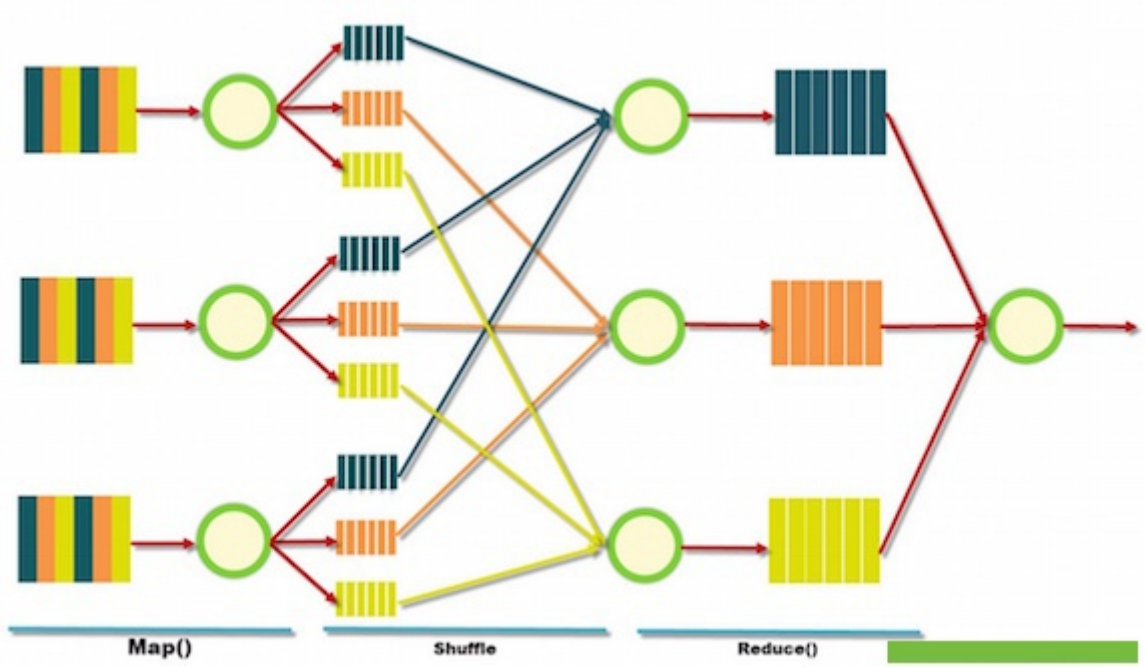
MapReduce的详细过程

写在前面的话

MapReduce作为hadoop的编程框架，是工程师最常接触的部分，也是除去了网络环境和集群配置之外对整个Job执行效率影响很大的部分，所以很有必要深入了解整个过程。本文写作的目的在于使得读者对整个MapReduce过程有比较细致的了解，当自己需要定制MapReduce行为时，知道该重写哪些类和方法。在写作时，我贴了部分认为重要的源码和接口，并跟着自己的理解，对于某些内容，结合了自己在工作中遇到的问题，给出了实践参考。

总体概览



[本图摘自http://blog.sqlauthority.com/]

比较High Level的来看，整个MapReduce过程分为三步：

- Map：读取输入，做初步的处理，输出形式的中间结果
- Shuffle：按照key对中间结果进行排序聚合，输出给reduce线程
- Reduce：对相同key的输入进行最终的处理，并将结果写入到文件中

用经典的WordCount例子来简单说明一下上面的过程。假设我们现在要做的是统计一个文本中单词的个数，我们将文件切分成几个部分，然后创建多个Map线程，处理这些输入，输出的中间结果是的形式，shuffle过程将同样Key的元组，也就是word相同的，分配到同样的reduce线程中，reduce线程汇总同一个word的元组个数，最终输出。

我这么一说，你是不是感觉已经理解MapReduce了？差不多吧，但是理解与深入理解是1与10000的差距，下面让我提几个细节方面的问题：

1. 原始数据是怎么切分的，又是以什么形式传递给Map线程的？
2. 有多少个map线程，怎样控制他们？
3. 输出写到磁盘的过程是怎样的？
4. 如果要保证同一个中间结果key交给同一个reduce，要不要排序？什么时候排序？
5. 满足什么条件的中间结果会调用一次reduce方法，满足什么条件的中间结果会交给一个reduce线程？
6. 有多少reduce线程，怎样控制他们？
7. 有多少输出文件？ ...

是不是有很多问题都看不懂啦？没关系，下面我就详细讲解这个过程。

Yarn的资源分配与任务调度

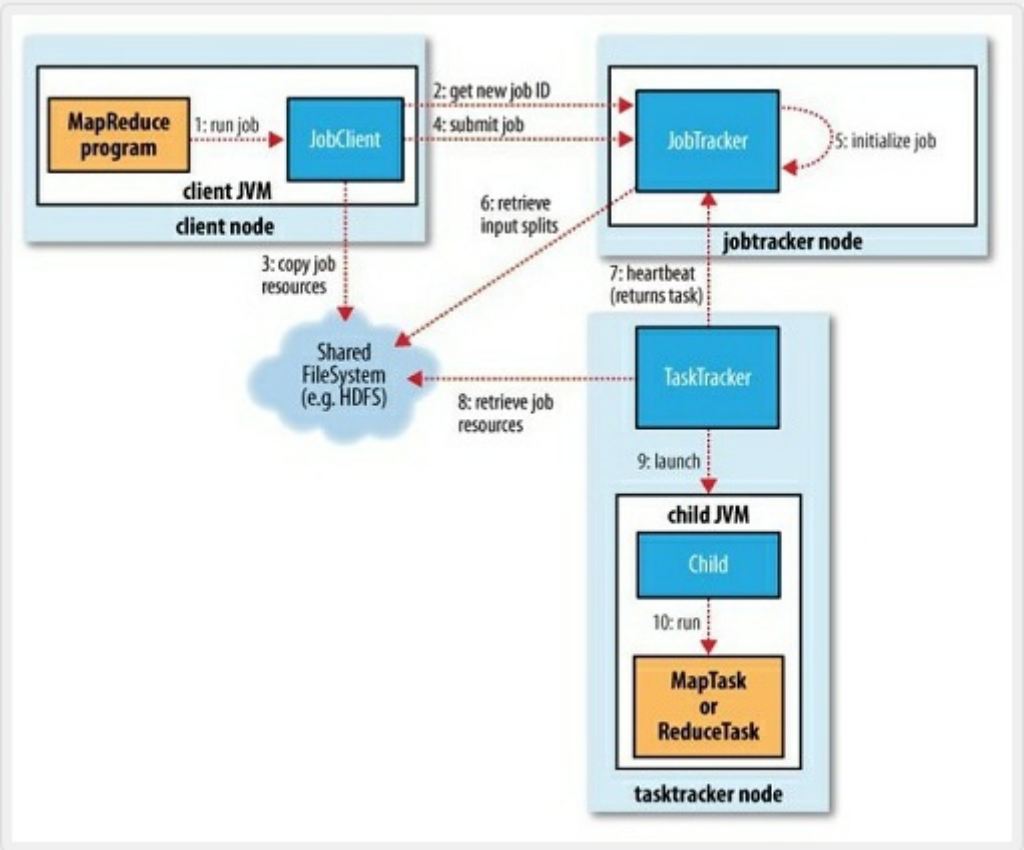
之所以要讲解这一部分，是因为MapReduce过程牵扯到了框架本身的东西，我们得知道计算线程是怎么来的，怎么没的。

Hadoop由1.0进化成2.0，变更还是很大的，1.0里整个job的资源分配，任务调度和监控管理都是由一个JobTracker来做的，扩展性很差，2.0对整个过程重新设计了一下，我们重点来看2.0的内容。

一个Job要在集群中运行起来，需要几个条件，首先，运算资源，可能包括内存，cpu等，其次，得有一个任务的调度算法，安排运行的先后顺序，最后，得知道工作进行的顺不顺利，并把情况及时的反馈给上级，以便及时的做出响应。下面分别说明。

下面我们首先看看1.0时代hadoop集群是怎么管理资源和调度任务的。

hadoop1.0的资源管理



[本图来自百度百科的“MapReduce”词条]

对于一个集群来说，资源有很多维度，比如内存，CPU等，1.0时代将节点上的资源切成等份，使用slot的概念来抽象，根据对资源占用情况的不同，又可细分为Map slot和reduce slot。slot代表一种运行的能力，像许可证一样，MapTask只有获得了Map slot后才可以执行，ReduceTask同理。对于一个节点，有多少slot是事先配置好的。

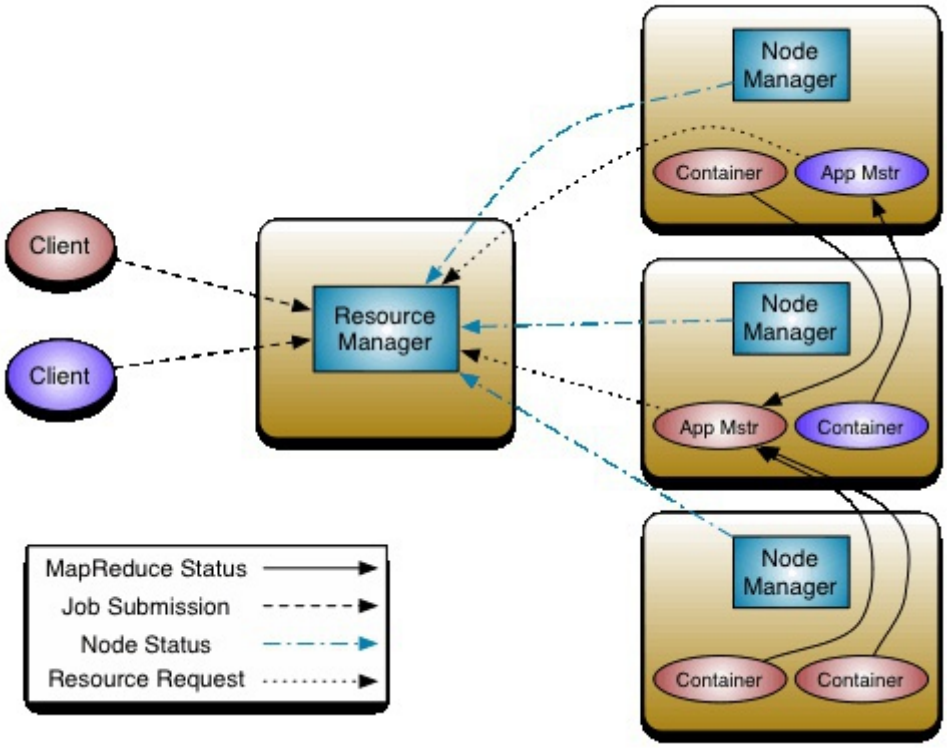
JobTracker和TaskTracker共同管理这些slot，其中JobTracker运行在Name Node上，负责资源的分配和任务的调度，TaskTracker运行在Data Node上，负责所在节点上资源的监控和task的管理。具体一点，当用户的任务提交给jobtracker之后，jobtracker根据任务的情况决定要启动多少MapTask和ReduceTask，然后根据TaskTracker反馈的slot使用情况（以及其他的因素，比如根据数据的存储情况），决定给哪几个TaskTracker分配多少个MapTask和多少个ReduceTask。接收到任务后，TaskTracker负责启动JVM来运行这些Task，并把运行情况实时反馈给JobTracker。

注意，TaskTracker**只有监控权，没有调度权**，也就是它只能把运行情况反馈给JobTracker，在他这里有多少个Task，当task失败时，重启task之类的管理权限，都在JobTracker那里。**JobTracker的任务管理是Task级别的**，也即JobTracker负责了集群资源的管理，job的调度，以及一个Job的每个Task的调度与运行。

打个比方，JobTracker是一个极度专权的君王，TaskTracer是大臣，君王握有所有的权利，大臣们被架空，君王说事情怎么做，底下的就得怎么做，大臣只管执行，并把进行情况告诉君王，如果事情搞砸了，大臣也不得擅作主张的重新做一遍，得上去请示君王，君王要么再给他一次机会，要么直接拖出去砍了，换个人完成。

极度专权早晚累死，而且一个人的力量终归是有限的，这也是1.0时代很大的问题。所以新时代采取了全新的设计。

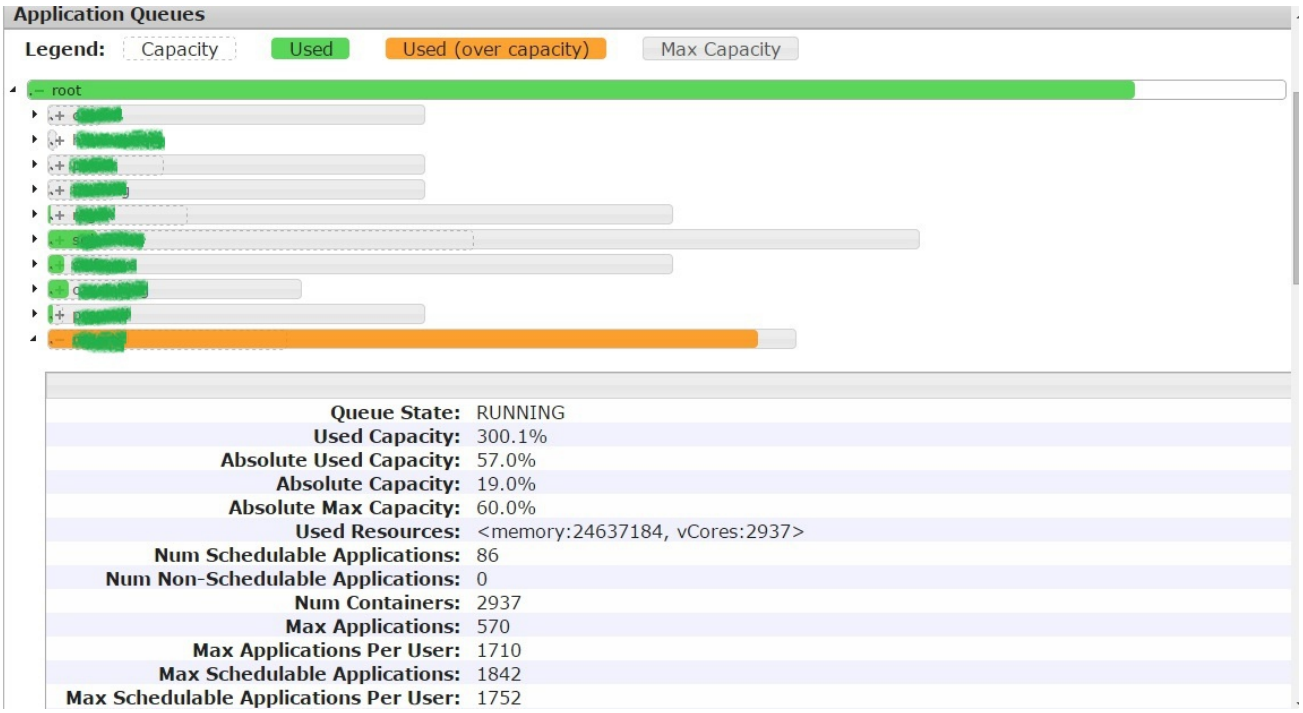
Yarn的资源控制与任务调度



[本图摘自http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html]

Yarn用Container的概念来抽象资源，Container描述了自己的位置，自己拥有的CPU，内存等资源的数量。Container跟任务完全独立了，是一个完全硬件的抽象。比1.0里使用计算时槽更加细粒度，也更易于理解。

资源控制由Resource Manage（RM)和Node Manager(NM)两个角色参与，其中Node Manager管理所在node上的container，并把资源的使用情况汇报给Resource Manager，Resource Manager通过Node Manager返回的信息，掌握着整个集群的资源情况。为了便于管理，Hadoop集群的管理员可以建立多个队列，每个队列配置一定量的资源，用户可以向一个或多个队列提交Job。作为集群的用户，可以到50030端口查看集群的队列的分配和负载情况。



当一个用户提交了一个job给ResourceManager，ResourceManager并不是直接衡量它所需的资源并调度，而是下放给一个Application Master（AM）的角色，这个AM全权负责用户提交的这个Job，它会根据Job的情况向RM申请资源，RM告诉AM它可以使用的Container的信息，AM再将自己Job的task放到这些Container中运行并监控。如果有失败的task，AM可以根据情况选择重启task。

有几个关键的点我列出来，以确保理解正确：

1. 集群的资源监控由RM与NM合作完成，任务调度与监控由RM与AM完成，结构更加清晰。

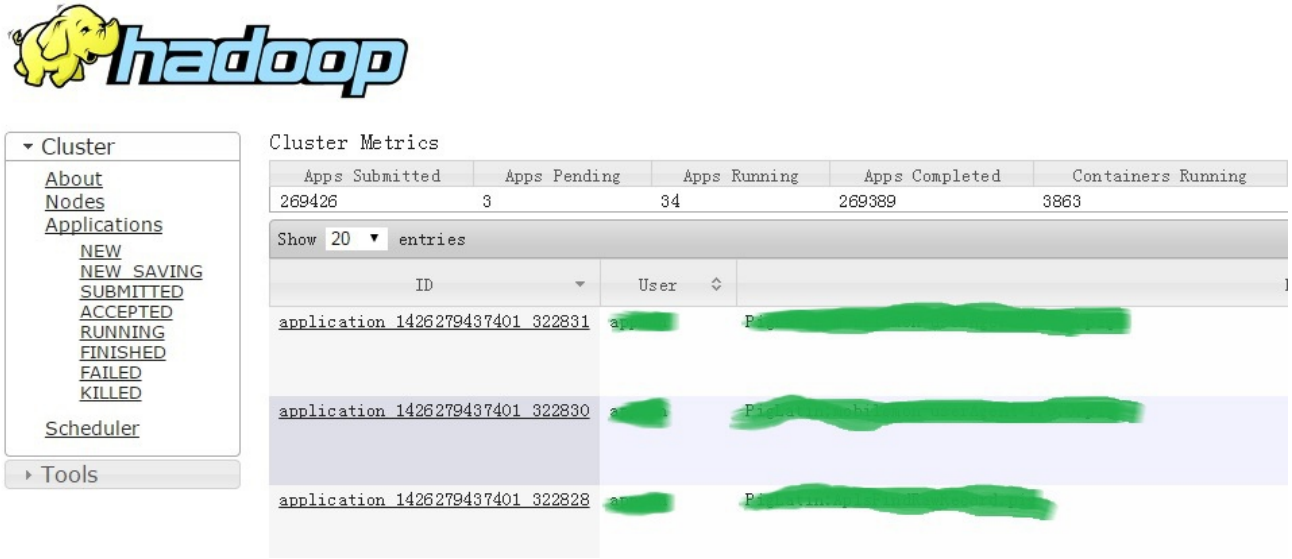
2. RM对任务的管理是Job级别的，即它只负责为整个Job分配资源，并交给AM去管理。RM得到了大大的解放。

3. 与TaskTracker相比，AM拥有更多的权利，它可以申请资源并全权负责task级别的运行情况。

4. 与TaskTracker相比，AM可以使用其他机器上的计算资源（即Container）。这些资源也不再有Map和Reduce的区别。

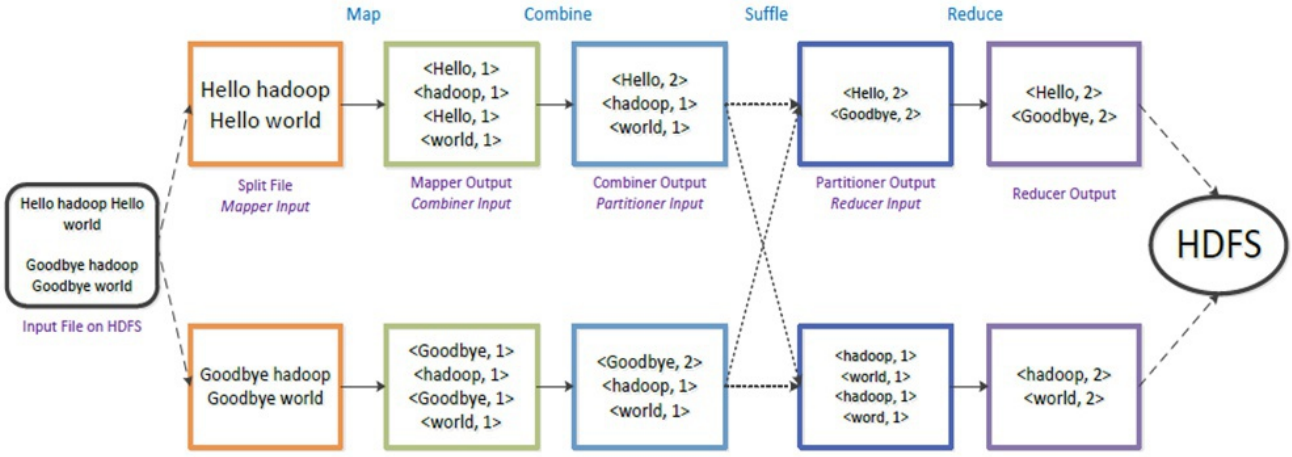
继续上面的例子。我用壮丁来比喻Container，壮丁有很多属性，比如家乡（location），力气（内存），财产（CPU），君王（RM）通过锦衣卫（NM）来掌握各个地方（Node）壮丁的使用情况。当有百姓提出一个要求（提交一个Job），比如兴修水利，君王不再事无巨细的过问这件事情，而是叫一个合适的大臣（AM）过来，比如此例中的水利大臣，问他需要多少人，多少钱，然后衡量一下国力，播一些壮丁给他用。水利大臣可以使用全国范围内的壮丁，对他们有绝对的领导权，让他们干嘛就得干嘛。事情要么圆满完成，水利大臣给君王报喜，要么发现难度太大啊，尝试了好多办法都失败了（job尝试次数到达一定数量），只好回去请罪。

君王遵循政务公开的原则，所有job的运行情况都可以通过50030端口查看：



好了，讲了这么一大通，我想关于Job怎么跑起来，task怎么来怎么没，应该有个概念了。用户将自己的代码上传到集群的一个client Node上，运行代码，代码里会对自己的job进行配置，比如输入在哪，有哪些依赖的jar包，输出写到哪，以什么格式写，然后提交给ResourceManager，ResourceManager会在一个Node上启动ApplicationMaster负责用户的这个Job，AM申请资源，得到RM的批准和分配后，在得到的Container里启动MapTask和ReduceTask，这两种task会调用我们编写的Mapper和Reducer等代码，完成任务。任务的运行情况可以通过web端口查看。

MapReduce计算框架最重要的两个类是Mapper和Reducer，用户可以继承这两个类完成自己的业务逻辑，下面以这两个类的输入输出为主线详细讲解整个过程。例子总是最容易被人理解的，所以讲解过程有看不懂的，可以回来查看这个简单的job。用户想使用MapReduce的过程统计一组文件中每个单词出现的次数，与经典的WordCount不同的是，要求大写字母开头的单词写到一个文件里面，小写的写到另一个文件中。



Mapper的输入

所谓源码之前，了无秘密，先上mapper的源码。

Mapper的源码

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

 /**
  * The <code>Context</code> passed on to the {@link Mapper} implementations.
  */
 public abstract class Context
   implements MapContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
 }

 /**
  * Called once at the beginning of the task.
  */
 protected void setup(Context context
                      ) throws IOException, InterruptedException {
   // NOTHING
 }

 /**
  * Called once for each key/value pair in the input split. Most applications
  * should override this, but the default is the identity function.
  */
 @SuppressWarnings("unchecked")
 protected void map(KEYIN key, VALUEIN value,
                    Context context) throws IOException, InterruptedException {
   context.write((KEYOUT) key, (VALUEOUT) value);
 }

 /**
  * Called once at the end of the task.
  */
 protected void cleanup(Context context
                        ) throws IOException, InterruptedException {
   // NOTHING
 }
```

```
  /**
   * Expert users can override this method for more complete control over the
   * execution of the Mapper.
   * @param context
   * @throws IOException
   */
  public void run(Context context) throws IOException, InterruptedException {
    setup(context);
    try {
      while (context.nextKeyValue()) {
        map(context.getCurrentKey(), context.getCurrentValue(), context);
      }
    } finally {
      cleanup(context);
    }
  }
}
```

可以简单的说，Mapper的输入来自于Context。我们先看一下MapContext的实现：

```
public class MapContextImpl<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
    extends TaskInputOutputContextImpl<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
    implements MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  private RecordReader<KEYIN,VALUEIN> reader;
  private InputSplit split;

  public MapContextImpl(Configuration conf, TaskAttemptID taskid,
                        RecordReader<KEYIN,VALUEIN> reader,
                        RecordWriter<KEYOUT,VALUEOUT> writer,
                        OutputCommitter committer,
                        StatusReporter reporter,
                        InputSplit split) {
    super(conf, taskid, writer, committer, reporter);
    this.reader = reader;
    this.split = split;
  }

  /**
```

```
   * Get the input split for this map.
   */
  public InputSplit getInputSplit() {
    return split;
  }

  @Override
  public KEYIN getCurrentKey() throws IOException, InterruptedException {
    return reader.getCurrentKey();
  }

  @Override
  public VALUEIN getCurrentValue() throws IOException, InterruptedException {
    return reader.getCurrentValue();
  }

  @Override
  public boolean nextKeyValue() throws IOException, InterruptedException {
    return reader.nextKeyValue();
  }

}
```

MapContextImpl类组合了两个类型的对象，即InputSplit和RecordReader，并封装了获取输入的Key和Value的方法，在深入探讨InputSplit和RecordReader之前，我们先看一下这个Context是怎么传递给我们编写的Mapper函数的。下面是我从MapTask类中摘出的一段代码：

```
public class MapTask extends Task {
private <INKEY,INVALUE,OUTKEY,OUTVALUE>
  void runNewMapper(final JobConf job,
                    final TaskSplitIndex splitIndex,
                    final TaskUmbilicalProtocol umbilical,
TaskReporter reporter
                    ) throws IOException, ClassNotFoundException,
                        InterruptedException {
    // make a task context so we can get the classes
    org.apache.hadoop.mapreduce.TaskAttemptContext taskContext =
      new org.apache.hadoop.mapreduce.task.TaskAttemptContextImpl(job,
```

```
                                        getTaskID(),
                                        reporter);
// make a mapper
org.apache.hadoop.mapreduce.Mapper<INKEY,INVALUE,OUTKEY,OUTVALUE> mapper =
  (org.apache.hadoop.mapreduce.Mapper<INKEY,INVALUE,OUTKEY,OUTVALUE>)
    ReflectionUtils.newInstance(taskContext.getMapperClass(), job);
// make the input format
org.apache.hadoop.mapreduce.InputFormat<INKEY,INVALUE> inputFormat =
  (org.apache.hadoop.mapreduce.InputFormat<INKEY,INVALUE>)
    ReflectionUtils.newInstance(taskContext.getInputFormatClass(), job);
// rebuild the input split
org.apache.hadoop.mapreduce.InputSplit split = null;
split = getSplitDetails(new Path(splitIndex.getSplitLocation()),
    splitIndex.getStartOffset());
LOG.info("Processing split: " + split);

org.apache.hadoop.mapreduce.RecordReader<INKEY,INVALUE> input =
  new NewTrackingRecordReader<INKEY,INVALUE>
    (split, inputFormat, reporter, taskContext);
...
org.apache.hadoop.mapreduce.MapContext<INKEY, INVALUE, OUTKEY, OUTVALUE>
    mapContext =
      new MapContextImpl<INKEY, INVALUE, OUTKEY, OUTVALUE>(job, getTaskID(),
          input, output,
          committer,
          reporter, split);

org.apache.hadoop.mapreduce.Mapper<INKEY,INVALUE,OUTKEY,OUTVALUE>.Context
        mapperContext =
          new WrappedMapper<INKEY, INVALUE, OUTKEY, OUTVALUE>().getMapContext(
              mapContext);
     try {
          input.initialize(split, mapperContext);
          mapper.run(mapperContext);
          ...
  }finally{
       closeQuietly(input);
       ...
  }
```

```
  }
}
```

从代码中可以看出，我们在客户端提交Job之前所进行的配置以JobContext的形式传递给了MapTask，在MapTask的runNewMapper()的方法中，它使用反射实例化出了用户指定的Mapper类和inputFormat类，并新建了InputSplit和RecorderReader用来实例化上面提交的MapContext，MapContext以参数的形式被传递给了包装类WrappedMapper，在对input进行初始化后，以

```
mapper.run(mapperContext);
```

的形式正式调用我们用户的代码。

InputSplit

源码中对InputSplit类的描述是：

```
/**
 * <code>InputSplit</code> represents the data to be processed by an
 * individual {@link Mapper}.
 *
 * <p>Typically, it presents a byte-oriented view on the input and is the
 * responsibility of {@link RecordReader} of the job to process this and present
 * a record-oriented view.
 */
public abstract class InputSplit {
  public abstract long getLength() throws IOException, InterruptedException;
  public abstract
  String[] getLocations() throws IOException, InterruptedException;
}
```

更易于理解的表述是，InputSplit决定了一个Map Task需要处理的输入。更进一步的，有多少个InputSplit，就对应了多少个处理他们的Map Task。从接口上来看，InputSplit中并没有存放文件的内容，而是指定了文件的文件的位置以及长度。

既然inputsplit与MapTask之间是一一对应的，那么我们就可以通过控制InputSplit的个数来调整MapTask的并行性。当文件量一定时，InputSplit越小，并行性越强。inputsplit的大小并不是任意的，

虽然最大值和最小值都可以通过配置文件来指定，但是最大值是不能超过一个block大小的。

Block是什么？用户通过HDFS的接口，看到的是一个完整文件层面，在HDFS底层，文件会被切成固定大小的Block，并冗余以达到可靠存储的目的。一般默认大小是64MB，可以调节配置指定。

InputSplit是从字节的角度来描述输入的，回头查看一下Mapper，它里面没有这种东西啊，用到的Key，Value是从哪来的？有请RecordReader。

RecordReader

按照惯例，先上源码：

```
public abstract class RecordReader<KEYIN, VALUEIN> implements Closeable {

  /**
   * Called once at initialization.
   * @param split the split that defines the range of records to read
   * @param context the information about the task
   * @throws IOException
   * @throws InterruptedException
   */
  public abstract void initialize(InputSplit split,
TaskAttemptContext context
                                  ) throws IOException, InterruptedException;

  /**
   * Read the next key, value pair.
   * @return true if a key/value pair was read
   * @throws IOException
   * @throws InterruptedException
   */
  public abstract
  boolean nextKeyValue() throws IOException, InterruptedException;

  /**
   * Get the current key
   * @return the current key or null if there is no current key
   * @throws IOException
   * @throws InterruptedException
```

```
   */
  public abstract
  KEYIN getCurrentKey() throws IOException, InterruptedException;

  /**
   * Get the current value.
   * @return the object that was read
   * @throws IOException
   * @throws InterruptedException
   */
  public abstract
  VALUEIN getCurrentValue() throws IOException, InterruptedException;

  /**
   * The current progress of the record reader through its data.
   * @return a number between 0.0 and 1.0 that is the fraction of the data read
   * @throws IOException
   * @throws InterruptedException
   */
  public abstract float getProgress() throws IOException, InterruptedException;

  /**
   * Close the record reader.
   */
  public abstract void close() throws IOException;
}
```

啊哈，InputSplit原来是RecordReader的一个参数啊。recordReader从InputSplit描述的输入里取出一个KeyValue，作为mapper.map()方法的输入，跑一遍Map方法。打个比方，InputSplit像一桌大餐，吃还是得一口一口吃，怎样算一口，就看RecordReader怎么实现了。

好了，如果我想自己实现InputSplit和RecordReader，应该写在哪呢？下面就讲InputFormat。

InputFormat

上文我们提到了InputFormat，这个类我们在配置Job的时候经常会指定它的实现类。先来看接口。

```
public abstract class InputFormat<K, V> {
  public abstract
    List<InputSplit> getSplits(JobContext context
                        ) throws IOException, InterruptedException;
  public abstract
  RecordReader<K,V> createRecordReader(InputSplit split,
TaskAttemptContext context
                                ) throws IOException,
                                    InterruptedException;
}
```

明白了吧，InputSplit是在getSplit函数里面算出来的，RecordReader也是在这里Create出来的。如果你想以自己的方式读取输入，就可以自己写一个InputFormat的实现类，重写里面的方法。

当然，如果你说我很懒，不想自己写怎么办？好办，之所以要用框架，很重要的一点就是人家提供了默认实现啦。WordCount里面一般用的是TextInputFormat，我们看一下它的实现。

```
ublic class TextInputFormat extends FileInputFormat<LongWritable, Text>
 implements JobConfigurable {
 public RecordReader<LongWritable, Text> getRecordReader(
                                InputSplit genericSplit, JobConf job,
                                Reporter reporter)
  throws IOException {

  reporter.setStatus(genericSplit.toString());
  String delimiter = job.get("textinputformat.record.delimiter");
  byte[] recordDelimiterBytes = null;
  if (null != delimiter) {
   recordDelimiterBytes = delimiter.getBytes(Charsets.UTF_8);
  }
  return new LineRecordReader(job, (FileSplit) genericSplit,
      recordDelimiterBytes);
  }
 }
```

有没有一下明白了的感觉？它实现了自己的getRecordReader方法，里面从配置中取了Delimiter，这个东西的默认值是"\n"！然后返回了以Delimiter划分的一个LineRecordReader，知道为什么你制定了InputFormat之后，Mapper里面读到的就是一行一行的输入了吧。

在我们加强版的WordCount里，也完全可以使用默认实现的TextInputFormat。关于Mapper的输入暂时就讲这些，下面我们来看Mapper的输出。

Mapper的输出

注意到上文贴出的Mapper的默认实现的map方法中，是将Key和Value直接写入到context当中，我们已经知道了context是从MapContextImpl来的，那这个Write方法是怎么回事?

Context.Write的来历

Write方法是它从MapContextImpl父类TaskInputOutputContextImpl继承来的，看一下这个类的部分代码:

```
public abstract class TaskInputOutputContextImpl<KEYIN,VALUEIN,KEYOUT,VALUEOUT>

extendsTaskAttemptContextImpl

implements TaskInputOutputContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

private RecordWriter<KEYOUT,VALUEOUT> output;
private OutputCommitter committer;

public TaskInputOutputContextImpl(Configuration conf, TaskAttemptID taskid,
RecordWriter<KEYOUT,VALUEOUT> output,
OutputCommitter committer,
StatusReporter reporter) {
super(conf, taskid, reporter);
this.output = output;
this.committer = committer;
}

/**

* Generate an output key/value pair.

*/
```

```
public void write(KEYOUT key,VALUEOUT value
) throws IOException, InterruptedException {
output.write(key, value);
}
}
```

注意到了有个RecordWriter类，跟我们在上文分析过的RecordReader一看就是兄弟嘛，作用你也肯定猜到了，就是将一个Key value对写入到输出文件中。回过头来看它的输出类是怎么从MapTask中传入的：

```
org.apache.hadoop.mapreduce.RecordWriter output = null;
// get an output object
if (job.getNumReduceTasks() == 0) {
output =
new NewDirectOutputCollector(taskContext, job, umbilical, reporter);
} else {
output = new NewOutputCollector(taskContext, job, umbilical, reporter);
}
```

判断条件满足时，说明这个Job没有ReduceTask，这时RecordWriter被实例化成了NewDirectOutputCollector，否则的话，实例化为NewOutputCollector。来具体看看这两个内部类。

```
private class NewDirectOutputCollector<K,V>
extends org.apache.hadoop.mapreduce.RecordWriter<K,V> {
private final org.apache.hadoop.mapreduce.RecordWriter out;
...
out = outputFormat.getRecordWriter(taskContext);
...
out.write(key, value);
}
```

前者直接调用了OutputFormat来实例化自己，我们写Job的时候一般会指定Job的OutputFormat，这个类在MapTask中是通过反射的方式引入的。可见，第一个分支的逻辑是会直接把map的输出写入到我们整个Job的输出当中。具体是怎么个写入的过程，我们留到reduce的输出中讲，毕竟那里才是最常规的会写输出文件的地方。

```
rivate class NewOutputCollector<K,V>
extends org.apache.hadoop.mapreduce.RecordWriter<K,V> {
private final MapOutputCollector<K,V> collector;
private final org.apache.hadoop.mapreduce.Partitioner<K,V> partitioner;
private final int partitions;

@SuppressWarnings("unchecked")
NewOutputCollector(org.apache.hadoop.mapreduce.JobContext jobContext,
        JobConf job,
        TaskUmbilicalProtocol umbilical,
TaskReporter reporter
    ) throws IOException, ClassNotFoundException {

    collector = createSortingCollector(job, reporter);
    partitions = jobContext.getNumReduceTasks();
    if (partitions > 1) {
        partitioner = (org.apache.hadoop.mapreduce.Partitioner<K,V>)
        ReflectionUtils.newInstance(jobContext.getPartitionerClass(), job);
    } else {
        partitioner = new org.apache.hadoop.mapreduce.Partitioner<K,V>() {
        @Override
        public int getPartition(K key, V value, int numPartitions) {
            return partitions - 1;
        }
    };
    }
}
@Override
public void write(K key, V value) throws IOException, InterruptedException {
    collector.collect(key, value,
    partitioner.getPartition(key, value, partitions));
}
@Override
public void close(TaskAttemptContext context
) throws IOException,InterruptedException {
    try {
        collector.flush();
    } catch (ClassNotFoundException cnf) {
        throw new IOException("can't find class ", cnf);
    }
```

```
    collector.close();
  }
  }
```

这个内部类有两个成员变量，一个是MapOutputCollector， 一个是Partitioner。最终的写入调用的是MapOutputCollector的Write方法完成的。Partitioner的名气更大一些，我们先来介绍。

Partitioner

但凡了解一点MapReduce的人应该都知道这个类，它的作用是根据Key将Map的输出分区，然后发送给Reduce线程。有多少个Partition，就对应有多少个Reduce线程。Reduce线程的个数是可以在配置文件中设定的。上面代码的逻辑就是先读一下这个配置，看一下需要分到少个分区，如果分区数少于1，就实例化出一个Partitioner的默认实现，否则的话，用反射读取用户设置的实现类。

我们一般只重写它的一个方法：getPartition，参数是一个Key Value对以及Partition的总数，比较常见的实现是取Key的hashcode再对总的分区数取模。

注意，为了提高整个job的运行速度，reduce task应该尽可能均匀的接收Map的输出。partition作为Map输出分配的唯一参考标准，映射规则至关重要，partition返回值一样的Map的输出，将会交给一个reduce task，在实际工作中，我们就遇到了partition返回值不合理，好多Mapper的输出都压在一个reduce的task上，造成这个reduce task执行非常缓慢，整体的job一直结束不了的情况。**尽可能均匀的分配**partition!

MapOutputCollector

这个Collector我们可以自己实现，不过不是很常见。它有一个默认实现，叫MapOutputBuffer。有关MapOutputBuffer的分析，文献[4]有非常清晰的解释，值得一看。

MapOutputBuffer

Combiner的意思是局部的reduce，它可以在job配置的时候指定，实现的逻辑也跟reduce一致，Combiner的作用是可以减少Mapper和Reducer之间传输的数据量。以我们上面大小写敏感的word

count来说，同一台机器上的Mapper输出，可以先合并一次，将n个合并成的形式，再传递给reducer。

我把这个类里关键的方法列一下，源码比较多，就不贴了，可以参照那篇帖子。

init

```
public void init(Context context) throws IOException, ClassNotFoundException;
```

做典型的初始化的工作。比较重要的有，取得partition的总数，取得溢出的阈值，指定排序函数（默认是qsort），输出缓存和index数组相关的初始化，有关内容压缩的初始化，启动溢出时写磁盘的线程。

collect

```
public synchronized void collect(K key, V value, int partition) throws IOException;
```

对外最常调用的接口。判定参数传入的参数类型与用户对job的配置一不一致（”Type mismatch in key from map\value “），当缓冲区没有足够的位置存放当前键值对时，将缓冲区的内容溢出写到磁盘，否则的话，序列化键值对，写入到缓冲区数组，并将这个键值对的位置信息连同partition编号写入到index数组里。

flush

```
public void flush() throws IOException, ClassNotFoundException, InterruptedException;
```

当map所有的输出都收集完了之后，处理残留在缓冲区，没有溢写到磁盘的数据。

sortAndSpill

```
private void sortAndSpill() throws IOException, ClassNotFoundException, InterruptedException;
```

溢写的关键逻辑，其中会调用排序函数和combiner。Combiner的逻辑与reducer的完全一样，相当于每个map线程的局部预处理，通过对局部数据的合并，来起到减少shuffle阶段数据量的作用。

spillSingleRecord

```
private void spillSingleRecord(K key, V value, int partition) throws IOException;
```

当缓冲区没有达到溢出条件，并且放不下当前这条记录的时候会调用的方法，主要用来处理大键值对的边界条件。这种情况直接写磁盘。

compare&&swap

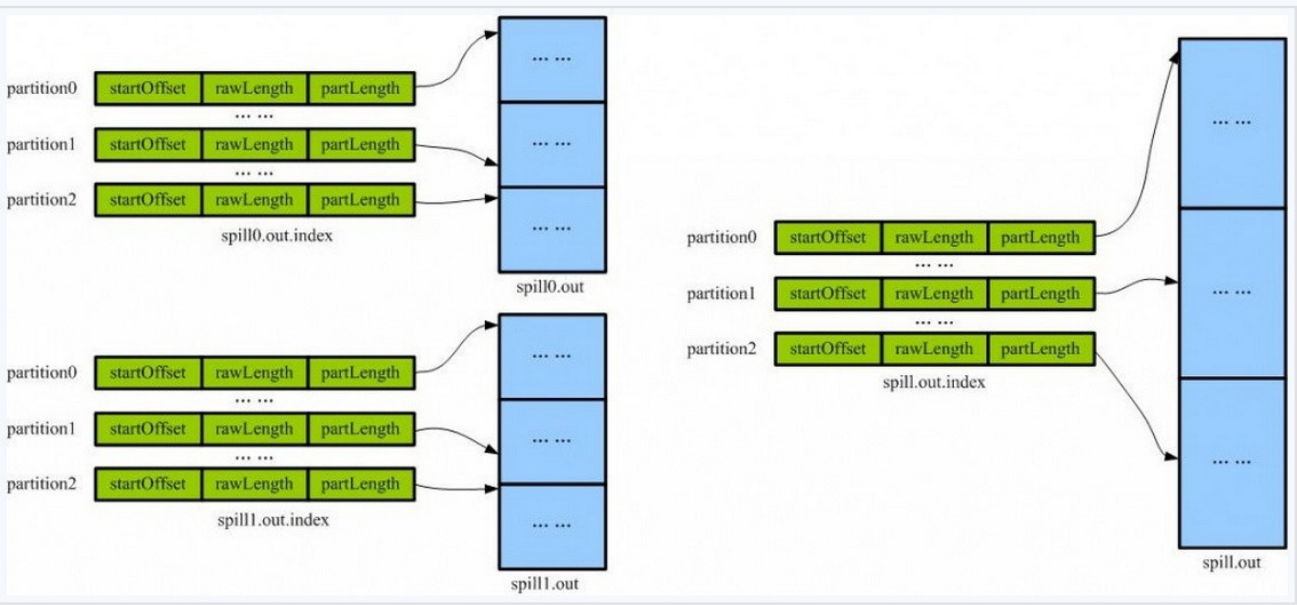
```
public int compare(int mi, int mj) {
    int kvi = this.offsetFor(mi % this.maxRec);
    int kvj = this.offsetFor(mj % this.maxRec);
    int kvip = this.kvmeta.get(kvi + 2);
    int kvjp = this.kvmeta.get(kvj + 2);
    return kvip != kvjp?kvip - kvjp:this.comparator.compare(this.kvbuffer, this.kvmeta.get(kvi + 1), this.kvmeta.get(kvi + 0) - this.kvmeta.get
(kvi + 1), this.kvbuffer, this.kvmeta.get(kvj + 1), this.kvmeta.get(kvj + 0) - this.kvmeta.get(kvj + 1));
}

public void swap(int mi, int mj) {
    int iOff = mi % this.maxRec * 16;
    int jOff = mj % this.maxRec * 16;
    System.arraycopy(this.kvbuffer, iOff, this.META_BUFFER_TMP, 0, 16);
    System.arraycopy(this.kvbuffer, jOff, this.kvbuffer, iOff, 16);
    System.arraycopy(this.META_BUFFER_TMP, 0, this.kvbuffer, jOff, 16);
}
```

从这两个函数可以猜出排序函数的行为。代码里出现的kvmeta就是上文中提到的index数组，他是kvbuffer的一种int视角，比较的对象就是它的两个元素，如果有乱序情况，交换的也是这两个元素的位置。

mergeParts

```
private void mergeParts() throws IOException, InterruptedException, ClassNotFoundException;
```

当文件全部溢出之后，会调用这个方法将小文件合并成大文件。



[本图摘自http://www.it165.net/pro/html/201402/9903.html]

合并前后的示意图还是很形象的。最终在shuffle的时候，只要根据index查找对应的数据就可以了。

业务场景

我一直没有想过MapTask是否会对输出自动排序，直到有一天我正真需要自己动手修改业务代码。

我在的组做的是数据处理，在我们的业务场景中，有两种数据结构，event和session，用户在电商网站上操作时，会在后台产生一系列的event，比如你查询了一件商品，后台就有一个查询event产生。event用guid和timestamp唯一标示，可能还含有其他的属性（比如ip等），guid可以简单的理解成用户的一种标示，event说白了是某个用户在某一时刻产生的某种动作。session的意思某个用户在一段连续时间内产生的动作集合，比event的抽象层次更高，它用sessionId和timestamp来标示，也有诸如这个session一共包含了多少个event这种统计信息。sessionId跟guid一样，某个用户在一定时间内是唯一的，session的timestamp取的是这段时间这个用户的第一个event的timestamp。

好了，我们需要写一个MapReduce的job，输入是event，输出是session。在map阶段，从event里面提取出key，然后同一个用户产生的event，应该一起在reduce阶段统计。既然有时序的问题，是不是在统计之前应该先排个序？可我翻遍了代码，都没有找到对key排序的逻辑，是前辈代码的巨大bug？

当然不是，在我们将guid与timestamp作为key输出时，MapTask已经按照这两个字段做了排序。注意，这种有序，指的只是当前MapTask局部输出的有序。从Mapper的输出，到真正Reducer的输入，还有很重要的一个过程要走。

Shuffle

从语义上说，Shuffle应该是Map和Reduce中间的过程，从源码的代码结构上看，shuffle过程是在reduceTask中得。前段时间在考公司的hadoop测试的时候，有这种变态的问题，说下面属于reduce过程的操作有。。至今不知道正确答案是什么。

ReduceTask有三个Phase，即copyPhase，sortPhase和reducePhase，主流的做法应该是将前两个phase归为Shuffle阶段，reducephase作为狭义的reduce过程。

ShuffleConsumerPlugin

Shuffle过程通过调用抽象类ShuffleConsumerPlugin来完成，它有个实现类，就叫做“Shuffle”。下面是Shuffle类最主要的run方法的实现：

```
@Override
 public RawKeyValueIterator run() throws IOException, InterruptedException {
  // Scale the maximum events we fetch per RPC call to mitigate OOM issues
  // on the ApplicationMaster when a thundering herd of reducers fetch events
  // TODO: This should not be necessary after HADOOP-8942
  int eventsPerReducer = Math.max(MIN_EVENTS_TO_FETCH,
      MAX_RPC_OUTSTANDING_EVENTS / jobConf.getNumReduceTasks());
  int maxEventsToFetch = Math.min(MAX_EVENTS_TO_FETCH, eventsPerReducer);

  // Start the map-completion events fetcher thread
  final EventFetcher<K,V> eventFetcher =
    new EventFetcher<K,V>(reduceId, umbilical, scheduler, this,
        maxEventsToFetch);
  eventFetcher.start();

  // Start the map-output fetcher threads
  boolean isLocal = localMapFiles != null;
  final int numFetchers = isLocal ? 1 :
    jobConf.getInt(MRJobConfig.SHUFFLE_PARALLEL_COPIES, 5);
  Fetcher<K,V>[] fetchers = new Fetcher[numFetchers];
  if (isLocal) {
    fetchers[0] = new LocalFetcher<K, V>(jobConf, reduceId, scheduler,
        merger, reporter, metrics, this, reduceTask.getShuffleSecret(),
```

```
        localMapFiles);
    fetchers[0].start();
  } else {
    for (int i=0; i < numFetchers; ++i) {
      fetchers[i] = new Fetcher<K,V>(jobConf, reduceId, scheduler, merger,
                                     reporter, metrics, this,
                                     reduceTask.getShuffleSecret());
      fetchers[i].start();
    }
  }

  // Wait for shuffle to complete successfully
  while (!scheduler.waitUntilDone(PROGRESS_FREQUENCY)) {
    reporter.progress();

    synchronized (this) {
      if (throwable != null) {
        throw new ShuffleError("error in shuffle in " + throwingThreadName,
                               throwable);
      }
    }
  }

  // Stop the event-fetcher thread
  eventFetcher.shutDown();

  // Stop the map-output fetcher threads
  for (Fetcher<K,V> fetcher : fetchers) {
    fetcher.shutDown();
  }

  // stop the scheduler
  scheduler.close();

  copyPhase.complete(); // copy is already complete
  taskStatus.setPhase(TaskStatus.Phase.SORT);
  reduceTask.statusUpdate(umbilical);

  // Finish the on-going merges...
```

```
    RawKeyValueIterator kvIter = null;
    try {
      kvIter = merger.close();
    } catch (Throwable e) {
      throw new ShuffleError("Error while doing final merge " , e);
    }

    // Sanity check
    synchronized (this) {
      if (throwable != null) {
        throw new ShuffleError("error in shuffle in " + throwingThreadName,
                               throwable);
      }
    }

    return kvIter;
  }
```

Shuffle的时候，会先判断是不是local run的，如果不是的话，会默认启动5个Fetcher线程拉取map的输出，Fetcher会先找到一个主机，确定这台机器上它要拉取的map task的输出，然后使用http协议获取response的stream，交给MapOutput类型的对象去完成具体的下载任务。

当文件拉取完成，就会进入sort阶段。注意到我们拉取到数据都是局部有序的，因此，排序的过程，实际上也就是一个Merge的过程。Copy phase结束之后，Shuffle会调用

```
kvIter = merger.close();
```

方法来得到排序完成的map的key value输出。

MapOutput

MapOutput有两个实现类，即OnDiskMapOutput和InMemoryMapOutput，具体哪一个被实例化，是看当前要shuffle的数据适不适合放到内存中。

OnDiskMapOutput的行为如下所示：

```
final int BYTES_TO_READ = 64 * 1024;
 byte[] buf = new byte[BYTES_TO_READ];
 while (bytesLeft > 0) {
  int n = input.read(buf, 0, (int) Math.min(bytesLeft, BYTES_TO_READ));
  if (n < 0) {
   throw new IOException("read past end of stream reading " +
                         getMapId());
  }
  disk.write(buf, 0, n);
  bytesLeft -= n;
  metrics.inputBytes(n);
  reporter.progress();
 }
```

InMemoryMapOutput的行为如下：

```
public static void readFully(InputStream in, byte buf[],
    int off, int len) throws IOException {
  int toRead = len;
  while (toRead > 0) {
   int ret = in.read(buf, off, toRead);
   if (ret < 0) {
    throw new IOException( "Premature EOF from inputStream");
   }
   toRead -= ret;
   off += ret;
  }
}
```

代码比较简单，前者有个buffer，一边读一边写文件，后者将数据缓存在一个byte数组里，跟类名看上去的行为完全一致。

当MapOutput拷贝方法shuffle返回时，Fetcher会调用Scheduler的copySucceed方法做一些收尾工作，比如将已经拷贝过的host从待拷贝列表中删除。比较重要的一点是，它会调用Mapoutput的commit方法。两种Mapoutput的实现在这里的差异不大，都会调用MergeManagerImpl的closeXXXXFile方法。

MapOutput负责的是将数据从集群中得其他机器上拉取过来，拉取到的数据怎么Merge到一起，就是MergeManagerImpl考虑的事情了。

MergeManagerImpl

MergeManagerImpl几乎handle了所有与merge相关的实现。他有两个（其实是三个）内部类，InMemeryMerger和OnDiskMerger，分别对应了前面的两种不同的MapOutput。

我们先看一下这两个Merger共同的父类MergeThread，比较容易理解它做得事情。

```
abstract class MergeThread<T,K,V> extends Thread {

 private LinkedList<List<T>> pendingToBeMerged;
 public synchronized void close() throws InterruptedException {
  closed = true;
  waitForMerge();
  interrupt();
 }

 public void startMerge(Set<T> inputs) {
  if (!closed) {
   numPending.incrementAndGet();
   List<T> toMergeInputs = new ArrayList<T>();
   Iterator<T> iter=inputs.iterator();
   for (int ctr = 0; iter.hasNext() && ctr < mergeFactor; ++ctr) {
    toMergeInputs.add(iter.next());
    iter.remove();
   }
   LOG.info(getName() + ": Starting merge with " + toMergeInputs.size() +
           " segments, while ignoring " + inputs.size() + " segments");
   synchronized(pendingToBeMerged) {
    pendingToBeMerged.addLast(toMergeInputs);
    pendingToBeMerged.notifyAll();
   }
  }
 }

 public synchronized void waitForMerge() throws InterruptedException {
  while (numPending.get() > 0) {
   wait();
  }
 }
```

```
    public void run() {
      while (true) {
        List<T> inputs = null;
        try {
          // Wait for notification to start the merge...
          synchronized (pendingToBeMerged) {
            while(pendingToBeMerged.size() <= 0) {
              pendingToBeMerged.wait();
            }
            // Pickup the inputs to merge.
            inputs = pendingToBeMerged.removeFirst();
          }

          // Merge
          merge(inputs);
        } catch (InterruptedException ie) {
          numPending.set(0);
          return;
        } catch(Throwable t) {
          numPending.set(0);
          reporter.reportException(t);
          return;
        } finally {
          synchronized (this) {
            numPending.decrementAndGet();
            notifyAll();
          }
        }
      }
    }

    public abstract void merge(List<T> inputs) throws IOException;
  }
```

容易看出，MergeThread有一个LinkedList，用于存放MapOutput得到的输出，startMerge方法会将这些输出添加到List中，run方法会不断的从中取出Mapoutput的输出，并调用merge方法，Close的时候会等待所有的merge过程结束。startMerge方法正是在MergeManagerImpl的closeXXXXMergedFile调用的。

这样整个过程就清晰一些了，Shuffle时调用Fetcher来下载Map的输出，Fetcher根据数据量的大小，判断是实例化InMemoryMapOutput还是OnDiskMapOutput，实例化出的MapOutput拉取完毕后，Fetcher通过一个Shuffle的scheduler调用Mapoutput的commit方法，commit方法中调用到MergeManagerImpl的closeXXXXMergedFile方法，这个方法又调用到MergeThread实现类中得startMerge方法，下载到得数据最终就被传递到了MergeThread的实现类了。

剩下的问题，就是怎么Merge了。

Merge

整个Merge的过程比较复杂，牵扯到得代码也比较多，我按照我的理解，在逻辑的层面简单叙述一下这个过程。

从整体上讲，Merge的过程是先Merge掉InMemory的，InMemory的结果也会加入到onDisk的待Merge队列中，最后补上一记finalMerge，合并起InMemory剩余的与onDisk剩余的。每种Merger的Merge操作最终都是交给一个叫Merger的工具类的静态方法实现的。

除了参数的不同，实际merge的过程是类似的。Merge就是将小文件合并成大文件，对于初始有序的数据，为了减少比较次数，每次应该合并数据最少的两组，也就是霍夫曼树的思想。从源码看，貌似是自己用ArrayList实现了一个。

InMemory的Merge行为是，先将InMemoryMapOutput中的buffer结构化成一组segment，segment含有需要merge的数据，最重要的，它含有这些数据的长度信息，这个信息会再霍夫曼树式的merge用到。接下来它会new出一个path对象用来存放merge的结果，一个Writer来写入，然后就会调用Merger工具类的相应merge方法进行实际的merge。在实际写入文件的时候，会判断有没有指定Combiner，也就是会不会对相同key的输出进行进一步的合并。InMemoryMerger的最终结果会写入到文件，并将这个文件的信息注册到onDiskMerger中，以便后续的合并。

onDiskMerger的行为与InMemoryMerger的行为基本一致，只是在调用Merger的时候给定了不同的参数。

finalMerge的行为是，先判断有没有inMemory的output，有的话构造出segment，合并，最终的结果是一个文件，添加到onDisk得output队里中，然后合并onDisk的ouput，比较特别的，finalMerge是有返回值的，最终合并的结果输出是RawKeyValueIterator类型，代表这一个reduce所接收到的所有输入。

MergeManagerImpl的close方法

在shuffle的run方法中，在copyPhase结束之后，调用了MergeManagerImpl的close方法，该方法的实现如下：

```
public RawKeyValueIterator close() throws Throwable {
  // Wait for on-going merges to complete
  if (memToMemMerger != null) {
    memToMemMerger.close();
  }
  inMemoryMerger.close();
  onDiskMerger.close();

  List<InMemoryMapOutput<K, V>> memory =
    new ArrayList<InMemoryMapOutput<K, V>>(inMemoryMergedMapOutputs);
  inMemoryMergedMapOutputs.clear();
  memory.addAll(inMemoryMapOutputs);
  inMemoryMapOutputs.clear();
  List<CompressAwarePath> disk = new ArrayList<CompressAwarePath>(onDiskMapOutputs);
  onDiskMapOutputs.clear();
  return finalMerge(jobConf, rfs, memory, disk);
 }
```

可见，在inMemoryMerger和onDiskMerger的close之后，会最终返回finalMerge的结果。这个RawKeyValueIterator会最为一个参数传递给reduce过程。Shuffle过程到此就算彻底结束了。

Reduce的输入

Shuffle过程结束之后，reduce阶段获得了RawKeyValueIterator类型的输入，ReduceTask的run方法会调用runNewReducer方法，该方法的签名如下：

```
private <INKEY,INVALUE,OUTKEY,OUTVALUE>
  void runNewReducer(JobConf job,
                     final TaskUmbilicalProtocol umbilical,
                     final TaskReporter reporter,
                     RawKeyValueIterator rIter,
                     RawComparator<INKEY> comparator,
                     Class<INKEY> keyClass,
```

```
                Class<INVALUE>valueClass
                ) throws IOException,InterruptedException,
                        ClassNotFoundException;
```

在这里面，会用反射的方式实例化出Reducer。

Reducer与Mapper非常相似，我贴一下他的实现：

```
public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {

 /**
  * The <code>Context</code> passed on to the {@link Reducer} implementations.
  */
 public abstract class Context
   implements ReduceContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
 }

 /**
  * Called once at the start of the task.
  */
 protected void setup(Context context
                      ) throws IOException, InterruptedException {
   // NOTHING
 }

 /**
  * This method is called once for each key. Most applications will define
  * their reduce class by overriding this method. The default implementation
  * is an identity function.
  */
 @SuppressWarnings("unchecked")
 protected void reduce(KEYIN key, Iterable<VALUEIN> values,Context context
                       ) throws IOException, InterruptedException {
   for(VALUEIN value: values) {
     context.write((KEYOUT) key, (VALUEOUT) value);
   }
 }

 /**
```

```
   * Called once at the end of the task.
   */
  protected void cleanup(Context context
                         ) throws IOException, InterruptedException {
    // NOTHING
  }

  /**
   * Advanced application writers can use the
   * {@link #run(org.apache.hadoop.mapreduce.Reducer.Context)} method to
   * control how the reduce task works.
   */
  public void run(Context context) throws IOException, InterruptedException {
    setup(context);
    try {
      while (context.nextKey()) {
        reduce(context.getCurrentKey(), context.getValues(), context);
        // If a back up store is used, reset it
        Iterator<VALUEIN> iter = context.getValues().iterator();
        if(iter instanceof ReduceContext.ValueIterator) {
          ((ReduceContext.ValueIterator<VALUEIN>)iter).resetBackupStore();
        }
      }
    } finally {
      cleanup(context);
    }
  }
}
```

Reducer的输入输出的信息同样是封装在Context中。Reducer与Mapper看上去很像，但是有很多关键的不同。比如reduce方法的参数，还有run方法的实现。

注意到，reduce方法的第二个参数，不再是一个VALUE类型，而是一个迭代器，意指key相同的value会一次性的扔给这个reduce方法，那么到底怎样算key相同呢？我们看到reduce方法是在run方法中调用的，第一个参数与Mapper相同，也是context的currentKey，第二个不一样，是从context获得的values，在ReduceContextImpl中，这个getValues方法，直接返回一个迭代器。

从语义上说，Reducer的reduce方法应该每次处理Key相同的那一组输入，那么到底什么样的一组key，算是相同的key呢？这个关键的问题由构造ReduceContext的一个不起眼的参数，GroupingComparator来解决。

GroupingComparator

我们知道，哪些Mapper的输出交给一个Reduce线程是由Partitioner决定的，但是这些输入并不是一次性处理的，举个例子，我们在做大小写敏感word count的时候，假设使用的partition策略是根据单词首字母大小来指定reducer，有2个reducer的话，"an"和"car"都会交给同一个reduce线程，但是统计每个单词个数的时候，他俩是不能混起来的，也就是**一个reduce线程实际上将整个输入分成了好多组，在每一组上运行了一次reduce的过程**。这个组怎么分，就是GroupingComparator做得事情。针对word count这个实例，我们应该将完全相等的两个单词作为一组，运行一次reduce的方法。

我们看一下GroupingComparator接口的定义：

```
public interface RawComparator<T> extends Comparator<T> {

  /**
   * Compare two objects in binary.
   * b1[s1:l1] is the first object, and b2[s2:l2] is the second object.
   *
   * @param b1 The first byte array.
   * @param s1 The position index in b1. The object under comparison's starting index.
   * @param l1 The length of the object in b1.
   * @param b2 The second byte array.
   * @param s2 The position index in b2. The object under comparison's starting index.
   * @param l2 The length of the object under comparison in b2.
   * @return An integer result of the comparison.
   */
  public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);

}
```

可见，它是从字节码的角度来判定是否相等的，具体比较哪一部分，可以根据我们自己的实现来控制。

经过了Shuffle过程，所有的输入都已经按照Key排序好了。所以，在判定两个Key要不要交给同一个Reduce方法时，只要判定相邻的两个key就可以了。这个比较的过程，实际上在我们在reduce方法中，对value的迭代器不断的取next的时候，就悄悄发生了。

业务场景

接着前面的业务场景，还是event和session的问题，我已经讲过一段时间内同一个guid的event应该划分成一个session，也就是event的key是guid和timestamp时，guid一样的event要按照timestamp排好序的顺序交给一个reduce方法来处理，因此，我们自己实现的GroupingComparator应该只比较event key的guid，按照guid来聚合。

Reduce的输出

在构造ReduceContext的时候，传入了两个跟输出相关的参数，一个是RecordWriter类型，一个是OutputCommitter类型。但是，当查看这两个对象构造的过程时，会发现他们的幕后boss居然是OutputFormat！这货看起来是不是非常眼熟？没错，我们在之前讲Map的输出时提到过一次，没有展开讲。它跟InputFormat的功能其实很呼应。

OutputFormat

按照惯例，我们还是来看看它的接口。

```
public abstract class OutputFormat<K, V> {

  /**
   * Get the {@link RecordWriter} for the given task.
   *
   * @param context the information about the current task.
   * @return a {@link RecordWriter} to write the output for the job.
   * @throws IOException
   */
  public abstract RecordWriter<K, V>
    getRecordWriter(TaskAttemptContext context
                    ) throws IOException, InterruptedException;

  /**
   * Check for validity of the output-specification for the job.
   *
   * <p>This is to validate the output specification for the job when it is
   * a job is submitted.  Typically checks that it does not already exist,
```

```
     * throwing an exception when it already exists, so that output is not
     * overwritten.</p>
     *
     * @param context information about the job
     * @throws IOException when output should not be attempted
     */
    public abstract void checkOutputSpecs(JobContext context
                                          ) throws IOException,
                                               InterruptedException;

    /**
     * Get the output committer for this output format. This is responsible
     * for ensuring the output is committed correctly.
     * @param context the task context
     * @return an output committer
     * @throws IOException
     * @throws InterruptedException
     */
    public abstract
    OutputCommitter getOutputCommitter(TaskAttemptContext context
                                       ) throws IOException, InterruptedException;
}
```

从源码的描述来看，OutputFormat主要做两件事情，一是验证job的输出配置是否合理（比如查看目标路径是否存在），二是提供一个RecordWriter的实现，来写入最终的输出。三个抽象方法，分别用于返回RecordWriter，返回OutputCommitter，以及验证输出的配置。

你可能会想不通一个输出为什么要搞这么复杂，反正一个reducer产生一个文件，指定一下目录，直接往里写不就行了吗？怎么还要recordWriter，还要committer的。我们回想一下hadoop设计的初衷，在不可靠的机器上，得到可靠的输出。也就是，hadoop的设计者认为一个task它很可能是会运行失败的，我们常常需要尝试多次，因此，除了写入操作之外，我们应该先写在临时目录，确定成功了，再提交到正式的输出目录里，这个工作其实就是committer做得，而recordWriter只要专注于写入操作就可以了。

我们当然可以从头开始写一个OutputFormat，但更一般的做法是，继承自一个典型的实现FileOutputFormat。

FileOutputFormat

下面是它对checkOutputSpecs的实现：

```
public void checkOutputSpecs(FileSystem ignored, JobConf job)
  throws FileAlreadyExistsException,
       InvalidJobConfException, IOException {
  // Ensure that the output directory is set and not already there
  Path outDir = getOutputPath(job);
  if (outDir == null && job.getNumReduceTasks() != 0) {
   throw new InvalidJobConfException("Output directory not set in JobConf.");
  }
  if (outDir != null) {
   FileSystem fs = outDir.getFileSystem(job);
   // normalize the output directory
   outDir = fs.makeQualified(outDir);
   setOutputPath(job, outDir);

   // get delegation token for the outDir's file system
   TokenCache.obtainTokensForNamenodes(job.getCredentials(),
                       new Path[] {outDir}, job);

   // check its existence
   if (fs.exists(outDir)) {
    throw new FileAlreadyExistsException("Output directory " + outDir +
                       " already exists");
   }
  }
 }
```

在最开始接触hadoop跑测试的时候，经常遇到FileAlreadyExistsException这个错误，原因就是没有删掉上一次跑的结果。直到现在，才知道原来是从这里抛出的啊。hadoop之所以这样设定，是为了防止因为粗心覆盖掉之前生成的数据，我觉得这是合理的。

FileOutputFormat还提供了一些好用的方法，比如下面这个：

```
public synchronized static String getUniqueFile(TaskAttemptContext context,
                       String name,
                       String extension) {
  TaskID taskId = context.getTaskAttemptID().getTaskID();
  int partition = taskId.getId();
```

```
    StringBuilder result = new StringBuilder();
    result.append(name);
    result.append('-');
    result.append(
        TaskID.getRepresentingCharacter(taskId.getTaskType()));
    result.append('-');
    result.append(NUMBER_FORMAT.format(partition));
    result.append(extension);
    return result.toString();
  }
```

这个方法提供了一种生成唯一输出文件名的功能，之所以需要这个，是因为多个task都输出，万一起名冲突就坏了。命名的规则是以用户提供的字符串开头，然后是task的类型（map-m， reduce-r），最后是这个task所属的partition。这种方式保证了在当前job生成的结果中文件名是唯一的。

标示出task得类型和partition有很大的好处，我们在实际工作中就有过这种体会。我们有一个每小时都会运行的job，跑完一个小时的数据需要15分钟的样子，但是每天0点都会跑的特别慢，也不报错，通过查看生成文件，我们发现标示为r的partition号为3的那个task总是最后生成文件，而且比其他partition的都要明显大，最终确定了是交给这个partition的数据太多造成的。

对于一开始用户提供的前缀，当然可以是任何形式，但是我们强烈建议缀上时间戳。在我们的实践中，当前小时生成的数据有可能需要拷贝回前面的文件夹，默认提供的命名方式只能保证在当前job生成的输出文件是唯一的，没法保证与之前的不冲突，我们的做法是在前缀上加时间戳，这样可以方便的分辨哪些是后来加入的文件。

OutputCommitter

从源码的注释，我们知道OutputCommitter负责下面的工作：

- 在job启动时setup job。例如，在job的启动期间建立临时的输出目录。
- 在job结束是clean up job。比如，job结束之后删掉临时输出目录。
- 建立task的临时输出
- 检测一个task需不需要提交自己的输出
- 提交task的输出
- 丢弃task的输出

这么列出来，感觉比较空洞，我讲一下我的理解。正如前文提到的，OutputCommitter的主要职责是建立起task执行的临时目录，然后验证这个task需不需要将自己的输出的结果提交，提交到哪里。对于产生的临时目录和写入的临时文件，也要负责清理干净。

OutputCommitter有很多需要实现的方法，我列一下：

```
public abstract class OutputCommitter {
  public abstract void setupJob(JobContext jobContext) throws IOException;
public void cleanupJob(JobContext jobContext) throws IOException { }
  public void commitJob(JobContext jobContext) throws IOException {
  cleanupJob(jobContext);
 }
  public void abortJob(JobContext jobContext, JobStatus.State state)
 throws IOException {
  cleanupJob(jobContext);
  }
  public abstract void setupTask(TaskAttemptContext taskContext)
 throws IOException;
  public abstract boolean needsTaskCommit(TaskAttemptContext taskContext)
 throws IOException;
  public abstract void commitTask(TaskAttemptContext taskContext)
 throws IOException;
  public abstract void abortTask(TaskAttemptContext taskContext)
 throws IOException;
  public boolean isRecoverySupported() {
    return false;
  }
  public void recoverTask(TaskAttemptContext taskContext)
 throwsIOException
 {}
```

方法名比较准确的反应方法需要实现的功能。下面我们看一下与FileOutputFormat对应的Committer。

FileOutputCommitter

前面已经提到了，OutputCommitter最重要的就是目录的建立删除以及拷贝，那么要理解一个Committer的行为，只要专注它是怎么操作目录的就可以了。

在FileOutputCommitter里，有三个四种目录，四种目录分别包括

- 最终的Job输出目录
- 临时的Job目录
- task的提交目录
- task的临时目录

task每次在task临时目录中工作，如果确定成功并且需要被提交，就会提交到task的提交目录中。task的提交目录实际上跟临时Job的目录是一个目录，当一个job的所有task都顺利执行之后，会从临时job目录提交到最终的输出目录。

之所以有这么多跳，其实还是基于task很可能会执行失败的假设，这种方式，在task失败的时候，可以直接清掉它的目录重来，效率上肯定要差一些。因此我的同事写过一个DirectFileOutputCommitter，当task执行成功时，直接提交到最终的工作目录。这种方式虽然在一定程度上提高了效率，可有个风险，当这个job失败需要重新执行的时候，就得事先清一下最终的输出目录。

在实践的时候，我们常常通过在一个目录下生成"_SUCCESS"文件来标记这个目录已经完成，一个很好的生成时机就是Committer的commitJob方法。

RecordWriter

这个类的介绍非常普通，它做的事情也很简单，就是将一对KeyValue的pair写入到输出文件中。他的接口很简单：

```
public abstract class RecordWriter<K, V> {
  /**
   * Writes a key/value pair.
   *
   * @param key the key to write.
   * @param value the value to write.
   * @throws IOException
   */
  public abstract void write(K key,V value
                             ) throws IOException, InterruptedException;

  /**
   * Close this <code>RecordWriter</code> to future operations.
   *
```

```
   * @param context the context of the task
   * @throws IOException
   */
  public abstract void close(TaskAttemptContext context
                         ) throws IOException, InterruptedException;
}
```

write方法用来写入，close方法用来释放资源。

FileOutputFormat中没有提供getRecordWriter的实现，我们来看一下实际工作中用的比较多的FileOutputFormat最有名的子类，TextOutputFormat中是怎么实现的。

```
public RecordWriter<K, V> getRecordWriter(FileSystem ignored,
                                          JobConf job,
                                          String name,
                                          Progressable progress)
    throws IOException {
    boolean isCompressed = getCompressOutput(job);
    String keyValueSeparator = job.get("mapreduce.output.textoutputformat.separator",
                                       "\t");
    if (!isCompressed) {
      Path file = FileOutputFormat.getTaskOutputPath(job, name);
      FileSystem fs = file.getFileSystem(job);
      FSDataOutputStream fileOut = fs.create(file, progress);
      return new LineRecordWriter<K, V>(fileOut, keyValueSeparator);
    } else {
      Class<? extends CompressionCodec> codecClass =
        getOutputCompressorClass(job, GzipCodec.class);
      // create the named codec
      CompressionCodec codec = ReflectionUtils.newInstance(codecClass, job);
      // build the filename including the extension
      Path file =
        FileOutputFormat.getTaskOutputPath(job,
                                           name + codec.getDefaultExtension());
      FileSystem fs = file.getFileSystem(job);
      FSDataOutputStream fileOut = fs.create(file, progress);
      return new LineRecordWriter<K, V>(new DataOutputStream
                                        (codec.createOutputStream(fileOut)),
```

```
                keyValueSeparator);
        }
    }
```

代码很好懂，key和value之间默认的分割符是"\t"，输出流用得是FSDataOutputStream，可压缩，可不压缩。LineRecordWriter是它的内部类，每次新起一行写入新数据，key value之间用分隔符分割。

至此，我就将MapReduce的过程全部讲完了，中间还有没说清楚的地方，后面我们继续晚上，再自己的经验积累一些后，应该会有更深刻的理解。

结语

终于将这篇博客写完了，本来想着是入职之前完成的，结果拖了快两个月。最初开始的时候也没想过会写这么多，看起源码来，一环一环的，就总想再搞得明白一些。里面掺杂了一些我工作中遇到的问题，不是很多，不过我觉得还是有一定的参考意义的。希望这篇博客能帮助到对hadoop感兴趣的你。

【参考文献】

1. http://www.ibm.com/developerworks/cn/opensource/os-cn-hadoop-yarn/
2. http://dongxicheng.org/mapreduce-nextgen/hadoop-1-and-2-resource-manage/
3. http://dongxicheng.org/mapreduce-nextgen/understand-yarn-container-concept/
4. http://www.it165.net/pro/html/201402/9903.html