11.大数据专业群教师 岗位试讲内容

注意事项：

1.每位考生试讲时间为10分钟；

2.试讲统一采用PPT讲授方式（自备U盘，如因U盘打不开课件，责任自负，U盘不能用考生姓名命名）；

3.试讲的考生在候考室抽签结束后在教案封面填写抽签号提交教案打印件（一式7份）给工作人员。教案不能透露任何个人信息，考生不得穿制服、单位工作服或有明显文字或图案标识的服装参加面试，凡透露个人信息的考生，扣减面试成绩的5%—20%，情节严重的，取消面试成绩。

教学内容：第7章 MapReduce

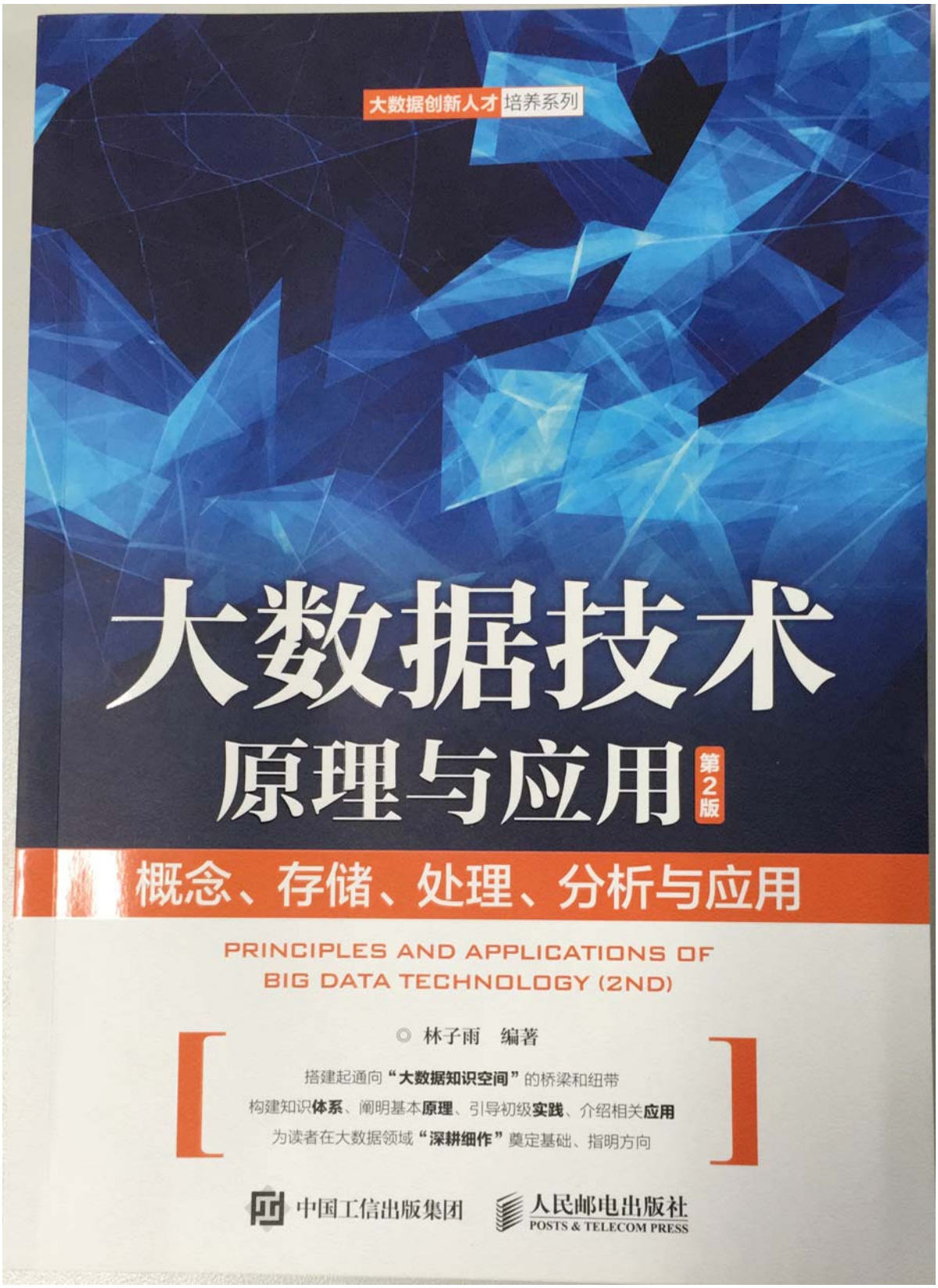
7.2 MapReduce的工作流程

7.3 实例分析：WordCount

教学重点： MapReduce算法执行过程，利用MapReduce算法实现词频统计案例实现，可自备教具及自备案例

教材信息：《大数据技术原理与应用》第2版，人民邮电出版社，2017年1月出版，林子雨编著。

教材封面


大数据创新人才培养系列
大数据技术
原理与应用
第2版
概念、存储、处理、分析与应用
PRINCIPLES AND APPLICATIONS OF
BIG DATA TECHNOLOGY (2ND)
林子雨 编著
搭建起通向“大数据知识空间”的桥梁和纽带
构建知识体系、阐明基本原理、引导初级实践、介绍相关应用
为读者在大数据领域“深耕细作”奠定基础、指明方向
中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

教学内容：第 7 章 MapReduce　7.2 MapReduce 的工作流程

7.3 实例分析：WordCount

输入元素，也可通过一个 Map 任务生成具有相同键的多个<key,value>。

Reduce 函数的任务就是将输入的一系列具有相同键的键值对以某种方式组合起来，输出处理后的键值对，输出结果会合并成一个文件。用户可以指定 Reduce 任务的个数（如 n 个），并通知实现系统，然后主控进程通常会选择一个 Hash 函数，Map 任务输出的每个键都会经过 Hash 函数计算，并根据哈希结果将该键值对输入相应的 Reduce 任务来处理。对于处理键为 k 的 Reduce 任务的输入形式为$<k,<v_1,v_2,\ldots,v_n>>$，输出为$<k,V>$。

下面给出一个简单实例。比如，我们想编写一个 MapReduce 程序来统计一个文本文件中每个单词出现的次数，对于表 7-1 中的 Map 函数的输入$<k_1,v_1>$而言，其具体数据就是<某一行文本在文件中的偏移位置，该行文本的内容>。用户可以自己编写 Map 函数处理过程，把文件中的一行读取后解析出每个单词，生成一批中间结果<单词,出现次数>，然后把这些中间结果作为 Reduce 函数的输入，Reduce 函数的具体处理过程也是由用户自己编写的，用户可以将相同单词的出现次数进行累加，得到每个单词出现的总次数。

7.2 MapReduce 的工作流程

理解 MapReduce 的工作流程，是开展 MapReduce 编程的前提。本节首先给出工作流程概述，并阐述 MapReduce 的各个执行阶段，最后对 MapReduce 的核心环节——Shuffle 过程进行详细剖析。

7.2.1 工作流程概述

大规模数据集的处理包括分布式存储和分布式计算两个核心环节。谷歌公司用分布式文件系统 GFS 实现分布式数据存储，用 MapReduce 实现分布式计算；而 Hadoop 则使用分布式文件系统 HDFS 实现分布式数据存储，用 Hadoop MapReduce 实现分布式计算。MapReduce 的输入和输出都需要借助于分布式文件系统进行存储，这些文件被分布存储到集群中的多个节点上。

MapReduce 的核心思想可以用“分而治之”来描述，如图 7-1 所示，也就是把一个大的数据集拆分成多个小数据块在多台机器上并行处理，也就是说，一个大的 MapReduce 作业，首先会被拆分成许多个 Map 任务在多台机器上并行执行，每个 Map 任务通常运行在数据存储的节点上，这样，计算和数据就可以放在一起运行，不需要额外的数据传输开销。当 Map 任务结束后，会生成以<key,value>形式表示的许多中间结果。然后，这些中间结果会被分发到多个 Reduce 任务在多台机器上并行执行，具有相同 key 的<key,value>会被发送到同一个 Reduce 任务那里，Reduce 任务会对中间结果进行汇总计算得到最后结果，并输出到分布式文件系统中。

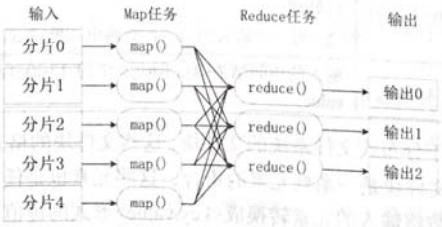


图 7-1　MapReduce 的工作流程

需要指出的是，不同的 Map 任务之间不会进行通信，不同的 Reduce 任务之间也不会发生任何信息交换；用户不能显式地从一台机器向另一台机器发送消息，所有的数据交换都是通过 MapReduce 框架自身去实现的。

在 MapReduce 的整个执行过程中，Map 任务的输入文件、Reduce 任务的处理结果都是保存在分布式文件系统中的，而 Map 任务处理得到的中间结果则保存在本地存储中（如磁盘）。另外，只有当 Map 处理全部结束后，Reduce 过程才能开始；只有 Map 需要考虑数据局部性，实现“计算向数据靠拢”，而 Reduce 则无需考虑数据局部性。

7.2.2　MapReduce 的各个执行阶段

下面是一个 MapReduce 算法的执行过程。

（1）MapReduce 框架使用 InputFormat 模块做 Map 前的预处理，比如验证输入的格式是否符合输入定义；然后，将输入文件切分为逻辑上的多个 InputSplit，InputSplit 是 MapReduce 对文件进行处理和运算的输入单位，只是一个逻辑概念，每个 InputSplit 并没有对文件进行实际切割，只是记录了要处理的数据的位置和长度。

（2）因为 InputSplit 是逻辑切分而非物理切分，所以还需要通过 RecordReader（RR）根据 InputSplit 中的信息来处理 InputSplit 中的具体记录，加载数据并转换为适合 Map 任务读取的键值对，输入给 Map 任务。

（3）Map 任务会根据用户自定义的映射规则，输出一系列的<key,value>作为中间结果。

（4）为了让 Reduce 可以并行处理 Map 的结果，需要对 Map 的输出进行一定的分区（Portition）、排序（Sort）、合并（Combine）、归并（Merge）等操作，得到<key,value-list>形式的中间结果，再交给对应的 Reduce 进行处理，这个过程称为 Shuffle。从无序的<key,value>到有序的<key,value-list>，这个过程用 Shuffle（洗牌）来称呼是非常形象的。

（5）Reduce 以一系列<key,value-list>中间结果作为输入，执行用户定义的逻辑，输出结果给 OutputFormat 模块。

（6）OutputFormat 模块会验证输出目录是否已经存在以及输出结果类型是否符合配置文件中的配置类型，如果都满足，就输出 Reduce 的结果到分布式文件系统。

MapReduce 工作流程中的各个执行阶段，具体如图 7-2 所示。

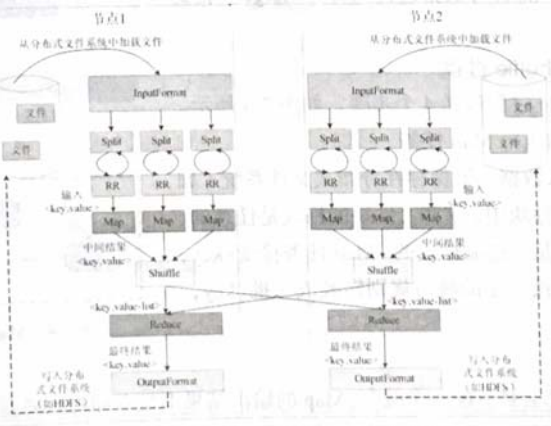


图 7-2　MapReduce 工作流程中的各个执行阶段

7.2.3 Shuffle 过程详解

Shuffle 过程是 MapReduce 整个工作流程的核心环节，理解 Shuffle 过程的基本原理，对于理解 MapReduce 流程至关重要。

1. Shuffle 过程简介

所谓 Shuffle，是指对 Map 输出结果进行分区、排序、合并等处理并交给 Reduce 的过程。因此，Shuffle 过程分为 Map 端的操作和 Reduce 端的操作，如图 7-3 所示，主要执行以下操作。

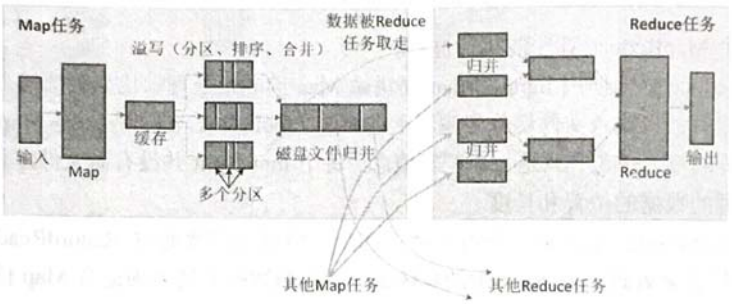


图 7-3 Shuffle 过程

（1）在 Map 端的 Shuffle 过程

Map 的输出结果首先被写入缓存，当缓存满时，就启动溢写操作，把缓存中的数据写入磁盘文件，并清空缓存。当启动溢写操作时，首先需要把缓存中的数据进行分区，然后对每个分区的数据进行排序（Sort）和合并（Combine），之后再写入磁盘文件。每次溢写操作会生成一个新的磁盘文件，随着 Map 任务的执行，磁盘中就会生成多个溢写文件。在 Map 任务全部结束之前，这些溢写文件会被归并（Merge）成一个大的磁盘文件，然后通知相应的 Reduce 任务来领取属于自己处理的数据。

（2）在 Reduce 端的 Shuffle 过程

Reduce 任务从 Map 端的不同 Map 机器领回属于自己处理的那部分数据，然后对数据进行归并（Merge）后交给 Reduce 处理。

2. Map 端的 Shuffle 过程

Map 端的 Shuffle 过程包括 4 个步骤，如图 7-4 所示。

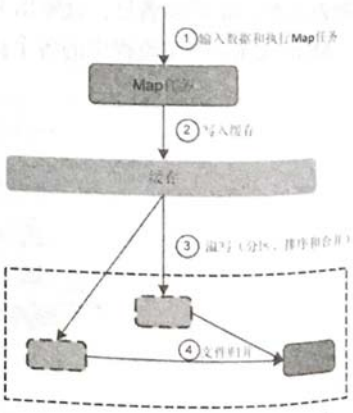


图 7-4 Map 端的 Shuffle 过程

（1）输入数据和执行 Map 任务

Map 任务的输入数据一般保存在分布式文件系统（如 GFS 或 HDFS）的文件块中，这些文件块的格式是任意的，可以是文档，也可以是二进制格式的。Map 任务接受<key, value>作为输入后，按一定的映射规则转换成一批<key, value>进行输出。

（2）写入缓存

每个 Map 任务都会被分配一个缓存，Map 的输出结果不是立即写入磁盘，而是首先写入缓存。在缓存中积累一定数量的 Map 输出结果以后，再一次性批量写入磁盘，这样可以大大减少对磁盘

I/O 的影响。因为，磁盘包含机械部件，它是通过磁头移动和盘片的转动来寻址定位数据的，每次寻址的开销很大，如果每个 Map 输出结果都直接写入磁盘，会引入很多次寻址开销，而一次性批量写入，就只需要一次寻址、连续写入，大大降低了开销。需要注意的是，在写入缓存之前，key 与 value 值都会被序列化成字节数组。

（3）溢写（分区、排序和合并）

提供给 MapReduce 的缓存的容量是有限的，默认大小是 100 MB。随着 Map 任务的执行，缓存中 Map 结果的数量会不断增加，很快就会占满整个缓存。这时，就必须启动溢写（Spill）操作，把缓存中的内容一次性写入磁盘，并清空缓存。溢写的过程通常是由另外一个单独的后台线程来完成的，不会影响 Map 结果往缓存写入，但是为了保证 Map 结果能够不停地持续写入缓存，不受溢写过程的影响，就必须让缓存中一直有可用的空间，不能等到全部占满才启动溢写过程，所以一般会设置一个溢写比例，如 0.8，也就是说，当 100 MB 大小的缓存被填满 80 MB 数据时，就启动溢写过程，把已经写入的 80 MB 数据写入磁盘，剩余 20 MB 空间供 Map 结果继续写入。

但是，在溢写到磁盘之前，缓存中的数据首先会被分区（Partition）。缓存中的数据是<key，value>形式的键值对，这些键值对最终需要交给不同的 Reduce 任务进行并行处理。MapReduce 通过 Partitioner 接口对这些键值对进行分区，默认采用的分区方式是采用 Hash 函数对 key 进行哈希后再用 Reduce 任务的数量进行取模，可以表示成 hash(key) mod R，其中 R 表示 Reduce 任务的数量，这样，就可以把 Map 输出结果均匀地分配给这 *R* 个 Reduce 任务去并行处理了。当然，MapReduce 也允许用户通过重载 Partitioner 接口来自定义分区方式。

对于每个分区内的所有键值对，后台线程会根据 key 对它们进行内存排序（Sort），排序是 MapReduce 的默认操作。排序结束后，还包含一个可选的合并（Combine）操作。如果用户事先没有定义 Combiner 函数，就不用进行合并操作。如果用户事先定义了 Combiner 函数，则这个时候会执行合并操作，从而减少需要溢写到磁盘的数据量。

所谓“合并”，是指将那些具有相同 key 的<key,value>的 value 加起来。比如，有两个键值对<“xmu” 1>和<“xmu” 1>，经过合并操作以后就可以得到一个键值对<“xmu” 2>，减少了键值对的数量。这里需要注意，Map 端的这种合并操作，其实和 Reduce 的功能相似，但是由于这个操作发生在 Map 端，所以我们只能称之为“合并”，从而有别于 Reduce。不过，并非所有场合都可以使用 Combiner，因为 Combiner 的输出是 Reduce 任务的输入，Combiner 绝不能改变 Reduce 任务最终的计算结果，一般而言，累加、最大值等场景可以使用合并操作。

经过分区、排序以及可能发生的合并操作之后，这些缓存中的键值对就可以被写入磁盘，并清空缓存。每次溢写操作都会在磁盘中生成一个新的溢写文件，写入溢写文件中的所有键值对都是经过分区和排序的。

（4）文件归并

每次溢写操作都会在磁盘中生成一个新的溢写文件，随着 MapReduce 任务的进行，磁盘中的溢写文件数量会越来越多。当然，如果 Map 输出结果很少，磁盘上只会存在一个溢写文件，但是通常都会存在多个溢写文件。最终，在 Map 任务全部结束之前，系统会对所有溢写文件中的数据进行归并（Merge），生成一个大的溢写文件，这个大的溢写文件中的所有键值对也是经过分区和排序的。

所谓“归并”，是指对于具有相同 key 的键值对会被归并成一个新的键值对。具体而言，对于若干个具有相同 key 的键值对$<k_1,v_1>$，$<k_1,v_2>$ …… $<k_1,v_n>$会被归并成一个新的键值对

$<k_1,<v_1,v_2,\cdots,v_n>>$。

另外，进行文件归并时，如果磁盘中已经生成的溢写文件的数量超过参数 min.num.spills.for.combine 的值时（默认值是 3，用户可以修改这个值），那么，就可以再次运行 Combiner，对数据进行合并操作，从而减少写入磁盘的数据量。但是，如果磁盘中只有一两个溢写文件时，执行合并操作就会“得不偿失”，因为执行合并操作本身也需要代价，因此不会运行 Combiner。

经过上述 4 个步骤以后，Map 端的 Shuffle 过程全部完成，最终生成的一个大文件会被存放在本地磁盘上。这个大文件中的数据是被分区的，不同的分区会被发送到不同的 Reduce 任务进行并行处理。JobTracker 会一直监测 Map 任务的执行，当监测到一个 Map 任务完成后，就会立即通知相关的 Reduce 任务来“领取”数据，然后开始 Reduce 端的 Shuffle 过程。

3. Reduce 端的 Shuffle 过程

相对于 Map 端而言，Reduce 端的 Shuffle 过程非常简单，只需要从 Map 端读取 Map 结果，然后执行归并操作，最后输送给 Reduce 任务进行处理。具体而言，Reduce 端的 Shuffle 过程包括 3 个步骤，如图 7-5 所示。

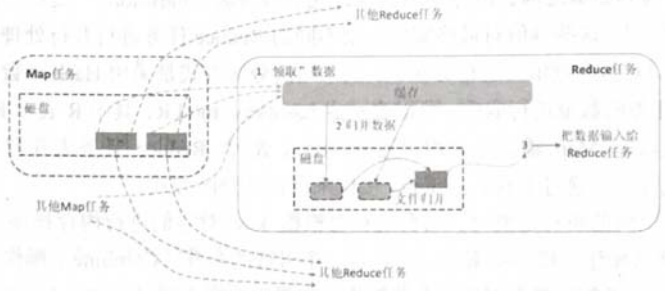


图 7-5 Reduce 端的 Shuffle 过程

（1）“领取”数据

Map 端的 Shuffle 过程结束后，所有 Map 输出结果都保存在 Map 机器的本地磁盘上，Reduce 任务需要把这些数据“领取”（Fetch）回来存放到自己所在机器的本地磁盘上。因此，在每个 Reduce 任务真正开始之前，它大部分时间都在从 Map 端把属于自己处理的那些分区的数据“领取”过来。每个 Reduce 任务会不断地通过 RPC 向 JobTracker 询问 Map 任务是否已经完成；JobTracker 监测到一个 Map 任务完成后，就会通知相关的 Reduce 任务来“领取”数据；一旦一个 Reduce 任务收到 JobTracker 的通知，它就会到该 Map 任务所在机器上把属于自己处理的分区数据领取到本地磁盘中。一般系统中会存在多个 Map 机器，因此 Reduce 任务会使用多个线程同时从多个 Map 机器领回数据。

（2）归并数据

从 Map 端领回的数据会首先被存放在 Reduce 任务所在机器的缓存中，如果缓存被占满，就会像 Map 端一样被溢写到磁盘中。由于在 Shuffle 阶段 Reduce 任务还没有真正开始执行，因此，这时可以把内存的大部分空间分配给 Shuffle 过程作为缓存。需要注意的是，系统中一般存在多个 Map 机器，Reduce 任务会从多个 Map 机器领回属于自己处理的那些分区的数据，因此缓存中的数据是来自不同的 Map 机器的，一般会存在很多可以合并（Combine）的键值对。当溢写过程启动时，具有相同 key 的键值对会被归并（Merge），如果用户定义了 Combiner，则归并后的数据还可以执行合并操作，减少写入磁盘的数据量。每个溢写过程结束后，都会在磁盘中生成一个溢写

文件，因此磁盘上会存在多个溢写文件。最终，当所有的 Map 端数据都已经被领回时，和 Map 端类似，多个溢写文件会被归并成一个大文件，归并的时候还会对键值对进行排序，从而使得最终大文件中的键值对都是有序的。当然，在数据很少的情形下，缓存可以存储所有数据，就不需要把数据溢写到磁盘，而是直接在内存中执行归并操作，然后直接输出给 Reduce 任务。需要说明的是，把磁盘上的多个溢写文件归并成一个大文件可能需要执行多轮归并操作。每轮归并操作可以归并的文件数量是由参数 io.sort.factor 的值来控制的（默认值是 10，可以修改）。假设磁盘中生成了 50 个溢写文件，每轮可以归并 10 个溢写文件，则需要经过 5 轮归并，得到 5 个归并后的大文件。

（3）把数据输入给 Reduce 任务

磁盘中经过多轮归并后得到的若干个大文件，不会继续归并成一个新的大文件，而是直接输入给 Reduce 任务，这样可以减少磁盘读写开销。由此，整个 Shuffle 过程顺利结束。接下来，Reduce 任务会执行 Reduce 函数中定义的各种映射，输出最终结果，并保存到分布式文件系统中（比如 GFS 或 HDFS）。

7.3 实例分析：WordCount

下面给出一个 WordCount 实例来阐述采用 MapReduce 解决实际问题的基本思路和具体实现过程。

7.3.1 WordCount 的程序任务

在编程语言的学习过程中，都会以“HelloWorld”程序作为入门范例，WordCount 就是类似“HelloWorld”的 MapReduce 入门程序，见表 7-2。表 7-3 给出了一个 WordCount 的输入和输出实例。

表 7-2 WordCount 程序任务

项目	描述
程序	WordCount
输入	一个包含大量单词的文本文件
输出	文件中每个单词及其出现次数（频数），并按照单词字母顺序排序，每个单词和其频数占一行，单词和频数之间有间隔

表 7-3 一个 WordCount 的输入和输出实例

输入	输出
Hello World Hello Hadoop Hello MapReduce	Hadoop 1 Hello 3 MapReduce 1 World 1

7.3.2 WordCount 的设计思路

首先，需要检查 WordCount 程序任务是否可以采用 MapReduce 来实现。在前文我们曾经提到，