Part 1

第一部分

Hadoop——一种分布式编程框架

本书第一部分所介绍的是理解与使用 Hadoop 的基础。首先描述 Hadoop 集群的硬件构成及系统安装与配置，然后从高层次阐述 MapReduce 框架，并让你的第一个 MapReduce 程序运行起来。

本部分内容

- 第 1 章　Hadoop 简介
- 第 2 章　初识 Hadoop
- 第 3 章　Hadoop 组件

第1章

Hadoop简介

1

本章内容

- 编写可扩展、分布式的数据密集型程序的基础知识
- 理解Hadoop和MapReduce
- 编写和运行一个基本的MapReduce程序

今天，我们正被数据所包围。人们上载视频、用手机照相、发短信给朋友、更新Facebook、网上留言及点击广告等，这使得机器产生和保留了越来越多的数据。你甚至此时此刻可能正在自己的电脑屏幕上阅读本书的电子版，并且可以确定的是，你在书店购买本书的记录已经被存为数据。[①]

数据的指数级增长首先向谷歌、雅虎、亚马逊和微软等这些处于市场领导地位的公司提出了挑战。它们需要遍历TB级和PB级数据来发现哪些网站更受欢迎，哪些书有需求，哪种广告吸引人。现有工具正变得无力处理如此大的数据集。谷歌率先推出了MapReduce，这是个用来应对其数据处理需求的系统。这个系统引起了广泛的关注，因为许多其他的企业同样面临数据膨胀的挑战，并且不是每个人都能够为自己重新量身订制一个专有的工具。Doug Cutting看到了机会并且领导开发了一个开源版本的MapReduce，称为Hadoop。随后，雅虎等公司纷纷响应，为其提供支持。今天，Hadoop已经成为许多互联网公司基础计算平台的一个核心部分，如雅虎、Facebook、LinkedIn和Twitter。许多传统的行业，如传媒业和电信业，也正在开始采用这个系统。我们将在第12章的案例中介绍《纽约时报》、中国移动和IBM等公司如何使用Hadoop。

Hadoop及大规模分布式数据处理，正在迅速成为许多程序员的一项重要技能。关系数据库、网络和安全这些在几十年前被认为是程序员可选技能的知识，今天已经成为一个高效程序员的必修课。同样，基本理解分布式数据处理将很快成为每个程序员的工具箱中不可或缺的一部分。斯坦福和卡内基-梅隆等一流的大学已经开始将Hadoop引入他们的计算机科学课程。这本书将会帮助你，一名执业的程序员，快速掌握Hadoop并用它来处理你的数据集。

---

① 当然，你读的是本正版书，对吗？

本章正式介绍Hadoop，找出它在分布式系统和数据处理系统方面的定位，并概述MapReduce编程模型。我们基于现有工具实现一个简单的单词统计示例，来彰显大型数据处理的挑战。然后，在使用Hadoop实现该示例之后，你会深刻体会Hadoop的简洁明了。我们还将讨论Hadoop的历史以及人们对MapReduce范式的一些观点。不过，让我先简单介绍一下为什么我写这本书，以及它为什么对你有用。

1.1 为什么写《Hadoop 实战》

实话实说，我第一次接触Hadoop即被其强大的能力所吸引，但随后在编写基本例程时却经历了一段令人沮丧的过程。虽然Hadoop官方网站上的文档相当全面，但是为简单的疑问找到直截了当的解答却并不总是那么容易。

写作本书的目的就是要解决这个问题。我不会关注过多的细节，相反，我提供的信息会有助于你快速创建可用代码，并会涉及在实践中最常遇到的更高级的话题。

1.2 什么是 Hadoop

按照正式的定义，Hadoop是一个开源的框架，可编写和运行分布式应用处理大规模数据。分布式计算是一个宽泛并且不断变化的领域，但Hadoop与众不同之处在于以下几点。

- 方便——Hadoop运行在由一般商用机器构成的大型集群上，或者如亚马逊弹性计算云（EC2）等云计算服务之上。
- 健壮——Hadoop致力于在一般商用硬件上运行，其架构假设硬件会频繁地出现失效。它可以从容地处理大多数此类故障。
- 可扩展——Hadoop通过增加集群节点，可以线性地扩展以处理更大的数据集。
- 简单——Hadoop允许用户快速编写出高效的并行代码。

Hadoop的方便和简单让其在编写和运行大型分布式程序方面占尽优势。即使是在校的大学生也可以快速、廉价地建立自己的Hadoop集群。另一方面，它的健壮性和可扩展性又使它胜任雅虎和Facebook最严苛的工作。这些特性使Hadoop在学术界和工业界都大受欢迎。

图1-1解释了如何与Hadoop集群交互。Hadoop集群是在同一地点用网络互连的一组通用机器。数据存储和处理都发生在这个机器“云”中[①]。不同的用户可以从独立的客户端提交计算“作业”到Hadoop，这些客户端可以是远离Hadoop集群的个人台式机。

并非所有分布式系统的构建都如图1-1所示的一样。下面，我们简要介绍一下其他的分布式系统，以便更好地展现Hadoop所依据的设计理念。

① 虽非绝对必要，但通常在一个Hadoop集群中的机器都是相对同构的x86 Linux服务器。而且它们几乎总是位于同一个数据中心，并通常在同一组机架里。

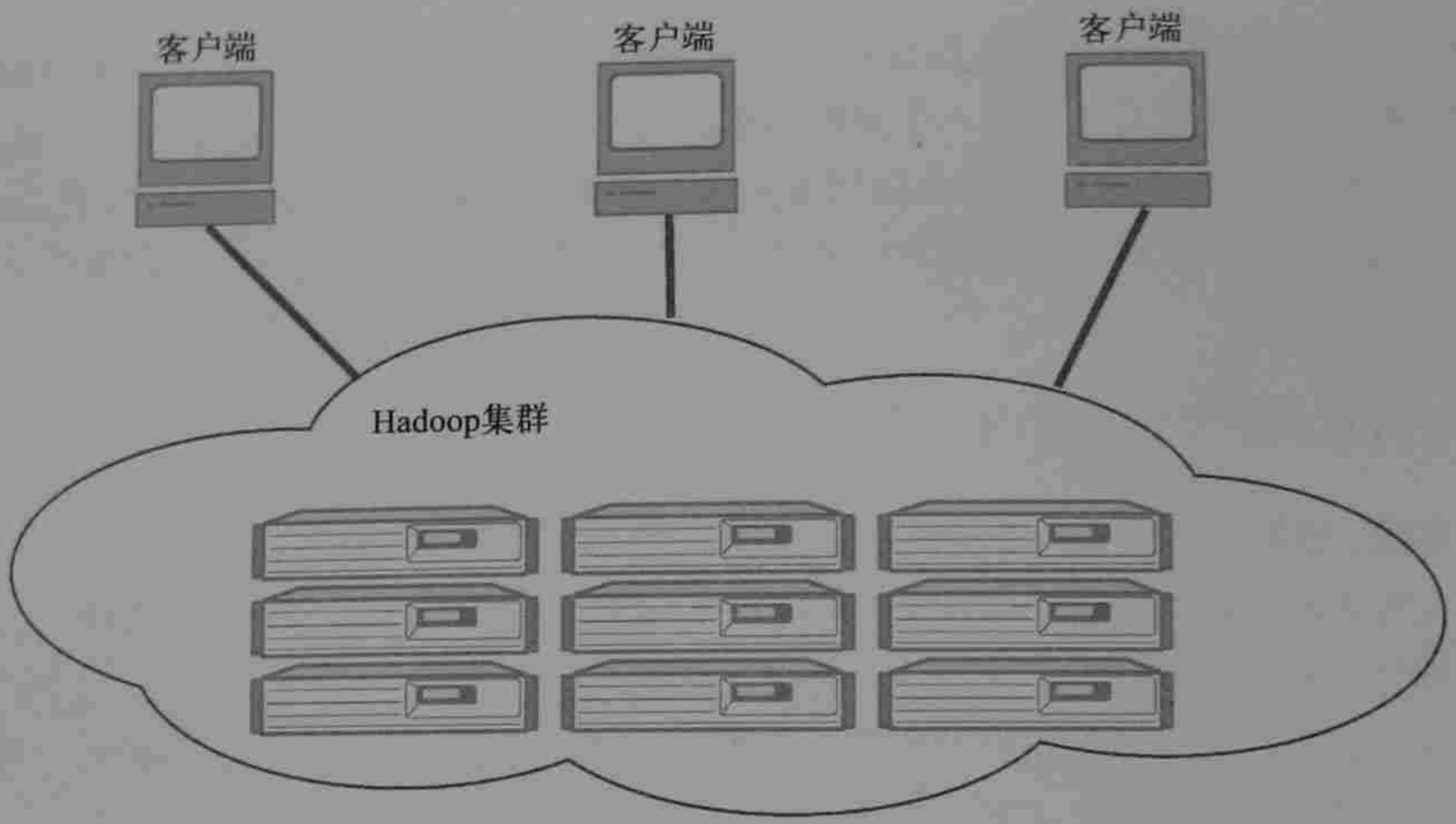

图1-1　一个Hadoop集群拥有许多并行的计算机，用以存储与处理大规模数据集。客户端计算机发送作业到计算云并获得结果

1.3　了解分布式系统和 Hadoop

摩尔定律在过去几十年间对我们都是适用的，但解决大规模计算问题却不能单纯依赖于制造越来越大型的服务器。有一种替代方案已经获得普及，即把许多低端/商用的机器组织在一起，形成一个功能专一的分布式系统。

为了理解盛行的分布式系统（俗称向外扩展）与大型单机服务器（俗称向上扩展）之间的对比，需要考虑现有I/O技术的性价比。对于一个有4个I/O通道的高端机，即使每个通道的吞吐量各为100 MB/sec，读取4 TB的数据集也需要3个小时！而利用Hadoop，同样的数据集会被划分为较小的块（通常为64 MB），通过Hadoop分布式文件系统（HDFS）分布在集群内多台机器上。使用适度的复制，集群可以并行读取数据，进而提供很高的吞吐量。而这样一组通用机器比一台高端服务器更加便宜！

前面的解释充分展示了Hadoop相对于单机系统的效率。现在让我们将Hadoop与其他分布式系统架构进行比较。一个众所周知的方法是SETI @ home，它利用世界各地的屏保来协助寻找外星生命。在SETI @ home，一台中央服务器存储来自太空的无线电信号，并在网上发布给世界各地的客户端台式机去寻找异常的迹象。这种方法将数据移动到计算即将发生的地方（桌面屏保）。经过计算后，再将返回的数据结果存储起来。

Hadoop在对待数据的理念上与SETI@home等机制不同。SETI @ home需要客户端和服务器之间重复地传输数据。这虽能很好地适应计算密集型的工作，但处理数据密集型任务时，由于数据规模太大，数据搬移变得十分困难。Hadoop强调把代码向数据迁移，而不是相反。参考图1-1，我们看到Hadoop的集群内部既包含数据又包含计算环境。客户端仅需发送待执行的MapReduce程序，而这些程序一般都很小（通常为几千字节）。更重要的是，代码向数据迁移的理念被应用

在Hadoop集群自身。数据被拆分后在集群中分布，并且尽可能让一段数据的计算发生在同一台机器上，即这段数据驻留的地方。

这种代码向数据迁移的理念符合Hadoop面向数据密集型处理的设计目标。要运行的程序（“代码”）在规模上比数据小几个数量级，更容易移动。此外，在网络上移动数据要比在其上加载代码更花时间。不如让数据不动，而将可执行代码移动到数据所在的机器上去。

现在你知道Hadoop是如何契合分布式系统的设计了，那就让我们看看它和通常的数据处理系统（SQL数据库）比较会怎么样。

1.4 比较 SQL 数据库和 Hadoop

鉴于Hadoop是一个数据处理框架，而在当前大多数应用中数据处理的主力是标准的关系数据库，那又是什么使得Hadoop更具优势呢？其中一个原因是，SQL（结构化查询语言）是针对结构化数据设计的，而Hadoop最初的许多应用针对的是文本这种非结构化数据。从这个角度来看，Hadoop比SQL提供了一种更为通用的模式。

若只针对结构化数据处理，则需要做更细致的比较。原则上，SQL和Hadoop可以互补，因为SQL是一种查询语言，它可将Hadoop作为其执行引擎[①]。但实际上，SQL数据库往往指代一整套传统技术，通过几个主要的厂商，面向一组历史悠久的应用进行优化。许多这些现有的商业数据库无法满足Hadoop设计所面向的需求。

考虑到这一点，让我们从特定的视角将Hadoop与典型SQL数据库做更详细的比较。

1. 用向外扩展代替向上扩展

扩展商用关系型数据库的代价是非常昂贵的。它们的设计更容易向上扩展。要运行一个更大的数据库，就需要买一个更大的机器。事实上，往往会看到服务器厂商在市场上将其昂贵的高端机标称为“数据库级的服务器”。不过有时可能需要处理更大的数据集，却找不到一个足够大的机器。更重要的是，高端的机器对于许多应用并不经济。例如，性能4倍于标准PC的机器，其成本将大大超过将同样的4台PC放在一个集群中。Hadoop的设计就是为了能够在商用PC集群上实现向外扩展的架构。添加更多的资源，对于Hadoop集群就是增加更多的机器。一个Hadoop集群的标配是十至数百台计算机。事实上，如果不是为了开发目的，没有理由在单个服务器上运行Hadoop。

2. 用键/值对代替关系表

关系数据库的一个基本原则是让数据按某种模式存放在具有关系型数据结构的表中。虽然关系模型具有大量形式化的属性，但是许多当前的应用所处理的数据类型并不能很好地适合这个模型。文本、图片和XML文件是最典型的例子。此外，大型数据集往往是非结构化或半结构化的。Hadoop使用键/值对作为基本数据单元，可足够灵活地处理较少结构化的数据类型。在Hadoop中，数据的来源可以有任何形式，但最终会转化为键/值对以供处理。

3. 用函数式编程（MapReduce）代替声明式查询（SQL）

SQL从根本上说是一个高级声明式语言。查询数据的手段是，声明想要的查询结果并让数据

① 其实这是Hadoop社区中的一个热点领域，我们将在第11章讨论其中一些领先的项目。

库引擎判定如何获取数据。在MapReduce中，实际的数据处理步骤是由你指定的，它很类似于SQL引擎的一个执行计划。SQL使用查询语句，而MapReduce则使用脚本和代码。利用MapReduce可以用比SQL查询更为一般化的数据处理方式。例如，你可以建立复杂的数据统计模型，或者改变图像数据的格式。而SQL就不能很好地适应这些任务。

另一方面，当数据处理非常适合于关系型数据结构时，有些人可能会发现使用MapReduce并不自然。那些习惯于SQL范式的人可能会发现用MapReduce来思考是一个挑战。我希望本书中的练习和示例能帮你更轻松地掌握MapReduce编程。不过值得注意的是，这里还有很多扩展可用，便于人们采用更熟悉的范式来编程，同时拥有Hadoop的可扩展性优势。事实上，使用某些扩展可采用一种类似SQL的查询语言，并自动将查询编译为可执行的MapReduce代码。我们将在第10章和第11章介绍其中的一些工具。

4. 用离线批量处理代替在线处理

Hadoop是专为离线处理和大规模数据分析而设计的，它并不适合那种对几个记录随机读写的在线事务处理模式。事实上，在本书写作时（以及在可预见的未来），Hadoop最适合一次写入、多次读取的数据存储需求。在这方面它就像SQL世界中的数据仓库。

你已经从宏观上看到Hadoop与分布式系统和SQL数据库之间的关系。那么，让我们开始学习如何用它来编程。为此，我们首先需要了解Hadoop中的MapReduce范式。

1.5　理解 MapReduce

你也许知道管道和消息队列等数据处理模型。这些模型可专用于数据处理应用的方方面面。Unix pipes就是一种最常见的管道。管道有助于进程原语的重用，已有模块的简单链接即可组成一个新的模块；消息队列则有助于进程原语的同步。程序员将数据处理任务以生产者或消费者的形式编写为进程原语，由系统来管理它们何时执行。

同样，MapReduce也是一个数据处理模型，它最大的优点是容易扩展到多个计算节点上处理数据。在MapReduce模型中，数据处理原语被称为mapper和reducer。分解一个数据处理应用为mapper和reducer有时是繁琐的，但是一旦以MapReduce的形式写好了一个应用程序，仅需修改配置就可以将它扩展到集群中几百、几千，甚至几万台机器上运行。正是这种简单的可扩展性使得MapReduce模型吸引了众多程序员。

MapReduce的几种写法

尽管已有不少关于MapReduce的论述，但并没有一个统一的写法。原始的谷歌文章和维基百科条目上使用驼峰大小写方式写为MapReduce。然而，谷歌自己在其网站的一些网页上采用了Map Reduce的写法（例如，http://research.google.com/roundtable/MR.html）。在Hadoop官方网站文档中，可以找到一个指向Map-Reduce Tutorial的链接。点击该链接，会进入一个名为Hadoop Map/Reduce Tuotorial的页面(http://hadoop.apache.org/core/docs/current/ mapred_tutorial.html)，其中是对Map/Reduce框架的解释。写法的差别也存在于不同的Hadoop组件，如NameNode (name node、name-node及namenode)、DataNode、JobTracker和TaskTracker。为了统一起见，我们在本书中全部采用驼峰大小写方式。（即MapReduce、NameNode、DataNode、JobTracker和 TaskTracker。）

1.5.1 动手扩展一个简单程序

在正式论述MapReduce之前，让我们先做一个练习，即扩展一个简单的程序让其处理一大段数据。然后，你会认识到扩展数据处理程序所面对的挑战，从而更好地体会MapReduce这种框架帮助你处理繁琐事务时的好处。

我们的练习是统计一组文档中的每个单词出现的次数。在这个例子中，我们的文档仅有一个文件，文件中只有一句话：

Do as I say, not as I do.

我们得到如右图所示单词统计的值。

单　词	计　数
as	2
do	2
i	2
not	1
say	1

我们将这个特定的练习称为单词统计。如果文档集合很小，一个简单的程序即可完成这项工作。可写为如下一段伪代码：

```
define wordCount as Multiset;
for each document in documentSet {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
display(wordCount);
```

该程序循环遍历所有的文档。对于每个文档，使用分词过程逐个地提取单词。对于每个单词，在多重集合wordCount中的相应项上加1。最后，display()函数打印出wordCount中的所有条目。

注意 多重集合中每个元素都有一个计数值。我们试图产生的单词统计是一个多重集合的典型例子。实际上，它通常用一个散列表来实现。

这个程序只适合处理少量文档，一旦文档数量激增，它就不能胜任了。例如，你想编写一个垃圾邮件过滤器，来获取接收到的几百万封垃圾邮件中经常使用的单词。使用单台计算机反复遍历所有文档将会非常费时。重写程序，让工作可以分布在多台机器上。每台机器处理这些文档的不同部分。当所有的机器都完成时，第二个处理阶段将合并这些结果。第一阶段要分布到多台机器上去的伪代码为：

```
define wordCount as Multiset;
for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
sendToSecondPhase(wordCount);
```

第二阶段的伪代码为：

```
define totalWordCount as Multiset;
```

```
for each wordCount received from firstPhase {
    multisetAdd (totalWordCount, wordCount);
}
```

这不是很难，对吗？但是，一些细节可能会妨碍它按预期工作。首先，我们忽略了文档读取的性能需求。如果文件都存在一个中央存储服务器上，那么瓶颈就是该服务器的带宽。让更多的机器参与处理的办法不会一直有效，因为有时存储服务器的性能会跟不上。因此，你需要将文档分开存放，使每台机器可以仅处理自己所存储的文档，从而消除单个中央存储服务器的瓶颈。这呼应了先前的观点，即在数据密集型分布式应用中存储和处理不得不紧密地绑定在一起。

该程序的另一个缺陷是wordCount（和totalWordCount）被存储在内存中。当处理大型文档集时，一个特定单词的数量就会超过一台机器的内存容量。英语有大约一百万个词，这个大小可以很轻松地放进一个iPod，但我们的单词统计程序将处理许多不在标准英语单词词典中的特殊单词。例如，我们必须处理诸如Hadoop这样的特定名称。我们必须统计错字，即使它们并不是真正的单词（例如，exampel），我们还需分别统计一个字的所有不同形式（例如，eat，ate，eaten和eating）。即使在文档中特定单词的数量可以被管理在内存中，在问题的定义上略有变化就可能会引起空间复杂度的爆炸。例如，我们不统计文档中的单词，而是去统计日志文件中的IP地址，或者Bigram的频率。那么我们处理的多重集合条目将达到数十亿，这超过了大多数商用计算机的内存容量。

注意 bigram是一对连续的单词。句子“Do as I say, not as I do”可分为以下的bigram：Do as, as I, I say, say not, not as, as I, I do。类似地，trigram是连续的三组词。bigram和trigram在自然语言处理中都非常重要。

wordCount也许无法放在内存中；我们将不得不改写我们的程序，以便在磁盘上存储该散列表。这意味着我们将实现一个基于磁盘的散列表，其中涉及大量的编码。

此外，请记住，第二阶段只有一台计算机，它将处理来自所有计算机在第一阶段计算wordCount的结果。wordCount的处理任务原本就相当繁重。当我们为第一阶段的处理提供充足的计算机时，第二阶段的单台计算机将成为瓶颈。最明显的问题是，我们能否按分布模式重写第二阶段，以便它可以通过增加更多的计算机来实现扩展？

答案是肯定的。为了使第二阶段以分布的方式运转，必须以某种方式将其分割到在多台计算机上，使之能够独立运行。需要在第一阶段之后将wordCount分区，使得第二阶段的每台计算机仅需处理一个分区。举一个例子，假设我们在第二阶段有26台计算机。我们让每台计算机上的wordCount只处理以特定字母开头的单词。例如，计算机A在第二阶段仅统计以字母a开头的单词。为了在第二阶段中实现这种划分，我们需要对第一阶段稍作修改。不再采用基于磁盘的散列表实现wordCount，而是划分出26个表：wordCount-a，wordCount-b等。每个表统计以特定字母开头的单词。经过第一阶段，来自该阶段所有计算机的wordCount-a结果将被发送到第二阶段的计算机A上，所有wordCount-b的结果将被发送到计算机B上，依次类推。第一阶段中的每台计算机都会将结果洗牌到第二阶段的计算机上。

这个单词统计程序现在正变得复杂。为了使它工作在一个分布式计算机集群上，我们发现需要添加以下功能。

- 存储文件到许多台计算机上（第一阶段）。
- 编写一个基于磁盘的散列表，使得处理不受内存容量限制。
- 划分来自第一阶段的中间数据（即wordCount）。
- 洗牌这些分区到第二阶段中合适的计算机上。

即使对于单词统计这样简单的程序，这都是繁重的工作，而我们甚至还没有涉及容错等问题。（如果在任务执行过程中一个计算机失效该怎么办？）这就是为什么我们需要一个像Hadoop一样的框架。当你用MapReduce模型来写应用程序，Hadoop将替你管理所有与可扩展性相关的底层问题。

1.5.2 相同程序在 MapReduce 中的扩展

MapReduce程序的执行分为两个主要阶段，为mapping和reducing。每个阶段均定义为一个数据处理函数，分别被称为mapper和reducer。在mapping阶段，MapReduce获取输入数据并将数据单元装入mapper。在reducing阶段，reducer处理来自mapper的所有输出，并给出最终结果。

简而言之，mapper意味着将输入进行过滤与转换，使reducer可以完成聚合。可以发现这和我们在扩展单词统计时的两个阶段惊人地相似。这种相似性并非偶然。MapReduce的设计建立在编写可扩展、分布式程序的丰富经验之上。这种两阶段的设计模式可以在许多程序的扩展中看到，成为MapReduce框架的基础。

上一节扩展分布式的单词统计程序时，我们不得不编写了partitioning和shuffling函数。它们与mapping和reducing一起形成常见的设计模式。但不同的是，partitioning和shuffling并不依赖特殊的数据处理应用，它们是通用的功能，MapReduce框架提供了可在大多数情况下工作的默认实现。

为了让mapping、reducing、partitioning和shuffling（以及几个我们尚未涉及的函数）能够无缝地在一起工作，我们需要在数据的通用结构上达成一致。它需要足够灵活和强大，以适应大多数的数据处理应用。MapReduce使用列表和键/值对作为其主要的数据原语。键与值通常为整数或字符串，但也可以是可忽略的假值，或者是复杂的对象类型。map和reduce函数必须遵循以下对键和值类型的约束。

	输　入	输　出
map	<k1, v1>	list(<k2, v2>)
reduce	<k2, list(v2)>	list(<k3, v3>)

在MapReduce框架中编写应用程序就是定制化mapper和reducer的过程。让我们看看完整的数据流。

(1) 应用的输入必须组织为一个键/值对的列表list(<k1, v1>)。输入格式可能看起来是不受约束的，但在实际中它非常简洁。用于处理多个文件的输入格式通常为list(<String filename, String file_content>)。用于处理日志文件这种大文件的输入格式为list(<Integer line_number, String log_event>)。

(2) 含有键/值对的列表被拆分，进而通过调用mapper的map函数对每个单独的键/值对<k1,

v1>进行处理。在这里，键k1经常被mapper所忽略。mapper转换每个<k1，v1>对并将之放入<k2，v2>对的列表中。这种转换的细节很大程度上决定了MapReduce程序的行为。值得注意的是，处理键/值对可以采用任意的顺序。而且，这种转换必须是封闭的，使得输出仅依赖于一个单独的键/值对。

对于单词统计，<String filename, String file_content>被输入mapper，而其中的filename被忽略。mapper可以输出一个<String word, Integer count>的列表，但也可以有更简单的形式。我们知道，计数值将在后续的阶段聚合，我们可以输出一个有重复条目的<String word, Integer 1>列表，并让完整的聚合稍后执行。这就是说，在输出列表中，可以出现一次键/值对<"foo", 3>，或者出现3次<"foo", 1>。可以看到，后者更容易编程。前者会获得一些性能上的优化，但是在完全掌握MapReduce框架之前，我们先不讨论这些优化。

(3) 所有mapper的输出（在概念上）被聚合到一个包含<k2,v2>对的巨大列表中。所有共享相同k2的对被组织在一起形成一个新的键/值对<k2, list(v2)>。框架让reducer来分别处理每一个被聚合起来的<k2, list(v2)>。回到单词统计的例子，一个文档的map输出的列表中可能出现三次<"foo", 1>，而另一个文档的map输出列表可能出现两次<"foo", 1>。reducer所看到的聚合的对为<"foo", list(1,1,1,1,1)>。在单词统计中，reducer的输出为<"foo", 5>，表示“foo”在文档集合中总计出现的次数。每一个reducer负责不同的单词。MapReduce框架自动搜集所有的<k3,v3>对，并将之写入文件。在单词统计例子中需要注意，k2和k3的数据类型是相同的，v2和v3也是相同的。但并不是所有的数据处理应用都是这样。

让我们基于MapReduce重写单词统计程序，看看所有这些是如何结合在一起的。代码清单1-1给出了伪代码。

代码清单1-1　单词统计中map和reduce函数的伪代码

```
map(String filename, String document) {
    List<String> T = tokenize(document);
    for each token in T {
        emit ((String)token, (Integer) 1);
    }
}
reduce(String token, List<Integer> values) {
    Integer sum = 0;
    for each value in values {
        sum = sum + value;
    }
    emit ((String)token, (Integer) sum);
}
```

以前我们提到map和reduce函数的输出都是列表。如伪代码所示，我们在框架中实际使用了一个特殊的函数emit()来逐个生成列表中的元素。这个emit()函数进一步将程序员从管理一个大列表的工作中解放出来。

这个代码看起来与1.5.1节的类似，而此时它却能够实际地以扩展方式运行。可见，Hadoop使得建立一个可扩展的分布式程序变容易了，对吧？现在让我们将这段伪代码变成一个Hadoop程序。

1.6 用 Hadoop 统计单词——运行第一个程序

既然已经了解了Hadoop和MapReduce的框架，让我们把它运行起来。在本章，Hadoop是运行在单机上的，可以是台式机或是笔记本。下一章将为你展示如何在集群上运行Hadoop，以满足实际部署的需要。在单机上运行Hadoop主要是为了完成开发的工作。

Linux是Hadoop公认的开发与生产平台。虽然Windows也可以支持开发模式，但你需要在节点上安装cygwin(http://www-cygwin.com/)来支持shell和Unix脚本。

> **注意** 有许多人声称已经成功地将Hadoop以开发模式运行在其他Unix的变体上，如Solaris和Mac OS X。事实上，如果在笔记本上开发，Hadoop的开发者似乎总是选择苹果的MacBook Pro笔记本，这在Hadoop大会上或者用户组会议上随处可见。

运行Hadoop需要Java1.6或更高版本。Mac用户可以从苹果公司获得，而其他操作系统的用户可以从Sun公司的网站下载最新的JDK，网址为http://java.sun.com/javase/downloads/index.jsp。安装后请记住Java的安装根目录，这在以后会用到。

要安装Hadoop，首先需要从网站http://hadoop.apache.org/core/releases.html上下载最近的稳定版本。在打开发布包后，编辑脚本conf/hadoop-env.sh将JAVA_HOME设置为刚才记下来的Java安装根目录。例如，在Mac OS X中，需要把这一行

```
# export JAVA_HOME=/usr/lib/j2sdk1.5-sun
```

替换为

```
export JAVA_HOME=/Library/Java/Home
```

Hadoop脚本以后会经常使用，让我们不加任何参数运行它，来看一看它的用法文档。输入：

```
bin/hadoop
```

我们得到

```
Usage: hadoop [--config confdir] COMMAND
这里COMMAND为下列其中一个：
namenode -format                    格式化DFS文件系统
secondarynamenode                   运行DFS的第二个namenode
namenode                            运行DFS的namenode
datanode                            运行一个DFS的datanode
dfsadmin                            运行一个DFS的admin 客户端
fsck                                运行一个DFS文件系统的检查工具
fs                                  运行一个普通的文件系统用户客户端
balancer                            运行一个集群负载均衡工具
jobtracker                          运行MapReduce的jobTracker节点
pipes                               运行一个Pipes作业
tasktracker                         运行一个MapReduce的taskTracker节点
job                                 处理MapReduce作业
version                             打印版本
jar <jar>                           运行一个jar文件
distcp <srcurl> <desturl>           递归地复制文件或者目录
```

```
archive -archiveName NAME <src>* <dest>    生成一个Hadoop档案
daemonlog                                  获取或设置每个daemon的log级别
或CLASSNAME                                运行名为CLASSNAME的类大多数命令会在使用w/o参数
                                           时打出帮助信息。
```

本书将涵盖Hadoop的各种命令。而当前我们仅需知道运行一个（java）Hadoop程序的命令为bin/hadoop jar <jar>。就像命令中显示的那样，用Java写的Hadoop程序被打包为jar执行文件。

幸运的是，我们无需先写一个Hadoop程序；默认安装中已经有几个我们可以使用的例程。下面的命令可以列出那些jar文件的例程：

```
bin/hadoop jar hadoop-*-examples.jar
```

你将看到一组已经被打包在Hadoop中的例程，其中一个就是单词统计程序，名为wordcount！这个程序重要的（内部）类显示在代码清单1-2中。我们将看到这个Java程序是如何实现代码清单1-1中单词统计伪代码中map和reduce功能的。我们将修改这个程序，理解如何让其行为发生变化。这里我们假设仅需按部就班地执行Hadoop程序，它就能够正常工作。

不指定任何参数执行wordcount将显示一些有关用法的信息：

```
bin/hadoop jar hadoop-*-examples.jar wordcount
```

显示出的参数列表为

```
wordcount [-m <maps>] [-r <reduces>] <input> <output>
```

唯一的参数是所需分析的文本文档的输入目录（<input>），以及程序填充结果的输出目录（<output>）。为了执行wordcount，我们需要首先生成一个输入目录

```
mkdir input
```

并放一些文档在里面。你可以添加任何的文本文档到这个目录中。为了解释，让我们从http://www.gpoaccess.gov/sou/下载一个2002年的国情咨文并添加进去。我们现在分析它的单词统计并看看结果：

```
bin/hadoop jar hadoop-*-examples.jar wordcount input output
more output/*
```

你将看到在文档中用到的所有单词的一个统计结果，它们按照字母顺序进行排列。鉴于你还没有开始写一行代码就能这样，这相当不错！但是，还需注意在这个wordcount程序中有一些不足。分词完全根据空格而不是根据标点符号，这使得“States”、“States.”和“States:”分别成为单独的单词。大小写也是一样，比如States和states会表示为不同的单词。另外，我们还希望能够略去在文档中仅显示一次或者两次的那些单词。

好在可以得到wordcount的源码，安装后它被放在src/examples/org/apache/hadoop/xamples/WordCount.java中。我们可以根据需要来修改它。首先我们建立一个playground的目录结构并复制这个程序。

```
mkdir playground
mkdir playground/src
mkdir playground/classes
cp src/examples/org/apache/hadoop/examples/WordCount.java
➥ playground/src/WordCount.java
```

在修改程序之前，我们在Hadoop框架中先编译和执行这个副本：

```
javac -classpath hadoop-*-core.jar -d playground/classes
➥ playground/src/WordCount.java
jar -cvf playground/wordcount.jar -C playground/classes/ .
```

你必须每次在运行Hadoop命令时删掉输出目录，因为它是自动生成的。

```
bin/hadoop jar playground/wordcount.jar
➥ org.apache.hadoop.examples.WordCount input output
```

再看一下输出目录中的文件。既然我们没有改变任何程序代码，结果应该与以前一样。我们只是编译了自己的副本，而不去运行预编译的版本。

现在我们准备修改WordCount来增加额外的功能。代码清单1-2显示了WordCount.java程序的一部分，去掉了注释和支撑性代码。

代码清单1-2 WordCount.java

```
public class WordCount extends Configured implements Tool {

  public static class MapClass 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 itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.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));
    }
  }

  ...
}
```

❶ 使用空格进行分词

❷ 把Token放入Text对象中[①]

❸ 输出每个Token的统计结果

① Token是分词的基本单位，以单词切分则每个单词为一个Token。——译者注

WordCount.java和我们的MapReduce伪代码之间主要的功能区别为，在WordCount.java中map()一次处理一行文本，而伪代码一次处理一个文档。这种区别仅通过观察WordCount.java可能并不容易看出来，因为这是Hadoop的默认配置。

代码清单1-2中的代码和代码清单1-1中的伪代码几乎是一样的，只是Java的语法让代码更加冗长。Map和Reduce函数是在WordCount的内部类中。你可能注意到它们是一些特殊的类，如LongWritable、IntWritable和Text，而不是更为常见的Java类Long、Integer和String。现在可以停下来想一想这些实现的细节。这些新类增加了Hadoop内部所需的串行化能力。

很容易找到我们要对程序进行修改的地方。我们看到在❶的位置上WordCount以默认配置使用了Java的StringTokenizer，这里仅基于空格来分词。为了在分词过程中忽略标准的标点符号，我们将它们加入到StringTokenizer的定界符列表中。

```
StringTokenizer itr = new StringTokenizer(line, " \t\n\r\f,.:;?![]'");
```

当遍历token的集合时，每个token被提取出来并放进一个Text对象❷。（在Hadoop中，特殊的Text类取代了String。）因为希望单词统计忽略大小写，我们在把它们转换为Text对象前先将所有的单词都变成小写。

```
word.set(itr.nextToken().toLowerCase());
```

最后，我们希望仅仅显示出现次数大于4次的单词。我们修改❸来使得输出结果仅搜集符合条件的单词统计。（这等价于在Hadoop中实现伪代码中的emit()）。

```
if (sum > 4) output.collect(key, new IntWritable(sum));
```

在修改了上述3行后，可以重新编译程序并执行，结果显示在表1-1中。

表1-1　2002年国情咨文中出现频次大于4的单词统计

11th (5)	citizens (9)	its (6)	over (6)	to (123)
a (69)	congress (10)	jobs (11)	own (5)	together (5)
about (5)	corps (6)	join (7)	page (7)	tonight (5)
act (7)	country (10)	know (6)	people (12)	training (5)
afghanistan (10)	destruction (5)	last (6)	protect (5)	united (6)
all (10)	do (6)	lives (6)	regime (5)	us (6)
allies (8)	every (8)	long (5)	regimes (6)	want (5)
also (5)	evil (5)	make (7)	security (19)	war (12)
America (33)	for (27)	many (5)	september (5)	was (11)
American (15)	free (6)	more (11)	so (12)	we (76)
americans (8)	freedom (10)	most (5)	some (6)	we've (5)
an (7)	from (15)	must (18)	states (9)	weapons (12)
and (210)	good (13)	my (13)	tax (7)	were (7)
are (17)	great (8)	nation (11)	terror (13)	while (5)
as (18)	has (12)	need (7)	terrorist (12)	who (18)

（续）

ask (5)	have (32)	never (7)	terrorists (10)	will (49)
at (16)	health (5)	new (13)	than (6)	with (22)
be (23)	help (7)	no (7)	that (29)	women (5)
been (8)	home (5)	not (15)	the (184)	work (7)
best (6)	homeland (7)	now (10)	their (17)	workers (5)
budget (7)	hope (5)	of (130)	them (8)	world (17)
but (7)	i (29)	on (32)	these (18)	would (5)
by (13)	if (8)	one (5)	they (12)	yet (8)
camps (8)	in (79)	opportunity (5)	this (28)	you (12)
can (7)	is (44)	or (8)	thousands (5)	
children (6)	it (21)	our (78)	time (7)	

我们看到有128个单词出现的次数大于4次。其中许多在几乎所有的英文文本中都经常出现，如a (69)、and (210)、i (29)、in (79)、the (184)等。我们还能发现一些单词，它们概述了美国当前所面临的问题：terror (13)、terrorist (12)、terrorists (10)、security (19)、weapons (12)、destruction (5)、afghanistan (10)、freedom (10)、jobs (11)、budget (7) 等。

1.7 Hadoop 历史

Hadoop开始时是Nutch的一个子项目，而Nutch又是Apache Lucene的一个子项目。这3个项目都是由Doug Cutting所创立的，每个项目在逻辑上都是前一个项目的演进。

Lucene是一个功能全面的文本索引和查询库。给定一个文本集合，开发者就可以使用Lucene引擎方便地在文档上添加搜索功能。桌面搜索、企业搜索，以及许多领域特定的搜索引擎使用的都是Lucene。作为Lucene的扩展，Nutch的目标可谓雄心勃勃，它试图以Lucene为核心建立一个完整的Web搜索引擎。Nutch为HTML提供了解析器，还具有网页抓取工具、链接图形数据库和其他网络搜索引擎的额外组件。Doug Cutting所设想的Nutch是开放与民主的，可以替代Google等商业产品的垄断技术。

除了增加了像抓取器和解析器这样的组件，网络搜索引擎与基本的文档搜索引擎的区别就在于规模。Lucene的目标是索引数百万的文档，但Nutch应该能够处理数十亿的网页，而不会带来过度的操作开销。这样Nutch就得运行在由商用硬件组成的分布式集群上。Nutch团队面临的挑战是解决软件可扩展性问题，即要在Nutch中建立一个层，来负责分布式处理、冗余、自动故障恢复和负载均衡。这些挑战绝非易事。

在2004年左右，Google发表了两篇论文来论述Google文件系统（GFS）和MapReduce框架。Google声称使用了这两项技术来扩展自己的搜索系统。Doug Cutting立即看到了这些技术可以适用于Nutch，接着他的团队实现了一个新的框架，将Nutch移植上去。这种新的实现马上提升了Nutch的可扩展性。它开始能够处理几亿个网页，并能够运行在几十个节点的集群上。Doug认识

到设计一个专门的项目可以充实两种网络扩展所需的技术，于是就有了Hadoop。雅虎在2006年1月聘用Doug，让他和一个专项团队一起改进Hadoop，并将其作为一个开源项目。两年后，Hadoop成为Apache的顶级项目。后来，在2008年2月19日，雅虎宣布其索引网页的生产系统采用了在10 000多个核的Linux集群上运行的Hadoop（见http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html）。Hadoop真正达到了万维网的规模！

这个名字是怎么来的

为软件项目命名时，Doug Cutting似乎总会得到家人的启发。Lucene是他妻子的中间名，也是她外祖母的名字。他的儿子在咿呀学语时，总把所有用于吃饭的词叫成Nutch，后来，他把一个黄色大象毛绒玩具叫做Hadoop。Doug说，他“在寻找一个名字，不是已经存在的域名或商标，所以我尝试生活中以前没有人用过的各种词汇。而孩子们很擅长创造单词。”

1.8　小结

Hadoop是一个通用的工具，它让新用户可以享受到分布式计算的好处。通过采用分布式存储、迁移代码而非迁移数据，Hadoop在处理大数据集时避免了耗时的数据传输问题。此外，数据冗余机制允许Hadoop从单点失效中恢复。你已经看到在Hadoop中使用MapReduce框架编写程序非常方便，而且同等重要的是，此时你不必担心如何分割数据、如何分配任务执行节点，或者如何管理节点间的通信。Hadoop为你处理这些事务，使你可以专注于那些最重要的事情——你的数据以及你想用它做什么。

下一章我们将深入Hadoop内部并探究构建Hadoop集群的更多细节。

1.9　资源

Hadoop的官方网站：http://hadoop.apache.org/。

Google文件系统和MapReduce的原始论文很值得一读，可以了解它们的底层设计和架构：

- *The Google File System*——http:// labs.google.com/papers/gfs.html
- *MapReduce: Simplified Data Processing on Large Clusters*——http://labs.google.com/papers/mapreduce.html