大数据管理
Big Data Management

张海腾

htzhang@ecust.edu.cn

第三章 分布式文件系统HDFS

- 3.1 分布式文件系统
- 3.2 HDFS简介
- 3.3 HDFS的相关概念
- 3.4 HDFS体系结构
- 3.5 HDFS的存储原理
- 3.6 HDFS的数据读写过程
- 3.7 HDFS编程实践

3.1 分布式文件系统

- 本地文件系统
- 分布式文件系统DFS：通过网络实现文件在多台主机上进行分布式存储的文件系统　　P46
  - 设计模式：Client/Server
    - 客户端以特定的通信协议通过网络与服务器建立连接，提出文件访问请求；
    - 客户端和服务器可以通过设置访问权来限制请求方对底层数据存储块的访问。
- GFS和HDFS（HDFS是GFS的开源实现）

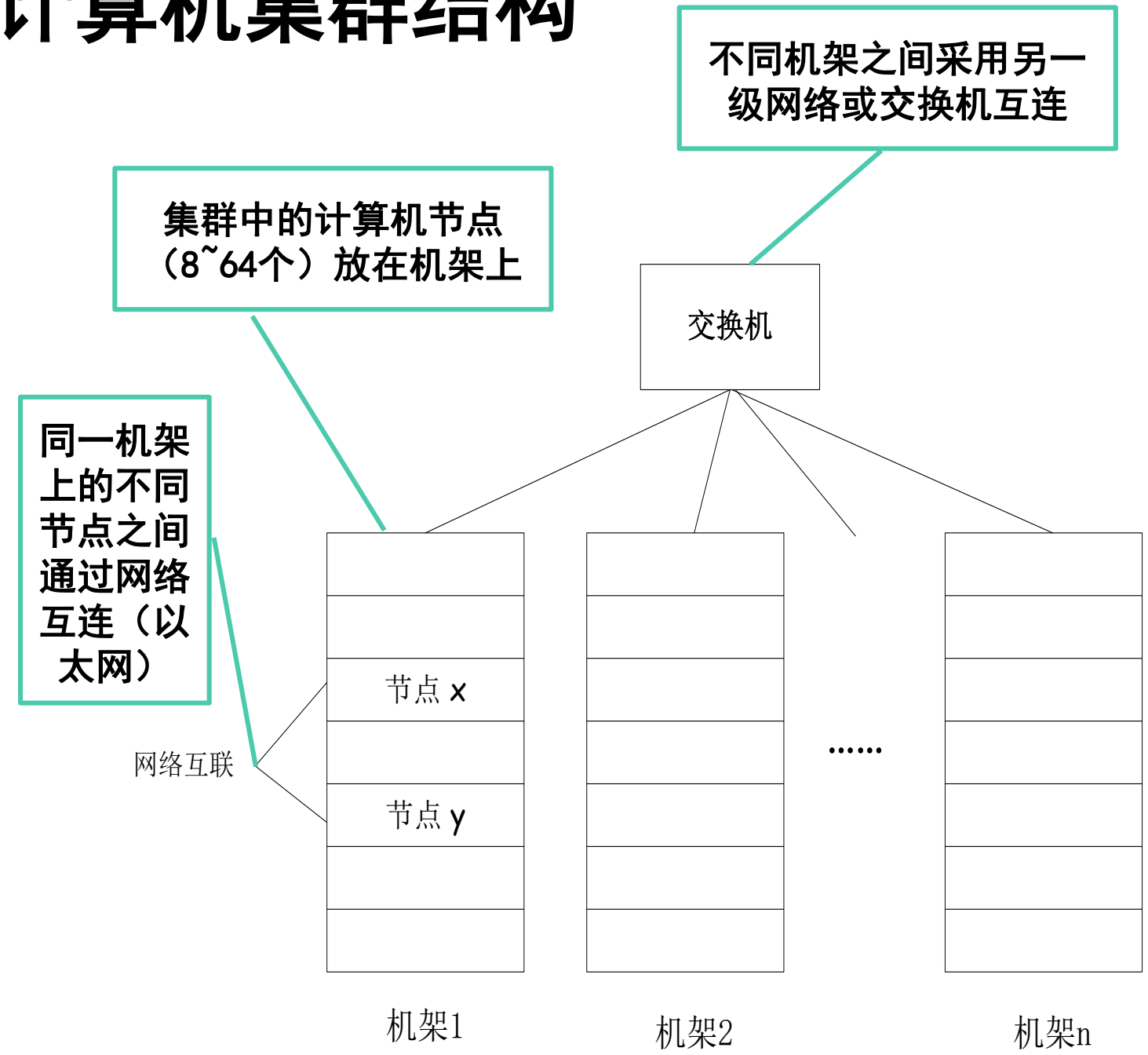
3.1 分布式文件系统

3.1.1 计算机集群结构

3.1.2 分布式文件系统的结构

3.1.1 计算机集群结构

- 分布式文件系统把文件分布存储到多个计算机节点上，**成千上万的计算机节点构成计算机集群**
- 与之前使用多个处理器和专用高级硬件的并行化处理装置不同的是，目前的**分布式文件系统所采用的计算机集群，都是由普通硬件构成的，大大降低了硬件上的开销**

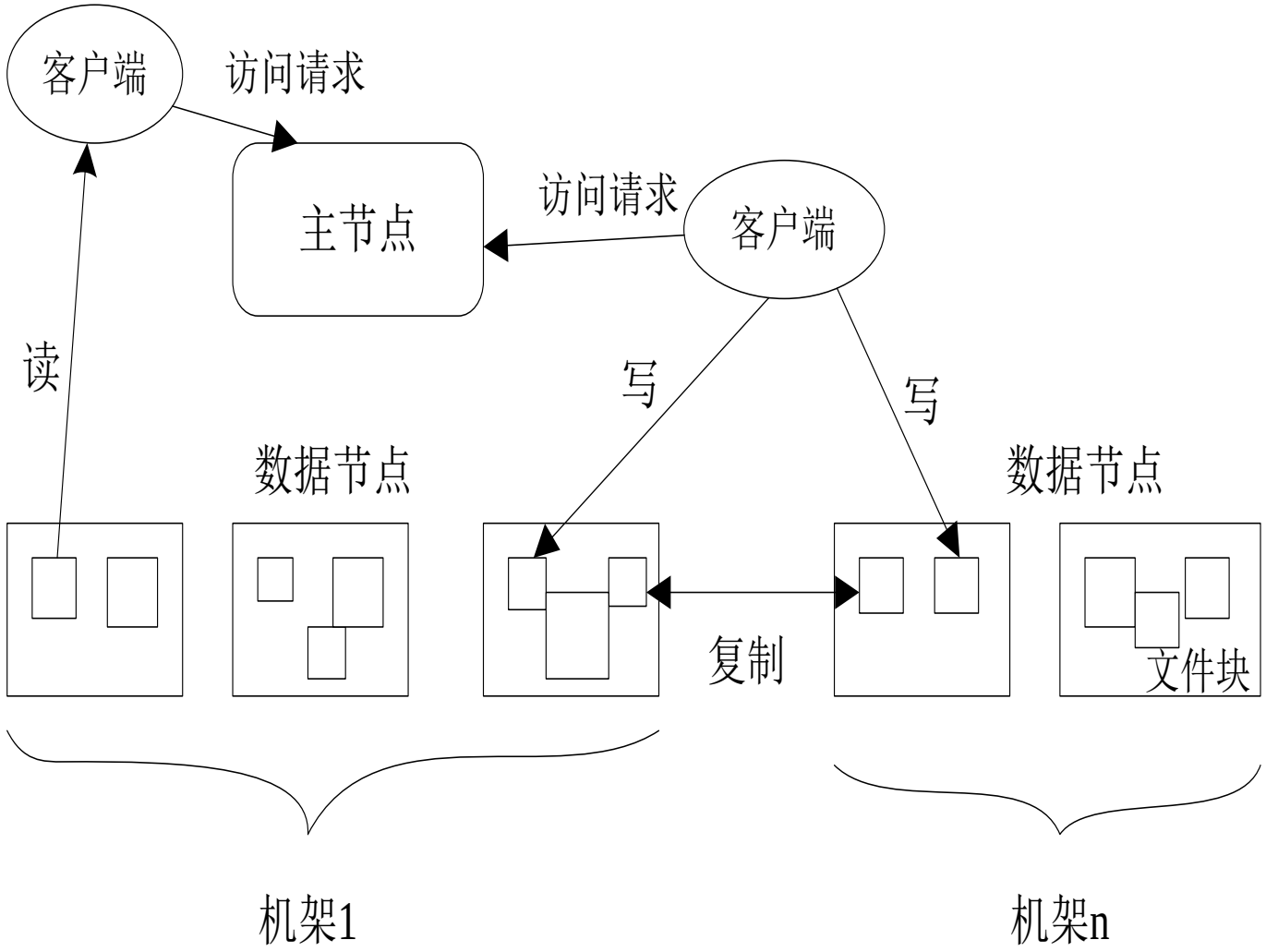


3.1.2 分布式文件系统的结构

- **一般的操作系统（Windows、Linux等）：**
  - 文件系统一般把磁盘空间划分为每512B为一组的磁盘块。
  - 文件系统的块（Block）通常是磁盘块的整数倍（每次读写的数据量必须是磁盘块大小的整数倍）
- **分布式文件系统：**
  - 文件被分成若干个块进行存储，块是数据读写的基本单元。
  - HDFS默认的一个块的大小是64MB。
  - 如果一个文件小于一个数据块的大小，它并不占用整个数据块的存储空间。例如：如果一个文件大小为100MB，它分成2个块：64MB +36MB。

3.1.2 分布式文件系统的结构

- 分布式文件系统在物理结构上是由计算机集群中的**多个节点**构成的
- 这些节点分为两类，一类叫“**主节点**”(Master Node)或者也被称为“**名称结点**”(NameNode)，
- 另一类叫“**从节点**”（Slave Node）或者也被称为“**数据节点**”(DataNode)

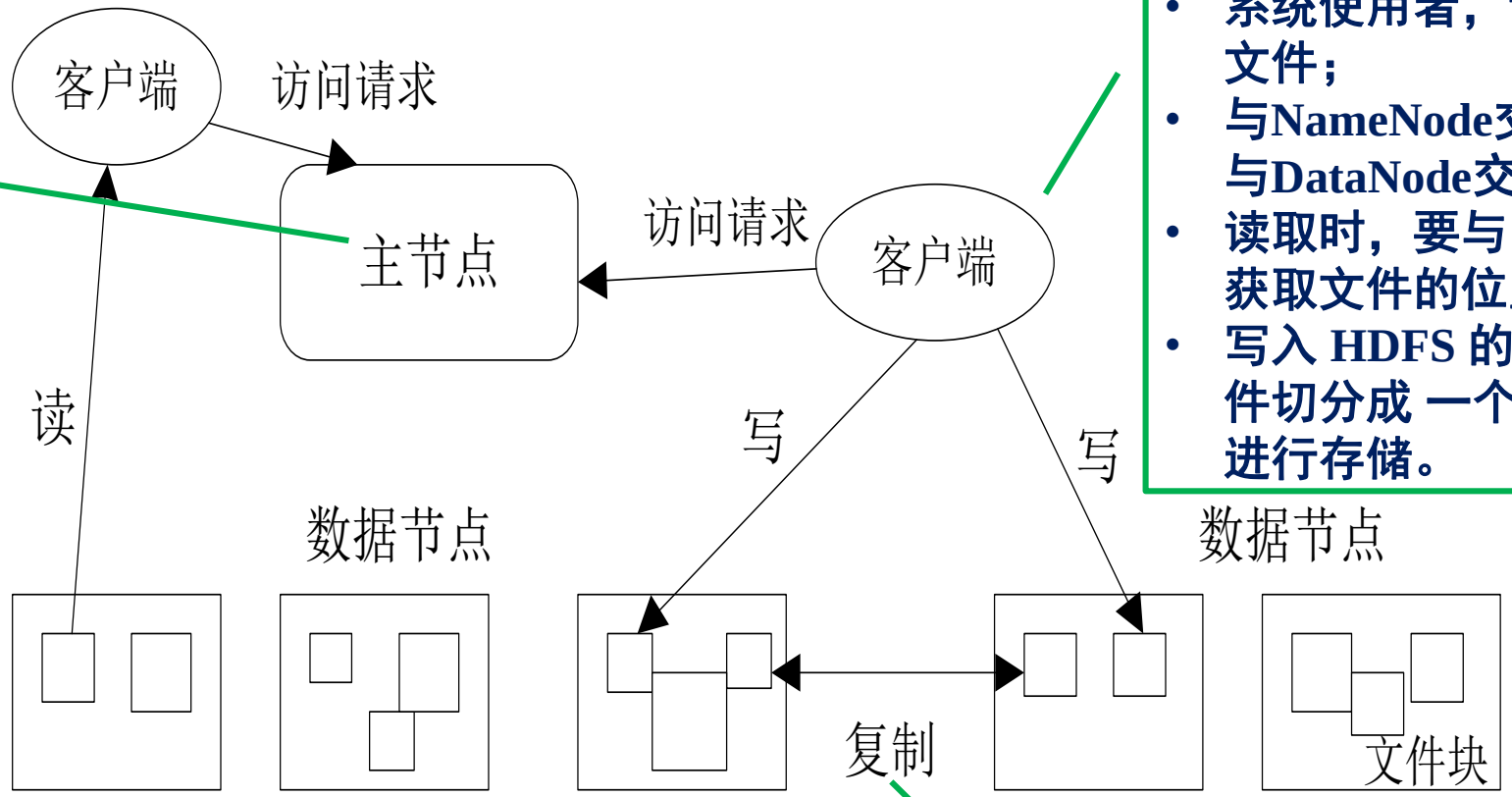


3.1.2 分布式文件系统的结构

NameNode的作用（主管、管理者）

- 管理HDFS的名称空间：负责文件和目录的创建、删除和重命名等
- 管理DataNode和文件块的映射关系（客户端只有访问NameNode才能找到请求的文件块所在的位置，再到相应位置读取所需文件块）
- **DataNode的作用（NameNode下达命令，DataNode具体执行）**
- 负责数据的存储和读取
  - 存储时：由NameNode分配存储位置，由客户端把数据直接写入相应DataNode；
  - 读取时：客户端从NameNode获得DataNode和文件块的映射关系，然后就可以到相应的位置访问文件块
- 根据NameNode的命令创建、删除数据块和冗余复制

3.1.2 分布式文件系统的结构



- 存放元数据（文件名称、大小、副本数、位置等信息）

- 系统使用者，调用HDFS API操作文件；
- 与NameNode交互获取文件元数据；与DataNode交互进行数据读写。
- 读取时，要与 NameNode 交互，获取文件的位置信息;
- 写入 HDFS 的时候，Client 将文件切分成 一个一个的Block，然后进行存储。

- 存储实际的数据块；
- 执行数据块的读/写操作。

- 分布式文件系统通常采用多副本策略。
- 文件快会被复制为多个副本，存储在不通的节点上，而且存储同一数据块的不同副本的各个节点会分布在不同的机架上。

3.1.3 分布式文件系统的设计需求

□透明性：

- **访问透明性**：用户能够通过相同的操作访问本地和远程文件资源。
- **位置透明性**：不改变路径名的前提下，不管文件副本数量和实际存储位置发生何种变化，对用户而言都是透明的。
- **性能和伸缩透明性**：系统中节点的增加和减少以及性能的变化对用户而言是透明的，用户感觉不到什么时候节点加入或退出
- **HDFS 只能提供一定程度的访问透明性，完全支持位置透明性、性能和伸缩透明性**

3.1.3 分布式文件系统的设计需求

并发控制

- 客户端对于文件的读写不应该影响其他客户端对同一文件的读写
- **HDFS任何时候只允许有一个程序写入某个文件**

文件复制

- 一个文件可以拥有不同位置的多个副本
- **HDFS采用多副本机制**

硬件和操作系统异构性

- 可以在不同的操作系统和计算机上实现同样的客户端和服务器端程序
- **HDFS 采用JAVA，具有很好的跨平台性**

3.1.3 分布式文件系统的设计需求

可伸缩性

- 支持节点的动态加入和退出
- **HDFS建立在大规模廉价机器上的分布式文件系统集群，具有良好的可伸缩性**

容错

- 保证文件服务器在客户端或服务器出现问题的时候能正常使用
- **HDFS 具有多副本机制和故障自动检测、恢复机制**

安全

- 保障系统的安全性
- **HDFS 安全性较弱**

思考题

□为什么分布式文件系统通常采用多副本存储？

- 为保证数据的完整性，分布式文件系统通常采用多副本存储。
- 文件块被复制为多个副本，存储在不同的节点上，存储同一文件块的不同副本的各个节点分布在不同的机架上。
- 这样，单个节点出现故障时，可以快速调用副本重启单个节点上的计算过程，而不用重启整个计算过程；整个机架出故障时，不会丢失所有文件块。
- 文件块的大小和副本个数通常可以由用户指定。

3.2 HDFS简介

- HDFS支持流数据读取和处理超大规模文件，运行在廉价的普通机器组成的集群上。
- HDFS设计目标：
  - 兼容廉价的硬件设备：快速检测硬件故障、进行自动恢复
  - 流数据读写：满足批量数据处理的要求，流式方式访问文件系统
  - 大数据集：HDFS文件通常可以达到GB甚至TB级别
  - 简单的文件模型：一次写入，多次读取
  - 强大的跨平台兼容性：支持JVM的机器都可以运行HDFS

3.2 HDFS简介

HDFS自身的应用局限性：

- **不适合低延迟数据访问**

  面向大规模数据批量处理，采用流式数据读取，数据吞吐率高，有较高的延迟，不适合需要低延迟（数十毫秒）的应用场合（应选择HBase）

- **无法高效存储大量小文件**

  过多小文件会给系统扩展性和性能带来很多问题

- **不支持多用户写入及任意修改文件**

  HDFS只允许一个文件有一个写入者，不允许多个用户对同一个文件执行写操作；只允许对文件执行追加操作，不支持随机写操作。

- **HDFS适合用来做大数据分析的底层存储服务，不适用于网盘等应用。**

思考题

◘为什么HDFS适合存储非常大的文件，不适合存储小文件？

1）首先，HDFS采用NameNode来管理文件系统的元数据，这些元数据被保存在内存中，从而使客户端可以快速获取文件实际存储位置。通常每个文件、文件夹和块的存储信息大约占150B，如果有1000万个小文件，每个文件对应一个块，则NameNode要保存这些元数据信息至少需要3GB的内存空间（150B*2*10000000=3GB）。这时，元数据检索的效率就很低，需要花费较多的时间找到一个文件的实际存储位置。如果扩展到数十亿个小文件，元数据所需占用的内存空间大大增加，硬件水平无法满足。

思考题

为什么HDFS适合存储非常大的文件，不适合存储小文件？

2）其次，用MapReduce处理大量小文件时，会产生过多的Map任务，线程管理开销会大大增加，因此，处理大量小文件的速度远远低于处理同样数量的大文件的速度。

3）再次，访问大量小文件的速度远远低于访问几个大文件的速度，因为访问大量小文件，需要不断从一个数据节点跳到另一个数据节点，严重影响性能。

3.3 HDFS相关概念

3.3.1 块

3.3.2名称节点和数据节点

3.3.3 第二名称节点

3.3.1 块

- HDFS中一个文件被分成多个块，以块作为存储单位。
- 默认一个块64MB（Hadoop2.X是128MB）块的大小可以在配置文件中设置。
- 块的大小远远大于普通文件系统，可以最小化寻址开销。

HDFS寻址开销不仅包括磁盘寻道开销，还包括数据块的定位开销，当客户端需要访问一个文件时，首先从名称节点获取组成这个文件的数据块的位置列表，然后根据位置列表获取实际存储各个数据块的数据节点的位置，最后，数据节点根据数据块信息在本地Linux文件系统中找到对应的文件，并把数据返回给客户端，设计成一个比较大的块，可以减少每个块儿中数据的总的寻址开销，相对降低了单位数据的寻址开销

3.3.1 块

HDFS采用抽象的块概念可以带来以下几个明显的好处：

- **支持大规模文件存储：**文件以块为单位进行存储，一个大规模文件可以被分拆成若干个文件块，不同的文件块可以被分发到不同的节点上，因此，一个文件的大小不会受到单个节点的存储容量的限制，可以远远大于网络中任意节点的存储容量
- **简化系统设计**：首先，大大简化了存储管理，因为文件块大小是固定的，这样就可以很容易计算出一个节点可以存储多少文件块；其次，方便了元数据的管理，元数据不需要和文件块一起存储，可以由其他系统负责管理元数据
- **适合数据备份：**每个文件块都可以冗余存储到多个节点上，大大提高了系统的容错性和可用性

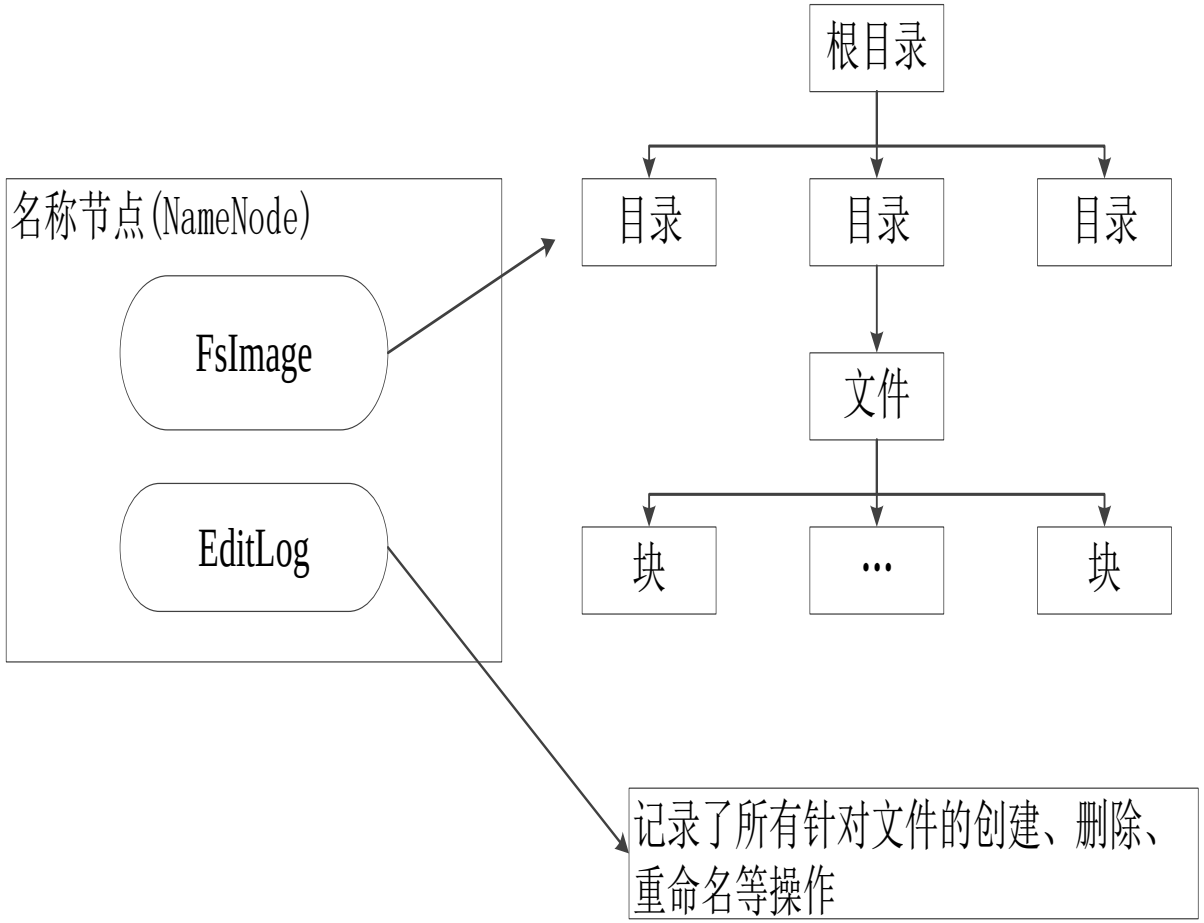
3.3.2名称节点和数据节点

- NameNode（Master Node）的作用是管理文件目录结构，是管理DataNode的，是整个HDFS集群的管家。
- NameNode维护两套数据：
  - 一套是文件目录与数据块之间的关系（静态的，存放在磁盘上，通过FsImage和Edits来维护）
  - 另一套是数据块与DataNode之间的关系（动态的，不持久化到磁盘，每当集群启动的时候，会自动建立这些信息）

3.3.2名称节点和数据节点

- NameNode负责管理分布式文件系统的命名空间(Namespace)，保存了两个核心的数据结构：

- **FsImage：** 用于维护文件系统树以及文件树中所有的文件和目录的元数据（Block数量、副本数量、权限等信息）
- **EditLog：** 记录了所有针对文件的创建、删除、打开、关闭、重命名等操作（不包括查询）相当于日志文件（EditLog）



NameNode的数据结构

3.3.2名称节点和数据节点

FsImage文件：

- FsImage文件没有记录每个块存储在哪个数据节点。而是由名称节点把这些映射信息保留在内存中。
- 当数据节点加入HDFS集群时，数据节点会把自己所包含的块列表告知给名称节点，此后会定期执行这种告知操作，以确保名称节点的块映射是最新的。

3.3.2名称节点和数据节点

补充：查看NameNode的目录结构

- 启动Hadoop，并用jps判断是否成功启动：

```
hadoop@dblab-VirtualBox:/usr/local/hadoop$ ./sbin/start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/hadoop/logs/hadoop-hadoop-nameno
de-dblab-VirtualBox.out
localhost: starting datanode, logging to /usr/local/hadoop/logs/hadoop-hadoop-datano
de-dblab-VirtualBox.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-hadoop
-secondarynamenode-dblab-VirtualBox.out
hadoop@dblab-VirtualBox:/usr/local/hadoop$ jps
9009 NameNode
9363 SecondaryNameNode
9143 DataNode
9485 Jps
```

3.3.2名称节点和数据节点

- NameNode的元数据的保存目录在以下配置文件中设置：/usr/local/hadoop/etc/hadoop/hdfs-site.xml
- 转到NameNode中元数据的保存目录并用tree命令查看：

```
hadoop@dblab-VirtualBox:/usr/local/hadoop$ cd tmp
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp$ cd dfs
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs$ tree name
name
└── current
    ├── edits_0000000000000000001-0000000000000000028
    ├── edits_0000000000000000029-0000000000000
    ├── edits_0000000000000000088-0000000000000
    ├── edits_0000000000000000089-0000000000000
    ├── edits_0000000000000000091-0000000000000
    ├── edits_0000000000000000123-0000000000000
    ├── edits_0000000000000002054-0000000000000002054
    ├── edits_0000000000000002055-0000000000000002056
    ├── edits_0000000000000002057-0000000000000002057
    ├── edits_0000000000000002058-0000000000000002059
    ├── edits_inprogress_0000000000000002060
    ├── fsimage_0000000000000002057
    ├── fsimage_0000000000000002057.md5
    ├── fsimage_0000000000000002059
    ├── fsimage_0000000000000002059.md5
    ├── seen_txid
    └── VERSION
└── in_use.lock

1 directory, 138 files
```

3.3.2名称节点和数据节点

- 查看到的文件的说明：
  - edits_0000xxxx -> 操作日志文件（记录了对HDFS的各种更新操作）
  - edits_inprogress_000xxx -> 当前打开可写的操作日志文件
  - fsimage_000xxx -> 文件系统镜像文件（某一时刻，HDFS文件系统的快照）
  - VERSION -> HDFS版本信息的描述文件
  - in_use.lock -> 一个锁文件，NameNode使用该文件为存储目录加锁，用来保证NameNode的唯一性，防止一台机器同时启动多个NameNode进程导致目录数据不一致。

3.3.2名称节点和数据节点

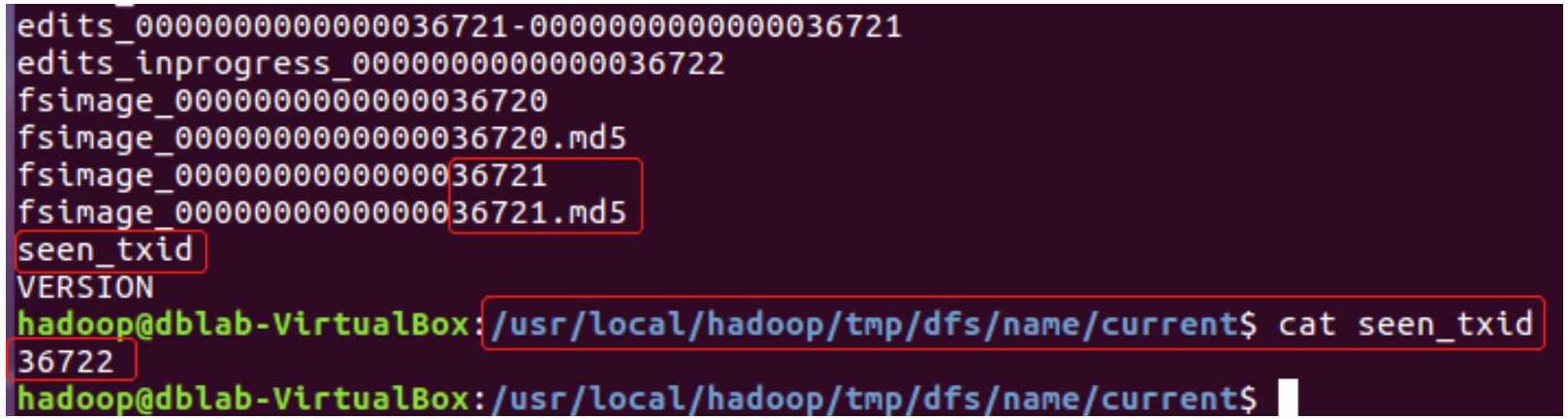
查看/usr/local/hadoop/tmp/dfs/name/current/中的VERSION文件：

```
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs/name/current$ cat VERSION
#Tue Mar 22 21:48:09 CST 2022
namespaceID=30824014
clusterID=CID-9e5549b3-1a15-4314-b36f-330733536097
cTime=1646198280830
storageType=NAME_NODE
blockpoolID=BP-1856939348-127.0.1.1-1646198280830
layoutVersion=-66
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs/name/current$
```

- **namespaceID/clusterID/blockpoolID：** 这些都是 HDFS 集群的唯一标识符；
- **cTime：** NameNode 存储系统创建时间，首次格式化文件系统这个属性是 0，当文件系统升级之后，该值会更新到升级之后的时间戳；
- **storageType：** 说明这个文件存储的是什么进程的数据结构信息（如果是 DataNode，storageType=DATA_NODE）；
- **layoutVersion：** 表示 HDFS 永久性数据结构的版本信息，是一个负整数。

3.3.2名称节点和数据节点

- 查看/usr/local/hadoop/tmp/dfs/name/current/中的seen_txid文件：



- /usr/local/hadoop/tmp/dfs/name/current/seen_txid 非常重要，是存放 transactionID 的文件，初始化之后是 0，它代表的是 namenode 里面的 edits_*文件的尾数，namenode 重启的时候，会按照 seen_txid 的数字，按序遍历edits_0000001~到 seen_txid 的数字。所以当HDFS 发生异常重启的时候，一定要比对 seen_txid 内的数字是不是 edits 最后的尾数。

3.3.2名称节点和数据节点

- 使用oiv命令查看fsimage文件
  - 语法：hdfs oiv -p 文件类型 -i 镜像文件 -o 转换后文件输出路径

```
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs/name/current$ hdfs oiv -p xml -i fsimage_0000000000000006521 -o ~/fsimage.xml
2022-03-29 21:25:28,896 INFO offlineImageViewer.FSImageHandler: Loading 6 strings
2022-03-29 21:25:29,131 INFO namenode.FSDirectory: GLOBAL serial map: bits=29 maxEntries=536870911
2022-03-29 21:25:29,131 INFO namenode.FSDirectory: USER serial map: bits=24 maxEntries=16777215
2022-03-29 21:25:29,131 INFO namenode.FSDirectory: GROUP serial map: bits=24 maxEntries=16777215
2022-03-29 21:25:29,132 INFO namenode.FSDirectory: XATTR serial map: bits=24 maxEntries=16777215
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs/name/current$ vim ~/fsimage.xml
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs/name/current$
```

```
1 <?xml version="1.0"?>
2 <fsimage><version><layoutVersion>-66</layoutVersion><onDiskVersion>1</onDiskVersion><oivRevision>a3b9c37a397ad4188041dd80621bdeefc46885f2</oiv
  Revision></version>
3 <NameSection><namespaceId>30824014</namespaceId><genstampV1>1000</genstampV1><genstampV2>1704</genstampV2><genstampV1Limit>0</genstampV1Limit>
  <lastAllocatedBlockId>1073742514</lastAllocatedBlockId><txid>6521</txid></NameSection>
4 <ErasureCodingSection>
5 <erasureCodingPolicy>
6 <policyId>1</policyId><policyName>RS-6-3-1024k</policyName><cellSize>1048576</cellSize><policyState>DISABLED</policyState><ecSchema>
7 <codecName>rs</codecName><dataUnits>6</dataUnits><parityUnits>3</parityUnits></ecSchema>
8 </erasureCodingPolicy>
9
10 <erasureCodingPolicy>
11 <policyId>2</policyId><policyName>RS-3-2-1024k</policyName><cellSize>1048576</cellSize><policyState>DISABLED</policyState><ecSchema>
12 <codecName>rs</codecName><dataUnits>3</dataUnits><parityUnits>2</parityUnits></ecSchema>
13 </erasureCodingPolicy>
```

3.3.2名称节点和数据节点

- 使用oev命令查看edits文件
  - 语法：hdfs oev -p 文件类型 -i 日志文件 -o 转换后文件输出路径

```
-rw-rw-r-- 1 hadoop hadoop      5 3月  26 23:07 seen_txid
-rw-rw-r-- 1 hadoop hadoop    212 3月  26 09:05 VERSION
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs/name/current$ hdfs oev -p xml -i edits_0000000000000006470-0000000000000006477 -o ~/edits.xml
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs/name/current$ vim ~/edits.xml
```

```
 1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
 2 <EDITS>
 3   <EDITS_VERSION>-66</EDITS_VERSION>
 4   <RECORD>
 5     <OPCODE>OP_START_LOG_SEGMENT</OPCODE>
 6     <DATA>
 7       <TXID>6470</TXID>
 8     </DATA>
 9   </RECORD>
10   <RECORD>
11     <OPCODE>OP_ADD</OPCODE>
12     <DATA>
13       <TXID>6471</TXID>
14       <LENGTH>0</LENGTH>
15       <INODEID>17696</INODEID>
16       <PATH>/user/hadoop/mywc.jar._COPYING_</PATH>
17       <REPLICATION>1</REPLICATION>
18       <MTIME>1648263112681</MTIME>
19       <ATIME>1648263112681</ATIME>
20       <BLOCKSIZE>134217728</BLOCKSIZE>
21       <CLIENT_NAME>DFSClient_NONMAPREDUCE_-1547413083_1</CLIENT_NAME>
22       <CLIENT_MACHINE>127.0.0.1</CLIENT_MACHINE>
23       <OVERWRITE>true</OVERWRITE>
```

3.3.2 名称节点和数据节点

NameNode的启动过程

- NameNode在启动的过程中处于“安全模式”，只能对外提供读操作，无法提供写操作。“安全模式”是为了让DataNode汇报自己当前所持有的Block信息给NameNode来补全元数据。启动过程结束后，系统就会退出安全模式，进入正常运行状态，对外提供读写操作；
- 在名称节点启动的时候，它会将FsImage文件中的内容加载到内存中，之后再执行EditLog文件中的各项操作，使得内存中的元数据和实际的同步，存在内存中的元数据支持客户端的读操作。

3.3.2 名称节点和数据节点

- 一旦在内存中成功建立文件系统元数据的映射，则创建一个新的FsImage文件和一个空的EditLog文件
- 名称节点起来之后，HDFS中的更新操作会重新写到EditLog文件中，因为FsImage文件一般都很大（GB级别的很常见），如果所有的更新操作都往FsImage文件中添加，这样会导致系统运行的十分缓慢，但是，如果往EditLog文件里面写就不会这样，因为EditLog 要小很多。

3.3.2 名称节点和数据节点

□数据节点DataNode/Slave Node

- DataNode是分布式文件系统HDFS的工作节点，负责数据的存储和读取，会根据客户端或者是NameNode的调度来进行数据的存储和检索，并且向NameNode定期发送自己所存储的块的列表
- 每个DataNode中的数据会被保存在各自节点的本地Linux文件系统中

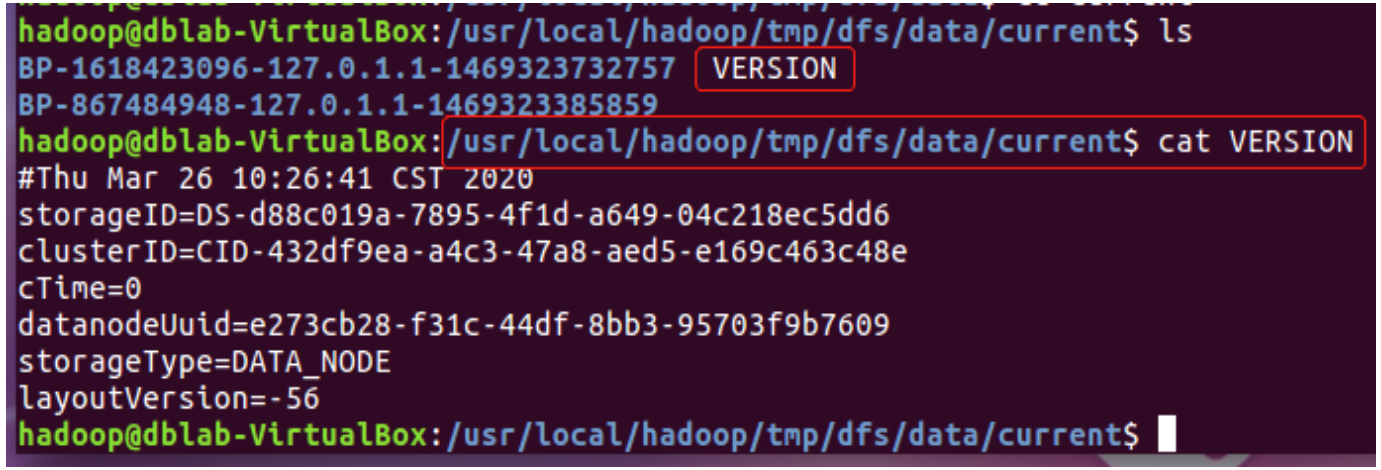
```
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs$ ls
data  name  namesecondary
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs$ tree data
data
├── current
│   ├── BP-1618423096-127.0.1.1-1469323732757
│   │   ├── current
│   │   │   ├── dfsUsed
│   │   │   ├── finalized
│   │   │   │   └── subdir0
│   │   │   │       └── subdir0
│   │   │   │           ├── blk_1073741881
│   │   │   │           ├── blk_1073741881_1057.meta
│   │   │   │           ├── blk_1073741882
│   │   │   │           ├── blk_1073741882_1058.meta
```

```
│   │   │   │           ├── blk_1073742002
│   │   │   │           └── blk_1073742002_1182.meta
│   │   │   ├── rbw
│   │   │   └── VERSION
│   │   ├── scanner.cursor
│   │   └── tmp
│   └── BP-867484948-127.0.1.1-1469323385859
│       ├── current
│       │   ├── dfsUsed
│       │   ├── finalized
│       │   ├── rbw
│       │   └── VERSION
│       ├── scanner.cursor
│       └── tmp
│   └── VERSION
└── in_use.lock

13 directories, 138 files
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs$
```

3.3.2 名称节点和数据节点

- /usr/local/hadoop/tmp/dfs/data/current/中的VERSION文件：



- **storageID**=DS-d88c019a-... //相对于DataNode来说是唯一的，用于在NameNode处标识DN
- **clusterID**=CID-432df9ea-a4c3-… //第一次连接NameNode时就会从中获取
- **cTime=0** //DataNode 存储系统创建时间，首次格式化文件系统这个属
- 性是 0，当文件系统升级之后，该值会更新到升级之后的时间戳；
- **storageType**=DATA_NODE //storageType将这个目录标志为DataNode数据存储目录
- **layoutVersion**=-18 //表示 HDFS 永久性数据结构的版本信息，是一个负整数。

3.3.2 名称节点和数据节点

□数据节点DataNode/Slave Node的工作机制

- DataNode启动后向NameNode注册，通过后周期性（1小时）的将本DataNode上保存的Block信息汇报给NameNode
- NameNode在接收到的Block信息以及该Block所在的DataNode信息等保存在内存中
- DataNode通过向NameNode发送心跳保持与其联系（3秒一次），心跳返回结果带有NameNode的命令，返回的命令为：如块的复制，删除某个数据块.....

3.3.2 名称节点和数据节点

- NameNode如果10分钟没有收到DataNode的心跳，则认为它已经lost，并copy其上的Block到其它DataNode
- DataNode在其文件创建后三周进行验证其checkSum的值是否和文件创建时的checkSum值一致

3.3.3 第二名称节点

NameNode运行期间Edits不断变大的问题

- 在NameNode运行期间，HDFS的所有更新操作都是直接写到Edits中，时间长了，Edits文件将会变得很大；
- 虽然这对NameNode运行的时候是没有什么明显影响的，但是，当NameNode重启的时候，NameNode需要先将FsImage里面的所有内容映像到内存中，然后再一条一条地执行Edits中的记录，当Edits文件非常大的时候，会导致NameNode启动操作非常慢，而在这段时间内HDFS系统处于安全模式，一直无法对外提供写操作，影响了用户的使用。

3.3.3 第二名称节点

□如何解决Edits不断变大的问题？----SecondaryNameNode

- SecondaryNameNode是HDFS架构中的一个组成部分，它是用来保存NameNode中对HDFS 元数据信息的备份，并减少NameNode重启的时间。SecondaryNameNode一般是单独运行在一台机器上。

- 查看SecondaryNameNode的目录结构：

```
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs$ ls
data  name  namesecondary
hadoop@dblab-VirtualBox:/usr/local/hadoop/tmp/dfs$ tree namesecondary
namesecondary
└── current
    ├── edits_0000000000000000001-0000000000000000028
    ├── edits_0000000000000000029-0000000000000000087
    ├── edits_0000000000000000088-0000000000000000088
    ├── edits_0000000000000000089-0000000000000000090
    ├── edits_0000000000000000123-0000000000000000129
    ├── edits_0000000000000000130-0000000000000000132
    ├── edits_0000000000000000134-0000000000000000346
    ├── edits_0000000000000002062-0000000000000002063
    ├── edits_0000000000000002064-0000000000000002065
    ├── edits_0000000000000002066-0000000000000002067
    ├── fsimage_0000000000000002065
    ├── fsimage_0000000000000002065.md5
    ├── fsimage_0000000000000002067
    ├── fsimage_0000000000000002067.md5
    └── VERSION
└── in_use.lock

1 directory, 104 files
```

防止一台机器同时启动多个SecondaryNameNode进程导致目录数据不一致

3.3.3 第二名称节点

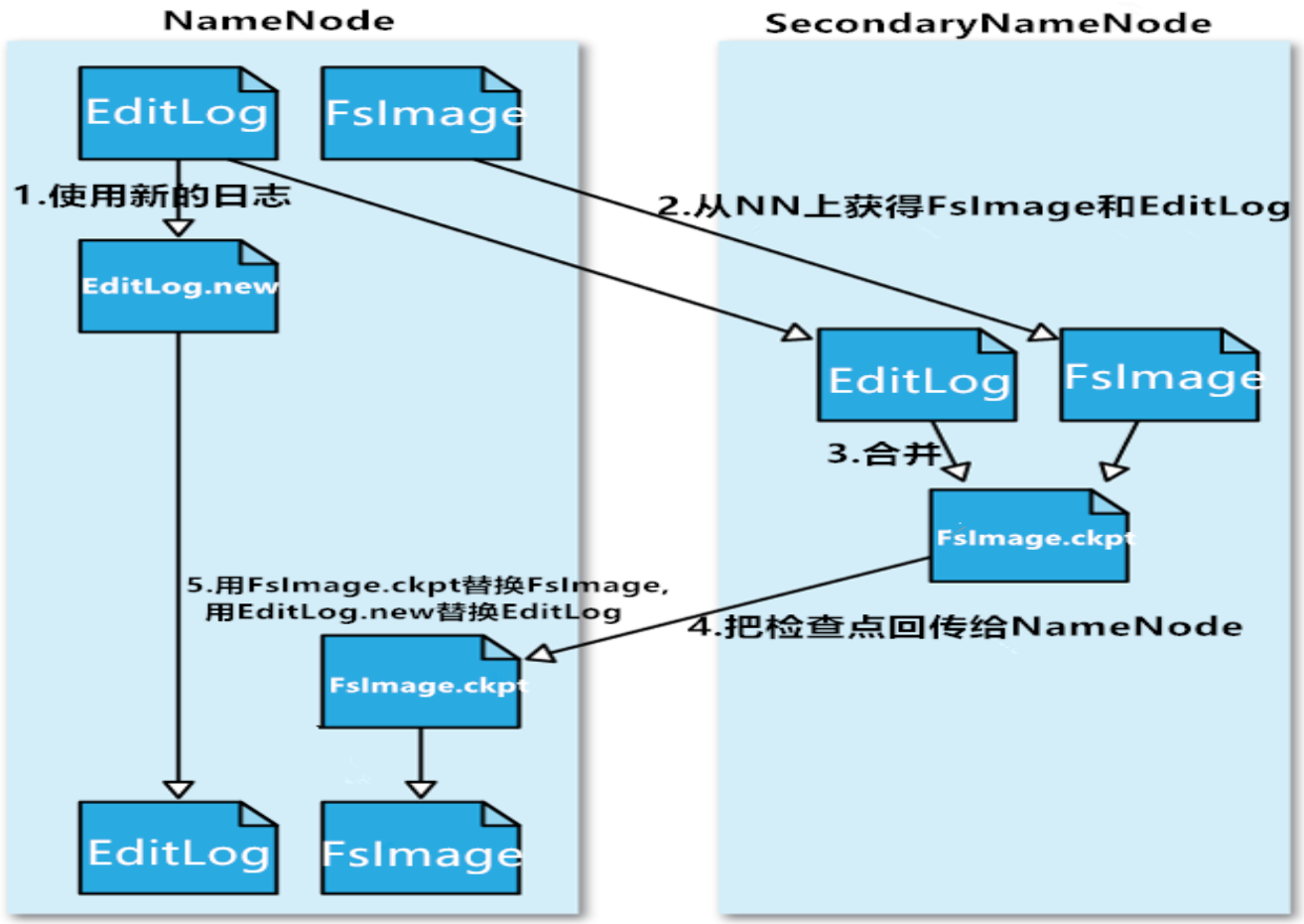
SecondaryNameNode的功能

- 首先，可以完成Edits和FsImage的合并操作，减小Edits文件大小，缩短NameNode重启时间；
- 其次，可以作为NameNode的“检查点”，保存NameNode中的元数据信息。

3.3.3 第二名称节点

SecondaryNameNode的工作过程

(1) SNN会定期和NN通信，请求其停止使用EditLog文件，暂时将新的写操作写到一个新的文件EditLog.new上来，这个操作是瞬间完成，上层写日志的函数完全感觉不到差别；

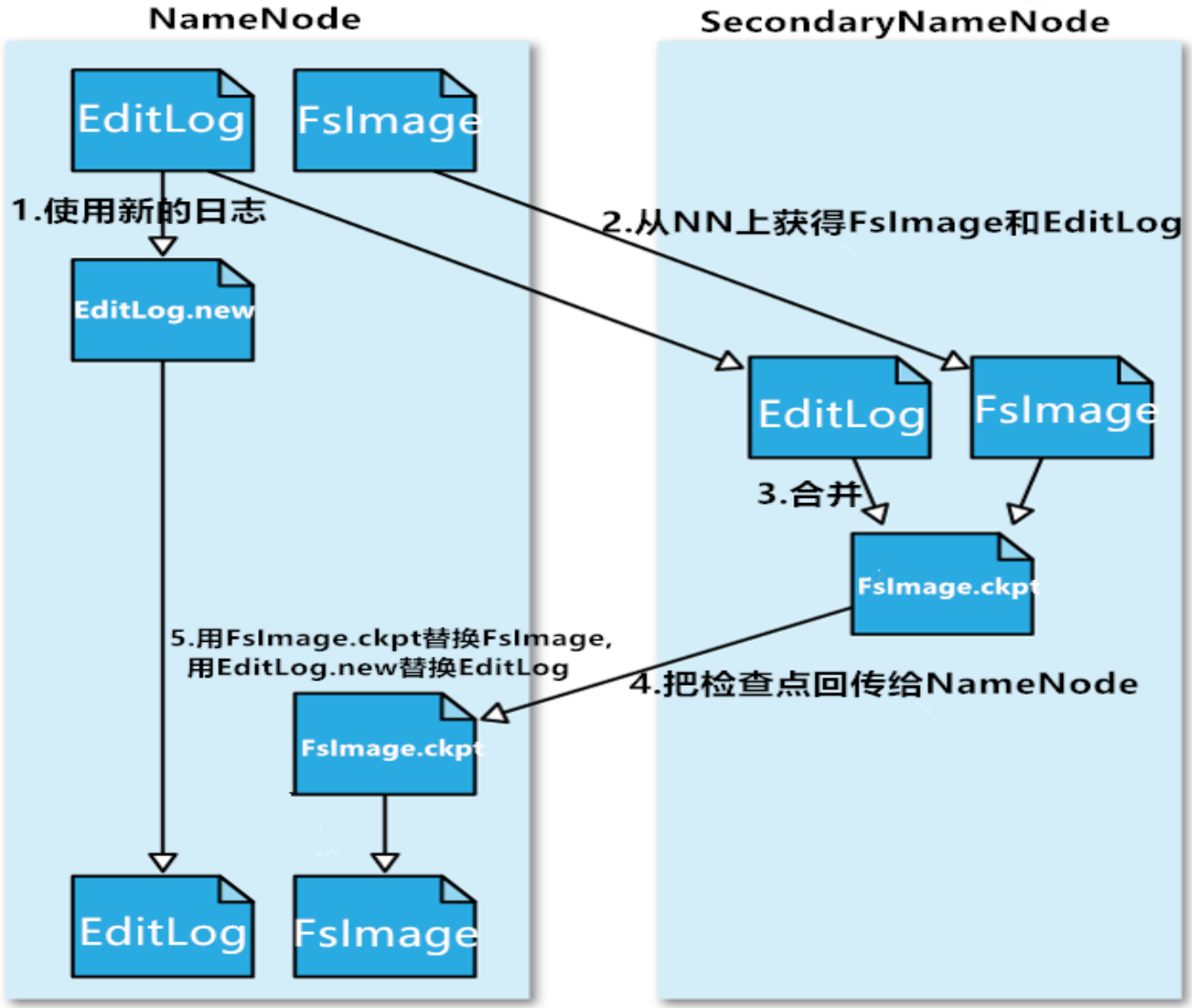


定期：当EditLog文件的大小达到一个临界值（默认是64MB）或者间隔一段时间（默认是1小时）的时候checkpoint会触发SecondaryNameNode进行工作。

3.3.3 第二名称节点

(2) SNN通过HTTP GET方式从NN上获取到FsImage和EditLog文件，并下载到本地的相应目录下；

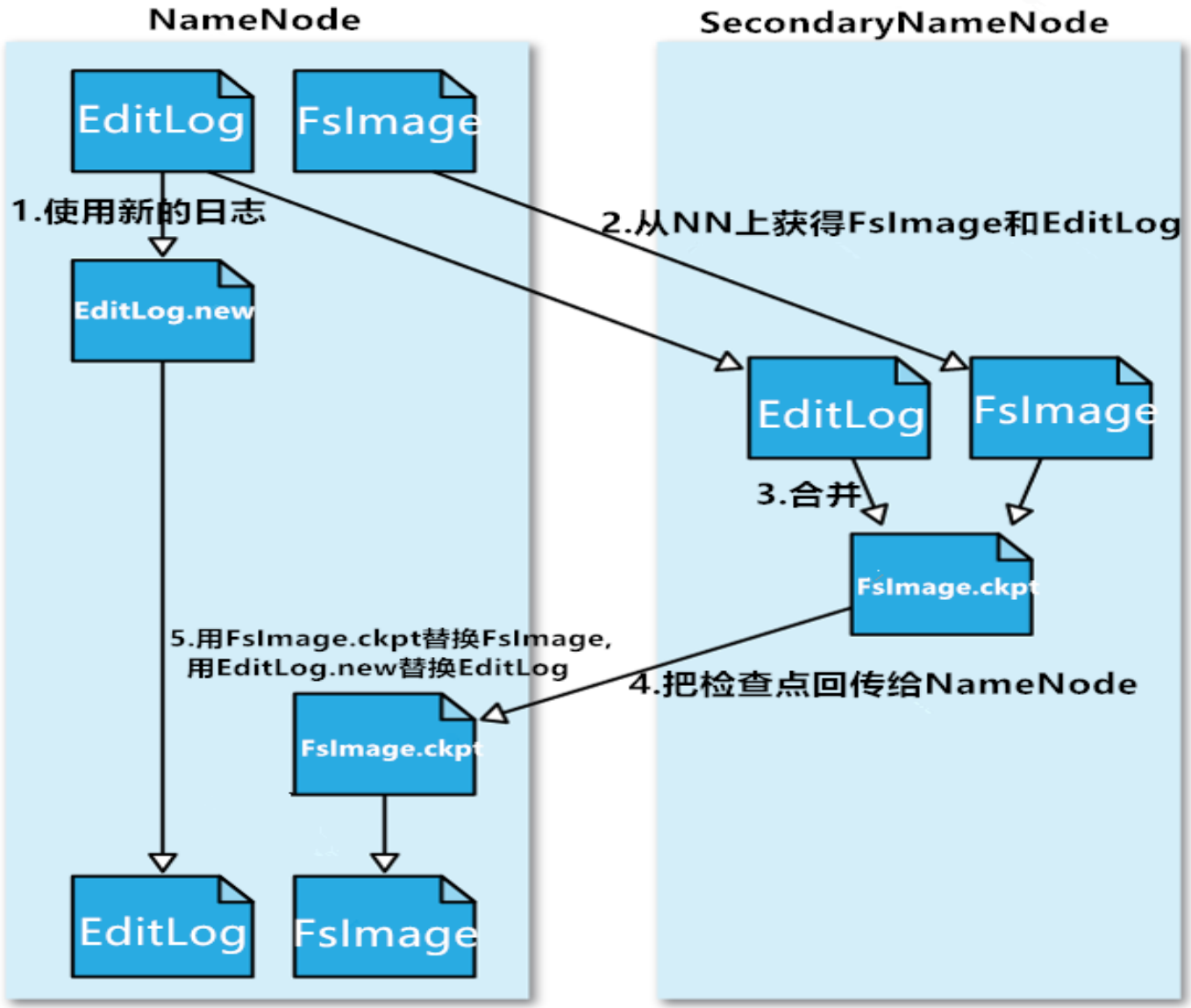
(3) SNN将下载下来的FsImage载入到内存，然后一条一条地执行EditLog文件中的各项更新操作，使得内存中的FsImage保持最新；这个过程就是EditLog和FsImage文件合并；



3.3.3 第二名称节点

(4) SNN执行完上述操作后，会通过post方式将新的FsImage文件发送到NN节点上

(5) NN将从SNN接收到的新的FsImage替换旧的FsImage文件，同时将EditLog.new替换EditLog文件，通过这个过程EditLog就变小了。



思考题

- **HDFS中NameNode、DataNode和SecondaryNameNode的职责分别是什么？**

- **NameNode：Master节点，只有一个，管理HDFS的名称空间和数据块映射信息；配置副本策略；处理客户端请求。**
- **DataNode：Slave节点，存储实际的数据；执行数据块的读写；汇报存储信息给NameNode。**
- **SecondaryNameNode：辅助NameNode，分担其工作量；定期合并FsImage和EditLog，推送给NameNode；紧急情况下，可辅助恢复NameNode，**

思考题

SecondaryNameNode是NameNode的热备份吗?

- 热备份：b是a的热备份，如果a坏掉。那么b立即运行代替a的工作。
- 冷备份：b是a的冷备份，如果a坏掉。那么b不能立即代替a工作。但是b上存储a的一些信息，减少a坏掉之后的损失。

答：SecondaryNameNode并非NameNode的热备份，当NameNode 挂掉的时候，它并不能立即替换 NameNode 并提供服务，但是SNN上存储NN上的一些信息，减少NN坏掉之后的损失。

3.4 HDFS体系结构

3.4.1 HDFS体系结构概述

3.4.2 HDFS命名空间管理

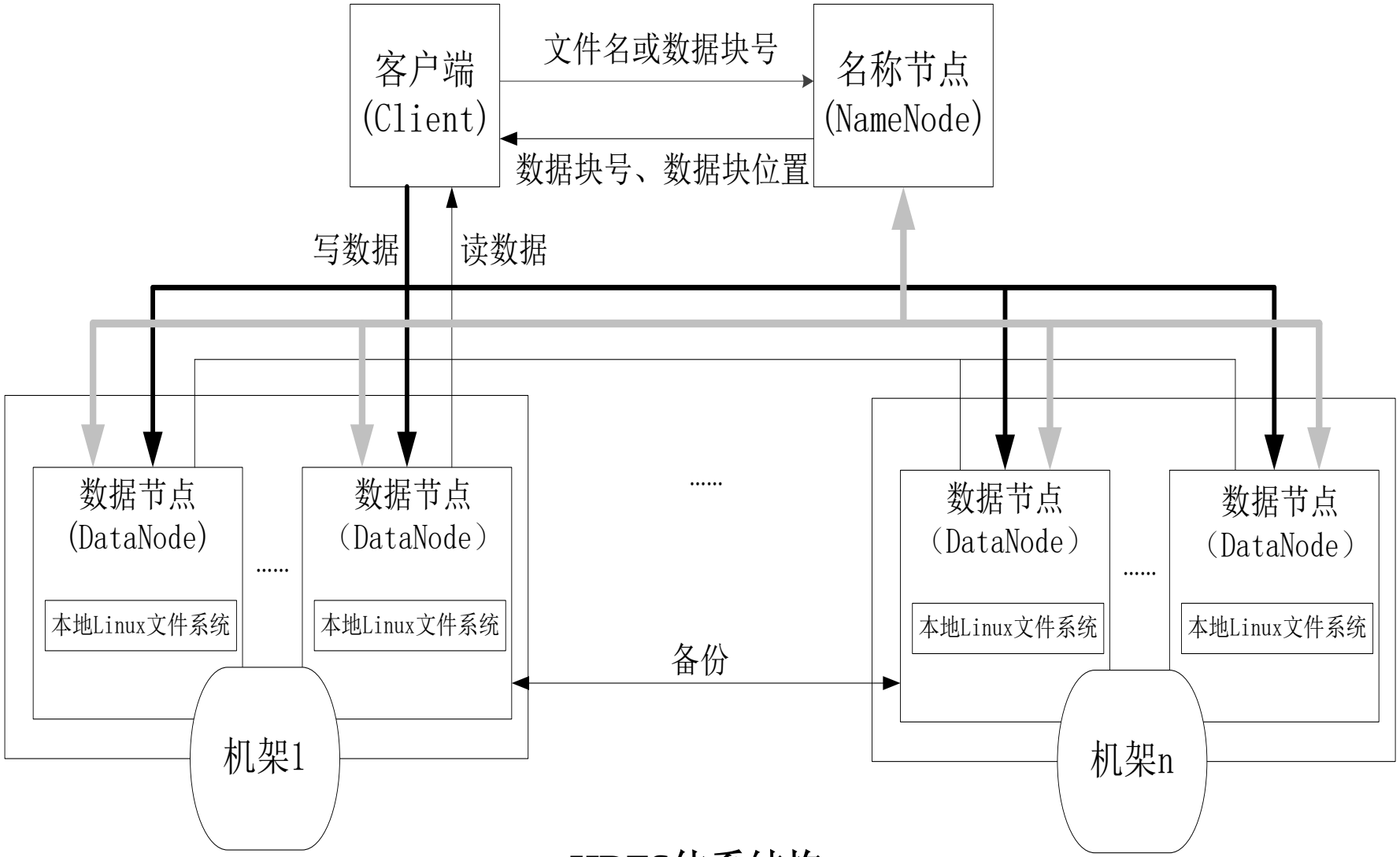
3.4.3 通信协议

3.4.4 客户端

3.4.5 HDFS体系结构的局限性

3.4.1 HDFS体系结构概述

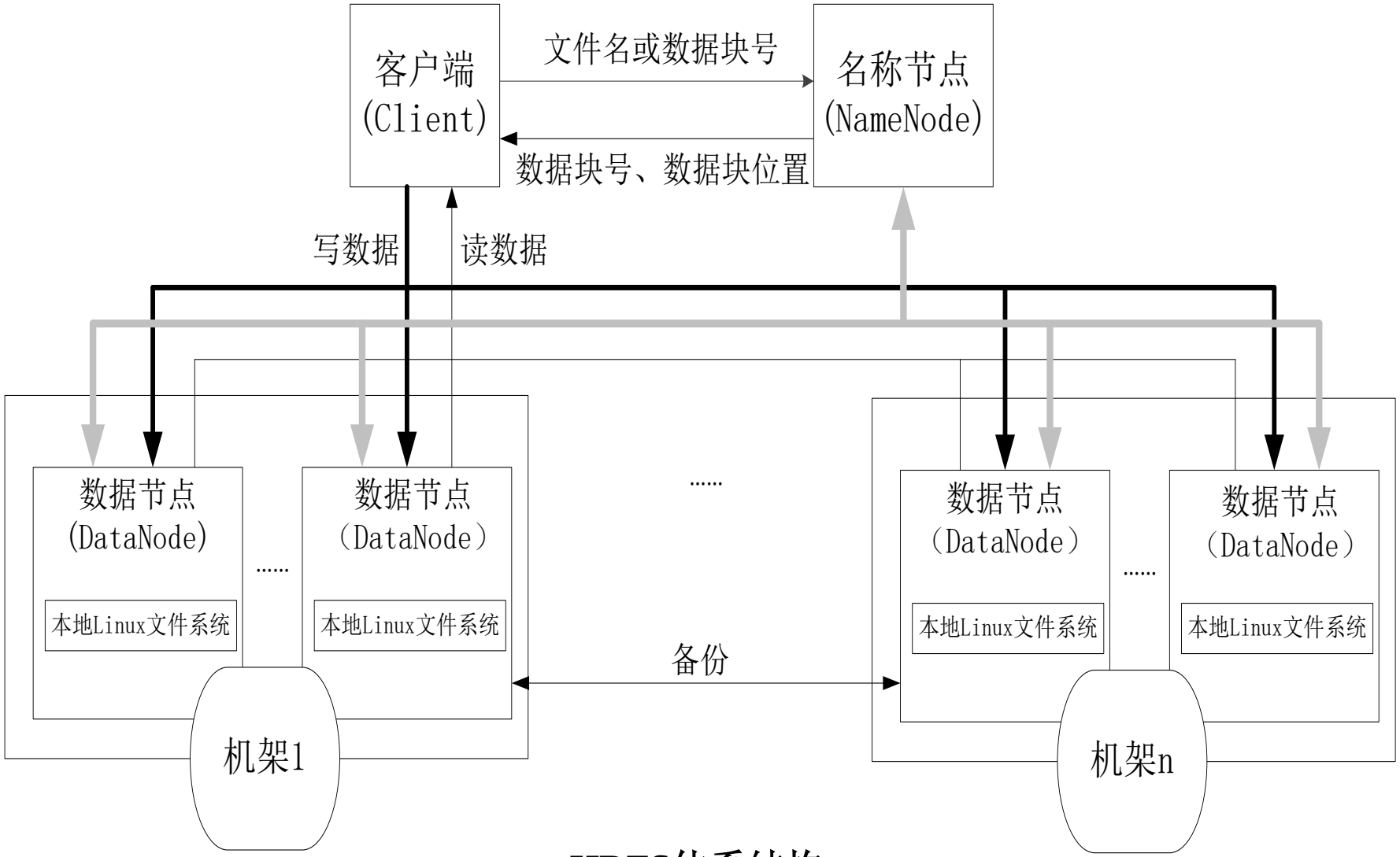
- HDFS采用了主从（Master/Slave）结构模型，一个HDFS集群包括一个名称节点（NameNode）和若干个数据节点（DataNode）



HDFS体系结构

3.4.1 HDFS体系结构概述

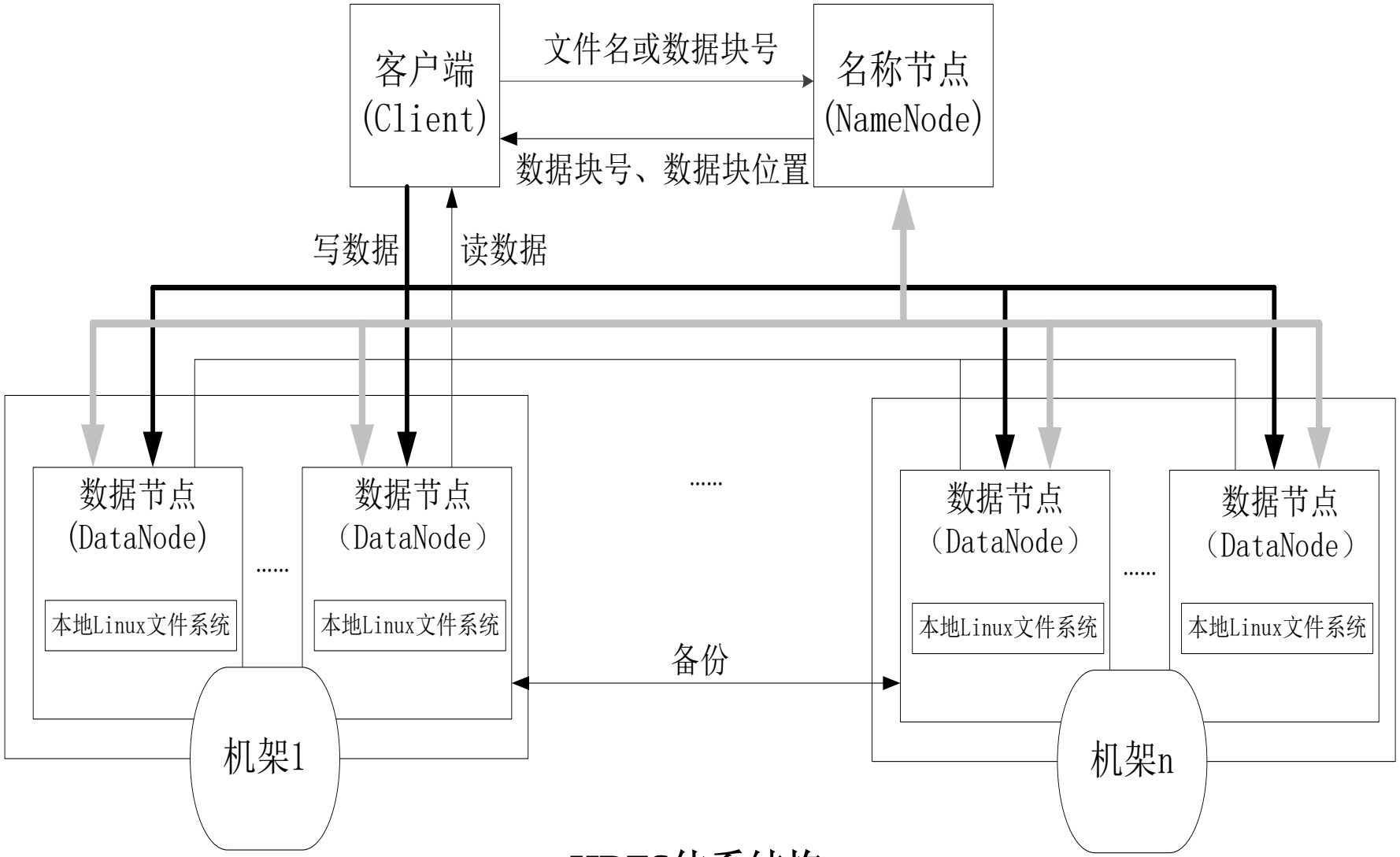
□NameNode作为中心服务器，负责管理文件系统的命名空间及客户端对文件的访问。



HDFS体系结构

3.4.1 HDFS体系结构概述

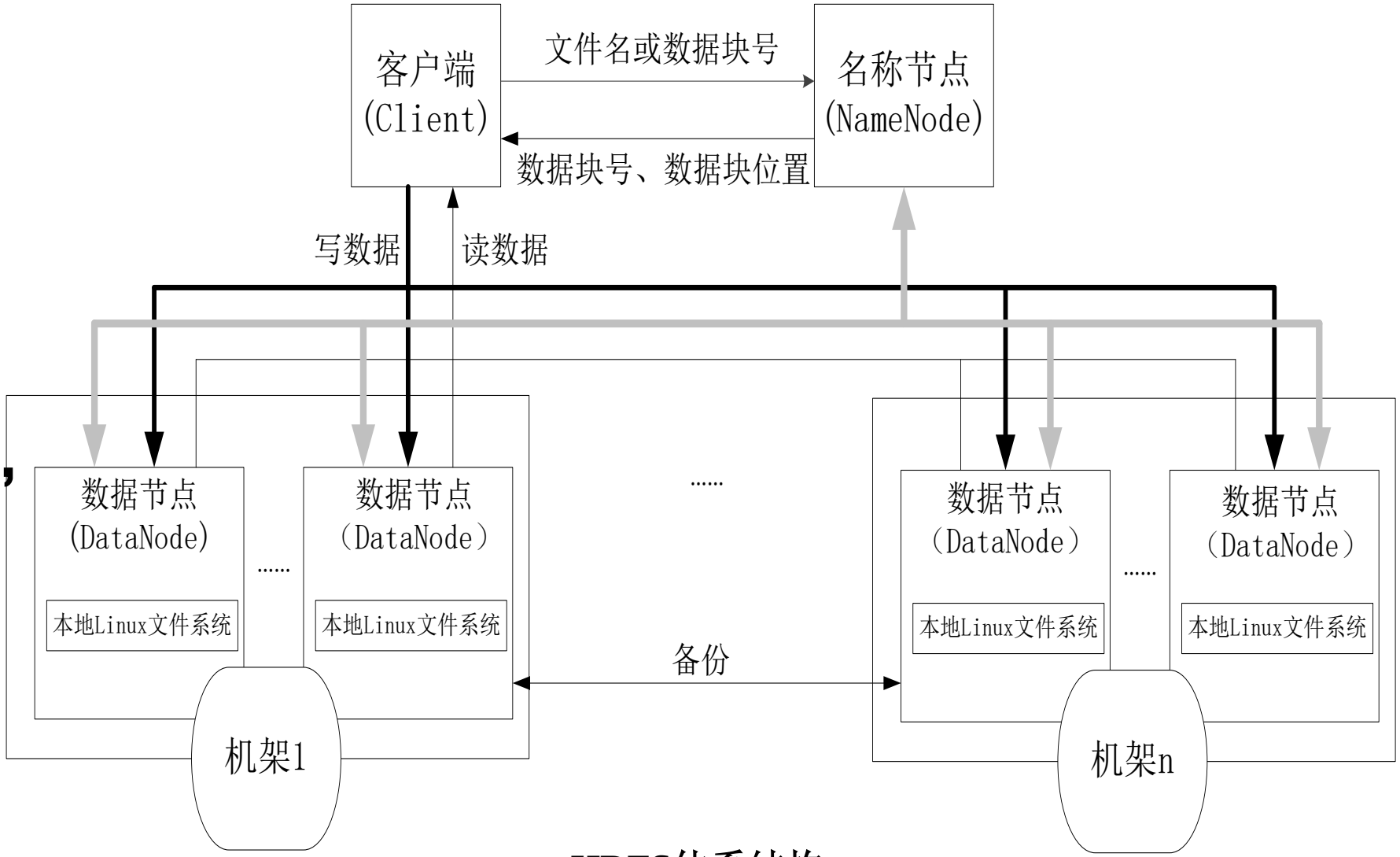
□DataNode一般是一个节点运行一个数据节点进程，负责处理文件系统客户端的读/写请求，在NameNode的统一调度下进行数据块的创建、删除和复制等操作。



HDFS体系结构

3.4.1 HDFS体系结构概述

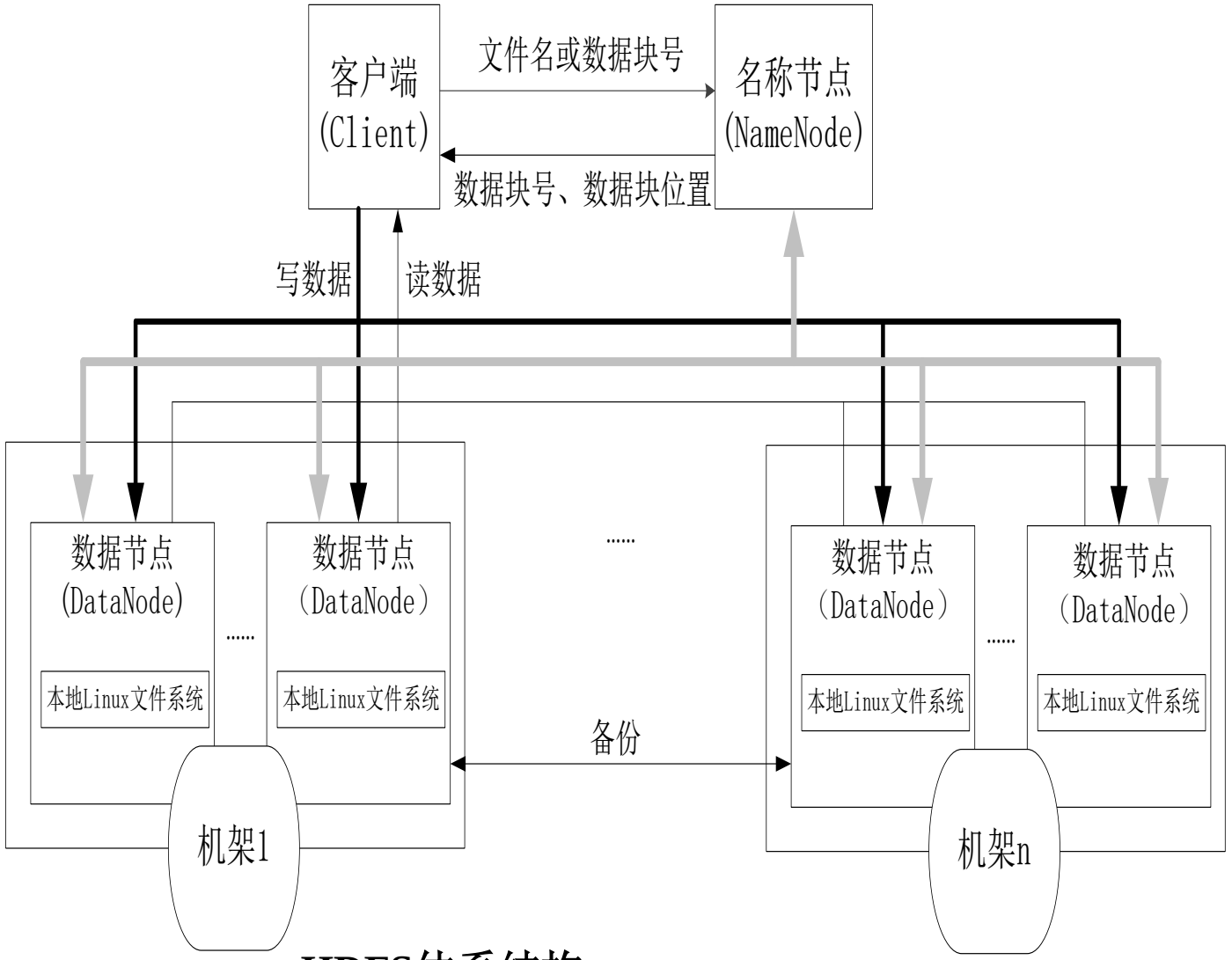
- 每个DataNode会周期性地向NameNode发送“心跳”信息，报告自己的状态，没有按时发送“心跳”信息的DataNode会被标记为“宕机”，不再给它分配任何I/O请求。



HDFS体系结构

3.4.1 HDFS体系结构概述

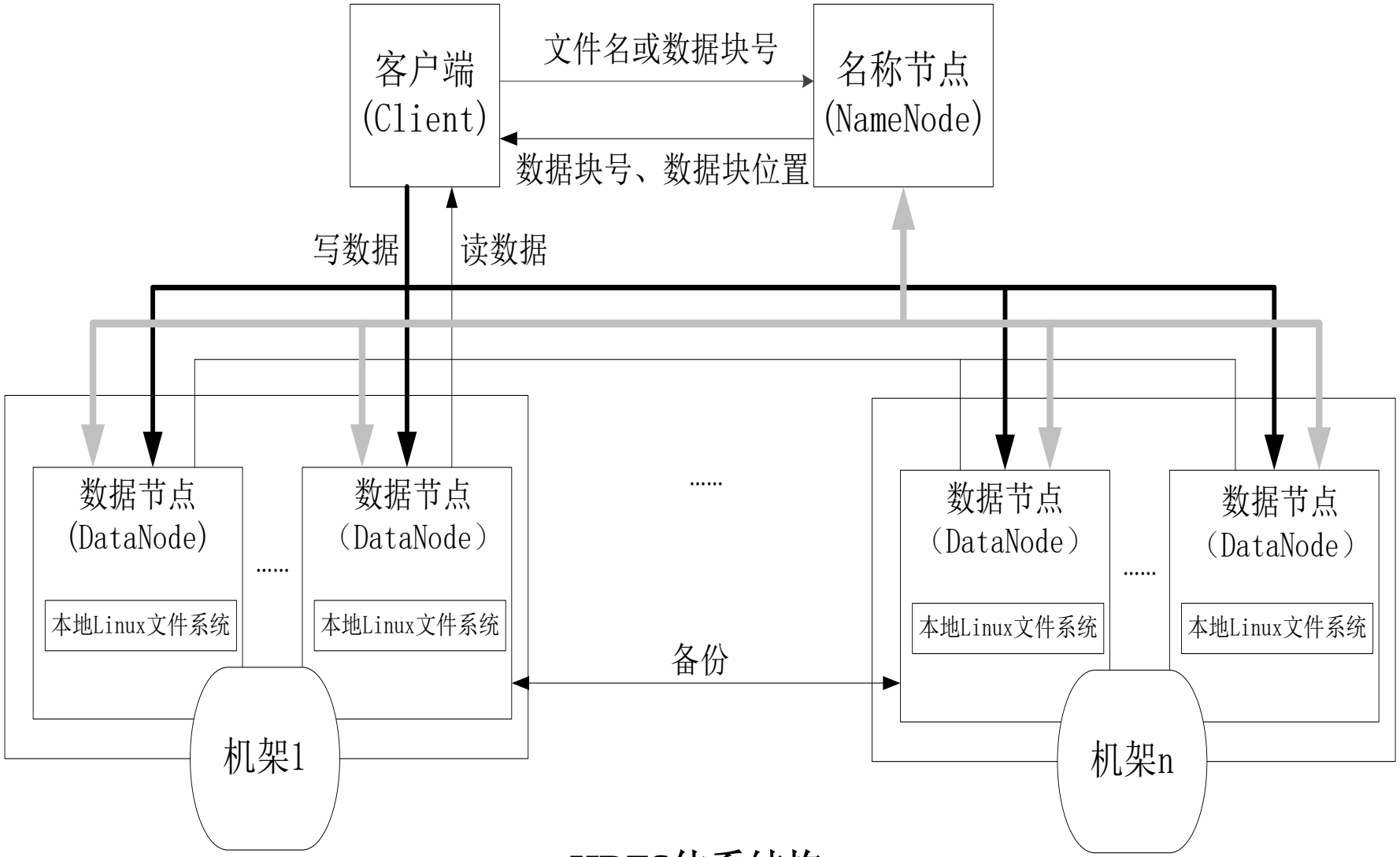
- 当客户端需要访问一个文件时，首先把文件名送给名称节点，名称节点根据文件名找到对应的数据块。
- 再根据每个数据块的信息找到实际存储各个数据块的数据节点的位置，并把数据节点位置发给客户端。
- 最后客户端直接访问这些数据节点获得数据。



HDFS体系结构

3.4.1 HDFS体系结构概述

- 在整个访问过程中，名称节点并不参与数据的传输。
- 这种设计方式，使得一个文件的数据能够在不同数据结点上**实现并发访问**，大大提高访问速度。



HDFS体系结构

3.4.2 HDFS命名空间管理

- HDFS的命名空间包含目录、文件和块
- 在HDFS1.0体系结构中，在整个HDFS集群中只有一个命名空间，并且只有唯一一个NameNode，该节点负责对这个命名空间进行管理
- HDFS使用的是传统的分级文件体系，因此，用户可以像使用普通文件系统一样，创建、删除目录和文件，在目录间转移文件，重命名文件等

3.4.3 通信协议

- HDFS是一个部署在集群上的分布式文件系统，因此，很多数据需要通过网络进行传输
- 所有的HDFS通信协议都是构建在TCP/IP协议基础之上的
- 客户端通过一个可配置的端口向NameNode主动发起TCP连接，并使用客户端协议与NameNode进行交互
- NameNode和DataNode之间则使用数据节点协议进行交互
- 客户端与DataNode的交互是通过RPC（Remote Procedure Call）来实现的。在设计上，NameNode不会主动发起RPC，而是响应来自客户端和DataNode的RPC请求

3.4.4 客户端

- 客户端是用户操作HDFS最常用的方式，HDFS在部署时都提供了客户端
- HDFS客户端是一个库，提供了HDFS文件系统接口，这些接口隐藏了HDFS实现中的大部分复杂性
- 严格来说，客户端并不算是HDFS的一部分
- 客户端可以支持打开、读取、写入等常见的操作，并且提供了类似Shell的命令行方式来访问HDFS中的数据
- 此外，HDFS也提供了Java API，作为应用程序访问文件系统的客户端编程接口

3.4.5 HDFS体系结构的局限性

HDFS只设置唯一一个NameNode，这样做虽然大大简化了系统设计，但也带来了一些明显的局限性，具体如下：

① **命名空间的限制：**NameNode是保存在内存中的，因此，NameNode能够容纳的对象（文件、块）的个数会受到内存空间大小的限制。

② **性能的瓶颈：**整个分布式文件系统的吞吐量，受限于单个名称节点的吞吐量。

③ **隔离问题：**由于集群中只有一个NameNode，只有一个命名空间，因此，无法对不同应用程序进行隔离。

④ **集群的可用性：**一旦这个唯一的NameNode发生故障，会导致整个集群不可用。

3.5 HDFS的存储原理

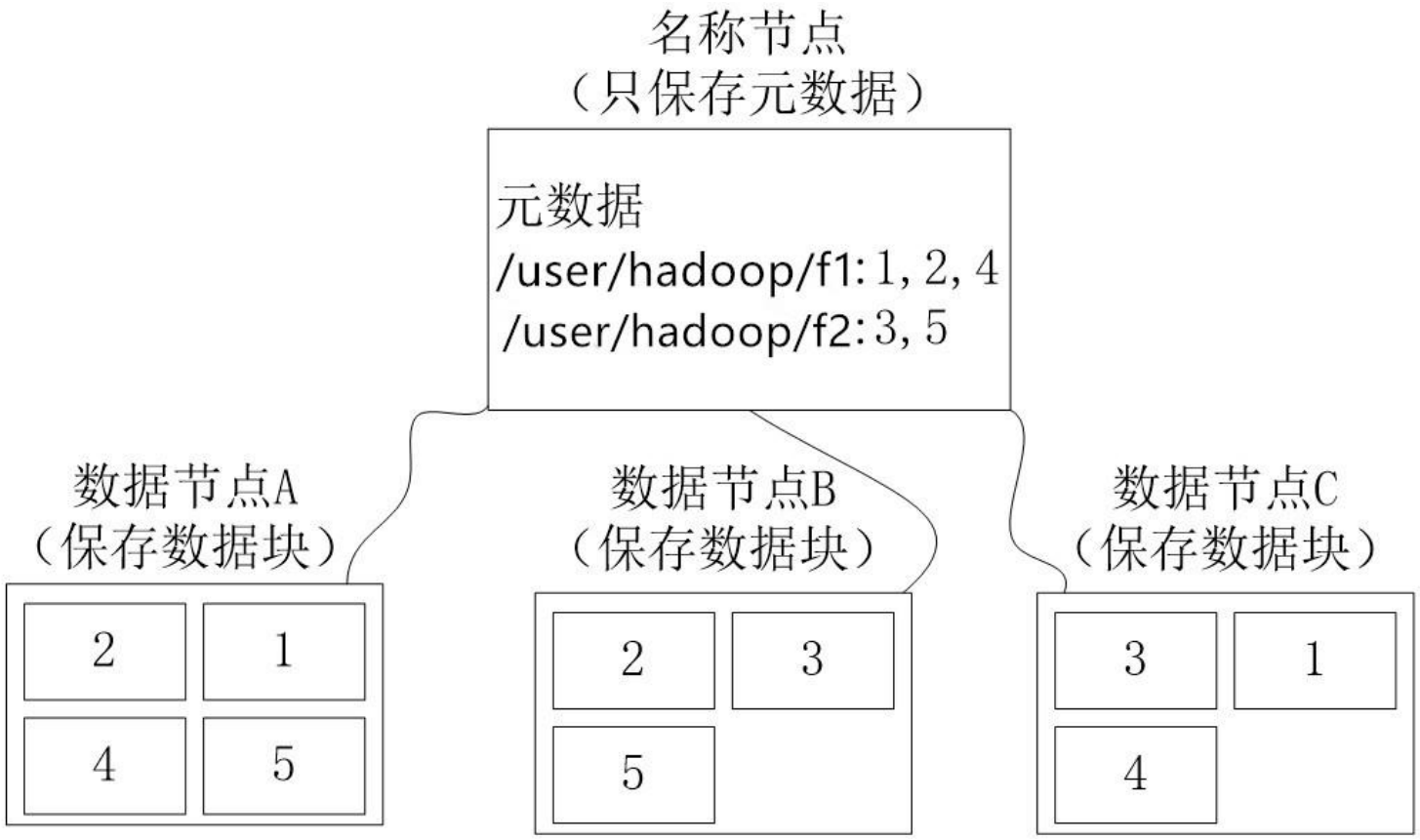
3.5.1 数据的冗余存储

3.5.2 数据存取策略

3.5.3 数据错误与恢复

3.5.1 数据的冗余存储

- 作为一个分布式文件系统，为了保证系统的容错性和可用性，HDFS采用了多副本方式对数据进行冗余存储（副本机制），通常**一个数据块的多个副本会被分布到不同的数据节点上**



- 如图所示：数据块1被分别存放到数据节点A和C上，数据块2被存放在数据节点A和B上，……

3.5.1 数据的冗余存储

- 这种多副本方式具有以下几个优点：

（1）加快数据传输速度：当多个客户端需要同时访问一个文件时，可以让各个客户端分别从不同的数据块副本中读取数据，大大加快了数据传输速度。

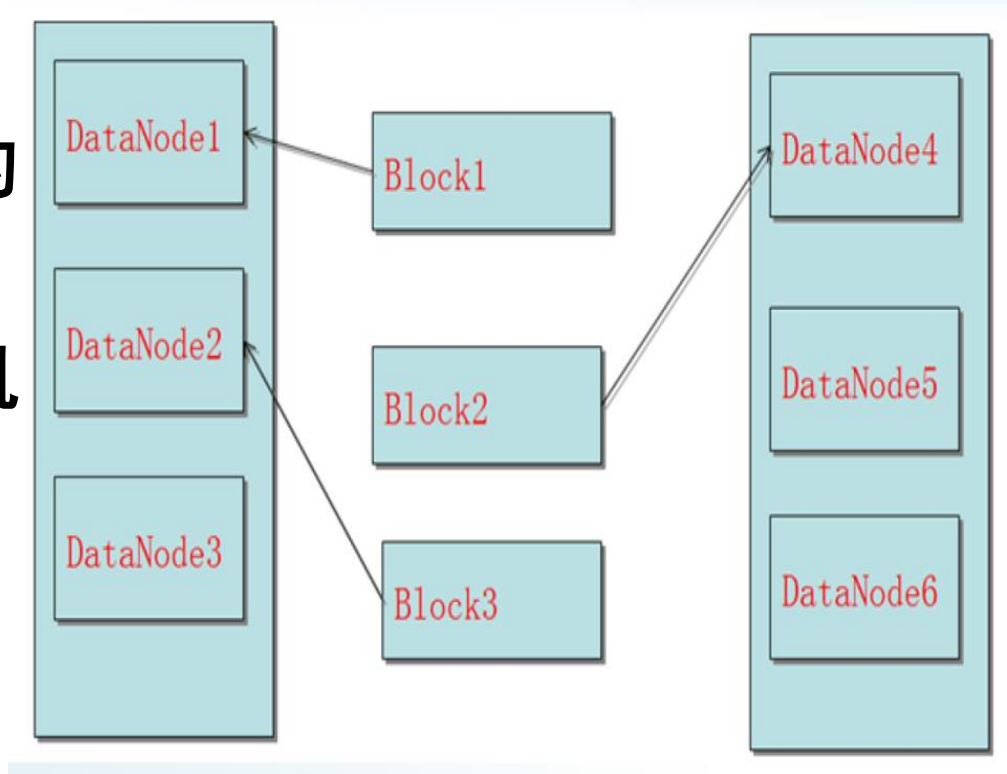
（2）容易检查数据错误：HDFS的数据节点之间通过网络传输数据，采用多个副本可以很容易判断数据传输是否出错。

（3）保证数据的可靠性：即使某个数据节点出现故障失效，也不会造成数据丢失。

3.5.2 数据存取策略

1. 数据存放

- HDFS默认的数据块副本数是3，是在配置文件hdfs-site.xml中设置的。
- 第一个副本：如果写操作请求来自集群内，则放置在发起写操作请求的数据节点上；如果是集群外提交，则随机挑选一台磁盘不太满、CPU不太忙的数据节点；
- 第二个副本：放置在与第一个副本不同的机架的数据节点上；
- 第三个副本：放置在与第一个副本相同机架的其他数据节点上；
- 更多副本：放置在集群中的随机节点



思考题

- **HDFS默认：数据副本不都放在同一的机架上，它的优缺点是什么？**

- 缺点：一个集群包含多个机架，不同机架之间的数据通信需要通过交换机或路由器，同一机架中不同机器之间的通信则不需要经过交换机和路由器，这样，同一机架中不同机器之间的通信要比不同机架之间机器的通信带宽大。写入数据的时候不能充分利用同一机架内部机器之间的带宽；
- 优点:（1）可以获得很高的数据可靠性；（2）读取数据的时候，可以在多个机架上并行读取数据；（3）可以更容易地实现系统内部负载均衡和错误处理。

3.5.2 数据存取策略

2. 数据读取

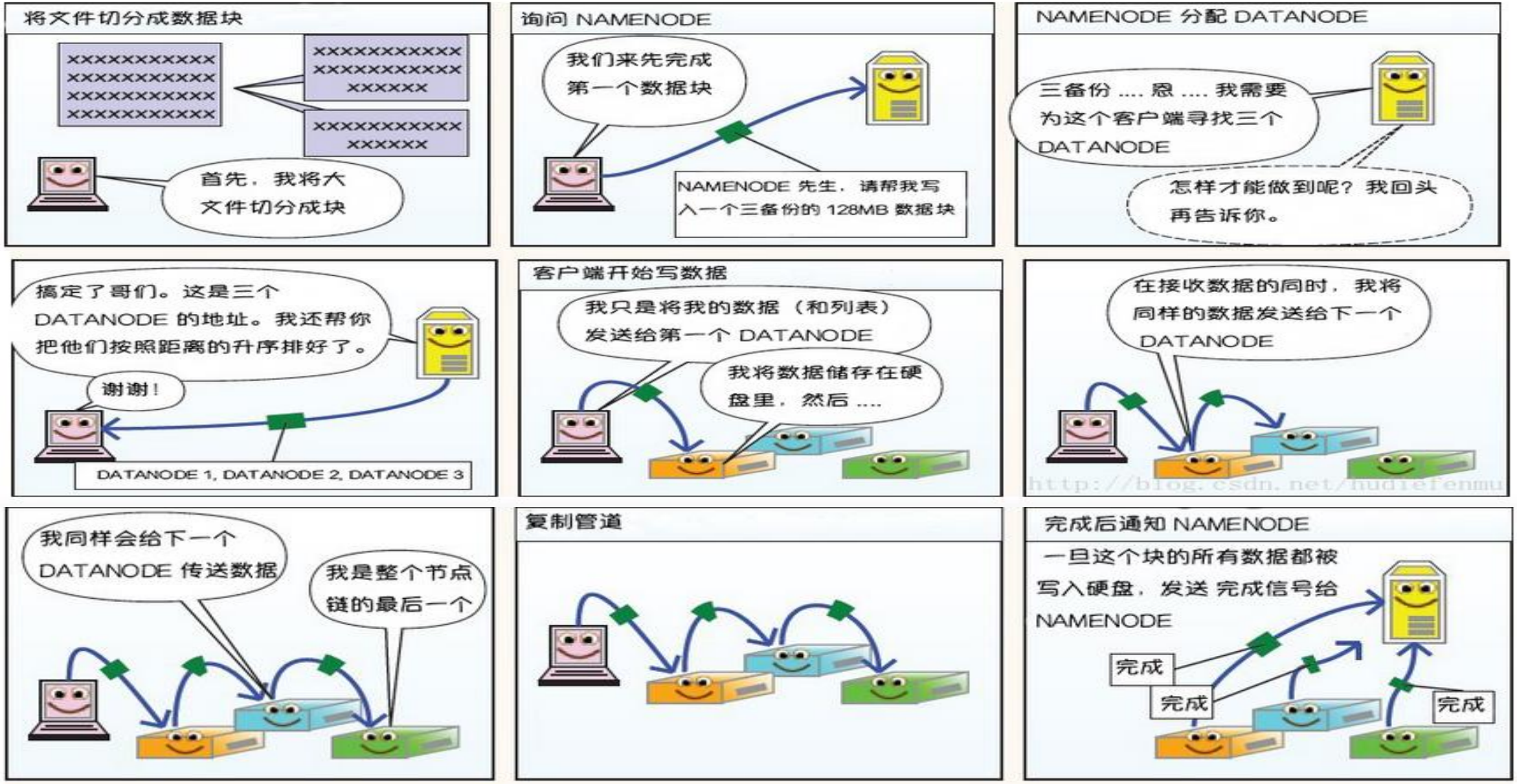
- HDFS提供了一个API可以确定一个数据节点所属的机架ID（称为“机架感知”），客户端也可以调用API获取自己所属的机架ID；
- 当客户端读取数据时，从NameNode获得数据块不同副本的存放位置列表，列表中包含了副本所在的DataNode，可以调用API来确定客户端和这些DataNode所属的机架ID，当发现某个数据块副本对应的机架ID和客户端对应的机架ID相同时，就优先选择该副本读取数据，如果没有发现，就随机选择一个副本读取数据。

3.5.2 数据存取策略

3. 数据复制(HDFS的数据复制采用了流水线复制的策略)

- 要复制的文件首先会被写入本地，并被切分成若干块。每个块都向HDFS集群中的名称节点发起写请求，名称节点会根据系统中各个数据节点的使用情况，选择一个**数据节点列表返回给客户端。**
- 客户端把数据首先写入列表中的第一个数据节点，同时把列表传给第一个数据节点。当第一个数据节点接收到4KB数据的时候，写入本地，并且向列表中的第二个数据节点发起连接请求，把自己已经接收到的4KB数据和列表传给第二个数据节点。
- 当第二个数据节点接收到4KB数据的时候，写入本地，并且向列表中的第三个数据节点发起连接请求，依次类推，列表中的多个数据节点形成一条数据复制的流水线。最后，当文件写完的时候，数据复制也同时完成。

3.5.2 数据存取策略


将文件切分成数据块
首先，我将大文件切分成块
询问 NAMENODE
我们来先完成第一个数据块
NAMENODE 先生，请帮我写入一个三备份的 128MB 数据块
NAMENODE 分配 DATANODE
三备份 .... 恩 .... 我需要为这个客户端寻找三个 DATANODE
怎样才能做到呢？我回头再告诉你。
搞定了哥们。这是三个 DATANODE 的地址。我还帮你把他们按照距离的升序排好了。
谢谢！
DATANODE 1, DATANODE 2, DATANODE 3
客户端开始写数据
我只是将我的数据（和列表）发送给第一个 DATANODE
我将数据储存在硬盘里，然后 ....
在接收数据的同时，我将同样的数据发送给下一个 DATANODE
我同样会给下一个 DATANODE 传送数据
我是整个节点链的最后一个
复制管道
完成后通知 NAMENODE
一旦这个块的所有数据都被写入硬盘，发送 完成信号给 NAMENODE
完成
完成
完成

3.5.3 数据错误与恢复

- HDFS具有较高的容错性，可以兼容廉价的硬件，它把硬件出错看作一种常态，而不是异常，并设计了相应的机制检测数据错误和进行自动恢复，主要包括：NameNode出错、DataNode出错和数据出错。

1. NameNode出错

- NameNode保存了所有的元数据信息，其中，最核心的两大数据结构是FsImage和EditLog，如果这两个文件发生损坏，那么整个HDFS实例将失效。
- 因此，HDFS设置了备份机制，把这些核心文件同步复制到备份服务器SecondaryNameNode上。当NameNode出错时，就可以根据备份服务器SecondaryNameNode中的FsImage和EditLog数据进行恢复。

3.5.3 数据错误与恢复

2. DataNode出错

- 每个DataNode会定期向NameNode发送“心跳”信息，向NameNode报告自己的状态,当DataNode发生故障，或者网络发生断网时，NameNode就无法收到来自一些DataNode的心跳信息，这时，这些DataNode就会被标记为“宕机”，节点上面的所有数据都会被标记为“不可读”，NameNode不会再给它们发送任何I/O请求。
- 这时，有可能出现一种情形，即由于一些DataNode的不可用，会导致一些数据块的副本数量小于冗余因子,NameNode会定期检查这种情况，一旦发现某个数据块的副本数量小于冗余因子，就会启动数据冗余复制，为它生成新的副本。

3.5.3数据错误与恢复

3. 数据出错

- 网络传输和磁盘错误等因素，都会造成数据错误。
- 客户端在读取到数据后，会采用md5和sha1对数据块进行校验，以确定读取到正确的数据。
- 在文件被创建时，客户端就会对每一个数据块进行信息摘录，并把这些信息写入到同一个路径的隐藏文件里面，当客户端读取文件的时候，会先读取该信息文件，然后，利用该信息文件对每个读取的数据块进行校验，如果校验出错，客户端就会请求到另外一个DataNode读取该数据块，并且向NameNode报告这个数据块有错误，NameNode会定期检查并且重新复制这个块。

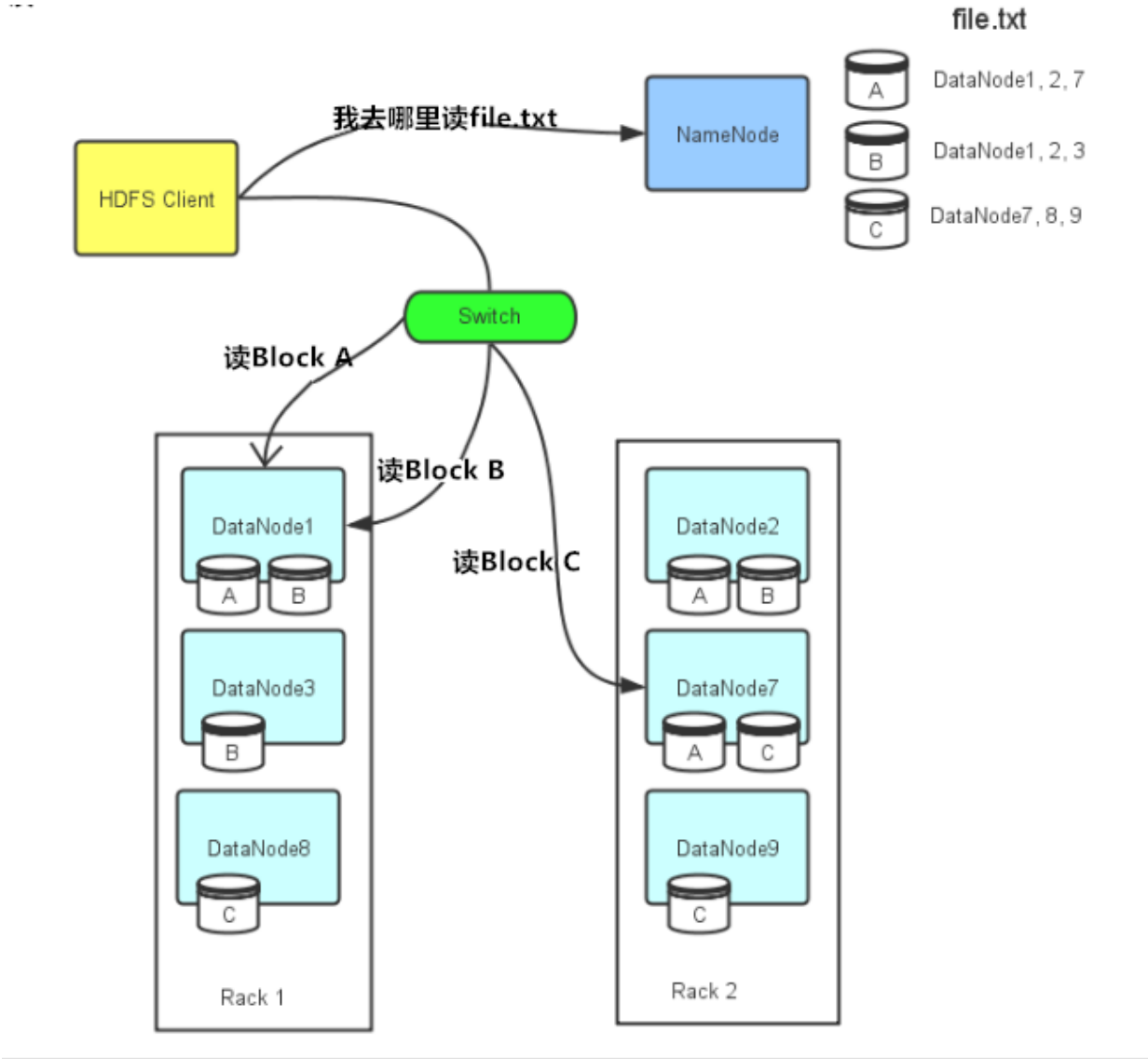
3.6 HDFS数据读写过程

- 3.6.1　读数据的过程
- 3.6.2　写数据的过程

3.6.1 读数据的过程

□读数据流程(hdfs dfs –get file.txt .)

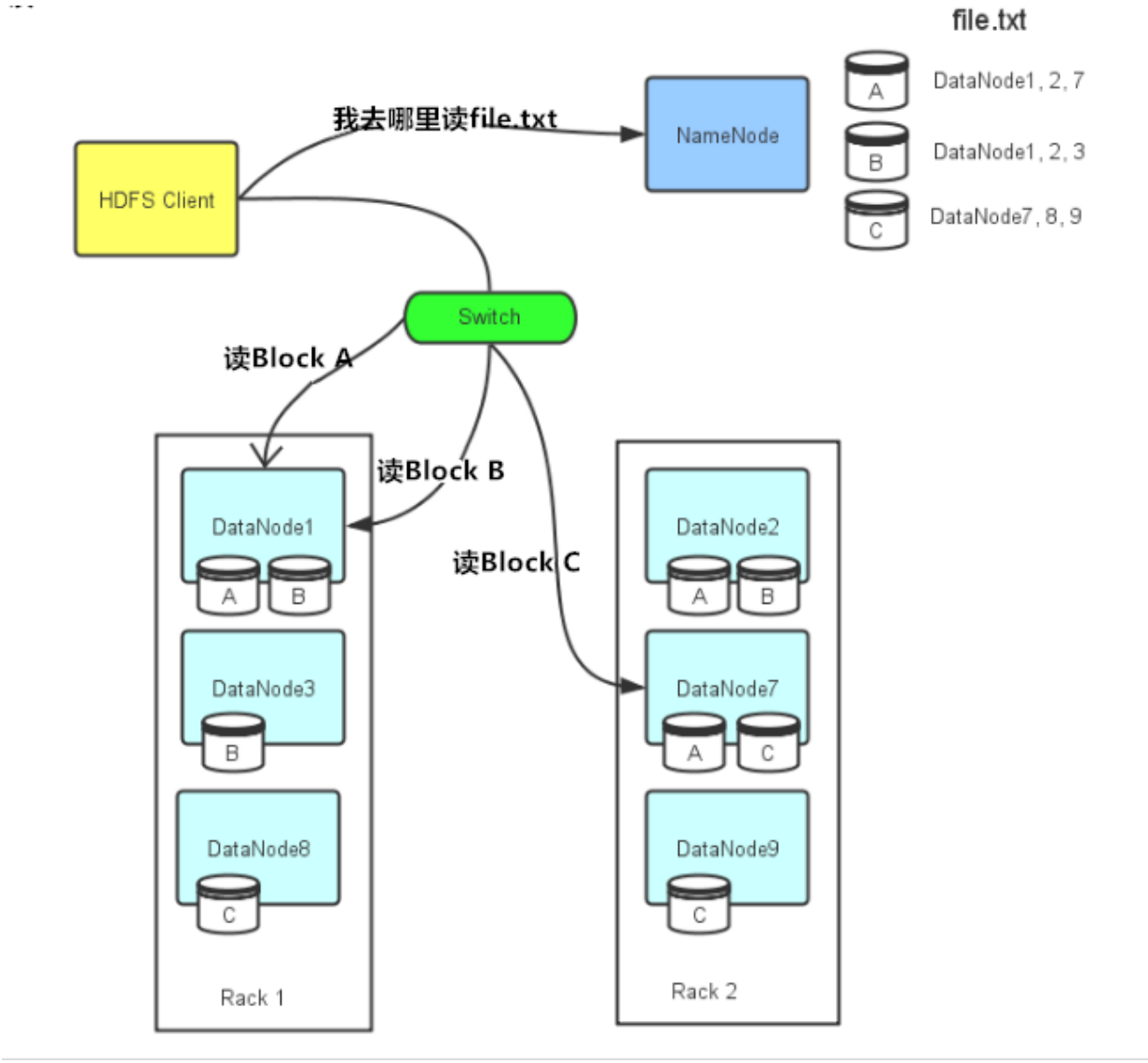
（1）首先，HDFS Client会先去联系NameNode，询问file.txt总共分为几个Block，这些Block分别存放在哪些DataNode上。由于每个Block都会存在几个副本，所以NameNode会把file.txt文件组成的Block对应的所有DataNode列表都返回给HDFS Client。



3.6.1 读数据的过程

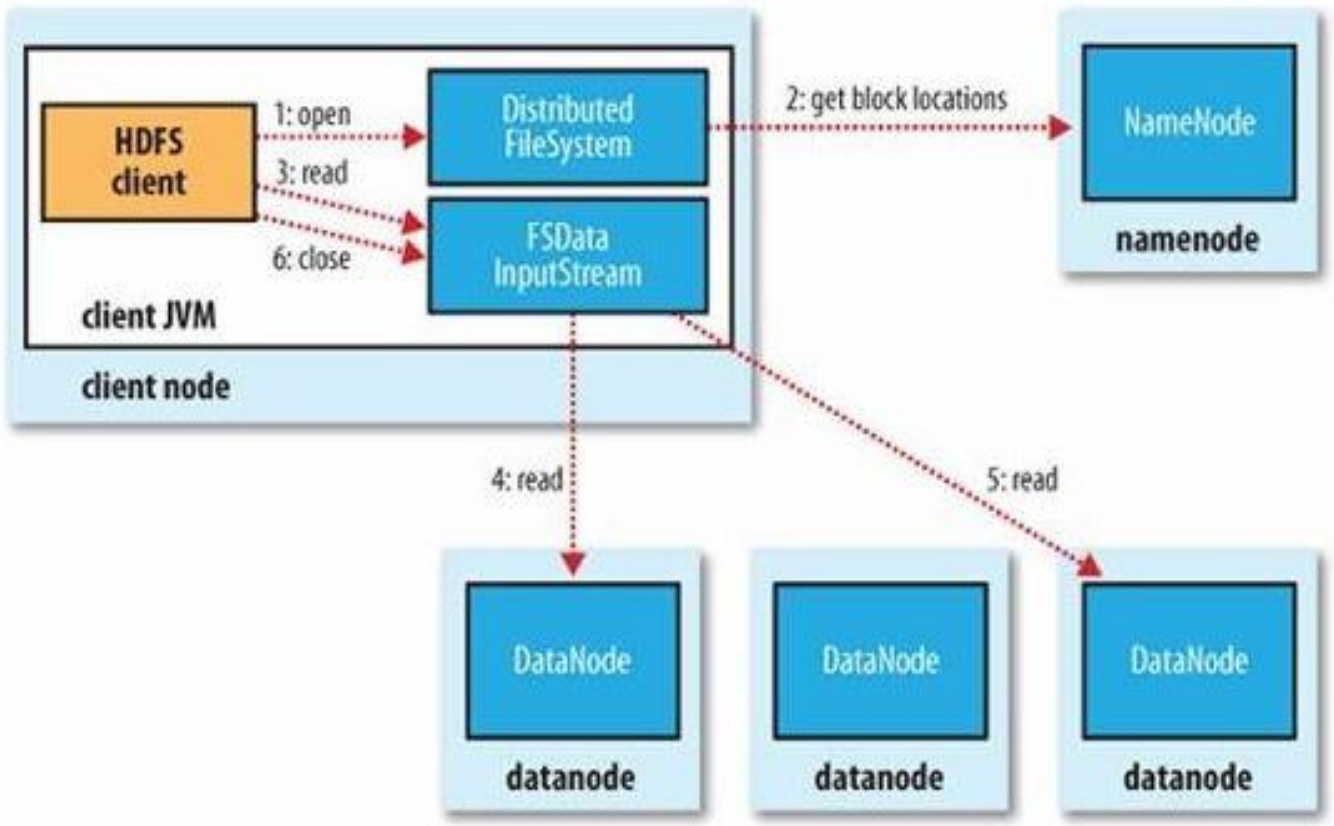
（2）第一个DataNode去读取对应的Block,比如Block A存储在DataNode 1,2,7,那么HDFS Client会到DataNode1去读取Block A，Block C存储在DataNode7,8,9那么HDFS Client就会到DataNode7去读取Block C.

（3）将读取的三个Block合并成一个完整的文件



3.6.1 读数据的过程

□HDFS数据读取代码解析：

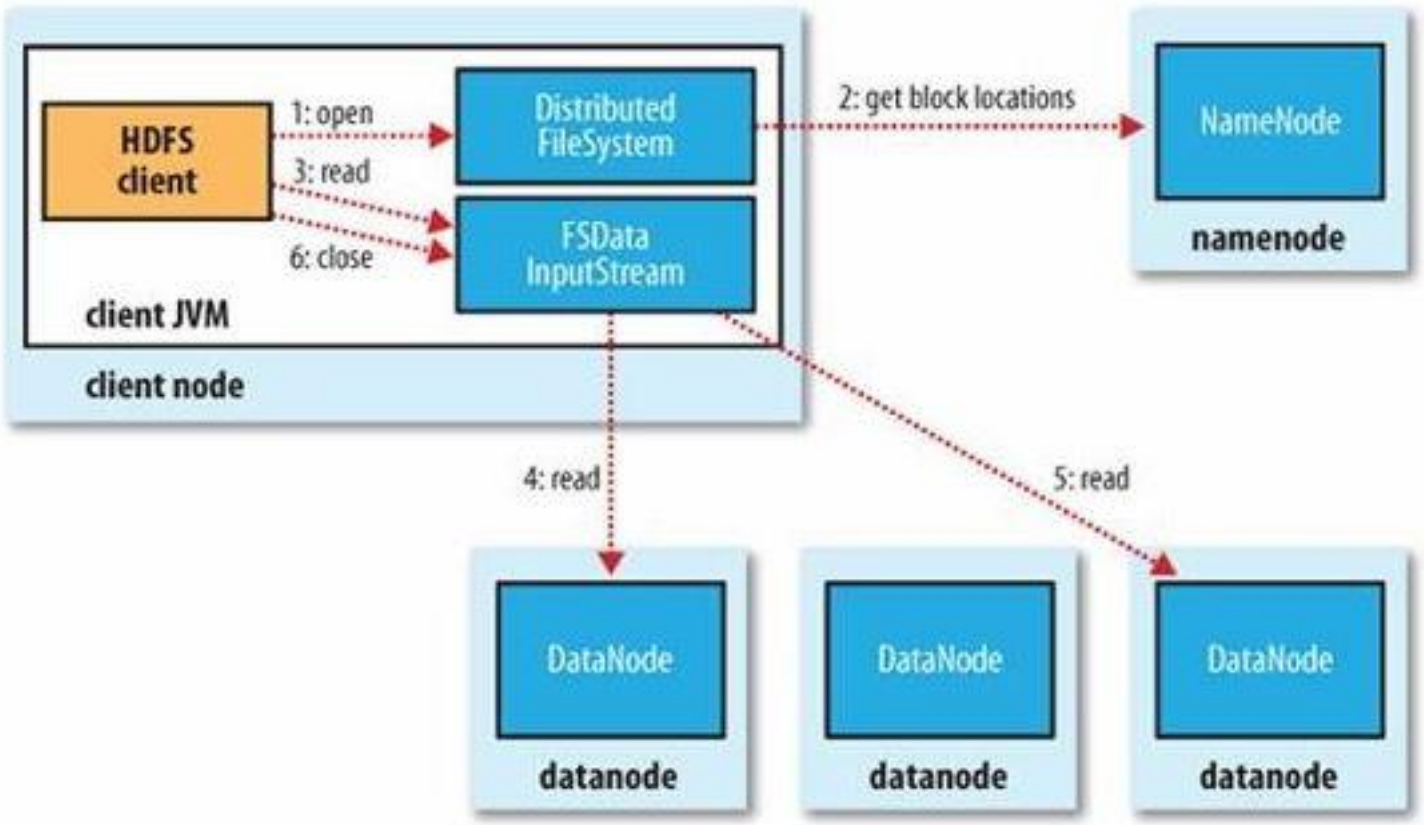


1.客户端首先调用FileSystem对象的open方法，其实获取的是一个DistributedFileSystem的实例。

- FileSystem：通用文件系统的抽象基类，Hadoop为FileSystem这个抽象类提供了许多具体的实现；
- DistributedFileSystem：FileSystem在HDFS文件系统中的实现。

3.6.1 读数据的过程

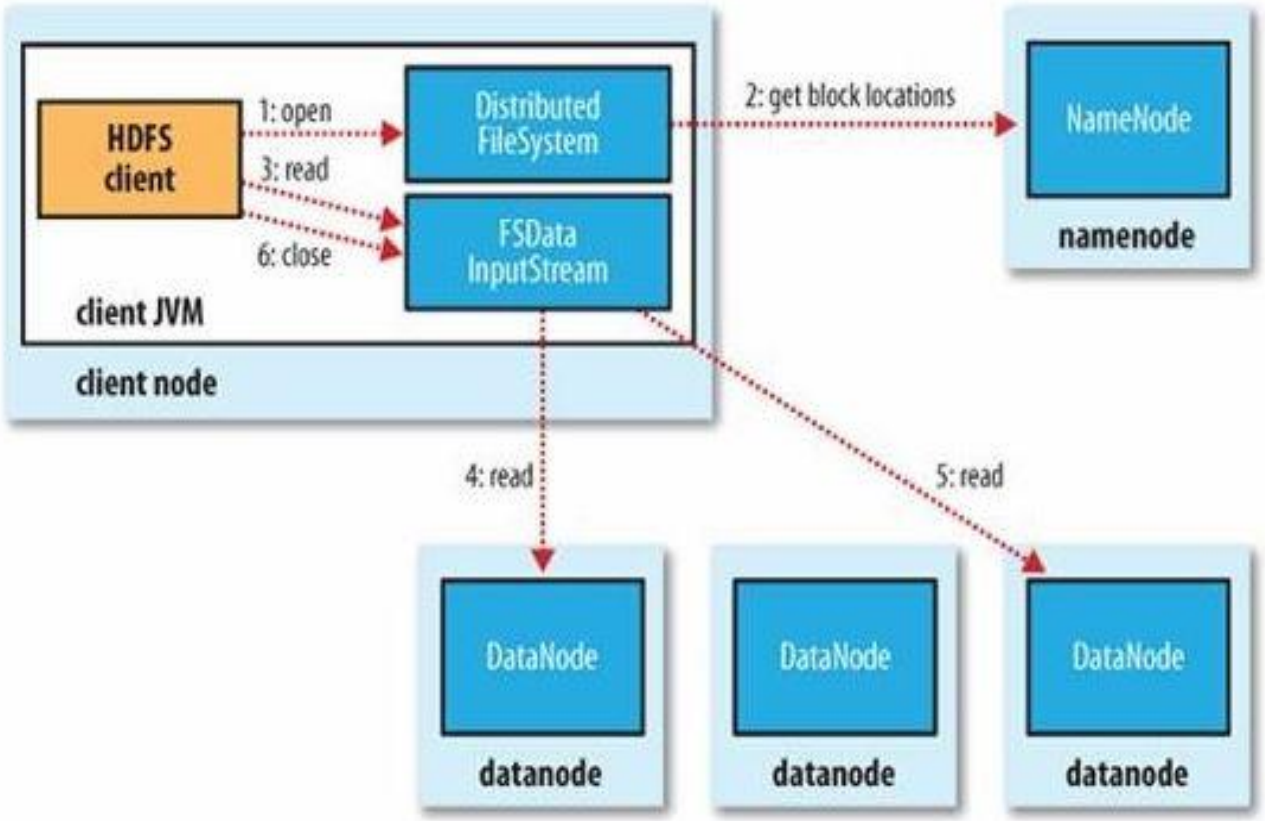
□HDFS数据读取代码解析：



2.DistributedFileSystem通过RPC(远程过程调用)获得文件的第一批block的locations，同一 block按照重复数会返回多个locations，这些locations按照hadoop拓扑结构排序，距离客户端近的排在前面。

3.6.1 读数据的过程

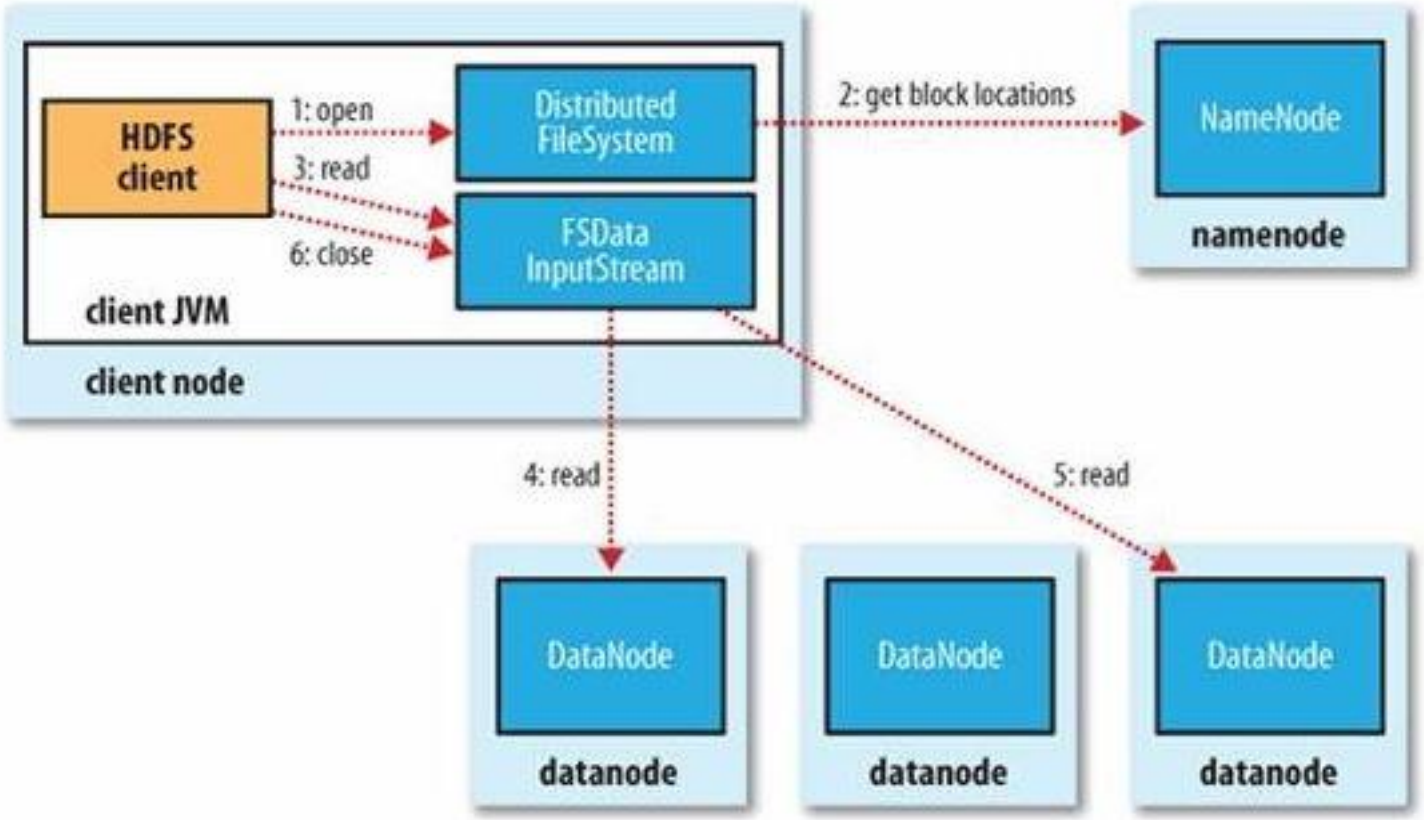
□HDFS数据读取代码解析：



（3） 前两步会返回一个FSDataInputStream对象，该对象会被封装成 DFSInputStream对象，DFSInputStream可以方便的管理datanode和namenode数据流。客户端调用read方法，DFSInputStream就会找出离客户端最近的datanode并连接datanode。

3.6.1 读数据的过程

□HDFS数据读取代码解析：

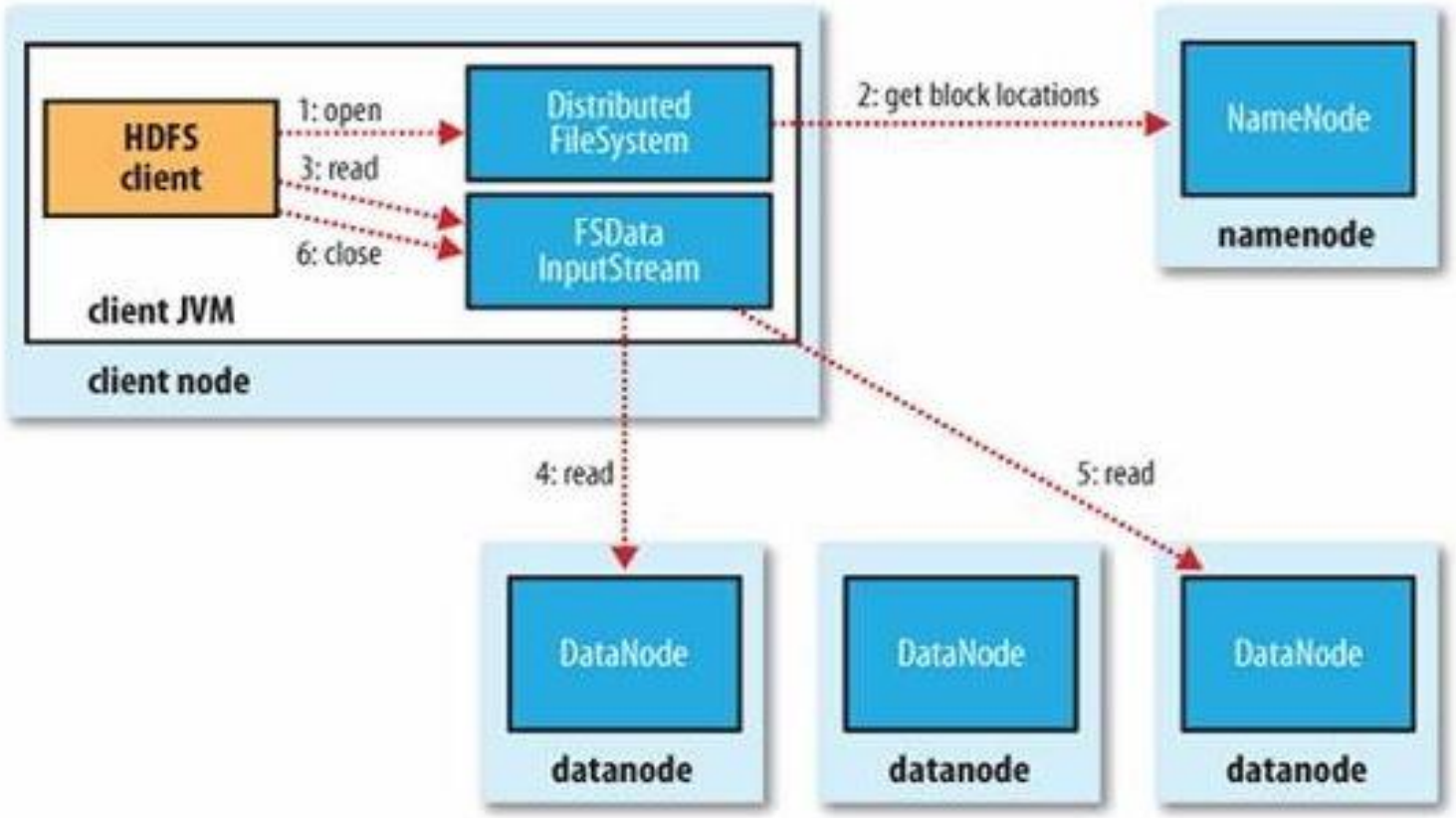


（4） 数据从datanode源源不断的流向客户端。

（5）如果第一个block块的数据读完了，就会关闭指向第一个block块的datanode连接，接着读取下一个block块。这些操作对客户端来说是透明的，从客户端的角度来看只是读一个持续不断的流。

3.6.1 读数据的过程

□HDFS数据读取代码解析：



（6）如果第一批block都读完了，FSInputStream就会去namenode拿下一批blocks的location，然后继续读，如果所有的block块都读完，这时就会关闭掉所有的流。

3.6.1 读数据的过程

□HDFS读数据JAVA代码：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class FileReader
{
    public static void main(String[] args)throws IOException
    {
```

3.6.1 读数据的过程

```
//加载配置信息
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS","hdfs://hacluster");
//指定读取的路径
        Path path = new Path("/tenant/user06/a.txt");
//获取文件系统
        FileSystem fs  = FileSystem.get(conf);
 //打开指定的路径
      FSDataInputStream input =  fs.create (path);
//封装为缓冲文本读取
        BufferedReader reader = new BufferedReader(new
InputStreamReader(input));
```

3.6.1 读数据的过程

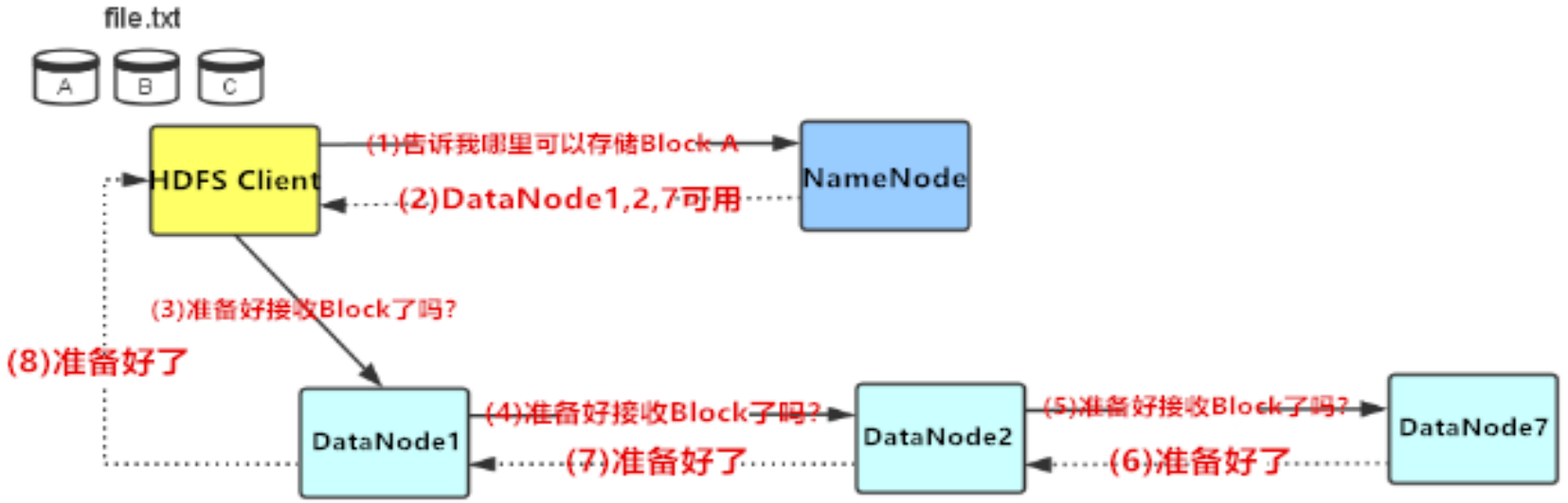
```
//读取一行记录
        String str =  reader.readLine();
//如果记录不为空
        while(str!=null)
        {
//打印输出
            System.out.println(str);
//读取一行记录
            str =  reader.readLine();
        }
//关闭流
        input.close();
        reader.close();
```

3.6.1 读数据的过程

```
//以FSDataInputStream流的方式读取
//      byte[] b = new byte[10];
// 读取到字符数组中
//      input.read(b);
//循环字节数组
//      for(int i = 0;i<b.length;i++)
//      {
//打印数组的每个字符
//          System.out.println((char)b[i]);
//      }
}
```

3.6.2 写数据的过程

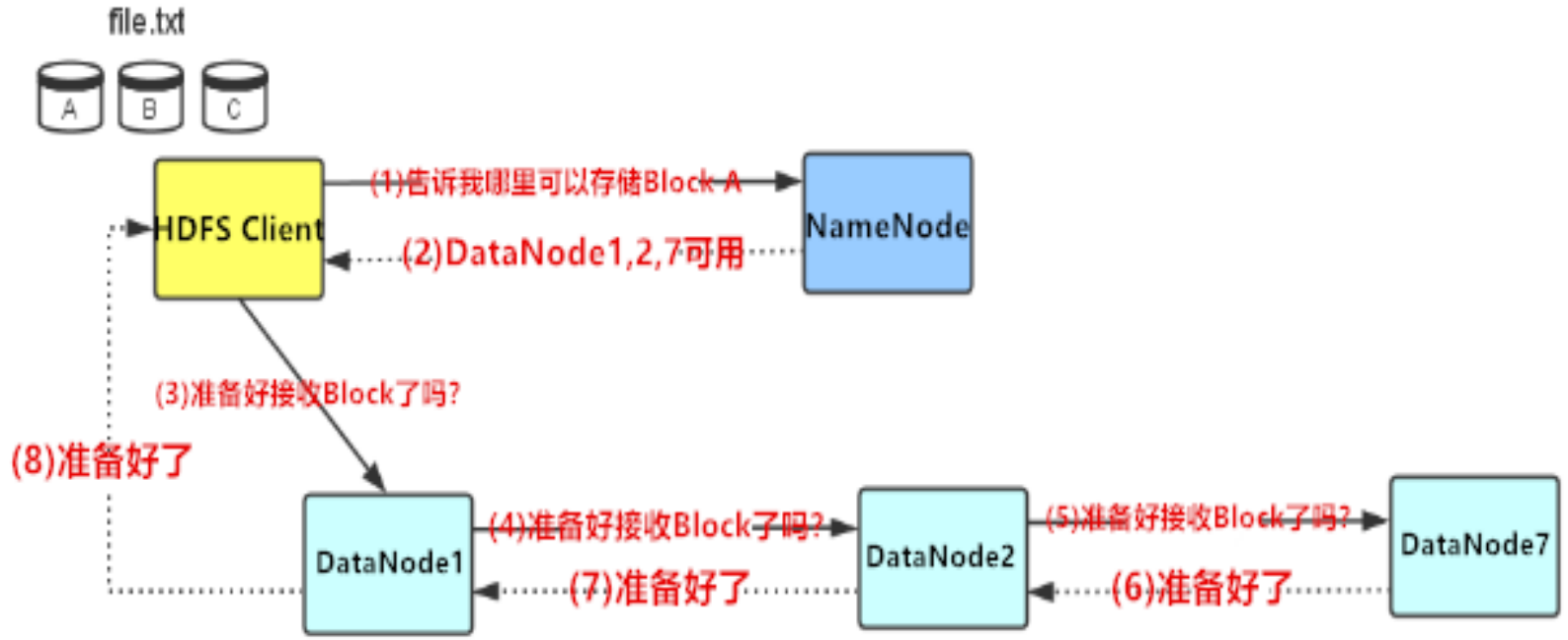
写数据之前



（1）首先，HDFS Client会去询问NameNode，看哪些DataNode可以存储Block A，file.txt文件的拆分是在HDFS Client中完成的，拆分成了3个Block(A B C)。因为NameNode存储着整个文件系统的元数据，它知道哪个DataNode上有空间可以存储这个Block A。

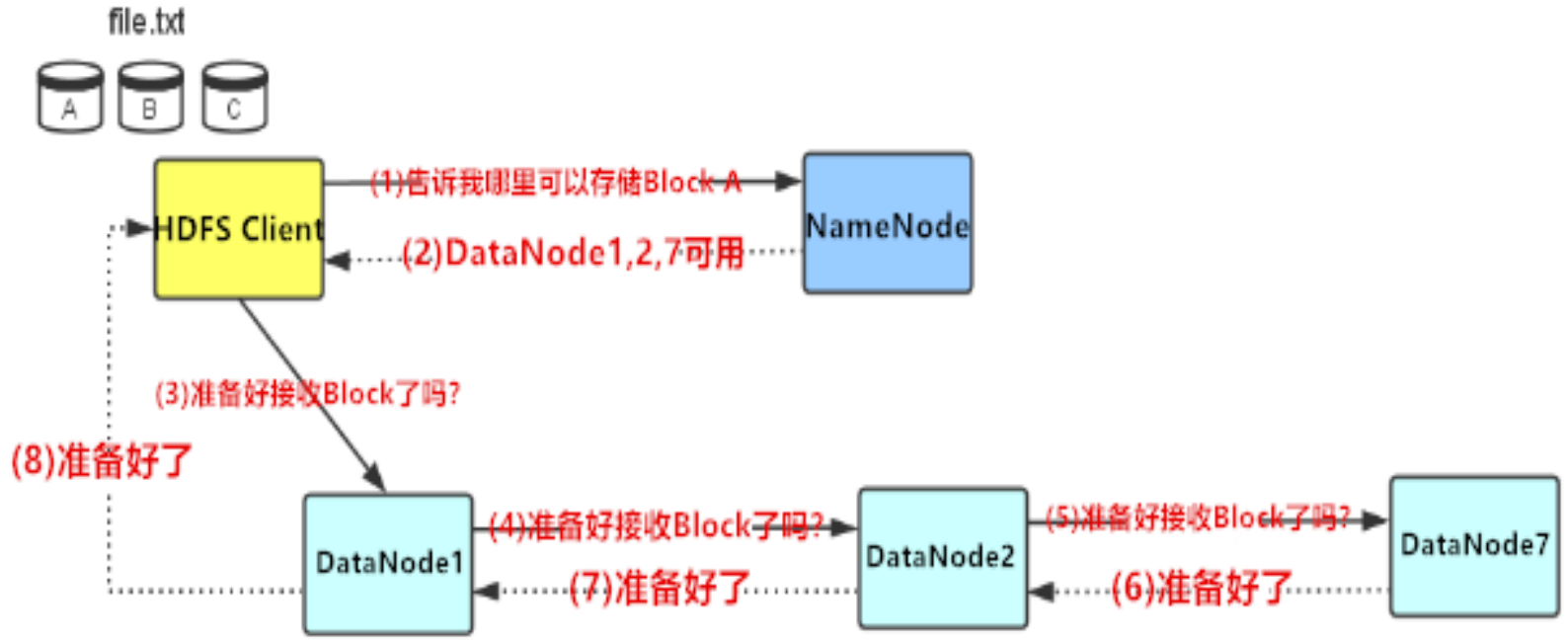
（2）NameNode通过查看它的元数据信息，发现DataNode1、2、7上有空间可以存储Block A,于是将此信息高速发送给HDFS Client。

3.6.2 写数据的过程



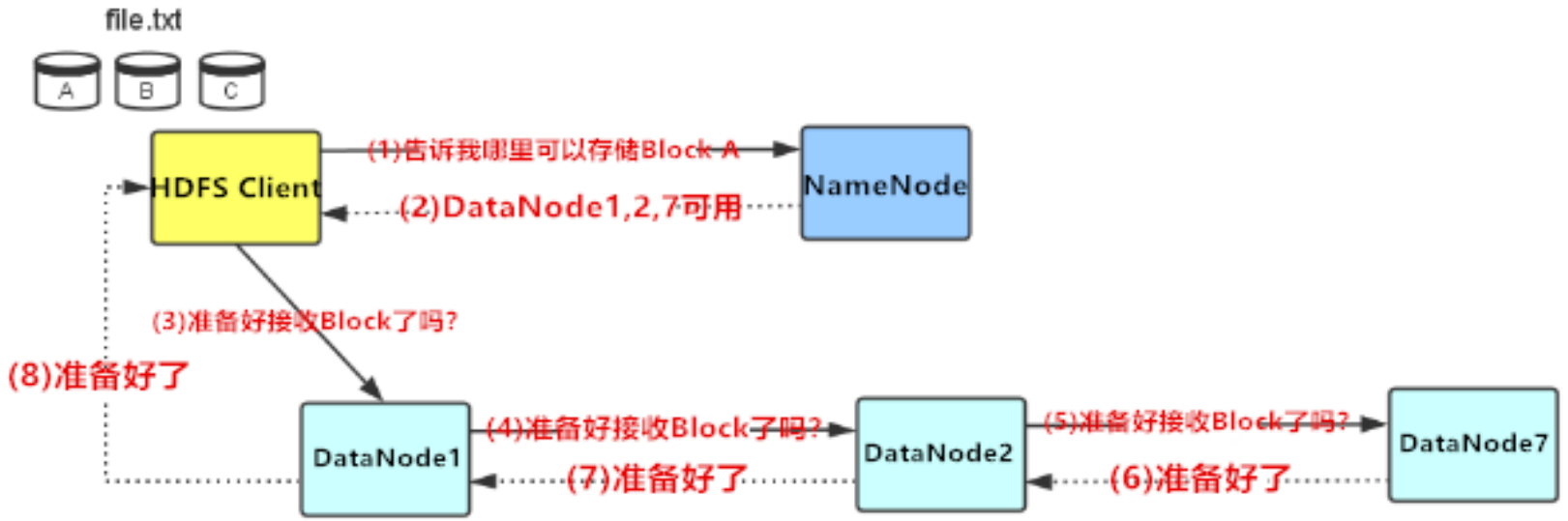
（3）HDFS Client接到NameNode返回的DataNode列表信息后，它会直接联系第一个DataNode-DataNode 1,让它准备接收Block A--实际上就是建立彼此之间的TCP连接。然后将Block A和NameNode返回的所有关于DataNode的元数据一并传给DataNode1。

3.6.2 写数据的过程



（4）在DataNode1与HDFS Client建立好TCP连接后，它会把HDFS Client要写Block A的请求顺序传给DataNode2（在与HDFS Client建立好TCP连接后从HDFS Client获得的DataNode信息），要求DataNode2也准备好接收Block A（建立DataNode2到DataNode1的TCP连接）。

3.6.2 写数据的过程



（5）同上，建立DataNode2到DataNode7的TCP连接

（6）当DataNode7准备好之后，它会通知DataNode2，表示可以开始接收Block A

（7）同理，当DataNode2准备好之后，它会通知DataNode1,表明可以开始接收Block A

（8）当HDFS Client接收到DataNode1的成功反馈信息后，说明这3个DataNode都已经准备好了，HDFS Client就会开始往这三个DataNode写入Block 。

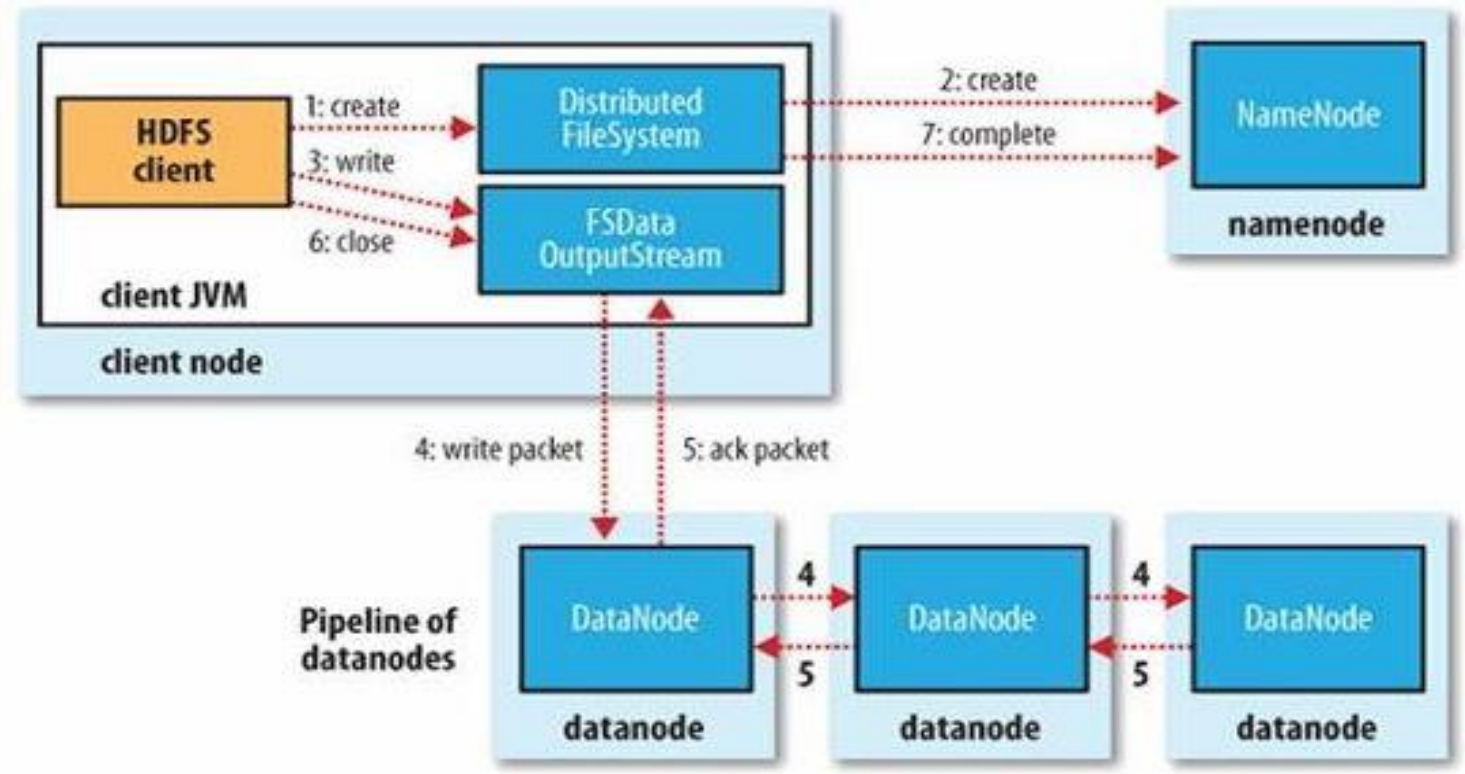
3.6.2 写数据的过程

写数据流程（hdfs dfs –put file.txt ./）

- 在DataNode1,2,7都准备好接收数据后，HDFS Client开始往DataNode1写入Block A数据。
- 同准备工作一样，当DataNode1接受完Block A数据后，它会顺序将Block A数据传输给DataNode2，然后DataNode2再传输给DataNode7。
- 每个DataNode在接受完Block A 数据后，会向发送者发送确认包，告诉它Block A数据已经接收完毕，当Block A成功写入3个DataNode之后，DataNode1会发送一个成功消息给HDFS Client，同时HDFS Client也会发一个Block A成功写入的消息给NameNode，NameNode会根据它接收到的消息更新它保存的文件系统元数据信息。
- 之后HDFS Client才能开始继续处理下一个Block：Block B。

3.6.2 写数据的过程

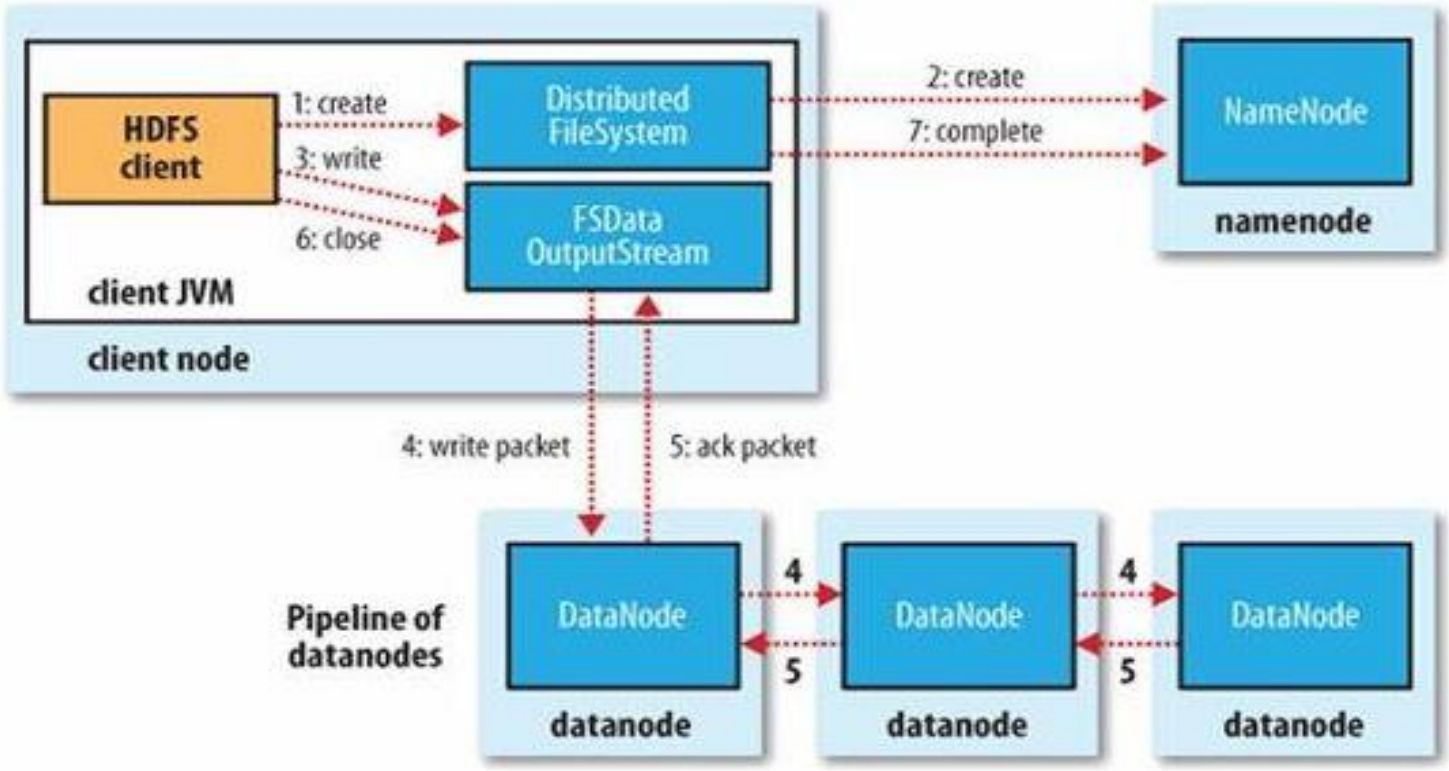
□HDFS写数据代码解析：



（1）客户端通过调用**DistributedFileSystem的create方法**，创建一个新的文件。

3.6.2 写数据的过程

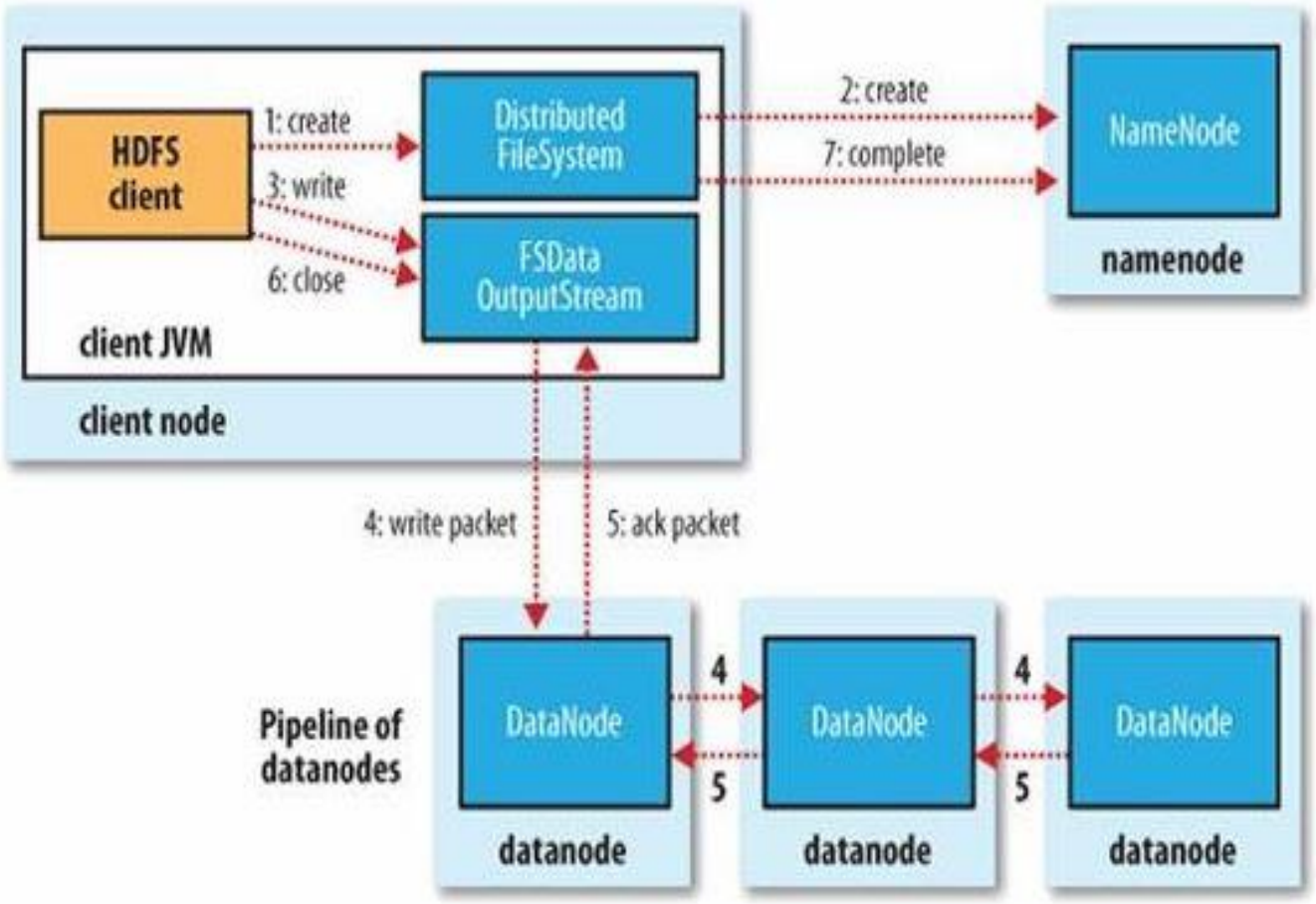
□HDFS写数据代码解析：



（2）DistributedFileSystem通过 RPC（远程过程调用）调用 NameNode，去创建一个没有blocks关联的新文件。创建前，NameNode 会做各种校验，比如文件是否存在，客户端有无权限去创建等。如果校验通过，NameNode 就会记录下新文件，否则就会抛出IO异常。

3.6.2 写数据的过程

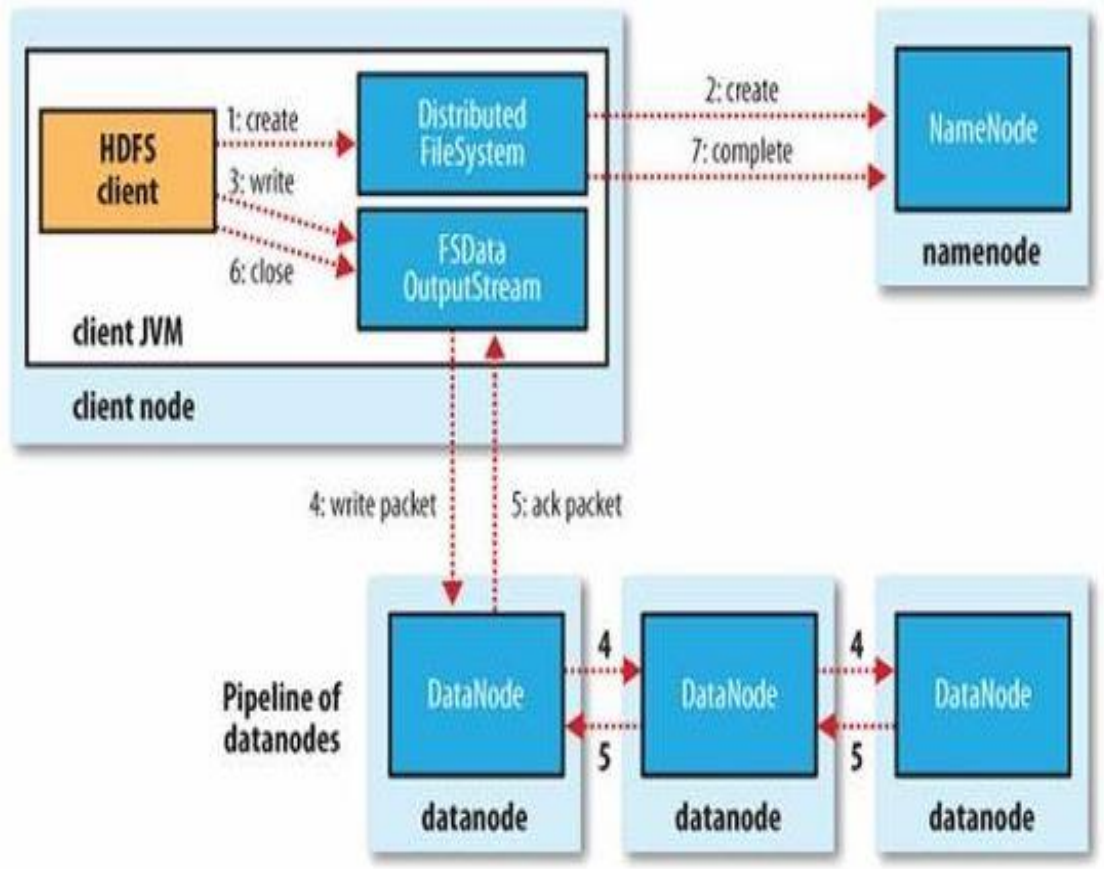
□HDFS写数据代码解析：



（3）前两步结束后会返回 FSDataOutputStream 的对象，和读文件的时候相似，FSDataOutputStream 被封装成 DFSOutputStream，DFSOutputStream 可以协调 NameNode和 DataNode。客户端开始写数据到 DFSOutputStream，DFSOutputStream会把数据切成一个个小packet，然后排成队列 data queue。

3.6.2 写数据的过程

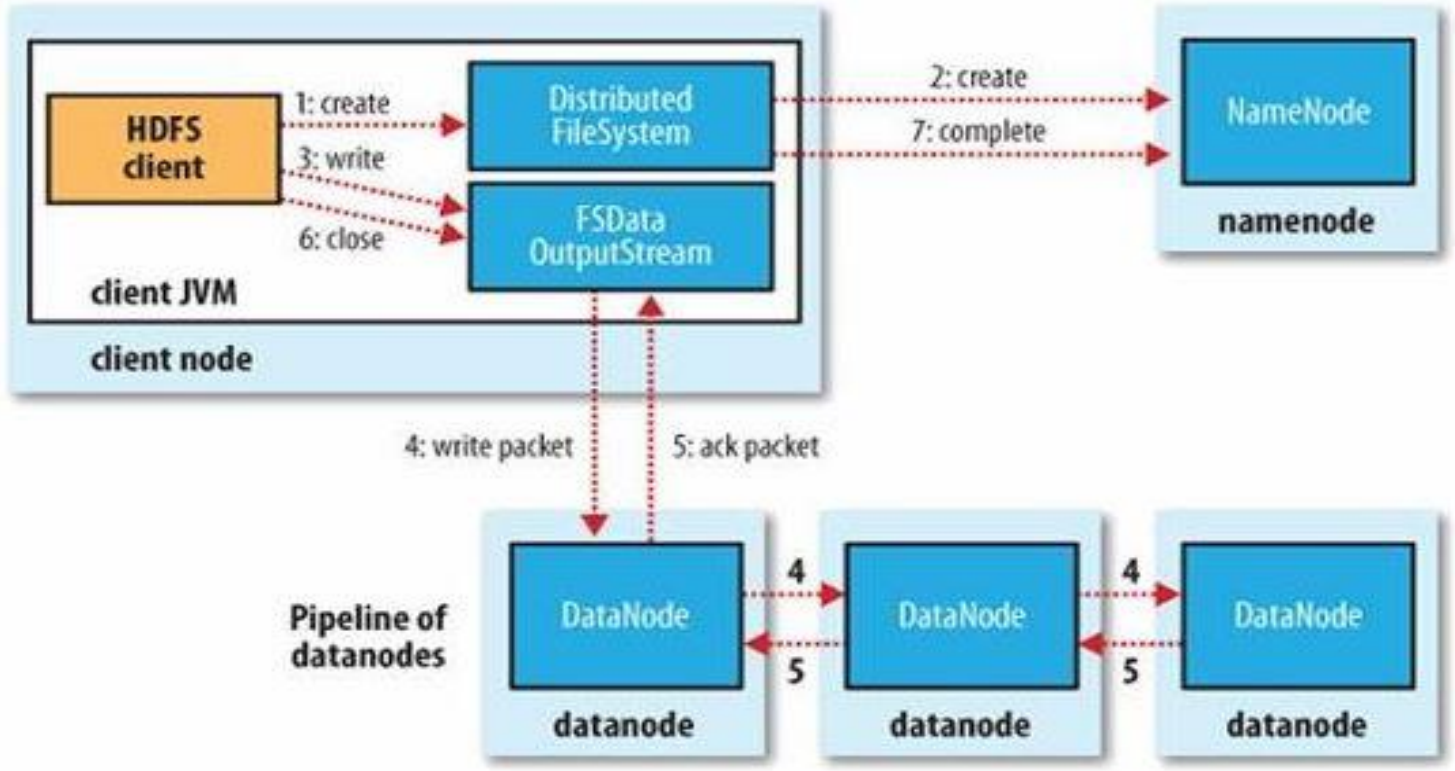
□HDFS写数据代码解析：



（4）DataStreamer 会去处理接受 data queue，它先问询 NameNode 这个新的 block 最适合存储的在哪几个 DataNode里，比如重复数是3，那么就找到3个最适合的 DataNode，把它们排成一个 pipeline。DataStreamer 把 packet 按队列输出到管道的第一个 DataNode 中，第一个 DataNode又把 packet 输出到第二个 DataNode 中，以此类推

3.6.2 写数据的过程

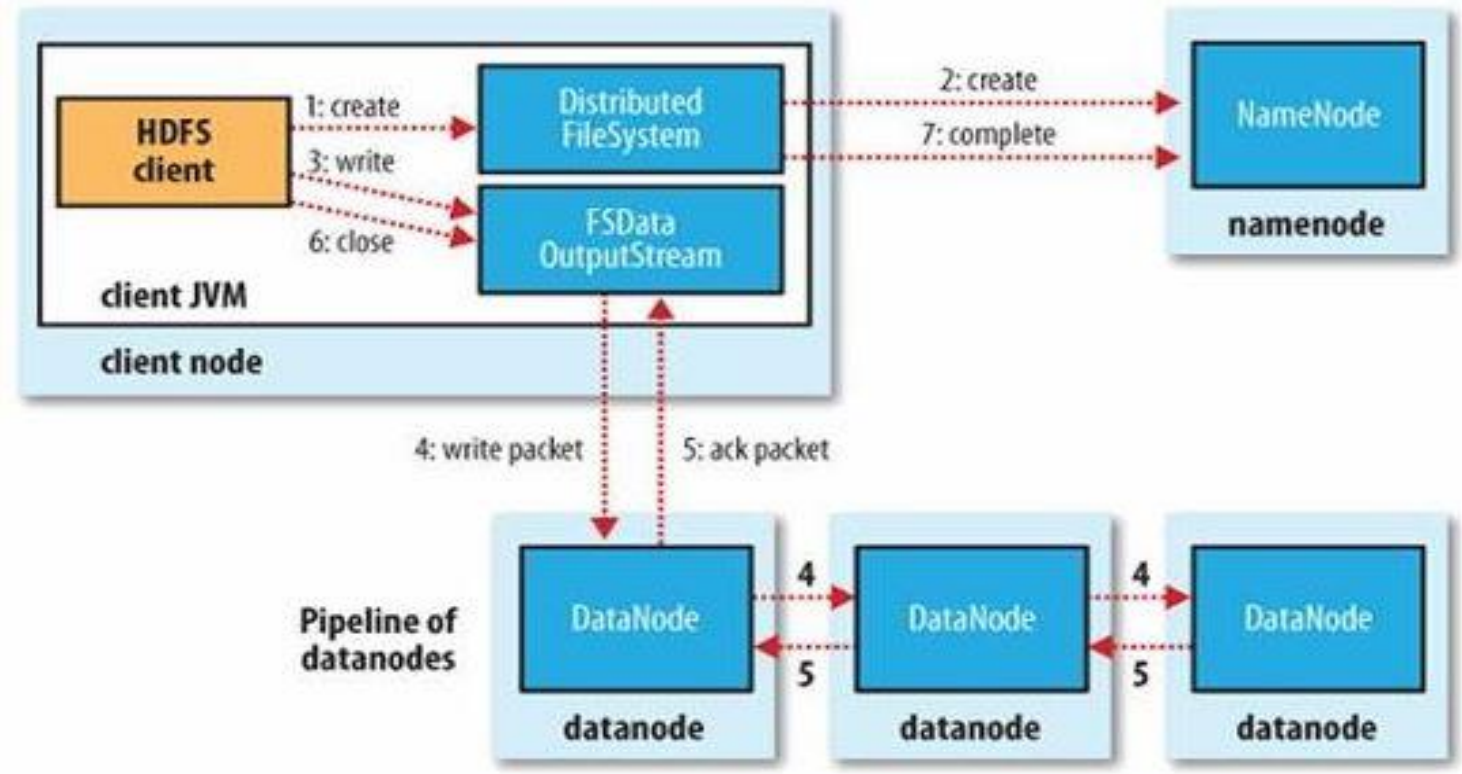
□HDFS写数据代码解析：



（5）DFSOutputStream 还有一个队列叫 ack queue，也是由 packet 组成，等待DataNode的收到响应，当pipeline中的所有DataNode都表示已经收到的时候，这时akc queue才会把对应的packet包移除掉。

3.6.2 写数据的过程

□HDFS写数据代码解析：



（6）客户端完成写数据后，调用close方法关闭写入流。

（7）DataStreamer 把剩余的包都刷到 pipeline 里，然后等待 ack 信息，收到最后一个 ack 后，通知 DataNode 把文件标示为已完成。

3.6.2 写数据的过程

□HDFS写数据JAVA代码：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
public class FileWriter
{
    public static void main(String[] args)throws IOException
{
```

3.6.2 写数据的过程

```
//加载配置信息
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://hacluster");
//指定写入的路径
        Path path = new Path("/tenant/user06/a.txt");
//获得文件系统
        FileSystem fs = FileSystem.get(conf);
//建立流连接
        //FSDataOutputStream output = fs.create(path);
    // 也可以在原来的文件上做追加
FSDataOutputStream output = fs.append(path);
```

3.6.2 写数据的过程

```
//封装为缓冲文本写入
        BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(output)); 30//写入字符串
        writer.write("\naaa\nbbbb \n ccc");
//写入一个空行
        // writer.newLine();
        // writer.write("hello world");
//刷新缓冲区，写入文件
        writer.flush();
// 关闭流
        output.close();
        writer.close();
        System.out.println("write successs");
    }
```

3.7 HDFS编程实践

3.7.1 HDFS常用命令

3.7.2 HDFS的Web界面

3.7.3 HDFS常用Java API及应用实例

参考：http://dblab.xmu.edu.cn/blog/2460-2/

本章小结

- 分布式文件系统是大数据时代解决大规模数据存储问题的有效解决方案，HDFS开源实现了GFS，可以利用由廉价硬件构成的计算机集群实现海量数据的分布式存储
- HDFS具有兼容廉价的硬件设备、流数据读写、大数据集、简单的文件模型、强大的跨平台兼容性等特点。但是，也要注意到，HDFS也有自身的局限性，比如不适合低延迟数据访问、无法高效存储大量小文件和不支持多用户写入及任意修改文件等
- 块是HDFS核心的概念，一个大的文件会被拆分成很多个块。HDFS采用抽象的块概念，具有支持大规模文件存储、简化系统设计、适合数据备份等优点

本章小结

- **HDFS采用了主从（MASTER/SLAVE）结构模型，一个HDFS集群包括一个名称节点和若干个数据节点。名称节点负责管理分布式文件系统的命名空间；数据节点是分布式文件系统HDFS的工作节点，负责数据的存储和读取**
- **HDFS采用了冗余数据存储，增强了数据可靠性，加快了数据传输速度。HDFS还采用了相应的数据存放、数据读取和数据复制策略，来提升系统整体读写响应性能。HDFS把硬件出错看作一种常态，设计了错误恢复机制**
- 本章最后介绍了**HDFS**的数据读写过程以及**HDFS**编程实践方面的相关知识

小练习

3、如何理解元数据？

元数据指：目录结构及文件分块位置信息，包括文件系统目录树信息（HDFS提供给客户端一个抽象目录树，访问形式：hdfs://NameNode的hostname:port，例如：hdfs://localhost:9000）

① 文件名、目录名

② 文件和目录的从属关系

③ 文件和目录的大小、创建和最后访问时间、权限

- 文件和数据块的对应关系：文件由哪些块组成
- 块的存放位置：机器名，数据块ID

思考题

2、三个DataNode，当有一个DataNode出现错误会怎样？

DataNode以数据块作为容错单位，通常一个数据块会备份到三个DataNode上，如果一个DataNode出错，则会去其他备份数据块的DataNode上读取，并且会把这个DataNode上的数据块再复制一份，以达到备份的效果。

3、HDFS中NameNode、DataNode和SecondaryNameNode的职责分别是什么？

NameNode：Master节点，只有一个，管理HDFS的名称空间和数据块映射信息；配置副本策略；处理客户端请求。

DataNode：Slave节点，存储实际的数据；执行数据块的读写；汇报存储信息给NameNode。

SecondaryNameNode：辅助NameNode，分担其工作量；定期合并FsImage和EditLog，推送给NameNode；紧急情况下，可辅助恢复NameNode，但SecondaryNameNode并非NameNode的热备。

小练习

4、Client端上传文件的过程是什么？

Client向NameNode发起文件写入的请求。NameNode根据文件大小和文件块配置情况，返回给Client它所管理部分DataNode的信息。Client将文件划分为多个Block，根据DataNode的地址信息，按顺序写入到每一个DataNode块中。

5、HDFS默认的Block Size是多少？

Hadoop1.X版本中是64MB；Hadoop2.X版本中是128MB

小练习

6、HDFS构架原则：

- **元数据与数据分离：文件本身的属性（即元数据）与文件所持有的数据分离；**
- **主/从架构：一个HDFS集群是由一个NameNode和一定数目的DataNode组成；**
- **一次写入多次读取：HDFS中的文件在任何时间只能有一个Writer。当文件被创建，接着写入数据，最后，一旦文件被关闭，就不能再修改；**
- **移动计算比移动数据更划算：数据运算，越靠近数据，执行运算的性能就越好，由于HDFS数据分布在不同机器上，要让网络的消耗最低，并提高系统的吞吐量，最佳方式是将运算的执行移到离它要处理的数据更近的地方，而不是移动数据。**