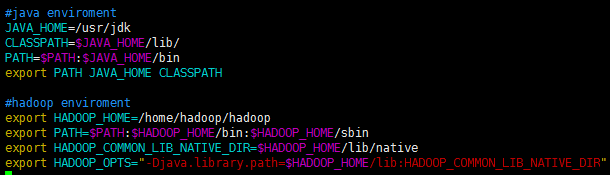
**Hadoop学习笔记**

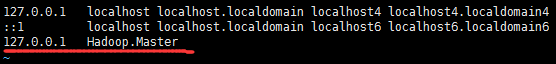
Hadoop环境搭建

由于之前没有记录，简单贴出来配置文件，看一下吧。

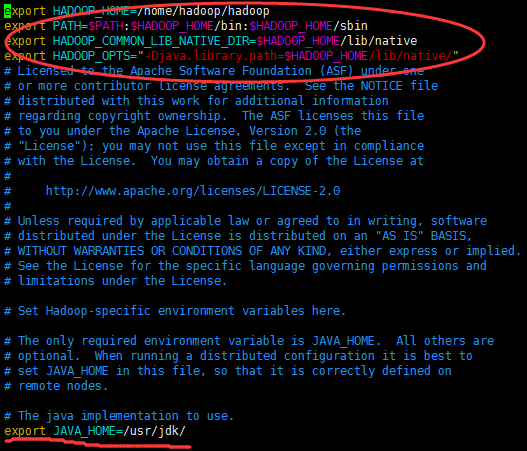
首先。编辑 /etc/profile 配置jdk环境以及hadoop环境



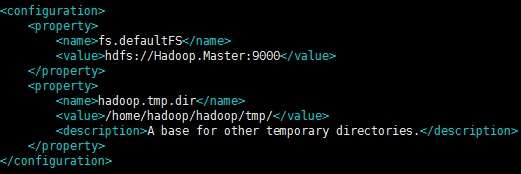
然后，配置一下hostname



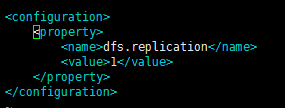
第一个配置文件,hadoop目录下etc/**hadoop-env.sh**



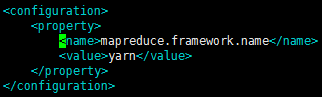
第二个配置文件core-site.xml。**第一个property配置的是HDFS的NameNode的地址，第二个property配置的内容用来指定Hadoop运行时产生的文件的存放目录。**



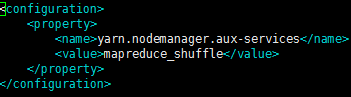
第三个配置文件hdfs-site.xml。**该配置文件用来指定HDFS保存数据副本的数量（现在是伪分布式，所以数量是1，将来的集群副本数量默认是3）。**



第四个配置文件mapred-site.xml。我们在hadoop目录下发现文件列表中只有mapred-site.xml.template而没有mapred-site.xml，因此我们需要先把mapred-site.xml.template的后缀.template去掉（即重命名）



第五个配置文件yarn-site.xml。**需要说明的是，配置的内容是NodeManager获取数据的方式shuffle。**



然后我们需要输入命令格式化我们的hdfs

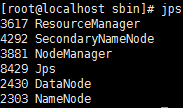
hdfs namenode -format

进入sbin目录下，启动dfs和yarn

./start-dfs.sh

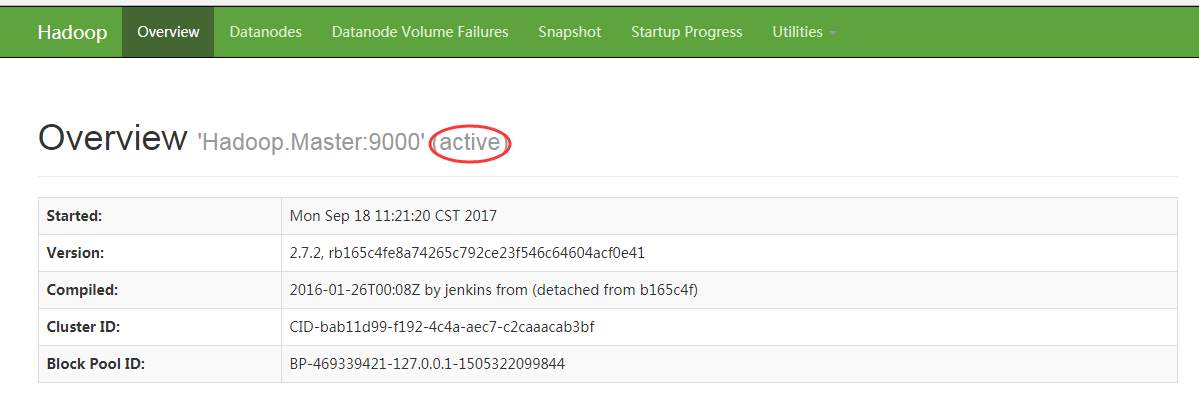
./start-yarn.sh

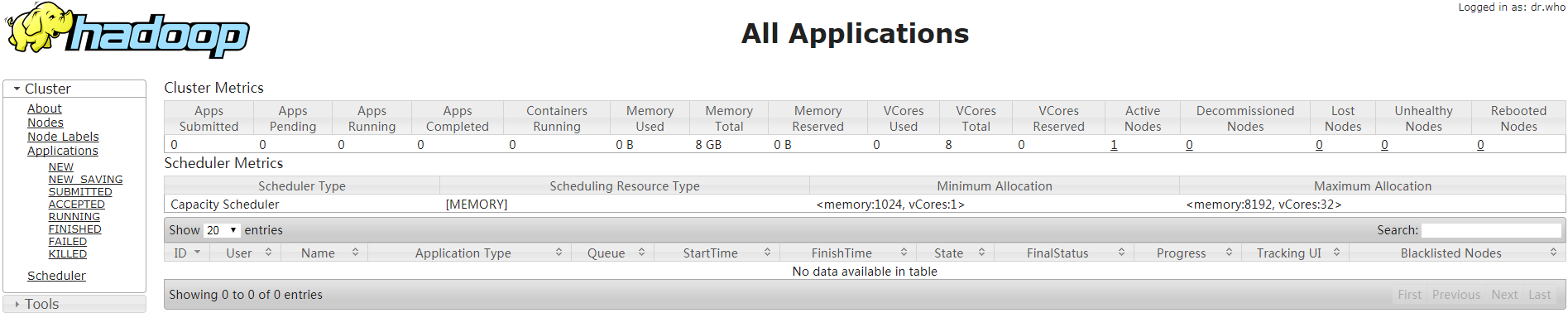
然后，用jps 命令查看启动的进程数，应该为6个才是正常



http://192.168.8.88:50070     (hdfs管理界面)

http://192.168.8.88:8088       (yarn管理界面)

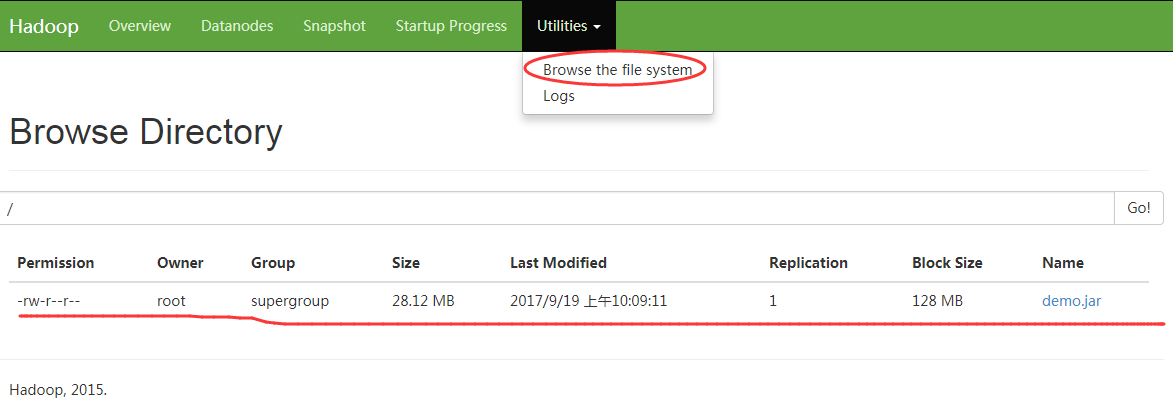




上传一个文件

格式：hadoop fs -put < local file > < hdfs file >

hadoop fs -put /home/testhu/demo.jar hdfs://Hadoop.Master:9000/demo.jar



查看hdfs文件命令

hadoop fs -ls hdfs://Hadoop.Master:9000/



**注：hdfs://Hadoop.Master:9000/代表hdfs的根路径，就是core-site.xml配置文件中配置的。**

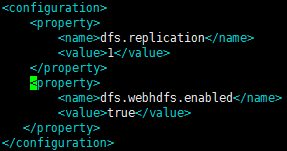
下载命令：

hadoop fs -get hdfs://Hadoop.Master:9000/demo.jar

问题记录：

背景：HDFS 页面中File information无法下载文件。

原因：hdfs-site.xml必须将dfs.webhdfs.enabled属性设置为true，才能使用webhdfs的一些功能。于是修改配置文件



停止hdfs 以及 yarn

stop-dfs.sh

stop-yarn.sh

然后重新启动hdfs以及yarn，此时出现问题，namenode无法启动。

查看hadoop-root-namenode-localhost.localdomain.log 这个namenode的日志发现报错。

java.io.IOException: There appears to be a gap in the edit log. We expected txid 1, but got txid 17.

网上说是元数据被破坏了，我就停止然后在启动元数据就坏了？搞不懂。

这个命令能解决namenode无法启动的问题：hadoop namenode -recover

但是不用还好一用，又来问题了。

datanode又无法启动了

查看datanode的日志hadoop-root-datanode-localhost.localdomain.log 发现报错：

2017-09-19 14:29:56,688 WARN org.apache.hadoop.hdfs.server.common.Storage: java.io.IOException: Incompatible clusterIDs in /home/hadoop/hadoop/tmp/dfs/data: namenode clusterID = CID-26df84e6-a33b-4c02-8f1f-a40042ab1b89; datanode clusterID = CID-bab11d99-f192-4c4a-aec7-c2caaacab3bf

2017-09-19 14:29:56,692 FATAL org.apache.hadoop.hdfs.server.datanode.DataNode: Initialization failed for Block pool <registering> (Datanode Uuid unassigned) service to Hadoop.Master/127.0.0.1:9000. Exiting.

java.io.IOException: All specified directories are failed to load.

…

….

然后又是百度一下。。。

网友给出的答案：

**原因是因为datanode的clusterID 和 namenode的clusterID 不匹配。**

**打开hdfs-site.xml里配置的datanode和namenode对应的目录，分别打开current文件夹里的VERSION，可以看到clusterID项正如日志里记录的一样，确实不一致，修改datanode里VERSION文件的clusterID 与namenode里的一致，再重新启动dfs（执行start-dfs.sh）再执行jps命令可以看到datanode已正常启动。**

**出现该问题的原因：在第一次格式化dfs后，启动并使用了hadoop，后来又重新执行了格式化命令（hdfs namenode -format)，这时namenode的clusterID会重新生成，而datanode的clusterID 保持不变。**

所以，解决办法就是修改VERSION文件中的clusterID

文件的位置在tmp/dfs/data/current/VERSION

然后修改成与namenode的clusterID一样就行了

namenode,datenode都启动了，然后又来问题了：

HDFS的主页面上显示There are 1 missing blocks. The following files may be corrupted

我修改配置之前上传的文件丢失了。

bin/hadoop fsck /

bin/hadoop dfs -rm /demo.jar  
bin/hadoop fsck /

只能删除了。。。

然后一切回归平静

好多疑问，我觉得和我用hadoop namenode -recover 这个命令有关。

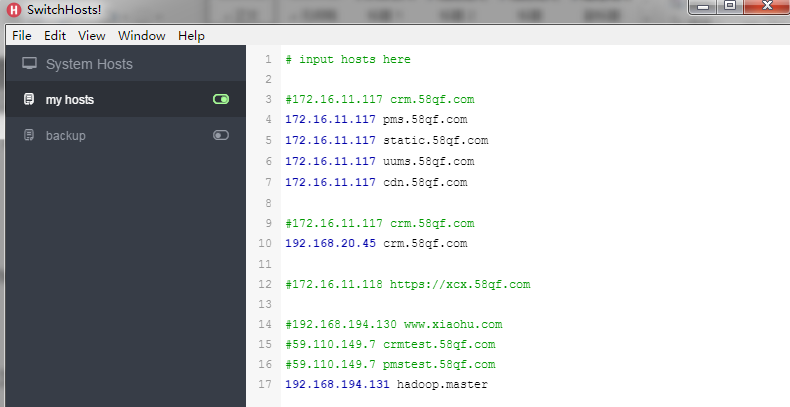
问：

重启一下为啥namenode 就不能启动了？每次namenode启动都要格式化么？

hadoop namenode -recover这个做了些什么？

继续之前的hadoop学习，之前的hadoop的hdfs页面下载文件下载不了，原因是他所跳转的网址是带有host标签的（hadoop.master），而我们访问的机器无法解析这个host标签，也就访问不了。

所以我们需要在访问的机器上配置host（hadoop.master）指向我们的hadoop服务。

由于我本机装了SwitchHosts 所以直接在这上面改了。

配置好之后就可以使用hdfs的web端页面的下载了。

Hadoop教学贴：

<http://blog.csdn.net/u012453843/article/category/6887358/3>

互联网电商项目 淘淘商城从无到有教学贴：

<http://blog.csdn.net/u012453843/article/category/6970308/8>

简单的java下载hdfs中的文件

首先，IDEA的POM文件引入下列的包。

<dependency>  
 <groupId>org.apache.hadoop</groupId>  
 <artifactId>hadoop-common</artifactId>  
 <version>2.7.2</version>  
</dependency>  
<dependency>  
 <groupId>org.apache.hadoop</groupId>  
 <artifactId>hadoop-hdfs</artifactId>  
 <version>2.7.2</version>  
</dependency>

下载文件这两个包就够了，

然后java代码如下：  
FileSystem fs = FileSystem.get(new URI("hdfs://Hadoop.Master:9000/"),new Configuration());  
InputStream is = fs.open(new Path("/demo.jar"));  
OutputStream os = new FileOutputStream("C:\\Users\\Administrator\\Desktop\\dd\\demo.jar");  
IOUtils.copyBytes(is, os, 4096, true);

上面的代码是要下载我hdfs根路径下，上传的一个jar包。

下面是向hdfs上传文件的java代码。

FileSystem fs = FileSystem.get(new URI("hdfs://Hadoop.Master:9000/"),new Configuration(),"root");  
InputStream is = new FileInputStream("C:\\Users\\Administrator\\Desktop\\周报.txt");  
OutputStream os = fs.create(new Path("/test.txt"));  
IOUtils.copyBytes(is, os, 4096, true);

上面的代码是将我桌面上的一个周报文本文件上传到hdfs的根路径下，并重命名为test.txt。

下面是删除代码，将hdfs中的demo.jar删除

/\*删除文件代码\*/  
//第一个参数是要删除的文件或者文件夹  
//第二个参数是是否递归删除，用于删除文件夹以及文件夹下所有文件  
FileSystem fs = FileSystem.get(new URI("hdfs://Hadoop.Master:9000/"),new Configuration(),"root");

Boolean flag = fs.delete(new Path("/demo.jar"),false);  
System.out.println(flag);

下面是创建文件夹代码

/\*创建文件夹\*/

FileSystem fs = FileSystem.get(new URI("hdfs://Hadoop.Master:9000/"),new Configuration(),"root");  
Boolean sign = fs.mkdirs(new Path("/dirTest"));  
System.out.println(sign);

后续的Hadoop知识有则补充，基本都以：

<http://blog.csdn.net/u012453843/article/category/6887358/3>

此教学贴为准

## Hadoop远程调试

简单做一下阐述，首先创建一个maven项目，并引入相关的jar包，以一个最基本的例子为准pom.xml文件如下：

<dependencies>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-common</artifactId>

<version>2.7.1</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-hdfs</artifactId>

<version>2.7.1</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

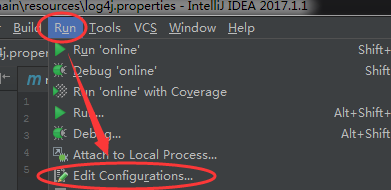
<version>2.7.1</version>

</dependency>

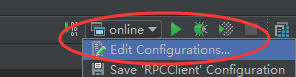
</dependencies>

然后我们项目里必须的一个配置是log4j.properties。

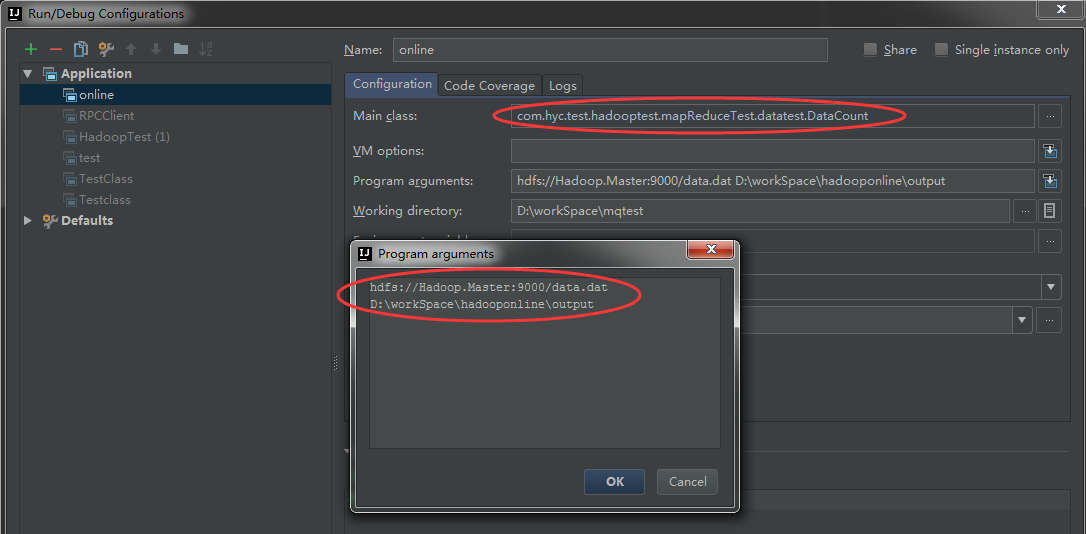
然后，我们需要配置项目的启动参数，点击IDEA的



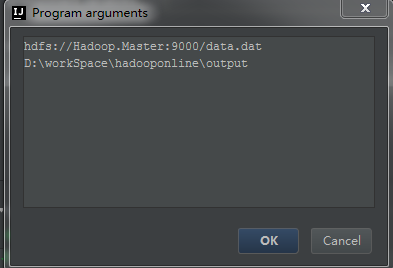
或者右上角



配置如下



第一个是启动main方法所在的class的文件，其中：



这里面的配置第一行配置是文件的输入目录，或者要输入的文件目录，第二行设置的是hadoop经过mapreduce解析后的输出路径。

然后，我们需要在main函数的第一行添加如下代码，用来配置hadoop远程调试的必要程序winutils.exe的所在根目录

System.setProperty("hadoop.home.dir", "C:\\Users\\Administrator\\Desktop\\hadoop-common-bin");

接下来就可以编写hadoop代码，进行远程调试了。

## Hadoop集群搭建

首先，在有jdk的基础上进行搭建，我们准备搭建一个namenode，两个datanado

vim /etc/profile

添加

#hadoop enviroment

export HADOOP\_HOME=/opt/soft/hadoop/hadoop-2.7.2

export PATH=$PATH:$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin

export HADOOP\_COMMON\_LIB\_NATIVE\_DIR=$HADOOP\_HOME/lib/native

export HADOOP\_OPTS="-Djava.library.path=$HADOOP\_HOME/lib:HADOOP\_COMMON\_LIB\_NATIVE\_DIR"

之后，source /etc/profile 使其生效

然后配置host ，vim /etc/hosts

192.168.194.131 Hadoop.Master

192.168.194.129 Hadoop.Slave1

192.168.194.130 Hadoop.Slave2

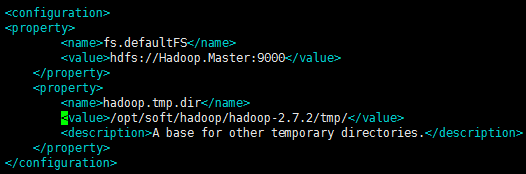
service network restart

然后我们将hadoop包解压，并开始修改配置文件hadoop-env.sh、core-site.xml、mapred-site.xml、hdfs-site.xml、yarn-site.xml、slaves

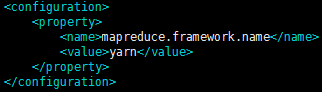
hadoop-env.sh



core-site.xml



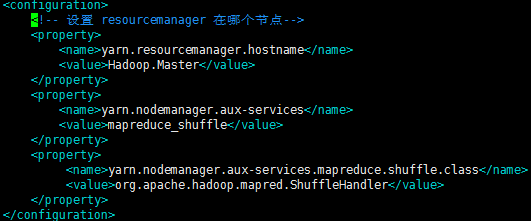
mapred-site.xml



hdfs-site.xml



yarn-site.xml



vim slaves

Hadoop.Slave1

Hadoop.Slave2

然后，将该hadoop，以及hosts文件、profile文件复制到另外两个linux服务器中。

配置主节点机免密码登录两个从节点机。。。（此处省略）

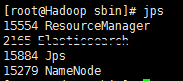
然后接下来就是namenode初始化以及启动主节点：

hadoop namenode -format

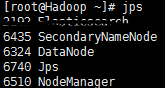
sbin/start-dfs.sh

sbin/start-yarn.sh

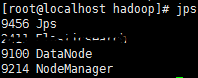
master



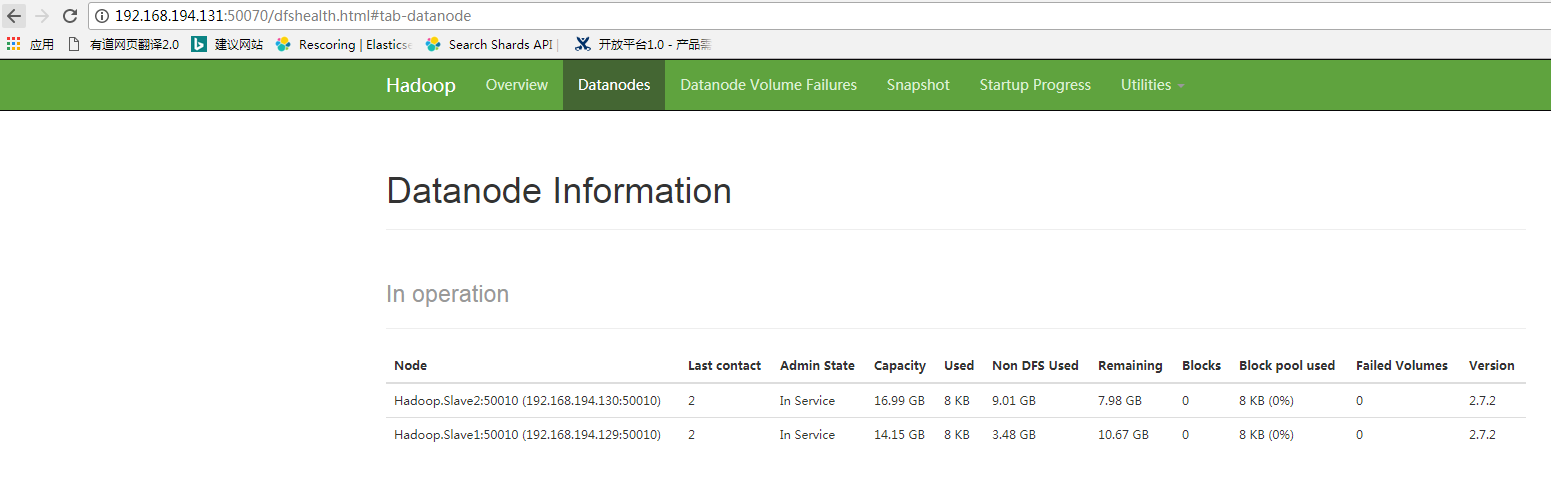
Slave1



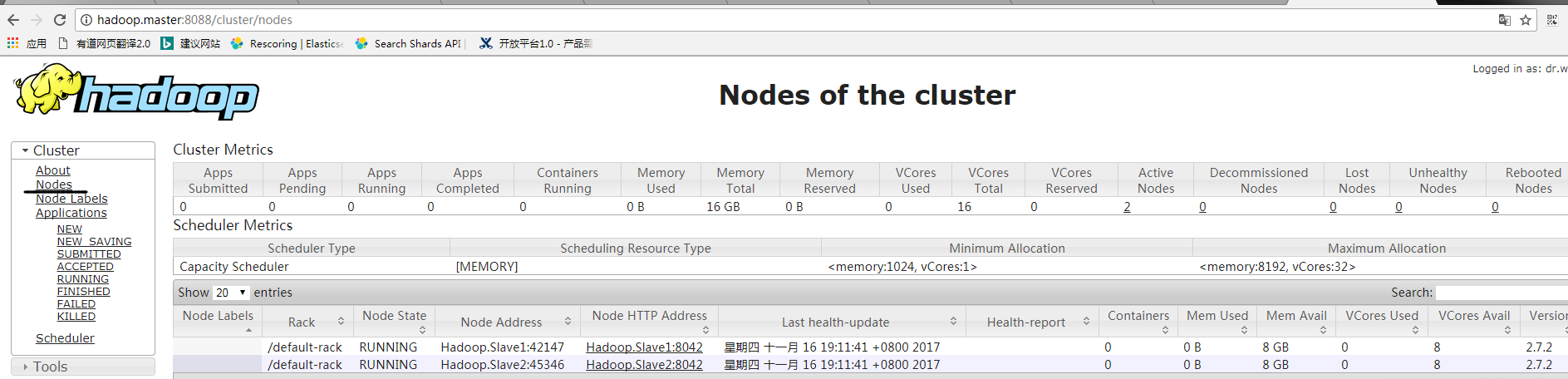
Slave2



<http://192.168.194.131:50070>



http:// 192.168.194.131:8088



Hadoop MapReduce结果分区：

使用场景：说一下为什么要用到分区的功能，这里我们举个例子，  
中国移动公司想要查看北京用户的打电话情况，手机信号来自附近的基站，要查看北京用户的信息得从全国所有的基站获取信息并一一筛选，假如我们不把用户按  
省市进行分别存放的话，每次我们想查看某个省市的信息时便需要从全国所有的基站信息中去一一查询，这样做，无疑效率是非常低的。假如我们把数据都分省市  
进行存放了，以后我们再想查看北京市的打电话信息便非常方便了，直接到存放北京市打电话信息的文件中查找就可以了。

Partitioner类是负责实现分区情况规则的就Java类，实现分区需要继承Partitioner并实现重写getPartition方法，写完内部类后，我们还需要在main方法中把ProviderPartitioner加载到Job当中，要不然Hadoop不知道你定义了一个内部类，也不会去执行它。我们还需要设置Reducer的数量。

**注意：这里需要说明的是，如果我们指定Reducer的数量少于Partitioner的数量，这样就会报错。但是如果我们设置的Reducer的数量多于Partitioner的数量的话，不会出错，只是会生成多余的空文件。因此我们的原则是宁可多起Reducer也不能少起，否则会有问题。**

具体例子参考mqtest中的hadooptest。

Hadoop 多次MapReduce操作：

使用场景：针对一组数据要做计算并排序，此时需要分开两次MapReduce操作，第一次MapReduce为求和操作，第二次MapReduce为针对第一次的结果进行排序操作。

具体用例参考mqtest中的sumAndSortest。

Hadoop Combiner编程

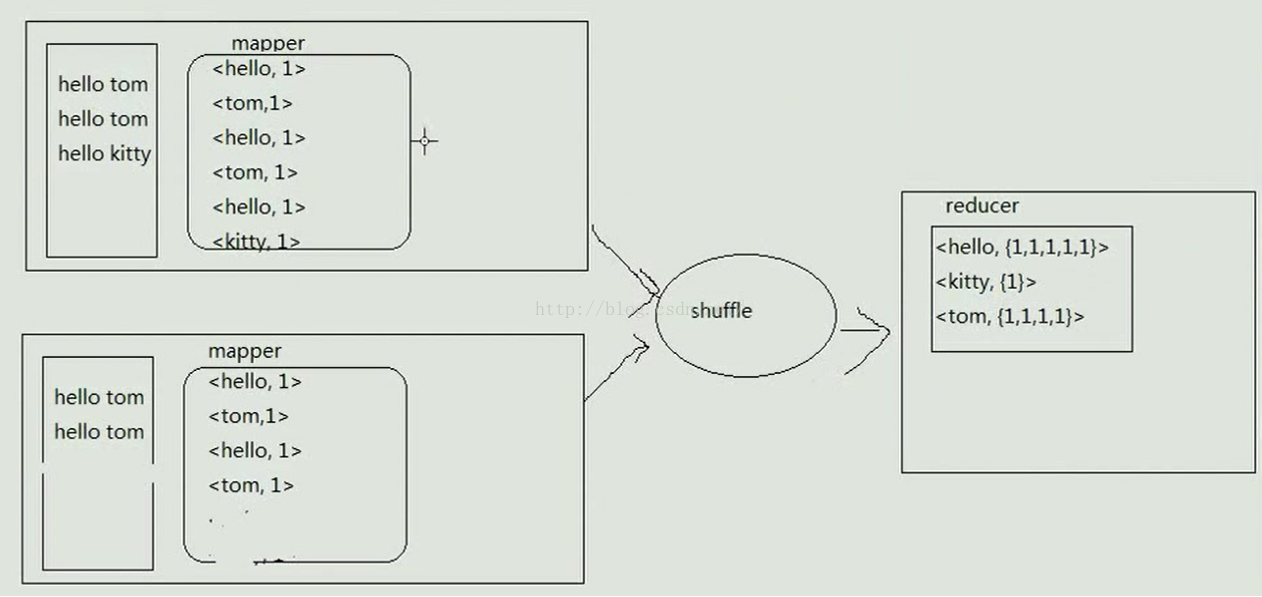
这节我们一起学习Combiner编程，每一个map可能会产生大量的输出，Combiner的作用就是在map端对输出先合并一次，以减少传输到Reducer的数据量。

       combiner最基本是实现本地key的归并，combiner具有类似本地reduce功能。

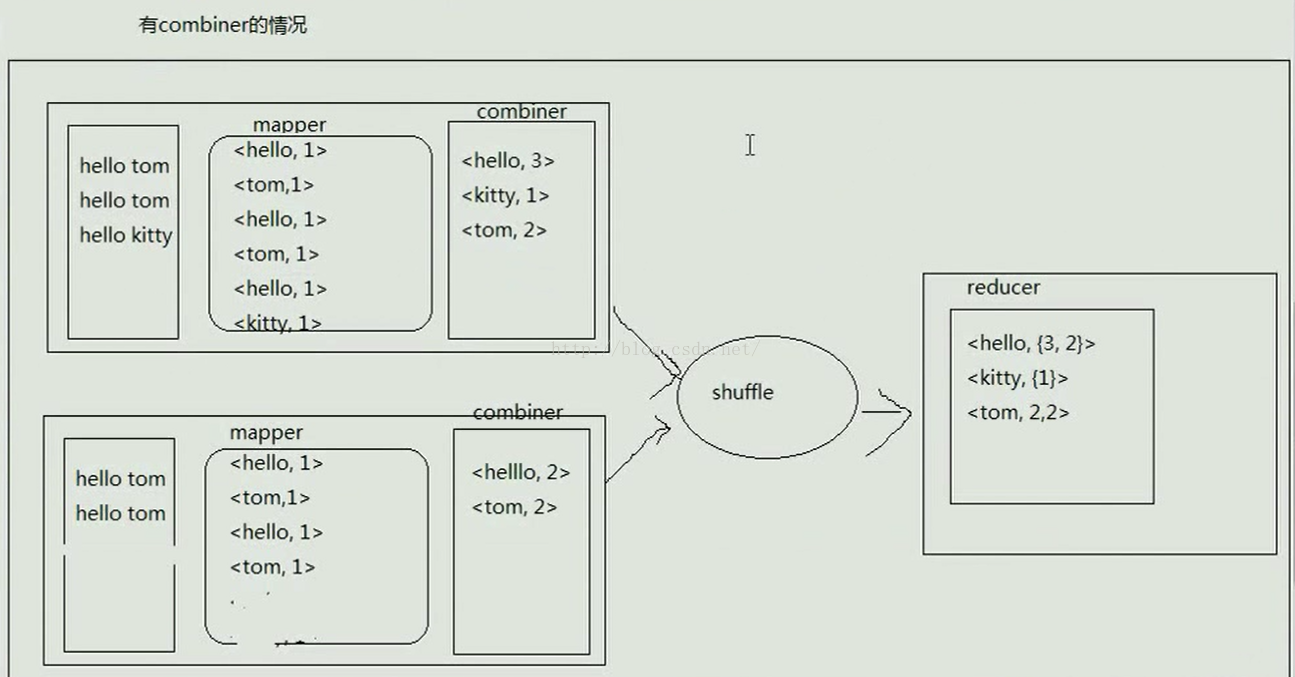
       如果不用combiner，那么所有的结果都是reduce完成，效率会相对低下，使用combiner，先完成的map会在本地聚合，提升速度。

       注意：Combiner的输出是Reducer的输入，如果Combiner是可插拔的，添加Combiner决不能改变最终的计算结果，在这种情况下Combiner只适用于那种Reduce的输入key/value和Reduce的输出key/value类型完全一致，且不影响最终结果的场景，比如累加、最大值。当然，Combiner也可以是不可插拔的，也就是说有与没有Combiner最终的计算结果是不一样的，不可插拔的应用场景是数据的筛选，比如我们可以在Map执行之后，Reduce执行之前先对数据进行筛选，把坏数据给过滤掉，这样传到Reduce的数据就都是好的数据了，这样最终的结果肯定是与没有筛选数据的结果是不一样的。

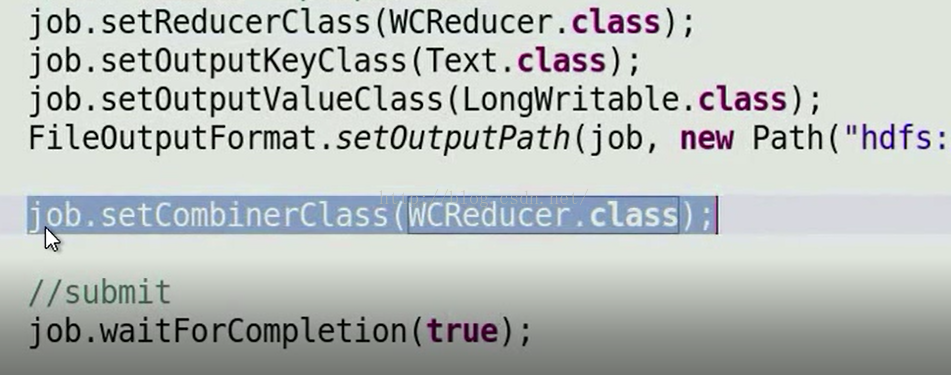
      上面说了一堆理论，可能大家还不理解Combiner的作用具体是什么，下面我们画个图来更好的理解Combiner做了什么。我们先来看一张没有Combiner的MapReduce工作原理，比如有两个mapper，第一个mapper读取的文件中有两个hello tom和一个hello kitty，mapper处理完后，数据会被处理成一个一个的map，同理，第二个mapper读取的文件中有两个hello tom，经mapper处理后也变成了一个一个的map，两个mapper处理完之后把结果给shuffle处理，经shuffle处理后结果变为<hello,{1,1,1,1,1}>、<kitty,{1}>、<tom,{1,1,1,1}>，然后交给reducer处理，如下图所示



        接下来我们再来看一副有Combiner的图片，如下图所示，我们看到Combiner存在的位置是在mapper端，它是在mapper执行完之后进行的，combiner会先对数据合并一下，第一个mapper读取的文件中hello出现3次，kitty出现1次，tom出现2次，因此经combiner处理后的结果是<hello,3>、<kitty,1>、<tom,2>，第二个mapper读取的文件中有两个hello和两个tom，经combiner处理之后的结果是<hello,2>、<tom,2>，然后把这些数据交给shuffle处理，shuffle处理后的结果是<hello,{3,2}>、<kitty,{1}>、<tom,{2,2}>，然后把这些数据交给reducer处理。而且我们可以看到经过Combiner处理后最终的结果并没有变化，这就是可插拔的。



        那么我们怎么使用Combiner呢，如下图所示，我们在Job中设置CombinerClass（job.setCombinerClass(WCReeducer.class)），其实Combiner是特殊的Reducer，因此如果是可插拔的话，我们可以把Reducer作为Combiner，如果不可插拔，我们可以定义一个Reducer来处理我们特殊的逻辑，然后在job中进行配置。



**所以所谓的Combiner实际上是一种在reduce之前操作的reduce（参考前面的图），从代码中看到其设定的操作类也是一个Reduce操作类。**

具体用例看mqtest 的wordcount用例。

Hadoop 生态圈 Sqoop安装与使用：

本人使用的Hadoop版本是2.7.2 对应的sqoop-1.4.6

Sqoop的运行需要Yarn和HDFS，它只需要知道Yarn的管理者角色ResourceManager和HDFS的管理者角色NameNode在哪台设备上就可以，我们在搭建集群的时候每台设备上都有相同的6个配置文件，因此我们把Sqoop放到任何一台设备上都可以。

那么，我们用不用配置Sqoop呢，如果你的环境已经配置了NameNode和ResourceManager在哪台设备上，那么我们不用配置任何内容便可以使用Sqoop，现在我们的集群环境中core-site.xml和hdfs-site.xml两个文件指定了NameNode在哪两台设备上，yarn-site.xml指定了ResourceManager在哪台设备上，因此我们不用作任何配置了。

    那么问题又来了，Sqoop是怎么知道我们集群的配置呢，其实Sqoop会自动去查找这台设备上HADOOP\_HOME的位置，找到之后它会自动读取hadoop下面的配置文件，从而它也就知道了NameNode和ResourceManager的位置。

将下载的sqoop的tar包解压 我们解压到的目录是/opt/soft/hadoop/sqoop-1.4.6

然后，配置下/etc/project

#sqoop enviroument

export SQOOP\_HOME=/opt/soft/hadoop/sqoop-1.4.6

export PATH=$PATH:$SQOOP\_HOME/bin

然后执行命令：source /etc/project

如果我们想与MySql做交互需要瞎子一个MySql的驱动包mysql-connector-java-5.1.40-bin.jar放到sqoop的lib目录下

启动命令：进入到sqoop的bin目录下 执行 ./sqoop

需要注意的是：我们的sqoop想直接访问MySql是不可以的，我们需要修改Mysql的Ip访问权限。

解决方法如下：

mysql -u root -p   
mysql>use mysql;   
mysql>select host from user where user='root';   
mysql>update user set host = '%' where user ='root';   
mysql>flush privileges;   
mysql>select host   from user where user='root';  
第一句是以权限用户root登录   
第二句：选择mysql库   
第三句：查看mysql库中的user表的host值（即可进行连接访问的主机/IP名称）   
第四句：修改host值（以通配符%的内容增加主机/IP地址），当然也可以直接增加IP地址   
第五句：刷新MySQL的系统权限相关表   
第六句：再重新查看user表时，有修改。。   
重起mysql服务即可完成。

好了执行完上面的所有操作，我们可以和mysql做交互了，Sqoop的基本全表导入导出语句如下

导出MySql到hdfs：

sqoop import --connect jdbc:mysql://192.168.20.143:3306/world --username root --password 111111 --table country --target-dir /world/country -m 1 -fields-terminated-by '\t'

导入MySql从hdfs：

./sqoop export --connect jdbc:mysql://192.168.20.143:3306/world --username root --password 111111 --export-dir /world/country -m 1 --table country\_copy --fields-terminated-by '\t'

**注：sqoop在导数据到hadoop，hive中的时候，如果sqoop的查询中存在相同名字的字段，将会导出失败，报错。所以在创建hadoop相关的表的时候尽量名称不要重复。**

列出几个基本参数：

-connect mysql链接URL

-username mysql登录用户名

-password mysql密码

-table 操作的表

-target-dir hdfs保存操作目录

-export-dir hdfs导出读取操作目录

-m mapreduce执行数量

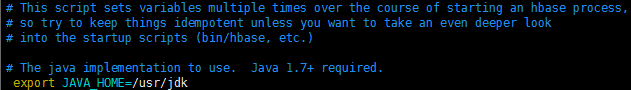
-fields-terminated-by 数据分隔符设置

Hadoop Hbase单节点安装：

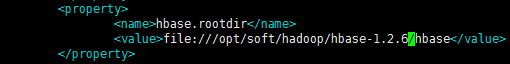
本人使用的Hadoop版本为2.7.2 对应的可用Hbase版本为hbase-1.2.6 ，安装过程很简单，首先下载hbase的tar包，上传你的服务器并解压。

然后，进入hbase 的conf目录下，修改两个配置文件，分别是hbase-env.sh和hbase-site.xml

hbase-env.sh需要配置JDK的所在目录：



hbase-site.xml 需要配置文件系统位置：我将它配置到我的hbase目录下的一个文件夹中



然后通过命令启动hbase：./bin/start-hbase.sh

然后通过jps命令查看是否有hbase进程：



说明启动成功，然后我们检查下hbase是否可用：

执行./hbaseshell命令：开启hbase命令行交互界面，输入help看是否有打印。。。

Hadoop Hbase集群搭建（非高可用）：

环境：hadoop-2.7.2 hbase-1.2.6 Centos7 linux服务器三台

对应的host：

192.168.194.131 Hadoop.Master （主节点）

192.168.194.129 Hadoop.Slave1 （从节点1）

192.168.194.130 Hadoop.Slave2 （从节点2）

首先，我们需要配置Hbase的环境变量vim /etc/profile 配置完成后source /etc/profile

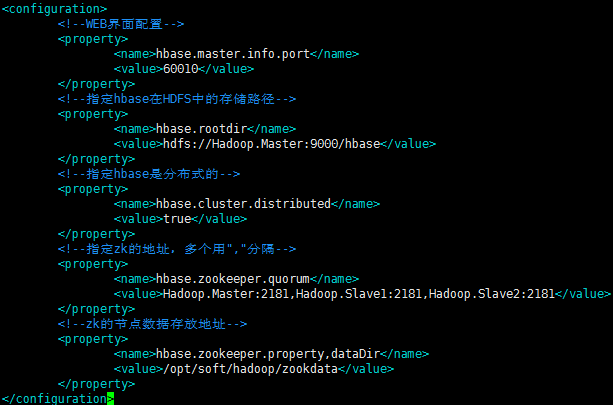
然后，进入到hbase的conf目录下，配置hbase-env.sh，如果不是高可用，只需要配置JDK就好了



如果是配置高可用的hbase，则需要将下面的配置行的值改为false，停止hbase使用自己的zookeeper，改为用自己的zookeeper集群。



然后，我们配置第二个配置文件hbase-site.xml，其中zk配置我们zk所在的服务器地址，由于我们不是高可用配置，并且用的是hbase自带的zk，所以这里我们配置三台虚拟机的地址即可



第三个配置文件，配置regionservers，用来添加被管理者的hbase的host



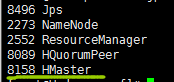
这里我们有两个从节点，将从节点配置为被管理者。

然后将hbase复制到另外两台服务器上，路径保持一致。

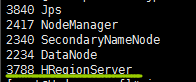
将三台机器上的hadoop目录中的etc/hadoop/下的两个文件core-site.xml、hdfs-site.xml复制到hbase的conf目录下。

最后，执行主节点上的启动脚本 ./bin/start-hbase.sh 。

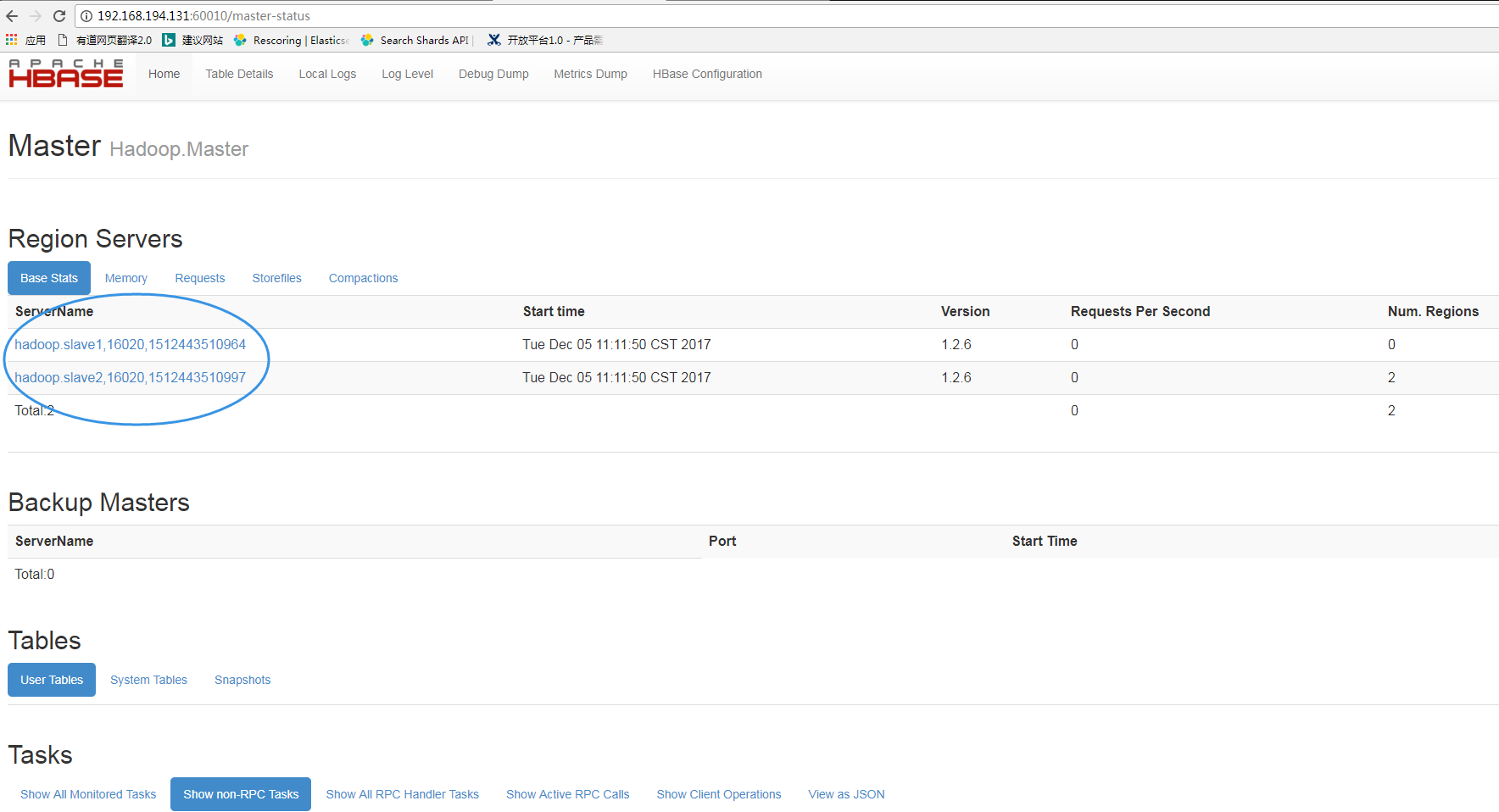
通过jps观察启动情况，主节点：



从节点：



访问主节点IP+60010端口，查看hbase管理界面。



Hadoop Hive单机安装：

环境：hadoop-2.7.2 hbase-1.2.6 hive-2.2.0

首先我们将hive的tar包上传我们的服务器，并解压。

然后我们需要做以下几个步骤：

1. 配置hive的环境变量
2. 配置hive-env.sh
3. 配置hive-site.xml
4. 添加MySql链接驱动包
5. Hive初始化MySql数据库
6. 启动检查

环境变量的配置就不详细说了，修改/etc/profile文件。

然后我们配置hive-env.sh，进入到hive的conf目录下面，我们可以用他的模板，也可以自己新建一个文件。模板是hive-env.sh.template 可以直接改名字用。

hive-env.sh配置如下：



接下来我们需要配置hive-site.xml，他的模板文件为hive-default.xml.template，直接改名，在里面做修改。

在修改hive-site.xml之前，我们需要去hadoop中创建几个hive要用到的目录。

bin/hadoop fs -mkdir -p /user/hive/warehouse

bin/hadoop fs -mkdir -p /user/hive/tmp

bin/hadoop fs -mkdir -p /user/hive/log

bin/hadoop fs -chmod -R 777 /user/hive/warehouse

bin/hadoop fs -chmod -R 777 /user/hive/tmp

bin/hadoop fs -chmod -R 777 /user/hive/log

然后我们要修改hive-site.xml：

下面是关于hive的配置

<property>

<name>hive.metastore.warehouse.dir</name>

<value>/user/hive/warehouse</value>

<description>location of default database for the warehouse</description>

</property>

<property>

<property>

<name>hive.exec.scratchdir</name>

<value>/user/hive/tmp</value>

</property>

<property>

<name>hive.querylog.location</name>

<value>/user/hive/log/hadoop</value>

<description>Location of Hive run time structured log file</description>

</property>

下面是关于MySql的修改

<property>

<name>javax.jdo.option.ConnectionURL</name>

<value>jdbc:mysql://192.168.20.143:3306/hive?characterEncoding=utf8&amp;useSSL=true&amp;createDatabaseIfNotExist=true</value>

<description>

JDBC connect string for a JDBC metastore.

To use SSL to encrypt/authenticate the connection, provide database-specific SSL flag in the connection URL.

For example, jdbc:postgresql://myhost/db?ssl=true for postgres database.

</description>

</property>

<property>

<name>javax.jdo.option.ConnectionDriverName</name>

<value>com.mysql.jdbc.Driver</value>

<description>Driver class name for a JDBC metastore</description>

</property>

<property>

<name>javax.jdo.option.ConnectionUserName</name>

<value>root</value>

<description>Username to use against metastore database</description>

</property>

<property>

<name>javax.jdo.option.ConnectionPassword</name>

<value>111111</value>

<description>password to use against metastore database</description>

</property>

在hive中创建一个tmp目录 /opt/soft/hadoop/hive-2.2.0/tmp

然后，把{system:java.io.tmpdir} 改成 /opt/soft/hadoop/hive-2.2.0/tmp

把 {system:user.name} 改成 {user.name}，hive-site.xml配置完成。

将MySql的jdbc驱动包上传到hive的lib目录下面。

接下来，一切准备好了，我们来初始化hive的MySql库

./bin/schematool -initSchema -dbType mysql

有如下输出，表示初始化成功

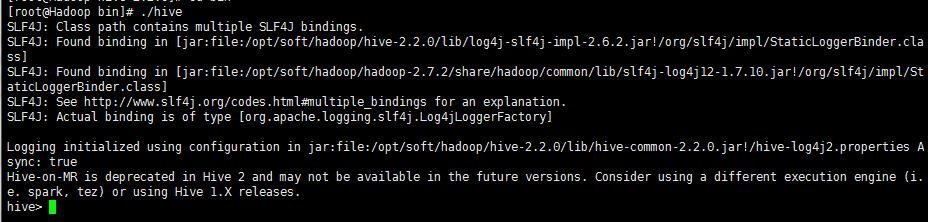
Starting metastore schema initialization to 2.1.0

Initialization script hive-schema-2.1.0.mysql.sql

Initialization script completed

schemaTool completed

最后，我们在hive的bin目录下执行hive命令启动hive，并进入hive的命令行。



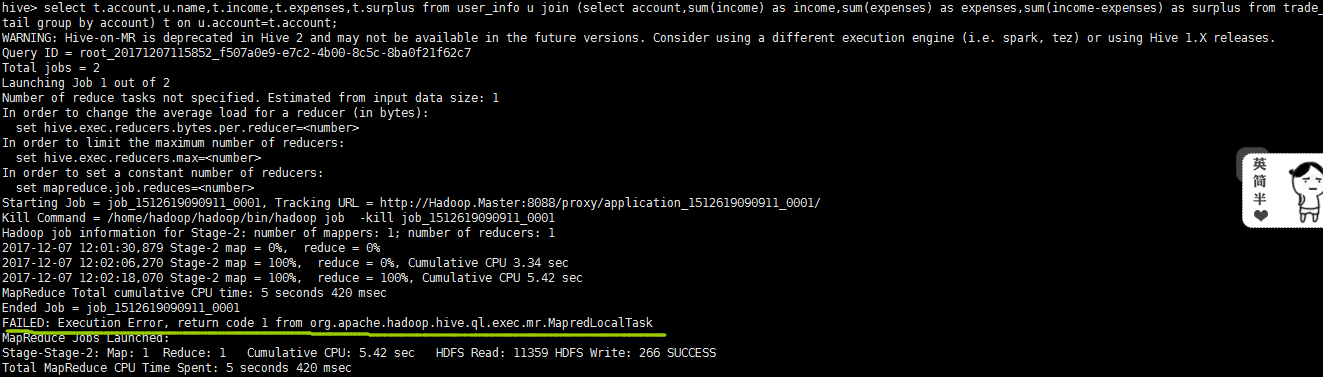
**注：hive中也有类似hadoop的分区的概念。**

Hive语法：

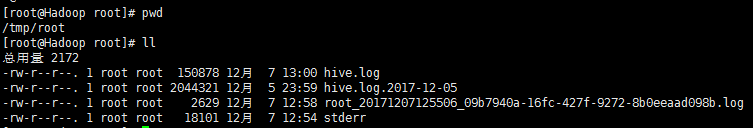
http://blog.csdn.net/hguisu/article/details/7256833

遇到的坑：

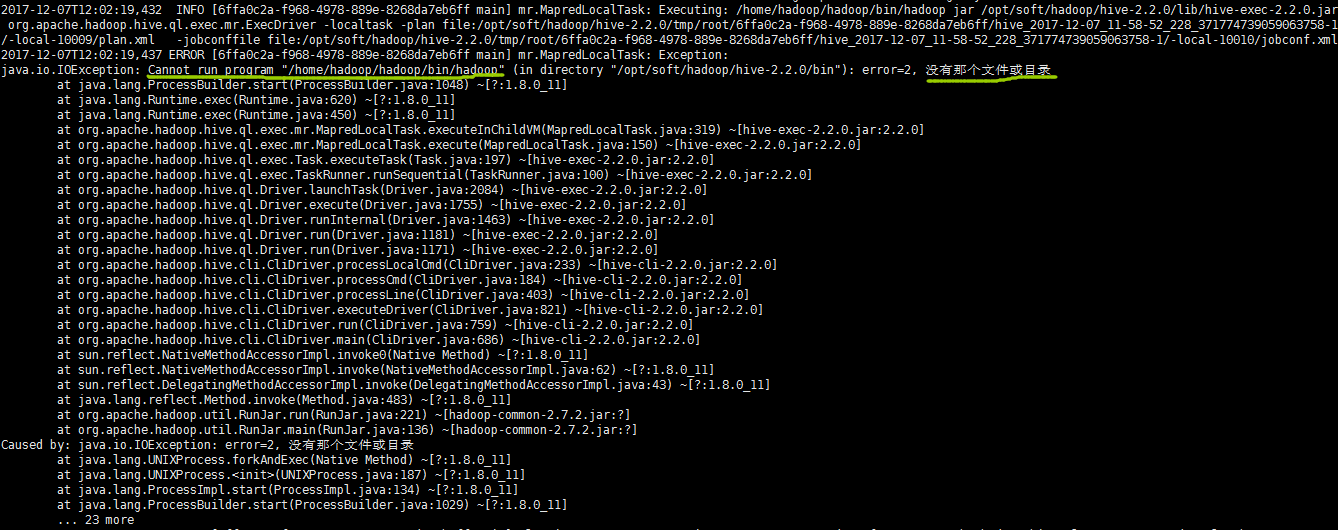
在执行hive的连表查询的时候，执行失败。



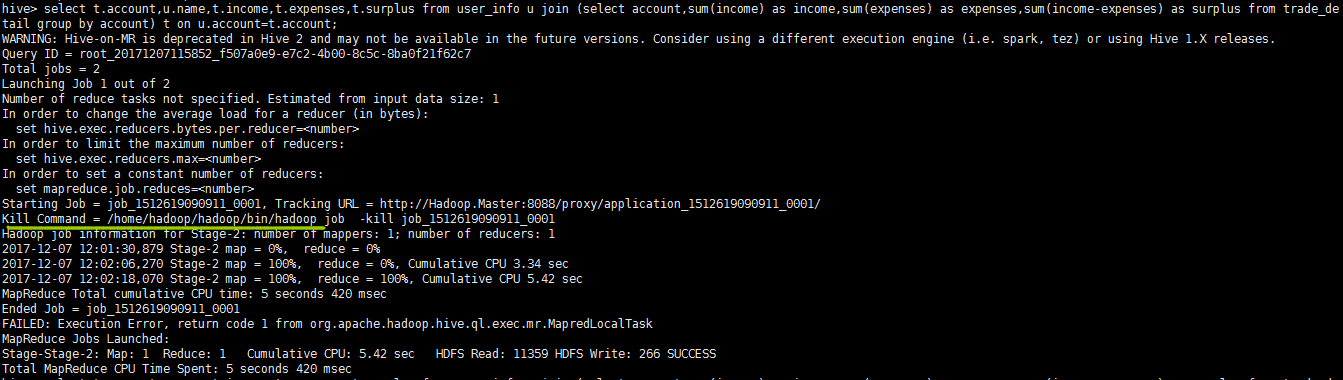
然后我们去查看hive的执行日志，hive的Log默认记录在/tmp/{用户名} 下



我们查看hive.log 有日下错误打印



并且我们在执行查询的时候，发现hive取得hadoop目录有问题



经过一番排查，发现我们的hadoop-env.sh配置的有问题：其中还有我刚开始学习hadoop的时候配置的错误的HADOOP\_HOME，正是这里的hadoop环境配置成了/home/hadoop/hadoop 所以我们将其注掉，然后执行source hadoop-env.sh 最后重启hadoop再次执行hive的连表查询的时候，就会正常了。

经过这次的问题，我发现了hive在取hadoop环境变量的时候，最后生效的既不是hive-env.sh中的HADOOP\_HOME，也不是/etc/profile中的HADOOP\_HOME，而是hadoop-env.sh中的HADOOP\_HOME（如果有配置）。

**注：如果修改了hadoop-env.sh文件，需要执行source hadoop-env.sh，然后重启hadoop。**

**Hive部分功能点：**

**1.Hive可以通过编写Java程序，来自定义我们的字段解析函数，也就是自定义UDF。**

**2.Hive可以直接读取HDFS中的文件，你需要做的只有新建了对应hdfs文件字段的表并且location对应着指定的hdfs目录，就能通过hive查表直接查询到hdfs文件中的内容。**

**3.Hive可以通过查询已有的表的数据，生成新的表：**

**“create table pv\_num row format delimited fields terminated by '\t' as select count(\*) from hmbbs”**

Hive HWI模式搭建：

环境hive-2.2.0：

首先，我们需要去hive的官方网站下载hive的src包

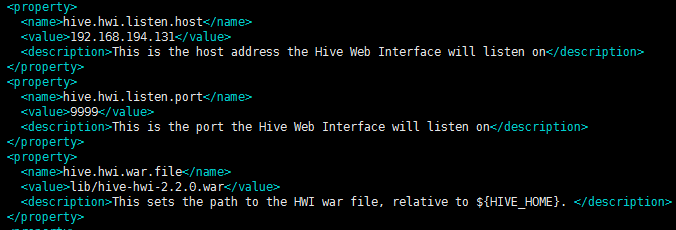
http://mirrors.shuosc.org/apache/hive/



下载之后，可以用IDE工具，也可以直接用DOS命令到apache-hive-2.2.0-src\hive-2.2.0\hwi目录下，执行jar cfM hive-hwi-2.2.0.war -C web . 进行打包（打成war包，注意最后有个点）

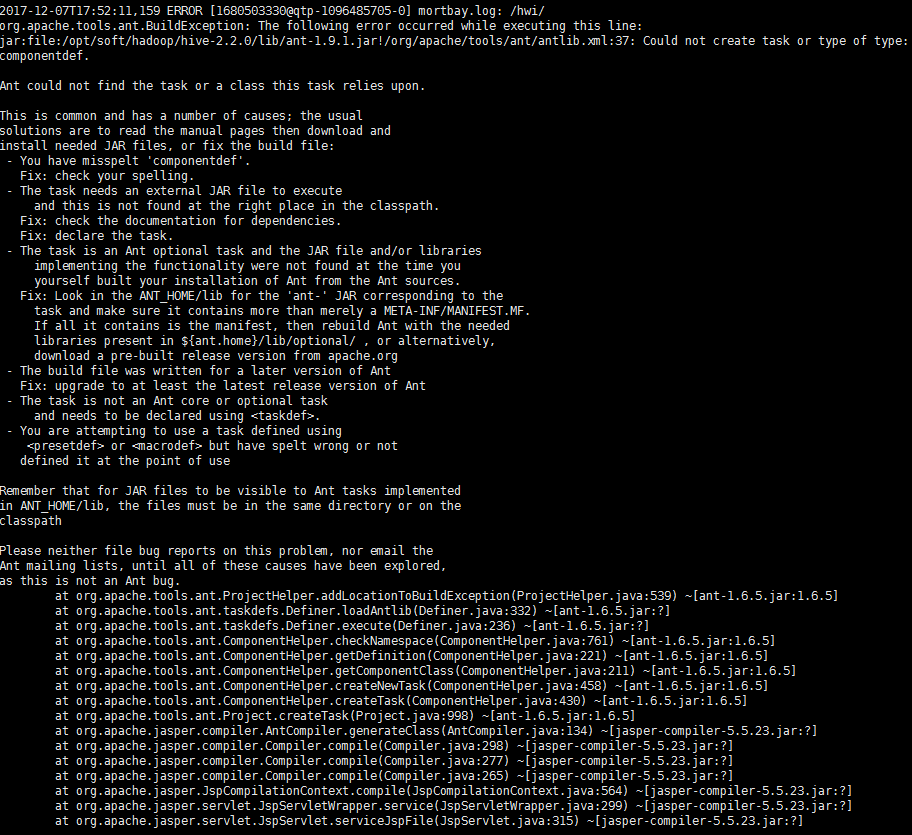
然后将这个hive-hwi-2.2.0.war上传到hive的lib目录下，并将jdk的tools.jar也放到lib包下。

然后进入hive的conf目录下编辑hive-site.xml

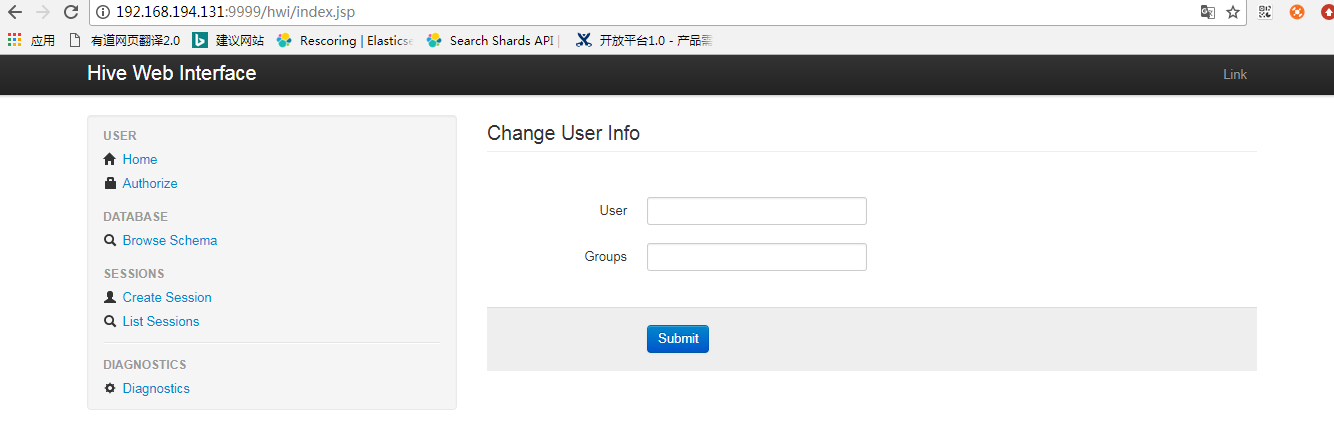


然后到hive的bin目录下执行命令：hive --service hwi

访问<http://192.168.194.131:9999/hwi/>，发现hive日志中报错



我发现我的hive的lib包下有两个ant的jar包：ant-1.6.5.jar 和 ant-1.9.1.jar，将那个低版本的删掉，再次启动hive：hive --service hwi 然后访问<http://192.168.194.131:9999/hwi/>。

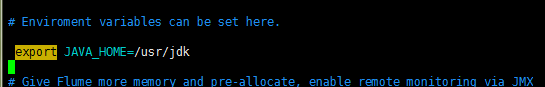


Hadoop flume日志收集框架搭建：

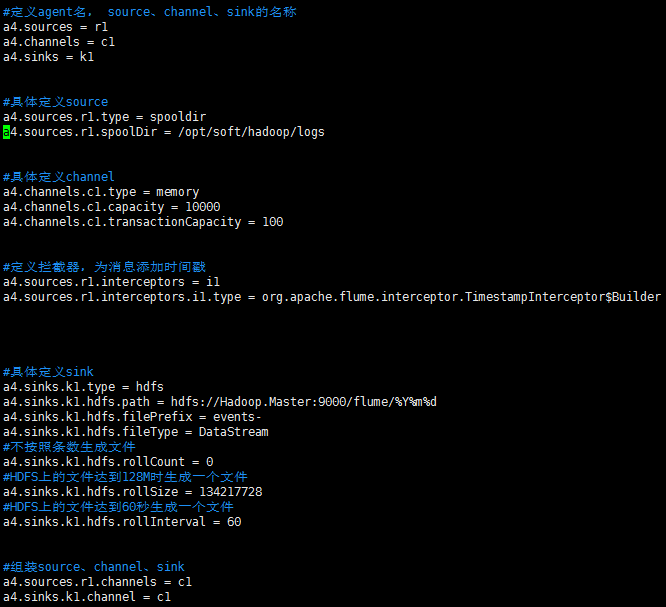
去官网下载flume最新版1.9.9，上传达哦我们的虚拟机，并解压。

进入flume目录中进入conf目录配置flume-env.sh

将Java的环境变量配置注释去掉，并配置正确的Java目录地址



然后我们在conf目录下，编辑我们的flume的组件配置（source、channel、sink的配置），我们给配置文件取名a4.conf，配置如下：



接下来我们需要将hadoop的以下配置文件以及jar包导入到flume中，将hadoop目录中etc/hadoop/目录下的hdfs-site.xml、core-site.xml 配置文件，以及hadoop目录中share/hadoop目录下的commons-configuration-1.6.jar、hadoop-hdfs-2.7.2.jar、hadoop-auth-2.7.2.jar、hadoop-common-2.7.2.jar这四个jar包，放到flume中的lib包下。

启动之前，我们必须要保证读取的日志目录真是存在，不然会报错。

然后我们启动flume：

./bin/flume-ng agent -n a4 -c conf -f conf/a4.conf -Dflume.root.logger=INFO,console

我们去hadoop中查看，在我们flume设定的hadoop目录下是否有我们的日志



成功！

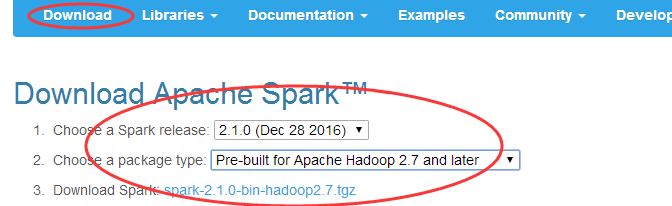
2017-12-12 周二

Hadoop Spark集群搭建，以及IDEA远程调试：

环境：Hadoop-2.7.2 jdk-1.8 scala-2-11-12 spark-2.1.0

spark2.0.0开始，只支持Java8版本了，并且Java8版本最好相对高一点，并且从spark2.1.0开始只支持scala2.11了

scala下载地址（选择2.11.X以上版本）：http://www.scala-lang.org/download/  
spark下载地址：http://spark.apache.org/downloads.html



由于我们的hadoop是2.7.2 版本，这里选择下载对应的spark版本。

首先，将scala和spark的tar包上传到我们的三台虚拟机上，并进行解压。

我们先来配置scala，在配置scala之前我们必须保证服务器上有Java8之后，配置scala的环境变量，编辑/etc/profile：



保存然后使其生效之后，执行scala -version观察输出：



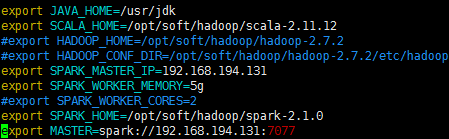
如果输出对应版本号则说明配置成功。

接下来我们配置spark，首先就是环境变量：



然后我们进入spark目录下的conf目录，去配置两个配置文件spark-env.sh slaves

首先我们编辑spark-env.sh，将spark-env.sh.template文件重命名

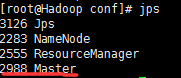


然后我们编辑slaves，将slaves.template文件重命名，并且将hosts文件中我们的两个hadoop从节点的host也来充当我们的spark的从节点

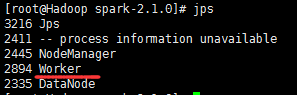


然后我们进入spark目录下的sbin目录中执行start-all.sh，用jps命令观察各个节点的启动效果。

Master：



Slave：



遇到的坑：

1.stop-all.sh执行关闭spark集群的时候，worker节点没有正常关闭

2.spark UI在单机节点的时候访问地址为 IP:4040 ，集群状态下 的访问地址为IP:8080。

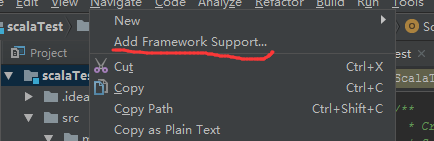
3.由于我的hadoop的环境变量中，将hadoop目录下的sbin目录配置到了PATH中，导致我在spark目录下的sbin目录下执行start-all.sh/stop-all.sh时，总是执行hadoop的start-all.sh/stop-all.sh，想执行spark的start-all.sh只能通过绝对路径执行

虚拟机上搭建好了Spark集群，我们如何用IDEA开发，远程调试呢？首先我们需要在本机的windos系统下安装scala，方式同Linux相同，只需要解压并配置环境变量即可。

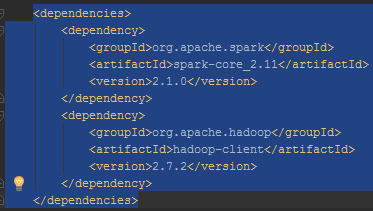
然后我们打开IDEA开发工具，点击File-->Settings-->Plugins-->Browse Repositories，搜索插件Scala，然后安装，重启IDEA。

再次打开IDEA

新建Project -> Scala -> IDEA，右键点击项目，通过Add Framework Support可以添加Maven支持



Pom文件中添加依赖



之后就可以按照官网推荐的例子编码了。

2018-01-05 周五

Druid 数据库单机安装部署：

首先，下载zookeeper、druid的tar安装包到服务器。

ZK下载地址：https://www.apache.org/dyn/closer.cgi/zookeeper/

Druid下载地址：<http://druid.io/downloads.html>

基本上安装过程很无脑，只是解压缩就好，但是配置和启动就。。。（全是泪）

根据官网的示例，首先我们安装并启动zk

tar -xzf zookeeper-3.4.10.tar.gz

cd zookeeper-3.4.10

cp conf/zoo\_sample.cfg conf/zoo.cfg

./bin/zkServer.sh start

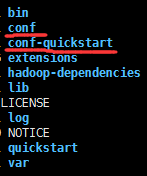
一顿操作猛如虎，接下来我们解压druid的tar包

tar -xzf druid-0.11.0-bin.tar.gz

cd druid-0.11.0

bin/init（初始化）

首先我们需要普及一点东西，正确与否还不确定。。。



这两个都是配置文件目录，进去看了下，不同的是，conf目录配置比较全，conf-quickstart是最轻量的配置，也就是字面上的快速启动。。。在启动的时候可以指定配置文件，这是两套配置文件

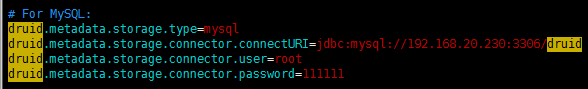
然后我们需要注意，一个配置文件conf/druid/\_common/common.runtime.properties 以及

conf-quickstart/druid/\_common/common.runtime.properties

common.runtime.properties这个配置文件里面记录了一些服务器，以及数据源方面的一些主要配置

如果需要全部的服务都能够启动，我们需要配置两处。

一个是mysql，用来存储druid的一些存储信息。



第二个是zookeeper（单机好像也可以不配置，默认的也可以，集群的就应该好好配置下了），划掉是因为不给后面看的人误导



单机可以在一个节点将所有druid的服务启动，但是必须先启动middleManager，否则其他的节点启动会有问题，数据源必须配置好否则overlord节点会有问题。

middleManager节点启动，灰色标记的地方表明了我们启动采用的配置文件为conf-quickstart目录下的配置文件

java `cat conf-quickstart/druid/middleManager/jvm.config | xargs` -cp "conf-quickstart/druid/\_common:conf-quickstart/druid/middleManager:lib/\*" io.druid.cli.Main server middleManager

historical节点启动：

java `cat conf-quickstart/druid/historical/jvm.config | xargs` -cp "conf-quickstart/druid/\_common:conf-quickstart/druid/historical:lib/\*" io.druid.cli.Main server historical

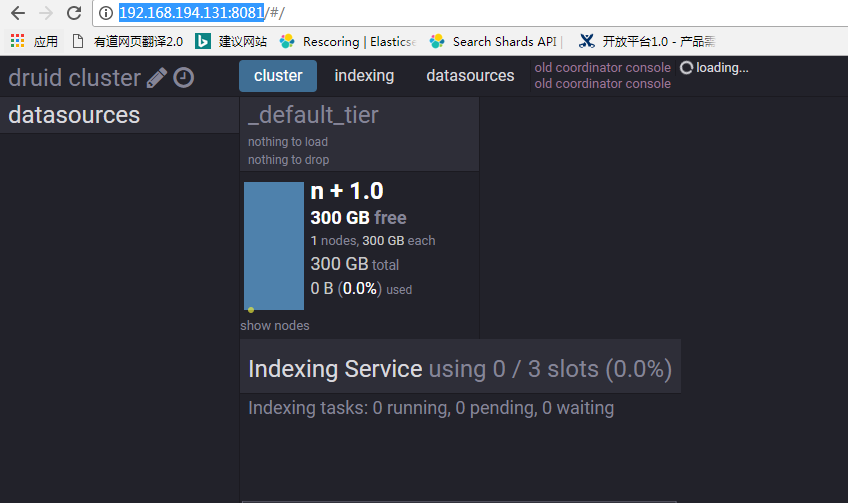
broker节点启动：

java `cat conf-quickstart/druid/broker/jvm.config | xargs` -cp "conf-quickstart/druid/\_common:conf-quickstart/druid/broker:lib/\*" io.druid.cli.Main server broker

coordinator节点启动：

java `cat conf-quickstart/druid/coordinator/jvm.config | xargs` -cp "conf-quickstart/druid/\_common:conf-quickstart/druid/coordinator:lib/\*" io.druid.cli.Main server coordinator

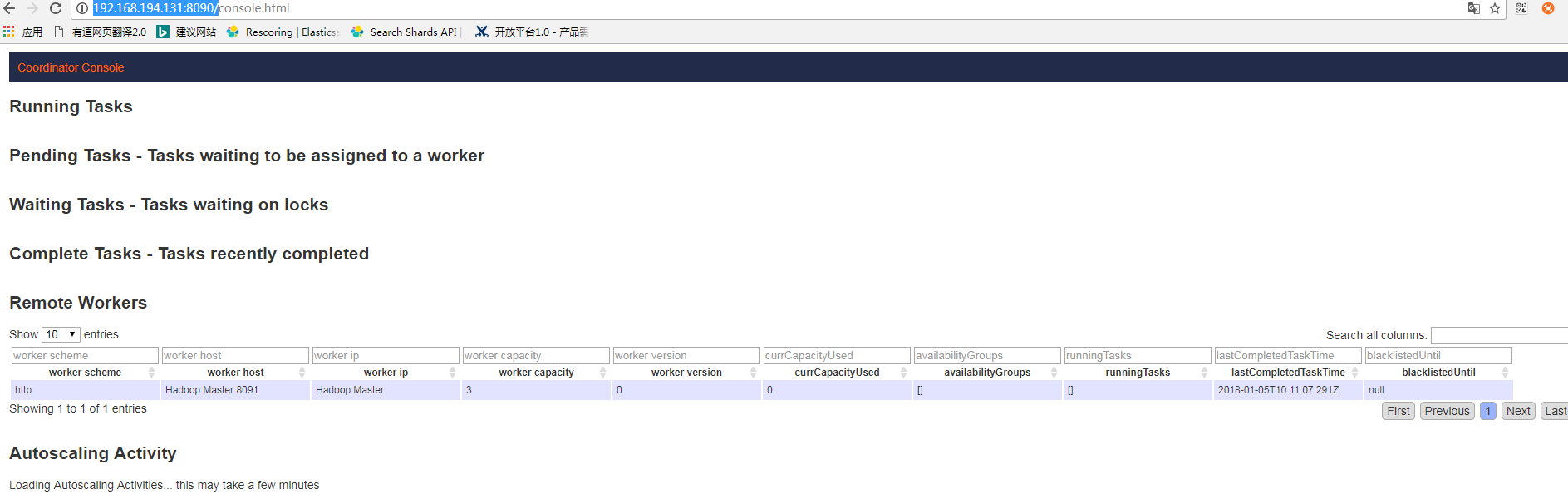
coordinator可视化界面：



overlord节点启动：

java `cat conf-quickstart/druid/overlord/jvm.config | xargs` -cp "conf-quickstart/druid/\_common:conf-quickstart/druid/overlord:lib/\*" io.druid.cli.Main server overlord

overload可视化界面



2018-01-15 周一

最近想学习Hbase 和 Druid ，Druid感觉有点入门困难，从应用范围上看先学Hbase吧。

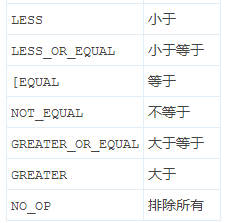
了解了一些HBase，感觉HBase的难点有两个，一个是RowKey的设计，还有就是针对RowKey无法查询的复杂查询的二级索引的实现。

简单的代码，建表CRDU操作都实现了，现在主要想学点高级的东西，Filter 和 Coprocessor

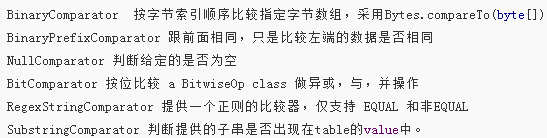
**Hbase filter 学习:**

首先需要了解与Filter联用的两个东西，过滤器操作符和比较器：

过滤器操作符：



比较器：



基本过滤器：

RowFilter：RowKey过滤器，一般用于针对RowKey进行过滤查询的时候后比较有用。

RowFilter(final CompareOp rowCompareOp, final ByteArrayComparable rowComparator)

例如：RowFilter filter = new RowFilter(CompareOp.EQUAL, new BinaryComparator(row));

QualifierFilter(final CompareOp op, final ByteArrayComparable qualifierComparator)：列过滤器，针对列族中的所有列进行过滤，类似RowFilter只不过是针对列名了而已。

例如：Filter filter = new QualifierFilter(CompareFilter.CompareOp.LESS\_OR\_EQUAL, new BinaryComparator("balance".getBytes()));

ValueFilter(final CompareOp valueCompareOp, final ByteArrayComparable valueComparator)：针对每个单元的值进行过滤，值符合条件的数据将会被加载。

例如：Filter filter = new ValueFilter(CompareFilter.CompareOp.EQUAL,new SubstringComparator("Kent"));

基于列，单元值的过滤器：

SingleColumnValueFilter(final byte [] family, final byte [] qualifier, final CompareOp compareOp, final byte[] value)：键值过滤器，指定返回列族中的某列，匹配对应的值的整行数据。

例如：Filter filter = new SingleColumnValueFilter("info".getBytes(),"city".getBytes(), CompareFilter.CompareOp.EQUAL, "Allamuchy".getBytes());返回info列族下city列值为Allamuchy的数据。

SingleColumnValueExcludeFilter(byte[] family, byte[] qualifier, CompareOp compareOp, byte[] value):与SingleColumnValueFilterTest使用方法相似，但是返回结果不包括指定的列

基于单元值的过滤器：

KeyOnlyFilter()：只返回每行的行键，值全部为空，对于只关注于行键的应用场景来说非常合适，这样忽略掉其值就可以减少传递到客户端的数据量，能起到一定的优化作用

例如：Filter filter = new KeyOnlyFilter();

TimestampsFilter(List<Long> timestamps)：返回指定时间戳版本的数据。

例如：List<Long> list = new ArrayList<>();  
list.add(1515839411077L);  
Filter filter = new TimestampsFilter(list);返回时间戳版本为1515839411077的数据

行过滤器：

PrefixFilter(final byte [] prefix)：前缀匹配rowKey进行查询，有点类似于RowFilter结合RegexComparator来实现，这里提供了更简单的用法。

例如：Filter filter = new PrefixFilter(Bytes.toBytes("ffe131f7"));

PageFilter(final long pageSize)：根据RowKey的排序进行分页查询，感觉没啥大用，因为只能传一个pagesize参数。

InclusiveStopFilter(final byte [] stopRowKey)：过滤出从起始行建到结束行键（包含）数据的的过滤器

例如：Filter filter = new InclusiveStopFilter(Bytes.toBytes("006c679b-8b24-4d86-b418-c5821ad69f7f"));

RandomRowFilter(float chance)：作用就是按照一定的几率（<=0会过滤掉所有的行，>=1会包含所有的行）来返回随机的结果集，对于同样的数据集，多次使用同一个RandomRowFilter会返回不通的结果集，对于需要随机抽取一部分数据的应用场景，可以使用此过滤器

例如：Filter filter = new RandomRowFilter(0.5f);

列过滤器：

ColumnPaginationFilter(final int limit, final int offset)：列分页过滤器：基于列进行分页，需要设置偏移量与返回数量

例如：Filter filter = new ColumnPaginationFilter(2,0);

ColumnCountGetFilter(final int n) 列计数过滤器，返回限定数量的列，可以指定需要返回的数量限制，这个过滤器更适合get操作，使用scan操作时需要注意，当某一行的列数量超过指定的数量，scan不会继续扫描下一行，而是停止本次扫描直接返回结果。

例如：Filter filter = new ColumnCountGetFilter(2);

FirstKeyOnlyFilter()：首列过滤器，只返回每一行的第一列数据

例如：Filter filter = new FirstKeyOnlyFilter();

FirstKeyValueMatchingQualifiersFilter(Set<byte[]> qualifiers

)首次匹配列过滤器：通过设置一组需要匹配的列，只要匹配到任意一个列就会停止这一行的扫描操作进行下一行的扫描。

下面代码设定两个需要匹配的列address，account\_number，只要匹配到任意一个列就会停止这一行的扫描，继续扫描下一行（针对scan）。存储结构上address位于account\_number之后。扫描过程中会首先匹配到account\_number，此时不会继续匹配下一个列，而是直接开始下一行的扫描。返回结果包括匹配到的列之前的所有列值（包含匹配列）。

例如：Set<byte[]> qualifiers = new HashSet<byte[]>();  
qualifiers.add(Bytes.toBytes("address"));  
qualifiers.add(Bytes.toBytes("account\_number"));  
Filter filter = new FirstKeyValueMatchingQualifiersFilter(qualifiers);

ColumnPrefixFilter(final byte [] prefix)：列前缀过滤器：针对每一行列名的前缀匹配。

例如：Filter filter = new ColumnPrefixFilter(Bytes.toBytes("a"));

MultipleColumnPrefixFilter(byte[][] prefixes)：列多前缀匹配过滤器：在列前缀匹配的基础上增加匹配列。

例如：byte[][] prefixes = new byte[][]{Bytes.toBytes("a"),Bytes.toBytes("b")};  
Filter filter = new MultipleColumnPrefixFilter(prefixes);

ColumnRangeFilter(byte[] minColumn, boolean minColumnInclusive, byte[] maxColumn, boolean maxColumnInclusive)：列范围匹配过滤器：设定列的范围进行匹配

参数解释：

minColumn - 列范围的最小值，如果为空，则没有下限；

minColumnInclusive - 列范围是否包含minColumn ；

maxColumn - 列范围最大值，如果为空，则没有上限；

maxColumnInclusive - 列范围是否包含maxColumn 。

例如：Filter filter = new ColumnRangeFilter(Bytes.toBytes("a"), false, Bytes.toBytes("d"), false);

允许用户指定一个参考列或引用列来过滤其他列的过滤器，过滤的原则是基于参考列的时间戳来进行筛选

DependentColumnFilter()

DependentColumnFilter(byte[] family,byte[] qulifier)

DependentColumnFilter(byte[] family,byte[] qulifier，boolean dropDependentColumn)

DependentColumnFilter(byte[] family,byte[] qulifier，boolean dropDependentColumn，CompareOp valueCompareOp, WritableByteArrayComparable valueComparator)

相关参数：

boolean dropDependentColumn -- 决定参考列被返回还是丢弃，为true时表示参考列被返回，为false时表示被丢弃

CompareOp valueCompareOp --  比较运算符

WritableByteArrayComparable valueComparator --  比较器

例如：Filter filter = new DependentColumnFilter(Bytes.toBytes("info"),Bytes.toBytes("account\_number"));

包装过滤器：

SkipFilter(Filter filter)：根据整行中的每个列来做过滤，只要存在一列不满足条件，整行都被过滤掉。SkipFilter包装了其它的过滤器，只要被包装的过滤器返回的结果中有一行的某一列或者某个KeyValue被过滤掉了，那么SkipFilter会将该列或者KeyValue所处的整行全部过滤。被包装的过滤器必须实现filterKeyValue()方法。因为SkipFilter会依靠filterKeyValue()返回的结果进行附加的处理

例如，如果一行中的所有列代表的是不同物品的重量，则真实场景下这些数值都必须大于零，我们希望将那些包含任意列值为0的行都过滤掉。

在这个情况下，我们结合ValueFilter（只返回满足指定值得那列）和SkipFilter共同实现该目的：

scan.setFilter(new SkipFilter(new ValueFilter(CompareOp.NOT\_EQUAL,new BinaryComparator(Bytes.toBytes(0))));

WhileMatchFilter(Filter filter)：WhileMatchFilter跟SkipFilter类似，不过区别之处在于WhileMatchFilter一旦找到某一行中的某些列值或者KeyValue不满足条件，那么整个Scan操作就会被终止。SkipFilter只是会将此行过滤，不作为返回值，但Scan操作会继续。

例如：

Filter filter1 = new RowFilter(CompareFilter.CompareOp.LESS,new BinaryComparator("ffb74d3c-5573-4ec3-ba83-364584c66c78".getBytes()));  
Filter filter2 = new WhileMatchFilter(filter1);

上例返回了RowKey字典值小于"ffb74d3c-5573-4ec3-ba83-364584c66c78"的所有行，当遇到不满足条件的那一列时，Scan将结束不在继续扫描，这个例子不是很清晰，如果我们这样：

Filter filter1 = new RowFilter(CompareFilter.CompareOp. GREATER

,new BinaryComparator("ffb74d3c-5573-4ec3-ba83-364584c66c78".getBytes()));  
Filter filter2 = new WhileMatchFilter(filter1);

将条件变成大于，将不会有加过返回，因为WhileMatchFilter的性质，当Scan到第一条的时候条件不满足，Scan结束，返回结果就是空的。

组合过滤器：

FilterList:组合过滤器，代表一个过滤器链，它可以包含一组即将应用于目标数据集的过滤器，过滤器间具有“与” FilterList.Operator.MUST\_PASS\_ALL 和“或” FilterList.Operator.MUST\_PASS\_ONE 关系。

构造方法：

[FilterList](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/FilterList.html" \l "FilterList(org.apache.hadoop.hbase.filter.Filter...))([Filter](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/Filter.html" \o "class in org.apache.hadoop.hbase.filter)... rowFilters)

[FilterList](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/FilterList.html" \l "FilterList(org.apache.hadoop.hbase.filter.FilterList.Operator))([FilterList.Operator](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/FilterList.Operator.html" \o "enum in org.apache.hadoop.hbase.filter) operator)

[FilterList](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/FilterList.html" \l "FilterList(org.apache.hadoop.hbase.filter.FilterList.Operator,%20org.apache.hadoop.hbase.filter.Filter...))([FilterList.Operator](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/FilterList.Operator.html" \o "enum in org.apache.hadoop.hbase.filter) operator, [Filter](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/Filter.html" \o "class in org.apache.hadoop.hbase.filter)... rowFilters)

[FilterList](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/FilterList.html" \l "FilterList(org.apache.hadoop.hbase.filter.FilterList.Operator,%20java.util.List))([FilterList.Operator](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/FilterList.Operator.html" \o "enum in org.apache.hadoop.hbase.filter) operator, [List](http://docs.oracle.com/javase/7/docs/api/java/util/List.html?is-external=true" \o "class or interface in java.util)<[Filter](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/Filter.html" \o "class in org.apache.hadoop.hbase.filter)> rowFilters)

[FilterList](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/FilterList.html" \l "FilterList(java.util.List))([List](http://docs.oracle.com/javase/7/docs/api/java/util/List.html?is-external=true" \o "class or interface in java.util)<[Filter](http://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/Filter.html" \o "class in org.apache.hadoop.hbase.filter)> rowFilters)

例如：

//第一个过滤器

Filter rowFilter = new RowFilter(CompareFilter.CompareOp.GREATER, new BinaryComparator(Bytes.toBytes("ffa7d23c-f73d-4525-9786-0e0e4b6e3815")));

//第二个过滤器

Filter valFilter = new ValueFilter(CompareFilter.CompareOp.EQUAL,new BinaryComparator(Bytes.toBytes("Hill")));

//初始化一个 AND 关系的组合过滤器

FilterList filterList = new FilterList(FilterList.Operator.MUST\_PASS\_ALL);  
filterList.addFilter(rowFilter);  
filterList.addFilter(valFilter);

-------------------------------------------------------------------------------

穿插一个东西protobuf：

protobuf是HBase必须的一个工具而且版本只能是2.5.0

首先下载protobuf-2.5.0.tar.gz 然后tar -zxvf解压

进入到protobuf-2.5.0目录

然后执行./configure -prefix= /opt/soft/hadoop/protobuf-2.5

然后make （会很久。。。）

再然后make check （会很久。。。）

最后make install （会很久。。。）

然后编辑 vim /etc/profile 配置环境变量，source /etc/profile



最后验证一下：



这时候我们还不能正常使用protobuf，，只是安装成功，还需要配置protobuf的共享lib目录才能使用

vim /etc/ld.so.conf

将protobuf的安装目录下的lib目录添加到/etc/ld.so.conf的后一行，然后执行

ldconfig使其生效

完成

-------------------------------------------------------------------------------

**HBase Coprocessor 学习：**

HBase 支持两种类型的Coprocessor（协处理器），Endpoint 和 Observer。

Endpoint 协处理器类似传统数据库中的存储过程，客户端可以调用这些 Endpoint 协处理器执行一段 Server 端代码，并将 Server 端代码的结果返回给客户端进一步处理，最常见的用法就是进行聚集操作。如果没有协处理器，当用户需要找出一张表中的最大数据，即 max 聚合操作，就必须进行全表扫描，在客户端代码内遍历扫描结果，并执行求最大值的操作。这样的方法无法利用底层集群的并发能力，而将所有计算都集中到 Client 端统一执行，势必效率低下。利用 Coprocessor，用户可以将求最大值的代码部署到 HBase Server 端，HBase 将利用底层 cluster 的多个节点并发执行求最大值的操作。即在每个 Region 范围内执行求最大值的代码，将每个 Region 的最大值在 Region Server 端计算出，仅仅将该 max 值返回给客户端。在客户端进一步将多个 Region 的最大值进一步处理而找到其中的最大值。这样整体的执行效率就会提高很多。

另外一种协处理器叫做 Observer Coprocessor，这种协处理器类似于传统数据库中的触发器，当发生某些事件的时候这类协处理器会被 Server 端调用。Observer Coprocessor 就是一些散布在 HBase Server 端代码中的 hook 钩子，在固定的事件发生时被调用。比如：put 操作之前有钩子函数 prePut，该函数在 put 操作执行前会被 Region Server 调用；在 put 操作之后则有 postPut 钩子函数。

**EndPoint代码开发示例:**

准备环境：Hadoop-2.7.2 、 HBase-1.2.6 、protobuf-2.5.0 、maven

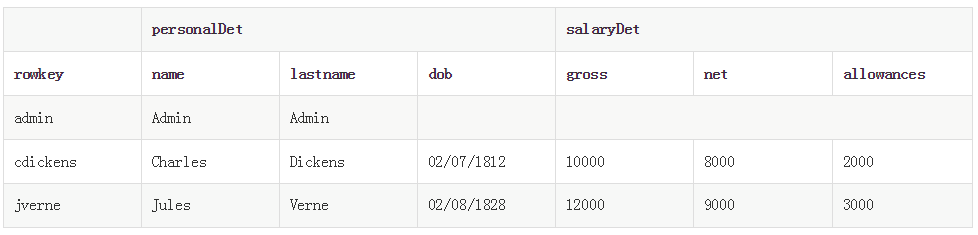
首先，我们先来介绍一下protobuf这个工具，这个工具是由[Google](http://www.baidu.com/link?url=EQgj0q3d8eVYDi1D93hcbZ0qrD0P8cKfW6hBgE5qdUcU-_utzEWlPlM_VDK7it0fIaWGKm_o2w_ETOQbtQxlLud-x0Qxm-i6Fsf81trK03m" \t "_blank)开发的非常牛逼的一个工具全称Google Protocol Buffer( 简称 Protobuf) 是 Google 公司内部的混合语言数据标准，目前已经正在使用的有超过 48,162 种报文格式定义和超过 12,183 个 .proto 文件。他们用于 RPC 系统和持续数据存储系统。

Protocol Buffers 是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。它很适合做数据存储或 RPC 数据交换格式。可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。目前提供了 C++、Java、Python 三种语言的 API。

我们需要通过protobuf为我们生成客户端与服务端的HBase Coprocessor协调器代码通讯的RPC接口。

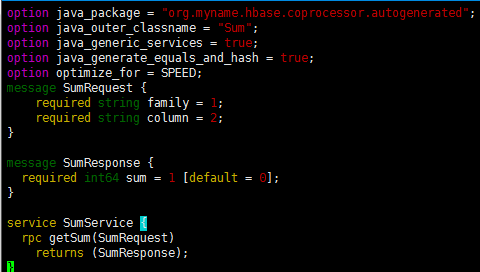
一下例子均为官网给出的例子：

数据：



使用这个表格的数据，实现了一个端点协处理器来计算所有员工工资的总和。

首先我们先用protobuf工具生成我们的RPC通讯接口：



这是官网给出的protobuf例子，大概意思就是在

org.myname.hbase.coprocessor.autogenerated包下生成一个Sum.Java，定义了请求参数格式以及响应参数格式，并且顶一个SumService内部类，类中定义了一个getSum方法入参格式以及出参格式分别为SumRequest和SumResponse

我们编辑好这个.proto文件之后，接下来将用他生成我们后面将要用到的RPC通讯接口，在.proto所在目录执行如下命令：

protoc --java\_out=输出目录 文件所在目录

例如：protoc --java\_out=/opt/soft/hadoop/protobuf-2.5.0/src ./sum.proto

将会在当前目录生成一个.proto中java\_package字段设置的目录，该目录下有我们需要的RPC通讯类。比如我们刚才.proto设置的java\_package字段为org.myname.hbase.coprocessor.autogenerated，java\_outer\_classname设置为Sum，那么将会生成一个org/myname/hbase/coprocessor/autogenerated/目录，目录下有一个Sum.java。

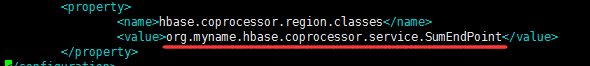
RPC通讯类生成之后，接下来我们开始编写HBase服务端的Coprocessor代码：

服务店代码我们需要做如下几步：

1. 创建maven项目；
2. Pom文件引入protobuf的jar依赖；

<dependency>  
 <groupId>com.google.protobuf</groupId>  
 <artifactId>protobuf-java</artifactId>  
 <version>2.5.0</version>  
</dependency>

1. 将protobuf生成的RPC Java文件放入我们的项目中；
2. 然后编写我们的实现代码，继承RPC Java类，实现Coprocessor以及CoprocessorService，重写我们要实现功能的方法；
3. 将项目打成jar包上传到各个HBase服务器节点中，路径为HBase的安装根目录的lib目录下
4. 将自定义的Coprocessor配置到我们HBase中，编辑HBase-site.xml,加入如下配置，并重启HBase集群



其中红线部分为我们的实现代码的全名称，这种配置Coprocessor的方式为静态配置，还有动态配置，后续介绍

1. 编写客户端代码，创建Maven项目，将RPC Java类放到指定包名目录下，调用即可

服务端源码：

package org.myname.hbase.coprocessor.service;  
  
import com.google.protobuf.RpcCallback;  
import com.google.protobuf.RpcController;  
import com.google.protobuf.Service;  
import org.apache.hadoop.hbase.Cell;  
import org.apache.hadoop.hbase.CellUtil;  
import org.apache.hadoop.hbase.Coprocessor;  
import org.apache.hadoop.hbase.CoprocessorEnvironment;  
import org.apache.hadoop.hbase.client.Scan;  
import org.apache.hadoop.hbase.coprocessor.CoprocessorException;  
import org.apache.hadoop.hbase.coprocessor.CoprocessorService;  
import org.apache.hadoop.hbase.coprocessor.RegionCoprocessorEnvironment;  
import org.apache.hadoop.hbase.protobuf.ResponseConverter;  
import org.apache.hadoop.hbase.regionserver.InternalScanner;  
import org.apache.hadoop.hbase.util.Bytes;  
import org.myname.hbase.coprocessor.autogenerated.Sum;  
  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.List;

public class SumEndPoint extends Sum.SumService implements Coprocessor, CoprocessorService {  
  
 private RegionCoprocessorEnvironment env;  
  
 @Override  
 public Service getService() {  
 return this;  
 }  
  
 @Override  
 public void start(CoprocessorEnvironment env) throws IOException {  
 if (env instanceof RegionCoprocessorEnvironment) {  
 this.env = (RegionCoprocessorEnvironment)env;  
 } else {  
 throw new CoprocessorException("Must be loaded on a table region!");  
 }  
 }  
  
 @Override  
 public void stop(CoprocessorEnvironment env) throws IOException {  
 // do nothing  
 }  
  
 @Override  
 public void getSum(RpcController controller, Sum.SumRequest request, RpcCallback done) {  
 Scan scan = new Scan();  
 scan.addFamily(Bytes.toBytes(request.getFamily()));  
 scan.addColumn(Bytes.toBytes(request.getFamily()), Bytes.toBytes(request.getColumn()));  
 Sum.SumResponse response = null;  
 InternalScanner scanner = null;  
 String gross = null;//从HBase中取出的收入字符串  
 long grossNum = 0L;//从字符串转化成数字long类型的收入字段  
 try {  
 scanner = env.getRegion().getScanner(scan);  
 List<Cell> results = new ArrayList<>();  
 boolean hasMore = false;  
 long sum = 0L;  
 do {  
 hasMore = scanner.next(results);  
 for (Cell cell : results) {  
 gross = Bytes.toString(CellUtil.cloneValue(cell));  
 grossNum = Long.parseLong(gross);  
 sum = sum + grossNum;  
 }  
 results.clear();  
 } while (hasMore);  
  
 response = Sum.SumResponse.newBuilder().setSum(sum).build();  
  
 } catch (IOException ioe) {  
 ResponseConverter.setControllerException(controller, ioe);  
 } finally {  
 if (scanner != null) {  
 try {  
 scanner.close();  
 } catch (IOException ignored) {}  
 }  
 }  
 done.run(response);  
 }  
}

客户端源码：

package hbase.coprocessor;  
  
import com.google.protobuf.ServiceException;  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.hbase.HBaseConfiguration;  
import org.apache.hadoop.hbase.TableName;  
import org.apache.hadoop.hbase.client.Connection;  
import org.apache.hadoop.hbase.client.ConnectionFactory;  
import org.apache.hadoop.hbase.client.Table;  
import org.apache.hadoop.hbase.client.coprocessor.Batch;  
import org.apache.hadoop.hbase.ipc.BlockingRpcCallback;  
import org.myname.hbase.coprocessor.autogenerated.Sum;  
  
import java.io.IOException;  
import java.util.Map;  
  
/\*\*  
 \* Created by Administrator on 2018/1/18.  
 \*/  
public class CoprocessorTest {  
 public static void main(String[] args) throws IOException {  
 Configuration conf = HBaseConfiguration.create();  
 // Use below code for HBase version 1.x.x or above.  
 conf.set("hbase.zookeeper.quorum","Hadoop.Master:2181,Hadoop.Slave1:2181,Hadoop.Slave2:2181");  
 Connection connection = ConnectionFactory.createConnection(conf);  
 TableName tableName = TableName.valueOf("Users");  
 Table table = connection.getTable(tableName);  
  
 //Use below code HBase version 0.98.xx or below.  
 //HConnection connection = HConnectionManager.createConnection(conf);  
 //HTableInterface table = connection.getTable("users");  
  
 final Sum.SumRequest request = Sum.SumRequest.newBuilder().setFamily("salaryDet").setColumn("gross")  
 .build();  
 try {  
 Map<byte[], Long> results = table.coprocessorService(Sum.SumService.class, null, null,  
 new Batch.Call<Sum.SumService, Long>() {  
 @Override  
 public Long call(Sum.SumService aggregate) throws IOException {  
 BlockingRpcCallback rpcCallback = new BlockingRpcCallback();  
 aggregate.getSum(null, request, rpcCallback);  
 Sum.SumResponse response = (Sum.SumResponse) rpcCallback.get();  
 return response.hasSum() ? response.getSum() : 0L;  
 }  
 });  
 for (Long sum : results.values()) {  
 System.out.println("Sum = " + sum);  
 }  
 } catch (ServiceException e) {  
 e.printStackTrace();  
 } catch (Throwable e) {  
 e.printStackTrace();  
 }  
 }  
}

具体代码参考项目（PS：官网的例子好像好久没更新了，有很多错误，例子还需要自己改改才能用）：

Server端hbaseCoprocessorServer/SumEndPoint

Client端：myhbase/CoprocessorTest

坑：

java.lang.IllegalArgumentException: offset (0) + length (8) exceed the capacity of the array: 5

at org.apache.hadoop.hbase.util.Bytes.explainWrongLengthOrOffset(Bytes.java:632)

at org.apache.hadoop.hbase.util.Bytes.toLong(Bytes.java:606)

at org.apache.hadoop.hbase.util.Bytes.toLong(Bytes.java:579)

at org.myname.hbase.coprocessor.service.SumEndPoint.getSum(SumEndPoint.java:64)

at org.myname.hbase.coprocessor.autogenerated.Sum$SumService.callMethod(Sum.java:1267)

at org.apache.hadoop.hbase.regionserver.HRegion.execService(HRegion.java:7875)

at org.apache.hadoop.hbase.regionserver.RSRpcServices.execServiceOnRegion(RSRpcServices.java:2008)

at org.apache.hadoop.hbase.regionserver.RSRpcServices.execService(RSRpcServices.java:1990)

at org.apache.hadoop.hbase.protobuf.generated.ClientProtos$ClientService$2.callBlockingMethod(ClientProtos.java:33652)

at org.apache.hadoop.hbase.ipc.RpcServer.call(RpcServer.java:2196)

at org.apache.hadoop.hbase.ipc.CallRunner.run(CallRunner.java:112)

at org.apache.hadoop.hbase.ipc.RpcExecutor.consumerLoop(RpcExecutor.java:133)

at org.apache.hadoop.hbase.ipc.RpcExecutor$1.run(RpcExecutor.java:108)

at java.lang.Thread.run(Thread.java:745)

上面的错误来源于这段代码：

Bytes.toLong(CellUtil.cloneValue(cell));

Cell中获取的是一个数字字符串12000，在调用Bytes.toLong的时候发生了转换错误，原因是HBase中存储格式之后字符串，而字符串格式的12000再调用CellUtil.cloneValue转换成byte数组的时候是5位（12000,5个字符，也就是5位），而long类型的转换需要8位字符，因此发生异常。

解决办法：

可以存储数据的时候就存储long类型的byte数组，取的时候就不会有上述的问题，但是存储到Hbase中的将会是一串乱码，数据可读性不好，并且在通过hive查询的时候也需要针对此类字段做特殊处理。

另外一种是通过代码处理：可以先将Cell的值转为字符串，然后再转成数字。

String gross = Bytes.toString(CellUtil.cloneValue(cell));  
Long grossNum = Long.parseLong(gross);

-------------------------------------------------------------------------------

**Observer代码开发示例：**

Observer协处理器的开发没有protobuf生成RPC的这一步骤，由于Observer功能有点类似于触发器，它并不需要客户端直接调用，而是通过我们的某个操作直接触发，所以不需要RPC代码生成。

Observer处理器又分为四个接口，分别观察处理不同事件，RegionObserver、RegionServerObserver、MasterObserver、WalObserver  
我们在开发Observer类型的处理器的时候西药继承这四个接口，但是接口中方法太多我们没有必要全部实现，于是HBase为我们分别提供了类似于适配器的便利类  
RegionObserver ==> BaseRegionObserver /用于观察一个region的事件，如Get 和Put操作  
RegionServerObserver ==> BaseMasterAndRegionObserver /观察与RegionServer的操作相关的事件，例如启动，停止或执行合并，提交或回滚。  
MasterObserver ==> BaseMasterAndRegionObserver /观察与HBase Master相关的事件，例如表创建，删除或架构修改。  
WalObserver ==> BaseWALObserver 观察与写入日志（WAL）相关的事件下面是一个官网Observer处理器示例。

步骤：

1. 创建项目，编写Observer代码
2. 将项目打成jar包，传到HDFS中
3. 将Observer注册到HBase
4. Client端调用方法测试触发动作

import org.apache.hadoop.hbase.Cell;  
import org.apache.hadoop.hbase.CellUtil;  
import org.apache.hadoop.hbase.client.Get;  
import org.apache.hadoop.hbase.client.Result;  
import org.apache.hadoop.hbase.client.Scan;  
import org.apache.hadoop.hbase.coprocessor.BaseRegionObserver;  
import org.apache.hadoop.hbase.coprocessor.ObserverContext;  
import org.apache.hadoop.hbase.coprocessor.RegionCoprocessorEnvironment;  
import org.apache.hadoop.hbase.filter.BinaryComparator;  
import org.apache.hadoop.hbase.filter.CompareFilter;  
import org.apache.hadoop.hbase.filter.Filter;  
import org.apache.hadoop.hbase.filter.RowFilter;  
import org.apache.hadoop.hbase.regionserver.InternalScanner;  
import org.apache.hadoop.hbase.regionserver.RegionScanner;  
import org.apache.hadoop.hbase.util.Bytes;  
  
import java.io.IOException;  
import java.util.Iterator;  
import java.util.List;  
  
/\*\*  
 \* 下面是一个官网Observer处理器示例：  
 \* Observer处理器的所有方法中  
 \* pre 代表前置方法动作  
 \* post 代表后置方法动作  
 \*  
 \* 例如：preGetOp()方法 为 执行get前调用的方法 postGetOp()为执行get后调用的方法  
 \*/  
public class AdminObserver extends BaseRegionObserver {  
 private static final byte[] ADMIN = Bytes.toBytes("admin");  
 private static final byte[] COLUMN\_FAMILY = Bytes.toBytes("details");  
 private static final byte[] COLUMN = Bytes.toBytes("Admin\_det");  
 private static final byte[] VALUE = Bytes.toBytes("You can't see Admin details");  
  
 /\*\*  
 \* 覆盖该preGetOp()方法（该preGet()方法不推荐使用）来检查客户端是否查询了值为admin的rowkey。  
 \* 如果是这样，返回一个空的结果。否则，正常处理请求。  
 \*  
 \* @param e  
 \* @param get  
 \* @param results  
 \* @throws IOException  
 \*/  
 @Override  
 public void preGetOp(ObserverContext<RegionCoprocessorEnvironment> e, Get get, List<Cell> results) throws IOException {  
 if (Bytes.equals(get.getRow(),ADMIN)) {  
 Cell c = CellUtil.createCell(get.getRow(),COLUMN\_FAMILY, COLUMN,  
 System.currentTimeMillis(), (byte)4, VALUE);  
 results.add(c);  
 e.bypass();//方法表明当前的协处理器的返回值应该被用来代替正常的HBase值,然后不再请求region读取实际的数据  
 }  
 }  
  
 /\*\*  
 \* 每当有scan动作的之前，设置一个过滤器过滤所有的RowKey为Admin的行，返回非Admin的结果行。  
 \*  
 \* 这种用法有一个缺点，就是当client端进行scan操作的时候，设置了Filter那么在此处设置的Filter将会覆盖之前的Filter，  
 \* 所以一般过滤动作我们都会在post动作中进行，例如下面的postScannerNext方法  
 \*  
 \* @param e  
 \* @param scan  
 \* @param s  
 \* @return  
 \* @throws IOException  
 \*/  
 @Override  
 public RegionScanner preScannerOpen(final ObserverContext e, final Scan scan,  
 final RegionScanner s) throws IOException {  
  
 Filter filter = new RowFilter(CompareFilter.CompareOp.NOT\_EQUAL, new BinaryComparator(ADMIN));  
 scan.setFilter(filter);  
 return s;  
 }  
  
 /\*\*  
 \* 将scan之后的结果中，移除所有RowKey为Admin的行  
 \*  
 \* @param e  
 \* @param s  
 \* @param results  
 \* @param limit  
 \* @param hasMore  
 \* @return  
 \* @throws IOException  
 \*/  
 @Override  
 public boolean postScannerNext(final ObserverContext e, final InternalScanner s,  
 final List results, final int limit, final boolean hasMore) throws IOException {  
 Result result = null;  
 Iterator iterator = results.iterator();  
 while (iterator.hasNext()) {  
 result = (Result)iterator.next();  
 if (Bytes.equals(result.getRow(), ADMIN)) {  
 iterator.remove();  
 break;  
 }  
 }  
 return hasMore;  
 }  
}

在这里，我们将用动态加载的方式加载Coprocessor，东塔加载的方式与静态加载方式的区别在于，静态加载Coprocessor有效范围为所有的表，动态加载的方式有效范围为设定表。

动态加载方式有两种：hbase shell方式 和 Java api方式

下面贴出我们注册Coprocessor的代码以及卸载Coprocessor的代码

注册Coprocessor：主要代码HTableDescriptor 下的 addCoprocessor

import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.hbase.\*;  
import org.apache.hadoop.hbase.client.HBaseAdmin;  
import org.myname.hbase.coprocessor.service.AdminObserver;  
  
import java.io.IOException;  
  
/\*\*  
 \* Coprocessor协调器通过java动态注册代码  
 \* 首先需要将jar包存入我们的HDFS目录下  
 \*/  
public class CoprocessorRegisiter {  
 public static void main(String[] args) throws IOException {  
 TableName tableName = TableName.valueOf("Users");  
 //指定HBase服务端协调器jar包在HDFS中的目录  
 String path = "hdfs://Hadoop.Master:9000/user/root/coprocessor.jar";  
 //连接到HBase的zk集群  
 Configuration conf = HBaseConfiguration.create();  
 conf.set("hbase.zookeeper.quorum","Hadoop.Master:2181,Hadoop.Slave1:2181,Hadoop.Slave2:2181");  
  
 HBaseAdmin admin = new HBaseAdmin(conf);  
 admin.disableTable(tableName);  
  
 HTableDescriptor hTableDescriptor = new HTableDescriptor(tableName);  
 HColumnDescriptor columnFamily1 = new HColumnDescriptor("personalDet");  
 columnFamily1.setMaxVersions(3);  
 hTableDescriptor.addFamily(columnFamily1);  
  
 HColumnDescriptor columnFamily2 = new HColumnDescriptor("salaryDet");  
 columnFamily2.setMaxVersions(3);  
  
 hTableDescriptor.addFamily(columnFamily2);  
 //添加我们开发的Observer协调器  
 hTableDescriptor.addCoprocessor(AdminObserver.class.getCanonicalName(), new Path(path),  
 Coprocessor.PRIORITY\_USER, null);  
 admin.modifyTable(tableName, hTableDescriptor);  
 admin.enableTable(tableName);  
 }  
}

卸载Coprocessor代码：主要方法HTableDescriptor 下的 removeCoprocessor

import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.hbase.HBaseConfiguration;  
import org.apache.hadoop.hbase.HColumnDescriptor;  
import org.apache.hadoop.hbase.HTableDescriptor;  
import org.apache.hadoop.hbase.TableName;  
import org.apache.hadoop.hbase.client.Admin;  
import org.apache.hadoop.hbase.client.Connection;  
import org.apache.hadoop.hbase.client.ConnectionFactory;  
import org.myname.hbase.coprocessor.service.AdminObserver;  
  
import java.io.IOException;  
  
/\*\*  
 \* Coprocessor协调器通过java动态卸载代码  
 \*/  
public class CoprocessorUninstall {  
 public static void main(String[] args) throws IOException {  
 TableName tableName = TableName.valueOf("Users");  
 //连接HBase的zk集群  
 Configuration conf = HBaseConfiguration.create();  
 conf.set("hbase.zookeeper.quorum","Hadoop.Master:2181,Hadoop.Slave1:2181,Hadoop.Slave2:2181");  
 Connection connection = ConnectionFactory.createConnection(conf);  
  
 Admin admin = connection.getAdmin();  
 admin.disableTable(tableName);  
  
 HTableDescriptor hTableDescriptor = new HTableDescriptor(tableName);  
 HColumnDescriptor columnFamily1 = new HColumnDescriptor("personalDet");  
 columnFamily1.setMaxVersions(3);  
 hTableDescriptor.addFamily(columnFamily1);  
  
 HColumnDescriptor columnFamily2 = new HColumnDescriptor("salaryDet");  
 columnFamily2.setMaxVersions(3);  
  
 hTableDescriptor.addFamily(columnFamily2);  
 //移除该表下面的指定Observer协调器  
 hTableDescriptor.removeCoprocessor(AdminObserver.class.getCanonicalName());  
  
 admin.modifyTable(tableName, hTableDescriptor);  
 admin.enableTable(tableName);  
 }  
}

-------------------------------------------------------------------------------

Hbase 0.92 版本出现了很多新功能引入了协处理器(coprocessors)，实现一些激动人心的新特性：能够轻易建立二次索引、复杂过滤器(谓词下推)以及访问控制等)

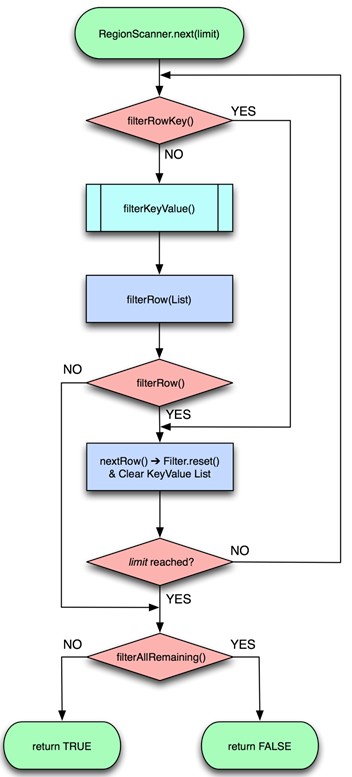
谓词下推：通俗理解，就是在实际数据读取和SQL实际执行之前预先执行条件语句（where）进行预处理和过滤。

**注：HBase存储的RowKey以及列族、每个列族中的列名 存储时都是按照字符串的字典排序后进行存储。**

-------------------------------------------------------------------------------

自定义过滤器：完整用例请参考hbaseFilterServer（服务端）项目以及myhbase（客户端）项目

HBase Filter执行流程图如下

[](http://photo.blog.sina.com.cn/showpic.html#blogid=7431c7c50101c5ig&url=http://s8.sinaimg.cn/orignal/7431c7c5gd34ef35632f7)

首先，学习自定义Filter之前，我们需要了解的几个必须类和方法：

filterRowKey(byte[],int,int): true的意思是放弃这一行；false的意思包括。

filterKeyValue(Cell): 决定是否包括或排除这个键值。请看Filter.ReturnCode。

filterRowCells(List): 允许直接修改提交的最终列表。

filterRow(): 最后一次基于筛选器调用序列删除整行的机会。Eg：如果一行不包含指定的列，则过滤它。

transform(KeyValue): 如果keyValue在内，让过滤器改变keyValue。

reset() : 在过滤新行之前重置筛选器状态。

filterAllRemaining(): true 的意思是行扫描结束，false意味着继续。

ReturnCode 枚举

INCLUDE 添加kv

INCLUDE\_AND\_NEXT\_COL 包含kv并查找下一列跳过旧版本的kv。

SKIP 跳过这个kv

NEXT\_COL 跳过此列。转到这一行的下一列。

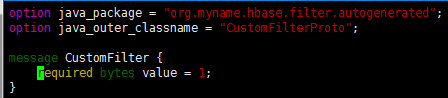
NEXT\_ROW 用列完成，跳到下一行。注意，filterRow()仍然会被调用。

SEEK\_NEXT\_USING\_HINT 寻找下一个作为过滤器提示的键。

自定义Filter相比自定义Coprocessor要简单些，因为自定义Filter不需要在HBase中注册。开发完成直接将jar包上传到HBase的lib目录下，或者HBASE\_CLASSPATH指定目录下即可。

下面我们来实现一个基于行的简单的Filter：通过传入指定RowKey，返回指定行数据，类似get的功能。

首先，我们通过自定义protocol 的RPC Java文件，实现客户端和服务端的参数传输：



这里有一个细节，就是待传入的参数value 的类型为 byte[]，在protocol如何表示的问题，当时纠结好久。。

Protocol类型与Java类型作比较:

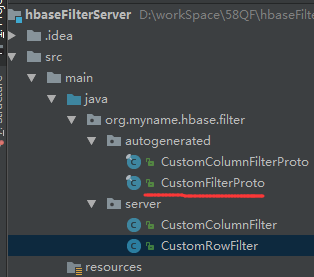


上面看到，对于Java中的byte，没有对应，参考了PrefixFilter的源码后，我们知道了，HBase中用ByteString来表示byte类型，但好像有不太对Protocol中是bytes啊。。。多了个s

经过一番誓言与设想以及源码的了解，最后确定了Protocol中的bytes 就能够对应Java中的byte[]

然后通过protocol命令将proto文件生成Java类。

新创建项目，将生成的protocol Java类引入到我们的项目中：



编写Filter实现代码：

package org.myname.hbase.filter.server;  
  
import com.google.protobuf.ByteString;  
import com.google.protobuf.InvalidProtocolBufferException;  
import org.apache.hadoop.hbase.Cell;  
import org.apache.hadoop.hbase.exceptions.DeserializationException;  
import org.apache.hadoop.hbase.filter.Filter;  
import org.apache.hadoop.hbase.filter.FilterBase;  
import org.myname.hbase.filter.autogenerated.CustomFilterProto;  
  
import java.io.IOException;  
import java.util.Arrays;  
  
/\*\*  
 \* 自定义Filter，实现了一个类似于get操作的Filter，传入一个rowkey，将对应行的数据返回。  
 \* 这是一个针对行的自定义Filter  
 \*/  
public class CustomRowFilter extends FilterBase {  
 private byte[] value = null;//传入的rowkey字节数组  
 private boolean filterRow = true;//整体标志位  
  
 /\*\*  
 \* 构造方法  
 \* @param value  
 \*/  
 public CustomRowFilter(byte[] value){  
 this.value = value;  
 }  
  
 /\*\*  
 \* 决定当前操作数据，进行数据是否添加到返回结果集的操作判断  
 \* 可以针对数据进行二次校验操作  
 \* ReturnCode 枚举  
 \* INCLUDE 添加kv  
 \* INCLUDE\_AND\_NEXT\_COL 包含kv并查找下一列跳过旧版本的kv。  
 \* SKIP 跳过这个kv  
 \* NEXT\_COL 跳过此列。转到这一行的下一列。  
 \* NEXT\_ROW 用列完成，跳到下一行。注意，filterRow()仍然会被调用。  
 \* SEEK\_NEXT\_USING\_HINT 寻找下一个作为过滤器提示的键。  
 \*  
 \* @param cell  
 \* @return  
 \* @throws IOException  
 \*/  
 @Override  
 public ReturnCode filterKeyValue(Cell cell) throws IOException {  
 if (this.filterRow == false) {  
 return ReturnCode.INCLUDE;//包含  
 }else {  
 return ReturnCode.NEXT\_ROW;//下一行  
 }  
 }  
  
  
 /\*\*  
 \* buffer - 包含行键的缓冲区  
 \* offset - 在缓冲区中距离行键开始时的偏移量  
 \* length - 行键的长度  
 \*  
 \* 针对每条数据进行筛选的具体实现方法  
 \*  
 \* false表示将当前这一行，加入返回结果集  
 \* true表示将当前这一行，不加入返回结果集  
 \*  
 \* @param buffer  
 \* @param offset  
 \* @param length  
 \* @return  
 \* @throws IOException  
 \*/  
 @Override  
 public boolean filterRowKey(byte[] buffer, int offset, int length) throws IOException {  
 byte[] rowKey = Arrays.copyOfRange(buffer, offset, offset + length);  
 String str = new String(rowKey);  
 if (new String(value).equals(str))  
 { // 在0(1)时间内返回，效率较高  
 this.filterRow = false;  
 }  
 return this.filterRow;  
 }  
  
 /\*\*  
 \* 每行完成之后重置标志位  
 \* 该方法每次验证完一行数据都会执行一次  
 \*  
 \*/  
 @Override  
 public void reset()  
 {  
 this.filterRow = true;  
 }  
  
 /\*\*  
 \* 返回该行是否被过滤  
 \* @return  
 \*/  
 @Override  
 public boolean filterRow()  
 {  
 return filterRow;  
 }  
  
  
 /\*\*  
 \* 必须重新实现的方法，参数序列化  
 \* 实现方式大致相同，只是Proto Java类不同  
 \*  
 \* @return  
 \* @throws IOException  
 \*/  
 @Override  
 public byte[] toByteArray() throws IOException {  
 CustomFilterProto.CustomFilter.Builder builder = CustomFilterProto.CustomFilter.newBuilder();  
  
 builder.setValue(ByteString.copyFrom(this.value));  
  
 return builder.build().toByteArray();  
 }  
  
 /\*\*  
 \* 必须重新实现的方法，参数反序列化  
 \* 实现方式大致相同，只是Proto Java类不同  
 \*  
 \* @param pbBytes  
 \* @return  
 \* @throws DeserializationException  
 \*/  
 public static Filter parseFrom(byte[] pbBytes) throws DeserializationException {  
 CustomFilterProto.CustomFilter proto;  
 try {  
 proto = CustomFilterProto.CustomFilter.parseFrom(pbBytes);  
 } catch (InvalidProtocolBufferException e) {  
 throw new DeserializationException(e);  
 }  
 byte[] rowKey = proto.getValue().toByteArray();  
 return new CustomRowFilter(rowKey);  
 }  
}

需要注意的是，自定义的filter必须实现两个方法：

/\*\*  
 \* 必须重新实现的方法，参数序列化  
 \* 实现方式大致相同，只是Proto Java类不同  
 \*  
 \* @return  
 \* @throws IOException  
 \*/  
@Override

public byte[] toByteArray() throws IOException{}

/\*\*  
 \* 必须重新实现的方法，参数反序列化  
 \* 实现方式大致相同，只是Proto Java类不同  
 \*  
 \* @param pbBytes  
 \* @return  
 \* @throws DeserializationException  
 \*/

public static Filter parseFrom(byte[] pbBytes) throws DeserializationException{}

将项目打成jar包，发送到每个HBase节点的lib目录下并重启HBase集群。

客户端测试服务，需要将自定义Filter的项目生成的jar包引入到项目中，进行调用



总结：

与自定义协调器有所不同，自定义Filter的 Proto定义的RPC Java类只负责想服务其传输参数，并没有想自定义协调器定义方法。

org.apache.hadoop.hbase.Cell 该类表示一个行数据的单元，针对列操作的Filter需要用到下面几个方法（具体用例参考hbaseFilterServer项目中的CustomColumnFilter）通过下面几个方法以及filterRowKey(byte[] buffer, int offset, int length)方法，可以触及到每行中的每列，并进行列操作。

cell.getQualifierArray() 获取HBase中所有列的byte[]数据  
cell.getQualifierOffset() 获取HBase中针对列的偏移量  
cell.getQualifierLength() 获取HBase中针对列的长度

方法filterRowKey：

/\*\*  
 \* buffer - 包含行键的缓冲区  
 \* offset - 在缓冲区中距离行键开始时的偏移量  
 \* length - 行键的长度  
 \*  
 \* 针对每条数据进行筛选的具体实现方法  
 \*  
 \* false表示将当前这一行，加入返回结果集  
 \* true表示将当前这一行，不加入返回结果集  
 \*/

public boolean filterRowKey(byte[] buffer, int offset, int length) throws IOException

-------------------------------------------------------------------------------

自定义比较器：