北京交通大學

数据采集与存储

学　号：　16281002

姓　名：　杜永坤

专　业：　计算机科学与技术

学　院：　计算机与信息技术学院

提交日期：　2019 年 05 月 27 日

目录

实验二
数据采集存储实验

1 实验目的

1. 理解 Kafka、HDFS、HBase、Hive 在 Hadoop 体系结构中的角色；
2. 熟悉 HDFS、Hbase 操作常用的 Shell 命令；
3. 熟悉 HDFS、HBase 操作常用的 Java API;
4. 熟悉 Hive 的 DDL 命令与 DML 操作;

2 实验环境

实验平台：基于实验一搭建的虚拟机 Hadoop 大数据实验平台上的 KAFKA 集群、HDFS、HBASE、HIVE；

编程语言：JAVA；

3 实验内容

3.1KAFKA 数据采集

3.1.1 使用 Kafka Shell 命令完成以下任务：

(1) 创建任意 topic
(2) 创建向该 topic 发送数据的生产者
(3) 创建订阅该 topic 的消费者

创建 topic

在 cluster1 上创建名称为 mykafka 的 topic，查看信息

```
[hadoop@cluster1 ~]$ kafka-topics.sh --create --zookeeper cluster1:2181,cluster2
:2181,cluster3:2181 --replication-factor 3 --partitions 1 --topic mykafka
Error while executing topic command Topic "mykafka" already exists.
kafka.common.TopicExistsException: Topic "mykafka" already exists.
        at kafka.admin.AdminUtils$.createOrUpdateTopicPartitionAssignmentPathInZ
K(AdminUtils.scala:187)
        at kafka.admin.AdminUtils$.createTopic(AdminUtils.scala:172)
        at kafka.admin.TopicCommand$.createTopic(TopicCommand.scala:93)
        at kafka.admin.TopicCommand$.main(TopicCommand.scala:55)
        at kafka.admin.TopicCommand.main(TopicCommand.scala)

[hadoop@cluster1 ~]$ kafka-topics.sh --list --zookeeper cluster1:2181,cluster2:2
181,cluster3:2181
mykafka
[hadoop@cluster1 ~]$ kafka-topics.sh --describe --zookeeper cluster1:2181,cluste
r2:2181,cluster3:2181
Topic:mykafka   PartitionCount:1        ReplicationFactor:3     Configs:
        Topic: mykafka  Partition: 0    Leader: 1       Replicas: 2,3,1 Isr: 1,2
,3
[hadoop@cluster1 ~]$ kafka-console-producer.sh --broker-list localhost:9092 --to
pic mykafka
[2019-05-17 14:15:12,684] WARN Property topic is not valid (kafka.utils.Verifiab
leProperties)
```

生产者

在 cluster1 上使用 Kafka-console-producer.sh，创建 producer，这终端就是输入源

消费者

在 cluster2 上使用 Kafka-console-consumer.sh，创建 consumer，终端显示 producer 在终端输入的信息

```
[2019-05-17 14:12:05,077] INFO Truncating log mykafka-0 to offset 4. (kafka.log.
Log)
[2019-05-17 14:12:05,106] INFO [ReplicaFetcherManager on broker 2] Added fetcher
 for partitions List([[mykafka,0], initOffset 4 to broker id:1,host:192.168.56.1
21,port:9092] ) (kafka.server.ReplicaFetcherManager)
[2019-05-17 14:12:05,113] INFO [ReplicaFetcherThread-0-1], Starting  (kafka.serv
er.ReplicaFetcherThread)

[hadoop@cluster2 ~]$
[hadoop@cluster2 ~]$ kafka-console-consumer.sh -zookeeper cluster1:2181,cluster2
:2181,cluster3:2181 --topic mykafka --from-beginning
[2019-05-17 14:17:00,167] INFO [ReplicaFetcherManager on broker 2] Removed fetch
er for partitions [mykafka,0] (kafka.server.ReplicaFetcherManager)
[2019-05-17 14:17:00,177] INFO [ReplicaFetcherThread-0-1], Shutting down (kafka.
server.ReplicaFetcherThread)
[2019-05-17 14:17:00,222] ERROR [ConsumerFetcherThread-console-consumer-46417_cl
uster2-1558073725258-72f5d338-0-1], Error for partition [mykafka,0] to broker 1:
class kafka.common.NotLeaderForPartitionException (kafka.consumer.ConsumerFetche
rThread)
[2019-05-17 14:17:00,508] INFO [ReplicaFetcherThread-0-1], Stopped  (kafka.serve
r.ReplicaFetcherThread)
[2019-05-17 14:17:00,509] INFO [ReplicaFetcherThread-0-1], Shutdown completed (k
afka.server.ReplicaFetcherThread)
```

```
hadoop@cluster1:~
failed due to Leader not local for partition [mykafka,0] on broker 1 (kafka.serv
er.ReplicaManager)
[2019-05-17 14:16:59,118] INFO Closing socket connection to /192.168.56.122. (ka
fka.network.Processor)
[2019-05-17 14:16:59,122] INFO Closing socket connection to /192.168.56.122. (ka
fka.network.Processor)
[2019-05-17 14:16:59,375] WARN [Replica Manager on Broker 1]: Fetch request with
 correlation id 586 from client ReplicaFetcherThread-0-1 on partition [mykafka,0
] failed due to Leader not local for partition [mykafka,0] on broker 1 (kafka.se
rver.ReplicaManager)
[2019-05-17 14:16:59,378] INFO Closing socket connection to /192.168.56.122. (ka
fka.network.Processor)
[2019-05-17 14:16:59,453] WARN [Replica Manager on Broker 1]: Fetch request with
 correlation id 577 from client ReplicaFetcherThread-0-1 on partition [mykafka,0
] failed due to Leader not local for partition [mykafka,0] on broker 1 (kafka.se
rver.ReplicaManager)
[2019-05-17 14:16:59,456] INFO Closing socket connection to /192.168.56.123. (ka
fka.network.Processor)
testinfo
[2019-05-17 14:19:02,972] INFO Closing socket connection to /127.0.0.1. (kafka.n
etwork.Processor)
testinfo 2
testinfo 3
```

```
hadoop@cluster2:~
 for partitions List([[mykafka,0], initOffset 4 to broker id:1,host:192.168.56.1
21,port:9092] ) (kafka.server.ReplicaFetcherManager)
[2019-05-17 14:12:05,113] INFO [ReplicaFetcherThread-0-1], Starting  (kafka.serv
er.ReplicaFetcherThread)

[hadoop@cluster2 ~]$
[hadoop@cluster2 ~]$ kafka-console-consumer.sh -zookeeper cluster1:2181,cluster2
:2181,cluster3:2181 --topic mykafka --from-beginning
[2019-05-17 14:17:00,167] INFO [ReplicaFetcherManager on broker 2] Removed fetch
er for partitions [mykafka,0] (kafka.server.ReplicaFetcherManager)
[2019-05-17 14:17:00,177] INFO [ReplicaFetcherThread-0-1], Shutting down (kafka.
server.ReplicaFetcherThread)
[2019-05-17 14:17:00,222] ERROR [ConsumerFetcherThread-console-consumer-46417_cl
uster2-1558073725258-72f5d338-0-1], Error for partition [mykafka,0] to broker 1:
class kafka.common.NotLeaderForPartitionException (kafka.consumer.ConsumerFetche
rThread)
[2019-05-17 14:17:00,508] INFO [ReplicaFetcherThread-0-1], Stopped  (kafka.serve
r.ReplicaFetcherThread)
[2019-05-17 14:17:00,509] INFO [ReplicaFetcherThread-0-1], Shutdown completed (k
afka.server.ReplicaFetcherThread)
testinfo
testinfo 2
testinfo 3
```

分别在 cluster2cluster1 上使用 **Ctrl+C** 退出。

退出后终端会显示传输内容的信息：

```
hadoop@cluster1:~
failed due to Leader not local for partition [mykafka,0] on broker 1 (kafka.serv
er.ReplicaManager)
[2019-05-17 14:16:59,118] INFO Closing socket connection to /192.168.56.122. (ka
fka.network.Processor)
[2019-05-17 14:16:59,122] INFO Closing socket connection to /192.168.56.122. (ka
fka.network.Processor)
[2019-05-17 14:16:59,375] WARN [Replica Manager on Broker 1]: Fetch request with
 correlation id 586 from client ReplicaFetcherThread-0-1 on partition [mykafka,0
] failed due to Leader not local for partition [mykafka,0] on broker 1 (kafka.se
rver.ReplicaManager)
[2019-05-17 14:16:59,378] INFO Closing socket connection to /192.168.56.122. (ka
fka.network.Processor)
[2019-05-17 14:16:59,453] WARN [Replica Manager on Broker 1]: Fetch request with
 correlation id 577 from client ReplicaFetcherThread-0-1 on partition [mykafka,0
] failed due to Leader not local for partition [mykafka,0] on broker 1 (kafka.se
rver.ReplicaManager)
[2019-05-17 14:16:59,456] INFO Closing socket connection to /192.168.56.123. (ka
fka.network.Processor)
testinfo
[2019-05-17 14:19:02,972] INFO Closing socket connection to /127.0.0.1. (kafka.n
etwork.Processor)
testinfo 2
testinfo 3
^C[hadoop@cluster1 ~]$
```

```
hadoop@cluster2:~
[2019-05-17 14:17:00,509] INFO [ReplicaFetcherThread-0-1], Shut
afka.server.ReplicaFetcherThread)
testinfo
testinfo 2
testinfo 3
^C[2019-05-17 14:22:33,000] ERROR Closing socket for /192.168.5
error (kafka.network.Processor)
java.io.IOException: Connection reset by peer
        at sun.nio.ch.FileDispatcherImpl.read0(Native Method)
        at sun.nio.ch.SocketDispatcher.read(SocketDispatcher.ja
        at sun.nio.ch.IOUtil.readIntoNativeBuffer(IOUtil.java:2
        at sun.nio.ch.IOUtil.read(IOUtil.java:197)
        at sun.nio.ch.SocketChannelImpl.read(SocketChannelImpl.
        at kafka.utils.Utils$.read(Utils.scala:380)
        at kafka.network.BoundedByteBufferReceive.readFrom(Boun
ive.scala:54)
        at kafka.network.Processor.read(SocketServer.scala:444)
        at kafka.network.Processor.run(SocketServer.scala:340)
        at java.lang.Thread.run(Thread.java:745)
Consumed 3 messages
[hadoop@cluster2 ~]$ ^C
[hadoop@cluster2 ~]$ [2019-05-17 14:22:37,244] INFO Closing soc
 /192.168.56.121. (kafka.network.Processor)
```

3.1.2 使用 Java API 编程实现以下任务：

(1) 实现生产者程序，向指定 topic 发送数据
(2) 实现消费者程序，从（1）中指定的 topic 中订阅数据并将消费得到的数据存到本地文件中。

生产者：

Java 代码：

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;

import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class MyProducer {

 public static void main(String[] args) {
        Properties props = new Properties();
        props.put("serializer.class",
"kafka.serializer.StringEncoder");
        props.put("metadata.broker.list", "localhost:9092");
    Producer<Integer, String> producer = new Producer<Integer,
String>(new ProducerConfig(props));
    String topic = "mykafka";

    File file = new File("testdata.txt");
    BufferedReader reader = null;
    try {
```

```
            reader = new BufferedReader(new FileReader(file));
            String tempString = null;
            int line = 1;
            while ((tempString = reader.readLine()) != null) {
                producer.send(new  KeyedMessage<Integer,  String>(topic,
tempString));
                System.out.println("Success  send  [" + line + "]
message ..");
                line++;
            }
            reader.close();
            System.out.println("Total send [" + line + "] messages ..");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e1) {}
            }
        }
        producer.close();

    }
}
```

编译 javac -cp /usr/local/kafka_2.10-0.8.2.1/libs/*: MyProducer.java
执行 java -cp /usr/local/kafka_2.10-0.8.2.1/libs/*: MyProducer

消费者：

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.io.File;
import java.io.FileNotFoundException;
```

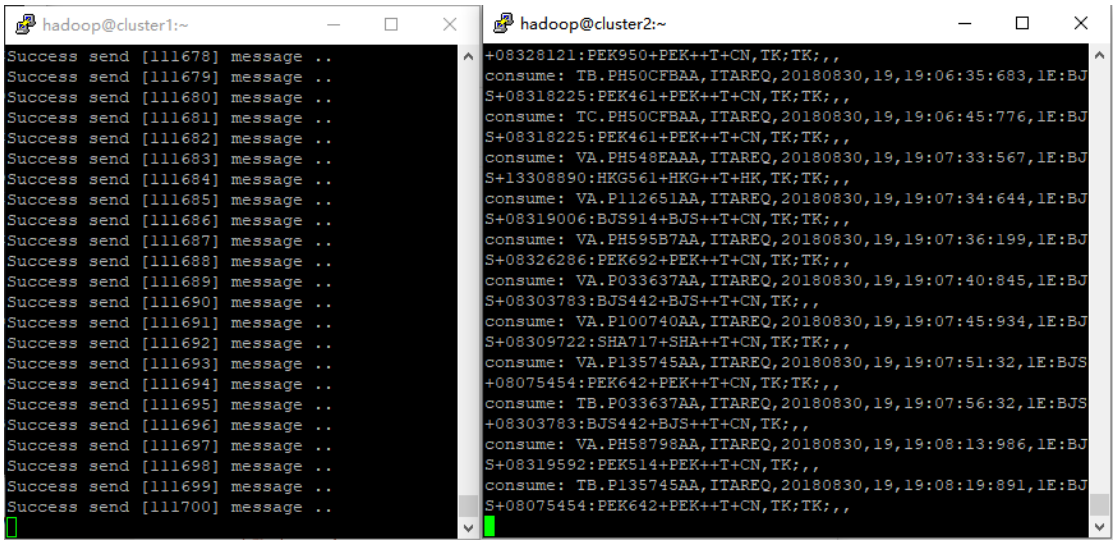
```
import java.io.FileReader;
import java.io.FileWriter;
import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

public class MyConsumer {

 public static void main(String[] args) {
     File fp = new File("testdata1.txt");
     static FileWriter fw = null;
     try {
         if (!fp.exists()) {
             fp.createNewFile(); // 创建输出的中间文件
         }
         fw = new FileWriter(fp);
     } catch (IOException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
     }

     String topic = "mykafka";

     ConsumerConnector consumer =
Consumer.createJavaConsumerConnector(createConsumerConfig());
     Map<String, Integer> topicCountMap = new HashMap<String,
Integer>();
     topicCountMap.put(topic, new Integer(1));
     Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
consumer.createMessageStreams(topicCountMap);
     KafkaStream<byte[], byte[]> stream =
consumerMap.get(topic).get(0);
     ConsumerIterator<byte[], byte[]> it = stream.iterator();
```

```
        while(it.hasNext())
        {try {
            fw.write(new String(it.next().message()));
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

            System.out.println("consume:          "          +          new
String(it.next().message()));
        }

    }

    private static ConsumerConfig createConsumerConfig() {
        Properties props = new Properties();
        props.put("group.id","group1");

props.put("zookeeper.connect","cluster1:2181,cluster2:2181,cluster3:
2181");
        props.put("zookeeper.session.timeout.ms", "400");
        props.put("zookeeper.sync.time.ms", "200");
        props.put("auto.commit.interval.ms", "1000");
        return new ConsumerConfig(props);
      }
}
```

编译 javac -cp /usr/local/kafka_2.10-0.8.2.1/libs/*: MyConsumer.java
执行 java -cp /usr/local/kafka_2.10-0.8.2.1/libs/*: MyConsumer

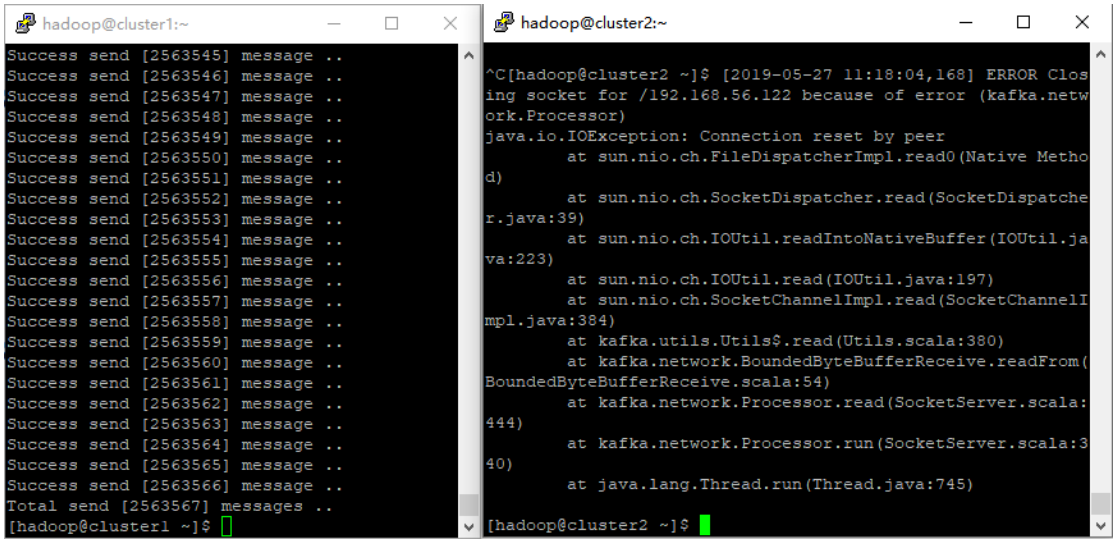
结果展示：

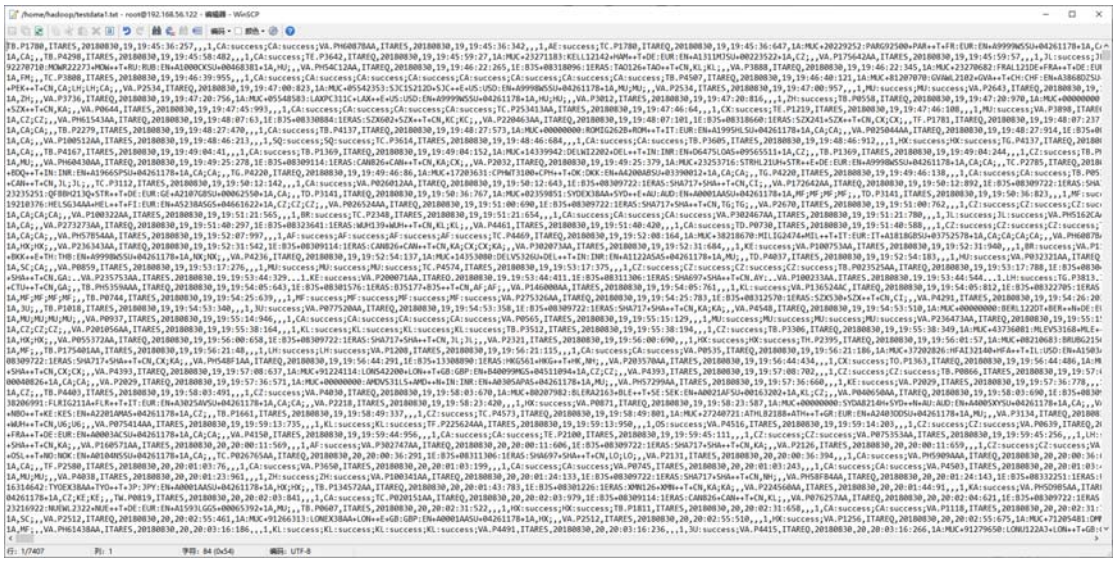
运行过程中：
Cluster1 运行生产者的程序，从数据集中获得数据，然后上传到 topic，然后 cluster2 上运行了消费者程序，消费 topic 的数据，并将数据保存到了本地。



运行结束后：

Cluster2 上多出保存数据的文件：





3.2 数据集介绍

本课程所选用的数据集由中国航信提供，是真实 GDS 系统预定日志。其中包日志数据分组标识，日志类型，时间戳等类型，每一行数据对应一条日志。数据样例如下：

```
VA.P2594,ITARES,20180830,19,19:45:37:470,,,1,MU:success;MU:success;
VA.P3928,ITARES,20180830,19,19:45:37:482,,,1,CZ:success;CZ:success;CZ:success;
VA.P4424,ITARES,20180830,19,19:45:37:511,,,1,CA:success;CA:success;
```

3.2.HDFS 数据存储

1) 利用 Hadoop 提供的 Shell 命令完成以下任务：

(1) 向 HDFS 中上传任意文本文件，如果指定的文件在 HDFS 中已经存在，由用户指定是追加到原有文件末尾还是覆盖原有的文件；

先 上传一个测试文件 testfile，内容如下：

```
hadoop@cluster1:~
oop-nodemanager-cluster2.out
cluster3: starting nodemanager, logging to /usr/local/hadoop-2.6.5/logs/yarn-had
oop-nodemanager-cluster3.out
cluster1: starting nodemanager, logging to /usr/local/hadoop-2.6.5/logs/yarn-had
oop-nodemanager-cluster1.out
[hadoop@cluster1 ~]$ ls
MyProducer.class  perl5    run.sh        testfile
MyProducer.java   run1.sh  testdata.txt  work
[hadoop@cluster1 ~]$ hdfs dfs -ls
ls: `.': No such file or directory
[hadoop@cluster1 ~]$ hdfs dfs -ls /
Found 1 items
drwxr-xr-x   - hadoop supergroup          0 2019-05-21 16:11 /test
[hadoop@cluster1 ~]$ hdfs dfs -ls /test
[hadoop@cluster1 ~]$ hdfs dfs -put testfile /test
[hadoop@cluster1 ~]$ hdfs dfs -ls /test
Found 1 items
-rw-r--r--   3 hadoop supergroup          7 2019-05-26 13:18 /test/testfile
[hadoop@cluster1 ~]$ hdfs dfs -cat /test/testfile
1
2
3

[hadoop@cluster1 ~]$
```

修改本地 testfile 的内容为 456

先是设定为追加：

```
[hadoop@cluster1 ~]$ hdfs dfs -appendToFile testfile /test/testfile
[hadoop@cluster1 ~]$ hdfs dfs -cat /test/testfile
1
2
3

4
5
6
[hadoop@cluster1 ~]$
```

执行命令后查看 testfile 内容

然后是覆盖直接使用 put -f 命令即可

```
[hadoop@cluster1 ~]$ hdfs dfs -put -f testfile /test/
[hadoop@cluster1 ~]$ hdfs dfs -cat /test/testfile
4
5
6
[hadoop@cluster1 ~]$
```

Java API 实现：

```
import java.io.FileInputStream;
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class CopyFromLocalFile {
    /**
     * 判断路径是否存在
     */
    public static boolean test(Configuration conf, String path) {
        try (FileSystem fs = FileSystem.get(conf)) {
            return fs.exists(new Path(path));
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }
    /**
     * 复制文件到指定路径 若路径已存在，则进行覆盖
     */
    public static void copyFromLocalFile(Configuration conf,
            String localFilePath, String remoteFilePath) {
        Path localPath = new Path(localFilePath);
        Path remotePath = new Path(remoteFilePath);
        try (FileSystem fs = FileSystem.get(conf)) {
            /* fs.copyFromLocalFile 第一个参数表示是否删除源文件，第二个
```

```
参数表示是否覆盖 */
            fs.copyFromLocalFile(false, true, localPath, remotePath);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     * 追加文件内容
     */
    public static void appendToFile(Configuration conf, String
localFilePath,
            String remoteFilePath) {
        Path remotePath = new Path(remoteFilePath);
        try (FileSystem fs = FileSystem.get(conf);
                FileInputStream in = new FileInputStream(localFilePath);)
{
            FSDataOutputStream out = fs.append(remotePath);
            byte[] data = new byte[1024];
            int read = -1;
            while ((read = in.read(data)) > 0) {
                out.write(data, 0, read);
            }
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 主函数
     */
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://localhost:9000");
        String localFilePath = "/home/hadoop/textfile"; // 本地路径
```

```
        String remoteFilePath = "/test/textfile"; // HDFS路径
        // String choice = "append"; // 若文件存在则追加到文件末尾
        String choice = "overwrite"; // 若文件存在则覆盖

        try {
            /* 判断文件是否存在 */
            boolean fileExists = false;
            if (CopyFromLocalFile.test(conf, remoteFilePath)) {
                fileExists = true;
                System.out.println(remoteFilePath + " 已存在.");
            } else {
                System.out.println(remoteFilePath + " 不存在.");
            }
            /* 进行处理 */
            if (!fileExists) { // 文件不存在，则上传
                CopyFromLocalFile.copyFromLocalFile(conf, localFilePath,
                        remoteFilePath);
                System.out.println(localFilePath + " 已上传至 " +
remoteFilePath);
            } else if (choice.equals("overwrite")) { // 选择覆盖
                CopyFromLocalFile.copyFromLocalFile(conf, localFilePath,
                        remoteFilePath);
                System.out.println(localFilePath + " 已覆盖 " +
remoteFilePath);
            } else if (choice.equals("append")) { // 选择追加
                CopyFromLocalFile.appendToFile(conf, localFilePath,
                        remoteFilePath);
                System.out.println(localFilePath + " 已追加至 " +
remoteFilePath);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

(2) 从 HDFS 中下载指定文件，如果本地文件与要下载的文件名称相同，则自动对下载的文件重命名；

```
[hadoop@cluster1 ~]$ if (hdfs dfs -test -e ~/testfile);
> then $(hdfs dfs -copyToLocal testfile ./testfile);
> else $(hdfs dfs -copyToLocal testfile ./testfile2);
> fi
[hadoop@cluster1 ~]$ ls
MyProducer.class  perl5    run.sh        testfile   work
MyProducer.java   run1.sh  testdata.txt  testfile2
[hadoop@cluster1 ~]$
```

可以看到本地存在 testfile 文件，执行命令后，新下载的文件被命名为 testfile2

JavaAPI 实现：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.fs.FileSystem;

import java.io.*;

public class CopyToLocal {
    /**
     * 下载文件到本地 判断本地路径是否已存在，若已存在，则自动进行重命名
     */
    public static void copyToLocal(Configuration conf, String remoteFilePath,
            String localFilePath) {
        Path remotePath = new Path(remoteFilePath);
        try (FileSystem fs = FileSystem.get(conf)) {
            File f = new File(localFilePath);
            /* 如果文件名存在，自动重命名(在文件名后面加上 _0, _1...) */
            if (f.exists()) {
                System.out.println(localFilePath + " 已存在.");
                Integer i = Integer.valueOf(0);
                while (true) {
                    f = new File(localFilePath + "_" + i.toString());
                    if (!f.exists()) {
                        localFilePath = localFilePath + "_" + i.toString();
                        break;
```

```
                } else {
                    i++;
                    continue;
                }
            }
            System.out.println("将重新命名为: " + localFilePath);
        }
        // 下载文件到本地
        Path localPath = new Path(localFilePath);
        fs.copyToLocalFile(remotePath, localPath);
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
/**
 * 主函数
 */
public static void main(String[] args) {
    Configuration conf = new Configuration();
    conf.set("fs.defaultFS", "hdfs://localhost:9000");
   String localFilePath = "/home/hadoop/textfile"; // 本地路径
   String remoteFilePath = "/test/textfile"; // HDFS 路径
    try {
        CopyToLocal.copyToLocal(conf,                remoteFilePath,
localFilePath);
        System.out.println("下载完成");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

(3) 显示 HDFS 中指定的文件的读写权限、大小、创建时间、路径等信息；

```
[hadoop@cluster1 ~]$ hdfs dfs -ls -h testfile
-rw-r--r--   3 hadoop supergroup          6 2019-05-26 13:23 testfile
[hadoop@cluster1 ~]$
```

Java API 实现：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.fs.FileSystem;

import java.io.*;
import java.text.SimpleDateFormat;

public class List {
    /**
     * 显示指定文件的信息
     */
    public static void ls(Configuration conf, String remoteFilePath) {
        try (FileSystem fs = FileSystem.get(conf)) {
            Path remotePath = new Path(remoteFilePath);
            FileStatus[] fileStatuses = fs.listStatus(remotePath);
            for (FileStatus s : fileStatuses) {
                System.out.println("路径: " + s.getPath().toString());
                System.out.println("权限: " +
s.getPermission().toString());
                System.out.println("大小: " + s.getLen());
                /* 返回的是时间戳,转化为时间日期格式 */
                long timeStamp = s.getModificationTime();
                SimpleDateFormat format = new SimpleDateFormat(
                        "yyyy-MM-dd HH:mm:ss");
                String date = format.format(timeStamp);
                System.out.println("时间: " + date);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 主函数
     */
```

```
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://localhost:9000");
        String remoteFilePath = "/test/textfile"; // HDFS 路径

        try {
            System.out.println("读取文件信息: " + remoteFilePath);
            List.ls(conf, remoteFilePath);
            System.out.println("\n 读取完成");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

(4) 给定 HDFS 中某一个目录,输出该目录下的所有文件的读写权限、大小、创建时间、路径等信息，如果该文件是目录，则递归输出该目录下所有文件相关信息；

```
[hadoop@cluster1 ~]$ hdfs dfs -ls -R -h /
drwxr-xr-x   - hadoop supergroup          0 2019-05-26 13:27 /test
-rw-r--r--   3 hadoop supergroup          6 2019-05-26 13:27 /test/testfile
drwxr-xr-x   - hadoop supergroup          0 2019-05-26 13:23 /user
drwxr-xr-x   - hadoop supergroup          0 2019-05-26 13:23 /user/hadoop
-rw-r--r--   3 hadoop supergroup          6 2019-05-26 13:23 /user/hadoop/testfi
le
[hadoop@cluster1 ~]$
```

Java API 实现：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.fs.FileSystem;

import java.io.*;
import java.text.SimpleDateFormat;

public class ListDir {
    /**
     * 显示指定文件夹下所有文件的信息（递归）
     */
    public static void lsDir(Configuration conf, String remoteDir) {
        try (FileSystem fs = FileSystem.get(conf)) {
```

```
            Path dirPath = new Path(remoteDir);
            /* 递归获取目录下的所有文件 */
            RemoteIterator<LocatedFileStatus>     remoteIterator     =
fs.listFiles(
                    dirPath, true);
            /* 输出每个文件的信息 */
            while (remoteIterator.hasNext()) {
                FileStatus s = remoteIterator.next();
                System.out.println("路径: " + s.getPath().toString());
                System.out.println("     权     限     :     "     +
s.getPermission().toString());
                System.out.println("大小: " + s.getLen());
                /* 返回的是时间戳,转化为时间日期格式 */
                Long timeStamp = s.getModificationTime();
                SimpleDateFormat format = new SimpleDateFormat(
                        "yyyy-MM-dd HH:mm:ss");
                String date = format.format(timeStamp);
                System.out.println("时间: " + date);
                System.out.println();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 主函数
     */
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://localhost:9000");
        String remoteDir = "/"; // HDFS路径

        try {
            System.out.println("(递归)读取目录下所有文件的信息: " +
```

```
remoteDir);
            ListDir.lsDir(conf, remoteDir);
            System.out.println("读取完成");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

(5) 删除 HDFS 中指定的文件；

```
hdfs dfs -rm /test/testfile
```

(6) 在 HDFS 中，将文件从源路径移动到目的路径。

```
hdfs dfs -mv testfile /usr/hadoop/
```

JavaAPI 实现：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.fs.FileSystem;
import java.io.*;
public class MoveFile {
    /**
     * 移动文件
     */
    public static boolean mv(Configuration conf, String remoteFilePath,
            String remoteToFilePath) {
        try (FileSystem fs = FileSystem.get(conf)) {
            Path srcPath = new Path(remoteFilePath);
            Path dstPath = new Path(remoteToFilePath);
            return fs.rename(srcPath, dstPath);
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }
```

```
    /**
     * 主函数
     */
    public static void main(String[] args) {
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://localhost:9000");
        String remoteFilePath = "/test/testfile"; // 源文件HDFS路径
        String remoteToFilePath = "/usr/hadoop/"; // 目的HDFS路径

        try {
            if (MoveFile.mv(conf, remoteFilePath, remoteToFilePath)) {
                System.out.println("将文件 " + remoteFilePath + " 移动到
"
                        + remoteToFilePath);
            } else {
                System.out.println("操作失败(源文件不存在或移动失败)");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3.3.HBASE 数据存储

1) 通过 Hbase 的 shell 命令创建 HBase 列式存储数据表格，其中每一行的数据格式如下：

行键 (number)	列簇 1(information)			列簇 2(score)		
	列名 (name)	列名 (sex)	列名 (age)	列名 (123001)	列名 (123002)	列名 (123003)
学号	姓名	性别	年龄	成绩	成绩	成绩

首先进入 HBase

HBase shell

然后创建表：

create 'student','information','score'

查看表：

```
hbase(main):002:0> scan 'student'
ROW                                    COLUMN+CELL
 2015001                               column=information:age, timestamp=1558927736736, value=23
 2015001                               column=information:name, timestamp=1558927736732, value=zhangsan
 2015001                               column=information:sex, timestamp=1558927736741, value=male
 2015001                               column=score:123001, timestamp=1558927736749, value=86
 2015001                               column=score:123003, timestamp=1558927736753, value=69
 2015002                               column=information:age, timestamp=1558927736763, value=22
 2015002                               column=information:name, timestamp=1558927736759, value=Mary
 2015002                               column=information:sex, timestamp=1558927736768, value=female
 2015002                               column=score:123002, timestamp=1558927736773, value=77
 2015002                               column=score:123003, timestamp=1558927736777, value=99
 2015003                               column=information:age, timestamp=1558927736787, value=24
 2015003                               column=information:name, timestamp=1558927736782, value=Lisi
 2015003                               column=information:sex, timestamp=1558927736793, value=male
 2015003                               column=score:123001, timestamp=1558927736797, value=98
 2015003                               column=score:123002, timestamp=1558927736801, value=95
3 row(s) in 0.0510 seconds

hbase(main):003:0>
```

2) 请使用 HBASE 提供的 API 编程，实现向 1）建立的 HBase 表中插入如下数据，并完成以下指定功能：

学生表（Student）

学号（S_No）	姓名（S_Name）	性别（S_Sex）	年龄（S_Age）
2015001	Zhangsan	male	23
2015002	Mary	female	22
2015003	Lisi	male	24

选课表（SC）

学号（SC_Sno）	课程号（SC_Cno）	成绩（SC_Score）
2015001	123001	86
2015001	123003	69
2015002	123002	77
2015002	123003	99
2015003	123001	98
2015003	123002	95

(1) addRecord(String tableName, String row, String[] fields, String[] values);向表 tableName、行 row（用 S_Name 表示）和字符串数组 files 指定的单元格中添加对应的数据 values。其中 fields 中每个元素如果对应的列族下还有相应的列限定符的话，用“columnFamily:column”表示。例如，同时向“Math”、“Computer Science”、“English”三列添加成绩时，字符串数组 fields 为{“Score:Math”,”Score;Computer Science”,”Score:English”}，数组 values 存储这三门课的成绩。

(2) scanColumn(String tableName, String column);浏览表 tableName 某一列的数据，如果某一行记录中该列数据不存在，则返回 null。要求当参数 column 为某一列族名称时，如果底下有若干个列限定符，则要列出每个列限定符代表的列的数据；当参数 column 为某一列具体名称（例如“Score:Math”）时，只需要列出该列的数据。

(3) deleteRow(String tableName, String row);删除表 tableName 中 row 指定的行的记录。

Java API 程序：

```
/*
 * 创建一个students表,并进行相关操作
 */
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.KeyValue;
import org.apache.hadoop.hbase.client.Delete;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.util.Bytes;
public class HBaseJavaAPI {
    // 声明静态配置
    private static Configuration conf = null;
    static {
        conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum", "192.168.56.121");
        conf.set("hbase.zookeeper.property.clientPort", "2181");
    }
    //判断表是否存在
    private static boolean isExist(String tableName) throws IOException {
        HBaseAdmin hAdmin = new HBaseAdmin(conf);
```

```
    return hAdmin.tableExists(tableName);
}
// 创建数据库表
public static void createTable(String tableName, String[] columnFamilys)
        throws Exception {
    // 新建一个数据库管理员
    HBaseAdmin hAdmin = new HBaseAdmin(conf);
    if (hAdmin.tableExists(tableName)) {
        System.out.println("表 "+tableName+" 已存在！");
        //System.exit(0);
    } else {
        // 新建一个students表的描述
        HTableDescriptor tableDesc = new HTableDescriptor(tableName);
        // 在描述里添加列族
        for (String columnFamily : columnFamilys) {
            tableDesc.addFamily(new HColumnDescriptor(columnFamily));
        }
        // 根据配置好的描述建表
        hAdmin.createTable(tableDesc);
        System.out.println("创建表 "+tableName+" 成功!");
    }
}

// 删除数据库表
public static void deleteTable(String tableName) throws Exception {
    // 新建一个数据库管理员
    HBaseAdmin hAdmin = new HBaseAdmin(conf);
    if (hAdmin.tableExists(tableName)) {
        // 关闭一个表
        hAdmin.disableTable(tableName);
        hAdmin.deleteTable(tableName);
        System.out.println("删除表 "+tableName+" 成功！");
    } else {
        System.out.println("删除的表 "+tableName+" 不存在！");
        System.exit(0);
```

```
        }
    }

    // 添加一条数据
    public static void addRecord(String tableName, String row,
            String columnFamily, String column, String value) throws Exception
{
        HTable table = new HTable(conf, tableName);
        Put put = new Put(Bytes.toBytes(row));// 指定行
        // 参数分别:列族、列、值
        put.add(Bytes.toBytes(columnFamily), Bytes.toBytes(column),
                Bytes.toBytes(value));
        table.put(put);
    }

    // 删除一条(行)数据
    public static void deleteRow(String tableName, String row) throws Exception
{
        HTable table = new HTable(conf, tableName);
        Delete del = new Delete(Bytes.toBytes(row));
        table.delete(del);
    }

    // 删除多条数据
    public static void delMultiRows(String tableName, String[] rows)
            throws Exception {
        HTable table = new HTable(conf, tableName);
        List<Delete> delList = new ArrayList<Delete>();
        for (String row : rows) {
            Delete del = new Delete(Bytes.toBytes(row));
            delList.add(del);
        }
        table.delete(delList);
    }
```

```
    // 获取一条数据
    public static void scanColumn(String tableName, String row) throws
Exception {
        HTable table = new HTable(conf, tableName);
        Get get = new Get(Bytes.toBytes(row));
        Result result = table.get(get);
        // 输出结果,raw 方法返回所有 keyvalue 数组
        for (KeyValue rowKV : result.raw()) {
            System.out.print("行名:" + new String(rowKV.scanColumn()) + " ");
            System.out.print("时间戳:" + rowKV.getTimestamp() + " ");
            System.out.print("列族名:" + new String(rowKV.getFamily()) + " ");
            System.out.print("列名:" + new String(rowKV.getQualifier()) + "
");
            System.out.println("值:" + new String(rowKV.getValue()));
        }
    }

    // 获取所有数据
    public static void getAllRows(String tableName) throws Exception {
        HTable table = new HTable(conf, tableName);
        Scan scan = new Scan();
        ResultScanner results = table.getScanner(scan);
        // 输出结果
        for (Result result : results) {
            for (KeyValue rowKV : result.raw()) {
                System.out.print("行名:" + new String(rowKV.scanColumn()) + "
");
                System.out.print("时间戳:" + rowKV.getTimestamp() + " ");
                System.out.print("列族名:" + new String(rowKV.getFamily()) +
" ");
                System.out
                        .print("列名:" + new String(rowKV.getQualifier()) + "
");
                System.out.println("值:" + new String(rowKV.getValue()));
            }
```

```
        }
    }

    // 主函数
    public static void main(String[] args) {
        try {
            String tableName = "student";
            // 第一步：创建数据库表：“student”
            String[] columnFamilys = { "information", "score" };
            HBaseJavaAPI.createTable(tableName, columnFamilys);
            // 第二步：向数据表的添加数据
            // 添加第一行数据
            if (isExist(tableName)) {
                HBaseJavaAPI.addRecord(tableName, "2015000", "information",
"name", "dyk");
                HBaseJavaAPI.addRecord(tableName, "2015000", "information",
"age", "20");
                HBaseJavaAPI.addRecord(tableName, "2015000", "information",
"sex", "boy");
                HBaseJavaAPI.addRecord(tableName, "2015000", "score",
"123001", "97");
                HBaseJavaAPI.addRecord(tableName, "2015000", "score",
"123002", "128");
                HBaseJavaAPI.addRecord(tableName, "2015000", "score",
"123003", "85");
                // 添加第二行数据
                HBaseJavaAPI.addRecord(tableName, "2015001", "information",
"name", "zhangsan");
                HBaseJavaAPI.addRecord(tableName, "2015001", "information",
"age", "23");
                HBaseJavaAPI.addRecord(tableName, "2015001", "information",
"sex", "male");
                HBaseJavaAPI.addRecord(tableName, "2015001", "score",
"123001","86");
                HBaseJavaAPI.addRecord(tableName, "2015001", "score",
```

```
"123003","69");
        //HBaseJavaAPI.addRecord(tableName, "2015001", "score", "123002","90");
        // 添加第三行数据
        HBaseJavaAPI.addRecord(tableName, "2015002", "information", "name", "Mary");
        HBaseJavaAPI.addRecord(tableName, "2015002", "information", "age", "22");
        HBaseJavaAPI.addRecord(tableName, "2015002", "information", "sex","female");
        HBaseJavaAPI.addRecord(tableName, "2015002", "score", "123002","77");
        HBaseJavaAPI.addRecord(tableName, "2015002", "score", "123003","99");
        //HBaseJavaAPI.addRecord(tableName, "2015002", "score", "english","99");
        HBaseJavaAPI.addRecord(tableName, "2015003", "information", "name", "Lisi");
        HBaseJavaAPI.addRecord(tableName, "2015003", "information", "age", "24");
        HBaseJavaAPI.addRecord(tableName, "2015003", "information", "sex","male");
        HBaseJavaAPI.addRecord(tableName, "2015003", "score", "123001","98");
        HBaseJavaAPI.addRecord(tableName, "2015003", "score", "123002","95");
        // 第三步：获取一条数据
        System.out.println("**********获取一条(2015000)数据**********");
        HBaseJavaAPI.scanColumn(tableName, "2015000");
        // 第四步：获取所有数据
        System.out.println("********** 获取所有数据**********");
        HBaseJavaAPI.getAllRows(tableName);
```

```
                // 第五步：删除一条数据
                System.out.println("************删除一条(2015000)数据************");
                HBaseJavaAPI.deleteRow(tableName, "2015000");
                HBaseJavaAPI.getAllRows(tableName);
                // 第六步：删除多条数据
                //System.out.println("************** 删除多条数据****************");
                //String rows[] = new String[] { "qingqing","xiaoxue" };
                //HBaseJavaAPI.delMultiRows(tableName, rows);
                //HBaseJavaAPI.getAllRows(tableName);
                // 第七步：删除数据库
                //System.out.println("*************** 删除数据库表**************");
                //HBaseJavaAPI.deleteTable(tableName);
                //System.out.println(" 表 "+tableName+" 存在吗？"+isExist(tableName));
            } else {
                System.out.println(tableName + "此数据库表不存在！");
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

编译：javac -cp /usr/local/hbase-1.2.6/lib/*: HBaseJavaAPI.java

运行：java -cp /usr/local/hbase-1.2.6/lib/*: HBaseJavaAPI

运行结果：

```
hadoop@cluster1:~
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further de
tails.
表 student 已存在!
***************获取一条(2015000)数据**************
行名:2015000 时间戳:1558927736707 列族名:information 列名:age 值:20
行名:2015000 时间戳:1558927736685 列族名:information 列名:name 值:dyk
行名:2015000 时间戳:1558927736713 列族名:information 列名:sex 值:boy
行名:2015000 时间戳:1558927736718 列族名:score 列名:123001 值:97
行名:2015000 时间戳:1558927736722 列族名:score 列名:123002 值:128
行名:2015000 时间戳:1558927736727 列族名:score 列名:123003 值:85
***************获取所有数据***************
行名:2015000 时间戳:1558927736707 列族名:information 列名:age 值:20
行名:2015000 时间戳:1558927736685 列族名:information 列名:name 值:dyk
行名:2015000 时间戳:1558927736713 列族名:information 列名:sex 值:boy
行名:2015000 时间戳:1558927736718 列族名:score 列名:123001 值:97
行名:2015000 时间戳:1558927736722 列族名:score 列名:123002 值:128
行名:2015000 时间戳:1558927736727 列族名:score 列名:123003 值:85
行名:2015001 时间戳:1558927736736 列族名:information 列名:age 值:23
行名:2015001 时间戳:1558927736732 列族名:information 列名:name 值:zhangsan
行名:2015001 时间戳:1558927736741 列族名:information 列名:sex 值:male
行名:2015001 时间戳:1558927736749 列族名:score 列名:123001 值:86
行名:2015001 时间戳:1558927736753 列族名:score 列名:123003 值:69
行名:2015002 时间戳:1558927736763 列族名:information 列名:age 值:22
行名:2015002 时间戳:1558927736759 列族名:information 列名:name 值:Mary
行名:2015002 时间戳:1558927736768 列族名:information 列名:sex 值:female
行名:2015002 时间戳:1558927736773 列族名:score 列名:123002 值:77
行名:2015002 时间戳:1558927736777 列族名:score 列名:123003 值:99
行名:2015003 时间戳:1558927736787 列族名:information 列名:age 值:24
行名:2015003 时间戳:1558927736782 列族名:information 列名:name 值:Lisi
行名:2015003 时间戳:1558927736793 列族名:information 列名:sex 值:male
行名:2015003 时间戳:1558927736797 列族名:score 列名:123001 值:98
行名:2015003 时间戳:1558927736801 列族名:score 列名:123002 值:95
************删除一条(2015000)数据************
行名:2015001 时间戳:1558927736736 列族名:information 列名:age 值:23
行名:2015001 时间戳:1558927736732 列族名:information 列名:name 值:zhangsan
行名:2015001 时间戳:1558927736741 列族名:information 列名:sex 值:male
行名:2015001 时间戳:1558927736749 列族名:score 列名:123001 值:86
行名:2015001 时间戳:1558927736753 列族名:score 列名:123003 值:69
行名:2015002 时间戳:1558927736763 列族名:information 列名:age 值:22
行名:2015002 时间戳:1558927736759 列族名:information 列名:name 值:Mary
行名:2015002 时间戳:1558927736768 列族名:information 列名:sex 值:female
行名:2015002 时间戳:1558927736773 列族名:score 列名:123002 值:77
行名:2015002 时间戳:1558927736777 列族名:score 列名:123003 值:99
行名:2015003 时间戳:1558927736787 列族名:information 列名:age 值:24
行名:2015003 时间戳:1558927736782 列族名:information 列名:name 值:Lisi
行名:2015003 时间戳:1558927736793 列族名:information 列名:sex 值:male
行名:2015003 时间戳:1558927736797 列族名:score 列名:123001 值:98
行名:2015003 时间戳:1558927736801 列族名:score 列名:123002 值:95
[hadoop@cluster1 ~]$
```

3.4.HIVE 数据存储（可选）

3.4.1 使用 shell 命令完成下列任务

(1) 任意创建一张表，先加载本地数据到该表；再查询该表，将结果输出到文件系统。

创建表：

create table test_table(id int, name string)row format delimited fields terminated by '\t';

创建 table.txt 文件，

内容为：

```
hadoop@cluster2:~
1       dyk
2       cjl
3       hhx
4       jfy
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"table.txt" [readonly][noeol] 4L, 23C
```

使用 load 语句：

LOAD DATA LOCAL INPATH "/home/hadoop/table.txt" OVERWRITE INTO TABLE test_table;

查看表：

Select * from test_table;

```
hive> create table test_table(id int, name string)row format delimited fields te
rminated by '\t';
OK
Time taken: 0.105 seconds
hive> LOAD DATA LOCAL INPATH "/home/hadoop/table.txt" OVERWRITE INTO TABLE test_
table;
Loading data to table default.test_table
Table default.test_table stats: [numFiles=1, numRows=0, totalSize=23, rawDataSiz
e=0]
OK
Time tselect * from test_table;
OK
1       dyk
2       cjl
3       hhx
4       jfy
Time taken: 0.075 seconds, Fetched: 4 row(s)
hive> exit
```

(2) 分别创建一个内部表，外部表，查看数据的存储位置，体会它们的区别。

创建内部表：

```
create table t1(
    id      int
   ,name    string
   ,hobby   array<string>
   ,add     map<String,string>
)
row format delimited
fields terminated by ','
collection items terminated by '-'
map keys terminated by ':'
;
```

```
Logging initialized using configuration in jar:file:/usr/local/apache-hive-1.1.0
-bin/lib/hive-common-1.1.0.jar!/hive-log4j.properties
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/local/hadoop-2.6.5/share/hadoop/common/li
b/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/apache-hive-1.1.0-bin/lib/hive-jdbc
-1.1.0-standalone.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
hive> create table t1(
    >     id      int
    >    ,name    string
    >    ,hobby   array<string>
    >    ,add     map<String,string>
    > )
    > row format delimited
    > fields terminated by ','
    > collection items terminated by '-'
    > map keys terminated by ':'
    > ;
OK
Time taken: 0.616 seconds
hive>
```

创建外部表：

```
create external table t2(
    id      int
   ,name    string
   ,hobby   array<string>
   ,add     map<String,string>
)
row format delimited
fields terminated by ','
collection items terminated by '-'
map keys terminated by ':'
```

location '/user/t2'

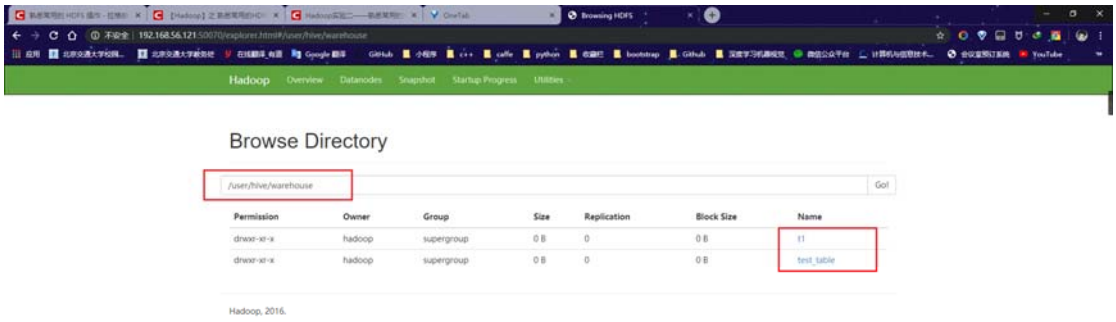
;

```
hive> create external table t2(
    >     id      int
    >    ,name    string
    >    ,hobby   array<string>
    >    ,add     map<String,string>
    > )
    > row format delimited
    > fields terminated by ','
    > collection items terminated by '-'
    > map keys terminated by ':'
    > location '/user/t2'
    > ;
OK
Time taken: 0.078 seconds
hive>
```

查看来两个表的信息：

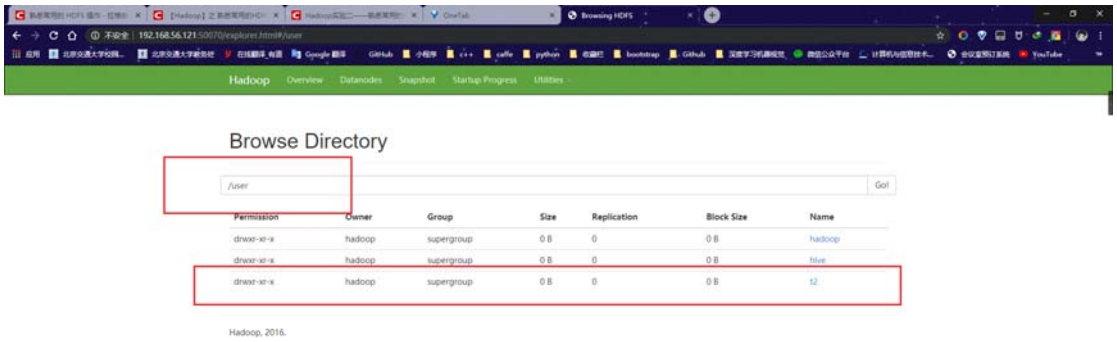
内部表：

```
OK
Time taken: 0.078 seconds
hive> desc formatted t1;
OK
# col_name              data_type               comment

id                      int
name                    string
hobby                   array<string>
add                     map<string,string>

# Detailed Table Information
Database:               default
Owner:                  hadoop
CreateTime:             Mon May 27 12:29:50 CST 2019
LastAccessTime:         UNKNOWN
Protect Mode:           None
Retention:              0
Location:               hdfs://cluster1:9000/user/hive/warehouse/t1
Table Type:             MANAGED_TABLE
Table Parameters:
        transient_lastDdlTime   1558931390

# Storage Information
```



外部表：

```
Time taken: 0.222 seconds, Fetched: 32 row(s)
hive> desc formatted t2;
OK
# col_name              data_type               comment

id                      int
name                    string
hobby                   array<string>
add                     map<string,string>

# Detailed Table Information
Database:               default
Owner:                  hadoop
CreateTime:             Mon May 27 12:30:34 CST 2019
LastAccessTime:         UNKNOWN
Protect Mode:           None
Retention:              0
Location:               hdfs://cluster1:9000/user/t2
Table Type:             EXTERNAL_TABLE
Table Parameters:
        EXTERNAL                TRUE
        transient_lastDdlTime   1558931434

# Storage Information
```



未被 external 修饰的是内部表（managed table），被 external 修饰的为外部表（external table）；内部表和外部表的区别主要是，内部表数据由 Hive 自身管理，外部表数据由 HDFS 管理；内部表数据存储的位置是 hive.metastore.warehouse.dir（默认：/user/hive/warehouse），外部表数据的存储位置由自己制定；删除内部表会直接删除元数据(metadata)及存储数据；删除外部表仅仅会删除元数据，HDFS 上的文件并不会被删除；对内部表的修改会将修改直接同步给元数据，而对外部表的表结构和分区进行修改，则需要修复（MSCK REPAIR TABLE table_name;）

(3) 列出 hive 指定的表的相关信息，例如表名，结构信息等；

desc formatted table_name;

```
hadoop@cluster2:~
hive> desc formatted t2;
OK
# col_name              data_type               comment

id                      int
name                    string
hobby                   array<string>
add                     map<string,string>

# Detailed Table Information
Database:               default
Owner:                  hadoop
CreateTime:             Mon May 27 12:30:34 CST 2019
LastAccessTime:         UNKNOWN
Protect Mode:           None
Retention:              0
Location:               hdfs://cluster1:9000/user/t2
Table Type:             EXTERNAL_TABLE
Table Parameters:
        EXTERNAL                TRUE
        transient_lastDdlTime   1558931434

# Storage Information
SerDe Library:          org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
InputFormat:            org.apache.hadoop.mapred.TextInputFormat
OutputFormat:           org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputForm
at
Compressed:             No
Num Buckets:            -1
Bucket Columns:         []
Sort Columns:           []
Storage Desc Params:
        colelction.delim        -
        field.delim             ,
        mapkey.delim            :
        serialization.format    ,
Time taken: 0.086 seconds, Fetched: 33 row(s)
hive>
```

(4) 清空指定的表的所有记录数据；

truncate table test_table;

```
hive> truncate table test_table;
OK
Time taken: 0.099 seconds
hive> select * from test_table;
OK
Time taken: 0.228 seconds
hive>
```

4 问题解决

4.1 Kafka JAVA API jar 包引用问题

使用 java API 需要导入 kafka 的 jar 包，在开始实验的时候，对于 jdk 不容易管理 jar 包，这个过程中使用 maven 进行了实验，但是在 CentOS 上，没有用户界面，对于维护一个项目很不容易，其中也遇到很多问题，版本覆盖等

等，这让第一步编译就进行不下去，后面还是采用了 jdk 引用外部包的方法：

javac -cp /usr/local/kafka_2.10-0.8.2.1/libs/*: MyConsumer.java

这个命令可以将 kafka/libs/目录下的所有 jar 包全部引用。

同理如果使用某个其他文件目录的 jar 包，修改对应的目录即可。

4.2 启动 HDFS datanode 启动失败问题

启动 HDFS 的时候遇到 datanode 不能启动的问题：

主要的原因就是多次格式化 HDFS：

接下来开始格式化：

// 启动journalnode（在所有datanode上执行，也就是cluster1, cluster2, cluster3）

$ hadoop-daemon.sh start journalnode

启动后使用jps命令可以看到JournalNode进程

// 格式化HDFS（在cluster1上执行）

$ hdfs namenode -format

// 格式化完毕后可关闭journalnode（在所有datanode上执行）

$ hadoop-daemon.sh stop journalnode

格式化只需要在第一次启动的时候进行一次即可。如果需要格式化，需要将

<property>

<name>dfs.datanode.data.dir</name>

<value>/home/hadoop_files/hadoop_data/hadoop/datanode</value>

</property>

该目录下的 current 文件夹删除掉，所有的节点上都要删除