# 1 hadoop生态圈

Commom：一些列组件和接口，用于分布式系统和通用I/O(序列化、javaRpc和持久化数据结构)

Avro：一种序列化系统，用于支持高效、夸语言的RPC和持久化数据数据村吃。

MapReduce:分布式数据处理模型和执行环境，运行于大型商用机集群。

HDFS：分布式文件系统。

Pig：数据流语言和运行环境，用于研究非常庞大的数据集。Pig运行在MapReduce和HDFS集群上。

Hive：一种分布式、按列存储的数据仓库。Hive管理HDFS中存储的数据，请提供基于sql的查询语句（由运行时引擎翻译成mapreduce作业）用以查询数据。

HBase：一种分布式的，按列存储的数据库。HBase使用HDFS作为底层存储，并且支持mapreduce的批量式计算和点查询。

Zookeeper：一种分布式的、可用性高的协调性服务。Zookeeper提供分布式锁之类的基本服务用与构建分布式应用。

Sqoop：用于在结构化数据存储和HDFS之间高效批量的传输数据。

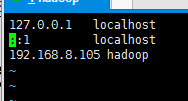
Oozie：该服务用于运行和调度Hadoop作业。

## 1.1 hadoop伪分布式搭建

基于2.7.1版本

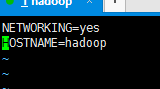
### 1.1.1 、更改/etc/hsots文件

vim /etc/hosts



### 1.1.2 更改/etc/sysconfig/network

vim /etc/sysconfig/network



source /etc/sysconfig/network

### 1.1.3 配置免ssh密码登录

生成秘钥：

ssh-keygen -t rsa

配置免密码登录：

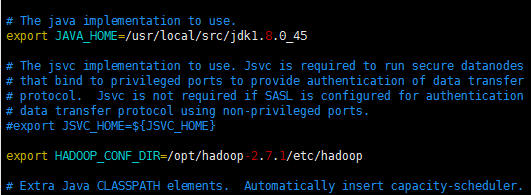
ssh -copy-id root@hadoop

### 1.1.4 配置文件配置

1. **在hadoop中配置hadoop-env.sh文件**

**vim hadoop-env.sh**

**修改JAVA\_HOME路径和HADOOP\_CONF\_DIR路径**



修改后执行：source hadoop-env.sh

1. 修改core-site.xml

vim core-site.xml

<configuration>

<!-- 用来指定hdfs的老大，namenode的地址 -->

<property>

<name>fs.default.name</name>

<value>hdfs://hadoop:9000</value>

</property>

<!-- 用来指定hadoop运行时产生文件的存放目录 -->

<property>

<name>hadoop.tmp.dir</name>

<value>file:/opt/hadoop-2.7.1/tmp</value>

</property>

</configuration>

1. hdfs-site.xml

Vim hdfs-site.xml

<configuration>

<!--指定hdfs保存副本的数量，包括自己，默认值为3，伪分布模式值设为1-->

<property>

<name>dfs.replication</name>

<value>1</value>

</property>

<!--设置hdfs的操作权限，false表示任何用户都可以在hdfs上操作-->

<property>

<name>dfs.permissions</name>

<value>false</value>

</property>

</configuration>

4 修改mapred-site.xml

需要拷贝一份，原来的模板为mapred-site.xml.template

Cp mapred-site.xml.template mapred-site.xml

Vim mapred-site.xml

<configuration>

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value>

</property>

</configuration>

5 修改yarn-site.xml

<configuration>

<!--指定yarn的老大resourcemanager的地址-->

<property>

<name>yarn.resourcemanager.hostname</name>

<value>hadoop</value>

</property>

<!--NodeManager获取数据的方式-->

<property>

<name>yarn.nodenameager.aux-services</name>

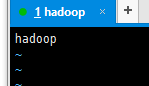
<value>mapreduce\_shuffle</value>

</property>

</configuration>

6 修改 slaves

Vim slaves



### 1.1.5配置环境变量

# vi /etc/profile

编辑内容：

export HADOOP\_HOME=/opt/hadoop-2.7.1

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

export HADOOP\_LOG\_DIR=/opt/hadoop-2.7.1/logs

export YARN\_LOG\_DIR=$HADOOP\_LOG\_DIR

# source /etc/profile  #使配置文件生效

### 1.1.6 启动

格式化namenode，需要哪个机器做namenode就在哪个上面执行

# cd /opt/hadoop-2.7.3/sbin

# hdfs namenode -format

启动hadoop

# cd /opt/hadoop-2.7.3/sbin

# start-all.sh

检查进程：

# jps

# 2 HDFS分布式文件系统

Hdfs是以流式数据访问模式存储超大文件。

* 超大文件
* 流式数据访问。Hdfs的构建思路：一次写入、多次读取的高效访问模式
* 商用硬件
* 低时延的数据访问。Hdfs是为了提高数据吞吐量应用优化的，这将以提高时延作为代价。对于低延迟的访问需求，hbase是更好的选择。
* 大量的小文件。Namenode的将文件系统的元数据存储在内存当中，所以文件系统所能存储的文件总数受限于内存的大小。
* 多用户写入，任意修改文件。Hdfs同时值能有一个writer，这个写操作只会将数据追加到文件的尾部，不支持多个用户同时写入，页不支持在文件的任意系统写入。

## 2.1 hdfs的概念

### 2.1.1 数据块

每个磁盘都会有默认的数据块大小，比如windos的4对其情况下，一个数据块的大小4K，HDFS文件系统的数据块是磁盘数据块大小的整数倍，默认为64MB，与单一磁盘上的文件系统相似，HDFS上的文件也别划分成块大小的多个分块（chunk），作为独立的存储单元。与文件系统不同的是，HDFS中小于一个块大小的文件不会占据整个块空间。

数据块越大可以减少磁盘寻址时间。

分块好处：

1. 一个文件的大小可以大于网络上任意一个磁盘的容量。
2. 使用抽象块而非整个文件作为存储单元，大大简化了存储子系统的优化。块的大小的固定，因此计算单个磁盘能存储多少个块相对容易很多。消除了对元数据的顾虑，块值存储文件的一部分，而其他系统可以单独管理这些元数据。

适合数据备份提高高可用

显示块的信息

Hadoop fsck / -files -blocks -locations

### 2.1.2 namenode和datanode

HDFS集群有两类节点以管理者-工作者模运行，即一个namenode和多个datanode，namenode管理文件系统的命名空间。它维护着文件系统和整棵树内所有的文件和目录，这些 信息会以两个文件形式永久的保存在本地磁盘上：命名空间镜像文件和编辑日志文件。Namenode页记录着每个文件的各个块所在数据节点信息，但它不会永久保存块的位置信息，因为这些信息会在系统启动时有各个数据节点重建。

客户端（client）代表用户通过与namenode和datanode的交互来访问整个文件系统。

如果运行的namenode节点服务器毁坏，文件系统上所有的文件将会丢失，因为我们不知道如何根据datanode块重建文件。因此hadoop提供两种容错机制：

1. 备份那些组成文件系统元数据持久状态的文件。将写操作同步，是原子操作，将持久状态写入本地的同时，写入一个远程挂载的网络文件系统上（NFS）。
2. 运行一个辅助namenode，作用是定期的通过编辑日志合并命名空间镜像，以防编辑日志过大，它会保存合并后的命名空间镜像副本，并在namenode发生故障时启用，但是辅助节点的数据总会滞后主节点，所以当主节点宕机时，难免会损失一部分数据，这时，一般吧存储在namenode上的元数据复制到辅助节点上作为新的namenode。

### 2.1.3联邦HDFS

因为Namenode中存储了文件系统中所有文件和文件数据块的引用关系，所以当文件数量过大时，用的内存也就越大，所以内存成为制约hdfs横向扩展的瓶颈，在2.x中引入了联邦hdfs可以通过添加namenode来实现横向扩展，每个namenode管理文件系统命名空间的一部分。

在联邦环境下，每个namenode维护者一个命名空间卷（namespace volume），包括命名空间的源数据和该命名空间下的所有数据块的数据块池。命名空间卷之间是相互独立的，两两之间不不想通信，甚至一个命名空间卷失效也不会影响其他namenode维护的命名空间卷的高可用。数据块池页不在切分，因此集群中的datanode需要注册到每个namenode上，并且存储着来自多个数据块池的数据块。

要访问联邦hdfs集群，客户端需要使用客户端挂载数据表将文件路径映射到namenode。该功能也可以通过ViewFileSystem和viewfs://URI进行配置和管理。

### 2.1.4 HDFS高可用

Hdfs配置了一对活动-备用（active-standby）namenode。当活动的namenode失效，备用的namenode就会接管它的任务并开始服务于来自客户端的请求，不会有任何明显的中断。修改如下：

1. namenode之间通过高可用的共享存储实现编辑日志的共享。可用架构在zokeeper上的bookkeeper这样的系统实现。当备用namenode接管之后，会将通读编辑日志直至末尾，以实现与namenode的状态同步，并继续读取由活动namenode写入新条目。
2. Datanode需要同时向两个namenode发送数据块处理报告。因为数据块的映射信息存储在内存而非磁盘。
3. 客户端需要使用特定的机制来处理namenode失效问题。

## 2.2 javaAPI

### 2.2.1 文件写入

|  |
| --- |
| *//获取hdfs文件系统* **public static** FileSystem getFs() **throws** URISyntaxException, IOException {  String url=**"hdfs://192.168.8.105:9000"**;  Configuration conf=**new** Configuration();  FileSystem fs=FileSystem.*get*(URI.*create*(url),conf);  **return** fs; } @Test **public void** testOpen() **throws** IOException, URISyntaxException {  FileSystem fs=*getFs*();  FSDataInputStream in=**null**;  **try** {  in = fs.open(**new** Path(**"/park02/3.txt"**));  IOUtils.*copyBytes*(in, System.***out***, 4096, **false**);  *//定位到文件的开头，从新读取，耗费资源，不建议使用* in.seek(0);  IOUtils.*copyBytes*(in,System.***out***,4096,**false**);  }**finally** {  IOUtils.*closeStream*(in);  fs.close();  } } |

### 2.2.2 写入数据

Public boolean mkdirs(path p)创建目录

|  |
| --- |
| **public void** testProgress() **throws** IOException {  String url=**"hdfs://192.168.8.105:9000"**;  Configuration conf=**new** Configuration();  FileSystem fs=FileSystem.*get*(URI.*create*(url),conf);  ***//创建节点可以指定是否覆盖现有文件，文件备份数量，写入文件是所用的缓冲区大小，文件块大小及文件权限  //当path不存在时会自动创建父目录，不需要显示的调用mkdirs  //progressable用于传递回调接口，可以吧数据写入datanode的进度通知给应用  //每次调用progress方法就是每次将64KB数据写入datanode管线后打印一个点显示的过程***OutputStream out=fs.create(**new** Path(**"/park04"**), **new** Progressable() {  **public void** progress() {  System.***out***.println(**"."**);  }  });  *//已有文件末尾追加数据  //OutputStream out2=fs.append(new Path("/park04"));* InputStream in =**new** BufferedInputStream(**new** FileInputStream(**"hello.txt"**));  IOUtils.*copyBytes*(in,out,4096,**true**); } |

### 2.2.3 查询文件系统

***1 .文件元数据： FileStatus***

封装了文件系统中文件和目录的元数据，包括文件长度、块大小、副本、修改时间、所有者以及权限信息。

FileSystem的getFileStatus（）方法用于获取文件或者目录的FileStatus对象。

|  |
| --- |
| **public** **void** testListStatus() **throws** IOException, URISyntaxException{  Configuration conf=**new** Configuration();  FileSystem fs=FileSystem.*get*(**new** URI("hdfs://192.168.8.105:9000"),conf);  FileStatus[] files=fs.listStatus(**new** Path("/park02"));  **for**(FileStatus f:files){  System.***out***.println(f);  }  fs.close();  } |

***2. 文件的块存储信息***

|  |
| --- |
| **public** **void** testBlockLocation() **throws** IOException, URISyntaxException{  Configuration conf=**new** Configuration();  FileSystem fs=FileSystem.*get*(**new** URI("hdfs://192.168.8.105:9000"),conf);  BlockLocation[] bls=fs.getFileBlockLocations(**new** Path("/park02/3.txt"), 0, 10);  **for**(BlockLocation bl:bls){  System.***out***.println(bl);  }  fs.close();  } |

***3 文件模式***

常见与单个操作处理大批文件的需求。在一个表达式中使用通配来匹配多个文件，无需列举每个文件和目录来指定输入，该操作成为“通配（globbing）”，hadoop为执行通配提供了两个FileSystem方法：

Public FileStatus[] globStatus(Path pathPattern)

Public FileStatus[] globStatus(Path pathPattern,PathFilter filter)

PathFiler示例：

|  |
| --- |
| **public** **class** RegexPathFilter **implements** PathFilter{  **private** **final** String regex;  **public** RegexPathFilter(String regex) {  **this**.regex=regex;  }  @Override  **public** **boolean** accept(Path path) {  **return** !path.toString().matches(regex);  }  } |

过滤器值只传递不匹配与正则表达式的文件。

Fs.globStatus(new Path(“/2007/\*/\*”),new RegexPathFiler(“^.\*/2007/12/31$”));

# 3 mapreduce框架

ResourceManager工作职能

1. 知道管理那些机器，即管理那些NodeManager
2. 要有检测机制，能够奸恶到NodeManager的转台变化，童工RPCC心跳来实现
3. 任务的分配和的调度，ResourceManger能够做到细粒度的任务分配，不如某一个任务需要占用多大的内存，需要多少计算资源

NodeManager工作职能

## 3.1 MR实现单词统计

实现步骤

1. 创建maven工程
2. 上传待处理的文件到HDFS上因为mr职能处理HDFS上的数据文件
3. 利用mr的API处理任务
4. 达成jar包，上传到inux虚拟机上，利用hadoop的命令运行jar包，即运行job任务

### 3.1.1Configuration配置

Configuration.xml编写

|  |
| --- |
| <?xml version=*"1.0"* ?>  <configuration>  <property>  <name>color</name>  <value>yellow</value>  </property>  <property>  <name>size</name>  <value>0</value>  <final>true</final>  </property>  <property>  <name>size-weight</name>  <value>${size},${weight}</value>  </property>  </configuration> |

Configuration conf=**new** Configuration();

conf.addResource("Configuration.xml");

//获的配置属性时设置默认值

conf.get("color","yellow")

### 3.1.2 WordcountMapper.java

|  |
| --- |
| /\*\*  \* MR计算有两个组件，分别是Mapper和reducer  \* Mapper组件  \*/  **public** **class** WordCountMapper **extends** Mapper<LongWritable, Text, Text, IntWritable>{  @Override  **protected** **void** map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, IntWritable>.Context context)  **throws** IOException, InterruptedException {  String line=value.toString();  String[] data=line.split(" ");  **for**(String word:data){  context.write(**new** Text(word),**new** IntWritable(1));  }  }  } |

### 3.1.3 wordCountReducer.java

|  |
| --- |
| /\*\*  \* Reducer组件会就收mapper组件的输出结果《mapper.context.write》  \* 第一个形参类型：mapper输出的key类型  \* 第二个形参类型：mapper输出的value类型  \*  \* 1、Mapper组件可以单独工作，但是Reducer工作必须依赖Mapper组件，因为需要Mapper组件的输出结果  \* 2、引入Reducer组件后，输出结果就是Reducer的输出结果  \*  \* reducer工作机制：  \* 1、根据key，把key对应的value值结合在一起，形成Iterable交给程序员  \* 所以，reduce方法执行的次数取决于Mapper输出的key  \* 2、Reducer会对key进行排序  \*/  public class WordCountReducer extends Reducer<Text, IntWritable, Text, Text> {  @Override  protected void reduce(Text key, Iterable<IntWritable> values, Reducer<Text, IntWritable, Text, Text>.Context context)  throws IOException, InterruptedException {  String result="";  int count=0;  for(IntWritable value:values){  count++;  }  result=count+"";  context.write(key, new Text(result));  // super.reduce(arg0, arg1, arg2);  }  } |

### 3.1.4 wordCountDriver.java

|  |
| --- |
| /\*\*  \* 输出结果key：每行行首偏移量  \* vlaue：每行内容  \* Mapper第一个形参对用的是key，所以LongWritable，这个形参是固定的  \* 第二个形参对应的是value，应为value是一行一行的内容，所以是Text，这个参数是固定的  \*  \* 1、Mapper组件会读取文件内容，并通过map方法的两个形参key，value，交给程序员，我们最主要的是拿到value，即每行内容  \* 2、每有一行数据就会触发一次map方法  \* 3、第三个和第四个形参，要和context输出的key和value类型对应 \*  \*/  public class WordCountDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  Configuration conf=new Configuration();  Job job=Job.getInstance(conf);    //指定Job工作运行的主类  job.setJarByClass(WordCountDriver.class);  //指定Mapper组件的运行类  job.setMapperClass(WordCountMapper.class);  //指定reducer的运行类  job.setReducerClass(WordCountReducer.class);    //指定mapper组件输出key的类型  job.setMapOutputKeyClass(Text.class);  //指定maooer组件输出的value类型，设置Text  job.setMapOutputValueClass(IntWritable.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(Text.class);    //设置job处理文件所在的HDFS  FileInputFormat.setInputPaths(job, new Path("hdfs://192.168.8.105:9000/wordcount/words.txt"));  //mr处理生成的结果是以文件的形式存储的，这个结果文件必须放在HDFS上，这个输出路径需要指定，会自行创建  FileOutputFormat.setOutputPath(job, new Path("hdfs://192.168.8.105:9000/wordcount/result2"));  //  job.waitForCompletion(true);  }  } |

## 3.2 mr面向对象和分区

以流量统计为例

### 3.2.1 对象编写和序列化

|  |
| --- |
| **public** **class** FlowBean **implements** Writable{  **private** String phone;  **private** String addr;  **private** String name;  **private** **int** flow;  //反序列化，必须和反序列化的顺序一致  @Override  **public** **void** readFields(DataInput in) **throws** IOException {  **this**.phone=in.readUTF();  **this**.addr=in.readUTF();  **this**.name=in.readUTF();  **this**.flow=in.readInt();  }  //序列化  @Override  **public** **void** write(DataOutput out) **throws** IOException {  out.writeUTF(phone);  out.writeUTF(addr);  out.writeUTF(name);  out.writeInt(flow);  }  **public** String getPhone() {  **return** phone;  }  **public** **void** setPhone(String phone) {  **this**.phone = phone;  }  。。。。。。Getter和setter  @Override  **public** String toString() {  **return** "FlowBean [phone=" + phone + ", addr=" + addr + ", name=" + name + ", flow=" + flow + "]";  }  } |

### 3.2.2 mapper

|  |
| --- |
| **public** **class** FlowMapper **extends** Mapper<LongWritable, Text, Text, FlowBean>{  @Override  **protected** **void** map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, FlowBean>.Context context)  **throws** IOException, InterruptedException {  String line=value.toString();  FlowBean flowBean=**new** FlowBean();  flowBean.setPhone(line.split(" ")[0]);  flowBean.setAddr(line.split(" ")[1]);  flowBean.setName(line.split(" ")[2]);  flowBean.setFlow(Integer.*parseInt*(line.split(" ")[3]));  context.write(**new** Text(line.split(" ")[2]), flowBean);  }  } |

### 3.2.3 reducer

|  |
| --- |
| **public** **class** FlowReducer **extends** Reducer<Text, FlowBean, Text, FlowBean>{  @Override  **protected** **void** reduce(Text key, Iterable<FlowBean> values, Reducer<Text, FlowBean, Text, FlowBean>.Context context)  **throws** IOException, InterruptedException {  FlowBean bean=**new** FlowBean();  **for**(FlowBean flow:values){  **if**(bean.getName()==**null**){  bean.setPhone(flow.getPhone());  bean.setName(flow.getName());  bean.setAddr(flow.getAddr());  }  bean.setFlow(bean.getFlow()+flow.getFlow());  }  context.write(key, bean);  }  } |

### 3.2.3 Partitioner设置分区

在driver中加入下面两句

//设置reducer的数量

job.setNumReduceTasks(3);

//设置自定义的分区类

job.setPartitionerClass(FlowPartioner.**class**);

|  |
| --- |
| /\*\*  \* 自己设置mapper输出分区  \*  \* 第一个泛型为，map输出的key类型  \* 第二个泛型为，map输出的value类型  \* //设置自定义的分区类  //job.setPartitionerClass(FlowPartioner.class);  \*/  **import** org.apache.hadoop.io.Text;  **import** org.apache.hadoop.mapreduce.Partitioner;  **public** **class** FlowPartioner **extends** Partitioner<Text, FlowBean>{  //第三个参数为分区数量，从0开始  @Override  **public** **int** getPartition(Text key, FlowBean value, **int** arg2) {  String addr=value.getAddr();  **if**(addr.equals("bj")){  **return** 0;  }  **if**(addr.equals("sz")){  **return** 1;  }**else**{  **return** 2;  }  }  } |

## 3.3 combiner

Mapper--->Combiner--->reducer

比如在比较最高气温中：

Max（0,29,2,34,43,32）=max(max(0,29,2),max(43,32))=max(29,43)=43

Combiner把mapper产生的数据先进行一次处理，然后在交给reducer。这样能够有效的减少mapper和reducer之间的数据传输量，但是combiner并不能完全取代reducer，因为仍然需要reducer函数处理不同map输出中具有相同键的记录。

实现与reducer相同，

Job.setCombinerClass(combiner.class);

## 3.4 多文件输入的mapper

|  |
| --- |
| **public** **class** ScoreMapper **extends** Mapper<LongWritable,Text,Text,Score>{  @Override  **protected** **void** map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, Score>.Context context)  **throws** IOException, InterruptedException {  FileSplit split=(FileSplit) context.getInputSplit();  //得到文件名的方法  Score score=**new** Score();  String filename=split.getPath().getName();  String line=value.toString();  **if**(filename.equals("chinese.txt")){  score.setChinese(Integer.*parseInt*(line.split(" ")[2]));  }  **if**(filename.equals("english.txt")){  score.setEnglish(Integer.*parseInt*(line.split(" ")[2]));  }  **if**(filename.equals("math.txt")){  score.setMath(Integer.*parseInt*(line.split(" ")[2]));  }  score.setName(line.split(" ")[1]);  context.write(**new** Text(line.split(" ")[1]), score);  }  } |

# 4 hadoop的I/O操作

## 4.1数据完整性

Hdfs会对写入的数据计算数据完整性，并在读取数据的时候计算校验和。他针对每一个由io.bytes.per.checksum指定字节的数据计算校验和。默认情况下为512字节，而CRC魂环冗余校验码的大小为4字节，所以存储校验和的额外开销小于1%。

Datanode负责收到数据后存储该数据并且计算校验和。

写入数据时，客户端将数据及其校验和发送到由一些列datanode构成的管线，管线的最后一个datanode负责验证校验和，如果错误，便抛给客户端ChecksumException异常。

读取数据时与每个datanode中存储的校验和比较。

如果发现有损坏的数据块，datanode会安排这个数据块的一个副本复制到另一个datanode，然后删除已损坏的数据块。

Configuration conf=new Configuration();

FileSystem fs=new RawLocalFileSystem();

Fs.initialize(null,conf);

获得文件的校验和文件路径：

FileSystem checksummedfs=new ChecksumFileSystem(conf);

checksummedfs.getChecksumFile();

## 4.2 数据的压缩

文件压缩有两大好处：减少文件存储的磁盘空间，加速数据在网络和磁盘上的传输。

1、 *//reduce压缩输出的api* FileOutputFormat.*setCompressOutput*(job,**true**);  
 FileOutputFormat.*setOutputCompressorClass*(job, GzipCodec.**class**);

Configuration conf=**new** Configuration();  
2、 *//设置map任务输出gzip压缩格式的代码* conf.setBoolean(**"mapred.compress.map.ouput"**,**true**);  
 conf.setClass(**"mapred.map.output.compress.codec"**,GzipCodec.**class**,CompressionCodec.**class**);  
  
 Job job=Job.*getInstance*(conf);

Codec实现了一种压缩-解压算法。在hadoop中一个对CompressionCodec 接口的实现代表一个codec。

1 压缩从标准文件流读取的输成绩然后写入标准输出流

|  |
| --- |
| **public** **class** CedecTest {  **public** **static** **void** main(String[] args) **throws** Exception {  String codecClassName="";  Class<?> codecClass=Class.*forName*(codecClassName);  Configuration conf=**new** Configuration();  CompressionCodec codec=(CompressionCodec) ReflectionUtils.*newInstance*(codecClass, conf);  CompressionOutputStream out=codec.createOutputStream(System.***out***);  IOUtils.*copyBytes*(System.***in***, out, 4096,**false**);  out.finish();  }  } |

## 4.3 序列化

RPC序列化的格式如下：

紧凑：存储格式紧凑，进而高效的使用存储空间。

快速：读/写数据的额外开销比较小

可扩展：可以透明的读取老格式的数据

互操作：可以使用不同语言读/写永久存储的数据

Text

Text.toString();

IntWritable

NullWritable

NullWritable.get();

ObjectWritble

Writable的集合类：

ArrayWritbale.MapWritable SortedMapWritable ArrayPrimitiveWritable

ArrayWritable writbale=new ArrayWritable(Text.class);

ArrayWritable和TwoDArrayWriable都有get（）、set()、toArray()方法，toArray（）用于新建该数组的一个浅拷贝。

ArrayPrimitiveWritable是对java数组类型的一个封装。调用set（）方法时，可以识别相应组件类型。

## 4.4 基于文件的数据结构

对于某些应用，我们需要一种特殊的数据结构来存储自己的数据。对于基于MapReduce的数据处理，将每个二进制数据大对象（blob）单独存放在各自的文件中不能实现可扩展性，所以Hadoop为此开发了跟多更高层次的容器。

### 4.4.1 SequenceFile

考虑日志文件，其中每一行代表一条日志记录。纯文本不合适记录二进制类型的数据。在这种情况下，hadoop的SequenceFile非常合适。将它作为日志文件的存储格式时，你可以自己选择键（比如LongWritable类表示时间戳），以及值可以是Writbale类型（用于表示日志记录的数量）

HDFS和mapreduce是针对大文件优化的，所以通过SequenceFile类型将小文件包装起来，可以获得更高效率的存储和处理。

1. 写操作

SequenceFIle.Writer writer=SequenceFile.*createWriter*(fs,conf,path,key.getClass(),value.getClass());

|  |
| --- |
| **public class** SequenceFileWrite {  **private static final** String[] ***DATA***={  **"One,two,buckle my shoe"**,  **"Three,four,shut the door"**,  **"Five,six,pick up sticks"**,  **"Seven,eight,lay them straight"**,  **"Nine,ten,a big fat hen"** };  **public static void** main(String[] args) **throws** IOException {  String uri=**"sequence.text"**;  Configuration conf=**new** Configuration();  FileSystem fs=FileSystem.*get*(URI.*create*(uri), conf);  Path path=**new** Path(uri);  IntWritable key=**new** IntWritable();  Text value=**new** Text();  SequenceFile.Writer writer=**null**;  **try**{  writer=SequenceFile.*createWriter*(fs,conf,path,key.getClass(),value.getClass());  **for**(**int** i=0;i<100;i++){  key.set(100-i);  value.set(***DATA***[i%***DATA***.**length**]);  System.***out***.printf(**"[%s]\t%s\t%s\n"**,writer.getLength(),key,value);  writer.append(key,value);  }  }**finally** {  IOUtils.*closeStream*(writer);  }  } } |

2 写操作

读取那条记录与使用的序列化框架有关系

如果使用writable则： public boolean next(Writable key,Writable value);

读取成功返回true，读到文件末尾或者取失败返回false；

对于非Writable框架（比如Thrift）：

Public Object next（Object key）；

Public Object getCurrentValue（Object val）；

如果返回部位null，则用getCurentVlaue获取值，如果为空，表示读到文件末尾

|  |
| --- |
| **public class** SequenceFileReader {  **public static void** main(String[] args) **throws** IOException {  String uri=**"sequence.text"**;  Configuration conf=**new** Configuration();  FileSystem fs=FileSystem.*get*(URI.*create*(uri),conf);  Path path=**new** Path(uri);  SequenceFile.Reader reader=**null**;  **try**{  reader=**new** SequenceFile.Reader(fs,path,conf);  Writable key= (Writable) ReflectionUtils.*newInstance*(reader.getKeyClass(),conf);  Writable value= (Writable) ReflectionUtils.*newInstance*(reader.getValueClass(),conf);  **long** position=reader.getPosition();  **while**(reader.next(key,value)){  *//替换特殊字符串，同步，出错后再次与记录边界同步* String syncSeen=reader.syncSeen()?**"\*"**:**""**;  System.***out***.printf(**"[%s%s]\t%s\t%s\n"**,position,syncSeen,key,value);  position=reader.getPosition();  }  }**finally** {  IOUtils.*closeStream*(reader);  }  } } |

顺序搜索文件位置1 ：reader.seek(234) 如果指定不是记录边界，调用next会出错。

2：reader.sync(long) 定位到position后的第一个同步点。

SequenceFile.Writer对象有一个sync（）方法，用于插入一个同步点。

### 4.4.2 MapFile

已经排序过的SequenceFIle，有索引，所以可以按键查找。可以将MapFile视作java。Util。Map的持久化形式，尽管他并没有实现map接口，他的大小可以超过保存在内存中一个map的大小。

1 写出操作

新建一个MapFile.Wribale实例，然后调用append（）顺序写入内容。必须顺序写入，键必须为WritableComparatable类型，值必须是Writable类型。

|  |
| --- |
| **public class** MapFileWriter {  **private static final** String[] ***DATA***={  **"One,two,buckle my shoe"**,  **"Three,four,shut the door"**,  **"Five,six,pick up sticks"**,  **"Seven,eight,lay them straight"**,  **"Nine,ten,a big fat hen"** };  **public static void** main(String[] args) **throws** IOException {  String uri=**"mapsequence.text"**;  Configuration conf=**new** Configuration();  FileSystem fs=FileSystem.*get*(URI.*create*(uri),  conf);  Path path=**new** Path(uri);   IntWritable key=**new** IntWritable();  Text value=**new** Text();  MapFile.Writer writer=**null**;  **try**{  writer=**new** MapFile.Writer(conf,fs,uri,key.getClass(),value.getClass());  **for**(**int** i=0;i<1024 ;i++){  key.set(i+1);  value.set(***DATA***[i%***DATA***.**length**]);  System.***out***.printf(**"%s\t%s\n"**,key,value);  writer.append(key,value);  }  }**finally** {  IOUtils.*closeStream*(writer);  }   } } |

输出是个文件夹，包含两个文件data和index，data是记录文件，index是索引文件，index中存储一部分键和data文件中键到其偏移量的映射。，默认每128个键会有一个索引，也可以用MapFile。Writer 的setIndexInterval（）方法来调整。

2 读取

顺序读取：

Public boolean next（WritableComparable key，Writable val）;

随机访问：

Public Writbale get（WritableComparatbale key，Writable val）；

例Text： Text value=new Text();

Reader.get(new IntWritable(433),value);

不要修改索引间隔后重建索引，可以读取索引时设置io.map.index.skip来加载一定比例的索引键。该属性通常为0，不跳过索引，如果设为3，表示一次跳过index中的两个索引（只读取三分之1的索引）。

3 MapFile的变种

SetFile是一个特殊的MapFile，用于存储Writable键的集合。键必须按照排好的顺序添加。

ArrayFile的键是一个整形，用于表示数组中元素的索引，而值是一个Writable值。

BloomMapFile提供了get（）方法的高性能实现，对于稀疏的文件和你有用。该类实现一个布隆过滤器来检测某个给定的键是否在map文件中。

### 4.4.3 SequenceFile转换为MapFile

1. 将SequenceFIle排序
2. 将排序后的文件改名为data，放入map文件夹下
3. 如果有计算校验和的文件，删除掉
4. 重建索引

|  |
| --- |
| **public class** MapFileFixer {  **public static void** main(String[] args) **throws** Exception {  String uri=**"mapsequence.text"**;  Configuration conf=**new** Configuration();  FileSystem fs=FileSystem.*get*(URI.*create*(uri), conf);  Path map=**new** Path(uri);  Path mapData=**new** Path(map, MapFile.***DATA\_FILE\_NAME***);   SequenceFile.Reader reader=**new** SequenceFile.Reader(fs,mapData,conf);  Class keyClass=reader.getKeyClass();  Class valueVlass=reader.getValueClass();  reader.close();  *//重建索引* **long** entries=MapFile.*fix*(fs,map,keyClass,valueVlass,**false**,conf);  System.***out***.printf(**"Created Mapfie %s with %d entries\n"**,map,entries);  } } |

## 4.5 输入格式

### FileInputFormat

提供addInputPath（job,Path） 和addInputPaths（job，String...）两个方法将一个或多个路径加入路径列表。

SetInputPath（job，Path...）设定完整的路径列表，替换前面调用中在job上所设所有的路径。

指定路径为目录，其内容不会被递归。将mapred.input.dir.recursive设置为true从而强制对输入目录进行递归的读取。

如果需要排除特定文件可以使用FileInputFormat的过滤器方法：(默认排除“.”,”\_”开头文件)

Public static void setInputPathFileter(Job job,Class<? Extends PathFiter> filter)；

1 KeyValueTextInputFormat

默认的TextInputFormat是以行偏移量为键，行为值。

KeyValueTextInputFormat可以设置行分隔符前的字符为值，分隔符后的值为值，分割复可以用mapreduce.input.keyvaluelinerecordreader.key.value.separator属性来指定分隔符，默认为一个制表符。

2 NLineInputFormat

如果希望mapper收到固定行的输入，可以用NlineInputFormat

N的默认值为1，可以用mapreduce.input.lineinputformat.linespermap属性指定。

3 二进制输入SequenceFileInputFormat

如果输入文件为SequenceFIle，则可以从文件本身获取键和值，可以同时处理MapFile和SequenceFile。

4 SequenceFileAsTextInputFormat

将顺序文件的键和值转换为Text对象输入。这个转换通过调用toString完成。

5 SequenceFileAsBinaryinputFormat

将顺序文件的键和值转换为BytesWritable对象。

### 很多小文件CombineFileInputFormat

使map操作中处理数据量与HDFS中文件 的块大小之间的耦合度降低了。

相对于小批量的文件，FileInputFormat为每个文件产生一个分片，一个分片一个map任务，就会有很多map任务，每次map操作都会产生额外的开销。小文件将增加运行作业而必须的寻址次数。还有hdfs的集群中文件的元数据读存储在内存中，大量小文件会占用namenode的内存。

把CombineFileInputFormat的最大分片大小设置成块数较小的整数倍，使每个map处理多个块，这样整个处理的时间减少了，因为少量的mapper运行减少了大量短时mapper所涉及的任务管理和开销。（通过mapred.max.split.size设置，以字节为单位）

使用CombineFileInputFormat需要创建一个实现类，并且实现getRecordReader()方法；

FileInputFormat类：

|  |
| --- |
| **public class** CombineSmallFileInputFormat **extends** CombineFileInputFormat<LongWritable,BytesWritable> {  @Override  **public** RecordReader<LongWritable, BytesWritable> createRecordReader(InputSplit split, TaskAttemptContext context) **throws** IOException {  CombineFileSplit combineFileSplit= (CombineFileSplit) split;  *//一定要通过CombineFileRecordReader来创建一个RecordReader  //构造方法必须是上面的类型和顺序，构造方法包含三个参数* CombineFileRecordReader<LongWritable,BytesWritable> recordReader=  **new** CombineFileRecordReader<>(combineFileSplit,context,CombineSmallFileRecordReader.**class**);  **try** {  recordReader.initialize(combineFileSplit,context);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return** recordReader;  } } |

CombineSmallFileRecordReader

|  |
| --- |
| **public class** CombineSmallFileRecordReader **extends** RecordReader<LongWritable,BytesWritable> {   **private** CombineFileSplit **combineFileSplit**;  **private** LineRecordReader **lineRecordReader**=**new** LineRecordReader();  **private** Path[] **paths**;  **private int totalLength**;  **private int currentIndex**;  **private float currentProgress**=0;  **private** LongWritable **currentKey**;  **private** BytesWritable **currentValue**=**new** BytesWritable();   **public** CombineSmallFileRecordReader(CombineFileSplit combineFileSplit,TaskAttemptContext context,Integer index){  **super**();  **this**.**combineFileSplit**=combineFileSplit;  **this**.**currentIndex**=index;*//当前要处理的小文件Block在CombineFileSplit中的索引* }  @Override  **public void** initialize(InputSplit split, TaskAttemptContext context) **throws** IOException, InterruptedException {  **this**.**combineFileSplit**= (CombineFileSplit) split;  *//处理CombineFileSplit中的小文件Block，因为使用LineRecordReader，需要构造一个Fileplit对象，然后才能读取数据* FileSplit fileSplit=**new** FileSplit(**combineFileSplit**.getPath(**currentIndex**),  **combineFileSplit**.getOffset(**currentIndex**),**combineFileSplit**.getLength(**currentIndex**),  **combineFileSplit**.getLocations());  **lineRecordReader**.initialize(fileSplit,context);  **this**.**paths**=**combineFileSplit**.getPaths();  **totalLength**=**paths**.**length**; context.getConfiguration().set(**"map.input.file.name"**,**combineFileSplit**.getPath(**currentIndex**).getName());  }  @Override  **public boolean** nextKeyValue() **throws** IOException, InterruptedException {  **if**(**currentIndex**>=0&&**currentIndex**<**totalLength**){  **return true**;  }**else**{  **return false**;  }  }  @Override  **public** LongWritable getCurrentKey() **throws** IOException, InterruptedException {  **currentKey**=**lineRecordReader**.getCurrentKey();  **return currentKey**;  }  @Override  **public** BytesWritable getCurrentValue() **throws** IOException, InterruptedException {  **byte**[] content=**lineRecordReader**.getCurrentValue().getBytes();  **currentValue**.set(content,0,content.**length**);  **return currentValue**;  }  @Override  **public float** getProgress() **throws** IOException, InterruptedException {  **if**(**currentIndex**>0&&**currentIndex**<**totalLength**){  **currentProgress**=(**float**) **currentIndex**/**totalLength**;  **return currentProgress**;  }  **return currentProgress**;  }  @Override  **public void** close() **throws** IOException {  **lineRecordReader**.close();  } } |

Mapper中：

|  |
| --- |
| **public class** CombineSmallFileMapper **extends** Mapper<LongWritable,ByteWritable,Text,Text>{  **private** Text **file**=**new** Text();  @Override  **protected void** map(LongWritable key, ByteWritable value, Context context) **throws** IOException, InterruptedException {  String filename=context.getConfiguration().get(**"map.input.file.name"**);  **file**.set(filename);   context.write(**file**,**new** Text(**new** String(value.toString().trim())));  } } |

### 避免切分

有些应用程序不希望文件被切分，而用一个mapper完整的处理整个文件，例如检查文件是否有序。

第一种方法：增加分片大小，将大小设置成大于要处理的文件；

第二种方法：使用FileInputFormat的具体子类，并且重载isSplitable()方法，把返回值设置为false。

### 整个文件当做一条记录

即使不分割文件，仍需要一个RecordReader来读取文件内容作为Record值：

Ps： 一个map处理一个小文件是低效的所以最好的办法是继承CombineFileInputFormat

|  |
| --- |
| **public** **class** WholeFileInputFormat **extends** FileInputFormat<NullWritable, ByteWritable>{  **protected** **boolean** isSplitable(JobContext context,Path faile){  **return** **false**;  }  @Override  **public** RecordReader<NullWritable, ByteWritable> createRecordReader(InputSplit split, TaskAttemptContext context)  **throws** IOException, InterruptedException {  RecordReader reader=**new** WholeFileRecordReader();  reader.initialize(split , context);  **return** reader;  }  } |

WholeFileRecordReader 负责将fileSplit转换成一条记录，该记录的键值为空

Processed表示记录是否被处理过

|  |
| --- |
| **public** **class** WholeFileRecordReader **extends** RecordReader{  **private** FileSplit fileSplit;  **private** Configuration conf;  **private** BytesWritable value=**new** BytesWritable();  **private** **boolean** processed=**false**;  @Override  **public** **void** initialize(InputSplit split, TaskAttemptContext context) **throws** IOException, InterruptedException {  **this**.fileSplit=(FileSplit) split;  **this**.conf=context.getConfiguration();  }  @Override  **public** **boolean** nextKeyValue() **throws** IOException, InterruptedException {  **if**(!processed){  **byte**[] contents=**new** **byte**[(**int**) fileSplit.getLength()];  Path file=fileSplit.getPath();  FileSystem fs=file.getFileSystem(conf);  FSDataInputStream in=**null**;  **try**{  in=fs.open(file);  IOUtils.*readFully*(in, contents,0,contents.length);  value.set(contents,0,contents.length);  }**finally**{  IOUtils.*closeStream*(in);  }  processed=**true**;  **return** **true**;  }  **return** **false**;  }  @Override  **public** **void** close() **throws** IOException {  }  @Override  **public** Object getCurrentKey() **throws** IOException, InterruptedException {  **return** NullWritable.*get*();  }  @Override  **public** Object getCurrentValue() **throws** IOException, InterruptedException {  **return** value;  }  @Override  **public** **float** getProgress() **throws** IOException, InterruptedException {  **return** processed?1.0f:0.0f;  }  } |

将若干和小文件打包成顺序文件的程序：

Mapper中：

Class SequenceFileOutputformat extends Mapper<>{

FileSplit split=(FileSplit)context.getInputSplit();

Path path=split.getPath();

Text key=new Text(path.toString);

Context.write(key,value);}

Driver中：

Job.setInputFormatClass(WholeFileInputFormat.class);

job.setOutputFormatClass(SequenceFileOutputformat.class);

### 把行号作为键输入格式

InputFormat

|  |
| --- |
| **public class** MyInputFormat **extends** FileInputFormat<IntWritable,Text> {  @Override  **public** RecordReader<IntWritable, Text> createRecordReader(InputSplit split, TaskAttemptContext context) **throws** IOException, InterruptedException {  **return new** MyFileRecordReader();  } } |

MyFileRecordReader

|  |
| --- |
| **public class** MyFileRecordReader **extends** RecordReader {  **private** FileSplit **fileSplit**;  **private** Configuration **conf**;  **private** IntWritable **key**;  **private** Text **value**;  **private** LineReader **reader**;  **private int count**;  @Override  **public void** initialize(InputSplit split, TaskAttemptContext context) **throws** IOException, InterruptedException {  **this**.**fileSplit**= (FileSplit) split;  **this**.**conf**=context.getConfiguration();  Path path=**fileSplit**.getPath();  FileSystem fs=path.getFileSystem(**conf**);  *//拿到输入流* FSDataInputStream in=fs.open(path);  **reader**=**new** LineReader(in);  }   *//当次方法的返回值为true时，就会被调用  //每当nextKeyValue（）调用已改getCurrentKey与getCurrentValue就会被调用一次  //每次调用一次就会把getCurrentKey就会把返回值交给map的输入key* @Override  **public boolean** nextKeyValue() **throws** IOException, InterruptedException {  **key**=**new** IntWritable();  **value**=**new** Text();  Text tmp=**new** Text();  **int** result=**reader**.readLine(tmp);  **if**(result==0){  *//证明没有数据可读* **return false**;  }**else**{  **count**++;  **key**.set(**count**);  **value**=tmp;  **return true**;  }  }   @Override  **public** Object getCurrentKey() **throws** IOException, InterruptedException {  **return key**;  }  @Override  **public** Object getCurrentValue() **throws** IOException, InterruptedException {  **return value**;  }  @Override  **public float** getProgress() **throws** IOException, InterruptedException {  **return** 0;  }  @Override  **public void** close() **throws** IOException {  **if**(**reader**!=**null**){  **reader**=**null**;  }  } } |

### Mapper中的文件信息

通过调用mapper上的context对象上的getInputSplit（）方法可以获得输入分片相关信息。

当输入格式源自FileInputFormat()时，该方法返回的InputSplit对象可以强制转换为FileSplit。

FileSplit split=(FileSplit)contxt.getInputSplit();

//获得文件名

String filename=Split.getPath().getName();

Long start=split.getStart();//分片的开始位置的偏移量

Long length=split.getLength();//分片的字节长度

### 多个格式文本输入MultipleInputs

MultipleInputs允许为每条输入路径指定InputFormat和Mapper

MultipleInputs.addInputPath(job,path1,TextInputFormat.class,Mpper1.class)

MultipleInputs.addInputPath(job,path2,TextInputFormat.class,Mpper2.class)

重载的方法可以不指定mapper

## 4.6 输出格式

### OutputFormat实现：

TextOutputFormat，输出键值可以指定，但默认以键值以制表符分割，可以使用mapreduce.output.textoutputformat.separator属性指定。

SequenceFileOutputFormat输出顺序文件，这种输出有益于后续map的输入。

SequenceFileAsBinaryOutputFormat将键值以二进制的格式写入SequenceFile中。

MapFileOutputFormat把MapFile作为输出，必须保证reducer的输出已经排序。

### 输出指定分隔符或格式

|  |
| --- |
| **public class** MyOutputFormat<K,V> **extends** FileOutputFormat<K,V> {  @Override  **public** RecordWriter<K, V> getRecordWriter(TaskAttemptContext job) **throws** IOException, InterruptedException {  Path path=**super**.getDefaultWorkFile(job,**""**);  Configuration conf=**new** Configuration();  FileSystem fs=path.getFileSystem(conf);  FSDataOutputStream out=fs.create(path);  **return new** MyRecordWriter<K,V>(out,**"|"**,**"\r\n"**);  } } |

RecordWriter：

|  |
| --- |
| **public class** MyRecordWriter<K,V> **extends** RecordWriter<K,V> {  **private** FSDataOutputStream **out**;  **private** String **keyValueSeparater**;  **private** String **lineSeparater**;  **public** MyRecordWriter(FSDataOutputStream out, String s, String s1) {  **this**.**out**=out;  **this**.**keyValueSeparater**=s;  **this**.**lineSeparater**=s1;  }  *//key 就是输出key。如果job任务里没有reduce，key就是mapper输出key  //如果有reduce，key时reduce的输出key  //输出value  //注意：写的时候注意顺序：key 分割符 value 行分割符  //这个是决定结果里的格式的，针对mapper输出的结果文件同样生效* @Override  **public void** write(K key, V value) **throws** IOException, InterruptedException {  **out**.write(key.toString().getBytes());  **out**.write(**keyValueSeparater**.getBytes());  **out**.write(value.toString().getBytes());  **out**.write(**lineSeparater**.getBytes());  }  @Override  **public void** close(TaskAttemptContext context) **throws** IOException, InterruptedException {  **out**.close();  } } |

### 多个输出

一个reducer输出多个文件用MultipleOutputFormat。

如我们想要一个气象站的数据输出到一个文件，那么有两种方法：

1. 对map结果分区，这种方法缺点，有的气象站数据大，有的气象站数据小，无法均匀的的利用reducer，利用气象站id分区的时候要先用其他途径获取全部的id。
2. 用MultipleOutputFormat类将数据写入多个文件，文件名可以是输出的键和值中的任意字符串。采用name-m-nnnnn形式的文件名用于map输出，采用name-r-nnnnn形式文件名reducer输出。

在reducer中添加如下两句：

MultipleOutputs<Text,Text> outputs=**new** MultipleOutputs<>(context);

outputs.write(key, **new** Text(result),”filename”);

Write的第三个参数可以包含路径分隔符（“/”），创建任意深度的子目录分割符是有可能的 例：

String basePath=String.format(“%s%s/part”,parser.getStationId,parser.getYear());

MultipleOutput.write(NullWritable.get(),value,basePath);

Write方法重载的实现：

在reducer中：

|  |
| --- |
| **public class** MyMultipleReducer **extends** Reducer<IntWritable,Text,IntWritable,Text> {  **private** MultipleOutputs **outs**;  @Override  **protected void** setup(Context context) **throws** IOException, InterruptedException {  **outs**=**new** MultipleOutputs(context);  }  @Override  **protected void** reduce(IntWritable key, Iterable<Text> values, Context context) **throws** IOException, InterruptedException {  **for**(Text value:values){  String val=value.toString();  **if**((key.toString().equals(**"1"**))){  **outs**.write(**"world"**,key,value);  }**else if**((key.toString().equals(**"2"**))){  **outs**.write(**"hadoop"**,key,value);  }**else**{  **outs**.write(**"1606"**,key,value);  }}}} |

在driver中：

|  |
| --- |
| MultipleOutputs.*addNamedOutput*(job,**"world"**, TextOutputFormat.**class**,IntWritable.**class**,Text.**class**); MultipleOutputs.*addNamedOutput*(job,**"hadoop"**, TextOutputFormat.**class**,IntWritable.**class**,Text.**class**); MultipleOutputs.*addNamedOutput*(job,**"1606"**, TextOutputFormat.**class**,IntWritable.**class**,Text.**class**); |

### 延迟输出

LazyOutputFormat封装一个输出格式，只有指定分区的第一条记录输出不为空时才正真创建文件，使用它，用JobConf和相关的输出格式作为参数来调用setOutputFormatClass（）。

# 5 MapReduce特性

## 5.1 计数器

在很多情况下，根据计数器的值统计特定事件发生的次数要比分析一大堆日志文件容易的多。

### 5.1.1 内置计数器

Hadoop为每个作业维护若干内置计数器。

内置计数器分组：

1 、MapReduce任务计数器

2 、文件计数器

3 、FileInputFormat计数器

4 、FileOutputFormat计数器

5 、作业计数器

### 5.1.2 用户定义java计数器

计数器的值可以在mapper或者reducer中添加，计数器由一个java枚举（enum）类型定义。

context.getCounter(Temperature.***AMLFORMED***);

集群情况下获取计数器值：

Cluster cluster=**new** Cluster(conf);  
Job job1=cluster.getJob(JobID.*forName*(**"jobId"**));  
Counters counters=job.getCounters();  
**long** missing=counters.findCounter(Temperature.***MiSSING***).getValue();

枚举类型:org.apache.hadoop.mapreduce.TaskCounter

## 5.2 排序

### 部分排序

基于mapIFle的查找技术，如果输出格式为MapFileOutputFormat，则输出30个map文件，可以基于这些文件进行查找操作。

job.setOutputFormatClass(MapFileOutputFormat.**class**);

从MapFiles集合中获取符合指定键的第一项记录。

MapFile.Reader[] readers= MapFileOutputFormat.*getReaders*(**new** Path(**""**),conf);  
Partitioner<IntWritable,Text> partitioner=**new** HashPartitioner<>();  
Text value=**new** Text();  
*//使用Partitioner找到包含指定键Reader实例,在通过reader的get（）方法得到键对应的值*Writable entry=MapFileOutputFormat.*getEntry*(readers,partitioner,key,value);

### 全排序

通过对键空间的采样，获得键的近似分布，并构建分区

采样器InputSampler的getSampler(job,InputFormat)返回一系列样本键

顺序文件由TotalOrderPartitioner来创建分区。

|  |
| --- |
| job.setPartitionerClass(TotalOrderPartitioner.**class**);  //采样率0。1 最大样本数1000， 最大分区10，只要满足任意限制就停止采样。 InputSampler.Sampler<IntWritable,Text> sampler=**new** InputSampler.RandomSampler<IntWritable, Text>(0.1,1000,10); InputSampler.*writePartitionFile*(job,sampler); Configuration conf1=**new** Configuration();  //为了和集群上运行的其他任务共享分区文件，InputSampler需将其所写的分区文件加入到分布式缓存中。 String partitionFile=TotalOrderPartitioner.*getPartitionFile*(conf1); URI partitionUri=**new** URI(partitionFile+**"#"**+TotalOrderPartitioner.***DEFAULT\_PATH***); DistributedCache.*addCacheFile*(partitionUri,conf); DistributedCache.*createSymlink*(conf); |