Hadoop分布式文件系统

内容

文件系统回顾

分布式文件系统对于本地文件系统的区别

HDFS

- HDFS文件系统原理
- HDFS读写过程

Hadoop文件系统

- 压缩
- 文件格式

文件系统回顾

文件系统的介质一般是永久存储的介质，如硬盘，光盘，磁带等；内存是易失性介质，硬盘是非易失性介质，掉电后仍然可以保存信息

存盘介质的特征

- 容量：硬盘的容量增长速度很快，现在已经有3TB以上的磁盘
- 速度：速度远远跟不上内存的速度，即使是顺序读的情况下速度也在100MB左右

文件系统的基本功能

- 文件系统是用户以及应用程序与底层的磁盘系统的接口。文件系统给上层的应用提供了以目录树为组织方式的名字空间，而底层则是以磁盘块数组为接口的磁盘
- 文件系统需要做好翻译工作

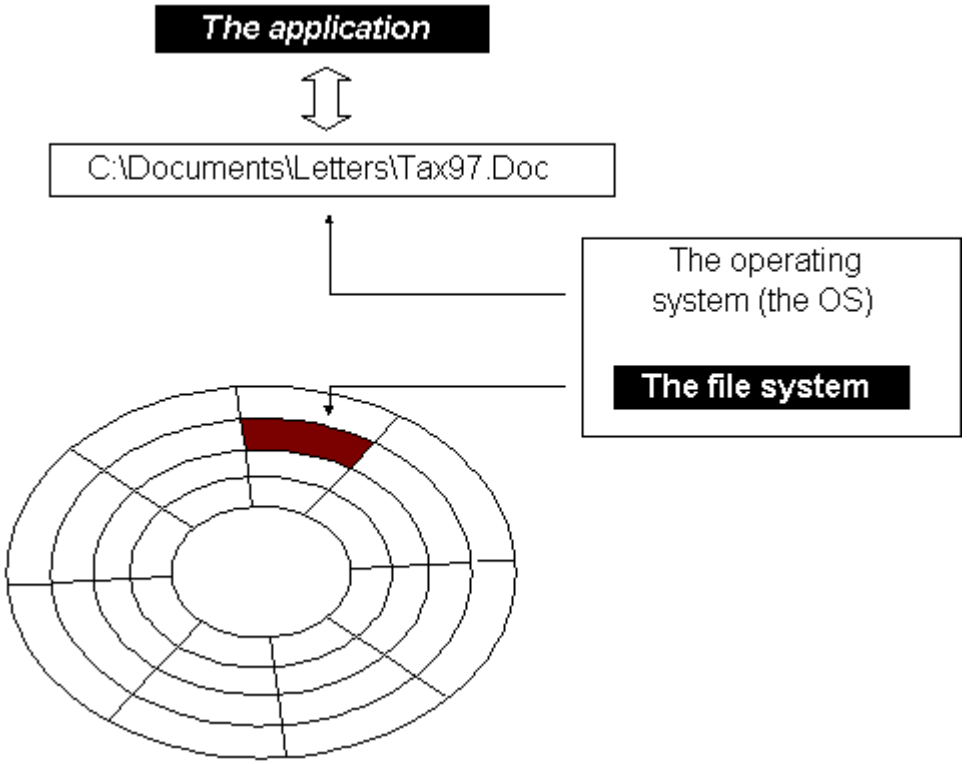
文件系统的基本功能：寻址与定位

应用程序访问目录树中的一个文件

操作系统将文件名交给文件系统

文件系统将文件名翻译为对应的磁盘的具体位置

磁盘转到具体的磁道，定位磁头，完成读写



文件系统中的相关概念

文件系统的目录树组织

文件系统的基本操作

文件系统的文件读写操作

文件系统元数据

文件系统的安全性策略

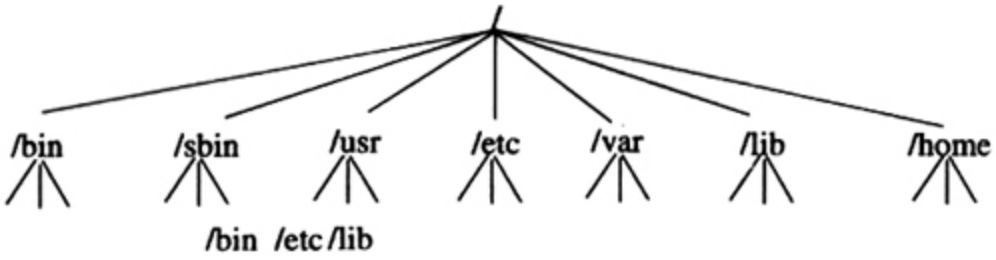
文件系统的目录树组织

不同的操作系统具有不同的文件目录树组织方式，有些目录具有特定的含义，用以特定的用途

Linux文件系统通过根目录树的组织方式，将所有的文件组织到同一个命名空间中，如果有不同的文件系统的化（例如多个磁盘）也会被安装到目录树的某一个目录下

Windows下的文件系统会根据不同的文件系统的情况分为多个不同的磁盘分别进行访问

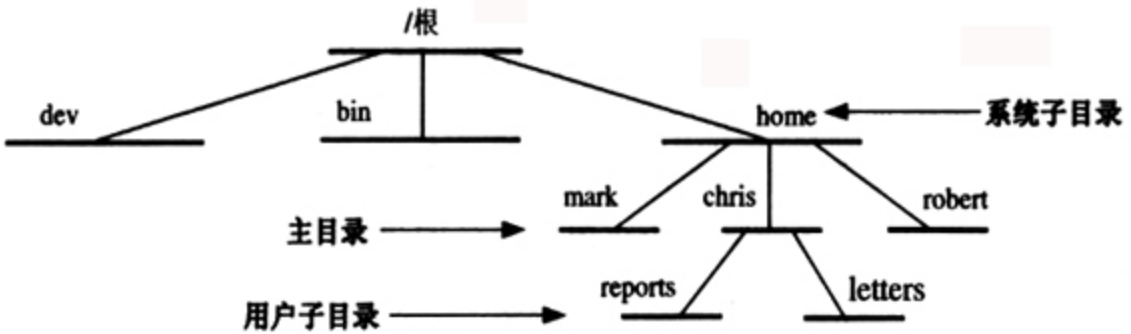
图1　Linux文件系统



DOS也采用目录树的结构，但是与Linux的略有不同，如图2所示。

图2　DOS文件系统

图3　从根目录开始的Linux文件结构



文件系统的基本操作

创建文件 删除文件 移动文件 文件改名

创建目录 删除目录 移动目录 目录改名

列目录下的所有文件以及子目录

文件系统的文件读写操作

文件系统的读写操作也具有标准的接口

- 文件的打开
- 文件的读取
- 文件的写入
- 文件的关闭

文件系统的读写操作实际是整个文件系统主要的操作，完成了整个文件系统绝大多数的数据流量的处理，提要整个系统的读写的高性能

（文件系统的不同部分，可能会有不同的指标需求）

文件系统元数据

文件系统的元数据是对于数据的描述，而不是数据的本身。

文件系统中针对文件的典型元数据：

- 文件在目录树中的目录名以及文件名
- 文件的大小以及在磁盘上的分布情况
- 文件的访问时间，修改时间
- 文件的用户数据等

文件系统总体的元数据

- 文件系统的编码信息
- 文件系统的格式化的信息
- 文件系统的可用空间等相关信息

文件系统的安全性策略

早期的文件系统没有安全性的策略，例如FAT文件系统没有安全性策略，整个系统没有用户的概念

当前文件系统都有一定的安全性策略，典型的UNIX文件系统策略有用户以及用户组的概念，在权限上区分rwx，即读，写，以及执行（对于目录是访问）

最新的文件系统以及操作系统会加入高级的安全功能，如NTFS支持保证文件和文件夹安全性的访问控制列表(ACL)，以及数据加密功能

分布式文件系统与本地文件系统

分布式文件系统需要提供什么功能？

- 文件系统目录树
- 文件的读写
- 文件的权限功能

分布式文件系统的构建考虑

- 文件系统同样需要提供目录树结构，这是向上提供的接口
- 分布式文件系统的底层是各个节点上的本地文件系统，而不是块设备的底层接口
  - 分布式文件系统由于不是需要底层块设备，因此不需要将模块放入到系统的内核，可以在应用层实现
  - 分布式文件系统底层的接口不是块设备的编程接口，每个节点都具有一个独立的操作系统，本机的数据管理可以交给本地文件系统去做

分布式文件系统提供的文件定位功能

与本地文件系统一样，分布式文件系统同样需要提供文件的定位功能，即从文件名定位到具体的数据的位置

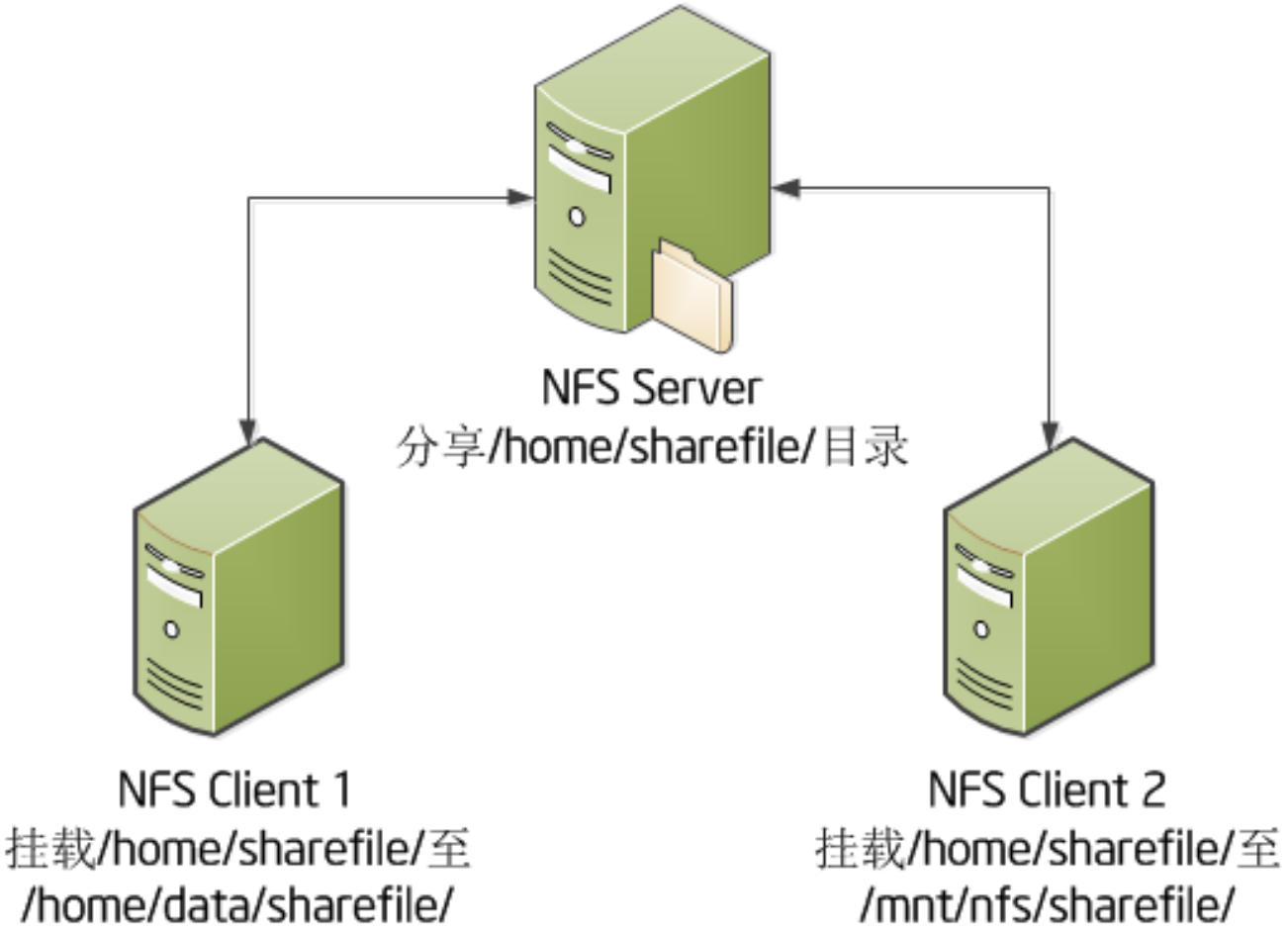
在本地文件系统中，数据的位置是磁盘的参数，而分布式文件系统的数据的位置是在系统中一个节点以及这个具体节点中的某一个文件

- 为何不需要定位到具体的磁盘中的位置？

分布式文件系统的本质的功能：将一个以目录树表达的文件翻译为一个具体的节点，而到磁盘的定位则可以交给本地文件系统去完成

最简单的分布式文件系统NFS（网络文件系统）

无需进行分布式环境的定位，所有的文件都保存在一个服务器中



HDFS与Google文件系统GFS

GFS的设计目的：为了存储Google内部大量的数据，主要是全球互联网的数据，需要极大的容量，为搜索引擎提供后备的存储支持。

Hadoop文件系统HDFS的设计思想来源于GFS，HDFS的基本结构与GFS一致。

Google搜索引擎对于文件系统的需求

Google需要一个分布式文件系统能够存储大量的数据

- 建立在大规模的廉价x86集群之上，并且集群模块还会出错

现有的文件系统无法满足Google对于存储数据的需求

- 整个硬件中的许多模块会出现出错的情况，出错会同时发生
- 有大量的超大规模的文件，文件大小会超过数百GB

读写特性：写入一次，多次读取。写入过程可能是并发的

读的过程是连续的读取，一次将一个文件全部内容读一遍

- 针对MapReduce优化

整个系统对于吞吐率的要求非常高，但是对于延迟不敏感

- 面向批处理

HDFS的特点

存储极大数目的信息（terabytes or petabytes），将数据保存到大量的节点当中。支持很大单个文件。

通过复制提供数据的高可靠性，单个或者多个节点不工作，对系统不会造成任何影响，数据仍然可用。

很高的系统吞吐量。

水平扩展：通过简单加入更多服务器的方式就能够扩展容量和吞吐量。最大的实用集群:4000个node。

针对MapReduce。HDFS对顺序读进行了优化；同时，使得数据尽可能根据其本地局部性进行访问与计算。

HDFS的基本设计

数据块

- 由于文件的规模十分庞大，文件将会被划分为多个大小为64MB的数据块进行存储，这个数据块的大小远远大于一般的文件系统数据块的大小
  - 减少元数据的量
  - 有利于顺序读写（在磁盘上数据顺序存放）

可靠性

- 为了保证数据的可靠性，数据通过副本的方式保存在多个节点之中，默认3个副本。副本选择也会考虑机架的情况，即不能把所有的副本都放在一个机架的服务器上，因为整个机架有可能同时掉电

系统设计简化

- 通过单个节点来保存文件系统的元数据，这个节点被称为文件系统的主节点NameNode，这个主节点用来协调整个系统的访问流程

HDFS的基本设计（2）

数据缓存

- 文件系统没有数据缓存，由于经常文件的访问是扫描式的，不具有局部性

访问接口

- 文件系统没有标谁的内核模块，所有用户程序通过编程库对文件系统进行访问
- 文件内容不允许覆盖更新overwrite
- 提供一个特殊的访问接口，在其它文件系统中不会出现，即追加操作Append，这也是由Google自己的文件读写特征引入的

HDFS系统结构中的主要模块

NameNode：

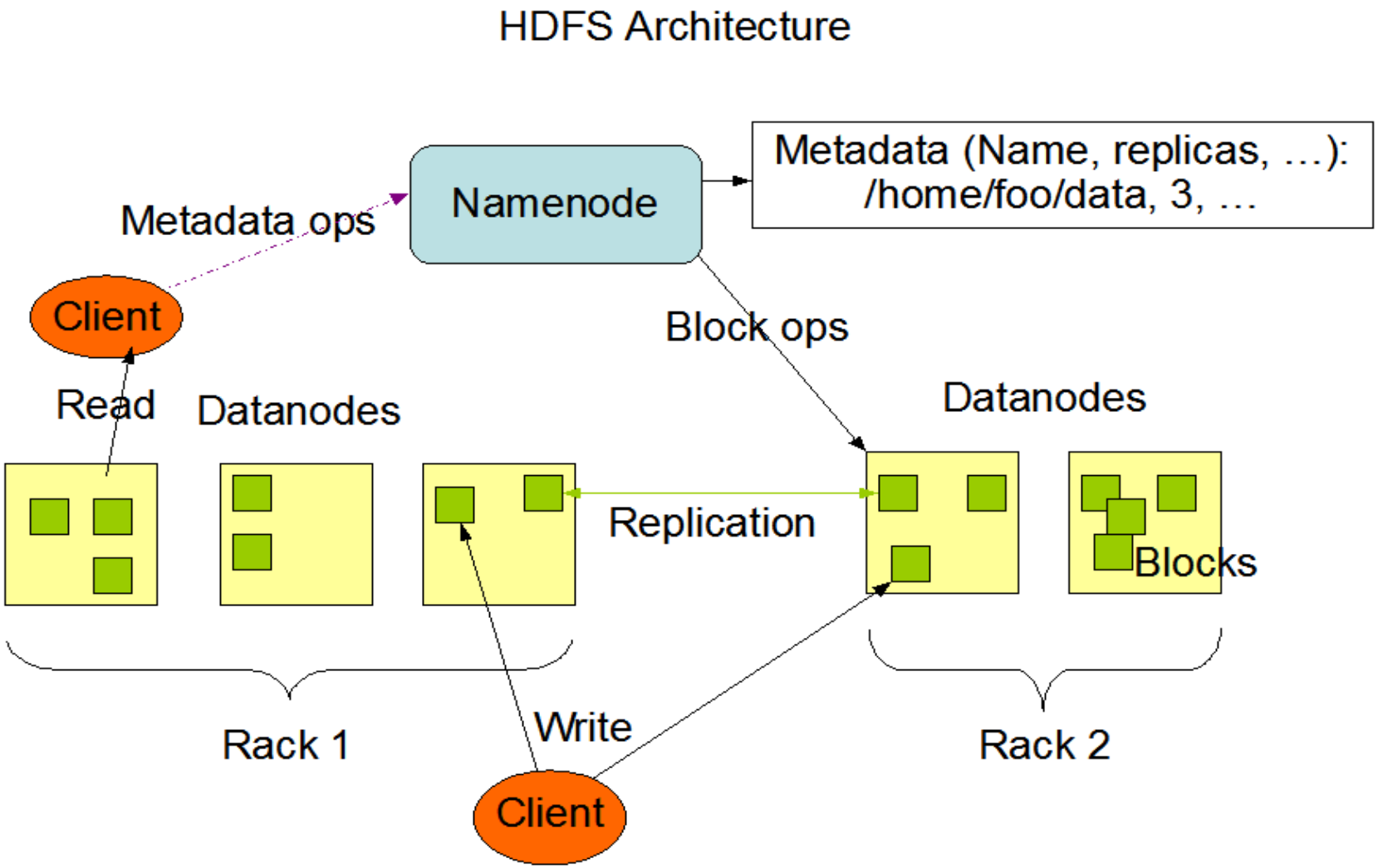
- 单台服务器，系统中的单点
- NameNode管理所有文件系统的元数据以及协调管理客户端对于数据的访问
- 管理集群节点和各种操作（如负载均衡）

DataNode：

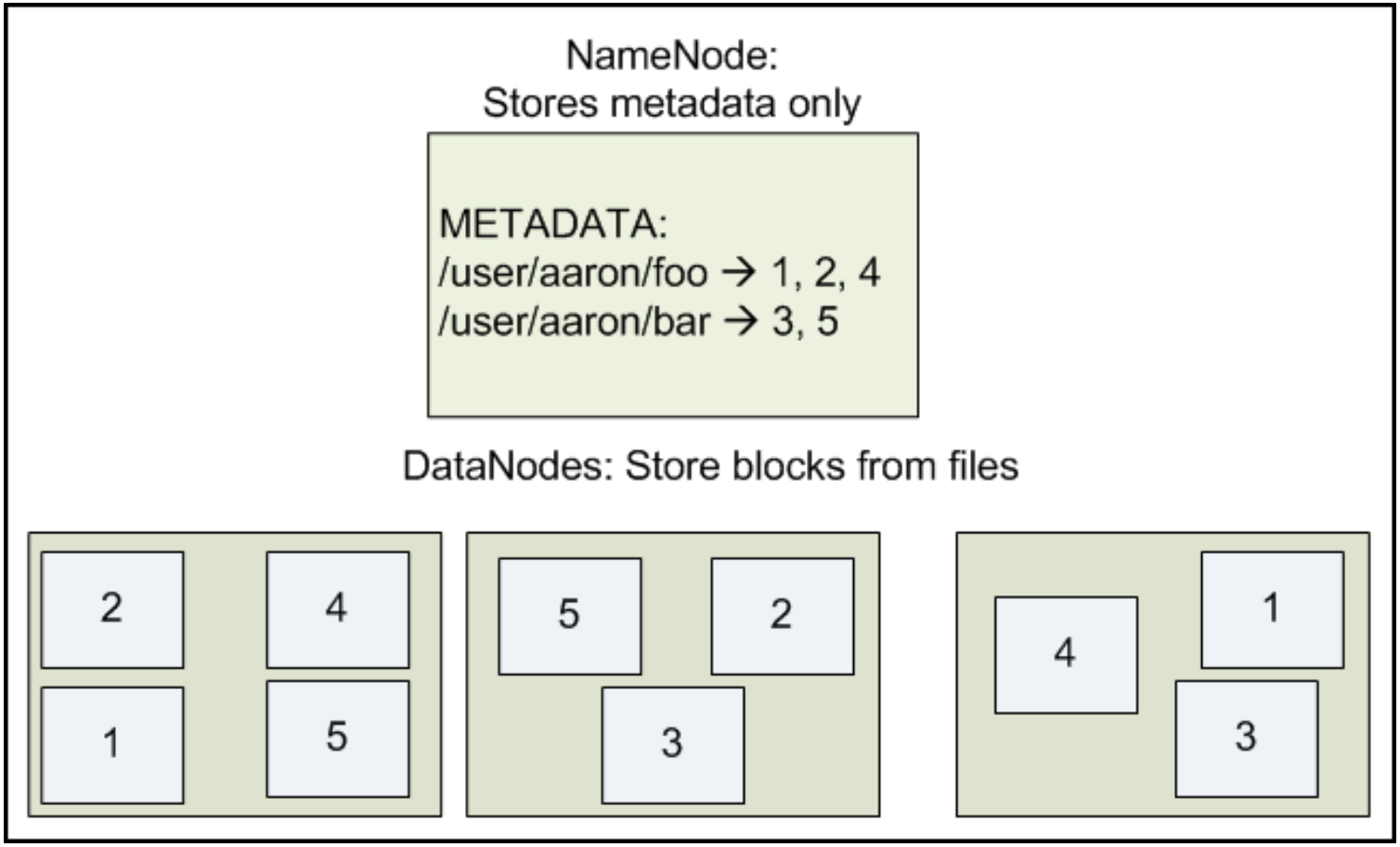
- 运行在集群中的绝大多数节点之上，每个节点一个DataNode进程
- 管理文件系统中的数据存储
- 从NameNode中接收命令，并对数据进行组织管理上的操作
- 响应文件系统客户端的读写访问命令，提供数据服务

数据通信的协议使用TCP/IP协议，使用RPC进行远程信息访问请求

HDFS体系结构


HDFS Architecture
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

HDFS数据分布设计


NameNode:
Stores metadata only
METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5
DataNodes: Store blocks from files
2
4
1
5
5
2
3
4
1
3

HDFS中的元数据

所有的全局的元数据都存放在NameNode中，包括以下的内容

- 文件以及数据块的名字空间
- 所有文件到数据块之间的映射关系
- 每个数据块副本的位置

所有的元数据都可以保存在内存中（一个64M数据块的元数据约为16个字节）

- 数据访问比较快，可以做缓存
- 能够降低访问的复杂度，简化访问模型

元数据操作需要记录日志

- NameNode针对关键的元数据更新记录操作日志：保存到磁盘中，进一步作副本，并作为快速恢复的检查点

NameNode

管理HDFS的元数据。

管理客户端对HDFS中文件的访问。客户端与NameNode交互的过程中，取到了所请求的文件块所对应的DataNode结点，执行文件的读写操作。

管理DataNode结点。包括DataNode结点的健康状态报告和其所在结点上数据块状态报告，以便能够及时处理失效的数据结点。

负载均衡

- 如果副本数目太少，生成副本
- 如果副本数目太多，删除副本
- 重新分布副本来进一步平滑存储以及访问的负载

NameNode (2)

垃圾搜集

- NameNode根据元数据和DataNode block report知道DataNode哪些块可删除
- NameNode定期通知DataNode删除相应块

过期副本的删除

- 整个系统在节点失效，并重新加入的时候产生过期副本数据
- 通过检查数据块的版本来探测到过期副本

DataNode

1、负责管理它所在结点上存储数据的读写

2、向Namenode结点报告状态。每个Datanode结点会周期性地向Namenode发送心跳信号和文件块状态报告。

3、执行数据的流水线复制。当文件系统客户端从Namenode获取到要进行复制的数据块列表后，完成文件块及其块副本的流水线复制。

fsimage,edits

为了最大化性能，NameNode将所有的元数据都保存在内存中。同时，将所有对元数据的修改都用log的形式持久化到磁盘上。

- fsimage中保存了整个文件系统元数据的一个快照
- Edits保存在这个快照之后所有对元数据修改操作的日志

当NameNode启动时，它首先从fsimage中读取快照，接着应用edits中的操作日志，从而得到最新的元数据状态。然后它将新的HDFS状态写入（fsimage）中，并使用一个空的edits文件开始正常操作

Safe Mode

刚启动的时候，NameNode会进入Safe Mode状态

- 合并fsimage和edits
- 等待每一个DataNode报告情况，更新数据块和DataNode的map
- 当绝大多数数据块（缺省99.9%）达到最小复制份数时，退出Safe Mode

退出安全模式的时候才进行副本复制操作，以及允许数据的写入

Secondary NameNode

因为NameNode只有在启动阶段才合并fsimage和edits，所以久而久之日志文件可能会变得非常庞大，特别是对大型的集群。日志文件太大的另一个副作用是下一次NameNode启动会花很长时间

SecondaryNameNode定期合并fsimage和edits日志，然后将更新后的fsimage回写到NameNode上。从而，将edits日志文件大小控制在一个限度下

工作原理和NameNode的启动完全一样

内存需求和NameNode类似， 所以通常secondaryNameNode和NameNode运行在不同的机器上。

HDFS可靠性

对于DataNode节点失效的问题：

- 通过心跳信息来获得节点的情况
- 当节点失效时，NameNode可以进行数据重新分布

集群数据的重新进行负载均衡

数据块的完整性通过数据校验获得，64KB进行一次校验

DataNode定期对其所拥有的数据块进行数据检验

关于单个NameNode服务器的讨论

单个NameNode可能存在的问题

- 性能的单点瓶颈：一个节点不能够负载所有的数据流量
- 可靠性的单点瓶颈：一个节点出现失效，整个系统无法运行

解决方法：

1. 高可靠、高性能的服务器
   - 使用多个元数据目录，同时存放多份fsimage和edits
   - 磁盘用RAID10或者RAID 5
2. Standby NameNode
   - 通过DRBD进行元数据的实时同步

HDFS的副本放置策略

数据副本对于文件系统的高可用性来说非常重要。HDFS默认会把数据存放3份

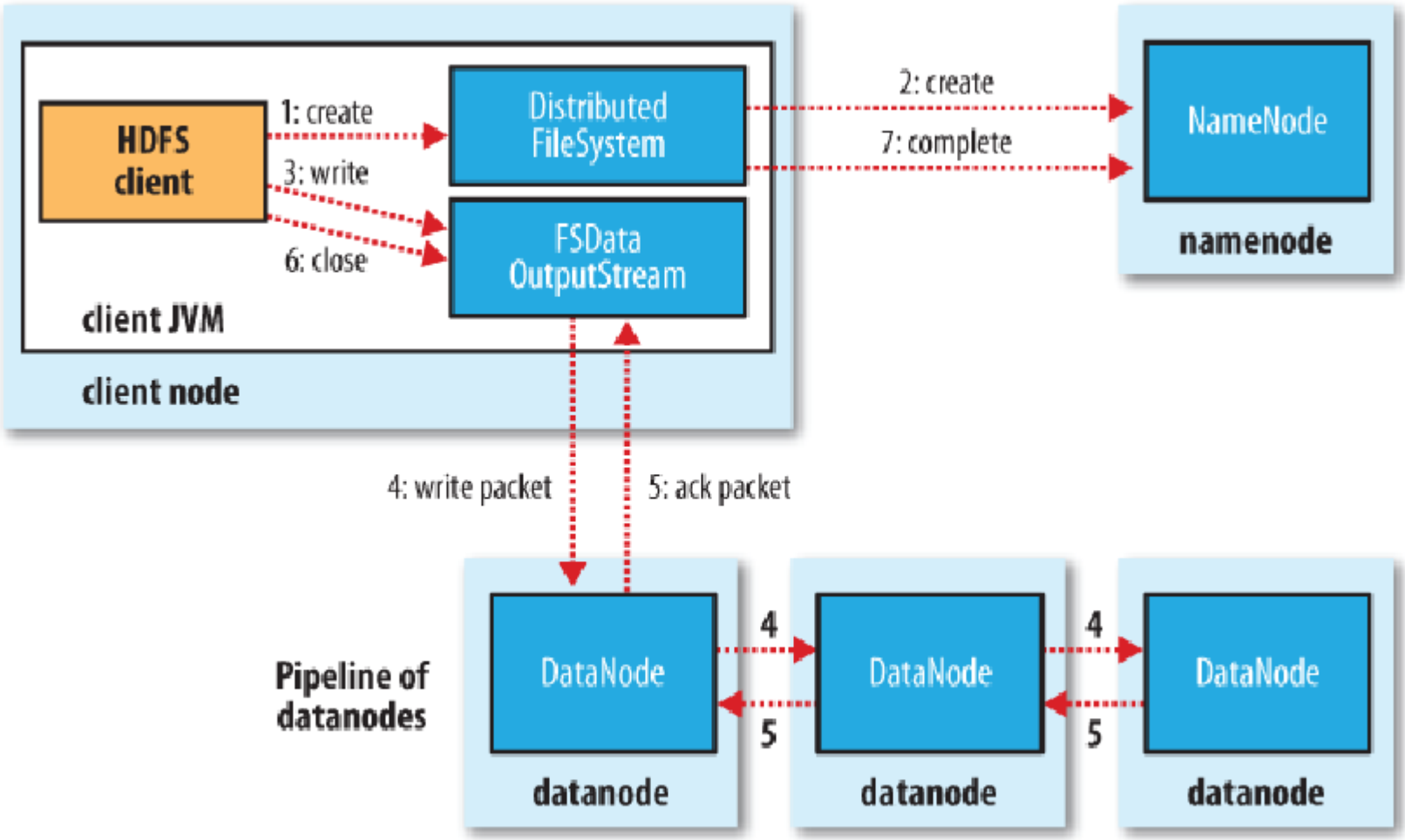
对于副本的选择来说，需要兼顾高可靠性以及高性能

- 数据副本的基本原则是不能把副本放置在一个节点上
- 不能把所有副本放在同一个机架上，以防止机架停电
- 为了提高性能，副本应该靠近客户端

HDFS的副本放置策略

- 第一份副本会优先放在本地DataNode（如果客户端在DataNode上的话）。
- 第二份副本会优先放在本地机架中的DataNode。
- 第三份副本会放在其他机架中的DataNode。

数据写入


HDFS client
1: create
3: write
6: close
Distributed FileSystem
FSData OutputStream
client JVM
client node
2: create
7: complete
NameNode
namenode
4: write packet
5: ack packet
Pipeline of datanodes
DataNode
4
5
DataNode
4
5
DataNode
datanode
datanode
datanode

数据写入：写流水线pipeline

当客户端开始写入时，会向Namenode申请新的block。NameNode返回用来存储副本的DataNode列表

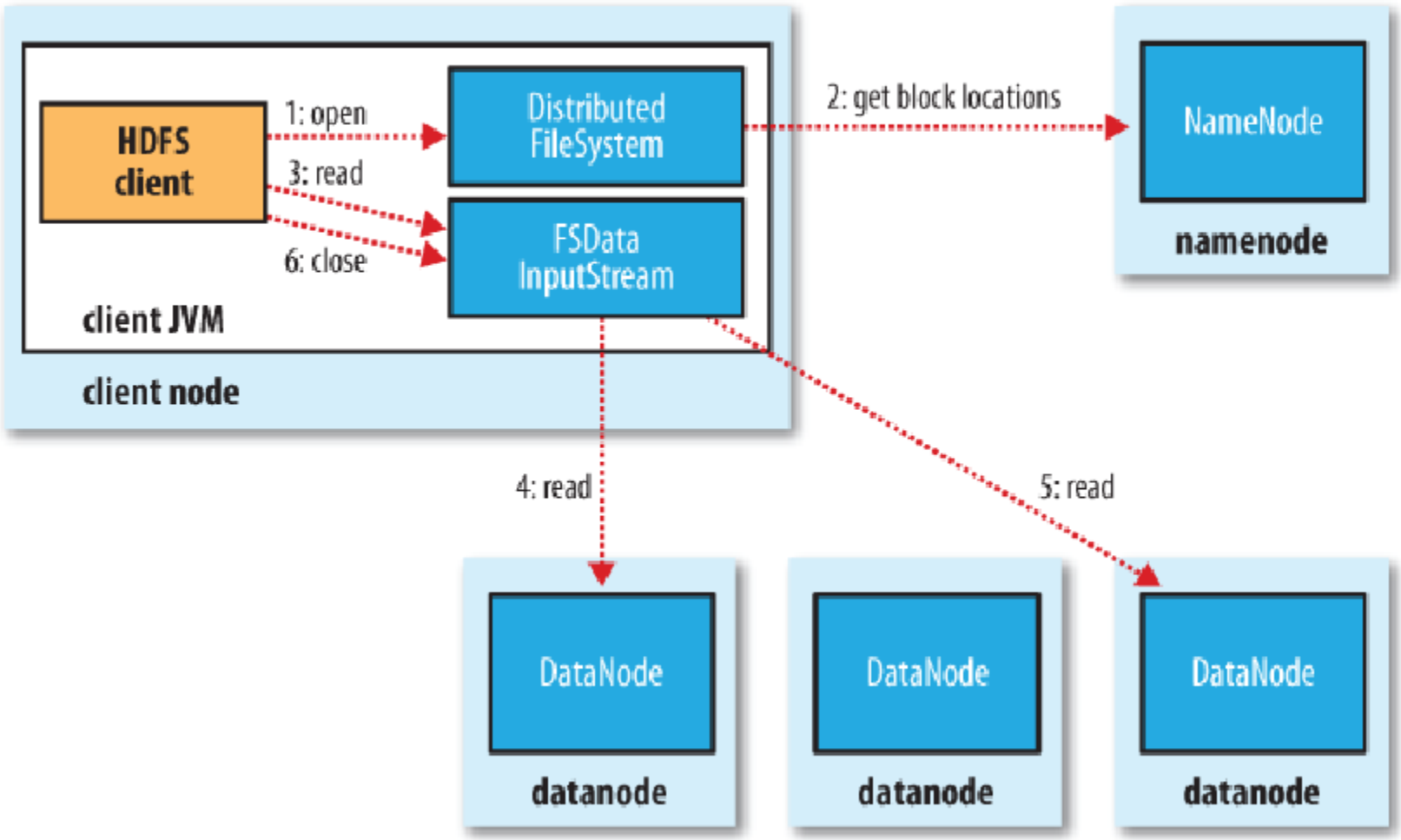
客户端通知第一个DataNode来建立写流水线pipeline

客户端将数据传给第一个DataNode，该DataNode在把数据本地存储的同时，将其传递给pipeline中的下一个DataNode，直到最后一个datanode，这种写数据的方式呈流水线的形式。

最后一个DataNode成功存储之后会返回一个ack packet，依次返回后，在pipeline里传递至客户端

如果传输过程中，有DataNode出现故障，那么当前pipeline会被关闭。NameNode分配一个DataNode以替换故障DataNode。剩余的block继续以pipeline的形式传输

数据读取


HDFS client
1: open
3: read
6: close
Distributed FileSystem
FSData InputStream
client JVM
client node
2: get block locations
NameNode
namenode
4: read
5: read
DataNode
datanode
DataNode
datanode
DataNode
datanode

数据读取过程

由于HDFS使用了3个副本进行数据存放，数据的读取过程如下：

- HDFS Client端首先从NameNode中获取对应数据块的分布情况。这个数据分布情况在运行的过程中可以一直缓存在HDFS Client的内部，以减少对于NameNode节点的打扰
- HDFS Client依据数据的分布情况从距离自己最近的副本中读取数据
  - IDH中实现了另外一个选取算法：根据最近的历史动态选取副本
- Client读取副本的同时会读取其Checksum文件，通过checksum来验证数据的正确性。

数据出错检测与出错处理

在HDFS中，每一块数据都需要经过校验，这样的话，客户端在数据读取的过程中就可以获知数据是否是正确的

客户端在检测出错误的时候，将出错的情况汇报给NameNode，NameNode会获知对应服务器的工作状态

NameNode在发现数据错误的时候，与丢失副本一样，发起数据重分布的过程，将正确的数据的副本数目进行恢复，并且在元数据中丢弃错误的数据块

HDFS的POSIX兼容性

HDFS不是标准的文件系统，而是建立在本地文件系统之上的应用层文件系统

HDFS与标准的POSIX文件系统并不兼容，因此在HDFS上面不能够运行程序，访问HDFS需要一个客户端

在数据的读写上，HDFS的POSIX不一致主要表现在以下方面：

- 数据读写方面，HDFS增加了Append操作，由文件系统确定写入地址，这一点是POSIX所没有的
- HDFS不允许对已有的文件进行修改（除了append）
- 在数据一致性方面POSIX不兼容，HDFS定义了自己的数据一致性模型

HDFS的一致性模型

HDFS定义了一个非严格的一致性模型，数据在文件系统中具有一定限制的一致性。以下是一致性模型中的两个定义：

- Consistent：一致性，所有的副本都具有相同的数据值
- Defined：明确的一致性：副本的数据反映了修改的状况，特别是修改的时间先后

HDFS一致性的特性：Relaxed Consistency，特性

- 数据不一定在所有副本中都是一样的，例如出现失败写的情况，这个时候数据有可能是不一致的
- 并发写成功的时候，数据是一致的，但不一定是明确的，即并不一定反映客户写入的时候的顺序过程

在出现不明确一致性的时候，由应用程序去自己解决这个问题

HDFS一致性的讨论

HDFS的一致性能够被有效实现，如果都需要明确的一致性，需要付出大量的努力，且没有必要

元数据的操作都是原子的，且可以序列化，因此元数据不会出现不一致的情况

HDFS的权限管理

类似于标准的UNIX文件权限管理

- rwxrwxrwx
- 判断十分简单，就是拿发出指令的用户名对文件进行权限检查

HDFS自己没有用户数据库

- 只要在连接Hadoop的机器上创建一个用户
- 然后su成该用户操作就可以

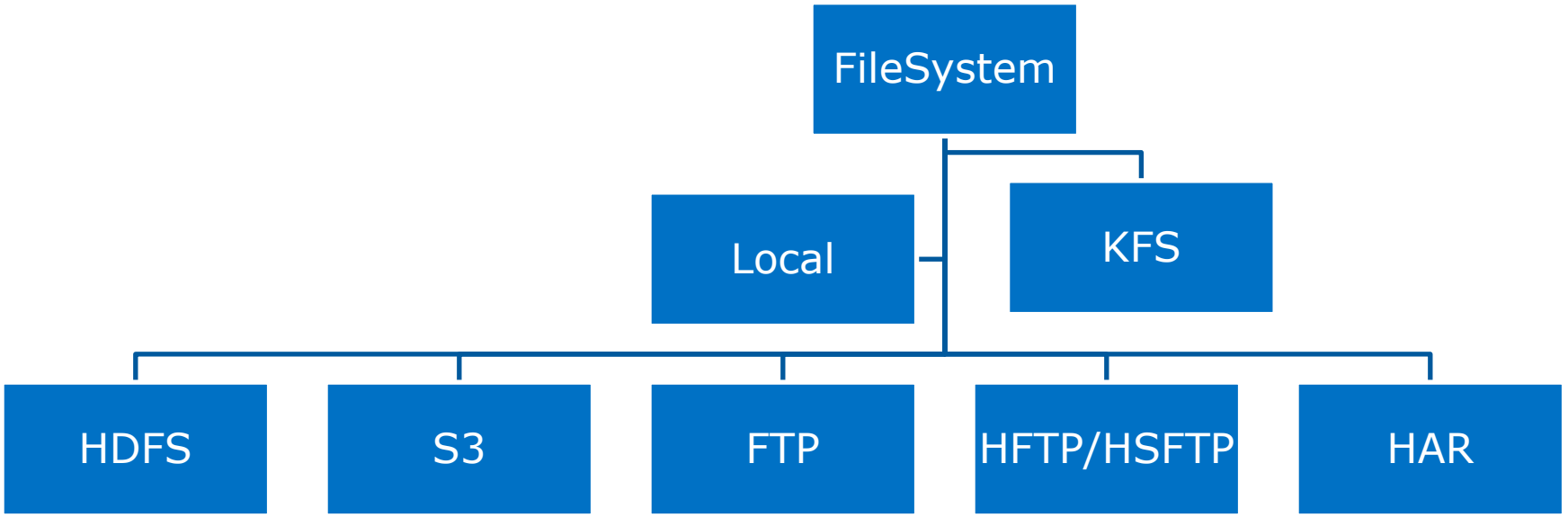
只有在使用Kerberos验证的情况下，才能提供足够的安全性

关于HDFS文件系统的总结

HDFS实际上是演示了如何在市场上可见的硬件水平上构建一个大规模的处理系统

- 从设计上就内建错误容忍机制
- 对大文件的优化，特别是数据追加以及读取
- 不局限于现有的文件系统接口，为了应用进行接口的扩展
- 尽量使用简化的设计，如单个的主服务器NameNode，简化系统的结构，便于理解与维护

Hadoop文件系统接口



Create specific FileSystem instance by configuration file

FileSystem Usage

```
Configuration conf = new Configuration();
try{
        FileSystem fs = FileSystem.get(conf);
        FSDataOutputStream out = fs.create(new Path(file));
        out.writeUTF(message);
        out.close();
}finally{
        fs.close();
}
```

压缩

Hadoop支持对文件的压缩

- 减少储存的内容
- 减少网络传输

压缩以Codec形式表达

IDH利用Codec接口实现了对文件和MapReduce过程的加密

压缩格式

Compression Format	Tool	Algorithm	Filename Extension	Multiple Files	Splittable
DEFLATE	N/A	DEFLATE	.deflate	NO	NO
gzip	gzip	DEFLATE	.gz	NO	NO
ZIP	zip	DEFLATE	.zip	YES	YES, at file boundaries
bzip2	bzip2	bzip2	.bz2	NO	YES
LZO	lzop	LZO	.lzo	NO	NO
Snappy	N/A	Snappy	.snappy	NO	YES

"Splittable" 列

- 表示该格式是否支持数据切割splitting （用于MapReduce），即是否可以定位到文件中某点，然后从这点开始读取。

压缩代码示例

```
String codecClassname = args[0];

Class<?> codecClass =Class.forName(codecClassname);

Configuration conf = new Configuration();

CompressionCodec codec = (CompressionCodec)

        ReflectionUtils.newInstance(codecClass, conf);

CompressionOutputStream out =
codec.createOutputStream(System.out);

IOUtils.copyBytes(System.in, out, 4096, false);

out.finish();
```

SequenceFile

用于保存大量的键值对

一些应用场景：

1. Log文件：
   - 键：时间戳
   - 值：log内容
2. 用于保存很多的小文件

SequenceFile代码示例

```
FileSystem fs=FileSystem.get(conf);

SequenceFile.Writer
writer=SequenceFile.createWriter(fs, conf, new
Path(file), LongWritable.class, UserWritable.class);

writer.append(key, value);

IOUtils.closeStream(writer);

fs.close();
```

MapFile

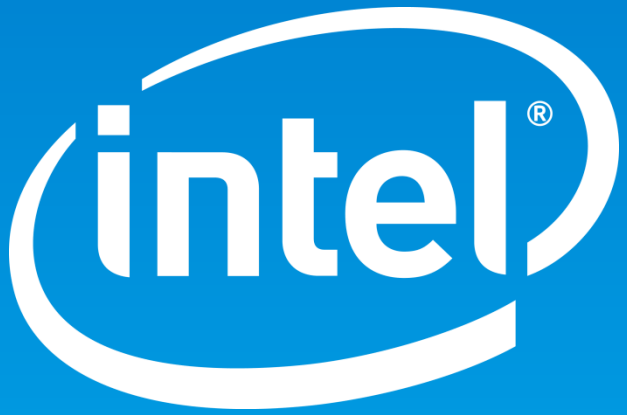
MapFile = 排序的SequenceFile + 内存Index

支持快速的查找

键类型必须继承自**WritableComparable**

值类型必须继承自**Writable**

小心内存占用和排序开销，仅支持小数据量


intel