目录

第 1 章 初识 Hadoop

古时候，人们用牛来拉重物，当一头牛拉不动一根圆木的时候，他们不曾想过培育个头更大的牛。同样，我们也不需要尝试更大的计算机，而是应该开发更多的计算系统。

--格蕾斯·霍珀

1.1 数据！数据

我们生活在数据时代！很难估计全球存储的电子数据总量是多少，但是据 IDC 估计 2006 年"数字全球"项目(digital universe)的数据总量为 0.18 ZB，并且预测到 2011 年这个数字将达到 1.8 ZB，为 2006 年的 10 倍 。1 ZB 相当于 10 的 21 次方字节的数据，或者相当于 1000 EB，1 000 000 PB，或者大家更熟悉的 10 亿 TB 的数据！这相当于世界上每个人一个磁盘驱动器的数量级。

这一数据洪流有许多来源。考虑下文：

纽约证券交易所每天产生 1 TB 的交易数据。

著名社交网站 Facebook 的主机存储着约 100 亿张照片，占据 PB 级存储空间。

Ancestry.com，一个家谱网站，存储着 2.5 PB 数据。

互联网档案馆(The Internet Archive)存储着约 2 PB 数据，并以每月至少 20 TB 的速度增长。

瑞士日内瓦附近的大型强子对撞机每年产生约 15 PB 的数据。

此外还有大量数据。但是你可能会想它对自己有何影响。大部分数据被锁定在最大的网页内容里面(如搜索引擎)或者是金融和科学机构，对不对？是不是所谓的"大数据"的出现会影响到较小的组织或个人？

我认为是这样的。以照片为例，我妻子的祖父是一个狂热的摄影爱好者，并且他成人之后，几乎一直都在拍照片。他的所有照片(中等格式、幻灯片和 35 mm 胶片)，在扫描成高解析度照片时，占了大约 10 GB 的空间。相比之下，我家去年一年用数码相机拍摄的照片就占用了 5 GB 的空间。我家产生照片数据的速度是我妻子祖父的 35 倍！并且，随着拍摄更多的照片变得越来越容易，这个速度还在增加中。

更常见的情况是，个人数据的产生量正在快速地增长。微软研究院的 MyLifeBits 项目(http://research.microsoft.com/en-us/projects/mylifebits/default.aspx)显示，在不久的将来，个人信息档案将可能成为普遍现象。MyLifeBits 是这样的一个实验：一个人与外界的联系(电话、邮件和文件)被抓取和存储供以后访问。收集的数据包括每分钟拍摄的照片等，导致整

个数据量达到每月 1 GB 的大小。当存储成本下降到使其可以存储连续的音频和视频时，服务于未来 MyLifeBits 项目的数据量将是现在的许多倍。

个人数据的增长的确是大势所趋，但更重要的是，计算机所产生的数据可能比人所产生的数据更大。机器日志、RFID 读取器、传感器网络、车载 GPS 和零售交易数据等，这些都会促使"数据之山越来越高"。

公开发布的数据量也在逐年增加。作为组织或企业，再也不能只管理自己的数据，未来的成功在很大程度上取决于它是否能从其他组织的数据中提取出价值。

这方面的先锋(如亚马逊网络服务器、Infochimps.org 或者 andtheinfo.org)的公共数据集，它们的存在就在于促进"信息共享"，任何人都可以共享并自由(或以 AWS 平台的形式，或以适度的价格)下载和分析这些数据。不同来源的信息混合处理后会带来意外的效果和至今难以想像的应用。

以 Astrometry.net 项目为例，这是一个研究 Flickr 网站上天体爱好者群中新照片的项目。它分析每一张上传的照片，并确定它是天空的哪一部分，或者是否是有趣的天体，如恒星或者星系。虽然这只是一个带实验性质的新服务，但是它显示了数据(这里特指摄影照片)的可用性并且被用来进行某些活动(图像分析)，而这些活动很多时候并不是数据创建者预先能够想像到的。

有句话是这么说的："算法再好，通常也难敌更多的数据。"意思是说对于某些问题(譬如基于既往偏好生成的电影和音乐推荐)，不论你的算法有多么猛，它们总是会在更多的数据面前无能为力(更不用说没有优化过的算法了)。

现在，我们有一个好消息和一个坏消息。好消息是有海量数据！坏消息是我们正在为存储和分析这些数据而奋斗不息。

1.2 数据的存储和分析

问题很简单：多年来硬盘存储容量快速增加的同时，访问速度-- 数据从硬盘读取的速度-- 却未能与时俱进。1990 年，一个普通的硬盘驱动器可存储 1370 MB 的数据并拥有 4.4 MB/s 的传输速度 ，所以，只需五分钟的时间就可以读取整个磁盘的数据。20 年过去了，1 TB 级别的磁盘驱动器是很正常的，但是数据传输的速度却在 100 MB/s 左右。所以它需要花两个半小时以上的时间读取整个驱动器的数据。

从一个驱动器上读取所有的数据需要很长的时间，写甚至更慢。一个很简单的减少读取时间的办法是同时从多个磁盘上读取数据。试想一下，我们拥有 100 个磁盘，每个存储百分之一的数据。如果它们并行运行，那么不到两分钟我们就可以读完所有的数据。

只使用一个磁盘的百分之一似乎很浪费。但是我们可以存储 100 个数据集，每个 1 TB，并让它们共享磁盘的访问。我们可以想像，此类系统的用户会很高兴看到共享访问可以缩短分析时间，并且，从统计角度来看，他们的分析工作会分散到不同的时间点，所以互相之间不会有太多干扰。

尽管如此，现在更可行的是从多个磁盘并行读写数据。

第一个需要解决的问题是硬件故障。一旦开始使用多个硬件设施，其中一个会出故障的概率是非常高的。避免数据丢失的常见做法是复制：通过系统保存数据的冗余副本，在故障发生时，可以使用数据的另一份副本。这就是冗余磁盘阵列的工作方式。Hadoop 的文件系统 HDFS(Hadoop Distributed Filesystem)也是一个例子，虽然它采取的是另一种稍有不同的方法，详见后文描述。

第二个问题是大部分分析任务需要通过某种方式把数据合并起来，即从一个磁盘读取的数据可能需要和另外 99 个磁盘中读取的数据合并起来才能使用。各种不同的分布式系统能够组合多个来源的数据，但是如何保证正确性是一个非常难的挑战。MapReduce 提供了一个编程模型，其抽象出上述磁盘读写的问题，将其转换为计算一个由成对键/值组成的数据集。这种模型的具体细节将在后面的章节讨论。但是目前讨论的重点是，这个计算由两部分组成：Map 和 Reduce。这两者的接口就是"整合"之地。就像 HDFS 一样，MapReduce 是内建可靠性这个功能的。

简而言之，Hadoop 提供了一个稳定的共享存储和分析系统。存储由 HDFS 实现，分析由 MapReduce 实现。纵然 Hadoop 还有其他功能，但这些功能是它的核心所在。

1.3 相较于其他系统

MapReduce 似乎采用的是一种蛮力方法。即，针对每个查询，每一个数据集-- 至少是很大一部分-- 都会被处理。但这正是它的能力。MapReduce 可以处理一批查询，并且它针对整个数据集处理即席查询并在合理时间内获得结果的能力也是具有突破性的。它改变了我们对数据的看法，并且解放了以前存储在磁带和磁盘上的数据。它赋予我们对数据进行创新的机会。那些以前需要很长时间才能获得答案的问题现在已经迎刃而解，但反过来，这又带来了新的问题和见解。

例如，Rackspace 的邮件部门 Mailtrust，用 Hadoop 处理邮件的日志。他们写的一个查询是找到其用户的地理分布。他们是这样说的：

"随着我们的壮大，这些数据非常有用，我们每月运行一次 MapReduce 任务来帮助我们决定哪些 Rackspace 数据中心需要添加新的邮件服务器。"

通过将数百 GB 的数据整合，借助于分析工具，Rackspace 的工程师得以了解这些数据，否则他们永远都不会了解，并且他们可以运用这些信息去改善他们为用户提供的服务。第 14 章将详细介绍 Rackspace 公司是如何运用 Hadoop 的。

1.3.1 关系型数据库管理系统

为什么我们不能使用数据库加上更多磁盘来做大规模的批量分析？为什么我们需要 MapReduce？

这个问题的答案来自于磁盘驱动器的另一个发展趋势：寻址时间的提高速度远远慢于传输速率的提高速度。寻址就是将磁头移动到特定位置进行读写操作的工序。它的特点是磁盘操作有延迟，而传输速率对应于磁盘的带宽。

如果数据的访问模式受限于磁盘的寻址，势必会导致它花更长时间(相较于流)来读或写大部分数据。另一方面，在更新一小部分数据库记录的时候，传统的 B 树(关系型数据库中使用的一种数据结构，受限于执行查找的速度)效果很好。但在更新大部分数据库数据的时候，B 树的效率就没有 MapReduce 的效率高，因为它需要使用排序/合并来重建数据库。

在许多情况下，MapReduce 能够被视为一种 RDBMS(关系型数据库管理系统)的补充。(两个系统之间的差异见表 1-1)。MapReduce 很适合处理那些需要分析整个数据集的问题，以批处理的方式，尤其是 Ad Hoc(自主或即时)分析。RDBMS 适用于点查询和更新(其中，数据集已经被索引以提供低延迟的检索和短时间的少量数据更新。MapReduce 适合数据被一次写入和多次读取的应用，而关系型数据库更适合持续更新的数据集。

表 1-1：关系型数据库和 MapReduce 的比较

	传统关系型数据库	MapReduce
数据大小	GB	PB
访问	交互型和批处理	批处理
更新	多次读写	一次写入多次读取
结构	静态模式	动态模式
集成度	高	低
伸缩性	非线性	线性

MapReduce 和关系型数据库之间的另一个区别是它们操作的数据集中的结构化数据的数量。结构化数据是拥有准确定义的实体化数据，具有诸如 XML 文档或数据库表定义的格式，符合特定的预定义模式。这就是 RDBMS 包括的内容。另一方面，半结构化数据比较宽松，虽然可能有模式，但经常被忽略，所以它只能用作数据结构指南。例如，一张电子表格，其中的结构便是单元格组成的网格，尽管其本身可能保存任何形式的数据。非结构化数据没有什么特别的内部结构，例如纯文本或图像数据。MapReduce 对于非结构化或半结构化

数据非常有效，因为它被设计为在处理时间内解释数据。换句话说：MapReduce 输入的键和值并不是数据固有的属性，它们是由分析数据的人来选择的。

关系型数据往往是规范的，以保持其完整性和删除冗余。规范化为 MapReduce 带来问题，因为它使读取记录成为一个非本地操作，并且 MapReduce 的核心假设之一就是，它可以进行(高速)流的读写。

Web 服务器日志是记录集的一个很好的非规范化例子(例如，客户端主机名每次都以全名来指定，即使同一客户端可能会出现很多次)，这也是 MapReduce 非常适合用于分析各种日志文件的原因之一。

MapReduce 是一种线性的可伸缩的编程模型。程序员编写两个函数-- map 函数和 Reduce 函数-- 每一个都定义一个键/值对集映射到另一个。这些函数无视数据的大小或者它们正在使用的集群的特性，这样它们就可以原封不动地应用到小规模数据集或者大的数据集上。更重要的是，如果放入两倍的数据量，运行的时间会少于两倍。但是如果是两倍大小的集群，一个任务任然只是和原来的一样快。这不是一般的 SQL 查询的效果。

随着时间的推移，关系型数据库和 MapReduce 之间的差异很可能变得模糊。关系型数据库都开始吸收 MapReduce 的一些思路(如 ASTER DATA 的和 GreenPlum 的数据库)，另一方面，基于 MapReduce 的高级查询语言(如 Pig 和 Hive)使 MapReduce 的系统更接近传统的数据库编程人员。

1.3.2 网格计算

高性能计算(High Performance Computing，HPC)和网格计算社区多年来一直在做大规模的数据处理，它们使用的是消息传递接口(Message Passing Interface，MPI)这样的 API。从广义上讲，高性能计算的方法是将作业分配给一个机器集群，这些机器访问共享文件系统，由一个存储区域网络(Storage Area Network，SAN)进行管理。这非常适用于以主计算密集型为主的作业，但当节点需要访问的大数据量(数百 GB 的数据，这是 MapReduce 实际开始"发光"的起点)时，这会成为一个问题，因为网络带宽成为"瓶颈"，所以计算节点闲置下来了。

MapReduce 尝试在计算节点本地存储数据，因此数据访问速度会因为它是本地数据而比较快。 这项"数据本地化"功能，成为 MapReduce 的核心功能并且也是它拥有良好性能的原因之一。意识到网络带宽在数据中心环境是最有价值的资源(到处复制数据会很容易的把网络带宽饱和)之后，MapReduce 便通过显式网络拓扑结构不遗余力地加以保护。请注意，这种安排不会排除 MapReduce 中的高 CPU 使用分析。

MPI 赋予程序员很大的控制，但也要求显式控制数据流机制，需要使用传统的 C 语言的功能模块完成(例如 socket)，以及更高级的算法来进行分析。而 MapReduce 却是在更高层面上完成任务，即程序员从键/值对函数的角度来考虑，同时数据流是隐含的。

在一个大规模分布式计算平台上协调进程是一个很大的挑战。最困难的部分是恰当的处理失效与错误-- 在不知道一个远程进程是否已经失败的时候-- 仍然需要继续整个计算。MapReduce 将程序员从必须考虑失败任务的情况中解放出来，它检测失败的 map 或者 reduce 任务，在健康的机器上重新安排任务。MapReduce 能够做到这一点，因为它是一个无共享的架构，这意味着各个任务之间彼此并不依赖。(这里讲得稍微简单了一些，因为 mapper 的输出是反馈给 reducer 的，但这由 MapReduce 系统控制。在这种情况下，相对于返回失败的 map，应该对返回 reducer 给予更多关注，因为它必须确保它可以检索到必要的 map 输出，如果不行，必须重新运行相关的 map 从而生成必要的这些输出。)因此，从程序员的角度来看，执行任务的顺序是无关紧要的。相比之下，MPI 程序必须显式地管理自己的检查点和恢复机制，从而把更多控制权交给程序员，但这样会加大编程的难度。

MapReduce 听起来似乎是一个相当严格的编程模型，而且在某种意义上看的确如此：我们被限定于键/值对的类型(它们按照指定的方式关联在一起)，mapper 和 reducer 彼此间的协作有限，一个接一个地运行(mapper 传输键/值对给 reducer)。对此，一个很自然的问题是：你是否能用它做点儿有用或普通的事情？

答案是肯定的。MapReduce 作为一个建立搜索索引产品系统，是由 Google 的工程师们开发出来的，因为他们发现自己一遍又一遍地解决相同的问题(MapReduce 的灵感来自传统的函数式编程、分布式计算和数据库社区)，但它后来被应用于其他行业的其他许多应用。我们惊喜地看到许多算法的变体在 MapReduce 中得以表示，从图像图形分析，到基于图表的问题，再到机器学习算法 。它当然不能解决所有问题，但它是一个很普遍的数据处理工具。

第 14 章将介绍一些 Hadoop 应用范例。

1.3.3 志愿计算

人们第一次听说 Hadoop 和 MapReduce 的时候，经常会问："和SETI@home有什么区别？"SETI，全称为 Search for Extra-Terrestrial Intelligence(搜寻外星人)，运行着一个称为SETI@home的项目(http://setiathome.berkeley.edu)。在此项目中，志愿者把自己计算机 CPU 的空闲时间贡献出来分析无线天文望远镜的数据借此寻外星智慧生命信号。SETI@home是最有名的拥有许多志愿者的项目，其他的还有 Great Internet Mersenne Prime Search(搜索大素数)与Folding@home项目(了解蛋白质构成及其与疾病之间的关系)。

志愿计算项目通过将他们试图解决的问题分为几个他们成为工作单元的块来工作，并将它们送到世界各地的电脑上进行分析。例如，SETI@home的工作单元大约是 0.35 MB 的

无线电望远镜数据，并且一个典型的计算机需要数小时或数天来分析。完成分析后，结果发送回服务器，客户端获得的另一项工作单元。作为防止欺骗的预防措施，每个工作单元必须送到三台机器上并且需要有至少两个结果相同才会被接受。

虽然SETI@home在表面上可能类似于 MapReduce(将问题分为独立的块，然后进行并行计算)，但差异还是显著的。SETI@home问题是 CPU 高度密集型的，使其适合运行于世界各地成千上万台计算机上，因为相对于其计算时间而言，传输工作单元的时间微不足道。志愿者捐献的是 CPU 周期，而不是带宽。

MapReduce 被设计为用来运行那些需要数分钟或数小时的作业，这些作业在一个聚集带宽很高的数据中心中可信任的专用硬件设备上运行。相比之下，SETI@home项目是在接入互联网的不可信的计算机上运行，这些计算机的网速不同，而且数据也不在本地。

1.4 Hadoop 发展简史

Hadoop 是 Doug Cutting-- Apache Lucene 创始人-- 开发的使用广泛的文本搜索库。Hadoop 起源于 Apache Nutch，后者是一个开源的网络搜索引擎，本身也是由 Lucene 项目的一部分。

Hadoop 名字的起源

Hadoop 这个名字不是一个缩写，它是一个虚构的名字。该项目的创建者，Doug Cutting 如此解释 Hadoop 的得名："这个名字是我孩子给一头吃饱了的棕黄色大象命名的。我的命名标准就是简短，容易发音和拼写，没有太多的意义，并且不会被用于别处。小孩子是这方面的高手。Googol 就是由小孩命名的。"

Hadoop 及其子项目和后继模块所使用的名字往往也与其功能不相关，经常用一头大象或其他动物主题(例如："Pig")。较小的各个组成部分给与更多描述性(因此也更俗)的名称。这是一个很好的原则，因为它意味着可以大致从其名字猜测其功能，例如，jobtracker 的任务就是跟踪 MapReduce 作业。

从头开始构建一个网络搜索引擎是一个雄心勃勃的目标，不只是要编写一个复杂的、能够抓取和索引网站的软件，还需要面临着没有专有运行团队支持运行它的挑战，因为它有那么多独立部件。同样昂贵的还有：据 Mike Cafarella 和 Doug Cutting 估计，一个支持此 10 亿页的索引需要价值约 50 万美元的硬件投入，每月运行费用还需要 3 万美元。 不过，他们相信这是一个有价值的目标，因为这会开放并最终使搜索引擎算法普及化。

Nutch 项目开始于 2002 年，一个可工作的抓取工具和搜索系统很快浮出水面。但他们意识到，他们的架构将无法扩展到拥有数十亿网页的网络。在 2003 年发表的一篇描述 Google 分布式文件系统(简称 GFS)的论文为他们提供了及时的帮助，文中称 Google 正在使用此文件系统。 GFS 或类似的东西，可以解决他们在网络抓取和索引过程中产生的大量的

文件的存储需求。具体而言，GFS 会省掉管理所花的时间，如管理存储节点。在 2004 年，他们开始写一个开放源码的应用，即 Nutch 的分布式文件系统(NDFS)。

2004 年，Google 发表了论文，向全世界介绍了 MapReduce。 2005 年初，Nutch 的开发者在 Nutch 上有了一个可工作的 MapReduce 应用，到当年年中，所有主要的 Nutch 算法被移植到使用 MapReduce 和 NDFS 来运行。

Nutch 中的 NDFS 和 MapReduce 实现的应用远不只是搜索领域，在 2006 年 2 月，他们从 Nutch 转移出来成为一个独立的 Lucene 子项目，称为 Hadoop。大约在同一时间，Doug Cutting 加入雅虎，Yahoo 提供一个专门的团队和资源将 Hadoop 发展成一个可在网络上运行的系统(见后文的补充材料)。在 2008 年 2 月，雅虎宣布其搜索引擎产品部署在一个拥有 1 万个内核的 Hadoop 集群上。

2008 年 1 月，Hadoop 已成为 Apache 顶级项目，证明它是成功的，是一个多样化、活跃的社区。通过这次机会，Hadoop 成功地被雅虎之外的很多公司应用，如 Last.fm、Facebook 和《纽约时报》。(一些应用在第 14 章的案例研究和 Hadoop 维基有介绍，Hadoop 维基的网址为http://wiki.apache.org/hadoop/PoweredBy。)

有一个良好的宣传范例，《纽约时报》使用亚马逊的 EC2 云计算将 4 TB 的报纸扫描文档压缩，转换为用于 Web 的 PDF 文件。 这个过程历时不到 24 小时，使用 100 台机器运行，如果不结合亚马逊的按小时付费的模式(即允许《纽约时报》在很短的一段时间内访问大量机器)和 Hadoop 易于使用的并行程序设计模型，该项目很可能不会这么快开始启动。

2008 年 4 月，Hadoop 打破世界纪录，成为最快排序 1TB 数据的系统。运行在一个 910 节点的群集，Hadoop 在 209 秒内排序了 1 TB 的数据(还不到三分半钟)，击败了前一年的 297 秒冠军。同年 11 月，谷歌在报告中声称，它的 MapReduce 实现执行 1TB 数据的排序只用了 68 秒。 在 2009 年 5 月，有报道宣称 Yahoo 的团队使用 Hadoop 对 1 TB 的数据进行排序只花了 62 秒时间。

Hadoop@Yahoo！

构建互联网规模的搜索引擎需要大量的数据，因此需要大量的机器来进行处理。Yahoo！Search 包括四个主要组成部分：Crawler，从因特网下载网页；WebMap，构建一个网络地图；Indexer，为最佳页面构建一个反向索引；Runtime(运行时)，回答用户的查询。WebMap 是一幅图，大约包括一万亿条边(每条代表一个网络链接)和一千亿个节点(每个节点代表不同的网址)。创建和分析此类大图需要大量计算机运行若干天。在 2005 年初，WebMap 所用的基础设施名为 Dreadnaught，需要重新设计以适应更多节点的需求。Dreadnaught 成功地从 20 个节点扩展到 600 个，但需要一个完全重新的设计，以进一步扩大。Dreadnaught 与 MapReduce 有许多相似的地方，但灵活性更强，结构更少。具体说来，每一个分段(fragment)，Dreadnaught 作业可以将输出发送到此作业下一阶段中的每一个分

段，但排序是在库函数中完成的。在实际情形中，大多数 WebMap 阶段都是成对存在的，对应于 MapReduce。因此，WebMap 应用并不需要为了适应 MapReduce 而进行大量重构。

Eric Baldeschwieler(Eric14)组建了一个小团队，我们开始设计并原型化一个新的框架(原型为 GFS 和 MapReduce，用 C++语言编写)，打算用它来替换 Dreadnaught。尽管当务之急是我们需要一个 WebMap 新框架，但显然，标准化对于整个 Yahoo! Search 平台至关重要，并且通过使这个框架泛化，足以支持其他用户，我们才能够充分运用对整个平台的投资。

与此同时，我们在关注 Hadoop(当时还是 Nutch 的一部分)及其进展情况。2006 年 1 月，雅虎聘请了 Doug Cutting，一个月后，我们决定放弃我们的原型，转而使用 Hadoop。相较于我们的原型和设计，Hadoop 的优势在于它已经在 20 个节点上实际应用过。这样一来，我们便能在两个月内搭建一个研究集群，并着手帮助真正的客户使用这个新的框架，速度比原来预计的快许多。另一个明显的优点是 Hadoop 已经开源，较容易(虽然远没有那么容易！)从雅虎法务部门获得许可在开源方面进行工作。因此，我们在 2006 年初设立了一个 200 个节点的研究集群，我们将 WebMap 的计划暂时搁置，转而为研究用户支持和发展 Hadoop。

Hadoop 大事记

2004 年-- 最初的版本(现在称为 HDFS 和 MapReduce)由 Doug Cutting 和 Mike Cafarella 开始实施。

2005 年 12 月-- Nutch 移植到新的框架，Hadoop 在 20 个节点上稳定运行。

2006 年 1 月-- Doug Cutting 加入雅虎。

2006 年 2 月-- Apache Hadoop 项目正式启动以支持 MapReduce 和 HDFS 的独立发展。

2006 年 2 月-- 雅虎的网格计算团队采用 Hadoop。

2006 年 4 月-- 标准排序(10 GB 每个节点)在 188 个节点上运行 47.9 个小时。

2006 年 5 月-- 雅虎建立了一个 300 个节点的 Hadoop 研究集群。

2006 年 5 月-- 标准排序在 500 个节点上运行 42 个小时(硬件配置比 4 月的更好)。

06 年 11 月-- 研究集群增加到 600 个节点。

06 年 12 月-- 标准排序在 20 个节点上运行 1.8 个小时，100 个节点 3.3 小时，500 个节点 5.2 小时，900 个节点 7.8 个小时。

07 年 1 月-- 研究集群到达 900 个节点。

07 年 4 月-- 研究集群达到两个 1000 个节点的集群。

08 年 4 月-- 赢得世界最快 1 TB 数据排序在 900 个节点上用时 209 秒。

08 年 10 月-- 研究集群每天装载 10 TB 的数据。

09 年 3 月-- 17 个集群总共 24 000 台机器。

09 年 4 月-- 赢得每分钟排序，59 秒内排序 500 GB(在 1400 个节点上)和 173 分钟内排序 100 TB 数据(在 3400 个节点上)。

1.5 Apache Hadoop 项目

今天，Hadoop 是一个分布式计算基础架构这把"大伞"下的相关子项目的集合。这些项目属于 Apache 软件基金会(http://hadoop.apache.org)，后者为开源软件项目社区提供支持。虽然 Hadoop 最出名的是 MapReduce 及其分布式文件系统(HDFS，从 NDFS 改名而来)，但还有其他子项目提供配套服务，其他子项目提供补充性服务。这些子项目的简要描述如下，其技术栈如图 1-1 所示。

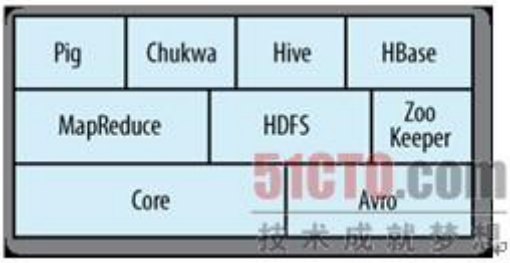


图 1-1：Hadoop 的子项目

Core

一系列分布式文件系统和通用 I/O 的组件和接口(序列化、Java RPC 和持久化数据结构)。

Avro

一种提供高效、跨语言 RPC 的数据序列系统，持久化数据存储。(在本书写作期间，Avro 只是被当作一个新的子项目创建，而且尚未有其他 Hadoop 子项目在使用它。)

MapReduce

分布式数据处理模式和执行环境，运行于大型商用机集群。

HDFS

分布式文件系统，运行于大型商用机集群。

Pig

一种数据流语言和运行环境，用以检索非常大的数据集。Pig 运行在 MapReduce 和 HDFS 的集群上。

Hbase

一个分布式的、列存储数据库。HBase 使用 HDFS 作为底层存储，同时支持 MapReduce 的批量式计算和点查询(随机读取)。

ZooKeeper

一个分布式的、高可用性的协调服务。ZooKeeper 提供分布式锁之类的基本服务用于构建分布式应用。

Hive

分布式数据仓库。Hive 管理 HDFS 中存储的数据，并提供基于 SQL 的查询语言(由运行时引擎翻译成 MapReduce 作业)用以查询数据。

Chukwa

分布式数据收集和分析系统。Chukwa 运行 HDFS 中存储数据的收集器，它使用 MapReduce 来生成报告。(在写作本书期间，Chukwa 刚刚从 Core 中的"contrib"模块分离出来独立成为一个独立的子项目。)

第 2 章 MapReduce 简介

MapReduce 是一种用于数据处理的编程模型。该模型非常简单。同一个程序 Hadoop 可以运行用各种语言编写的 MapReduce 程序。在本章中，我们将看到用 Java，Ruby，Python 和 C++这些不同语言编写的不同版本。最重要的是，MapReduce 程序本质上是并行的，因此可以将大规模的数据分析交给任何一个拥有足够多机器的运营商。MapReduce 的优势在于处理大型数据集，所以下面首先来看一个例子。

2.1 一个气象数据集

在我们这个例子里，要编写一个挖掘气象数据的程序。分布在全球各地的气象传感器每隔一小时便收集当地的气象数据，从而积累了大量的日志数据。它们是适合用 MapReduce 进行分析的最佳候选，因为它们是半结构化且面向记录的数据。

数据的格式

我们将使用 National Climatic Data Center(国家气候数据中心，NCDC，网址为 http://www.ncdc.noaa.gov/)提供的数据。数据是以面向行的 ASCII 格式存储的，每一行便是

一个记录。该格式支持许多气象元素，其中许多数据是可选的或长度可变的。为简单起见，我们将重点讨论基本元素(如气温)，这些数据是始终都有且有固定宽度的。

例 2-1 显示了一个简单的示例行，其中一些重要字段加粗显示。该行已被分成多行以显示出每个字段，在实际文件中，字段被整合成一行且没有任何分隔符。

例 2-1：国家气候数据中心数据记录的格式

```
0057
332130      # USAF weather station identifier
99999       # WBAN weather station identifier
19500101    # observation date
0300        # observation time
4
+51317      # latitude (degrees × 1000)
+028783     # longitude (degrees × 1000)
FM-12
+0171       # elevation (meters)
99999
V020
320         # wind direction (degrees)
1           # quality code
N
0072
1
00450       # sky ceiling height (meters)
1           # quality code
C
N
010000      # visibility distance (meters)
1           # quality code
N
9
-0128       # air temperature (degrees Celsius × 10)
1           # quality code
-0139       # dew point temperature (degrees Celsius × 10)
1           # quality code
10268       # atmospheric pressure (hectopascals × 10)
1           # quality code
```

数据文件按照日期和气象站进行组织。从 1901 年到 2001 年，每一年都有一个目录，每一个目录都包含一个打包文件，文件中的每一个气象站都带有当年的数据。例如，1990 年的前面的数据项如下：

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

因为实际生活中有成千上万个气象台，所以整个数据集由大量较小的文件组成。通常情况下，我们更容易、更有效地处理数量少的大型文件，因此，数据会被预先处理而使每年记录的读数连接到一个单独的文件中。（具体做法请参见附录 C）

2.2 使用 Unix Tools 来分析数据

在全球气温数据中每年记录的最高气温是多少？我们先不用 Hadoop 来回答这一问题，因为答案中需要提供一个性能标准(baseline)和一种检查结果的有效工具。

对于面向行的数据，传统的处理工具是 awk。例 2-2 是一个小的程序脚本，用于计算每年的最高气温。

例 2-2：一个用于从 NCDC 气象记录中找出每年最高气温的程序

```
#!/usr/bin/env bash
for year in all/*
do
  echo -ne 'basename $year .gz'"\t"
  gunzip -c $year | \
    awk '{ temp = substr($0, 88, 5) + 0;
           q = substr($0, 93, 1);
           if (temp !=9999 && q ~ /[01459]/
  && temp > max) max = temp }
          END { print max }'
done
```

该脚本循环遍历压缩文件，首先显示年份，然后使用 awk 处理每个文件。awk 脚本从数据中提取两个字段：气温和质量代码。气温值通过加上一个 0 变成一个整数。接下来，执行测试，从而判断气温值是否有效(值 9999 代表在 NCDC 数据集缺少值)，质量代码显示的读数是有疑问还是根本就是错误的。如果读数是正确的，那么该值将与目前看到的最大值进行比较，如果该值比原先的最大值大，就替换掉目前的最大值。当文件中所有的行都已处理完并打印出最大值后，END 块中的代码才会被执行。

下面是某次运行结果的开始部分：

```
1. %./max_temperature.sh
2. 1901 317
3. 1902 244
4. 1903 289
5. 1904 256
6. 1905 283
7. ...
```

由于源文件中的气温值按比例增加到 10 倍，所以结果 1901 年的最高气温是 31.7°C(在本世纪初几乎没有多少气温读数会被记录下来，所以这是可能的)。为完成对跨越一世纪这么长时间的查找，程序在 EC2 High-CPU Extra Large Instance 机器上一共运行了 42 分钟。

为加快处理，我们需要并行运行部分程序。从理论上讲，这很简单：我们可以通过使用计算机上所有可用的硬件线程来处理在不同线程中的各个年份的数据。但是这之中存在一些问题。

首先，划分成大小相同的作业块通常并不容易或明显。在这种情况下，不同年份的文件，大小差异很大，所以一些线程会比其他线程更早完成。即使它们继续下一步的工作，但是整个运行中占主导地位的还是那些运行时间很长的文件。另一种方法是将输入数据分成固定大小的块，然后把每块分配到各个进程。

其次，独立线程运行结果在合并后，可能还需要进一步的处理。在这种情况下，每年的结果是独立于其他年份，并可能通过连接所有结果和按年份排序这两种方式来合并它们。如果使用固定大小的块这种方法，则此类合并会更紧凑。对于这个例子，某年的数据通常被分割成几个块，每个进行独立处理。我们将最终获得每个数据块的最高气温，所以最后一步是寻找这些每年气温值中的最大值。

最后，我们仍然受限于一台计算机的处理能力。如果手中所有的处理器都使用上都至少需要 20 分钟，那就只能这样了。我们不能使它更快。另外，一些数据集的增长会超出一台计算机的处理能力。当我们开始使用多台计算机时，整个大环境中的许多其他因素将发

挥作用，可能由于协调性和可靠性的问题而出现当机等错误。谁运行整个作业？我们如何处理失败的进程？

因此，尽管并行处理可行，但实际上它非常复杂。使用 Hadoop 之类的框架非常有助于处理这些问题。

2.3 使用 Hadoop 进行数据分析

为了更好地发挥 Hadoop 提供的并行处理机制的优势，我们必须把查询表示成 MapReduce 作业。经过一些本地的小规模测试，我们将能够在机器集群上运 行它。

2.3.1 map 和 reduce

MapReduce 的工作过程分为两个阶段：map 阶段和 reduce 阶段。每个阶段都有键/值对作为输入和输出，并且它们的类型可由程序员选择。程序员还具体定义了两个函数：map 函数和 reduce 函数。

我们在 map 阶段输入的是原始的 NCDC 数据。我们选择的是一种文本输入格式，以便数据集的每一行都会是一个文本值。键是在文件开头部分文本行起始处的偏移量，但我们没有这方面的需要，所以将其忽略。

map 函数很简单。我们使用 map 函数来找出年份和气温，因为我们只对它们有兴趣。在本例中，map 函数只是一个数据准备阶段，通过这种方式来建立数据，使得 reducer 函数能在此基础上进行工作：找出每年的最高气温。map 函数也是很适合去除已损记录的地方：在这里，我们将筛选掉缺失的、不可靠的或错误的气温 数据。

为了全面了解 map 的工作方式，我们思考下面几行示例的输入数据(考虑到页面篇幅，一些未使用的列已被去除，用省略号表示)：

```
1. 0067011990999991950051507004...9999999N9+00001+99999999999...
2. 0043011990999991950051512004...9999999N9+00221+99999999999...
3. 0043011990999991950051518004...9999999N9-00111+99999999999...
4. 0043012650999991949032412004...0500001N9+01111+99999999999...
5. 0043012650999991949032418004...0500001N9+00781+99999999999...
```

这些行以键/值对的方式来表示 map 函数：

```
1. (0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
2. (106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
3. (212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
4. (318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
5. (424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

键是文件中的行偏移量，而这往往是我们在 map 函数中所忽视的。map 函数的功能仅仅提取年份和气温(以粗体显示)，并将其作为输出被发送。(气温值已被解释为整数)

```
1. (1950, 0)
2. (1950, 22)
3. (1950, ?11)
4. (1949, 111)
5. (1949, 78)
```

map 函数的输出先由 MapReduce 框架处理，然后再被发送到 reduce 函数。这一处理过程根据键来对键/值对进行排序和分组。因此，继续我们的示例，reduce 函数会看到如下输入：

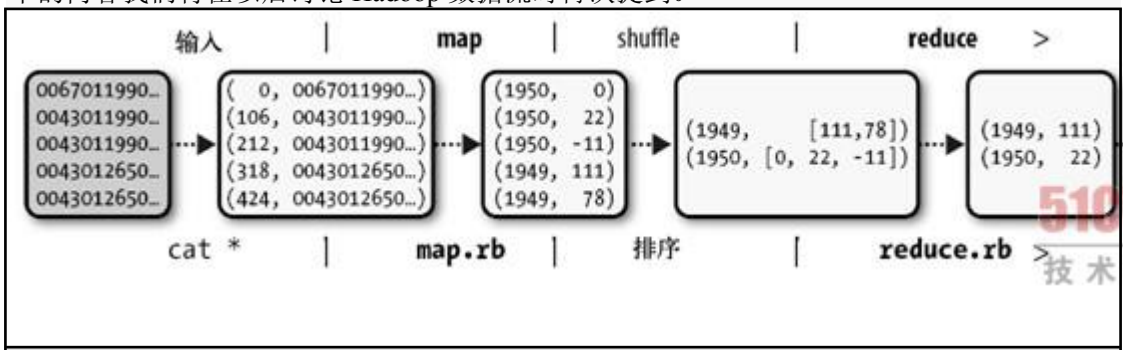
```
1. (1949, [111, 78])
2. (1950, [0, 22, ?11])
```

每年的年份后都有一系列气温读数。所有 reduce 函数现在必须重复这个列表并从中找出最大的读数：

```
1. (1949, 111)
2. (1950, 22)
```

这是最后的输出：全球气温记录中每年的最高气温。

整个数据流如图 2-1 所示。在图的底部是 Unix 的管道，模拟整个 MapReduce 的流程，其中的内容我们将在以后讨论 Hadoop 数据流时再次提到。



（点击查看大图）图 2-1：MapReduce 的逻辑数据流

2.3.2 Java MapReduce（1）

在明白 MapReduce 程序的工作原理之后，下一步就是要用代码来实现它。我们需要三样东西：一个 map 函数、一个 reduce 函数和一些来运行作业的代码。map 函数是由一个 Mapper 接口来实现的，其中声明了一个 map()方法。例 2-3 显示了我们的 map 函数的实现。

例 2-3：最高气温示例的 Mapper 接口

```
1. import java.io.IOException;
2.
3. import org.apache.hadoop.io.IntWritable;
4. import org.apache.hadoop.io.LongWritable;
5. import org.apache.hadoop.io.Text;
6. import org.apache.hadoop.mapred.MapReduceBase;
7. import org.apache.hadoop.mapred.Mapper;
8. import org.apache.hadoop.mapred.OutputCollector;
9. import org.apache.hadoop.mapred.Reporter;
10.
11.public class MaxTemperatureMapper extends MapReduceBase
12.  implements Mapper<LongWritable, Text, Text, IntWritable> {
13.
14.  private static final int MISSING = 9999;
15.
16.  public void map(LongWritable key, Text value,
17.    OutputCollector<Text, IntWritable> output, Reporter reporter)
18.    throws IOException {
19.
20.  String line = value.toString();
21.  String year = line.substring(15, 19);
22.  int airTemperature;
23.  if (line.charAt(87) == '+') { // parseInt
   doesn't like leading plus signs
24.    airTemperature = Integer.parseInt(line.substring(88, 92));
25.  } else {
26.    airTemperature = Integer.parseInt(line.substring(87, 92));
27.  }
28.  String quality = line.substring(92, 93);
29.  if (airTemperature != MISSING && quality.matches("[01459]")) {
30.    output.collect(new Text(year), new IntWritable(airTemperature));
31.  }
32. }
33.}
```

该 Mapper 接口是一个泛型类型，它有 4 个形式参数类型，由它们来指定 map 函数的输入键、输入值、输出键和输出值的类型。就目前的示例来说，输入键是一个长整数偏移

量，输入的值是一行文本，输出的键是年份，输出的值是气温(整数)。Hadoop 规定了自己的一套可用于网络序列优化的基本类型，而不是使用内置的 Java 类型。这些都可以在 org.apache.hadoop.io 包中找到。现在我们使用的是 LongWritable 类型(相当于 Java 的 Long 类型)、Text 类型(相当于 Java 的 String 类型)和 IntWritable 类型(相当于 Java 的 Integer 类型)。

map()方法需要传入一个键和一个值。我们将一个包含 Java 字符串输入行的 Text 值转换成 Java 的 String 类型，然后利用其 substring()方法提取我们感兴趣的列。

map()方法还提供了一个 OutputCollector 实例来写入输出内容。在这种情况下，我们写入年份作为一个 Text 对象(因为我们只使用一个键)，用 IntWritable 类型包装气温值。我们只有在气温显示出来后并且它的质量代码表示的是正确的气温读数时才写入输出记录。

reduce 函数同样在使用 Reducer 时被定义，如例 2-4 所示。

例 2-4：最高气温示例的 Reducer

```
1.  import java.io.IOException;
2.  import java.util.Iterator;
3.
4.  import org.apache.hadoop.io.IntWritable;
5.  import org.apache.hadoop.io.Text;
6.  import org.apache.hadoop.mapred.MapReduceBase;
7.  import org.apache.hadoop.mapred.OutputCollector;
8.  import org.apache.hadoop.mapred.Reducer;
9.  import org.apache.hadoop.mapred.Reporter;
10.
11. public class MaxTemperatureReducer extends MapReduceBase
12.   implements Reducer<Text, IntWritable, Text, IntWritable> {
13.
14.   public void reduce(Text key, Iterator<IntWritable> values,
15.       OutputCollector<Text, IntWritable> output, Reporter reporter)
16.       throws IOException {
17.
18.     int maxValue = Integer.MIN_VALUE;
19.     while (values.hasNext()) {
20.       maxValue = Math.max(maxValue, values.next().get());
21.     }
22.     output.collect(key, new IntWritable(maxValue));
23.   }
```

```
24. }
```

同样，四个形式参数类型用于指定 reduce 函数的输入和输出类型。reduce 函数的输入类型必须与 map 函数的输出类型相匹配：Text 类型和 IntWritable 类型。在这种情况下，reduce 函数的输出类型是 Text 和 IntWritable 这两种类型，前者是年份的类型而后者是最高气温的类型，在这些输入类型之中，我们遍历所有气温，并把每个记录进行比较直到找到一个最高的为止。

第三部分代码运行的是 MapReduce 作业(请参见例 2-5)。

2.3.2 Java MapReduce（2）

例 2-5：在气象数据集中找出最高气温的应用程序

```
1. import java.io.IOException;
2.
3. import org.apache.hadoop.fs.Path;
4. import org.apache.hadoop.io.IntWritable;
5. import org.apache.hadoop.io.Text;
6. import org.apache.hadoop.mapred.FileInputFormat;
7. import org.apache.hadoop.mapred.FileOutputFormat;
8. import org.apache.hadoop.mapred.JobClient;
9. import org.apache.hadoop.mapred.JobConf;
10.
11.public class MaxTemperature {
12.
13.  public static void main(String[] args) throws IOException {
14.    if (args.length != 2) {
15.    System.err.println("Usage: MaxTemperature <input path> <output
16.        path>");
17.    System.exit(-1);
18.    }
19.
20.    JobConf conf = new JobConf(MaxTemperature.class);
21.    conf.setJobName("Max temperature");
22.
23.    FileInputFormat.addInputPath(conf, new Path(args[0]));
24.    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
25.
26.    conf.setMapperClass(MaxTemperatureMapper.class);
27.    conf.setReducerClass(MaxTemperatureReducer.class);
```

```
28.
29.     conf.setOutputKeyClass(Text.class);
30.     conf.setOutputValueClass(IntWritable.class);
31.
32.     JobClient.runJob(conf);
33.   }
34.}
```

JobConf 对象指定了作业的各种参数。它授予你对整个作业如何运行的控制权。当我们在 Hadoop 集群上运行这个作业时，我们把代码打包成一个 JAR 文件(Hadoop 会在集群分发这个包)。我们没有明确指定 JAR 文件的名称，而是在 JobConf 构造函数中传送一个类，Hadoop 会找到这个包含此类的 JAR 文件。

在创建 JobConf 对象后，我们将指定输入和输出的路径。通过调用 FileInputFormat 内的静态方法 addInputPath()来定义输入的路径，它可以是单个文件、目录(本例中，输入的内容组成此目录下所有文件)或文件模式的路径。同时，addInputPath()可被调用多次从而实现使用多路径输入。

输出路径(其中只有一个)是在 FileOutputFormat 内的静态方法 setOutputPath()来指定的。它指定了 reduce 函数输出文件写入的目录。在运行作业前该目录不应该存在，否则 Hadoop 会报错并且拒绝运行任务。这种预防措施是为了防止数据丢失(一个长时间的任务可能非常恼人地被另一个意外覆盖)。

接下来，通过 setMapperClass()和 setReducerClass()这两个方法来指定要使用的 map 和 reduce 类型。

setOutputKeyClass()和 setOutputValueClass()方法控制 map 和 reduce 函数的输出类型，正如本例所示，这两个方法往往是相同的。如果它们不同，那么 map 的输出类型可设置成使用 setMapOutputKeyClass()和 setMapOutputValue-Class()方法。

输入的类型通过输入格式来控制，我们没有设置，因为我们使用的是默认的 TextInputFormat(文本输入格式)。

在设置了定义 map 和 reduce 函数的类之后，运行作业的准备工作就算完成了。JobClient 内的静态方法 runJob()会提交作业并等待它完成，把进展情况写入控制台。

运行测试

写完 MapReduce 作业之后，拿一个小型的数据集进行测试以排除与代码直接有关的问题，这是常规做法。首先，以独立模式安装 Hadoop（详细说明请参见附录 A）。在这种模式下，Hadoop 运行中使用本地带 job runner（作业运行程序）的文件系统。让我们用前面讨论过的五行代码的例子来测试它（考虑到页面，这里已经对输出稍做修改和重新排版）：

```
% export HADOOP_CLASSPATH=build/classes
% hadoop MaxTemperature input/ncdc/sample.txt output
09/04/07 12:34:35 INFO jvm.JvmMetrics:
Initializing JVM Metrics with processName=Job
Tracker, sessionId=
09/04/07 12:34:35 WARN mapred.JobClient:
Use GenericOptionsParser for parsing the
arguments. Applications should implement Tool for the same.
09/04/07 12:34:35 WARN mapred.JobClient:
No job jar file set. User classes may not
be found. See JobConf(Class) or JobConf#setJar(String).
09/04/07 12:34:35 INFO mapred.FileInput
Format: Total input paths to process : 1
09/04/07 12:34:35 INFO mapred.JobClient:
Running job: job_local_0001
09/04/07 12:34:35 INFO mapred.FileInput
Format: Total input paths to process : 1
09/04/07 12:34:35 INFO mapred.MapTask:
numReduceTasks: 1
09/04/07 12:34:35 INFO mapred.MapTask:
io.sort.mb = 100
09/04/07 12:34:35 INFO mapred.MapTask:
data buffer = 79691776/99614720
09/04/07 12:34:35 INFO mapred.MapTask:
record buffer = 262144/327680
09/04/07 12:34:35 INFO mapred.MapTask:
Starting flush of map output
09/04/07 12:34:36 INFO mapred.MapTask:
Finished spill 0
09/04/07 12:34:36 INFO mapred.TaskRunner:
Task:attempt_local_0001_m_000000_0 is
done. And is in the process of commiting
```

```
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
file:/Users/tom/workspace/htdg/input/n
cdc/sample.txt:0+529
09/04/07 12:34:36 INFO mapred.TaskRunner: Task '
attempt_local_0001_m_000000_0' done.
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.Merger: Merging
1 sorted segments
09/04/07 12:34:36 INFO mapred.Merger: Down to
the last merge-pass, with 1 segments
left of total size: 57 bytes
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.TaskRunner:
Task:attempt_local_0001_r_000000_0 is done
. And is in the process of commiting
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.TaskRunner:
Task attempt_local_0001_r_000000_0 is
allowed to commit now
09/04/07 12:34:36 INFO mapred.
FileOutputCommitter: Saved output of task
'attempt_local_0001_r_000000_0' to file:/
Users/tom/workspace/htdg/output
09/04/07 12:34:36 INFO mapred.
LocalJobRunner: reduce > reduce
09/04/07 12:34:36 INFO mapred.TaskRunner:
Task 'attempt_local_0001_r_000000_0' done.
09/04/07 12:34:36 INFO mapred.JobClient:
map 100% reduce 100%
09/04/07 12:34:36 INFO mapred.JobClient:
Job complete: job_local_0001
09/04/07 12:34:36 INFO mapred.JobClient:
Counters: 13
09/04/07 12:34:36 INFO mapred.JobClient:
FileSystemCounters
09/04/07 12:34:36 INFO mapred.JobClient:
FILE_BYTES_READ=27571
```

```
42. 09/04/07 12:34:36 INFO mapred.JobClient:
    FILE_BYTES_WRITTEN=53907
43. 09/04/07 12:34:36 INFO mapred.JobClient:
    Map-Reduce Framework
44. 09/04/07 12:34:36 INFO mapred.JobClient:
    Reduce input groups=2
45. 09/04/07 12:34:36 INFO mapred.JobClient:
    Combine output records=0
46. 09/04/07 12:34:36 INFO mapred.JobClient:
    Map input records=5
47. 09/04/07 12:34:36 INFO mapred.JobClient:
    Reduce shuffle bytes=0
48. 09/04/07 12:34:36 INFO mapred.JobClient:
    Reduce output records=2
49. 09/04/07 12:34:36 INFO mapred.JobClient:
    Spilled Records=10
50. 09/04/07 12:34:36 INFO mapred.JobClient:
    Map output bytes=45
51. 09/04/07 12:34:36 INFO mapred.JobClient:
    Map input bytes=529
52. 09/04/07 12:34:36 INFO mapred.JobClient:
    Combine input records=0
53. 09/04/07 12:34:36 INFO mapred.JobClient:
    Map output records=5
54. 09/04/07 12:34:36 INFO mapred.JobClient:
    Reduce input records=5
```

2.3.2 Java MapReduce（3）

如果 Hadoop 命令是以类名作为第一个参数，它就会启动一个 JVM 来运行这个类。使用命令比直接使用 Java 更方便，因为前者把类的路径(及其依赖关系)加入 Hadoop 的库中，并获得 Hadoop 的配置。要添加应用程序类的路径，我们需要定义一个 HADOOP_CLASSPATH 环境变量，Hadoop 脚本会来执行相关操作。

注意：以本地(独立)模式运行时，本书所有程序希望都以这种方式来设置 HADOOP_CLA-SSPATH。命令必须在示例代码所在的文件夹下被运行。

运行作业所得到的输出提供了一些有用的信息。(无法找到作业 JAR 文件的相关信息是意料之中的，因为我们是在本地模式下没有 JAR 的情况下运行的。在集群上运行时，不会看到此警告。)例如，我们可以看到，这个作业被给予了一个 ID job_local_0001，并且它

运行了一个 map 任务和一个 reduce 任务(使用 attempt_local_0001_m_000000_0 和 attempt_local_0001_r_000000_0 两个 ID)。在调试 MapReduce 作业时，知道作业和任务的 ID 是非常有用的。

输出的最后一部分叫"计数器"(Counter)，显示了在 Hadoop 上运行的每个作业产生的统计信息。这些对检查处理的数据量是否符合预期非常有用。例如，我们可以遵循整个系统中记录的数目：5 个 map 输入产生了 5 个 map 的输出，然后 5 个 reduce 输入产生两个 reduce 输出。

输出被写入 output 目录，其中每个 reducer 包括一个输出文件。作业包含一个 reducer，所以我们只能找到一个文件，名为 part-00000：

```
1. %cat output/part-00000
2. 1949 111
3. 1950 22
```

这个结果和我们之前手动寻找的结果一样。我们可以把前面这个结果解释为在 1949 年的最高气温记录为 11.1℃，而在 1950 年为 2.2℃。

新的 Java Mapreduce API

Hadoop 最新版 Java MapReduce Release 0.20.0 的 API 包括一个全新的 MapReduce Java API，有时也称为"context object"(上下文对象)，旨在使 API 在未来更容易扩展。新的 API 类型上不兼容以前的 API，所以，以前的应用程序需要重写才能使新的 API 发挥其作用 。

新的 API 和旧的 API 之间有下面几个明显的区别。

新的 API 倾向于使用抽象类，而不是接口，因为这更容易扩展。例如，你可以添加一个方法(用默认的实现)到一个抽象类而不需修改类之前的实现方法。在新的 API 中，Mapper 和 Reducer 是抽象类。

新的 API 是在 org.apache.hadoop.mapreduce 包(和子包)中的。之前版本的 API 则是放在 org.apache.hadoop.mapred 中的。

新的 API 广泛使用 context object(上下文对象)，并允许用户代码与 MapReduce 系统进行通信。例如，MapContext 基本上充当着 JobConf 的 OutputCollector 和 Reporter 的角色。

新的 API 同时支持"推"和"拉"式的迭代。在这两个新老 API 中，键/值记录对被推 mapper 中，但除此之外，新的 API 允许把记录从 map()方法中拉出，这也适用于 reducer。"拉"式的一个有用的例子是分批处理记录，而不是一个接一个。

新的 API 统一了配置。旧的 API 有一个特殊的 JobConf 对象用于作业配置，这是一个对于 Hadoop 通常的 Configuration 对象的扩展(用于配置守护进程，请参见 5.1 节)。在新的 API 中，这种区别没有了，所以作业配置通过 Configuration 来完成。

作业控制的执行由 Job 类来负责，而不是 JobClient，它在新的 API 中已经荡然无存。

例 2-6 使用新 API 重写了 MaxTemperature 的代码，不同之处用黑体字突出显示。

例 2-6：使用新的 context object(上下文对象)MapReduce API 在气象数据集中查找最高气温

```
1. public class NewMaxTemperature {
2.   static class NewMaxTemperatureMapper
3.     extends Mapper<LongWritable, Text, Text, IntWritable> {
4.
5.     private static final int MISSING = 9999;
6.
7.     public void map(LongWritable key, Text value, Context context)
8.         throws IOException, InterruptedException {
9.
10.      String line = value.toString();
11.      String year = line.substring(15, 19);
12.      int airTemperature;
13.      if (line.charAt(87) == '+') { // parseInt doesn't like
14.        leading plus signs
15.        airTemperature = Integer.parseInt(line.substring(88, 92));
16.      } else {
17.        airTemperature = Integer.parseInt(line.substring(87, 92));
18.      }
19.      String quality = line.substring(92, 93);
20.      if (airTemperature !=MISSING && quality.matches("[01459]")){
21.      context.write(new Text(year), new
22.          IntWritable(airTemperature));
23.    }
24.  }
25.}
26.
27.static class NewMaxTemperatureReducer
28.  extends Reducer<Text, IntWritable, Text, IntWritable> {
29.
```

```
30.   public void reduce(Text key, Iterable<IntWritable> values,
31.       Context context)
32.       throws IOException, InterruptedException {
33.
34.     int maxValue = Integer.MIN_VALUE;
35.     for (IntWritable value : values) {
36.       maxValue = Math.max(maxValue, value.get());
37.     }
38.     context.write(key, new IntWritable(maxValue));
39.   }
40. }
41.
42. public static void main(String[] args) throws Exception {
43.   if (args.length != 2) {
44.     System.err.println("Usage: NewMaxTemperature <input path>
45.     <output path>");
46.     System.exit(-1);
47.   }
48.
49.   Job job = new Job();
50.   job.setJarByClass(NewMaxTemperature.class);
51.
52.   FileInputFormat.addInputPath(job, new Path(args[0]));
53.   FileOutputFormat.setOutputPath(job, new Path(args[1]));
54.
55.   job.setMapperClass(NewMaxTemperatureMapper.class);
56.   job.setReducerClass(NewMaxTemperatureReducer.class);
57.
58.   job.setOutputKeyClass(Text.class);
59.   job.setOutputValueClass(IntWritable.class);
60.
61.   System.exit(job.waitForCompletion(true) ? 0 : 1);
62.
63.   }
64. }
```

65. **2.4 分布化**

66. 前面展示了 MapReduce 针对小量输入的工作方式，现在是时候整体了解系统并进入大数据流作为输入了。为简单起见，我们的例子到目前为止都使用本地文件系统中的文件。然而，为了分布化，我们需要把数据存储在分布式文件系统中，典型的如 HDFS(详情参见第 3 章)，以允许 Hadoop 把 MapReduce 的计算移到承载部分数据的各台机器。下面我们就来看看这是如何工作的。

67. **2.4.1 数据流（1）**

68. 首先是一些术语的说明。MapReduce 作业(job)是客户端执行的单位：它包括输入数据、MapReduce 程序和配置信息。Hadoop 通过把作业分成若干个小任务(task)来工作，其包括两种类型的任务：map 任务和 reduce 任务。

69. 有两种类型的节点控制着作业执行过程：jobtracker 和多个 tasktracker。jobtracker 通过调度任务在 tasktracker 上运行，来协调所有运行在系统上的作业。Tasktracker 运行任务的同时，把进度报告传送到 jobtracker，jobtracker 则记录着每项任务的整体进展情况。如果其中一个任务失败，jobtracker 可以重新调度任务到另外一个 tasktracker。Hadoop 把输入数据划分成等长的小数据发送到 MapReduce，称为输入分片(input split)或分片。Hadoop 为每个分片(split)创建一个 map 任务，由它来运行用户自定义的 map 函数来分析每个分片中的记录。

70. 拥有许多分片就意味着处理每个分片的时间与处理整个输入的时间相比是比较小的。因此，如果我们并行处理每个分片，且分片是小块的数据，那么处理过程将有一个更好的负载平衡，因为更快的计算机将能够比一台速度较慢的机器在作业过程中处理完比例更多的数据分片。即使是相同的机器，没有处理的或其他同时运行的作业也会使负载平衡得以实现，并且在分片变得更细时，负载平衡质量也会更佳。

71. 另一方面，如果分片太小，那么管理分片的总时间和 map 任务创建的总时间将决定作业的执行的总时间。对于大多数作业，一个理想的分片大小往往是一个 HDFS 块的大小，默认是 64 MB，虽然这可以根据集群进行调整(对于所有新建文件)或在新建每个文件时具体进行指定。

72. map 任务的执行节点和输入数据的存储节点是同一节点，Hadoop 的性能达到最佳。这就是所谓的 data locality optimization(数据局部性优化)。现在我们应该清楚为什么最佳分片的大小与块大小相同：它是最大的可保证存储在单个节点上的数据量。如果分区跨越两个块，那么对于任何一个 HDFS 节点而言，基本不可能同时存储这两数据块，因此此分布的某部分必须通过网络传输到节点，这与使用本地数据运行 map 任务相比，显然效率更低。

73. map 任务把输出写入本地硬盘，而不是 HDFS。这是为什么？因为 map 的输出作为中间输出：而中间输出则被 reduce 任务处理后产生最终的输出，一旦作业完成，map 的输出就可以删除了。因此，把它及其副本存储在 HDFS 中，难免有些小题大做。如果该节点上运行的 map 任务在 map 输出给 reduce 任务处理之前崩溃，那么 Hadoop 将在另一个节点上重新运行 map 任务以再次创建 map 的输出。

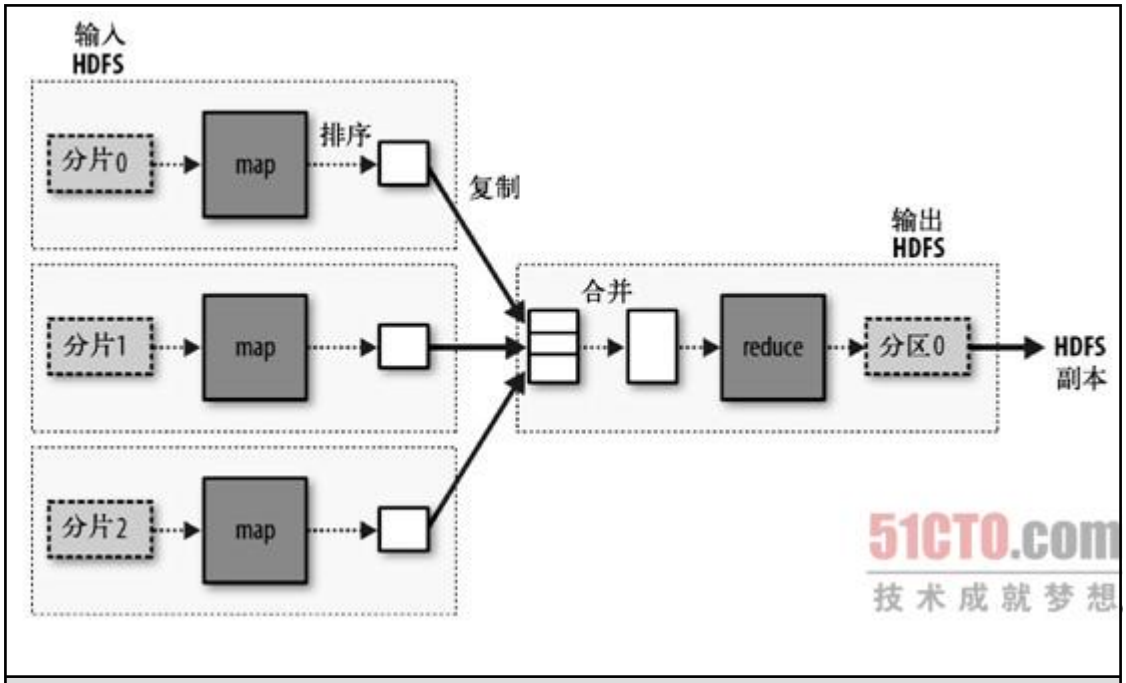
74. reduce 任务并不具备数据本地读取的优势-- 一个单一的 reduce 任务的输入往往来自于所有 mapper 的输出。在本例中，我们有一个单独的 reduce 任务，其输入是由所有 map 任务的输出组成的。因此，有序 map 的输出必须通过网络传输到 reduce 任务运行的节点，并在那里进行合并，然后传递到用户定义的 reduce 函数中。为增加其可靠性，reduce 的输出通常存储在 HDFS 中。如第 3 章所述，对于每个 reduce 输出的 HDFS 块，第一个副本存储在本地节点上，其他副本存储在其他机架节点中。因此，编写 reduce 的输出确实十分占用网络带宽，但是只和正常的 HDFS 写管线的消耗一样。

75. 一个单一的 reduce 任务的整个数据流如图 2-2 所示。虚线框表示节点，虚线箭头表示数据传输到一个节点上，而实线的箭头表示节点之间的数据传输。

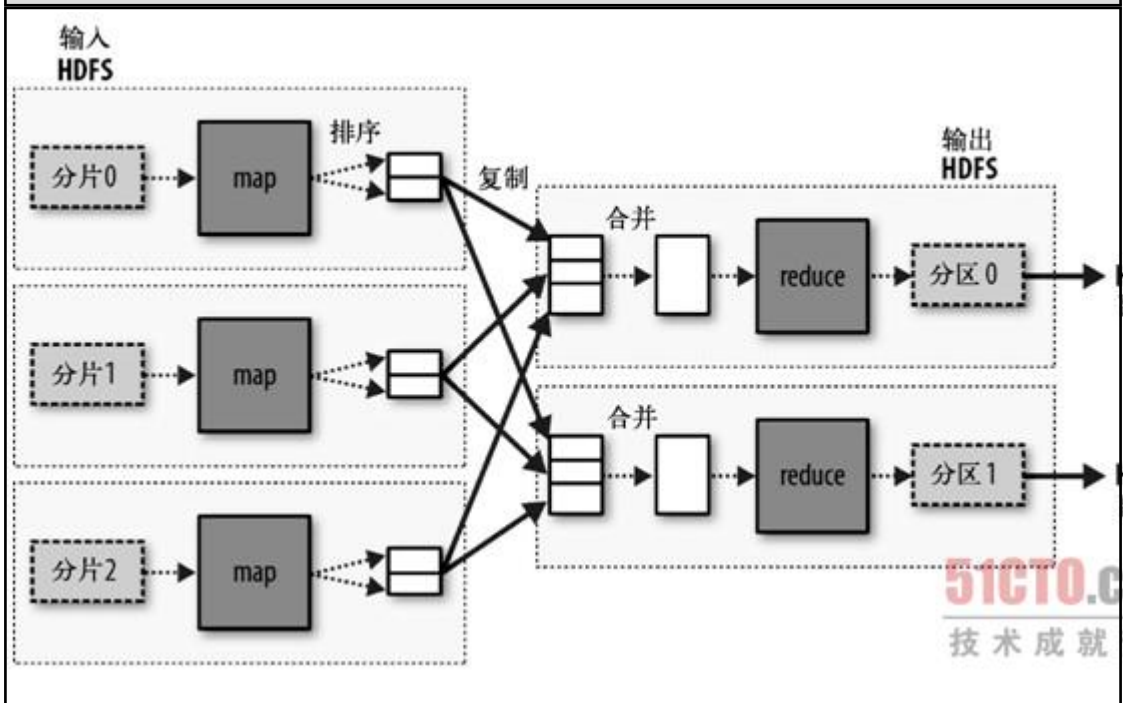
76. reduce 任务的数目并不是由输入的大小来决定，而是单独具体指定的。在第 7 章的 7.1 节中，将介绍如何为一个给定的作业选择 reduce 任务数量。

77. 如果有多个 reducer，map 任务会对其输出进行分区，为每个 reduce 任务创建一个分区(partition)。每个分区包含许多键(及其关联的值)，但每个键的记录都在同一个分区中。分区可以通过用户定义的 partitioner 来控制，但通常是用默认的分区工具，它使用的是 hash 函数来形成"木桶"键/值，这种方法效率很高。

78. 一般情况下，多个 reduce 任务的数据流如图 2-3 所示。此图清楚地表明了 map 和 reduce 任务之间的数据流为什么要称为"shuffle"(洗牌)，因为每个 reduce 任务的输入都由许多 map 任务来提供。shuffle 其实比此图所显示的更复杂，并且调整它可能会对作业的执行时间产生很大的影响，详见 6.4 节。



（点击查看大图）图 2-2：MapReduce 中单一 reduce 任务的数据流图



（点击查看大图）图 2-3：多个 reduce 任务的 MapReduce 数据流

2.4.1 数据流（2）

最后，也有可能不存在 reduce 任务，不需要 shuffle 的时候，这样的情况是可能的，因为处理可以并行进行(第 7 章有几个例子讨论了这个问题)。在这种情况下，唯一的非本地节点数据传输是当 map 任务写入到 HDFS 中(见图 2-4)。

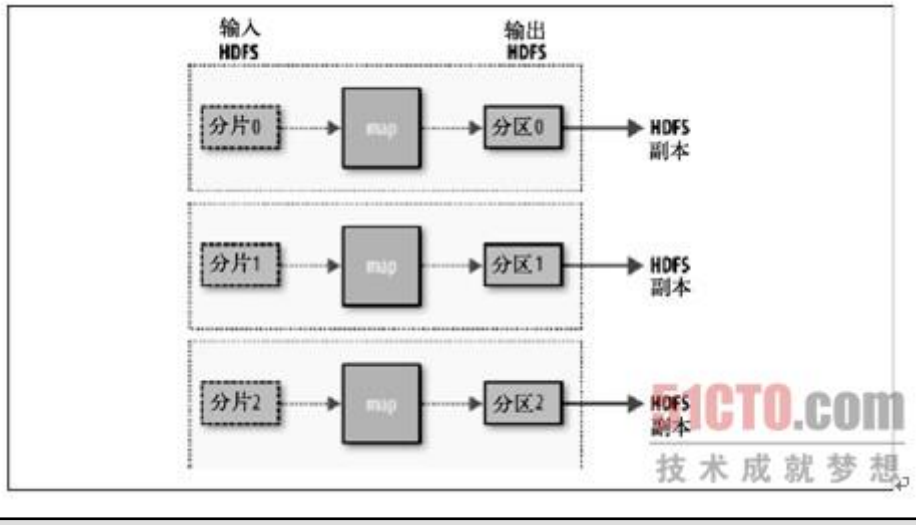


图 2-4：MapReduce 中没有 reduce 任务的数据流

集群的可用带宽限制了 MapReduce 作业的数量，因此 map 和 reduce 任务之间数据传输的代价是最小的。Hadoop 允许用户声明一个 combiner，运行在 map 的输出上-- 该函数的输出作为 reduce 函数的输入。由于 combiner 是一个优化方法，所以 Hadoop 不保证对于某个 map 的输出记录是否调用该方法，调用该方法多少次。换言之，不调用该方法或者调用该方法多次，reducer 的输出结果都一样。

combiner 的规则限制着可用的函数类型。我们将用一个例子来巧妙地加以说明。以前面的最高气温例子为例，1950 年的读数由两个 map 处理(因为它们在不同的分片中)。假设第一个 map 的输出如下：

```
1. (1950, 0)
2. (1950, 20)
3. (1950, 10)
```

第二个 map 的输出如下：

```
1. (1950, 25)
2. (1950, 15)
```

reduce 函数再调用时被传入以下数字：

```
1. (1950, [0, 20, 10, 25, 15])
```

因为 25 是输入值中的最大数，所以输出如下：

```
1. (1950, 25)
```

我们可以用 combiner，像 reduce 函数那样，为每个 map 输出找到最高气温。reduce 函数被调用时将被传入如下数值：

```
1. (1950, [20, 25])
```

然而，reduce 输出的结果和以前一样。更简单地说，我们可以像下面这样，对本例中的气温值进行如下函数调用：

```
1. max(0, 20, 10, 25, 15) = max(max(0, 20, 10),
   max(25, 15)) = max(20, 25) = 25
```

并非所有函数都有此属性。例如，如果我们计算平均气温，便不能用 mean 作为 combiner，因为：

```
1. mean(0, 20, 10, 25, 15) = 14
```

但是：

```
1. mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15
```

combiner 并不能取代 reduce 函数。(为什么呢？reduce 函数仍然需要处理来自不同的 map 给出的相同键记录。)但它可以帮助减少 map 和 reduce 之间的数据传输量，而正因为此，是否在 MapReduce 作业中使用 combiner 是需要慎重考虑的。

2.4.2 具体定义一个 combiner

让我们回到 Java MapReduce 程序，combiner 是用 reducer 接口来定义的，并且该应用程序的实现与 MaxTemperatureReducer 中的函数相同。唯一需要强调的不同是如何在 JobConf 上设置 combiner 类(见例 2-7)。

例 2-7：使用 combiner 高效查找最高气温

```
1. public class MaxTemperatureWithCombiner {
2.
3.   public static void main(String[] args) throws IOException {
4.     if (args.length != 2) {
5.         System.err.println("Usage: MaxTemperatureWithCombiner <input

6.           path> " +"<output path>");
7.         System.exit(-1);
8.     }
9.
10.
11.    JobConf conf = new JobConf(MaxTemperatureWithCombiner.class);
12.    conf.setJobName("Max temperature");
13.
14.    FileInputFormat.addInputPath(conf, new Path(args[0]));
```

```
15.     FileOutputFormat.setOutputPath(conf, new Path(args[1]));
16.
17.     conf.setMapperClass(MaxTemperatureMapper.class);
18.     conf.setCombinerClass(MaxTemperatureReducer.class);
19.     conf.setReducerClass(MaxTemperatureReducer.class);
20.
21.     conf.setOutputKeyClass(Text.class);
22.     conf.setOutputValueClass(IntWritable.class);
23.
24.     JobClient.runJob(conf);
25.   }
26. }
```

2.4.3 运行分布式 MapReduce 作业

同一个程序将在一个完整的数据集中直接运行而不做更改。这是 MapReduce 的优势之一：它扩充数据大小和硬件规模。这里有一个运行结果：在一个 10 节点 EC2 High-CPU Extra Large lnstance，程序只用 6 分钟就运行完了。

我们将在第 5 章分析程序在集群上运行的机制。

2.5 Hadoop 流

Hadoop 提供了一个 API 来运行 MapReduce，并允许你用除 java 以外的语言来编写自己的 map 和 reduce 函数。Hadoop 流使用 Unix 标准流作为 Hadoop 和程序之间的接口，所以可以使用任何语言，只要编写的 MapReduce 程序能够读取标准输入，并写入到标准输出。

流适用于文字处理(尽管 0.21.0 版本也可以处理二进制流)，在文本模式下使用时，它有一个面向行的数据视图。map 的输入数据把标准输入流传输到 map 函数，其中是一行一行的传输，然后再把行写入标准输出。一个 map 输出的键/值对是以单一的制表符分隔的行来写入的。reduce 函数的输入具有相同的格式-- 通过制表符来分隔的键/值对-- 传输标准输入流。reduce 函数从标准输入流读入行，然后为保证结果的有序性用键来排序，最后将结果写入标准输出。

下面使用流来重写按年份寻找最高气温的 MapReduce 程序。

2.5.1 Ruby 语言

例 2-8 中的 map 函数是用 ruby 来写的。

例 2-8：用于查找最高气温的 map 函数(ruby 版)

```
#!/usr/bin/env ruby

STDIN.each_line do |line|
  val = line
  year, temp, q = val[15,4], val[87,5], val[92,1]
  puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

程序通过执行 STDIN(IO 类型的一个全局常量)中的每行，不断从标准输入中反复查找。它把相关的字段从每行中取出，如果气温有效，就过一个 tab 字符\t 把年份和气温通隔开，然后输入标准输出(使用 puts)。

注意：指出流和 Java MapReduce API 之间的差异是十分值得的。 Java 的 API 是主要面向一次处理一个 map 函数的记录。该框架调用 Mapper 的 map()方法来处理输入中的每条记录，然而 map 程序可以决定如何处理输入流，例如，它可以轻松地读取和同一时间处理多行，因为它控制着多行读取。用户的 Java map 实现是压栈记录，但它仍可以考虑处理多行，具体做法是将 mapper 中实例变量中之前的行汇聚在一起。 在这种情况下，需要实现 close()方法，以便了解最后的记录何时被读取，从而完成对最后一组记录行的处理。

由于该脚本只运行在标准输入和输出上，所以只需使用 Unix 管道而不使用 Hadoop 来进行测试，这是不具有说服力的：

```
% cat input/ncdc/sample.txt | src/main/ch02/ruby/max_

temperature_map.rb
1950 +0000
1950 +0022
1950 -0011
1949 +0111
1949 +0078
```

例 2-9 显示的 reduce 函数稍微复杂一些。

例 2-9：用于查找最高气温的 Reduce 函数(Ruby 版本)

```
#!/usr/bin/env ruby

last_key, max_val = nil, 0
STDIN.each_line do |line|
  key, val = line.split("\t")
```

```
6.    if last_key && last_key != key
7.      puts "#{last_key}\t#{max_val}"
8.      last_key, max_val = key, val.to_i
9.    else
10.     last_key, max_val = key, [max_val, val.to_i].max
11.   end
12. end
13. puts "#{last_key}\t#{max_val}" if last_key
```

同样，程序遍历标准输入中的行，但是当我们处理每组键时，我们要存储一些状态。在这种情况下，这个键是气象站的标识符，我们把看到的最后一个键和迄今为止见到的最高气温存储到那个键组中。MapReduce 框架确保键是有序的，以便让我们知道，如果一个键与其上一个键不同，可将其它移入一个新的键组。相比之下，Java API 则提供每一个键组的迭代器，我们只能在流中使用程序来找到键组的边界。

我们从每行中取出键和值，随后，如果刚刚完成一组键(last_key＆last_key =键)，则在将新值更新为最高气温之前，写入键和该组的最高气温，它们由一个制表符来分隔。如果我们没有结束一个组，只能更新当前键的最高气温。

程序的最后一行保证了输入的最后一组键会有一行输出。

现在可以用 UNIX 管线(来模拟整个 MapReduce 的管线，它与图 2-1 显示的 Unix 管线是相同的)。

```
1.  % cat input/ncdc/sample.txt | src/main/ch02
    /ruby/max_temperature_map.rb | \
2.  sort | src/main/ch02/ruby/max_temperature_reduce.rb
3.  1949 111
4.  1950 22
```

输出结果和 Java 程序一样，所以下一步是用 Hadoop 来运行它。

Hadoop 命令不支持 Streaming 函数，因此，我们需要指定 JAR 文件流与 jar 选项。流程序的选项指定了输入和输出路径选项，map 和 reudce 脚本，如下所示：

```
1.  % hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-
2.    streaming.jar \
3.    -input input/ncdc/sample.txt \
4.    -output output \
5.    -mapper src/main/ch02/ruby/max_temperature_map.rb \
6.    -reducer src/main/ch02/ruby/max_temperature_reduce.rb
```

在一个集群上运行一个庞大的数据集时，要使用-combiner 选项来设置 combiner。

从 0.21.0 版开始，combiner 可以是任何一个流命令。对于早期版本的 combiner，只能用 Java 来编写，所以作为变通的方法，一般都在 mapper 中手动合并，从而避开 Java 语言。在这种情况下，我们可以把 mapper 改成管线：

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-
  streaming.jar \
  -input input/ncdc/all \
  -output output \
  -mapper "ch02/ruby/max_temperature_map.rb | sort |
   ch02/ruby/max_temperature_reduce.rb" \
  -reducer src/main/ch02/ruby/max_temperature_reduce.rb \
  -file src/main/ch02/ruby/max_temperature_map.rb \
  -file src/main/ch02/ruby/max_temperature_reduce.rb
```

还要注意-file 的使用，在集群上运行流程序把脚本传输到集群上时，可使用此选项。

2.5.2 Python

Hadoop 流支持任何可以从标准输入读取和写入到标准输出中的编程语言，因此对于更熟悉 Python 的读者，下面提供了同一个例子的 Python 版本 。map 脚本参见例 2-10，reduce 脚本参见例 2-11。

例 2-10：用于查找最高气温的 Map 函数(Python 版本)

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  val = line.strip()
   (year, temp, q) = (val[15:19], val[87:92], val[92:93])
  if (temp != "+9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

例 2-11 用于查找最高气温的 Reduce 函数(Python 版本)

```
#!/usr/bin/env python

import sys

```

```
(last_key, max_val) = (None, 0)
for line in sys.stdin:
  (key, val) = line.strip().split("\t")
  if last_key and last_key != key:
    print "%s\t%s" % (last_key, max_val)
    (last_key, max_val) = (key, int(val))
  else:
    (last_key, max_val) = (key, max(max_val, int(val)))

if last_key:
  print "%s\t%s" % (last_key, max_val)
```

我们可以测试程序且用在 Ruby 中用过的相同的方法来运行作业。比如，运行一个测试：

```
% cat input/ncdc/sample.txt |

src/main/ch02/python/max_temperature_map.py | \
sort | src/main/ch02/python/max_temperature_reduce.py
1949 111
1950 22
```

2.6 Hadoop 管道

Hadoop 管道是 Hadoop MapReduce 的 C++接口的代称。与流不同，流使用标准输入和输出让 map 和 reduce 节点之间相互交流，管道使用 sockets 作为 tasktracker 与 C++编写的 map 或者 reduce 函数的进程之间的通道。JNI 未被使用。

我们将用 C++重写贯穿本章的示例，然后，我们将看到如何使用管道来运行它。例 2-12 显示了用 C++语言编写的 map 函数和 reduce 函数的源代码。

例 2-12：用 C++语言编写的最高气温程序

```
#include <algorithm>
#include <limits>
#include <string>

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

class MaxTemperatureMapper : public HadoopPipes::Mapper {
```

```
10. public:
11.   MaxTemperatureMapper(HadoopPipes::TaskContext& context) {
12.   }
13.   void map(HadoopPipes::MapContext& context) {
14.     std::string line = context.getInputValue();
15.     std::string year = line.substr(15, 4);
16.     std::string airTemperature = line.substr(87, 5);
17.     std::string q = line.substr(92, 1);
18.     if (airTemperature != "+9999" &&
19.         (q == "0" || q == "1" || q == "4" || q == "5" || q == "9")) {
20.       context.emit(year, airTemperature);
21.     }
22.   }
23. };
24.
25.
26. class MapTemperatureReducer : public HadoopPipes::Reducer {
27. public:
28.   MapTemperatureReducer(HadoopPipes::TaskContext& context) {
29.   }
30.   void reduce(HadoopPipes::ReduceContext& context) {
31.     int maxValue = INT_MIN;
32.     while (context.nextValue()) {
33.       maxValue = std::max(maxValue,
34.         HadoopUtils::toInt(context.getInputValue()));
35.     }
36.     context.emit(context.getInputKey(),
37.       HadoopUtils::toString(maxValue));
38.   }
39. };
40.
41. int main(int argc, char *argv[]) {
42.   return HadoopPipes::runTask(HadoopPipes::TemplateFactory
43.                         <MaxTemperatureMapper,
    MapTemperatureReducer>());
44. }
```

此应用程序连接到 Hadoop C++库，后者是一个用于与 tasktracker 子进程进行通信的轻量级的封装器。通过扩展在 HadoopPipes 命名空间的 Mapper 和 Reducer 类且提供 map()和

reduce()方法的实现，我们便可定义 map 和 reduce 函数。这些方法采用了一个上下文对象(MapContext 类型或 ReduceContext 类型)，后者提供读取输入和写入输出，通过 JobConf 类来访问作业配置信息。本例中的处理过程非常类似于 Java 的处理方式。

与 Java 接口不同，C++接口中的键和值是字节缓冲，表示为标准模板库(Standard Template Library，STL)的字符串。这使接口变得更简单，尽管它把更重的负担留给了应用程序的开发人员，因为开发人员必须将字符串 convert to and from 表示 to 和 from 两个逆操作。开发人员必须在字符串及其他类型之间进行转换。这一点在 MapTemperatureReducer 中十分明显，其中，我们必须把输入值转换为整数的输入值(使用 HadoopUtils 中的便利方法)，在最大值被写出之前将其转换为字符串。在某些情况下，我们可以省略这个转化，如在 MaxTemperatureMapper 中，它的 airTemperature 值从来不用转换为整数，因为它在 map()方法中从来不会被当作数字来处理。

main()方法是应用程序的入口点。它调用 HadoopPipes::runTask，连接到从 Mapper 或 Reducer 连接到 Java 的父进程和 marshals 数据。runTask()方法被传入一个 Factory 参数，使其可以创建 Mapper 或 Reducer 的实例。它创建的其中一个将受套接字连接中 Java 父进程控制。我们可以用重载模板 factory 方法来设置一个 combiner(combiner)、partitioner(partitioner)、记录读取函数(record reader)或记录写入函数(record writer)。

编译运行

现在我们可以用 Makerfile 编译连接例 2-13 的程序。

例 2-13：C++版本的 MapReduce 程序的 Makefile

```
1. CC = g++
2. CPPFLAGS = -m32 -I$(HADOOP_INSTALL)/c++/$(PLATFORM)/include
3.
4. max_temperature: max_temperature.cpp
5.      $(CC) $(CPPFLAGS) $< -Wall
6.      -L$(HADOOP_INSTALL)/c++/$(PLATFORM)/lib -lhadooppipes \
7.      -lhadooputils -lpthread -g -O2 -o $@
```

在 Makefile 中应当设置许多环境变量。除了 HADOOP_INSTALL(如果遵循附录 A 的安装说明，应该已经设置好)，还需要定义平台，指定操作系统、体系结构和数据模型(例如，32 位或 64 位)。我在 32 位 Linux 系统的机器编译运行了如下内容：

```
1. % export PLATFORM=Linux-i386-32
2. % make
```

在成功完成之前，当前目录中便有 max_temperature 可执行文件。

要运行管道作业，我们需要在伪分布式(pseudo_distrinuted)模式下(其中所有守护进程运行在本地计算机上)运行 Hadoop，其中的安装步骤参见附录 A。管道不在独立模式(本地运行)中运行，因为它依赖于 Hadoop 的分布式缓存机制，仅在 HDFS 运行时才运行。

Hadoop 守护进程开始运行后，第一步是把可执行文件复制到 HDFS，以便它们启动 map 和 reduce 任务时，它能够被 tasktracker 取出：

```
1. % hadoop fs -put max_temperature bin/max_temperature
```

示例数据也需要从本地文件系统复制到 HDFS：

```
1. % hadoop fs -put input/ncdc/sample.txt sample.txt
```

现在可以运行这个作业。为了使其运行，我们用 Hadoop 管道命令，使用-program 参数来传递在 HDFS 中可执行文件的 URI。

```
1. % hadoop pipes\
2.    -D hadoop.pipes.java.recordreader\
3.    -D hadoop.pipes.java.recordwriter\
4.    inpit sample.txt\
5.    output output
6.    program bin/max_temperature
```

我们使用-D 选项来指定两个属性：hadoop.pipes.java.recordreader 和 hadoop.pipes.java.recordwriter，这两个属性都被设置为 true，表示我们并没有指定一个 C+++记录读取函数或记录写入函数，但我们要使用默认的 Java 设置(用来设置文本输入和输出)。管道还允许你设置一个 Java mapper，reducer，combiner 或 partitioner。事实上，在任何一个作业中，都可以混合使用 Java 类或 C++类。

结果和用其他语言编写的程序所得的结果一样。

第 3 章 Hadoop 分布式文件系统

当数据集超过一个单独的物理计算机的存储能力时，便有必要将它分布到多个独立的计算机。管理着跨计算机网络存储的文件系统称为分布式文件系统。因为它们是基于网络的，所有网络编程的复杂性都会随之而来，所以分布式文件系统比普通磁盘文件系统更复杂。举例来说，使这个文件系统能容忍节点故障而不损失数据就是一个极大的挑战。

Hadoop 有一个被称为 HDFS 的分布式系统，全称为 Hadoop Distributed Filesystem。(有时可能简称为 DFS，在非正式情况或是文档和配置中，其实是一样的。) HDFS 是 Hadoop 的旗舰级文件系统，也是本章的重点，但 Hadoop 实际上有一个综合性的文件系统抽象，因而接下来我们将看看 Hadoop 如何与其他文件系统集成(如本地文件系统或 Amazon S3)。

3.1 HDFS 的设计

HDFS 是为以流式数据访问模式存储超大文件而设计的文件系统，在商用硬件的集群上运行。让我们仔细看看下面的明。

超大文件

"超大文件"在这里指几百 MB，几百 GB 甚至几百 TB 大小的文件。目前已经有 Hadoop 集群存储 PB(petabytes)级的数据了。

流式数据访问

HDFS 建立在这样一个思想上：一次写入、多次读取模式是最高效的。一个数据集通常由数据源生成或复制，接着在此基础上进行各种各样的分析。每个分析至少都会涉及数据集中的大部分数据 (甚至全部)，因此读取整个数据集的时间比读取第一条记录的延迟更为重要。

商用硬件

Hadoop 不需要运行在昂贵并且高可靠性的硬件上。它被设计运行在商用硬件(在各种零售店都能买到的普通硬件)的集群上，因此至少对于大的集群来说，节点故障的几率还是较高的。HDFS 在面对这种故障时，被设计为能够继续运行而让用户察觉不到明显的中断。

同时，那些并不适合 HDFS 的应用也是值得研究的。在目前，HDFS 还不太适用于某些领域，不过日后可能会有所改进。

低延迟数据访问

需要低延迟访问数据在毫秒范围内的应用并不适合 HDFS。HDFS 是为达到高数据吞吐量而优化的，这有可能会以延迟为代价。目前，对于低延迟访问，HBase(参见第 12 章)是更好的选择。

大量的小文件

名称节点(namenode)存储着文件系统的元数据，因此文件数量的限制也由名称节点的内存量决定。根据经验，每个文件，索引目录以及块占大约 150 个字节。因此，举例来说，如果有一百万个文件，每个文件占一个块，就至少需要 300 MB 的内存。虽然存储上百万的文件是可行的，十亿或更多的文件就超出目前硬件的能力了。

多用户写入，任意修改文件

HDFS 中的文件只有一个写入者，而且写操作总是在文件的末尾。它不支持多个写入者，或是在文件的任意位置修改。(可能在以后这些会被支持，但它们也相对不那么高效。)

3.2 HDFS 的概念

3.2.1 块

一个磁盘有它的块大小，代表着它能够读写的最小数据量。文件系统通过处理大小为一个磁盘块大小的整数倍数的数据块来运作这个磁盘。文件系统块一般为几千字节，而磁盘块一般为 512 个字节。这些信息，对于仅仅在一个文件上读或写任意长度的文件系统用户来说是透明的。但是，有些工具会维护文件系统，如 df 和 fsck， 它们都在系统块级上操作。

HDFS 也有块的概念，不过是更大的单元，默认为 64 MB。与单一磁盘上的文件系统相似，HDFS 上的文件也被分为以块为大小的分块，作为单独的单元存储。但与其不同的是，HDFS 中小于一个块大小的文件不会占据整个块的空间。如果没有特殊指出，"块"在本书中就指代 HDFS 中的块。

为何 HDFS 中的一个块那么大？

HDFS 的块比磁盘的块大，目的是为了减小寻址开销。通过让一个块足够大，从磁盘转移数据的时间能够远远大于定位这个块开始端的时间。因此，传送一个由多个块组成的文件的时间就取决于磁盘传输送率。

我们来做一个速算，如果寻址时间在 10 毫秒左右，传输速率是 100 兆/秒，为了使寻址时间为传输时间的 1%，我们需要 100 MB 左右的块大小。而默认的大小实际为 64 MB，尽管很多 HDFS 设置使用 128 MB 的块。这一数字将在以后随着新一代磁盘驱动带来的传输速度加快而继续调整。

当然这种假定不应该如此夸张。MapReduce 过程中的 map 任务通常是在一个时间内运行操作一个块，因此如果任务数过于少(少于集群上的节点数量)，作业的运行速度显然就比预期的慢。

在分布式文件系统中使用抽象块会带来很多好处。第一个最明显的好处是，一个文件可以大于网络中任意一个磁盘的容量。文件的分块(block，后文有些地方也简称为"块")不需要存储在同一个磁盘上，因此它们可以利用集群上的任意一个磁盘。其实，虽然不常见，但对于 HDFS 集群而言，也可以存储一个其分块占满集群中所有磁盘的文件。

第二个好处是，使用块抽象单元而不是文件会简化存储子系统。简单化是所有系统的追求，但对于故障种类繁多的分布式系统来说尤为重要的。存储子系统控制的是块，简化了存储管理。(因为块的大小固定，计算一个磁盘能存多少块就相对容易)，也消除了对元数据的顾虑(块只是一部分存储的数据-而文件的元数据，如许可信息，不需要与块一同存储，这样一来，其他系统就可以正交地管理元数据。)

不仅如此，块很适合于为提供容错和实用性而做的复制操作。为了应对损坏的块以及磁盘或机器的故障，每个块都在少数其他分散的机器(一般为 3 个)进行复制。如果一个块损坏了，系统会在其他地方读取另一个副本，而这个过程是对用户透明的。一个因损坏或机器故障而丢失的块会从其他候选地点复制到正常运行的机器上，以保证副本的数量回到正常水平。(参见第 4 章的"数据的完整性"小节，进一步了解如何应对数据损坏。)同样，有些应用程序可能选择为热门的文件块设置更高的副本数量以提高集群的读取负载量。

与磁盘文件系统相似，HDFS 中 fsck 指令会显示块的信息。例如，执行以下命令将列出文件系统中组成各个文件的块(参见第 10 章的"文件系统查看(fsck)"小节)：

```
1. % hadoop fsck / -files -blocks
```

3.2.2 名称节点与数据节点

HDFS 集群有两种节点，以管理者-工作者的模式运行，即一个名称节点(管理者)和多个数据节点(工作者)。名称节点管理文件系统的命名空间。它维护着这个文件系统树及这个树内所有的文件和索引目录。这些信息以两种形式将文件永久保存在本地磁盘上：命名空间镜像和编辑日志。名称节点也记录着每个文件的每个块所在的数据节点，但它并不永久保存块的位置，因为这些信息会在系统启动时由数据节点重建。

客户端代表用户通过与名称节点和数据节点交互来访问整个文件系统。客户端提供一个类似 POSIX(可移植操作系统界面)的文件系统接口，因此用户在编程时并不需要知道名称节点和数据节点及其功能。

数据节点是文件系统的工作者。它们存储并提供定位块的服务(被用户或名称节点调用时)，并且定时的向名称节点发送它们存储的块的列表。

没有名称节点，文件系统将无法使用。事实上，如果运行名称节点的机器被毁坏了，文件系统上所有的文件都会丢失，因为我们无法知道如何通过数据节点上的块来重建文件。因此，名称节点能够经受故障是非常重要的，Hadoop 提供了两种机制来确保这一点。

第一种机制就是复制那些组成文件系统元数据持久状态的文件。Hadoop 可以通过配置使名称节点在多个文件系统上写入其持久化状态。这些写操作是具同步性和原子性的。一般的配置选择是，在本地磁盘上写入的同时，写入一个远程 NFS 挂载(mount)。

另一种可行的方法是运行一个二级名称节点，虽然它不能作为名称节点使用。这个二级名称节点的重要作用就是定期的通过编辑日志合并命名空间镜像，以防止编辑日志过大。这个二级名称节点一般在其他单独的物理计算机上运行，因为它也需要占用大量 CPU 和内存来执行合并操作。它会保存合并后的命名空间镜像的副本，在名称节点失效后就可以使用。

但是，二级名称节点的状态是比主节点滞后的，所以主节点的数据若全部丢失，损失仍在所难免。在这种情况下，一般把存在 NFS 上的主名称节点元数据复制到二级名称节点上并将其作为新的主名称节点运行。

详情请参见第 10 章的"文件系统镜像与编辑日志"小节。

3.3 命令行接口

现在我们将通过命令行与 HDFS 交互。HDFS 还有很多其他接口，但命令行是最简单的，同时也是许多开发者最熟悉的。

我们将在一台机器上运行 HDFS，所以请先参照附录 A 中在伪分布模式下设置 Hadoop 的说明。稍后将介绍如何在集群上运行 HDFS 从而为我们提供伸缩性与容错性。

在我们设置伪分布配置时，有两个属性需要进一步解释。首先是 fs.default.name，设置为 hdfs://localhost/, 用来为 Hadoop 设置默认文件系统。文件系统是由 URI 指定的，这里我们已使用了一个 hdfs URI 来配置 HDFS 为 Hadoop 的默认文件系统。HDFS 的守护程序将通过这个属性来决定 HDFS 名称节点的宿主机和端口。我们将在 localhost 上运行，默认端口为 8020。这样一来，HDFS 用户将通过这个属性得知名称节点在哪里运行以便于连接到它。

第二个属性 dfs.replication，我们设为 1，这样一来，HDFS 就不会按默认设置将文件系统块复制 3 份。在单独一个数据节点上运行时，HDFS 无法将块复制到 3 个数据节点上，所以会持续警告块的副本不够。此设置可以解决这个问题。

基本文件系统操作

文件系统已经就绪，我们可以执行所有其他文件系统都有的操作，例如，读取文件，创建目录，移动文件，删除数据，列出索引目录，等等。输入 hadoop fs -help 命令即可看到所有命令详细的帮助文件。

首先将本地文件系统的一个文件复制到 HDFS：

```
1. % hadoopfs -copyFromLocal input/docs/quangle.
   txt hdfs://localhost/user/tom/quangle.txt
```

该命令调用 Hadoop 文件系统的 shell 命令 fs，提供一系列的子命令。在这里，我们执行的是-copyFromLocal。本地文件 quangle.txt 被复制到运行在 localhost 上的 HDFS 实体中的/user/tom/quangle.txt 文件。其实我们可以省略 URI 的格式与主机而选择默认设置，即省略 hdfs://localhost,就像 core-site.xml 中指定的那样。

```
1. % hadoop fs -copyFromLocal input/docs/quangle.
   txt /user/tom/quangle.txt
```

也可以使用相对路径，并将文件复制到 home 目录，即/user/tom：

```
1. % hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt
```

我们把文件复制回本地文件系统，看看是否一样：

```
1.      % hadoop fs -copyToLocal quangle.txt quangle.copy.txt
2. % md5 input/docs/quangle.txt quangle.copy.txt
3. MD5 (input/docs/quangle.txt) = a16f231da6b05e2ba7a339320e7dacd9
4. MD5 (quangle.copy.txt) = a16f231da6b05e2ba7a339320e7dacd9
```

MD5 分析结果是一样的，表明这个文件在 HDFS 之旅中得以幸存并完整。

最后，我们看一下 HDFS 文件列表。我们创建一个目录来看看它在列表中如何显示：

```
1. % hadoop fs -mkdir books
2. % hadoop fs -ls .
3. Found 2 items
4. drwxr-xr-x   - tom supergroup          0
   2009-04-02 22:41 /user/tom/books
5. -rw-r--r--   1 tom supergroup        118
   2009-04-02 22:29 /user/tom/quangle.txt
```

返回的信息结果与 Unix 命令 ls -l 的输出非常相似，仅有细微差别。第一列显示的是文件格式。第二列是这个文件的副本数(这在 Unix 文件系统是没有的)。由于我们设置的默认副本数在网站范围内为 1，所以这里显示的也都是 1。这一列的开头目录栏是空的，因为副本的概念并没有应用-- 目录是作为元数据并存在名称节点中的，而非数据节点。第三列和第四列显示文件的所属用户和组别。第五列是文件的大小，以字节显示，目录大小为 0。第六列和第七列是文件的最后修改日期与时间。最后的第八列是文件或目录的绝对路径。

HDFS 中的文件许可

HDFS 对于文件及目录有与 POSIX 非常相似的许可模式。

共有三种形式的许可：读取许可(r)、写入许可(w)和执行许可(x)。读取文件或列出目录内容时需要读取许可。写入一个文件，或是在一个目录上创建或删除文件或目录，需要写入许可。对于文件而言执行许可可以忽略因为 HDFS 中不能执行文件(与 POSIX 不同)，但在访问一个目录的子项时是需要的。

每个文件和目录都有一个所属用户、所属组别和模式。这个模式是由所属用户的许可、组内其他成员的许可及其他用户的许可组成。

客户端的标识是通过它正在运行的进程的 username(名称)和 groups(组别)来确定的。由于客户端是远程的，任何人都可以简单地在远程系统上创建一个账户来进行访问。因此，许可只能在一个合作的团体中的用户中使用，作为共享文件系统资源和防止数据意外损失的机制，而不能在一个敌意的环境中保护资源。但是，除去这些缺点，为防止用户或自动工具及程序意外修改或删除文件系统的重要部分，使用许可还是值得的(这也是默认的配置，参见 dfs.permissions 属性)。

如果启用了许可检查，所属用户许可与组别许可都会被检查，以确认用户的用户名与所属用户许可是否相同，确认他是否属于此用户组的成员；若不符，则检查其他许可。

这里有一个超级用户的概念，超级用户是名称节点进程的标识。对于超级用户，系统不会执行任何许可检查。

3.4 Hadoop 文件系统（1）

Hadoop 有一个抽象的文件系统概念，HDFS 只是其中的一个实现。Java 抽象类 org.apache.hadoop.fs.FileSystem 展示了 Hadoop 的一个文件系统，而且有几个具体实现，如表 3-1 所示。

文件系统	URI 方案	Java 实现(全部在 org.apache.hadoop)	描述
Local	file	fs.LocalFileSystem	针对有客户端校验和的本地连接磁盘使用的文件系统。针对没有校验和的本地文件系统使用 RawLocalFileSystem。详情参见第 4 章
HDFS	hdfs	hdfs.Distributed-FileSystem	Hadoop 的分布式文件系统。HDFS 被设计为结合使用 Map-Reduce 实现高效工作
HFTP	hftp	hdfs.HftpFileSystem	一个在 HTTP 上提供对 HDFS 只读访问的文件系统(虽然其名称为 HFTP，但它与 FTP 无关)。通常与 distcp 结合使用(参见第 3 章)，在运行不同版本 HDFS 的集群间复制数据
HSFTP	hsft	hdfs.Hsftp-	在 HTTPS 上提供对

	p	FileSystem	HDFS 只读访问的文件系统(同上，与 FTP 无关)
HAR	har	fs.HarFileSystem	一个构建在其他文件系统上来存档文件的文件系统。Hadoop 存档一般在 HDFS 中的文件存档时使用，以减少名称节点内存的使用
KFS(Cloud-Store)	kfs	fs.kfs.Kosmos-FleSystem	cloudstore(其前身是 Kosmos 文件系统)是相似于 HDFS 或是 Google 的 GFS 的文件系统，用 C++编写。详情可参见 *http://kosmosfs.sourceforge.net/*
FTP	ftp	fs.ftp.FTP-FileSystem	由 FTP 服务器支持的文件系统
S3(本地)	s3n	fs.s3native.Native-S3FileSystem.	由 Amazon S3 支持的文件系统。可参见 *http://wiki.apache.org/hadoop/AmazonS3*
S3(基于块)	s3	fs.s3.S3FileSystem	由 Amazon S3 支持的文件系统，以块格式存储文件(与 HDFS 很相似)来解决 S3 的 5 GB 文件大小限制

Hadoop 提供了许多文件系统的接口，它一般使用 URI 方案来选取合适的文件系统实例交互。举例来说，我们在前一小节中研究的文件系统 shell 可以操作所有的 Hadoop 文件系统。列出本地文件系统根目录下的文件，输入以下命令：

```
1. % hadoop fs -ls file:///
```

尽管运行那些可访问任何文件系统的 MapReduce 程序是可行的(有时也很方便)，但在处理大量数据时，仍然需要选择一个有最优本地数据的分布式文件系统，如 HDFS 或者 KFS(参见第 1 章)。

3.4 Hadoop 文件系统（2）

接口

Hadoop 是用 Java 编写的，所有 Hadoop 文件系统间的相互作用都是由 Java API 调解的。举个例子，文件系统的 shell 就是一个 Java 应用，它使用 Java 文件系统类来提供文件系统操作。这些接口在 HDFS 中被广泛应用，因为 Hadoop 中的其他文件系统一般都有访问基本文件系统的工具(FTP 的 FTP 客户，S3 的 S3 工具等)，但它们大多数都能和任意一个 Hadoop 文件系统协作。

Thrift

因为 Hadoop 的文件系统接口是 Java API，所以其他非 Java 应用访问 Hadoop 文件系统会比较麻烦。在"Thriftfs"分类单元中的 Thrift API 通过将 Hadoop 文件系统展示为一个 Apache Thrift 服务来弥补这个不足，使得任何有 Thrift 绑定的语言都能轻松地与 Hadoop 文件系统互动，如 HDFS。

使用 Thrift API，需要运行提供 Thrift 服务的 Java 服务器，以代理的方式访问 Hadoop 文件系统。你的应用程序在访问 Thrift 服务时，后者实际上就和它运行在同一台机器上。

Thrift API 包含很多其他语言的预生成 stub，包含 C++，Perl, PHP, Python 及 Ruby。Thrift 支持不同版本，因此我们可以从同一个客户代码中访问不同版本的 Hadoop 文件系统(不过必须运行针对每个版本的代理)。

关于安装与使用教程，请参阅 src/contrib/thriftfs 目录中关于 Hadoop 分布的文档。

C 语言库

Hadoop 提供了反映 Java 文件系统接口的名为 libhdfs 的 C 语言库(它被编写为一个访问 HDFS 的 C 语言库，但其实可以访问任意 Hadoop 文件系统)。它会使用 Java 本地接口(JNI)调用一个 Java 文件系统客户。

C API 与 Java 的非常相似，但它一般比 Java 的滞后，因此目前还不支持一些新特征。相关资料可参见 libhdfs/docs/api 目录中关于 Hadoop 分布的 C API 文档。

Hadoop 中有预先建好的 32 位 Linux 的 libhdfs 二元码，但对于其他平台，需要使用 http://wiki.apache.org/hadoop/LibHDFS的教程自己编写。

FUSE

用户空间文件系统(Filesystem in Userspace，FUSE)允许一些文件系统整合为一个 Unix 文件系统在用户空间中执行。通过使用 Hadoop 的 Fuse-DFS 分类模块，任意一个 Hadoop 文件系统(不过一般为 HDFS)都可以作为一个标准文件系统进行挂载。我们随后便

可以使用 Unix 的工具(如 ls 和 cat)与这个文件系统交互，还可以通过任意一种编程语言使用 POSIX 库来访问文件系统。

Fuse-DFS 是用 C 语言实现的，使用 libhdfs 作为与 HDFS 的接口。要想了解如何编译和运行 Fuse-DFS，可参见 src/contrib./fuse-dfs 中的 Hadoop 分布目录。

WebDAV

WebDAV 是一系列支持编辑和更新文件的 HTTP 的扩展。在大部分操作系统中，WebDAV 共享都可以作为文件系统进行挂载，因此借由 WebDAV 来向外提供 HDFS(或其他 Hadoop 文件系统)，可以将 HDFS 作为一个标准文件系统进行访问。

在本书写作期间，Hadoop 中的 WebDAV 支持(通过对 Hadoop 调用 Java API 来实现)仍在开发中，要想了解最新动态，可访问https://issues,apache.org/jira/browse/ HADOOP-496。

其他 HDFS 接口

对于 HDFS 有两种特定的接口。

HTTP

HDFS 定义了一个只读接口用来在 HTTP 上检索目录列表和数据。目录列表由名称节点的嵌入式 Web 服务器(运行在 50070 端口)以 XML 格式提供服务，文件数据由数据节点通过它们的 Web 服务器 (运行在 50075 端口)传输。这个协议并不拘泥于某个 HDFS 版本，因此用户可以编写使用 HTTP 从运行不同版本 Hadoop 的 HDFS 集群中读取数据的客户端应用。HftpFileSystem 就是其中一种：一个通过 HTTP 与 HDFS 交流的 Hadoop 文件系统(HsftpFileSystem 是 HTTPS 的变体)。

FTP

尽管本书写作期间尚未完成(https://issues.apache.org/jira/browse/HADOOP-3199)，但我们还是要提一下，还有一个对 HDFS 的 FTP 接口，它允许使用 FTP 协议与 HDFS 交互。这个接口很方便，它使用现有 FTP 客户端与 HDFS 进行数据的传输。

对 HDFS 的 FTP 接口与 FTPFileSystem 不可混为一谈，此接口的目的是将任意 FTP 服务器向外暴露为 Hadoop 文件系统。

3.5 Java 接口

在本小节，我们要深入探索 Hadoop 的 Filesystem 类：与 Hadoop 的文件系统交互的 API。 虽然我们主要关注的是 HDFS 的实现 DistributedFileSystem，但总体来说，还是应该努力编写不同于 FileSsytem 抽象类的代码，以保持其在不同文件系统中的可移植性。这是

考验编程能力的最佳手段，因为我们很快就可以使用存储在本地文件系统中的数据来运行测试了。

3.5.1 从 Hadoop URL 中读取数据

要从 Hadoop 文件系统中读取文件，一个最简单的方法是使用 java.net.URL 对象来打开一个数据流，从而从中读取数据。一般的格式如下：

```
1.      InputStream in = null;
2.  try {
3.       in = new URL("hdfs://host/path").openStream();
4.       // process in
5.  } finally {
6.       IOUtils.closeStream(in);
7.  }
```

这里还需要一点工作来让 Java 识别 Hadoop 文件系统的 URL 方案，就是通过一个 FsUrlStreamHandlerFactory 实例来调用在 URL 中的 setURLStreamHandler-Factory 方法。这种方法在一个 Java 虚拟机中只能被调用一次，因此一般都在一个静态块中执行。这个限制意味着如果程序的其他部件(可能是不在你控制中的第三方部件)设置一个 URLStreamHandlerFactory，我们便无法再从 Hadoop 中读取数据。下一节将讨论另一种方法。

例 3-1 展示了以标准输出显示 Hadoop 文件系统的文件的程序，它类似于 Unix 的 cat 命令。

例 3-1：用 URLStreamHandler 以标准输出格式显示 Hadoop 文件系统的文件

```
1.  public class URLCat {
2.
3.    static {
4.      URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
5.    }
6.
7.    public static void main(String[] args) throws Exception {
8.      InputStream in = null;
9.      try {
10.       in = new URL(args[0]).openStream();
11.       IOUtils.copyBytes(in, System.out, 4096, false);
12.     } finally {
13.       IOUtils.closeStream(in);
```

```
14.     }
15.   }
16.}
```

我们使用 Hadoop 中简洁的 IOUtils 类在 finally 子句中关闭数据流，同时复制输入流和输出流之间的字节(本例中是 System.out)。copyBytes 方法的最后两个参数，前者是要复制的缓冲的大小，后者表示复制结束后是否关闭数据流。这里是将输入流关掉了，而 System.out 不需要关闭。

下面是一个运行示例：

```
1.      % hadoop URLCat hdfs://localhost/user/tom/quangle.txt
2. On the top of the Crumpetty Tree
3. The Quangle Wangle sat,
4. But his face you could not see,
5. On account of his Beaver Hat.
```

3.5.2 使用 FileSystem API 读取数据

如前一小节所解释的，有时不能在应用中设置 URLStreamHandlerFactory。这时，我们需要用 FileSystem API 来打开一个文件的输入流。

文件在 Hadoop 文件系统中显示为一个 Hadoop Path 对象(不是一个 java.io.File 对象，因为它的语义与本地文件系统关联太紧密)。我们可以把一个路径视为一个 Hadoop 文件系统 URI，如 hdfs://localhost/user/tom/quangle.txt。

FileSystem 是一个普通的文件系统 API，所以首要任务是检索我们要用的文件系统实例，这里是 HDFS。取得 FileSystem 实例有两种静态工厂方法：

```
1. public static FileSystem get(Configuration conf)
   throws IOException
2. ublic static FileSystem get(URI uri,
   Configuration conf) throws IOException
```

Configuration 对象封装了一个客户端或服务器的配置，这是用从类路径读取而来的配置文件(如 conf/core-site.xml)来设置的。第一个方法返回的是默认文件系统(在 conf/core-site.xml 中设置的，如果没有设置过，则是默认的本地文件系统)。第二个方法使用指定的 URI 方案及决定所用文件系统的权限，如果指定 URI 中没有指定方案，则退回默认的文件系统。

有了 FileSystem 实例后，我们调用 open()来得到一个文件的输入流：

```
1. public FSDataInputStream open(Path f) throws IOException
```

```
2. ublic abstract FSDataInputStream open(Path f,
   int bufferSize) throws IOException
```

第一个方法使用默认 4 KB 的缓冲大小。

将它们合在一起，我们可以在例 3-2 中重写例 3-1。

例 3-2：直接使用 FileSystem 以标准输出格式显示 Hadoop 文件系统的文件

```
1. public class FileSystemCat {
2.   public static void main(String[] args) throws Exception {
3.     String uri = args[0];
4.     Configuration conf = new Configuration();
5.     FileSystem fs = FileSystem.get(URI.create(uri), conf);
6.     InputStream in = null;
7.     try {
8.       in = fs.open(new Path(uri));
9.       IOUtils.copyBytes(in, System.out, 4096, false);
10.    } finally {
11.      IOUtils.closeStream(in);
12.    }
13.  }
14.}
```

程序运行结果如下：

```
1.     % hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
2. On the top of the Crumpetty Tree
3. The Quangle Wangle sat,
4. But his face you could not see,
5. On account of his Beaver Hat.
6. FSDataInputStream
```

FileSystem 中的 open()方法实际上返回的是一个 FSDataInputStream，而不是标准的 java.io 类。这个类是 java.io.DataInputStream 的一个子类，支持随机访问，这样就可以从流的任意位置读取数据了。

```
1.     package org.apache.hadoop.fs;
2.
3. public class FSDataInputStream extends DataInputStream
4.      implements Seekable, PositionedReadable {
5.      // implementation elided
```

```
6. }
```

Seekable 接口允许在文件中定位，并提供一个查询方法，用于查询当前位置相对于文件开始处的偏移量(getPos())：

```
1. public interface Seekable {
2.   void seek(long pos) throws IOException;
3.   long getPos() throws IOException;
4.   boolean seekToNewSource(long targetPos) throws IOException;
5. }
```

调用 seek()来定位大于文件长度的位置会导致 IOException 异常。与 java.io.InputStream 中的 skip()不同，seek()并没有指出数据流当前位置之后的一点，它可以移到文件中任意一个绝对位置。

应用程序开发人员并不常用 seekToNewSource()方法。此方法一般倾向于切换到数据的另一个副本并在新的副本中寻找 targetPos 指定的位置。HDFS 内部就采用这样的方法在数据节点故障时为客户端提供可靠的数据输入流。

例 3-3 是例 3-2 的简单扩展，它将一个文件两次写入标准输出：在写一次后，定位到文件的开头再次读入数据流。

例 3-3：通过使用 seek 两次以标准输出格式显示 Hadoop 文件系统的文件

```
1. public class FileSystemDoubleCat {
2.
3.   public static void main(String[] args) throws Exception {
4.     String uri = args[0];
5.     Configuration conf = new Configuration();
6.     FileSystem fs = FileSystem.get(URI.create(uri), conf);
7.     FSDataInputStream in = null;
8.     try {
9.       in = fs.open(new Path(uri));
10.      IOUtils.copyBytes(in, System.out, 4096, false);
11.      in.seek(0); // go back to the start of the file
12.      IOUtils.copyBytes(in, System.out, 4096, false);
13.    } finally {
14.      IOUtils.closeStream(in);
15.    }
16.  }
17.}
```

在一个小文件上运行得到以下结果：

```
% hadoop FileSystemDoubleCat hdfs://localhost/user/tom/quangle.txt

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDataInputStream 也实现了 PositionedReadable 接口，从一个指定位置读取一部分数据：

```
public interface PositionedReadable {

  public int read(long position, byte[] buffer,
  int offset, int length)
        throws IOException;

  public void readFully(long position, byte[]
  buffer, int offset, int length)
        throws IOException;

  public void readFully(long position, byte[]
  buffer) throws IOException;
}
```

read()方法从指定 position 读取指定长度的字节放入缓冲 buffer 的指定偏离量 offset。返回值是实际读到的字节数：调用者需要检查这个值，它有可能小于指定的长度。readFully()方法会读出指定字节由 length 指定的数据到 buffer 中或在只接受 buffer 字节数组的版本中，再读取 buffer.length 字节(这儿指的是第三个函数)，若已经到文件末，将会抛出 EOFException。

所有这些方法会保留文件当前位置并且是线程安全的，因此它们提供了在读取文件(可能是元数据)的主要部分时访问其他部分的便利方法。其实，这只是使用 Seekable 接口的实现，格式如下：

```
    long oldPos = getPos();
```

```
2. try {
3.   seek(position);
4.   // read data
5. } finally {
6.   seek(oldPos);
7. }
```

最后务必牢记，seek()是一个相对高开销的操作，需要慎重使用。我们需要依靠流数据构建应用访问模式(如使用 MapReduce)，而不要大量执行 seek 操作。

3.5.3 写入数据

FileSystem 类有一系列创建文件的方法。最简单的是给拟创建的文件指定一个路径对象，然后返回一个用来写的输出流：

```
1. public FSDataOutputStream create(Path f) throws IOException
```

这个方法有重载的版本允许我们指定是否强制覆盖已有的文件、文件副本数量、写入文件时的缓冲大小、文件块大小以及文件许可。

注意：create()方法为需要写入的文件而创建的父目录可能原先并不存在。虽然这样很方便，但有时并不希望这样。如果我们想在父目录不存在时不执行写入，就必须在调用 exists()首先检查父目录是否存在。

还有一个用于传递回调接口的重载方法 Progressable，如此一来，我们所写的应用就会被告知数据写入数据节点的进度：

```
1. package org.apache.hadoop.util;
2. ublic interface Progressable {
3.       public void progress();
```

新建文件的另一种方法是使用 append()在一个已有文件中追加(也有一些其他重载版本)：

```
1. public FSDataOutputStream append(Path f) throws IOException
```

这个操作允许一个写入者打开已有文件并在其末尾写入数据。有了这个 API，会产生无边界文件的应用，以日志文件为例，就可以在重启后在已有文件上继续写入。此添加操作是可选的，并不是所有 Hadoop 文件系统都有实现。HDFS 支持添加，但 S3 就不支持了。

例 3-4 展示了如何将本地文件复制到 Hadoop 文件系统。我们在每次 Hadoop 调用 progress()方法时，也就是在每 64 KB 数据包写入数据节点管道后打印一个句号来展示整个

过程。(注意，这个动作并不是 API 指定的，因此在 Hadoop 后面的版本中大多被改变了。API 仅仅是让我们注意到"发生了一些事"。)

例 3-4：将本地文件复制到 Hadoop 文件系统并显示进度

```
1. public class FileCopyWithProgress {
2.   public static void main(String[] args) throws Exception {
3.     String localSrc = args[0];
4.     String dst = args[1];
5.     InputStream in = new BufferedInputStream(new
   FileInputStream(localSrc));
6.
7.
8.     Configuration conf = new Configuration();
9.     FileSystem fs = FileSystem.get(URI.create(dst), conf);
10.    OutputStream out = fs.create(new Path(dst), new Progressable() {
11.      public void progress() {
12.        System.out.print(".");
13.      }
14.    });
15.
16.    IOUtils.copyBytes(in, out, 4096, true);
17.  }
18.}
```

典型用途：

```
1. % hadoop FileCopyWithProgress input/docs/1400-8.txt
   hdfs://localhost/user/tom/1400-8.txt
2. ...............
```

目前，其他 Hadoop 文件系统在写入时都不会调用 progress()。通过后面几章的描述，我们会感到进度之于 MapReduce 应用的重要性。

FSDataOutputStream

FileSystem 中的 create()方法返回了一个 FSDataOutputStream，与 FSDataInputStream 类似，它也有一个查询文件当前位置的方法：

```
1. package org.apache.hadoop.fs;
2.
```

```
3. public class FSDataOutputStream extends
   DataOutputStream implements Syncable {
4.
5.   public long getPos() throws IOException {
6.     // implementation elided
7.   }
8.
9.    // implementation elided
10.}
```

但是，与 FSDataInputStream 不同，FSDataOutputStream 不允许定位。这是因为 HDFS 只允许对一个打开的文件顺序写入，或向一个已有文件添加。换句话说，它不支持除文件尾部的其他位置的写入，这样一来，写入时的定位就没有什么意义。

3.5.4 目录

fIlesystem 提供了一个创建目录的方法：

```
1. public boolean mkdirs(Path f) throws IOException
```

这个方法会创建所有那些必要但不存在的父目录，就像 java.io.File 的 mkdirs()。如果目录(以及所有父目录)都创建成功，它会返回 true。

我们常常不需要很确切地创建一个目录，因为调用 create()写入文件时会自动生成所有的父目录。

3.5.5 查询文件系统（1）

文件元数据：Filestatus

任何文件系统的一个重要特征是定位其目录结构及检索其存储的文件和目录信息的能力。FileStatus 类封装了文件系统中文件和目录的元数据，包括文件长度、块大小、副本、修改时间、所有者以及许可信息。

FileSystem 的 getFileStatus()提供了获取一个文件或目录的状态对象的方法。例 3-5 展示了它的用法。

例 3-5：展示文件状态信息

```
1. public class ShowFileStatusTest {
2.
3.   private MiniDFSCluster cluster; // use an
   in-process HDFS cluster for testing
4.   private FileSystem fs;
```

```
5.
6.    @Before
7.    public void setUp() throws IOException {
8.      Configuration conf = new Configuration();
9.      if (System.getProperty("test.build.data") == null) {
10.       System.setProperty("test.build.data", "/tmp");
11.     }
12.     cluster = new MiniDFSCluster(conf, 1, true, null);
13.     fs = cluster.getFileSystem();
14.     OutputStream out = fs.create(new Path("/dir/file"));
15.     out.write("content".getBytes("UTF-8"));
16.     out.close();
17.   }
18.
19.   @After
20.   public void tearDown() throws IOException {
21.     if (fs != null) { fs.close(); }
22.     if (cluster != null) { cluster.shutdown(); }
23.   }
24.
25.   @Test(expected = FileNotFoundException.class)
26.   public void throwsFileNotFoundForNonExistentFile()
    throws IOException {
27.     fs.getFileStatus(new Path("no-such-file"));
28.   }
29.
30.   @Test
31.   public void fileStatusForFile() throws IOException {
32.     Path file = new Path("/dir/file");
33.     FileStatus stat = fs.getFileStatus(file);
34.     assertThat(stat.getPath().toUri().getPath(), is("/dir/file"));
35.     assertThat(stat.isDir(), is(false));
36.     assertThat(stat.getLen(), is(7L));
37.     assertThat(stat.getModificationTime(),
38.         is(lessThanOrEqualTo(System.currentTimeMillis())));
39.     assertThat(stat.getReplication(), is((short) 1));
40.     assertThat(stat.getBlockSize(), is(64 * 1024 * 1024L));
41.     assertThat(stat.getOwner(), is("tom"));
```

```
42.     assertThat(stat.getGroup(), is("supergroup"));
43.     assertThat(stat.getPermission().toString(), is("rw-r--r--"));
44.   }
45.
46.   @Test
47.   public void fileStatusForDirectory() throws IOException {
48.     Path dir = new Path("/dir");
49.     FileStatus stat = fs.getFileStatus(dir);
50.     assertThat(stat.getPath().toUri().getPath(), is("/dir"));
51.     assertThat(stat.isDir(), is(true));
52.     assertThat(stat.getLen(), is(0L));
53.     assertThat(stat.getModificationTime(),
54.         is(lessThanOrEqualTo(System.currentTimeMillis())));
55.     assertThat(stat.getReplication(), is((short) 0));
56.     assertThat(stat.getBlockSize(), is(0L));
57.     assertThat(stat.getOwner(), is("tom"));
58.     assertThat(stat.getGroup(), is("supergroup"));
59.     assertThat(stat.getPermission().toString(), is("rwxr-xr-x"));
60.   }
61.
62. }
```

如果文件或目录不存在，即会抛出 FileNotFoundException 异常。如果你只对文件或目录是否存在有兴趣，exists()方法会更方便：

```
1. public boolean exists(Path f) throws IOException
```

列出文件

查找一个文件或目录的信息很实用，但有时我们还需要能够列出目录的内容。这就是listStatus()方法的功能：

```
1.     public FileStatus[] listStatus(Path f)
   throws IOException
2. public FileStatus[] listStatus(Path f,
   PathFilter filter) throws IOException
3. public FileStatus[] listStatus(Path[] files)
   throws IOException
4. public FileStatus[] listStatus(Path[] files,
   PathFilter filter) throws IOException
```

传入参数是一个文件时，它会简单地返回长度为 1 的 FileStatus 对象的一个数组。当传入参数是一个目录时，它会返回 0 或者多个 FileStatus 对象，代表着此目录所包含的文件和目录。

重载方法允许我们使用 PathFilter 来限制匹配的文件和目录，示例参见后文。如果把路径数组作为参数来调用 listStatus 方法，其结果是与依次对每个路径调用此方法，再将 FileStatus 对象数组收集在一个单一数组中的结果是相同的，但是前者更为方便。这在建立从文件系统树的不同部分执行的输入文件的列表时很有用。例 3-6 是这种思想的简单示范。注意 FIleUtil 中 stat2Paths()的使用，它将一个 FileStatus 对象数组转换为 Path 对象数组。

例 3-6：显示一个 Hadoop 文件系统中一些路径的文件信息

```
1. public class ListStatus {
2.
3.   public static void main(String[] args) throws Exception {
4.     String uri = args[0];
5.     Configuration conf = new Configuration();
6.     FileSystem fs = FileSystem.get(URI.create(uri), conf);
7.
8.     Path[] paths = new Path[args.length];
9.     for (int i = 0; i < paths.length; i++) {
10.      paths[i] = new Path(args[i]);
11.    }
12.
13.    FileStatus[] status = fs.listStatus(paths);
14.    Path[] listedPaths = FileUtil.stat2Paths(status);
15.    for (Path p : listedPaths) {
16.      System.out.println(p);
17.    }
18.  }
19.}
```

3.5.5 查询文件系统（2）

我们可以使用这个程序得到一个路径集的整个目录列表。

```
1.     % hadoop ListStatus hdfs://localhost/ hdfs://localhost/user/tom
2. hdfs://localhost/user
3. hdfs://localhost/user/tom/books
4. hdfs://localhost/user/tom/quangle.txt
```

文件格式

在一步操作中处理批量文件，这个要求很常见。举例来说，处理日志的 MapReduce 作业可能会分析一个月的文件，这些文件被包含在大量目录中。Hadoop 有一个通配的操作，可以方便地使用通配符在一个表达式中核对多个文件，不需要列举每个文件和目录来指定输入。Hadoop 为执行通配提供了两个 FileSystem 方法：

```
1. public FileStatus[] globStatus(Path pathPattern) throws IOException
2. ublic FileStatus[] globStatus(Path pathPattern,
   PathFilter filter) throws IOException
```

globStatus()返回了其路径匹配于所供格式的 FileStatus 对象数组，按路径排序。可选的 PathFilter 命令可以进一步指定限制匹配。

Hadoop 支持的一系列通配符与 Unix bash 相同(见表 3-2)。

表 3-2：通配符及其作用

通配符	名称	匹配
*	星号	匹配 0 或多个字符
?	问号	匹配单一字符
[ab]	字符类别	匹配{a,b}中的一个字符

续表

通配符	名称	匹配
[^ab]	非字符类别	匹配不是{a,b}中的一个字符
[a-b]	字符范围	匹配一个在{a,b}范围内的字符(包括 ab)，a 在字典顺序上要小于或等于 b
[^a-b]	非字符范围	匹配一个不在{a,b}范围内的字符(包括 ab)，a 在字典顺序上要小于或等于 b
{a,b}	或选择	匹配包含 a 或 b 中的一个的语句
\c	转义字符	匹配元字符 c

假设有日志文件存储在按日期分层组织的目录结构中。如此一来，便可以假设 2007 年最后一天的日志文件就会以/2007/12/31 的命名存入目录。假设整个文件列表如下：

```
1. /2007/12/30
2. /2007/12/31
3. /2008/01/01
4. /2008/01/02
```

以下是一些文件通配符及其扩展。

通配符	扩展
/*	/2007/2008
/*/*	/2007/12 /2008/01
/*/12/*	/2007/12/30 /2007/12/31
/200?	/2007 /2008
/200[78]	/2007 /2008
/200[7-8]	/2007 /2008
/200[^01234569]	/2007 /2008
/*/*/{31,01}	/2007/12/31 /2008/01/01
/*/*/3{0,1}	/2007/12/30 /2007/12/31
/*/{12/31,01/01}	/2007/12/31 /2008/01/01

PathFilter 对象

通配格式不是总能够精确地描述我们想要访问的文件集合。比如，使用通配格式排除一个特定的文件就不太可能。FileSystem 中的 listStatus()和 globStatus()方法提供了可选的 PathFilter 对象，使我们能够通过编程方式控制匹配：

```
1. package org.apache.hadoop.fs;
2.
3. ublic interface PathFilter {
4.    boolean accept(Path path);
```

PathFilter 与 java.io.FileFilter 一样，是 Path 对象而不是 File 对象。

例 3-7 展示了一个 PathFilter，用于排除匹配一个正则表达式的路径。

例 3-7：一个 PathFilter，用于排除匹配一个正则表达式的路径

```
1. public class RegexExcludePathFilter implements PathFilter {
2.
3.   private final String regex;
```

```
4.
5.    public RegexExcludePathFilter(String regex) {
6.      this.regex = regex;
7.    }
8.
9.    public boolean accept(Path path) {
10.     return !path.toString().matches(regex);
11.   }
12. }
```

这个过滤器只留下与正则表达式不同的文件。我们将它与预先剔除一些文件集合的通配配合：过滤器用来优化结果。例如：

```
1. fs.globStatus(new Path("/2007/*/*"),
2. ew RegexExcludeFilter("^.*/2007/12/31$"))
```

<pre><ol class="dp-xml"><li class="alt">fs.globStatus(new Path("/2007/*/*"),   ew RegexExcludeFilter("^.*/2007/12/31$")) </pre>

3.5.6 删除数据

使用 FileSystem 中的 delete()可以永久性删除文件或目录。

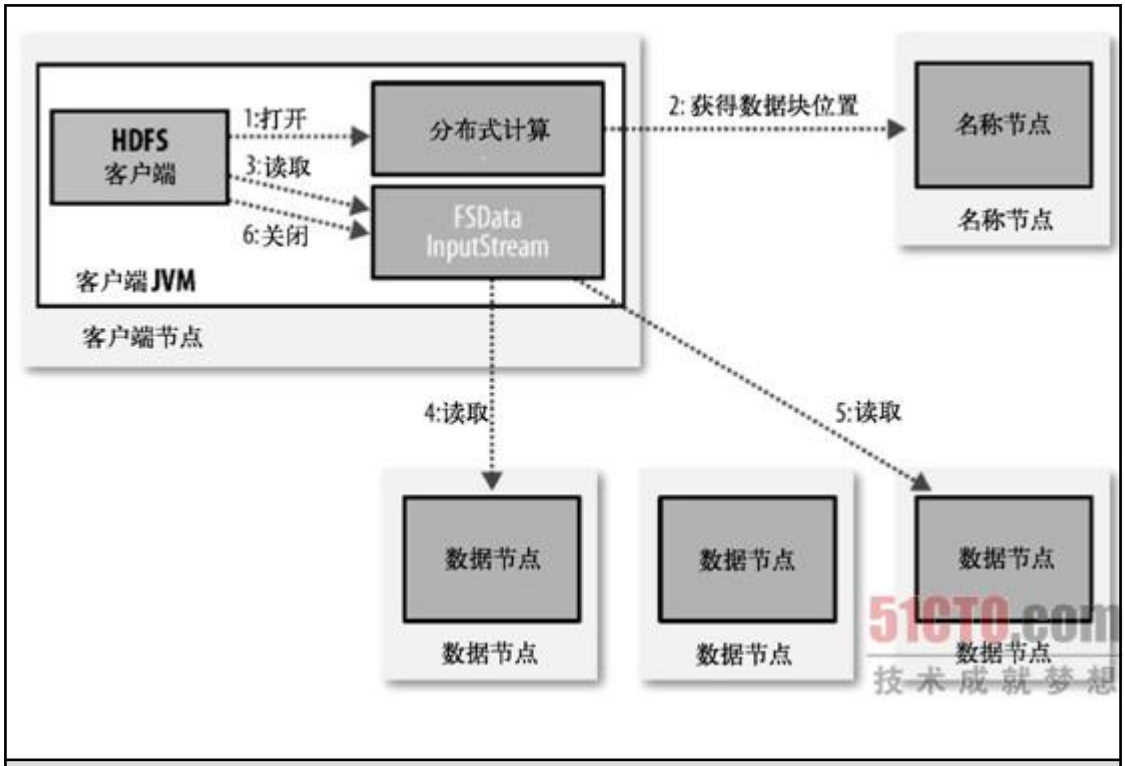
```
1. public boolean delete(Path f, boolean recursive) throws IOException
```

如果传入的 f 是一个文件或空目录，那么 recursive 的值就会被忽略。只有在 recrusive 值为 true 时，一个非空目录以及其内容才会被删除(否则会抛出 IOException 异常)。

3.6 数据流

3.6.1 文件读取剖析

为了了解客户端及与之交互的 HDFS、名称节点和数据节点之间的数据流是怎样的，我们可参考图 3-1，其中显示了在读取文件时一些事件的主要顺序。



（点击查看大图）图 3-1：客户端从 HDFS 中读取数据

客户端通过调用 FileSystem 对象的 open()来读取希望打开的文件，对于 HDFS 来说，这个对象是分布式文件系统(图 3-1 中的步骤 1)的一个实例。DistributedFileystem 通过使用 RPC 来调用名称节点，以确定文件开头部分的块的位置(步骤 2)。对于每一个块，名称节点返回具有该块副本的数据节点地址。此外，这些数据节点根据它们与客户端的距离来排序(根据网络集群的拓扑；参见后文补充材料"网络拓扑与 Hadoop")。如果该客户端本身就是一个数据节点(比如在一个 MapReduce 任务中)，便从本地数据节点中读取。

Distributed FilesyStem 返回一个 FSData InputStream 对象(一个支持文件定位的输入流)给客户端读取数据。FSData InputStream 转而包装了一个 DFSInputStream 对象。

接着，客户端对这个输入流调用 read()(步骤 3)。存储着文件开头部分的块的数据节点地址的 DFSInputStream 随即与这些块最近的数据节点相连接。通过在数据流中重复调用 read()，数据会从数据节点返回客户端(步骤 4)。到达块的末端时，DFSInputStream 会关闭与数据节点间的联系，然后为下一个块找到最佳的数据节点(步骤 5)。客户端只需要读取一个连续的流，这些对于客户端来说都是透明的。

客户端从流中读取数据时，块是按照 DFSInputStream 打开与数据节点的新连接的顺序读取的。它也会调用名称节点来检索下一组需要的块的数据节点的位置。一旦客户端完成读取，就对文件系统数据输入流调用 close()(步骤 6)。

在读取的时候，如果客户端在与数据节点通信时遇到一个错误，那么它就会去尝试对这个块来说下一个最近的块。它也会记住那个故障的数据节点，以保证不会再对之后的块

进行徒劳无益的尝试。客户端也会确认从数据节点发来的数据的校验和。如果发现一个损坏的块，它就会在客户端试图从别的数据节点中读取一个块的副本之前报告给名称节点。

这个设计的一个重点是，客户端直接联系数据节点去检索数据，并被名称节点指引到每个块中最好的数据节点。因为数据流动在此集群中是在所有数据节点分散进行的，所以这种设计能使 HDFS 可扩展到最大的并发客户端数量。同时，名称节点只不过是提供块位置请求(存储在内存中，因而非常高效)，不是提供数据。否则如果客户端数量增长，名称节点会快速成为一个"瓶颈"。

网络拓扑与 Hadoop

两个节点在一个本地网络中被称为"彼此的近邻"是什么意思？在高容量数据处理中，限制因素是我们在节点间传送数据的速率--带宽很稀缺。这个想法便是将两个节点间的带宽作为距离的衡量标准。

衡量节点间的带宽，实际上很难实现(它需要一个稳定的集群，并且在集群中成对的节点的数量的增长要是节点数量的平方)，不及 Hadoop 采用一个简单的方法，把网络看作一棵树，两个节点间的距离是距离它们最近的共同祖先的总和。该树中的等级是没有被预先设定的，但是它对于相当于数据中心、框架和一直在运

行的节点的等级是共同的。这个想法是，对于以下每个场景，可用带宽依次减少：

相同节点中的进程

同一机架上的不同节点

同一数据中心的不同机架上的节点

不同数据中心的节点

例如，假设节点 n1 在数据中心 d1 中的机架 r1 上。这被表示成/d1/r1/n1。利用这种标记，这里给出四种描述的距离：

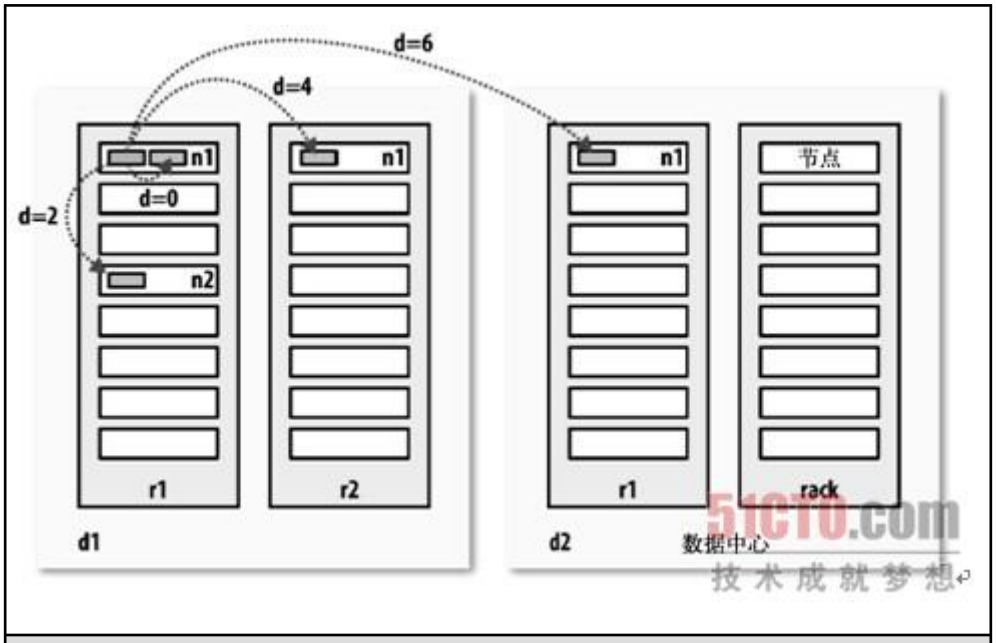
距离(/d1/r1/n1, /d1/r1/n1)=0(相同节点中的进程)

距离(/d1/r1/n1, /d1/r1/n2)=2(同一机架上的不同节点)

距离(/d1/r1/n1, /d1/r2/n3)=4(同一数据中心的不同机架上的节点)

距离(/d1/r1/n1, /d2/r3/n4)=6(不同数据中心的节点)

这在图 3-2 中用图示形式表达(数学爱好者会注意到这是一个距离公制的例子)。



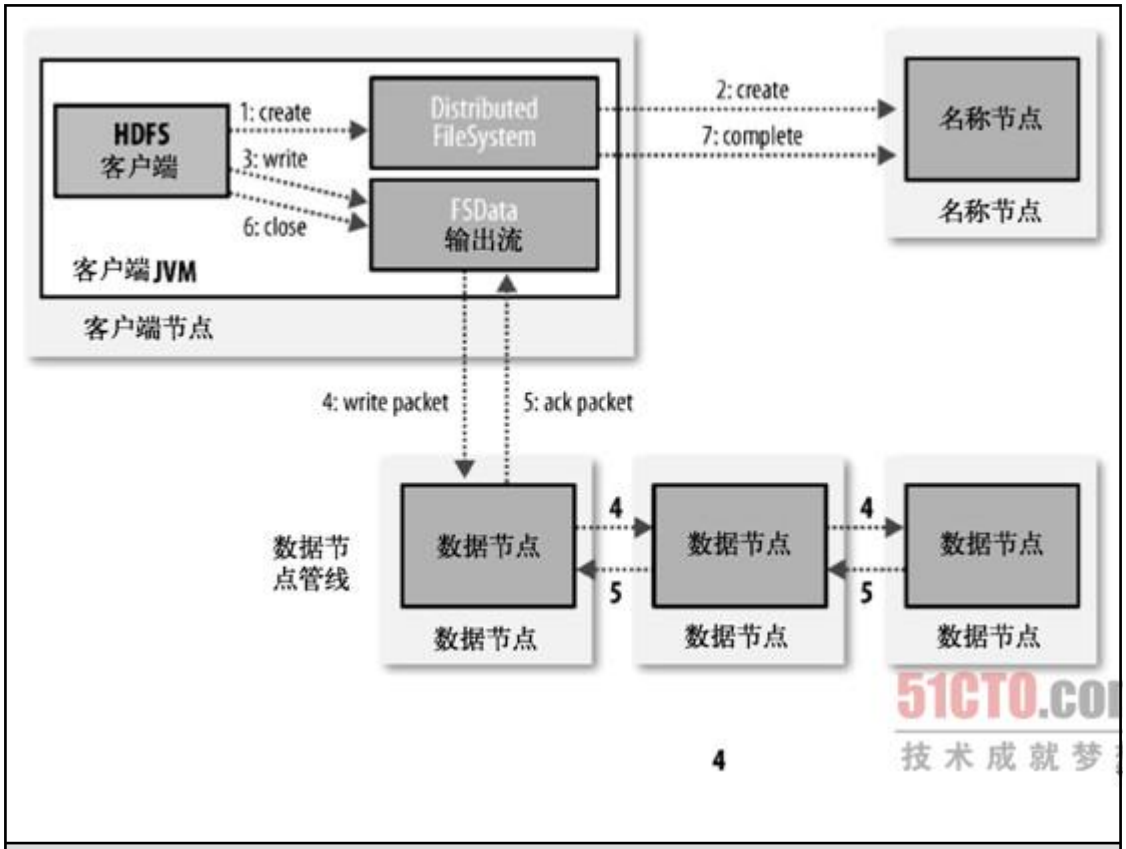
（点击查看大图）图 3-2：Hadoop 中的网络距离

我们必须意识到，Hadoop 无法预测网络拓扑结构。它需要一定帮助，我们将在第 9 章讨论如何配置拓扑。不过在默认情况下，假设网络是平的(一个单层的等级制)，或者换句话说，所有节点都在同一数据中心的同一机架。小的集群可能如此，所以不需要进一步的配置。

3.6.2 文件写入剖析

接下来我们要看文件是如何写入 HDFS 的。尽管比较详细，但对于理解数据流还是很有用的，因为它清楚地说明了 HDFS 的连贯模型(coherency model)。

我们考虑的情况是创建一个新的文件，向其写入数据后关闭该文件。参见图 3-3。



（点击查看大图）图 3-3：客户端对 HDFS 写入数据

客户端通过在 DistributedFilesystem 中调用 create()来创建文件(图 3-3 的步骤 1)。DistributedFilesystem 一个 RPC 去调用名称节点，在文件系统的命名空间中创建一个新的文件，没有块与之相联系(步骤 2)。名称节点执行各种不同的检查以确保这个文件不会已经存在，并且客户端有可以创建文件的适当的许可。如果这些检查通过，名称节点就会生成一个新文件的记录；否则，文件创建失败并向客户端抛出一个 IOException 异常。分布式文件系统返回一个文件系统数据输出流，让客户端开始写入数据。就像读取事件一样，文件系统数据输出流控制一个 DFSoutPutstream，负责处理数据节点和名称节点之间的通信。

在客户端写入数据时(步骤 3)，DFSoutPutstream 将它分成一个个的包，写入内部的队列，称为数据队列。数据队列随数据流流动，数据流的责任是根据适合的数据节点的列表来要求这些节点为副本分配新的块。这个数据节点的列表形成一个管　　线-- 我们假设这个副本数是 3，所以有 3 个节点在管线中。数据流将包分流给管线中第一个的数据节点，这个节点会存储包并且发送给管线中的第二个数据节点。同样地，第二个数据节点存储包并且传给管线中第三个(也是最后一个)数据节点(步骤 4)。

DFSoutputStream 也有一个内部的包队列来等待数据节点收到确认，称为确认队列。一个包只有在被管线中所有节点确认后才会被移出确认队列(步骤 5)。

如果在有数据写入期间，数据节点发生故障，则会执行下面的操作，当然这对写入数据的客户端而言，是透明的。首先管线被关闭，确认队列中的任何包都会被添加回数据队

列的前面，以确保数据节点从失败的节点处是顺流的，不会漏掉任意一个包。当前的块在正常工作的数据节点中被给予一个新的身份并联系名称节点，以便能在故障数据节点后期恢复时其中的部分数据块会被删除。故障数据节点会从管线中删除并且余下块的数据会被写入管线中的两个好的数据节点。名称节点注意到块副本不足时，会在另一个节点上安排创建一个副本。随后，后续的块会继续正常处理。

在一个块被写入期间多个数据节点发生故障的可能性虽然有但很少见。只要 dfs.replication.min 的副本(默认为 1)被写入，写操作就是成功的，并且这个块会在集群中被异步复制，直到满足其目标副本数(dfs.replication 的默认设置为 3)。

客户端完成数据的写入后，就会在流中调用 close()(步骤 6)。在向名称节点发送完信息之前，此方法会将余下的所有包放入数据节点管线并等待确认(步骤 7)。名称节点已经知道文件由哪些块组成(通过 Data streamer 询问块分配)，所以它只需在返回成功前等待块进行最小量的复制。

副本的放置

名称节点如何选择哪个数据节点来保存副本？我们需要在可靠性与写入带宽和读取带宽之间进行权衡。例如，因为副本管线都在单独一个节点上运行，所以把所有副本都放在一个节点基本上不会损失写入带宽，但这并没有实现真的冗余(如果节点发生故障，那么该块中的数据会丢失)。同样，离架读取的带宽是很高的。另一个极端，把副本放在不同的数据中心会最大限度地增大冗余，但会以带宽为代价。即使在相同的数据中心(所有的 Hadoop 集群到目前为止都运行在同一个数据中心)，也有许多不同的放置策略。其实，Hadoop 在发布的 0.17.0 版中改变了放置策略来帮助保持块在集群间有相对平均的分布(第 10 章详细说明了如何保持集群的平衡)。

Hadoop 的策略是在与客户端相同的节点上放置第一个副本(若客户端运行在

集群之外，就可以随机选择节点，不过系统会避免挑选那些太满或太忙的节点)。

第二个副本被放置在与第一个不同的随机选择的机架上(离架)。第三个副本被放置在与第二个相同的机架上，但放在不同的节点。更多的副本被放置在集群中的随机节点上，不过系统会尽量避免在相同的机架上放置太多的副本。

一旦选定副本放置的位置，就会生成一个管线，会考虑到网络拓扑。副本数为 3 的管道看起来如图 3-4 所示。

图 3-4：一个典型的副本管线

总的来说，这样的方法在稳定性(块存储在两个机架中)、写入带宽(写入操作只需要做一个单一网络转换)、读取性能(选择从两个机架中进行读取)和集群中块的分布(客户端只在本地机架写入一个块)之间，进行了较好的平衡。

3.6.3 一致模型

文件系统的一致模型描述了对文件读写的数据可见性。HDFS 为性能牺牲了一些 POSIX 请求，因此一些操作可能比想像的困难。

在创建一个文件之后，在文件系统的命名空间中是可见的，如下所示：

```
1. Path p = new Path("p");
2. Fs.create(p);
3. assertThat(fs.exists(p),is(true));
```

但是，写入文件的内容并不保证能被看见，即使数据流已经被刷新。所以文件长度显示为 0：

```
1. Path p = new Path("p");
2. OutputStream out = fs.create(p);
3. out.write("content".getBytes("UTF-8"));
4. out.flush();
5. assertThat(fs.getFileStatus(p).getLen(),is(0L));
```

一旦写入的数据超过一个块的数据，新的读取者就能看见第一个块。对于之后的块也是这样。总之，它始终是当前正在被写入的块，其他读取者是看不见它的。

HDFS 提供一个方法来强制所有的缓存与数据节点同步，即在文件系统数据输出流使用 sync()方法。在 sync()返回成功后，HDFS 能保证文件中直至写入的最后的数据对所有新的读取者而言，都是可见且一致的。万一发生冲突(与客户端或 HDFS)，也不会造成数据丢失：

```
1. Path p = new Path("p");
2. FSDataOutputStream out = fs.create(p);
3. out.write("content".getBytes("UTF-8"));
4. out.flush();
5. out.sync();
6. assertThat(fs.getFileStatus(p).getLen(),
   is(((long) "content".length())));
```

此行为类似于 Unix 中的 fsync 系统调用-- 为一个文件描述符提交缓冲数据。例如，利用 Java API 写入一个本地文件，我们肯定能够看到刷新流和同步之后的内容：

```
1. FileOutputStream out = new FileOutputStream(localFile);
2. out.write("content".getBytes("UTF-8"));
3. out.flush(); // flush to operating system
4. out.getFD().sync(); // sync to disk
5. assertThat(localFile.length(), is(((long) "content".length())));
```

在 HDFS 中关闭一个文件其实还执行了一个隐含的 sync()：

```
1. Path p = new Path("p");
2. OutputStream out = fs.create(p);
3. out.write("content".getBytes("UTF-8"));
4. out.close();
5. assertThat(fs.getFileStatus(p).getLen(),
   is(((long) "content".length())));
```

应用设计的重要性

这个一致模型与具体设计应用程序的方法有关。如果不调用 sync()，那么一旦客户端或系统发生故障，就可能失去一个块的数据。对很多应用来说，这是不可接受的，所以我们应该在适当的地方调用 sync()，例如在写入一定的记录或字节之后。尽管 sync()操作被设计为尽量减少 HDFS 负载，但它仍然有开销，所以在数据健壮性和吞吐量之间就会有所取舍。应用依赖就比较能接受，通过不同的 sync()频率来衡量应用程序，最终找到一个合适的平衡。

3.7 通过 distcp 进行并行复制

前面的 HDFS 访问模型都集中于单线程的访问。例如通过指定文件通配，我们可以对一部分文件进行处理，但是为了高效，对这些文件的并行处理需要新写一个程序。Hadoop 有一个叫 distcp(分布式复制)的有用程序，能从 Hadoop 的文件系统并行复制大量数据。

distcp 一般用于在两个 HDFS 集群中传输数据。如果集群在 Hadoop 的同一版本上运行，就适合使用 hdfs 方案：

```
1. % hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar
```

这将从第一个集群中复制/foo 目录(和它的内容)到第二个集群中的/bar 目录下，所以第二个集群会有/bar/foo 目录结构。如果/bar 不存在，则新建一个。我们可以指定多个源路径，并且所有的都会被复制到目标路径。源路径必须是绝对路径。

默认情况下，distcp 会跳过目标路径已经有的文件，但可以通过提供的-overwrite 选项进行覆盖。也可以用-update 选项来选择只更新那些修改过的文件。

注意：使用-overwrite 和-update 中任意一个(或两个)选项会改变源路径和目标路径的含义。这可以用一个例子清楚说明。如果改变先前例子中第一个集群的子树/foo 下的一个文件，就能通过运行对第二个集群的改变进行同步：

```
1. % hadoop distcp -update hdfs://namenode1/foo hdfs://namenode2/bar/foo
```

目标路径需要末尾这个额外的子目录/foo，因为源目录下的内容已被复制到目标目录下。(如果熟悉 rsync，你可以想像-overwrite 或-update 项对源路径而言，如同添加一个隐含的斜杠。)

如果对 discp 操作不是很确定，最好先对一个小的测试目录树进行尝试。

有很多选项可以控制分布式复制行为，包括预留文件属性，忽略故障和限制复制的文件或总数据的数量。运行时不带任何选项，可以看到使用说明。

distcp 是作为一个 MapReduce 作业执行的，复制工作由集群中并行运行的 map 来完成。这里并没有 reducer。每个文件都由一个单一的 map 进行复制，并且 distcp 通过将文件分成大致相等的文件来为每个 map 数量大致相同的数据。

map 的数量是这样确定的。通过让每一个 map 复制数量合理的数据以最小化任务建立所涉及的开销，是一个很好的想法，所以每个 map 的副本至少为 256 MB。(除非输入的总大小较少，否则一个 map 就足以操控全局。)例如，1 GB 的文件会被分成 4 个 map 任务。如果数据很大，为限制带宽和集群的使用而限制映射的数量就变得很有必要。map 默认的最大数量是每个集群节点(tasktracker)有 20 个。例如，复制 1000 GB 的文件到一个 100 个节点的集群，会分配 2000 个 map(每个节点 20 个 map)，所以平均每个会复制 512 MB。通

过对 distcp 指定-m 参数，会减少映射的分配数量。例如，-m 1000 会分配 1000 个 map，平均每个复制 1 GB。

如果想在两个运行着不同版本 HDFS 的集群上利用 distcp，使用 hdfs 协议是会失败的，因为 RPC 系统是不兼容的。想要弥补这种情况，可以使用基于 HTTP 的 HFTP 文件系统从源中进行读取。这个作业必须运行在目标集群上，使得 HDFS RPC 版本是兼容的。使用 HFTP 重复前面的例子：

```
1. % hadoop distcp hftp://namenode1:50070/foo hdfs://namenode2/bar
```

注意，需要在 URI 源中指定名称节点的 Web 端口。这是由 dfs.http.address 的属性决定的，默认值为 50070。

保持 HDFS 集群的平衡

向 HDFS 复制数据时，考虑集群的平衡相当重要。文件块在集群中均匀地分布时，HDFS 能达到最佳工作状态。回顾前面 1000 GB 数据的例子，通过指定-m 选项为 1，即由一个单一的 map 执行复制工作，它的意思是，不考虑速度变慢和未充分利用集群资源，每个块的第一个副本会存储在运行 map 的节点上(直到磁盘被填满)。第二和第三个副本分散在集群中，但这一个节点并不会平衡。通过让 map 的数量多于集群中节点的数量，我们便可避免这个问题。鉴于此，最好首先就用默认的每个节点 20 个 map 这个默认设置来运行 distcp。

然而，这也并不总能阻止一个集群变得不平衡。也许想限制 map 的数量以便一些节点可以被其他作业使用。若是这样，可以使用 balancer 工具(参见第 10 章)继续改善集群中块的分布。

3.8 Hadoop 归档文件

每个文件以块方式存储，块的元数据存储在名称节点的内存里，此时存储一些小的文件，HDFS 会较低效。因此，大量的小文件会耗尽名称节点的大部分内存。(注意，相较于存储文件原始内容所需要的磁盘空间，小文件所需要的空间不会更多。例如，一个 1 MB 的文件以大小为 128 MB 的块存储，使用的是 1 MB 的磁盘空间，而不是 128 MB。)

Hadoop Archives 或 HAR 文件，是一个更高效的将文件放入 HDFS 块中的文件存档设备，在减少名称节点内存使用的同时，仍然允许对文件进行透明的访问。具体说来，Hadoop Archives 可以被用作 MapReduce 的输入。

3.8.1 使用 Hadoop Archives

Hadoop Archives 通过使用 archive 工具根据一个文件集合创建而来。这些工具运行一个 MapReduce 作业来并行处理输入文件，因此我们需要一个 MapReduce 集群去运行使用它。HDFS 中有一些我们希望归档的文件：

```
% hadoop fs -lsr /my/files
-rw-r--r--    1 tom supergroup      1 2009-04-09 19:13 /my/files/a
drwxr-xr-x   - tom supergroup      0 2009-04-09 19:13 /my/files/dir
-rw-r--r--    1 tom supergroup      1 2009-04-
09 19:13 /my/files/dir/b
```

现在我们可以运行 archive 指令：

```
% hadoop archive -archiveName files.har /my/files /my
```

第一个选项是归档文件名称，这里是 file.har。HAR 文件总是有一个.har 扩展名，这是必需的，具体理由见后文描述。接下来把文件放入归档文件。这里我们只归档一个源树，即 HDFS 下/my/files 中的文件，但事实上，该工具接受多个源树。最后一个参数是 HAR 文件的输出目录。让我们看看这个归档文件是怎么创建的：

```
% hadoop fs -ls /my
Found 2 items
drwxr-xr-x    - tom supergroup      0 2009-04-09
19:13 /my/files
drwxr-xr-x    - tom supergroup      0 2009-04-09
19:13 /my/files.har
% hadoop fs -ls /my/files.har
Found 3 items
-rw-r--r--    10 tom supergroup    165 2009-04-09
19:13 /my/files.har/_index
-rw-r--r--    10 tom supergroup     23 2009-04-09
19:13 /my/files.har/_masterindex
-rw-r--r--     1 tom supergroup      2 2009-04-09
19:13 /my/files.har/part-0
```

这个目录列表展示了一个 HAR 文件的组成部分：两个索引文件和部分文件的集合(本例中只有一个)。这些部分文件包含已经链接在一起的大量原始文件的内容，并且索引使我们可以查找那些包含归档文件的部分文件，包括它的起始点和长度。但所有这些细节对于使用 har URI 方案与 HAR 文件交互的应用都是隐藏的，HAR 文件系统是建立在基础文件系统上的(本例中是 HDFS)。以下命令以递归方式列出了归档文件中的文件：

```
% hadoop fs -lsr har:///my/files.har
```

```
2. drw-r--r--    - tom supergroup       0 2009-04-09
   19:13 /my/files.har/my
3. drw-r--r--    - tom supergroup       0 2009-04-09
   19:13 /my/files.har/my/files
4. -rw-r--r--   10 tom supergroup       1 2009-04-09
   19:13 /my/files.har/my/files/a
5. drw-r--r--    - tom supergroup       0 2009-04-09
   19:13 /my/files.har/my/files/dir
6. -rw-r--r--   10 tom supergroup       1 2009-04-09
   19:13 /my/files.har/my/files/dir/b
```

如果 HAR 文件所在的文件系统是默认的文件系统，这就非常直观易懂。但如果想使用在其他文件系统中的 HAR 文件，就需要使用一个不同于正常情况的 URI 路径格式。以下两个指令作用相同，例如：

```
1. % hadoop fs -lsr har:///my/files.har/my/files/dir
2. % hadoop fs -lsr har://hdfs-localhost:8020/my/files.har/my/files/dir
```

注意第二个格式，仍以 har 方案表示一个 HAR 文件系统，但是是由 hdfs 指定基础的文件系统方案，后面加上一个横杠和 HDFS host(localhost)和端口(8020)。我们现在算是明白为什么 HAR 文件必须要有.har 扩展名了。通过查看权限和路径及.har 扩展名的组成，HAR 文件系统将 har URI 转换成为一个基础文件系统的 URI。在本例中是 hdfs://localhost:8020/user/tom/files.har。路径的剩余部分是文件在归档文件中的路径：/user/tom/files/dir。

要想删除一个 HAR 文件，需要使用删除的递归格式，因为对于基础文件系统来说，HAR 文件是一个目录。

```
1. % hadoop fs -rmr /my/files.har
```

2. **3.8.2 不足**

3. 对于 HAR 文件，还需要了解它的一些不足。创建一个归档文件会创建原始文件的一个副本，因此需要与要归档(尽管创建了归档文件后可以删除原始文件)的文件同样大小的磁盘空间。虽然归档的文件能被压缩(HAR 文件在这方面像 tar 文件)，但是目前还不支持档案压缩。

4. 一旦创建，Archives 便不可改变。要增加或移除文件，必须重新创建归档文件。事实上，这对那些写后便不能改的文件来说不是问题，因为它们可以定期成批归档，比如每日或每周。

5. 如前所述，HAR 文件可以用作 MapReduce 的输入。然而，没有归档 InputFormat 可以打包多个文件到一个单一的 MapReduce，所以即使在 HAR 文件中处理许多小文件，也仍然低效的。第 7 章讨论了解决此问题的另一种方法。