

MapReduce 是怎样工作

在本章，我们将要看看MapReduce的工作细节，这些知识将为我们随后的两章学习编写MapReduce高级编程奠定基础。

剖析MapReduce作业运行

可以通过在一个job 对象上调用简单的方法：submit()来运行一个MapReduce 作业，也可以调用WaitForCompletion（），他们用于提交一个以前没有提交过的作业，然后等待他的完成。这两个方法隐藏了大量的处理细节，本节将要揭示hadoop运行作业时所采取的措施。

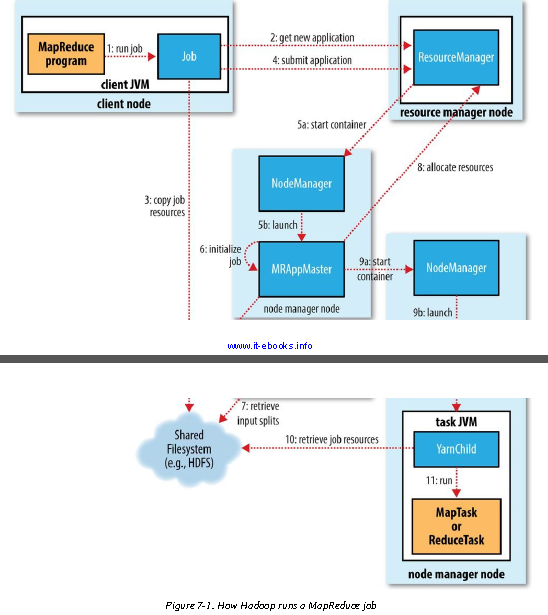


图7-1阐述了整个过程，在高层次角度，他有5主要的组件：

1，客户端，用于提交MapReduce作业

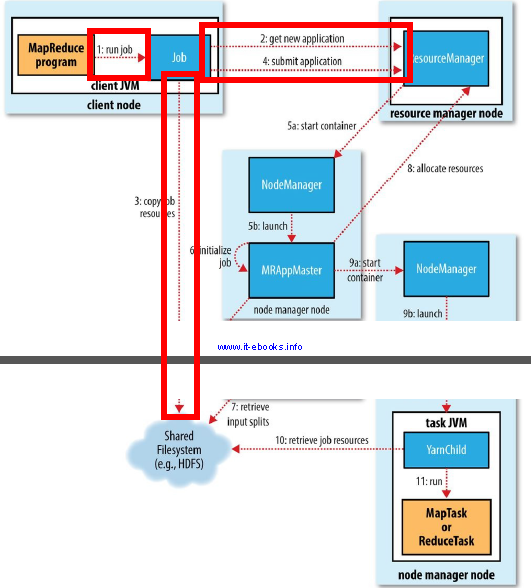
2，YARN 资源管理者，用于协调集群上计算资源的分发

3，YARN node manager，用于运行和监视集群上本机的计算容器

4，MapReduce application master，用于调度MapReduce 作业的运行，application master和MapRedu t作业运行在容器中，容器是被resource manager调度，被node manager管理的一个组件

5，分布式文件系统（一般为HDFS），他被用户在不同的节点之间共享job 的文件

作业的提交

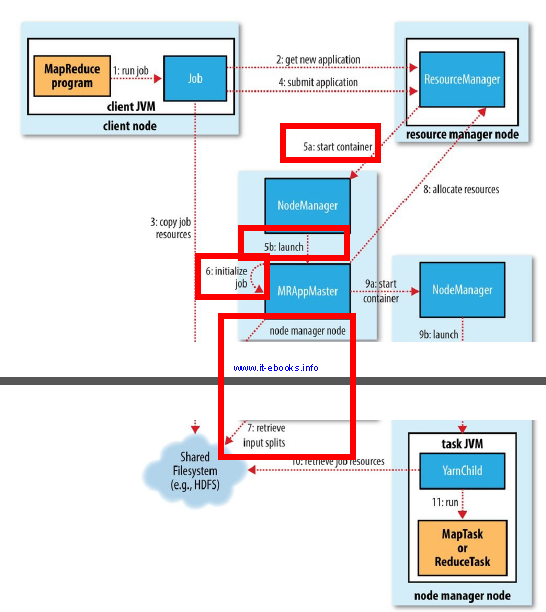


Job的submit（）方法创建一个内部的JobSummiter实例，并且调用submitJobInternal（）方法，作业提交后，waitFormCompletion（）每秒获取作业的进度，并且如果发现自上次报告后有改变，就把进度报告到控制台，当作业运行成功后，如果成功就系那是job的计数器，如果程序运行失败，则把导致作业失败的错误记录到控制台

JobSummiter所实现的作业提交过程如下所示：

* 像resource manager请求一个作业的ID，用于MapReduce job的ID，见step 2
* 检查作业的输出说明。例如，如果输出文件夹没有被制定或者已经存在了，作业就不提交，错误抛回给MapReduce程序
* 计算作业的输入分片。如果分片无法计算，比如因为输入路径不存在，作业就不提交，粗无返回给MapReduce程序
* 将运行作业所需要的资源，包括job JAR文件，配置文件，计算所需要的分片。复制到一个以作业ID命名的目录下的共享文件系统中（step 3），作业JAR的副本较多（有mapreduce.client.submit.file.replication属性进行配置，默认值为10），因此在作业运行的时候，集群中可以有很多个副本工node manager去访问
* 在resource manager 上调用submitApplication来提交作业（step 4）

作业的初始化



当resource manager 接收到他的submitApplication的方法调用后，他会把这个请求切换到YARN scheduler中， scheduler在nodemanager上分配一个container，然后resource manager在此container上运行application master进程，（step 5a和step 5b）

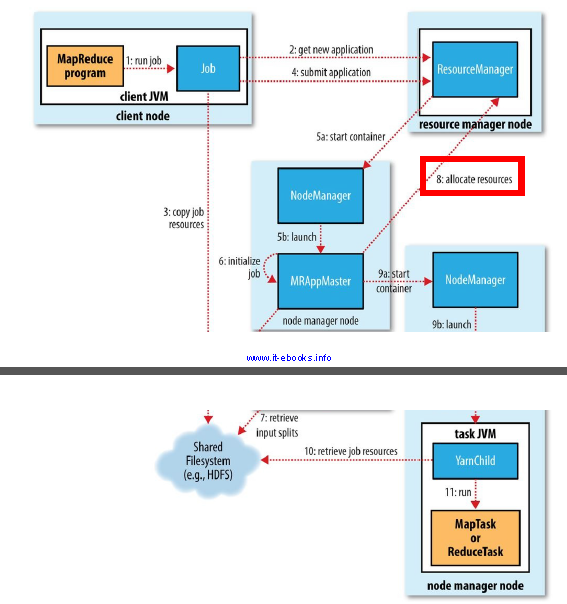
对于MapReduce jobs ,application master他是一个Java 应用程序，他的主类是MRAppMaster，初始化job包括创建一个bookkeeping对象的成员去不断追踪作业的运行（step 6），然后，application master他会从共享文件系统中获取客户端已经计算好的输入分片（step 7），为每一个分片创建一个map task object，reduce task objects的数量由mapreduce.job.reduces属性决定（这个属性通过调用job的setNumReduceTasks（）方法），Tasks在这个地方被赋予一个ID

Application master 必须决定如何运行这些构成MapReduce的作业的tasks，如果这个job是非常小的，application master 将会在自己所在的JVM中运行这些tasks，这种情况发生在：如果application master 判断分发这些任务到新的containers，并行运行这些任务，比顺序运行这些任务在一个节点上要消耗更多，这样的任务被称作为uberized，或者就像运行一个uber task。

什么样的标准才算是一个小的job？默认情况下，一个小job，他的mapper个数小于10个，仅仅只有一个reduce，并且他输入的文件大小小于一个HDFS 块的大小（笔记：这些值可以通过设置一个job的mapreduce.job.ubertask.maxmaps,mapreduce.job.ubertask.maxreduces,mapreduce.job.ubertask.maxbytes来进行改变），Uber tasks 必须明确的通过设置mapreduce.job.ubertask.enable为true来启用。

最后，当所有的task在可以运行的时候，application master 会在OutputCommitter上调用setupJob（）方法，OutputCommitter默认是FileOutputCommitter,他会为job的最终输出创建一个最终的输出目录文件夹，并且会创建task output的临时工作目录空间。

任务的分发

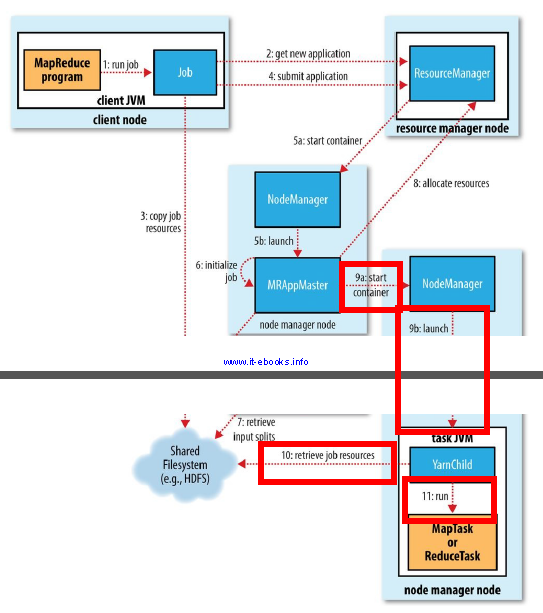


如果一个job不能作为uber task来运行，application master将会从resource manager所有的map和reudce任务请求containers（step 8），首先请求map tasks所需要的containers，因为它比reduce task拥有更高的优先权，reduce的排序解析开始之前，所有的map任务都必须完成。直到map tasks的5%已经运行完成，reduce才能够启动（慢启动）。

Reduce tasks能够运行在 集群的任何地方，但是，对于map任务，scheduler会考虑数据的本地性约束，在最理想的情况下，任务是数据本地化的，也就是任务运行在输入分片所在的节点上。同样，任务也可能是机架本地化的（rack local），任何输入分片在同一个机架，但不在同一个节点上，一些任务既不是数据本地化，也不是机架本地化，而是从与他们自身运行的不同机架上检索数据。可以通过查看作业的计数器得知每类任务的比例。

对于tasks，请求也包含了内存和cpu的需求，默认情况下，每一个map和reduce被分配1024M的内存，一个cpu，一个job的这些值的配置通过以下属性进行配置：mapreduce.map.memory.mb,mapreduce.reduce.memory.mb,mapreduce.map.cpu.vcores,mapreduce.reduce.cpu.vcores。

任务的执行



一旦resource manager 的scheduler为一个task在一个节点上分配了一个container，application master联系node manager来启动这个container（step 9a和step9b），这个任务被一个java应用程序来执行，他的主类是YarnChild。在运行任务之前，他会下载task需要的资源，包括job 的configuration ，JAR file等其他在分布式缓存中的文件（step 10），最后，运行map或者reduce task（step 11）。

YarnChild运行在一个专用的JVM中，所以出现在用户自定义的map或者reduce函数中任何bugs不会影响node manager，例如导致崩溃或者挂起等。

每一个task 都能执行setup和commit actions，它运行在task自己的JVM中，并有作业的OutputCommitter确定。Commit动作用于把task的临时输出目录移动到最终的输出目录，提交协议确保当推理执行（speculative execution）可用时，只有一个任务副本被提交，其他的都将被取消。

Streming

略

进度和状态的更新

MapReduce 作业是长时间运行的批量作业，运行时间范围从数分钟到数小时。这是一个很长的时间段，所以对于用户而言，能够得知作业进展 是很重要的。一个作业和他的每个人物都有一个状态，包括：作业或任务的状态（比如，运行状态，成功状态，失败状态），map和reduce的仅需，作业计数器的值，状态消息或描述（可以由用户代码来设置）。这些状态信息在作业运行期间是不断改变的，他们是如何与客户端通信的呢？

任务在运行期间，对其进度（progress，即任务完成的百分比）保持跟踪。对map任务，任务进度是已经处理输入所占的比例，对于reduce任务，情况稍微复杂点，但系统仍然会估计已处理reduce输入的比例。整个过程分成三部分，与shuffle的三个阶段相对应。比如，如果任务已经执行reducer一半的输入，那么任务的进度便是5/6。因为已经完成复制和排序阶段（每个占1/3），并且已经完成reduce阶段的一半（1/6）

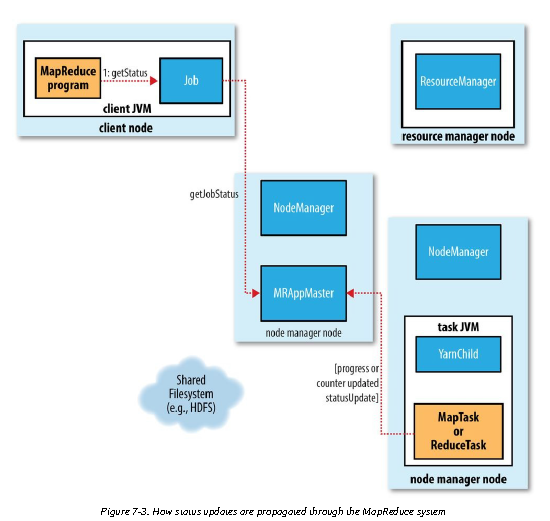
任务也有一组计数器，负责对任务运行过程中各个事件进行计数，这些计数器要么内置于框架中 ，例如已写入的map输出记录数，要么用户自己定义。

在map或者reduce任务运行的时候，任务每三秒钟通过umbilical接口向application master汇报进度和状态（包含计数器），作为作业的汇聚视图。

资源管理器的web UI展示了正在运行的应用以及连接到对应的application master，每个application master展示MapReduce作业的进度等进一步细节。

在作业运行期间，客户端每秒查询application master 来接受最新的状态（这个间隔可以通过mapreduce.client.progressmonitor.pollinterval进行设置），客户端也可以使用job的getStatus（）方法来得到一个JobStatus的实例，后者包含作业的所有状态信息。

下图对方法调用进行了图解



作业的完成

当application master接收到最后一个任务已完成的通知后，便把状态设置为“成功”，然后，在job状态时，便知道任务已经完成，于是job打印一条信息告诉用户，然后从waitForCompletion（）方法返回。Job的统计信息和计数值也在这时输出到控制台。

如果application master有相应的设置，也会发送一个HTTP作业通知。希望收到毁掉指令的客户端可以通过mapreduce.job.end-notification.url属性来进行这项设置。

最后，作业完成后，application master和任务容器清理其工作状态（内部的输出将被删除），OutputCommitter的作业清理方法会被调用。作业历史服务器保存作业的信息供用户需要时查询。

失败

在实际情况下，用户代码错误不断，进程崩溃，机器故障等等，使用hadoop最主要的好处之一是它能够处理此类故障并让你能够完成作业。

任务运行失败

首先考虑子任务失败的情况。最常见的情况是map任务或reduce任务中的用户代码抛出运行异常。如果发生这种错误，子任务JVM进程会在退出之前向application master 发送错误报告，错误报告最后被记入用户日志。Application master 将此任务尝试标记为失败，释放container以便运行另外一个任务。

对于Streaming 任务，如果Streaming进程以非零退出代码退出，则被标记为failed。这种行为由stream.non.zero.exit.is.failure属性来控制（默认值为true）

另一种错误情况是子进程JVM突然退出，可能由于JVM软件缺陷而导致MapReduce用户代码由于某些特殊原因造成JVM退出。在这种情况下，node manager注意到进程已经退出然后报告给application master以便application master将此任务尝试标记为失败。

任务挂起的处理方式则有不同。一旦application master注意到已经有一段时间没有收到进度的更新，便会将任务标记为failed。在此之后，JVM子进程将被杀次。任务失败的超时间隔通常为10分钟，可以以作业为基础（或以集群为基础）将mapreduce.task.timeout属性设置为毫秒为单位的值。

超时设置为0将关闭超时判定，所以长时间运行的任务永远不会被标记为failed。在这种情况下，被挂起的任务永远不会释放他的container，随着时间的推移最终降低整个集群的效率。因此，尽量避免这种设置，同时充分确保每个任务能够定期汇报其进度。

Application master被告知一个任务尝试失败后，将重新调度任务的执行。Application master会尝试避免重新调度以前尝试失败的node manager上的任务。此外，一个任务的失败次数超过4次，将不会再重试。这个值是可以设置的：对于map任务，运行任务的最多尝试次数又mapreduce.map.maxattemptes属性控制；而对于reduce任务，则由mapreduce.reduce.maxattempts 属性控制。在默认情况下，如果任何一个任务失败次数大于4次（或配置的最大重试次数），整个作业都会失败。

对于一些应用程序，我们不希望一点有少数几个任务失败就中止运行这个作业，因为即使有任务失败，作业的一些结果可能还是可用的。在这种情况下，可以为作业设置在不触发作业失败的情况下允许任务失败的最大百分比。Map任务和reduce任务可以独立控制 ，分别通过mapreduce.map.failures.maxpercent和mapreduce.reduce.failures.maxpercent属性来设置。

任务尝试也是可以被killed，但与失败不同，任务尝试可以被中止是因为他是一个推测副本，或因为他所处的node manager失败，导致node manager将它上面运行的所有任务尝试标记为killed。被中止的任务尝试不会被计入任务运行尝试次数（由mapreduce.map.maxattempts和mapreduce.reduce.maxattempts设置）,因为尝试中止并不是任务的过错。

用户也可以使用web UI 或命令行方式（输入hadoop job 查看相应的选项）来中止或取消任务尝试。也可以采用相同的机制来中止作业。

Application Master失败

就像MapReduce任务可以进行多次尝试来使任务成功，在YARN application也被重试。最大的application master的重试次数由mapreduce.am.max-attempts 属性进行设置，默认为2次，所以在失败重试2次之后任务将会失败。

YARN指定在集群中application master重试次数，个人的应用程序不能超过这个限制，这个限制由yarn.resourcemanager.am.max-attemptes，默认是2，所以你想增加 MapReduce application master的重试次数，也必须增加此参数的设置。

恢复工作的流程如下：一个application master 定期发送一个心跳给resource manager，当application master失败的时候，resource manager将会感触到application master已经失败，然后在一个新的container中重新启动一个application master，在这种情况下，applicaition master将会通过作业历史恢复任务的运行状态，恢复默认是开启的，但是可以通过设置yarn.app.mapreduce.am.job.recovery.enable为false来关闭。

客户端从application master获取进度报告，如果application master失败，客户端需要获取新的实例，在job初始化期间，客户端向resource manager获取新的application master的地址，然后缓存起来以便不用每次都需要重新向resource manager获取applicaition的地址，但是，如果 application master失败，此时客户端向applicaition master获取最新状态将会超时，此时，客户端将返回resource manager去询问一个新的application master的新的地址，这些都用户来说都是透明的。

Node Manager 失败

如果node manager失败，就会停止向resource manager发送心跳（或者发送时间间隔时间很长），并从resource manager的节点资源管理器池移除，默认值10分钟的属性yarn.resourcemanager.nm.lineness-monitor.expiry-interval-ms决定着resource manager认为node manager失败之前的等待时间。

如果应用程序的运行失败次数过高，那么node manager可能会被拉黑，即使node manager本身没有失败。对于MapReduce，如果一个node manager上有三个任务失败，application master就会尽量将任务调度到不同的节点上。用户可以通过mapreduce.job.maxtaskfailures.per.tracker设置该属性。

注意：resource manager不执行拉黑操作，因此新作业中的任务可能被调度到故障节点上，即使这些故障节点已经被运行早期作业的application master 拉黑。

Resource manager失败

Resource manager失败是非常严重的，因为没有它，任何作业或者任务都不能运行。默认配置下，resource manager 是单节点故障的，只要这个节点的机器出现故障，所有正在运行的任务都会失败，而且不能恢复。

实现高可用，在配置中配置并运行两个resource manager 是非常必要的，如果active resource manager失败，standby resource manager代替他，对于客户端没有任何停止。

正在运行的作业的信息都被存储在高可用的存储设备上（ZooKeeper或者HDFS上），所以standby 能够恢复active失败前主要的状态。Node manager的信息没有被存储起来，原因就是当node manager发送心跳给新的resource manager的时候，resource manager可以快速重建node manager。

当新的reource manager启动的时候，他从存储系统中应用程序的信息，然后重新启动所有正在集群中运行的所有的application master，他不会把应用程序计数到失败重试次数中（所以他不会被yarn.resourcemanager.am.max-attempts计数加一），因为他不是由于应用程序代码错误，而是由于系统杀死的，in practice,the application master restart is not an issue for MapReduce applications since they recover the work done by completed tasks

Resource manager 从standby转变为active是由failover controller 处理的。Failover controller默认是自动的，他使用ZooKeeper选举机制来保证同一时间只有一个resource manager，不用于HDFS 的HA，这个failover controller不是运行在一个单独的进程中，它默认是嵌入在resource manager中的，也可以同时配置多个failover ，但是不推荐。

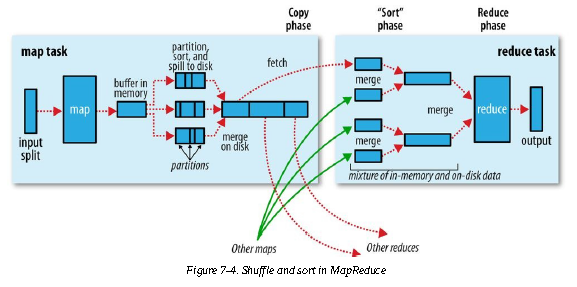
客户端和node manager必须配置来处理resource manager，因为现在有两个resource manager，他们尝试去尝试去连接着两个resource manager去发现active resource manager。如果active resource manager出现故障，当standby 变成active后去尝试连接active resource manager。

Shuffle and Sort

MapReduce确保每个reducere的输入都是按键进行排序的。系统执行排序----将map的输出作为输入传递给reducer----他被称作shuffle。这本节，我们将学习shuffle是如何工作的，因为它有助于我们理解工作机制（如果需要优化MapReduce程序）。Shuffle输入不断被优化和改进的代码库的一部分，因此，下面的描述有必要隐藏一些细节，从许多方面来看，shuffle是MapReduce的“心脏”，是奇迹发生的地方。

The Map Side

Map函数开始产生输出时，他不是简单地将它写到磁盘。这个过程更复杂，他利用缓冲的方式写到内存并且处于效率的考虑进行预排序，下图展示了该过程：



每个map任务都有一个环形内存缓冲区用于存储任务输出。在默认情况下，缓冲区的大小为100MB，此值可以通过改变mapreduce.task.io.sort.mb属性来调节。一旦缓冲内容达到阈值（mapreduce.map.sort.spill.precent,默认为0.8或者80%），一个后台线程便开始把内容溢出到（spill）磁盘。在溢出写到磁盘过程中，map输出继续写到缓存区，但如果在此期间缓冲区被填满，map会被阻塞直到磁盘过程完成。溢出写过程按轮询方式将缓冲区中的内容写到mapreduce.cluster.local.dir属性指定的作业特定子目录的目录中。

在写磁盘之前，线程首先根据数据最终要传的reducer把数据划分成相应的分区（partition）。在每个分区中，后台线程按键在内存中进行排序，如果有一个combiner，他就在排序后的输出上运行。运行combiner使得map输出结果更紧凑，所以减少写到磁盘的数据传递给reducer的数据。

每次内存缓冲区达到溢出阈值，就会新建一个溢出文件（spill file），一次在map任务完成其最后一个输出记录之后，会有几个溢出文件。在任务完成之前，溢出文件被合并成一个已分区且排序的输出文件。配置属性mapreduce.task.io.sort.factor控制着一次最多能合并多少流，默认是10.

如果至少存在3个溢出文件（通过mapreduce.map.combine.minspills属性设置）时，则combiner就会在输出文件写道磁盘之前再次运行。前面曾讲过，combiner可以在输入上反复运行，但不影响最终结果。如果只有一或者两个溢出文件，那么对map输出的减少方面不值得调用combiner，不会为该map输出再次运行combiner。

在将压缩map输出写到磁盘的过程中对它进行压缩往往是个很好的主要，因为这样会写磁盘的速度更快，节约磁盘空间，并且减少传给reducer的数据量。在默认情况下，输出是不压缩的，但只要将mapreduce.map.output.compress设置为true，就可以轻松启用此功能。使用的压缩库由mapreduce.map.output.compression.codec指定 ， Hadoop的压缩codec



Reducer通过HTTP方式得到输出文件的分区。用于文件分区的工作线程的数量由mapreduce.shuffle.max.threads属性控制。此设置针对的是每一个node manager，而不是针对每个map任务。默认为0表示是处理器数量的两倍

The Reducer Side

现在转到处理过程的reduce部分。map输出文件位于map任务的机器的本地磁盘（尽管map输出经常写到map 运行的本地磁盘，但reduce输出并不是这样），现在，机器要为分区文件运行reduce任务。而且，reduce任务需要集群上若干个map任务的map输出作为其特殊的分区文件。每个map任务的完成时间可能是不同的，因此只要有一个任务完成，reduce任务就开始复制其输出。这就是reduce任务的复制阶段。Reduce任务有少量复制线程，因为能够并行取得map输出。默认值是5个线程，但这个默认值可以通过设置mapreduce.reduce.shuffle.parallel.copies属性来改变。

Reduce如何知道要从哪台机器取得map输出呢？

Map任务完成后，他们会通过心跳通知application master。因此，对已指定作业，application master知道map的输出和hosts之间的映射关系。Reducer中的一个现成定期询问application master以便获取map输出的位置，知道获得所有输出位置。

由于第一个reducer可能失败，因此node manager并没有在第一个reducer线索到map输出时就立即从磁盘上删除它们。相反，node manager会等待，知道resource manager告诉它删除map输出，这是作业完成后执行的。

如果map输出相当小，会被复制到reduce任务JVM的内存（缓冲区大小由mapreduce.reduce.shuffle.input.buffer.percent属性控制，它用于指定用于此用途的堆空间的百分比），否则，map输出被复制到磁盘。一旦内存缓冲区大爱道阈值大小（由mapreduce.reduce.shuffle.merge.percent决定）或达到map输出阈值（由mapreduce.reduce.merge.inmem.threshold控制），则合并后溢出写到磁盘中。如果指定combiner，则在合并期间运行它以降低写入磁盘的数据量。

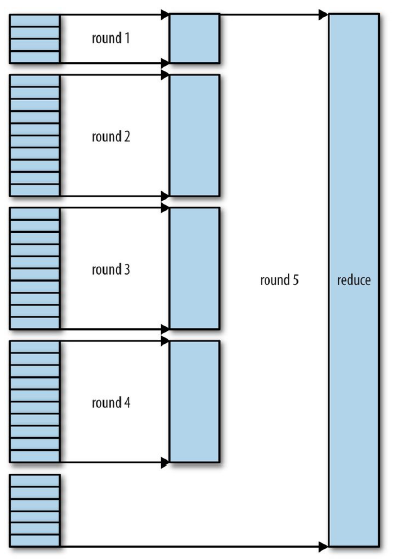
随着磁盘上副本增多，后台线程会将他们合并为更大的，排好序的文件，这会为后面的合并节省一些时间。注意，为了合并，压缩的map输出（通过map任务）都必须在内存中被解压缩。

复制完所有map输出后，reduce任务进入排序阶段（更恰当的说法是合并阶段，因为排序实在map端进行的），这个阶段将合并map输出，维持其顺序排序。这是循环进行的。比如，如果有50个map输出，而合并因子是10（10为默认设置，有io.sort.factor属性设置，与map的合并类似），合并将进行5趟。每趟将10个文件合并成一个文件，因此最后由5个中间文件。

在最后阶段，即reduce阶段，直接把数据输入reduce函数，从而省略了一次磁盘往返行程，并没有将这5个文件合并成一个已排序的文件作为最后一趟。最后的合并可以来自内存和磁盘片段。

每趟合并的文件数实际上比示例中展示有所不同。目标是合并最小数量的文件以便满足最后一趟的合并系数。因此如果有40个文件，我们不会再四趟中每趟合并10个文件从而得到4个文件。相反，第一趟只合并4个文件，随后的三趟合并所有10个文件。在最后一趟中，4个已合并的文件和余下的6个（未合并的）文件合计10个文件，该过程如下图所示，

注意，这并没有改变合并次数，他只是一个优化措施，目的是尽量减少写到磁盘的数据量，因为最后 一趟总是直接合并到reduce。



在reduce阶段，reduce函数被强制把其输出按照key进行排序。并且数据时直接输出到文件系统，典型的是HDFS，在HDFS下，由于node manager是运行在一个datanode上，所以第一个副本会被写入到本地磁盘中。

Configuration Tuning

现在我们已经有比较好的基础来理解如何调优shuffle过程，过程来提高MapReduce性能。下面两张表中总结了相关设置和默认值，这些设置以作业为单位，默认值适用于常规作业。

|  |  |  |  |
| --- | --- | --- | --- |
| Property name | Type | Default value | Description |
| Mapreduce.task.io.sort.mb | Int | 100 | 排序map输出时所用的内存缓冲区的大小，以M为单位 |
| Mapreduce.map.sort.spill.precent | Float | 0.80 | Map输出内存缓冲和用来开始磁盘溢出写过程的记录边界索引，这两者使用比例的阈值 |
| Mapreduce.task.io.sort.factor | Int | 10 | 排序文件时，一次最多合并流数。这个属性也在reduce中使用。将此值增加到100是很常见的 |
| Mapreduce.map.combine.minspills | Int | 3 | 运行combiner所需要的最小溢出文件数（如果已指定combiner） |
| Mapreduce.map.output.compress | Boolean | False | 压缩map输出 |
| Mapreduce.map.output.compress.codec | Class name | Org.apache.hadoop.io.compress.DefaultCoder | 用于map输出的压缩编码解码器 |
| Mapreduce.shuffle.max.threads | Int | 0 | 每个node manager的工作线程数，用于将map输出到reducer。这是集群范围的设置，不能有单个作业设置。0意味着是可用precessors的两倍 |

|  |  |  |  |
| --- | --- | --- | --- |
| Property name | Type | Default value | Description |
| Mapreduce.reduce.shuffle.parallelcopies | Int | 5 | 用于把map输出复制到reducer的线程数 |
| Mapreduce.reduce.shuffle.maxfetchfailures | Int | 10 | The number of times a reducer tries to fetch a map output before reporting the error |
| Mapreduce.task.io.sort.factor | Int | 10 |  |
| Mapreduce.reduce.shuffle.input.buffer.percent | Float | 0.70 | 在shuffle的复制阶段，分配给map输出的缓冲区占堆空间的百分比 |
| Mapreduce.redude.shuffle.merge.percent | Float | 0.66 | Map输出缓冲区（由mapreduce.reducer.shuffle.merge.percent定义）的阈值使用比例，用于启动合并输出和磁盘溢出写的过程 |
| Mapreduce.reducer.merger.inmem.threshold | Int | 1000 | 启动合并输出和磁盘易出血过程的map输出的阈值数。0或者更小的数意味着没有阈值限制，溢出写行为由mapreduce.reduce.shuffle.merge.percent |
| Mapreduce.reduce.input.buffer.percent | Float | 0.0 | 在reduce过程中，在内存中保存map输出的空间占整个堆空间的比例。Reduce阶段开始时，内存中的map输出大小不能大于这个值。默认情况下，在reduce任务开始之前，所有map输出都合并到磁盘上，以便为reducer提供尽可能多的内存，然而，如果reducer需要的内存较少，可以增加此值来最小化访问磁盘的数量 |

总的原则是给shuffle过程尽可能多的内存。然而，有一个平衡问题，也就是要确保map函数和reduce函数能得到足够的内存来运行。这就是为什么编写map函数和reduce函数时尽量少用内存的原因，它们不应该无限使用内存（例如，应避免在map中堆积数据）。

运行map任务和reduce任务的JVM，其内存大小由mapreduce.child.java.opts属性设置。任务节点上的内存大小应该尽量大。

在map段，可以通过避免多次溢写出磁盘来获取最佳性能；一次是最佳的情况。如果能估算map输出大小。就可以合理地设置mapreduce.task.io.sort.\*属性来尽可能减少溢出写的次数。具体而言，如果可以，就要增加mapreduce.task.io.sort.mb的值。MapReuce计数器计算在作业运行整个阶段中溢出写磁盘的记录数，这对于调优很有帮助。注意，这个计数器包括map和reduce两端的溢出写。

在reduce段，中间数据全部驻留在内存时，就能获得最佳性能。在默认情况下，这是不可能发生的，因为所有内存一般都预留给reduce函数。但如果reduce函数的内存需求不大，把mapreduce.reduce.merge.inmen.threshold设置为0，把mapreduce.reduce.input.buffer.percent设置为1.0（或者一个更低的值）就可以提升性能。

2008年4月，Hadoop在通过TB字节排序基准测试中获胜，他使用了一个优化方法，即将中间数据保存在reduce端的内存中。

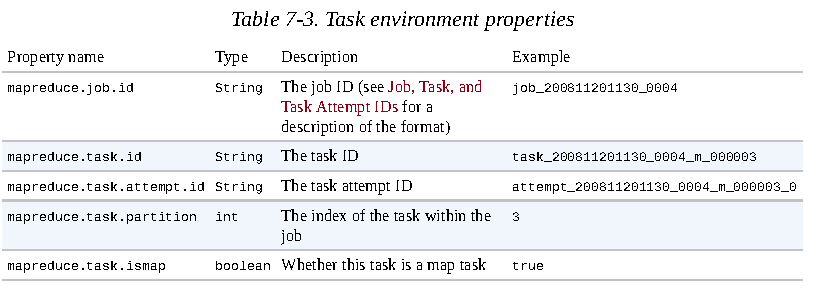
更常见的情况是，Hadoop使用默认为4KB的缓冲区，这是很低的，因此应该在集群中增加这个值io.file.buffer.size.

Task Execution

前面讲了MapReduce系统是如何工作的，在本节，我们将了解MapReduce用户对任务执行的更多控制。

任务执行环境

Hadoop为map任务或reduce任务提供运行环境相关信息。例如，map任务可以知道它处理的问价的名称，map任务或reduce任务可以得知任务的尝试次数。下表的属性可以从作业的配置信息中获取，通过为mapper或者reduer提供一个configure（）方法实现（其中，配置信息作为参数进行传递），便可获得这一信息。在新版本的API中，这些属性可以从传递给Mapper或Reducer的所有方法的相关对象中获取。



推测执行（Speculative Execution）

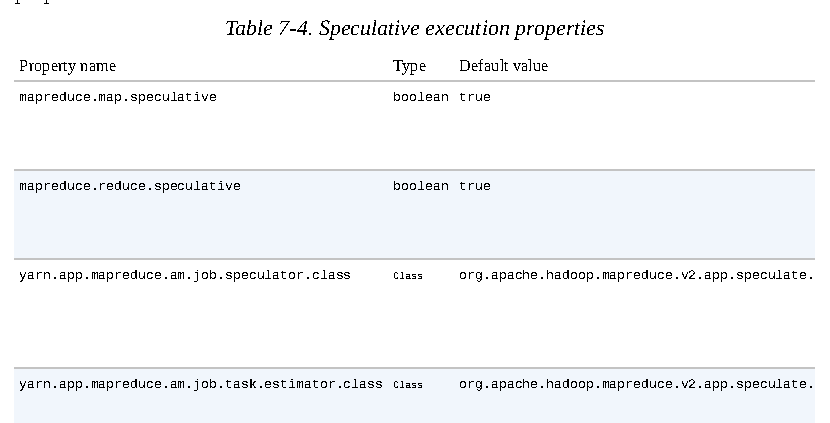
MapReduce模型将作业分解成任务，然后并行地运行任务以使作业的整体执行时间少于各个任务顺序执行的时间。这使作业执行时间对运行缓慢的任务很敏感，因为只运行一个缓慢的任务会使整个作业所用的时间远远大于执行其他任务的时间。当一个作业由几百或几千个任务组成时，可能出现少数“拖后腿”的任务，这是很常见的。

任务执行缓慢可能有很多原因，包括硬件老化或软件配置错误，但是，检测具体原因很困难，因为任务总能够成功完成，尽管比预计执行时间长。Hadoop不会尝试诊断或修复执行慢的任务，相反，在一个任务运行比预期慢的时候，他会尽量检测，并启动另一个相同的任务作为备份。这就是所谓的任务的“推测执行”（speculative execution）

必须认识到一点：如果同时启动两个重复的任务，他们会互相竞争，导致推测执行无法工作。这对集群资源是一种浪费。相反，只有在一个作业的所有任务都启动之后在启动推测任务，并且只针对那些已运行一段时间（至少一分钟）且比作业中其他任务平均进度慢的任务。一个任务成功完成后，任何正在运行的重复任务都被中止，因为已经不再需要它们了。因此，如果原任务在推测任务前完成，推测任务就会被中止；同样，如果推测任务先完成，那么缘任务就会被中止。

推测任务是一种优化策略，它并不能使作业的运行更可靠。如果有一些软件缺陷会造成任务挂起或运行速度缓慢，依靠推测执行来避免这些问题显然是不明智的，并且不能可靠地运行，因为效果南通的软件缺陷可能会影响推测式任务。应该修复软件缺陷，式任务不会挂起或运行速度缓慢。

在默认情况下，推测执行是启动的。可以基于集群或基于每个作业，单独为map任务和reduce任务启用或禁用该功能。相关的属性如下表。



为什么会想到关闭推测执行？推测执行的目的是减少作业执行时间，但这是以集群效率为代价的。在一个繁忙的集群中，推测执行会减少整个的吞吐量，因为冗余任务的执行时会减少作业的执行时间。因此，一些集群管理者倾向于在集群上关闭此选项，而让用户更具各别任务需要而开启该功能。Hadoop老版本尤其如此，因为在调度推测任务时，会过度使用推测执行方法。

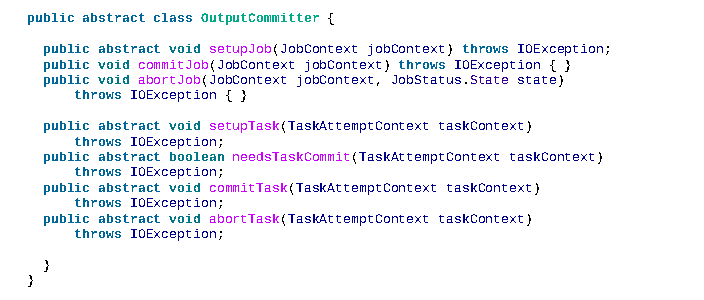
对于reduce任务，关闭推测执行是有益的，因为任意重复的reduce任务都必须将获取map输出作为最先的任务，这可能会大幅度地增加集群上的网络传输。

关闭推测执行的另一种情况是考虑到非幂等的任务。然而在很多情况下，将任务携程幂等的并使用OutputCommitter来提升任务成功时输出到最后位置的速度，这是可行的 。

关于OutputCommittters

Hadoop MapReduce使用一个提交协议来确保作业和任务都完全成功或失败。这个行为通过对作业使用OutputCommitter来实现，在老版本MapReduce API中通过调用JobConf的setOutputCommitter（）或配置中的mapred.output.committer.class来设置。在新版本的MapReduce API中，OutputCommitter由OutputFormat通过它的getOutputCommitter（）方法确定。默认值为FileOutputCommitter，这对基于文件的MapReduce是适合的。可以定制已有的OutputCommitter或者在需呀对作业或任务进行特殊的安排或清理时，甚至还可以写一个新的实现。

OoutputCommitter 的API如下所示（在新旧版本中的MapReduce API中）



SetupJob（）方法在作业运行前被调用，通常用来执行初始化操作，FileOutputCommitter（）方法通常为任务创建最后的输出目录${mapreduce.outout.fileoutputformat.outputdir}，以及一个临时的工作空间。

如果作业成功，就调用commitJob（）方法，在默认的基于文件的实践中，它用于删除临时的工作空间并在输出目录中创建一个\_SUCCESS的隐藏的标志文件，一次告知文件系统的客户端该作业成功完成了。如果作业不成功，就通过状态对象调用abortJob（），意味着该作业是否失败或终止（例如用户终止）。在默认的实现中，将删除作业的临时工作空间。

在任务级别上的操作与此类似。在任务执行之前先调用setupTask（）方法，默认的实现不做任何事情，因为针对任务输出命名的临时目录是在写任务输出的时候被创建。

任务的提交阶段是可选的，并通过从needsTaskCommit（）返回的false值关闭它。这使得执行框架不必为任务运行分布提交协议，也不需要commitTask（）或者abortTask（）。当一个任务没有写输出时，FileOutputCommitter将会跳过提交阶段。

执行框架保证特定任务在有很多任务尝试的情况下只有一个任务会被提交，其他的则被取消。这种情况是可能出现的，因为第一次尝试处于某个原因而失败（这种情况下将被取消），提交的是稍后成功的尝试。另一种情况是如果两个任务尝试作为推测副本同时运行，则提交先完成的，而另一个被取消。

任务附件文件

对于map任务或reduce任务的输出，常用的写方法是通过OutputCollector来收集键/值对。有一些应用需要比单个键/值对模式更灵活的方式。因此直接将map或reduce任务的输出文件写道分布式文件系统，如HDFS。

注意，要确保同一个任务的多个实例不向同一个文件进行写操作。如前所述，OutputCommitter协议解决了该问题。如果应用程序将附属文件导入其任务的工作目录中，那么成功完成的这些任务就会将其附属文件自动推送到输出目录，而失败的任务，其附属任务也被删除。

任务通过从其配置文件查询mapreduce.task.output.dir属性属性值找到其工作目录。另一中方法，使用Java API的MapReduce程序可以调用FileOutputFormat上的getWorkOutputPath（）静态方法获取描述工作目录的Path对象。执行框架在执行任务之前首先会创建工作目录，因此不需要我们创建。

举一个例子，假设有一个程序用来转换图片文件的格式。一种实现方式是用一个只有map任务的作业，其中每个map执行一组药转换的图片。如果map任务把转换后的图像写到其工作目录，那么在任务完成之后，这些图像就会被传到输出目录。