scheme是hdfs，对本地文件系统，scheme是file。其中scheme和authority参数都是可选的，如果未加指定，就会使用配置中指定的默认scheme。例如：hdfs://namenode:namenodePort/parent/child，可以表示成/parent/child（假设配置文件是namenode:namenodePort）

1. 大多数FS Shell命令的行为和对应的LINUX Shell命令类似。

##### HDFS的Shell命令

Hadoop fs = hdfs dfs

1. **帮助help**

|  |
| --- |
| -ls  功能：输出这个命令参数手册  示例：hadoop fs -help |

1. **查询**

|  |
| --- |
| -help  功能：显示目录信息  使用命令：  bin/hdfs dfs -ls / 这条执行会列出/目录下的文件和目录  hdfs dfs -ls -R /这条会列出/目录下的左右文件，由于有-R参数，会在文件夹和子文件夹下执行ls操作。 |

1. **添加文件夹**

|  |
| --- |
| -mkdir  功能：在hdfs上创建目录  示例：hdfs dfs -mkdir -p /test/input/ |

1. **从本地复制到HDFS中**

|  |
| --- |
| -copyFromLocal 或 -put  功能：从本地剪切粘贴到hdfs  示例：hdfs dfs -copyFromLocal   /root/city.csv  /test/input |

1. **把文件从hdfs取回本地**

|  |
| --- |
| -copyToLocal      或 -get  功能：从hdfs剪切粘贴到本地  示例：hdfs dfs -copyToLocal /test/input/ city.csv /root/a.csv |

1. **查看hdfs文件中的内容**

|  |
| --- |
| -cat  功能：查看hdfs上文件  示例：hdfs dfs -cat /test/input/city.csv  -tail  功能：显示一个文件的末尾  示例：hdfs dfs -tail /test/input/city.csv  -text  功能：以字符形式打印一个文件的内容  示例：hdfs dfs -text /test/input/city.csv |

1. **删除文件**

|  |
| --- |
| -rm -f  功能：删除hdfs中存储的某个文件  示例：hdfs dfs -rm -f /test/input/city.csv |

1. **删除文件夹**

|  |
| --- |
| -rm -r  功能：删除hdfs中的某个文件夹  示例：hdfs dfs -rm -r /test/input/ |

1. **Hdfs中的文件拷贝**

|  |
| --- |
| -cp  功能：从hdfs的一个路径拷贝hdfs的另一个路径  示例：hdfs dfs -cp /test/input/city.csv /input/city.csv |

1. **系统信息**

|  |
| --- |
| -df  功能：统计文件系统的可用空间信息  示例：hdfs dfs -df -h /    -du  功能：统计文件夹的大小信息  示例：hdfs dfs -du -s -h /test    -count  功能：统计一个指定目录下的文件节点数量  示例：hdfs dfs -count / |

#### 1.3　使用JAVA-API存储数据

##### 1.使用FileSystem访问HDFS文件系统

使用JAVA-API操作HDFS，首先要在项目中引入和hadoop相同版本的hadoopjar包进入项目如图4-4。

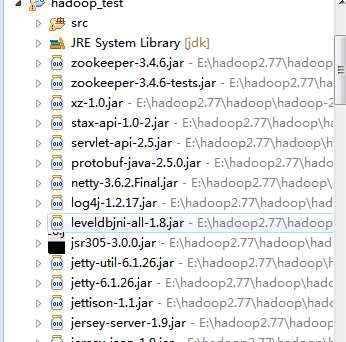


图4-4 引包效果图

【例4-1】创建一个操作HDFS的测试类：

|  |
| --- |
| **public** **class** HDFSTest {  **private** **static** **final** String ***HDFS*** = "hdfs://192.168.137.171:9000/";  **public** **static** **void** main(String[] args) **throws** Exception {  //HDFS读写的配置文件  Configuration conf = **new** Configuration();  //访问HDFS返回FileSystem 对象  FileSystem fs = FileSystem.*get*(URI.*create*(***HDFS*** ),conf);  //关闭HDFS的访问  fs.close();  }  } |

Configuration conf = **new** Configuration();是加载默认的配置文件hdfs-site.xml，new Configuration();的时候，它就会去加载jar包中的hdfs-default.xml，然后再加载classpath下的hdfs-site.xml。

fs = FileSystem.get(URI.create(uri), conf);是使用FileSystem访问HDFS文件系统。

##### 2.查看文件或目录是否存在

【例4-2】创建一个查看文件或目录是否存在的方法isExist（folder），来判断这个folder路径。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS(HDFSTest中的定义 )路径的HDFS；返回FileSystem 对象。
2. 我们通过FileSystem 对象调用exists（path）方法判断。
3. 返回true表示folder路径存在，反之不存在。
4. 调用close()方法关闭对HDFS的访问。

|  |
| --- |
| **private** **void** isExist(String folder) **throws** Exception{  FileSystem fs = FileSystem.*get*(**new** URI(***HDFS***),**new** Configuration());  Path path = **new** Path(folder);  //判断在HDFS上指定路径文件或目录是否存在  **if**(fs.exists(path)){  System.***out***.println("HDFS 上存在:" + folder);  }**else**{  System.***out***.println("HDFS 上不存在:" + folder);  }  //关闭HDFS的访问  fs.close();  } |

##### 3.创建HDFS目录

【例4-3】创建一个创建HDFS目录的方法mkdir（folder），来创个这个folder路径对应的目录。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS(HDFSTest中的定义 ) 路径的HDFS；返回FileSystem 对象。
2. 我们通过FileSystem 对象调用exists（path）方法判断。
3. 返回true表示folder路径存在，反之不存在。
4. 如果folder路径存在，在调用mkdirs(path)创建文件目录。
5. 调用close()方法关闭对HDFS的访问。

|  |
| --- |
| **public** **void** mkdir(String folder) **throws** Exception{  //与hdfs建立联系  FileSystem fs = FileSystem.*get*(**new** URI(***HDFS***),**new** Configuration());  Path path = **new** Path(folder);  **if** (!fs.exists(path)) {  //在HDFS上指定的路径创建目录  fs.mkdirs(path);  System.***out***.println("创建成功: " + folder);  }**else**{  System.***out***.println("文件夹已存在:" + folder);  }  fs.close();  } |

##### 4.删除HDFS目录

【例4-4】创建一个列出目录下的文件或目录名称的方法rmr（folder），来删除这个folder路径的目录。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS(HDFSTest中的定义 ) 路径的HDFS；返回FileSystem 对象。
2. 我们通过FileSystem 对象调用delete（path）方法删除folder对应路径的目录。
3. 调用close()方法关闭对HDFS的访问。

|  |
| --- |
| **private** **void** rmr(String folder) **throws** Exception{  //与hdfs建立联系  FileSystem fs = FileSystem.*get*(**new** URI(***HDFS***),**new** Configuration());  Path path = **new** Path(folder);  //删除HDFS上指定路径的目录  fs.delete(path);  System.***out***.println("删除成功:" + folder);  fs.close();  } |

##### 5.列出目录下的文件或目录名称

【例4-5】创建一个列出目录下的文件或目录名称的方法ls(folder)，来查看这个folder路径下的文件及目录。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS(HDFSTest中的定义 ) 路径的HDFS；返回FileSystem 对象。
2. 我们通过FileSystem 对象调用listStatus（path）获取其信息FileStatus[]对象。
3. 通过遍历FileStatus[]对象，打印输出显示。
4. 调用close()方法关闭对HDFS的访问。

|  |
| --- |
| **private** **void** ls(String folder) **throws** IOException, URISyntaxException {  //与hdfs建立联系  FileSystem fs = FileSystem.*get*(**new** URI(***HDFS***),**new** Configuration());  Path path = **new** Path(folder);  //得到该目录下的全部文件  FileStatus[] fileList = fs.listStatus(path);  **for** (FileStatus f : fileList) {  System.***out***.printf("name: %s | folder: %s | size: %d\n", f.getPath(), f.isDir() , f.getLen());  }  fs.close();  } |

##### 6.上传文件

【例4-6】创建一个上传文件的方法put(local,remote)，把文件从local下载到remote。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS(HDFSTest中的定义 ) 路径的HDFS；返回FileSystem 对象；
2. 在开始下面操作前请确认local 本地路径以及remote HDFS上路径的正确性；
3. 我们通过FileSystem 对象调用copyFromLocalFile(local,remote)即可；
4. 调用close()方法关闭对HDFS的访问。

|  |
| --- |
| **private** **void** put(String local, String remote) **throws** Exception {  // 建立联系  FileSystem fs = FileSystem.*get*(**new** URI(***HDFS***), **new** Configuration());  fs.copyFromLocalFile(**new** Path(local), **new** Path(remote));  System.***out***.println("从:" + local + " 上传到 : " + remote);  fs.close();  } |

##### 7.下载文件

【例4-7】创建一个下载文件的方法get(local,remote)，把文件从remote下载到local。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS(HDFSTest中的定义 ) 路径的HDFS；返回FileSystem 对象。
2. 在开始下面操作前请确认local 本地路径以及remote HDFS上路径的正确性；
3. 我们通过FileSystem 对象调用copyToLocalFile(local,remote)即可。
4. 调用close()方法关闭对HDFS的访问。

|  |
| --- |
| **private** **void** get(String remote, String local) **throws** Exception {  // 建立联系  FileSystem fs = FileSystem.*get*(**new** URI(***HDFS***), **new** Configuration());  fs.copyToLocalFile(**new** Path(remote), **new** Path(local));  System.***out***.println("从 : " + remote + " 下载到 :" + local);  fs.close();  } |

##### 8.删除文件

【例4-8】创建一个显示文件的方法rm(folder)，来删除这个folder路径的文件。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS(HDFSTest中的定义 ) 路径的HDFS；返回FileSystem 对象；
2. 我们通过FileSystem 对象调用deleteOnExit(path)判断文件是否存在；
3. 文件存在，再调用delete(path)方法，删除文件即可。
4. 调用close()方法关闭对HDFS的访问。

|  |
| --- |
| **private** **void** rm(String folder) **throws** Exception {  //与hdfs建立联系  FileSystem fs = FileSystem.*get*(**new** URI(***HDFS***),**new** Configuration());  Path path = **new** Path(folder);  **if**(fs.deleteOnExit(path)){  fs.delete(path);  System.***out***.println("删除成功:" + folder);  }**else**{  System.***out***.println("文件不存在！");  }  fs.close();  } |

##### 9.显示文件

【例4-9】创建一个显示文件的方法cat(folder)，来查看这个folder路径对应的文件。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS(HDFSTest中的定义 ) 路径的HDFS；返回FileSystem 对象；
2. 我们通过FileSystem 对象调用open(folder),返回FSDataInputStream 对象；
3. 在用IOUtils解析FSDataInputStream 对象对象即可。
4. 调用close()方法关闭对HDFS的访问。

|  |
| --- |
| **private** **void** cat(String folder) **throws** IOException, URISyntaxException {  // 与hdfs建立联系  FileSystem fs = FileSystem.*get*(**new** URI(***HDFS***),**new** Configuration());  Path path = **new** Path(folder);  FSDataInputStream fsdis = **null**;  System.***out***.println("cat: " + folder);  **try** {  fsdis =fs.open(path);  IOUtils.*copyBytes*(fsdis, System.***out***, 4096, **false**);  } **finally** {  IOUtils.*closeStream*(fsdis);  fs.close();  }  } |

### 任务2 HDFS文件存储格式

#### 2.1　存储格式概述

hdfs 文件存储格式大致上分为两大类 面向行的行存储和面向列的列存储两类。

行存储：同时将一整行的数据存储在一起的一种连续的存储方式,例如SequenceFile,MapFile,Avro Datafile。采用这种连续的存储方式，如果只需要访问某行中的一小段数据，必须需要将整行读入内存中，推迟序列化可以缓解这个缺点，但是读取整行数据是不可避免。行存储适用于将一整行数据需要同时处理的情况下。

列存储：列存储会把文件切分成若干列分数据,每一列存储数据在一起,在需要那一列数据时读取那一列,不需要的就可以跳过不需要的列,适用于处理行的一小部分字段的情况,但是这种格式的读写需要更多的内存空间来缓存行在内存中。例如parquet ORCfile,RCfile,列存储不适合流式写入,写入一旦失败文件将无法恢复，而行存储的数据在写入失败时可以返回到最后的同步点继续写入，所以flume采用行存储。

#### 2.2　行存储

##### 1.SequenceFile 存储格式

[SequenceFile](http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/io/SequenceFile.html)是一个由二进制key/value对组成的平面文件。它可以在map/reduce过程中的input/output 的format时被使用。在map/reduce过程中，map处理文件的临时输出就是使用SequenceFile处理过的。

SequenceFile分别提供了Writer，Reader和Sorter类，用于写，读和排序。

SequenceFile的操作中有三种压缩方式：

不压缩数据直接存储键/值。

压缩value值不压缩key值存储的存储方式。

key/value值都压缩的方式存储。

SequenceFile提供了若干Writer的构造静态获取 ,所述[SequenceFile.Reader](http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/io/SequenceFile.Reader.html)充当桥接模式和可以读取SequenceFile.Writer中的任何方式的压缩数据。三种不同的压缩方式是共用一个数据头，流方式的读取会先读取头字节去判断是哪种方式的压缩，然后根据压缩方式去解压缩并反序列化字节流数据，得到可识别的数据。

SequenceFile也各有优缺点：

SequenceFile优点：

1.支持基于记录(Record)或块(Block)的数据压缩。

2.支持splitable，能够作为MapReduce的输入分片。

3.修改简单：主要负责修改相应的业务逻辑，而不用考虑具体的存储格式。

SequenceFile的缺点 :需要一个合并文件的过程，且合并后的文件不方便查看。

【例4-10】实例测试：

写入：这里先创建一个写入的方法sequenceFileWriter(urls)，urls是要写入文件的 DFS文件路径。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS；返回FileSystem 对象fs；
2. 再通过SequenceFile.createWriter（）,返回SequenceFile.Writer对象writer；
3. 在用writer.append(key, value)写入。
4. 最后调用IOUtils.*closeStream*(writer)关闭writer对象。

读取：这里先创建一个写入的方法sequenceFileReader(urls)，urls是要读取文件的 DFS文件路径。

1. 首先我们通过FileSystem.get(conf)方法访问HDFS；返回FileSystem 对象fs；
2. 再通过 SequenceFile.Reader（）,返回SequenceFile.Reader 对象reader；
3. 在用分别使用2个对象接收key、value。
4. 在用 **while** (reader.next(key, value))遍历
5. 最后调用IOUtils.*closeStream*(reader)关闭reader对象。

|  |
| --- |
| **import** org.apache.hadoop.conf.Configuration;  **import** org.apache.hadoop.fs.FileSystem;  **import** org.apache.hadoop.fs.Path;  **import** org.apache.hadoop.io.\*;  **import** org.apache.hadoop.util.ReflectionUtils;  **import** java.io.IOException;  **import** java.net.URI;  **public** **class** SequenceFileTest {  **public** **static** **final** String ***path*** = "hdfs://192.168.137.171:9000/test/test";  **private** **static** **final** String[] ***DATA*** = { "asdf", "bbbb", "cccc", "dddd"};    **public** **static** **void** sequenceFileWriter(String urls) **throws** IOException {  Configuration conf = **new** Configuration();  //链接hdfs://192.168.137.171:9000/返回FileSystem对象fs  FileSystem fs = FileSystem.*get*(URI.*create*("hdfs://192.168.137.171:9000/"),conf);  Path path = **new** Path(urls);  @SuppressWarnings("deprecation")  // 不压缩  //SequenceFile.Writer writer = SequenceFile.createWriter(fs, conf, path, IntWritable.class, Text.class,SequenceFile.CompressionType.NONE);  // value压缩  //SequenceFile.Writer writer = SequenceFile.createWriter(fs, conf, path, IntWritable.class, Text.class,SequenceFile.CompressionType.RECORD);  //key|value压缩|块压缩  //SequenceFile.createWriter() 返回SequenceFile.Writer对象 writer  //path 是SequenceFile.Writer要写入文件的 DFS文件路径  //fs 是链接 访问的HDFS  //Text.class、 IntWritable.class 是写入的数据的key和value  SequenceFile.Writer writer = SequenceFile.createWriter(fs, conf, path, Text.**class**, IntWritable.**class**,SequenceFile.CompressionType.***BLOCK***);  Text key = **new** Text();  IntWritable value = **new** IntWritable();  **for**(**int** i = 0; i < ***DATA***.length; i++) {  key.set(***DATA***[i]);  value.set(i);  System.***out***.println("["+writer.getLength()+"] "+key+" "+value+"" );  writer.append(key, value);  }  IOUtils.*closeStream*(writer);  }  **public** **static** **void** sequenceFileReader(String urls) **throws** IOException {  Configuration conf = **new** Configuration();  FileSystem fs = FileSystem.*get*(URI.*create*("hdfs://192.168.137.171:9000/"),conf);  Path path = **new** Path(urls);  @SuppressWarnings("deprecation")  //path 是SequenceFile.Writer要写入文件的 DFS文件路径  //fs 是链接 访问的HDFS  SequenceFile.Reader reader = **new** SequenceFile.Reader(fs, path, conf);  //分别获取可以和value  Writable key = (Writable) ReflectionUtils.*newInstance*(reader.getKeyClass(), conf);  Writable value = (Writable) ReflectionUtils.*newInstance*(reader.getValueClass(), conf);  **while** (reader.next(key, value)) {    System.***out***.println(""+key+" "+value+"");  }  IOUtils.*closeStream*(reader);  }  **public** **static** **void** main(String[] args) **throws** IOException {  System.***out***.println("运行结果如下：");  *sequenceFileWriter*(***path***);  *sequenceFileReader*(***path***);  }  } |

运行结果如图4-5所示。

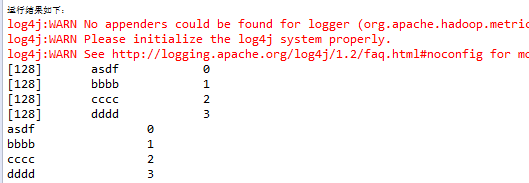


图4-5 【例4-10】运行结果图

##### 2.MapFile 存储格式

MapFile由两部分组成，分别是data和index，index可以看做文件的数据索引，主要记录了每个Record的key值，以及该Record在文件中的偏移位置。MapFile被访问时，首先加载索引文件index到内存，通过索引文件内的映射关系可快速锁定Record所在文件位置，这个方式HDFS方式较类似。与SequenceFile想比较MapFile的检索效率是高效的，但是消耗的内存也相对较高。

【例4-11】实例测试：

|  |
| --- |
| **import** java.io.IOException;  **import** java.net.URI;  **import** org.apache.hadoop.conf.Configuration;  **import** org.apache.hadoop.fs.FileSystem;  **import** org.apache.hadoop.io.IOUtils;  **import** org.apache.hadoop.io.IntWritable;  **import** org.apache.hadoop.io.MapFile;  **import** org.apache.hadoop.io.Text;    **public** **class** MapFileTest {    **public** **static** **final** String ***path*** = "hdfs://192.168.137.171:9000/test/MapFile";  **public** **static** **final** String ***HDFS*** = "hdfs://192.168.137.171:9000/";  **private** **static** **final** String[] ***DATA*** = { "asdf", "bbbb", "cccc", "dddd"};    **private** **static** **void** mapFileWriter() **throws** IOException {  Configuration conf = **new** Configuration();  FileSystem fs = FileSystem.*get*(URI.*create*(***HDFS***), conf);  MapFile.Writer writer = **new** MapFile.Writer(conf, fs, ***path***, Text.**class**, IntWritable.**class**);  Text key = **new** Text();  IntWritable value = **new** IntWritable();  **for**(**int** i = 0; i < ***DATA***.length; i++) {  key.set(***DATA***[i]);  value.set(i);  System.***out***.println("["+i+"] "+key+" "+value+"" );  writer.append(key, value);  }  //通过writer向文档中写入记录  IOUtils.*closeStream*(writer);//关闭write流  }    **private** **static** **void** mapFileReader() **throws** IOException {  Configuration conf = **new** Configuration();  FileSystem fs = FileSystem.*get*(URI.*create*(***HDFS***), conf);  MapFile.Reader reader = **new** MapFile.Reader(fs, ***path***, conf);  //通过writer向文档中写入记录  Text key = **new** Text();  IntWritable value = **new** IntWritable();  **while** (reader.next(key, value)) {  System.***out***.println(key +" "+value);  }  IOUtils.*closeStream*(reader);//关闭write流  }  **public** **static** **void** main(String[] args) **throws** IOException {  *mapFileWriter*();  *mapFileReader*();  }  } |

#### 2.2　列存储

##### 2.2.1 Parquet存储格式

Parquet文件是以二进制方式存储的，是不可以直接读取和修改的，Parquet文件是自解析的，文件中包括该文件的数据和元数据。在HDFS文件系统和Parquet文件中存在如下几个概念：

块（Block）：Block是HDFS上的最小的副本单位，HDFS会把一个Block存储在本地的一个文件同时维护分散在不同的机器上的多个副本中，通常情况下每个Block的大小为256M、512M等。

HDFS文件：一个HDFS的文件，必须包含文件元数据，它不需要实际包含数据，实际的数据分散存储在多个Block中。

行组：按照行将数据物理上划分为多个单元，每一个行组包含一定的行数，在一个HDFS文件中至少存储一个行组，Parquet读写的时候会将整个行组缓存在内存中，所以如果每一个行组的大小是由内存大的小决定的。

列块(Column Chunk)：在一个行组中每一列保存在一个列块中，行组中的所有列连续的存储在这个行组文件中。不同的列块可能使用不同的算法进行压缩。

页(Page)：每一个列块划分为多个页，一个页是最小的编码的单位，在同一个列块的不同页可能使用不同的编码方式。

通常情况下，存储Parquet数据的时，会按照HDFS的Block大小设置行组的大小，由于每一个Mapper任务处理数据的最小单位是一个Block，这样可以把每一个行组由一个Mapper任务处理，增大任务执行并行度。

Parquet文件的格式如下图图4-5所示：

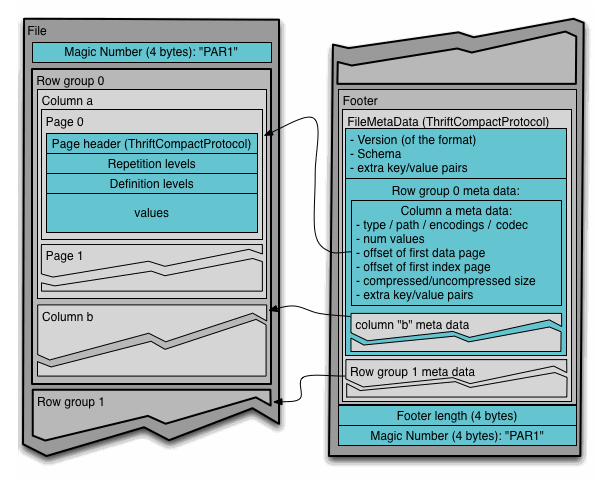


图4-5 Parquet文件的格式

##### 2.2.1 ORC文件格式

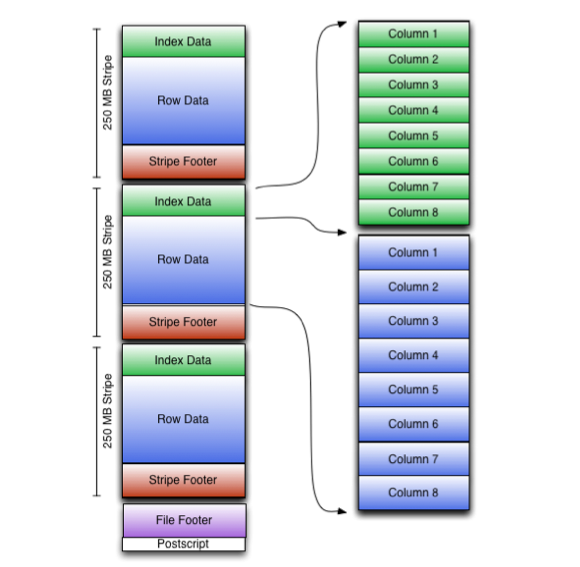
ORC和Parquet类似，也是以二进制方式存储的，所以也是不可以直接读取和修改的，ORC文件也是自解析的，它包含许多的元数据，这些元数据都是同构ProtoBuffer进行序列化的。ORC的文件结构入图6，其中涉及到如下的概念：

图4-6 ORC文件结构

1. ORC文件：保存在文件系统上的普通二进制文件，一个ORC文件中可以包含多个stripe，每一个stripe包含多条记录，这些记录按照列进行独立存储，对应到Parquet中的row group的概念。
2. 文件级元数据：包括文件的描述信息PostScript、文件meta信息（包括整个文件的统计信息）、所有stripe的信息和文件schema信息。
3. stripe：一组行形成一个stripe，每次读取文件是以行组为单位的，一般为HDFS的块大小，保存了每一列的索引和数据。
4. stripe元数据：保存stripe的位置、每一个列的在该stripe的统计信息以及所有的stream类型和位置。
5. row group：索引的最小单位，一个stripe中包含多个row group，默认为10000个值组成。
6. stream：一个stream表示文件中一段有效的数据，包括索引和数据两类。索引stream保存每一个row group的位置和统计信息，数据stream包括多种类型的数据，具体需要哪几种是由该列类型和编码方式决定。

在ORC文件中保存了三个层级的统计信息，分别为文件级别、stripe级别和row group级别的，他们都可以用来根据Search ARGuments（谓词下推条件）判断是否可以跳过某些数据，在统计信息中都包含成员数和是否有null值，并且对于不同类型的数据设置一些特定的统计信息。