第 3 章

Hadoop 分布式文件系统

当数据集的大小超过一台独立的物理计算机的存储能力时，就有必要对它进行分区(partition)并存储到若干台单独的计算机上。管理网络中跨多台计算机存储的文件系统称为分布式文件系统(distributed filesystem)。该系统架构于网络之上，势必会引入网络编程的复杂性，因此分布式文件系统比普通磁盘文件系统更为复杂。例如，使文件系统能够容忍节点故障且不丢失任何数据，就是一个极大的挑战。

Hadoop 自带一个称为 HDFS 的分布式文件系统，即 Hadoop Distributed Filesystem。在非正式文档或旧文档以及配置文件中，有时也简称为 DFS，它们是一回事儿。HDFS 是 Hadoop 的旗舰级文件系统，也是本章的重点，但实际上 Hadoop 是一个综合性的文件系统抽象，因此接下来我们将了解将 Hadoop 与其他存储系统集成的途径，例如本地文件系统和 Amazon S3 系统。

3.1 HDFS 的设计

HDFS 以流式数据访问模式来存储超大文件，运行于商用硬件集群上。[①]让我们仔细看看下面的描述。

- **超大文件** “超大文件”在这里指具有几百 MB、几百 GB 甚至几百 TB

① Robert Chansler 等人在文章“The Hadoop Distributed File System”(Hadoop 分布系统)中详细叙述了 HDFS 的架构(*http://www.aosabook.org/en/hdfs.html*)，该文章内容可以参见 Amy Brown 和 Greg Wilson 等人的文章“The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks”。

大小的文件。目前已经有存储 PB 级数据的 Hadoop 集群了。[①]

- **流式数据访问** HDFS 的构建思路是这样的：一次写入、多次读取是最高效的访问模式。数据集通常由数据源生成或从数据源复制而来，接着长时间在此数据集上进行各种分析。每次分析都将涉及该数据集的大部分数据甚至全部，因此读取整个数据集的时间延迟比读取第一条记录的时间延迟更重要。

- **商用硬件** Hadoop 并不需要运行在昂贵且高可靠的硬件上。它是设计运行在商用硬件(在各种零售店都能买到的普通硬件[②])的集群上的，因此至少对于庞大的集群来说，节点故障的几率还是非常高的。HDFS 遇到上述故障时，被设计成能够继续运行且不让用户察觉到明显的中断。

  同样，那些不适合在 HDFS 上运行的应用也值得研究。目前 HDFS 对某些应用领域并不适合，不过以后可能会有所改进。

- **低时间延迟的数据访问** 要求低时间延迟数据访问的应用，例如几十毫秒范围，不适合在 HDFS 上运行。记住，HDFS 是为高数据吞吐量应用优化的，这可能会以提高时间延迟为代价。目前，对于低延迟的访问需求，HBase(参见第 20 章)是更好的选择。

- **大量的小文件** 由于 namenode 将文件系统的元数据存储在内存中，因此该文件系统所能存储的文件总数受限于 namenode 的内存容量。根据经验，每个文件、目录和数据块的存储信息大约占 150 字节。因此，举例来说，如果有一百万个文件，且每个文件占一个数据块，那至少需要 300 MB 的内存。尽管存储上百万个文件是可行的，但是存储数十亿个文件就超出了当前硬件的能力。[③]

- **多用户写入，任意修改文件** HDFS 中的文件写入只支持单个写入者，而且写操作总是以“只添加”方式在文件末尾写数据。它不支持多个写入者的操作，也不支持在文件的任意位置进行修改。可能以后会支持这些操

① 参见 Konstantin V. Shvachko 和 Arun C. Murthy 在 2008 年 9 月 30 日发表的文章，标题为“Scaling Hadoop to 4000 nodes at Yahoo！”(Yahoo 将 Hadoop 扩展应用到 4000 个节点)，详情可以访问 *http://bit.ly/scaling_hadoop*。

② 详见第 10 章的典型计算机规格。

③ 对于 HDFS 可扩展性限制的阐述，请参见 Konstantin V. Shvachko 在 2010 年 4 月发表的论文，标题为“HDFS Scalability: The limits to growth”(HDFS 的扩展性：增长的极限)，网址为 *http://bit.ly/limits_to_growth*。

作，但它们相对比较低效。

3.2 HDFS 的概念

3.2.1 数据块

每个磁盘都有默认的数据块大小，这是磁盘进行数据读/写的最小单位。构建于单个磁盘之上的文件系统通过磁盘块来管理该文件系统中的块，该文件系统块的大小可以是磁盘块的整数倍。文件系统块一般为几千字节，而磁盘块一般为 512 字节。这些信息(文件系统块大小)对于需要读/写文件的文件系统用户来说是透明的。尽管如此，系统仍然提供了一些工具(如 *df* 和 *fsck*)来维护文件系统，由它们对文件系统中的块进行操作。

HDFS 同样也有块(block)的概念，但是大得多，默认为 128 MB。与单一磁盘上的文件系统相似，HDFS 上的文件也被划分为块大小的多个分块(chunk)，作为独立的存储单元。但与面向单一磁盘的文件系统不同的是，HDFS 中小于一个块大小的文件不会占据整个块的空间(例如，当一个 1MB 的文件存储在一个 128 MB 的块中时，文件只使用 1 MB 的磁盘空间，而不是 128 MB)。如果没有特殊指出，本书中提到的“块”特指 HDFS 中的块。

HDFS 中的块为什么这么大？

HDFS 的块比磁盘的块大，其目的是为了最小化寻址开销。如果块足够大，从磁盘传输数据的时间会明显大于定位这个块开始位置所需的时间。因而，传输一个由多个块组成的大文件的时间取决于磁盘传输速率。

我们来做一个速算，如果寻址时间约为 10 ms，传输速率为 100 MB/s，为了使寻址时间仅占传输时间的 1%，我们要将块大小设置约为 100 MB。默认的块大小实际为 128 MB，但是很多情况下 HDFS 安装时使用更大的块。以后随着新一代磁盘驱动器传输速率的提升，块的大小会被设置得更大。

但是这个参数也不会设置得过大。MapReduce 中的 map 任务通常一次只处理一个块中的数据，因此如果任务数太少(少于集群中的节点数量)，作业的运行速度就会比较慢。

对分布式文件系统中的块进行抽象会带来很多好处。第一个最明显的好处是，一个文件的大小可以大于网络中任意一个磁盘的容量。文件的所有块并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进行存储。事实上，尽管不常见，但对于整个 HDFS 集群而言，也可以仅存储一个文件，该文件的块占满集群中所有的磁盘。

第二个好处是，使用抽象块而非整个文件作为存储单元，大大简化了存储子系统的设计。简化是所有系统的目标，但是这对于故障种类繁多的分布式系统来说尤为重要。将存储子系统的处理对象设置为块，可简化存储管理(由于块的大小是固定的，因此计算单个磁盘能存储多少个块就相对容易)。同时也消除了对元数据的顾虑(块只是要存储的大块数据，而文件的元数据，如权限信息，并不需要与块一同存储，这样一来，其他系统就可以单独管理这些元数据)。

不仅如此，块还非常适合用于数据备份进而提供数据容错能力和提高可用性。将每个块复制到少数几个物理上相互独立的机器上(默认为 3 个)，可以确保在块、磁盘或机器发生故障后数据不会丢失。如果发现一个块不可用，系统会从其他地方读取另一个复本，而这个过程对用户是透明的。一个因损坏或机器故障而丢失的块可以从其他候选地点复制到另一台可以正常运行的机器上，以保证复本的数量回到正常水平(参见 5.1 节对数据完整性的讨论，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一些常用的文件块设置更高的复本数量进而分散集群中的读取负载。

与磁盘文件系统相似，HDFS 中 `fsck` 指令可以显示块信息。例如，执行以下命令将列出文件系统中各个文件由哪些块构成，详情可以参见 11.1.4 节对文件系统检查(fsck)的讨论：

```
% hdfs fsck / -files -blocks
```

3.2.2 namenode 和 datanode

HDFS 集群有两类节点以管理节点-工作节点模式运行，即一个 namenode(管理节点)和多个 datanode(工作节点)。namenode 管理文件系统的命名空间。它维护着文件系统树及整棵树内所有的文件和目录。这些信息以两个文件形式永久保存在本地磁盘上：命名空间镜像文件和编辑日志文件。namenode 也记录着每个文件中各个块所在的数据节点信息，但它并不永久保存块的位置信息，因为这些信息会在系统启动时根据数据节点信息重建。

客户端(client)代表用户通过与 namenode 和 datanode 交互来访问整个文件系统。客户端提供一个类似于 POSIX(可移植操作系统界面)的文件系统接口，因此用户在编程时无需知道 namenode 和 datanode 也可实现其功能。

datanode 是文件系统的工作节点。它们根据需要存储并检索数据块(受客户端或 namenode 调度)，并且定期向 namenode 发送它们所存储的块的列表。

没有 namenode，文件系统将无法使用。事实上，如果运行 namenode 服务的机器毁坏，文件系统上所有的文件将会丢失，因为我们不知道如何根据 datanode 的块重建文件。因此，对 namenode 实现容错非常重要，Hadoop 为此提供两种机制。

第一种机制是备份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 namenode 在多个文件系统上保存元数据的持久状态。这些写操作是实时同步的，且是原子操作。一般的配置是，将持久状态写入本地磁盘的同时，写入一个远程挂载的网络文件系统(NFS)。

另一种可行的方法是运行一个辅助 namenode，但它不能被用作 namenode。这个辅助 namenode 的重要作用是定期合并编辑日志与命名空间镜像，以防止编辑日志过大。这个辅助 namenode 一般在另一台单独的物理计算机上运行，因为它需要占用大量 CPU 时间，并且需要与 namenode 一样多的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，并在 namenode 发生故障时启用。但是，辅助 namenode 保存的状态总是滞后于主节点，所以在主节点全部失效时，难免会丢失部分数据。在这种情况下，一般把存储在 NFS 上的 namenode 元数据复制到辅助 namenode 并作为新的主 namenode 运行。(注意，也可以运行热备份 namenode 代替运行辅助 namenode，具体参见 3.2.5 节对 HDFS 高可用性的讨论。)

关于文件系统镜像和编辑日志的更多讨论，请参见 11.1.1 节。

3.2.3 块缓存

通常 datanode 从磁盘中读取块，但对于访问频繁的文件，其对应的块可能被显式地缓存在 datanode 的内存中，以堆外块缓存(off-heap *block cache*)的形式存在。默认情况下，一个块仅缓存在一个 datanode 的内存中，当然可以针每个文件配置 datanode 的数量。作业调度器(用于 MapReduce、Spark 和其他框架的)通过在缓存块的 datanode 上运行任务，可以利用块缓存的优势提高读操作的性能。例如，连接(join)操作中使用的一个小的查询表就是块缓存的一个很好的候选。

用户或应用通过在缓存池(*cache pool*)中增加一个 *cache directive* 来告诉 namenode 需要缓存哪些文件及存多久。缓存池是一个用于管理缓存权限和资源使用的管理性分组。

3.2.4 联邦 HDFS

namenode 在内存中保存文件系统中每个文件和每个数据块的引用关系，这意味着对于一个拥有大量文件的超大集群来说，内存将成为限制系统横向扩展的瓶颈(参见 10.3.2 节)。在 2.x 发行版本系列中引入的联邦 HDFS 允许系统通过添加 namenode 实现扩展，其中每个 namenode 管理文件系统命名空间中的一部分。例如，一个 namenode 可能管理*/user* 目录下的所有文件，而另一个 namenode 可能管理*/share* 目录下的所有文件。

在联邦环境下，每个 namenode 维护一个命名空间卷(namespace volume)，由命名空间的元数据和一个数据块池(block pool)组成，数据块池包含该命名空间下文件的所有数据块。命名空间卷之间是相互独立的，两两之间并不相互通信，甚至其中一个 namenode 的失效也不会影响由其他 namenode 维护的命名空间的可用性。数据块池不再进行切分，因此集群中的 datanode 需要注册到每个 namenode，并且存储着来自多个数据块池中的数据块。

要想访问联邦 HDFS 集群，客户端需要使用客户端挂载数据表将文件路径映射到 namenode。该功能可以通过 `ViewFileSystem` 和 `viewfs：//URI` 进行配置和管理。

3.2.5 HDFS 的高可用性

通过联合使用在多个文件系统中备份 namenode 的元数据和通过备用 namenode 创建监测点能防止数据丢失，但是依旧无法实现文件系统的高可用性。namenode 依旧存在单点失效(SPOF, single point of failure)的问题。如果 namenode 失效了，那么所有的客户端，包括 MapReduce 作业，均无法读、写或列举(list)文件，因为 namenode 是唯一存储元数据与文件到数据块映射的地方。在这一情况下，Hadoop 系统无法提供服务直到有新的 namenode 上线。

在这样的情况下，要想从一个失效的 namenode 恢复，系统管理员得启动一个拥有文件系统元数据副本的新的 namenode，并配置 datanode 和客户端以便使用这个新的 namenode。新的 namenode 直到满足以下情形才能响应服务：(1)将命名空间的

映像导入内存中；(2)重演编辑日志；(3)接收到足够多的来自 datanode 的数据块报告并退出安全模式。对于一个大型并拥有大量文件和数据块的集群，namenode 的冷启动需要 30 分钟，甚至更长时间。

系统恢复时间太长，也会影响到日常维护。事实上，预期外的 namenode 失效出现概率很低，所以在现实中，计划内的系统失效时间实际更为重要。

Hadoop2 针对上述问题增加了对 HDFS 高可用性(HA)的支持。在这一实现中，配置了一对活动-备用(active-standby) namenode。当活动 namenode 失效，备用 namenode 就会接管它的任务并开始服务于来自客户端的请求，不会有任何明显中断。实现这一目标需要在架构上做如下修改。

- namenode 之间需要通过高可用共享存储实现编辑日志的共享。当备用 namenode 接管工作之后，它将通读共享编辑日志直至末尾，以实现与活动 namenode 的状态同步，并继续读取由活动 namenode 写入的新条目。
- datanode 需要同时向两个 namenode 发送数据块处理报告，因为数据块的映射信息存储在 namenode 的内存中，而非磁盘。
- 客户端需要使用特定的机制来处理 namenode 的失效问题，这一机制对用户是透明的。
- 辅助 namenode 的角色被备用 namenode 所包含，备用 namenode 为活动的 namenode 命名空间设置周期性检查点。

可以从两种高可用性共享存储做出选择：NFS 过滤器或群体日志管理器(QJM，*quorum journal manager*)。QJM 是一个专用的 HDFS 实现，为提供一个高可用的编辑日志而设计，被推荐用于大多数 HDFS 部署中。QJM 以一组日志节点(journal node)的形式运行，每一次编辑必须写入多数日志节点。典型的，有三个 journal 节点，所以系统能够忍受其中任何一个的丢失。这种安排与 ZooKeeper 的工作方式类似，当然必须认识到，QJM 的实现并没使用 ZooKeeper。(然而，值得注意的是，HDFS HA 在选取活动的 namenode 时确实使用了 ZooKeeper 技术，详情参见下一章。)

在活动 namenode 失效之后，备用 namenode 能够快速(几十秒的时间)实现任务接管，因为最新的状态存储在内存中：包括最新的编辑日志条目和最新的数据块映射信息。实际观察到的失效时间略长一点(需要 1 分钟左右)，这是因为系统需要保守确定活动 namenode 是否真的失效了。

在活动 namenode 失效且备用 namenode 也失效的情况下，当然这类情况发生的概率非常低，管理员依旧可以声明一个备用 namenode 并实现冷启动。这类情况并不会比非高可用(non-HA)的情况更差，并且从操作的角度讲这是一个进步，因为上述处理已是一个标准的处理过程并植入 Hadoop 中。

故障切换与规避

系统中有一个称为故障转移控制器(failover controller)的新实体，管理着将活动 namenode 转移为备用 namenode 的转换过程。有多种故障转移控制器，但默认的一种是使用了 ZooKeeper 来确保有且仅有一个活动 namenode。每一个 namenode 运行着一个轻量级的故障转移控制器，其工作就是监视宿主 namenode 是否失效(通过一个简单的心跳机制实现)并在 namenode 失效时进行故障切换。

管理员也可以手动发起故障转移，例如在进行日常维护时。这称为“平稳的故障转移”(*graceful failover*)，因为故障转移控制器可以组织两个 namenode 有序地切换角色。

但在非平稳故障转移的情况下，无法确切知道失效 namenode 是否已经停止运行。例如，在网速非常慢或者网络被分割的情况下，同样也可能激发故障转移，但是先前的活动 namenode 依然运行着并且依旧是活动 namenode。高可用实现做了更进一步的优化，以确保先前活动的 namenode 不会执行危害系统并导致系统崩溃的操作，该方法称为“规避”(fencing)。

同一时间 QJM 仅允许一个 namenode 向编辑日志中写入数据。然而，对于先前的活动 namenode 而言，仍有可能响应并处理客户过时的读请求，因此，设置一个 SSH 规避命令用于杀死 namenode 的进程是一个好主意。当使用 NFS 过滤器实现共享编辑日志时，由于不可能同一时间只允许一个 namenode 写入数据(这也是为什么推荐 QJM 的原因)，因此需要更有力的规避方法。规避机制包括：撤销 namenode 访问共享存储目录的权限（通常使用供应商指定的 NFS 命令）、通过远程管理命令屏蔽相应的网络端口。诉诸的最后手段是，先前活动 namenode 可以通过一个相当形象的称为“一枪爆头”*STONITH*，shoot the other node in the head)的技术进行规避，该方法主要通过一个特定的供电单元对相应主机进行断电操作。

客户端的故障转移通过客户端类库实现透明处理。最简单的实现是通过客户端的配置文件实现故障转移的控制。HDFS URI 使用一个逻辑主机名，该主机名映射到一对 namenode 地址(在配置文件中设置)，客户端类库会访问每一个 namenode 地址直至处理完成。

3.3 命令行接口

现在我们通过命令行交互来进一步认识 HDFS。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。

参照附录 A 中伪分布模式下设置 Hadoop 的说明，我们先在一台机器上运行 HDFS。稍后介绍如何在集群上运行 HDFS，以提供可扩展性与容错性。

在我们设置伪分布配置时，有两个属性项需要进一步解释。第一项是 `fs.defaultFS`，设置为 `hdfs://localhost/`，用于设置 Hadoop 的默认文件系统。[①]文件系统是由 URI 指定的，这里我们已使用 `hdfs` URI 来配置 HDFS 为 Hadoop 的默认文件系统。HDFS 的守护程序通过该属性项来确定 HDFS namenode 的主机及端口。我们将在 localhost 默认的 HDFS 端口 8020 上运行 namenode。这样一来，HDFS 客户端可以通过该属性得知 namenode 在哪里运行进而连接到它。

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本设为 3。在单独一个 datanode 上运行时，HDFS 无法将块复制到 3 个 datanode 上，所以会持续给出块复本不足的警告。设置这个属性之后，上述问题就不会再出现了。

文件系统的基本操作

至此，文件系统已经可以使用了，我们可以执行所有常用的文件系统操作，例如，读取文件，新建目录，移动文件，删除数据，列出目录，等等。可以输入 `hadoop fs -help` 命令获取每个命令的详细帮助文件。

首先从本地文件系统将一个文件复制到 HDFS：

```
% hadoop fs -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`，后者提供了一系列子命令，在这个例子中，我们执行的是`-copyFromLocal`。本地文件 *quangle.txt* 被复制到运行在 localhost 上的 HDFS 实例中，路径为*/user/tom/quangle.txt*。事实上，我们可以简

① Hadoop 1 中，该属性的名称为 `fs.default.name`。Hadoop2 中使用了许多新的属性名称，旧名称均不再使用(详情参见 6.2.2 节)。本书使用的均为新的属性名称。

化命令格式以省略主机的 URI 并使用默认设置，即省略 `hdfs://localhost`，因为该项已在 *core-site.xml* 中指定。

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

我们也可以使用相对路径，并将文件复制到 HDFS 的 home 目录中，本例中为 */user/tom*：

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

我们把文件复制回本地文件系统，并检查是否一致：

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt
% md5 input/docs/quangle.txt quangle.copy.txt
 MD5 (input/docs/quangle.txt) = e7891a2627cf263a079fb0f18256ffb2
 MD5 (quangle.copy.txt) = e7891a2627cf263a079fb0f18256ffb2
```

MD5 键值相同，表明这个文件在 HDFS 之旅中得以幸存并保存完整。

最后，看一下 HDFS 文件列表。我们新建一个目录，看它在列表中怎么显示：

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x - tom supergroup 0 2014-10-04 13:22 books
-rw-r--r-- 1 tom supergroup 119 2014-10-04 13:21 quangle.txt
```

返回的结果信息与 Unix 命令 `ls -l` 的输出结果非常相似，仅有细微差别。第 1 列显示的是文件模式。第 2 列是这个文件的备份数(这在传统 Unix 文件系统是没有的)。由于我们在整个文件系统范围内设置的默认复本数为 1，所以这里显示的也都是 1。这一列的开头目录为空，因为本例中没有使用复本的概念，目录作为元数据保存在 namenode 中，而非 datanode 中。第 3 列和第 4 列显示文件的所属用户和组别。第 5 列是文件的大小，以字节为单位，目录为 0。第 6 列和第 7 列是文件的最后修改日期与时间。最后，第 8 列是文件或目录的名称。

HDFS 中的文件访问权限

针对文件和目录，HDFS 的权限模式与 POSIX 的权限模式非常相似。

一共提供三类权限模式：只读权限(r)、写入权限(w)和可执行权限(x)。读取文件或列出目录内容时需要只读权限。写入一个文件或是在一个目录上新建及删除文件或目录，需要写入权限。对于文件而言，可执行权限可以忽略，因为你

不能在 HDFS 中执行文件(与 POSIX 不同)，但在访问一个目录的子项时需要该权限。

每个文件和目录都有所属用户(owner)、所属组别(group)及模式(mode)。这个模式是由所属用户的权限、组内成员的权限及其他用户的权限组成的。

在默认情况下，Hadoop 运行时安全措施处于停用模式，意味着客户端身份是没有经过认证的。由于客户端是远程的，一个客户端可以在远程系统上通过创建和任一个合法用户同名的账号来进行访问。当然，如果安全设施处于启用模式，这些都是不可能的(详情见 10.4 节关于安全性的有关论述)。无论怎样，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，启用权限控制还是很重要的(这也是默认的配置，参见 `dfs.permissions.enabled` 属性)

如果启用权限检查，就会检查所属用户权限，以确认客户端的用户名与所属用户是否匹配，另外也将检查所属组别权限，以确认该客户端是否是该用户组的成员；若不符，则检查其他权限。

这里有一个超级用户(super-user)的概念，超级用户是 namenode 进程的标识。对于超级用户，系统不会执行任何权限检查。

3.4 Hadoop 文件系统

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` 定义了 Hadoop 中一个文件系统的客户端接口，并且该抽象类有几个具体实现，其中和 Hadoop 紧密相关的见表 3-1。

表 3-1. Hadoop 文件系统

文件系统	URI 方案	Java 实现(都在 org.apache.hadoop 包中)	描述
`Local`	`file`	`fs.LocalFileSystem`	使用客户端校验和的本地磁盘文件系统。使用 `RawLocalFileSystem` 表示无校验和的本地磁盘文件系统。详情参见 5.1.2 节
`HDFS`	`hdfs`	`hdfs.DistributedFileSystem`	Hadoop 的分布式文件系统。将 HDFS 设计成与 MapReduce 结合使用，可以实现高性能

续表

文件系统	URI 方案	Java 实现(都在 org.apache.hadoop 包中)	描述
WebHDFS	Webhdfs	Hdfs.web.WebHdfsFileSystem	基于 HTTP 的文件系统，提供对 HDFS 的认证读/写访问。详情参见 3.4 节相关内容
Secure WebHDFS	swebhdfs	hdfs.web.SWebHdfsFileSystem	WebHDFS 的 HTTPS 版本
HAR	har	fs.HarFileSystem	一个构建在其他文件系统之上用于文件存档的文件系统。Hadoop 存档文件系统通常用于将 HDFS 中的多个文件打包成一个存档文件，以减少 namenode 内存的使用。使用 hadoop 的 achive 命令来创建 HAR 文件
View	viewfs	viewfs.ViewFileSystem	针对其他 Hadoop 文件系统的客户端挂载表。通常用于为联邦 namenode 创建挂载点，详情参见 3.2.4 节
FTP	ftp	fs.ftp.FTPFileSystem	由 FTP 服务器支持的文件系统
S3	S3a	fs.s3a.S3AFileSystem	由 Amazon S3 支持的文件系统。替代老版本的 s3n(S3 原生)实现
Azure	wasb	fs.azure.NativeAzure FileSystem	由 Microsoft Azure 支持的文件系统
Swift	swift	fs.swift.snative. SwiftNativeFileSystem	由 OpenStack Swift 支持的文件系统

Hadoop 对文件系统提供了许多接口，它一般使用 URI 方案来选取合适的文件系统实例进行交互。举例来说，我们在前一小节中遇到的文件系统命令行解释器可以操作所有的 Hadoop 文件系统命令。要想列出本地文件系统根目录下的文件，可以输入以下命令：

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

尽管运行的 MapReduce 程序可以访问任何文件系统(有时也很方便)，但在处理大数据集时，建议你还是选择一个有数据本地优化的分布式文件系统，如 HDFS(参见 2.4 节)。

接口

Hadoop 是用 Java 写的，通过 Java API 可以调用大部分 Hadoop 文件系统的交互操作。例如，文件系统的命令解释器就是一个 Java 应用，它使用 Java 的 FileSystem 类来提供文件系统操作。其他一些文件系统接口也将在本小节中做简

单介绍。这些接口通常与 HDFS 一同使用，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(对于 FTP，有 FTP 客户端；对于 S3，有 S3 工具，等等)，但它们大多数都能用于任何 Hadoop 文件系统。

1. HTTP

Hadoop 以 Java API 的形式提供文件系统访问接口，非 Java 开发的应用访问 HDFS 会很不方便。由 WebHDFS 协议提供的 HTTP REST API 则使得其他语言开发的应用能够更方便地与 HDFS 交互。注意，HTTP 接口比原生的 Java 客户端要慢，所以不到万不得已，尽量不要用它来传输特大数据。

通过 HTTP 来访问 HDFS 有两种方法：直接访问，HDFS 守护进程直接服务于来自客户端的 HTTP 请求；通过代理(一个或多个)访问，客户端通常使用 `DistributedFileSystem` API 访问 HDFS。这两种方法如图 3-1 所示。两者都使用了 WebHDFS 协议。

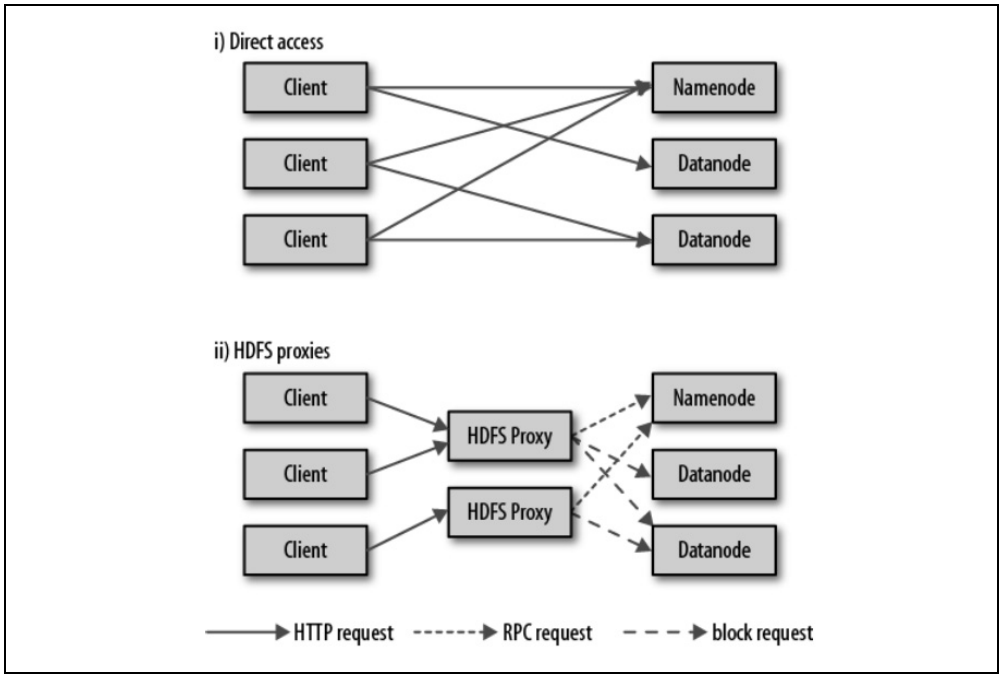

图 3-1. 通过 HTTP 直接访问 HDFS 或者通过多个 HDFS 代理访问 HDFS

在第一种情况中，namenode 和 datanode 内嵌的 web 服务器作为 WebHDFS 的端节点运行。(由于 `dfs.webhdfs.enabled` 被设置为 `true`，WebHDFS 默认是启用状

态。)文件元数据操作由 namenode 管理，文件读(写)操作首先被发往 namenode，由 namenode 发送一个 HTTP 重定向至某个客户端，指示以流方式传输文件数据的目的或源 datanode。

第二种方法依靠一个或者多个独立代理服务器通过 HTTP 访问 HDFS。(由于代理服务是无状态的，因此可以运行在标准的负载均衡器之后。)所有到集群的网络通信都需要经过代理，因此客户端从来不直接访问 namenode 或 datanode。使用代理服务器后可以使用更严格的防火墙策略和带宽限制策略。通常情况下都通过代理服务器，实现在不同数据中心中部署的 Hadoop 集群之间的数据传输，或从外部网络访问云端运行的 Hadoop 集群。

HttpFS 代理提供和 WebHDFS 相同的 HTTP(和 HTTPS)接口，这样客户端能够通过 `webhdfs`(`swebhdfs`) URI 访问这两类接口。HttpFS 代理的启动独立于 namenode 和 datanode 的守护进程，使用 *httpfs.sh* 脚本，默认在一个不同的端口上监听(端口号 14000)。

2. C 语言

Hadoop 提供了一个名为 *libhdfs* 的 C 语言库，该语言库是 Java `FileSystem` 接口类的一个镜像(它被写成访问 HDFS 的 C 语言库，但其实它可以访问任何一个 Hadoop 文件系统)。它使用 Java 原生接口(JNI, Java Native Interface)调用 Java 文件系统客户端。同样还有一个 *libwebhdfs* 库，该库使用了前述章节描述的 WebHDFS 接口。

这个 C 语言 API 与 Java 的 API 非常相似，但它的开发滞后于 Java API，因此目前一些新的特性可能还不支持。可以在 Apache Hapdoop 二进制压缩发行包的 *include* 目录中找到头文件 *hdfs.h*。

Apache Hapdoop 二进制压缩包自带预先编译好的 *libhdfs* 二进制编码，支持 64 位 Linux。但对于其他平台，需要按照源代码树顶层的 *BUILDING.txt* 指南自行编译。

3. NFS

使用 Hadoop 的 NFSv3 网关将 HDFS 挂载为本地客户端的文件系统是可行的。然后你可以使用 Unix 实用程序(如 `ls` 和 `cat`) 与该文件系统交互，上传文件，通过任意一种编程语言调用 POSIX 库来访问文件系统。由于 HDFS 仅能以追加模式写文件，因此可以往文件末尾添加数据，但不能随机修改文件。

关于如何配置和运行 NFS 网关，以及如何从客户端连接网关，可以参考 Hadoop 相关文档资料。

4. FUSE

用户空间文件系统(FUSE, Filesystem in Userspace，)允许将用户空间实现的文件系统作为 Unix 文件系统进行集成。通过使用 Hadoop 的 Fuse-DFS 功能模块，HDFS(或任何一个 Hadoop 文件系统)均可以作为一个标准的本地文件系统进行挂载。Fuse-DFS 是用 C 语言实现的，使用 *libhdfs* 作为访问 HDFS 的接口。在写操作时，Hadoop NFS 网关对于挂载 HDFS 来说是更健壮的解决方案，相比 Fuse-DFS 而言应优先选择。

3.5 Java 接口

在本小节中，我们要深入探索 Hadoop 的 `Filesystem` 类：它是与 Hadoop 的某一文件系统进行交互的 API。[①] 虽然我们主要聚焦于 HDFS 实例，即 `DistributedFileSystem`，但总体来说，还是应该集成 `FileSystem` 抽象类，并编写代码，使其在不同文件系统中可移植。这对测试你编写的程序非常重要，例如，你可以使用本地文件系统中的存储数据快速进行测试。

3.5.1 从 Hadoop URL 读取数据

要从 Hadoop 文件系统读取文件，最简单的方法是使用 `java.net.URL` 对象打开数据流，从中读取数据。具体格式如下：

```
InputStream in = null;
 try {
      in = new URL("hdfs://host/path").openStream();
      // process in
 } finally {
      IOUtils.closeStream(in);
 }
```

让 Java 程序能够识别 Hadoop 的 `hdfs URL` 方案还需要一些额外的工作。这里采用的方法是通过 `FsUrlStreamHandlerFactory` 实例调用 `java.net.URL` 对象的

① 在 Hadoop 2 及后续版本中，新增一个名为 `FileContext` 的文件系统接口，该接口能够更好地处理多文件系统问题(例如，单个 `FileContext` 接口能够解决多文件系统方案)，并且该接口更简明，更一致。然而，`FileSystem` 仍然在广泛使用中。

setURLStreamHandlerFactory()方法。每个 Java 虚拟机只能调用一次这个方法，因此通常在静态方法中调用。这个限制意味着如果程序的其他组件(如不受你控制的第三方组件)已经声明一个 URLStreamHandlerFactory 实例，你将无法使用这种方法从 Hadoop 中读取数据。下一节将讨论另一种备选方法。

范例 3-1 展示的程序以标准输出方式显示 Hadoop 文件系统中的文件，类似于 Unix 中的 cat 命令。

范例 3-1. 通过 URLStreamHandler 实例以标准输出方式显示 Hadoop 文件系统的文件

```
public class URLCat {

  static {
    URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
  }

  public static void main(String[] args) throws Exception {
    InputStream in = null;
    try {
      in = new URL(args[0]).openStream();
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

我们可以调用 Hadoop 中简洁的 IOUtils 类，并在 finally 子句中关闭数据流，同时也可以在输入流和输出流之间复制数据(本例中为 System.out)。copyBytes 方法的最后两个参数，第一个设置用于复制的缓冲区大小，第二个设置复制结束后是否关闭数据流。这里我们选择自行关闭输入流，因而 System.out 不必关闭输入流。

下面是一个运行示例：[①]

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop URLCat 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.
```

① 这段文字来自爱德华·李尔(Edward Lear)的诗歌“The Quangle Wangle’s Hat”。

3.5.2 通过 FileSystem API 读取数据

正如前一小节所解释的，有时根本不可能在应用中设置 URLStreamHandlerFactory 实例。在这种情况下，我们需要用 FileSystem API 来打开一个文件的输入流。

Hadoop 文件系统中通过 Hadoop Path 对象(而非 java.io.File 对象，因为它的语义与本地文件系统联系太紧密)来代表文件。可以将路径视为一个 Hadoop 文件系统 URI，如 hdfs://localhost/user/tom/quangle.txt。

FileSystem 是一个通用的文件系统 API，所以第一步是检索我们需要使用的文件系统实例，这里是 HDFS。获取 FileSystem 实例有下面这几个静态工厂方法：

```
public static FileSystem get(Configuration conf) throws IOException
Public static FileSystem get(URI uri, Configuration conf) throws IOException
public static FileSystem get(URI uri, Configuration conf, String user)
  throws IOException
```

Configuration 对象封装了客户端或服务器的配置，通过设置配置文件读取类路径来实现(如 *etc/hadoop/core-site.xml*)。第一个方法返回的是默认文件系统(在 *core-site.xml* 中指定的，如果没有指定，则使用默认的本地文件系统)。第二个方法通过给定的 URI 方案和权限来确定要使用的文件系统，如果给定 URI 中没有指定方案，则返回默认文件系统。第三，作为给定用户来访问文件系统，对安全来说是至关重要。详情可以参见 10.4 节。

在某些情况下，你可能希望获取本地文件系统的运行实例，此时你可以使用的 getLocal()方法很方便地获取。

```
public static LocalFileSystem getLocal(Configuration conf) throws IOException
```

有了 FileSystem 实例之后，我们调用 open()函数来获取文件的输入流：

```
Public FSDataInputStream open(Path f) throws IOException
Public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException
```

第一个方法使用默认的缓冲区大小 4 KB。

最后，我们重写范例 3-1，得到范例 3-2。

范例 3-2. 直接使用 FileSystem 以标准输出格式显示 Hadoop 文件系统中的文件

```
public class FileSystemCat {
  public static void main(String[] args) throws Exception {
    String uri = args[0];
```

```
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    InputStream in = null;
    try {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

程序运行结果如下：

```
% hadoop FileSystemCat 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.
```

FSDataInputStream 对象

实际上，FileSystem 对象中的 open()方法返回的是 FSDataInputStream 对象，而不是标准的 java.io 类对象。这个类是继承了 java.io.DataInputStream 的一个特殊类，并支持随机访问，由此可以从流的任意位置读取数据。

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

public class FSDataInputStream extends DataInputStream
      implements Seekable, PositionedReadable {
      // implementation elided
 }
```

Seekable 接口支持在文件中找到指定位置，并提供一个查询当前位置相对于文件起始位置偏移量(getPos())的查询方法：

```
public interface Seekable {
  void seek(long pos) throws IOException;
  long getPos() throws IOException;
}
```

调用 seek()来定位大于文件长度的位置会引发 IOException 异常。与 java.io.InputStream 的 skip()不同，seek()可以移到文件中任意一个绝对位置，skip()则只能相对于当前位置定位到另一个新位置。

范例 3-3 是对范例 3-2 的简单扩展，它将一个文件写入标准输出两次：在一次写完

之后，定位到文件的起始位置再次以流方式读取该文件并输出。

范例 3-3. 使用 seek()方法，将 Hadoop 文件系统中的一个文件在标准输出上显示两次

```
public class FileSystemDoubleCat {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    FSDataInputStream in = null;
    try {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in, System.out, 4096, false);
      in.seek(0); // go back to the start of the file
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

在一个小文件上运行的结果如下：

```
% 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 类也实现了 PositionedReadable 接口，从一个指定偏移量处读取文件的一部分：

```
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()方法从文件的指定 position 处读取至多为 length 字节的数据并存入缓冲区 buffer 的指定偏离量 offset 处。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 length 长度。readFully()方法将指定

length 长度的字节数数据读取到 buffer 中(或在只接受 buffer 字节数组的版本中，读取 buffer.length 长度字节数据)，除非已经读到文件末尾，这种情况下将抛出 EOFException 异常。

所有这些方法会保留文件当前偏移量，并且是线程安全的(FSDataInputStrean 并不是为并发访问设计的，因此最好为此新建多个实例)，因此它们提供了在读取文件的主体时，访问文件其他部分(可能是元数据)的便利方法。

最后务必牢记，seek()方法是一个相对高开销的操作，需要慎重使用。建议用流数据来构建应用的访问模式(比如使用 MapReduce)，而非执行大量 seek()方法。

3.5.3 写入数据

FileSystem 类有一系列新建文件的方法。最简单的方法是给准备建的文件指定一个 Path 对象，然后返回一个用于写入数据的输出流：

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

此方法有多个重载版本，允许我们指定是否需要强制覆盖现有的文件、文件备份数量、写入文件时所用缓冲区大小、文件块大小以及文件权限。

create()方法能够为需要写入且当前不存在的文件创建父目录。尽管这样很方便，但有时并不希望这样。如果希望父目录不存在就导致文件写入失败，则应该先调用 exists()方法检查父目录是否存在。另一种方案是使用 FileContext，允许你可以控制是否创建父目录。

还有一个重载方法 Progressable 用于传递回调接口，如此一来，可以把数据写入 datanode 的进度通知给应用：

```
package org.apache.hadoop.util;

public interface Progressable {
  public void progress();
}
```

另一种新建文件的方法是使用 append()方法在一个现有文件末尾追加数据(还有其他一些重载版本)：

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

这样的追加操作允许一个 writer 打开文件后在访问该文件的最后偏移量处追加数据。有了这个 API，某些应用可以创建无边界文件，例如，应用可以在关闭日志

文件之后继续追加日志。该追加操作是可选的，并非所有 Hadoop 文件系统都实现了该操作。例如，HDFS 支持追加，但 S3 文件系统就不支持。

范例 3-4 显示了如何将本地文件复制到 Hadoop 文件系统。每次 Hadoop 调用 progress()方法时，也就是每次将 64 KB 数据包写入 datanode 管线后，打印一个时间点来显示整个运行过程。注意，这个操作并不是通过 API 实现的，因此 Hadoop 后续版本能否执行该操作，取决于该版本是否修改过上述操作。API 只是让你知道“正在发生什么事情”。

范例 3-4. 将本地文件复制到 Hadoop 文件系统

```
public class FileCopyWithProgress {
  public static void main(String[] args) throws Exception {
    String localSrc = args[0];
    String dst = args[1];
    InputStream in = new BufferedInputStream(new FileInputStream(localSrc));

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(dst), conf);
    OutputStream out = fs.create(new Path(dst), new Progressable() {
      public void progress() {
        System.out.print(".");
      }
    });

    IOUtils.copyBytes(in, out, 4096, true);
  }
}
```

典型应用如下：

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

目前，其他 Hadoop 文件系统写入文件时均不调用 progress()方法。后面几章将展示进度对 MapReduce 应用的重要性。

FSDataOutputStream 对象

FileSystem 实例的 create()方法返回 FSDataOutputStream 对象，与 FSDataInputStream 类相似，它也有一个查询文件当前位置的方法：

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

public class FSDataOutputStream extends DataOutputStream implements Syncable {

  public long getPos() throws IOException {
```

```
      // implementation elided
    }

    // implementation elided
  }
```

但与 FSDataInputStream 类不同的是，FSDataOutputStream 类不允许在文件中定位。这是因为 HDFS 只允许对一个已打开的文件顺序写入，或在现有文件的末尾追加数据。换句话说，它不支持在除文件末尾之外的其他位置进行写入，因此，写入时定位就没有什么意义。

3.5.4 目录

Filesystem 实例提供了创建目录的方法：

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

这个方法可以一次性新建所有必要但还没有的父目录，就像 java.io.File 类的 mkdirs()方法。如果目录(以及所有父目录)都已经创建成功，则返回 true。

通常，你不需要显式创建一个目录，因为调用 create()方法写入文件时会自动创建父目录。

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都是提供其目录结构浏览和检索它所存文件和目录相关信息的功能。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、复本、修改时间、所有者以及权限信息。

FileSystem 的 getFileStatus()方法用于获取文件或目录的 FileStatus 对象。范例 3-5 显示了它的用法。

范例 3-5. 展示文件状态信息

```
public class ShowFileStatusTest {

  private MiniDFSCluster cluster; // use an in-process HDFS cluster for testing
  private FileSystem fs;

  @Before
  public void setUp() throws IOException {
    Configuration conf = new Configuration();
    if (System.getProperty("test.build.data") == null) {
      System.setProperty("test.build.data", "/tmp");
```

```
    }
    cluster = new MiniDFSCluster.Builder(conf).build();
    fs = cluster.getFileSystem();
    OutputStream out = fs.create(new Path("/dir/file"));
    out.write("content".getBytes("UTF-8"));
    out.close();
  }

  @After
  public void tearDown() throws IOException {
    if (fs != null) { fs.close(); }
    if (cluster != null) { cluster.shutdown(); }
  }

  @Test(expected = FileNotFoundException.class)
  public void throwsFileNotFoundForNonExistentFile() throws IOException {
    fs.getFileStatus(new Path("no-such-file"));
  }

  @Test
  public void fileStatusForFile() throws IOException {
    Path file = new Path("/dir/file");
    FileStatus stat = fs.getFileStatus(file);
    assertThat(stat.getPath().toUri().getPath(), is("/dir/file"));
    assertThat(stat.isDirectory(), is(false));
    assertThat(stat.getLen(), is(7L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 1));
    assertThat(stat.getBlockSize(), is(128 * 1024 * 1024L));
    assertThat(stat.getOwner(), is(System.getProperty("user.name")));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rw-r--r--"));
  }

  @Test
  public void fileStatusForDirectory() throws IOException {
    Path dir = new Path("/dir");
    FileStatus stat = fs.getFileStatus(dir);
    assertThat(stat.getPath().toUri().getPath(), is("/dir"));
    assertThat(stat.isDirectory(), is(true));
    assertThat(stat.getLen(), is(0L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 0));
    assertThat(stat.getBlockSize(), is(0L));
    assertThat(stat.getOwner(),is(System.getProperty("user.name")));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rwxr-xr-x"));
  }

}
```

如果文件或目录均不存在，会抛出一个 `FileNotFoundException` 异常。但如果只是想检查文件或目录是否存在，那么调用 `exists()`方法会更方便：

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

2. 列出文件

查找一个文件或目录相关的信息很实用，但通常还需要能够列出目录中的内容。这就是 FileSystem 的 listStatus()方法的功能：

```
public FileStatus[] listStatus(Path f) throws IOException
public FileStatus[] listStatus(Path f, PathFilter filter) throws IOException
public FileStatus[] listStatus(Path[] files) throws IOException
public FileStatus[] listStatus(Path[] files, PathFilter filter) throws IOException
```

当传入的参数是一个文件时，它会简单转变成以数组方式返回长度为 1 的 FileStatus 对象。当传入参数是一个目录时，则返回 0 或多个 FileStatus 对象，表示此目录中包含的文件和目录。

它的重载方法允许使用 PathFilter 来限制匹配的文件和目录，可以参见 3.5.5 节提供的例子。最后，如果指定一组路径，其执行结果相当于依次轮流传递每条路径并对其调用 listStatus()方法，再将 FileStatus 对象数组累积存入同一数组中，但该方法更为方便。在从文件系统树的不同分支构建输入文件列表时，这是很有用的。范例 3-6 简单显示了这个方法。注意 Hadoop 的 FileUtil 中 stat2Paths()方法的使用，它将一个 FileStatus 对象数组转换为一个 Path 对象数组。

范例 3-6. 显示 Hadoop 文件系统中一组路径的文件信息

```
public class ListStatus {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);

    Path[] paths = new Path[args.length];
    for (int i = 0; i < paths.length; i++) {
      paths[i] = new Path(args[i]);
    }
    FileStatus[] status = fs.listStatus(paths);
    Path[] listedPaths = FileUtil.stat2Paths(status);
    for (Path p : listedPaths) {
      System.out.println(p);
    }
  }
}
```

我们可以用这个程序来显示一组路径集目录列表的并集：

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

3. 文件模式

在单个操作中处理一批文件是一个很常见的需求。例如，一个用于处理日志的MapReduce 作业可能需要分析一个月内包含在大量目录中的日志文件。在一个表达式中使用通配符来匹配多个文件是比较方便的，无需列举每个文件和目录来指定输入，该操作称为“通配”(globbing)。Hadoop 为执行通配提供了两个FileSystem 方法：

```
public FileStatus[] globStatus(Path pathPattern) throws IOException
public FileStatus[] globStatus(Path pathPattern, PathFilter filter)
    throws IOException
```

globStatus()方法返回路径格式与指定模式匹配的所有 FileStatus 对象组成的数组，并按路径排序。PathFilter 命令作为可选项可以进一步对匹配结果进行限制。

Hadoop 支持的通配符与 Unix bash shell 支持的相同(参见表 3-2)。

表 3-2. 通配符及其含义

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	匹配{a,b}集合中的一个字符
[^ab]	非字符类	匹配非{a,b}集合中的一个字符
[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 中的一个的表达式
\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，2007 年最后一天的日志文件就会保存在名为*/2007/12/31* 的目录中。假设整个文件列表如下所示：

```
/
├── 2007/
│   └── 12/
│       ├── 30/
│       └── 31/
└── 2008/
    └── 01/
        ├── 01/
        └── 02/
```

一些文件通配符及其扩展如下所示。

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

4. PathFilter 对象

通配符模式并不总能够精确地描述我们想要访问的文件集。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，以编程方式控制通配符：

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

public interface PathFilter {
   boolean accept(Path path);
}
```

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

范例 3-7 显示了 PathFilter 用于排除匹配正则表达式的路径。

范例 3-7. PathFilter 用于排除匹配正则表达式的路径

```
public class RegexExcludePathFilter implements PathFilter {

  private final String regex;

  public RegexExcludePathFilter(String regex) {
    this.regex = regex;
  }

  public boolean accept(Path path) {
    return !path.toString().matches(regex);
  }
}
```

这个过滤器只传递不匹配于正则表达式的文件。在通配符选出一组需要包含的初始文件之后，过滤器可优化其结果。如下示例将扩展到/2007/12/30：

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

以 Path 为代表，过滤器只能作用于文件名。不能针对文件的属性(例如创建时间)来构建过滤器。但是，过滤器却能实现通配符模式和正则表达式都无法完成的匹配任务。例如，如果将文件存储在按照日期排列的目录结构中(如前一节中讲述的那样)，则可以写一个 Pathfilter 选出给定时间范围内的文件。

3.5.6 删除数据

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

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

如果 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，非空目录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 剖析文件读取

为了了解客户端及与之交互的 HDFS、namenode 和 datanode 之间的数据流是什么样的，我们可参考图 3-2，该图显示了在读取文件时事件的发生顺序。

客户端通过调用 FileSyste 对象的 open()方法来打开希望读取的文件，对于

HDFS 来说，这个对象是 DistributedFileSystem 的一个实例(图 3-2 中的步骤 1)。DistributedFileSystem 通过使用远程过程调用(RPC)来调用 namenode，以确定文件起始块的位置(步骤 2)。对于每一个块，namenode 返回存有该块副本的 datanode 地址。此外，这些 datanode 根据它们与客户端的距离来排序(根据集群的网络拓扑；参见 3.6.1 节的补充材料)。如果该客户端本身就是一个 datanode (比如，在一个 MapReduce 任务中)，那么该客户端将会从保存有相应数据块复本的本地 datanode 读取数据(参见图 2-2 及 10.3.5 节)。

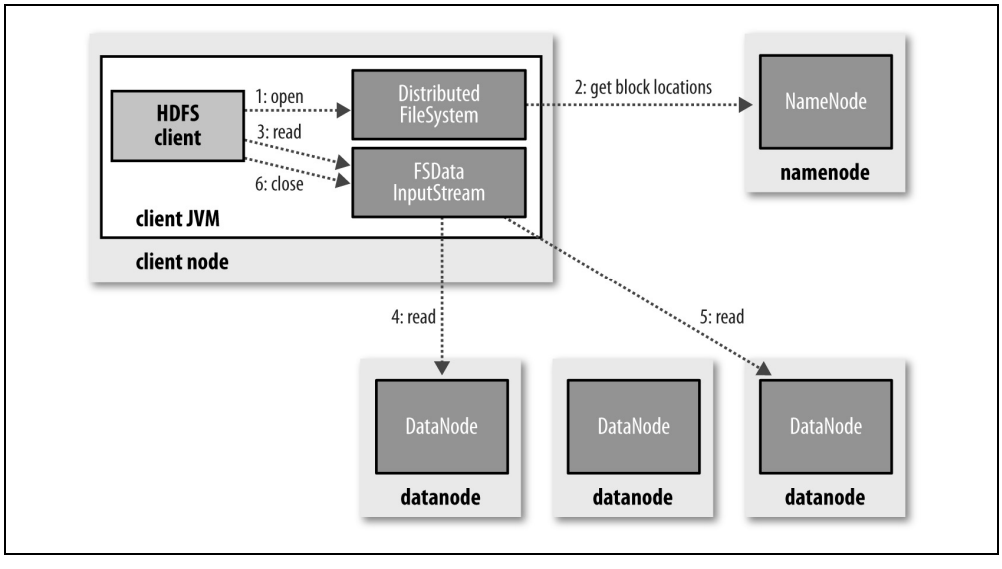

图 3-2. 客户端读取 HDFS 中的数据

DistributedFileSystem 类返回一个 FSDataInputStream 对象(一个支持文件定位的输入流)给客户端以便读取数据。FSDataInputStream 类转而封装 DFSInputStream 对象，该对象管理着 datanode 和 namenode 的 I/O。

接着，客户端对这个输入流调用 read()方法(步骤 3)。存储着文件起始几个块的 datanode 地址的 DFSInputStream 随即连接距离最近的文件中第一个块所在的 datanode。通过对数据流反复调用 read()方法，可以将数据从 datanode 传输到客户端(步骤 4)。到达块的末端时，DFSInputStream 关闭与该 datanode 的连接，然后寻找下一个块的最佳 datanode(步骤 5)。所有这些对于客户端都是透明的，在客户看来它一直在读取一个连续的流。

客户端从流中读取数据时，块是按照打开 DFSInputStream 与 datanode 新建连接的顺序读取的。它也会根据需要询问 namenode 来检索下一批数据块的 datanode 的位置。一旦客户端完成读取，就对 FSDataInputStream 调用 close()方法(步骤 6)。

在读取数据的时候，如果 DFSInputStream 在与 datanode 通信时遇到错误，会尝试从这个块的另外一个最邻近 datanode 读取数据。它也记住那个故障 datanode，以保证以后不会反复读取该节点上后续的块。DFSInputStream 也会通过校验和确认从 datanode 发来的数据是否完整。如果发现有损坏的块，DFSInputStream 会试图从其他 datanode 读取其复本，也会将被损坏的块通知给 namenode。

这个设计的一个重点是，客户端可以直接连接到 datanode 检索数据，且 namenode 告知客户端每个块所在的最佳 datanode。由于数据流分散在集群中的所有 datanode，所以这种设计能使 HDFS 扩展到大量的并发客户端。同时，namenode 只需要响应块位置的请求(这些信息存储在内存中，因而非常高效)，无需响应数据请求，否则随着客户端数量的增长，namenode 会很快成为瓶颈。

网络拓扑与 Hadoop

在本地网络中，两个节点被称为“彼此近邻”是什么意思？在海量数据处理中，其主要限制因素是节点之间数据的传输速率——带宽很稀缺。这里的想法是将两个节点间的带宽作为距离的衡量标准。

不用衡量节点之间的带宽，实际上很难实现(它需要一个稳定的集群，并且在集群中两两节点对数量是节点数量的平方)，adoop 为此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它们到最近共同祖先的距离总和。该树中的层次是没有预先设定的，但是相对于数据中心、机架和正在运行的节点，通常可以设定等级。具体想法是针对以下每个场景，可用带宽依次递减：

- 同一节点上的进程
- 同一机架上的不同节点
- 同一数据中心中不同机架上的节点
- 不同数据中心中的节点[①]

例如，假设有数据中心 *d1* 机架 *r1* 中的节点 *n1*。该节点可以表示为/*d1*/*r1*/*n1*。利用这种标记，这里给出四种距离描述：

① 在本书写作期间，Hadoop 仍然不适合跨数据中心运行。

- *distance(/d1/r1/n1, /d1/r1/n1)*=0(同一节点上的进程)
- *distance(/d1/r1/n1, /d1/r1/n2)*=2(同一机架上的不同节点)
- *distance(/d1/r1/n1, /d1/r2/n3)*=4(同一数据中心中不同机架上的节点)
- *distance(/d1/r1/n1, /d2/r3/n4)*=6(不同数据中心中的节点)

示意图参见图 3-3(数学爱好者会注意到，这是一个测量距离的例子)。

最后，我们必须意识到 Hadoop 无法自动发现你的网络拓扑结构。它需要一些帮助(我们将在 10.1.2 节的“网络拓扑”中讨论如何配置网络拓扑)。不过在默认情况下，假设网络是扁平化的只有一层，或换句话说，所有节点都在同一数据中心的同一机架上。规模小的集群可能如此，不需要进一步配置。

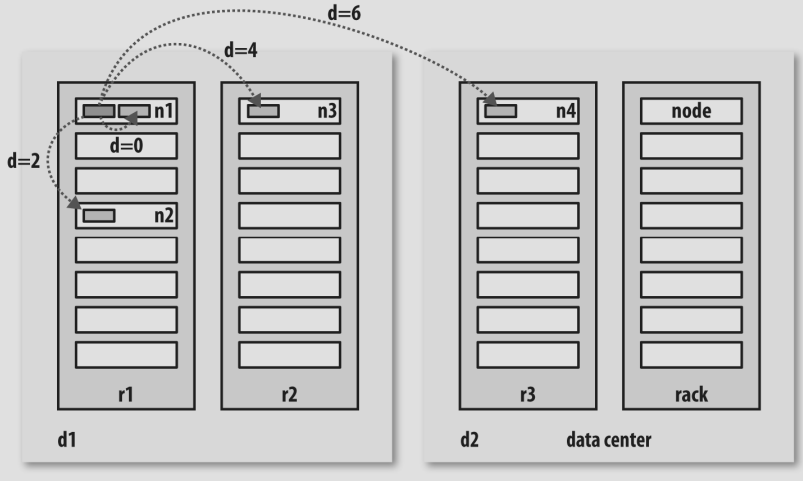

图 3-3. Hadoop 中的网络距离

3.6.2 剖析文件写入

接下来我们看看文件是如何写入 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 的一致模型。

我们要考虑的情况是如何新建一个文件，把数据写入该文件，最后关闭该文件。如图 3-4 所示。

客户端通过对 `DistributedFileSystem` 对象调用 `create()`来新建文件(图 3-4 中的步骤 1)。`DistributedFileSystem` 对 namenode 创建一个 RPC 调用，在文件

系统的命名空间中新建一个文件，此时该文件中还没有相应的数据块(步骤 2)。namenode 执行各种不同的检查以确保这个文件不存在以及客户端有新建该文件的权限。如果这些检查均通过，namenode 就会为创建新文件记录一条记录；否则，文件创建失败并向客户端抛出一个 IOException 异常。DistributedFileSystem 向客户端返回一个 FSDataOutputStream 对象，由此客户端可以开始写入数据。就像读取事件一样，FSDataOutputStream 封装一个 DFSoutPutstream 对象，该对象负责处理 datanode 和 namenode 之间的通信。

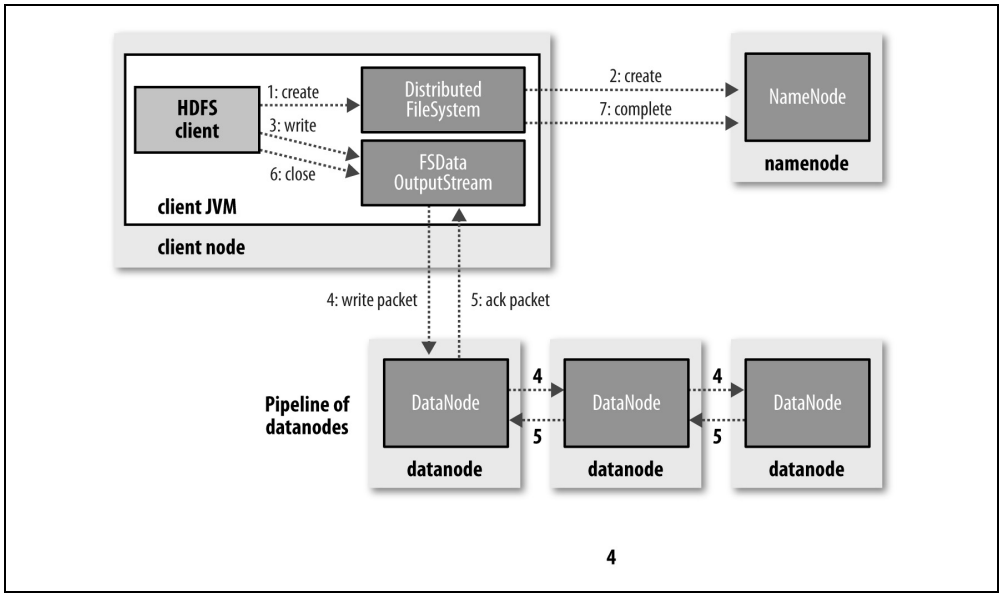

图 3-4. 客户端将数据写入 HDFS

在客户端写入数据时(步骤 3)，DFSOutputStream 将它分成一个个的数据包，并写入内部队列，称为“数据队列”(data queue)。DataStreamer 处理数据队列，它的责任是挑选出适合存储数据复本的一组 datanode，并据此来要求 namenode 分配新的数据块。这一组 datanode 构成一个管线——我们假设复本数为 3，所以管线中有 3 个节点。DataStreamer 将数据包流式传输到管线中第 1 个 datanode，该 datanode 存储数据包并将它发送到管线中的第 2 个 datanode。同样，第 2 个 datanode 存储该数据包并且发送给管线中的第 3 个(也是最后一个)datanode (步骤 4)。

DFSOutputStream 也维护着一个内部数据包队列来等待 datanode 的收到确认回执，称为“确认队列”(ack queue)。收到管道中所有 datanode 确认信息后，该数

据包才会从确认队列删除(步骤 5)。

如果任何 datanode 在数据写入期间发生故障，则执行以下操作(对写入数据的客户端是透明的)。首先关闭管线，确认把队列中的所有数据包都添加回数据队列的最前端，以确保故障节点下游的 datanode 不会漏掉任何一个数据包。为存储在另一正常 datanode 的当前数据块指定一个新的标识，并将该标识传送给 namenode，以便故障 datanode 在恢复后可以删除存储的部分数据块。从管线中删除故障 datanode，基于两个正常 datanode 构建一条新管线。余下的数据块写入管线中正常的 datanode。namenode 注意到块复本量不足时，会在另一个节点上创建一个新的复本。后续的数据块继续正常接受处理。

在一个块被写入期间可能会有多个 datanode 同时发生故障，但非常少见。只要写入了 `dfs.namenode.replication.min` 的复本数(默认为 `1`)，写操作就会成功，并且这个块可以在集群中异步复制，直到达到其目标复本数(`dfs.replication` 的默认值为 3)。

客户端完成数据的写入后，对数据流调用 `close()`方法(步骤 6)。该操作将剩余的所有数据包写入 datanode 管线，并在联系到 namenode 告知其文件写入完成之前，等待确认(步骤 7)。namenode 已经知道文件由哪些块组成(因为 `Datastreamer` 请求分配数据块)，所以它在返回成功前只需要等待数据块进行最小量的复制。

复本怎么放

namenode 如何选择在哪个 datanode 存储**复本**(replica)？这里需要对可靠性、写入带宽和读取带宽进行权衡。例如，把所有复本都存储在一个节点损失的写入带宽最小(因为复制管线都是在同一节点上运行)，但这并不提供真实的冗余(如果节点发生故障，那么该块中的数据会丢失)。同时，同一机架上服务器间的读取带宽是很高的。另一个极端，把复本放在不同的数据中心可以最大限度地提高冗余，但带宽的损耗非常大。即使在同一数据中心(到目前为止，所有 Hadoop 集群均运行在同一数据中心内)，也有多种可能的数据布局策略。

Hadoop 的默认布局策略是在运行客户端的节点上放第 1 个复本 (如果客户端运行在集群之外，就随机选择一个节点，不过系统会避免挑选那些存储太满或太忙的节点)。第 2 个复本放在与第一个不同且随机另外选择的机架中节点上(离架)。第 3 个复本与第 2 个复本放在同一个机架上，且随机选择另一个节点。其他复本放在集群中随机选择的节点上，不过系统会尽量避免在同一个的机架上放太多复本。

一旦选定复本的放置位置，就根据网络拓扑创建一个管线。如果复本数为 3，则有图 3-5 所示的管线。

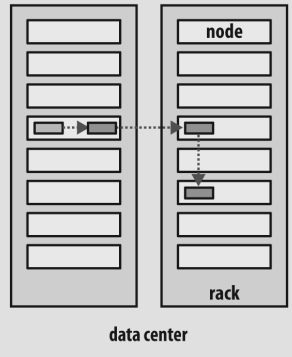

图 3-5. 一个典型的复本管线

总的来说，这一方法不仅提供很好的稳定性(数据块存储在两个机架中)并实现很好的负载均衡，包括写入带宽(写入操作只需要遍历一个交换机)、读取性能(可以从两个机架中选择读取)和集群中块的均匀分布(客户端只在本地机架上写入一个块)。

3.6.3 一致模型

文件系统的一致模型(coherency model)描述了文件读/写的数据可见性。HDFS 为性能牺牲了一些 POSIX 要求，因此一些操作与你期望的可能不同。

新建一个文件之后，它能在文件系统的命名空间中立即可见，如下所示：

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

但是，写入文件的内容并不保证能立即可见，即使数据流已经刷新并存储。所以文件长度显示为 0：

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

当写入的数据超过一个块后，第一个数据块对新的 reader 就是可见的。之后的块

也不例外。总之，当前正在写入的块对其他 reader 不可见。

HDFS 提供了一种强行将所有缓存刷新到 datanode 中的手段，即对 FSDataOutputStream 调用 hflush()方法。当 hflush ()方法返回成功后，对所有新的 reader 而言，HDFS 能保证文件中到目前为止写入的数据均到达所有 datanode 的写入管道并且对所有新的 reader 均可见：

```
Path p = new Path("p");
FSDataOutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.hflush();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

注意，hflush()不保证 datanode 已经将数据写到磁盘上，仅确保数据在 datanode 的内存中(因此，如果数据中心断电，数据会丢失)。为确保数据写入到磁盘上，可以用 hsync()替代[①]。

hsync()操作类似于 POSIX 中的 fsync()系统调用，该调用提交的是一个文件描述符的缓冲数据。例如，利用标准 Java API 数据写入本地文件，我们能够在刷新数据流且同步之后看到文件内容：

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

在 HDFS 中关闭文件其实隐含了执行 hflush()方法：

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

对应用设计的重要性

这个一致模型和设计应用程序的具体方法息息相关。如果不调用 hflush()或 hsync()方法，就要准备好在客户端或系统发生故障时可能会丢失数据块。对很多应用来说，这是不可接受的，所以需要在适当的地方调用 hflush()方法，例如

① 在 Hadoop 1.x 中，hflush()被称为 sync()，hsync()不存在。

在写入一定的记录或字节之后。尽管 hflush()操作被设计成尽量减少 HDFS 负载，但它有许多额外的开销(hsync()的开销更大)，所以在数据鲁棒性和吞吐量之间就会有所取舍。怎样权衡与具体的应用相关，通过度量应用程序以不同频率调用 hflush()或 hsync()时呈现出的性能，最终选择一个合适的调用频率。

3.7 通过 distcp 并行复制

前面着重介绍单线程访问的 HDFS 访问模型。例如，通过指定文件通配符，可以对一组文件进行处理，但是为了提高性能，需要写一个程序来并行处理这些文件。Hadoop 自带一个有用程序 *distcp*，该程序可以并行从 Hadoop 文件系统中复制大量数据，也可以将大量数据复制到 Hadoop 中。

Distcp 的一种用法是替代 hadoop fs -cp。例如，我们可以将文件复制到另一个文件中：[①]

```
% hadoop distcp file1 file2
```

也可以复制目录：

```
% hadoop distcp dir1 dir2
```

如果 *dir2* 不存在，将新建 *dir2*，目录 *dir1* 的内容全部复制到 *dir2* 下。可以指定多个源路径，所有源路径下的内容都将被复制到目标路径下。

如果 *dir2* 已经存在，那么目录 *dir1* 将被复制到 *dir2* 下，形成目录结构 *dir2/dir1*。如果这不是你所需的，你可以补充使用-overwrite 选项，在保持同样的目录结构的同时强制覆盖原有文件。你也可以使用-update 选项，仅更新发生变化的文件。用一个示例可以更好解释这个过程。如果我们修改了 *dir1* 子树中一个文件，我们能够通过运行以下命令将修改同步到 *dir2* 中：

```
% hadoop distcp -update dir1 dir2
```

如果不确定 *discp* 操作的效果，最好先在一个小的测试目录树下试运行。

① 即使对于单个文件复制，由于 hadoop fs -cp 通过运行命令的客户端进行文件复制，因此更倾向于使用 *distcp* 变种复制大文件。

distcp 是作为一个 MapReduce 作业来实现的，该复制作业是通过集群中并行运行的 map 来完成。这里没有 reducer。每个文件通过一个 map 进行复制，并且 *distcp* 试图为每一个 map 分配大致相等的数据来执行，即把文件划分为大致相等的块。默认情况下，将近 20 个 map 被使用，但是可以通过为 *distcp* 指定`-m` 参数来修改 map 的数目。

关于 distcp 的一个常见使用实例是在两个 HDFS 集群间传送数据。例如，以下命令在第二个集群上为第一个集群*/foo* 目录创建了一个备份：

```
% hadoop distcp -update -delete -p hdfs://namenode1/foo hdfs://namenode2/foo
```

`-delete` 选项使得 *distcp* 可以删除目标路径中任意没在源路径中出现的文件或目录，`-P` 选项意味着文件状态属性如权限、块大小和复本数被保留。当你运行不带参数的 *distcp* 时，能够看到准确的用法。

如果两个集群运行的是 HDFS 的不兼容版本，你可以将 `webhdfs` 协议用于它们之间的 *distcp*：

```
% hadoop distcp webhdfs://namenode1:50070/foo webhdfs://namenode2:50070/foo
```

另一个变种是使用 HttpFs 代理作为 *distcp* 源或目标(又一次使用了 `webhdfs` 协议)，这样具有设置防火墙和控制带宽的优点，详情参见 3.4 节对 HTTP 的讨论。

保持 HDFS 集群的均衡

向 HDFS 复制数据时，考虑集群的均衡性是相当重要的。当文件块在集群中均匀分布时，HDFS 能达到最佳工作状态，因此你想确保 *distcp* 不会破坏这点。例如，如果将`-m` 选项指定为 1，即由一个 map 来执行复制作业，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复本将存储到运行 map 的节点上(直到磁盘被填满)。第二和第三个复本将分散在集群中，但这一个节点是不均衡的。将 map 的数量设定为多于集群中节点的数量，可以避免这个问题。鉴于此，最好首先使用默认的每个节点 20 个 map 来运行 *distcp* 命令。

然而，这也并不总能阻止集群的不均衡。也许想限制 map 的数量以便另外一些节点可以运行其他作业。若是这样，可以用均衡器(balancer)工具(参见 11.1.4 节对均衡器的讨论)，进而改善集群中块分布的均匀程度。