获取教材和讲义 PPT 等各种课程资料请访问 http://dblab.xmu.edu.cn/node/422

=课程教材由林子雨老师根据网络资料编著=

厦门大学计算机科学系教师 林子雨 编著

http://www.cs.xmu.edu.cn/linziyu

2013 年 9 月

前言

本教程由厦门大学计算机科学系教师林子雨编著，可以作为计算机专业研究生课程《大数据技术基础》的辅助教材。

本教程的主要内容包括：大数据概述、大数据处理模型、大数据关键技术、大数据时代面临的新挑战、NoSQL 数据库、云数据库、Google Spanner、Hadoop、HDFS、HBase、MapReduce、Zookeeper、流计算、图计算和 Google Dremel 等。

本教程是林子雨通过大量阅读、收集、整理各种资料后精心制作的学习材料，与广大数据库爱好者共享。教程中的内容大部分来自网络资料和书籍，一部分是自己撰写。对于自写内容，林子雨老师拥有著作权。

本教程 PDF 文档及其全套教学 PPT 可以通过网络免费下载和使用（下载地址：http://dblab.xmu.edu.cn/node/422）。教程中可能存在一些问题，欢迎读者提出宝贵意见和建议！

本教程已经应用于厦门大学计算机科学系研究生课程《大数据技术基础》，欢迎访问 2013 班级网站 http://dblab.xmu.edu.cn/node/423。

林子雨的 E-mail 是：ziyulin@xmu.edu.cn。

林子雨的个人主页是：http://www.cs.xmu.edu.cn/linziyu。

林子雨于厦门大学海韵园

2013 年 9 月

第 4 章 MapReduce

厦门大学计算机科学系教师 林子雨 编著

个人主页：http://www.cs.xmu.edu.cn/linziyu

课程网址：http://dblab.xmu.edu.cn/node/422

2013 年 9 月

第 4 章 MapReduce

MapReduce 是一种编程模型，用于大规模数据集（大于 1TB）的并行运算。概念“Map（映射）”和“Reduce（归约）”以及它们的主要思想，都是从函数式编程语言里借用而来的，同时也包含了从矢量编程语言里借来的特性。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。

本章介绍 MapReduce 的相关知识，内容要点如下：

- 分布式并行编程：编程方式的变革
- MapReduce 模型概述
- Map 和 Reduce 函数
- MapReduce 工作流程
- 并行计算的实现
- 实例分析：WordCount
- 新 MapReduce 框架 Yarn

4.1 分布式并行编程：编程方式的变革

根据摩尔定律，约每隔 18 个月，CPU 性能会提高一倍。然而，由于晶体管电路已经逐渐接近其物理上的性能极限，摩尔定律在 2005 年左右开始失效。随着互联网时代的到来，软件编程方式发生了重大的变革，基于大规模计算机集群的分布式并行编程是将来软件性能提升的主要途径。基于集群的分布式并行编程能够让软件与数据同时运行在连成一个网络的许多台计算机上，由此获得海量计算能力。

在摩尔定律的作用下，以前程序员根本不用考虑计算机的性能会跟不上软件的发展，因为约每隔 18 个月，CPU 的主频就会增加一倍，性能也将提升一倍，软件根本不用做任何改变，就可以享受免费的性能提升。然而，由于晶体管电路已经逐渐接近其物理上的性能极限，摩尔定律在 2005 年左右开始失效了，人类再也不能期待单个 CPU 的速度每隔 18 个月就翻一倍，为我们提供越来越快的计算性能。Intel、AMD、IBM 等芯片厂商开始从多核这个角

度来挖掘 CPU 的性能潜力，多核时代以及互联网时代的到来，将使软件编程方式发生重大变革，基于多核的多线程并发编程以及基于大规模计算机集群的分布式并行编程是将来软件性能提升的主要途径。

许多人认为这种编程方式的重大变化将带来一次软件的并发危机，因为我们传统的软件方式基本上是单指令单数据流的顺序执行，这种顺序执行十分符合人类的思考习惯，却与并发并行编程格格不入。基于集群的分布式并行编程，能够让软件与数据同时运行在连成一个网络的许多台计算机上，这里的每一台计算机均可以是一台普通的 PC 机。这样的分布式并行环境的最大优点是，可以很容易地通过增加计算机来扩充新的计算节点，并由此获得不可思议的海量计算能力，同时又具有相当强的容错能力，一批计算节点失效也不会影响计算的正常进行以及结果的正确性。Google 就是这么做的，他们使用了叫做 MapReduce 的并行编程模型进行分布式并行编程，运行在叫做 GFS (Google File System)的分布式文件系统上，为全球亿万用户提供搜索服务。

Hadoop 实现了 Google 的 MapReduce 编程模型，提供了简单易用的编程接口，也提供了它自己的分布式文件系统 HDFS，与 Google 不同的是，Hadoop 是开源的，任何人都可以使用这个框架来进行并行编程。如果说分布式并行编程的难度足以让普通程序员望而生畏的话，开源的 Hadoop 的出现，则极大地降低了它的门槛。你会发现，基于 Hadoop 编程非常简单，无需任何并行开发经验，你也可以轻松地开发出分布式的并行程序，并让其令人难以置信地同时运行在数百台机器上，然后在短时间内完成海量数据的计算。你可能会觉得你不可能会拥有数百台机器来运行你的并行程序，而事实上，随着“云计算”的普及，任何人都可以轻松获得这样的海量计算能力。例如，现在 Amazon 公司的云计算平台 Amazon EC2 已经提供了这种按需计算的租用服务。

掌握一点分布式并行编程的知识对将来的程序员是必不可少的，Hadoop 是如此地简便好用，何不尝试一下呢？也许你已经急不可耐地想试一下基于 Hadoop 的编程是怎么回事了，但毕竟这种编程模型与传统的顺序程序大不相同，掌握一点基础知识才能更好地理解基于 Hadoop 的分布式并行程序是如何编写和运行的。因此，这里会先介绍一下 MapReduce 的计算模型和 Hadoop 中的分布式文件系统 HDFS，然后介绍 Hadoop 是如何实现并行计算的。

4.2 MapReduce 模型概述

MapReduce 是 Google 公司的核心计算模型，这是一个令人惊讶的、简单却又威力巨大

的模型，它将复杂的、运行于大规模集群上的并行计算过程高度地抽象到了两个函数：Map 和 Reduce。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（归约）”，以及它们的主要思想，都是从函数式编程语言里借来的，同时包含了从矢量编程语言里借来的特性。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。

一个 MapReduce 作业（job）通常会把输入的数据集切分为若干独立的数据块，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 框架和分布式文件系统是运行在一组相同的节点上的，也就是说，计算节点和存储节点通常在一起。这种配置允许框架在那些已经存好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 框架由单独一个 master JobTracker 和每个集群节点一个 slave TaskTracker 共同组成。这个 master 负责调度构成一个作业的所有任务，这些任务分布在不同的 slave 上，master 监控它们的执行，重新执行已经失败的任务。而 slave 仅负责执行由 master 指派的任务。

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）。然后，Hadoop 的 job client 提交作业（jar 包/可执行程序等）和配置信息给 JobTracker，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 应用程序则不一定要用 Java 来写。

4.3 Map 和 Reduce 函数

MapReduce 计算模型的核心是 map 和 reduce 两个函数，这两个函数由用户负责实现，功能是按一定的映射规则将输入的<key，value>转换成另一个或一批<key，value>输出（见表 4-1）。

表 4-1 Map 和 Reduce

函数	输入	输出	说明
Map	<k1,v1>	List(<k2,v2>)	1.将小数据集进一步解析成一批 <key,value>对，输入 Map 函数中进行处理。 2.每一个输入的 <k1,v1> 会输出一批 <k2,v2>。 <k2,v2>是计算的中间结果。
Reduce	<k2,List(v2)>	<k3,v3>	输入的中间结果 <k2,List(v2)> 中的 List(v2)表示是一批属于同一个 k2 的 value

以一个计算文本文件中每个单词出现的次数的程序为例，<k1，v1>可以是 <行在文件中的偏移位置，文件中的一行>，经 Map 函数映射之后，形成一批中间结果 <单词，出现次数>，而 Reduce 函数则可以对中间结果进行处理，将相同单词的出现次数进行累加，得到每个单词的总的出现次数。

基于 MapReduce 计算模型编写分布式并行程序非常简单，程序员的主要编码工作就是实现 Map 和 Reduce 函数，其它的并行编程中的种种复杂问题，如分布式存储、工作调度、负载均衡、容错处理、网络通信等，均由 MapReduce 框架(比如 Hadoop)负责处理，程序员完全不用操心。

4.4MapReduce 工作流程

4.4.1 工作流程概述

图 4-1 说明了用 MapReduce 来处理大数据集的过程，这个 MapReduce 的计算过程简而言之，就是将大数据集分解为成百上千的小数据集，每个(或若干个)数据集分别由集群中的一个节点(一般就是一台普通的计算机)进行处理并生成中间结果，然后这些中间结果又由大量的节点进行合并，形成最终结果。

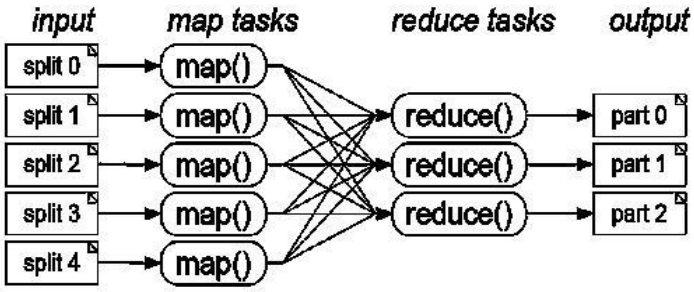


图 4-1 MapReduce 工作流程

MapReduce 的输入一般来自 HDFS 中的文件，这些文件分布存储在集群内的节点上。运行一个 MapReduce 程序会在集群的许多节点甚至所有节点上运行 mapping 任务，每一个 mapping 任务都是平等的：mappers 没有特定“标识物”与其关联。因此，任意的 mapper

都可以处理任意的输入文件。每一个 mapper 会加载一些存储在运行节点本地的文件集来进行处理（注：这是移动计算，把计算移动到数据所在节点，可以避免额外的数据传输开销）。

当 mapping 阶段完成后，这阶段所生成的中间“键值对”数据必须在节点间进行交换，把具有相同键的数值发送到同一个 reducer 那里。Reduce 任务在集群内的分布节点同 mappers 的一样。这是 MapReduce 中唯一的任务节点间的通信过程。map 任务之间不会进行任何的信息交换，也不会去关心别的 map 任务的存在。相似地，不同的 reduce 任务之间也不会有通信。用户不能显式地从一台机器发送信息到另外一台机器；所有数据传送都是由 Hadoop MapReduce 平台自身去做的，这些是通过关联到数值上的不同键来隐式引导的。这是 Hadoop MapReduce 的可靠性的基础元素。如果集群中的节点失效了，任务必须可以被重新启动。如果任务已经执行了有副作用（side-effect）的操作，比如说，跟外面进行通信，那共享状态必须存在可以重启的任务上。消除了通信和副作用问题，那重启就可以做得更优雅些。

4.4.2 MapReduce 各个执行阶段

一般而言，Hadoop 的一个简单的 MapReduce 任务执行流程如下（如图 4-2 所示）：

1) JobTracker 负责分布式环境中实现客户端创建任务并提交；

2)InputFormat 模块负责做 Map 前的预处理，主要包括以下几个工作：验证输入的格式是否符合 JobConfig 的输入定义，可以是专门定义或者是 Writable 的子类。将 input 的文件切分为逻辑上的输入 InputSplit，因为在分布式文件系统中 blocksize 是有大小限制的，因此大文件会被划分为多个较小的 block。通过 RecordReader 来处理经过文件切分为 InputSplit 的一组 records，输出给 Map。因为 InputSplit 是逻辑切分的第一步，如何根据文件中的信息来具体切分还需要 RecordReader 完成。

3)将 RecordReader 处理后的结果作为 Map 的输入，然后 Map 执行定义的 Map 逻辑，输出处理后的<key,value>对到临时中间文件。

4)Shuffle&Partitioner：在 MapReduce 流程中，为了让 reduce 可以并行处理 map 结果，必须对 map 的输出进行一定的排序和分割，然后再交给对应的 reduce，而这个将 map 输出进行进一步整理并交给 reduce 的过程，就称为 shuffle。Partitioner 是选择配置，主要作用是在多个 Reduce 的情况下，指定 Map 的结果由某一个 Reduce 处理，每一个 Reduce 都会有单独的输出文件。

6)Reduce 执行具体的业务逻辑，即用户编写的处理数据得到结果的业务，并且将处理

结果输出给 OutputFormat。

7)OutputFormat 的作用是，验证输出目录是否已经存在以及输出结果类型是否符合 Config 中配置类型，如果都成立，则输出 Reduce 汇总后的结果。

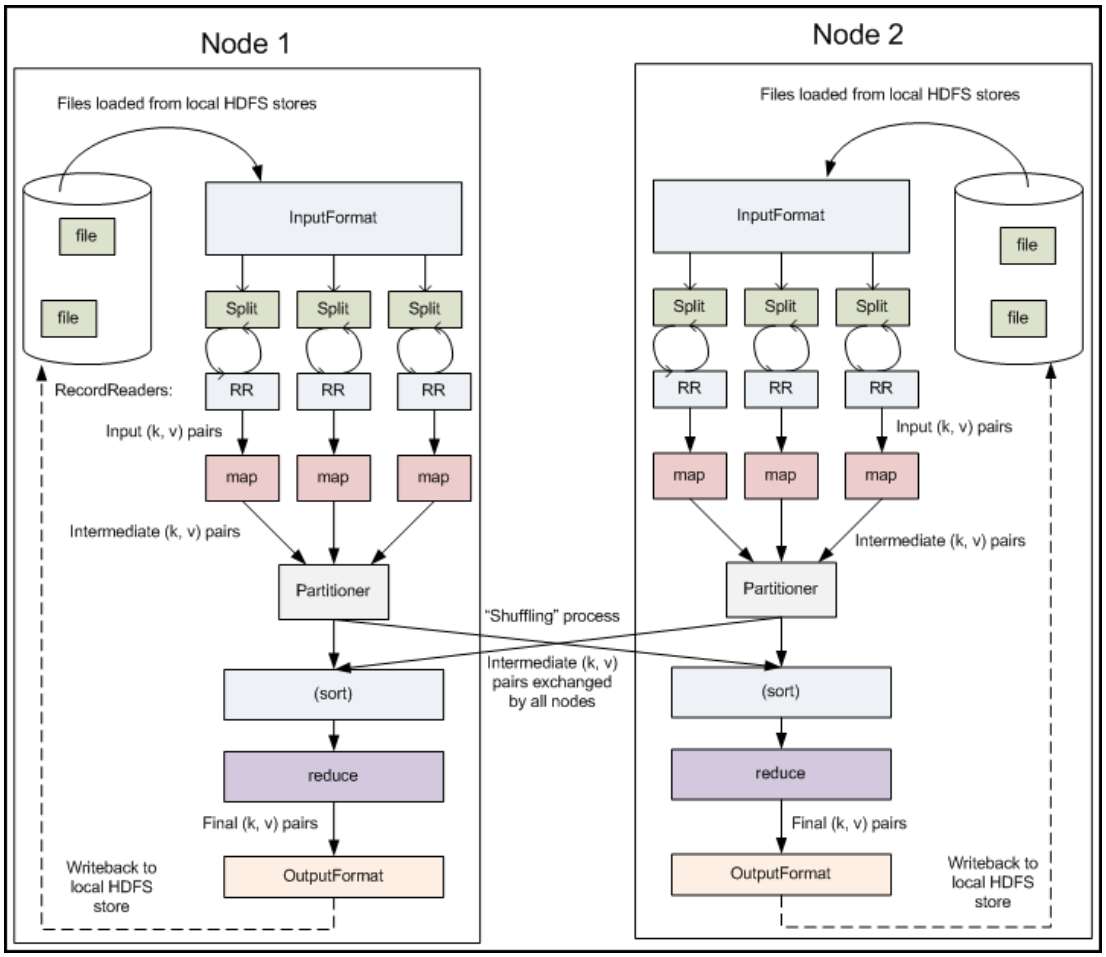


图 4-2 Hadoop MapReduce 工作流程中的各个执行阶段

下面简单介绍一下 MapReduce 过程的各个部分，包括输入文件、输入格式、输入块、记录读取器、Mapper、Partion&Shuffle、Reducer、输出格式、RecordWriter。

- **输入文件**

文件是 MapReduce 任务的数据的初始存储地。正常情况下，输入文件一般是存在 HDFS 里。这些文件的格式可以是任意的；我们可以使用基于行的日志文件，也可以使用二进制格式，多行输入记录或其它一些格式。这些文件会很大——数十 GB 或更大。

- **输入格式**

InputFormat 类定义了如何分割和读取输入文件，它提供了以下几个功能：

（1） 选择作为输入的文件或对象；

（2） 定义把文件划分到任务的 InputSplits；

（3） 为 RecordReader 读取文件提供了一个工厂方法。

Hadoop 自带了好几个输入格式。其中有一个抽象类叫 FileInputFormat，所有操作文件的 InputFormat 类都是从它那里继承功能和属性。当开启 Hadoop 作业时，FileInputFormat 会得到一个路径参数，这个路径内包含了所需要处理的文件，FileInputFormat 会读取这个文件夹内的所有文件（注：默认不包括子文件夹内的），然后它会把这些文件拆分成一个或多个 InputSplit。你可以通过 JobConf 对象的 setInputFormat()方法来设定应用到你的作业输入文件上的输入格式。表 4-2 给出了 MapReduce 提供的输入格式。

表 4-2 MapReduce 提供的输入格式

输入格式	描述	键	值
TextInputFormat	默认格式，读取文件的行	行的字节偏移量	行的内容
KeyValueInputFormat	把行解析为键值对	第一个 tab 字符前的所有字符	行剩下的内容
SequenceFileInputFormat	Hadoop 定义的高性能二进制格式	用户自定义	用户自定义

默认的输入格式是 TextInputFormat，它把输入文件每一行作为单独的一个记录，但不做解析处理。这对那些没有被格式化的数据或是基于行的记录来说是很有用的，比如日志文件。更有趣的一个输入格式是 KeyValueInputFormat，这个格式也是把输入文件每一行作为单独的一个记录。然而不同的是，TextInputFormat 把整个文件行当做值数据，KeyValueInputFormat 则是通过搜寻 tab 字符来把行拆分为“键值对”。这在把一个 MapReduce 的作业输出作为下一个作业的输入时显得特别有用，因为，默认的 map 输出格式正是按 KeyValueInputFormat 格式输出数据。最后来讲讲 SequenceFileInputFormat，它用于读取特殊的、特定于 Hadoop 的二进制文件，这些文件包含了很多能让 Hadoop 的 mapper 快速读取数据的特性。Sequence 文件是块压缩的，并提供了对几种数据类型（不仅仅是文本类型）直接的序列化与反序列化操作。Squence 文件可以作为 MapReduce 任务的输出数据，并且用它做一个 MapReduce 作业到另一个作业的中间数据是很高效的。

- **输入块（InputSplit）**

一个输入块描述了构成 MapReduce 程序中单个 map 任务的一个单元。把一个 MapReduce 程序应用到一个数据集上，即是指一个作业，会由几个（也可能几百个）任务组成。Map 任务可能会读取整个文件，但一般是读取文件的一部分。默认情况下，FileInputFormat 及其

子类会以 64MB 为基数来拆分文件（与 HDFS 的 Block 默认大小相同，Hadoop 建议 Split 大小与此相同）。你可以在 hadoop-site.xml（注：0.20.*以后是在 mapred-default.xml 里）文件内设定 mapred.min.split.size 参数来控制具体划分大小，或者在具体 MapReduce 作业的 JobConf 对象中重写这个参数。通过以块形式处理文件，我们可以让多个 map 任务并行地操作一个文件。如果文件非常大的话，这个特性可以通过并行处理大幅地提升性能。更重要的是，因为多个块（Block）组成的文件可能会分散在集群内的好几个节点上，这样就可以把任务调度在不同的节点上；因此，所有的单个块都是本地处理的，而不是把数据从一个节点传输到另外一个节点。当然，日志文件可以以明智的块处理方式进行处理，但是，有些文件格式不支持块处理方式。针对这种情况，你可以写一个自定义的 InputFormat，这样你就可以控制你文件是如何被拆分（或不拆分）成文件块的。

输入格式定义了组成 mapping 阶段的 map 任务列表，每一个任务对应一个输入块。接下来根据输入文件块所在的物理地址，这些任务会被分派到对应的系统节点上，可能会有多个 map 任务被分派到同一个节点上。任务分派好后，节点开始运行任务，尝试去最大并行化执行。节点上的最大任务并行数由 mapred.tasktracker.map.tasks.maximum 参数控制。

- **记录读取器（RecordReader）**

InputSplit 定义了如何切分工作，但是没有描述如何去访问它。RecordReader 类则是实际地用来加载数据，并把数据转换为适合 mapper 读取的键值对。RecordReader 实例是由输入格式定义的，默认的输入格式 TextInputFormat，提供了一个 LineRecordReader，这个类会把输入文件的每一行作为一个新的值，关联到每一行的键则是该行在文件中的字节偏移量。RecordReader 会在输入块上被重复地调用，直到整个输入块被处理完毕，每一次调用 RecordReader 都会调用 Mapper 的 map()方法。

- **Mapper**

Mapper 执行了 MapReduce 程序第一阶段中有趣的用户定义的工作。给定一个键值对，map()方法会生成一个或多个键值对，这些键值对会被送到 Reducer 那里。对于整个作业输入部分的每一个 map 任务（输入块），每一个新的 Mapper 实例都会在单独的 Java 进程中被初始化，mapper 之间不能进行通信。这就使得每一个 map 任务的可靠性不受其它 map 任务的影响，只由本地机器的可靠性来决定。

- **Partition & Shuffle**

当第一个 map 任务完成后，节点可能还要继续执行更多的 map 任务，但这时候也开始把 map 任务的中间输出交换到需要它们的 reducer 那里去，这个把“map 输出”移动到 reducer

那里去的过程就叫做 shuffle。每一个 reduce 节点会被分配得到“map 输出的键集合”中的一个不同的子集合，这些子集合（被称为“partitions”）是 reduce 任务的输入数据。每一个 map 任务生成的键值对，可能会隶属于任意的 partition，有着相同键的数值总是在一起被 reduce，不管它是来自那个 mapper 的。因此，所有的 map 节点必须就把不同的中间数据发往何处达成一致。Partitioner 类就是用来决定给定键值对的去向，默认的分类器（partitioner）会计算键的哈希值，并基于这个结果来把键赋到相应的 partition 上。

- **Reducer**

每一个 reduce 任务负责对那些关联到相同键上的所有数值进行归约（reducing），每一个节点收到的中间键集合，在被送到具体的 reducer 那里前就已经自动被 Hadoop 排序过了。每个 reduce 任务都会创建一个 Reducer 实例，这是一个用户自定义代码的实例，负责执行特定作业的第二个重要的阶段。对于每一个已被赋予到 reducer 的 partition 内的键来说，reducer 的 reduce()方法只会调用一次，它会接收一个键和关联到键的所有值的一个迭代器，迭代器会以一个未定义的顺序返回关联到同一个键的值。reducer 也要接收一个 OutputCollector 和 Report 对象，它们像在 map()方法中那样被使用。

- **输出格式**

提供给 OutputCollector 的键值对会被写到输出文件中，写入的方式由输出格式控制。OutputFormat 的功能跟前面描述的 InputFormat 类很像，Hadoop 提供的 OutputFormat 的实例会把文件写在本地磁盘或 HDFS 上，它们都是继承自公共的 FileInputFormat 类。每一个 reducer 会把结果输出写在公共文件夹中一个单独的文件内，这些文件的命名一般是 part-nnnnn，其中，nnnnn 是关联到某个 reduce 任务的 partition 的 id，输出文件夹通过 FileOutputFormat.setOutputPath()来设置。你可以通过具体 MapReduce 作业的 JobConf 对象的 setOutputFormat()方法来设置具体用到的输出格式。表 4-3 给出了 Hadoop 已提供的输出格式。

表 4-3 Hadoop 提供的输出格式

输出格式	描述
TextOutputFormat	默认的输出格式， 以 "key \t value" 的方式输出行
SequenceFileOutputFormat	输出二进制文件，适合于读取为子 MapReduce 作业的输入
NullOutputFormat	忽略收到的数据，即不做输出

Hadoop 提供了一些 OutputFormat 实例用于写入文件，基本的（默认的）实例是

TextOutputFormat，它会以一行一个键值对的方式把数据写入一个文本文件里。这样后面的 MapReduce 任务就可以通过 KeyValueInputFormat 类，简单地重新读取所需的输入数据了，而且也适合于人的阅读。还有一个更适合于在 MapReduce 作业间使用的中间格式，那就是 SequenceFileOutputFormat，它可以快速地序列化任意的数据类型到文件中，而对应 SequenceFileInputFormat 则会把文件反序列化为相同的类型并提交为下一个 Mapper 的输入数据，方式和前一个 Reducer 的生成方式一样。NullOutputFormat 不会生成输出文件并丢弃任何通过 OutputCollector 传递给它的键值对，如果你在要 reduce()方法中显式地写你自己的输出文件，并且不想 Hadoop 框架输出额外的空输出文件，那这个类是很有用的。

- **RecordWriter**

这个跟 InputFormat 中通过 RecordReader 读取单个记录的实现很相似，OutputFormat 类是 RecordWriter 对象的工厂方法，用来把单个的记录写到文件中，就像是 OuputFormat 直接写入的一样。

Reducer 输出的文件会留在 HDFS 上供你的其它应用使用，比如，另外一个 MapReduce 作业，或者一个给人工检查的单独程序。

4.4.3 Shuffle 过程详解

在 MapReduce 流程中，为了让 reduce 可以并行处理 map 结果，必须对 map 的输出进行一定的排序和分割，然后再交给对应的 reduce，而这个将 map 输出进行进一步整理并交给 reduce 的过程，就称为 shuffle。Shuffle 过程是 MapReduce 工作流程的核心，也被称为奇迹发生的地方。要想理解 MapReduce，Shuffle 是必须要了解的。

从图 4-3 中可以看出，shuffle 过程包含在 map 和 reduce 两端中，描述着数据从 map task 输出到 reduce task 输入的这段过程。在 map 端的 shuffle 过程是对 map 的结果进行划分（partition）、排序（sort）和 spill（溢写），然后，将属于同一个划分的输出合并在一起，并写到磁盘上，同时按照不同的划分将结果发送给对应的 reduce（map 输出的划分与 reduce 的对应关系由 JobTracker 确定）。reduce 端又会将各个 map 送来的属于同一个划分的输出进行合并（merge），然后对合并的结果进行排序，最后交给 reduce 处理。下面将从 map 和 reduce 两端详细介绍 shuffle 过程。

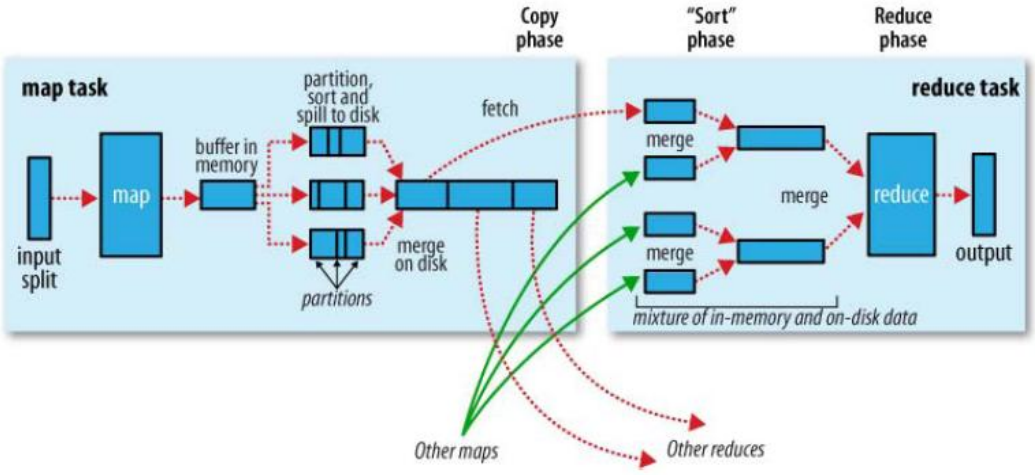


图 4-3 Shuffle 过程

4.4.3.1 map 端的 shuffle 过程

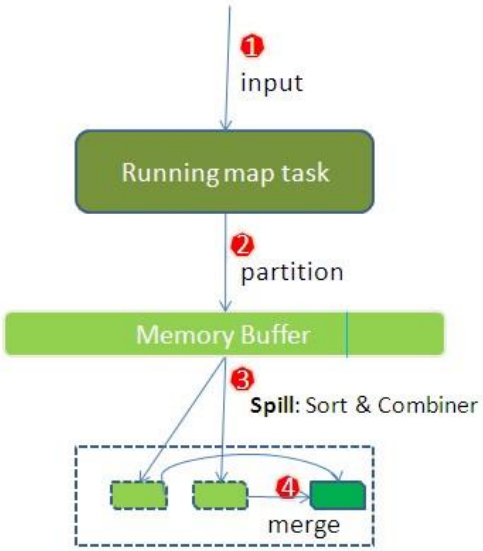


图 4-4 map 端的 shuffle 过程

图 4-4 是某个假想的 map task 的运行情况，可以清楚地说明划分（partition）、排序（sort）与合并（combiner）到底作用在 MapReduce 工作流程的哪个阶段，希望让大家清晰地了解从 map 数据输入到 map 端所有数据准备好的全过程。

整个流程可以包含四步。简单地说，每个 map task 都有一个内存缓冲区，存储着 map 的输出结果，当缓冲区快满的时候，需要将缓冲区的数据以一个临时文件的方式存放到磁盘，当整个 map task 结束后，再对磁盘中这个 map task 产生的所有临时文件做合并，生成最终的正式输出文件，然后等待 reduce task 来拉数据。当然这里的每一步都可能包含着多个步骤与细节，下面对每个步骤的细节进行说明。

第 1 步：在 map task 执行时，它的输入数据来源于 HDFS 的 block，当然在 MapReduce 概念中，map task 只读取 split。Split 与 block 的对应关系可能是多对一，默认是一对一。在 WordCount 例子里，假设 map 的输入数据都是像“aaa”这样的字符串。

第 2 步：在经过 mapper 的运行后，我们得知 mapper 的输出是这样一个 key/value 对：key 是“aaa”，value 是数值 1。注意，当前 map 端只做加 1 的操作，在 reduce task 里才去合并结果集。假设这个 job 有 3 个 reduce task，到底当前的“aaa”应该交由哪个 reduce 去做呢，是需要现在决定的。

MapReduce 提供 Partitioner 接口，它的作用就是根据 key 或 value 及 reducer 的数量来决定当前的这个“键值对”输出数据最终应该交由哪个 reduce task 处理。默认对 key 进行哈希以后再用 reduce task 数量进行取模，即 hash(key) mod R，其中 R 表示 reducer 的数量。默认的取模方式只是为了平均 reduce 的处理能力，如果用户自己对 Partitioner 有需求，可以订制并设置到 job 上。

假设在我们的例子中，“aaa”经过 Partitioner 后返回 0，也就是这个“键值对”应当交由第一个 reducer 来处理。接下来，需要将数据写入内存缓冲区中，缓冲区的作用是批量收集 map 结果，减少磁盘 IO 的影响。我们的 key/value 对以及 Partition 的结果都会被写入缓冲区。当然写入之前，key 与 value 值都会被序列化成字节数组。整个内存缓冲区就是一个字节数组。

第 3 步：这个内存缓冲区是有大小限制的，默认是 100MB。当 map task 的输出结果很多时，就可能会撑爆内存，所以，需要在一定条件下将缓冲区中的数据临时写入磁盘，然后重新利用这块缓冲区。这个从内存往磁盘写数据的过程被称为 Spill，中文可译为“溢写”，字面意思很直观。这个溢写是由单独线程来完成，不影响往缓冲区写 map 结果的线程。溢写线程启动时，不应该阻止 map 的结果输出，所以，整个缓冲区有个溢写的比例 spill.percent。这个比例默认是 0.8，也就是当缓冲区的数据已经达到阈值（buffer size * spill percent = 100MB * 0.8 = 80MB），溢写线程启动，锁定这 80MB 的内存，执行溢写过程。Map task 的输出结果还可以往剩下的 20MB 内存中写，互不影响。

当溢写线程启动后，需要对这 80MB 空间内的 key 做排序(Sort)。排序是 MapReduce 模型默认的行为，这里的排序也是对序列化的字节做的排序。

另外，为了进一步减少从 map 端到 reduce 端需要传输的数据量，还可以在 map 端执行 combine 操作。对于 WordCount 例子，就是简单地统计单词出现的次数，如果在同一个 map task 的结果中有很多个像“aaa”一样出现多次的 key，我们就应该把它们的值合并到一块，

这个过程叫 reduce，也叫 combine。比如，有些 map 输出数据可能像这样：<”aaa”, 1>，<”aaa”, 1>，经过 combine 操作以后就可以得到<”aaa”, 2>。但是，在 MapReduce 的术语中，reduce 单纯是指 reduce 端执行从多个 map task 取数据做计算的过程。除 reduce 外，非正式地合并数据只能算做 combine 了。实际上，MapReduce 中将 Combiner 等同于 Reducer。

如果 client 设置过 Combiner，那么现在就是使用 Combiner 的时候了。将有相同 key 的 key/value 对的 value 加起来，减少溢写到磁盘的数据量。Combiner 会优化 MapReduce 的中间结果，所以它在整个模型中会多次使用。那么，哪些场景才能使用 Combiner 呢？从这里分析，Combiner 的输出是 Reducer 的输入，Combiner 绝不能改变最终的计算结果。所以，一般而言，Combiner 只应该用于那种 Reduce 的输入 key/value 与输出 key/value 类型完全一致、且不影响最终结果的场景，比如累加、最大值等。Combiner 的使用一定得慎重，如果用好，它对 job 执行效率有帮助，反之，则会影响 reduce 的最终结果。

第 4 步：每次溢写会在磁盘上生成一个溢写文件，如果 map 的输出结果真的很大，有多次这样的溢写发生，磁盘上相应的就会有多个溢写文件存在。当 map task 真正完成时，内存缓冲区中的数据也全部溢写到磁盘中形成一个溢写文件。最终，磁盘中会至少有一个这样的溢写文件存在(如果 map 的输出结果很少，当 map 执行完成时，只会产生一个溢写文件)，因为最终的文件只允许有一个，所以需要将这些溢写文件归并到一起，这个过程就叫做 Merge。Merge 是怎样的？如前面的例子，“aaa”从某个 map task 读取过来时值是 5，从另外一个 map 读取时值是 8，因为它们有相同的 key，所以得 merge 成 group。什么是 group？对于“aaa”而言，merge 后得到的 group 就是像这样的：<”aaa”, {5, 8, 2, …}>，数组中的值就是从不同溢写文件中读取出来的，然后再把这些值合并起来。请注意，因为 merge 是将多个溢写文件合并到一个文件，所以可能也有相同的 key 存在，在这个过程中如果 client 设置过 Combiner，也会使用 Combiner 来合并相同的 key。

至此，map 端的所有工作都已结束，最终生成的这个文件也存放在 TaskTracker 够得着的某个本地目录内。每个 reduce task 不断地通过 RPC 从 JobTracker 那里获取 map task 是否完成的信息，如果 reduce task 得到通知，获知某台 TaskTracker 上的 map task 执行完成，Shuffle 的后半段过程开始启动。

4.4.3.2 reduce 端的 shuffle 过程

简单地说，reduce task 在执行之前的工作就是不断地拉取当前 job 里每个 map task 的最

终结果，然后对从不同地方拉取过来的数据不断地做 merge，也最终形成一个文件作为 reduce task 的输入文件（如图 4-5 所示）。

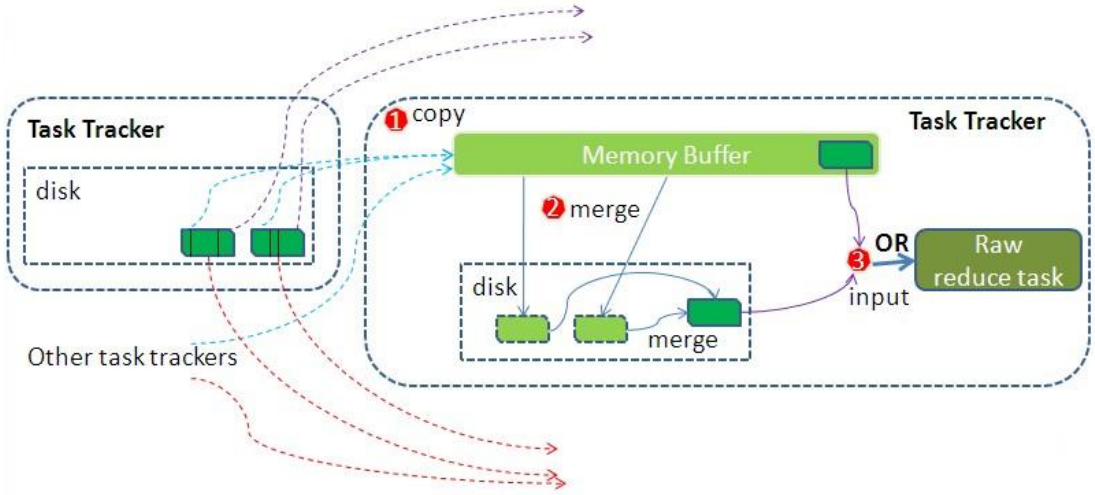


图 4-5 reduce 端的 shuffle 过程

和 map 端的细节图（图 4-4）一样，Shuffle 在 reduce 端的过程也能用图 4-5 上标明的 3 个步骤来概括。当前 reduce 拉取数据的前提是，它要从 JobTracker 那里获知有哪些 map task 已执行结束。Reducer 真正运行之前，所有的时间都是在拉取数据，做 merge，且不断重复地在做。如前面的方式一样，下面分成 3 步描述 reduce 端的 Shuffle 细节。

第 1 步：复制过程，简单地拉取数据。Reduce 进程启动一些数据复制线程(Fetcher)，通过 HTTP 方式请求 map task 所在的 TaskTracker 获取 map task 的输出文件。因为 map task 早已结束，这些文件就归 TaskTracker 管理在本地磁盘中。

第 2 步：Merge 阶段。这里的 merge 如 map 端的 merge 动作，只是数组中存放的是不同 map 端复制来的数值。复制过来的数据会先放入内存缓冲区中，这里的缓冲区大小要比 map 端的更为灵活，因为 Shuffle 阶段 Reducer 不运行，所以应该把绝大部分的内存都给 Shuffle 用。这里需要强调的是，merge 有三种形式：1)内存到内存；2)内存到磁盘；3)磁盘到磁盘。默认情况下第一种形式不启用，让人比较困惑，是吧。当内存中的数据量到达一定阈值，就启动内存到磁盘的 merge。与 map 端类似，这也是溢写的过程，这个过程中如果你设置有 Combiner，也是会启用的，然后在磁盘中生成了众多的溢写文件。第二种 merge 方式一直在运行，直到没有 map 端的数据时才结束，然后启动第三种磁盘到磁盘的 merge 方式生成最终的那个文件。

第 3 步：Reducer 的输入文件。不断地 merge 后，最后会生成一个“最终文件”。为什么加引号？因为这个文件可能存在于磁盘上，也可能存在于内存中。对我们来说，当然希望

它存放于内存中，直接作为 Reducer 的输入，但默认情况下，这个文件是存放于磁盘中的。当 Reducer 的输入文件已定，整个 Shuffle 才最终结束。然后就是 Reducer 执行，把结果放到 HDFS 上。

4.5并行计算的实现

MapReduce 计算模型非常适合在大量计算机组成的大规模集群上并行运行。图 4-1 中的每一个 Map 任务和每一个 Reduce 任务均可以同时运行于一个单独的计算节点上，可想而知，其运算效率是很高的，那么这样的并行计算是如何做到的呢？实际上，这里涉及到了三个方面的内容（见图 4-6）：数据分布存储、分布式并行计算和本地计算。这三者共同合作，完成并行分布式计算的任务。

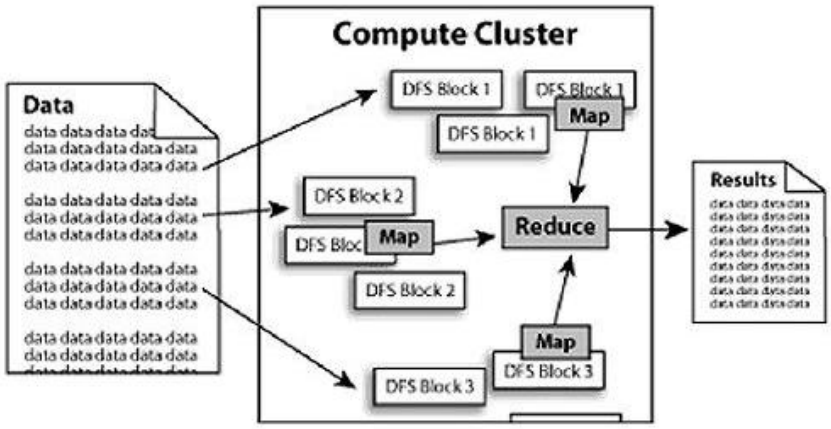


图 4-6 分布存储与并行计算

4.5.1 数据分布存储

如图 4-7 所示，Hadoop 中的分布式文件系统 HDFS 由一个管理节点(NameNode)和 N 个数据节点(DataNode)组成，每个节点均是一台普通的计算机。在使用上同我们熟悉的单机上的文件系统非常类似，一样可以建目录、创建、复制、删除文件、查看文件内容等。但其底层实现上是把文件切割成 Block，然后这些 Block 分散地存储于不同的 DataNode 上，每个 Block 还可以复制数份存储于不同的 DataNode 上，达到容错容灾之目的。NameNode 则是整个 HDFS 的核心，它通过维护一些数据结构，记录了每一个文件被切割成了多少个 Block，这些 Block 可以从哪些 DataNode 中获得，各个 DataNode 的状态等重要信息。

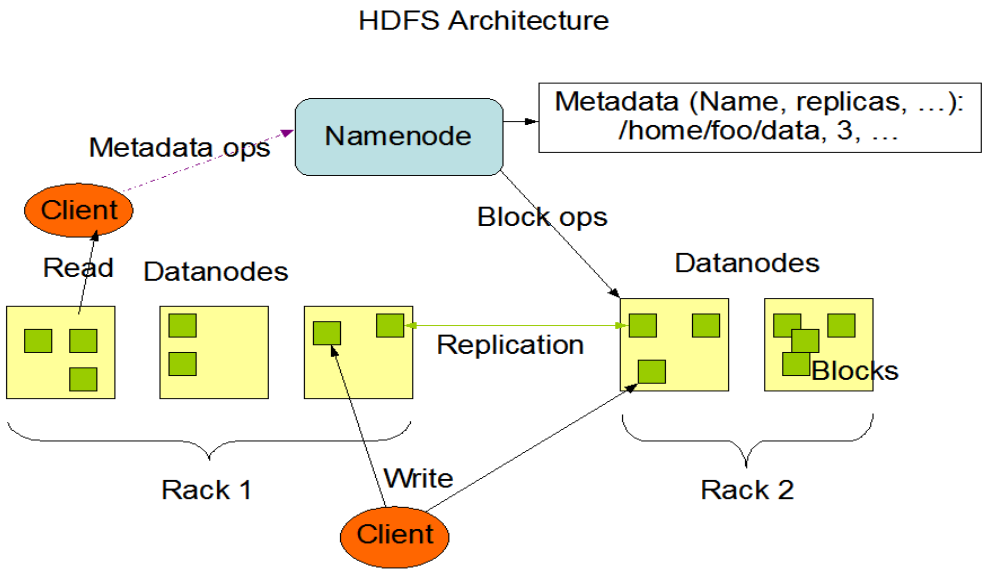


图 4-7 HDFS 的体系结构

4.5.2 分布式并行计算

Hadoop 中有一个作为主控的 JobTracker（见图 4-8），用于调度和管理其它的 TaskTracker，JobTracker 可以运行于集群中任一台计算机上。TaskTracker 负责执行任务，必须运行于 DataNode 上，即 DataNode 既是数据存储节点，也是计算节点。JobTracker 将 Map 任务和 Reduce 任务分发给空闲的 TaskTracker，让这些任务并行运行，并负责监控任务的运行情况。如果某一个 TaskTracker 出故障了，JobTracker 会将其负责的任务转交给另一个空闲的 TaskTracker 重新运行。

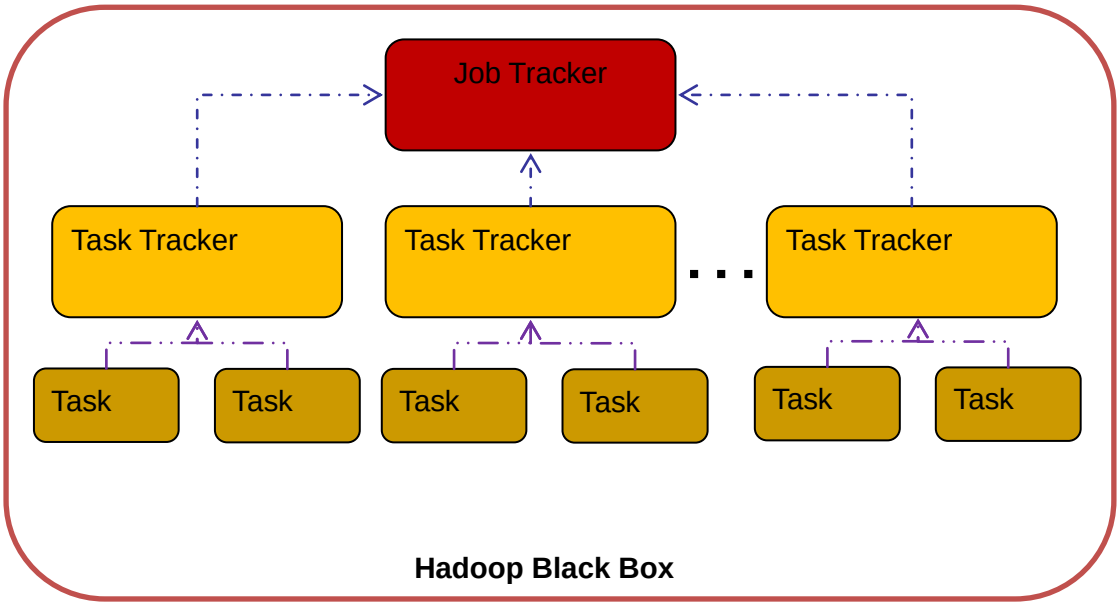


图 4-8 Hadoop Job Tracker

在进行 MapReduce 的任务调度时，首先要保证 master 节点的 NameNode、SecondaryNameNode、JobTracker 和 slaves 节点的 DataNode、TaskTracker 都已经启动。通常来说，MapRedcue 作业是通过 JobClient.rubJob(job)方法向 master 节点的 JobTracker 提交的，JobTracker 接到 JobClient 的请求后把其加入作业队列中。JobTracker 一直在等待 JobClient 通过 RPC 向其提交作业，而 TaskTracker 一直通过 RPC 向 JobTracker 发送心跳信号询问有没有任务可做，如果有，则请求 JobTracker 派发任务给它执行。如果 JobTracker 的作业队列不为空，则 TaskTracker 发送的心跳将会获得 JobTracker 给它派发的任务。这是一个主动请求的任务，slave 的 TaskTracker 主动向 master 的 JobTracker 请求任务。当 TaskTracker 接到任务后，通过自身调度在本 slave 建立起 Task 执行任务。图 4-9 是 MapReduce 任务请求调度的过程示意图。

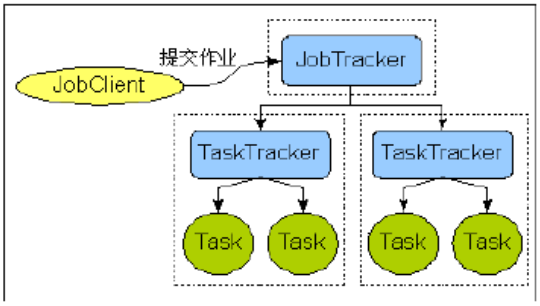


图 4-9 MapReduce 的任务调度

具体来说，MapReduce 任务请求调度过程包括两个步骤：

（1）JobClient 提交作业

JobClient.runJob(job)静态方法会实例化一个 JobClient 的实例，然后用此实例的 submitJob(job)方法向 JobTracker 提交作业。此方法会返回一个 RunningJob 对象，它用来跟踪作业的状态。作业提交完毕后，JobClient 会根据此对象开始关注作业的进度，直到作业完成。submitJob(job)内部是通过调用 submitJobInternal(job)方法完成实质性的作业提交的。submitJobInternal(job)方法首先会向 Hadoop 分布系统文件系统（HDFS）依次上传三个文件 job.jar、job.split 和 job.xml。job.jar 里面包含了执行此任务需要的各种类，比如 Mapper、Reducer 等实现；job.split 是文件分块的相关信息，比如有数据分多少个块、块的大小(默认 64M)等。job.xml 是有关的作业配置，例如 Mapper,Combiner,Reducer 的类型，输入输出格式的类型等。

（2）JobTacker 调度作业

JobTracker 接到 JobClient 提交的作业后，即在 JobTracker.submitJob(job)方法中，首先产

生一个 JobInProgress 对象。此对象代表一道作业，它的作用是维护这道作业的所有信息，包括作业相关信息 JobProfile 和最近作业状态 JobStatus，并将作业所有规划的 Task 登记到任务列表中。随后 JobTracker 将此 JobInProgress 对象通过 listener.jobAdded(job)方法加入到调度队列中，并用一个成员变量 jobs 来维护所有的作业。然后等到有 TaskTracker 空闲，使用 JobTracker.AssignTask(tasktracker）来请求任务，如果调度队列不空，程序便通过调度算法取出一个 task 交给来请求的 TaskTracker 去执行。至此，整个任务分配过程基本完成。

4.5.3 本地计算

数据存储在哪一台计算机上，就由这台计算机进行这部分数据的计算，这样可以减少数据在网络上的传输，降低对网络带宽的需求。在 Hadoop 这样的基于集群的分布式并行系统中，计算节点可以很方便地扩充，因它所能够提供的计算能力近乎是无限的。但是，由是数据需要在不同的计算机之间流动，故而网络带宽变成了瓶颈，是非常宝贵的，因此，“本地计算”是最有效的一种节约网络带宽的手段，业界把这形容为“移动计算比移动数据更经济”。

4.5.4 任务粒度

把原始大数据集切割成小数据集时，通常让小数据集小于或等于 HDFS 中一个 Block 的大小(缺省是 64M)，这样能够保证一个小数据集位于一台计算机上，便于本地计算。有 M 个小数据集待处理，就启动 M 个 Map 任务，注意这 M 个 Map 任务分布于 N 台计算机上并行运行，Reduce 任务的数量 R 则可由用户指定。

4.5.5 Partition

Partition 是选择配置，主要作用是在多个 Reduce 的情况下，指定 Map 的结果由某一个 Reduce 处理，每一个 Reduce 都会有单独的输出文件。把 Map 任务输出的中间结果按 key 的范围划分成 R 份(R 是预先定义的 Reduce 任务的个数)，划分时通常使用 hash 函数，如: hash(key) mod R，这样可以保证某一段范围内的 key，一定是由一个 Reduce 任务来处理，可以简化 Reduce 的过程。

4.5.6 Combine

在 partition 之前，还可以对中间结果先做 combine，即将中间结果中有相同 key 的 <key，value>对合并成一对。Combiner 是可选择的，它的主要作用是在每一个 Map 执行完分析以后，在本地优先做 Reduce 的工作，减少在 Reduce 过程中的数据传输量。combine 的过程与 Reduce 的过程类似，很多情况下就可以直接使用 Reduce 函数，但 combine 是作为 Map 任务的一部分，在执行完 Map 函数后紧接着执行的。Combine 能够减少中间结果中<key，value>对的数目，从而减少网络流量。

4.5.7 Reduce 任务从 Map 任务节点取中间结果

Map 任务的中间结果在做完 Combine 和 Partition 之后，以文件形式存于本地磁盘。中间结果文件的位置会通知主控 JobTracker，JobTracker 再通知 Reduce 任务到哪一个 DataNode 上去取中间结果。注意所有的 Map 任务产生中间结果均按其 Key 用同一个 Hash 函数划分成了 R 份，R 个 Reduce 任务各自负责一段 Key 区间。每个 Reduce 需要向许多个 Map 任务节点取得落在其负责的 Key 区间内的中间结果，然后执行 Reduce 函数，形成一个最终的结果文件。

4.5.8 任务管道

有 R 个 Reduce 任务，就会有 R 个最终结果，很多情况下这 R 个最终结果并不需要合并成一个最终结果。因为这 R 个最终结果又可以做为另一个计算任务的输入，开始另一个并行计算任务。

4.6 实例分析：WordCount

如果想统计过去 10 年计算机论文中出现次数最多的几个单词，看看大家都在研究些什么，那收集好论文后，该怎么办呢？可以大致采用以下几种方法：

- 方法一：可以写一个小程序，把所有论文按顺序遍历一遍，统计每一个遇到的单词的出现次数，最后就可以知道哪几个单词最热门了。这种方法在数据集比较小时，是非常有效的，而且实现最简单，用来解决这个问题很合适。

- 方法二：写一个多线程程序，并发遍历论文。这个问题理论上是可以高度并发的，因为统计一个文件时不会影响统计另一个文件。当我们的机器是多核或者多处理器，方法二肯定比方法一高效。但是，写一个多线程程序要比方法一困难多了，我们必须自己同步共享数据，比如要防止两个线程重复统计文件。
- 方法三：把作业交给多个计算机去完成。我们可以使用方法一的程序，部署到 N 台机器上去，然后把论文集分成 N 份，一台机器跑一个作业。这个方法跑得足够快，但是部署起来很麻烦，我们要人工把程序复制分发到别的机器，要人工把论文集分开，最痛苦的是还要把 N 个运行结果进行整合（当然我们也可以再写一个程序）。
- 方法四：让 MapReduce 来帮帮我们吧！MapReduce 本质上就是方法三，但是如何拆分文件集，如何复制分发程序，如何整合结果，这些都是框架定义好的。我们只要定义好这个任务（用户程序），其它都交给 MapReduce 去处理。

4.6.1 WordCount 设计思路

WordCount 例子如同 Java 中的“HelloWorld”经典程序一样是 MapReduce 的入门程序。计算出文件中各个单词的频数，要求输出结果按照单词的字母顺序进行排序，每个单词和其频数占一行，单词和频数之间有间隔。比如，输入一个文件，其内容如下：

hello world

hello hadoop

hello mapreduce

对应上面给出的输入样例，其输出样例为：

hadoop 1

hello 3

mapreduce 1

world 1

上面这个应用实例的解决方案很直接，就是将文件内容切分成单词。然后将所有相同的单词聚集到一起，最后，计算单词出现的次数进行输出。针对 MapReduce 并行程序设计原则可知，解决方案中的内容切分步骤和数据不相关，可以并行化处理，每个拿到原始数据的机器只要将输入数据切分成单词就可以了。所以，可以在 map 阶段完成单词切分的任务。另外，相同单词的频数计算也可以并行化处理。根据实例要求来看，不同单词之间的频数不

相关，所以，可以将相同的单词交给一台机器来计算频数，然后输出最终结果。这个过程可以交给 reduce 阶段完成。至于将中间结果根据不同单词进行分组后再发送给 reduce 机器，这正好是 MapReduce 过程中的 shuffle 能够完成的。至此，这个实例的 MapReduce 程序就设计出来的。Map 阶段完成由输入数据到单词切分的工作，shuffle 阶段完成相同单词的聚集和分发工作（这个过程是 MapReduce 的默认过程，不用具体配置），reduce 阶段完成接收所有单词并计算其频数的工作。由于 MapReduce 中传递的数据都是<key, value>形式的，并且 shuffle 排序聚集分发是按照 key 值进行的，所以，将 map 的输出设计成由 word 作为 key，1 作为 value 的形式，它表示单词出现了 1 次（map 的输入采用 Hadoop 默认的输入方式，即文件的一行作为 value，行号作为 key）。Reduce 的输入是 map 输出聚集后的结果，即<key, value-list>，具体到这个实例就是<word, {1,1,1,1,…}>，reduce 的输出会设计成与 map 输出相同的形式，只是后面的数值不再是固定的 1，而是具体算出的 word 所对应的频数。

4.6.2 WordCount 代码

采用 MapReduce 进行词频统计的 WordCount 代码如下：

```
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> <out>");
            System.exit(2);
        }
        Job job = new Job(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);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

4.6.3 过程解释

map 操作的输入是<key, value>形式，其中，key 是文档中某行的行号，value 是该行的内容。map 操作会将输入文档中每一个单词的出现输出到中间文件中去（如图 4-10 所示）。

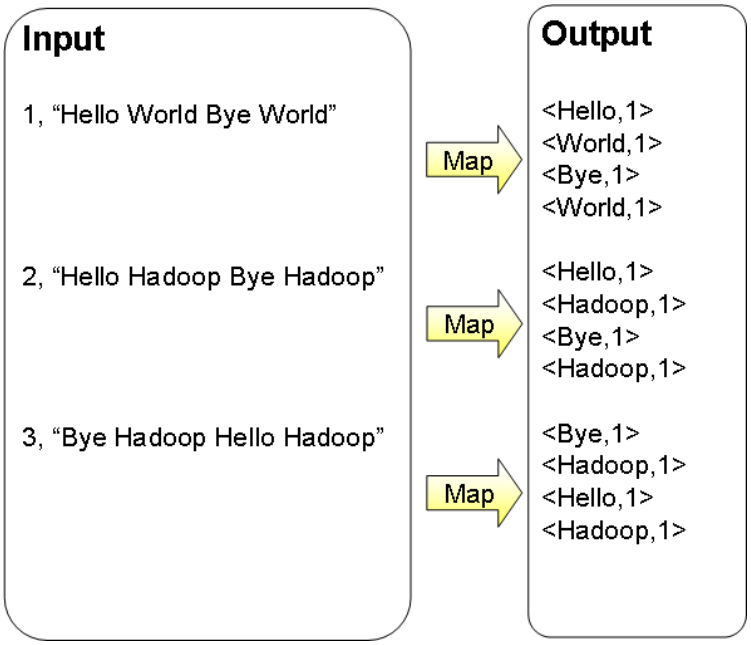


图 4-10 Map 过程示意图

Reduce 操作的输入是单词和出现次数的序列（如图 4-11 所示）。用上面的例子来说，就是 <“Hello”， [1,1,1]>， <“World”，[1,1]>， <“Bye”，[1,1,1]>， <“Hadoop”， [1,1,1,1]> 等。然后根据每个单词，算出总的出现次数。

最后输出排序后的最终结果就会是：<“Bye”，3>，<“Hadoop”，4> ，<“Hello”，3>，<“World”，2>。

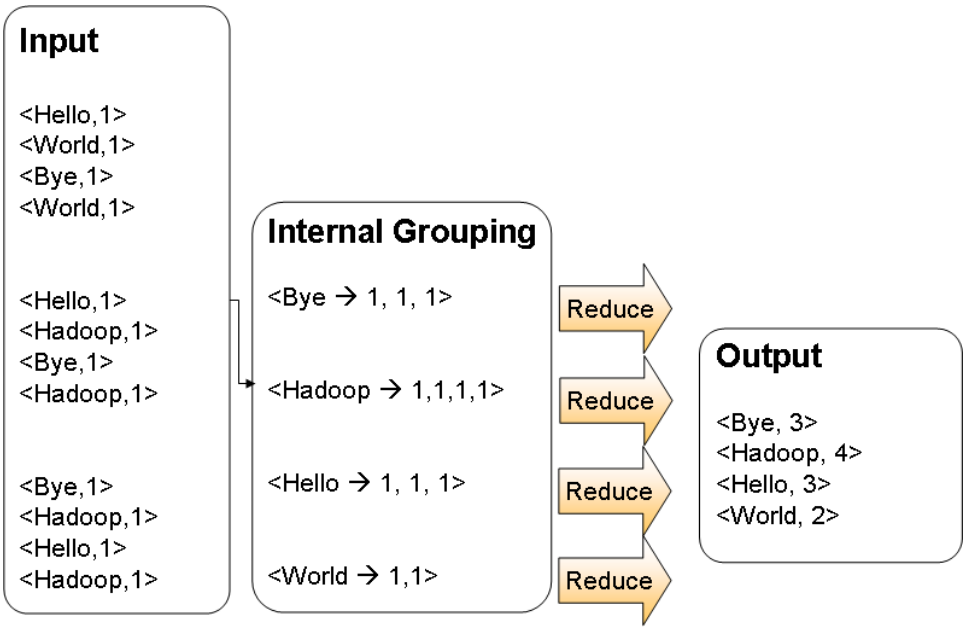


图 4-11 Reduce 过程示意图

整个 MapReduce 过程实际的执行顺序是：

- MapReduce Library 将 Input 分成 M 份，这里的 Input Splitter 也可以是多台机器并

行 Split；

- Master 将 M 份 Job 分给空闲状态的 M 个 worker 来处理；
- 对于输入中的每一个<key， value> 进行 Map 操作，将中间结果缓冲在内存里；
- 定期地（或者根据内存状态）将缓冲区中的中间信息刷写到本地磁盘上，并且把文件信息传回给 Master（Master 需要把这些信息发送给 Reduce worker）。这里最重要的一点是，在写磁盘的时候，需要将中间文件做 Partition（比如 R 个）。拿上面的例子来举例，如果把所有的信息存到一个文件，Reduce worker 又会变成瓶颈。我们只需要保证相同 Key 能出现在同一个 Partition 里面就可以把这个问题分解。
- R 个 Reduce worker 开始工作，从不同的 Map worker 的 Partition 那里拿到数据，用 key 进行排序（如果内存中放不下需要用到外部排序）。很显然，排序（或者说 Group）是 Reduce 函数之前必须做的一步。这里面很关键的是，每个 Reduce worker 会去从很多 Map worker 那里拿到 X(0<X<R) Partition 的中间结果，这样，所有属于这个 Key 的信息已经都在这个 worker 上了。
- Reduce worker 遍历中间数据，对每一个唯一 Key，执行 Reduce 函数（参数是这个 key 以及相对应的一系列 Value）。
- 执行完毕后，唤醒用户程序，返回结果（最后应该有 R 份 Output，每个 Reduce Worker 一个）。

可见，这里的“分”（Divide）体现在两步，分别是将输入分成 M 份，以及将 Map 的中间结果分成 R 份。将输入分开通常很简单，而把 Map 的中间结果分成 R 份，通常用”hash(key) mod R”这个结果作为标准，保证相同的 Key 出现在同一个 Partition 里面。当然，使用者也可以指定自己的 Partition 函数，比如，对于 Url Key，如果希望同一个 Host 的 URL 出现在同一个 Partition，可以用”hash(Hostname(urlkey)) mod R”作为 Partition 函数。

另外，对于上面的例子来说，每个文档中都可能会出现成千上万的 <“the”，1>这样的中间结果，琐碎的中间文件必然导致传输上的开销。因此，MapReduce 还支持用户提供 Combiner 函数。这个函数通常与 Reduce 函数有相同的实现，不同点在于 Reduce 函数的输出是最终结果，而 Combiner 函数的输出是 Reduce 函数的输入。图 4-10 中的 map 过程输出结果，如果采用 Combiner 函数后，则可以得到如图 4-12 所示的输出，这个输出结果可以作为 reduce 过程的输入（如图 4-13 所示）。

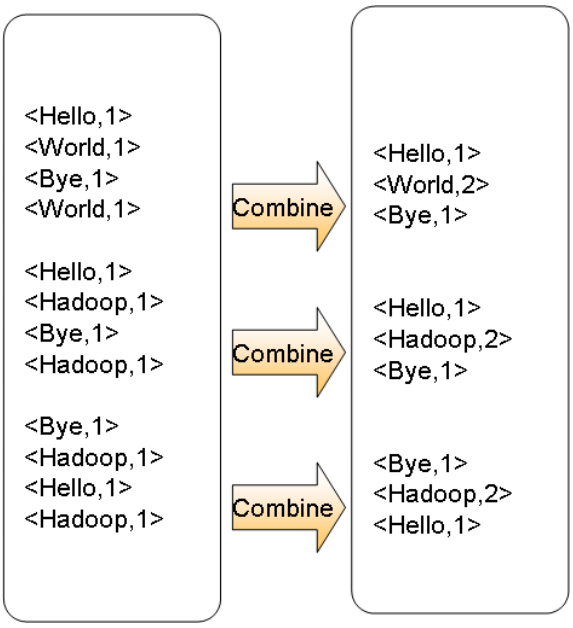


图 4-12 Combine 过程示意图

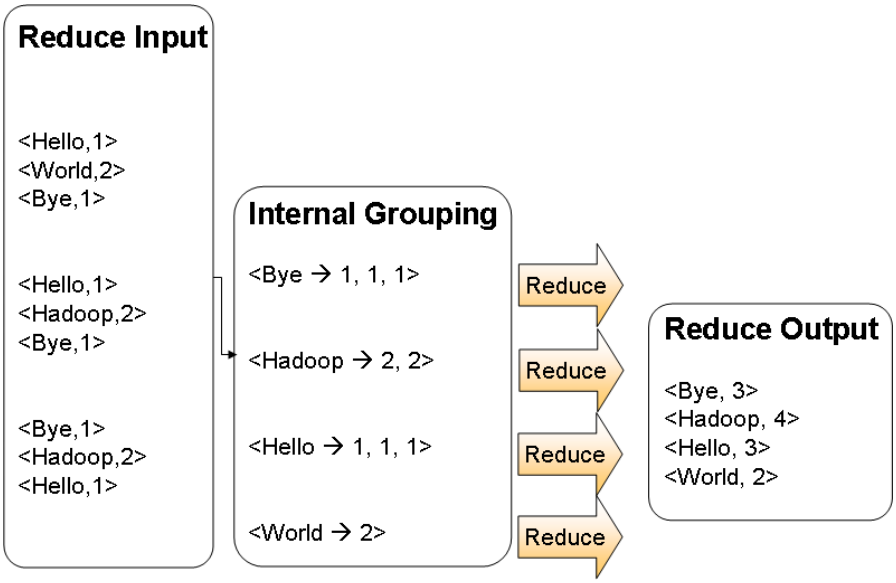


图 4-13 以 Combine 输出结果作为输入的 Reduce 过程示意图

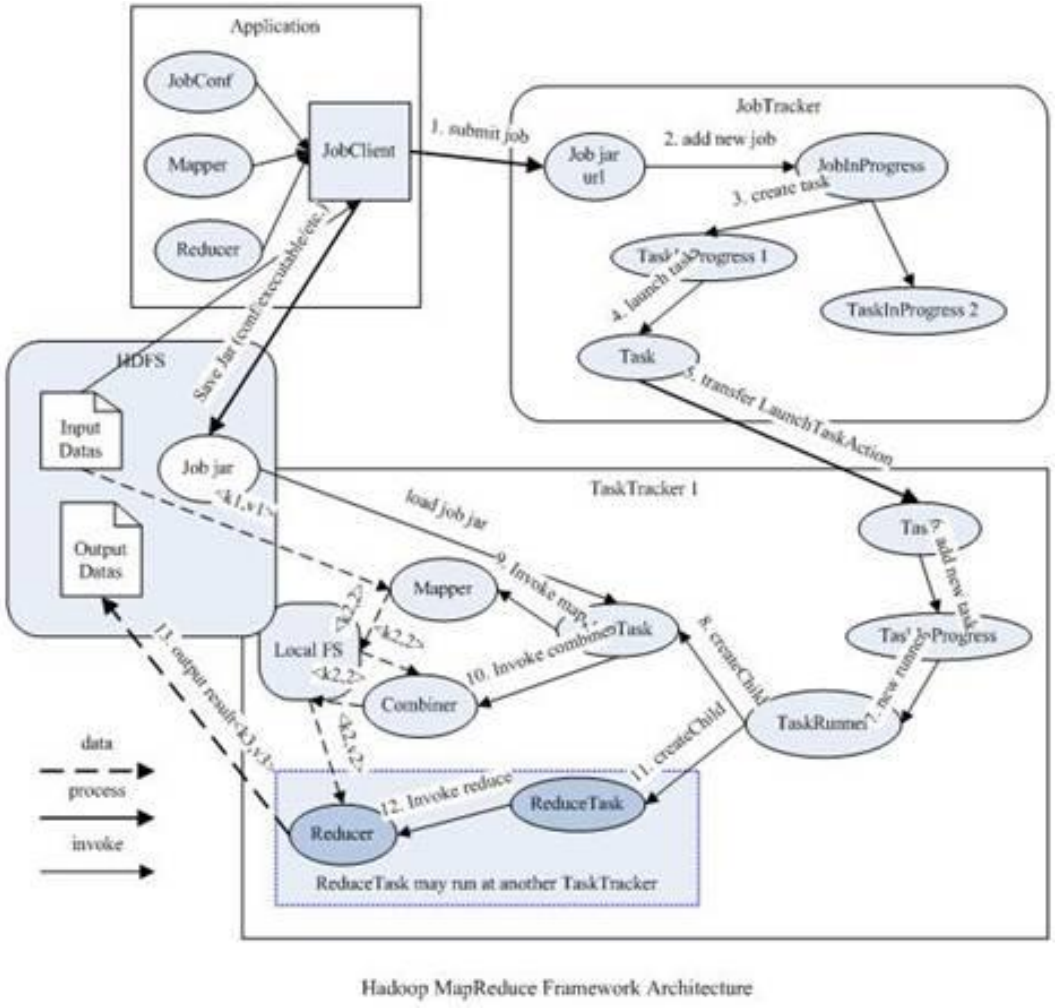
4.7新 MapReduce 框架 Yarn

Yarn 带来了巨大的改变，改变了 Hadoop 计算组件（MapReduce）切分和重新组成处理任务的方式，因为，Yarn 把 MapReduce 的追踪组件切分成两个不同部分：资源管理器和应用调度。这样的数据管理工具，有助于更加轻松地同时运行 MapReduce 或 Storm 这样的任务以及 HBase 等服务。Hadoop 共同创始人之一 Doug Cutting 表示：“它使得其他不是 MapReduce 的工作负载现在可以更有效地与 MapReduce 分享资源。现在这些系统可以动态地分享资源，资源也可以设置优先级”。

Cutting 和 Bhandarkar 都承认，这种方法是受到了 Apache 项目“Mesos”集群管理系统以及谷歌 Borg 和 Omega 秘密项目的一些影响。

Yarn 的出现，使得 Hadoop 变成一个针对数据中心的操作系统，支持广泛的应用。

4.7.1 原 Hadoop MapReduce 框架的问题



Hadoop MapReduce Framework Architecture

图 4-14 原 Hadoop MapReduce 架构

从图 4-14 中可以清楚地看出原 MapReduce 程序的流程及设计思路：

- 首先用户程序 (JobClient) 提交了一个 job，job 的信息会发送到 Job Tracker 中，Job Tracker 是 MapReduce 框架的中心，它需要与集群中的机器定时通信 (heartbeat)，需要管理哪些程序应该跑在哪些机器上，需要管理所有 job 失败、重启等操作。
- TaskTracker 是 MapReduce 集群中每台机器都有的一个部分，它做的事情主要是监视自己所在机器的资源情况。
- TaskTracker 同时监视当前机器的 tasks 运行状况。TaskTracker 需要把这些信息通

过 heartbeat 发送给 JobTracker，JobTracker 会搜集这些信息以确定新提交的 job 分配运行在哪些机器上。图 4-14 虚线箭头就是表示消息的“发送—接收”的过程。

可以看得出，原来的 MapReduce 架构是简单明了的，在最初推出的几年，也得到了众多的成功案例，获得业界广泛的支持和肯定。但是，随着分布式系统集群的规模和其工作负荷的增长，原框架的问题逐渐浮出水面，主要的问题集中如下：

- JobTracker 是 MapReduce 的集中处理点，存在单点故障。
- JobTracker 完成了太多的任务，造成了过多的资源消耗，当 MapReduce job 非常多的时候，会造成很大的内存开销，潜在来说，也增加了 JobTracker 失败的风险，这也是业界普遍总结出来的一个结论,即老 Hadoop 的 MapReduce 只能支持 4000 节点主机的上限。
- 在 TaskTracker 端，以 MapReduce 任务的数目作为资源的表示过于简单，没有考虑到 CPU 和内存的占用情况，如果两个大内存消耗的任务被调度到了一块，很容易出现 OOM（Out of Memory）。
- 在 TaskTracker 端，把资源强制划分为 map task slot 和 reduce task slot, 如果当系统中只有 map task 或者只有 reduce task 的时候，会造成资源的浪费，也就是前面提过的集群资源利用的问题。
- 源代码层面分析的时候，会发现代码非常难读，常常因为一个类（class）做了太多的事情，代码量达 3000 多行，造成类的任务不清晰，增加 bug 修复和版本维护的难度。
- 从操作的角度来看，现在的 Hadoop MapReduce 框架在有任何重要的或者不重要的变化时(例如 bug 修复，性能提升和特性化)，都会强制进行系统级别的升级更新。更糟的是，它不管用户的喜好，强制让分布式集群系统的每一个用户端同时更新。这些更新会让用户为了验证他们之前的应用程序是不是适用新的 Hadoop 版本而浪费大量时间。

4.7.2 新 Hadoop Yarn 框架原理及运作机制

从业界使用分布式系统的变化趋势和 Hadoop 框架的长远发展来看，MapReduce 的 JobTracker/TaskTracker 机制需要大规模的调整来修复它在可扩展性、内存消耗、线程模型、可靠性和性能上的缺陷。在过去的几年中，Hadoop 开发团队做了一些 bug 的修复，但是，

最近这些修复的成本越来越高，这表明对原框架做出改变的难度越来越大。

为从根本上解决旧 MapReduce 框架的性能瓶颈，促进 Hadoop 框架的更长远发展，从 0.23.0 版本开始，Hadoop 的 MapReduce 框架完全重构，发生了根本的变化。新的 Hadoop MapReduce 框架命名为 MapReduceV2 或者叫 Yarn，其架构图如图 4-15 所示。

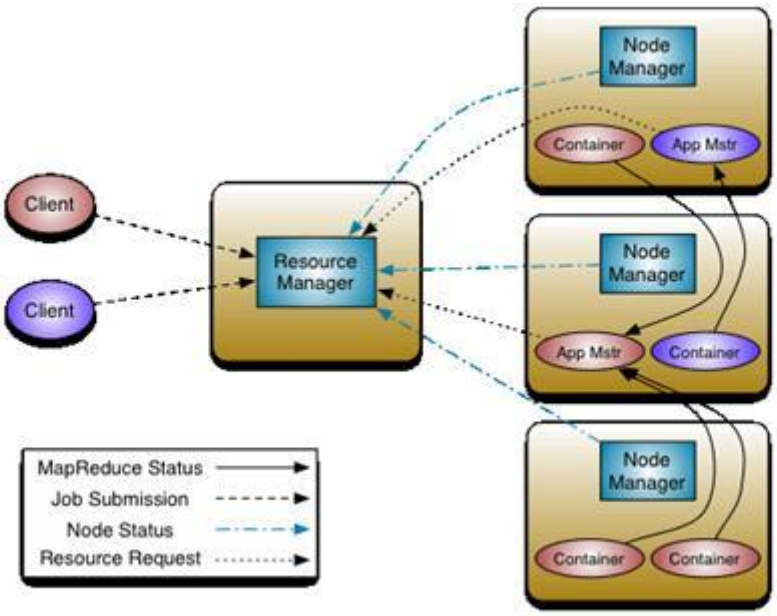


图 4-15 新的 Hadoop MapReduce 框架（Yarn）架构

重构根本的思想是将 JobTracker 两个主要的功能分离成单独的组件，这两个功能是资源管理和任务调度/监控。新的资源管理器全局管理所有应用程序计算资源的分配，每一个应用的 ApplicationMaster 负责相应的调度和协调。一个应用程序无非是一个单独的传统的 MapReduce 任务或者是一个 DAG(有向无环图) 任务。ResourceManager 和每一台机器的节点管理服务器能够管理用户在那台机器上的进程并能对计算进行组织。

事实上，每一个应用的 ApplicationMaster 是一个详细的框架库，它结合从 ResourceManager 获得的资源，和 NodeManager 协同工作来运行和监控任务。

图 4-15 中 ResourceManager 支持分层级的应用队列，这些队列享有集群一定比例的资源。从某种意义上讲，它就是一个纯粹的调度器，它在执行过程中不对应用进行监控和状态跟踪。同样，它也不能重启因应用失败或者硬件错误而运行失败的任务。

ResourceManager 是基于应用程序对资源的需求进行调度的；每一个应用程序需要不同类型的资源，因此就需要不同的容器。资源包括内存、CPU、磁盘、网络等等。可以看出，这同原来的 MapReduce 中固定类型的资源使用模型有显著区别，原来的那种模型给集群的使用会带来负面的影响。资源管理器提供一个调度策略的插件，它负责将集群资源分配给多

个队列和应用程序。调度插件可以基于现有的能力调度和公平调度模型。

图 4-15 中 NodeManager 是每一台机器框架的代理，是执行应用程序的容器，监控应用程序的资源使用情况 (CPU、内存、硬盘、网络) 并且向调度器汇报。

每一个应用的 ApplicationMaster 的职责有：向调度器索要适当的资源容器，运行任务，跟踪应用程序的状态和监控它们的进程，处理任务的失败原因。

4.7.3 新旧 Hadoop MapReduce 框架比对

（1）新旧 MapReduce 框架的变化

新的 Yarn 框架相对旧 MapRduce 框架而言，其配置文件、启停脚本及全局变量等发生了一些变化。

新旧 MapReduce 框架的客户端没有发生变化，其调用 API 及接口，大部分保持兼容，这也是为了对开发使用者透明化，使其不必对原有代码做大的改变。但是，原框架中核心的 JobTracker 和 TaskTracker 不见了，取而代之的是 ResourceManager、ApplicationMaster 与 NodeManager 三个部分，三者的具体功能如下：

- **ResourceManager**：ResourceManager 是一个中心的服务，它做的事情是调度、启动每一个 Job 所属的 ApplicationMaster，并且监控 ApplicationMaster 的存在情况。细心的读者会发现：Job 里面所在的 task 的监控、重启等等内容不见了。这就是 ApplicationMaster 存在的原因。ResourceManager 负责作业与资源的调度。接收 JobSubmitter 提交的作业，按照作业的上下文(Context) 信息以及从 NodeManager 收集来的状态信息，启动调度过程，分配一个 Container 作为 ApplicationMaster。
- **NodeManager**：功能比较专一，就是负责 Container 状态的维护，并向 ResourceManager 保持心跳。
- **ApplicationMaster**：负责一个 Job 生命周期内的所有工作，类似老的框架中 JobTracker。但是要注意，每一个 Job（不是每一种）都有一个 ApplicationMaster，它可以运行在 ResourceManager 以外的机器上。

（2）Yarn 框架相对于老的 MapReduce 框架的优势

Yarn 框架相对于老的 MapReduce 框架什么优势呢？具体表现在以下几个方面：

- 这个设计大大减小了 JobTracker（也就是现在的 ResourceManager）的资源消耗，并且让监测每一个 Job 子任务 (tasks) 状态的程序分布式化了，更安全、更优雅。
- 在新的 Yarn 中，ApplicationMaster 是一个可变更的部分，用户可以对不同的编程模型写自己的 ApplicationMaster，让更多类型的编程模型能够跑在 Hadoop 集群中，可以参考 Hadoop Yarn 官方配置模板中的 mapred-site.xml 配置。
- 对于资源的表示以内存为单位 (在目前版本的 Yarn 中，没有考虑 CPU 的占用)，比之前以剩余 slot 数目更合理。
- 老的框架中，JobTracker 一个很大的负担就是监控 job 下的 tasks 的运行状况，现在，这个部分就扔给 ApplicationMaster 做了，而 ResourceManager 中有一个模块叫做 ApplicationsMasters(注意不是 ApplicationMaster)，它是监测 ApplicationMaster 的运行状况，如果出问题，会将其在其他机器上重启。
- Container 是 Yarn 为了将来作资源隔离而提出的一个框架。这一点应该借鉴了 Mesos 的工作，目前是一个框架，仅仅提供 java 虚拟机内存的隔离，Hadoop 团队的设计思路应该后续能支持更多的资源调度和控制，既然资源表示成内存量，那就没有了之前的 map slot/reduce slot 分开造成集群资源闲置的尴尬情况。

本章小结

本章介绍了 MapReduce 计算模型的基本原理、工作流程，阐述了 Hadoop 中实现并行计算的相关机制以及 MapReduce 的任务调度过程；接下来，以一个实例演示了 MapReduce 的详细执行过程；最后，介绍了新 MapReduce 框架 Yarn 的原理及运作机制。

参考文献

[1] MapReduce. 百度百科.

[2] 分布式计算开源框架 Hadoop 入门实践. http://www.chinacloud.cn/show.aspx?id=463&cid=12

[3] Dean, Jeffrey & Ghemawat, Sanjay (2004). "MapReduce: Simplified Data Processing on Large Clusters". Retrieved Apr. 6, 2005

[4] 陆嘉恒. Hadoop 实战. 机械工业出版社. 2011 年.

[5] 其他网络来源.

附录 1:任课教师介绍

林子雨(1978－),男,博士,厦门大学计算机科学系助理教授,主要研究领域为数据库,数据仓库,数据挖掘.
主讲课程：《大数据技术基础》
办公地点：厦门大学海韵园科研 2 号楼
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu