第 1 章

Hadoop 简介

本章内容

- 什么是 Hadoop
- Hadoop 项目及其结构
- Hadoop 体系结构
- Hadoop 与分布式开发
- Hadoop 计算模型——MapReduce
- Hadoop 数据管理
- Hadoop 集群安全策略
- 本章小结

1.1 什么是 Hadoop

1.1.1 Hadoop 概述

Hadoop 是 Apache 软件基金会旗下的一个开源分布式计算平台。以 Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）和 MapReduce（Google MapReduce 的开源实现）为核心的 Hadoop 为用户提供了系统底层细节透明的分布式基础架构。HDFS 的高容错性、高伸缩性等优点允许用户将 Hadoop 部署在低廉的硬件上，形成分布式系统；MapReduce 分布式编程模型允许用户在不了解分布式系统底层细节的情况下开发并行应用程序。所以用户可以利用 Hadoop 轻松地组织计算机资源，从而搭建自己的分布式计算平台，并且可以充分利用集群的计算和存储能力，完成海量数据的处理。经过业界和学术界长达 10 年的锤炼，目前的 Hadoop 1.0.1 已经趋于完善，在实际的数据处理和分析任务中担当着不可替代的角色。

1.1.2 Hadoop 的历史

Hadoop 的源头是 Apache Nutch，该项目始于 2002 年，是 Apache Lucene 的子项目之一。2004 年，Google 在“操作系统设计与实现”（Operating System Design and Implementation，OSDI）会议上公开发表了题为 *MapReduce: Simplified Data Processing on Large Clusters*（《MapReduce: 简化大规模集群上的数据处理》）的论文之后，受到启发的 Doug Cutting 等人开始尝试实现 MapReduce 计算框架，并将它与 NDFS（Nutch Distributed File System）结合，用以支持 Nutch 引擎的主要算法。由于 NDFS 和 MapReduce 在 Nutch 引擎中有着良好的应用，所以它们于 2006 年 2 月被分离出来，成为一套完整而独立的软件，并命名为 Hadoop。到了 2008 年年初，Hadoop 已成为 Apache 的顶级项目，包含众多子项目。它被应用到包括 Yahoo! 在内的很多互联网公司。现在的 Hadoop1.0.1 版本已经发展成为包含 HDFS、MapReduce 子项目，与 Pig、ZooKeeper、Hive、HBase 等项目相关的大型应用工程。

1.1.3 Hadoop 的功能与作用

我们为什么需要 Hadoop 呢？众所周知，现代社会的信息增长速度很快，这些信息中又积累着大量数据，其中包括个人数据和工业数据。预计到 2020 年，每年产生的数字信息中将会有超过 1/3 的内容驻留在云平台中或借助云平台处理。我们需要对这些数据进行分析处理，以获取更多有价值的信息。那么我们如何高效地存储管理这些数据、如何分析这些数据呢？这时可以选用 Hadoop 系统。在处理这类问题时，它采用分布式存储方式来提高读写速度和扩大存储容量；采用 MapReduce 整合分布式文件系统上的数据，保证高速分析处理数据；与此同时还采用存储冗余数据来保证数据的安全性。

Hadoop 中的 HDFS 具有高容错性，并且是基于 Java 语言开发的，这使得 Hadoop 可以部署在低廉的计算机集群中，同时不限于某个操作系统。Hadoop 中 HDFS 的数据管理能力、

MapReduce 处理任务时的高效率以及它的开源特性，使其在同类分布式系统中大放异彩，并在众多行业和科研领域中被广泛应用。

1.1.4　Hadoop 的优势

Hadoop 是一个能够让用户轻松架构和使用的分布式计算平台。用户可以轻松地在 Hadoop 上开发运行处理海量数据的应用程序。它主要有以下几个优点：

- 高可靠性。Hadoop 按位存储和处理数据的能力值得人们信赖。
- 高扩展性。Hadoop 是在可用的计算机集簇间分配数据完成计算任务的，这些集簇可以方便地扩展到数以千计的节点中。
- 高效性。Hadoop 能够在节点之间动态地移动数据，以保证各个节点的动态平衡，因此其处理速度非常快。
- 高容错性。Hadoop 能够自动保存数据的多份副本，并且能够自动将失败的任务重新分配。

1.1.5　Hadoop 应用现状和发展趋势

由于 Hadoop 优势突出，基于 Hadoop 的应用已经遍地开花，尤其是在互联网领域。Yahoo! 通过集群运行 Hadoop，用以支持广告系统和 Web 搜索的研究；Facebook 借助集群运行 Hadoop 来支持其数据分析和机器学习；搜索引擎公司百度则使用 Hadoop 进行搜索日志分析和网页数据挖掘工作；淘宝的 Hadoop 系统用于存储并处理电子商务交易的相关数据；中国移动研究院基于 Hadoop 的“大云”（BigCloud）系统对数据进行分析并对外提供服务。

2008 年 2 月，作为 Hadoop 最大贡献者的 Yahoo! 构建了当时最大规模的 Hadoop 应用。他们在 2000 个节点上面执行了超过 1 万个 Hadoop 虚拟机器来处理超过 5PB 的网页内容，分析大约 1 兆个网络连接之间的网页索引资料。这些网页索引资料压缩后超过 300TB。Yahoo! 正是基于这些为用户提供了高质量的搜索服务。

Hadoop 目前已经取得了非常突出的成绩。随着互联网的发展，新的业务模式还将不断涌现，Hadoop 的应用也会从互联网领域向电信、电子商务、银行、生物制药等领域拓展。相信在未来，Hadoop 将会在更多的领域中扮演幕后英雄，为我们提供更加快捷优质的服务。

1.2　Hadoop 项目及其结构

现在 Hadoop 已经发展成为包含很多项目的集合。虽然其核心内容是 MapReduce 和 Hadoop 分布式文件系统，但与 Hadoop 相关的 Common、Avro、Chukwa、Hive、HBase 等项目也是不可或缺的。它们提供了互补性服务或在核心层上提供了更高层的服务。图 1-1 是 Hadoop 的项目结构图。

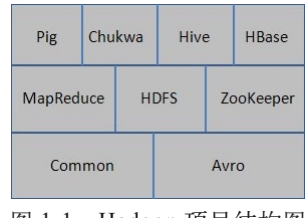

图 1-1　Hadoop 项目结构图

下面将对 Hadoop 的各个关联项目进行更详细的介绍。

1）Common：Common 是为 Hadoop 其他子项目提供支持的常用工具，它主要包括 FileSystem、RPC 和串行化库。它们为在廉价硬件上搭建云计算环境提供基本的服务，并且会为运行在该平台上的软件开发提供所需的 API。

2）Avro：Avro 是用于数据序列化的系统。它提供了丰富的数据结构类型、快速可压缩的二进制数据格式、存储持久性数据的文件集、远程调用 RPC 的功能和简单的动态语言集成功能。其中代码生成器既不需要读写文件数据，也不需要使用或实现 RPC 协议，它只是一个可选的对静态类型语言的实现。

Avro 系统依赖于模式（Schema），数据的读和写是在模式之下完成的。这样可以减少写入数据的开销，提高序列化的速度并缩减其大小；同时，也可以方便动态脚本语言的使用，因为数据连同其模式都是自描述的。

在 RPC 中，Avro 系统的客户端和服务端通过握手协议进行模式的交换，因此当客户端和服务端拥有彼此全部的模式时，不同模式下相同命名字段、丢失字段和附加字段等信息的一致性问题就得到了很好的解决。

3）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。映射（Map）、化简（Reduce）的概念和它们的主要思想都是从函数式编程语言中借鉴而来的。它极大地方便了编程人员——即使在不了解分布式并行编程的情况下，也可以将自己的程序运行在分布式系统上。MapReduce 在执行时先指定一个 Map（映射）函数，把输入键值对映射成一组新的键值对，经过一定处理后交给 Reduce，Reduce 对相同 key 下的所有 value 进行处理后再输出键值对作为最终的结果。

图 1-2 是 MapReduce 的任务处理流程图，它展示了 MapReduce 程序将输入划分到不同的 Map 上、再将 Map 的结果合并到 Reduce、然后进行处理的输出过程。详细介绍请参考本章 1.3 节。

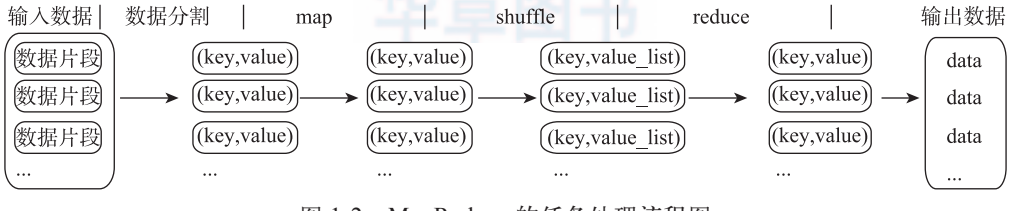

图 1-2　MapReduce 的任务处理流程图

4）HDFS：HDFS 是一个分布式文件系统。因为 HDFS 具有高容错性（fault-tolerent）的特点，所以它可以设计部署在低廉（low-cost）的硬件上。它可以通过提供高吞吐率（high throughput）来访问应用程序的数据，适合那些有着超大数据集的应用程序。HDFS 放宽了对可移植操作系统接口（POSIX，Portable Operating System Interface）的要求，这样可以实现以流的形式访问文件系统中的数据。HDFS 原本是开源的 Apache 项目 Nutch 的基础结构，最后它却成为了 Hadoop 基础架构之一。

以下几个方面是 HDFS 的设计目标：

- 检测和快速恢复硬件故障。硬件故障是计算机常见的问题。整个 HDFS 系统由数百甚至数千个存储着数据文件的服务器组成。而如此多的服务器则意味着高故障率，因此，故障的检测和快速自动恢复是 HDFS 的一个核心目标。
- 流式的数据访问。HDFS 使应用程序流式地访问它们的数据集。HDFS 被设计成适合进行批量处理，而不是用户交互式处理。所以它重视数据吞吐量，而不是数据访问的反应速度。
- 简化一致性模型。大部分的 HDFS 程序对文件的操作需要一次写入，多次读取。一个文件一旦经过创建、写入、关闭就不需要修改了。这个假设简化了数据一致性问题和高吞吐量的数据访问问题。
- 通信协议。所有的通信协议都是在 TCP/IP 协议之上的。一个客户端和明确配置了端口的名字节点（NameNode）建立连接之后，它和名字节点的协议便是客户端协议（Client Protocal）。数据节点（DataNode）和名字节点之间则用数据节点协议（DataNode Protocal）。

关于 HDFS 的具体介绍请参考本章 1.3 节。

5）Chukwa：Chukwa 是开源的数据收集系统，用于监控和分析大型分布式系统的数据。Chukwa 是在 Hadoop 的 HDFS 和 MapReduce 框架之上搭建的，它继承了 Hadoop 的可扩展性和健壮性。Chukwa 通过 HDFS 来存储数据，并依赖 MapReduce 任务处理数据。Chukwa 中也附带了灵活且强大的工具，用于显示、监视和分析数据结果，以便更好地利用所收集的数据。

6）Hive：Hive 最早是由 Facebook 设计的，是一个建立在 Hadoop 基础之上的数据仓库，它提供了一些用于对 Hadoop 文件中的数据集进行数据整理、特殊查询和分析存储的工具。Hive 提供的是一种结构化数据的机制，它支持类似于传统 RDBMS 中的 SQL 语言的查询语言，来帮助那些熟悉 SQL 的用户查询 Hadoop 中的数据，该查询语言称为 Hive QL。与此同时，传统的 MapReduce 编程人员也可以在 Mapper 或 Reducer 中通过 Hive QL 查询数据。Hive 编译器会把 Hive QL 编译成一组 MapReduce 任务，从而方便 MapReduce 编程人员进行 Hadoop 系统开发。

7）HBase：HBase 是一个分布式的、面向列的开源数据库，该技术来源于 Google 论文《Bigtable：一个结构化数据的分布式存储系统》。如同 Bigtable 利用了 Google 文件系统（Google File System）提供的分布式数据存储方式一样，HBase 在 Hadoop 之上提供了类似于 Bigtable 的能力。HBase 不同于一般的关系数据库，原因有两个：其一，HBase 是一个适合于非结构化数据存储的数据库；其二，HBase 是基于列而不是基于行的模式。HBase 和 Bigtable 使用相同的数据模型。用户将数据存储在一个表里，一个数据行拥有一个可选择的键和任意数量的列。由于 HBase 表是疏松的，用户可以为行定义各种不同的列。HBase 主要用于需要随机访问、实时读写的大数据（Big Data）。具体介绍请参考第 12 章。

8）Pig：Pig 是一个对大型数据集进行分析、评估的平台。Pig 最突出的优势是它的结构能够经受住高度并行化的检验，这个特性使得它能够处理大型的数据集。目前，Pig 的底层

由一个编译器组成，它在运行的时候会产生一些 MapReduce 程序序列，Pig 的语言层由一种叫做 Pig Latin 的正文型语言组成。有关 Pig 的具体内容请参考第 14 章。

9）ZooKeeper：ZooKeeper 是一个为分布式应用所设计的开源协调服务。它主要为用户提供同步、配置管理、分组和命名等服务，减轻分布式应用程序所承担的协调任务。ZooKeeper 的文件系统使用了我们所熟悉的目录树结构。ZooKeeper 是使用 Java 编写的，但是它支持 Java 和 C 两种编程语言。有关 ZooKeeper 的具体内容请参考第 15 章。

上面讨论的 9 个项目在本书中都有相应的章节进行详细的介绍。

1.3 Hadoop 体系结构

如上文所说，HDFS 和 MapReduce 是 Hadoop 的两大核心。而整个 Hadoop 的体系结构主要是通过 HDFS 来实现分布式存储的底层支持的，并且它会通过 MapReduce 来实现分布式并行任务处理的程序支持。

下面首先介绍 HDFS 的体系结构。HDFS 采用了主从（Master/Slave）结构模型，一个 HDFS 集群是由一个 NameNode 和若干个 DataNode 组成的。其中 NameNode 作为主服务器，管理文件系统的命名空间和客户端对文件的访问操作；集群中的 DataNode 管理存储的数据。HDFS 允许用户以文件的形式存储数据。从内部来看，文件被分成若干个数据块，而且这若干个数据块存放在一组 DataNode 上。NameNode 执行文件系统的命名空间操作，比如打开、关闭、重命名文件或目录等，它也负责数据块到具体 DataNode 的映射。DataNode 负责处理文件系统客户端的文件读写请求，并在 NameNode 的统一调度下进行数据块的创建、删除和复制工作。图 1-3 所示为 HDFS 的体系结构。

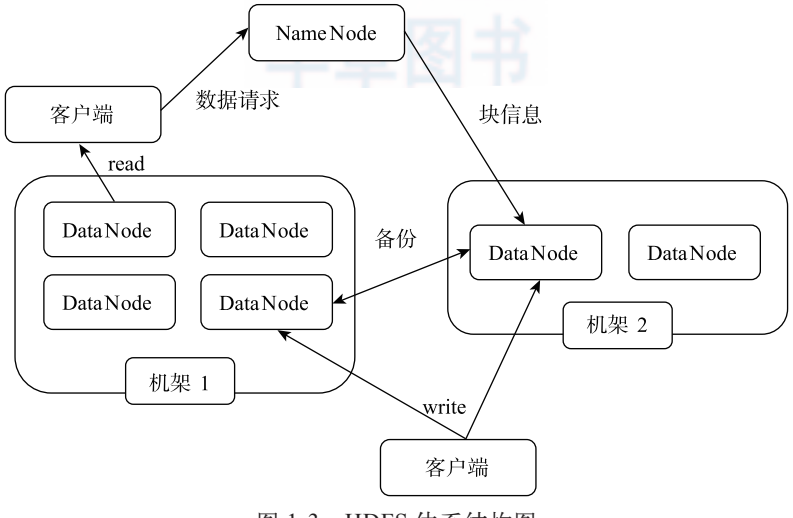

图 1-3　HDFS 体系结构图

NameNode 和 DataNode 都可以在普通商用计算机上运行。这些计算机通常运行的是

GNU/Linux 操作系统。HDFS 采用 Java 语言开发，因此任何支持 Java 的机器都可以部署 NameNode 和 DataNode。一个典型的部署场景是集群中的一台机器运行一个 NameNode 实例，其他机器分别运行一个 DataNode 实例。当然，并不排除一台机器运行多个 DataNode 实例的情况。集群中单一 NameNode 的设计大大简化了系统的架构。NameNode 是所有 HDFS 元数据的管理者，用户需要保存的数据不会经过 NameNode，而是直接流向存储数据的 DataNode。

接下来介绍 MapReduce 的体系结构。MapReduce 是一种并行编程模式，利用这种模式软件开发者可以轻松地编写出分布式并行程序。在 Hadoop 的体系结构中，MapReduce 是一个简单易用的软件框架，基于它可以将任务分发到由上千台商用机器组成的集群上，并以一种可靠容错的方式并行处理大量的数据集，实现 Hadoop 的并行任务处理功能。MapReduce 框架是由一个单独运行在主节点的 JobTracker 和运行在每个集群从节点的 TaskTracker 共同组成的。主节点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从节点上。主节点监控它们的执行情况，并且重新执行之前失败的任务；从节点仅负责由主节点指派的任务。当一个 Job 被提交时，JobTracker 接收到提交作业和其配置信息之后，就会将配置信息等分发给从节点，同时调度任务并监控 TaskTracker 的执行。

从上面的介绍可以看出，HDFS 和 MapReduce 共同组成了 Hadoop 分布式系统体系结构的核心。HDFS 在集群上实现了分布式文件系统，MapReduce 在集群上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，MapReduce 在 HDFS 的基础上实现了任务的分发、跟踪、执行等工作，并收集结果，二者相互作用，完成了 Hadoop 分布式集群的主要任务。

1.4　Hadoop 与分布式开发

我们通常所说的分布式系统其实是分布式软件系统，即支持分布式处理的软件系统。它是在通信网络互联的多处理机体系结构上执行任务的系统，包括分布式操作系统、分布式程序设计语言及其编译（解释）系统、分布式文件系统和分布式数据库系统等。Hadoop 是分布式软件系统中文件系统层的软件，它实现了分布式文件系统和部分分布式数据库系统的功能。Hadoop 中的分布式文件系统 HDFS 能够实现数据在计算机集群组成的云上高效的存储和管理，Hadoop 中的并行编程框架 MapReduce 能够让用户编写的 Hadoop 并行应用程序运行得以简化。下面简单介绍一下基于 Hadoop 进行分布式并发编程的相关知识，详细的介绍请参看后面有关 MapReduce 编程的章节。

Hadoop 上并行应用程序的开发是基于 MapReduce 编程模型的。MapReduce 编程模型的原理是：利用一个输入的 key/value 对集合来产生一个输出的 key/value 对集合。MapReduce 库的用户用两个函数来表达这个计算：Map 和 Reduce。

用户自定义的 Map 函数接收一个输入的 key/value 对，然后产生一个中间 key/value 对的集合。MapReduce 把所有具有相同 key 值的 value 集合在一起，然后传递给 Reduce 函数。

用户自定义的 Reduce 函数接收 key 和相关的 value 集合。Reduce 函数合并这些 value 值，形成一个较小的 value 集合。一般来说，每次调用 Reduce 函数只产生 0 或 1 个输出的 value 值。通常我们通过一个迭代器把中间 value 值提供给 Reduce 函数，这样就可以处理无法全部放入内存中的大量的 value 值集合了。

图 1-4 是 MapReduce 的数据流图，体现 MapReduce 处理大数据集的过程。简而言之，这个过程就是将大数据集分解为成百上千个小数据集，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合并，形成最终结果。图 1-4 也说明了 MapReduce 框架下并行程序中的两个主要函数：Map、Reduce。在这个结构中，用户需要完成的工作是根据任务编写 Map 和 Reduce 两个函数。

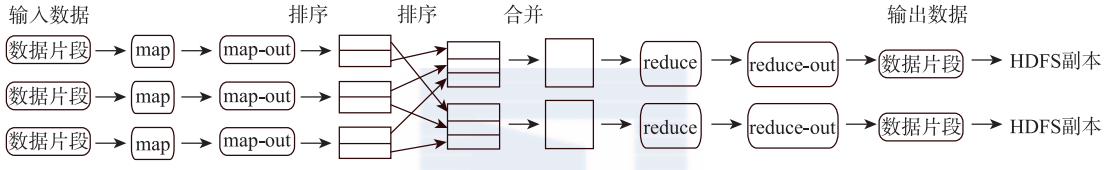

图 1-4　MapReduce 数据流图

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 1-4 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？下面将简单介绍一下其原理。

1. 数据分布存储

Hadoop 分布式文件系统（HDFS）由一个名字节点（NameNode）和多个数据节点（DataNode）组成，每个节点都是一台普通的计算机。在使用方式上 HDFS 与我们熟悉的单机文件系统非常类似，利用它可以创建目录，创建、复制、删除文件，并且可以查看文件内容等。但文件在 HDFS 底层被切割成了 Block，这些 Block 分散地存储在不同的 DataNode 上，每个 Block 还可以复制数份数据存储在不同的 DataNode 上，达到容错容灾的目的。NameNode 则是整个 HDFS 的核心，它通过维护一些数据结构来记录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，以及各个 DataNode 的状态等重要信息。

2. 分布式并行计算

Hadoop 中有一个作为主控的 JobTracker，用于调度和管理其他的 TaskTracker。JobTracker 可以运行于集群中的任意一台计算机上；TaskTracker 则负责执行任务，它必须运行于 DataNode 上，也就是说 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 出了故障，JobTracker 会将其负责的任务转交给另一个空闲的 TaskTracker 重新运行。

3. 本地计算

数据存储在哪一台计算机上，就由哪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 这类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，因此它所能够提供的计算能力近乎无限。但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。“本地计算”是一种最有效的节约网络带宽的手段，业界将此形容为“移动计算比移动数据更经济”。

4. 任务粒度

在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（默认是 64MB），这样能够保证一个小数据集是位于一台计算机上的，便于本地计算。假设有 *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，它们将并行运行，Reduce 任务的数量 *R* 则可由用户指定。

5. 数据分割（Partition）

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 任务来处理的，可以简化 Reduce 的过程。

6. 数据合并（Combine）

在数据分割之前，还可以先对中间结果进行数据合并（Combine），即将中间结果中有相同 key 的 <key, value> 对合并成一对。Combine 的过程与 Reduce 的过程类似，在很多情况下可以直接使用 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，从而降低网络流量。

7. Reduce

Map 任务的中间结果在执行完 Combine 和 Partition 之后，以文件形式存储于本地磁盘上。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 TaskTracker 上去取中间结果。注意，所有的 Map 任务产生的中间结果均按其 key 值通过同一个 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。

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* 个最终结果又可以作为另一个计算任务的输入，开始另一个并行计算任务，这也就形成了任务管道。

这里简要介绍了在并行编程方面 Hadoop 中 MapReduce 编程模型的原理、流程、程序结构和并行计算的实现，MapReduce 程序的详细流程、编程接口、程序实例等请参见后面章节。

1.5 Hadoop 计算模型——MapReduce

MapReduce 是 Google 公司的核心计算模型，它将运行于大规模集群上的复杂的并行计算过程高度地抽象为两个函数：Map 和 Reduce。Hadoop 是 Doug Cutting 受到 Google 发表的关于 MapReduce 的论文启发而开发出来的。Hadoop 中的 MapReduce 是一个使用简易的软件框架，基于它写出来的应用程序能够运行在由上千台商用机器组成的大型集群上，并以一种可靠容错的方式并行处理上 T 级别的数据集，实现了 Hadoop 在集群上的数据和任务的并行计算与处理。

一个 Map/Reduce 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。框架会先对 Map 的输出进行排序，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，Map/Reduce 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，这样可以使整个集群的网络带宽得到非常高效的利用。

Map/Reduce 框架由一个单独的 Master JobTracker 和集群节点上的 Slave TaskTracker 共同组成。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 监控它们的执行情况，并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。

在 Hadoop 上运行的作业需要指明程序的输入 / 输出位置（路径），并通过实现合适的接口或抽象类提供 Map 和 Reduce 函数。同时还需要指定作业的其他参数，构成作业配置（Job Configuration）。在 Hadoop 的 JobClient 提交作业（JAR 包 / 可执行程序等）和配置信息给 JobTracker 之后，JobTracker 会负责分发这些软件和配置信息给 slave 及调度任务，并监控它们的执行，同时提供状态和诊断信息给 JobClient。

1.6 Hadoop 数据管理

前面重点介绍了 Hadoop 及其体系结构与计算模型 MapReduce，现在开始介绍 Hadoop 的数据管理，主要包括 Hadoop 的分布式文件系统 HDFS、分布式数据库 HBase 和数据仓库工具 Hive。

1.6.1 HDFS 的数据管理

HDFS 是分布式计算的存储基石，Hadoop 分布式文件系统和其他分布式文件系统有很多类似的特性：

- 对于整个集群有单一的命名空间；
- 具有数据一致性，都适合一次写入多次读取的模型，客户端在文件没有被成功创建之前是无法看到文件存在的；

- 文件会被分割成多个文件块，每个文件块被分配存储到数据节点上，而且会根据配置由复制文件块来保证数据的安全性。

通过前面的介绍和图 1-3 可以看出，HDFS 通过三个重要的角色来进行文件系统的管理：NameNode、DataNode 和 Client。NameNode 可以看做是分布式文件系统中的管理者，主要负责管理文件系统的命名空间、集群配置信息和存储块的复制等。NameNode 会将文件系统的 Metadata 存储在内存中，这些信息主要包括文件信息、每一个文件对应的文件块的信息和每一个文件块在 DataNode 中的信息等。DataNode 是文件存储的基本单元，它将文件块（Block）存储在本地文件系统中，保存了所有 Block 的 Metadata，同时周期性地将所有存在的 Block 信息发送给 NameNode。Client 就是需要获取分布式文件系统文件的应用程序。接下来通过三个具体的操作来说明 HDFS 对数据的管理。

（1）文件写入

1）Client 向 NameNode 发起文件写入的请求。

2）NameNode 根据文件大小和文件块配置情况，返回给 Client 所管理的 DataNode 的信息。

3）Client 将文件划分为多个 Block，根据 DataNode 的地址信息，按顺序将其写入到每一个 DataNode 块中。

（2）文件读取

1）Client 向 NameNode 发起文件读取的请求。

2）NameNode 返回文件存储的 DataNode 信息。

3）Client 读取文件信息。

（3）文件块（Block）复制

1）NameNode 发现部分文件的 Block 不符合最小复制数这一要求或部分 DataNode 失效。

2）通知 DataNode 相互复制 Block。

3）DataNode 开始直接相互复制。

作为分布式文件系统，HDFS 在数据管理方面还有值得借鉴的几个功能：

- 文件块（Block）的放置：一个 Block 会有三份备份，一份放在 NameNode 指定的 DataNode 上，另一份放在与指定 DataNode 不在同一台机器上的 DataNode 上，最后一份放在与指定 DataNode 同一 Rack 的 DataNode 上。备份的目的是为了数据安全，采用这种配置方式主要是考虑同一 Rack 失败的情况，以及不同 Rack 之间进行数据复制会带来的性能问题。
- 心跳检测：用心跳检测 DataNode 的健康状况，如果发现问题就采取数据备份的方式来保证数据的安全性。
- 数据复制（场景为 DataNode 失败、需要平衡 DataNode 的存储利用率和平衡 DataNode 数据交互压力等情况）：使用 Hadoop 时可以用 HDFS 的 balancer 命令配置 Threshold 来平衡每一个 DataNode 的磁盘利用率。假设设置了 Threshold 为 10%，那

么执行 balancer 命令时，首先会统计所有 DataNode 的磁盘利用率的平均值，然后判断如果某一个 DataNode 的磁盘利用率超过这个平均值，那么将会把这个 DataNode 的 Block 转移到磁盘利用率低的 DataNode 上，这对于新节点的加入十分有用。

- 数据校验：采用 CRC32 做数据校验。在写入文件块的时候，除了会写入数据外还会写入校验信息，在读取的时候则需要先校验后读入。
- 单个 NameNode：如果单个 NameNode 失败，任务处理信息将会记录在本地文件系统和远端的文件系统中。
- 数据管道性的写入：当客户端要写入文件到 DataNode 上时，首先会读取一个 Block，然后将其写到第一个 DataNode 上，接着由第一个 DataNode 将其传递到备份的 DataNode 上，直到所有需要写入这个 Block 的 DataNode 都成功写入后，客户端才会开始写下一个 Block。
- 安全模式：分布式文件系统启动时会进入安全模式（系统运行期间也可以通过命令进入安全模式），当分布式文件系统处于安全模式时，文件系统中的内容不允许修改也不允许删除，直到安全模式结束。安全模式主要是为了在系统启动的时候检查各个 DataNode 上数据块的有效性，同时根据策略进行必要的复制或删除部分数据块。在实际操作过程中，如果在系统启动时修改和删除文件会出现安全模式不允许修改的错误提示，只需要等待一会儿即可。

1.6.2 HBase 的数据管理

HBase 是一个类似 Bigtable 的分布式数据库，它的大部分特性和 Bigtable 一样，是一个稀疏的、长期存储的（存在硬盘上）、多维度的排序映射表，这张表的索引是行关键字、列关键字和时间戳。表中的每个值是一个纯字符数组，数据都是字符串，没有类型。用户在表格中存储数据，每一行都有一个可排序的主键和任意多的列。由于是稀疏存储的，所以同一张表中的每一行数据都可以有截然不同的列。列名字的格式是“<family>:<label>”，它是由字符串组成的，每一张表有一个 family 集合，这个集合是固定不变的，相当于表的结构，只能通过改变表结构来改变表的 family 集合。但是 label 值相对于每一行来说都是可以改变的。

HBase 把同一个 family 中的数据存储在同一个目录下，而 HBase 的写操作是锁行的，每一行都是一个原子元素，都可以加锁。所有数据库的更新都有一个时间戳标记，每次更新都会生成一个新的版本，而 HBase 会保留一定数量的版本，这个值是可以设定的。客户端可以选择获取距离某个时间点最近的版本，或者一次获取所有版本。

以上从微观上介绍了 HBase 的一些数据管理措施。那么 HBase 作为分布式数据库在整体上从集群出发又是如何管理数据的呢？

HBase 在分布式集群上主要依靠由 HRegion、HMaster、HClient 组成的体系结构从整体上管理数据。

HBase 体系结构有三大重要组成部分：

- HBaseMaster：HBase 主服务器，与 Bigtable 的主服务器类似。

- HRegionServer：HBase 域服务器，与 Bigtable 的 Tablet 服务器类似。
- HBase Client：HBase 客户端是由 org.apache.hadoop.HBase.client.HTable 定义的。

下面将对这三个组件进行详细的介绍。

（1）HBaseMaster

一个 HBase 只部署一台主服务器，它通过领导选举算法（Leader Election Algorithm）确保只有唯一的主服务器是活跃的，ZooKeeper 保存主服务器的服务器地址信息。如果主服务器瘫痪，可以通过领导选举算法从备用服务器中选择新的主服务器。

主服务器承担着初始化集群的任务。当主服务器第一次启动时，会试图从 HDFS 获取根或根域目录，如果获取失败则创建根或根域目录，以及第一个元域目录。在下次启动时，主服务器就可以获取集群和集群中所有域的信息了。同时主服务器还负责集群中域的分配、域服务器运行状态的监视、表格的管理等工作。

（2）HRegionServer

HBase 域服务器的主要职责有服务于主服务器分配的域、处理客户端的读写请求、本地缓冲区回写、本地数据压缩和分割域等功能。

每个域只能由一台域服务器来提供服务。当它开始服务于某域时，它会从 HDFS 文件系统中读取该域的日志和所有存储文件，同时还会管理操作 HDFS 文件的持久性存储工作。客户端通过与主服务器通信获取域和域所在域服务器的列表信息后，就可以直接向域服务器发送域读写请求，来完成操作。

（3）HBaseClient

HBase 客户端负责查找用户域所在的域服务器地址。HBase 客户端会与 HBase 主机交换消息以查找根域的位置，这是两者之间唯一的交流。

定位根域后，客户端连接根域所在的域服务器，并扫描根域获取元域信息。元域信息中包含所需用户域的域服务器地址。客户端再连接元域所在的域服务器，扫描元域以获取所需用户域所在的域服务器地址。定位用户域后，客户端连接用户域所在的域服务器并发出读写请求。用户域的地址将在客户端被缓存，后续的请求无须重复上述过程。

综上所述，在 HBase 的体系结构中，HBase 主要由主服务器、域服务器和客户端三部分组成。主服务器作为 HBase 的中心，管理整个集群中的所有域，监控每台域服务器的运行情况等；域服务器接收来自服务器的分配域，处理客户端的域读写请求并回写映射文件等；客户端主要用来查找用户域所在的域服务器地址信息。

1.6.3　Hive 的数据管理

Hive 是建立在 Hadoop 上的数据仓库基础构架。它提供了一系列的工具，用来进行数据提取、转化、加载，这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 的查询语言，称为 Hive QL，它允许熟悉 SQL 的用户用 SQL 语言查询数据。作为一个数据仓库，Hive 的数据管理按照使用层次可以从元数据存储、数据存储和数据交换三方面来介绍。

（1）元数据存储

Hive 将元数据存储在 RDBMS 中，有三种模式可以连接到数据库：

- Single User Mode：此模式连接到一个 In-memory 的数据库 Derby，一般用于 Unit Test。
- Multi User Mode：通过网络连接到一个数据库中，这是最常用的模式。
- Remote Server Mode：用于非 Java 客户端访问元数据库，在服务器端启动一个 MetaStoreServer，客户端利用 Thrift 协议通过 MetaStoreServer 来访问元数据库。

（2）数据存储

首先，Hive 没有专门的数据存储格式，也没有为数据建立索引，用户可以非常自由地组织 Hive 中的表，只需要在创建表的时候告诉 Hive 数据中的列分隔符和行分隔符，它就可以解析数据了。

其次，Hive 中所有的数据都存储在 HDFS 中，Hive 中包含 4 种数据模型：Table、External Table、Partition 和 Bucket。

Hive 中的 Table 和数据库中的 Table 在概念上是类似的，每一个 Table 在 Hive 中都有一个相应的目录来存储数据。例如，一个表 pvs，它在 HDFS 中的路径为：/wh/pvs，其中，wh 是在 hive-site.xml 中由 ${hive.metastore.warehouse.dir} 指定的数据仓库的目录，所有的 Table 数据（不包括 External Table）都保存在这个目录中。

（3）数据交换

数据交换主要分为以下几部分，如图 1-5 所示。

- 用户接口：包括客户端、Web 界面和数据库接口。
- 元数据存储：通常存储在关系数据库中，如 MySQL、Derby 等。
- 解释器、编译器、优化器、执行器。
- Hadoop：利用 HDFS 进行存储，利用 MapReduce 进行计算。

用户接口主要有三个：客户端、数据库接口和 Web 界面，其中最常用的是客户端。Client 是 Hive 的客户端，当启动 Client 模式时，用户会想要连接 Hive Server，这时需要指出 Hive Server 所在的节点，并且在该节点启动 HiveServer。Web 界面是通过浏览器访问 Hive 的。

Hive 将元数据存储在数据库中，如 MySQL、Derby 中。Hive 中的元数据包括表的名字、表的列、表的分区、表分区的属性、表的属性（是否为外部表等）、表的数据所在目录等。

解释器、编译器、优化器完成 Hive QL 查询语句从词法分析、语法分析、编译、优化到查询计划的生成。生成的查询计划存储在 HDFS 中，并且随后由 MapReduce 调用执行。

Hive 的数据存储在 HDFS 中，大部分的查询由 MapReduce 完成（包含 * 的查询不会生成 MapRedcue 任务，比如 select * from tbl）。

以上从 Hadoop 的分布式文件系统 HDFS、分布式数据库 HBase 和数据仓库工具 Hive 入手介绍了 Hadoop 的数据管理，它们都通过自己的数据定义、体系结构实现了数据从宏观到微观的立体化管理，完成了 Hadoop 平台上大规模的数据存储和任务处理。

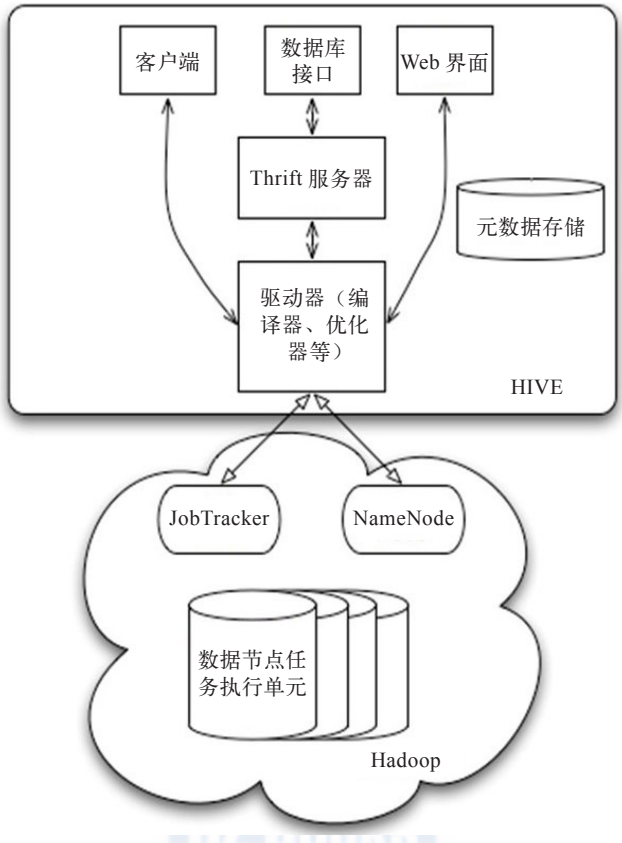

图 1-5　Hive 数据交换图

1.7　Hadoop 集群安全策略

众所周知，Hadoop 的优势在于其能够将廉价的普通 PC 组织成能够高效稳定处理事务的大型集群，企业正是利用这一特点来构架 Hadoop 集群、获取海量数据的高效处理能力的。但是，Hadoop 集群搭建起来后如何保证它安全稳定地运行呢？旧版本的 Hadoop 中没有完善的安全策略，导致 Hadoop 集群面临很多风险，例如，用户可以以任何身份访问 HDFS 或 MapReduce 集群，可以在 Hadoop 集群上运行自己的代码来冒充 Hadoop 集群的服务，任何未被授权的用户都可以访问 DataNode 节点的数据块等。经过 Hadoop 安全小组的努力，在 Hadoop 1.0.0 版本中已经加入最新的安全机制和授权机制（Simple 和 Kerberos），使 Hadoop 集群更加安全和稳定。下面从用户权限管理、HDFS 安全策略和 MapReduce 安全策略三个方面简要介绍 Hadoop 的集群安全策略。有关安全方面的基础知识如 Kerberos 认证等读者可自行查阅相关资料。

（1）用户权限管理

Hadoop 上的用户权限管理主要涉及用户分组管理，为更高层的 HDFS 访问、服务访问、Job 提交和配置 Job 等操作提供认证和控制基础。

Hadoop 上的用户和用户组名均由用户自己指定，如果用户没有指定，那么 Hadoop 会调用 Linux 的“whoami”命令获取当前 Linux 系统的用户名和用户组名作为当前用户的对应名，并将其保存在 Job 的 user.name 和 group.name 两个属性中。这样用户所提交 Job 的后续认证和授权以及集群服务的访问都将基于此用户和用户组的权限及认证信息进行。例如，在用户提交 Job 到 JobTracker 时，JobTracker 会读取保存在 Job 路径下的用户信息并进行认证，在认证成功并获取令牌之后，JobTracker 会根据用户和用户组的权限信息将 Job 提交到 Job 队列（具体细节参见本小节的 HDFS 安全策略和 MapReduce 安全策略）。

Hadoop 集群的管理员是创建和配置 Hadoop 集群的用户，它可以配置集群，使用 Kerberos 机制进行认证和授权。同时管理员可以在集群的服务（集群的服务主要包括 NameNode、DataNode、JobTracker 和 TaskTracker）授权列表中添加或更改某确定用户和用户组，系统管理员同时负责 Job 队列和队列的访问控制矩阵的创建。

（2）HDFS 安全策略

用户和 HDFS 服务之间的交互主要有两种情况：用户机和 NameNode 之间的 RPC 交互获取待通信的 DataNode 位置，客户机和 DataNode 交互传输数据块。

RPC 交互可以通过 Kerberos 或授权令牌来认证。在认证与 NameNode 的连接时，用户需要使用 Kerberos 证书来通过初试认证，获取授权令牌。授权令牌可以在后续用户 Job 与 NameNode 连接的认证中使用，而不必再次访问 Kerberos Key Server。授权令牌实际上是用户机与 NameNode 之间共享的密钥。授权令牌在不安全的网络上传输时，应给予足够的保护，防止被其他用户恶意窃取，因为获取授权令牌的任何人都可以假扮成认证用户与 NameNode 进行不安全的交互。需要注意的是，每个用户只能通过 Kerberos 认证获取唯一一个新的授权令牌。用户从 NameNode 获取授权令牌之后，需要告诉 NameNode：谁是指定的令牌更新者。指定的更新者在为用户更新令牌时应通过认证确定自己就是 NameNode。更新令牌意味着延长令牌在 NameNode 上的有效期。为了使 MapReduce Job 使用一个授权令牌，用户应将 JobTracker 指定为令牌更新者。这样同一个 Job 的所有 Task 都会使用同一个令牌。JobTracker 需要保证这一令牌在整个任务的执行过程中都是可用的，在任务结束之后，它可以选择取消令牌。

数据块的传输可以通过块访问令牌来认证，每一个块访问令牌都由 NameNode 生成，它们都是特定的。块访问令牌代表着数据访问容量，一个块访问令牌保证用户可以访问指定的数据块。块访问令牌由 NameNode 签发被用在 DataNode 上，其传输过程就是将 NameNode 上的认证信息传输到 DataNode 上。块访问令牌是基于对称加密模式生成的，NameNode 和 DataNode 共享了密钥。对于每个令牌，NameNode 基于共享密钥计算一个消息认证码（Message Authentication Code，MAC）。接下来，这个消息认证码就会作为令牌验证器成为令牌的主要组成部分。当一个 DataNode 接收到一个令牌时，它会使用自己的共享密钥重新

计算一个消息认证码，如果这个认证码同令牌中的认证码匹配，那么认证成功。

（3）MapReduce 安全策略

MapReduce 安全策略主要涉及 Job 提交、Task 和 Shuffle 三个方面。

对于 Job 提交，用户需要将 Job 配置、输入文件和输入文件的元数据等写入用户 home 文件夹下，这个文件夹只能由该用户读、写和执行。接下来用户将 home 文件夹位置和认证信息发送给 JobTracker。在执行过程中，Job 可能需要访问多个 HDFS 节点或其他服务，因此，Job 的安全凭证将以 <String key, binary value> 形式保存在一个 Map 数据结构中，在物理存储介质上将保存在 HDFS 中 JobTracker 的系统目录下，并分发给每个 TaskTracker。Job 的授权令牌将 NameNode 的 URL 作为其关键信息。为了防止授权令牌过期，JobTracker 会定期更新授权令牌。Job 结束之后所有的令牌都会失效。为了获取保存在 HDFS 上的配置信息，JobTracker 需要使用用户的授权令牌访问 HDFS，读取必需的配置信息。

任务（Task）的用户信息沿用生成 Task 的 Job 的用户信息，因为通过这个方式能保证一个用户的 Job 不会向 TaskTracker 或其他用户 Job 的 Task 发送系统信号。这种方式还保证了本地文件有权限高效地保存私有信息。在用户提交 Job 后，TaskTracker 会接收到 JobTracker 分发的 Job 安全凭证，并将其保存在本地仅对该用户可见的 Job 文件夹下。在与 TaskTracker 通信的时候，Task 会用到这个凭证。

当一个 Map 任务完成时，它的输出被发送给管理此任务的 TaskTracker。每一个 Reduce 将会与 TaskTracker 通信以获取自己的那部分输出，此时，就需要 MapReduce 框架保证其他用户不会获取这些 Map 的输出。Reduce 任务会根据 Job 凭证计算请求的 URL 和当前时间戳的消息认证码。这个消息认证码会和请求一起发到 TaskTracker，而 TaskTracker 只会在消息认证码正确并且在封装时间戳的 *N* 分钟之内提供服务。在 TaskTracker 返回数据时，为了防止数据被木马替换，应答消息的头部将会封装根据请求中的消息认证码计算而来的新消息认证码和 Job 凭证，从而保证 Reduce 能够验证应答消息是由正确的 TaskTracker 发送而来。

1.8　本章小结

本章首先介绍了 Hadoop 分布式计算平台：它是由 Apache 软件基金会开发的一个开源分布式计算平台。以 Hadoop 分布式文件系统（HDFS）和 MapReduce（Google MapReduce 的开源实现）为核心的 Hadoop 为用户提供了系统底层细节透明的分布式基础架构。由于 Hadoop 拥有可计量、成本低、高效、可信等突出特点，基于 Hadoop 的应用已经遍地开花，尤其是在互联网领域。

本章接下来介绍了 Hadoop 项目及其结构，现在 Hadoop 已经发展成为一个包含多个子项目的集合，被用于分布式计算，虽然 Hadoop 的核心是 Hadoop 分布式文件系统和 MapReduce，但 Hadoop 下的 Common、Avro、Chukwa、Hive、HBase 等子项目提供了互补性服务或在核心层之上提供了更高层的服务。紧接着，简要介绍了以 HDFS 和 MapReduce 为核心的 Hadoop 体系结构。

本章之后又从分布式系统的角度介绍了 Hadoop 是如何做到并行计算和数据管理的。分布式计算平台 Hadoop 实现了分布式文件系统和分布式数据库。Hadoop 中的分布式文件系统 HDFS 能够实现数据在电脑集群组成的云上高效的存储和管理功能，Hadoop 中的并行编程框架 MapReduce 基于 HDFS 来保证用户可以编写应用于 Hadoop 的并行应用程序。本章又介绍了 Hadoop 的数据管理，主要包括 Hadoop 的分布式文件系统 HDFS、分布式数据库 HBase 和数据仓库工具 Hive。它们都有自己完整的数据定义和体系结构，以及实现数据从宏观到微观的立体管理数据办法，这都为 Hadoop 平台的数据存储和任务处理打下了基础。

本章最后还介绍了关于 Hadoop 的一些基本的安全策略，包括用户权限管理、HDFS 安全策略和 MapReduce 安全策略，为用户的实际使用提供了参考。本章中的许多内容在本书后面的章节中会详细介绍。

第 2 章

Hadoop 的安装与配置

本章内容

- 在 Linux 上安装与配置 Hadoop
- 在 Mac OSX 上安装与配置 Hadoop
- 在 Windows 上安装与配置 Hadoop
- 安装和配置 Hadoop 集群
- 日志分析及几个小技巧
- 本章小结

Hadoop 的安装非常简单，大家可以在官网上下载到最新的几个版本，截至本书截稿时，Hadoop 的最新版本是 1.0.1，下载网址为 http://apache.etoak.com//hadoop/core/。

Hadoop 是为了在 Linux 平台上使用而开发的，但是在一些主流的操作系统如 UNIX、Windows 甚至 Mac OS X 系统上 Hadoop 也运行良好。不过，在 Windows 上运行 Hadoop 稍显复杂，首先必须安装 Cygwin 来模拟 Linux 环境，然后才能安装 Hadoop。

本章将介绍在 Linux、Mac OS X 和 Windows 系统上安装最新的 Hadoop1.0.1 版本，其中，Linux 系统是 Ubuntu 11.10，Mac OS X 系统是 10.7.3 版本，Windows 系统采用 Windows Xp sp3。这些安装步骤均由笔者成功实践过，大家可直接参照执行。

2.1 在 Linux 上安装与配置 Hadoop

在 Linux 上安装 Hadoop 之前，需要先安装两个程序：

1）JDK 1.6（或更高版本）。Hadoop 是用 Java 编写的程序，Hadoop 的编译及 MapReduce 的运行都需要使用 JDK。因此在安装 Hadoop 前，必须安装 JDK 1.6 或更高版本。

2）SSH（安全外壳协议），推荐安装 OpenSSH。Hadoop 需要通过 SSH 来启动 Slave 列表中各台主机的守护进程，因此 SSH 也是必须安装的，即使是安装伪分布式版本（因为 Hadoop 并没有区分开集群式和伪分布式）。对于伪分布式，Hadoop 会采用与集群相同的处理方式，即按次序启动文件 conf/slaves 中记载的主机上的进程，只不过在伪分布式中 Salve 为 localhost（即为自身），所以对于伪分布式 Hadoop，SSH 一样是必需的。

2.1.1 安装 JDK 1.6

下面介绍安装 JDK 1.6 的具体步骤。

（1）下载和安装 JDK 1.6

确保可以连接到互联网，从 http://www.oracle.com/technetwork/java/javase/downloads 页面下载 JDK 1.6 安装包（文件名类似 jdk-***-linux-i586.bin，不建议安装 JDK 1.7 版本，因为并不是所有软件都支持 1.7 版本）到 JDK 安装目录（本章假设 IDK 安装目录均为 /usr/lib/jvm/jdk）。

（2）手动安装 JDK 1.6

在终端下进入 JDK 安装目录，并输入命令：

```
sudo chmod u+x jdk-***-linux-i586.bin
```

修改完权限之后就可以进行安装了，在终端输入命令：

```
sudo -s ./jdk-***-linux-i586.bin
```

安装结束之后就可以开始配置环境变量了。

（3）配置环境变量

输入命令：

```
sudo gedit /etc/profile
```

输入密码，打开 profile 文件。

在文件最下面输入如下内容：

```
#set Java Environment
export JAVA_HOME=/usr/lib/jvm/jdk
export CLASSPATH=".:$JAVA_HOME/lib:$CLASSPATH"
export PATH="$JAVA_HOME/:$PATH"
```

这一步的意义是配置环境变量，使系统可以找到 JDK。

（4）验证 JDK 是否安装成功

输入命令：

```
java –version
```

会出现如下 JDK 版本信息：

```
java version "1.6.0_22"
Java(TM) SE Runtime Environment (build 1.6.0_22-b04)
Java HotSpot(TM) Client VM (build 17.1-b03, mixed mode, sharing)
```

如果出现上述 JDK 版本信息，说明当前安装的 JDK 并未设置成 Ubuntu 系统默认的 JDK，接下来还需要手动将安装的 JDK 设置成系统默认的 JDK。

（5）手动设置系统默认 JDK

在终端依次输入命令：

```
sudo update-alternatives --install /usr/bin/java java /usr/lib/jvm/jdk/bin/java 300
sudo update-alternatives --install /usr/bin/javac javac /usr/lib/jvm/jdk/bin/javac 300
sudo update-alternatives --config java
```

接下来输入 java –version 就可以看到所安装的 JDK 的版本信息了。

2.1.2　配置 SSH 免密码登录

同样以 Ubuntu 为例，假设用户名为 u：

1）确认已经连接上互联网，然后输入命令：

```
sudo apt-get install ssh
```

2）配置为可以免密码登录本机。首先查看在 u 用户下是否存在 .ssh 文件夹（注意 ssh 前面有“.”，这是一个隐藏文件夹），输入命令：

```
ls –a /home/u
```

一般来说，安装 SSH 时会自动在当前用户下创建这个隐藏文件夹，如果没有，可以手动创建一个。

接下来，输入命令（注意下面命令中不是双引号，是两个单引号）：

```
ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
```

解释一下，ssh-keygen 代表生成密钥；-t（注意区分大小写）表示指定生成的密钥类型；

dsa 是 dsa 密钥认证的意思，即密钥类型；-P 用于提供密语；-f 指定生成的密钥文件。

在 Ubuntu 中，~ 代表当前用户文件夹，此处即 /home/u。

这个命令会在 .ssh 文件夹下创建 id_dsa 及 id_dsa.pub 两个文件，这是 SSH 的一对私钥和公钥，类似于钥匙和锁，把 id_dsa.pub（公钥）追加到授权的 key 中去。

输入命令：

```
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

这条命令的功能是把公钥加到用于认证的公钥文件中，这里的 authorized_keys 是用于认证的公钥文件。

至此免密码登录本机已配置完毕。

3）验证 SSH 是否已安装成功，以及是否可以免密码登录本机。

输入命令：

```
ssh -version
```

显示结果：

```
OpenSSH_5.8p1 Debian-7ubuntu1, OpenSSL 1.0.0e 6 Sep 2011
Bad escape character 'rsion'.
```

显示 SSH 已经安装成功了。

输入命令：

```
ssh localhost
```

会有如下显示：

```
The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is 8b:c3:51:a5:2a:31:b7:74:06:9d:62:04:4f:84:f8:77.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
Linux master 2.6.31-14-generic #48-Ubuntu SMP Fri Oct 16 14:04:26 UTC 2011 i686

To access official Ubuntu documentation, please visit:
http://help.ubuntu.com/

Last login: Sat Feb 18 17:12:40 2012 from master
admin@Hadoop:~$
```

这说明已经安装成功，第一次登录时会询问是否继续链接，输入 yes 即可进入。

实际上，在 Hadoop 的安装过程中，是否免密码登录是无关紧要的，但是如果不配置免密码登录，每次启动 Hadoop 都需要输入密码以登录到每台机器的 DataNode 上，考虑到一般的 Hadoop 集群动辄拥有数百或上千台机器，因此一般来说都会配置 SSH 的免密码登录。

2.1.3 安装并运行 Hadoop

介绍 Hadoop 的安装之前，先介绍一下 Hadoop 对各个节点的角色定义。

Hadoop 分别从三个角度将主机划分为两种角色。第一，最基本的划分为 Master 和

Slave，即主人与奴隶；第二，从 HDFS 的角度，将主机划分为 NameNode 和 DataNode（在分布式文件系统中，目录的管理很重要，管理目录相当于主人，而 NameNode 就是目录管理者）；第三，从 MapReduce 的角度，将主机划分为 JobTracker 和 TaskTracker（一个 Job 经常被划分为多个 Task，从这个角度不难理解它们之间的关系）。

Hadoop 有官方发行版与 cloudera 版，其中 cloudera 版是 Hadoop 的商用版本，这里先介绍 Hadoop 官方发行版的安装方法。

Hadoop 有三种运行方式：单机模式、伪分布式与完全分布式。乍看之下，前两种方式并不能体现云计算的优势，但是它们便于程序的测试与调试，所以还是很有意义的。

你可以在以下地址获得 Hadoop 的官方发行版：http://www.apache.org/dyn/closer.cgi/Hadoop/core/。

下载 hadoop-1.0.1.tar.gz 并将其解压，本书后续都默认将 Hadoop 解压到 /home/u/ 目录下。

（1）单机模式配置方式

安装单机模式的 Hadoop 无须配置，在这种方式下，Hadoop 被认为是一个单独的 Java 进程，这种方式经常用来调试。

（2）伪分布式 Hadoop 配置

可以把伪分布式的 Hadoop 看做只有一个节点的集群，在这个集群中，这个节点既是 Master，也是 Slave；既是 NameNode，也是 DataNode；既是 JobTracker，也是 TaskTracker。

伪分布式的配置过程也很简单，只需要修改几个文件。

进入 conf 文件夹，修改配置文件。

指定 JDK 的安装位置：

```
Hadoop-env.sh:
export JAVA_HOME=/usr/lib/jvm/jdk
```

这是 Hadoop 核心的配置文件，这里配置的是 HDFS（Hadoop 的分布式文件系统）的地址及端口号。

```
conf/core-site.xml:
<configuration>
     <property>
         <name>fs.default.name</name>
         <value>hdfs://localhost:9000</value>
     </property>
</configuration>
```

以下是 Hadoop 中 HDFS 的配置，配置的备份方式默认为 3，在单机版的 Hadoop 中，需要将其改为 1。

```
conf/hdfs-site.xml:
<configuration>
     <property>
         <name>dfs.replication</name>
         <value>1</value>
```

```
    </property>
</configuration>
```

以下是 Hadoop 中 MapReduce 的配置文件，配置 JobTracker 的地址及端口。

```
conf/mapred-site.xml:
<configuration>
    <property>
        <name>mapred.job.tracker</name>
        <value>localhost:9001</value>
    </property>
</configuration>
```

接下来，在启动 Hadoop 前，需要格式化 Hadoop 的文件系统 HDFS。进入 Hadoop 文件夹，输入命令：

```
bin/Hadoop NameNode -format
```

格式化文件系统，接下来启动 Hadoop。

输入命令，启动所有进程：

```
bin/start-all.sh
```

最后，验证 Hadoop 是否安装成功。

打开浏览器，分别输入网址：

```
http://localhost:50030 (MapReduce 的 Web 页面)
http://localhost:50070 (HDFS 的 Web 页面)
```

如果都能查看，说明 Hadoop 已经安装成功。

对于 Hadoop 来说，启动所有进程是必须的，但是如果有必要，你依然可以只启动 HDFS（start-dfs.sh）或 MapReduce（start-mapred.sh）。

关于完全分布式的 Hadoop 会在 2.4 节详述。

2.2 在 Mac OSX 上安装与配置 Hadoop

由于现在越来越多的人使用 Mac Book，故笔者在本章中增加了在 Mac OS X 上安装与配置 Hadoop 的内容，供使用 Mac Book 的读者参考。

2.2.1 安装 Homebrew

Mac OS X 上的 Homebrew 是类似于 Ubuntu 下 apt 的一种软件包管理器，利用它可以自动下载和安装软件包，安装 Homebrew 之后，就可以使用 Homebrew 自动下载安装 Hadoop。安装 Homebrew 的步骤如下：

1）从 Apple 官方下载并安装内置 GCC 编译器——Xcode（现在版本为 4.2）。安装 Xcode 主要是因为一些软件包的安装依赖于本地环境，需要在本地编译源码。Xcode 的下载地址为 https://developer.apple.com/xcode/。

2）使用命令行安装 Homebrew，输入命令：

```
/usr/bin/ruby -e "$(/usr/bin/curl -fksSL https://raw.github.com/mxcl/homebrew/
master/Library/Contributions/install_homebrew.rb)"
```

这个命令会将 Homebrew 安装在 /usr/local 目录下，以保证在使用 Homebrew 安装软件包时不用使用 sudo 命令。安装完成后可以使用 brew –v 命令查看是否安装成功。

2.2.2　使用 Homebrew 安装 Hadoop

安装完 Homebrew 之后，就可以在命令行输入下面的命令来自动安装 Hadoop。自动安装的 Hadoop 在 /usr/local/Cellar/hadoop 路径下。需要注意的是，在使用 brew 安装软件时，会自动检测安装包的依赖关系，并安装有依赖关系的包，在这里 brew 就会在安装 Hadoop 时自动下载 JDK 和 SSH，并进行安装。

```
brew install hadoop
```

2.2.3　配置 SSH 和使用 Hadoop

接下来需要配置 SSH 免密码登录和启动 Hadoop。由于其步骤和内容与 Linux 的配置完全相同，故这里不再赘述。

2.3　在 Windows 上安装与配置 Hadoop

2.3.1　安装 JDK 1.6 或更高版本

相对于 Linux，JDK 在 Windows 上的安装过程更容易，你可以在 http://www.java.com/zh_CN/download/manual.jsp 下载到最新版本的 JDK。这里再次申明，Hadoop 的编译及 MapReduce 程序的运行，很多地方都需要使用 JDK 的相关工具，因此只安装 JRE 是不够的。

安装过程十分简单，运行安装程序即可，程序会自动配置环境变量（在之前的版本中还没有这项功能，新版本的 JDK 已经可以自动配置环境变量了）。

2.3.2　安装 Cygwin

Cygwin 是在 Windows 平台下模拟 UNIX 环境的一个工具，只有通过它才可以在 Windows 环境下安装 Hadoop。可以通过下面的链接下载 Cygwin：http://www.cygwin.com/。

双击运行安装程序，选择 install from internet。

根据网络状况，选择合适的源下载程序。

进入 select packages 界面，然后进入 Net，选中 OpenSSL 及 OpenSSH（如图 2-1 所示）。

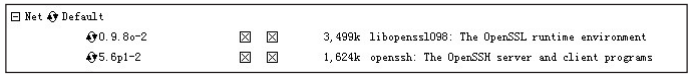

图 2-1　勾选 openssl 及 openssh

如果打算在 Eclipse 上编译 Hadoop，还必须安装 Base Category 下的 sed（如图 2-2 所示）。

图 2-2　勾选 sed

另外建议安装 Editors Category 下的 vim，以便在 Cygwin 上直接修改配置文件。

2.3.3　配置环境变量

依次右击“我的电脑”，在弹出的快捷菜单中依次单击“属性”→“高级系统设置”→“环境变量”，修改环境变量里的 path 设置，在其后添加 Cygwin 的 bin 目录。

2.3.4　安装 sshd 服务

单击桌面上的 Cygwin 图标，启动 Cygwin，执行 ssh-host-config 命令，当要求输入 Yes/No 时，选择输入 No。当显示“Have fun”时，表示 sshd 服务安装成功。

2.3.5　启动 sshd 服务

在桌面上的“我的电脑”图标上右击，在弹出的快捷菜单中单击“管理”命令，启动 CYGWIN sshd 服务，或者直接在终端下输入下面的命令启动服务：

```
net start sshd
```

2.3.6　配置 SSH 免密码登录

执行 ssh-keygen 命令生成密钥文件。按如下命令生成 authorized_keys 文件：

```
cd ~/.ssh/
cp id_rsa.pub authorized_keys
```

完成上述操作后，执行 exit 命令先退出 Cygwin 窗口，如果不执行这一步操作，后续的操作可能会遇到错误。

接下来，重新运行 Cygwin，执行 ssh localhost 命令，在第一次执行时会有提示，然后输入 yes，直接回车即可。

2.3.7　安装并运行 Hadoop

在 Windows 上安装 Hadoop 与在 Linux 上安装的过程一样，这里就不再赘述了，不过有两点需要注意：

1）在配置 conf/hadoop-evn.sh 文件中 Java 的安装路径时，如果路径之间有空格，需要将整个路径用双引号引起来。例如可以进行配置：

```
export JAVA_HOME="/cygdrive/c/Program Files/Java/jdk1.6.0_22"
```

其中 cygdrive 表示安装 cygdrive 之后系统的根目录。

另外一种办法是在 cygwin 窗口使用类似下面的命令创建文件链接，使后面的文件指向 Windows 下安装的 JDK，然后将 conf/hadoop-env.sh 中 JDK 配置为此链接文件：

```
$ ln -s /cygdrive/c/Program\ Files/Java/jdk1.6.0_22 /usr/local/jdk
```

2）在配置 conf/mapred-site.xml 文件时，应增加对 mapred.child.tmp 属性的配置，配置的值应为一个 Linux 系统的绝对路径，如果不配置，Job 在运行时就会报错。具体配置为：

```
<property>
    <name>mapred.child.tmp</name>
    <value>/home/Administrator/hadoop-1.0.1/tmp</value>
</property>
```

同样需要在 conf/core-site.xml 文件中为 hadoop.tmp.dir 属性配置一个和 mapred.child.tmp 属性相似的绝对路径。

2.4　安装和配置 Hadoop 集群

2.4.1　网络拓扑

通常来说，一个 Hadoop 的集群体系结构由两层网络拓扑组成，如图 2-3 所示。结合实际应用来看，每个机架中会有 30~40 台机器，这些机器共享一个 1GB 带宽的网络交换机。在所有的机架之上还有一个核心交换机或路由器，通常来说其网络交换能力为 1GB 或更高。可以很明显地看出，同一个机架中机器节点之间的带宽资源肯定要比不同机架中机器节点间丰富。这也是 Hadoop 随后设计数据读写分发策略要考虑的一个重要因素。

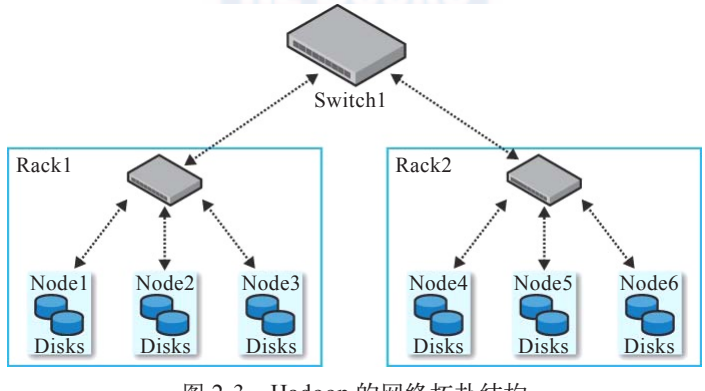

图 2-3　Hadoop 的网络拓扑结构

2.4.2　定义集群拓扑

在实际应用中，为了使 Hadoop 集群获得更高的性能，读者需要配置集群，使 Hadoop 能够感知其所在的网络拓扑结构。当然，如果集群中机器数量很少且存在于一个机架中，那么就不用做太多额外的工作；而当集群中存在多个机架时，就要使 Hadoop 清晰地知道每台

机器所在的机架。随后，在处理 MapReduce 任务时，Hadoop 就会优先选择在机架内部做数据传输，而不是在机架间传输，这样就可以更充分地使用网络带宽资源。同时，HDFS 可以更加智能地部署数据副本，并在性能和可靠性间找到最优的平衡。

在 Hadoop 中，网络的拓扑结构、机器节点及机架的网络位置定位都是通过树结构来描述的。通过树结构来确定节点间的距离，这个距离是 Hadoop 做决策判断时的参考因素。NameNode 也是通过这个距离来决定应该把数据副本放到哪里的。当一个 Map 任务到达时，它会被分配到一个 TaskTracker 上运行，JobTracker 节点则会使用网络位置来确定 Map 任务执行的机器节点。

在图 2-3 中，笔者使用树结构来描述网络拓扑结构，主要包括两个网络位置：交换机 / 机架 1 和交换机 / 机架 2。因为图 2-3 中的集群只有一个最高级别的交换机，所以此网络拓扑可简化描述为 / 机架 1 和 / 机架 2。

在配置 Hadoop 时，Hadoop 会确定节点地址和其网络位置的映射，此映射在代码中通过 Java 接口 DNSToSwitchMaping 实现，代码如下：

```
public interface DNSToSwitchMapping {
    public List<String> resolve(List<String> names);
}
```

其中参数 names 是 IP 地址的一个 List 数据，这个函数的返回值为对应网络位置的字符串列表。在 opology.node.switch.mapping.impl 中的配置参数定义了一个 DNSToSwitchMaping 接口的实现，NameNode 通过它确定完成任务的机器节点所在的网络位置。

在图 2-3 的实例中，可以将节点 1、节点 2、节点 3 映射到 / 机架 1 中，节点 4、节点 5、节点 6 映射到 / 机架 2 中。事实上在实际应用中，管理员可能不需要手动做额外的工作去配置这些映射关系，系统有一个默认的接口实现 ScriptBasedMapping。它可以运行用户自定义的一个脚本区完成映射。如果用户没有定义映射，它会将所有的机器节点映射到一个单独的网络位置中默认的机架上；如果用户定义了映射，那么这个脚本的位置由 topology.script.file.name 的属性控制。脚本必须获取一批主机的 IP 地址作为参数进行映射，同时生成一个标准的网络位置给输出。

2.4.3 建立和安装 Cluster

要建立 Hadoop 集群，首先要做的就是选择并购买机器，在机器到手之后，就要进行网络部署并安装软件了。安装和配置 Hadoop 有很多方法，这部分内容在前文已经详细讲解过（见 2.1 节、2.2 节和 2.3 节），同时还告诉了读者在实际部署时应该考虑的情况。

为了简化我们在每个机器节点上安装和维护相同软件的过程，通常会采用自动安装法，比如 Red Hat Linux 下的 Kickstart 或 Debian 的全程自动化安装。这些工具先会记录你的安装过程，以及你对选项的选择，然后根据记录来自动安装软件。同时它们会在每个进程结尾提供一个钩子执行脚本，在对那些不包含在标准安装中的最终系统进行调整和自定义时这是非常有用的。

下面我们将具体介绍如何部署和配置 Hadoop。Hadoop 为了应对不同的使用需求（不管是开发、实际应用还是研究），有着不同的运行方式，包括单机式、单机伪分布式、完全分布式等。前面已经详细介绍了在 Windows、MacOSX 和 Linux 下 Hadoop 的安装和配置。下面将对 Hadoop 的分布式配置做具体的介绍。

1. Hadoop 集群的配置

在配置伪分布式的过程中，大家也许会觉得 Hadoop 的配置很简单，但那只是最基本的配置。

Hadoop 的配置文件分为两类。

1）只读类型的默认文件：src/core/core-default.xml、src/hdfs/hdfs-default.xml、src/mapred/mapred-default.xml、conf/mapred-queues.xml。

2）定位（site-specific）设置：conf/core-site.xml、conf/hdfs-site.xml、conf/mapred-site.xml、conf/mapred-queues.xml。

除此之外，也可以通过设置 conf/Hadoop-env.sh 来为 Hadoop 的守护进程设置环境变量（在 bin/ 文件夹内）。

Hadoop 是通过 org.apache.hadoop.conf.configuration 来读取配置文件的。在 Hadoop 的设置中，Hadoop 的配置是通过资源（resource）定位的，每个资源由一系列 name/value 对以 XML 文件的形式构成，它以一个字符串命名或以 Hadoop 定义的 Path 类命名（这个类是用于定义文件系统内的文件或文件夹的）。如果是以字符串命名的，Hadoop 会通过 classpath 调用此文件。如果以 Path 类命名，那么 Hadoop 会直接在本地文件系统中搜索文件。

资源设定有两个特点，下面进行具体介绍。

1）Hadoop 允许定义最终参数（final parameters），如果任意资源声明了 final 这个值，那么之后加载的任何资源都不能改变这个值，定义最终资源的格式是这样的：

```
<property>
    <name>dfs.client.buffer.dir</name>
    <value>/tmp/Hadoop/dfs/client</value>
    <final>true</final>    // 注意这个值
</property>
```

2）Hadoop 允许参数传递，示例如下，当 tenpdir 被调用时，basedir 会作为值被调用。

```
<property>
    <name>basedir</name>
    <value>/user/${user.name}</value>
<property>

<property>
    <name>tempdir</name>
    <value>${basedir}/tmp</value>
</property>
```

前面提到，读者可以通过设置 conf/Hadoop-env.sh 为 Hadoop 的守护进程设置环境变量。一般来说，大家至少需要在这里设置在主机上安装的 JDK 的位置（JAVA_HOME），以

使 Hadoop 找到 JDK。大家也可以在这里通过 HADOOP_*_OPTS 对不同的守护进程分别进行设置，如表 2-1 所示。

表 2-1 Hadoop 的守护进程配置表

守护进程（Daemon）	配置选项（Configure Options）
NameNode	HADOOP_NAMENODE_OPTS
DataNode	HADOOP_DATANODE_OPTS
SecondaryNameNode	HADOOP_SECONDARYNAMENODE_OPTS
JobTracker	HADOOP_JOBTRACKER_OPTS
TaskTracker	HADOOP_TASKTRACKER_OPTS

例如，如果想设置 NameNode 使用 parallelGC，那么可以这样写：

```
export HADOOP_NameNode_OPTS="-XX:+UseParallelGC ${HADOOP_NAMENODE_OPTS}"
```

在这里也可以进行其他设置，比如设置 Java 的运行环境（HADOOP_OPTS），设置日志文件的存放位置（HADOOP_LOG_DIR），或者 SSH 的配置（HADOOP_SSH_OPTS），等等。

关于 conf/core-site.xml、conf/hdfs-site.xml、conf/mapred-site.xml 的配置如表 2-2 ～表 2-4 所示。

表 2-2 conf/core-site.xml 的配置

参数（Parameter）	值（Value）
fs.default.name	NameNode 的 IP 地址及端口

表 2-3 conf/hdfs-site.xml 的配置

参数（Parameter）	值（Value）
dfs.name.dir	NameNode 存储名字空间及汇报日志的位置
dfs.data.dir	DataNode 存储数据块的位置

表 2-4 conf/mapred-site.xml 的配置

参数（Parameter）	值（Value）
mapreduce.jobtracker.address	JobTracker 的 IP 地址及端口
mapreduce.jobtracker.system.dir	MapReduce 在 HDFS 上存储文件的位置，例如 /Hadoop/mapred/system/
mapreduce.cluster.local.dir	MapReduce 的缓存数据存储在文件系统中的位置
mapred.tasktracker.{map|reduce}.tasks.maximum	每台 TaskTracker 所能运行的 Map 或 Reduce 的 task 最大数量
dfs.hosts/dfs.hosts.exclude	允许或禁止的 DataNode 列表
mapreduce.jobtracker.hosts.filename/ mapreduce.jobtracker.hosts.exclude.filename	允许或禁止的 TaskTrackers 列表
mapreduce.cluster.job-authorization-enabled	布尔类型，表示 Job 存取控制列表是否支持对 Job 的观察和修改

一般而言，除了规定端口、IP 地址、文件的存储位置外，其他配置都不是必须修改的，可以根据读者的需要决定采用默认配置还是自己修改。还有一点需要注意的是，以上配置都被默认为最终参数（final parameters），这些参数都不可以在程序中再次修改。

接下来可以看一下 conf/mapred-queues.xml 的配置列表，如表 2-5 所示。

表 2-5　conf/mapred-queues.xml 的配置

标签或属性（Tag/Attribute）	值（Value）	是否可刷新
queues	配置文件的根元素	无意义
aclsEnabled	布尔类型 <queues> 标签的属性，表示存取控制列表是否支持控制 Job 的提交及所有 queue 的管理	是
queue	<queues> 的子元素，定义系统中的 queue	无意义
name	<queue> 的子元素，代表名字	否
state	<queue> 的子元素，代表 queue 的状态	是
acl-submit-job	<queue> 的子元素，定义一个能提交 Job 到该 queue 的用户或组的名单列表	是
acl-administer-job	<queue> 的子元素，定义一个能更改 Job 的优先级或能杀死已提交到该 queue 的 Job 用户或组的名单列表	是
properties	<queues> 的子元素，定义优先调度规则	无意义
property	<properties> 的子元素	无意义
key	<property> 的子元素	调度程序指定
value	<property> 的属性	调度程序指定

相信大家不难猜出表 2-5 的 conf/mapred-queues.xml 文件是用来做什么的，这个文件就是用来设置 MapReduce 系统的队列顺序的。queues 是 JobTracker 中的一个抽象概念，可以在一定程度上管理 Job，因此它为管理员提供了一种管理 Job 的方式。这种控制是常见且有效的，例如通过这种管理可以把不同的用户划分为不同的组，或分别赋予他们不同的级别，并且会优先执行高级别用户提交的 Job。

按照这个思想，很容易想到三种原则：

- 同一类用户提交的 Job 统一提交到同一个 queue 中；
- 运行时间较长的 Job 可以提交到同一个 queue 中；
- 把很快就能运行完成的 Job 划分到一个 queue 中，并且限制 queue 中 Job 的数量上限。

queue 的有效性很依赖在 JobTracker 中通过 mapreduce.jobtracker.taskscheduler 设置的调度规则（scheduler）。一些调度算法可能只需要一个 queue，不过有些调度算法可能很复杂，需要设置很多 queue。

对 queue 大部分设置的更改都不需要重新启动 MapReduce 系统就可以生效，不过也有一些更改需要重启系统才能有效，具体如表 2-5 所示。

conf/mapred-queues.xml 的文件配置与其他文件略有不同，配置格式如下：

```
<queues aclsEnabled="$aclsEnabled">
        <queue>
```

```
<name>$queue-name</name>
<state>$state</state>
<queue>
  <name>$child-queue1</name>
  <properties>
    <property key="$key" value="$value"/>
      ...
  </properties>
  <queue>
    <name>$grand-child-queue1</name>
    ...
  </queue>
</queue>
<queue>
  <name>$child-queue2</name>
    ...
</queue>
  ...
  ...
  ...
  <queue>
   <name>$leaf-queue</name>
   <acl-submit-job>$acls</acl-submit-job>
   <acl-administer-jobs>$acls</acl-administer-jobs>
   <properties>
     <property key="$key" value="$value"/>
     ...
   </properties>
  </queue>
 </queue>
</queues>
```

以上这些就是 Hadoop 配置的主要内容，其他关于 Hadoop 配置方面的信息，诸如内存配置等，如果有兴趣可以参阅官方的配置文档。

2. 一个具体的配置

为了方便阐述，这里只搭建一个有三台主机的小集群。

相信大家还没有忘记 Hadoop 对主机的三种定位方式，分别为 Master 和 Slave，JobTracker 和 TaskTracker，NameNode 和 DataNode。在分配 IP 地址时我们顺便规定一下角色。

下面为这三台机器分配 IP 地址及相应的角色：

```
10.37.128.2—master,namonode,jobtracker—master（主机名）
10.37.128.3—slave,dataNode,tasktracker—slave1（主机名）
10.37.128.4—slave,dataNode,tasktracker—slave2（主机名）
```

首先在三台主机上创建相同的用户（这是 Hadoop 的基本要求）：

1）在三台主机上均安装 JDK 1.6，并设置环境变量。

2）在三台主机上分别设置 /etc/hosts 及 /etc/hostname。

hosts 这个文件用于定义主机名与 IP 地址之间的对应关系。

/etc/hosts:

```
127.0.0.1 localhost
10.37.128.2 master
10.37.128.3 slave1
10.37.128.4 slave2
```

hostname 这个文件用于定义 Ubuntu 的主机名。

/etc/hostname:

```
“你的主机名”（如 master，slave1 等）
```

3）在这三台主机上安装 OpenSSH，并配置 SSH 可以免密码登录。

安装方式不再赘述，建立 ~/.ssh 文件夹，如果已存在，则无须创建。生成密钥并配置 SSH 免密码登录本机，输入命令：

```
ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

将文件复制到两台 Slave 主机相同的文件夹内，输入命令：

```
scp authorized_keys slave1:~/.ssh/
scp authorized_keys slave2:~/.ssh/
```

查看是否可以从 Master 主机免密码登录 Slave，输入命令：

```
ssh slave1
ssh slave2
```

4）配置三台主机的 Hadoop 文件，内容如下。

conf/Hadoop-env.sh:

```
export JAVA_HOME=/usr/lib/jvm/jdk
```

conf/core-site.xml:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
  <property>
     <name>fs.default.name</name>
     <value>hdfs://master:9000</value>
  </property>
<property>
     <name>hadoop.tmp.dir</name>
     <value>/tmp</value>
</property>
</configuration>
```

conf/hdfs-site.xml:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
   <name>dfs.replication</name>
   <value>2</value>
</property>
</configuration>
```

conf/mapred-site.xml:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
 <property>
    <name>mapred.job.tracker</name>
   <value>master:9001</value>
</property>
</configuration>
```

conf/masters:

```
master
```

conf/slaves:

```
slave1
slave2
```

5）启动 Hadoop。

```
bin/Hadoop NameNode -format
bin/start-all.sh
```

你可以通过以下命令或者通过 http://master:50070 及 http://master:50030 查看集群状态。

```
Hadoop dfsadmin -report
```

2.5 日志分析及几个小技巧

如果大家在安装的时候遇到问题，或者按步骤安装完成却不能运行 Hadoop，那么建议仔细查看日志信息。Hadoop 记录了详尽的日志信息，日志文件保存在 logs 文件夹内。

无论是启动还是以后会经常用到的 MapReduce 中的每一个 Job，或是 HDFS 等相关信息，Hadoop 均存有日志文件以供分析。

例如：NameNode 和 DataNode 的 namespaceID 不一致，这个错误是很多人在安装时都

会遇到的。日志信息为：

```
java.io.IOException: Incompatible namespaceIDs in /root/tmp/dfs/data:namenode
namespaceID = 1307672299; datanode namespaceID = 389959598
```

若 HDFS 一直没有启动，读者可以查询日志，并通过日志进行分析，日志提示信息显示了 NameNode 和 DataNode 的 namespaceID 不一致。

这个问题一般是由于两次或两次以上格式化 NameNode 造成的，有两种方法可以解决，第一种方法是删除 DataNode 的所有资料，第二种方法就是修改每个 DataNode 的 namespaceID（位于 /dfs/data/current/VERSION 文件中）或修改 NameNode 的 namespaceID（位于 /dfs/name/current/VERSION 文件中）。使其一致。

下面这两种方法在实际应用也可能会用到。

1）重启坏掉的 DataNode 或 JobTracker。当 Hadoop 集群的某单个节点出现问题时，一般不必重启整个系统，只须重启这个节点，它会自动连入整个集群。

在坏死的节点上输入如下命令即可：

```
bin/Hadoop-daemon.sh start datanode
bin/Hadoop-daemon.sh start jobtracker
```

2）动态加入 DataNode 或 TaskTracker。下面这条命令允许用户动态地将某个节点加入到集群中。

```
bin/Hadoop-daemon.sh --config ./conf start datanode
bin/Hadoop-daemon.sh --config ./conf start tasktracker
```

2.6　本章小结

本章主要讲解了 Hadoop 的安装和配置过程。Hadoop 的安装过程并不复杂，基本配置也简单明了，其中有几个关键点：

- Hadoop 主要是用 Java 语言写的，它无法使用一般 Linux 预装的 OpenJDK，因此在安装 Hadoop 前要先安装 JDK（版本要在 1.6 以上）；
- 作为分布式系统，Hadoop 需要通过 SSH 的方式启动处于 slave 上的程序，因此必须安装和配置 SSH。

由此可见，在安装 Hadoop 前需要安装 JDK 及 SSH。

Hadoop 在 Mac OS X 上的安装与 Linux 雷同，在 Windows 系统上的安装与在 Linux 上有一点不同，就是在 Windows 系统上需要通过 Cygwin 模拟 Linux 环境，而 SSH 的安装也需要在安装 Cygwin 时进行选择，请不要忘了这一点。

集群配置只要记住 conf/Hadoop-env.sh、conf/core-site.xml、conf/hdfs-site.xml、conf/mapred-site.xml、conf/mapred-queues.xml 这 5 个文件的作用即可，另外 Hadoop 有些配置是可以在程序中修改的，这部分内容不是本章的重点，因此没有详细说明。

第 3 章

MapReduce 计算模型

本章内容

- 为什么要用 MapReduce
- MapReduce 计算模型
- MapReduce 任务的优化
- Hadoop 流
- Hadoop Pipes
- 本章小结

2004 年，Google 发表了一篇论文，向全世界的人们介绍了 MapReduce。现在已经到处都有人在谈论 MapReduce（微软、雅虎等大公司也不例外）。在 Google 发表论文时，MapReduce 的最大成就是重写了 Google 的索引文件系统。而现在，谁也不知道它还会取得多大的成就。MapReduce 被广泛地应用于日志分析、海量数据排序、在海量数据中查找特定模式等场景中。Hadoop 根据 Google 的论文实现了 MapReduce 这个编程框架，并将源代码完全贡献了出来。本章就是要向大家介绍 MapReduce 这个流行的编程框架。

3.1　为什么要用 MapReduce

MapReduce 的流行是有理由的。它非常简单、易于实现且扩展性强。大家可以通过它轻易地编写出同时在多台主机上运行的程序，也可以使用 Ruby、Python、PHP 和 C++ 等非 Java 类语言编写 Map 或 Reduce 程序，还可以在任何安装 Hadoop 的集群中运行同样的程序，不论这个集群有多少台主机。MapReduce 适合处理海量数据，因为它会被多台主机同时处理，这样通常会有较快的速度。

下面来看一个例子。

引文分析是评价论文好坏的一个非常重要的方面，本例只对其中最简单的一部分，即论文的被引用次数进行了统计。假设有很多篇论文（百万级），且每篇论文的引文形式如下所示：

```
References
David M. Blei, Andrew Y. Ng, and Michael I. Jordan.
2003. Latent dirichlet allocation. Journal of Machine
Learning Research, 3:993-1022.
Samuel Brody and Noemie Elhadad. 2010. An unsupervised
aspect-sentiment model for online reviews. In
NAACL '10.
Jaime Carbonell and Jade Goldstein. 1998. The use of
mmr, diversity-based reranking for reordering documents
and producing summaries. In SIGIR '98, pages
335-336.
Dennis Chong and James N. Druckman. 2010. Identifying
frames in political news. In Erik P. Bucy and
R. Lance Holbert, editors, Sourcebook for Political
Communication Research: Methods, Measures, and
Analytical Techniques. Routledge.
Cindy Chung and James W. Pennebaker. 2007. The psychological
function of function words. Social Communication:
Frontiers of Social Psychology, pages 343-
359.
G¨unes Erkan and Dragomir R. Radev. 2004. Lexrank:
graph-based lexical centrality as salience in text summarization.
J. Artif. Int. Res., 22(1):457-479.
Stephan Greene and Philip Resnik. 2009. More than
words: syntactic packaging and implicit sentiment. In
NAACL '09, pages 503-511.
```

```
Aria Haghighi and Lucy Vanderwende. 2009. Exploring
content models for multi-document summarization. In
NAACL '09, pages 362-370.
Sanda Harabagiu, Andrew Hickl, and Finley Lacatusu.
2006. Negation, contrast and contradiction in text processing.
```

在单机运行时，想要完成这个统计任务，需要先切分出所有论文的名字存入一个 Hash 表中，然后遍历所有论文，查看引文信息，一一计数。因为文章数量很多，需要进行很多次内外存交换，这无疑会延长程序的执行时间。但在 MapReduce 中，这是一个 WordCount 就能解决的问题。

3.2 MapReduce 计算模型

要了解 MapReduce，首先需要了解 MapReduce 的载体是什么。在 Hadoop 中，用于执行 MapReduce 任务的机器有两个角色：一个是 JobTracker，另一个是 TaskTracker。JobTracker 是用于管理和调度工作的，TaskTracker 是用于执行工作的。一个 Hadoop 集群中只有一台 JobTracker。

3.2.1 MapReduce Job

在 Hadoop 中，每个 MapReduce 任务都被初始化为一个 Job。每个 Job 又可以分为两个阶段：Map 阶段和 Reduce 阶段。这两个阶段分别用两个函数来表示，即 Map 函数和 Reduce 函数。Map 函数接收一个 <key, value> 形式的输入，然后产生同样为 <key, value> 形式的中间输出，Hadoop 会负责将所有具有相同中间 key 值的 value 集合到一起传递给 Reduce 函数，Reduce 函数接收一个如 <key, (list of values)> 形式的输入，然后对这个 value 集合进行处理并输出结果，Reduce 的输出也是 <key, value> 形式的。

为了方便理解，分别将三个 <key, value> 对标记为 <k1, v1>、<k2, v2>、<k3, v3>，那么上面所述的过程就可以用图 3-1 来表示了。

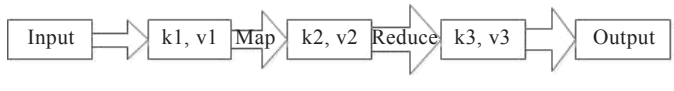

图 3-1 MapReduce 程序数据变化的基本模型

3.2.2 Hadoop 中的 Hello World 程序

上面所述的过程是 MapReduce 的核心，所有的 MapReduce 程序都具有图 3-1 所示的结构。下面我再举一个例子详述 MapReduce 的执行过程。

大家初次接触编程时学习的不论是哪种语言，看到的第一个示例程序可能都是“Hello World”。在 Hadoop 中也有一个类似于 Hello World 的程序。这就是 WordCount。本节会结合这个程序具体讲解与 MapReduce 程序有关的所有类。这个程序的内容如下：

```
package cn.edu.ruc.cloudcomputing.book.chapter03;
```

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

   public static class Map extends MapReduceBase implements Mapper<LongWritable,
   Text, Text, IntWritable> {
     private final static IntWritable one = new IntWritable(1);
     private Text word = new Text();

     public void map(LongWritable key, Text value, OutputCollector<Text,
     IntWritable> output, Reporter reporter) throws IOException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         output.collect(word, one);
       }
     }
   }

   public static class Reduce extends MapReduceBase implements Reducer<Text,
   IntWritable, Text, IntWritable> {
     public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
     IntWritable> output, Reporter reporter) throws IOException {
       int sum = 0;
       while (values.hasNext()) {
         sum += values.next().get();
       }
       output.collect(key, new IntWritable(sum));
     }
   }

   public static void main(String[] args) throws Exception {
     JobConf conf = new JobConf(WordCount.class);
     conf.setJobName("wordcount");

     conf.setOutputKeyClass(Text.class);
     conf.setOutputValueClass(IntWritable.class);

     conf.setMapperClass(Map.class);
     conf.setReducerClass(Reduce.class);

     conf.setInputFormat(TextInputFormat.class);
```

```
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        JobClient.runJob(conf);
    }
}
```

同时，为了叙述方便，设定两个输入文件，如下：

```
echo "Hello World Bye World" > file01
echo "Hello Hadoop Goodbye Hadoop" > file02
```

看到这个程序，相信很多读者会对众多的预定义类感到很迷惑。其实这些类非常简单明了。首先，WordCount 程序的代码虽多，但是执行过程却很简单，在本例中，它首先将输入文件读进来，然后交由 Map 程序处理，Map 程序将输入读入后切出其中的单词，并标记它的数目为 1，形成 <word , 1> 的形式，然后交由 Reduce 处理，Reduce 将相同 key 值（也就是 word）的 value 值收集起来，形成 <word, list of 1> 的形式，之后将这些 1 值加起来，即为单词的个数，最后将这个 <key, value> 对以 TextOutputFormat 的形式输出到 HDFS 中。

针对这个数据流动过程，我挑出了如下几句代码来表述它的执行过程：

```
JobConf conf = new JobConf(MyMapre.class);
conf.setJobName("wordcount");

conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);

conf.setMapperClass(Map.class);
conf.setReducerClass(Reduce.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
```

首先讲解一下 Job 的初始化过程。Main 函数调用 Jobconf 类来对 MapReduce Job 进行初始化，然后调用 setJobName() 方法命名这个 Job。对 Job 进行合理的命名有助于更快地找到 Job，以便在 JobTracker 和 TaskTracker 的页面中对其进行监视。接着就会调用 setInputPath() 和 setOutputPath() 设置输入输出路径。下面会结合 WordCount 程序重点讲解 Inputformat()、OutputFormat()、Map()、Reduce() 这 4 种方法。

1. InputFormat() 和 InputSplit

InputSplit 是 Hadoop 中用来把输入数据传送给每个单独的 Map，InputSplit 存储的并非数据本身，而是一个分片长度和一个记录数据位置的数组。生成 InputSplit 的方法可以通过 Inputformat() 来设置。当数据传送给 Map 时，Map 会将输入分片传送到 InputFormat() 上，InputFormat() 则调用 getRecordReader() 方法生成 RecordReader，RecordReader 再通过 creatKey()、creatValue() 方法创建可供 Map 处理的 <key, value> 对，即 <k1, v1>。简而言之，

InputFormat() 方法是用来生成可供 Map 处理的 <key, value> 对的。

Hadoop 预定义了多种方法将不同类型的输入数据转化为 Map 能够处理的 <key, value> 对，它们都继承自 InputFormat，分别是：

- BaileyBorweinPlouffe.BbpInputFormat
- ComposableInputFormat
- CompositeInputFormat
- DBInputFormat
- DistSum.Machine.AbstractInputFormat
- FileInputFormat

其中，FileInputFormat 又有多个子类，分别为：

- CombineFileInputFormat
- KeyValueTextInputFormat
- NLineInputFormat
- SequenceFileInputFormat
- TeraInputFormat
- TextInputFormat

其中，TextInputFormat 是 Hadoop 默认的输入方法，在 TextInputFormat 中，每个文件（或其一部分）都会单独作为 Map 的输入，而这是继承自 FileInputFormat 的。之后，每行数据都会生成一条记录，每条记录则表示成 <key, value> 形式：

- key 值是每个数据的记录在数据分片中的字节偏移量，数据类型是 LongWritable；
- value 值是每行的内容，数据类型是 Text。

也就是说，输入数据会以如下的形式被传入 Map 中：

```
file01:
0  hello world bye world
file02
0  hello hadoop bye hadoop
```

因为 file01 和 file02 都会被单独输入到一个 Map 中，因此它们的 key 值都是 0。

2. OutputFormat()

对于每一种输入格式都有一种输出格式与其对应。同样，默认的输出格式是 TextOutputFormat，这种输出方式与输入类似，会将每条记录以一行的形式存入文本文件。不过，它的键和值可以是任意形式的，因为程序内部会调用 toString() 方法将键和值转换为 String 类型再输出。最后的输出形式如下所示：

```
Bye 2
Hadoop 2
Hello 2
World 2
```

3. Map() 和 Reduce()

Map() 方法和 Reduce() 方法是本章的重点，从前面的内容知道，Map() 函数接收经过 InputFormat 处理所产生的 <k1, v1>，然后输出 <k2, v2>。WordCount 的 Map() 函数如下：

```
public class MyMapre {
   public static class Map extends MapReduceBase implements Mapper<LongWritable,
   Text, Text, IntWritable> {
     private final static IntWritable one = new IntWritable(1);
     private Text word = new Text();

     public void map(LongWritable key, Text value,
OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
         output.collect(word, one);
     }
   }
}
```

Map() 函数继承自 MapReduceBase，并且它实现了 Mapper 接口，此接口是一个范型类型，它有 4 种形式的参数，分别用来指定 Map() 的输入 key 值类型、输入 value 值类型、输出 key 值类型和输出 value 值类型。在本例中，因为使用的是 TextInputFormat，它的输出 key 值是 LongWritable 类型，输出 value 值是 Text 类型，所以 Map() 的输入类型即为 <LongWritable, Text>。如前面的内容所述，在本例中需要输出 <word, 1> 这样的形式，因此输出的 key 值类型是 Text，输出的 value 值类型是 IntWritable。

实现此接口类还需要实现 Map() 方法，Map() 方法会负责具体对输入进行操作，在本例中，Map() 方法对输入的行以空格为单位进行切分，然后使用 OutputCollect 收集输出的 <word, 1>，即 <k2, v2>。

下面来看 Reduce() 函数：

```
public static class Reduce extends MapReduceBase implements Reducer<Text,
IntWritable, Text, IntWritable> {
   public void reduce(Text key, Iterator<IntWritable> values,
   OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
    sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
    }
}
```

与 Map() 类似，Reduce() 函数也继承自 MapReduceBase，需要实现 Reducer 接口。Reduce() 函数以 Map() 的输出作为输入，因此 Reduce() 的输入类型是 <Text, IneWritable>。

而 Reduce() 的输出是单词和它的数目，因此，它的输出类型是 <Text, IntWritable>。Reduce() 函数也要实现 Reduce() 方法，在此方法中，Reduce() 函数将输入的 key 值作为输出的 key 值，然后将获得的多个 value 值加起来，作为输出的 value 值。

4. 运行 MapReduce 程序

读者可以在 Eclipse 里运行 MapReduce 程序，也可以在命令行中运行 MapReduce 程序，但是在实际应用中，还是推荐到命令行中运行程序。按照第 2 章介绍的步骤，首先安装 Hadoop，然后输入编译打包生成的 JAR 程序，如下所示（以 Hadoop-0.20.2 为例，安装路径是 ~/hadoop）：

```
mkdir FirstJar
javac -classpath ~/hadoop/hadoop-0.20.2-core.jar -d  FirstJar
WordCount.java
jar -cvf wordcount.jar -C FirstJar/ .
```

首先建立 FirstJar，然后编译文件生成 .class，存放到文件夹 FirstJar 中，并将 FirstJar 中的文件打包生成 wordcount.jar 文件。

接着上传输入文件（输入文件是 file01，file02，存放在 ~/input）：

```
~/hadoop/bin/hadoop dfs -mkdir input
~/hadoop/bin/hadoop dfs -put ~/input/file0* input
```

在此上传过程中，先建立文件夹 input，然后上传文件 file01、file02 到 input 中。

最后运行生成的 JAR 文件，为了叙述方便，先将生成的 JAR 文件放入 Hadoop 的安装文件夹中（HADOOP_HOME），然后运行如下命令。

```
~/hadoop/bin/hadoop jar wordcount.jar WordCount input output
11/01/21 20:02:38 WARN mapred.JobClient: Use GenericOptionsParser for parsing the
arguments. Applications should implement Tool for the same.
11/01/21 20:02:38 INFO mapred.FileInputFormat: Total input paths to process : 2
11/01/21 20:02:38 INFO mapred.JobClient: Running job: job_201101111819_0002
11/01/21 20:02:39 INFO mapred.JobClient: map 0% reduce 0%
11/01/21 20:02:49 INFO mapred.JobClient: map 100% reduce 0%
11/01/21 20:03:01 INFO mapred.JobClient: map 100% reduce 100%
11/01/21 20:03:03 INFO mapred.JobClient: Job complete: job_201101111819_0002
11/01/21 20:03:03 INFO mapred.JobClient: Counters: 18
11/01/21 20:03:03 INFO mapred.JobClient: Job Counters
11/01/21 20:03:03 INFO mapred.JobClient: Launched reduce tasks=1
11/01/21 20:03:03 INFO mapred.JobClient: Launched map tasks=2
11/01/21 20:03:03 INFO mapred.JobClient: Data-local map tasks=2
11/01/21 20:03:03 INFO mapred.JobClient: FileSystemCounters
11/01/21 20:03:03 INFO mapred.JobClient: FILE_BYTES_READ=100
11/01/21 20:03:03 INFO mapred.JobClient: HDFS_BYTES_READ=46
11/01/21 20:03:03 INFO mapred.JobClient: FILE_BYTES_WRITTEN=270
11/01/21 20:03:03 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=31
11/01/21 20:03:03 INFO mapred.JobClient: Map-Reduce Framework
11/01/21 20:03:04 INFO mapred.JobClient: Reduce input groups=4
11/01/21 20:03:04 INFO mapred.JobClient: Combine output records=0
```

```
11/01/21 20:03:04 INFO mapred.JobClient: Map input records=2
11/01/21 20:03:04 INFO mapred.JobClient: Reduce shuffle bytes=106
11/01/21 20:03:04 INFO mapred.JobClient: Reduce output records=4
11/01/21 20:03:04 INFO mapred.JobClient: Spilled Records=16
11/01/21 20:03:04 INFO mapred.JobClient: Map output bytes=78
11/01/21 20:03:04 INFO mapred.JobClient: Map input bytes=46
11/01/21 20:03:04 INFO mapred.JobClient: Combine input records=0
11/01/21 20:03:04 INFO mapred.JobClient: Map output records=8
11/01/21 20:03:04 INFO mapred.JobClient: Reduce input records=8
```

Hadoop 命令（注意不是 Hadoop 本身）会启动一个 JVM 来运行这个 MapReduce 程序，并自动获取 Hadoop 的配置，同时把类的路径（及其依赖关系）加入到 Hadoop 的库中。以上就是 Hadoop Job 的运行记录，从这里面可以看到，这个 Job 被赋予了一个 ID 号：job_201101111819_0002，而且得知输入文件有两个（Total input paths to process：2），同时还可以了解 Map 的输入输出记录（record 数及字节数），以及 Reduce 的输入输出记录。比如说，在本例中，Map 的 task 数量是 2 个，Reduce 的 Task 数量是一个；Map 的输入 record 数是 2 个，输出 record 数是 8 个等。

可以通过命令查看输出文件输出文件为：

```
bye 2
hadoop 2
hello 2
world 2
```

5. 新的 API

从 0.20.2 版本开始，Hadoop 提供了一个新的 API。新的 API 是在 org.apache.hadoop.mapreduce 中的，旧版的 API 则在 org.apache.hadoop.mapred 中。新的 API 不兼容旧的 API，WordCount 程序用新的 API 重写如下：

```
package cn.ruc.edu.cloudcomputing.book.chaptero3;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;
import org.apache.hadoop.util.*;

   public class WordCount extends Configured implements Tool {
      public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
              private final static IntWritable one = new IntWritable(1);
              private Text word = new Text();
              public void map(LongWritable key, Text value, Context context)
              throws IOException, InterruptedException {
```

```
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
              word.set(tokenizer.nextToken());
              context.write(word, one);
            }
          }
        }

    public static class Reduce extends Reducer<Text, IntWritable, Text,
    IntWritable> {
       public void reduce(Text key, Iterable<IntWritable> values, Context context)
       throws IOException, InterruptedException {
        int sum = 0;
         for (IntWritable val : values) {
           sum += val.get();
         }
         context.write(key, new IntWritable(sum));
    }
}

public int run(String [] args) throws Exception {
     Job job = new Job(getConf());
     job.setJarByClass(WordCount.class);
     job.setJobName("wordcount");

     job.setOutputKeyClass(Text.class);
     job.setOutputValueClass(IntWritable.class);

     job.setMapperClass(Map.class);
     job.setReducerClass(Reduce.class);

     job.setInputFormatClass(TextInputFormat.class);
     job.setOutputFormatClass(TextOutputFormat.class);

     FileInputFormat.setInputPaths(job, new Path(args[0]));
     FileOutputFormat.setOutputPath(job, new Path(args[1]));

     boolean success = job.waitForCompletion(true);
     return success ? 0 : 1;
}

    public static void main(String[] args) throws Exception {
         int ret = ToolRunner.run(new WordCount(), args);
         System.exit(ret);
    }
}
```

从这个程序可以看到新旧 API 的几个区别：

- 在新的 API 中，Mapper 与 Reducer 已经不是接口而是抽象类。而且 Map 函数与

Reduce 函数也已经不再实现 Mapper 和 Reducer 接口，而是继承 Mapper 和 Reducer 抽象类。这样做更容易扩展，因为添加方法到抽象类中更容易。

- 新的 API 中更广泛地使用了 context 对象，并使用 MapContext 进行 MapReduce 间的通信，MapContext 同时充当 OutputCollector 和 Reporter 的角色。
- Job 的配置统一由 Configurartion 来完成，而不必额外地使用 JobConf 对守护进程进行配置。
- 由 Job 类来负责 Job 的控制，而不是 JobClient，JobClient 在新的 API 中已经被删除。

这些区别，都可以在以上的程序中看出。

此外，新的 API 同时支持“推”和“拉”式的迭代方式。在以往的操作中，<key, value> 对是被推入到 Map 中的，但是在新的 API 中，允许程序将数据拉入 Map 中，Reduce 也一样。这样做更加方便程序分批处理数据。

3.2.3 MapReduce 的数据流和控制流

前面已经提到了 MapReduce 的数据流和控制流的关系，本节将结合 WordCount 实例具体解释它们的含义。图 3-2 是上例中 WordCount 程序的执行流程。

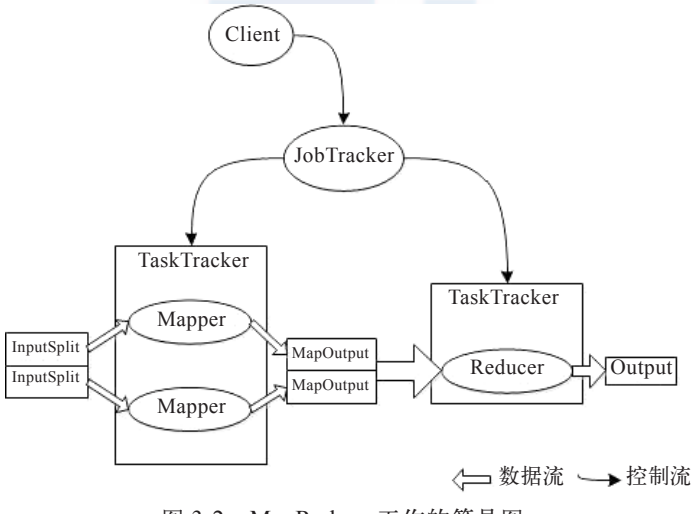

图 3-2 MapReduce 工作的简易图

由前面的内容知道，负责控制及调度 MapReduce 的 Job 的是 JobTracker，负责运行 MapReduce 的 Job 的是 TaskTracker。当然，MapReduce 在运行时是分成 Map Task 和 Reduce Task 来处理的，而不是完整的 Job。简单的控制流大概是这样的：JobTracker 调度任务给 TaskTracker，TaskTracker 执行任务时，会返回进度报告。JobTracker 则会记录进度的进行状况，如果某个 TaskTracker 上的任务执行失败，那么 JobTracker 会把这个任务分配给另一台 TaskTracker，直到任务执行完成。

这里更详细地解释一下数据流。上例中有两个 Map 任务及一个 Reduce 任务。数据首先

按照 TextInputFormat 形式被处理成两个 InputSplit，然后输入到两个 Map 中，Map 程序会读取 InputSplit 指定位置的数据，然后按照设定的方式处理该数据，最后写入到本地磁盘中。注意，这里并不是写到 HDFS 上，这应该很好理解，因为 Map 的输出在 Job 完成后即可删除了，因此不需要存储到 HDFS 上，虽然存储到 HDFS 上会更安全，但是因为网络传输会降低 MapReduce 任务的执行效率，因此 Map 的输出文件是写在本地磁盘上的。如果 Map 程序在没来得及将数据传送给 Reduce 时就崩溃了（程序出错或机器崩溃），那么 JobTracker 只需要另选一台机器重新执行这个 Task 就可以了。

Reduce 会读取 Map 的输出数据，合并 value，然后将它们输出到 HDFS 上。Reduce 的输出会占用很多的网络带宽，不过这与上传数据一样是不可避免的。如果大家还是不能很好地理解数据流的话，下面有一个更具体的图（WordCount 执行时的数据流），如图 3-3 所示。

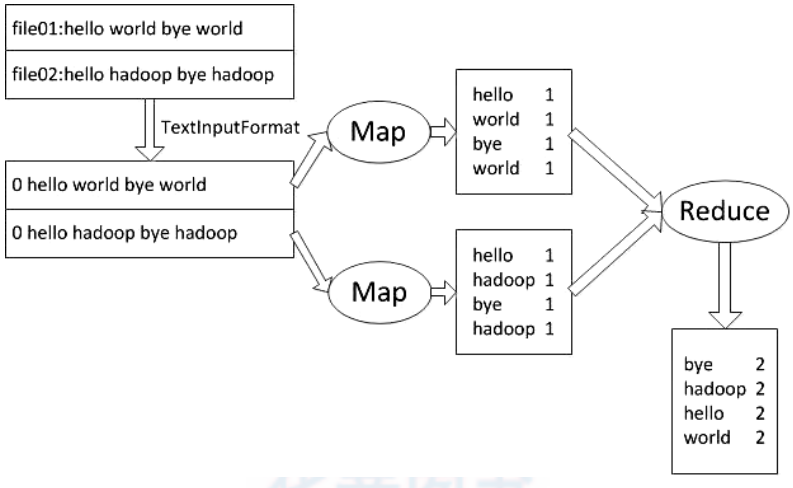

图 3-3　WordCount 数据流程图

相信看到图 3-3，大家就会对 MapReduce 的执行过程有更深刻的了解了。

除此之外，还有两种情况需要注意：

1）MapReduce 在执行过程中往往不止一个 Reduce Task，Reduce Task 的数量是可以程序指定的。当存在多个 Reduce Task 时，每个 Reduce 会搜集一个或多个 key 值。需要注意的是，当出现多个 Reduce Task 时，每个 Reduce Task 都会生成一个输出文件。

2）另外，没有 Reduce 任务的时候，系统会直接将 Map 的输出结果作为最终结果，同时 Map Task 的数量可以看做是 Reduce Task 的数量，即有多少个 Map Task 就有多少个输出文件。

3.3　MapReduce 任务的优化

相信每个程序员在编程时都会问自己两个问题“我如何完成这个任务”，以及“怎么能让程序运行得更快”。同样，MapReduce 计算模型的多次优化也是为了更好地解答这两个问题。

MapReduce 计算模型的优化涉及了方方面面的内容，但是主要集中在两个方面：一是计算性能方面的优化；二是 I/O 操作方面的优化。这其中，又包含六个方面的内容。

1. 任务调度

任务调度是 Hadoop 中非常重要的一环，这个优化又涉及两个方面的内容。计算方面：Hadoop 总会优先将任务分配给空闲的机器，使所有的任务能公平地分享系统资源。I/O 方面：Hadoop 会尽量将 Map 任务分配给 InputSplit 所在的机器，以减少网络 I/O 的消耗。

2. 数据预处理与 InputSplit 的大小

MapReduce 任务擅长处理少量的大数据，而在处理大量的小数据时，MapReduce 的性能就会逊色很多。因此在提交 MapReduce 任务前可以先对数据进行一次预处理，将数据合并以提高 MapReduce 任务的执行效率，这个办法往往很有效。如果这还不行，可以参考 Map 任务的运行时间，当一个 Map 任务只需要运行几秒就可以结束时，就需要考虑是否应该给它分配更多的数据。通常而言，一个 Map 任务的运行时间在一分钟左右比较合适，可以通过设置 Map 的输入数据大小来调节 Map 的运行时间。在 FileInputFormat 中（除了 CombineFileInputFormat），Hadoop 会在处理每个 Block 后将其作为一个 InputSplit，因此合理地设置 block 块大小是很重要的调节方式。除此之外，也可以通过合理地设置 Map 任务的数量来调节 Map 任务的数据输入。

3. Map 和 Reduce 任务的数量

合理地设置 Map 任务与 Reduce 任务的数量对提高 MapReduce 任务的效率是非常重要的。默认的设置往往不能很好地体现出 MapReduce 任务的需求，不过，设置它们的数量也要有一定的实践经验。

首先要定义两个概念——Map/Reduce 任务槽。Map/Reduce 任务槽就是这个集群能够同时运行的 Map/Reduce 任务的最大数量。比如，在一个具有 1200 台机器的集群中，设置每台机器最多可以同时运行 10 个 Map 任务，5 个 Reduce 任务。那么这个集群的 Map 任务槽就是 12000，Reduce 任务槽是 6000。任务槽可以帮助对任务调度进行设置。

设置 MapReduce 任务的 Map 数量主要参考的是 Map 的运行时间，设置 Reduce 任务的数量就只需要参考任务槽的设置即可。一般来说，Reduce 任务的数量应该是 Reduce 任务槽的 0.95 倍或是 1.75 倍，这是基于不同的考虑来决定的。当 Reduce 任务的数量是任务槽的 0.95 倍时，如果一个 Reduce 任务失败，Hadoop 可以很快地找到一台空闲的机器重新执行这个任务。当 Reduce 任务的数量是任务槽的 1.75 倍时，执行速度快的机器可以获得更多的 Reduce 任务，因此可以使负载更加均衡，以提高任务的处理速度。

4. Combine 函数

Combine 函数是用于本地合并数据的函数。在有些情况下，Map 函数产生的中间数据会有很多是重复的，比如在一个简单的 WordCount 程序中，因为词频是接近与一个 zipf 分布的，每个 Map 任务可能会产生成千上万个 <the, 1> 记录，若将这些记录一一传送给 Reduce 任

务是很耗时的。所以，MapReduce 框架运行用户写的 combine 函数用于本地合并，这会大大减少网络 I/O 操作的消耗。此时就可以利用 combine 函数先计算出在这个 Block 中单词 the 的个数。合理地设计 combine 函数会有效地减少网络传输的数据量，提高 MapReduce 的效率。

在 MapReduce 程序中使用 combine 很简单，只需在程序中添加如下内容：

```
job.setCombinerClass(combine.class);
```

在 WordCount 程序中，可以指定 Reduce 类为 combine 函数，具体如下：

```
job.setCombinerClass(Reduce.class);
```

5. 压缩

编写 MapReduce 程序时，可以选择对 Map 的输出和最终的输出结果进行压缩（同时可以选择压缩方式）。在一些情况下，Map 的中间输出可能会很大，对其进行压缩可以有效地减少网络上的数据传输量。对最终结果的压缩虽然会减少数据写 HDFS 的时间，但是也会对读取产生一定的影响，因此要根据实际情况来选择（第 7 章中提供了一个小实验来验证压缩的效果）。

6. 自定义 comparator

在 Hadoop 中，可以自定义数据类型以实现更复杂的目的，比如，当读者想实现 k-means 算法（一个基础的聚类算法）时可以定义 k 个整数的集合。自定义 Hadoop 数据类型时，推荐自定义 comparator 来实现数据的二进制比较，这样可以省去数据序列化和反序列化的时间，提高程序的运行效率（具体会在第 7 章中讲解）。

3.4　Hadoop 流

Hadoop 流提供了一个 API，允许用户使用任何脚本语言写 Map 函数或 Reduce 函数。Hadoop 流的关键是，它使用 UNIX 标准流作为程序与 Hadoop 之间的接口。因此，任何程序只要可以从标准输入流中读取数据并且可以写入数据到标准输出流，那么就可以通过 Hadoop 流使用其他语言编写 MapReduce 程序的 Map 函数或 Reduce 函数。

举个最简单的例子（本例的运行环境：Ubuntu，Hadoop-0.20.2）：

```
bin/hadoop jar contrib/streaming/hadoop-0.20.2-streaming.jar -input input -output
output -mapper /bin/cat -reducer usr/bin/wc
```

从这个例子中可以看到，Hadoop 流引入的包是 hadoop-0.20.2-streaming.jar，并且具有如下命令：

```
-input      指明输入文件路径
-output     指明输出文件路径
-mapper     指定 map 函数
-reducer    指定 reduce 函数
```

Hadoop 流的操作还有其他参数，后面会一一列出。

3.4.1 Hadoop 流的工作原理

先来看 Hadoop 流的工作原理。在上例中，Map 和 Reduce 都是 Linux 内的可执行文件，更重要的是，它们接受的都是标准输入（stdin），输出的都是标准输出（stdout）。如果大家熟悉 Linux，那么对它们一定不会陌生。执行上一节中的示例程序的过程如下所示。

程序的输入与 WordCount 程序是一样的，具体如下：

```
file01:
hello world bye world
file02
hello hadoop bye hadoop
```

输入命令：

```
bin/hadoop jar contrib/streaming/hadoop-0.20.2-streaming.jar -input input -output
output -mapper /bin/cat -reducer /usr/bin/wc
```

显示：

```
packageJobJar: [/root/tmp/hadoop-unjar7103575849190765740/] [] /tmp/
streamjob2314757737747407133.jar tmpDir=null
11/01/23 02:07:36 INFO mapred.FileInputFormat: Total input paths to process : 2
11/01/23 02:07:37 INFO streaming.StreamJob: getLocalDirs(): [/root/tmp/mapred/local]
11/01/23 02:07:37 INFO streaming.StreamJob: Running job: job_201101111819_0020
11/01/23 02:07:37 INFO streaming.StreamJob: To kill this job, run:
11/01/23 02:07:37 INFO streaming.StreamJob: /root/hadoop/bin/hadoop job  -Dmapred.
job.tracker=localhost:9001 -kill job_201101111819_0020
11/01/23 02:07:37 INFO streaming.StreamJob: Tracking URL: http://localhost:50030/
jobdetails.jsp?jobid=job_201101111819_0020
11/01/23 02:07:38 INFO streaming.StreamJob:  map 0%  reduce 0%
11/01/23 02:07:47 INFO streaming.StreamJob:  map 100%  reduce 0%
11/01/23 02:07:59 INFO streaming.StreamJob:  map 100%  reduce 100%
11/01/23 02:08:02 INFO streaming.StreamJob: Job complete: job_201101111819_0020
11/01/23 02:08:02 INFO streaming.StreamJob: Output: output
```

程序的输出是：

```
2       8       46
```

wc 命令用来统计文件中的行数、单词数与字节数，可以看到，这个结果是正确的。

Hadoop 流的工作原理并不复杂，其中 Map 的工作原理如图 3-4 所示（Reduce 与其相同）。

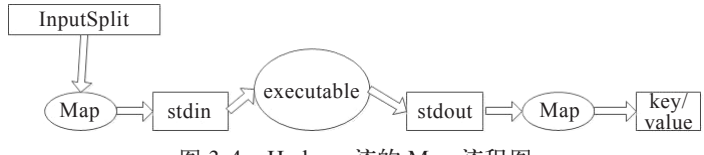

图 3-4　Hadoop 流的 Map 流程图

当一个可执行文件作为 Mapper 时，每一个 Map 任务会以一个独立的进程启动这个可执行文件，然后在 Map 任务运行时，会把输入切分成行提供给可执行文件，并作为它的标准输

入（stdin）内容。当可执行文件运行出结果时，Map 从标准输出（stdout）中收集数据，并将其转化为 <key, value> 对，作为 Map 的输出。

Reduce 与 Map 相同，如果可执行文件做 Reducer 时，Reduce 任务会启动这个可执行文件，并且将 <key, value> 对转化为行作为这个可执行文件的标准输入（stdin）。然后 Reduce 会收集这个可执行文件的标准输出（stdout）的内容。并把每一行转化为 <key, value> 对，作为 Reduce 的输出。

Map 与 Reduce 将输出转化为 <key , value> 对的默认方法是：将每行的第一个 tab 符号（制表符）之前的内容作为 key，之后的内容作为 value。如果没有 tab 符号，那么这一行的所有内容会作为 key，而 value 值为 null。当然这是可以更改的。

值得一提的是，可以使用 Java 类作为 Map，而用一个可执行程序作为 Reduce；或使用 Java 类作为 Reduce，而用可执行程序作为 Map。例如：

```
/bin/hadoop  jar contrib/streaming/hadoop-0.20.2-streaming.jar
 -input myInputDirs -output myOutputDir -mapper
org.apache.hadoop.mapred.lib.IdentityMapper -reducer /bin/wc
```

3.4.2　Hadoop 流的命令

Hadoop 流提供自己的流命令选项及一个通用的命令选项，用于设置 Hadoop 流任务。首先介绍一下流命令。

1. Hadoop 流命令选项

Hadoop 流命令具体内容如表 3-1 所示。

表 3-1　Hadoop 流命令

参　数	可选 / 必选	参　数	可选 / 必选
-input	必选	-cmdenv	可选
-output	必选	-inputreader	可选
-mapper	必选	-verbose	可选
-reducer	必选	-lazyOutput	可选
-file	可选	-numReduce tasks	可选
-inputformat	可选	-mapdebug	可选
-outputformat	可选	-reducedebug	可选
-partitioner	可选	-io	可选
-combiner	可选		

表 3-1 所示的 Hadoop 流命令中，必选的 4 个很好理解，分别用于指定输入 / 输出文件的位置及 Map/Reduce 函数。在其他的可选命令中，这里我们只解释常用的几个。

- -file

-file 指令用于将文件加入到 Hadoop 的 Job 中。上面的例子中，cat 和 wc 都是 Linux

系统中的命令，而在 Hadoop 流的使用中，往往需要使用自己写的文件（作为 Map 函数或 Reduce 函数）。一般而言，这些文件是 Hadoop 集群中的机器上没有的，这时就需要使用 Hadoop 流中的 -file 命令将这个可执行文件加入到 Hadoop 的 Job 中。

- -combiner

这个命令用来加入 combiner 程序。

- -inputformat 和 -outputformat

这两个命令用来设置输入输出文件的处理方法，这两个命令后面的参数必须是 Java 类。

2. Hadoop 流通用的命令选项

Hadoop 流的通用命令用来配置 Hadoop 流的 Job。需要注意的是，如果使用这部分配置，就必须将其置于流命令配置之前，否则命令会失败。这里简要列出命令列表（如表 3-2 所示），供大家参考。

表 3-2 Hadoop 流的 Job 设置命令

参　数	可选 / 必选	参　数	可选 / 必选
-conf	可选	-files	可选
-D	可选	-libjars	可选
-fs	可选	-archives	可选
-jt	可选		

3.4.3 两个例子

从上面的内容可以知道，Hadoop 流的 API 是一个扩展性非常强的框架，它与程序相连的部分只有数据，因此可以接受任何适用于 UNIX 标准输入 / 输出的脚本语言，比如 Bash、PHP、Ruby、Python 等。

下面举两个非常简单的例子来进一步说明它的特性。

1. Bash

MapReduce 框架是一个非常适合在大规模的非结构化数据中查找数据的编程模型，grep 就是这种类型的一个例子。

在 Linux 中，grep 命令用来在一个或多个文件中查找某个字符模式（这个字符模式可以代表字符串，多用正则表达式表示）。

下面尝试在如下的数据中查找带有 Hadoop 字符串的行，如下所示。

输入文件为：

```
file01:
hello      world bye world
file02:
hello      hadoop bye hadoop
```

reduce 文件为：

```
reduce.sh:
grep hadoop
```

输入命令为：

```
bin/hadoop jar contrib/streaming/hadoop-0.20.2-streaming.jar -input input -output
output -mapper /bin/cat -reducer ~/Desktop/test/reducer.sh -file ~/Desktop/test/
reducer.sh
```

结果为：

```
hello      hadoop bye hadoop
```

显然，这个结果是正确的。

2. Python

对于 Python 来说，情况有些特殊。因为 Python 是可以编译为 JAR 包的，如果将程序编译为 JAR 包，那么就可以采用运行 JAR 包的方式来运行了。

不过，同样也可以用流的方式运行 Python 程序。请看如下代码：

```
Reduce.py
#!/usr/bin/python

import sys;

def generateLongCountToken(id):
    return "LongValueSum:" + id + "\t" + "1"
def main(argv):
    line = sys.stdin.readline();
    try:
        while line:
            line = line[:-1];
            fields = line.split("\t");
            print generateLongCountToken(fields[0]);
            line = sys.stdin.readline();
    except "end of file":
        return None
if __name__ == "__main__":
     main(sys.argv)
```

使用如下命令来运行：

```
bin/hadoop jar contrib/streaming/hadoop-0.20.2-streaming.jar -input input -output
pyoutput -mapper reduce.py -reducer aggregate -file reduce.py
```

注意其中的 aggregate 是 Hadoop 提供的一个包，它提供一个 Reduce 函数和一个 combine 函数。这个函数实现一些简单的类似求和、取最大值最小值等的功能。

3.5 Hadoop Pipes

Hadoop Pipes 提供了一个在 Hadoop 上运行 C++ 程序的方法。与流不同的是，流使用的是标准输入输出作为可执行程序与 Hadoop 相关进程间通信的工具，而 Pipes 使用的是 Sockets。先看一个示例程序 wordcount.cpp：

```
#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

const std::string WORDCOUNT = "WORDCOUNT";
const std::string INPUT_WORDS = "INPUT_WORDS";
const std::string OUTPUT_WORDS = "OUTPUT_WORDS";

class WordCountMap: public HadoopPipes::Mapper {
public:
  HadoopPipes::TaskContext::Counter* inputWords;

  WordCountMap(HadoopPipes::TaskContext& context) {
    inputWords = context.getCounter(WORDCOUNT, INPUT_WORDS);
  }

  void map(HadoopPipes::MapContext& context) {
    std::vector<std::string> words =
      HadoopUtils::splitString(context.getInputValue(), " ");
    for(unsigned int i=0; i < words.size(); ++i) {
      context.emit(words[i], "1");
    }
    context.incrementCounter(inputWords, words.size());
  }
};

class WordCountReduce: public HadoopPipes::Reducer {
public:
  HadoopPipes::TaskContext::Counter* outputWords;

  WordCountReduce(HadoopPipes::TaskContext& context) {
    outputWords = context.getCounter(WORDCOUNT, OUTPUT_WORDS);
  }

  void reduce(HadoopPipes::ReduceContext& context) {
    int sum = 0;
    while (context.nextValue()) {
      sum += HadoopUtils::toInt(context.getInputValue());
    }
    context.emit(context.getInputKey(), HadoopUtils::toString(sum));
    context.incrementCounter(outputWords, 1);
  }
};
```

```
int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(HadoopPipes::TemplateFactory<WordCountMap,
  WordCountReduce>());
}
```

这个程序连接的是一个 C++ 库，结构类似于 Java 编写的程序。如新版 API 一样，这个程序使用 context 方法读入和收集 <key, value> 对。在使用时要重写 HadoopPipes 名字空间下的 Mapper 和 Reducer 函数，并用 context.emit() 方法输出 <key, value> 对。main 函数是应用程序的入口，它调用 HadoopPipes::runTask 方法，这个方法由一个 TemplateFactory 参数来创建 Map 和 Reduce 实例，也可以重载 factory 设置 combiner()、partitioner()、record reader、record writer。

接下来，编译这个程序。这个编译命令需要用到 g++，读者可以使用 apt 自动安装这个程序。g++ 的命令格式如下所示：

```
apt-get install g++
```

然后建立文件 Makerfile，如下所示：

```
HADOOP_INSTALL=" 你的 hadoop 安装文件夹 "
PLATFORM=Linux-i386-32（如果是 AMD 的 CPU，请使用 Linux-amd64-64）

CC = g++
CPPFLAGS = -m32 -I$(HADOOP_INSTALL)/c++/$(PLATFORM)/include

wordcount: wordcount.cpp
$(CC) $(CPPFLAGS) $< -Wall -L$(HADOOP_INSTALL)/c++/$(PLATFORM)/lib -lhadooppipes
-lhadooputils -lpthread -g -O2 -o $@
注意在 $(CC) 前有一个 <tab> 符号，这个分隔符是很关键的。
```

在当前目录下建立一个 WordCount 可执行文件。

接着，上传可执行文件到 HDFS 上，这是为了 TaskTracker 能够获得这个可执行文件。这里上传到 bin 文件夹内。

```
~/hadoop/bin/hadoop fs -mkdir bin
~/hadoop/bin/hadoop dfs -put wordcount bin
```

然后，就可以运行这个 MapReduce 程序了，可以采用两种配置方式运行这个程序。一种方式是直接在命令中运行指定配置，如下所示：

```
~/hadoop/bin/hadoop pipes\
-D hadoop.pipes.java.recordreader=true\
-D hadoop.pipes.java.recordwriter=true\
-input input\
-output Coutput\
-program bin/wordcount
```

另一种方式是预先将配置写入配置文件中，如下所示：

```
<?xml version="1.0"?>
<configuration>
```

```
  <property>
    // Set the binary path on DFS
    <name>hadoop.pipes.executable</name>
    <value>bin/wordcount</value>
  </property>
  <property>
    <name>hadoop.pipes.java.recordreader</name>
    <value>true</value>
  </property>
  <property>
    <name>hadoop.pipes.java.recordwriter</name>
    <value>true</value>
  </property>
</configuration>
```

然后通过如下命令运行这个程序：

```
~/hadoop/bin/hadoop pipes -conf word.xml -input input -output output
```

将参数 hadoop.pipes.executable 和 hadoop.pipes.java.recordreader 设置为 true 表示使用 Hadoop 默认的输入输出方式（即 Java 的）。同样的，也可以设置一个 Java 语言编写的 Mapper 函数、Reducer 函数、combiner 函数和 partitioner 函数。实际上，在任何一个作业中，都可以混用 Java 类和 C++ 类。

3.6 本章小结

本章主要介绍了 MapReduce 的计算模型，其中的关键内容是一个流程和四个方法。一个流程指的是数据流程，输入数据到 <k1, v1>、<k1, v1> 到 <k2, v2>、<k2, v2> 到 <k3, v3>、<k3, v3> 到输出数据。四个方法就是这个数据转换过程中使用的方法（分别是 InputFormat、Map、Reduce、OutputFormat），以及其对应的转换过程。除此之外，还介绍了 MapReduce 编程框架的几个优化方法，以及 Hadoop 流和 Hadoop Pipes，后者是在 Hadoop 中使用脚本文件及 C++ 编写 MapReduce 程序的方法。