# 3.1 HDFS的设计（重点）

1.背景：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，使文件系统能容忍节点故障且不丢失任何数据。

2.Hadoop自带一个称为HDFS的分布式文件系统，即 Hadoop Distributed Filesystem。

3.HDFS以流式数据访问模式来存储超大文件，运用于商用硬件集群上。

1. 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大小的文件**。目前已经有存储PB级数据的Hadoop集群了。
2. 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**。
3. 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。
4. 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是更好的选择。
5. 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，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300MB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。
6. 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**。

# 3.2 HDFS的概念

## 3.2.1 数据块（重点）

1.**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。

2.**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（文件系统块的大小）**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。

3.**HDFS同样也有块(block)的概念**，但是大的多，**默认为128MB**。与单一磁盘上文件系统相似，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128MB的块中时，文件只使用1MB的磁盘空间，而不是128MB）。

4.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时间**。因而，**传输一共由多个块组成的大文件的时间取决于磁盘传输速率**。

比如寻址时间是10ms，传输速率为100MB/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100MB。默认的块实际大小为128MB。

但在很多情况下HDFS安装时使用更大的块，以后随着磁盘驱动传输速率而提升，块的大小会被设置得更大。

但这个**参数也不会设置过大**。**MapReduce中的map任务通常一次只处理一个块中得数据，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**。

5.**分布式文件系统中的块进行抽象**会带来很多好处。

1. 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的所有磁盘。
2. 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，其它系统就可以单独管理这些元数据）
3. **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地。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。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。

6.HDFS中fsck指令可以显示块信息。hdfs fsck / -files -blocks+

## 3.2.2 NameNode和DataNode（重点）

1.**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)**。

2.**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**。

3.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。

4.**DataNode是文件系统的工作节点**。他们根据需要**存储并检索数据块**（受客户端或NameNode调度），**并定期向NameNode发送它们所存储的块的列表**。

5.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。

6.Hadoop提供两种容错机制：

1. **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)。
2. **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）

## 3.2.3 块缓存（重点）

1.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**被显示地缓存在DataNode的内存中**，以**堆外块缓存**（off-heap block cache）的形式存在。

2.默认情况下，**一个块仅缓存在一个DataNode内存中**，**当然可以针对每个文件配置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**。

3.**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

1.**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**，**内存将称为限制系统横向扩展的瓶颈**。

2.在2.X发行版本中引入的联邦HDFS允许系统通过添加NameNode进行扩展，其中每个**NameNode管理文件系统命名空间中的一部分**。比如一个NameNode管理/user下的全部文件，另一个可能管理/share目录下的所有文件。

3.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。

## 3.2.5 HDFS的高可用性（重点）

1.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创建检测点**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**。

2.在这样的情况下，从一个失效的NameNode恢复，系统管理员需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。

3.新的**NameNode直到满足以下情形才能相应任务**：

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。

4.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30mins，甚至更长时间。

5.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。

6.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**，不会有任何明显的中断。实现这一目标需要在架构上做出如下修改：

* 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**用户来讲是透明的**
* 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

7.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**。

8.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组日志节点**。典型的，有**三个journal节点**，所以系统能忍受其中任何一个丢失。这种安排与Zookeeper的工作方式类似，**当然QJM的实现并没使用Zookeeper**。(然而，HDFS HA在选举活动的NameNode确实使用了Zookeeper技术)

9.在活动NameNode失效之后，备用NameNode能够**快速(几十秒时间)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**。

10.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。

### 故障切换与规避

1.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来保证有且只有一个活动NameNode**。

2.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**。

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

4.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被分割的情况下，同样也可能发生故障转移，但是先前的活动NameNode依然运行着并且依旧是活动NameNode**。

5.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)。

6.同一时间QJM(群体日志管理器)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，仍有可能相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**。

7.之所以不使用NFS过滤器实现共享编辑日志，是因为**不可能同一时间只允许一个NameNode写入数据**。

8.规避机制：

* 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STONITH**(shoot the other node in the head, 一枪爆头)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**。

9.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(HDFS统一资源标识符)使用一个逻辑主机名，该主机名映射到一对NameNode地址**(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

1.设置伪分布式配置时，两个属性需要进一步解释。

* fs.defaultFS，设置为hdfs://localhost/，用于设置**Hadoop的默认文件系统**。文件系统是URI(统一资源标识符)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的主机和端口。**在localhost默认的HDFS端口8020上运行NameNode**。这样一来，HDFS客户端科研通过该属性得知NameNode在哪里运行进而连接到它。
* dfs.replication，**设为1，这样一来，HDFS就不会按默认设置将文件块复本设为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。设置之后，就不会出现上述问题。

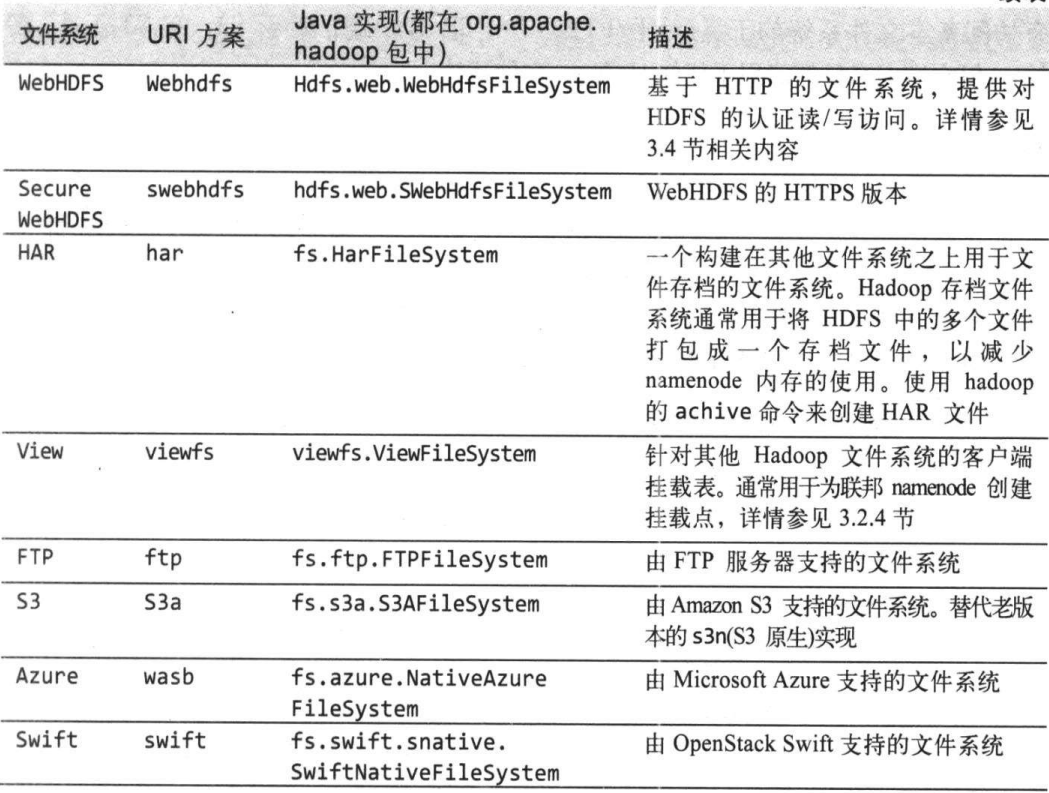
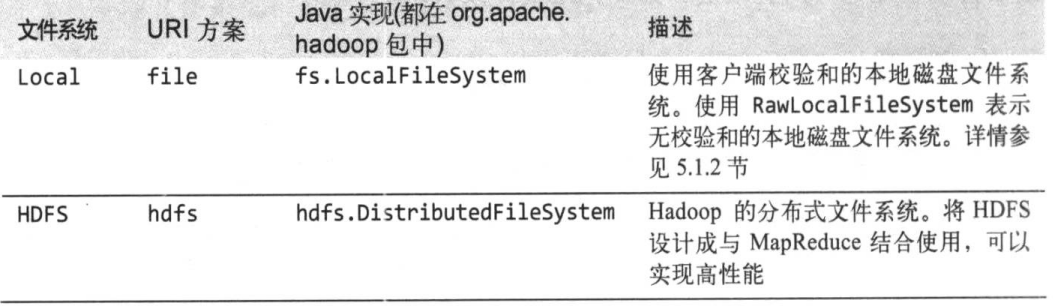
2.HDFS中的文件访问权限：

* 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目录，需要写入权限。对于文件，可执行权限可忽略不计，因为**不能在HDFS中执行文件**。
* 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**。
* 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**(dfs.permissions.enabled)
* 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**。
* 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

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紧密关系如下。

Hadoop文件系统：



2.Hadoop对文件系统提供了很多接口，**它一般使用URI方案来选取合适的文件系统实例进行交互**。

3.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，还是选择数据本地优化的分布式文件系统，比如HDFS。

# 3.5 Java接口

### 使用FileSystem API读取数据

1.FileSystem类是一个Hadoop的某一文件系统进行交互的API。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。

2.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。

* 第一个方法返回的是默认文件系统(core-site.xml指定)，默认缓冲区大小是4KB
* 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
* 第三个方法作为给定用户来访问文件系统，对安全来说是至关重要。

3.FSDataInputStream对象：

* **FileSystem对象**中的open()方法返回的是**FSDataInputStream对象**，而不是标准的java.io对象。这个类继承了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**
* 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偏移量(getPos())的查询方法
* FSDataInputStream类实现了PositionedReadable接口，从一个指定偏移量处读取文件的一部分
* 所有方法会**保留文件当前偏移**量，并且**线程安全的**(FSDataInputStream并不是为并发访问设计的，因此最好为此创建多个示例)，因此它们提供了读取文件的主体时，访问文件其他部分(也可能是元数据)的便利方法。

### 写入数据

1.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

2.此方法有多个重载版本，允许我们指定是否需要**强制覆盖现有文件**、**文件备份数量、写入文件时所用缓存区大小**、**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。Progessable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。

3.另一种新建文件的方法是使用append()方法在访问该文件的**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**。

4.下面范例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64KB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。

package org.Test\_3\_4;  
  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.FileSystem;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.IOUtils;  
import org.apache.hadoop.util.Progressable;  
  
import java.io.BufferedInputStream;  
import java.io.FileInputStream;  
import java.io.InputStream;  
import java.io.OutputStream;  
import java.net.URI;  
  
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 configuration = new Configuration();  
 FileSystem fs = FileSystem.get(URI.create(dst), configuration);  
 OutputStream out = fs.create(new Path((dst)), new Progressable() {  
 @Override  
 public void progress() {  
 System.out.print('.');  
 }  
 });  
 IOUtils.copyBytes(in, out, 4096, true);  
 }  
}

5.FSDataOutputStream对象：

* FileSystem实例的create()方法返回FSDataOutputStream对象，与FSDataInputStream类类似，它也有一个查询文件当前位置的方法getPos()。
* 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。它不支持在除文件末尾之外的其他位置进行写入，因此，写入时定位也没什么意义。

### 目录

1.FileSystem实例提供了创建目录的方法：

public boolean mkdirs(Path f) throws IOExecption

2.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。通常不需要显式创建目录，因为create()方法写入文件时会自动创建父目录。

### 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就是**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。

如果文件目录不存在，会抛出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()方法更方便。

2.列出文件：listStatus()

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

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**。

其重载方法允许使用PathFilter来限制匹配的文件和目录。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对象数组累计存入同一数组中，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，这很有用。

3.文件模式：

HDFS支持在表达式使用通配符来匹配多个文件，无需列举每个文件和目录在指定输入，该操作为“通配”(globbing)。Hadoop存在两个FileSystem方法globStatus()和PathFilter对象执行通配。

### 删除数据

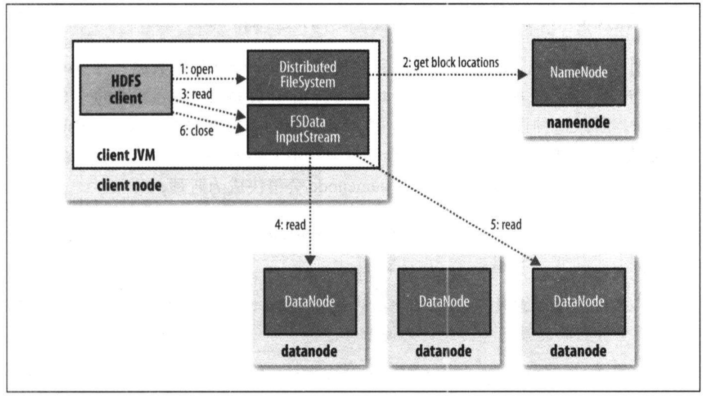
1.使用FileSystem的delete()方法可以永久性删除文件或目录。

## 3.6 数据流（重点）

### 3.6.1 剖析文件读取（重点）

1.客户端通过调用**FileSystem对象的open()方法来打开希望读取的文件**，对于HDFS来说，对象是**DistributedFileSystem的一个实例**(下图步骤1)。

1. DistributedFileSystem通过**远程调用**(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**)。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
2. **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
3. 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DataNode**。
4. 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)。
5. 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)。
6. **所有这些操作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**。
7. 客户端从**流中读取数据时**，**块是按照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)
8. 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**。它也会记住那个**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，也会将被破坏的块通知给NameNode**。
9. 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。
10. 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，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很快成为瓶颈。



2.在面试中我们这样回答HDFS读数据流程：

1. 客户端通过Distributed FileSystem向NameNode请求下载文件，NameNode通过查询元数据，找到文件块所在的DataNode地址
2. 挑选一台DataNode(就近原则，然后随机)服务器，请求读取数据
3. DataNode开始传输数据给客户端(从磁盘里面读取数据输入流，以packet为单位来做校验)
4. 客户端以packet为单位接收，先在本地缓存，然后写入目标文件
5. 一次传输剩下的数据块，直到整个文件合并完成

### **网络拓扑与Hadoop（重点）**

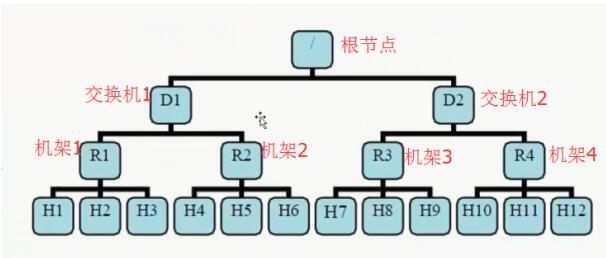
1.在海量数据处理中，其**主要限制因素是节点之间数据的传输速率——带宽**很稀缺。这里将两个节点的带宽作为距离的衡量标准。

2.不同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，并且集群中两两节点对数量是节点数量的平方）。

3.Hadoop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公共祖先的距离总和**。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
* 同一数据中心不同机架上的节点
* 不同数据中心的节点(目前Hadoop不支持跨数据中心运行)



1.distance(/D1/R1/H1,/D1/R1/H1)=0 相同的 datanode

2.distance(/D1/R1/H1,/D1/R1/H2)=2 同一 rack 下的不同 datanode

3.distance(/D1/R1/H1,/D1/R1/H4)=4 同一 IDC(互联网数据中心（机房）)下的不同 datanode

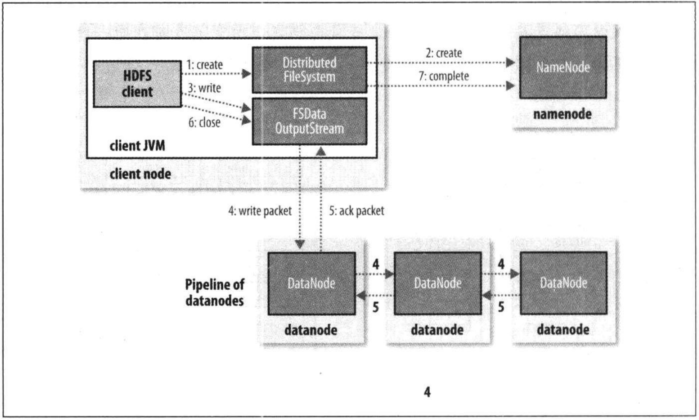
4.distance(/D1/R1/H1,/D2/R3/H7)=6 不同 IDC 下的 datanode

Hadoop是无法自动发现你的网络拓扑结构。它需要管理员来人工配置。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。规模小的集群不需要进一步配置。

### 3.6.2 剖析文件写入（重点）

1.我们考虑的问题是：如何新建一个文件，把数据写入该文件，最后关闭该文件。

1. 客户端通过对DistributedFileSystem对象调用create()来新建文件(步骤1)。
2. **DistributedFileSystem对NameNode创建一个远程调用**(RPC)，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)。
3. **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。
4. 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
5. 在客户端写入数据时(步骤3)，**DFSOutputStream将它分成一个个数据包**，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。
6. **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)。
7. **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，该数据包才会确认队列删除(步骤5)
8. 如果任何DataNode在数据写入期间发生故障，则执行以下操作(对写入数据的客户是透明的)：
   1. **首先关闭管线**，确认把队列中所有**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。
   2. 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。
   3. 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。后续数据块继续正常接受处理。
9. 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)
10. **客户端完成数据的写入后**，对数据流**调用close()方法**(步骤6)。
11. 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**



2.复本如何存储？

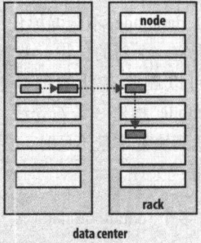
1. NameNode如何选择在哪个DataNode存储复本(replica)。这里需要对**可靠性、写入带宽和读取带宽**进行权衡。
2. 比如：把所有**复本都存储在一个节点损失的写入带宽最小**(因为复制管线都是在同一节点上运行)，但这并不提供真实的冗余(**如果节点发生故障，那么该块中的数据会丢失**)。同时，**同一机架上服务器间的读取带宽是很高**的。
3. 另一个极端，把**复本放在不同的数据中心可以最大限度地提高冗余**，但**带宽的损耗非常大**。即使在同一**数据中心**（到目前为止，所有Hadoop集群均运行在同一数据中心内），也有多种可能的数据布局策略。

3.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慢或太忙的节点**）

4.第2个复本放在与第1个复本**不同且随机另外选择的机架中节点上**（离架）。

5.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**尽量避免在同一个机架上放太多复本**。

6.一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。



7.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)。

8.面试中我们这样回答HDFS读文件操作：

1. 客户端发出请求hdfs dfs -put /etc/profile /qf/data
2. 客户端通过Distributed FileSystem模块向NameNode请求上传文件，NameNode检查目标文件是否已存在，父目录是否存在
3. NameNode返回是否可以上传
4. 客户端请求第一个block(0~128MB, dfs.block.size)上传到哪几个datanode服务器上
5. NameNode返回3个DataNode节点，分别为dn1、dn2、dn3
6. 客户端通过FSDataOutputStream模块请求dn1上传数据，dn1收到请求会继续调用dn2，然后dn2调用dn3，将这个通信管道建立完成
7. dn1、dn2、dn3逐级应答客户端
8. 客户端开始往dn1上传第一个block（先从磁盘读取数据放到一个本地内存缓存），以packet（dfs.write.packet.size，默认64K）为单位，dn1收到一个packet就会传给dn2，dn2传给dn3；dn1每传一个packet会放入一个应答队列等待应答
9. 当一个block传输完成之后，客户端再次请求NameNode上传第二个block的服务器(重复3~7步)

### 3.6.3 一致模型

1.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为性能牺牲一些POSIX要求，因此操作可能不同。

2.新建一个文件后，它能在**文件系统的命名空间中立即可见**。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**。

3.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**。

4.HDFS提供了一种**强行将所有缓存刷新到DataNode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可见**。

5.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，数据中心断电，数据会丢失)。为确保数据写入到磁盘上，可用hsync()替代。

6.hsync()操作类似于POSIX中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，使用标准Java API数据写入本地文件，我们能够刷新数据流且同步之后看到内容。

7.对应用设计的重要性：

这个一致模型呵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可能会丢失数据块。对很多应用来说是不能接受的，所以需要在合适的地方调用hflush()方法，例如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()开销更大)，所以在数据鲁棒性和吞吐量之间会有取舍。

# 3.7 通过distcp并行复制

1.distcp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。

2.distcp的一种替代用法是：hdfs dfs -cp。例如，将文件/目录复制到另一个文件/目录：

hadoop distcp file1 file2hadoop distcp dir1 dir2

3.可以使用-overwrite选项覆盖，以及-update选项更新。

hadoop distcp -overwrite dir1 dir2hadoop distcp -update dir1 dir2

4.**distcp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。

5.默认情况下，将近20个map被使用，但是可以通过distop -m来指定map数目。

6.distop可以在两个HDFS集群间传送数据。

hadoop distop -update -delete -p hdfs://namenode1/foo hdfs://namenode2/foo

7.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distcp时，能够看到准确用法。

8.如果不兼容使用webhdfs://namenode1:50070/foo来运行。