第5章

分布式计算模型MapReduce

MapReduce是Hadoop系统的核心组件之一，它是一种可用于大数据并行处理的计算模型、框架和平台，主要解决海量数据的分析计算，是目前分布式计算模型中应用较为广泛的一种。MapReduce计算模型由Google提出，后来由Apache Hadoop项目进行了开源实现。除了Hadoop，MapReduce计算模型也被其他分布式计算系统广泛采用，如Apache Spark、Apache Flink等。这些系统在MapReduce计算模型的基础上，进一步优化和扩展了计算能力，提供更多的计算模式和灵活性，适用于更多种类的数据处理和分析任务。

本章将介绍MapReduce的特点，分析MapReduce的基本思想、算法原理、工作流程、基本架构，通过入门案例帮助读者理解其工作原理，通过排序的进阶案例带读者实践MapReduce的编程过程。

5.1 问题导入

在大数据处理领域中，如何高效地处理和分析海量数据始终是一个巨大的挑战。传统的数据处理方法往往无法应对这种规模的需求，因此需要一种能够在分布式环境中高效运行的数据处理模型。MapReduce应运而生，它通过将复杂的计算任务分解成小块并行处理，从而显著提高了大数据处理的效率。

然而，随着数据规模的不断增长和多样化需求的增加，MapReduce本身也面临着一系列挑战和问题：

- 如何有效地分配和管理计算资源？
- 如何处理数据的倾斜问题，避免计算负载的不均衡？
- 如何优化Map和Reduce阶段的性能，减少数据传输的开销？

这些问题是MapReduce在实际应用中需要重点解决的关键点。在本章中，将针对这些问题进行详细探讨，并介绍MapReduce模型的基本概念、工作原理以及实际应用案例，帮助读者全面理解这一强大的分布式计算模型。

首先从MapReduce的概述开始，介绍其发展历史以及在大数据处理中的重要地位。接下来，将详细解释MapReduce的优缺点，分析其在不同应用场景中的表现。之后，深入研究MapReduce的工作原理，包括其计算模型的基本思想和数据处理流程，帮助读者掌握其内部机制和操作方法。最后，通过两个实际应用案例——文本分词统计计算和海量数据排序处理，展示MapReduce在解决实际问题中的强大能力。

5.2　MapReduce 概述

MapReduce 是被广泛采用的计算模型，该模型一经面世便受到广泛关注并迅速普及。要想了解这其中的原因，还要从其发展和优缺点说起。

5.2.1　MapReduce 的发展

谷歌在 2003—2006 年连续发表了 3 篇很有影响力的文章，分别阐述了 GFS、MapReduce 和 Bigtable 的核心思想。其中，MapReduce 是谷歌公司的核心计算模型，运行在分布式文件系统 GFS 上，应用于谷歌的大规模数据处理和分布式计算任务。

Hadoop 项目最早是由 Doug Cutting 在 2006 年创建的，受 MapReduce 模型启发，Doug Cutting 将其作为 Hadoop 的计算框架，并以 Apache Nutch 作为初始基础进行开发。与谷歌类似，Hadoop MapReduce 运行在分布式文件系统 HDFS 上。Hadoop MapReduce 要比谷歌 MapReduce 的使用门槛低很多，程序员即使没有任何分布式程序开发经验，也可以很轻松地开发出分布式程序并部署到计算机集群中。随着 Hadoop 的普及和使用，Hadoop MapReduce 逐渐成为处理大数据的事实标准。

Hadoop 2.0 引入了 YARN，取代了旧版 Hadoop 的 MapReduce 作业调度器。YARN 将 Hadoop 的资源管理和作业调度功能进行了解耦，提供了更灵活和可扩展的资源管理机制，使得 Hadoop MapReduce 能够更好地支持多种计算模型和应用。

随着大数据计算需求的不断增长，Hadoop 生态系统中涌现了更多的计算模型和框架，如 Apache Spark、Apache Flink 等。这些新的计算模型在 Hadoop MapReduce 的基础上进行了优化和扩展，提供更高性能和更多的计算能力，逐渐成为 Hadoop 生态系统的重要组成部分。

Hadoop 3.x 版本继续对 Hadoop MapReduce 进行改进和优化。Hadoop 3.x 引入了更多的功能和改进，包括更快的数据复制和数据恢复、更好的资源管理和容错性，进一步提升了 Hadoop MapReduce 的性能和可靠性。

总体而言，Hadoop MapReduce 经历了从初始版本到成熟阶段，再到与 YARN 和新计算模型的整合，不断发展和演进，成为大数据处理的重要组件之一。

同时，随着技术的发展，Hadoop 生态系统还出现了更多的计算模型和框架，为大数据处理提供了更多的选择和灵活性。

5.2.2　MapReduce 的优缺点

MapReduce 作为一种分布式计算模型和编程框架，在大数据处理中具有很多优点，但也有一些缺点。

1. 优点

(1) MapReduce 易于编程。

相比于其他复杂的分布式计算框架，MapReduce 提供了简单、清晰的编程接口，容易学习和使用，降低了分布式计算的门槛。用户简单地实现一些接口，就可以完成一个分布式程序，这个分布式程序可以分布到大量廉价的服务器上运行。这个特点使得 MapReduce 编程变得非常流行。

(2) 具有良好的扩展性。

当用户的计算资源不能得到满足时,可以通过简单地增加机器来扩展 MapReduce 的计算能力。随着计算节点的增加,系统的处理能力可以线性扩展,适应不断增长的数据量和计算需求。

(3) 具有高容错性。

MapReduce 设计的初衷就是使程序能够部署在廉价的服务器上,这就要求它具有很高的容错性。例如,其中一台服务器宕机了,MapReduce 可以把上面的计算任务转移到另外一个节点上运行,不至于这个任务运行失败,而且这个过程不需要人工参与,而完全是由 Hadoop 内部完成的。

(4) 适合 PB 级以上海量数据的离线处理。

MapReduce 可以实现上千台服务器集群并发工作,提供海量数据处理能力。MapReduce 适用于离线批处理场景,特别擅长处理大规模的数据集,如日志分析、数据清洗、数据转换等任务。

2. 缺点

(1) 不擅长实时计算。

MapReduce 适用于批处理场景,不适合处理实时数据和流式数据,因为 MapReduce 需要在每次作业完成后再次启动一个新的作业。流式计算的输入数据是动态的,而 MapReduce 的输入数据集是静态的,不能动态变化。这是因为 MapReduce 自身的设计特点决定了数据源必须是静态的。

(2) 低效的迭代计算。

MapReduce 在处理迭代计算时效率较低,由于每个 MapReduce 作业之间需要进行磁盘读写和数据传输,导致迭代计算的性能较差。

(3) 不擅长 DAG(有向无环图)计算。

多个应用程序存在依赖关系,后一个应用程序的输入为前一个的输出。在这种情况下,MapReduce 并不是不能做,而是使用后 MapReduce 会生成大量的中间数据,需要进行磁盘读写和数据传输,可能导致 I/O 开销较大,导致性能非常低下。

5.3 MapReduce 工作原理

本节详细介绍 MapReduce 的工作原理。通过分解数据处理任务为 Map 阶段和 Reduce 阶段,并在分布式计算节点上进行并行执行,MapReduce 实现了高效、可靠的大数据处理。本节将深入探讨 MapReduce 的基本思想、内部流程、数据划分、Map 阶段、Shuffle 阶段、Reduce 阶段等关键内容,帮助读者深入理解 MapReduce 的工作机制和优势。

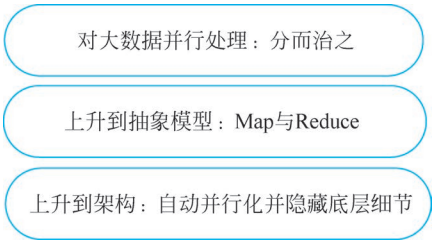

图 5-1 MapReduce 基本思想

5.3.1 MapReduce 计算模型介绍

1. MapReduce 基本思想

使用 MapReduce 处理大数据的基本思想包括 3 个层面,如图 5-1 所示。

首先,对大数据采取分而治之的思想。对相互间不具有计算依赖关系的大数据实现并行处理,最自然的办法就是采取分而治之的策略。

其次，把分而治之的思想上升到抽象模型。为了克服 MPI 等并行计算方法缺少高层并行编程模型这一缺陷，MapReduce 借鉴了 Lisp 函数式语言中的思想，用 Map 和 Reduce 两个函数提供了高层的并行编程抽象模型。

最后，把分而治之的思想上升到架构层面，统一架构为程序员隐藏系统层的实现细节。

MPI 等并行计算方法缺少统一的计算框架支持，程序员需要考虑数据存储、划分、分发、结果收集、错误恢复等诸多细节，为此，MapReduce 设计并提供了统一的计算框架，为程序员隐藏了绝大多数系统层面的处理细节。

(1) 分而治之。

分而治之是 MapReduce 的核心思想，它表示把一个大规模的数据集切分成很多小的单独的数据集，然后放在多个机器上同时处理。其计算思想如图 5-2 所示。

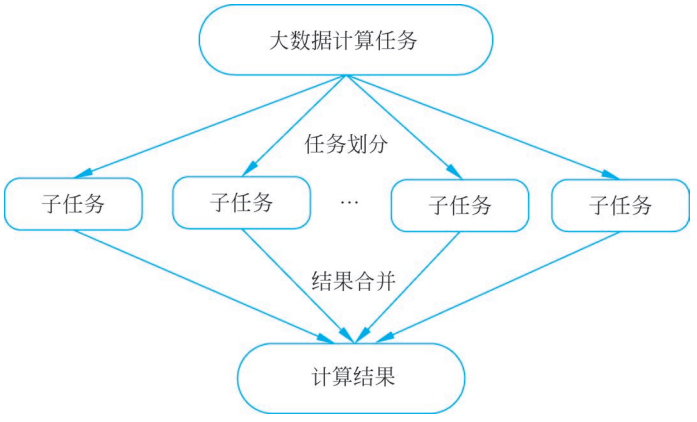

图 5-2　分而治之计算思想

并行计算的第一个重要问题是如何划分计算任务或者计算数据以便对划分的子任务或数据块同时进行计算。但是，一些计算问题的前后数据项之间存在很强的依赖关系，无法进行划分，只能串行计算。

对于不可拆分的计算任务或相互间有依赖关系的数据无法进行并行计算。一个大数据若可以分为具有同样计算过程的数据块，并且这些数据块之间不存在数据依赖关系，则提高处理速度的最好办法就是并行计算。

(2) 高度抽象为两个阶段：Map(映射)与 Reduce(归约)。

MapReduce 任务过程是分为两个处理阶段。

① Map 阶段。Map 阶段的主要作用是“分”，即把复杂的任务分解为若干“简单的任务”来并行处理。Map 阶段的这些任务可以并行计算，彼此间没有依赖关系。

② Reduce 阶段。Reduce 阶段的主要作用是“合”，即对 Map 阶段的结果进行全局汇总。

事实上，在 Map 阶段和 Reduce 阶段之间，还需要数据的传输和排序，通常把这个阶段叫作 Shuffle 阶段。Shuffle 阶段是 MapReduce 的中间阶段，是 Map 阶段和 Reduce 阶段之间的连接。在 Shuffle 阶段，MapReduce 会根据 Map 输出的键，将相同键的值进行分组，然后将这些分组的数据传递给对应的 Reduce 任务。因此也经常会说有 3 个阶段，如图 5-3 所示。

(3) 自动实现并行化。

MapReduce 提供了一个统一的计算框架来完成计算任务的划分和调度，数据的分布存储和划分，处理数据与计算任务的同步，结果数据的收集整理，系统通信、负载平衡、计算性能优

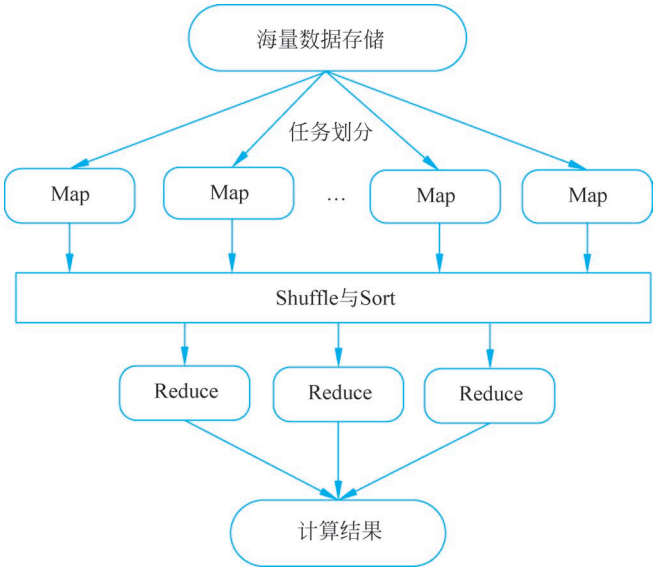

图 5-3 Map 与 Reduce

化、系统节点出错检测和失效恢复处理等。

MapReduce 通过抽象模型和计算框架把需要做什么与具体怎么做分开了，为程序员提供了一个抽象和高层的编程接口和框架，程序员仅需要关心其应用层的具体计算问题，仅需要编写少量的处理应用本身计算问题的程序代码。

与具体完成并行计算任务相关的诸多系统层细节被隐藏起来，交给计算框架去处理。从分布代码的执行，大到数千个小到单个的节点集群的自动调度使用，程序员都不用操心。

MapReduce 计算架构提供的主要功能包括以下几点。

(1) 任务调度。

提交的一个计算作业(Job)将被划分为很多个计算任务。

任务调度功能主要负责为这些划分后的计算任务分配和调度计算节点(Map 节点或 Reduce 节点)，同时负责监控这些节点的执行状态，以及 Map 节点执行的同步控制，也负责进行一些计算性能优化处理。例如，对最慢的计算任务采用多备份执行，选最快完成者作为结果。

(2) 数据/程序互定位。

为了减少数据通信量，一个基本原则是本地化数据处理，即一个计算节点尽可能处理其本地磁盘上分布存储的数据，这实现了代码向数据的迁移。

当无法进行这种本地化数据处理时，再寻找其他可用节点并将数据从网络上传送给该节点(数据向代码迁移)，但将尽可能从数据所在的本地机架上寻找可用节点以减少通信延迟。

(3) 出错处理。

在以低端商用服务器构成的大规模 MapReduce 计算集群中，节点硬件(主机、磁盘、内存等)出错和软件有缺陷是常态。因此，MapReduce 架构需要能检测并隔离出错节点，并调度分配新的节点接管出错节点的计算任务。

(4) 分布式数据存储与文件管理。

海量数据处理需要一个良好的分布数据存储和文件管理系统作为支撑，系统要能够把海量数据分布存储在各节点的本地磁盘上，还要保持整个数据在逻辑上成为一个完整的数据文件。为了提供数据存储容错机制，系统还要提供数据块的多备份存储管理能力。

（5）合并和划分。

为了减少数据通信开销，中间结果数据进入 Reduce 节点前需要进行合并（combine）处理，即把具有同样主键的数据合并到一起避免重复传送。

一个 Reduce 节点所处理的数据可能会来自多个 Map 节点，因此，Map 节点输出的中间结果需使用一定的策略进行适当的划分（partition）处理，保证相关数据发送到同一个 Reduce 节点上。

2. Map 和 Reduce 函数

使用 MapReduce 执行计算任务时，每个任务的执行过程都会被分为两个阶段，分别是 Map 和 Reduce，其中 Map 阶段用于对原始数据进行处理，Reduce 阶段用于对 Map 阶段的结果进行汇总，得到最终结果。这两个阶段的模型如图 5-4 所示。

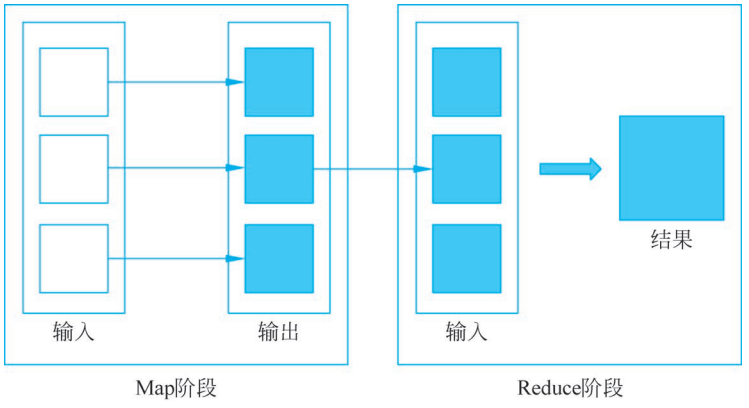

图 5-4　MapReduce 简易模型

MapReduce 编程模型借鉴了函数式程序设计语言的设计思想，其程序实现过程是通过 map 和 reduce 函数来完成的。从数据格式上来看，map 函数接收的数据格式是键值对，产生的输出结果也是键值对形式，reduce 函数会将 map 函数输出的键值对作为输入，把相同 key 值的 value 进行汇总，输出新的键值对。接下来，通过一张图来描述 MapReduce 的简易数据流模型，具体如图 5-5 所示。

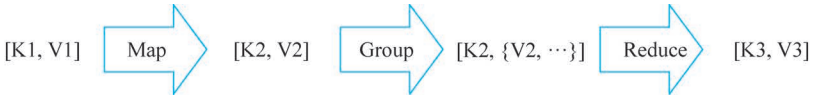

图 5-5　MapReduce 简易数据流模型

对于图 5-5 描述的 MapReduce 简易数据流模型说明如下。

（1）将原始数据处理成键值对< K1，V1 >形式。

（2）将解析后的键值对< K1，V1 >传给 map 函数，map 函数会根据映射规则，将键值对< K1，V1 >映射为一系列中间结果形式的键值对< K2，V2 >。

（3）将中间形式的键值对< K2，V2 >形成< K2，{V2，…}>形式传给 reduce 函数处理，把具有相同 key 的 value 合并在一起，产生新的键值对< K3，V3 >，此时的键值对< K3，V3 >就是最终输出的结果。

5.3.2　MapReduce 数据处理流程

那么，MapReduce 到底是如何运行的呢？按照时间顺序，MapReduce 任务执行包括输入分片 Map、Shuffle 和 Reduce 等阶段，一个阶段的输出正好是下一阶段的输入，上述各阶段的

关系和流程如图 5-6 所示。

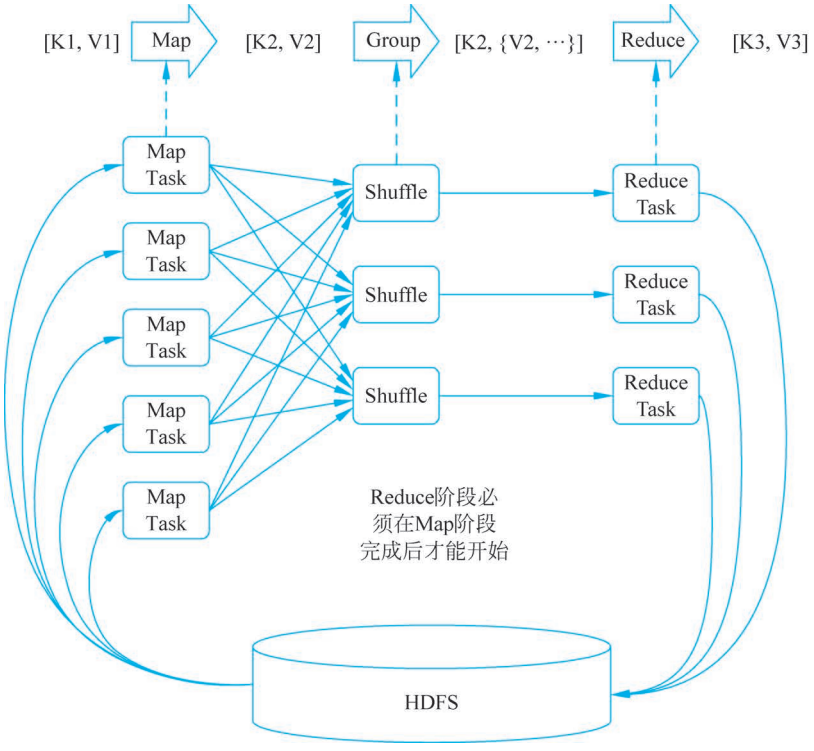

图 5-6　MapReduce 执行阶段和流程

图 5-6 从整体角度很好地表示了 MapReduce 的大致阶段划分和工作流程。下面结合上文的示例文件更加深入和详细地介绍上述过程，结合单词计数实例的 MapReduce 执行阶段和流程如图 5-7 所示。

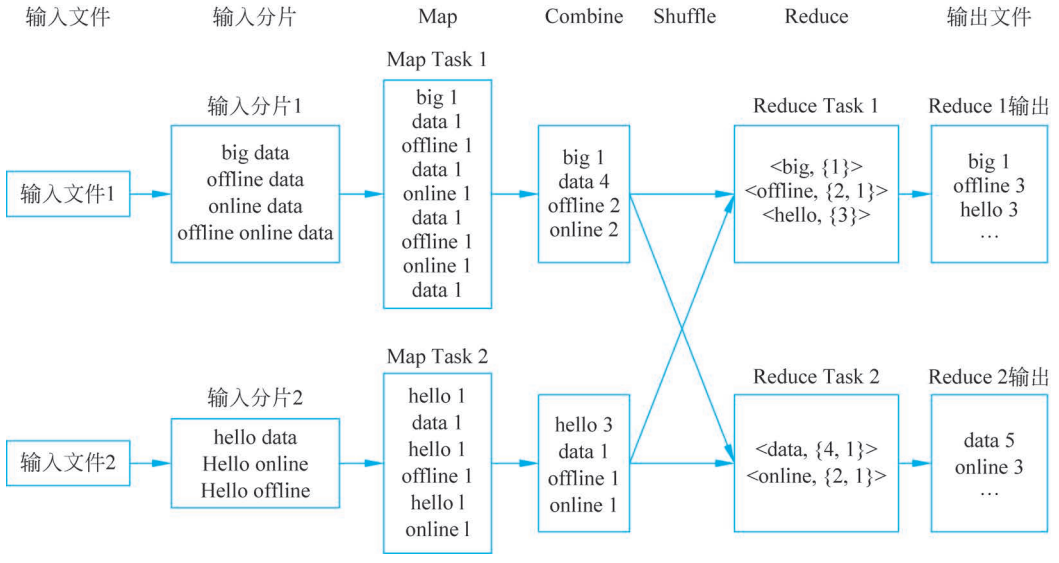

图 5-7　结合单词计数实例的 MapReduce 执行阶段和流程

1. 输入分片

在进行 Map 计算之前，MapReduce 会根据输入文件计算输入分片(input split)。每个输入分片对应一个 Map 任务(Map Task)，输入分片存储的并非数据本身，而是一个分片长度和

一个记录数据的位置的数组。输入分片往往和 HDFS 的块关系很密切，假如设定 HDFS 的块大小是 64MB，如果输入只有一个文件，大小为 150MB，那么 MapReduce 会把此大文件切分为 3 片（分别为 64MB、64MB 和 22MB），同样，如果输入为两个文件，其大小分别是 20MB 和 100MB，那么 MapReduce 会把 20MB 文件作为一个输入分片，100MB 则切分为两个即 64MB 和 36MB 的输入分片。对于上述实例文件 1 和 2，由于非常小，因此分别被作为输入分片 1 和输入分片 2 输入 Map 任务 1 和 Map 任务 2 中（此处只为说明问题，实际处理中应将小文件进行合并，否则如果输入多个文件而且文件大小均远小于块大小，会导致生成多个不必要的 Map 任务，这也是 MapReduce 优化计算的一个关键点）。

2. Map 阶段

在 Map 阶段，各 Map 任务会接收到所分配到的输入分片，并调用 Map 函数，逐行执行并输出键值对。例如对于本例，Map Task 1 将会接收到输入分片 1，并调用 Map 函数，其输出将为如下的键值对。

```
big 1
data 1
offline 1
data 1
online 1
data 1
offline 1
online 1
data 1
```

3. Combine 阶段

Combine 阶段是可选的。Combine 其实也是一种 Reduce 操作，但它是一个本地化的 Reduce 操作，是 Map 运算的本地后续操作，主要是在 Map 计算出中间文件前做一个简单的合并重复键值的操作，例如上述文件 1 中 data 出现了 4 次，Map 计算时如果碰到一个 data 的单词就会记录为 1，这样就重复计算了 4 次，Map 任务输出就会有冗余，这样后续处理和网络传输都被消费不必要的资源，因此通过 Combine 操作可以解决和优化此问题，但这一操作是有风险的，使用它的原则是 Combine 的输出不会影响到 Reduce 计算的最终输入。例如，如果计算只是求总数、最大值及最小值，可以使用 Combine 操作，但是如果做平均值计算使用 Combine，最终的 Reduce 计算结果就会出错。

4. Shuffle 阶段

Map 任务的输出必须经过一个 Shuffle 的阶段才能交给 Reduce 处理。Shuffle 阶段是 MapReduce 的核心，它的性能直接影响整个 MapReduce 的性能，下面将详细介绍其过程和原理。

什么是 Shuffle 呢？一般理解为数据从 Map 任务输出到 Reduce 任务输入的过程，它决定了 Map 任务的输出如何高效地传送给 Reduce 任务，如图 5-8 所示。

总体来说，Shuffle 阶段包含在 Map 和 Reduce 两个阶段中。

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

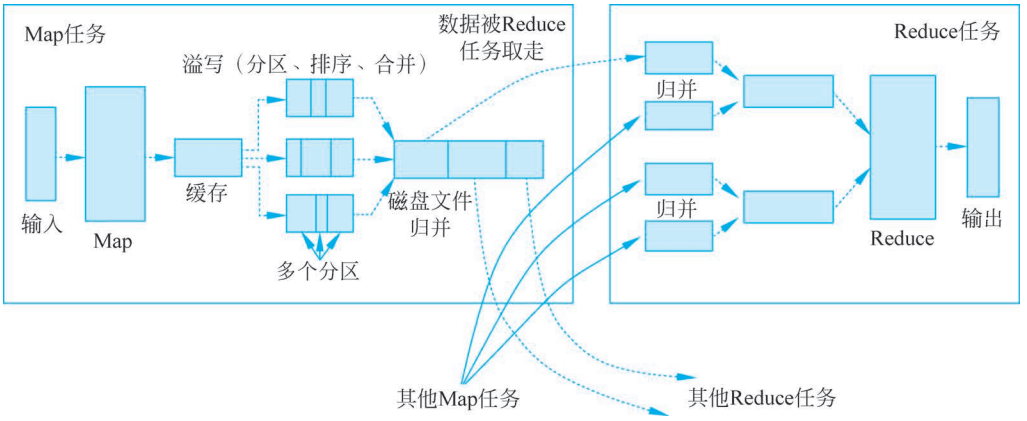

图 5-8　Shuffle 过程

领取属于自己处理的数据。

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后交给 Reduce 处理。

下面从 Map 和 Reduce 两端详细介绍 Shuffle 阶段。

（1）Map 阶段的 Shuffle。

通常 MapReduce 计算的都是海量数据，而且 Map 输出还需要对结果进行排序，内存开销很大，因此完全在内存中完成是不可能也是不现实的，所以 Map 输出时会在内存中开启一个环形内存缓冲区，并且在配置文件中为这个缓冲区设定了一个阈值（默认为 80%，可自定义修改此配置）。同时，Map 还为输出操作启动一个守护线程，如果缓存区的内存使用达到了阈值，那么这个守护线程就会把这 80%的内存区内容写到磁盘上，这个过程就叫分隔。另外的 20%内存可以供 Map 输出继续使用，写入磁盘和写入内存操作是互不干扰的，如果缓存区被撑满了，那么 Map 就会阻塞写入内存的操作，待写入磁盘操作完成后再继续执行写入内存操作，如图 5-9 所示。

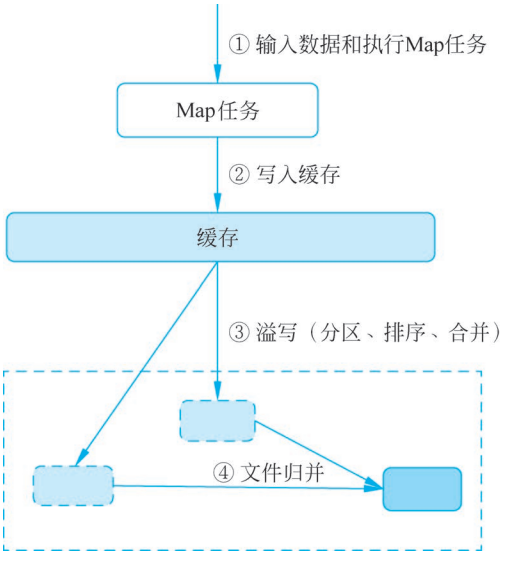

图 5-9　Map 阶段的 Shuffle

缓冲区内容分隔到磁盘前，会首先进行分区操作，分区的数目由 Reduce 的数目决定。对于本例，Reduce 数目为 2 个，那么分区数就是 2，然后对于每个分区，后台线程还会按照键值对需要写出的数据进行排序，如果配置了 combine 函数，还会进行 combine 操作，以使得更少的数据被写入磁盘并发送给 Reduce。

每次的分隔操作都会生成一个分隔文件，全部的 Map 输出完成后，可能会有很多的分隔文件，因此在 Map 任务结束前，还要进行归并操作，这些溢写文件会归并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。

至此，Map 阶段的所有工作都已结束，最终生成的文件也会存放在 NodeManager（YARN 的组件，在 Hadoop 3. x 中代替了 TaskTracker 执行任务）能访问的某个本地目录内。每个 Reduce 任务不断地从 ApplicationMaster（在 Hadoop 3. x 中取代了 JobTracker 作为应用程序执行管理器）那里获取 Map 任务是否完成的信息，如果 Reduce 任务得到通知，获知某台 NodeManager 上的 Map 任务执行完成，Shuffle 的后半段过程，也就是 Reduce 阶段的 Shuffle，便开始启动。

（2）Reduce 阶段的 Shuffle。

Shuffle 在 Reduce 阶段可以分为 3 个阶段：复制 Map 输出、归并阶段和 Reduce 任务处理，如图 5-10 所示。

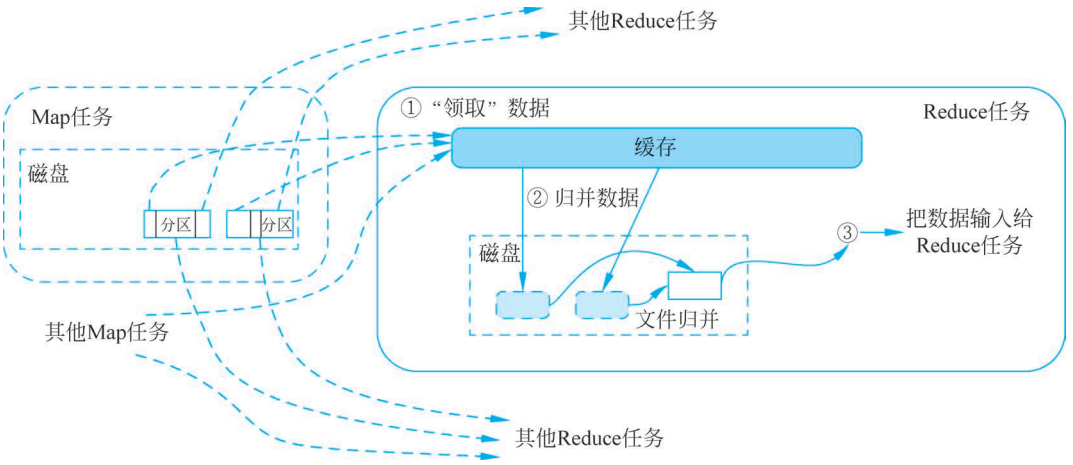

图 5-10　Reduce 端的 Shuffle

① 复制 Map 输出。如上文所述，Map 任务完成后，会通知 ApplicationMaster，然后 ApplicationMaster 会进一步通知 ResourceManager（YARN 的组件，在 Hadoop 3. x 中代替了 JobTracker 作为资源管理器）。这些通知会通过心跳机制完成。对于 Job 来说，ResourceManager 记录了 Map 输出和 NodeManager 的映射关系。同时 Reduce 也会定期向 Application Master 获取 Map 是否输出以及输出位置的信息，一旦拿到输出位置，Reduce 任务就会启动复制线程，通过 HTTP 方式请求 Map 任务所在的 NodeManager 获取其输出文件。因为 Map 任务早已结束，这些文件就被 NodeManager 存储在 Map 任务所在的本地磁盘中。

② 归并阶段。此处的归并和 Map 阶段的归并类似。复制过来的数据会首先放入内存缓冲区中，这里的内存缓冲区大小比 Map 阶段要灵活很多，它基于 JVM 的 heap size 设置，由于 Shuffle 阶段 Reduce 任务并不运行，绝大部分内存应该给 Shuffle 使用；同时此 Shuffle 的归并阶段根据要处理的数据量不同，也可能会有分隔到磁盘的过程，如果设置了 Combine 函数，

Combine 操作也会进行。

从 Map 阶段的 Shuffle 过程到 Reduce 阶段的 Shuffle 过程，都提到了归并，那么归并究竟是怎样的呢？如本文的例子，Map Task 1 对于 offline 键的值为 2，而 Map Task 2 的 offline 键值为 1，那么归并就是将 offline 的键值归并为 group，本例即为<offline,{2,1}>。

③ Reduce 任务处理。不断归并后，最后会生成一个最终结果（可能在内存，也可能在磁盘）。至此 Reduce 任务的输入准备完毕，下一步就是真正的 Reduce 操作。

5. Reduce 阶段

经过 Map 和 Reduce 阶段的 Shuffle 过程后，Reduce 任务的输入终于准备完毕，相关的数据已经被归并和汇总，Reduce 任务只需调用 Reduce 函数即可。对于本例来说，就是对每个键调用 sum 逻辑合并 value 并输出到 HDFS 即可。例如，对于 Reduce 任务的 offline 的键，只需将集合{2,1}相加，输出 offline 3 即可。

至此，整个 MapReduce 的详细流程和原理介绍完毕，从上述整个过程可以看出，Shuffle 是整个流程最为核心的部分，也是最为复杂的部分，当然也是 MapReduce 魔力发生的地方。理解 MapReduce 的关键就是理解 Shuffle 过程。

5.3.3 MapReduce 基本架构

MapReduce 1.x 的版本采用 Master/Slave 的主从架构，MapReduce 包含 4 部分，分别为 Client、JobTracker、TaskTracker 和 Task。在 Hadoop 2.x 版本后，去掉了 JobTracker 和 TaskTracker，用 ResourceManager 和 NodeManager 替代。本书中将以 3.x 版本为例，讲解 MapReduce 的架构，如图 5-11 所示。

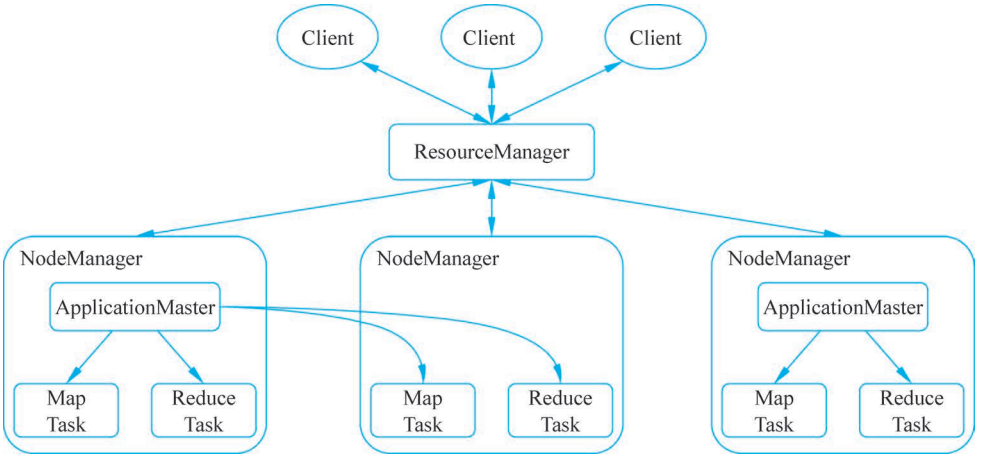

图 5-11 MapReduce 的架构

1. Client

在 Hadoop 3.x 中，每个 Job 仍然由 Client 提交，但是现在 Client 将应用程序以及配置参数打包为 jar 文件并存储在 HDFS 中。然后，Client 通过 ResourceManager（取代了 JobTracker）来提交作业。ResourceManager 是一个全局的资源管理器，负责整个集群的资源监控和作业调度。

2. ResourceManager

ResourceManager 取代了 Hadoop 1.x 版本中的 JobTracker，负责资源监控和作业调度。它监控所有 NodeManager（取代了 TaskTracker）和作业的健康状况，一旦发现失败，就会将相

应的任务转移到其他节点。ResourceManager 会跟踪任务的执行进度、资源使用量等信息，并将这些信息提供给任务调度器。

3. NodeManager

NodeManager 取代了 Hadoop 1.x 版本中的 TaskTracker。它运行在集群中的每个节点上，负责管理本节点上的资源和任务执行。NodeManager 会周期性地通过心跳机制将本节点上资源的使用情况和任务的运行进度汇报给 ResourceManager。同时，它接收 ResourceManager 发送过来的命令并执行相应的操作（如启动新任务、杀死任务等）。

4. Task

Task 仍然分为 Map Task 和 Reduce Task 两种，它们由 ApplicationMaster 启动。在 HDFS 中，数据以固定大小的块为基本单位存储，而在 MapReduce 任务处理过程中，数据被划分为数据分片。Map Task 负责处理输入数据的分片，将其转换为键值对，然后将结果传递给 ApplicationMaster。Reduce Task 负责处理 Map Task 的输出结果，并生成最终的输出。任务的分发、跟踪和执行等工作由 ApplicationMaster、ResourceManager 和 NodeManager 共同协调完成。

用户提交一个应用程序时，ResourceManager 为该应用程序分配一个 ApplicationMaster。ApplicationMaster 运行在集群中的一个计算节点上，并负责向 ResourceManager 请求所需的资源，例如内存和 CPU。一旦 ApplicationMaster 获得了所需资源，它就负责协调应用程序中的各任务的执行。具体而言，ApplicationMaster 会与 ResourceManager 交互，以获取合适的计算节点来执行 Map Task 和 Reduce Task，并确保任务在节点上成功运行。同时，ApplicationMaster 会监控任务的执行进度和健康状态，并将相关信息汇报给 ResourceManager。这样，ResourceManager 可以根据任务的执行情况进行动态的资源分配和作业调度，以实现集群资源的最优利用。

综上所述，HDFS 和 MapReduce 仍然共同组成了 Hadoop 分布式集群的核心。HDFS 提供了分布式文件系统的支持，而 MapReduce 实现了分布式计算和任务处理。HDFS 在 MapReduce 任务处理过程中提供了对文件操作和存储等的支持，而 MapReduce 在 HDFS 基础上实现了任务的分发、跟踪和执行等工作，两者相互作用，完成了 Hadoop 分布式集群的主要任务。Hadoop 3.x 版本通过引入 ResourceManager 和 NodeManager 的改进，提高了集群的稳定性和扩展性。

5.3.4　MapReduce 任务提交详解

对于用户提交的任务，Hadoop 是如何调度并执行的呢？

从 Hadoop 3.x 的架构可以看出，Hadoop 作业的执行主要由 ResourceManager、NodeManager 和 ApplicationMaster 负责完成。

客户端编写好的 Hadoop 作业是一个 Job，Job 被提交给 ResourceManager 后，ResourceManager 会为该 Job 分配一个全局唯一的 Application ID，并且创建一个 ApplicationMaster 来管理该作业的执行。接着，ResourceManager 检查该 Job 指定的输出目录是否存在、输入文件是否存在，如果不存在，则抛出错误。同时，ResourceManager 会根据输入文件计算输入分片。这些检查通过后，ResourceManager 会为 Job 配置所需的资源，并分配 NodeManager 来运行 ApplicationMaster。

ApplicationMaster 运行在集群中的某个计算节点上，负责与 ResourceManager 交互请求资源，并与 NodeManager 协调任务的执行。一旦 ApplicationMaster 获得了所需资源，它就会协调应用程序中的各任务的执行。具体而言，ApplicationMaster 会与 ResourceManager 交

互，以获取合适的计算节点来执行 Map Task 和 Reduce Task，并监控任务的执行进度和健康状态。

Job 被分配到 NodeManager 上的计算节点后，ApplicationMaster 会向该节点的 NodeManager 发送任务启动请求，即启动 Map Task 和 Reduce Task。NodeManager 接收到启动请求后，在本节点上执行 Map Task 和 Reduce Task。

作业调度器通过心跳机制监控 ApplicationMaster 和 NodeManager 的状态和进度，并计算出整个 Job 的状态和进度。当 Job 的所有任务都成功完成时，ApplicationMaster 会通知 ResourceManager，ResourceManager 将整个 Job 状态置为成功。当查询 Job 运行状态时（注意，这是一个异步操作），客户端会获得 Job 完成的通知。如果 Job 中途失败，MapReduce 也会有相应的机制处理。一般情况下，如果不是程序本身存在缺陷（bug），MapReduce 的错误处理机制能够保证提交的 Job 能正常完成。

5.4 MapReduce 应用案例

如同在学习一门编程语言时总是会以 HelloWorld 输出作为第一个入门案例，在大数据计算框架的学习中，往往会采用 WordCount 案例作为第一个编程实例，因为 WordCount 案例既兼顾了数据对磁盘的压力，又兼顾了单词排序时对内存的压力，最后以统计时间作为性能评判标准。从需求而言又相对简单易懂，对于大数据计算框架初学者而言是一个相对合适的案例。下面将对这个案例进行分析和实现。

5.4.1 文本分词统计

1. 需求分析

WordCount 需求和其单词直译表述一致，即单词统计。具体需求为：给定一个全部由单词组成的文件，统计文件中每个单词出现的次数。

对于单纯的单词统计而言，相信掌握基本编程语言的读者都可以轻易实现。通用的实现思路是：将文件进行全局排序，使所有的单词按一定顺序排好，相同单词在一起。随后进行遍历，逐个取出其中的每个单词，对于重复单词而言，计数器加一；当出现不同单词时，将上一个单词和计数器的数字输出，然后计数器清零，开始统计下一个单词出现的次数，直到所有单词统计完成为止。

使用上述步骤进行编码的开发，任何一门语言都是可以实现 WordCount 需求的，但是在大数据的需求场景之下，却会发现很明显的问题。因为上述实现思路的第一步是全局排序，而计算机基础知识告诉我们，排序只能发生在内存当中，因此内存的大小决定了以此设计思路进行编码开发时所能统计的单词文件总大小。

在现实企业环境中，每天的增量数据往往是以 TB 计的，而企业全量数据更是 PB（拍字节）甚至 EB（艾字节）级的数量级，未来还可能更多，因而数据的增长必定超过单服务器内存的增长，且现在企业服务器也多是 GB（吉字节）级内存。

不难看出，就最简单的单词统计需求来说，在不考虑数据量的情况下，实现是很简单的，但是当数据量超过单台服务器内存极限时，曾经的实现方式又无法采用任何编程语言完成了。也就是说，针对大数据的场景时，需要重新思考一种全新的编程模型与实现思路，这就是本章重点介绍的 MapReduce。

2. 实现思路

该案例的整体设计思路仍然是排序、遍历、输出这样一个过程，而需要变化的是“分而治之”这样设计思想的一个引入。具体来说，基于分布式文件存储 HDFS，可以针对每个块进行单独排序，随后将局部完成排序的若干小文件 a1～an，按照单词顺序，归并排序到一个新的文件 b，而此时文件 b 则是一个全局排序完成的文件了，再将文件 b 进行遍历，输出累加值即可。

上述排序的小文件 a1～an 的处理过程，在 MapReduce 编程模型中被称为 Map 阶段；文件 b 的处理过程，在 MapReduce 编程模型中被称为 Reduce 阶段。

3. 案例实现

MapReduce 程序开发涉及在 Windows 本地环境中进行开发和测试，然后将项目打包在分布式集群环境中运行，以保证开发的高效性。因此，首先需要准备 Windows 基础环境并在该环境下完成功能测试。在开发过程中，一般会使用 Maven 工程来管理依赖，并针对 Map 和 Reduce 核心代码编写相应的模块。整个开发过程需要遵循 MapReduce 框架的设计规范，将数据处理逻辑分解成 Map、Reduce 和 Driver 3 个模块，以实现分布式计算和数据处理的目标。

1）环境准备

（1）检查 Windows 环境是否正常，具体详情可回顾 4.3 节内容。

（2）针对 Maven 工程导入如下 pom 文件依赖。

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.8.2</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>3.3.6</version>
        <exclusions>
            <exclusion>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-log4j12</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.3.6</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-hdfs</artifactId>
        <version>3.3.6</version>
    </dependency>
```

```
        <dependency>
            <groupId>jdk.tools</groupId>
            <artifactId>jdk.tools</artifactId>
            <version>1.8</version>
            <scope>system</scope>
            <systemPath>${JAVA_HOME}/lib/tools.jar</systemPath>
        </dependency>
    </dependencies>
```

2）核心代码

MapReduce 基础核心代码分为 Map、Reduce 和 Driver 3 个模块。下面逐一对每个模块进行详细介绍。

（1）Map 阶段。

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;
public class WordCountMap extends Mapper<LongWritable, Text, Text, IntWritable>{
    private final Text text = new Text();
    private final IntWritable one = new IntWritable(1);
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException{
        String line = value.toString();
        String[] words = line.split(" ");
        for (String word : words) {
            text.set(word);
            context.write(text, one);
        }
    }
}
```

首先需要明确的是，MapReduce 不是某个编程语言，而是一种分布式并行计算框架。简单来说，MapReduce 提供了一套大数据分布式处理的标准模板。这个模板中预先实现了许多功能，例如分布式计算、排序和迭代计算等。然而，框架设计者无法预先了解用户需求的多样性与独特性。同样，对于框架的使用者，完全手动实现诸如分布式计算等功能将失去使用框架的意义。因此，MapReduce 框架提前提取了大数据处理中的公共逻辑部分，并留出足够的开放接口供使用者按照规范进行自定义实现。

回到 Map 部分的代码，类定义中的继承语句也就不难理解了。

```
public class WordCountMap extends Mapper<LongWritable, Text, Text, IntWritable>
```

在使用 MapReduce 设计者预先定义的模板和功能时，需要通过类继承来继承之前定义的功能，其中 Mapper 类就是其中之一。

紧接着是 4 个泛型参数，它们分别代表 Map 阶段的输入 key 的类型、输入 value 的类型、输出 key 的类型和输出 value 的类型。根据前面对 MapReduce 编程模型的介绍，可以了解到 MapReduce 的处理过程都是以键值对的形式存在的。因此，在 Map 阶段进行数据读取时，MapReduce 就已经开始使用键值对的方式接收数据了。具体形式的键值对如何进行接收将

在下面详细介绍。最后需要注意的是，此处的泛型类型都是一些复杂的序列化包装类，关于包装类与普通数据类型的关系如表 5-1 所示。

表 5-1　Java 普通数据类型与 Hadoop 包装类的对应关系

Java 类型	Hadoop Writable 类型	Java 类型	Hadoop Writable 类型
Boolean	BooleanWritable	Double	DoubleWritable
Byte	ByteWritable	String	Text
Int	IntWritable	Map	MapWritable
Float	FloatWritable	Array	ArrayWritable
Long	LongWritable		

所谓序列化，就是把内存中的对象转换为字节序列或其他数据传输协议，以便于存储到磁盘或进行网络传输。在大数据处理中，通常需要将数据持久化保存，或者将数据发送到远程计算机进行后续的计算。序列化是实现这些需求的关键。

一般来说，可计算和排序的数据只存在于内存中，一旦关机或断电，数据将会丢失。此外，这样的数据只能由本地的进程使用，不能被发送到网络上的其他计算机。然而，序列化技术可以存储这样的可计算数据，并且可以将它们发送到远程计算机进行后续的计算。

Java 本身提供了一种序列化对象类型，即 Serializable 接口，但它是一个重量级序列化框架。序列化后的对象附带许多额外的信息，例如校验信息、Header(序列化对象的头部信息)和继承体系，这不利于在网络中高效传输数据。为了解决这个问题，Hadoop 自己开发了一套序列化机制，即 Writable 接口。

在 MapReduce 中，“Text text = new Text();”和 “IntWritable one = new IntWritable(1);”这两句代码是对 Map 阶段输出数据的定义。根据继承类的泛型，可以知道 Map 阶段所需要输出的是 Text 类型和 IntWritable 类型。由于这些都是复杂对象类型，因此需要通过构造函数进行对象的创建和初始化。

在 Mapper 类中，map 方法是继承自 Mapper 类的方法。该方法在底层已经实现了一些功能。在默认情况下，MapReduce 按行读取数据，一次读取一行数据，因此 Map 阶段的输入 key 是每行数据的下标索引，而输入 value 是每行的具体内容。

在 Map 阶段的 map 方法中，“String line = value.toString();” 用于将 Text 类型的输入 value 转换为 Java 的 String 类型进行操作。“String[] words = line.split(" ");” 将一行内容按空格进行拆分，得到一个包含每个单词的 String 数组 words，这将用于后续的单词统计。

接下来，使用增强 for 循环遍历 words 数组，取出其中的每个单词。然后，将每个单词设置到之前定义的 Text 类型的 text 中，并将 text 和预先初始化为 1 的 IntWritable 类型的 one 写入 context(上下文)中。Map 阶段会调用这个方法多次，每次都处理一行数据，并将单词作为 key、1 作为 value 写入 context 中。

需要明确的是，由于 MapReduce 是基于 Hadoop 框架的核心组成部分之一，Map 阶段的执行是按照分布式并行处理的逻辑进行运算。每个 Map 任务针对不同的 HDFS 块进行分布式并行处理，而不是全局键值对映射。在 Map 阶段完成后，数据已经以(单词，1)的形式映射，为后续的 Reduce 阶段做好了准备，Reduce 阶段将只需要对相同 key 的数据进行 value 值的累加即可完成单词统计的需求。

(2) Reduce 阶段。

通过前面对 MapReduce 编程模型的介绍，已经知道 Map 阶段主要将数据转换为键值对的映射，而 Reduce 阶段主要用于迭代计算。在本次需求中，Map 将文本数据映射成了单词和

1 的键值对，而 Reduce 对每个键值对进行遍历，那么排序在哪里执行呢？如果没有排序，Reduce 的遍历将会是无序的。

实际上，MapReduce 作为大数据计算框架，在默认情况下已经提供了默认的排序方式，即按照键值对中 key 的字典序进行全局排序。这个排序过程同样是分布式进行的。在每个块所在的节点完成键值对映射后，借助节点内存完成局部排序。一个块的大小默认为 128MB，因此服务器可以承受并快速完成局部排序。

完成每个块节点的局部排序后，再使用归并排序的方式，逐一取出每个节点的数据，按照相同 key 在一起的顺序发送到需要进行 Reduce 计算的节点，保持每个节点局部排序的顺序，以此完成在 Reduce 节点的全局排序。

因此，到达 Reduce 节点的数据已经是一个完成了全局排序的数据。这个排序过程发生在 Map 阶段之后和 Reduce 阶段之前，并由 MapReduce 框架自动完成。当然，未来在学习 MapReduce 高阶内容时，会详细介绍如何控制排序方式，以及自定义更多的排序条件。但在本次需求中，可以信赖 MapReduce 框架默认的全局排序，以便正确执行 Reduce 阶段的逻辑。

当明确了 Map 阶段和 Reduce 阶段发生的事情后，再来看一下 Reduce 阶段的代码。

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;
public class WordCountReduce extends Reducer<Text, IntWritable, Text, IntWritable>{
    private IntWritable result = new IntWritable();
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException{
        int sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
public class WordCountReduce extends Reducer<Text, IntWritable,Text,IntWritable>
```

在 Reduce 阶段中，与 Map 阶段类似，Reduce 阶段的类也继承自 MapReduce 框架的 Reducer 类，用于调用针对分布式处理的方法。Reducer 类有 4 个泛型参数，其作用与 Map 阶段的泛型相同。这 4 个泛型分别表示 Reduce 阶段的输入 key 类型、输入 value 类型、输出 key 类型和输出 value 类型，并且同样需要遵循 Hadoop 的序列化类型规范。

需要注意的是，Reduce 阶段的输入 key 和 value 的类型总是与 Map 阶段的输出 key 和 value 类型一致。这是因为 Map 阶段的输出数据是为 Reduce 阶段的输入做准备的。尽管在 MapReduce 框架中会为用户执行排序等操作，但基本数据格式保持不变。Map 阶段的输出格式决定了 Reduce 阶段的输入格式。对于当前的单词统计需求，Map 阶段将数据映射成了（key：单词，value：1）这样的 Text 和 IntWritable 类型作为输出，因此 Reduce 阶段的输入数据格式也是相同类型。

最终的目标是统计每个单词出现的次数，因此 Reduce 阶段的输出类型也是 Text 和 IntWritable，用于保存单词和对应的统计次数。

```
private IntWritable result = new IntWritable();
```

在 Reduce 阶段的代码中，首先定义了一个 IntWritable 类型的变量 result，用于存储每个单词的统计总数。这个变量将被放入 context 中，因此必须是 Hadoop 的序列化类型。由于 reduce 方法会被框架循环执行，为了避免不断地创建新的变量，result 在 reduce 方法外部进行定义，以节省资源并提高效率。

```
protected void reduce(Text key, Iterable<IntWritable> values, Context context)
```

reduce 方法是继承自 Reducer 类的，和 Mapper 类中的 map 方法类似，reduce 方法在底层已经实现了一些功能。reduce 方法的输入数据来自 Map 阶段的输出数据。在 Map 阶段，输出数据是一组以相同单词为 key、1 为 value 的数据，但在 reduce 方法中，这些数据被组织成一个 Iterable 类型的迭代器。这是因为 MapReduce 框架在完成全局排序后，将相同 key 的 value 数据放入同一个迭代器中，每个 reduce 方法将处理一个 key 及其对应的所有 value。因此，reduce 方法会按照不同的 key 进行循环执行，对每个 key 的 value 进行迭代计算。

reduce 方法的第三个参数 context 是整个 MapReduce 框架的数据主线，用户需要将计算得到的数据填入其中，并遵循 Reducer 类所定义的后两个泛型数据类型。

```
int sum = 0;
```

为了进行累加计算，首先在 reduce 方法内部定义一个名为 sum 的变量，用于计算每个 key 的总数。由于每次 reduce 方法重新执行时，都需要将 sum 变量清零，它被定义为基本数据类型 int，不需要考虑内存占用问题。

```
for(IntWritable value: values){
sum = sum + value.get();
}
```

通过增强 for 循环遍历迭代器对象 values，取出 values 中的每一个值。对于当前需求，values 实际上是一系列的 1，将这些值累加到 sum 变量中，得到了当前 key 出现的总次数，即当前单词的统计次数。

```
result.set(sum);
context.write(key,result);
```

将 sum 的值赋给 IntWritable 类型的变量 result，因为 sum 是基本数据类型，无法直接传输给 context。然后将 result 作为一个完整的键值对，与当前 key 一起放入 context 中，完成对当前单词的统计工作。随后，框架会不断循环调用 reduce 方法，实现对所有键值对的迭代计算，从而得到每个单词的出现次数。

（3）Driver 阶段。

在定义好 Map 阶段和 Reduce 阶段后，还需要用户设置当前任务的主程序。也就是说，用户需要显示地声明使用哪一个 Map 和使用哪一个 Reduce，包括任务名称、程序的输入输出目录等，都需要在此处定义。MapReduce 作为一个计算框架，本身是拥有许多默认设置的，但也有一些设置需要用户显式地进行声明，这些声明都在 Driver 类中进行。

```
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import java.io.IOException;
public class WordCountDriver {
     public static void main(String[] args) throws IOException, ClassNotFoundException,
InterruptedException {
         //创建任务
         Configuration conf = new Configuration();
         Job job = Job.getInstance(conf);
         //设置任务名字
         job.setJobName("MyWordCount");
         //设置主类名字
         job.setJarByClass(WordCountDriver.class);
         //设置 Map 类
         job.setMapperClass(WordCountMap.class);
         //设置 Reduce 类
         job.setReducerClass(WordCountReduce.class);
         //设置 Map 阶段输出的 key 的数据类型
         job.setMapOutputKeyClass(Text.class);
         //设置 Map 阶段输出的 value 的数据类型
         job.setMapOutputValueClass(IntWritable.class);
         //设置 Reduce 的输出的 key 数据类型
         job.setOutputKeyClass(Text.class);
         //设置 Reduce 阶段的输出 value 数据类型
         job.setOutputValueClass(IntWritable.class);
         //设置程序的输入目录和输出目录
         //本地运行时,直接指定桌面上的 output 文件夹作为输出路径
          FileInputFormat.setInputPaths(job, new Path("file:///C:/Users/<username>/Desktop/
word.txt"));
          FileOutputFormat.setOutputPath(job, new Path("file:///C:/Users/<username>/Desktop/
output"));
         //执行任务并等待完成
         boolean success = job.waitForCompletion(true);
         System.exit(success ? 0 : 1);
     }
}
```

代码本身对许多内容做出了说明,此处不一一赘述。仅对其中需要特别强调和注意的部分做出说明和解释。

```
FileInputFormat.setInputPaths(job, new Path("file:///C:/Users/<username>/Desktop/word.txt"));
FileOutputFormat.setOutputPath(job, new Path("file:///C:/Users/<username>/Desktop/output"));
```

此处采用给定的参数设置输入和输出路径,也可以改为接收外部参数的方式。输入文件代表任务需要统计单词的文件,输出目录代表任务需要将统计结果输出到什么位置。

4. 案例测试

在上述的 WordCountDriver 代码中,已经设置了 MapReduce 任务的输入路径为桌面上的 word.txt 文件,并将输出结果保存在桌面上的 output 文件夹中。现在来测试验证这个任务是否能够正确地统计 word.txt 文件中每个单词出现的次数。

首先,将 word.txt 文件的内容保存到桌面上,文件中的内容如下所示。

```
hadoop hdfshdfs hbase spark flink mepreduce
hdfs hadoop flink hive hive spark
habse flink flink hdfs hbase spark flink
hdfs hbase spark flink hdfs hbase spark flink
hdfs hbase spark flink
hdfs hbase spark flink hdfs hbase spark flink
hdfs hbase spark flink hdfs hbase spark flink hdfs hbase spark flink
```

接下来，编译并运行 WordCountDriver 类。MapReduce 任务将会执行，读取 word.txt 文件并进行单词统计，结果将保存在桌面上的 output 文件夹中。

然后，查看输出结果。打开桌面上的 output 文件夹，将会看到生成的输出文件（通常是 part-r-00000）。该文件中包含了每个单词及其对应的出现次数。结果如图 5-12 所示。

输出结果显示了每个单词及其在 word.txt 中出现的次数。与预期的一致，这样就验证了 MapReduce 任务的正确性。

```
flink      13
habse      1
hadoop     2
hbase      10
hdfs       10
hdfshdfs   1
hive       2
mepreduce          1
spark      11
```

图 5-12　WordCount 案例的输出文件

5. MapReduce 编程规范

用户编写的 MapReduce 程序分成 3 部分：Mapper、Reducer 和 Driver。

1）Mapper

（1）用户自定义的 Mapper 要继承自己的父类。

（2）Mapper 的输入数据是键值对的形式，其中键值对的类型可自定义。

（3）Mapper 中的业务逻辑写在 map 方法中。

（4）Mapper 的输出数据是键值对的形式，其中键值对的类型可自定义。

（5）map 方法对每个输入的键值对调用一次，在默认情况下是一行数据调用一次。

2）Reducer

（1）用户自定义的 Reducer 要继承 MapReduce 的 Reducer 父类。

（2）Reducer 的输入数据类型对应 Mapper 的输出数据类型，也是一个键值对。

（3）Reducer 的业务逻辑写在 reduce 方法中。

（4）ReduceTask 进程对每一组相同 key 的键值对组调用一次 reduce 方法。

3）Driver

Driver 相当于 YARN 集群的客户端，用于提交用户整个程序到 YARN 集群，提交的是封装了 MapReduce 程序相关运行参数的 job 对象。

5.4.2　海量数据排序

排序是 MapReduce 框架中最重要的操作之一。Map 任务和 Reduce 任务均会对数据按照 key 进行排序。默认排序是按照字典顺序排序，实现该排序的方法是快速排序。

对于 Map 任务，它会将处理的结果暂时放到一个缓冲区中，当缓冲区使用率达到一定阈值后，再对缓冲区中的数据进行一次排序，并将这些有序数据写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行一次合并，以将这些文件合并成一个大的有序文件。

对于 Reduce 任务，它从每个 Map 任务上远程复制相应的数据文件，如果文件大小超过一定阈值，则放到磁盘上，否则放到内存中。如果磁盘上文件数目达到给定阈值，则进行一次合并以生成一个更大文件；如果内存中文件大小或者数目超过一定阈值，则进行一次合并后将数据写到磁盘上。当所有数据复制完毕后，Reduce 任务统一对内存和磁盘上的所有数据进行一次归并排序。

将 MapReduce 中的排序做以下分类。

- 部分排序：MapReduce 根据输入记录的键对数据集排序，保证输出的每个文件内部有序。
- 全局排序：最终输出结果只有一个文件，且文件内部有序。实现方式是只设置一个 Reduce 任务。但该方法在处理大型文件时效率极低，因为一台机器处理所有文件，完全丧失了 MapReduce 所提供的并行架构。
- 分组排序：在 Reduce 端对 key 进行分组。在接收的 key 为 bean 对象时，想让一个或几个字段相同(全部字段比较不相同)的 key 进入同一个 reduce 方法时，可以采用分组排序。
- 二次排序：在自定义排序过程中，调用自定义对象 IntPair 的 compare 方法进行第二次排序。如果 compareTo 中的判断条件为两个即为二次排序。

1. 案例需求

现在有一份日志文件，里面包含了一部分用户访问网站的流量数据，字段分别为手机号、上行流量、下行流量、总流量。数据中，每个字段之间都用空格进行了分隔。

日志文件名为 phone_data.txt，数据如下：

```
13726230503 2481 24681 27162
13826544101 264 0 264
13926435656 131 1512 1643
13926251106 0 240 240
18211575961 527 2160 2687
84138413 4116 1432 5548
13560439658 1116 954 2070
15920133257 3156 2936 6092
13719199419 240 0 240
13660577991 6960 690 7650
15013685858 1938 3538 5476
15989002119 180 180 360
13560439658 918 4938 5856
13480253104 1938 2910 4848
13602846565 3008 3720 6728
13922314466 210 210 420
13502468823 2845 2430 5284
18320173382 9531 2410 11941
13925057413 7335 110349 117684
13760778710 2481 24681 27162
13560436666 1116 654 1770
13560436666 1720 2420 4140
```

要求根据手机号的总流量进行倒序排序，以此为依据获取当前活跃用户。

2. 原理分析

Writable 是 Hadoop 的序列化格式，Hadoop 定义了这样一个 Writable 接口. 一个类要支持可序列化只需实现这个接口即可。

另外，Writable 有一个子接口是 WritableComparable，WritableComparable 既可实现序列化，又可以对 key 进行比较，这里可以通过自定义 key 实现 WritableComparable 来实现排序功能。

自定义 bean 对象作为 key 传输，需要实现 WritableComparable 接口重写 compareTo 方法，就可以实现排序。

```
@Override
public int compareTo(FlowBean o) {
    int result;
    //按照总流量大小,倒序排列
    if (sumFlow > o.getSumFlow()) {
        result = -1;
    } else if (sumFlow < o.getSumFlow()) {
        result = 1;
    } else {
        result = 0;
    }
    return result;
}
```

MapReduce 框架中 Shuffle 阶段的排序是默认行为,不管是否需要都会进行排序把所有的字段封装成 bean 对象,并且指定 bean 对象作为 key 输出。

3. 案例实现

在 IDEA 中创建一个新的项目 mr-sort,项目的 pom 文件与前面 WordCount 案例一样。

(1) 自定义分区类。

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.WritableComparable;
public class FlowBean implements WritableComparable<FlowBean> {
    private long upFlow;      //上行流量
    private long downFlow;    //下行流量
    private long sumFlow;     //总流量
    public FlowBean() {
        super();
    }
    public FlowBean(long upFlow, long downFlow) {
        super();
        this.upFlow = upFlow;
        this.downFlow = downFlow;
        sumFlow = upFlow + downFlow;
    }
    //序列化
    @Override
    public void write(DataOutput out) throws IOException {
        out.writeLong(upFlow);
        out.writeLong(downFlow);
        out.writeLong(sumFlow);
    }
    //反序列化
    @Override
    public void readFields(DataInput in) throws IOException {
        upFlow = in.readLong();
        downFlow = in.readLong();
        sumFlow = in.readLong();
    }
    //比较
    @Override
    public int compareTo(FlowBean bean) {
```

```
            int result;
            //核心比较条件判断
            if (sumFlow > bean.getSumFlow()) {
                result = -1;
            } else if (sumFlow < bean.getSumFlow()) {
                result = 1;
            } else {
                result = 0;
            }
            return result;
        }
        public long getUpFlow() {
            return upFlow;
        }
        public void setUpFlow(long upFlow) {
            this.upFlow = upFlow;
        }
        public long getDownFlow() {
            return downFlow;
        }
        public void setDownFlow(long downFlow) {
            this.downFlow = downFlow;
        }
        public long getSumFlow() {
            return sumFlow;
        }
        public void setSumFlow(long sumFlow) {
            this.sumFlow = sumFlow;
        }
        @Override
        public String toString() {
            return upFlow + "\t" + downFlow + "\t" + sumFlow;
        }
    }
```

自定义分区 FlowBean 类实现了 WritableComparable 接口，包含 3 个属性，分别代表上行流量、下行流程、总流量。在 FlowBean 中重写了 compareTo 方法，按照总流量排倒序。compareTo 方法用于将当前对象与方法的参数进行比较。

- 如果指定的数与参数相等则返回 0。
- 如果指定的数小于参数则返回－1。
- 如果指定的数大于参数则返回 1。

（2）编写 Mapper 类。

```
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class FlowCountSortMapper extends Mapper<LongWritable, Text, FlowBean, Text>{
    FlowBean bean = new FlowBean();
    Text v = new Text();
    @Override
     protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
```

```
            //1.获取一行数据
            String line = value.toString();
            //2.分割字段
            String[] fields = line.split("\t");
            //3.封装对象
            String phoneNbr = fields[0];
            long upFlow = Long.parseLong(fields[1]);
            long downFlow = Long.parseLong(fields[2]);
            bean.setUpFlow(upFlow);
            bean.setDownFlow(downFlow);
            bean.setSumFlow(upFlow + downFlow);
            v.set(phoneNbr);
            //4.输出
            context.write(bean, v);
        }
    }
```

map 方法中读取了日志文件中的数据，把流量数值赋值给 FlowBean 对象的属性，此时 Mapper 输出的 key 是记录流量数值的 FlowBean 对象，value 是这一行日志数据。默认按照 key 进行排序，即根据 FlowBean 中重写的 compareTo 方法，进行排序。

（3）编写 Reducer 类。

```
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class FlowCountSortReducer extends Reducer<FlowBean, Text, Text, FlowBean>{
    @Override
    protected void reduce(FlowBean key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
        //循环输出,避免总流量相同情况
        for (Text text : values) {
            context.write(text, key);
        }
    }
}
```

这个 Reducer 类用于处理 MapReduce 过程中的排序结果。根据 FlowBean 对象的总流量进行排序，将结果按照降序输出。在总流量相同的情况下，为了避免数据被覆盖，需要循环输出相同总流量的结果。最终输出的键值对是< Text，FlowBean >，其中 Text 表示手机号码，FlowBean 表示对应的流量数据。

（4）编写 Driver 类。

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class FlowCountSortDriver {
     public static void main(String[] args) throws ClassNotFoundException, IOException,
InterruptedException {
        //1.获取配置信息,或者 job 对象实例
```

```
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);
        //2.指定本程序的 jar 包所在的本地路径
        job.setJarByClass(FlowCountSortDriver.class);
        //3.指定本业务 job 要使用的 Mapper/Reducer 业务类
        job.setMapperClass(FlowCountSortMapper.class);
        job.setReducerClass(FlowCountSortReducer.class);
        //4.指定 Mapper 输出数据的 key-value 类型
        job.setMapOutputKeyClass(FlowBean.class);
        job.setMapOutputValueClass(Text.class);
        //5.指定最终输出的数据的 key-value 类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);
        //6.指定 job 的输入原始文件所在路径
        FileInputFormat.setInputPaths(job, new Path("C:/Users/YourUsername/Desktop/phone_data.
txt"));
        FileOutputFormat.setOutputPath(job, new Path("C:/Users/YourUsername/Desktop/output2"));
        //7.将 job 中配置的相关参数,以及 job 所用的 Java 类所在的 jar 包,提交给 YARN 去运行
        boolean result = job.waitForCompletion(true);
        System.exit(result ? 0 : 1);
    }
}
```

这个代码是驱动程序,用于配置和提交 MapReduce 任务,并将 MapReduce 任务的输出结果保存在指定的输出路径。

4. 案例测试

在上述的 FlowCountSortDriver 代码中,已经设置了 MapReduce 任务的输入路径为桌面上的 phone_data.txt 文件,并将输出结果保存在桌面上的 output2 文件夹中。现在来测试验证这个任务是否能够对输入的流量数据进行排序。

13925057413	7335	110349	117684
13760778710	2481	24681	27162
13726230503	2481	24681	27162
18320173382	9531	2410	11941
13660577991	6960	690	7650
13602846565	3008	3720	6728
15920133257	3156	2936	6092
13560439658	918	4938	5856
84138413	4116	1432	5548
15013685858	1938	3538	5476
13502468823	2845	2430	5275
13480253104	1938	2910	4848
13560436666	1720	2420	4140
18211575961	527	2160	2687
13560439658	1116	954	2070
13560436666	1116	654	1770
13926435656	131	1512	1643
13922314466	210	210	420
15989002119	180	180	360
13826544101	264	0	264
13719199419	240	0	240
13926251106	0	240	240

图 5-13 排序结果

接下来,编译并运行 FlowCountSortDriver 类。MapReduce 任务将会执行,读取 phone_data.txt 文件并进行自定义的排序,结果将保存在桌面上的 output2 文件夹中。

然后,查看输出结果。打开桌面上的 output2 文件夹,将会看到生成的输出文件 part-r-00000。结果如图 5-13 所示。

如图 5-13 所示,其中每一行包含一个手机号码和对应的上行流量与下行流量。手机号码将按照总流量大小从大到小排列,最大的总流量排在前面。至此,通过 MapReduce 完成了对输入的流量数据的排序和统计任务,方便后续分析和处理。

本章小结

在本章中,深入研究了分布式计算模型 MapReduce,该模型作为大数据处理领域的重要工具,在实践中展现出了强大的能力和广泛的应用。通过概述、工作原理、入门案例(WordCount)和进阶案例(排序),探索了 MapReduce 的核心概念、实现原理以及在实际场景中的应用。

MapReduce 作为一种强大的分布式计算框架,在处理大规模数据时表现出了显著的优

势。其简单易用的特点使得开发者可以更加专注于业务逻辑的实现，而无须过多关注底层的分布式计算细节。同时，MapReduce 的可扩展性也使得其能够轻松应对不断增长的数据规模，为大数据处理提供了可靠的解决方案。

通过本章的学习，读者可以了解到 MapReduce 是如何通过将任务分解成 Map 阶段和 Reduce 阶段，并通过中间结果的分区和合并来实现数据的并行处理和计算的。此外，还通过实际案例展示了 MapReduce 在不同场景下的应用，包括简单的数据统计和复杂的数据排序，为读者提供了丰富的编程实践经验。

随着大数据技术的不断发展和完善，MapReduce 作为其中的重要组成部分将继续在各领域发挥重要作用。从互联网企业到金融、医疗、电商等各行业，MapReduce 都有着广泛的应用场景，为数据处理和分析提供了强大的支持。同时，随着人工智能、机器学习等技术的不断成熟，MapReduce 也将与这些新兴技术相结合，为数据驱动的创新和发展注入新的动力。

在未来，可以通过不断学习和实践，更好地利用 MapReduce 等工具，应对日益增长的数据挑战，推动数据处理和分析的进步。同时，也需要关注 MapReduce 在实际应用中可能遇到的挑战和限制，不断优化和改进其性能和功能，以满足不断变化的数据处理需求。

综上所述，MapReduce 作为分布式计算领域的重要技术，将在未来继续发挥着重要作用，为大数据处理和分析提供了可靠的解决方案，推动数据驱动的创新和发展。

课后习题

1. 单选题

（1）MapReduce 是由（　　）公司首先提出并实现的。

A. Apache Software Foundation　　B. Google

C. IBM　　D. Microsoft

（2）Hadoop MapReduce 运行在（　　）上。

A. GFS　　B. HDFS　　C. NTFS　　D. FAT32

（3）Hadoop 2.0 引入了（　　）新技术来取代旧版的 MapReduce 作业调度器。

A. YARN　　B. HBase　　C. Spark　　D. Flink

（4）MapReduce 模型中的 Map 阶段的主要作用是（　　）。

A. 对数据进行全局汇总

B. 将复杂的任务分解为若干简单的任务

C. 合并中间结果

D. 排序数据

（5）在 MapReduce 中，Shuffle 阶段的主要作用是（　　）。

A. 将数据从 Map 任务传输到 Reduce 任务

B. 对数据进行预处理

C. 对数据进行安全加密

D. 将数据存储到磁盘

（6）MapReduce 的优点不包括（　　）。

A. 易于编程　　B. 具有良好的扩展性

C. 适合实时计算　　D. 具有高容错性

（7）在 MapReduce 的(　　)可以实现对数据的迭代计算。

A. Map 阶段　　B. Shuffle 阶段　　C. Reduce 阶段　　D. Cleanup 阶段

（8）Hadoop 生态系统中,(　　)组件是用来进行实时数据管道和流处理应用程序的。

A. Sqoop　　B. Flume　　C. Kafka　　D. Hive

（9）在 MapReduce 任务执行过程中,负责全局资源管理和作业调度的组件是(　　)。

A. ApplicationMaster　　B. ResourceManager

C. NodeManager　　D. TaskTracker

（10）MapReduce 中的合并是在(　　)阶段使用的。

A. Map　　B. Shuffle　　C. Reduce　　D. Cleanup

2. 思考题

（1）MapReduce 模型为什么在大数据处理中得到广泛应用?

（2）在 MapReduce 任务中,为什么需要进行 Shuffle 阶段?