# 1.1 MapReduce 远程调试&作业调优&工作流

## 1.1.1 远程调试

1.当一个**MapReduce任务失败并且没有足够多的记录信息来诊断错误时**，可以选择用调试器运行该任务。在集群上运行作业时，很难使用调试器，因为不知道哪个节点处理哪部分输入，所以不能在错误发生之前安装调试器。然而，下面方法可以使用：

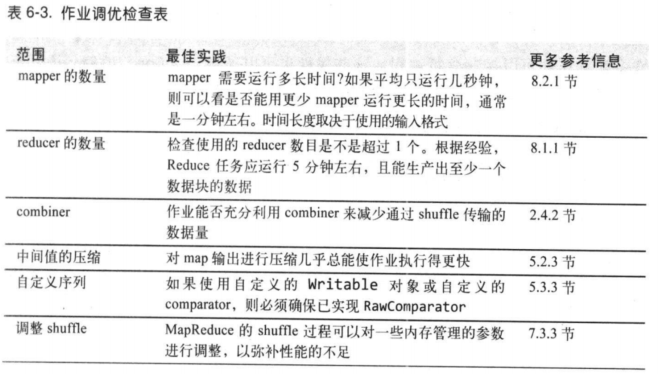
* **在本地重新产生错误**：对于特定的输入，失败的任务通常总会失败。可以尝试通过下载致使任务失败的文件到本地运行重现问题，这可以使用到调试器。
* **使用JVM调试选项**：失败的常见原因是任务JVM中Java的内存溢出。可以讲mapred.child.java.opts设为包含-XX:HeapDumpOnOutOfMemoryError-XX:HeapDumpPath = /path/to/dumps。该设置将产生一个堆转储，这可以通过jhat或Eclipse Memory Analyzer这样的工具来检查。注意，JVM选项应当添加到由mapred.child.java.opts指定的已有内存设置中。
* **使用任务分析**：Java的profiler提供了很多JVM的内部细节，Hadoop提供了分析作业中部分任务的机制。

2.在一些情况下**保存失败的任务尝试的中间结果文件对于以后的检查是很有用的**，特别是在任务工作路径中建立转储或配置文件。

3.也可以保存成功任务的中间结果文件，以便解释任务没有失败。这是，将属性mapreduce.task.files.preserve.filepattern 设置为一个正则表达式。

## 1.1.2 作业调优(重点)

1.在开始任务级别的分析或优化之前，必须仔细研究下表所示的检查内容。



## 1.1.3 MapReduce的工作流

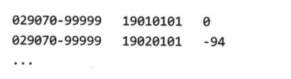
下面给出了一个例子，将这个复杂问题转换为MapReduce的工作流。

1.假设我们想找到每个气象台**每年每天的最高气温记录的均值**。例如，要计算029070~99999气象台的1月1日的每日最高气温的均值，我们将从这个气象台的1901年1月1日，1902年1月1日，直到2000年的1月1日的气温中找出每日最高气温的均值。

2.MapReduce计算要分为以下两个阶段：

1. 计算每对station-date的每日最高气温。该例子中的MapReduce程序是最高气温程序的变种，不同之处在于本例中的键是一个综合的station-date对，而不只是年份
2. 计算每个station-day-month键的每日最高气温的均值。mapper从上一个作业得到的输出记录(station-date，最高气温值)，丢掉年份部分，将其值投影到记录(station-day-month，最高气温值)。然后reducer为每个station-day-month键计算最高气温值的均值。

3.第一阶段的输出看上去就是我们想要的气象台的信息。范例中的mean\_max\_daily\_temp.sh脚本提供了Hadoop Streaming 的一个实现:



4.前两个字段形成键，最后一列是指定气象台和日期所有记录中的最高气温。第二阶段计算这些年份中每日最高气温的平均值:

image.png

5.以上是气象台029070~99999整个世纪中1月1日的日均最高气温为-6.8℃。

6.**一个作业可以包含多个(简单的)MapReduce步骤**，这样整个作业由多个可分解的、可维护的mapper和reducer 组成。本书第V部分提到的一些实例学习包括使用MapReduce来解决的大量实际问题，在每个例子中，数据处理任务都是使用两个或更多MapReduce作业来实现的。对于理解如何将问题分解成MapReduce 工作流，第V部分所提供的详细介绍非常有价值。

7.相对于我们已经做的，mapper和 reducer 完全可以进一步分解。**mapper一般执行输入格式解析、投影(选择相关的字段)和过滤(去掉无关记录)**。在前面的mapper中，我们在一个mapper 中实现了所有这些函数。然而，还可以将这些函数分割到不同的 mapper，然后使用 Hadoop自带的 ChainMapper类库将它们连接成一个mapper。结合使用ChainReducer，你可以在一个MapReduce作业中运行一系列的mapper，再运行一个reducer和另一个mapper链。

## 1.1.4 关于JobControl

1.当MapReduce工作流中的作业不止一个时，问题随之而来:如何管理这些作业按顺序执行?有几种方法，其中主要考虑是否有一个线性的作业链或一个**更复杂的作业有向无环图**(DAG，directed acyclic graph)。

2.对于线性链表，**最简单的方法是一个接一个地运行作业**，等前一个作业运行结束后再运行下一个：

image.png

3.如果一个作业失败，**runJob()方法就抛出一个IOException**，这样一来，**管道中后面的作业就无法执行**。根据具体的应用程序，你可能想捕获异常，并清除前一个作业输出的中间数据。

4.这种方法类似新的MapReduce API，**除了需要Job上的waitForCompletion()方法的布尔返回值：true表示作业成功，而false表示失败**。

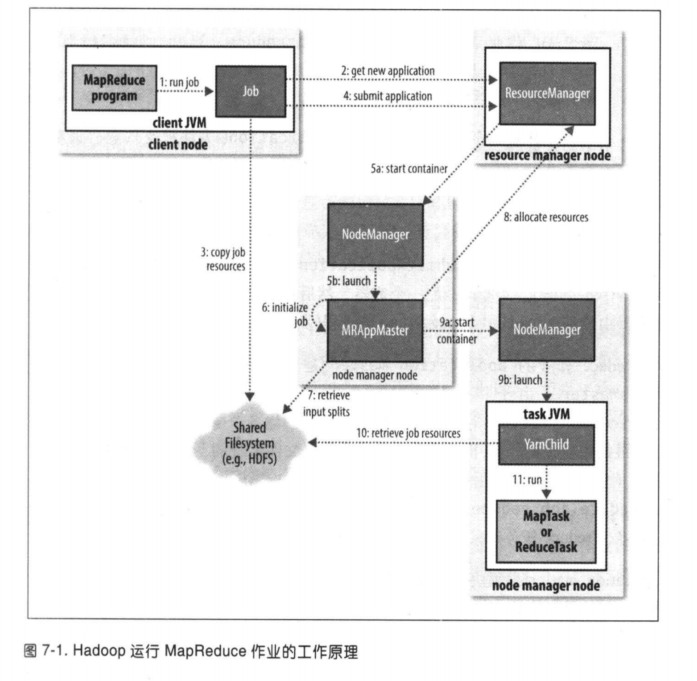
5.对于比线性链表更复杂的结构，有相关的类库可以帮助你合理安排工作流。它们也适用于线性链表或一次性作业。最简单的是 org.apache.hadoop.mapreduce.jobcontrol包中的JobControl类。JobControl的实例表示一个**作业的运行图**，你可以加入作业配置，然后告知JobControl实例作业之间的依赖关系。在一个线程中运行JobControl时，它将按照依赖顺序来执行这些作业。也可以查看进程，在作业结束后，可以查询作业的所有状态和每个失败相关的错误信息。如果一个作业失败，Jobcontrol将不执行与之有依赖关系的后续作业。

# 1.2 MapReduce 工作机制(重点)

1.可以通过一个**简单的方法调用来运行MapReduce作业**：**Job对象的submit()方法**。注意，**也可以调用waitForCompletion()，它用于提交以前没有提交过的作业，并等待它的完成**。submit()方法调用封装了大量的处理细节。

2.Hadoop运行MapReduce的工作原理如下图。在最高层，有以下5个独立的实体。

* 客户端，提交MapReduce作业
* **YARN资源管理器**，负责协调集群上计算机资源的分配
* **YARN节点管理器**，负责启动和监视集群中机器上的计算容器(container)
* **MapReduce的application master**，**负责协调运行MapReduce作业的任务**。它和MapReduce任务在容器中运行，这些容器由资源管理器分配并由节点管理器管理
* 分布式文件系统(一般为HDFS)，用来与其他实体间共享作业文件



## 1.2.1 作业的提交

1.**Job的submit()方法创建一个内部的JobSummiter实例**，并且调用其submitJobInternal()方法(图步骤1)。提交作业后，**waitForCompletion()每秒轮询作业的进度，如果发现自上次报告后有改变，便把进度报告到控制台**。作业完成后，如果成功，**就显示作业计数器**；**如果失败，则导致作业失败的错误记录到控制台**。

2.JobSummiter所实现的作业提交过程如下所述：

* **向资源管理器请求一个新应用ID，用于MapReduce作业ID**(步骤2)。
* **检查作业的输出说明**。例如**，如果没有指定目录或输出目录已经存在，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，错误返回给MapReduce程序**。
* **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)复制到一个以作业ID命名的目录下的共享文件系统中**(步骤3)。**作业JAR的复本较多**(由mapreduce.client.submit.file.replication属性控制，默认值为10)，**因此在运行作业的任务时，集群中有很多个复本可供节点管理器访问**。
* 通过调用**资源管理器**的submitApplication()方法提交作业(步骤4)。

## 1.2.2 作业的初始化

1.资源管理器调用它的submitApplication()消息后，便将**请求传递给YARN调度器**(scheduler)。调度器分配一个容器，然后**节点管理器在资源管理器的管理下在容器中启动application master的进程(步骤5a, 5b)**。

2.**MapReduce作业的application master是一个Java应用程序**，它的主类是**MRAppMaster**。由于将**接受来自任务的进度和完成报告**(步骤6)，因此**application master对作业的初始化是通过创建多个簿记对象以保持对作业进度的跟踪来完成的**。

3.接下来，它**接受来自共享文件系统的、在客户端计算的输入分片**(步骤7)。然后对**每一个分片创建一个map任务对象**以及由mapreduce.job.reduces属性(通过作业的setNumReduceTasks()方法设置)确定的**多个reduce任务对象**。**任务ID在此时分配**。

4.**application master必须决定如何构成MapReduce作业的各个任务**。**如果作业很小，就选择和自己在同一个JVM上运行任务**。与在一个节点上顺序运行这些任务相比，**当application master判断在新的容器中分配和运行任务的开销大于并行运行它们的开销时**，就会发生这一情况。这样的作业称为uberized，或者作为uber任务运行。

5.哪些作业是小作业？**默认情况下，小作业就是少于10个mapper 且只有1个reducer且输入大小小于一个HDFS块的作业**(通过设置mapreduce.job.ubertask.maxmaps ,mapreduce.job.ubertask.maxreduces和 mapreduce.job.ubertask.maxbytes可以改变这几个值)。必须明确启用**Uber任务**(对于单个作业，或者是对整个集群)，具体方法是将mapreduce.job.ubertask.enable设置为true。

6.最后，在任何任务运行之前，application master调用setupJob()方法设置outputCommitter。FileOutputCommitter为默认值，**表示将建立作业的最终输出目录及任务输出的临时工作空间**。提交协议(commit protocol)将在7.4.3节介绍。

## 1.2.3 任务的分配

1.如果作业不适合作为uber任务运行，**那么application master就会为该作业中的所有map任务和reduce任务向资源管理器请求容器**(步骤8)。**首先为Map任务发出请求**，该**请求优先级要高于reduce任务的请求**，这是因为**所有的map任务必须在reduce的排序阶段启动前完成**。直到有**5%的map任务已经完成时，为reduce任务的请求才会发出**。

2.**reduce任务能够在集群中任意位置运行**，**但是map任务的请求有着数据本地化的局限，这也是调度器所关注的**。

3.**在理想的情况下，任务是数据本地化(data local)的，意味着任务在分片驻留的同一节点上运行**。

4.**可选的情况是，任务可能是机架本地化(rack local)的，即和分片在同一机架而非同一节点上运行**。

5.**有一些任务既不是数据本地化，也不是机架本地化，它们会从别的机架，而不是运行所在机架上获取自己的数据**。对于一个特定的作业运行，可以通过查看作业的计数器来确定在每个本地化层次上运行的任务的数量。

6.**请求也为任务指定了内存需求和CPU 数**。**在默认情况下，每个map任务和reduce任务都分配到1024 MB的内存和一个虚拟的内核**，**这些值可以在每个作业的基础上进行配置**(遵从于10.3.3节描述的最大值和最小值)，分别通过4个属性来设置mapreduce.map.memory.mb、mapreduce.reduce.memory.mb、mapreduce.map.cpu.vcores和 mapreduce.reduce.cpu.vcoresp.memory.mb。

## 1.2.4 任务的执行

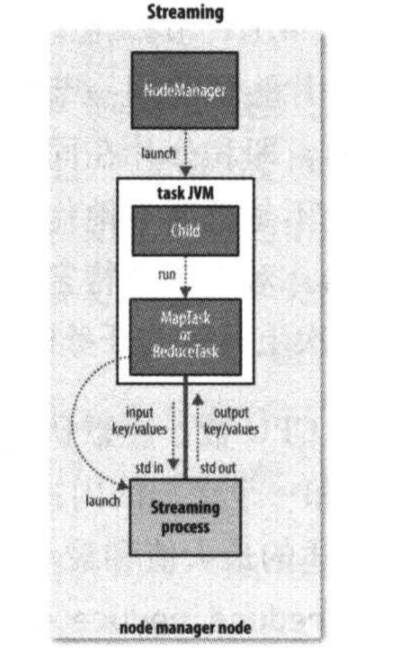
1.一旦**资源管理器的调度器为任务分配了一个特定节点上的容器**，application master就通过与节点管理器通信来启动容器(步骤9a和9b)。该任务由主类为YarnChild的一个Java应用程序执行。在它运行任务之前，**首先将任务需要的资源本地化**，包括作业的配置、JAR文件和所有来自分布式缓存的文件(步骤10)。最后，运行map任务和reduce任务(步骤11)。

2.**YarnChild在指定的JVM中运行**，因此**用户定义的map或reduce函数**(甚至是YarnChild)中的**任何缺陷不会影响到节点管理器，例如导致其崩溃或挂起**。

3.**每个任务都能够执行搭建(setup)和提交(commit)动作**，它们和任务本身在同一个JVM中运行，并由作业的OutputCommitter确定。对于基于文件的作业，提交动作将任务输出由临时位置搬移到最终位置。提交协议确保当推测执行被启用时，只有一个任务副本被提交，其他都被取消。

### Streaming

1.Streaming运行特殊的map任务和reduce任务，目的是运行用户提供的可执行程序，并与之通信。



2.Streaming任务使用**标准输入和输出流与进程**(可以用任何语言写)进行**通信**。在任务执行过程中，**Java进程都会把输入键-值对传给外部的进程**，后者通过用户定义的map函数和reduce 函数来执行它并把输出键-值对传回Java进程。从节点管理器的角度看，就像其子进程自己在运行map或reduce代码一样。

## 1.2.5 进度和状态的更新

1.**MapReduce 作业是长时间运行的批量作业**，**运行时间范围从数秒到数小时**。这可能是一个很长的时间段，所以对于用户而言，能够得知关于作业进展的一些反馈是很重要的。**一个作业和它的每个任务都有一个状态(status)**，包括：**作业或任务的状态**(比如，运行中，成功完成，失败)、**map和reduce 的进度**、**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期间不断改变，它们是如何与客户端通信的呢?

2.任务在运行时，**对其进度(progress，即任务完成百分比)保持追踪**。对**map任务**，**任务进度是已处理输入所占的比例**。对reduce任务，情况稍微有点复杂，**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**(详情参见7.3节)。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**，这是因为已经**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)**。

### MapReduce中进度的组成

1.**进度并不总是可测量的**，但是虽然如此，它能告诉Hadoop有个任务正在做一些事情。比如，正在写输出记录的任务是有进度的，**即使此时这个进度不能用需要写的总量的百分比来表示**(因为即便是产生这些输出的任务，也可能不知道需要写的总量)。

2.**进度报告**很重要。构成进度的所有操作如下：

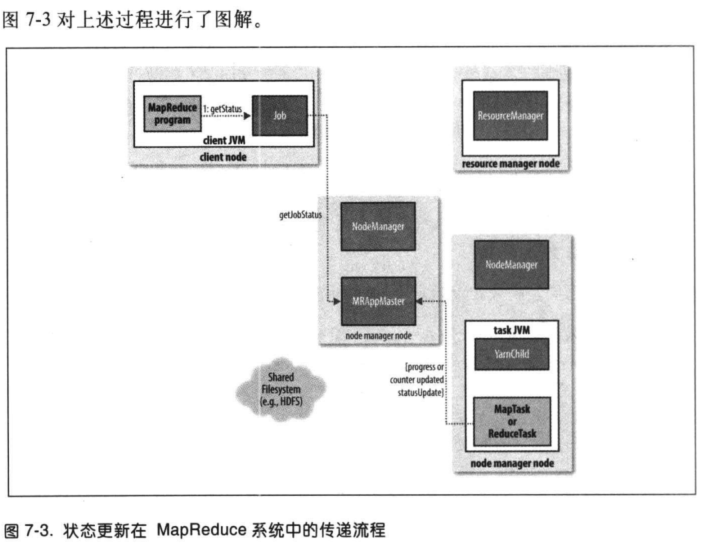
* 读入一条输入记录(在mapper或reducer中)
* 写入一条输出记录(在mapper或reducer中)
* 设置状态描述(通过Reporter或TaskAttemptContext的setStatus()方法)
* 增加计数器的值(使用Reporter的incrCounter()方法或Counter的increment()方法)
* 调用Reporter或TaskAttemptContext的progress()方法

3.**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(详情参见2.3.2节)，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，要么由用户自己定义。

4.当map任务或reduce任务运行时，**子进程和自己的父application master通过umbilical接口通信**。每隔3秒钟，**任务通过这个umbilical接口向自己的application master报告进度和状态**(包含计数器)，application master会形成一个作业的汇聚视图(aggregate view)。

5.**资源管理器的界面显示了所有运行中的应用程序**，并且分别有**链接指向这些应用各自的application master界面**，这些界**面展示了MapReduce作业的更多细节**，包括其**进度**。

6.在作业期间，**客户端每秒钟轮询一次**application master以**接收最新状态**(轮询间隔通过mapreduce.client.progressmonitor.pollinterval设置)。**客户端也可以使用Job 的 getstatus()方法得到一个Jobstatus 的实例，后者包含作业的所有状态信息**。



## 1.2.6 作业的完成

1.当**application master收到作业最后一个任务已完成的通知后**，便把**作业的状态设置为“成功”**。然后，在Job轮询状态时，便知道**任务已成功完成**，于是Job打印一条消息告知用户，然后从 waitForcompletion()方法返回。**Job 的统计信息和计数值也在这个时候输出到控制台**。

2.最后，作业完成时，**application master和任务容器清理其工作状态**(这样中间输出将被删除)，OutputCommitter的 commitJob(）方法会被调用。**作业信息由作业历史服务器存档，以便日后用户需要时可以查询**。

# 1.3 失败(重点)

## 1.3.1 任务运行失败

1.在现实情况，可能遇到很多问题。使用Hadoop最主要的好处之一是处理此类故障并让你能够成功完成作业。我们需要考虑以下实体失败：**任务**、**application master**、**节点管理器**和**资源管理器**。

2.首先考虑任务失败的情况。**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**。如果发生这种情况，任务JVM会在退出之前向其父application master发送错误报告。错误报告最后被记入用户日志。**application master将此次任务尝试标记为failed**(失败)，并释放以便资源可以为其他任务使用。

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

4.另一种失败模式是**任务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，**节点管理器会注意到进程已经退出**，并通知**application master将此次任务尝试标记为失败**。

5.**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，便会将任务标记为失败**。在此之后，任务JVM进程将被自动杀死。任务被认为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)进行设置，对应的属性为mapreduce.task.timeout，单位为毫秒。

6.**超时(timeout)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失败**。在这种情况下**，被挂起的任务永远不会释放它的容器并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。参见7.1.5节的补充材料“MapReduce中进度的组成”。

7.**application master被告知一个任务尝试失败后**，将**重新调度该任务的执行**。application master 会试图**避免在以前失败过的节点管理器上重新调度该任务**。此外，如果一个**任务失败过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由mapreduce.map.maxattempts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，**如果任何任务失败次数大于4(或最多尝试次数被配置为4)，整个作业都会失败**。

8.对于一些应用程序，**我们不希望一旦有少数几个任务失败就中止运行整个作业**，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。针对map任务和reduce任务的设置可以通过mapreduce.map.failures.maxpercent和mapreduce.reduce.failures.maxpercent这两个属性来完成。

9.**任务尝试(task attempt)也是可以中止的(killed)**，这与失败不同。**任务尝试可以被中止是因为它是一个推测副本**(相关详情可以参见7.4.2节)或**因为它所处的节点管理器失败**，导致**application master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)，**因为尝试被中止并不是任务的过错**。

## 1.3.2 application master运行失败

1.**YARN中的应用程序在运行失败的时候有几次尝试机会**，就像 **MapReduce任务在遇到硬件或网络故障时要进行几次尝试一样**。运行MapReduce application master的最多尝试次数由mapreduce.am.max-attempts属性控制。默认值是2，即如果 MapReduce application master 失败两次，便不会再进行尝试，作业将失败。

2.**YARN对集群上运行的YARN application master的最大尝试次数加以了限制**，单**个的应用程序不可以超过这个限制**。该限制由yarn.resourcemanager. am.max-attempts属性设置，默认值是2，这样如果你想增加MapReduce application master的尝试次数，**你也必须增加集群上 YARN的设置**。

3.**恢复的过程如下**。**application master向资源管理器发送周期性的心跳**，当**application master失败时，资源管理器将检测到该失败并在一个新的容器(由节点管理器管理)中开始一个新的master实例**。对于Mapreduce application master，**它将使用作业历史来恢复失败的应用程序所运行任务的状态，使其不必重新运行**。默认情况下恢复功能是开启的，但可以通过设置yarn.app.mapreduce.am.job.recovery.enable 为false来关闭这个功能。

4.**MapReduce客户端向application master 轮询进度报告**，但是如果**它的 application master运行失败，客户端就需要定位新的实例**。在作业初始化期间，**客户端向资源管理器询问并缓存 application master的地址**，使其每次需要向**application master查询时不必重载资源管理器**。但是，如果**application master运行失败，客户端就会在发出状态更新请求时经历超时**，这时客户端会**折回向资源管理器请求新的application master的地址。这个过程对用户是透明的**。

## 1.3.3 节点管理器运行失败

1.如果**节点管理器由于崩溃或运行非常缓慢而失败**，就会**停止向资源管理器发送心跳信息**(或发送频率很低)。如果10 分钟内(可以通过属性yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms 设置，以毫秒为单位)**没有收到一条心跳信息**，**资源管理器将会通知停止发送心跳信息的节点管理器**，并且将**其从自己的节点池中移除以调度启用容器**。

2.在失败的**节点管理器**上运行的所有任务或application master都用之前两节描述的机制进行恢复。另外，对于那些曾经在失败的节点管理器上运行且成功完成的map任务，如果属于未完成的作业，那么application master会安排它们重新运行。**这是由于这些任务的中间输出驻留在失败的节点管理器的本地文件系统中，可能无法被reduce任务访问的缘故**。

3.如果**应用程序的运行失败次数过高**，那么**节点管理器可能会被拉黑**，即使节点管理器自己并没有失败过。由application master管理黑名单，对于MapReduce，**如果一个节点管理器上有超过三个任务失败，application master就会尽量将任务调度到不同的节点上**。用户可以通过作业属性mapreduce.job.maxtaskfailures.per.tracker设置该阈值。

## 1.3.4 资源管理器运行失败

1.**资源管理器失败是一个非常严重的问题**，没有资源管理器，作业和任务容器将无法启动。在默认的配置中，资源管理器是个单点故障，这是由于在机器失败的情况下(尽管不太可能发生)，所有运行的作业都失败且不能被恢复。

2.为获得**高可用性(HA)**，在**双机热配置下**，运行一对**资源管理器是必要的**。如果主资源管理器失败了，那么备份资源管理器能够接替，且客户端不会感到明显的中断。

3.关于所有**运行中的应用程序的信息存储在一个高可用的状态存储区中**(由ZooKeeper或HDFS备份)，**这样备机可以恢复出失败的主资源管理器的关键状态**。

4.节点管理器信息没有存储在状态存储区中，因为当节点管理器发送它们的第一个心跳信息时，节点管理器的信息能以相当快的速度被新的资源管理器重构。(同样要注意，由于任务是由application master管理的，因此任务不是资源管理器的状态的一部分。这样，要存储的状态量比MapReduce 1 中 jobtracker要存储的状态量更好管理。)

5.当**新资源管理器启动后**，它从状**态存储区中读取应用程序的信息**，然后为集群中运行的所有应用程序重启 application master。**这个行为不被计为失败的应用程序尝试**(所以不会计入yarn.resourcemanager.am.max-attempts)，这是因为**应用程序并不是因为程序代码错误而失败，而是被系统强行中止的**。实际情况中，**application master重启不是MapReduce应用程序的问题，因为它们是恢复已完成的任务的工作**(详见7.2.2节)**。**

6.**资源管理器从备机到主机的切换是由故障转移控制器(failover controller)处理的**。默认的故障转移控制器是自动工作的，**使用ZooKeeper的 leader选举机制(leader election)以确保同一时刻只有一个主资源管理器**。**不同于HDFS高可用性**(详见3.2.5节)的**实现**，**故障转移控制器不必是一个独立的进程**，为配置方便，默认情况下嵌入在资源管理器中。故障转移也可以配置为手动处理，但不建议这样。

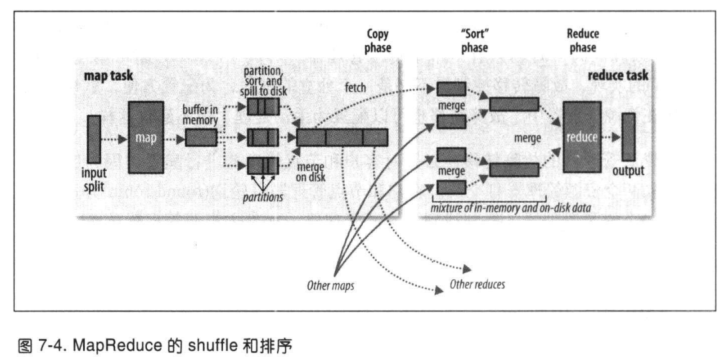
7.为**应对资源管理器的故障转移**，必须对**客户和节点管理器进行配置**，因为他们可能是在和**两个资源管理器打交道**。**客户和节点管理器以轮询(round-robin)方式试图连接每一个资源管理器，直到找到主资源管理器**。**如果主资源管理器故障，他们将再次尝试直到备份资源管理器变成主机**。

# 1.4 shuffle和排序(重点)

## 1.4.1 Map端

1.**MapReduce确保每个reducer的输入都是按键排序的**。**系统执行排序、将map输出作为输入传给reducer的过程称为shuffle**。

2.map函数开始产生输出时，并不是简单地将它写到磁盘。它利用**缓冲的方式**写到内存并出于**效率的考虑进行预排序**。下图展示了这个过程。



3.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**，这个值可以通过改变mapreduce.task.io.sort.mb属性来调整。一旦缓冲区内容达到阈值(mapreduce.map.sort.spill.percent，默认为0.80，或80%)，**一个后台线程便开始把内容溢出(spill)到磁盘**。

4.在**溢出写到磁盘过程中**，**map输出继续写到缓冲区**，但如果在此期间**缓冲区被填满**，**map会被阻塞直到写磁盘过程完成**。**溢出写过程按轮询方式将缓冲区中的内容写到**mapreduce.cluster.local.dir属性在**作业特定子目录下指定的目录中**。

5.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，后台线程按键进行内存中排序，如果有一个combiner函数，它就在排序后的输出上运行。**运行combiner函数使得map输出结果更紧凑**，**因此减少写到磁盘的数据和传递给reducer的数据**。

6.**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)**，因此在map任务写完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已排序的输出文件**。配置属性mapreduce.task.io.sort.factor**控制着一次最多能合并多少流**，**默认值是10**。

7.如果**至少存在3个溢出文件**(通过mapreduce.combine.minspills属性设置)时，**combiner就会在输出文件写到磁盘之前再次运行**。**combiner可以在输入上反复运行，但并不影响最终结果**。如果只有1或2个溢出文件，由于map输出规模减少，因而不值得调用combiner带来的开销，因此不会为该map输出再次运行combiner。

8.**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意**，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.compress.codec指定。

9.reducer通过HTTP得到输出文件的分区。用于文件分区的工作线程的数量由任务的mapreduce.shuffle.max.threads属性控制，此设置针对的是每一个节点管理器，而不是针对每个map任务。默认值О将最大线程数设置为机器中处理器数量的两倍。

## 1.4.2 Reduce端

1.现在转到处理过程的reduce部分。**map输出文件位于运行map任务的tasktracker的本地硬盘**(注意，但reduce输出并不这样)，现在，tasktracker需要为分区文件运行reduce任务。并且，**reduce任务需要集群上若干个map任务的map的输出作为其特殊的分区文件**。

2.每个map任务的完成时间可能不同，因此在每个任务完成时，**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**，因此能够并行取得map输出。默认值是5个线程，但这个默认值可以修改设置mapreduce.reduce.shuffle.parallelcopies属性。

3.**reducer如何知道要从哪台机器取得map输出呢**？

**map任务成功完成后**，它们会使用**心跳机制**通知它们的application master。因此，对于指定作业，**application master知道map输出和主机位置之间的映射关系**。reducer中的一个线程定期询问master以便获取map输出主机的位置，直到获得所有输出位置。

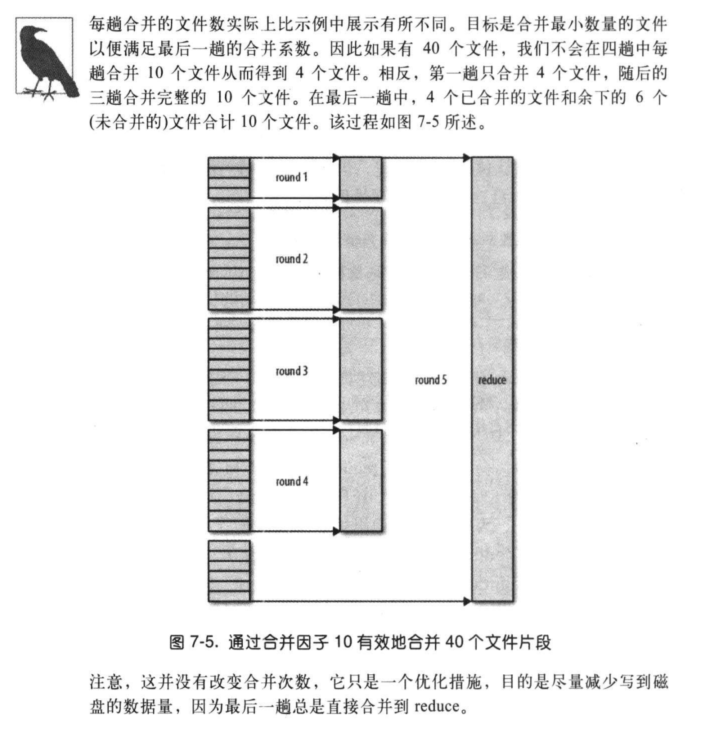
由于第一个reducer可能失败，**因此主机并没有在第一个reducer检索到map输出时就立即从磁盘上删除它们**。相反，等待master，直到application master告知它删除map输出，这是作业完成后执行的。

4.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，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**内存缓冲区达到阈值大小**(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硬盘的数据量**。

5.随着磁盘上副本增多，**后台线程会将它们合并为更大的、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**。

6.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**，**因为排序是在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，由mapreduce.task.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**，**因此最后有5个中间文件**。

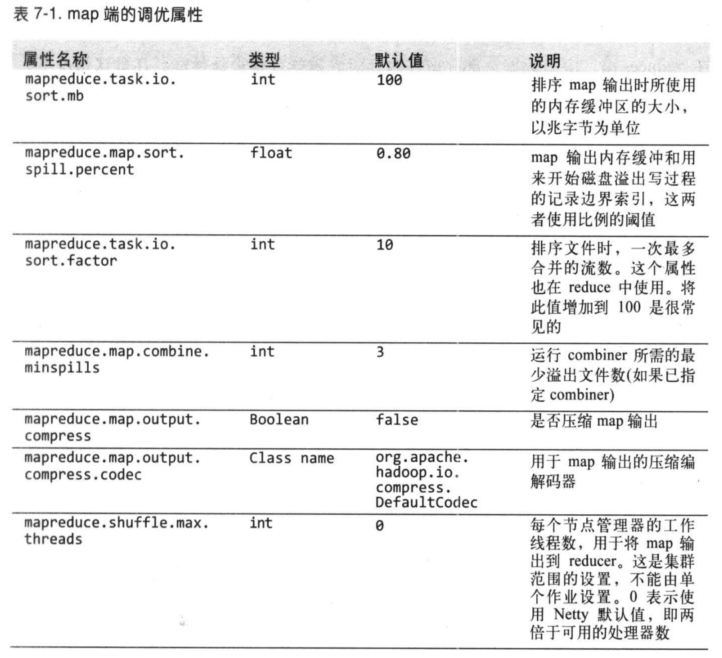
7.在**最后阶段，即reduce阶段**，直接把**数据输入reduce函数**，从而**省略了一次磁盘往返行程**，并没有将这5个文件合并成一个已排序的文件作为最后一趟。**最后的合并可以来自内存和磁盘阶段**。



8.在reduce阶段，**对已排序输出中的每个键调用reduce函数**。此阶段的**输出直接写到输出文件系统**，一般为**HDFS**。如果采用HDFS，**由于节点管理器也运行数据节点，所以第一个块复本将被写到本地磁盘**。

## 1.4.3 配置调优

1.现在我们已经有比较好的基础来理解如何**调优 shuffle过程来提高MapReduce 性能**。下表总结了相关**设置和默认值**，这些设置**以作业为单位**(除非有特别说明)，**默认值适用于常规作业**。

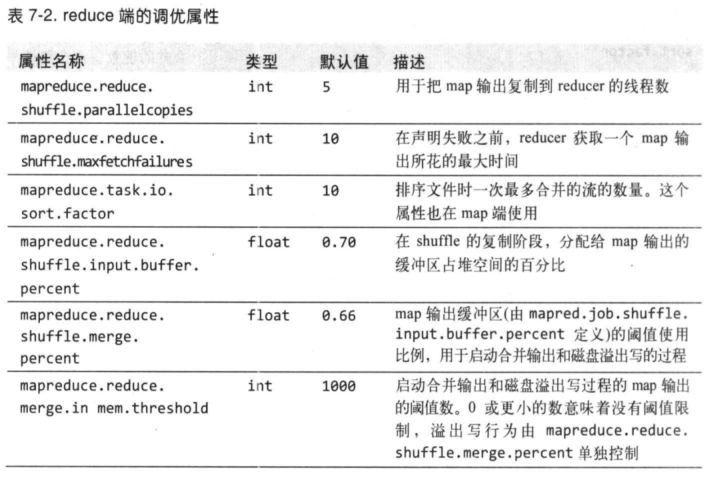


2.**调优的总原则是给shuffle过程尽量多提供内存空间**。然而，有一个平衡问题，也就是要确保**map函数和reduce函数能得到足够的内存在运行**。这就是为什么**写map函数和reduce函数时尽量少用内存的原因**，**它们不应该无限使用内存**(例如，**避免在map中堆积数据**)。

3.运行map任务和reduce任务的JVM，其内存大小由mapred.child.java.opts属性进行配置。任务节点上的内存应该尽可能设置大些。

4.在map端，**可以通过避免多次溢出写磁盘来获得最佳性能：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**。MapReduce 计数器(“SPILLED\_RECORDS”,参见9.1节“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。

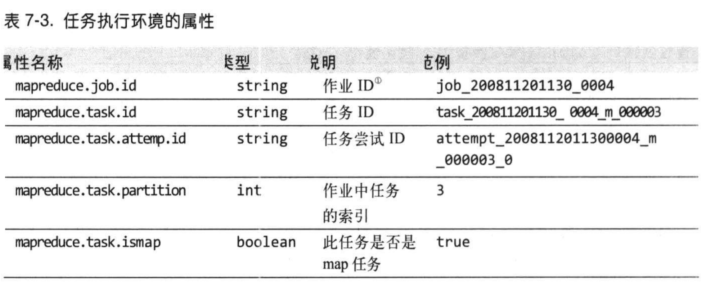
5.在reduce端，**中间数据全部驻留在内存时**，**就能获得最佳性能**。在默认情况下，这是不可能发生的，因为所有内存一般都预留给reduce函数(map输出缓冲区执行操作包括reduce函数、合并输出和溢写过程)。但如果reduce 函数的内存需求不大，把mapreduce.reduce.merge.inmem.threshold 设置为0，把mapreduce.reduce.input.buffer.percent 设置为1.0(或一个更低的值，详见表7-2)就可以提升性能。



# 1.5 任务的执行(重点)

## 1.5.1 任务执行环境

1.**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文件的名称**(参见8.2.1节)，**map任务或reduce任务可以得知任务的尝试次数**。表中的属性可以从作业的配置信息中获得，在老版本的MapReduce API中通过为Mapper或Reducer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。



2.Hadoop**设置作业配置参数作为Streaming程序的环境变量**。但它用下下划线来代替非字母数字的符号，以确保名称的合法性。

## 1.5.2 推测执行(重点)

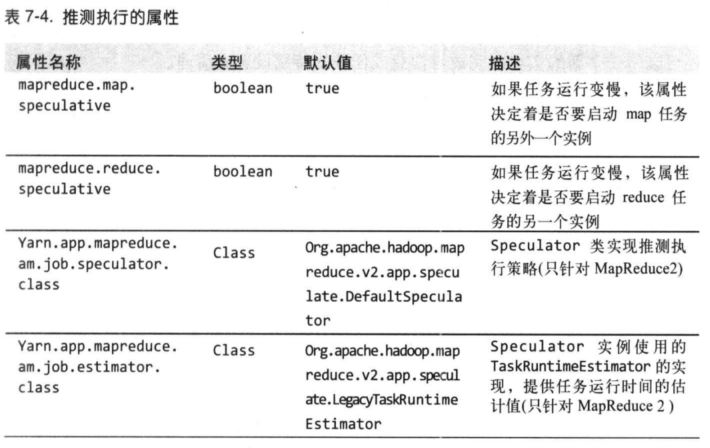
1.**MapReduce模型将作业分解成任务**，**然后并行地运行任务以使作业的整体执行时间少于各个任务顺序执行的时间**。**这使作业执行时间对运行缓慢的任务很敏感**，**因为只运行一个缓慢的任务会使整个作业所用的时间远远长于执行其他任务的时间**。当一个**作业由几百或几千个任务组成时，可能出现少数“拖后腿”的任务，这是很常见的**。

2.任务执行缓慢可能有多种原因，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，它会尽量检测**，并**启动另一个相同任务作为备份**。**这就是所谓的任务的“推测执行”**。

3.必须认识到一点:**如果同时启动两个重复的任务**，它们会**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，**调度器跟踪作业中所有相同类型**(map 和 reduce)**任务的进度**，并且仅仅**启动运行速度明显低于平均水平的那一小部分任务的推测副本**。**一个任务成功完成后，任何正在运行的重复任务都将被中止**，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，推测任务就会被终止**;同样，**如果推测任务先完成，那么原任务就会被中止**。

4.**推测执行是一种优化措施**，**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减慢**，**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**，因为相同的软件缺陷可能会影响推测式任务。应该修复软件缺陷，使任务不会挂起或运行速度减慢。

5.在默认情况下，**推测执行是启用的**。**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**。相关的属性如表7-4所示。



6.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，推测执行会减少整体的吞吐量，因为**冗余任务的执行时会减少作业的执行时间**。因此，**一些集群管理员倾向于在集群上关闭此选项**，**而让用户根据个别作业需要而开启该功能**。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式。

7.对于**reduce任务，关闭推测执行是有益的，因为任意重复的reduce任务都必须将取得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**。

8.**关闭推测执行的另一种情况是为了非幂等(nonidempotent)任务**。然而在很多情况下，将任务**写成幂等的并使用OutputCommitter来提升任务成功时输出到最后位置的速度**，这是可行的。