第5章

MapReduce分布式计算

5.1 MapReduce 简介

Hadoop MapReduce 是一个快速、高效、简单用于编写并行处理大数据程序及应用在大集群上的编程框架。其前身是 Google 公司的 MapReduce，它是 Google 公司的核心计算模型，将复杂的、运行于大规划集群上的并行计算过程高度地抽象到了两个函数：Map 和 Reduce。它适合用 MapReduce 来处理的数据集(或任务)，需要满足一个基本要求：待处理的数据集可以分解成许多小的数据集，而且每一个小数据集都可以完全并行地进行处理。概念 Map(映射)和 Reduce(归纳)及它们的主要思想，都是从函数式编程语言中借来的，同时包含了从矢量编程语言中借来的特性。Hadoop MapReduce 极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。

5.1.1 MapReduce 架构

和 HDFS 一样，MapReduce 也是采用 Master/Slave 的架构，其架构如图 5-1 所示。

它主要由 Client、JobTracker、TaskTracker 及 Task 4 个部分组成。

(1) Client 会在用户端通过 Client 类将应用配置参数打包成 jar 文件存储到 hdfs，并把路径提交到 Jobtracker，然后由 JobTracker 创建每一个 Task(即 MapTask 和 ReduceTask)，并将它们分发到各个 TaskTracker 服务中去执行。

(2) JobTracker 负责资源监控和作业调度。JobTracker 监控所有 TaskTracker 与 job 的健康状况，一旦发现失败，就将相应的任务转移到其他节点。同时，JobTracker 会跟踪任务的执行进度、资源使用量等信息，并将这些信息告诉任务调度器，而调度器会在资源出现空闲时，选择合适的任务使用这些资源。在 Hadoop 中，任务调度器是一个可插拔的模块，用户可以根据自己的需要设计相应的调度器。

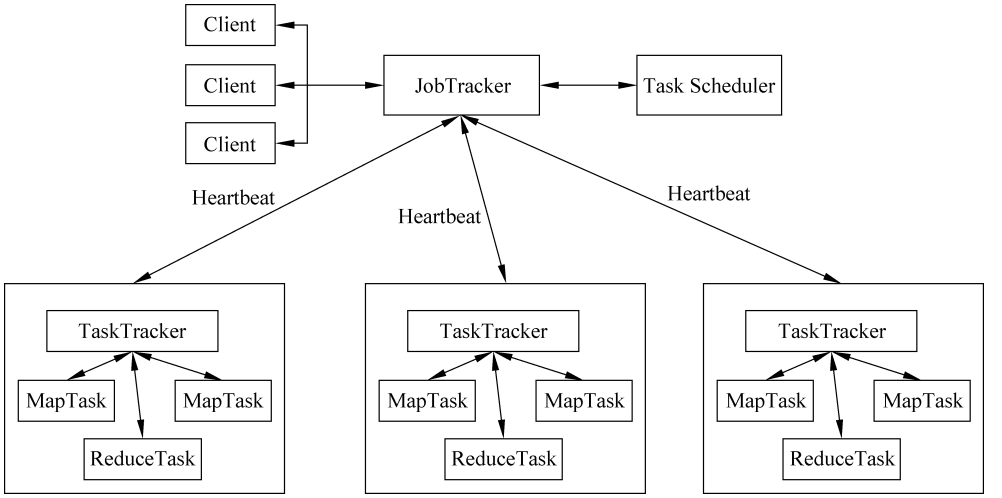

图 5-1 MapReduce 架构图

(3) TaskTracker 会周期性地通过 Heartbeat 将本节点上资源的使用情况和任务的运行进度汇报给 JobTracker,同时接收 JobTracker 发送过来的命令并执行相应的操作(如启动新任务、结束任务等)。TaskTracker 使用 slot 等量划分本节点上的资源量。slot 代表计算资源(CPU、内存等)。一个 Task 获取一个 slot 后才有机会运行,而 Hadoop 调度器的作用就是将各个 TaskTracker 上的空闲 slot 分配给 Task 使用。slot 分为 Map slot 和 Reduce slot 两种,分别供 MapTask 和 ReduceTask 使用。TaskTracker 通过 slot 数目(可配置参数)限定 Task 的并发度。

(4) TaskTracker 分为 MapTask 和 ReduceTask 两种,均由 TaskTracker 启动。HDFS 以固定大小的 block 为基本单位存储数据,而对于 MapReduce 而言,其处理单位是 split。split 是一个逻辑概念,只包含一些元数据信息,如数据起始位置、数据长度、数据所在节点等。它的划分方法完全由用户自行决定。但需要注意的是,split 的多少决定了 MapTask 的数目,因为每个 split 只会交给一个 MapTask 处理。split 和 block 的关系如图 5-2 所示。

MapTask 的执行过程如图 5-3 所示。由图 5-3 可知,MapTask 先将对应的 split 迭代解析成一个个 key/value 对,依次调用用户自定义的 map() 函数进行处理,最终将临时结果存放到本地磁盘上,其中临时数据被分成若干个 partition,每个 partition 将被一个 ReduceTask 处理。

ReduceTask 的执行过程如图 5-4 所示。该过程分为以下 3 个阶段。

(1) 从远程节点上读取 MapTask 中间结果(称为“shuffle 阶段”)。

(2) 按照 key/value 对进行排序(称为“sort 阶段”)。

(3) 依次读取< key, value list >,调用用户自定义的 reduce() 函数处理,并将最终结果保存到 HDFS 上(称为“reduce 阶段”)。

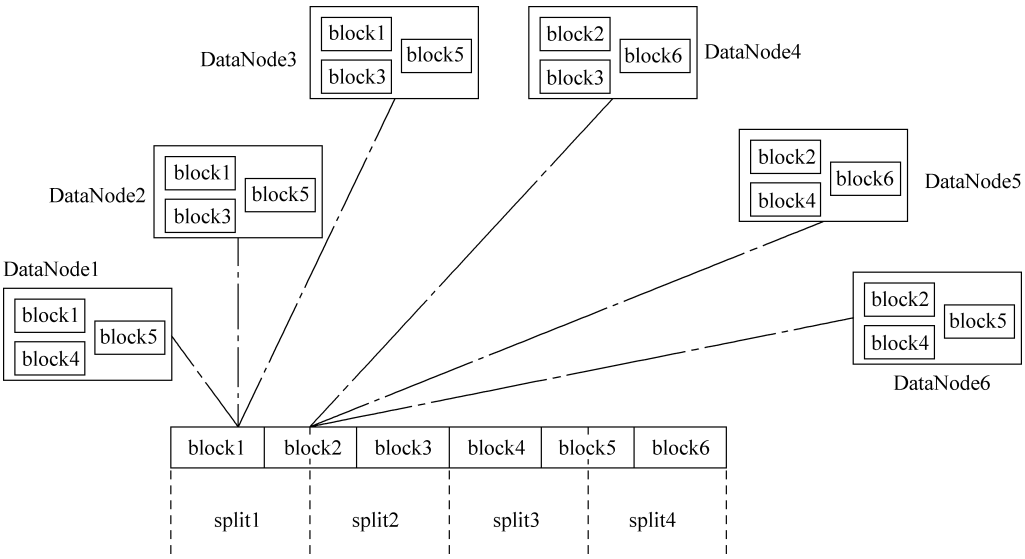

图 5-2　split 和 block 的关系

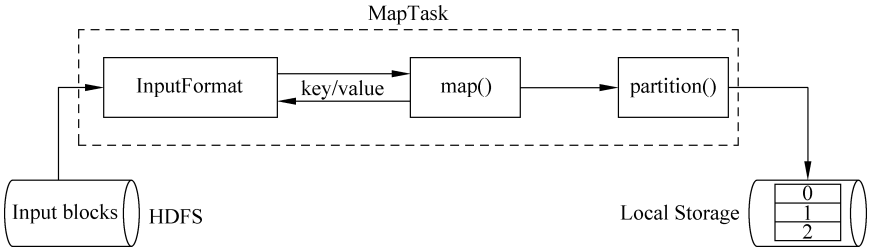

图 5-3　MapTask 的执行过程

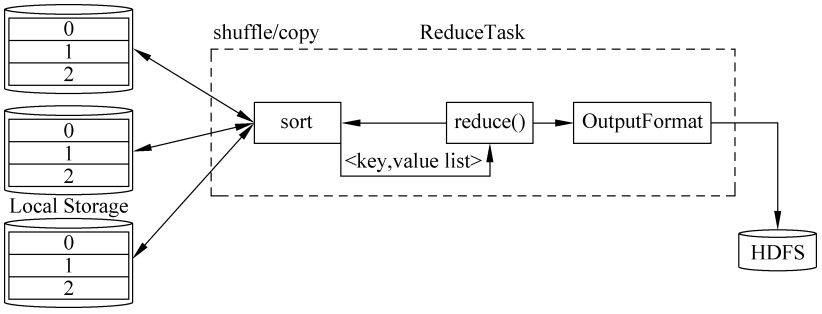

图 5-4　ReduceTask 的执行过程

5.1.2　MapReduce 的原理

MapReduce 采用的是“分而治之”的策略，当我们处理大规模的数据时，将这些数据拆解成多个部分，并利用集群的多个节点同时进行数据处理，然后对各个节点得到的中间结果进行汇总，经过进一步的计算，得到最终结果。

一个 MapReduce 作业(job)通常会把输入的数据集切分为若干个独立的数据块,由 map 任务(task)以完全并行的方式处理它们。框架会对 map 的输出先进行排序,然后把结果输入给 reduce 任务。通常,作业的输入和输出都会被存储在文件系统中,整个框架负责任务的调度和监控,以及重新执行已经失败的任务。

通常,MapReduce 框架的计算节点和存储节点是运行在一组相同的节点上,也就是说,运行 MapReduce 框架和运行 HDFS 文件系统的节点通常是在一起的。这种配置允许框架在那些已经存好的数据的节点上高效地调度任务,这可以使整个集群的网络带宽被非常高效地利用。

MapReduce 框架由一个主节点(ResourceManager)、多个子节点(运行 NodeManager)和 MRAppMaster(每个任务一个)共同组成。应用程序至少应该指明输入/输出的位置(路径),并通过实现合适的接口或抽象类提供 map 和 reduce 函数,再加上其他作业的参数,就构成了作业配置(job configuration)。Hadoop 的 job client 提交作业(jar 包/可执行程序等)和配置信息给 ResourceManager,后者负责分发这些软件和配置信息给 slave、调度任务且监控它们的执行,同时提供状态和诊断信息给 job-client。

虽然 Hadoop 框架是用 Java 实现的,但 MapReduce 应用程序不一定要用 Java 来写,也可以使用 Ruby、Python、C++等来编写。

MapReduce 框架的流程如图 5-5 所示。

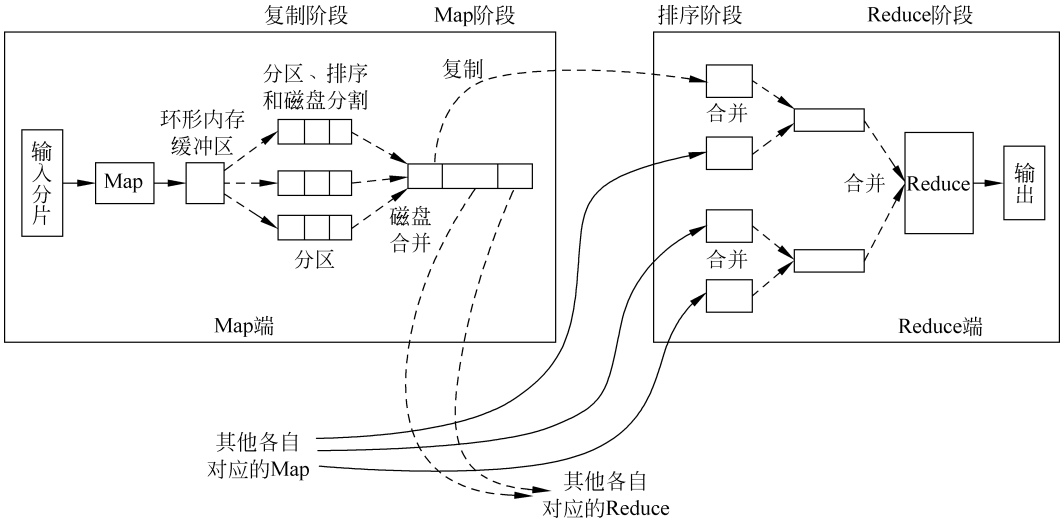

图 5-5 MapReduce 框架的流程图

针对上面的流程可以分为两个阶段来描述。

1. Map 阶段

(1) InputFormat 根据输入文件产生键值对,并传送到 Mapper 类的 map 函数中。

(2) Map 输出键值对到一个没有排序的缓冲内存中。

(3) 当缓冲内存达到给定值或 map 任务完成,在缓冲内存中的键值对就会被排序,

然后输出到磁盘中的溢出文件中。

(4) 如果有多个溢出文件,那么就会整合这些文件到一个文件中,且是排序的。

(5) 这些排序过的、在溢出文件中的键值对会等待 Reducer 类的获取。

2. Reduce 阶段

(1) Reducer 类获取 Mapper 类的记录,然后产生另外的键值对,最后输出到 HDFS 中。

(2) Reducer 类中的 reduce 方法针对每个 key 调用一次。

(3) Shuffle: 相同的 key 被传送到同一个的 Reducer 类中。

(4) 当有一个 Mapper 类完成后,Reducer 类就开始获取相关数据,所有的溢出文件会被排到一个内存缓冲区中。

(5) 当 Reducer 所有相关的数据都传输完成后,所有溢出文件就会被整合和排序。

(6) 当内存缓冲区满了后,就会在本地磁盘产生溢出文件。

(7) Reducer 类输出到 HDFS。

5.1.3　MapReduce 的工作机制

1. MapReduce 运行图

MapReduce 运行图如图 5-6 所示。

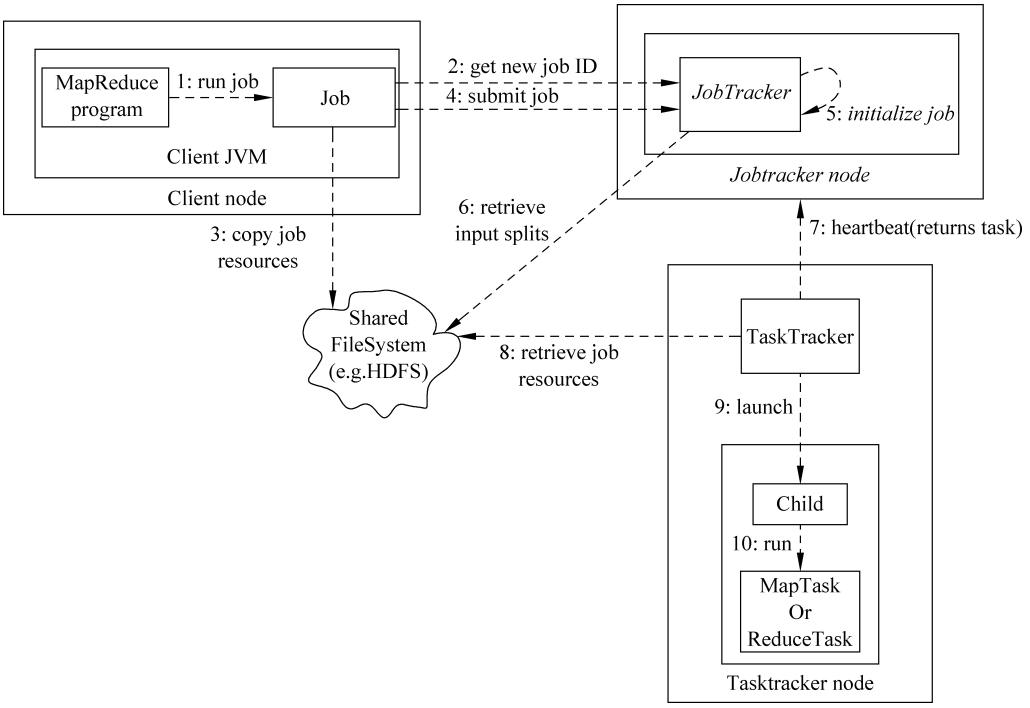

图 5-6　MapReduce 运行图

2. 运行解析

1) 作业的提交

(1) 此方法调用 submit()。在 submit()方法里面连接 JobTracker,即生成一个内部 JobSummitter(实际上是 new JobClient()),在 new JobClient()里面生成一个 JobSubmissionProtocol 接口(JobTracker 实现了此接口)对象 jobSubmitClient(是它连接或对应着 JobTracker),在 submit()方法里面也调用 JobClient. submitJobInternal(conf)方法返回一个 RunningJob(步骤 1)。

(2) 参数 true 说明要调用方法 jobClient. monitorAndPrintJob()即检查作业的运行情况(每秒一次),如果有变化就报告给控制台。

jobClient. submitJobInternal()所实现的提交作业过程如下。

① 向 JobTracker 请求一个新的 job ID(步骤 2)。

② 检查作业的输出路径,如果未指定或已存在,则不提交作业并抛错误给程序。

③ 计算并生成作业的输入分片,如果路径不存在,则不提交作业并抛错误给程序。

④ 将运行作业所需要的资源(包括作业 jar 文件、配置文件和计算所得的输入分片)复制到 JobTracker 的文件系统中以 job ID 命名的目录下(即 HDFS 中)。作业 jar 副本较多(mapred. submit. replication = 10)(步骤 3)。

⑤ 告知 JobTracker 作业准备执行(真正地提交作业 jobSubmitClient. submitJob())(步骤 4)。

2) 作业的初始化

(1) JobTracker 接收到对其 submitJob()方法的调用后,将其放入内部队列,交由 job scheduler 进行调度,并对其进行初始化,包括创建一个正在运行作业的对象——封装任务和记录信息(步骤 5)。

(2) 为了创建任务运行列表,Job Scheduler 首先从共享文件系统中获取已计算好的输入分片信息(步骤 6),然后为每个分片创建一个 map 任务。

(3) 创建的 reduce 任务数量由 Job 的 mapred. reduce. task 属性决定(setNumReduceTasks()设置),schedule 创建相应数量的 reduce 任务。任务在此时被指定 ID。

(4) 除了 map 和 reduce 任务,还有 setupJob 和 cleanupJob 需要建立:由 TaskTrackers 在所有 map 开始前和所有 reduce 结束后分别执行,这两个方法在 OutputCommitter 中(默认是 FileOutputCommitter)。setupJob()创建输出目录和任务的临时工作目录;cleanupJob()删除临时工作目录。

3) 作业的分配

(1) 每个 TaskTracker 定期发送心跳给 JobTracker,告知自己还活着,并附带消息说明自己是否已准备好接受新任务。JobTracker 以此来分配任务,并使用心跳的返回值与 TaskTracker 通信(步骤 7)。JobTracker 利用调度算法先选择一个 job,然后再选此 job 的一个 task 分配给 TaskTracker。

(2) 每个 TaskTracker 会有固定数量的 map 和 reduce 任务槽,数量由 TaskTracker

核的数量和内存大小来决定。JobTracker 会先将 TaskTracker 的所有的 map 槽填满，然后才填此 TaskTracker 的 reduce 任务槽。

(3) JobTracker 分配 map 任务时会选取与输入分片最近的 TaskTracker，分配 reduce 任务用不着考虑数据本地化。

4) 任务的执行

(1) TaskTracker 分配到一个任务后，首先从 HDFS 中把作业的 jar 文件及运行所需要的全部文件(DistributedCache 设置的)复制到 TaskTracker 本地(步骤 8)。

(2) TaskTracker 为任务新建一个本地工作目录，并把 jar 文件的内容解压到这个文件夹下。

(3) TaskTracker 新建一个 TaskRunner 实例来运行该任务(步骤 9)。

(4) TaskRunner 启动一个新的 JVM 来运行每个任务(步骤 10)，以便客户的 map/reduce 不会影响 TaskTracker。

5) 进度和状态的更新

一个作业和它的每个任务都有一个状态，包括作业或任务的运行状态(running、successful、failed)、map 和 reduce 的进度、计数器值、状态消息或描述。

map 进度标准是处理输入所占比例；reduce 是 copy\merge\reduce 整个进度的比例。

Child JVM 有独立的线程，每隔 3 秒检查任务更新标志，如果有更新，就会报告给此 TaskTracker。

TaskTracker 每隔 5 秒给 JobTracker 发心跳。

JobTracker 合并这些更新，产生一个表明所有运行作业及其任务状态的全局试图。

JobClient. monitorAndPrintJob()每秒查询这些信息。

6) 作业的完成

当 JobTracker 收到最后一个任务(this will be the special job cleanup task)的完成报告后，便把 job 状态设置为 successful。

job 得到完成信息便从 waitForCompletion()返回。

最后，JobTracker 清空作业的工作状态，并指示 TaskTracker 也清空作业的工作状态(如删除中间输出)。

3. 失败解析

1) 任务失败

(1) 子任务失败。当 map 或 reduce 子任务中的代码抛出异常，JVM 进程会在退出之前向父进程 TaskTracker 发送错误报告，TaskTracker 会将此(任务尝试)task attempt 标记为 failed 状态，释放一个槽以便运行另外一个任务。

(2) JVM 失败。JVM 突然退出，即 JVM 错误，这时 TaskTracker 会注意到进程已经退出，标记为 failed。

① 任务失败有重试机制，重试次数 map 任务设置是由 mapred. map. max. attempts 属性控制的，reduce 是由 mapred. reduce. max. attempts 属性控制的。

② 一些 job 可以完成总体任务的一部分就能够接受，这个百分比由 mapred. map. failures. precent 和 mapred. reduce. failures. precent 参数控制。

③ 任务尝试(task attempt)是可以中止(killed)的。

2) TaskTracker 失败

作业运行期间，TaskTracker 会通过心跳机制不断与系统 JobTracker 通信，如果某个 TaskTracker 运行缓慢或失败及出现故障，TaskTracker 就会停止或很少向 JobTracker 发送心跳，JobTracker 会注意到此 TaskTracker 发送心跳的情况，从而将此 TaskTracker 从等待任务调度的 TaskTracker 池中移除。

(1) 如果是 map 并且成功完成，JobTracker 会安排此 TaskTracker 上成功运行的 map 任务返回。

(2) 如果是 reduce 并且成功完成，则数据直接使用，因为 reduce 只要执行完就会把输出写到 HDFS 上。

(3) 如果它们属于未完成的作业，那么 reduce 阶段无法获取该 TaskTracker 上的本地 map 输出文件，任何任务都需要重新调度。

另外，即使 TaskTracker 没有失败，如果它上面的失败任务远远高于集群的平均失败任务数，也会被列入黑名单。可以通过重启从 JobTracker 的黑名单中移除。

3) JobTracker 失败

JobTracker 失败应该说是最严重的一种失败方式了，而且在 Hadoop 中存在单点故障的情况下是相当严重的，因为在这种情况下作业最终将失败，尽管这种故障的概率极小。未来版本可以通过启动多个 JobTracker，在这种情况只运行一个主的 JobTracker，通过一种机制来确定哪个是主的 JobTracker。

5.2 MapReduce 操作实践


视频讲解

5.2.1 MapReduce WordCount 编程实例

词频统计是最简单也是最能体现 MapReduce 思想的程序之一，可以称为 MapReduce 版“Hello World”，该程序的完整代码可以在 Hadoop 安装包的 src/examples 目录下找到。词频统计主要完成的功能是：统计一系列文本文件中每个单词出现的次数。本节通过分析源代码帮助读者厘清 MapReduce 程序的基本结构。

1. WordCount 代码分析

MapReduce 框架自带的示例程序 WordCount 只包含 Mapper 类和 Reduce 类，其他全部使用默认类。下面为 WordCount 源代码分析。

1) Mapper 类

Map 过程需要继承 org. apache. hadoop. mapreduce 包中的 Mapper 类，并重写 map 方法。

```
  public static class TokenizerMapper extends
    Mapper < Object, Text, Text, IntWritable > {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  //map 方法,划分一行文本,读一单词写出的一<单词,1 >
  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);                    //写出<单词,1 >
    }
  }
}
```

2）Reduce 类

```
  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();        //相当于< Hello,1 >< Hello,1 >将两个 1 相加
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

3）主函数

```
public static void main(String[ ] args) throws Exception {     //主方法,函数入口
  Configuration conf = new Configuration();                     //实例化配置文件类
  String[ ] otherArgs =
      new GenericOptionsParser(conf, args).getRemainingArgs();
  if (otherArgs.length != 2) {
    System.err.println("Usage: wordcount < in > < out >");
    System.exit(2);
  }
  Job job = new Job(conf, "word count");                        //实例化 Job 类
  job.setJarByClass(WordCount.class);                           //设置主类名
  job.setMapperClass(TokenizerMapper.class);                    //指定使用上述自定义 Map 类
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
```

```
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1])); //设置输出结果文件位置
    System.exit(job.waitForCompletion(true) ? 0 : 1);            //提交任务并监控任务状态
  }
```

4）提交 WordCount

```
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();
    //map 方法,划分一行文本,读一单词写出的一<单词,1>
    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);        //写出<单词,1>
      }
    }
  }

  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();          //相当于<Hello,1><Hello,1>将两个 1 相加
      }
      result.set(sum);
      context.write(key, result);
    }
  }

public static void main(String[] args) throws Exception {   //主方法,函数入口
    Configuration conf = new Configuration();               //实例化配置文件类
    String[] otherArgs =
        new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");                  //实例化 Job 类
    job.setJarByClass(WordCount.class);                     //设置主类名
    job.setMapperClass(TokenizerMapper.class);              //指定使用上述自定义 Map 类
```

```
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1])); //设置输出结果文件位置
        System.exit(job.waitForCompletion(true) ? 0 : 1);            //提交任务并监控任务状态
    }
}
```

2. 打包运行

(1) 在 Eclipse 中选择 File→Export 选项,导出 jar 包。

(2) 新建两个文本文件并上传到 HDFS:

```
[root@master ~]# hdfs dfs -mkdir /input
[root@master ~]# hdfs dfs -rm /input/*
[root@master ~]# hdfs dfs -put sw*.txt /input
```

(3) 运行 jar 包:

```
[root@master ~]# hadoop jar WordCountTest.jar WordCountTest /input /output
```

(4) 查看运行结果:

```
[root@master ~]# hdfs dfs -ls /output
[root@master ~]# hdfs dfs -text /output/part-r-00000
```

5.2.2　MapReduce 倒排索引编程实例

1. 简介

倒排索引是文档检索系统中最常用的数据结构,被广泛地应用于全文搜索引擎。它主要用来存储某个单词(或词组)在一个文档或一组文档中的存储位置的映射,即提供了一种根据内容来查找文档的方式。由于不是根据文档来确定文档所包含的内容,而是进行了相反的操作,因而称为倒排索引(inverted index)。通常情况下,倒排索引由一个单词(或词组)及相关的文档列表组成,文档列表中的文档或者标识文档的 ID 号,或者指定文档所在位置的 URI。

2. 分析与设计

本节实现的倒排索引主要关注的信息为单词、文档 URI 及词频。下面根据 MapReduce 的处理过程给出倒排索引的设计思路。

1) Map 过程

首先使用默认的 TextInputFormat 类对输入文件进行处理,得到文本中每行的偏移

量及其内容。显然，Map 过程首先必须分析输入的< key，value >对，得到倒排索引中需要的 3 个信息：单词、文档 URI 和词频。这里存在两个问题：第一，< key，value >对只能有两个值，作为 key 或 value 值；第二，通过一个 Reduce 过程无法同时完成词频统计和生成文档列表，所以必须增加一个 Combine 过程完成词频统计。

这里将单词和 URI 组成 key 值，将词频作为 value，这样做的好处是可以利用 MapReduce 框架自带的 Map 端排序，将同一个文档的相同单词的词频组成列表，传递给 Combine 过程，实现类似于 WordCount 的功能。

2）Combine 过程

经过 map 方法处理后，Combine 过程将 key 值相同的 value 值累加，得到一个单词在文档中的词频。如果单词在文档中词频的输出作为 Reduce 过程的输入，在 Shuffle 过程时将面临一个问题：所有具有相同单词的记录（由单词、URI 和词频组成）应该交由同一个 Reduce 处理，但当前的 key 值无法保证这一点，所以必须修改 key 值和 value 值。这次将单词作为 key 值，URI 和词频组成 value 值。这样做的好处是可以利用 MapReduce 框架默认的 HashPartitioner 类完成 Shuffle 过程，将所有相同的单词发送给同一个 Reduce 处理。

3）Reduce 过程

经过上述两个过程后，Reduce 过程只需要将 key 值相同的 value 值组合成倒排索引文件所需的格式即可，剩下的事情就可以直接交给 MapReduce 框架进行处理了。

4）需要解决的问题

本节设计的倒排索引在文件数目上没有限制，但是单个文件不宜过大（具体值与默认 HDFS 块大小及相关配置有关），要保证每个文件对应一个 split。否则，由于 Reduce 过程没有进一步统计词频，最终结果可能会出现词频未统计完全的单词。因此可以通过重写 Inputformat 类将每一个文件作为一个 split，避免上述情况；或者执行两次 MapReduce，第一次用于统计词频，第二次用于生成倒排索引。除此之外，还可以利用复合键值对等实现包含更多信息的倒排索引。

3. 倒排索引完整源代码

```
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.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```
import org.apache.hadoop.util.GenericOptionsParser;

public class InvertedIndex {
    public static class InvertedIndexMapper extends Mapper<Object, Text, Text, Text> {
        private Text keyInfo = new Text();
        private Text valueInfo = new Text();
        private FileSplit split;

        public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
            split = (FileSplit)context.getInputSplit();
            StringTokenizer itr = new StringTokenizer(value.toString());

            while(itr.hasMoreTokens()) {
                keyInfo.set(itr.nextToken() + ":" + split.getPath().toString());
                valueInfo.set("1");
                context.write(keyInfo, valueInfo);
            }
        }

    }
    public static class InvertedIndexCombiner extends Reducer<Text, Text, Text, Text> {
        private Text info = new Text();
        public void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
            int sum = 0;
            for(Text value : values) {
                sum += Integer.parseInt(value.toString());
            }
            int splitIndex = key.toString().indexOf(":");
            info.set(key.toString().substring(splitIndex + 1) + ":" + sum);
            key.set(key.toString().substring(0, splitIndex));
            context.write(key, info);
        }
    }
    public static class InvertedIndexReducer extends Reducer<Text, Text, Text, Text> {
        private Text result = new Text();
        public void reducer(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
            String fileList = new String();
            for(Text value : values) {
                fileList += value.toString() + ";";
            }
            result.set(fileList);
            context.write(key, result);
        }
    }
    public static void main(String[] args) throws Exception{
        //TODO Auto-generated method stub
```

```
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if(otherArgs.length != 2) {
            System.err.println("Usage: wordcount < in > < out >");
            System.exit(2);
        }
        Job job = new Job(conf, "InvertedIndex");
        job.setJarByClass(InvertedIndex.class);
        job.setMapperClass(InvertedIndexMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setCombinerClass(InvertedIndexCombiner.class);
        job.setReducerClass(InvertedIndexReducer.class);

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

        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

程序运行和 WordCount 同样原理。

小结

本章首先阐述了 MapReduce 架构，然后介绍了 MapReduce 的工作原理和 MapReduce 的工作机制，最后重点介绍了基于 MapReduce 架构的 WordCount 编程实例和倒排索引编程实例。

习题

1. 简述 MapReduce 架构。
2. 简述 MapReduce 的工作原理。
3. 简述 MapReduce 的工作机制。
4. 编写 MapReduce WordCount。
5. 实现 MapReduce 倒排索引编程。