

报告题目： Hadoop MapReduce源码分析报告

课程名称： 大规模软件系统构建技术导论

学生姓名：叶金韬

专 业： 软件工程

学 号： 3150102210

目录

**一，框架介绍3**

**二，分析模块3**

**三，从简单样例开始3**

**四，Mapper及其相关类4**

**五，Reducer及其相关类9**

**六，Configuration类10**

**七，Context具体实现14**

**7.1. MapContextImpl14**

**7.2. ReduceContextImpl16**

**八，关于Job 以及 Map & Reduce 调用流程19**

**8.1. MapTask20**

**8.2. ReduceTask24**

**九，总结29**

**参考和附录30**

Hadoop MapReduce源码分析报告

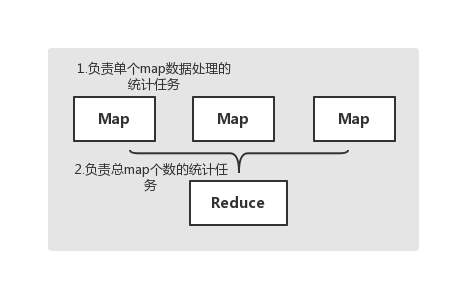
3150102210 叶金韬

1. **框架介绍**

Hadoop是一款可靠的用于大规模并行运算的开源软件。它可以用大规模的计算机集群来并行处理大规模的数据，但只需要我们使用简单的编程模型即可。它可以自己检测并处理运算过程中的错误，不需要依赖于硬件的高度可靠性，因此运算的结果可靠性很高。

1. **分析模块**

Hadoop Project最新的稳定版本已经到了2.8.0，也是本次准备分析的版本。Hadoop结果一系列的发展，以及变得非常的庞大。它主要由四个模块组成：Hadoop Common（支持其他模块运行的工具集）、HDFS（一种分布式文件系统）、Hadoop YARN（一种任务调度和集群资源管理的框架）以及Hadoop MapReduce（基于YARN的并行计算模块）。本次分析的模块是MapReduce模块，分析的包主要涉及2个：org.apache.hadoop.mapreduce（新）和org.apache.hadoop.mapred（旧）。



图一，Map-Reduce整体模型

1. **从简单样例开始**

应用MapReduce最简单的一个应用就是统计词频。在提供的WordCount.java就是一个例子。该类中有两个静态类TokenizerMapper（继承泛型类Mapper）和IntSumReduce（继承泛型类Reducer），分别处理Map和Reduce两个过程。

对于该样例，Map过程中输入的是<Object, Text>的键值对（Text为全部文本），输出的是<Text, IntWritable>结果键值对，即单词-次数键值对。对应未进行统计的结果，不论单词相不相同，都给其词频赋值“1”。即最初的结果集是键可重复集，值都为1。

|  |
| --- |
| //在对应覆盖的map函数中对于最初结果的处理  …  while (itr.hasMoreTokens()) {  word.set(itr.nextToken());  context.write(word, one);  }… |

Reduce过程即为规约过程，在该样例中，覆盖的reduce方法将相同的单词规约成一个键值对（作为结果），其值随着词频数增加而增加。下述的context就是最后的结果（对于某一个词）0。汇总所有Reduce结果将得到最后我们所希望的词频结果。

|  |
| --- |
| …  for (IntWritable val : values) {  sum += val.get();  }…  context.write(key, result); |

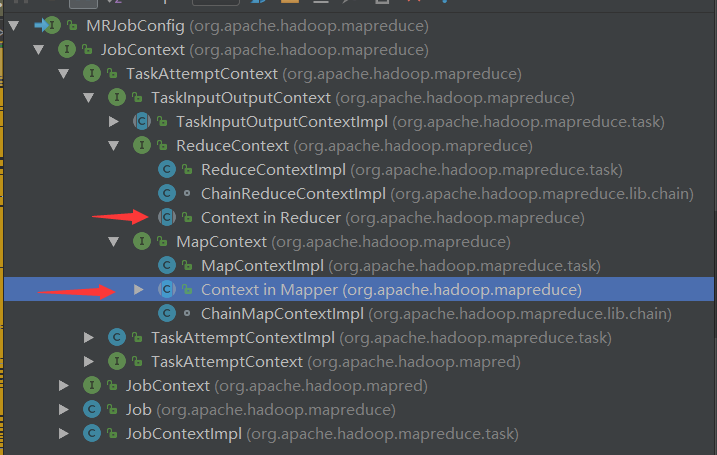
最后回到主方法中，开始创建默认的Configuration的实例，它可以来访问Configuration源的一些运行参数。该实例的默认配置源来自于两个配置文件core-default.xml（只读）和core-site.xml（site-specified）。然后定义一个Job，设定程序的类目，实现Map的类，实现Reduce的类，定义输入输出的类型的文件。最重要的是最后一步，提交并检测结果，也就是Job类的waitForCompletion()方法。完成后退出程序。

以上就是WordCount基本运行流程。其中涉及了一些陌生的类和方法（如Mapper, Reducer, Configuration, Job以及相关的），将在下文进行具体分析。

1. **Mapper以及相关类**

Mapper<KEYIN, VALURIN, KEYOUT, VALUEOUT>实际上是一个非常简单的类，因而我们常继承该类来自定义自己的Mapper。

Mapper里面有一个Context类，它继承MapContext接口，由第三节例子的源码可以推测，Context是一个信息载体，可以存储结果信息，也可以存储一些Job Configure信息，是Mapper和Reducer的桥梁。Mapper把数据写入Context，然后传入Reducer，Reducer把结果写入Context中返回给主程序。Context的数据还可以写到HDFS系统中。Context类的继承树如下，实际上，Mapper和Reducer类中的Context类没有重写继承而来的方法（对于TaskInputOutputContext接口来说），所以它们的继承是一致的。



图二，Context类继承树

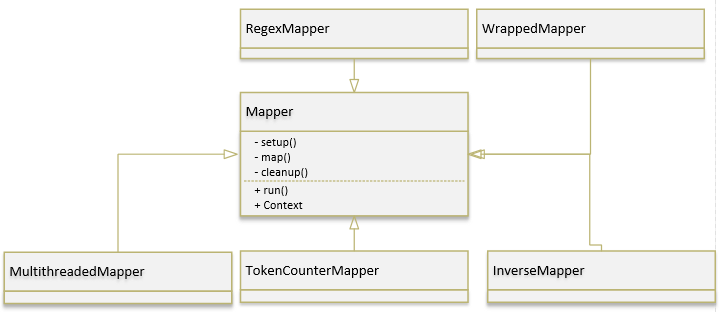
而Context的具体实现在ReduceContextImpl（对应Reducer）和MapContextImpl（对应Mapper）类中。

除了Context类之外，Mapper类还实现或者要求我们实现下列几个方法：

1. **setup(Context context)**：在一个任务开始的时候被调用一次，由于系统传入了Context参数，我们可以通过覆盖/重载该方法来获得Configuration等一些变量；
2. **map(KEYIN, VALUEIN, Context)**：关键的map函数，当输入分割后每个键值对都会调用该方法一次。默认的map仅是将传入的键值对拷贝到Context中（这里的Context代表KEYOUT和VALUEOUT），所以必然要重新覆盖该方法来实现Map的功能；
3. **cleanup(Context context)**: 当一个任务要结束时最后执行一次，用于资源的释放和清理。Hadoop并没有将清理放在map中是因为每次处理一次键值对就要释放资源是重复的，效率不高。
4. **run(Context context)**: 执行Map的全部流程，首先是setup，然后对已经分割后的数据（即参数中的context）进行map，当所有的map结束之后，最后进行清理工作。若map过程中存在异常，则直接进行清理，不会有其它问题。在默认的run方法中，每次分割的map是以线性的方式进行，时间复杂度为O(n)，因此为了提高效率，可考虑重写覆盖该方法，事实上Mapper的一些子类也将该方法override了。下面是默认的run关键代码。

|  |
| --- |
| setup(context);//First SETUP try {  while (context.nextKeyValue()) {//Next MAP  map(context.getCurrentKey(), context.getCurrentValue(), context);  } } finally {  cleanup(context);//Finally CLEAN-UP } |

由上述可知，默认的Mapper必然满足不了我们对并行运算的需求，因而在Hadoop框架内为我们实现了一些常用的Mapper，它们都继承与默认的Mapper类。常用的Mapper在包org.apache.hadoop.mapreduce.lib.map以及其它的包中中有提供，具体如下（仅仅挑了五个，实际上有许许多多的个性化Mapper）：



图三，Hadoop提供的一些Mapper（不全）

下面对于某些常用的Mapper进行分析：

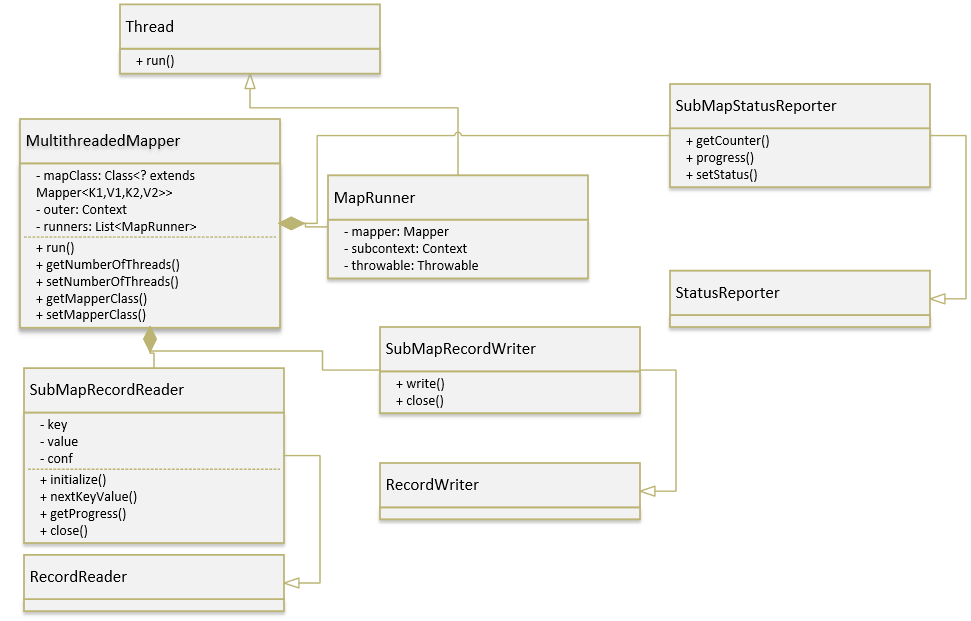
1. **TokenCounterMapper**。该Mapper仅重写了map函数，它将文本按词分割，然后一个一个进行键值对的创建，每个词对应的值为1，且键值对的键集可重复（也即词的一个列表）。显而易见，第三小节的WordCount例程可以直接使用TokenCounterMapper，只需使用代码job.setMapperClass(TokenCounterMapper.class)设置即可，无需自定义Mapper；

|  |
| --- |
| //KEY CODES  …  StringTokenizer itr = new StringTokenizer(value.toString());  while (itr.hasMoreTokens()) {  word.set(itr.nextToken());  context.write(word, one);  }  … |

1. **InverseMapper**。这个Mapper更加简单，它仅仅调换了传入的Key和Value（即原来的Key变为Value，而原来的Value变为了Key）；
2. **WrappedMapper**。这个Mapper并没有对基类的Mapper的四个方法进行修改，而是自定义了Context（继承于Mapper里的Context）。这个Context也是只是仅仅简单实现了上层的接口，并没有对信息进行修改。因而这个类仅仅对Mapper进行包装以允许Context的自定义而已；
3. **RegexMapper**。顾名思义，该类对正则表达式进行匹配分割。该类重写了两个方法（setup和map），并增加成员变量pattern和group用于匹配字符串（来分割数据集）。该类在map函数中对传入的字符串利用pattern进行匹配，然后将匹配的项生成<Matcher, 1>的键值对集；

|  |
| --- |
| //KEY CODES  …  String text = value.toString();  Matcher matcher = pattern.matcher(text);  while (matcher.find()) {  context.write(new Text(matcher.group(group)), new LongWritable(1));  }  … |

1. **MultithreadedMapper**。这个类有点复杂，它用多线程执行一个Mapper。关于该类的UML图如下：



图四，MultithreadedMapper类图

可以从类图中看到，它有一个成员叫mapClass，它指定了另外一个Mapper类（由mapred.map.multitreadedrunner.class设置），实际上运行的是指定的那个Mapper类而非这个MultithreadedMapper。

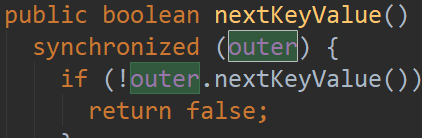
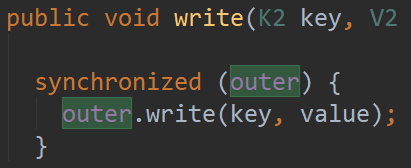
在该类重写的run方法中，首先设置context和指定工作的Mapper：

|  |
| --- |
| outer = context;  int numberOfThreads = getNumberOfThreads(context);  mapClass = getMapperClass(context);//SET CONTEXT & MAPPER |

然后启动多个MapRunner子线程来执行Map操作。MapRunner由上图可知它继承了Thread类，包含独享的Context（即subcontext）和指定的Mapper（即mapper），因此可以利用多线程来调用所需的Mapper进行Map操作（Throwable在run()中进行异常处理）。最后使用join()等待子线程执行完毕，得到map后的结果。

|  |
| --- |
| runners = new ArrayList<MapRunner>(numberOfThreads);  for(int i=0; i < numberOfThreads; ++i) {  MapRunner thread = new MapRunner(context);  thread.start();  runners.add(i, thread);//Start MAP-RUNNER  }  for(int i=0; i < numberOfThreads; ++i) {  MapRunner thread = runners.get(i);  thread.join();//Finally Join  …  } |

而在MapRunnder的构造函数中发现，它使用了独立的SubMapRecordReader, SubMapRecordWriter和SubMapStatusReporter，并且Reader和Writer在操作键值对的时候要进行同步（即是线程安全的），这是通过互斥访问MultithreadedMapper的Context实例outer实现的（访问的outer会被synchronized，也即被加锁了）。而由于对读写操作是线程安全的，所以当进行I/O密集型作业时，性能可能不会得到提高（因为有很多线程等待一个线程的I/O），反而有可能降低性能。



图五，对outer的互斥处理

除了这些Mapper之外，还有如ChainMapper, DelegatingMapper, IdentityMapper等其他的Mapper，有着不同的功能和用途，这里就不再细述。

1. **Reducer以及相关类**

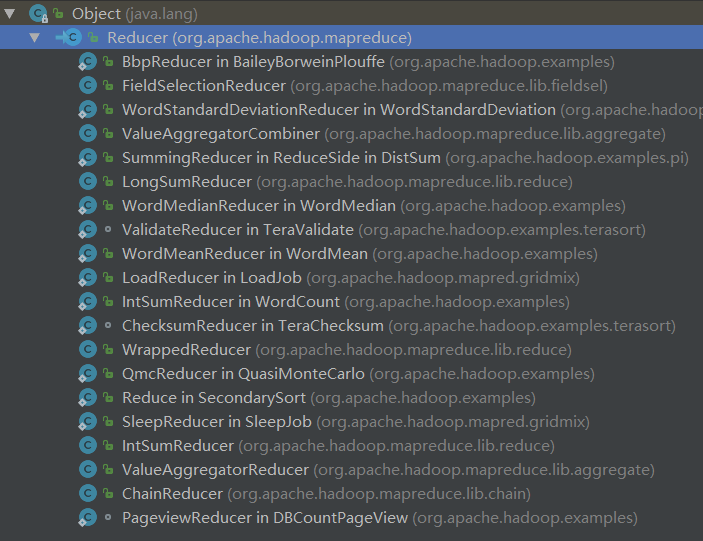
Reducer是MapReduce计算模型里的有一个十分重要的模块，它将一组共享一个键的中间值规约成一小组值。在Hadoop提供的代码里，默认的Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>也是一个非常简单的类，因而我们也常常定义自己的Reducer。

和Mapper相似，默认的Reducer类也有内部的Context类（继承ReduceContext，然而并没有更改继承而来的方法，因而本质上和Mapper里的Context一样），其功能也在上文中有所描述。

同样，Reducer类里也实现或者要求我们实现下列几个方法：

1. **setup(Context context)**：和Mapper中的setup类似，在一个Task开始的时候调用一次，来进行一些初始化的工作，或者进行获取Configuration参数等等。
2. **reduce(KEYIN, Iterable<VALUEIN>, Context)**：这是Reducer的业务逻辑的核心（同比与Mapper里的map函数），参数传入一个键和与该键相关的一组值（即Iterable参数）。然后通过一些规约的方法，最后将结果写入Context里（对于KEYOUT, VALUEOUT），该组键值对就处理完毕了。默认的Reducer对于reduce的操作是很简单的，也仅仅是把传来的参数重新写回去而已，没有进行任何数据上的修改，因此也是需要对该函数进行重写和覆盖来满足我们的需求。
3. **cleanup(Context context)**：和Mapper里的cleanup一样，当reduce工作完成或者在一个Task运行完成的时候最后调用一次，进行资源的释放和清理。
4. **run(Context context)**：基本同Mapper里的run方法，其运行的方法也是比较简单，对传入的Context（一般是Mapper处理好然后给Reducer的），线性遍历所有的key，然后得到所有相关的value得到一个Iterable实例，然后进行reduce操作，最后清理。

同Mapper，Hadoop也为我们提供了一些已经个性化好的Reducer子类，以方便我们直接调用。被实现的Reducer子类的数量比Mapper的要少许多，但是也有一些比较重要的，比如FieldSelectionReducer, ValueAggregatorCombiner, LongSumReducer, WrappedReducer（对应前面的WrappedMapper）, IntSumReducer（对应TokenCountMapper）, ValueAggregatorReducer, ChainReducer（用于流水链式的ChainMapper的处理）等等。注意到这里没有用多线程的Reducer，可能开发者认为这没有必要。



图六，Hadoop中实现的Reducer继承树

1. **Configuration类**

第四以及第五小节分析了Mapper和Reducer的基本算法和某些子类的具体实现。但是如何将Mapper和Reducer协同起来工作，真正实现对一个大数据集的Map & Reduce是上文没有涉及的。这里先回到第三小节的样例中，在主程序的第一行创建了Configuration实例conf，这一节来分析Hadoop的Configuration。

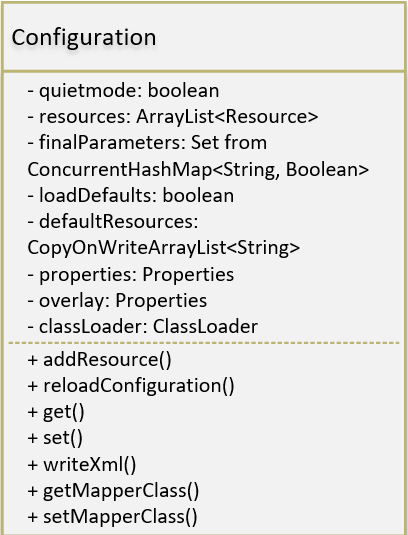
Hadoop中的Configuration并没有使用Java库的util.Properties来管理配置文件，而是使用自己独特的配置文件系统并使用自己的API（即Configuration类）来处理配置信息。

根据Configuration代码注释，配置文件采用的是XML格式的文档，默认情况下使用/hadoop-project-dist/hadoop-common/core-default.xml（只读）和core-site.xml，在基本情况下已经够用，当然我们也可以自定义自己的配置文件。默认配置文件根元素为configuration，一般只包含子元素property，每个property元素就是一个配置项，不支持分层和分级。每个配置项有name, value以及描述的description，其中final元素是固定不变不可修改的（可防止配置项值被覆盖）。



图七，Configuration配置文件部分代码

Configuration类主要的成员如下图：



图八，Configuration类主要成员

上面有一些成员变量，其中ArrayList的resources保存了通过addResources()添加的Configuration资源，该方法可以通过5种方式使用，分别是InputStream, Path, ClassPath, URL和现有的Configuration对象（该方法重载了这五个参数）。假如要加载默认的配置文件，则调用addDefaultResource()即可。

然后是properties, overlay, finalParameters。前两个是Java库的Properties类实例，Hadoop的配置文件解析之后会以键值对的形式存在properties中。第二个变量来存储已经改变的配置项，它不是通过解析XML文件得到的。而第三个用于存放final的键值对的键。

然后是classLoader变量，它是一个类加载器，可以加载类资源，也可以加载相关其他的资源，在通过资源名称查找资源的getResource()方法中，使用了classLoader。

|  |
| --- |
| public URL getResource(String name) {  return classLoader.getResource(name);  } |

添加资源可以通过addResource()方法，它将资源添加到Configuration对象中，但不会立即加载，而是将资源加入resources列表中，然后调用reloadConfiguration()函数清空properties和finalParameters。（addResource()方法都会调用addResourceObject()方法）

|  |
| --- |
| public synchronized void reloadConfiguration() {  properties = null; // trigger reload  finalParameters.clear(); // clear site-limits  }    private synchronized void addResourceObject(Resource resource) {  resources.add(resource); // add to resources  reloadConfiguration();  } |

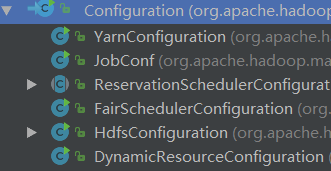
而properties中的资源数据只会在需要的时候被加载，加载的函数为getProps()，当properties为空的时候触发loadResources()加载资源，这种延迟性的加载策略避免了不必要的加载，提高了性能。当overlay不为空的时候将其所有的资源加载到properties以适应配置的变化，降低了运行错误率。该方法是线程安全的，里面的数据会被保护以保证不被其他线程肆意修改。

|  |
| --- |
| protected synchronized Properties getProps() {  if (properties == null) {  properties = new Properties();  Map<String, String[]> backup =  new ConcurrentHashMap<String, String[]>(updatingResource);  loadResources(properties, resources, quietmode);  if (overlay != null) {  properties.putAll(overlay);  for (Map.Entry<Object,Object> item: overlay.entrySet()) {  String key = (String)item.getKey();  String[] source = backup.get(key);  if(source != null) {  updatingResource.put(key, source);  }  }  }  }  return properties; } |

最后就是获取和设置Configuration对象的配置信息。可以用get()方法，通过键的值来获取配置信息，该方法重载了21个参数，参数可以说基本类型，也可以是Hadoop中的常用类型。假如获得的信息不存在则会返回defaultValue。其他的方法也会依赖于这个方法。

设置配置信息，可使用set()方法，它只是简单调用了setProperty()的方法，将传入的键值对传入properties和overlay变量中。

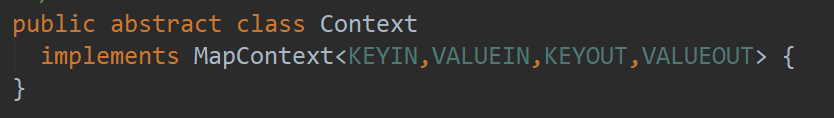
总而言之，Configuration类在实际程序运行和配置文件中提供了一个接口和桥梁，通过它来将XML文件中定义的配置应用到实际程序中。而在Hadoop源代码中，提供了不同的Configuration类，它们都继承该类来满足需求，比如YarnConfiguration, JobConf, FairSchedulerConfiguration等等，在MapReduce提交的任务中，JobConf是最常用的。

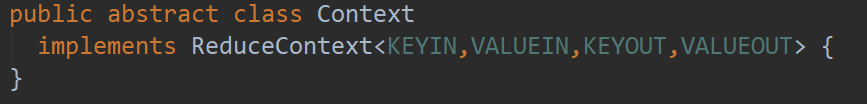


图九，Configuration继承树

1. **Context的具体实现**

在第四、五小节中提及了一个重要的类叫做Context。可是在Mapper和Reducer中，Context却是以一个抽象类表现的，里面没有作任何的实现。



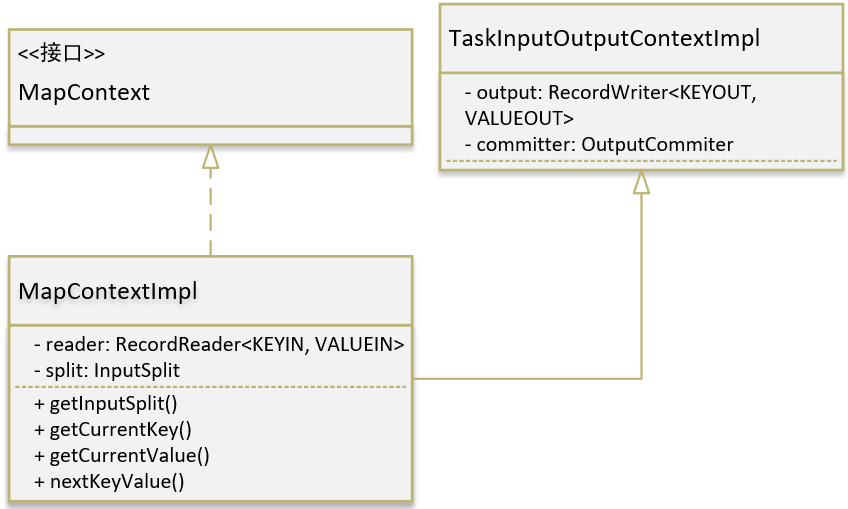


图十，Mapper & Reducer中对Context的定义

不过根据图二的继承树，其实可以发现Context在以下两个类中实现，分别是MapContextImpl和ReduceContextImpl类，对应的实现是MapContext和ReduceContext接口，实际上若无特别自定义，调用的代码在那两个实现的类中。

1. **MapContextImpl**

首先来看关于该类的UML类图：



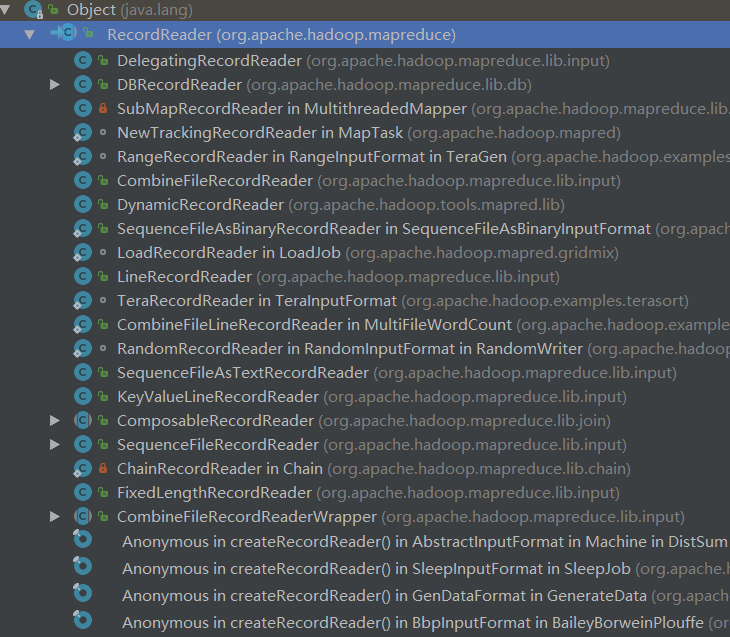
图十一，MapContext UML类图

在构造函数中，其提供了下面的几个参数：Configuration（配置参数）, RecordReader（读取器）, RecordWriter（输出器）, OutputCommitter（输出提交器）, StatudReporter（状态报告器）, InputSplit（数据分割器）。其中第二个和最后一个参数不是继承而来的，其他是继承而来的。

RecordReader<KEYIN, VALUEIN>类用于对键值对的输入，它有下列几个方法：

* initialize()，需要提供InputSplit以及TaskAttemptContext参数，分别代表着记录读取的范围以及关于这个Task的信息，用与读取初始化，只在开头调用一次；
* nextKeyValue()，用于读取下一个键值对；
* getCurrentKey()/getCurrentValue()，返回当前读取的键/值；

RecordReader将InputSplit参数解析并转化成一个个Task键值对。然后这些键值对就可以交给Mapper来进行map处理了。RecordReader有许多实现的类，具体如下：



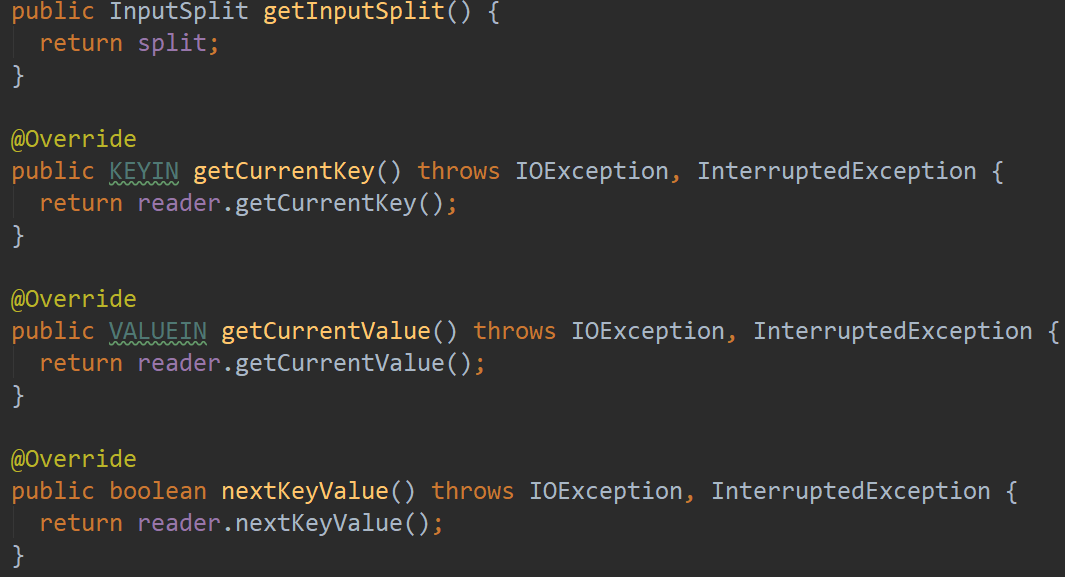
图十二，RecordReader类继承树

以LineRecordReader为例，读取数据的时候都会调用nextKeyValue()方法，该Reader是一行一行读取的，即以‘\n’为分隔符，让后放到value中，而key中的设定就是读取当前文本的位置。由此，文件转换成<K, V>对就实现了。具体见代码：

|  |
| --- |
| …//KEY CODES  if (key == null) {  key = new LongWritable();  }  key.set(pos);  if (value == null) {  value = new Text();//Default value  }  int newSize = 0;  // We always read one extra line, which lies outside the upper  // split limit i.e. (end - 1)  while (getFilePosition() <= end || in.needAdditionalRecordAfterSplit()) {  if (pos == 0) {  newSize = skipUtfByteOrderMark();  } else {  newSize = in.readLine(value, maxLineLength, maxBytesToConsume(pos));  pos += newSize;  }  … |

而LineRecordReader用split（分片）信息获取文件的信息，并准备好进行解析的过程在initialize()函数实现，而且支持压缩文件。

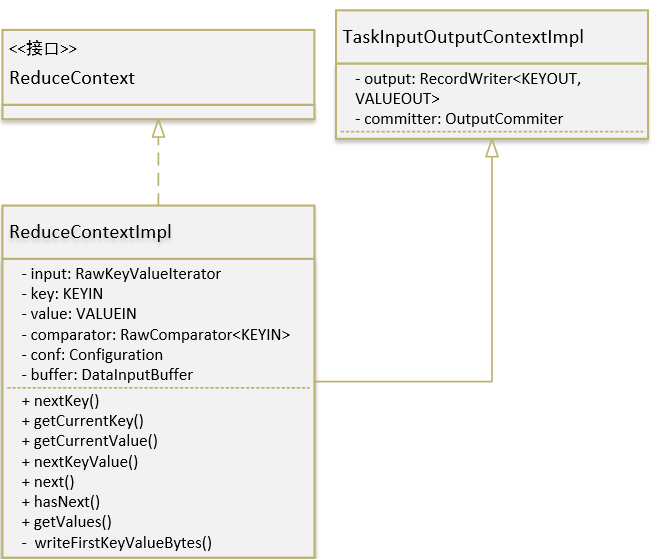
而在MapContextImpl中只是实现了上层接口的getInputSplit()，getCurrentKey/Vaue()以及nextKeyValue()，这些函数的返回值都是该类的成员reader决定的。



图十三，实现的四个方法

1. **ReduceContextImpl**

该类是ReduceContext类的实现，该类和上面的类相比相对要复杂一下。该类的主要的方法有nextKey()以及nextKeyValue()，分别返回下一个键（不同的）以及下一个键值对。而传给ReduceContext的内容是先被拷贝，合并，然后进行排序的，因此可以方便地获取下一个不同的键和下一个键值对，并对其进行了分类。传入ReduceContext的数据使用了DataInputBuffer()，使用缓冲区可以提高I/O性能。



图十四，ReduceContext UML类图

从上述类图中可以得到两个重要的方法，nextKey()和nextKeyValue()，这两个方法在reduce过程中通过迭代器获取键值对的时候是非常重要的。首先是nextKey()方法。该方法目的就是获得下一个不同的从未处理过的key。由于reduce的输入数据是分组的且又有Mapper的局部排序处理（下文会叙述）,所以相同的key它们的键值对都是紧挨着的，所以实现该方法的代码比较简单，只要迭代键值对并判断下一个key是否与当前相同并做出适当停止并返回即可。代码如下

|  |
| --- |
| public boolean nextKey() throws IOException,InterruptedException {  while (hasMore && nextKeyIsSame) {  nextKeyValue();  }  if (hasMore) {  if (inputKeyCounter != null) {  inputKeyCounter.increment(1);  }  return nextKeyValue();  } else {  return false;  }  } |

而nextKeyValue()就是尝试获取下一个键的值，其中涉及到2个参数firstValue和nextKeyIsSame，第一个表示当前key的第一个value，第二个表示当前key是否和下一个key相同，这个参数很重要，对于nextKey()方法，需要获取这个参数来得到下一个不同的key。具体代码如下：

|  |
| --- |
| public boolean nextKeyValue() throws IOException, InterruptedException {  if (!hasMore) {  key = null;  value = null;  return false;  }  firstValue = !nextKeyIsSame; //Judge whether first key value or not  DataInputBuffer nextKey = input.getKey();  currentRawKey.set(nextKey.getData(), nextKey.getPosition(),  nextKey.getLength() - nextKey.getPosition());  buffer.reset(currentRawKey.getBytes(), 0, currentRawKey.getLength());  key = keyDeserializer.deserialize(key);//Get key  DataInputBuffer nextVal = input.getValue();  buffer.reset(nextVal.getData(), nextVal.getPosition(), nextVal.getLength()  - nextVal.getPosition());  value = valueDeserializer.deserialize(value);//Get value  currentKeyLength = nextKey.getLength() - nextKey.getPosition();  currentValueLength = nextVal.getLength() - nextVal.getPosition();  if (isMarked) {  backupStore.write(nextKey, nextVal);  }  hasMore = input.next();  if (hasMore) {  nextKey = input.getKey();  nextKeyIsSame = comparator.compare(currentRawKey.getBytes(), 0,  currentRawKey.getLength(),  nextKey.getData(),  nextKey.getPosition(),  nextKey.getLength()-nextKey.getPosition()) == 0;//Check key of next KV is same or not  } else {  nextKeyIsSame = false;//No data  }  inputValueCounter.increment(1);  return true;  } |

总体来说，这两个类（MapContext以及ReduceContext）都没有对上层的类的方法进行重写，只是对上层接口进行了不同的实现，所以它们的继承是一致的。

1. **关于Job 以及 Map & Reduce 调用流程**

回到第三节的主程序，在代码最后调用了waitForCompletion()方法，对任务上传执行，并返回运行结果。

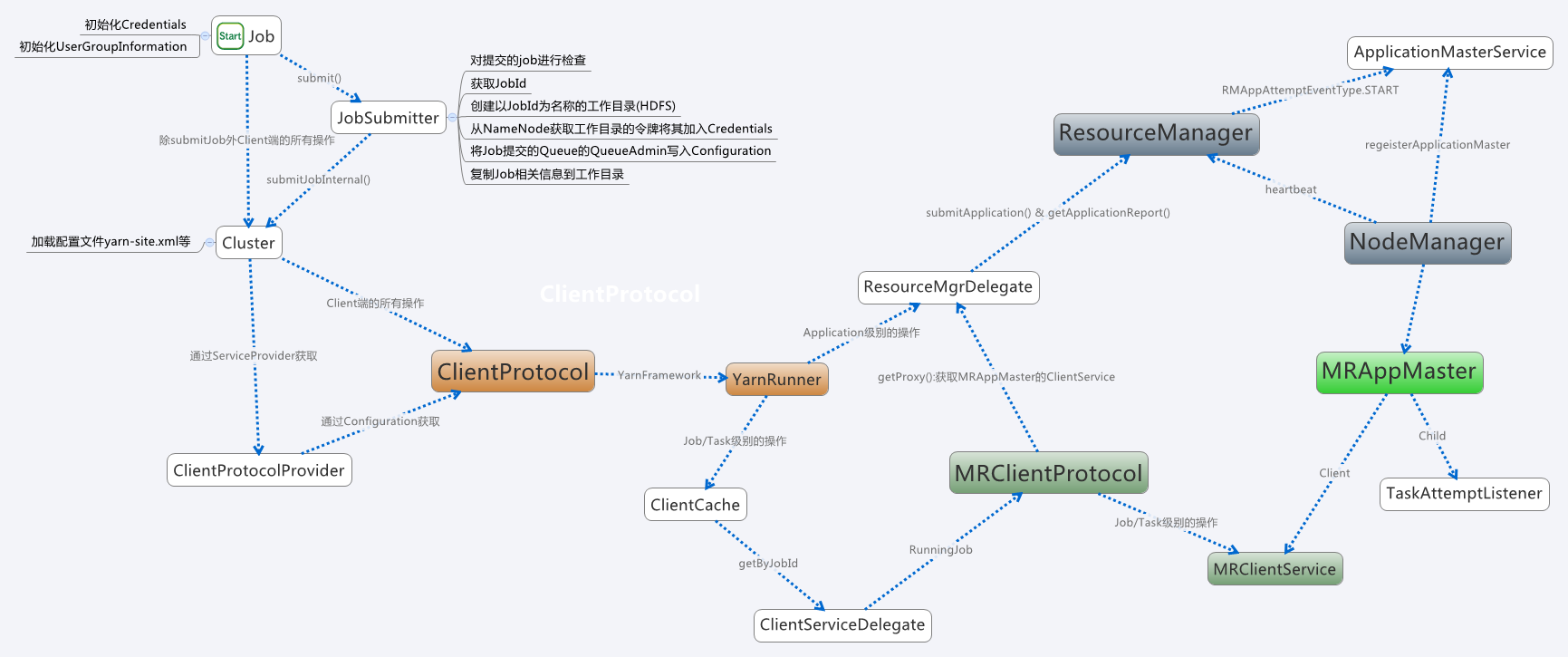
该函数会调用Job类的函数submit()，然后将任务调给JobSubmitter的submitJobInternal()方法处理。JobSubmitter处理好参数后，直接递交给ClientProtocol实例submitClient来提交这个Job。在Hadoop 2.x中，通常实现这个接口的类是YARNRunner, YARNRunner会启动MR和AM，并把任务信息提交给RM，在提交之前，对一个Job分片的创建已经完成了。

|  |
| --- |
| // Create the splits for the job  LOG.debug("Creating splits at " + jtFs.makeQualified(submitJobDir));  int maps = writeSplits(job, submitJobDir);  conf.setInt(MRJobConfig.NUM\_MAPS, maps);  LOG.info("number of splits:" + maps); |

具体的调用代码：

|  |
| --- |
| 1. job.waitForCompletion(true) //In MAIN 2. if (state == JobState.DEFINE) { //In Job.waitForCompletion()   submit();  }   1. return submitter.submitJobInternal(Job.this, cluster); //In Job.submit() 2. submitClient.submitJob(jobId, submitJobDir.toString(), job.getCredentials());//In JobSubmitter.submitJobInternal() |

下面是基于YARN的Job提交的框图：



图十五，Job的提交

如何调用Mapper和Reducer也是要说明的。Job类继承了JobContextImpl类并实现了JobContext接口，它定义了一系列的成员变量，其中就包含了JobConf对象实例conf，它存储了如作业ID，作业的Mapper类，Reducer类，输入输出格式等等，除了ID，其它都是只读的。当一个Job被提交了后任务调度器（YARN中的模块）可以通过conf这个变量在分配MapTask和ReduceTask时来调用指定的Mapper, Reducer以及其它模块。而调用Mapper和Reducer主要是MapTask类和ReduceTask类

1. **MapTask**

**1.1.** MapTask启动时会调用其run()方法，其代码如下：

|  |
| --- |
| public void run(final JobConf job, final TaskUmbilicalProtocol umbilical)  throws IOException, ClassNotFoundException, InterruptedException {  this.umbilical = umbilical;//1  if (isMapTask()) {  if (conf.getNumReduceTasks() == 0) {  mapPhase = getProgress().addPhase("map", 1.0f);  } else {  mapPhase = getProgress().addPhase("map", 0.667f);  sortPhase = getProgress().addPhase("sort", 0.333f);  }  }  TaskReporter reporter = startReporter(umbilical);//2  boolean useNewApi = job.getUseNewMapper();//3  initialize(job, getJobID(), reporter, useNewApi);//4  // check if it is a cleanupJobTask //5  if (jobCleanup) {  runJobCleanupTask(umbilical, reporter);  return;  }  if (jobSetup) {  runJobSetupTask(umbilical, reporter);  return;  }  if (taskCleanup) {  runTaskCleanupTask(umbilical, reporter);  return;  }  if (useNewApi) {  runNewMapper(job, splitMetaInfo, umbilical, reporter);//6  } else {  runOldMapper(job, splitMetaInfo, umbilical, reporter);  }  done(umbilical, reporter);  } |

1. 对于注释1，该语句负责与TaskTracker进行通信，可获得必要的对象信息；
2. 对于注释2，启动TaskReporter线程，它是Task的内部类，用于提交任务状态，为了不影响主线程，所以其另开了一个线程；
3. 对于注释3，其来得到这个Task使用的是新的API还是旧的API；
4. 对于注释4，初始化Task，将其状态设置为RUNNING，并获得输入输出的格式、提交器以及输出路径，然后获取当前内存、CPU等资源信息；
5. 对于注释5，判断这个Task是用于清理的/初始化的，若是用于清理的，则会运行runJobCleanupTask()来清理这个Job，若是初始化则会调用runJobSetupTask()方法来为建立Job做准备；
6. 对于注释6，这是真正执行Mapper的步骤，根据使用API的版本，调用是使用哪种Mapper，下面的内容只分析runNewMapper();

**1.2**. 下面就分析执行Mapper的方法runNewMapper()的运行流程。该方法传入四个参数JobConf, TaskSplitIndex, TaskUmblicalProtocol和TaskReporter。

1. 首先得到该Task的Context;

|  |
| --- |
| org.apache.hadoop.mapreduce.TaskAttemptContext taskContext = new org.apache.hadoop.mapreduce.task.TaskAttemptContextImpl(job, getTaskID(),reporter); |

1. 然后获得我们指定的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); |

1. 然后获得输入的格式和Split信息并对其进行重建；

|  |
| --- |
| org.apache.hadoop.mapreduce.InputFormat<INKEY,INVALUE> inputFormat =  (org.apache.hadoop.mapreduce.InputFormat<INKEY,INVALUE>)  ReflectionUtils.newInstance(taskContext.getInputFormatClass(), job);  org.apache.hadoop.mapreduce.InputSplit split = null;  split = getSplitDetails(new Path(splitIndex.getSplitLocation()),  splitIndex.getStartOffset()); |

1. 根据InputFormat来创建RecorderReader对象，默认是LineRecordReader，该对象用来读取分片中的内容，然后生成RecordWriter用于输出内容；

|  |
| --- |
| org.apache.hadoop.mapreduce.RecordReader<INKEY,INVALUE> input =  new NewTrackingRecordReader<INKEY,INVALUE>  (split, inputFormat, reporter, taskContext);//Just Record Reader |

1. 开始调用RecorderReader实例的initialize()方法并开始读取文件分片的内容；
2. 执行Mapper，调用mapper.run(context)方法即可，这是Map的核心阶段，在map后会将结果输出；
3. 最后关闭输入输出流；

以上是执行Mapper的流程，不过有些地方还需要说明。在代码中有一段为：

|  |
| --- |
| if (job.getNumReduceTasks() == 0) {  output =  new NewDirectOutputCollector(taskContext, job, umbilical, reporter);  } else {  output = new NewOutputCollector(taskContext, job, umbilical, reporter);  } |

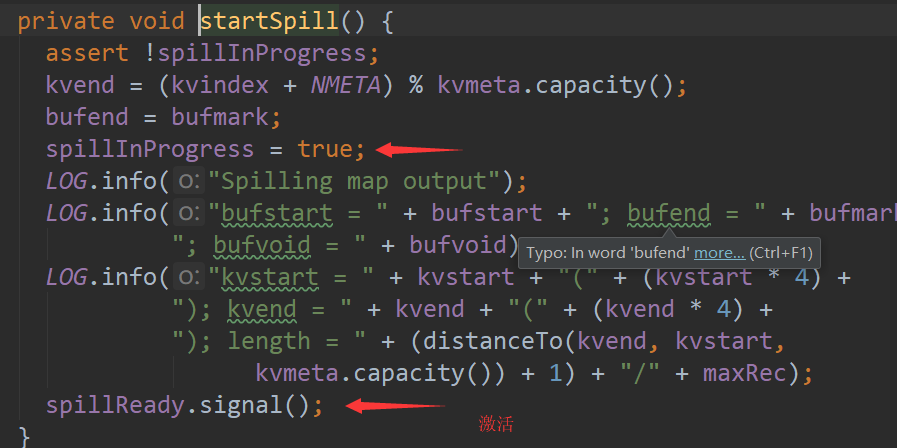
该代码段用于获得输出对象。在Job实例里可以获得Reduce Task的数目的信息，假如这个Job没有Reduce的流程，那么会将Mapper输出的信息直接写入HDFS中，它会创建RecordWriter输出流out，然后调用out.write(K, V)来写入HDFS中；假如Job是有Reduce流程的，系统会创建NewOutputCollector对象给output，这个类是MapTask的一个内部类，主要的成员有MapOutputCollector<K,V> collector和mapreduce.Partitioner<K,V> partitioner，第一个成员的write()方法会调用collector.collect(key, value,partitioner.getPartition(key, value, partitions))，它回将结果写入缓冲区中（下面会叙述），而第二个成员是将Mapper输出的数据进行分区，然后汇总对应的Reducer上。

**1.3.** 上一段涉及了一个/一组类叫MapOutputCollector，在Mapper输出结果的时候会被使用，实际上它是一个接口，而在MapTask中是以MapOutputBuffer类来实现的。该类使用了缓冲区来存储输出数据，当达到一定阈值后会将缓冲区的数据写回硬盘中，而在此过程中生产者仍旧可以不断往缓冲区添加数据，实现了读写并行。

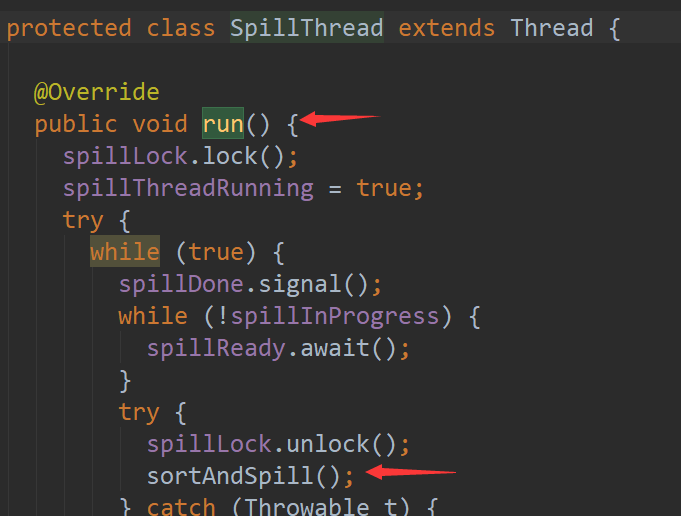
缓冲区以byte数组kvbuffer表示，用来存实际的键值对数据。MapOutputCollect类的核心方法是collect()，它每次都会检查缓冲区是否超过阈值，假如超过了就会调用spill线程将数据写回硬盘中，当然传入的K-V对可以被序列化并写入缓冲区中。

|  |
| --- |
| bufferRemaining -= METASIZE;  if (bufferRemaining <= 0) { //Check  spillLock.lock();  …  startSpill() //Start spill  spillLock.unlock(); //Finish} |

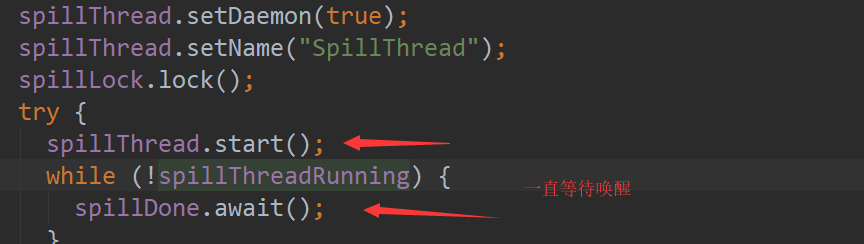
观察该类的init()函数就可以发现，spill的线程已经被启动了，而且其run()方法一直等待被唤醒，若被唤醒则会调用sortAndSpill()方法（startSpill()方法会唤醒该线程），该方法会对分区的数据先局部排序，然后写到磁盘中，也即做到了局部排序。



图十六，startSpill()激活spillThread线程

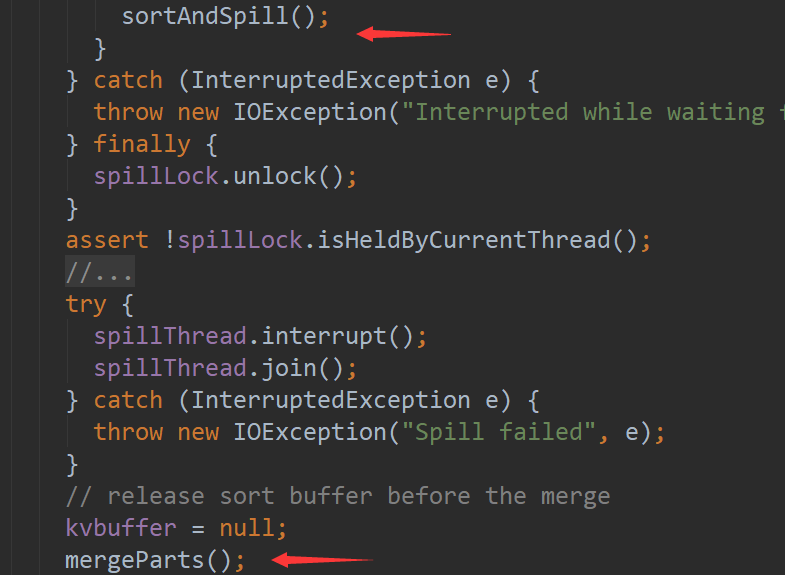


图十七，SpillThread的run()实际调用了sortAndSpill()方法



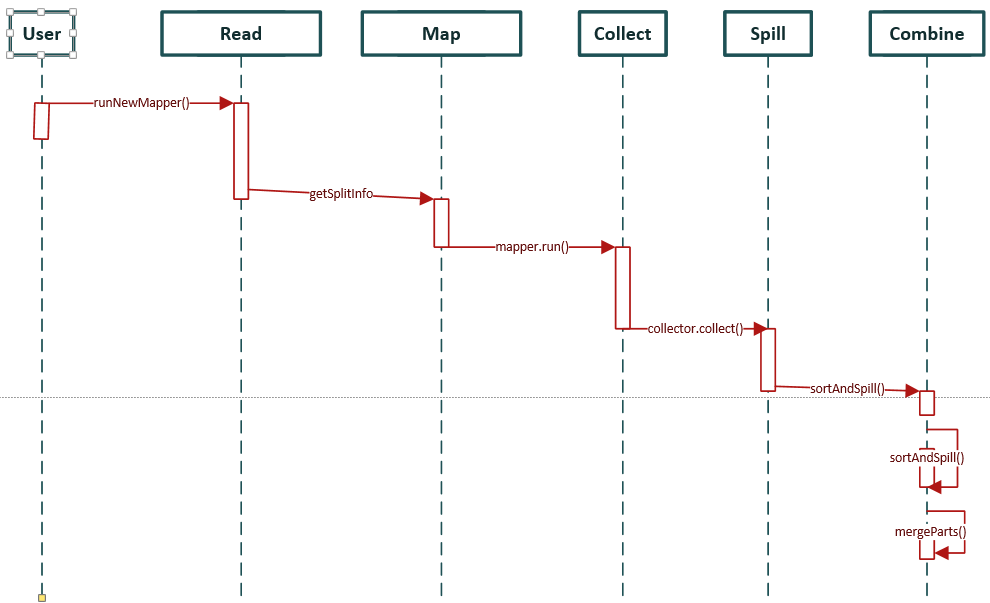
图十八，spillThread是启动并等待激活状态

**1.4.** 最后的会关闭输入输出流，当调用output.close()时会调用NewOutputCollector.close()方法，它会执行flush()操作，剩余的缓冲区数据通过sortAndSpill()写入本地文件，最后调用mergeParts()方法来为一个Task合并成一个文件，给Reducer用。



图十九，flush()方法调用的两个关键方法

至此，Map的任务就算完成了。总体时序图如下：



图二十，Map时序图

1. **ReduceTask**

**2.1.** ReduceTask类用于处理Reduce，相比与MapTask要稍微简单一点。启动该Task也需要调用其run()方法，所以先从该方法开始入手：

|  |
| --- |
| public void run(JobConf job, final TaskUmbilicalProtocol umbilical)  throws IOException, InterruptedException, ClassNotFoundException {  job.setBoolean(JobContext.SKIP\_RECORDS, isSkipping());  if (isMapOrReduce()) { //1  copyPhase = getProgress().addPhase("copy");  sortPhase = getProgress().addPhase("sort");  reducePhase = getProgress().addPhase("reduce");  }  TaskReporter reporter = startReporter(umbilical);  // 2  boolean useNewApi = job.getUseNewReducer();  initialize(job, getJobID(), reporter, useNewApi);  // 3  if (jobCleanup) {  runJobCleanupTask(umbilical, reporter);  return;  }  if (jobSetup) {  runJobSetupTask(umbilical, reporter);  return;  }  if (taskCleanup) {  runTaskCleanupTask(umbilical, reporter);  return;  }  codec = initCodec();  RawKeyValueIterator rIter = null;  ShuffleConsumerPlugin shuffleConsumerPlugin = null;    Class combinerClass = conf.getCombinerClass();  CombineOutputCollector combineCollector =  (null != combinerClass) ?  new CombineOutputCollector(reduceCombineOutputCounter, reporter, conf) : null;  Class<? extends ShuffleConsumerPlugin> clazz =  job.getClass(MRConfig.SHUFFLE\_CONSUMER\_PLUGIN, Shuffle.class, ShuffleConsumerPlugin.class);    shuffleConsumerPlugin = ReflectionUtils.newInstance(clazz, job);  ShuffleConsumerPlugin.Context shuffleContext = new ShuffleConsumerPlugin.Context(...);  shuffleConsumerPlugin.init(shuffleContext);  rIter = shuffleConsumerPlugin.run();//4  // free up the data structures  mapOutputFilesOnDisk.clear();    sortPhase.complete();  setPhase(TaskStatus.Phase.REDUCE);  statusUpdate(umbilical);  Class keyClass = job.getMapOutputKeyClass();  Class valueClass = job.getMapOutputValueClass();  RawComparator comparator = job.getOutputValueGroupingComparator();  if (useNewApi) {//5  runNewReducer(job, umbilical, reporter, rIter, comparator,  keyClass, valueClass);  } else {  runOldReducer(job, umbilical, reporter, rIter, comparator,  keyClass, valueClass);  }  shuffleConsumerPlugin.close();  done(umbilical, reporter);  } |

1. 对于注释1，添加Reduce过程中的阶段，可知需要有Copy, Sort和Reduce阶段；
2. 对于注释2，默认使用新的API并进行Reduce Task的初始化；
3. 对于注释3，这几个if语句根据不同的任务类型来进行不同的操作，类似于MapTask的几个if语句的判断，如有进行清理工作（在结束阶段，并删除工作目录）和初始阶段（建立工作目录）；
4. 对于注释4，前面创建了ShuffleComsumerPlugin的对象，并得到其Context对象，对其初始化，最后执行其run()方法，在这期间会做三件事：Copy, Merge以及Sort，然后返回一个键值对迭代器对象（里面有一组K-V对）以便于下面进行的Reduce运算。这在下文会详细叙述；
5. 对于注释5，对Mapper的数据进行处理之后，接下来就进入Reduce阶段，这里只分析调用新API的Reduce流程，结果的文件写到HDFS上，也即执行了write操作；
   1. 首先分析Reduce Task的Copy, Merge 和 Sort流程。进入Shuffle类（它是ShuffleComsumerPlugin接口的实现类）的run()方法。首先是Copy，Shuffle类会建立一组Fetcher线程，加入只是运行本地则只建立一个Fetcher，否则默认是5个。然后分别唤醒这些Fetcher线程，让其从不同的机器上并行拷贝Mapper输出的数据。

|  |
| --- |
| 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();  }  } |

然后进入Merge和Sort阶段，其实这两个阶段是一起运行的，也即一次的多路归并排序。先创建RawKeyValueIterator键值对迭代器，然后通过merger进行归并排序，归并排序之后一个Task生成一个文件，最后返回在起始位置的kvIter。

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

* 1. 在Shuffle, Merge, Sort完成之后，就要进入Reduce阶段。我们进入runNewReducer()方法，关键代码如下：

|  |
| --- |
| org.apache.hadoop.mapreduce.TaskAttemptContext taskContext =  new org.apache.hadoop.mapreduce.task.TaskAttemptContextImpl(job,  getTaskID(), reporter);  org.apache.hadoop.mapreduce.Reducer<INKEY,INVALUE,OUTKEY,OUTVALUE> reducer =  (org.apache.hadoop.mapreduce.Reducer<INKEY,INVALUE,OUTKEY,OUTVALUE>)  ReflectionUtils.newInstance(taskContext.getReducerClass(), job);  org.apache.hadoop.mapreduce.RecordWriter<OUTKEY,OUTVALUE> trackedRW =  new NewTrackingRecordWriter<OUTKEY, OUTVALUE>(this, taskContext);  ...  org.apache.hadoop.mapreduce.Reducer.Context  reducerContext = createReduceContext(reducer, job, getTaskID(),  rIter, reduceInputKeyCounter,  reduceInputValueCounter,  trackedRW,  committer,  reporter,comparator,keyClass,  valueClass);  try {  reducer.run(reducerContext);  } finally {  trackedRW.close(reducerContext);  } |

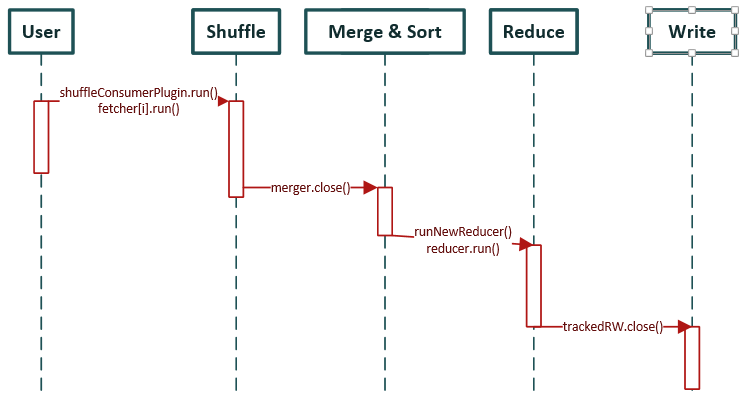
这里的rIter参数是mapred.Merger.MergerQueue类，然后复制给新的rawIter，然后rIter重新实现了RawKeyValueIterator接口以跟踪和汇报进度。

然后就是构造任务配置类和获取指定的Reducer类实例，然后创建RecordWriter实例trackedRW进行结果的输出。

之后就将rIter以及trackedRW等信息传入Context，构造了一个管理读写的配置对象。ReduceContext的对输入的实现使用迭代器的操作，在TaskInputOutputContext实现了输出的方法，其write()方法会直接调用trackedRW的write()方法。

配置完成后，将调用reducer的run()方法执行Reduce操作，最后关闭输出流。整体的Reduce的流程以及写回结果数据的流程也就结束了。

至此，Reduce Task的任务就结束了，总体的时序图由下所示：



图二十一，Reduce Task运行时序图

总而言之，对于Map-Reduce运算流程可以归结为如下：

1. **Map阶段**

* **Read**: 读取数据源，并根据分片生成一个个键值对；
* **Map**: 处理解析的键值对，产生新的键值对（丢给map函数处理）；
* **Collect**: 输出结果，存于缓冲区；
* **Spill**: 若内存满，则将数据写回硬盘生成临时文件；
* **Combine**: 合并临时文件。

1. **Reduce阶段**

* **Shuffle**: ReduceTask会到各个MapTask复制一份数据，若内存区满，则写回硬盘；
* **Merge**: 合并内存和磁盘上的文件；
* **Sort**: MapTask对其局部进行排序，ReduceTask对所得到的文件进行一次归并排序；
* **Reduce**: 将数据集递交给reduce函数就行规约运算；
* **Write**: 将运算结果写到HDFS上。

1. **总结**

本文对Hadoop的MapReduce框架进行了简要的分析，涉及到了Job的提交，Mapper以及Reducer执行的四个阶段，Mapper和Reducer进行Map和Reduce的一些示例的算法，MapTask以及ReduceTask执行流程（即如何真正执行Map以及Reduce）等等。不过由于自己是第一次接触分布式计算，第一次接触到如此大规模的软件项目，感觉自己在阅读源码的时候不能很好地理解和消化，感觉很吃力。再加上自己对分布式并行计算模型的不了解，对于其设计的精髓不能很好的理解，所以本文源码的阅读结果，其深度和广度也是远远不够的。

不过通过本次源码的阅读，我对MapReduce的计算模型有了一定的了解，开阔了视野。并且Hadoop在代码设计上也是很不错的，虽然整体代码十分庞大，但是各个项目，各个包的耦合性比较低，而每个类都各司其职，内聚性比较高，符合“高内聚低耦合”的原则。各个功能通常以接口形式实现，层次结构分明，已于代码的重用以及扩展。这些都是我以后在写程序的时候可以借鉴的。

总体来说，虽然本次源码阅读与分析的难度有点高，自己的理解水平有所不足，但是还是学到了一些东西，有所收获的。

**参考资料**：

[1] http://hadoop.apache.org/docs/r2.8.0/

[2] http://www.cnblogs.com/lxf20061900/p/3792190.html

[3] http://www.cnblogs.com/lxf20061900/p/3794514.html

**附录（WordCount.java源代码）:**

|  |
| --- |
| /\*\*  \* Licensed to the Apache Software Foundation (ASF) under one  \* or more contributor license agreements. See the NOTICE file  \* distributed with this work for additional information  \* regarding copyright ownership. The ASF licenses this file  \* to you under the Apache License, Version 2.0 (the  \* "License"); you may not use this file except in compliance  \* with the License. You may obtain a copy of the License at  \*  \* http://www.apache.org/licenses/LICENSE-2.0  \*  \* Unless required by applicable law or agreed to in writing, software  \* distributed under the License is distributed on an "AS IS" BASIS,  \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  \* See the License for the specific language governing permissions and  \* limitations under the License.  \*/  package org.apache.hadoop.examples;  import java.io.IOException;  import java.util.StringTokenizer;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.Mapper;  import org.apache.hadoop.mapreduce.Reducer;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  import org.apache.hadoop.util.GenericOptionsParser;  public class WordCount {  public static class TokenizerMapper  extends Mapper<Object, Text, Text, IntWritable>{    private final static IntWritable one = new IntWritable(1);  private Text word = new Text();    public void map(Object key, Text value, Context context  ) throws IOException, InterruptedException {  StringTokenizer itr = new StringTokenizer(value.toString());  while (itr.hasMoreTokens()) {  word.set(itr.nextToken());  context.write(word, one);  }  }  }    public static class IntSumReducer  extends Reducer<Text,IntWritable,Text,IntWritable> {  private IntWritable result = new IntWritable();  public void reduce(Text key, Iterable<IntWritable> values,  Context context  ) throws IOException, InterruptedException {  int sum = 0;  for (IntWritable val : values) {  sum += val.get();  }  result.set(sum);  context.write(key, result);  }  }  public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();  if (otherArgs.length < 2) {  System.err.println("Usage: wordcount <in> [<in>...] <out>");  System.exit(2);  }  Job job = Job.getInstance(conf, "word count");  job.setJarByClass(WordCount.class);  job.setMapperClass(TokenizerMapper.class);  job.setCombinerClass(IntSumReducer.class);  job.setReducerClass(IntSumReducer.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);  for (int i = 0; i < otherArgs.length - 1; ++i) {  FileInputFormat.addInputPath(job, new Path(otherArgs[i]));  }  FileOutputFormat.setOutputPath(job,  new Path(otherArgs[otherArgs.length - 1]));  System.exit(job.waitForCompletion(true) ? 0 : 1);  }  } |