



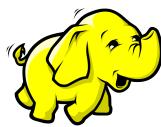
# 北京化工大学

Beijing University of Chemical Technology

## 大数据平台搭建实验手册

姓    名:	罗登
学    院:	信息科学与技术
专    业:	计算机科学与技术
班    级:	计科 1605
学    号:	2016011186

## 目录

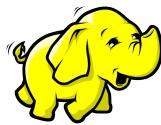


## 0.1. 前言

本实验报告主要基于章鱼大数据实训平台 (<https://train.ipieuvre.com/>) 完成，共有三台虚拟机，中间出现了一次 Slave1 宕机丢失的情况。后来重新申请找回。

在其中搭建了以 Hadoop 平台为主的软件工具，包括 Spark、Hive、HBase、Zookeeper、Kafka、Flume、Sqoop、Storm 等。在初期，老师只要求完成 Hadoop、Spark、Hive、HBase 与 Zookeeper 的搭建和测试，并要求结合课程视频和实验任务完成。由于之前有过一些 Linux 操作和 Hadoop 分布式搭建的经验，在操作的时候还是比较轻车熟路，后面的安装也是在完成安装测试后尽量添加编程实践部分，相当于是提高部分。在搭建 Hadoop HA（高可用）时，耗费了较多的时间。查找了很多的配置，最后才完成。而在搭建好平台性质的 Hadoop 或 HDFS 后，其他的软件安装起来都比较的方便。由于章鱼的视频课程中，涉及到了队列工具 Kafka、数据转换工具 Sqoop、日志聚集工具 Flume 的内容，因此也在上面完成了这几项的搭建和测试。加上身边有同学在学习 Storm 相关的课程内容，在一块讨论交流后，也产生了一些兴趣，因此也添加到了集群中来。并一起完成了一些小的编程案例。

结合章鱼大数据的视频教程、训练项目、网上收集到的一些资料，对每个软件框架完成了基本的搭建测试（运行 Shell 命令和自带例子）和简单的编程实践（WordCount、表操作等）。内容以广度为主，不是很深入。主要作为个人学习备忘和记录。



## 0.2. 安装基础依赖

在开始安装和搭建 Hadoop 开发环境之前，需要安装一些基础的开发环境和软件。包括：

1. Java 运行环境和 JDK。因为 Hadoop 等相关软件包都是运行在 Java 环境之上的，所以 JDK 必不可少。这里安装 JDK1.8。
2. SSH 服务和客户端。在 Hadoop 运行启动时，需要 Master 主节点访问到 Slave 从节点，发起远程过程调用。因此需要安装 SSH 服务程序和配置免密登录。
3. MySQL 服务。Hive 需要用到 MySQL 存储元数据信息。用`apt install mysql-server`即可。
4. 相关开发软件工具，如文本编辑器 VS Code、Vim 插件、Eclipse、IDEA 等常用的开发工具，方便编辑`xml`文件，能够提高效率。依个人喜好安装。

### 第一节 环境简介

为确保兼容性，本实验安装的各个软件包版本如下表??：完整的`.bashrc`配置：

表 1: 主机与各节点 IP 地址

主机名	IP 地址
Master	172.100.2.220
Slave1	172.100.5.239
Slave2	172.100.3.99

表 2: 软件包版本及依赖

软件名	版本
java	1.8
maven	3.6.1
hadoop	3.1.2
zookeeper	3.5.5
hive	3.3.1
hbase	2.2.0
scala	2.1.3 SDKMAN install
spark	2.4.3 with hadoop2.6+
flume	1.9.0
sqoop	1.4.7
kafka	2.12-2.30
storm	2.0

## .bashrc 配置

```
# java
export JAVA_HOME=/opt/java
export PATH=$JAVA_HOME/bin:$PATH

# hadoop
export HADOOP_HOME=/opt/hadoop
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export PATH=$HADOOP_HOME/bin:$PATH
export PATH=$HADOOP_HOME/sbin:$PATH

# zookeeper
export ZOOKEEPER_HOME=/home/zhangyu/zookeeper
export PATH=$ZOOKEEPER_HOME/bin:$PATH
export ZOOKEEPER_CONF_DIR=$ZOOKEEPER_HOME/conf

# hive config
export HIVE_HOME=/home/zhangyu/hive
export HIVE_CONF_DIR=$HIVE_HOME/conf
export PATH=$HIVE_HOME/bin:$PATH
CLASSPATH==$CLASSPATH:$HIVE_HOME/bin

# hbase config
export HBASE_HOME=/home/zhangyu/hbase
export HBASE_CONF_DIR=$HBASE_HOME/conf
export PATH=$HBASE_HOME/bin:$PATH

# maven config
export PATH=/home/zhangyu/maven/bin:$PATH

# add the spark bin
export SPARK_HOME=/home/zhangyu/spark
export SPARK_CONF_DIR=$SPARK_HOME/conf
export PATH=$SPARK_HOME/bin:$PATH

# flume
```

```
export FLUME_HOME=/home/zhangyu/flume
export FLUME_CONF_DIR=/home/zhangyu/flume/conf
export PATH=$FLUME_HOME/bin:$PATH

# sqoop
export SQOOP_HOME=/home/zhangyu/sqoop
export SQOOP_CONF_DIR=/home/zhangyu/sqoop/conf
export PATH=$SQOOP_HOME/bin:$PATH

# kafka
export KAFKA_HOME=/home/zhangyu/kafka
export KAFKA_CONF_DIR=/home/zhangyu/kafka/config
export PATH=$KAFKA_HOME/bin:$PATH

#THIS MUST BE AT THE END OF THE FILE FOR SDKMAN TO WORK!!!
export SDKMAN_DIR="/home/zhangyu/.sdkman"
[[ -s "/home/zhangyu/.sdkman/bin/sdkman-init.sh" ]] && source "/home/zhangyu/.sdkman/bin/sdkman-init.sh"
```

## 第二节 安装 Java 与 JDK

安装 JDK 需要去 Oracle 官网下载 JDK1.8 版本，这里要求 1.8，因为 Hadoop 平台等很多软件包要求在 1.8 以上运行，但是太新又不行。

而且由于其 License 的变化，需要在 Oracle 注册账号才能下载安装。

注册并下载后，完成简单的环境变量配置即可。

### 安装 JDK 过程

```
$ tar zxvf jdk1.8.0xxx.tar.gz
$ sudo mv jdk1.8.0xxx /opt/
$ cd /opt/
$ sudo mv jdk1.8.0xxx java
$ sudo chown zhangyu:zhangyu java
$ vi ~/.bashrc
# Append these lines
# java
export JAVA_HOME=/opt/java
export PATH=$JAVA_HOME/bin:$PATH
```



图 1: Oracle 官网注册安装 JDK

```
$ source ~/.bashrc
$ java -version
# 之后用 scp 将 Java 分发到各个节点上去
# 完成同样的配置
```

### 第三节 配置网络和 SSH 免密登录

修改/etc/hosts文件，添加主机名与 IP 地址映射关系。

hosts 文件

```
127.0.0.1      localhost
172.100.2.220    master
172.100.5.239    slave1
172.100.3.99     slave2
```

这样可以避免在其他配置的时候写硬的 IP 地址，不方便修改。这样人类可读性更好。再将此文件分发到其他的几个节点上去，保持端口映射相同。为了避免每天机器都会关闭重新启动时，/etc/hosts文件被恢复，在/home/zhangyu 目录下保存一份副本，每次将其覆盖即可。

在家目录下生成密钥：

```
ssh keygen
```

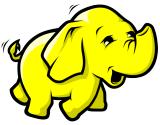
出现询问回车即可，就可以看到生成出来的公钥。

使用下列命令将秘钥复制到各个节点上。这样即可配置好免密登录。

### hosts 文件

```
ssh-copy-id master
ssh-copy-id slave1
ssh-copy-id slave2
# 测试是否正常将公钥分发完成免密登录
ssh master
ssh slave1
ssh slave2
```

中间出现询问和要求输入密码，输入`yes`和密码`zhangyu`即可。后面测试过程中，如果正常登录且不需要输入密码，则说明配置成功了。否则需要重新配置



## 0.3. 搭建分布式 Hadoop

### 第一节 简介

Hadoop 包含了 3 个核心组件，分别是

- HDFS 分布式文件系统
- Mapreduce 并行计算框架
- YARN 集群资源调度管理工具

这三项是整个生态系统的核心组件，如果没有配置好，后面的操作都无法完成，同时也是最难配置的一块，因此也是我花了最多时间的一部分。

首先从分布式配置开始，尽管仅有`SecondaryNameNode`的分布式配置可用性、可靠性都不高，但是是所有配置的基础。

### 第二节 配置文件

在配置好 JDK1.8 和 SSH 免密码登录后，配置分布式 Hadoop 就比较简单了，只需要修改 5 个配置文件：

`core-site.xml`, `hdfs-site.xml`, `yarn-site.xml`, `mapred-site.xml`, 和`hadoop-env.sh`。

#### core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="configuration.xsl"?>
```

```
<configuration>
<property>
    <name>hadoop.tmp.dir</name>
    <value>/data/tmp/hadoop/tmp</value>
</property>
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://master:9000/</value>
    <description>NameNode URI</description>
</property>
</configuration>
```

在简单分布式模式下，`core-site.xml`只要设置 Hadoop 数据保存的目录和对外暴露的名称节点端口即可。这里是 Master。

hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="configuration.xsl"?>
<configuration>
<property>
    <name>dfs.namenode.name.dir</name>
    <value>/data/tmp/hadoop/hdfs/name</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>/data/tmp/hadoop/hdfs/data</value>
</property>
<property>
    <name>dfs.replication</name>
    <value>1</value>
</property>
<property>
    <name>dfs.permissions</name>
    <value>false</value>
</property>
</configuration>
```

`hdfs-site.xml`设置的是名称节点的元数据和数据节点的数据的本地保存位置。以及数据的副本数，这里设置为 1，减小机器负担，默认为 3。还有其他用户访问的权限检查，这里设置为`false`，减少后面的麻烦。

yarn-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="configuration.xsl"?>
<configuration>
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>
</configuration>
```

`yarn-site.xml`这里设置的`yarn.nodemanager.aux-services`是指定 YARN 获取数据的方式，从 MapReduce 中获取。

mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="configuration.xsl"?>
<configuration>
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
</configuration>
```

其中指定调度 MapReduce 作业的框架为 YARN。在`hadoop-env.sh`中写入：

hadoop-env.sh

```
export JAVA_HOME=/opt/java/
export HADOOP_HOME=/opt/hadoop
export HADOOP_CONF_DIR=${HADOOP_HOME}/etc/hadoop
```

用于找到`JAVA_HOME`, `HADOOP_HOME`, 以及 Hadoop 配置文件的位置等信息。

### 第三节 启动查看

首先用 Jps 查看系统中的 Java 进程。分别到不同节点上去`jps`。

## Master 节点

```
1020 DataNode  
1522 NodeManager  
1419 ResourceManager  
1892 Jps  
1224 SecondaryNameNode  
860 NameNode
```

## Slave1 节点

```
1991 Jps  
1020 DataNode  
1522 NodeManager
```

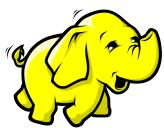
## Slave2 节点

```
4850 Jps  
3050 DataNode  
2876 NodeManager
```

其中，主节点上有 5 个进程，从节点上有 3 个进程，编号不是重要的。`DataNode`和`NameNode` 以及 Master 上的`SecondaryNameNode`，是 HDFS 启动时需要的进程；`NodeManager`和`ResourceManager`是 YARN 启动后需要看到的进程。

也可以在浏览器中查看 Web 端口的运行情况，这个在后面 HA 部分??详述。

如果没有正常启动，可以查看 Hadoop 目录下面的`logs\`文件夹下面查找日志情况。日志一般比较长，可以查到最后查看报错信息即可。该配置比较简单，多半是通信或者配置语法错误导致。



## 0.4. 安装 Zookeeper

### 第一节 简介

Zookeeper（动物管理员）相当于集群的管家，完成了集群中领导选举的工作，确保了集群中只有一个活跃的 Master，避免了“脑裂”现象。在 Hadoop HA 和 HBase、Kafka 等中均需要用到，一般来说都会在启动 HDFS 之前启动它。配置起来也不是很复杂。

### 第二节 配置文件

配置 Zookeeper 主要修改的文件就是`conf\zoo.cfg`这个文件。

```
zoo.cfg
```

```
tickTime=2000
initLimit=5
syncLimit=2
# 上面3项保持默认即可
# zookeeper临时数据的保持目录，重要
dataDir=/home/zhangyu/zookeeperdata
# 连接端口，保持默认即可
clientPort=2181
#maxClientCnxns=60
# 这里需要配置3台服务器的地址
server.1=master:2888:8888
server.2=slave1:2888:8888
server.3=slave2:2888:8888
```

修改这个之后呢，还需要创建其中配置的`dataDir=/home/zhangyu/zookeeperdata`文件夹，并在其中创建一个文件`myid`，写入其 id，要与`server.x`中对应。如 Master 是 server1，则在 myid 文件中写 1；Slave1 是 server2，则在 myid 文件中写 2。

完成配置之后分别在各台机器上使用`bin/zkServer.sh`启动服务即可。

```
$ zkServer.sh start
```

可以使用

```
$ zkServer.sh status
```

查看本机器状态，Leader 或 Follower。同时使用 Jps 查看进程可以发现多了`QuorumPeerMain`进程。

### 第三节 zkCli 命令行应用实践

Zookeeper 作为集群的管理者，在很多情况都会被用到，不光是 Hadoop 的 HA 机制，包括后面可以看到 Storm 的 HA 机制、Kafka 等的运行都要求前提是启动 ZK 集群。在作为后台服务的协调集群的同时，Zookeeper 还提供了 zkCli.sh 来方便查看集群中的各个应用的健康信息。

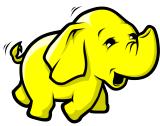
```
zhangyu@fb56ea429073:~/kafka$ zkCli.sh
Connecting to localhost:2181
2019-07-17 01:05:22,811 [myid:] - INFO  [main:Environment@109] - Client environment:zookeeper.version=3.5.5-390fe37ea45dee01bf87dc1c042b5e3dcce88653, built on 05/03/2019 12:07 GMT
2019-07-17 01:05:22,827 [myid:] - INFO  [main:Environment@109] - Client environment:host.name=slave2
2019-07-17 01:05:22,827 [myid:] - INFO  [main:Environment@109] - Client environment:java.version=1.8
.0_212
```

启动 ZKCli 进入命令行模式。

```
[admin, brokers, cluster, config, consumers, controller, controller_epoch, hadoop-ha, hbase,[0/1831]
nge_notification, latest_producer_id_block, log_dir_event_notification, rmstore, storm, yarn-leader-
election, zookeeper]
[zk: localhost:2181(CONNECTED) 1] [2019-07-17 01:06:50,390] INFO [GroupMetadataManager brokerId=2] R
emoved 0 expired offsets in 0 milliseconds. (kafka.coordinator.group.GroupMetadataManager)

[zk: localhost:2181(CONNECTED) 1] ls /admin
[delete_topics]
[zk: localhost:2181(CONNECTED) 2] ls /brokers
[ids, seqid, topics]
[zk: localhost:2181(CONNECTED) 3] ls /brokers/topics
[__consumer_offsets, my_test, mytest, t_test, test]
[zk: localhost:2181(CONNECTED) 4] ls /hadoop_ha
Node does not exist: /hadoop_ha
[zk: localhost:2181(CONNECTED) 5] ls /hadoop-ha
[lldcluster]
[zk: localhost:2181(CONNECTED) 6]
```

用类似 Linux 命令 `ls` 查看运行的各个应用的信息。



## 0.5. 搭建 Hadoop HA

### 第一节 简介

上面配置的 Hadoop 尽管有了 `SecondaryNameNode` 等机制来避免名称节点单点故障，但是依然很不健壮，单点故障依然是存在的。Hadoop2 中引入了 HA (High Available) 高可用机制，来确保 Hadoop 集群能够运行 24 小时不宕机。需要搭配 Zookeeper 实现高可用。

### 第二节 配置文件

HA 的配置相对比较复杂，主要需要修改的文件与配置分布式 Hadoop 时相同，可以参考第??部分，重复部分不再赘述。

core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="configuration.xsl"?>
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://lldcluster</value>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/data/tmp/hadoop/tmp</value>
```

```

</property>
<property>
    <name>ha.zookeeper.quorum</name>
    <value>master:2181,slave1:2181,slave2:2181,slave3:2181</value>
</property>
<property>
    <name>hadoop.proxyuser.zhangyu.hosts</name>
    <value>*</value>
</property>
<property>
    <name>hadoop.proxyuser.zhangyu.groups</name>
    <value>*</value>
</property>
</configuration>

```

- **fs.defaultFS**, HDFS 对外暴露的服务端口。这里设置成了一个集群端口，而不是一个名称节点的端口。因为在 HA 中，一个集群中可能会有多个名称节点，但 Active 的只有一个，具体是哪个取决于 Zookeeper。这里没有指定端口，使用的是默认的 8020（新版本 Hadoop 的端口）。
- **ha.zookeeper.quorum**指定了 ZK 集群有哪些机器。2181 是 ZK Client 的默认端口，也可以在**zoo.cfg**中配置。
- **hadoop.proxyuser.zhangyu.hosts**该配置和下面的配置是为了解决 Hive 运行时权限验证的问题，这里设置为全部主机和用户组，即免去验证。

hdfs-site.xml

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="configuration.xsl"?>
<configuration>
<property>
    <name>dfs.nameservices</name>
    <value>ldcluster</value>
</property>
<property>
    <name>dfs.ha.namenodes.ldcluster</name>

```

```
<value>master,slave1</value>
</property>
<property>
    <name>dfs.namenode.name.dir</name>
    <value>/data/name</value>
</property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>/data/data</value>
</property>
<property>
    <name>dfs.namenode.rpc-address.ldcluster.master</name>
    <value>master:8020</value>
</property>
<property>
    <name>dfs.namenode.rpc-address.ldcluster.slave1</name>
    <value>slave1:8020</value>
</property>
# master和slave1的http地址
<property>
    <name>dfs.namenode.http-address.ldcluster.master</name>
    <value>master:9870</value>
</property>
<property>
    <name>dfs.namenode.http-address.ldcluster.slave1</name>
    <value>slave1:9870</value>
</property>
# 3台JournalNode地址，后台跟名字，  

# 但后面的名字不能与nameService相同
<property>
    <name>dfs.namenode.shared.edits.dir</name>
    <value>qjournal://master:8485;slave2:8485;  

    slave1:8485/ldcluster5200</value>
</property>
# 配置客户端调用接口
<property>
```

```
<name>dfs.client.failover.proxy.provider.</name>
<value>org.apache.hadoop.hdfs.server.
namenode.ha.ConfiguredFailoverProxyProvider</value>
</property>
<property>
  <name>dfs.ha.fencing.methods</name>
  <value>sshfence</value>
</property>
<property>
  <name>dfs.ha.fencing.ssh.private-key-files</name>
  <value>/home/zhangyu/.ssh/id_dsa</value>
</property>
<property>
  <name>dfs.ha.fencing.methods</name>
  <value>sshfence</value>
</property>
<property>
  <name>dfs.ha.fencing.ssh.connect-timeout</name>
  <value>30000</value>
</property>
#配置journalnode目录
<property>
  <name>dfs.journalnode.edits.dir</name>
  <value>/opt/hadoop/journaldata</value>
</property>
<property>
  <name>dfs.ha.automatic-failover.enabled</name>
  <value>true</value>
</property>
</configuration>
```

- **dfs.nameservices**, HDFS 的名称服务, 非常重要, 而且要与**core-site.xml**中的设置相对应。
- **dfs.ha.namenodes.ldcluster**, 集群中的名称节点数。在当前名称节点宕机后, 这些节点都有可能转换为 Active 状态。
- **dfs.namenode.name.dir**, 保存名称节点元数据的目录。这里设置了很上面分布式不同的目

录，方便切换。

- `dfs.datanode.data.dir`, 同上。
- `dfs.namenode.rpc-address.ldcluster.master`, 名称节点 RPC 通信的地址。设置默认的 8020。注意要与`nameservices`与`namenodes`对应。
- `dfs.namenode.rpc-address.ldcluster.slave1`, 同上。
- `dfs.namenode.http-address.ldcluster.master`, 是查看 HTTP 服务也就是后面的 Web 界面的地址。原来是 50070, 新版本是 9870。
- `dfs.namenode.http-address.ldcluster.slave1`, 同上。
- `dfs.namenode.shared.edits.dir`, 这里设置 JournalNode 的地址。HA 是通过共享日志信息来查看是否有机器发生故障的。`qjournal`是协议的名称，所有的机器都要加入进来，端口不用修改用默认。
- `dfs.client.failover.proxy.provider.ldcluster`, 提供故障恢复的 Java 类，需要指定为: org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider, 不需要修改。
- `dfs.ha.fencing.methods`, 故障后需要将原来的故障 NameNode 阻拦，采用`sshfence`方式阻拦。
- `dfs.ha.fencing.ssh.private-key-files`, SSH 私钥保存位置。
- `dfs.ha.fencing.ssh.connect-timeout`, SSH 连接 timeout 时长，设置为 30000。
- `dfs.journalnode.edits.dir`, 保存 JournalNode 数据的目录，需要设置，且需要提前创建。
- `dfs.ha.automatic-failover.enabled`, 设置开启自动故障恢复。

yarn-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="configuration.xsl"?>
<configuration>
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>
<!-- open the log aggregation function -->
```

```
<property>
    <name>yarn.log-aggregation-enable</name>
    <value>true</value>
</property>
<!-- set the log aggregation retain time 7 days -->
<property>
    <name>yarn.log-aggregation.retain-seconds</name>
    <value>604800</value>
</property>
<property>
    <name>yarn.resourcemanager.ha.enabled</name>
    <value>true</value>
</property>
<property>
    <name>yarn.resourcemanager.cluster-id</name>
    <value>yarn-ha</value>
</property>
<property>
    <name>yarn.resourcemanager.ha.rm-ids</name>
    <value>rm1,rm2</value>
</property>
<property>
    <name>yarn.resourcemanager.hostname.rm1</name>
    <value>master</value>
</property>
<property>
    <name>yarn.resourcemanager.hostname.rm2</name>
    <value>slave1</value>
</property>
<property>
    <name>yarn.resourcemanager.webapp.address.rm1</name>
    <value>master:8088</value>
</property>
<property>
    <name>yarn.resourcemanager.webapp.address.rm2</name>
    <value>slave1:8088</value>
```

```
</property>
<property>
    <name>hadoop.zk.address</name>
    <value>master:2181,slave1:2181,slave2:2181</value>
</property>
<property>
    <name>yarn.application.classpath</name>
    <value>/opt/hadoop/etc/hadoop
        opt/hadoop/share/hadoop/common/lib/*:
        /opt/hadoop/share/hadoop/common/*:
        /opt/hadoop/share/hadoop/hdfs:
        /opt/hadoop/share/hadoop/hdfs/lib/*:
        /opt/hadoop/share/hadoop/hdfs/*:
        /opt/hadoop/share/hadoop/mapreduce/lib/*:
        /opt/hadoop/share/hadoop/mapreduce/*:
        /opt/hadoop/share/hadoop/yarn
        opt/hadoop/share/hadoop/yarn/lib/*:
        /opt/hadoop/share/hadoop/yarn/*</value>
</property>
</configuration>
```

- `yarn.resourcemanager.cluster-id`, 同`hdfs-site.xml`中的`nameservices`类似, 为集群设置一个 id, 可自定义。
- `yarn.resourcemanager.ha.rm-ids`, 资源管理器的 id。这里设置为`rm1,rm2`, 需要在后面明确指定主机地址。
- `yarn.resourcemanager.hostname.rm1`, 指定`rm1`为 master 机器。
- `yarn.resourcemanager.hostname.rm2`, 同上指定`rm2`。
- `yarn.resourcemanager.webapp.address.rm1`, 通信地址。
- `yarn.resourcemanager.webapp.address.rm2`, 同上。
- `hadoop.zk.address`, 指定 ZK 集群地址。
- `yarn.application.classpath`, 指定 CLASSPATH。在执行 MapReduce 测试程序和 Hive 操作时, 出现找不到主类的错误。可以在这里指定。
- `yarn.log-aggregation-enable`, 为开启日志聚合功能, 不涉及 HA。

- `yarn.log-aggregation.retain-seconds`, 日志保留的天数, 这里是 7 天。

### mapred-site.xml

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="configuration.xsl"?>
<configuration>
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
<property>
    <name>mapreduce.jobhistory.address</name>
    <value>master:10020</value>
</property>
<property>
    <name>mapreduce.jobhistory.webapp.address</name>
    <value>master:19888</value>
</property>
<property>
    <name>mapreduce.application.classpath</name>
    <value>$HADOOP_HOME/share/hadoop/mapreduce/*,
        $HADOOP_HOME/share/hadoop/mapreduce/lib/*</value>
</property>
</configuration>

```

- `mapreduce.application.classpath`, 设置 MapReduce 的 CLASSPATH, 还是为了解决前面的报错。
- `mapreduce.jobhistory.address`, JobHistory 的地址。
- `mapreduce.jobhistory.webapp.address`, 查看 JobHistory 的 Web 地址。

配置好之后, 启动进程的顺序也很重要。

### 启动步骤

```

# 格式化 ZK 集群
hdfs zkfc -formatZK
# 格式化名称节点

```

```
hdfs namenode --format
# 将Master上格式化后名称节点目录下的内容
# 复制到Slave1对应配置的目录下面
# 不这样执行Slave1上的NameNode将无法启动
scp -r /data/name/* slave1:/data/name
# 启动HDFS，可以看到NameNode, DataNode,
# JournalNode, ZKFailoverController等节点的启动
start-dfs.sh
# 启动YARN
start-yarn.sh
# 可以看到ResourceManager和NodeManager在Master和Slave1上启动
jps
# 输出所有Java进程
```

### 第三节 HDFS Shell 操作

关于 HDFS 的 Shell 命令行操作，比较简单，基本与常规的 Linux Shell 操作类似，只是在前面加上`hadoop fs`或`hdfs dfs`，个人常用后者。

如 Linux `ls` 对应于`hadoop fs -ls`或`hdfs dfs -ls`。类似还有`hadoop fs -mkdir`或`hdfs dfs -mkdir`。

上传文件为`hdfs dfs -put`+ 目标的文件目录。下载文件为`hdfs dfs -get`+ 目标文件。

### 第四节 Hadoop Web UI

可以通过 `http://master:9870` 访问 HDFS 和查看整个集群信息概览。这里的 master 指的是名称节点的主机名。如果 master 挂了，可以使用其他的节点，如 slave1 访问。这里也体现了 HA 的强大。

Namenode information - Mozilla Firefox

百度一下，你就知道

Namenode information

All Applications

master:9870/dfshealth.html#tab-overview

Hadoop Overview Datanodes Datanode Volume Failures Snapshot Startup Progress Utilities

## Overview 'master:8020' (active)

Namespace:	Idcluster
Namenode ID:	master
Started:	Fri Jul 12 02:11:16 +0000 2019
Version:	3.1.2, r1019dde65bcf12e05ef48ac71e84550d589e5d9a
Compiled:	Tue Jan 29 01:39:00 +0000 2019 by sunilg from branch-3.1.2
Cluster ID:	CID-f12592bf-e585-4fa2-9f3d-7473407a0bbb
Block Pool ID:	BP-2021266335-172.100.2.220-1562478391785

## Summary

Security is off.

Safemode is off.

所有应用程序 Namenode information ... Terminal 终端 - zhangyu...

262 files and directories, 123 blocks (123 replicated blocks, 0 erasure coded block groups) = 385 total filesystem object(s).

Heap Memory used 85.88 MB of 195.5 MB Heap Memory. Max Heap Memory is 910.5 MB.

Non Heap Memory used 72.86 MB of 74.19 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	5 TB
Configured Remote Capacity:	0 B
DFS Used:	19.9 MB (0%)
Non DFS Used:	2.28 TB
DFS Remaining:	2.47 TB (49.35%)
Block Pool Used:	19.9 MB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	3 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	12
Number of Blocks Pending Deletion	0

在这里可以看到整个集群的概览信息，包括名称节点是否是活跃的，数据节点的活跃情况，日志节点信息

The screenshot shows a Mozilla Firefox browser window titled "Browsing HDFS - Mozilla Firefox". The address bar displays "master:9870/explorer.html#/sogou-data/500w/500w". The main content area is titled "Browse Directory" and shows a single file entry: "sogou.500w.utf8". The file has the following details:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	zhangyu	supergroup	547.09 MB	Jul 12 08:18	3	128 MB	sogou.500w.utf8

Below the table, it says "Showing 1 to 1 of 1 entries". On the right, there is a dropdown menu under "Utilities" with options: "Browse the file system", "Logs", "Metrics", "Configuration", and "Process Thread Dump". At the bottom left, it says "Hadoop, 2018." and at the bottom right, there are "Previous", "1", and "Next" buttons.

浏览 HDFS 文件系统是非常有用的功能，还可以查看日志、配置等信息

第二是通过 `http://master:8088` 查看 MapReduce 作业的运行信息。这里包括运行的 Hadoop MapReduce Example，用户自己编写提交到集群运行的 Jar 包，Hive 查询等。可以看到作业的运行情况，使用的资源情况，运行还有多长时间结束等。

```

zhangyu@bd2a17a1c41a:~/datas$ hdfs dfs -cat /data/wordcount/out/*
Scheduler Metrics
Scheduler Type: Capacity Scheduler
Scheduling Resource Type: [memory-mb (unit=Mi), vcores]
<memory:1024, vcores:1>
Capacity Scheduler
[memory-mb (unit=Mi), vcores]
<memory:1024, vcores:1>
Show 20 entries
ID User Name Application Queue Application Priority StartTime FinishTime State FinalStatus
application_1562897494343_0002 zhangyu word count MAPREDUCE default 0 Fri Jul 12 02:25:26 +0000 2019 N/A RUNNING UNDEF
Showing 1 to 1 of 1 entries
zhangyu@bd2a17a1c41a:~/datas$ 

```

提交运行 MapReduce 的 WordCount 例子的结果

The screenshot shows the Hadoop Web UI interface. On the left, there is a sidebar with navigation links: Cluster (About, Nodes, Node Labels, Applications, Scheduler, Tools), and Tools (NEW, NEW\_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED). The main content area is titled "All Applications". It includes sections for "Cluster Metrics", "Cluster Nodes Metrics", and "Scheduler Metrics". The "Scheduler Metrics" section shows the same data as the terminal output above, including the application ID, user, name, type, queue, priority, start time, finish time, state, and final status. The application listed is "application\_1562897494343\_0002" run by "zhangyu" for "word count" under "MAPREDUCE" in the "default" queue with priority 0, starting at "Fri Jul 12 02:25:26 +0000 2019" and currently in "RUNNING" state.

The screenshot shows the Apache Hadoop Web UI interface. At the top, there are tabs for 'hive web port\_百度搜索', 'Hive- Hive Web Interface', '编译Hadoop-eclipse-plus', 'Browsing HDFS', 'All Applications', and a '+' button. Below the tabs, the URL is 'slave1:8088/cluster'. The main content area displays cluster statistics:

Processors Running	Memory Used	Memory Total	Memory Reserved	Vcores Used	Vcores Total	Vcores Reserved
	3 GB	24 GB	0 B	2	24	0

Nodes summary:

Bad Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes	Shutdown Nodes
0	0	0	0	0

Allocation settings:

Minimum Allocation	Maximum Allocation	Maximum Cluster Application Priority
<memory:1024, vCores:1>	<memory:8192, vCores:4>	0

Search bar and sorting headers:

FinishTime	State	FinalStatus	Running Containers	Allocated CPU Vcores	Allocated Memory MB	Reserved CPU Vcores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
------------	-------	-------------	--------------------	----------------------	---------------------	---------------------	--------------------	------------	--------------	----------	-------------	-------------------

Application details:

2012-07-12 22:25:49	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	0.0	0.0	0.0	<a href="#">History</a>	0
---------------------	----------	-----------	-----	-----	-----	-----	-----	-----	-----	-------------------------	---

Pagination controls at the bottom: First, Previous, 1, Next, Last.

可以看到所有的提交的作业的运行情况、资源占用情况、运行还有多长时间结束等信息

## 第五节 MapReduce 编程实践

下面以 WordCount 例子作为编程实例记录下来，关于 MapReduce 的原理，这里不做过多的讲解。

首先，在本地创建 3 个文件：[file001](#)、[file002](#)、[file003](#)。

file001

```
Hello world
Connected world
```

file002

```
One world
One dream
```

file001

```
Hello Hadoop
Hello Map
Hello Reduce
```

将上述 3 个文件放入 `input` 文件夹中, 用 `hdfs dfs -put` 命令将 `input` 文件夹上传至 DFS 上。下面就可以开始编程了。

### POM 依赖

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>3.1.2</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>3.1.2</version>
</dependency>
```

首先是编写 Map 程序: Hadoop MapReduce 框架已经在类 Mapper 中实现了 Map 任务的基本功能。为了实现 Map 任务, 开发者只需要继承类 Mapper, 并实现该类的 `map` 函数。为实现单词计数的 Map 任务, 首先为类 Mapper 设定好输入类型和输出类型。这里, Map 函数的输入是 `<key,value>` 形式, 其中, key 是输入文件中一行的行号, value 是该行号对应的一行内容。所以, Map 函数的输入类型为 `<IntWritable, Text>`。Map 函数的功能为完成文本分割工作, Map 函数的输出也是 `<key, value>` 形式, 其中, key 是单词, value 为该单词出现的次数。所以, Map 函数的输出类型为 `<Text, IntWritable>`。以下是单词计数程序的 Map 任务的实现代码。

### CoreMapper.java

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;
import java.util.StringTokenizer;

public class CoreMapper extends Mapper<Object, Text, Text,
    IntWritable> {

    private final IntWritable one = new IntWritable(1);
    private Text word = new Text();

    @Override
```

```

protected void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {
    StringTokenizer stringTokenizer = new StringTokenizer(value.
        toString());
    while(stringTokenizer.hasMoreTokens()) {
        word.set(stringTokenizer.nextToken());
        context.write(word, one);
    }
}
}

```

在上述代码中，实现 Map 任务的类为CoreMapper。该类首先将需要输出的两个变量one 和label 进行初始化。

- 变量one 的初始值直接设置为 1，表示某个单词在文本中出现过。
- Map 函数的前两个参数是函数的输入参数，value 为 Text 类型，是指每次读入文本的一行，key 为 Object 类型，是指输入的行数据在文本中的行号。

`StringTokenizer`类将 value 变量中文本的一行文字进行拆分，拆分后的单词放在`tokenizer`列表中。然后程序通过循环对每一个单词进行处理，把单词放在`label`中，把`one`作为单词计数。在函数的整个执行过程中，`one`的值一直是 1。在该实例中，`key`没有被明显地使用到。`context`是 Map 函数的一种输出方式，通过使用该变量，可以直接将中间结果存储在其中。

编写 MapReduce 程序的第二个任务就是编写 Reduce 程序。在单词计数任务中，Reduce 需要完成的任务就是把输入结果中的数字序列进行求和，从而得到每个单词的出现次数。

在执行完 Map 函数之后，会进入 Shuffle 阶段，在这个阶段中，MapReduce 框架会自动将 Map 阶段的输出结果进行排序和分区，然后再分发给相应的 Reduce 任务去处理。经过 Map 端 Shuffle 阶段后的结果如表 3 所示。

CoreReducer.java

```

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class CoreReducer extends Reducer<Text, IntWritable, Text,
    IntWritable> {

```

```

private IntWritable result = new IntWritable();

@Override
protected void reduce(Text key, Iterable<IntWritable> values,
    Context context) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val :
        values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
}

```

最后是 Main 函数，需要在 Main 函数中指定 Job，然后将定义的 Mapper 和 Reducer 添加到 Job 中。

WordCount.java

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class WordCount {
    public static void main(String[] args) throws IOException,
        ClassNotFoundException, InterruptedException {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "WordCount");
        job.setJarByClass(WordCount.class);
        job.setOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
    }
}

```

```

        job.setMapperClass(CoreMapper.class);
        job.setReducerClass(CoreReducer.class);
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

```

运行代码，需要将代码进行打包，然后提交到集群中才能运行。这里采用 Maven Package 的方法打包应用，形成 Jar 包后用 `hadoop jar` 命令提交。

### 提交 WordCount 到集群

```
# arg1 jar包名称, arg2 主类名 arg3 输入目录 arg4 输出目录
$ hadoop jar hadoopdemo-1.0.jar WordCount /input /output
```

程序运行较慢，运行之后的输出结果如下图：

```
(base) zhangyu@bd2a17alc41a:~/project/hadoopdemo/target$ hdfs dfs -ls /output
Found 2 items
-rw-r--r-- 3 zhangyu supergroup      0 2019-07-18 11:07 /output/_SUCCESS
-rw-r--r-- 3 zhangyu supergroup 66 2019-07-18 11:07 /output/part-r-00000
(base) zhangyu@bd2a17alc41a:~/project/hadoopdemo/target$ hdfs dfs -cat /output/*
Connected      1
Hadoop 1
Hello 4
Map 1
One 2
Reduce 1
dream 1
world 3
(base) zhangyu@bd2a17alc41a:~/project/hadoopdemo/target$
```

图 2: 提交 WordCount 到集群运行结果



### 第一节 简介

Hive 提供了简化 Mapreduce 查询的功能，可以输入类 SQL 语句（HQL 或 HiveQL）完成对 HDFS 中数据的查询。是非常重要的一个工具。

### 第二节 配置文件

## hive-env.sh

```
HADOOP_HOME=/opt/hadoop  
export HIVE_CONF_DIR=/home/zhangyu/hive/conf
```

## hive-site.xml

```
<configuration>  
  <property>  
    <name>javax.jdo.option.ConnectionURL</name>  
    <value>jdbc:mysql://master:3306/hive?  
      createDatabaseIfNotExist=true;  
      characterEncoding=latin1</value>  
  </property>  
  <property>  
    <name>javax.jdo.option.ConnectionDriverName</name>  
    <value>com.mysql.jdbc.Driver</value>  
  </property>  
  <property>  
    <name>javax.jdo.option.ConnectionUserName</name>  
    <value>root</value>  
  </property>  
  <property>  
    <name>javax.jdo.option.ConnectionPassword</name>  
    <value>password</value>  
  </property>  
</configuration>
```

启动 Hive 之前，需要配置确保安装和配置好了 MySQL，否则会报拒绝连接错误。

## Hive 启动步骤

```
# 更新软件源，如果必要的话  
$ sudo apt update  
# 安装mysql，设置root用户密码：password  
$ sudo apt install mysql-server  
$ mysql -u root -p password  
mysql> create database hive;  
# 由于配置的是远程访问  
# 可能需要设置mysql访问的主机权限
```

```
mysql> use mysql;
mysql> select user, host from user;
mysql> flush privilege;
mysql> exit;
# 初次连接，使用 schema-tool 初始化 mysql 中 Hive 数据库中的表
$ schema-tool --initSchema
# 测试运行
$ hive
```

### 第三节 Hive 查询与 Shell 操作

Hive 定义了一套自己的 SQL，简称 HQL，它与关系型数据库的 SQL 略有不同，但支持了绝大多数的语句如 DDL、DML 以及常见的聚合函数、连接查询、条件查询。DDL 操作（数据定义语言）包括：Create、Alter、Show、Drop 等。

1. create database - 创建新数据库
2. alter database - 修改数据库
3. drop database - 删除数据库
4. create table - 创建新表
5. alter table - 变更（改变）数据库表
6. drop table - 删除表
7. create index - 创建索引（搜索键）
8. drop index - 删除索引
9. show table - 查看表

DML 操作（数据操作语言）包括：Load 、Insert、Update、Delete、Merge。

1. load data - 加载数据
2. insert into - 插入数据
3. insert overwrite - 覆盖数据（insert ... values 从 Hive 0.14 开始可用。）
4. update table - 更新表（update 在 Hive 0.14 开始可用，并且只能在支持 ACID 的表上执行）
5. delete from table where id = 1; - 删除表中 ID 等于 1 的数据（delete 在 Hive 0.14 开始可用，并且只能在支持 ACID 的表上执行）

## 6. merge - 合并 (MERGE 在 Hive 2.2 开始可用，并且只能在支持 ACID 的表上执行)

注意：频繁的 update 和 delete 操作已经违背了 Hive 的初衷。不到万不得已的情况，还是使用增量添加的方式最好。

```
$ wget http://192.168.1.100:60000/allfiles/hive3/buyer_log
$ wget http://192.168.1.100:60000/allfiles/hive3/buyer_favorite
$ hive

hive> create table buyer_log(id string,
buyer_id string,dt string,
ip string,opt_type string)
row format delimited fields
terminated by '\t' stored as textfile;
hive> create table buyer_favorite(
      buyer_id string,goods_id string,dt string)
row format delimited fields
terminated by '\t' stored as textfile;
hive> select * from buyer_log limit 10;
hive> load data local inpath
'/data/hive3/buyer_log' into table buyer_log;
hive> load data local inpath
'/data/hive3/buyer_favorite' into table buyer_favorite;
hive> select * from buyer_log limit 10;
hive> select l.dt,f.goods_id from buyer_log l,buyer_favorite f
where l.buyer_id = f.buyer_id limit 10;
```

Hive 的语法与 SQL 类似，不做详细的分析，需要时也能找到很多的资料。下面是将提供的搜狗数据导入到 Hive 中的一些脚本。

数据说明如下，其中的分隔符为\t：

sogou-data 数据格式说明

访问时间	用户 id	查询关键字	浏览器返回排名	点击顺序	点击 url
------	-------	-------	---------	------	--------

20111230000005	57375476989eea12893c0c3811607bcf	奇艺高清	1	1	http://www.qiyi.com/
20111230000005	66c5bb7774e31d0a22278249b26bc83a	凡人修仙传	3	1	http://www.booksy.org/BookDetail.aspx?BookID=1050804&Level=1
20111230000007	b97920521c78de70ac38e3713f524b50	本本联盟	1	1	http://www.bblianmeng.com/
20111230000008	6961d0c97fe93701fc9c0d861d096cd9	华南师范大学图书馆	1	1	http://lib.scnu.edu.cn/
20111230000008	f2f5a2lc764aebe1ebafcc2871e086f	在线代理	2	1	http://proxyie.cn/
20111230000009	96994a0480e7e1edcaef67b20d8816b7	伟大导演	1	1	http://movie.douban.com/review/1128960/
20111230000009	698956eb07815439fef5f46e9a4503997	youku	1	1	http://www.youku.com/
20111230000009	599cd26984f72ee68b2b6ebfccf6aed	安徽合肥365房产网	1	1	http://hf.house365.com/
20111230000010	f577230df7b6c532837cd16ab731f874	哈萨克网址大全	1	1	http://www.kz321.com/
20111230000010	285f88780dd0659f5fc8acc7cc4949f2	IQ数码	1	1	http://www.iqshuma.com/

sogou-data.sql

```

drop table if exists sogou_data1;
create table sogou_data1(
    dt string,
    user_id string,
    search_word string,
    rank int,
    seq int,
    url string
)
row format delimited fields
terminated by '\t' stored as textfile;
load data local inpath '/home/zhangyu/download/sogou-data/500w/sogou
.500w.utf8' into table sogou_data1;

```

用 Hive 来做查询，效率要比编写 MR 程序再打包提交要高效很多，在实践使用过程中也能够体会到，Hive 查询的时延不是很高，体验较好。

## 第四节 Hive JDBC 编程

使用 IDEA 开发 Hive JDBC 编程，用 Maven 管理项目依赖。

启动 Hive 服务，注意这里使用的不是 Cli：

启动 Hive 服务

```
$ hive --service hiveserver2 &
```

查看 Hive 服务

```
$ netstat -nptl | grep 10000
# 出现监听进程，说明启动正常
```

启动 beeline，并尝试连接

```
$ beeline
$ !connect jdbc:hive2://master:10000
```

在这里连接的时候总是报错，无法连接成功，查看后发现主要是授权的问题，可是用户名和密码都输入正确了。后来通过搜索查看到问题所在，修改 Hadoop 配置 `core-site.xml` 文件，添加如下内容取消权限检查，便能够成功进入了。

```
<property>
    <name>hadoop.proxyuser.zhangyu.hosts</name>
```

```

<value>*</value>
</property>
<property>
    <name>hadoop.proxyuser.zhangyu.groups</name>
    <value>*</value>
</property>

```

下面开始新建项目编程：

### POM 文件依赖

```

<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-jdbc</artifactId>
    <version>1.2.1</version>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>3.1.2</version>
</dependency>

```

注意这里的版本用的是 1.2.1，而不是 Hive 的 3.1.1。因为发现使用 3.1.1 时，有依赖无法解决，一直飘红。所以找了一个使用量比较高的版本 1.2.1，因为测试比较简单，仅仅是输出打印，降低版本后能够正常运行了。

### HiveDemo.java

```

import java.sql.*;

public class HiveDemo {
    public static void main(String[] args) {
        try {
            Connection connection = null;
            String driverName = "org.apache.hive.jdbc.HiveDriver";
            Class.forName(driverName);
            // arg1: connect url, arg2: username, arg3: password
            connection = DriverManager.getConnection("jdbc:hive2://
                master:10000/default", "root", "password");
            Statement statement = connection.createStatement();
            ResultSet resultSet = null;
        }
    }
}

```

```
String sql = "select * from sogou_data1";
System.out.println("Now running: " + sql);
resultSet = statement.executeQuery(sql);
while(resultSet.next()) {
    System.out.println(resultSet.getString(1) + '\t'
        + resultSet.getString(2) + '\t'
        + resultSet.getString(3) + '\t'
        + resultSet.getString(4) + '\t'
        + resultSet.getString(5) + '\t'
    );
}
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}
}
```

程序运行截图如下：

而且发现，IDEA 提供了开箱即用的数据库连接工具，并支持 Hive 连接。配置成功后即可在集成开发环境中编写查询语句了，而且提供了很好的语法提示支持，算是一个彩蛋吧。

```

import java.sql.*;

public class HiveDemo {
    public static void main(String[] args) {
        try {
            Connection connection = null;
            String driverName = "org.apache.hive.jdbc.HiveDriver";
            Class.forName(driverName);
            // arg1: connect url, arg2: username, arg3: password
            connection = DriverManager.getConnection("jdbc:hive2://master:10000/default", "user", "r");
            Statement statement = connection.createStatement();
            ResultSet resultSet = null;
            String sql = "select * from sogou data1";
            System.out.println("Now running: " + sql);
            resultSet = statement.executeQuery(sql);
            while(resultSet.next()) {
                System.out.println(resultSet.getString("columnIndex_1") + '\t'
                    + resultSet.getString("columnIndex_2") + '\t'
                    + resultSet.getString("columnIndex_3") + '\t');
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

The screenshot shows the IntelliJ IDEA interface with the code editor displaying the above Java code. Below the code editor is a run output window showing the results of the query:

搜索词	次数
歪歪电影频道带色的	1 1
个性网	1 1
植物大战僵尸下载	1 1
一对基友的对话	youku 1 1
揪痧吧	3 2
惩罚耽美道具	4 3
笑话网	10 2
电视剧国门英雄剧情介绍	1 1
金玉峰	8 1
20个月宝宝食谱	4 4
殷人尊神，率民以事神，先鬼而后礼	4 1
幕府 2 没声音	8 4
2012年人教版小学二年级上册语文期末试题	10 6
湖北众鑫药业在哪？	2 1
哈尔滨交通违章查询	8 4
哪有学习安装消防器材的学校	4 1

图 3: Hive JDBC 编程运行效果

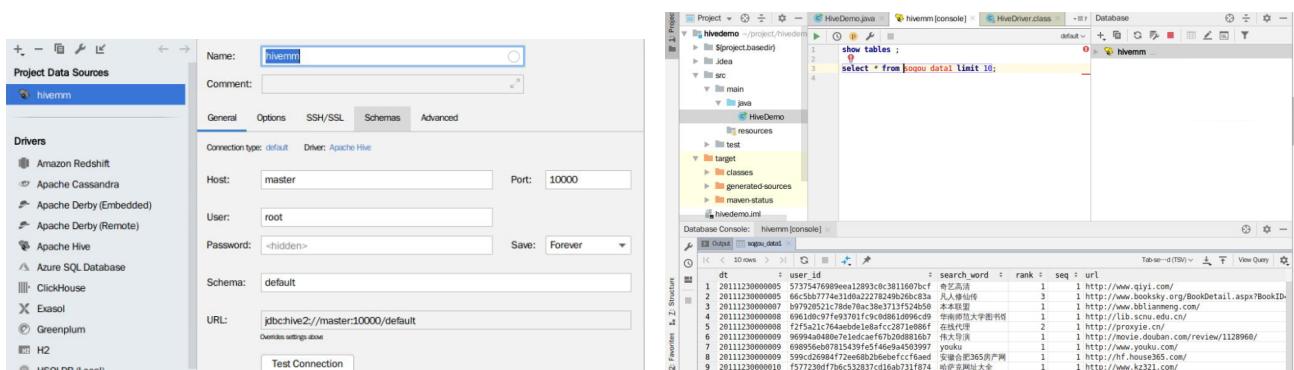
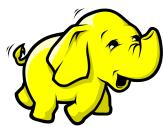


图 4: IDEA 对 Hive 的良好支持



## 0.7. 安装 HBase

### 第一节 简介

HBase 的设计来源于 Google 公司的 BigTable 论文，相当于是一个数据库，其查询的实时性较高。也是 Hadoop 生态系统中的一个很重要的工具。

### 第二节 配置文件

hbase-env.sh

```
export JAVA_HOME=/opt/java
export HBASE_MANAGES_ZK=false
export HADOOP_HOME=/opt/hadoop
HBASE_CLASSPATH=$HBASE_CLASSPATH:$HADOOP_HOME/etc/hadoop
```

hbase-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>hbase.rootdir</name>
  <value>hdfs://ldcluster:8020/hbase</value>
</property>
<property>
  <name>hbase.cluster.distributed</name>
  <value>true</value>
</property>
<property>
  <name>hbase.unsafe.stream.capability.enforce</name>
  <value>false</value>
</property>
<property>
  <name>hbase.zookeeper.quorum</name>
  <value>master,slave1,slave2</value>
</property>
```

```
<property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/home/zhangyu/zookeeperdata/</value>
</property>
</configuration>
```

我在配置 HBase 时，经常出现 HMaster 进程自动退出的问题，查看日志，调试了很长的时间。多半是因为访问时权限的问题。可以在[hdfs-site.xml](#)文件夹中加入相应的权限设置选项，取消权限设置。

并且按照官网推荐，在 HBase 的配置目录下面，添加一个指向[hdfs-site.xml](#)的软连接。

启动 HBase，需要先启动 HBase 服务。

启动 HBase

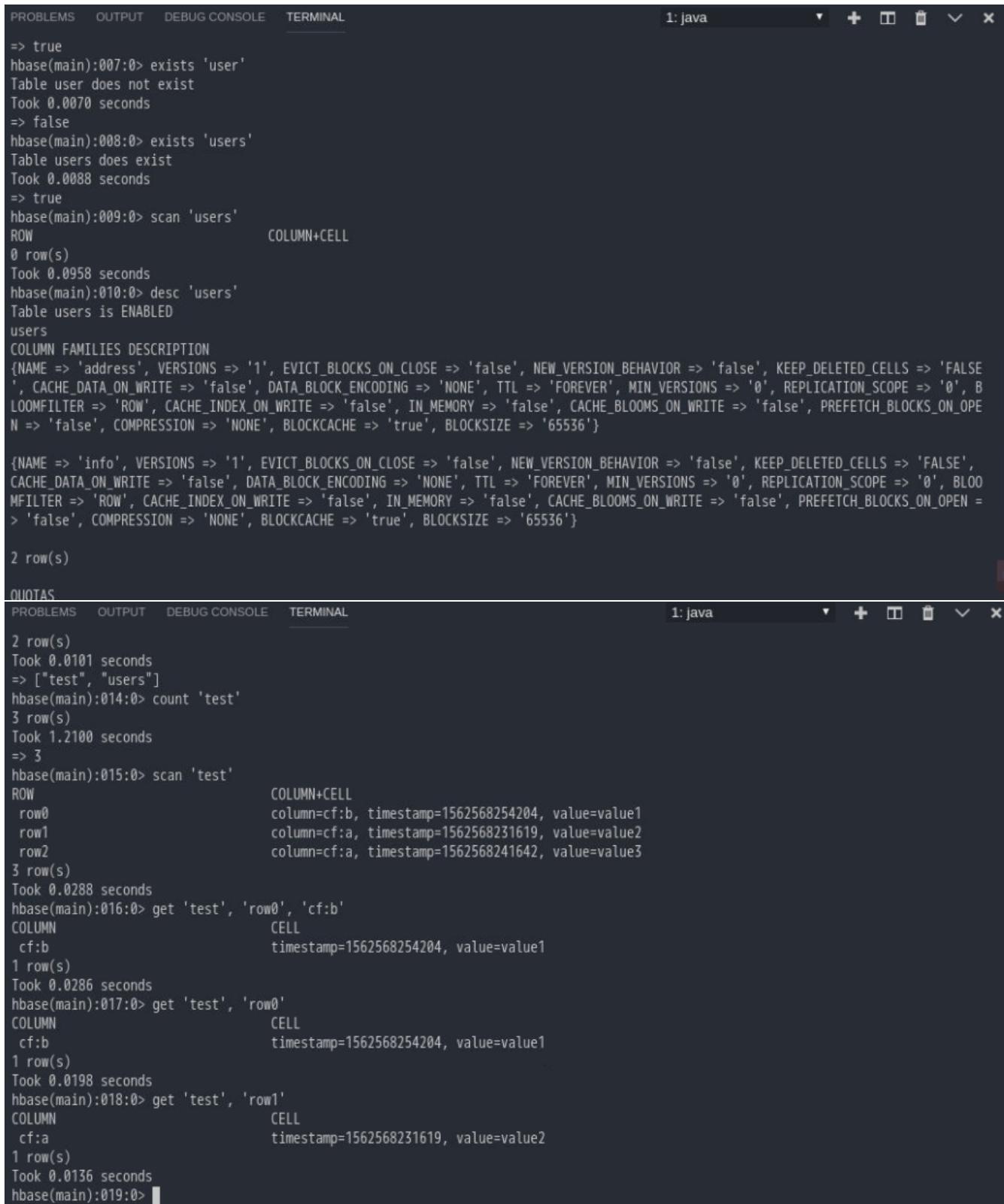
```
$ start-hbase.sh
$ jps
# 使用Jps查看进程可以看到主节点上多了HMaster进程和HRegionServer进程
# 在从节点上多了HRegionServer进程
# 如果出现过几秒后HMaster进程结束的情况，需要查看日志进行排查
```

### 第三节 HBase Shell 操作

下面是一些操作运行效果的截图：

表 3: HBase 操作指南

操作	命令表达式
创建表	create ”表名”, ”列族 1”, ”列族 2”, …, ”列族 n”
列出表	list
获取表的描述	desc ”表名”
修改表	alter ”表名”，应先 disable ”表名”
删除表	drop ”表名”，应先 disable ”表名”
判断表是否存在	exists ”表名”
判断表是否是 enable	is_enabled ”表名”
判断表是否是 disable	is_disabled ”表名”
插入数据	put ”表名”, ”行键”, ”列族名:列名”, ”值”
获取数据	get ”表名”, ”行键”, ”列族名:列名”
全表扫描	scan ”表名”
删除数据	delete ”表名”, ”行键”, ”列族名:列名”
查询行数	count ”表名”
清空该表	truncate ”表名”
更新表中数据	执行插入语句覆盖原来数据



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: java
=> true
hbase(main):007:0> exists 'user'
Table user does not exist
Took 0.0070 seconds
=> false
hbase(main):008:0> exists 'users'
Table users does exist
Took 0.0088 seconds
=> true
hbase(main):009:0> scan 'users'
ROW COLUMN+CELL
0 row(s)
Took 0.0958 seconds
hbase(main):010:0> desc 'users'
Table users is ENABLED
users
COLUMN FAMILIES DESCRIPTION
{NAME => 'address', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_BEHAVIOR => 'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN VERSIONS => '0', REPLICATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_MEMORY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOCKS_ON_OPEN => 'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536'}

{NAME => 'info', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_BEHAVIOR => 'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN VERSIONS => '0', REPLICATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_MEMORY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOCKS_ON_OPEN => 'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536'}

2 row(s)

QUOTAS
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: java
2 row(s)
Took 0.0101 seconds
=> ["test", "users"]
hbase(main):014:0> count 'test'
3 row(s)
Took 1.2100 seconds
=> 3
hbase(main):015:0> scan 'test'
ROW COLUMN+CELL
row0 column=cf:b, timestamp=1562568254204, value=value1
row1 column=cf:a, timestamp=1562568231619, value=value2
row2 column=cf:a, timestamp=1562568241642, value=value3
3 row(s)
Took 0.0288 seconds
hbase(main):016:0> get 'test', 'row0', 'cf:b'
COLUMN CELL
cf:b timestamp=1562568254204, value=value1
1 row(s)
Took 0.0286 seconds
hbase(main):017:0> get 'test', 'row0'
COLUMN CELL
cf:b timestamp=1562568254204, value=value1
1 row(s)
Took 0.0198 seconds
hbase(main):018:0> get 'test', 'row1'
COLUMN CELL
cf:a timestamp=1562568231619, value=value2
1 row(s)
Took 0.0136 seconds
hbase(main):019:0>

```

可以看出 HBase 的操作比较简单，这是其灵活性和扩展性的一种保障。

## 第四节 HBase Java API 使用

HBase 的 Java API 主要包括 5 大类的操作：HBase 的配置、HBase 表的管理、列族的管理、列的管理、数据的操作。

- 1) `org.apache.hadoop.hbase.HBaseConfiguration`。

`HBaseConfiguration` 类用于管理 HBase 的配置信息，使用举例如下。

```
static Configuration cfg = HBaseConfiguration.create();
```

- 2) `org.apache.hadoop.hbase.client.Admin`

`Admin` 是 Java 接口类型，不能直接用该接口来实例化一个对象，而是必须通过调用 `Connection.getAdmin()` 方法，来调用返回子对象的成员方法。该接口用来管理 HBase 数据库的表信息。它提供的方法包括创建表，删除表，列出表项，使表有效或无效，以及添加或删除表列族成员等。

创建表使用的例子如下。

```
Configuration configuration = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(
    configuration);
Admin admin = connection.getAdmin();
if(admin.tableExists(tableName)) { //如果存在要创建的表，那么先删除，再创建
    admin.disableTable(tableName);
    admin.deleteTable(tableName);
}
admin.createTable(tableDescriptor);
admin.disableTable(tableName);
HColumnDescriptor hd = new HColumnDescriptor(columnFamily);
admin.addColumn(tableName, hd);
```

- 3) `org.apache.hadoop.hbase.HTableDescriptor`

`HTableDescriptor` 包含了表的详细信息。创建表时添加列族使用的例子如下。

```
HTableDescriptor tableDescriptor = new HTableDescriptor(tableName)
    ;// 表的数据模式
tableDescriptor.addFamily(new HColumnDescriptor("name")); // 增加列族
tableDescriptor.addFamily(new HColumnDescriptor("age"));
```

```
tableDescriptor.addFamily(new HColumnDescriptor("gender"));
admin.createTable(tableDescriptor);
```

#### 4) org.apache.hadoop.hbase.client.Table

Table 是 Java 接口类型，不可以用 Table 直接实例化一个对象，而是必须通过调用 `connection.getTable()` 的一个子对象，来调用返回子对象的成员方法。这个接口可以用来和 HBase 表直接通信，可以从表中获取数据、添加数据、删除数据和扫描数据。例子如下。

```
Configuration configuration = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(
    configuration);
Table table = connection.getTable();
ResultScanner scanner = table.getScanner(family);
```

#### 5) org.apache.hadoop.hbase.client.Put

Put 类用来对单元执行添加数据操作。给表里添加数据的例子如下

```
Configuration configuration = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(
    configuration);
Table table = connection.getTable();
Put put = new Put ("*1111".getBytes()); //一个 Put 代表一行数据，行
//键为构造方法中传入的值
put.addColumn("name".getBytes(), null, "Ghander".getBytes()); //本行数
//据的第一列
put.addColumn("age".getBytes(), null, "20".getBytes()); // 本行数据的
//第二列
put.addColumn("gender".getBytes(), null, "male".getBytes()); // 本行数
//据的第三列
put.add("score".getBytes(), "Math".getBytes(), "99".getBytes()); // 本
//行数据的第四列
table.put(put);
```

#### 6) org.apache.hadoop.hbase.client.Get

Get 类用来获取单行的数据。获取指定单元的数据的例子如下。

```
Configuration configuration = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(
    configuration);
```

```
Table table = connection.getTable();
Get g = new Get(rowKey.getBytes());
Result rs = table.get(g);
```

#### 7) org.apache.hadoop.hbase.client.Result

Result 类用来存放 Get 或 Scan 操作后的查询结果，并以<key,value>的格式存储在映射表中。获取指定单元的数据的例子如下。

```
Configuration configuration = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(
    configuration);
Table table = connection.getTable();
Get g = new Get(rowKey.getBytes());
Result rs = table.get(g);
for (KeyValue kv : rs.raw()) {
    System.out.println("rowkey:" + new String(kv.getRow()));
    System.out.println("Column Family:" + new String(kv.getFamily()));
    System.out.println("Column :" + new String(kv.getQualifier()));
    System.out.println("value :" + new String(kv.getValue()));
}
```

#### 8) org.apache.hadoop.hbase.client.Scan

Scan 类可以用来限定需要查找的数据，如版本号、起始行号、终止行号、列族、列限定符、返回值的数量的上限等。设置 Scan 的列族、时间戳的范围和每次最多返回的单元数目的例子如下。

```
Scan scan = new Scan();
scan.addFamily(Bytes.toBytes("columnFamily1"));
scan.setTimeRange(1, 3);
scan.setBatch(1000);
```

#### 9) org.apache.hadoop.hbase.client.ResultScanner

ResultScanner 类是客户端获取值的接口，可以用来限定需要查找的数据，如版本号、起始行号、终止行号、列族、列限定符、返回值的数量的上限等。获取指定单元的数据的例子如下。

```
Scan scan = new Scan();
```

```
scan.addColumn(Bytes.toBytes("columnFamily1"), Bytes.toBytes("column1"));
scan.setTimeRange(1,3);
scan.setBatch(10);
Configuration configuration = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(
    configuration);
Table table = connection.getTable();
try {
    ResultScanner resultScanner = table.getScanner(scan);
    Result rs = resultScanner.next();
    for (; rs != null; rs = resultScanner.next()){
        for (KeyValue kv : rs.list()){
            System.out.print("-----");
            System.out.print("rowkey:" + new String(kv.getRow()));
            System.out.print("Column Family: " + new String(
                kv.getFamily()));
            System.out.print("Column :" + new String(
                kv.getQualifier()));
            System.out.print("value :" + new String(
                kv.getValue()));
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

下面演示一个具体的编程实例来学习如何使用 HBase Java API 解决实际问题。在本实例中，首先创建一个学生成绩表 scores，用来存储学生各门课程的考试成绩，然后向 scores 添加数据。

表 scores 的概念视图如图??所示，用学生的名字 name 作为行键，年级 grade 是一个只有一个列的列族，score 是一个列族，每一门课程都是 score 的一个列，如 english、math、Chinese 等。score 的列可以随时添加。

例如，后续学生又参加了其他课程的考试，如 computing、physics 等，那么就可以添加到 score 列族。因为每个学生参加考试的课程也会不同，所以，并不一定表中的每一个单元都会有值。在该实例中，要向学生成绩表 scores 中添加的数据如图??所示。

下面利用 Maven 和 IDEA 创建工程，在其中添加下面的依赖：

name	grade	score		
		english	math	chinese

图 5: 学生成绩表 scores 的概念视图

name	grade	score		
		english	math	chinese
dandan	6	95	100	92
sansan	6	87	95	98

图 6: 学生成绩表 scores 的数据

## 添加 Hbase Maven 依赖

```
<!-- https://mvnrepository.com/artifact/org.apache.hbase/hbase -->
<dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase</artifactId>
    <version>2.2.0</version>
    <type>pom</type>
</dependency>
```

新建主程序，写入下面代码框架：

## 主程序框架

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;
public class StudentScores {
    public static Configuration configuration; //HBase 配置信息
    public static Connection connection; //HBase 连接
    public static void main (String [] agrs) throws IOException{
        init(); //建立连接
        createTable(); //建表
        insertData(); //添加课程成绩
        insertData(); //添加课程成绩
    }
}
```

```

        insertData(); //添加课程成绩
        getData(); //浏览课程成绩
        close(); //关闭连接
    }

    public static void init () {.....} //建立连接
    public static void close () {.....} //关闭连接
    public static void createTable (){.....} //创建表
    public static void insertData () {.....} //添加课程成绩
    public static getData() {.....} //浏览操程成绩
}

```

然后分别实现下面的几个函数：

init 函数，用于初始化连接

```

public static void init() {
    configuration = HBaseConfiguration.create();
    configuration.set("hbase.rootdir", "hdfs://ldcluster:8020/
        hbase");
    try {
        connection = ConnectionFactory.createConnection(
            configuration);
        admin = connection.getAdmin();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

close 函数，用于结束断开连接

```

public static void close() {
    try {
        if (admin != null) {
            admin.close();
        }
        if (connection != null) {
            connection.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

    }
}
```

### createTable 函数

```

public static void createTable(String myTableName, String[] colFamily)
) throws IOException {
    TableName tableName = TableName.valueOf(myTableName);
    if (admin.tableExists(tableName)) {
        System.out.println("The " + myTableName + " exists!");
    } else {
        HTableDescriptor hTableDescriptor = new
            HTableDescriptor(tableName);
        for (String str :
                colFamily) {
            HColumnDescriptor columnDescriptor = new
                HColumnDescriptor(str);
            hTableDescriptor.addFamily(columnDescriptor);
        }
        admin.createTable(hTableDescriptor);
    }
}
```

### 向表中插入数据的函数

```

public static void insertData(String tableName, String rowKey, String
    colFamily, String col, String val) throws IOException {
    Table table = connection.getTable(TableName.valueOf(tableName));
    Put put = new Put(rowKey.getBytes());
    put.addColumn(colFamily.getBytes(), col.getBytes(), val.
        getBytes());
    table.put(put);
    table.close();
}
```

### 从表中获取数据的函数

```

public static void getData(String tableName, String rowKey, String
    colFamily, String col) throws IOException {
```

```
Table table = connection.getTable(TableName.valueOf(tableName));
Get get = new Get(rowKey.getBytes());
get.addColumn(colFamily.getBytes(), col.getBytes());
Result result = table.get(get);
System.out.println(new String(result.getValue(colFamily.
    getBytes(), col.getBytes())));
table.close();
}
```

完整的主函数如下：

### 完整的主函数

```
public static void main(String[] args) throws IOException {
    String tableName = "scores";
    String[] colFamily = {"grade", "score"};

    init();
    createTable(tableName, colFamily);
    insertData(tableName, "dandan", "grade", "", "6");
    insertData(tableName, "dandan", "score", "english", "95");
    insertData(tableName, "dandan", "score", "math", "100");
    insertData(tableName, "dandan", "score", "chinese", "92");

    insertData(tableName, "sansan", "grade", "", "6");
    insertData(tableName, "sansan", "score", "english", "87");
    insertData(tableName, "sansan", "score", "math", "95");
    insertData(tableName, "sansan", "score", "chinese", "98");

    getData(tableName, "dandan", "score", "math");
    getData(tableName, "sansan", "score", "english");
    getData(tableName, "sansan", "grade", "");

    close();
}
```

运行结果如下：

```
Run: StudentScores x
/opt/java/bin/java ...
log4j:WARN No appenders could be found for logger (org.apache.hadoop.util.Shell).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
The scores exists!
100
87
6
Process finished with exit code 0
```

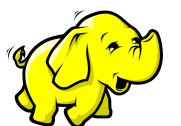
Terminal Messages Run TODO Event Log

可以在控制台看到输出的结果，但是需要运行等待的时间较长。

```
hbase(main):009:0> scan "scores"
ROW                                     COLUMN+CELL
dandan                                  column=grade:, timestamp=1563416876831, value=6
dandan                                  column=score:chinese, timestamp=1563416876859, value=92
dandan                                  column=score:english, timestamp=1563416876839, value=95
dandan                                  column=score:math, timestamp=1563416876852, value=100
samsan                                 column=grade:, timestamp=1563416876876, value=6
samsan                                 column=score:chinese, timestamp=1563416876894, value=98
samsan                                 column=score:english, timestamp=1563416876880, value=87
samsan                                 column=score:math, timestamp=1563416876884, value=95
2 row(s)
Took 0.0204 seconds
hbase(main):010:0>
```

在 Hbase shell 中也可以看到表的信息

总结：采用 HBase Shell 操作的实时性较好，但是灵活性不够，适合演示或小量数据。而采用 Java API 编程能够加深对 HBase 各个模块的理解，而且更加灵活，适合各种复杂大数据量情况。



## 0.8. 安装 Spark

### 第一节 简介

Spark 是一个内存计算框架，其速度能够在很大程度上超过 Hadoop 的 Mapreduce，并逐渐取代其，在大数据分析和处理上占据了重要的地位。Spark 的安装有本地模式安装、Standalone 模式安装和集群模式安装。集群模式需要借助 YARN 或 Meso，而且与 Stadalone 模式配置类似。这里采用 Spark on YARN 实现集群。

### 第二节 通过 SDKMAN 安装 Scala

运行 Spark 或者 Spark 开发需要 Scala 运行环境，这里采用了官网推荐的通过 SDKMAN 方法安装。当然也可以采用下载二进制包然后配置环境变量的方式，不过多描述。

#### 安装 Scala

```
$ sudo apt install curl
$ curl -s "https://get.sdkman.io" | bash
```

```
# this take a while
$ source ~/.bashrc
$ sdk install scala
# 查看 scala 所在的目录
$ whereis scala
# 将其配置到 .bashrc 环境变量中
# 输入 scala 进行测试
$ scala
scala>
```

### 第三节 配置文件

在安装好 Scala 和具备 Hadoop 环境后，即可进行 Spark 的安装和部署。

spark-env.sh

```
#!/usr/bin/env bash

export SPARK_DIST_CLASSPATH=$( /opt/hadoop/bin/hadoop classpath )

export JAVA_HOME=/opt/java
export SCALA_HOME=/home/zhangyu/.sdkman/candidates/scala/2.13.0
export SPARK_WORKING_MEMORY=1g #每一个 worker 节点上可用的最大内存
export SPARK_MASTER_IP=master #驱动器节点 IP
export SPARK_CONF_DIR=/home/zhangyu/spark/conf
export HADOOP_HOME=/opt/hadoop
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

slaves

```
master
slave1
slave2
```

而且可以设置 `log4j.properties.template` 文件设置输出日志级别，避免打印过多的 INFO 级别的日志信息。

## 第四节 测试 Spark Shell

Spark shell 是一个特别适合快速开发 Spark 程序的工具。即使你对 Scala 不熟悉，仍然可以使用这个工具快速应用 Scala 操作 Spark。Spark shell 使得用户可以和 Spark 集群交互，提交查询，这便于调试，也便于初学者使用 Spark。Spark shell 是非常方便的，因为它很大程度上基于 Scala REPL(Scala 交互式 shell，即 Scala 解释器)，并继承了 Scala REPL(读取-求值-打印-循环)(Read-Evaluate-Print-Loop) 的所有功能。运行 spark-shell，则会运行 spark-submit，spark-shell 其实是对 spark-submit 的一层封装。

下面是 Spark shell 的运行原理图

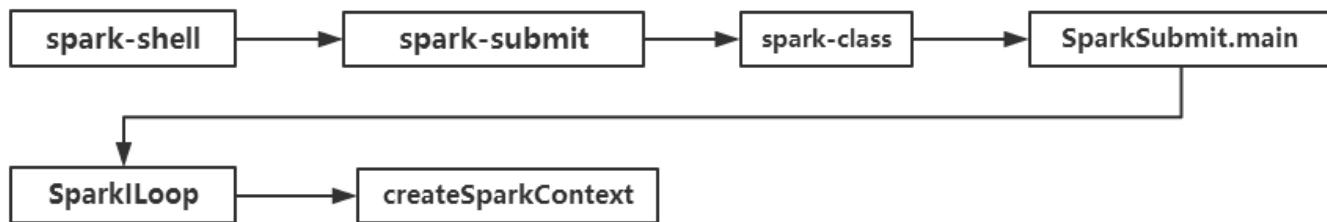


图 7: Spark shell 的运行原理图

RDD 有两种类型的操作，分别是 Transformation（返回一个新的 RDD）和 Action（返回 values）。

- Transformation: 根据已有 RDD 创建新的 RDD 数据集 build
  - 1) map(func): 对调用 map 的 RDD 数据集中的每个 element 都使用 func，然后返回一个新的 RDD，这个返回的数据集是分布式的数据集。
  - 2) filter(func) : 对调用 filter 的 RDD 数据集中的每个元素都使用 func，然后返回一个包含使 func 为 true 的元素构成的 RDD。
  - 3) flatMap(func): 和 map 很像，但是 flatMap 生成的是多个结果。
  - 4) mapPartitions(func): 和 map 很像，但是 map 是每个 element，而 mapPartitions 是每个 partition。
  - 5) mapPartitionsWithSplit(func): 和 mapPartitions 很像，但是 func 作用的是其中一个 split 上，所以 func 中应该有 index。
  - 6) sample(withReplacement,fraction,seed): 抽样。
  - 7) union(otherDataset): 返回一个新的 dataset，包含源 dataset 和给定 dataset 的元素的集合。
  - 8) distinct([numTasks]): 返回一个新的 dataset，这个 dataset 含有的是源 dataset 中的 distinct 的 element。
  - 9) groupByKey(numTasks): 返回 (K,Seq[V])，也就是 Hadoop 中 reduce 函数接受的 key-value list。

- 10) `reduceByKey(func,[numTasks])`: 就是用一个给定的 `reduce func` 再作用在 `groupByKey` 产生的 `(K,Seq[V])`, 比如求和, 求平均数。
- 11) `sortByKey([ascending],[numTasks])`: 按照 key 来进行排序, 是升序还是降序, `ascending` 是 boolean 类型。
- Action: 在 RDD 数据集运行计算后, 返回一个值或者将结果写入外部存储
    - 1) `reduce(func)`: 就是聚集, 但是传入的函数是两个参数输入返回一个值, 这个函数必须是满足交换律和结合律的。
    - 2) `collect()`: 一般在 `filter` 或者足够小的结果的时候, 再用 `collect` 封装返回一个数组。
    - 3) `count()`: 返回的是 dataset 中的 element 的个数。
    - 4) `first()`: 返回的是 dataset 中的第一个元素。
    - 5) `take(n)`: 返回前 n 个 elements。
    - 6) `takeSample(withReplacement, num, seed)`: 抽样返回一个 dataset 中的 num 个元素, 随机种子 seed。
    - 7) `saveAsTextFile (path)`: 把 dataset 写到一个 textfile 中, 或者 HDFS, 或者 HDFS 支持的文件系统中, Spark 把每条记录都转换为一行记录, 然后写到 file 中。
    - 8) `saveAsSequenceFile(path)`: 只能用在 key-value 对上, 然后生成 SequenceFile 写到本地或者 Hadoop 文件系统。
    - 9) `countByKey()`: 返回的是 key 对应的个数的一个 map, 作用于一个 RDD。
    - 10) `foreach(func)`: 对 dataset 中的每个元素都使用 func。

### 获取实验测试数据

```
wget -P ./spark5 file_ip:60000/allfiles/spark5/orders
wget -P ./spark5 file_ip:60000/allfiles/spark5/order_items

wget file_ip:60000/allfiles/spark3/wordcount/buyer_favorite
wget file_ip:60000/allfiles/spark3/distinct/buyer_favorite
wget -P ./sort file_ip:60000/allfiles/spark3/sort/goods_visit

wget -P ./join file_ip:60000/allfiles/spark3/join/orders
wget -P ./join file_ip:60000/allfiles/spark3/join/order_items

wget -P ./avg file_ip:60000/allfiles/spark3/avg/goods
wget -P ./avg file_ip:60000/allfiles/spark3/avg/goods_visit
```

## 测试 Shell 操作

```
$ hdfs dfs -mkdir /spark5
$ hdfs dfs -put /data/spark5/orders /spark5
$ hdfs dfs -put /data/spark5/order_items /spark5
$ spark-shell

scala> val sqlContext = new org.apache.spark.sql.SQLContext(sc)
scala> import sqlContext.implicits._
scala> case class Orders(order_id:String,order_number:String,buyer_id
   :String,create_dt:String)
scala> val dforders = sc.textFile("/myspark5/orders").map(_.split('\t')).map(line=>Orders(line(0),line(1),line(2),line(3))).toDF()
scala> dforders.registerTempTable("orders")
scala> sqlContext.sql("show tables").map(t=>"tableName is:"+t(0)).collect().foreach(println)
scala> sqlContext.sql("select order_id,buyer_id from orders").collect
scala>
scala> import org.apache.spark.sql._
scala> import org.apache.spark.sql.types._
scala> val rddorder_items = sc.textFile("/myspark5/order_items")
scala> val roworder_items = rddorder_items.map(_.split("\t")).map(p=>Row(p(0),p(1),p(2)))
scala> val schemaorder_items = "item_id order_id goods_id"
scala> val schema = StructType(schemaorder_items.split(" ")).map(fieldName=>StructField(fieldName, StringType, true))
scala> val dforder_items = sqlContext.applySchema(roworder_items, schema)
scala> dforder_items.registerTempTable("order_items")
scala>
scala> sqlContext.sql("show tables").map(t=>"tableName is:"+t(0)).collect().foreach(println)
scala> sqlContext.sql("select order_id,goods_id from order_items ").collect
scala>
scala> sqlContext.sql("select orders.buyer_id, order_items.goods_id
   from order_items join orders on order_items.order_id=orders.
   order_id ").collect
```

```
scala>
```

一些测试过程中的效果截图：

Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_212)  
Type in expressions to have them evaluated.  
Type :help for more information.

```
scala> [T01 0: bash- 1: bash* "bd2a17a1c41a" 11:39 12-7月-19

zhangyu@bd2a17a1c41a:~/datas/spark$ hdfs dfs -ls -R /spark* <memory:1024,vCores:1> <memory>
drwxr-xr-xED - zhangyu supergroup 0 2019-07-12 03:49 /spark/avg <memory>
-rw-r--r--ED 3 zhangyu supergroup 208799 2019-07-12 03:49 /spark/avg/goods <memory>
-rw-r--r--ED 3 zhangyu supergroup 82421 2019-07-12 03:49 /spark/avg/goods_visit <memory>
drwxr-xr-x - zhangyu supergroup Application 0 2019-07-12 03:48 /spark/distinct finalStatus Running Allocated
drwxr-xr-x - zhangyu supergroup Type 0 2019-07-12 03:50 /spark/join <memory> Containers CPU
-rw-r--r-- 3 zhangyu supergroup 328 2019-07-12 03:50 /spark/join/order_items <memory> Vcores
-rw-r--r-- 3 zhangyu supergroup 460 2019-07-12 03:50 /spark/join/orders No data available in table
drwxr-xr-x - zhangyu supergroup entries 0 2019-07-12 03:49 /spark/sort <memory>
-rw-r--r-- 3 zhangyu supergroup 104 2019-07-12 03:49 /spark/sort/goods_visit <memory>
drwxr-xr-x - zhangyu supergroup 0 2019-07-12 03:50 /spark/spark5 <memory>
-rw-r--r-- 3 zhangyu supergroup 315 2019-07-12 03:50 /spark/spark5/order_items <memory>
-rw-r--r-- 3 zhangyu supergroup 460 2019-07-12 03:50 /spark/spark5/orders <memory>
drwxr-xr-x - zhangyu supergroup 0 2019-07-12 03:37 /spark/wordcount <memory>
-rw-r--r-- 3 zhangyu supergroup 1019 2019-07-12 03:37 /spark/wordcount/buyer_favorite <memory>
zhangyu@bd2a17a1c41a:~/datas/spark$
```

buyer\_favorite

用户 id	商品 id	收藏 日期
10181	1000481	2010-04-04 16:54:31
20001	1001597	2010-04-07 15:07:52
20001	1001560	2010-04-07 15:08:27

20042	1001368	2010-04-08 08:20:30
20067	1002061	2010-04-08 16:45:33
20056	1003289	2010-04-12 10:50:55

```
Tools
scala> rdd.map(line => (line.split('\t')(0),1)).reduceByKey(_+_).collect
res4: Array[(String, Int)] = Array((20042,1), (20055,1), (20064,1), (20054,6), (20001,2), (10181,1),
(20067,1), (20056,12), (20076,5))
      Showing 0 to 0 of 0 entries
scala> rdd.map(line => (line.split('\t')(1),1)).reduceByKey(_+_).collect
res5: Array[(String, Int)] = Array((1003055,1), (1003103,2), (1001679,1), (1010675,1), (1003064,1),
(1002061,1), (1002429,1), (1003101,1), (1002427,1), (1003066,2), (1010183,1), (1000481,1), (1010178,
1), (1003290,1), (1003326,1), (1002420,2), (1003292,1), (1001368,1), (1001560,1), (1001597,1), (1003
289,1), (1002422,2), (1003094,1), (1003100,3))
scala>
```

Spark 实现 WordCount，比 Hadoop 要简单的多

```
Scheduler
scala> val rdd = sc.textFile("hdfs://ldcluster:8020/spark/wordcount/buyer_favorite")
rdd: org.apache.spark.rdd.RDD[String] = hdfs://ldcluster:8020/spark/wordcount/buyer_favorite MapPart
itionsRDD[1] at textFile at <console>:24
      Showing 0 to 0 of 0 entries
      No data available in table
scala> rdd.count
res2: Long = 30
      Showing 0 to 0 of 0 entries
      No data available in table
scala> rdd.map(line => line.split('\t')(1)).distinct.collect
res3: Array[String] = Array(1003055, 1003103, 1001679, 1010675, 1003064, 1002061, 1002429, 1003101,
1002427, 1003066, 1010183, 1000481, 1010178, 1003290, 1003326, 1002420, 1003292, 1001368, 1001560, 1
001597, 1003289, 1002422, 1003094, 1003100)
scala>
```

[0] 0:bash- 1:bash 2:bash\* "bd2a17a1c41a" 22:00 12-7月-19

Spark 实现去重，查看哪些商品被收藏

goods\_visit

商品 ID	点击次数
1010037	100
1010102	100
1010152	97
1010178	96
1010280	104

```
scala> val rdd = sc.textFile("hdfs://ldcluster:8020/spark/sort/goods_visit")      No data available in table
rdd: org.apache.spark.rdd.RDD[String] = hdfs://ldcluster:8020/spark/sort/goods_visit MapPartitionsRD
D[1] at textFile at <console>:24
      Showing 0 to 0 of 0 entries
      No data available in table
scala> rdd.map(line => (line.split('\t')(1).toInt, line.split('\t')(0))).sortByKey(true).collect
res0: Array[(Int, String)] = Array((96,1010178), (96,1010603), (97,1010152), (97,1010637), (100,1010
037), (100,1010102), (103,1010320), (104,1010280), (104,1010510))
scala>
```

[0] 0:bash- 1:bash 2:bash\* "bd2a17a1c41a" 22:16 12-7月-19

Spark 对用户记录进行排序，实现按购买数升序排列。

orders

订单ID	订单号	用户ID	下单日期
52304	111215052630	176474	2011-12-15 04:58:21
52303	111215052629	178350	2011-12-15 04:45:31
52302	111215052628	172296	2011-12-15 03:12:23
52301	111215052627	178348	2011-12-15 02:37:32
52300	111215052626	174893	2011-12-15 02:18:56

## order items

明细 ID	订单 ID	商品 ID
252578	52293	1016840
252579	52293	1014040
252580	52294	1014200
252581	52294	1001012
252582	52294	1022245

## Spark 实现两张表的 Join 操作

## 第五节 测试 Spark SQL

Spark SQL 的前身是 Shark，Shark 是伯克利实验室 Spark 生态环境的组件之一，它能运行在 Spark 引擎上，从而使得 SQL 查询的速度得到 10-100 倍的提升，但是，随着 Spark 的发展，由于 Shark 对于 Hive 的太多依赖（如采用 Hive 的语法解析器、查询优化器等等），制约了 Spark 的 One Stack Rule Them All 的既定方针，制约了 Spark 各个组件的相互集成，所以提出了 SparkSQL 项目。SparkSQL 抛弃了原有 Shark 的代码，汲取了 Shark 的一些优点，如内存列存储（In-MemoryColumnarStorage）、Hive 兼容性等，重新开发了 SparkSQL 代码；由于摆脱了对 Hive 的依赖性，SparkSQL 无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便。SQLContext 具体的执行过程如下：

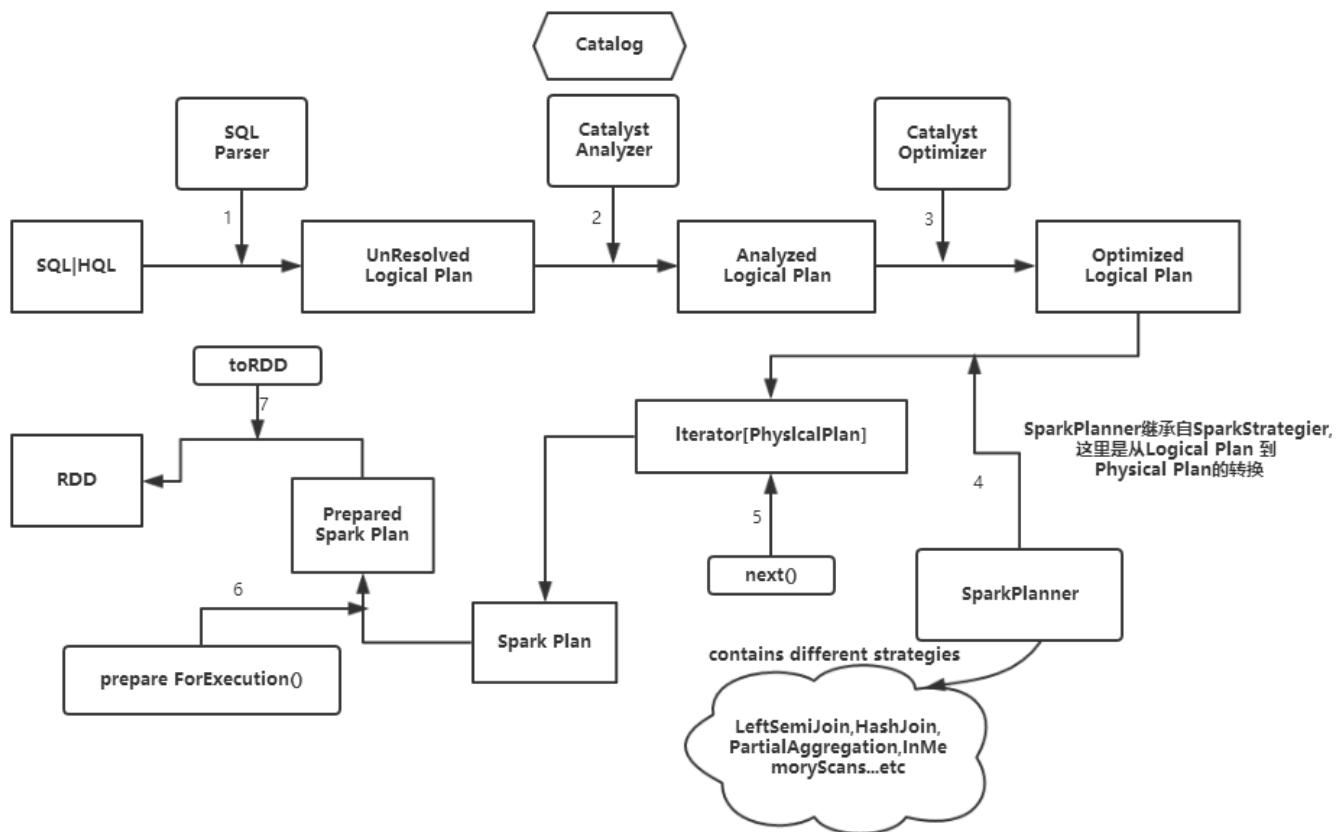


图 8: Spark SQL 执行流程

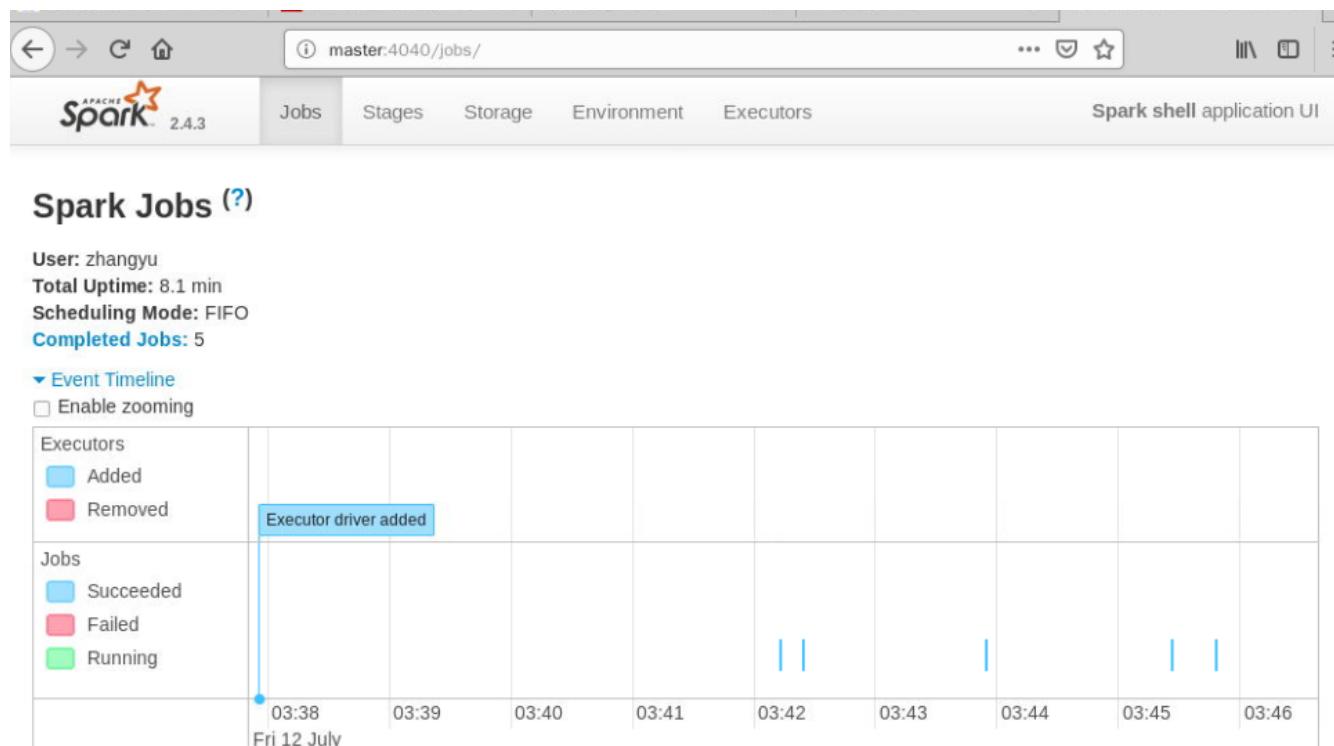
- 1) SQL | HQL 语句经过 SqlParse 解析成 UnresolvedLogicalPlan。
- 2) 使用 analyzer 结合数据字典（catalog）进行绑定，生成 resolvedLogicalPlan，在这个过程中，Catalog 提取出 SchemRDD，并注册类似 case class 的对象，然后把表注册进内存中。
- 3) Analyzed Logical Plan 经过 Catalyst Optimizer 优化器优化处理后，生成 Optimized Logical Plan，该过程完成以后，以下的部分在 Spark core 中完成。

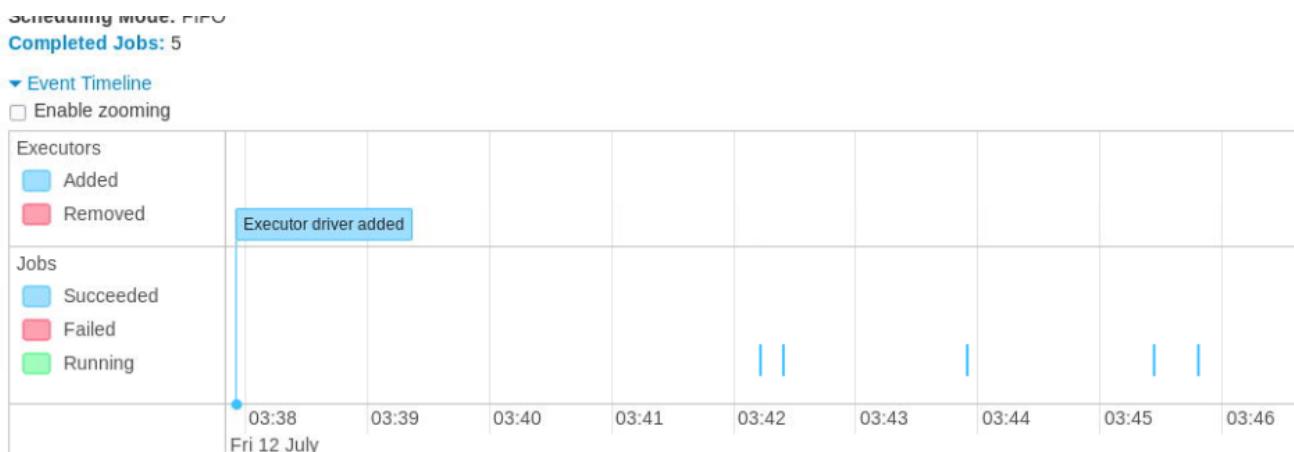
- 4) Optimized Logical Plan 的结果交给 SparkPlanner, 然后 SparkPlanner 处理后交给 PhysicalPlan, 经过该过程后生成 Spark Plan。
- 5) 使用 SparkPlan 将 LogicalPlan 转换成 PhysicalPlan。
- 6) 使用 prepareForExecution() 将 PhysicalPlan 转换成可执行物理计划。
- 7) 使用 execute() 执行可执行物理计划。
- 8) 生成 DataFrame。

在整个运行过程中涉及到多个 SparkSQL 的组件, 如 SqlParse、analyzer、optimizer、SparkPlan 等等。

## 第六节 Spark Web UI

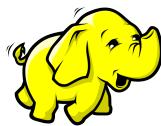
打开浏览器可以通过 <http://master:4040> 端口查看 Spark Job 的运行情况。





#### ▼ Completed Jobs (5)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	collect at <console>:26 collect at <console>:26	2019/07/12 03:45:47	77 ms	2/2	4/4
3	collect at <console>:26 collect at <console>:26	2019/07/12 03:45:26	60 ms	1/1	2/2
2	collect at <console>:26 collect at <console>:26	2019/07/12 03:43:54	0.6 s	2/2	4/4
1	count at <console>:26 count at <console>:26	2019/07/12 03:42:23	91 ms	1/1	2/2



## 0.9. 安装 Flume

### 第一节 简介

在电商、销售等各类系统中，日志是一种非常重要的信息载体，应用会产生大量的日志信息。将不同形式、不同渠道的日志收集起来非常重要且困难。Flume 是一种可靠的分布式日志收集工具。能够通过定义源 Sources、通道 Channels 和目的地 Sinks，方便各种日志信息的收集，汇总到 HDFS 中，方便后续的通过 Mapreduce 或者 Hive、HBase 等的分析处理。

### 第二节 安装配置

Flume 的安装和配置非常的容易，主要是 `flume-env.sh` 和 `flume.conf` 两个文件。

`flume-env.sh`

```
# Environment variables can be set here.
export JAVA_HOME=/opt/java
#FLUME_CLASSPATH=""
```

在 `flume-env.sh` 中写入 `JAVA_HOME` 的位置即可。

Flume 的运行需要通过 `flume.conf` 来指定，下面是 `flume.conf` 的一个配置实例。

#### flume-conf.properties

```
# The configuration file needs to define the sources,
# the channels and the sinks.

# Sources, channels and sinks are defined per agent,
# in this case called 'agent'

agent.sources = seqGenSrc
agent.channels = memoryChannel
agent.sinks = loggerSink

# For each one of the sources, the type is defined
agent.sources.seqGenSrc.type = seq

# The channel can be defined as follows.

agent.sources.seqGenSrc.channels = memoryChannel

# Each sink's type must be defined
agent.sinks.loggerSink.type = logger

#Specify the channel the sink should use
agent.sinks.loggerSink.channel = memoryChannel

# Each channel's type is defined.

agent.channels.memoryChannel.type = memory

# Other config values specific to each type of channel(sink or source
# )

# can be defined as well

# In this case, it specifies the capacity of the memory channel
agent.channels.memoryChannel.capacity = 100
```

在 Flume 的配置文件中，需要指定一个 Agent。Agent 内部包含了 Sources、Channels 和 Sinks。表明了日志数据的流动方向。下面是 Flume 的架构图：

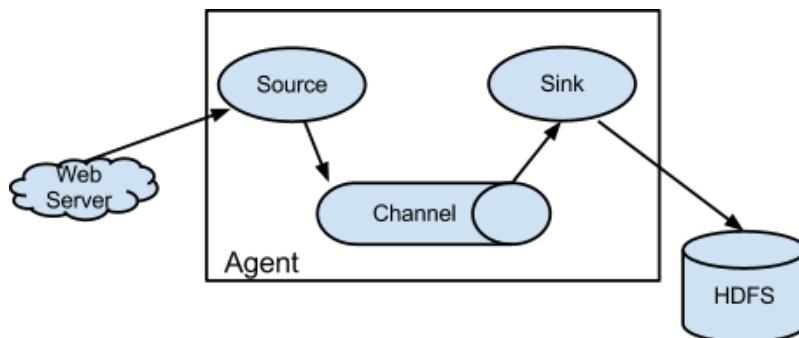


图 9: Flume 的基础架构

在上面这个例子中，指定了 Agent 的名称为agent，Agent 的日志来源是seqGenSrc，Agent 的通道是memoryChannel内存，Agent 的目的地是loggerSink。

注意这里指定的都是名称，在后面.type具体指定类型。

在指定完类型后,agent.sources.seqGenSrc.channels和agent.sinks.loggerSink.channel分别指定了源的通道和目的地的通道，这就像是将源端与目的端连接起来了一样。这里要注意channels和channel的区别，在Flume中，一个源是可以和多个Channel对应的，而一个目的与一个Channel对应。可以在拓扑逻辑上见到它们的区别，这里的配置也表明了这一点。

运行这个例子需要输入以下命令：

启动 Flume 自动的配置模板

```
bin/flume-ng agent -n agent -c conf -f conf/flume-conf.properties
```

其中，-c指明配置文件的目录，在有多个配置文件和需要找到JAVA\_HOME时都需要。-f指定读取那个配置文件，-n表示Agent的名称。

### 第三节 测试和运用

如上节所介绍，Flume的运行主要通过指定配置来指定。Flume的日志来源、通道和目的地都可以很多，灵活搭配使用。下面是一个Example，

example.conf

```
# example.conf: A single-node Flume configuration
# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1
# Describe/configure the source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444
# Describe the sink
a1.sinks.k1.type = logger
# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
# Bind the source and sink to the channel
a1.sources.r1.channels = c1
```

```
a1.sinks.k1.channel = c1
```

上面的配置定义了一个名称为a1的 Agent，监听 44444 端口上的数据作为输入的源、一个在内存中的缓冲作为事件的通道，并且将控制台日志输出作为事件的目的地。通过下面命令启动：

### 启动监听端口的例子

```
bin/flume-ng agent --conf conf --conf-file example.conf --name a1 -Dflume.root.logger=INFO,console
```

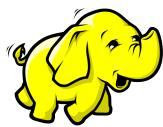
在启动后，新建一个窗口，或者将终端分屏，使用telnet向 44444 端口发送数据，可以检测到 Flume 监听端接收到数据。

The screenshot shows a terminal window with two main sections. The top section displays the Flume agent logs from the command line:

```
Terminal 终端 - zhangyu@bd2a17a1c41a: ~
文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)
tHandler.run(NetcatSource.java:328)] Chars read = 7
2019-07-15 04:25:50,389 (netcat-handler-0) [DEBUG - org.apache.flume.source.NetcatSource$NetcatSocketHandler.run(NetcatSource.java:328)] Events processed = 1
2019-07-15 04:25:50,841 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 48 65 6C 6C 6F 0D
Hello. }
2019-07-15 04:25:56,075 (netcat-handler-0) [DEBUG - org.apache.flume.source.NetcatSource$NetcatSocketHandler.run(NetcatSource.java:328)] Chars read = 13
2019-07-15 04:25:56,875 (netcat-handler-0) [DEBUG - org.apache.flume.source.NetcatSource$NetcatSocketHandler.run(NetcatSource.java:328)] Events processed = 1
2019-07-15 04:25:56,876 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 48 48 6C 6C 6C 77 6F 72 6C 64
0D
HHllllworld. }
2019-07-15 04:26,01,032 (conf-file-poller-0) [DEBUG - org.apache.flume.node.PollingPropertiesFileConfigurationProvider$FileWatcherRunnable.run(PollingPropertiesFileConfigurationProvider.java:131)] Checking file:example.conf for changes
```

The bottom section shows a Telnet session connected to localhost port 44444. The user types "Hello" and "HHllllworld", which are echoed back to the terminal. Red arrows point to the "Hello" and "HHllllworld" inputs, and red boxes highlight the log entries for these events in the Flume logs.

图 10: 监听 Telnet 的例子



## 0.10. 安装 Sqoop

### 第一节 简介

Sqoop 是一种数据导入和导出的工具，方便将关系型数据库（RDBMS）如 MySQL、Oracle 中的数据导入到 HDFS、Hive 或者是 HBase 中，从而可以通过 Mapreduce 或在 Spark 程序进行处理，再将结果导出到 RDBMS 中。也是一个比较重要且实用的工具。

### 第二节 安装配置

sqoop-env.sh

```
#Set path to where bin/hadoop is available
export HADOOP_COMMON_HOME=/opt/hadoop
#Set path to where hadoop-*--core.jar is available
export HADOOP_MAPRED_HOME=/opt/hadoop
#set the path to where bin/hbase is available
export HBASE_HOME=/home/zhangyu/hbase
#Set the path to where bin/hive is available
export HIVE_HOME=/home/zhangyu/hive
export HCATA_HOME=/home/zhangyu/hive/hcatalog
#Set the path for where zookeeper config dir is
export ZOOCFGDIR=/home/zhangyu/zookeeper/conf
```

### 第三节 测试和运用

将配置好的sqoop/文件夹分发到各台机器上，即可开始使用。

可以输入sqoop help查看命令帮助。

sqoop help

```
$ sqoop help
usage: sqoop COMMAND [ARGS]

Available commands:
  codegen           Generate code to interact with database records
  create-hive-table Import a table definition into Hive
```

```
eval           Evaluate a SQL statement and display the results
export         Export an HDFS directory to a database table
help           List available commands
import          Import a table from a database to HDFS
import-all-tables  Import tables from a database to HDFS
import-mainframe   Import mainframe datasets to HDFS
list-databases    List available databases on a server
list-tables       List available tables in a database
version          Display version information
```

See 'sqoop help COMMAND' for information on a specific command.

Sqoop 的基本用法如下：

use-sqoop.sh

```
# 列出数据库
$ sqoop list-databases \
> connect jdbc:mysql://master:3306/ \
> -username root \
> -password password
# 列出数据库中的表
$ sqoop list-tables \
> connect jdbc:mysql://master:3306/hive \
> -username root \
> -password password
# 导入数据到HDFS
$ sqoop import \
> connect jdbc:mysql://master:3306/mysqoop \
> -username root \
> -password password \
> -table student \
> -m 1 \
> -target-dir /mysqoop/test/mystu
# 导入数据到HBase
$ sqoop import \
> connect jdbc:mysql://master:3306/mysqoop \
> -username root \
```

```
> -password password \
> -table student \
> -hbase-create-table \
> -hbase-table mystu \
> -column-family mycf \
> -hbase-row-key id
```

当然也可以用一些命令的别名 `sqoop-toolname`, 这样更加简短, 如 `sqoop-import` 导入数据, `sqoop-export` 导出数据等。

有时在 Shell 中输入长串的命令很容易输错, Sqoop 可以通过加载配置选项文件的方式来设定。

```
$ sqoop --options-file /users/homer/work/import.txt --table TEST
```

import.txt

```
import
--connect
jdbc:mysql://localhost/db
--username
foo
```

上面命令等同于:

```
$ sqoop import --connect jdbc:mysql://localhost/db --username foo --
table TEST
```

可以在 options file 里面加入空行、注释等, 使其可读性更高。

使用 Sqoop 时可能会报一些错误, 主要是 MySQL 数据库连接的错误, 要提前将 `mysql-connector-xxx.jar` 文件拷贝到 `sqoop/lib` 文件夹下, 而且驱动的版本也要合适, 否则也会报错。

在连接到数据库服务器时, 有一些点值得注意:

Sqoop 连接数据库的语法是这样的:

Sqoop 连接数据库的语法

```
$ sqoop import --connect jdbc:mysql://database.example.com/employees
```

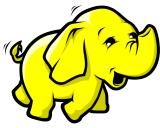
需要注意的是, 最好不要用 `localhost` 这样的地址, 因为在 Sqoop 中, 是一个分布式的数据导入导出工具, 如果指定为 `localhost` 那么每台机器都会在自己的 `localhost` 上连接, 这样必然连接失败, 因此要指定明确的 ip 地址或者是 hostname。可以通过指定 `--username` 和 `--password` 或 `--password-file` 来获取数据库授权。

可以在导入数据时通过 `--query` 指定查询数据库语句来选择性导入表中数据。

### query-example

```
$ sqoop import \
--query 'SELECT a.* , b.* FROM a JOIN b on (a.id == b.id) WHERE
$CONDITIONS' \
-m 1 --target-dir /user/foo/joinresults
```

-m 表明并行执行的 MapReduce 任务数。



## 0.11. 安装 Kafka

### 第一节 简介

Kafka 是一种分布式队列工具，可以与 Flume 结合实现日志的收集和分发，也可以和 Storm 结合使用完成流式信息的处理。需要借助 Zookeeper 协调工作。

Kafka 的作用和设置与 Flume 有很多相似的地方，下面可做一个对比。

- Flume
  - 1. 适合多个生产者
  - 2. 适合与 Hadoop 生态圈对接
  - 3. 适合下游消费者不多的情况
  - 4. 适合数据安全性不高的情况，如内存通道（Memory Channel）
- Kafka
  - 1. 适合下游消费者众多的情况
  - 2. 适合在线的实时日志处理
  - 3. 适合数据安全性较高的情况

常见的一个模型是线上数据 → Flume → Kafka → Flume → HDFS。

### 第二节 配置文件

Kafka 有很多的配置可以添加，但是大部分保持默认即可，因此启动起来也很容易。启动简单的 3 个节点的集群需要配置的文件有：

- `server.properties`，启动服务的基础配置，设置`broker.id`和 ZK 集群信息。
- `producer.properties`，配置 Producer 信息。

- `consumer.properties`, 配置 Consumer 信息。
- `log4j.properties`, 配置日志输出信息。

Kafka 使用了 Zookeeper 来做集群管理, 在启动 Kafka 之前, 要确保启动了 ZK 集群, 如果没有配置 ZK, 也可以使用默认的 ZK 服务, 快速启动。

### 快速启动 Kafka

```
# 启动自带ZK
$ bin/zookeeper-server-start.sh config/zookeeper.properties
# 然后启动Kafka
$ bin/kafka-server-start.sh config/server.properties
# 创建一个Topic
$ bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --
  replication-factor 1 --partitions 1 --topic test
# 查看Topic
$ bin/kafka-topics.sh --list --bootstrap-server localhost:9092
test
# 现在可以发送一些信息
$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
  test
This is a message
This is another message
# 新建一个窗口接受(消费)信息
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --
  topic test --from-beginning
This is a message
This is another message
```

上面的例子就启动了一个单节点的单 Broker 的 Kafka 进程, 并通过了简单的生产者消费者实验测试。在 Kafka 中, 也可以启动多个 Broker 的集群,

```
$ cp config/server.properties config/server-1.properties
$ cp config/server.properties config/server-2.properties
```

在下面两个文件中写入下面信息:

config/server-1.properties

```
broker.id=1
listeners=PLAINTEXT://:9093
log.dirs=/tmp/kafka-logs-1
```

config/server-2.properties

```
broker.id=2
listeners=PLAINTEXT://:9093
log.dirs=/tmp/kafka-logs-2
```

上面配置中的`broker.id`要不同，由于是在单台机器上测试，因此这里设置了不同的监听端口号和日志输出目录，防止各个 broker 覆盖各自的数据。

启动它们：

启动多 Broker 集群

```
$ bin/kafka-server-start.sh config/server-1.properties &
...
$ bin/kafka-server-start.sh config/server-2.properties &
...
# 创建新的 Topic
$ bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --
  replication-factor 3 --partitions 1 --topic my-replicated-topic
```

由于有多个 Broker，使用`--describe`查看各个 Broker 在干什么。

```
$ bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --
  topic my-replicated-topic
Topic:my-replicated-topic    PartitionCount:1    ReplicationFactor:3
  Configs:
    Topic: my-replicated-topic    Partition: 0    Leader: 1
      Replicas: 1,2,0  Isr: 1,2,0
```

关于输出，第一行描述了所有的分区信息，这里的分区数是 1，后面的每行描述了分区的详细信息，因此后面只有 1 行。可以从缩进关系中看出区别。

下面测试在多台机器多个节点上使用 Kafka，这时已经启动了 ZK 集群而不是用自带的。在下列文件中加入下面内容：

server.properties

```
# 注意这里每台机器要不一样
broker.id=0
port=9092
# 这里每台机器也要求是不一样的，要与配置ZK集群中相同
host.name=master
listeners=PLAINTEXT://:9092
log.dirs=/home/zhangyu/kafka/data
```

- `broker.id`, 表示区分 Broker 的 ID, 每台机器都要修改, 要不一样。
- `port`, 表示服务的端口。
- `log.dirs`, 存放 log 日志的目录。需要提前创建。

`producer.properties`和`consumer.properties`文件可以写好之后在启动时指定配置文件的方式, 也不配置, 可以在启动时直接在命令行输入参数。后面演示的是后者。这样更灵活, 但是略显繁琐。生产时应该通过前者来配置。

### 第三节 测试和运用

启动流程:

start-kafka.sh

```
# 启动 Kafka, 需要先启动 ZK 服务, 假设已经启动了
bin/kafka-server-start.sh config/server.properties
# 创建主题 Topic
bin/kafka-topics.sh --create --zookeeper master:2181 \
--replication-factor 1 \
--partitions 1 \
--topic test
# 查看有哪些主题
bin/kafka-topics.sh --list \
--zookeeper master:2181
# 开始生产数据
bin/kafka-console-producer.sh \
--broker-list master:9092 \
--topic test
# 新建另一个窗口开始消费数据
bin/kafka-console-consumer.sh \
--bootstrap-server master:2181 \
--topic test \
--from-beginning
```

注意, 将`kafka`文件夹直接拷贝到不同机器上后, 修改`broker.id`的值后, 每台机器上都需要启动 Kafka 服务。启动之后就可以新建窗口, 进行后面的测试了。

```
(base) zhangyu@bd2a17a1c41a:~$ kafka-topics.sh --list
Exception in thread "main" java.lang.IllegalArgumentException: Only one of --bootstrap-server or --zookeeper must be specified
  at kafka.admin.TopicCommand$TopicCommandOptions.checkArgs(TopicCommand.scala:628)
  at kafka.admin.TopicCommand$.main(TopicCommand.scala:50)
  at kafka.admin.TopicCommand.main(TopicCommand.scala)
(base) zhangyu@bd2a17a1c41a:~$ kafka-topics.sh --list --zookeeper master:2181
--consumer_offsets
my_test Docker: use correct name of packaged 5.2.1 jar in docker-compose.yml
t_test Merge remote-tracking branch 'upstream/5.1.2-post' into 5.2.0-post
test.xml
(base) zhangyu@bd2a17a1c41a:~$ kafka-topics.sh --list --zookeeper master:2181[2019-07-16 06:14:22,02
4] INFO [GroupMetadataManager brokerId=0] Removed 0 expired offsets in 0 milliseconds. (kafka.coordinat
or.group.GroupMetadataManager)
  at org.apache.kafka.common.utils.KafkaFutureImpl.get(KafkaFutureImpl.java:107)
last year
--consumer_offsets set roxworkdir to work around shebang length limit
my_test
t_test
test
(base) zhangyu@bd2a17a1c41a:~$ kafka-topics.sh --list --zookeeper slave1:2181
--consumer_offsets
my_test
t_test
test
(base) zhangyu@bd2a17a1c41a:~$ kafka-topics.sh --describe --zookeeper slave2:2181 --topic my_test
Topic:my_test PartitionCount:2 ReplicationFactor:3 Configs:
  Topic: my_test Partition: 0 Leader: 0      Replicas: 0,2,3 Isr: 0,2,3
  Topic: my_test Partition: 1 Leader: 1      Replicas: 1,2,3 Isr: 1,2,3
(base) zhangyu@bd2a17a1c41a:~$ [0] 0:ssh 1:ssh-M 2:bash* "bd2a17a1c41a" 14:15 16-7月 -19
```

用 `kafka-topics --list` 查看当前有的 Topics。

```
(kafka.admin.TopicCommand$) Examples
(base) zhangyu@bd2a17a1c41a:~$ kafka-topics.sh --describe --zookeeper master:2181 --topic t_test
Topic:t_test PartitionCount:5 ReplicationFactor:2 Configs:
  Topic: t_test Partition: 0 Leader: 1      Replicas: 1,0      Isr: 1,0
  Topic: t_test Partition: 1 Leader: 1      Replicas: 0,1      Isr: 1,0
  Topic: t_test Partition: 2 Leader: 1      Replicas: 1,0      Isr: 1,0
  Topic: t_test Partition: 3 Leader: 1      Replicas: 0,1      Isr: 1,0
  Topic: t_test Partition: 4 Leader: 1      Replicas: 1,0      Isr: 1,0
[0] 0:ssh 1:ssh-M 2:[tmux]* "bd2a17a1c41a" 14:16 16-7月 -19
```

用 `kafka-topics --describe` 查看 Topic 的详细描述。

The screenshot shows a terminal window with the following content:

```

Terminal 终端 - zhangyu@bd2a17a1c41a: ~
文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)
zhangyu@bd2a17a1c41a:~$ cd kafka/
zhangyu@bd2a17a1c41a:~/kafka$ ./bin/kafka-topics.sh --create --topic t_test \
--partitions 5 --replication-factor 2
WARNING: Due to limitations in metric names, topics with a period(.) or underscore(_) could collide. To avoid issues it is best to use either, but not both.
Created topic t_test.
zhangyu@bd2a17a1c41a:~/kafka$ jps
30018 Nimbus
61876 LogViewerServer
38820 Supervisor
773 QuorumPeerMain
26406 Kafka
43479 UI Server
1676 Jps
zhangyu@bd2a17a1c41a:~/kafka$ kafka-console-consumer.sh --bootstrap-server master:2181 --topic t_
>Hello world!
>Nihao
>PERFORMANCE
>zhelishi Beijing
>Beijing Huanyingni
>Kafka
>Hadoop PROJECT INFO
>HHH
>spark SYSTEM
>zookeeper
>storm CLIENTS
[0] 0:bash- 1:java*

```

右侧 pane 显示了 Kafka 的一些高级功能：

- PUBLISH & SUBSCRIBE**: 提供了读写流 (Read and write stream) 和写可伸缩流 (Write scalable stream) 的功能。
- STORE**: 提供了存储流 (Store streams of data) 的功能，描述为适合实时事件处理的应用程序。

底部状态栏显示："bd2a17a1c41a" 18:10 13-7月-19。

可以看到左侧 Producer 产生的数据直接就被右侧 Consumer 消费了。

## 第四节 Java API 实践

Kafka 提供了关于 Consumer 和 Provider 的高级 API 和低级 API，使用高级 API 可以快速启动，但是灵活性较差，而低级 API 比较灵活，应用较多，下面介绍高级 API。使用 Kafka 的 Java API 能够更加灵活地设置处理流程。

首先在 Maven 中加入依赖：

pom 文件中加入下面的依赖

```

<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>${kafka.version}</version>
</dependency>

```

`kafka.version` 中填入对应的 Kafka 版本。

然后编写调用 Kafka 关于生产者和消费者的 API 实现对应逻辑。

## CustomerProducer.java

```
import org.apache.kafka.clients.producer.*;

import java.util.Properties;

public class CustomerProducer {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "master:9092");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer", "org.apache.kafka.common.
            serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.
            serialization.StringSerializer");
        props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, "com.pancm
            .kafka.luod.CustomerPartitioner");

        Producer<String, String> producer = new KafkaProducer<>(props
        );
        for (int i = 0; i < 100; i++) {
            producer.send(new ProducerRecord<String, String>("mytest"
                , Integer.toString(i), Integer.toString(i)), new
                Callback() {
                    @Override
                    public void onCompletion(RecordMetadata metadata,
                        Exception exception) {
                        if (exception != null) {
                            System.out.println("发送成功, partition: " +
                                metadata.partition() + "offset: " +
                                metadata.offset());
                        } else {
                            System.out.println("发送失败");
                        }
                    }
                });
        }
    }
}
```

```
        }
    }
}
producer.close();
}
}
```

CustomerConsumer.java

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.util.*;

public class CustomerConsumer {
    private static Object List;
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "master:9092");
        props.put("group.id", "test");
        props.put("enable.auto.commit", "false");
        props.put("key.deserializer", "org.apache.kafka.common.
            serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.
            serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(
            props);
        /* 用于订阅一个 Topic */
        consumer.subscribe(Collections.singletonList("mytest"));
        /* 用于消费多个 Topics */
        consumer.subscribe(Arrays.asList("foo", "bar"));
        final int minBatchSize = 200;
        List<ConsumerRecord<String, String>> buffer = new ArrayList
```

```

<>();

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.println("record.topic() = " + record.topic()
            () + "---record.offset() = "
            + record.offset() + "--- record.value() = "
            + record.value()
        );
        buffer.add(record);
    }
    if (buffer.size() >= minBatchSize) {
//        insertIntoDb(buffer);
        consumer.commitSync();
        buffer.clear();
    }
}
}
}
}

```

在测试的时候，需要在后台用命令行启动对应的 Consumer 或 Provider 进程，并且要有对应的 Topic。

前面提到 Flume 与 Kafka 的对比和应用场景，下面是一个 Flume 与 Kafka 结合的配置书写。

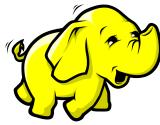
以 Kafka 作为 Flume Sink 的一个应用配置

```

# agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1
# source
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F -c +0 ~/data/flume.log
a1.sources.r1.shell = /bin/bash -c
# sink

```

```
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.bootstrap.servers = master:9092,slave1:9092,slave2
:9092,slave3:9092
a1.sinks.k1.kafka.topic = mytest
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1
# channel
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
# bind source and channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```



## 0.12. 安装 Storm

### 第一节 简介

Storm 是一款实时流处理框架，以实时性为特点，其延迟比 Spark Streaming 还要低，为毫秒级。多与 Kafka 等工具结合起来使用。其安装配置需要搭配 Zookeeper 进行。

Hadoop、Sqark 和 Storm 是 Apache 基金会的三个顶级项目。Storm 与 Hadoop 相比的特点主要是流式计算，而 Hadoop 是批处理作业。Storm 的运行可以没有 HDFS，只要有 Zookeeper 就可以，但是 Storm 也可以和 HDFS 交互，也属于生态系统重要组成。

Storm 的核心组件：

- **Nimbus:** 即 Storm 的 Master，负责资源分配和任务调度。一个 Storm 集群只有一个 Nimbus。
- **Supervisor:** 即 Storm 的 Slave，负责接收 Nimbus 分配的任务，管理所有 Worker，一个 Supervisor 节点中包含多个 Worker 进程。
- **Worker:** 工作进程，每个工作进程中都有多个 Task。
- **Task:** 任务，在 Storm 集群中每个 Spout 和 Bolt 都由若干个任务（tasks）来执行。每个任务都与一个执行线程相对应。
- **Topology:** 计算拓扑，Storm 的拓扑是对实时计算应用逻辑的封装，它的作用与 MapReduce 的任务（Job）很相似，区别在于 MapReduce 的一个 Job 在得到结果之后总会结束，而

拓扑会一直在集群中运行，直到你手动去终止它。拓扑还可以理解成由一系列通过数据流（Stream Grouping）相互关联的 Spout 和 Bolt 组成的的拓扑结构。

- Stream: 数据流（Streams）是 Storm 中最核心的抽象概念。一个数据流指的是在分布式环境中并行创建、处理的一组元组（tuple）的无界序列。数据流可以由一种能够表述数据流中元组的域（fields）的模式来定义。
- Spout: 数据源（Spout）是拓扑中数据流的来源。一般 Spout 会从一个外部的数据源读取元组然后将他们发送到拓扑中。根据需求的不同，Spout 既可以定义为可靠的数据源，也可以定义为不可靠的数据源。一个可靠的 Spout 能够在它发送的元组处理失败时重新发送该元组，以确保所有的元组都能得到正确的处理；相对应的，不可靠的 Spout 就不会在元组发送之后对元组进行任何其他的处理。一个 Spout 可以发送多个数据流。
- Bolt: 拓扑中所有的数据处理均是由 Bolt 完成的。通过数据过滤（filtering）、函数处理（functions）、聚合（aggregations）、联结（joins）、数据库交互等功能，Bolt 几乎能够完成任何一种数据处理需求。一个 Bolt 可以实现简单的数据流转换，而更复杂的数据流变换通常需要使用多个 Bolt 并通过多个步骤完成。
- Stream grouping: 为拓扑中的每个 Bolt 的确定输入数据流是定义一个拓扑的重要环节。数据流分组定义了在 Bolt 的不同任务（tasks）中划分数据流的方式。在 Storm 中有八种内置的数据流分组方式。
- Reliability: 可靠性。Storm 可以通过拓扑来确保每个发送的元组都能得到正确处理。通过跟踪由 Spout 发出的每个元组构成的元组树可以确定元组是否已经完成处理。每个拓扑都有一个“消息延时”参数，如果 Storm 在延时时间内没有检测到元组是否处理完成，就会将该元组标记为处理失败，并会在稍后重新发送该元组。

## 第二节 安装配置

首先需要修改 `storm-env.sh` 文件和 `storm_env.ini` 文件。

storm-env.sh

```
#!/bin/bash
export JAVA_HOME=/opt/java
export STORM_CONF_DIR=/home/zhangyu/storm/conf
```

storm\_env.ini

```
# Environment variables in the following section will be used
# in storm python script. They override the environment variables
# set in the shell.
```

```
[environment]

# The java implementation to use. If JAVA_HOME is not found we expect
# java to be in path
JAVA_HOME:/opt/java

# JVM options to be used in "storm jar" command
#STORM_JAR_JVM_OPTS:
```

在其中主要配置的是 Java 运行环境。更多的配置更主要需要修改[storm.yaml](#)文件。

### storm.yaml

```
#####
# These MUST be filled in for a storm configuration

storm.zookeeper.servers:
  - "master"
  - "slave1"
  - "slave2"

nimbus.seeds: ["slave1", "slave2"]
ui.port: 9090

supervisor.slots.ports:
  - 6700
  - 6701
  - 6702
  - 6703

storm.local.dir: "/home/zhangyu/storm/data"
storm.health.check.dir: "healthchecks"
storm.health.check.timeout.ms: 5000
## Locations of the drpc servers
drpc.servers:
  - "master"
  - "slave1"
  - "slave2"
```

- `storm.zookeeper.servers`配置的是 ZK 服务的集群。这里就填入配置好的 ZK 集群地址，ZK 集群的配置可以见上面。
- `nimbus.seeds`是有机会成为活跃`nimbus`的机器。
- `ui.port`原来的 UI 端口是 8080，这里使用时总是出现端口占用的情况，因此指定为 9090。

- `supervisor.slots.ports` 这个指定的是 supervisor 运行时 slots 可用的端口。这里采用的是官网上推荐的默认配置，指定了 4 个，表明最大会有 4 个 Worker 进程在一台机器上运行，具体的 Worker 数量，可以在代码中指定。
- `storm.local.dir`, 本地存储数据目录，需要提前创建。
- `storm.health.check.dir`, 存储健康检测信息目录。
- `storm.health.check.timeout.ms`, 健康检测时长。
- `drpc.servers`。DRPC (Distribuition Remote Process Call) 分布式远程过程调用服务，指定服务地址。

### 第三节 测试和运用

启动安装配置好后可以启动 Storm。启动顺序如下：

start-storm.sh 启动 storm 的顺序

```
echo "start storm nimbus"
storm nimbus &
echo "start storm health check"
storm node-health-check &
echo "start storm supervisor"
storm supervisor &
echo "start storm web ui"
storm ui &
echo "start storm log viewer"
storm logviewer &
```

注意上面的命令加了`&`，指定其进程在后台运行。需要在不同的节点上都启动，但是活跃的 Nimbus 只有一个，这个由 ZK 完成选取。启动后可以在 Web 浏览器 9090 (依据上面配置) 查看 Storm UI，启动了 LogViewer，可以在 8000 端口查看各个 Nimbus 和 Supervisor 的日志信息。

The screenshot shows the Apache Storm UI interface running in a browser window titled "Storm UI". The address bar indicates the URL is "master:9090". The UI is divided into several sections:

- Cluster Summary:** A table showing cluster statistics:

Version	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
2.0.0	3	0	12	12	0	0
- Nimbus Summary:** A table listing Nimbus hosts and their status:

Host	Port	Status	Version	Uptime
slave2	6627	Leader	2.0.0	3h 8m 6s
master	6627	Not a Leader	2.0.0	8m 25s
slave1	6627	Not a Leader	2.0.0	8m 2s

Showing 1 to 3 of 3 entries
- Owner Summary:** A table showing owner information (empty in this view).

The screenshot shows the Storm UI interface at master:9090. It includes two main sections: 'Topology Summary' and 'Supervisor Summary'.

**Topology Summary:**

- Header: Name, Owner, Status, Uptime, Num workers, Num executors, Num tasks, Replication count, Assigned Mem (MB), Scheduler Info, Topology Version, Storm Version.
- Message: No data available in table.
- Message: Showing 0 to 0 of 0 entries.

**Supervisor Summary:**

- Header: Host, Id, Uptime, Slots, Used slots, Avail slots, Used Mem (MB), Version.
- Data:

Host	Id	Uptime	Slots	Used slots	Avail slots	Used Mem (MB)	Version
master (log)	100b696f-f8a0-4971-9a6e-b8b16c970c4d- 172.100.2.220	4m 57s	4	0	4	0	2.0.0
slave1 (log)	ddafbc5d-1bbc-4e60- a952-8c65652d1358-172.100.5.239	7m 37s	4	0	4	0	2.0.0
slave2 (log)	13ae62a8-7af2-452f-bc11- b2990bb9d502-172.100.3.99	3h 7m 43s	4	0	4	0	2.0.0
- Message: No data available in table.
- Message: Showing 0 to 0 of 0 entries.

Storm UI 上查看很多集群相关有用的信息

```

parableReporter], blacklist.scheduler.resume.time.secs=1800, drpc.childopts=-Xmx768m, nimbus.task.launch.secs=120, logviewer
2019-07-13 06:02:04.162 o.a.s.d.s.Slot main [INFO] SLOT master:6700 Starting in state empty - assignment null
2019-07-13 06:02:04.162 o.a.s.d.s.Slot main [INFO] SLOT master:6701 Starting in state empty - assignment null
2019-07-13 06:02:04.163 o.a.s.d.s.Slot main [INFO] SLOT master:6702 Starting in state empty - assignment null
2019-07-13 06:02:04.163 o.a.s.d.s.Slot main [INFO] SLOT master:6703 Starting in state empty - assignment null
2019-07-13 06:02:04.179 o.a.s.d.s.Supervisor main [INFO] Starting supervisor with id 100b696f-f8a0-4971-9a6e-b8b16c970c4d-17
2019-07-13 06:02:04.184 o.a.s.d.m.ClientMetricsUtils main [INFO] Using statistics reporter plugin:org.apache.storm.daemon.me
2019-07-13 06:02:04.186 o.a.s.d.m.r.JmxPreparableReporter main [INFO] Preparing...
2019-07-13 06:02:04.215 o.a.s.m.StormMetricsRegistry main [INFO] Started statistics report plugin...
2019-07-13 06:02:05.015 o.a.s.u.NimbusClient Thread-4 [INFO] Found leader nimbus : slave2:6627
2019-07-13 06:02:19.446 o.a.s.u.Utils Thread-5 [INFO] Halting after 1 seconds
2019-07-13 06:02:19.448 o.a.s.d.s.Supervisor Thread-6 [INFO] Shutting down supervisor 100b696f-f8a0-4971-9a6e-b8b16c970c4d-17
2019-07-13 06:02:19.449 o.a.s.e.EventManagerImp Thread-4 [INFO] Event manager interrupted
2019-07-13 06:02:19.451 o.a.s.s.o.a.c.f.i.CuratorFrameworkImpl Curator-Framework-0 [INFO] backgroundOperationsLoop exiting
2019-07-13 06:02:19.455 o.a.s.s.o.a.z.ZooKeeper Thread-6 [INFO] Session: 0x302240b9e77000d closed
2019-07-13 06:02:19.455 o.a.s.s.o.a.z.ClientCnxn main-EventThread [INFO] EventThread shut down for session: 0x302240b9e77000
2019-07-13 06:02:30.336 o.a.s.v.ConfigValidation main [WARN] task.heartbeat.frequency.secs is a deprecated config please see
2019-07-13 06:02:30.500 o.a.s.v.ConfigValidation main [WARN] task.heartbeat.frequency.secs is a deprecated config please see
2019-07-13 06:02:30.593 o.a.s.s.o.a.c.u.Compatibility main [INFO] Running in ZooKeeper 3.4.x compatibility mode
2019-07-13 06:02:30.594 o.a.s.s.o.a.c.u.Compatibility main [INFO] Using emulated InjectSessionExpiration
2019-07-13 06:02:30.634 o.a.s.z.ClientZookeeper main [INFO] Starting ZK Curator

```

点击 log 查看各个节点的 log 信息，需要启动 logviewer 服务

## 第四节 Java 编程实例

Storm 以实时流计算为特色，能够在很大程度上弥补 MapReduce 作业在实时性上的缺点和不足。利用 Storm 提供的 Java API，可以自定义数据源 Spout 和数据处理点 Bolt。下面分析是一个利用 Storm 官方提供的 Word Count Example 编写的 WordCount 例子。

编写 Storm 程序，主要需要定义 Spout、Bolt 和 Topology。Spout 的定义一般是实现 [IRichSpout](#) 接口或继承 [BaseRichSpout](#) 类。对应 Bolt 类一般是实现 [IRichBolt](#) 接口或继承 [BaseRichBolt](#) 类。两种都是可以的，后者更为简单一些，是对前者的一层封装。

下面是用于随机生成句子的 Spout 类：

RandomSentenceSpout.java

```

package io.storm.spout;

import org.apache.storm.spout.SpoutOutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichSpout;

```

```
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Map;
import java.util.Random;

public class RandomSentenceSpout extends BaseRichSpout {
    private static final Logger LOG = LoggerFactory.getLogger(
        RandomSentenceSpout.class);
    SpoutOutputCollector collector;
    Random rand;

    @Override
    public void open(Map<String, Object> conf, TopologyContext
        context, SpoutOutputCollector collector) {
        this.collector = collector;
        rand = new Random();
    }

    @Override
    public void nextTuple() {
        String[] sentences = new String[]{
            sentence("the cow jumped over the moon"),
            sentence("an apple a day keeps the doctor away"),
            sentence("four score and seven years ago"),
            sentence("snow white and the seven dwarfs"),
            sentence("i am at two with nature")
        };
        final String sentence = sentences[rand.nextInt(sentences.
            length)];
        LOG.debug("Emitting tuple: {}", sentence);
    }
}
```

```
        collector.emit(new Values(sentence));
    }

protected String sentence(String input) {
    return input;
}

@Override
public void ack(Object id) {
}

@Override
public void fail(Object id) {
}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word"));
}

// Add unique identifier to each tuple, which is helpful for
// debugging
public static class TimeStamped extends RandomSentenceSpout {
    private final String prefix;

    public TimeStamped() {
        this("");
    }

    public TimeStamped(String prefix) {
        this.prefix = prefix;
    }

    @Override
    protected String sentence(String input) {
```

```
@Override
public void nextTuple() {
    String[] sentences = new String[]{
        sentence("the cow jumped over the moon"),
        sentence("an apple a day keeps the doctor
away"),
        sentence("four score and seven years ago"),
        sentence("snow white and the seven dwarfs")
    }, sentence("i am at two with nature")
};

final String sentence = sentences[rand.nextInt(sentences.
length)];

LOG.debug("Emitting tuple: {}", sentence);

collector.emit(new Values(sentence));
}
```

`nextTuple()`是其中处理业务的核心方法。该方法会被放在一个无限运行的循环中，不断地通过`emit`发送出数据，就像是“水龙头”一样。这也是 Storm 与 MapReduce 程序的区别，在定义好拓扑后，Storm 是不断运行的。

其他几个必须要实现的方法中（主要继承至[IRichSout](#)）分别有对应的功能。

下面是定义拓扑结构的类以及两个处理节点 `SplitSentence` 和 `WordCount` 都简单起见作为静态类写在了内部，分别用来将句子分割为单词串和将单词进行词频统计。

## WordCountApp.java

```
package io.storm;  
  
import io.storm.spout.RandomSentenceSpout;
```

```
import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.StormSubmitter;
import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.*;
import org.apache.storm.topology.base.BaseBasicBolt;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;

import java.util.HashMap;
import java.util.Map;

public class WordCountApp {
    public static void main(String[] args) throws Exception {
        int waitTime = 100;
        String topologyName = "Word-Count";
        Config conf = new Config();

        WordCountApp app = new WordCountApp();
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new RandomSentenceSpout(), 3);
        builder.setBolt("split", new SplitSentence(), 8).
            shuffleGrouping("spout");
        builder.setBolt("count", new WordCount(), 12).fieldsGrouping(
            "split", new Fields("word"));

        if (args != null && args.length > 0) {
            topologyName = args[0];
            StormSubmitter.submitTopology(topologyName, conf, builder
                .createTopology());
        } else {
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology(topologyName, conf, builder.
                createTopology());
        }
    }
}
```

```
        Thread.sleep(1000 * waitTime);
        cluster.killTopology(topologyName);
        cluster.shutdown();
    }
}

public static class SplitSentence implements IRichBolt {

    private OutputCollector collector = null;

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
        return null;
    }

    @Override
    public void prepare(Map<String, Object> topoConf,
                        TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }

    @Override
    public void execute(Tuple input) {
        String sentence = input.getStringByField("word");
        String[] words = sentence.split(" ");
        for(String word: words) {
            this.collector.emit(new Values(word));
        }
    }

    @Override
    public void cleanup() {
```

```
    }

}

public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple, BasicOutputCollector
        collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null) {
            count = 0;
        }
        count++;
        counts.put(word, count);

        System.out.println("-----Count Content
-----");
        for(String key:counts.keySet()) {
            System.out.println("key: " + key + " -> " + "count: "
                + counts.get(key));
        }
        System.out.println("-----Content     end
-----");
        collector.emit(new Values(word, count));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer
        ) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

`execute`是实现 Bolt 逻辑的主要方法,`prepare`用于在处理之前做一些准备,`declareOutputFields`

声明输出的Field。

```
// 将句子进行分词
@Override
public void execute(Tuple input) {
    String sentence = input.getStringByField("word");
    String[] words = sentence.split(" ");
    for(String word: words) {
        this.collector.emit(new Values(word));
    }
}

// 实现统计词频
@Override
public void execute(Tuple tuple, BasicOutputCollector collector) {
    String word = tuple.getString(0);
    Integer count = counts.get(word);
    if (count == null) {
        count = 0;
    }
    count++;
    counts.put(word, count);
    collector.emit(new Values(word, count));
}
```

注意里面设置了集群模式提交和本地模式运行的判断：

```
if (args != null && args.length > 0) {
    topologyName = args[0];
    StormSubmitter.submitTopology(topologyName, conf, builder.
        createTopology());
} else {
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology(topologyName, conf, builder.
        createTopology());
    Thread.sleep(1000 * waitTime);
    cluster.killTopology(topologyName);
    cluster.shutdown();
}
```

本地模式需要设置运行的结束时间，在这里不是无限循环运行的。

下面代码用于设置拓扑结构、分组策略以及并发程度，这些会影响作业的执行情况。

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 3);
builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split",
    new Fields("word"));
```

在本地运行的结果如下：

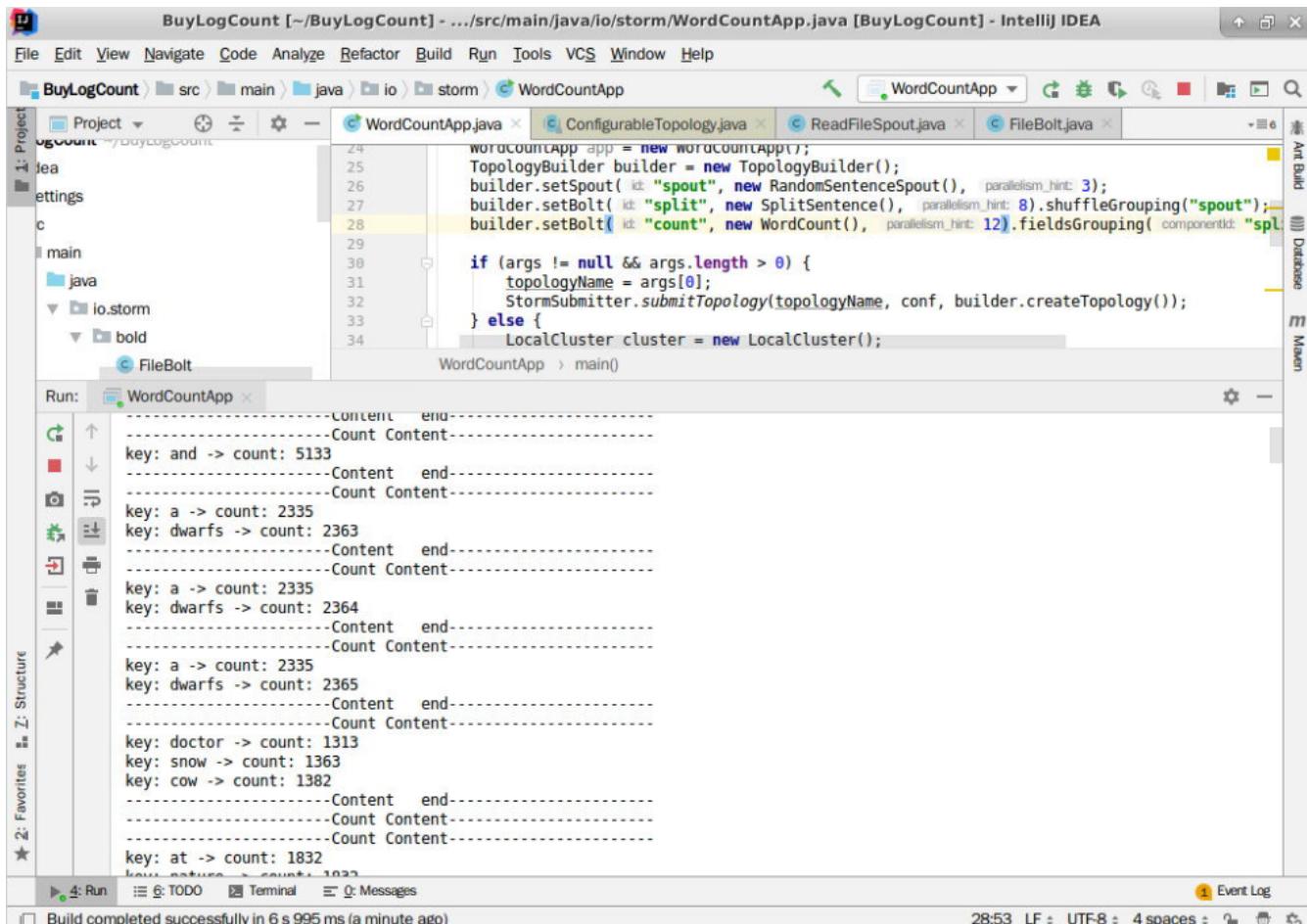


图 11: 词频统计程序本地运行情况

还可以通过下面命令将 Jar 包提交到集群中进行运行。

#### 提交 Storm 集群 Jar 包的命令

```
# arg1: jar包名 arg2: 主函数入口, 要拷贝全路径引用 arg3: 运行的拓扑名
storm jar BuyLogCount-1.0-SNAPSHOT.jar io.storm.WordCountApp
WordCount
```

提交集群运行后，可以在 UI 界面中查看相关拓扑运行情况。

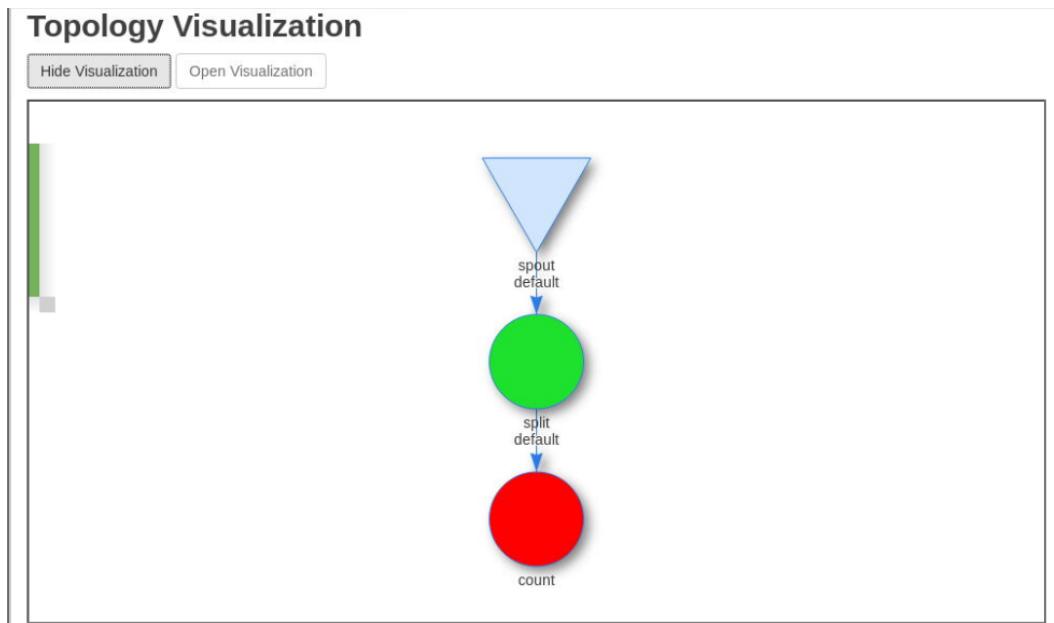


图 12: 查看可视化拓扑结构，该例子比较简单，只有一个 Spout 和两个顺序的 Bolt

## Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	2749020	1629060	0		
3h 0m 0s	2749020	1629060	0		
1d 0h 0m 0s	2749020	1629060	0		
All time	2749020	1629060	0		

## Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked		Failed		Error Host	Error Port	Last error	Error Time
						Acked	Failed	Acked	Failed				
spout 3	3	443520	443520	0	0	0	0	0	0				

Showing 1 to 1 of 1 entries

## Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host		Error Port		Last error		Error Time	
											Acked	Failed	Error Host	Error Port	Last error	Error Time		
count 12	12	12	1119960	0	1.067	3.876	1119980	3.877	1120000	0								
split 8	8	8	1185540	1185540	0.003	0.029	185220	0.000	0	0								

查看拓扑运行 Worker 进程、数据压力等统计信息