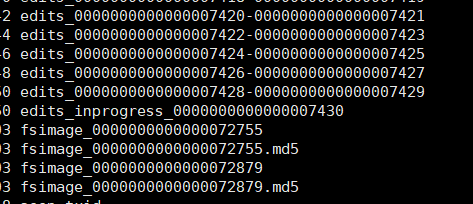
# HDFS

## editLog 和 FsImage

名称节点负责管理命名空间，保存了两个数据结构：FsImage 和 EditLog

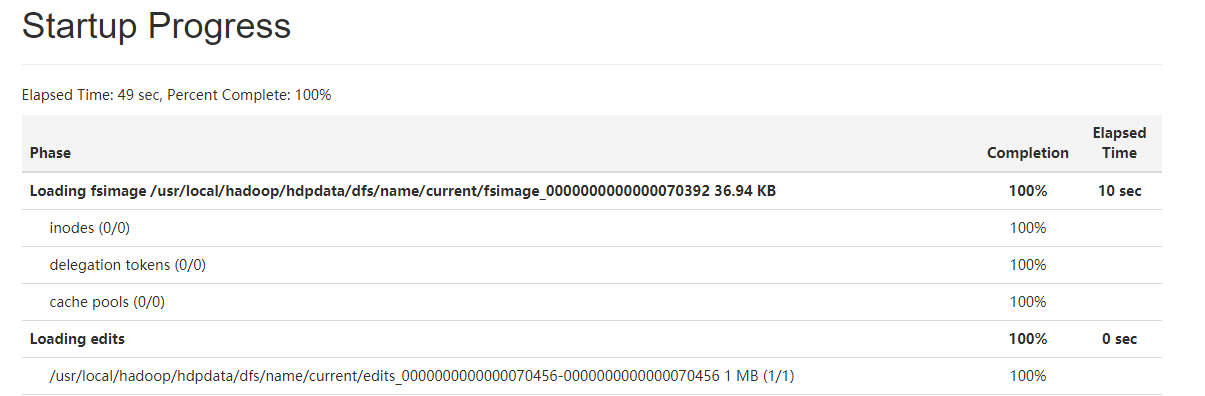
FsImage用于维护文件系统树以及文件树中所有的文件和文件夹的元数据。

EditLog中记录了所有针对文件的创建、删除和重命名等操作。



名称节点中记录了每个文件中各个块所在的数据节点的位置信息，但是并不会持久化存储这些信息，而是在系统每次启动时扫描所有数据系欸但重构得到这些信息。

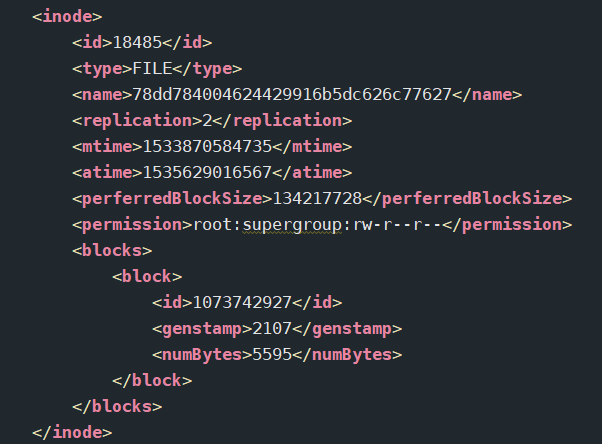
名称节点启动时，将FsImage的内容加载到内存中，然后执行EditLog文件中的各项操作，使得内存中的元数据保持最新，然后会创建一个新的FsImage文件和一个空的EditLog文件。

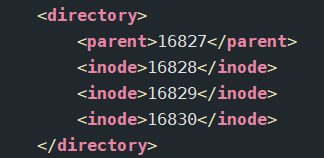


名称节点启动时会进入安全模式，只读。

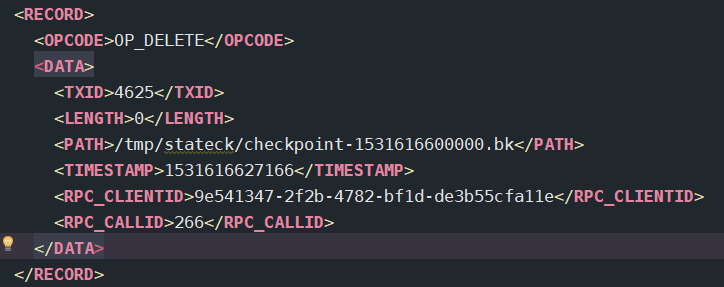
Hdfs的更新操作先写入EditLog，每隔一段时间再滚动EditLog，与FsImage进行合并。

FsImage的实际保存形式，其内部使用inode去维护每个文件或者目录的信息。其保存了文件id，类型，名称，副本数，创建和修改时间，权限，块大小，块id等信息。





FsImage



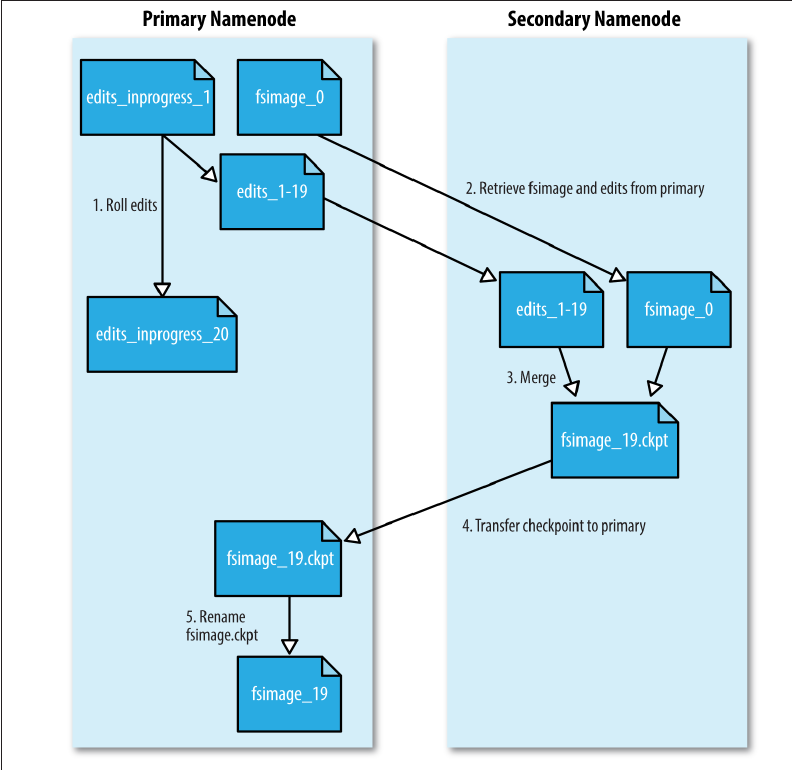


editLog

## 第二名称节点

完成EditLog与FsImage的合并，减小EditLog的大小，缩短名称空间的启动时间。

可以作为名称节点的检查点，保存名称节点的元数据信息。



## hadoop启动命令

$>start-all.sh //start-dfs.sh + start-yarn.sh

$>start-dfs.sh //hadoop-daemon.sh start namenode

//hadoop-daemons.sh start datanode

$>start-yarn.sh //yarn-daemon.sh start resourcemanager

//yarn-daemons.sh start nodemanager

hadoop-daemon.sh 可以在单个节点中进行操作，无需每次都要对整个集群进行操作

## hdfs dfsadmin

**hdfs的管理员工具**

Note: Administrative commands can only be run as the HDFS superuser.

[-report [-live] [-dead] [-decommissioning]]

**[-safemode <enter | leave | get | wait>]**

**[-saveNamespace]**

**[-rollEdits]**

[-restoreFailedStorage true|false|check]

[-refreshNodes]

**[-setQuota <quota> <dirname>...<dirname>]**

**[-clrQuota <dirname>...<dirname>]**

**[-setSpaceQuota <quota> [-storageType <storagetype>] <dirname>...<dirname>]**

**[-clrSpaceQuota [-storageType <storagetype>] <dirname>...<dirname>]**

[-finalizeUpgrade]

[-rollingUpgrade [<query|prepare|finalize>]]

[-refreshServiceAcl]

[-refreshUserToGroupsMappings]

[-refreshSuperUserGroupsConfiguration]

[-refreshCallQueue]

[-refresh <host:ipc\_port> <key> [arg1..argn]

[-reconfig <datanode|...> <host:ipc\_port> <start|status>]

[-printTopology]

[-refreshNamenodes datanode\_host:ipc\_port]

[-deleteBlockPool datanode\_host:ipc\_port blockpoolId [force]]

[-setBalancerBandwidth <bandwidth in bytes per second>]

[-fetchImage <local directory>]

**[-allowSnapshot <snapshotDir>]**

**[-disallowSnapshot <snapshotDir>]**

[-shutdownDatanode <datanode\_host:ipc\_port> [upgrade]]

[-getDatanodeInfo <datanode\_host:ipc\_port>]

[-metasave filename]

[-triggerBlockReport [-incremental] <datanode\_host:ipc\_port>]

[-help [cmd]]

### safemode

hdfs dfsadmin –savemode

安全模式下名称节点只读，不能复制和删除块。

名称节点启动时自动进入安全模式，当满足最小副本数的百分比时间自动退出。也可以手动进入和退出。

### [-saveNamespace]

保存名称空间 保存当前的名称空间到存储目录中，并且重置editlog

### [-setQuota <quota> <dirname>...<dirname>]

设置目录下的所有文件的总个数。如果1，表示空目录。

$>hdfs dfsadmin -setQuota 1 dir1 dir2 //设置目录配额

$>hdfs dfsadmin -clrQuota 1 dir1 dir2 //清除配额管理

### [-setSpaceQuota <quota> [-storageType <storagetype>] <dirname>...<dirname>]

计算目录下的所有文件的总大小.包括副本数.

空间配置至少消耗384M的空间大小(目录本身会占用384M的空间)。

$>hdfs dfsadmin -setSpaceQuota 3 data

$>echo -n a > k.txt

$>hdfs dfs -put k.txt data2

$>hdfs dfsadmin -clrSpaceQuota dir1 //清除配额管理

### [-allowSnapshot <snapshotDir>]

迅速对文件(夹)进行备份。不产生新文件，使用差值存储。默认是禁用快照,先启用。

创建快照不会产生新的文件，会在editlog中保存操作

$>hdfs dfsadmin -allowSnapShot dir1 //在dir1启用快照

$>hdfs dfsadmin -disallowSnapShot dir1 //在dir1启用快照

$>hdfs dfs -createSnapshot dir ss1 //创建快照

$>hdfs dfs -renameSnapshot dir ss1 ss2 //重命名

$>hdfs dfs -deleteSnapshot dir ss1 //删除快照

### 节点的服役和退役

[添加新节点]

1.在dfs.include文件中包含新节点名称,该文件在nn的本地目录。

[白名单]

[s201:/soft/hadoop/etc/dfs.include.txt]

s202

s203

s204

s205

2.在hdfs-site.xml文件中添加属性.

<property>

<name>dfs.hosts</name>

<value>/soft/hadoop/etc/dfs.include.txt</value>

</property>

3.在nn上刷新节点

$>hdfs dfsadmin -refreshNodes

4.在slaves文件中添加新节点ip(主机名)

s202

s203

s204

s205 //新添加的

5.单独启动新的节点中的datanode

[s205]

$>hadoop-daemon.sh start datanode

[退役]

1.添加退役节点的ip到黑名单,不要更新白名单.

[/soft/hadoop/etc/dfs.hosts.exclude.txt]

s205

2.配置hdfs-site.xml

<property>

<name>dfs.hosts.exclude</name>

<value>/soft/hadoop/etc/dfs.hosts.exclude.txt</value>

</property>

3.刷新nn节点

$>hdfs dfsadmin -refreshNodes

4.查看webUI,节点状态在decommisstion in progress.

5.当所有的要退役的节点都报告为Decommissioned,数据转移工作已经完成。

6.从白名单删除节点,并刷新节点

[s201:/soft/hadoop/etc/dfs.include.txt]

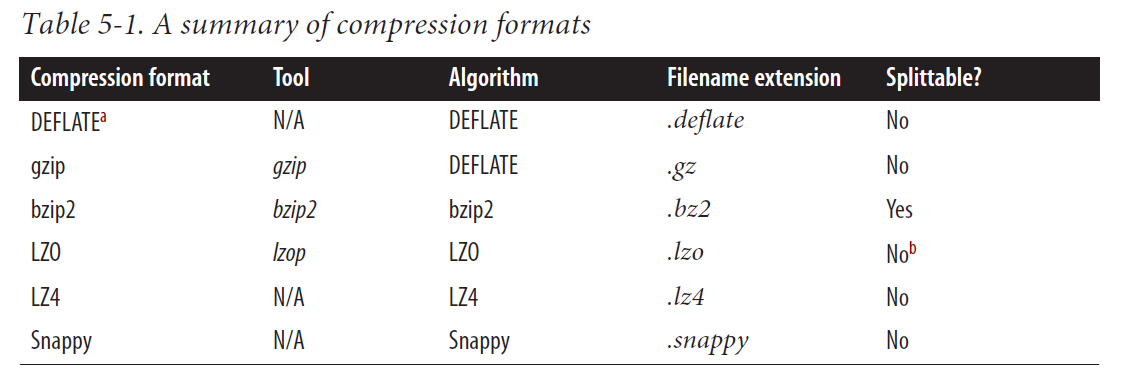
...

$>hdfs dfsadmin -refreshNodes

7.从slaves文件中删除退役节点

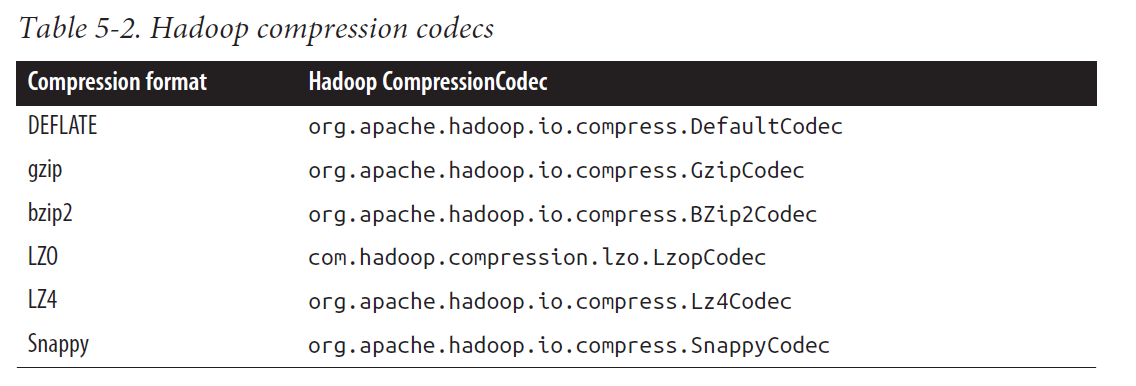
## hadoop中的文件压缩

文件压缩的两个优点：1.降低存储空间 2.占用带宽，加快网络传输速率（权威指南P104）



使用到的类

Codecs是解压缩算法的实现。



## 基于文件的数据结构——SequenceFile

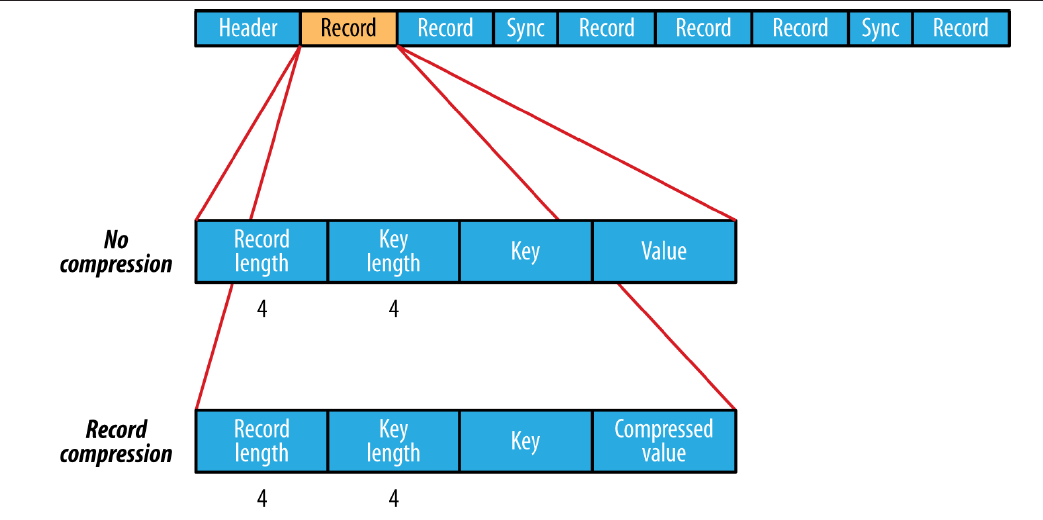
SequenceFile提供一个二进制key-value对的持久化数据结构。例如对于一个日志文件格式，可以把时间作为key，日志记录作为value(Writable子类)。

SequenceFile结构可以使*小文件*更加高效。

***sync points*** （同步点）– A sync point is a point in the stream that can be used to resynchronize with a record boundary if the reader is “lost”—for example, after seeking to an arbitrary position in the stream. 同步点由SequenceFile.writer写入，它是随机的，以小于1%的存储空间存储在文件流中便于索引。同步点总是在记录边界之间的，而不会写在key和value之间。调用reader.sync(position)后，reader将把索引移动到position之后的下一个同步点，然后从这个同步点开始的下个record边界开始读取key-value。

使用reader.seek(position)时，如果position的位置不在边界则会获取失败。

**SequenceFile的格式**。

*The internal structure of a sequence file with no compression and with record compression*

## 基于文件的数据结构——MapFile

A MapFile is a sorted SequenceFile with an index to permit lookups by key.

Mapfile是SequenceFile的一个有序版本。

MapFile中包含两个SequenceFile，一个是index，一个是data。

data中包含了左右数据，而数据是按key有序排列的。

index中包含了部分map中的部分key(每128个key建立一个索引)，index可以从data中加载数据进内存。

## HDFS的高可用(High Availability -- HA)

使用第二名称节点可以创建检查点防止数据丢失，但它不能实现文件系统的高可用，名称节点依然会存在单点故障问题(a single point of failure)。所谓的高可用就是在一个节点故障（宕机）时，可以自动切换到另外一个备份节点(standy)。

Hadoop 2 开始加入了HDFS的高可用。有两个名称节点 active-standy。如果active节点故障了，那么standy节点会自动接管客户端的请求，这期间不会产生一个明显的中断。

* namenodes达到HA必须共享editLog的存储。standby节点启动之后，它会同步读取active节点的editlog，并且继续读取active中新的操作。
* datanode必须把块报告发送给两个节点，因为块映射存储在节点的内存中，而不是硬盘上。
* 客户端必须进行配置，处理namenode failover（故障切换）。
* 第二名称节点的角色被纳入stanby节点，它会周期性检查active namenode

HA的共享存储有两种方式：NFS Filter 和 ***quorum journal manager***(QJM)。 QJM是专用的HDFS高可用实现，它唯一的功能是提供editlog的高可用，也是HDFS推荐使用的。

QJM以一组journal nodes的方式运行，每次edit操作必须写入大部分的journal nodes。一般情况有3个journal nodes，因此系统可以容错其中一个。***这个运行机制像zookeeper，不过QJM不是利用ZK实现的。（注意：HDFS的HA是用ZK选举出active节点的，这个和journal nodes没有关系）***

如果active节点宕机了，那么standby节点会迅速地接管（几十秒内），因为它内存中保留了最新的状态：最新的editlog和block mapping。

### Failover（容错）and fencing

故障切换是通过failover controller管理的。failover controller默认的实现是使用ZK，确保只有一个节点是active状态。每个namenode运行一个轻量级的Failover controller进程，它会监测名称节点的故障（使用心跳机制），namenode故障会触发failover。

有时由于网络延迟会导致故障切换，但是原来的active依然运行着并且以为自己是active的，HA的实现会避免对这个节点的破环，因此需要使用***fencing***。 QJM某一时刻只允许一个namenode写入editlog。然而，原来的active节点依然会读取客户端的请求，因此要使用SSH fencing 命令杀死该进程。

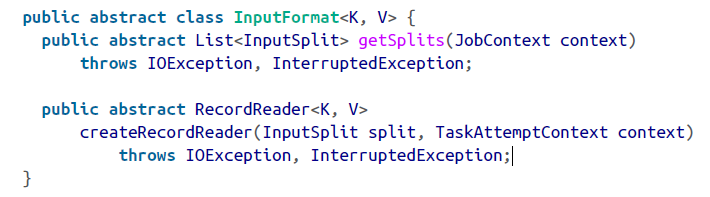
# MapReduce

## 输入格式

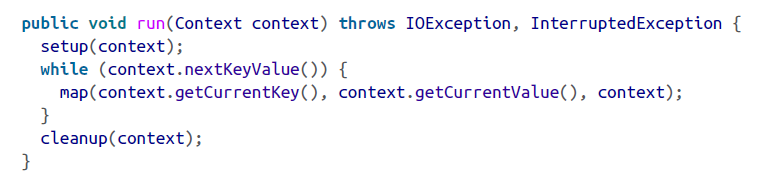
### Input Splits and Records

每个map任务拥有一个split，每个split分成多个record，而map任务以键值对的方式获取record。split和record都是逻辑分割，它们并不是真正的从物理上切割文件。在数据库的环境下，一个切片相当于表中某个范围内的行记录。

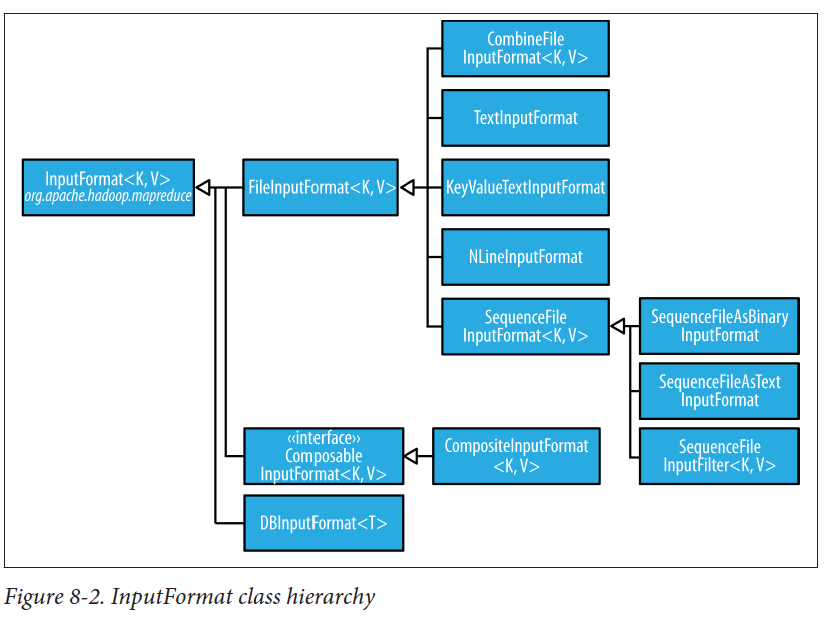
在Hadoop中，是由InputFormat创建切片的，它把创建切片并且把它分成多个记录。

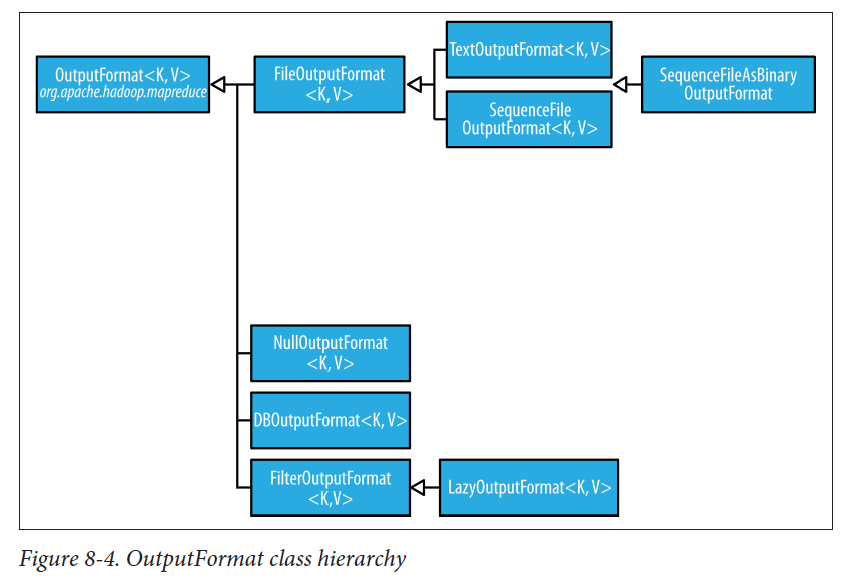


InputFormat中，getSplits()方法用于计算作业切片，然后把它们发送到application master，application master会使用存储路径规划map任务，map任务把split传入createRecorderReader()方法获取一个RecordReader，RR不仅是一个record的迭代器，map任务使用它生成key-value对，传到Mapper的map()函数中。



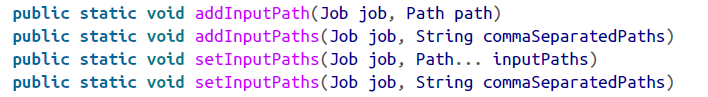
run()方法运行时，context.nextkeyValue()方法会不断迭代，把key-value对填入mapper中，key-value是从RR中取回放置到context中的。





### FileInputFormat

FileInputFormat是文件输入的基类。（抽象类，继承InputFormat）



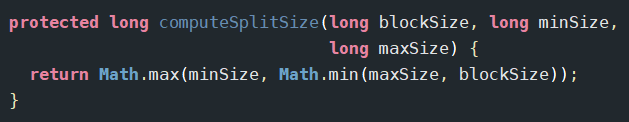
**setInputPathFilter()**可以用于在输入路径中排除某个文件

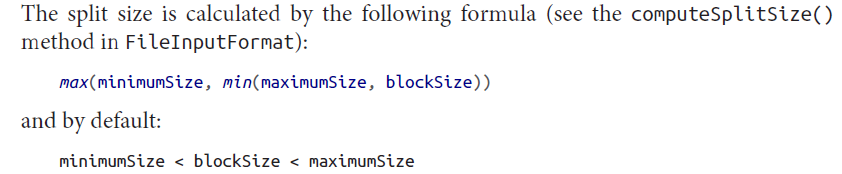
FileInputFormat的切片。给定若干文件，FileInputFormat如何切片？

FileInputFormat一般只切分大文件（意味着比HDFS的文件块大的文件）。***一般切片的大小就是HDFS块的大小***，可使用大部分场景应用。如果是小文件，一个文件就对应一个split。

***最小切片大小为1字节。最大为Java中Long的大小。***

FileInputFormat中计算切片大小的方法





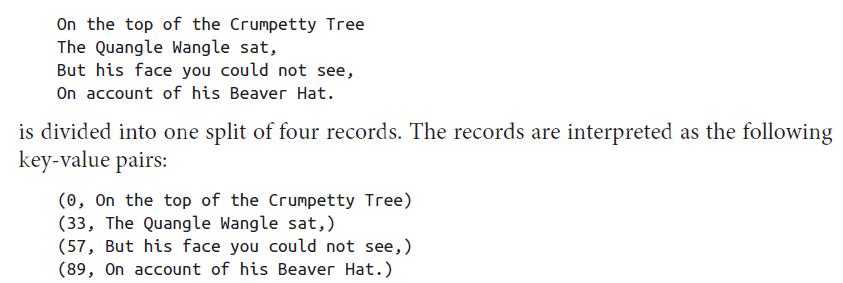
因此这里取的是中间值。

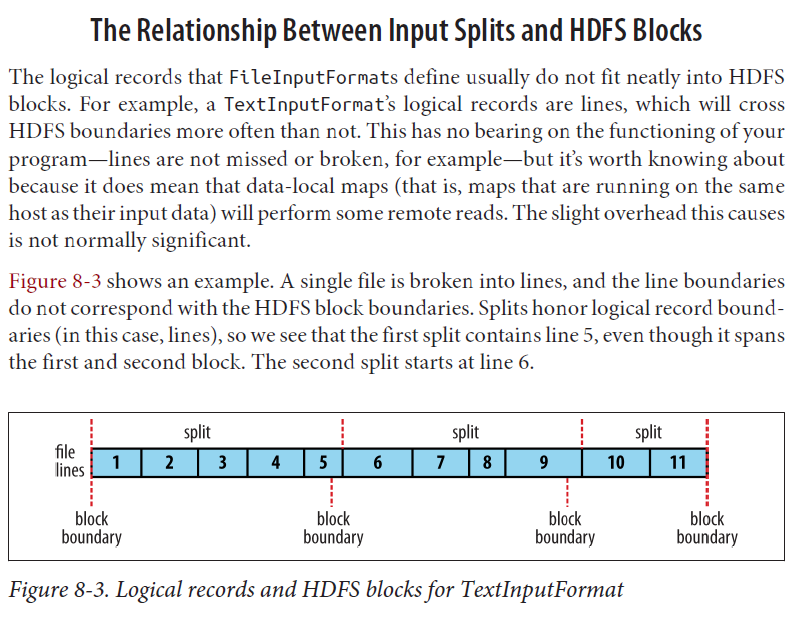
小文件问题

如果很多文件不足块大小，那么这些文件各自独立成一个split。假设一个1G的大文件分成128M的切片，会有8个split；而如果有1000个100k的文件，那么就要1000个split，作业时间相差10-100倍。

### Text Input

TextInputFormat是默认的InputFormat。每个record是输入的一行，key是LongWritable，字节偏移量，value是一行的内容。





**KeyValueTextInputFormat**可以直接读取一行文本中的key和value。

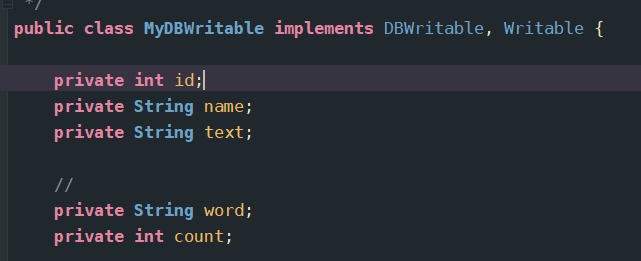
比如针对一行的数据hello world ，可以直接将hello读取为key，world读取为value，中间的分割符可以自定义，默认是\t。

### Database Input (and Output)

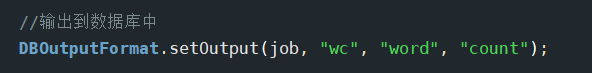
DBInputFormat是从数据库读取数据的输入格式，底层使用的是jdbc。

DBOutputFormat可以把输出写入数据库中。

需要自定义Writable，继承DBWritable,Writable，类中声明的数据库的字段。并且实现相关的方法。







## MapReduce各个执行阶段

（1）MR框架使用InputFormat模块做Map前的预处理，将输入文件切分为逻辑上的多个InputSplit，InputSplit是MapReduce对文件进行处理和运算的输入单位，只是一个逻辑概念，它并没有对文件进行实际的物理切割，只是记录了要处理的数据位置和长度。分片split确定的是Map的任务数，而最终每个Map是否有实际的任务是不确定的。

***例如，读取一行文本，如果分片比行偏移量要小，RR在读取时依然会按照行来读取，而不会以split为单位进行读取。在读取下一个split时，RR会跳过第一行读取到的split位置之后的内容（因为上一个map已经读取了），而直接从下一行开始读取。这就体现了InputSplit只是逻辑上的切分，而实际的物理切分是由RR读取是决定的。***

（2）RecordReader（RR）根据InputSplit中的信息来处理InputSplit中的具体记录，加载数据并转换为合适的Map任务读取的键值对，输入给Map任务。

（3）Map任务会根据用户自定义的映射规则，输入一系列的<key,value>作为中间结果。

（4）为了让Reduce可以并行处理Map的姐u共，需要对Map的输出惊醒一定的分区、排序、合并、归并等操作，得到<key,value-list>形式的中间结果。

（5）Reduce以一些列的<key,value-list>中间结果作为输入，执行用户定义的逻辑，输出结果给OutputFormat模块。

（6）OutputFormat模块会验证输出目录是否已经存在以及输出结果类型是否符合配置文件中的配置类型，如果满足就输出Reduce的结果到Hdfs中。



MapReduce工作流程中的各个执行阶段

## Map端的Shuffle过程

**The term *shuffle* is actually imprecise, since in some contexts it refers to only the part of the process where map outputs are fetched by reduce tasks. In this section, we take it to mean the whole process, from the point where a map produces output to where a reduce consumes input.**

主要包括四个步骤

(1)输入数据和执行map任务

(2)写入缓存。map的输出结果首先写入缓存，积累一定数量之后再一次写入磁盘。

(3)溢写(分区、排序和合并)

MapReduce的缓存默认是100MB，当达到溢写比例后（一般为0.8，即80MB），启动溢写过程，剩余的20MB继续接收Map写过的写入。

溢写之前，需要经过分区、排序和合并。

**分区(Partition)**。缓存中的键值对最终要交给不同的Reduce任务进行并行处理。MapReduce通过Partitioner接口对这些键值进行分区，默认采用Hash函数对key进行哈希再用Reduce的任务数量进行驱魔，这样就可以把Map输出的结果均匀的分布再R个Reduce任务处理。

**排序(Sort)**。每个分区内的键值对，后台线程会根据key对它们进行内存排序，这是mapReduce的默认操作。

**合并(Combiner)**(*可选*)。合并指的是把相同的key的value值加起来。只是发生在map端的。Combiner需要用户事先定义，可以减少溢写到磁盘的数据量。Combiner绝不能改变Reduce任务的最终计算结果。Combiner其实是在map端预先进行reduce操作，因此在JavaAPI中Job.setCombinerClass(Class<?> extends Reduce)传入的是Reduce的类。

结果以上步骤，缓存中的键值对就可以被写入磁盘，并清空缓存。

每次溢写都产生一个新文件，每个文件内都是经过分区并有序的。

(4)文件归并。把溢写出来的文件进行归并(Merge)，生成一个大的溢写文件，这个文件所有的键值对都是经过分区和排序的。相同的key会被合并成一个新的键值对。即<k1,<v1,v2,v3,..>>,<k2,<v1,v2,..>>。

经过以上四步，Map端的shuffle过程全部完成，最终生成一个大文件存储到本地磁盘，这个大文件的数据是被分区的，最终要被发送到不同的Reduce任务进行并行处理。



## Reduce端的Shuffle过程

（1）领取数据。

Map端的Shuffle过程结束后，Reduce任务需要把这些数据领取(Fetch)回来存放到自己所在的机器上。每个Reduce任务会不断地通过RPC向JobTracker询问Map任务是否已经完成，一个Map任务完成之后，JobTracker会检测到并且通知相关的Reduce任务来领取数据。一般Reduce会启动多个线程同时从不同的Map机器中领取数据。

（2）归并数据。

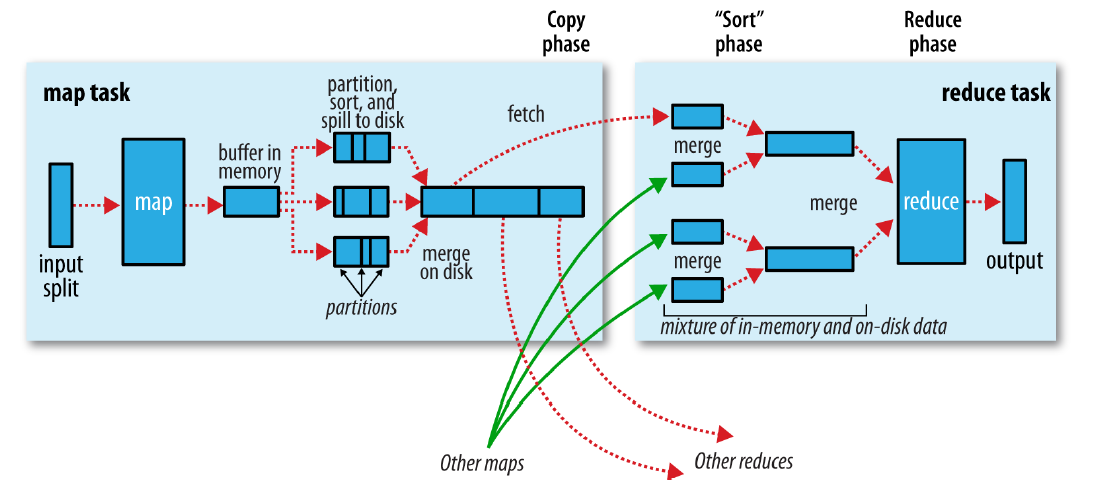
领取回来的数据同样要先保存的Reduce端的缓存中，沾满后溢写。缓存中的数据来自不同的Map机器，一般会存在很多相同的key，可以进行Combiner操作。溢写启动时，相同的key 的键值对会并归并(Merge)，如果定义了Combiner，归并后还可以执行合并操作。每个溢写结束，磁盘中都会生成新的溢写文件。所有的Map数据被领回之后，多个溢写文件会归并成一个大文件，归并时会进行排序。

把多个溢写文件归并成大文件时可能需要多轮归并。每次归并的文件数量由io.sort.factor确定。

（3）把数据输入给Reduce任务。

归并之后的若干个大文件，不会继续归并成新的大文件，而是直接输入给Reduce热任务。





shuffle过程

## Local模式运行MR流程

1.创建外部Job(mapreduce.Job),设置配置信息

2.通过jobsubmitter将job.xml + split等文件写入临时目录

3.通过jobSubmitter提交job给localJobRunner,

4.LocalJobRunner将外部Job 转换成成内部Job

5.内部Job线程，开放分线程执行job

6.job执行线程分别计算Map和reduce任务信息并通过线程池孵化新线程执行MR任务。

## 集群环境下运行MR流程

## 远程调试

使用windows客户端远程调试服务器上的程序

**1.设置服务器java vm的-agentlib:jdwp选项.**

[server]

//windwos

//set

JAVA\_OPTS=%JAVA\_OPTS%-agentlib:jdwp=transport=dt\_socket,address=8888,server=y,suspend=n

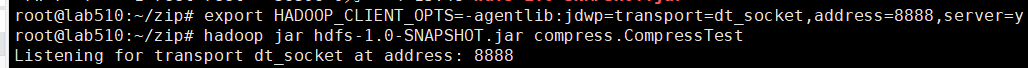
//linux

export HADOOP\_CLIENT\_OPTS=-agentlib:jdwp=transport=dt\_socket,address=8888,server=y,suspend=y

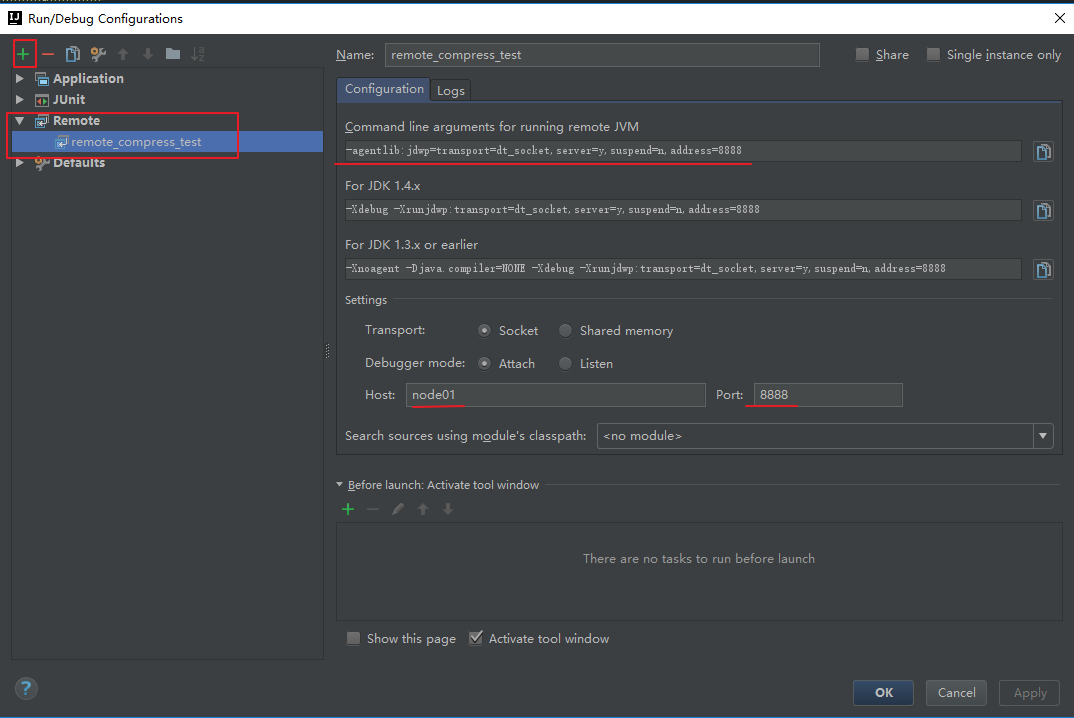
**2.在server启动java程序**

hadoop jar HdfsDemo.jar com.it18zhang.hdfs.mr.compress.TestCompress

**3.server会暂挂在8888.**

 Listening ...

**4.客户端(idea)通过远程调试连接到远程主机的8888.**



**5.添加断点，开启debug，客户端就可以调试了。**

## 使用mvn命令下载工件中的所有依赖

进入pom.xml所在目录，运行cmd：

mvn -DoutputDirectory=./lib

-DgroupId=com.it18zhang

-DartifactId=HdfsDemo

-Dversion=1.0-SNAPSHOT dependency:copy-dependencies

## 多输入问题

map的输入可能是多种格式的，例如文本格式、SequenceFile格式等，这时需要通过MulitipleInputs.addInputPath()添加路径，并且设置输入格式，对应的Mapper类，每个输入格式应该按实际情况对应一个Mapper类。

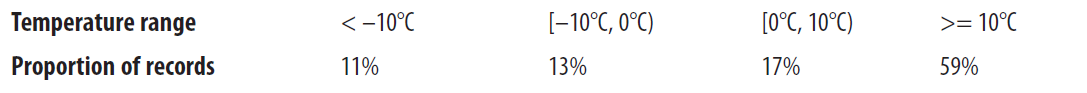
## 全排序

三种。在有多个reduce的情况下，会输出多个结果，part-r-00000,part-r-00001..

在全排序时，每个输出文件的key相对于前一个文件的key都是有序的。

1.只定义1个reduce。一般不用这种方式，因为大批量的数据会导致数据倾斜问题。

2.自定义分区函数 Partitioner，自定义分区区间。一般也不推荐，因为这个需要实现了解数据的内容，才可以手动地去分解区间。如果分区本来的数据分布就不均衡，那么就很难手动设置分区的范围。



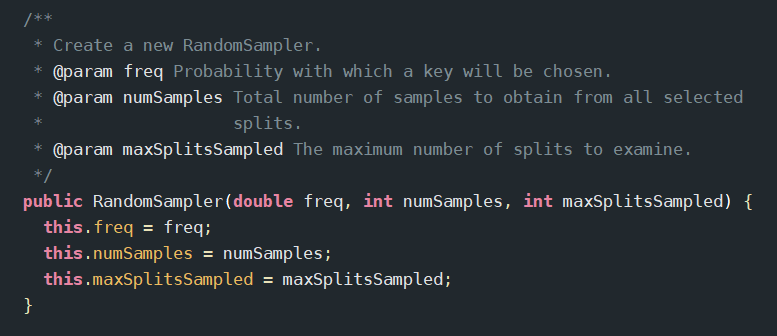
3.使用采样器。

为了更好地进行分区，我们需要更加了解整个数据的数据分布状态。在key空间分布中使用***采样***的方式进行相对公平的分区更加合理。

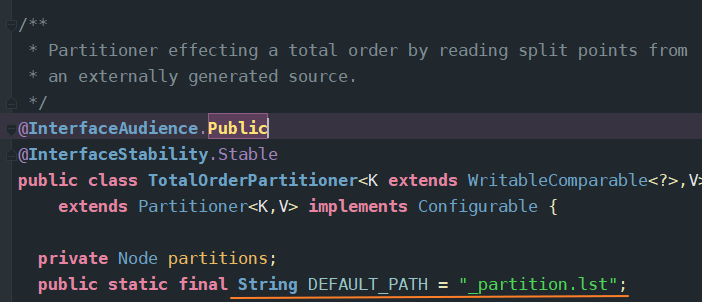
采样指的是观察数据集的子集去估算的获取key的分布状态，然后进行分区。Hadoop为我们实现的***采样器***。

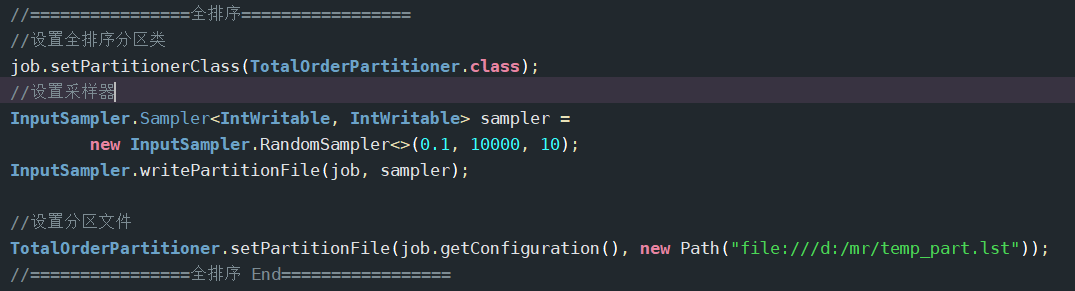
RandomSampler是InputSampler中的其中一个采样器（内部类）。

构造器参数包括采样率，总样本数，最大分片数。



TotalOrderPartitioner需要读取外部的区间序列文件(sequenceFile)进行分区。默认的位置在项目的根路径\_partition.lst。这个文件保存了分区的边界。



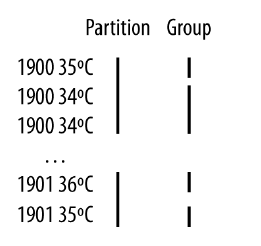


## 二次排序

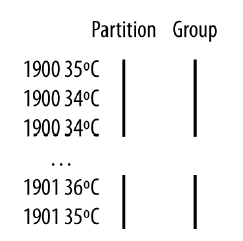
从Map端到Reduce端通常是按照key进行分区和排序的，而value值的出现是不稳定的，也就是说MR程序的写入不依赖于value。但是可以通过二次排序对value进行排序。

例如，如果有一批年份和对应的一些气温数据，需要找出最高气温，一般的做法是通过遍历某个年份的value值去获取该年份的最大值，如果通过二次排序就不需要进行value的遍历。这种情况可以通过自定义key进行：把年份和气温都放在key当中，然后利用key的有序性进行排序和转换。

通过自定义分区，保证相同年份的key进入同一个reduce中，但它只能保证同一个Reduce可以接收同一年的所有数据，但是并不能改变分区中的分组。



*因此需要控制分区中的分组情况，把所有同一年的数据放到同一个分组中。*这样就可以在一组中对根据气温对key进行排序，然后获取第一个或者最后一个值即最值。



综上，对value进行排序的方法如下

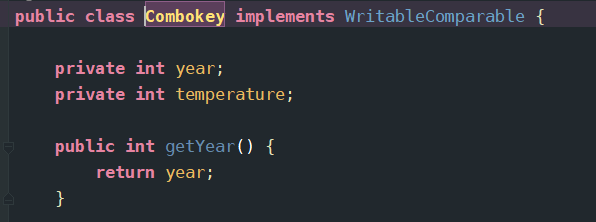
• Make the key a composite of the natural key and the natural value.

• The sort comparator should order by the composite key (i.e., the natural key *and* natural value).

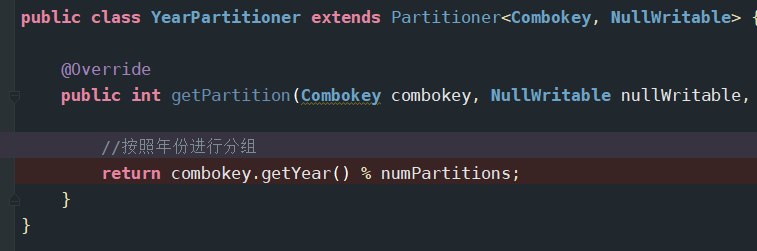
• The partitioner and grouping comparator for the composite key should consider

only the natural key for partitioning and grouping.

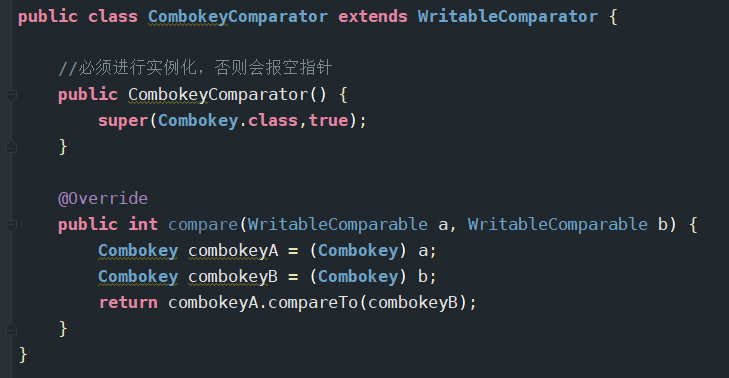
* 组合key，需要继承WritableComparable序列化类



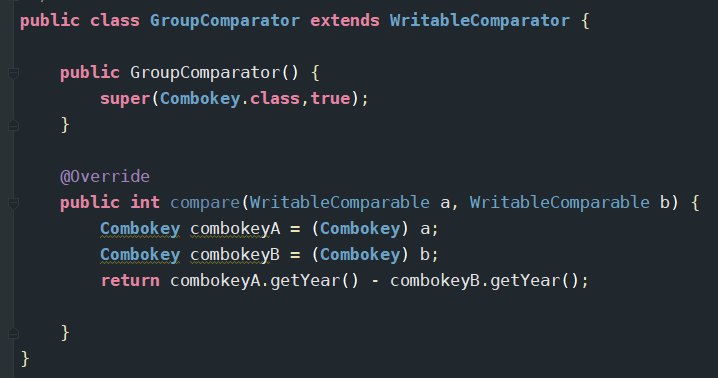
* 自定义的partitioner –把所有的相同年的key放到一个***reduce节点***中



* 自定义Sort Comparator ：对组合key进行排序，先按年，再按气温排序。compare方法的参数时WritableCompareble类型，否则不会进入此方法。

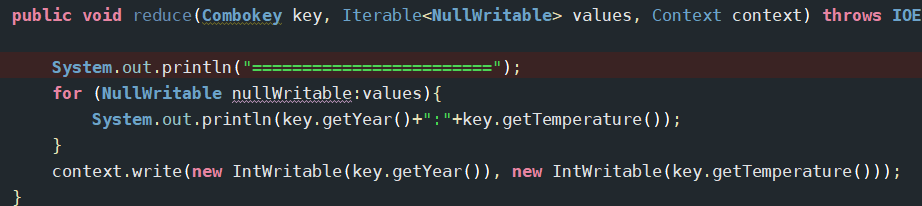


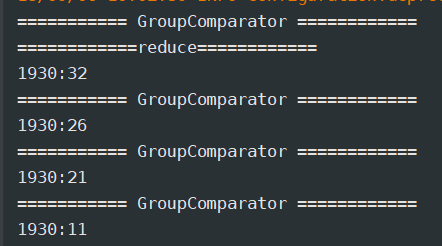
* 自定义group Comparator ：进入Reduce后，把相同年份的组合key放到同一个group中，这里的group是在reduce端。如果year相等就进入同一个组，否则降序排列。



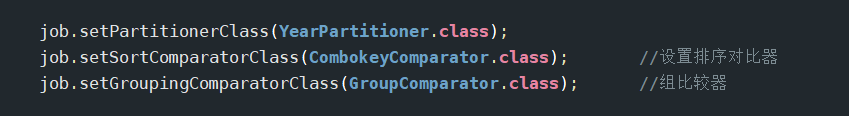
* 经过分组比较器之后（这是在***同一个Reduce节点***进行的），所有相同year的comboky键值对会聚合到同一个***reduce函数***中。

由于是倒序排列，因此第一次获取combokey的值，即可获取到最大值。





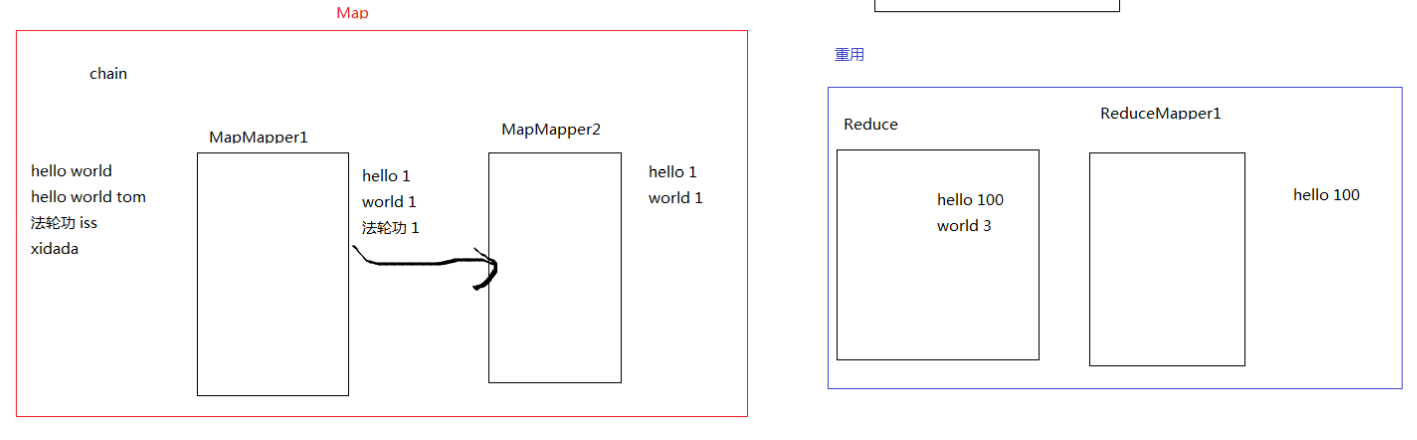
* 设置相关的类。



* **执行过程**：map产生输出->partitioner分区->组合key - combokeyComparator比较排序，这时map过程的shuffle端结束，然后根据partitioner进入不同的reduce节点中，每个节点又会根据分组比较器GroupComparator，根据自定义的比较条件，相同分组进入同一个reduce函数。
* **int compare(o1,o2)方法 返回正数：o1排在o2后面,返回负数：o1在o2后面。**

**自定义组合key比较器是在map端的分区结束后排序使用的，而分组比较器是在reduce节点上向reduce函数分发数据时使用的。**

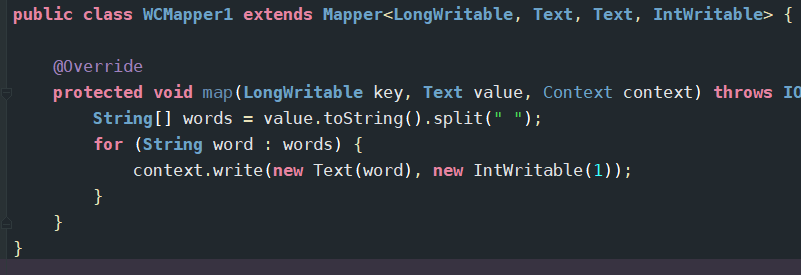
## 链式处理



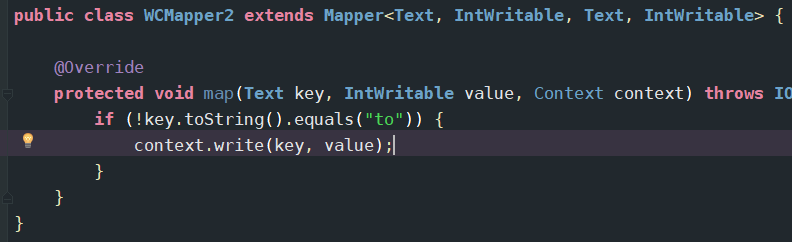
MapTask的链式处理：一个Map节点中存在多个map任务，而且一个map的输出是下一个map的输入，这样就形成了链式的Map。

ReduceTask的链式处理：只能有一个reduce任务，但可以在链式reduce中添加mapper，接收reduce端的输出。

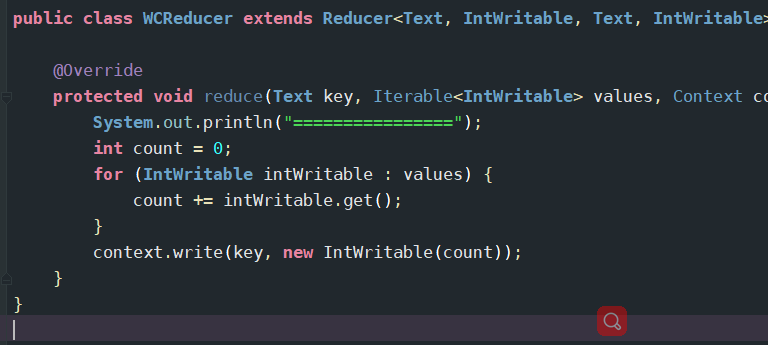
//WCMapper1进行分割



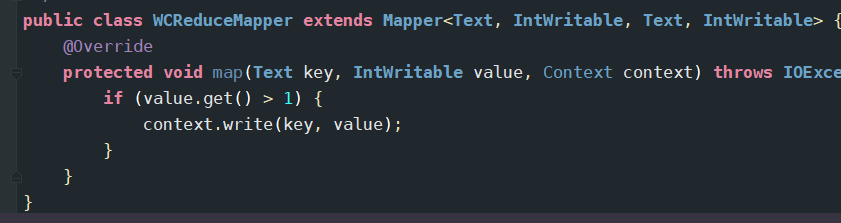
//WCMapper2进行过滤，把不需要的key值过滤，不输出到reduce端。



//Reducer进行正常的单词统计



//WCReduceMapper把单词出现少于1次的过滤掉



//通过ChainMapper和ChainReducer进行串联

