

MapReduce介绍

概述

MapReduce是一个**分布式的计算框架**（编程模型），最初由谷歌的工程师开发，基于GFS的分布式计算框架。后来Cutting根据《Google Mapreduce》,设计了基于HDFS的Mapreduce分布式计算框架。

MR框架对于程序员的最大意义在于，不需要掌握分布式计算编程，不需要考虑分布式编程里可能存在的种种难题，比如任务调度和分配、文件逻辑切块、位置追溯、工作。这样，程序员能够把大部分精力放在核心业务层面上，**大大简化了分布式程序的开发和调试周期。**

MapReduce框架的节点组成结构

JobTracker / ResourceManager：任务调度者，管理多个TaskTracker。ResourceManager是hadoop2.0版本之后引入了yarn，有yarn来管理hadoop之后，jobtracker就被替换成了ResourceManager

TaskTracker / NodeManager：任务执行者

相关配置

1.修改 mapred-site.xml

这个文件初始时是没有的，有的是模板文件，mapred-site.xml.template
所以需要拷贝一份，并重命名为mapred-site.xml

执行：cp mapred-site.xml.template mapred-site.xml

 **配置如下：**

```
<configuration>
<property>
<!-- 指定mapreduce运行在yarn上-->
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
</configuration>
```

```
<configuration>
<property>
<!--指定mapreduce运行在yarn上-->
<name>mapreduce.framework.name</name>
```

```
<value>yarn</value>
</property>
</configuration>
```

yarn是资源协调工具，

2.修改yarn-site.xml

 配置如下：

```
<configuration>
<!-- Site specific YARN configuration properties -->
<property>
<!--指定yarn的老大 resoucemanager的地址-->
<name>yarn.resourcemanager.hostname</name>
<value>hadoop01</value>
</property>
<property>
<!--NodeManager获取数据的方式-->
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
```

```
<configuration>
<!-- Site specific YARN configuration properties -->
<property>
<!--指定yarn的老大 resoucemanager的地址-->
<name>yarn.resourcemanager.hostname</name>
<value>hadoop01</value>
</property>
<property>
<!--NodeManager获取数据的方式-->
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
</configuration>
```

补充：

在实际工作中，namenode节点和ResourceManager节点不在同一台服务器上。

此外，在slaves文件中，配置的datanode和nodemanager的服务列表，即dn和nm这两个进程在同一台服务器上。

启动指令：start-yarn.sh

如果要启动HDFS+MapReduce: start-all.sh

MapReduce实现WordCount

MR入门代码示例

先单独运行map任务，得到的结果文件：

```
hello 1
hello 1
hello 1
world 1
1605 1
```

然后加入reduce任务，查看迭代器的内容

```
1605 1 ,
hello 1,1,1,
world 1,
```

Map任务代码：

```
/**
```

- * HDFS负责存储数据，MR计算框架负责计算HDFS上的数据，比如/park/word.txt。即MR想要计算某个文件，这个文件必须先要在HDFS上进行存储。

- * MR计算文件，会对文件进行逻辑切块，每一块是一个InputSplit，每有一个InputSplit，会启动一个Map任务。逻辑切块的大小是128MB。

- * 现在假设word.txt正好是128MB，则会有一个InputSplit》》有一个map任务》》分配给集群里某一个TaskTracker处理。

- * 如果小于128MB，也是一个InputSplit》》有一个map任务》》分配给集群里某一个TaskTracker处理。

- * 如果是129MB，两个InputSplit》》两个map任务》》此时根据集群里TaskTracker数量及状态分配map任务，这个工作MR框架会做的。不需要程序员来考虑。

- * 此外，因为涉及到了逻辑切块，所以涉及到：要处理的文件在HDFS的位置信息（在哪台datanode节点上）、计算start、end位置信息以及位置追溯等问题，统统都交给MR框架来做。不需要考虑。

- *

- * MapReduce计算框架抽象出两个任务阶段，Map任务阶段和Reduce任务阶段。

- *

- * Map任务阶段的目标：从文件里，按行读取数据。一行一行的进行处理。

- *

- * 如何利用MR框架编写Map阶段代码？

- * 1. 写一个类，继承这个类：org.apache.hadoop.mapreduce.Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>

- * 2. 定义Mapper的4个泛型类型。

- * 3. 重写map()方法，在map方法里编写map任务代码

- *

- * map函数()参数说明：

* 之前曾提到：一个InputSpilt会对应一个Map任务，在InputSpilt里，实际就包含着待处理的文件数据。比如现在要处理的word.txt数据如下：

```
* hello world
* hello hadoop
* hello 1605
*
```

* 然后MR框架会按行进行处理，并形成key/value对。默认的情况下，key是一个长整型，表示行的行首的偏移量。value是一个文本类型，代表每一行数据。

```
*
* key：每一行行首的偏移量（用处不大）
* value：每一行数据（主要用的是这个）
* context:我们利用context 输出我们自定义的输出map<E,V>
*
* 最后需要牢记的是：每有一行数据，就会调用一次map()方法。
*
*/
```

```
public class WordCountMapper extends Mapper<LongWritable, Text, Text,
LongWritable>{
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
Text, LongWritable>.Context context)
        throws IOException, InterruptedException {

        String line=value.toString();
        String[] data=line.split(" ");

        for(String word:data) {
            context.write(new Text(word), new LongWritable(1));
        }
    }
}
```

Reduce任务代码：

```
public class WordCountReducer extends Reducer<Text, LongWritable, Text, Text>{

    @Override
    protected void reduce(Text key, Iterable<LongWritable> values,
        Reducer<Text, LongWritable, Text, Text>.Context context) throws
        IOException, InterruptedException {

        String result="";
        for(LongWritable value:values) {
            result=result+value.get()+",";
        }
        context.write(key,new Text(result));
    }
}
```

Driver

```
public class WordCountDriver {

    public static void main(String[] args) throws Exception {
```

```

Configuration conf=new Configuration();
Job job=Job.getInstance(conf);

job.setJarByClass(WordCountDriver.class);
job.setMapperClass(WordCountMapper.class);
job.setReducerClass(WordCountReducer.class);

job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LongWritable.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

FileInputFormat.setInputPaths(job, new
Path("hdfs://192.168.234.21:9000/wordcount"));

FileOutputFormat.setOutputPath(job, new
Path("hdfs://192.168.234.21:9000/wordcount/result"));

job.waitForCompletion(true);
}
}

```

WordCount示例代码

Mapper代码：

```

public class WCMapper extends Mapper<LongWritable, Text, Text, LongWritable> {

    /**
     * 一个Block对应一个Mapper
     * 一个行出发一次Mapper中的map
     * Key:map的输入，是当前处理的行在文件中的偏移量
     * value:map的输入，是当前行的内容
     * context:环境对象，用来输出数据
     */
    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
    Text, LongWritable>.Context context)
        throws IOException, InterruptedException {
        String line = value.toString();
        String [] words = line.split(" ");
        for(String word : words){
            context.write(new Text(word), new LongWritable(1));
        }
    }
}

```

Reducer代码：

```

public class WReducer extends Reducer<Text, LongWritable, Text, LongWritable> {
    /**
     * Reducer进行数据的合并处理
     * key:reduce的输入，是map的输出
     * values:reduce的输入，是map输出中键相同的值的集合
     * context:环境对象，可以输出数据
     */
    @Override
    protected void reduce(Text key, Iterable<LongWritable> values,
        Reducer<Text, LongWritable, Text, LongWritable>.Context context)
        throws IOException, InterruptedException {
        String word = key.toString();
        long count = 0;
        for(LongWritable l : values){
            count += l.get();
        }

        context.write(new Text(word), new LongWritable(count));
    }
}

```

Driver代码：

```

public class WDriver {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf );


        job.setJarByClass(WDriver.class);

        job.setMapperClass(WMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(LongWritable.class);
        FileInputFormat.setInputPaths(job, new Path("/wc/words.txt"));

        job.setReducerClass(WReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);
        FileOutputFormat.setOutputPath(job, new Path("/wc/result01"));

        job.waitForCompletion(true);
    }
}

```

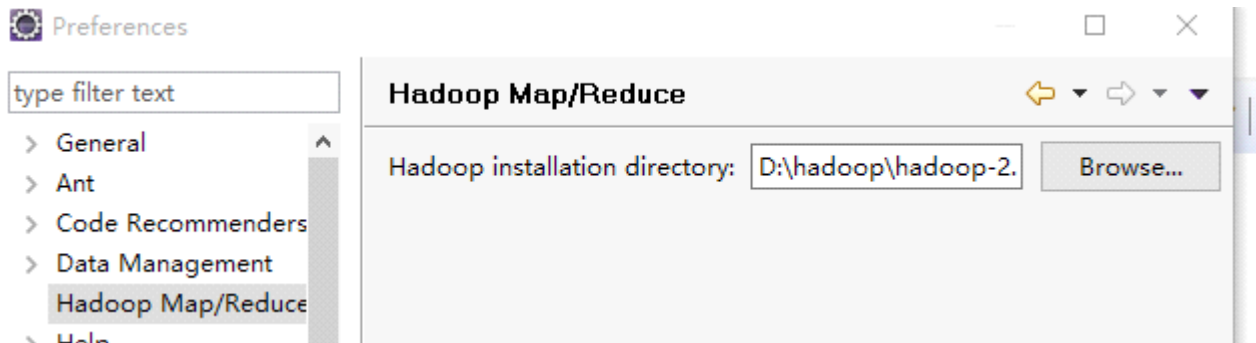
 然后将项目打成jar包，上传到虚拟机上，执行：

hadoop -jar xxx.jar 执行

通过插件实现WordCount

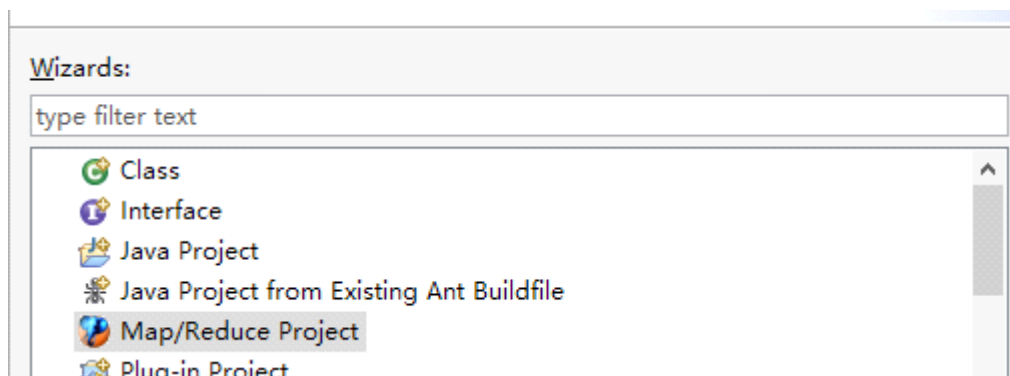
实现步骤：

1.windows=>Hadoop mapreduce => 选择hadoop的安装目录



屏幕剪辑的捕获时间：2016/9/5 20:34

2.new mapreduce project，然后会看到已经有现成的jar包了



3.创建Mapper

4.创建Reducer

5.创建Driver

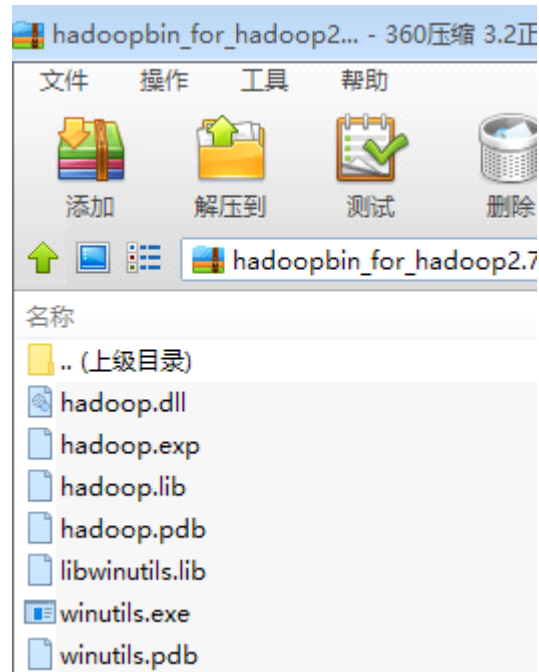
6.在Driver上右键，run as on hadoop

7.如果报null错误:

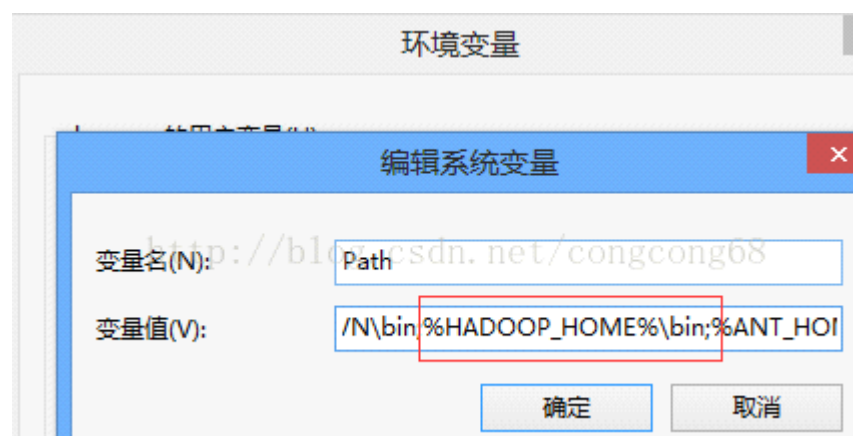
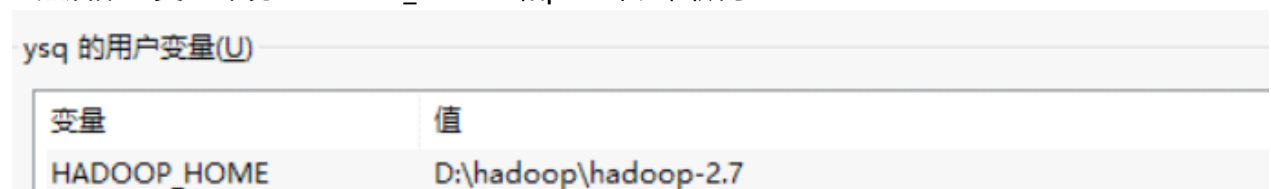

```
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Exception in thread "main" java.lang.NullPointerException
    at java.lang.ProcessBuilder.start(ProcessBuilder.java:1010)
    at org.apache.hadoop.util.Shell.runCommand(Shell.java:483)
    at org.apache.hadoop.util.Shell.run(Shell.java:456)
    at org.apache.hadoop.util.Shell$ShellCommandExecutor.execute(Shell.java:722)
    at org.apache.hadoop.util.Shell.execCommand(Shell.java:815)
    at org.apache.hadoop.util.Shell.execCommand(Shell.java:798)
```

解决办法：

7.将hadoopbin_for_hadoop2里的所有文件，复制到hadoop的安装目录的bin目录下，然后配置下hadoop的bin目录的环境变量。



8.然后配置变量环境HADOOP_HOME和path，如图所示：



10.将插件文件拷贝到C盘 windows的System32目录下

- 11.在hadoop bin目录下，双击执行winutils.exe
- 12.重启IDE，然后测试插件是否可用
- 13.如果不打印日志，引入log4j.properties文件即可

MapReduce入门知识点

2018年8月27日 14:27

Mapper组件知识点

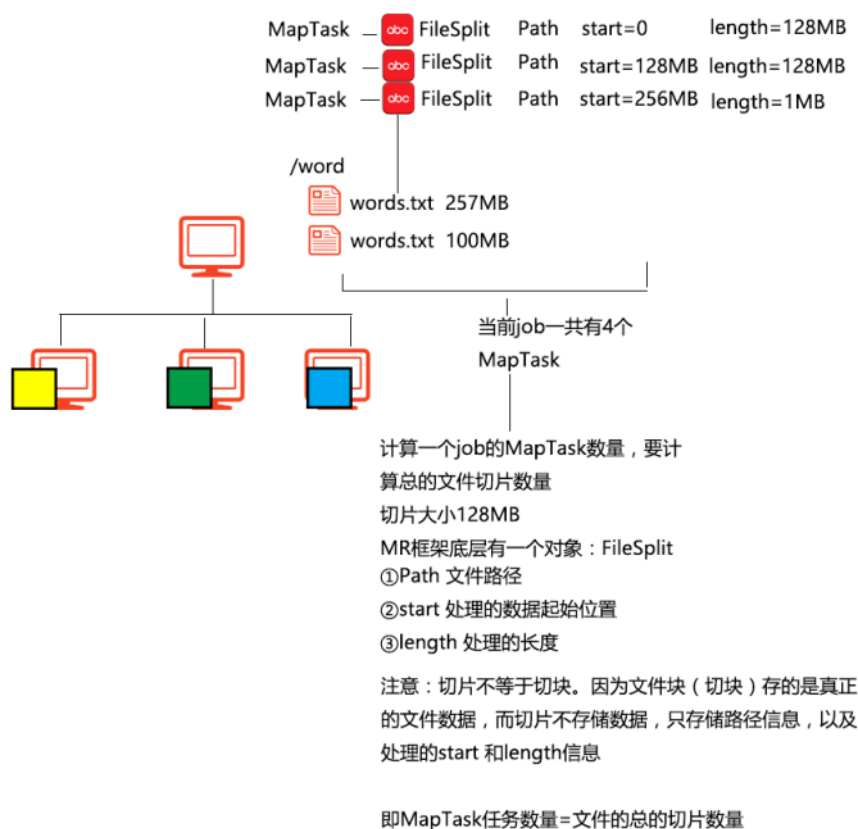
1.Mapper组件开发方式：写一个类，继承Mapper

2.Mapper组件的作用是定义 每一个MapTask具体要怎么处理数据。

比如一个文件，257MB，会生成3个MapTask。即三个MapTask处理逻辑是一样的

只是每个MapTask处理的数据不一样。

针对2的补充:



3.第一个泛型类型：LongWritable，对应的Mapper的输入key。

输入key是每行的行首偏移量

4.第二个泛型类型：Text，对应的Mapper的输入Value。

输入value是每行的内容

5.第三个泛型类型：对应的Mapper的输出key，这是程序员根据业务来定义

6.第四个泛型类型：对应的Mapper的输出value，这是程序员根据业务来定义

7.注意：初学时，第一个和第二个泛型写死。第三个和第四个不固定

8.Writable机制是Hadoop自身的序列化机制，常用的类型：

①LongWritable ②Text(String) ③IntWritable ④NullWritable

9.定义MapTask的任务逻辑是通过重写map()方法来实现的。

读取一行数据就会调用一次此方法，同时会把输入key和输入value传给程序员

10.在实际开发中，最重要的是拿到输入value(每行内容)

11.输出方法：通过context.write(输出key，输出value)

12.开发一个MapReduce程序（job），Mapper可以单独存储，此时，最后的输出的结果文件内容就是Mapper的输出。

13.Reducer组件不能单独存在，因为Reducer要依赖于Mapper的输出。当引入了Reducer之后，最后输出的结果文件的结果就是Reducer的输出。

Reduce组件知识点

1.reduce组件用于接收mapper组件的输出

2.reduce第一个泛型类型是reduce的输入key,需要和mapper的输出key类型一致

3.第二个泛型类型是reduce的输入value，需要和mapper的输出value类型一致

4.第三个泛型类型是reduce的输出key类型，根据具体业务决定

5.第四个泛型类型是reduce的输出value类型，根据具体业务决定

6.reduce收到map的输出，会按相同的key做聚合，形成key Iterable 形式然后通过reduce方法传给程序员。

7.reduce方法中的Iterable是一次性的，即遍历一次之后，再遍历，里面就没有数据了。所以，在某些业务场景，会涉及到多次操作此迭代器，处理的方法是：①先创建一个List ②把Iterable装到List ③多次去使用List即可

案例——求平均值

tom 69
tom 84
tom 68
jary 89
jary 90
jary 81
jary 35
rose 23
rose 100
rose 230



Mapper代码：

```
public class AverageMapper 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(" ");

        String name=data[0];

        int score=Integer.parseInt(data[1]);
```

```

        context.write(new Text(name), new IntWritable(score));

    }

}

```



Reduce代码：

```

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

    @Override

    protected void reduce(Text name, Iterable<IntWritable> scores,

        Reducer<Text, IntWritable, Text, IntWritable>.Context context)

        throws IOException, InterruptedException {

        int i=0;

        int score=0;

        for(IntWritable data:scores){

            score=score+data.get();

            i++;

        }

        int average=score/i;

        context.write(name, new IntWritable(average));

    }

}

```



Driver代码：

```

public class AverageDriver {

    public static void main(String[] args) throws Exception {

        Configuration conf=new Configuration();
    }
}

```

```

Job job=Job.getInstance(conf);

job.setJarByClass(AverageDriver.class);

job.setMapperClass(AverageMapper.class);
job.setReducerClass(AverageReducer.class);

job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);


job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);


FileInputFormat.setInputPaths(job, new
Path("hdfs://192.168.234.21:9000/average"));

FileOutputFormat.setOutputPath(job, new
Path("hdfs://192.168.234.21:9000/average/result"));

job.waitForCompletion(true);

}

}

```

案例——数据去重

192. 168. 234. 21

192. 168. 234. 22

192. 168. 234. 21

192. 168. 234. 21

192. 168. 234. 23

192. 168. 234. 21

192. 168. 234. 21

192. 168. 234. 21

192. 168. 234. 25

192. 168. 234. 21

192. 168. 234. 21

192. 168. 234. 26

192. 168. 234. 21

192. 168. 234. 27

192. 168. 234. 21

192. 168. 234. 27

192. 168. 234. 21

192. 168. 234. 29

192. 168. 234. 21

192. 168. 234. 26

192. 168. 234. 21

192. 168. 234. 25

192. 168. 234. 25

192.168.234.21

192.168.234.22

192.168.234.21



Mapper代码：

```
public class DistinctMapper extends Mapper<LongWritable, Text, Text, NullWritable>
{

    @Override

    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text,
    Text, NullWritable>.Context context)

        throws IOException, InterruptedException {

        context.write(value, NullWritable.get());

    }

}
```



Reduce代码：

```
public class DistinctReducer extends Reducer<Text, NullWritable, Text, NullWritable>
{

    @Override

    protected void reduce(Text key, Iterable<NullWritable> values,

        Reducer<Text, NullWritable, Text, NullWritable>.Context context)

        throws IOException, InterruptedException {

        context.write(key, NullWritable.get());

    }

}
```

```
}
```

```
}
```



Driver:

```
public class DistinctDriver {

    public static void main(String[] args) throws Exception {

        Configuration conf=new Configuration();

        Job job=Job.getInstance(conf);

        job.setJarByClass(DistinctDriver.class);

        job.setMapperClass(DistinctMapper.class);

        job.setReducerClass(DistinctReducer.class);

        job.setMapOutputKeyClass(Text.class);

        job.setMapOutputValueClass(NullWritable.class);

        job.setOutputKeyClass(Text.class);

        job.setOutputValueClass(NullWritable.class);

        FileInputFormat.setInputPaths(job, new

        Path("hdfs://192.168.234.21:9000/distinct"));

        FileOutputFormat.setOutputPath(job, new

        Path("hdfs://192.168.234.21:9000/distinct/result"));

    }

}
```

```
        job.waitForCompletion(true);  
    }  
}
```

案例一求最大值最小值

假设我们需要处理一批有关天气的数据，其格式如下：

按照ASCII码存储，每行一条记录。每行共24个字符（包含符号在内）

第9、10、11、12字符为年份，第20、21、22、23字符代表温度，求每年的最高温度

2329999919500515070000

9909999919500515120022

9909999919500515180011

9509999919490324120111

6509999919490324180078

9909999919370515070001

9909999919370515120002

9909999919450515180001

6509999919450324120002

8509999919450324180078

在map过程中，通过对每一行字符串的解析，得到年-温度的key-value对作为输出：

(1950, 0)

(1950, 22)

(1950, 11)

(1949, 111)

(1949, 78)

(1937, 1)

(1945, 1)

(1945, 2)

(1945, 78)

在reduce过程，将map过程中的输出，按照相同的key将value放到同一个列表中作为reduce的输入

(1950, [0, 22, 11])

(1949, [111, 78])

(1937, [1, 2])

(1945, [1, 2, 78])

在reduce过程中，在列表中选择出最大的温度，将年-最大温度的key-value作为输出：

(1950, 22)

(1949, 111)

(1937, 1)

(1945, 78)



Mapper代码：

```
public class MaxMapper 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 year=line.substring(8,12);

        int tempature=Integer.parseInt(line.substring(18,22));

        context.write(new Text(year), new IntWritable(tempature));

    }

}
```



Reducer代码：

```
public class MaxReducer extends Reducer<Text, IntWritable, Text, IntWritable>{

    @Override

    protected void reduce(Text year, Iterable<IntWritable> tempratures,

        Reducer<Text, IntWritable, Text, IntWritable>.Context context)

        throws IOException, InterruptedException {

        int max=0;

        for(IntWritable data:tempratures){

            if(max<data.get()){

                max=data.get();

            }

        }

        context.write(year, new IntWritable(max));

    }

}
```



Driver代码：

```
public class MaxDriver {

    public static void main(String[] args) throws Exception {

        Configuration conf=new Configuration();

        Job job=Job.getInstance(conf);

    }

}
```

```
job.setJarByClass(MaxDriver.class);

job.setMapperClass(MaxMapper.class);

job.setReducerClass(MaxReducer.class);


job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);


job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);


FileInputFormat.setInputPaths(job, new
Path("hdfs://192.168.234.21:9000/max"));

FileOutputFormat.setOutputPath(job, new
Path("hdfs://192.168.234.21:9000/max/result"));

job.waitForCompletion(true);

}

}
```

课后作业：案例——找爷孙关系

2018年8月1日 12:59

案例文件：

褚英 努尔哈赤
皇太极 努尔哈赤
多尔袞 努尔哈赤
多铎 努尔哈赤
豪格 皇太极
福临(顺治) 皇太极
福全 福临(顺治)
玄烨(康熙) 福临(顺治)

第一列是孩子辈，第二列是父母辈。现在要得到爷孙辈的关系

比如最后的输出结果：

爷爷辈:[努尔哈赤]-->孙子辈:[福临(顺治), 豪格]

爷爷辈:[皇太极]-->孙子辈:[玄烨(康熙), 福全]