第2讲 HDFS

辜希武
IDC实验室
1694551702@qq.com

Distributed File System

- 数据量越来越多，在一个操作系统管辖的范围存不下了，那么就分配到更多的操作系统管理的磁盘中，但是不方便管理和维护，因此迫切需要一种系统来管理多台机器上的文件，这就是分布式文件管理系统 。
- 是一种允许文件通过网络在多台主机上分享的文件系统，可让多机器上的多用户分享文件和存储空间。
- 通透性。让实际上是通过网络来访问文件的动作，由程序与用户看来，就像是访问本地的磁盘一般。
- 容错。即使系统中有某些节点脱机，整体来说系统仍然可以持续运作而不会有数据损失。
- 分布式文件管理系统很多，hdfs只是其中一种。适用于一次写入多次查询的情况，不支持并发写情况。

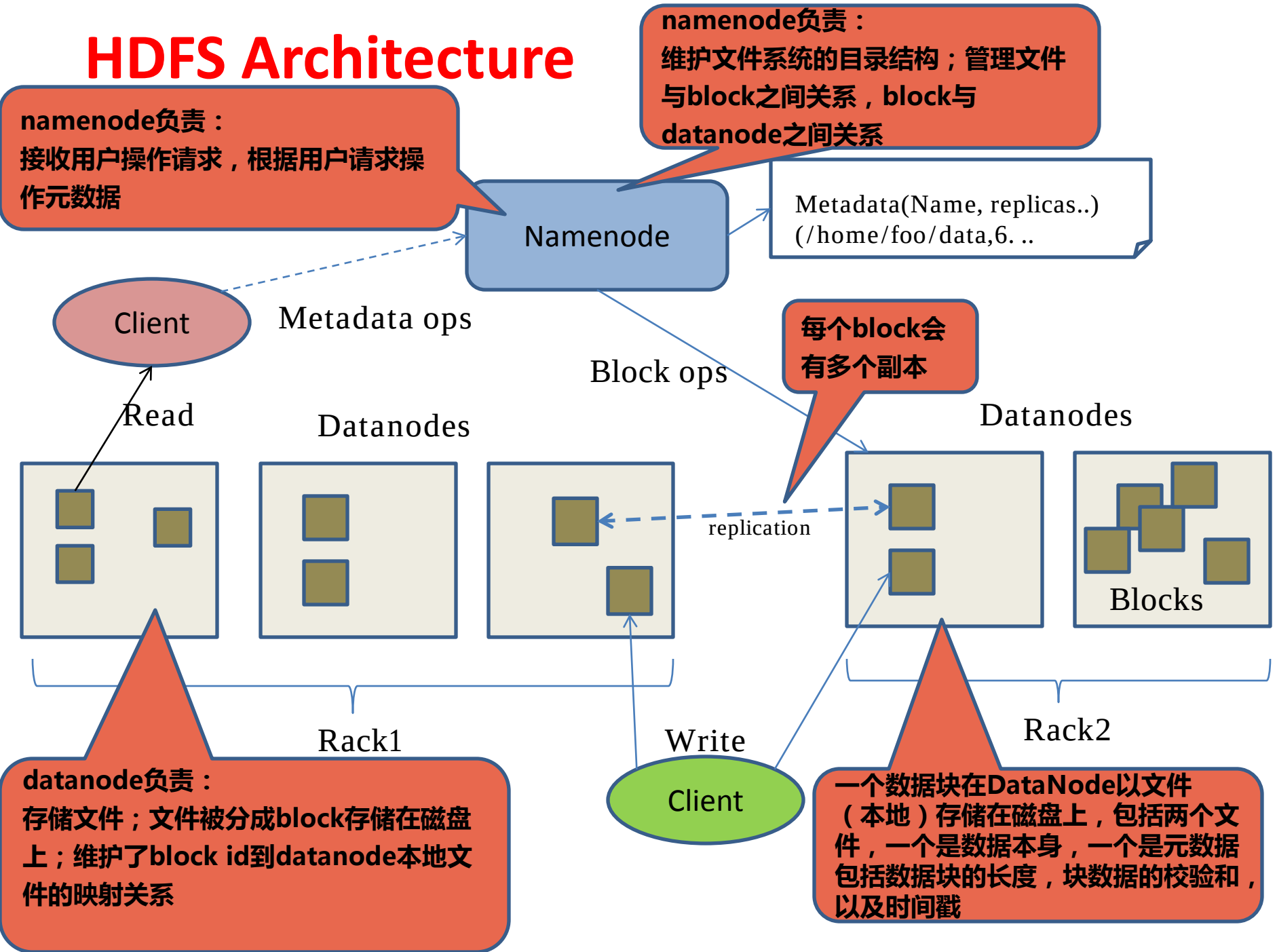
Hadoop文件系统

- 在Hadoop中，有一个文件系统的抽象，它提供了文件系统实现的各类接口，HDFS只是这个抽象文件系统的一个实例。
- 这个高层的文件系统抽象类是org.apache.hadoop.fs.FileSystem，这个抽象类有很多具体的文件系统的实现，一些主要的具体文件系统实现如下表所示。
- Hadoop提供了许多文件系统的接口，最重要的有：
  - Shell命令接口：提供了丰富的Shell命令
  - Java API：提供JAVA API编程接口

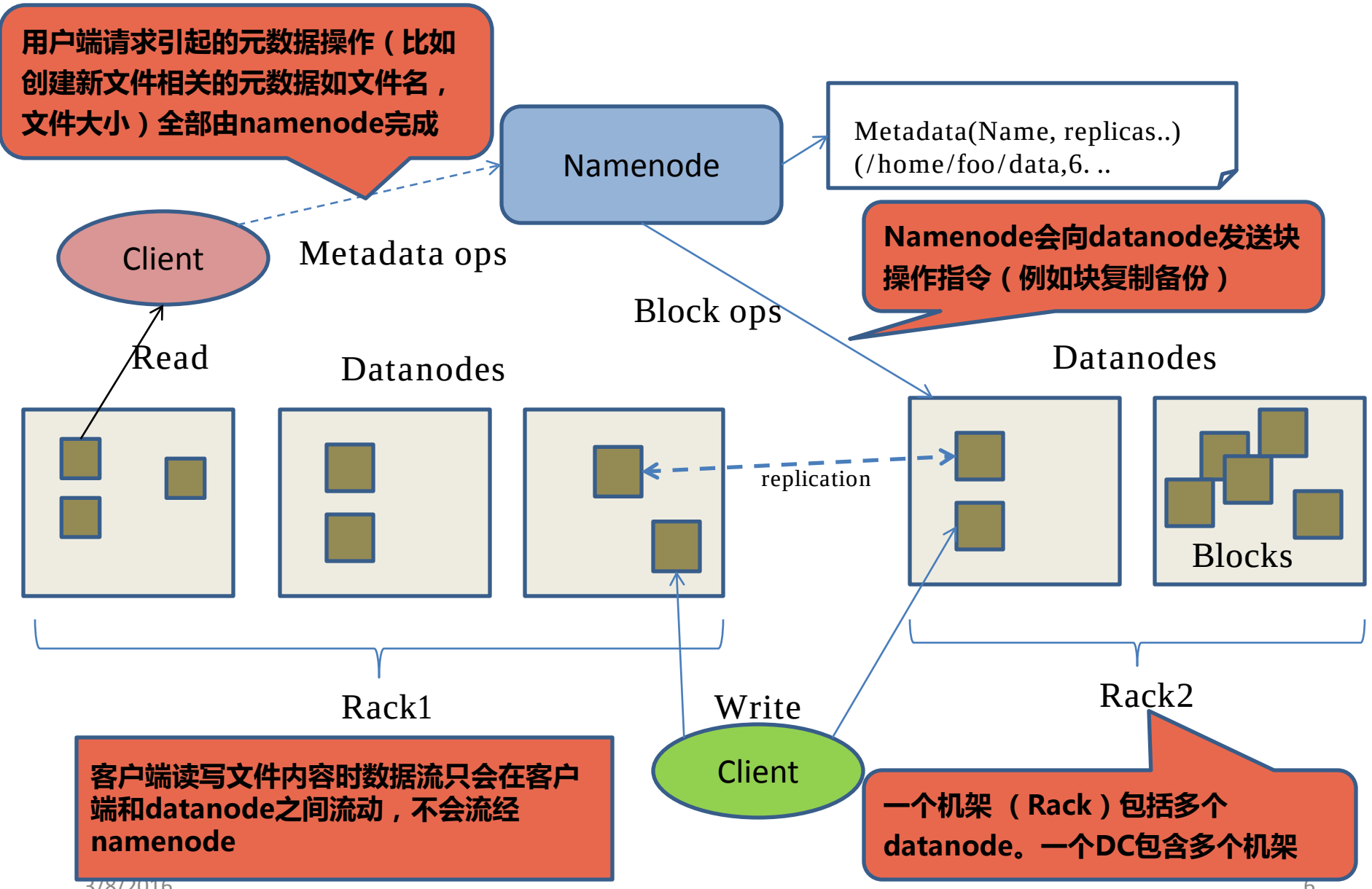
Hadoop文件系统

文件系统	URI方案	Java实现 (org.apache.hadoop)	定义
Local	file	fs.LocalFileSystem	支持有客户端校验和的本地文件系统。不带有校验和的本地文件系统在fs.RawLocalFileSystem中实现
HDFS	hdfs	hdfs.DistributedFileSystem	Hadoop的分布式文件系统
HFTP	hftp	hdfs.HftpFileSystem	支持通过HTTP方式以只读方式访问HDFS（和FTP没任何关系）
HSFTP	hsftp	hdfs.HsftpFileSystem	支持通过HTTPS方式以只读方式访问HDFS
HAR	har	fs.HarFileSystem	构建在其它文件系统之上，用于文件的归档，归档文件主要用于减少Namenode的内存使用
FTP	ftp	fs.ftp.FTPFileSystem	由FTP服务器支持的文件系统

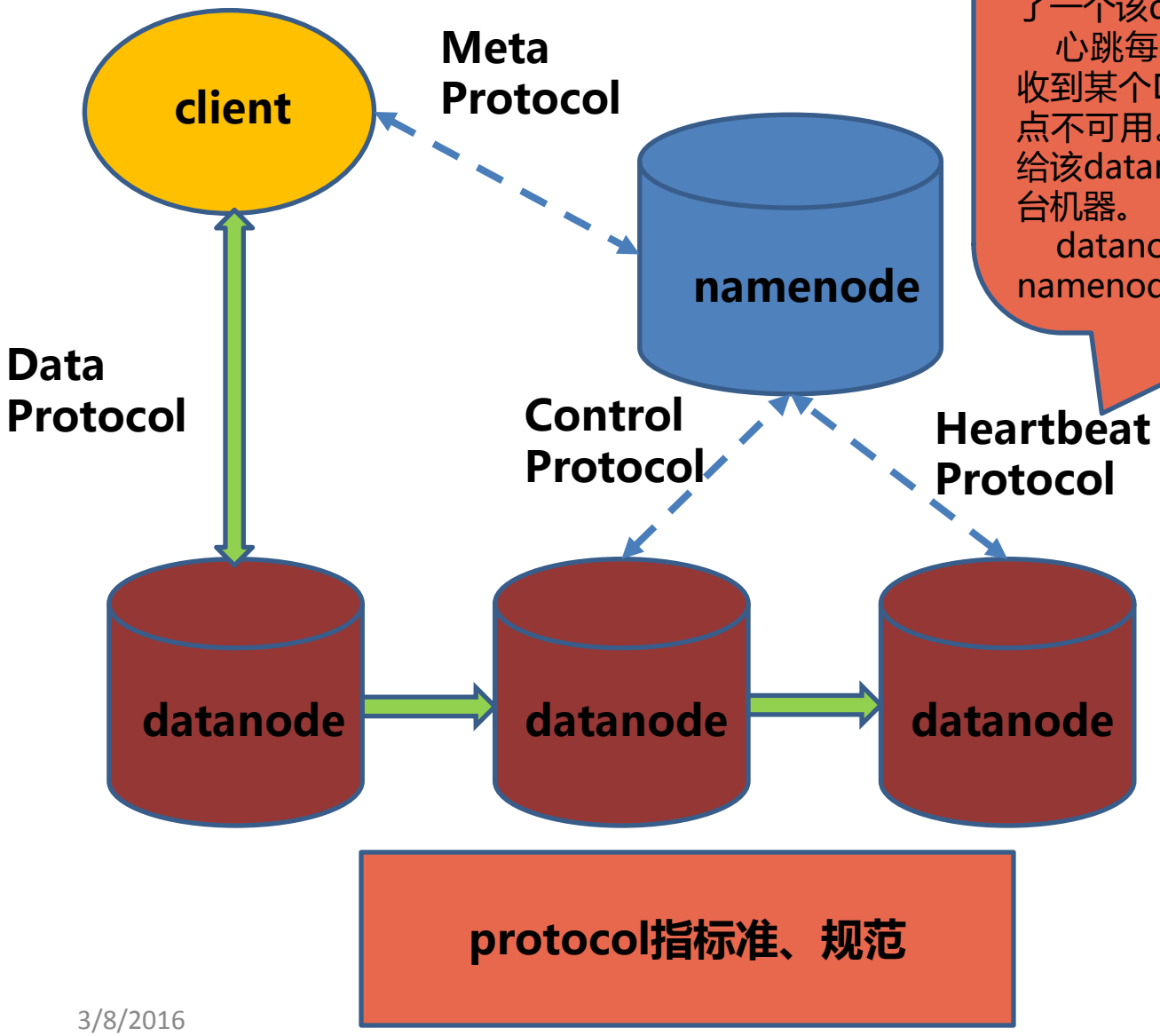
HDFS Architecture



HDFS Architecture



HDFS Architecture

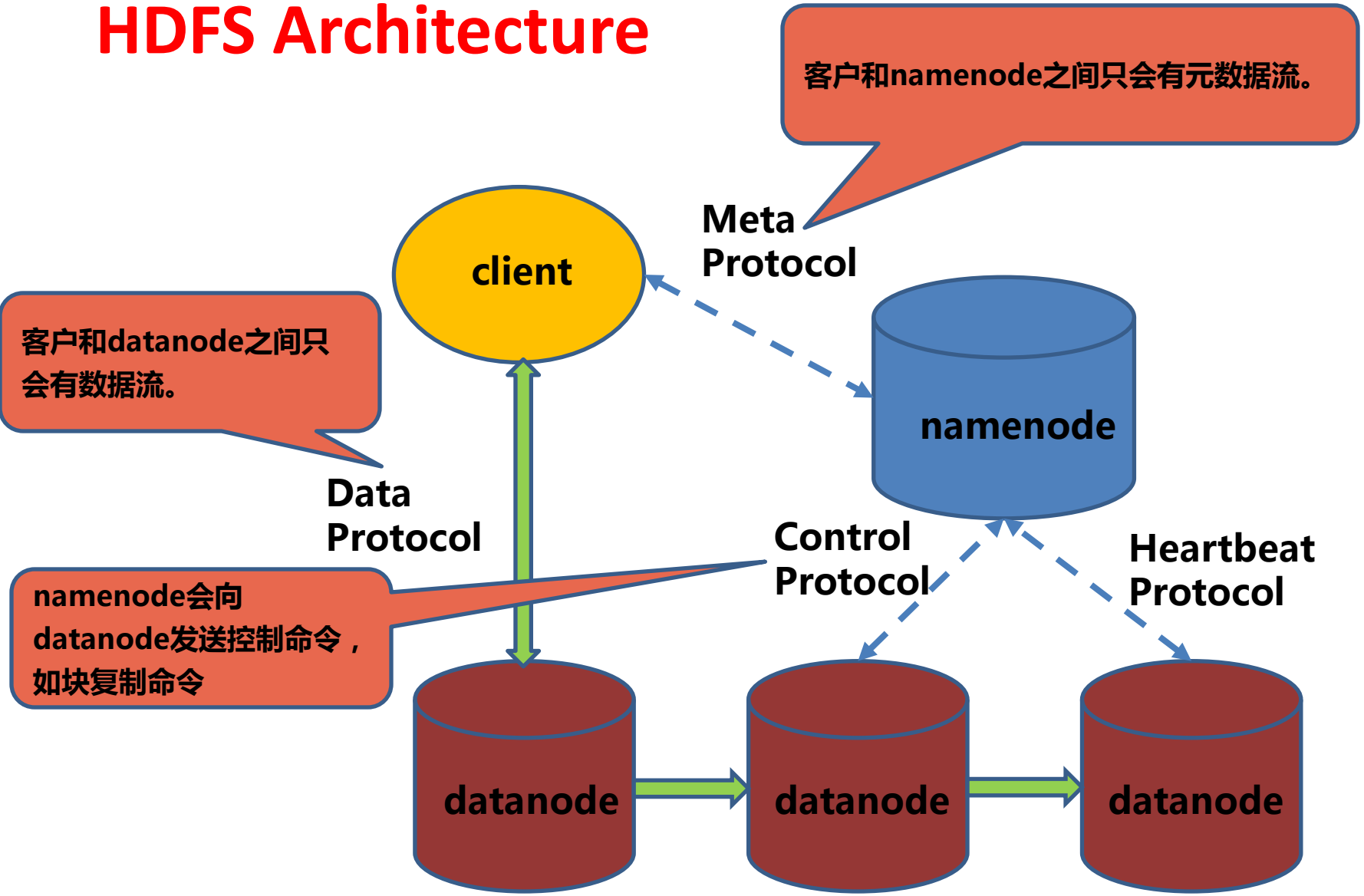


namenode全权管理数据块的复制，它周期性地从每个datanode接收心跳信号和块状态报告。接收到心跳信号意味着该datanode节点工作正常。块状态报告包含了一个该datanode上所有数据块的列表。

心跳每3秒一次，如果超过10分钟没有收到某个DataNode 的心跳，则认为该节点不可用。心跳返回结果带有namenode给该datanode的命令如复制块数据到另一台机器。

datanode 周期性（1小时）的向namenode上报所有的块信息。

HDFS Architecture



HDFS的特点

- 处理超大文件
  - 处理数百MB，甚至TB数量级的文件。目前在实际应用中，HDFS可以用来管理PB级的数据。
- 流式访问数据
  - HDFS的设计是以满足“一次写入，多次读取”类型的任务为目标。一个数据集一旦由数据源生成，就会被复制分发到不同的存储节点中，然后响应各种数据分析任务的请求。
  - 请求读取整个数据集比读取一条记录更高效。
- 运行于廉价的商用机器集群上
  - 对硬件要求不高，只需运行在廉价的商用机器集群上
  - 意味着大型集群中出现节点故障的概率比较高，这就要求在设计时充分考虑数据的可靠性，安全性和高可用性

HDFS不适合做什么

- 不适合低延迟数据访问
  - 如果要处理一些用户要求时间比较短的低延迟用户请求，则HDFS不适合。
  - HDFS是为处理大规模数据分析处理任务而设计的，设计目标不是低延迟，而是高数据吞吐量，因此更适合离线任务
- 无法高效存储大量小文件
  - Hadoop需要用Namenode来管理文件系统的元数据，因此文件数量大小要由Namenode来决定。例如平均每个文件的元数据占100字节，100万个文件就要占100M内存。如果文件更多，Namenode处理元数据的时间就不可接受。
- 不支持多用户写入和任意修改文件
  - 一个文件只有一个写入者
  - 文件只支持在文件尾追加

HDFS Command Line Interface(CLI)

- **File System Shell (fs)**
  - Invoked as follows:

```
hadoop fs <args>
```

  - Example:
    - Listing the current directory in hdfs

```
hadoop fs  – ls
```

HDFS Command Line Interface(CLI)

- **FS shell commands take paths URIs as argument**
  - URI format:

```
scheme://path
```

- **Scheme:**
  - For the local file system, the scheme is file
  - For HDFS, the scheme is hdfs

```
hadoop fs  - copyFromLocal
    file://myfile.txt
    hdfs://localhost/user/keith/myfile.txt
```

HDFS Command Line Interface(CLI)

- **Many POSIX-like commands**
  - cat, chgrp, chmod, chown, cp, ls, mkdir, mv, rm,…
- **Some HDFS-specific commands**
  - copyFromLocal, copyToLocal, get, put,…

HDFS – Specific commands

- **copyFromLocal / put**
  - Copy files from the local file system into hdfs

```
hadoop fs -copyFromLocal <localsrc>  <dst>
```

  - or

```
hadoop fs -put <localsrc>  <dst>
```

```
hadoop fs  – copyFromLocal
    file://myfile.txt
    hdfs://localhost/user/keith/myfile.txt
```

HDFS – Specific commands

- **copyToLocal / get**
  - Copy files from hdfs into the local file system

```
hadoop fs -copyToLocal [-ignorecrc] [-crc]
        <src> <localdst>
```

  - or

```
hadoop fs -get [-ignorecrc] [-crc]
        <src> <localdst>
```

Hadoop Commands Reference

http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/CommandsManual.html

HDFS Java API

- 对HDFS中的文件操作主要涉及以下几个类：
  - **Configuration**类：该类的对象封装了客户端或者服务器的配置。
  - **FileSystem**类：该类是个**抽象类**，只能通过该类的**get**静态方法得到具体类的对象,如**DistributedFileSystem**对象
  - **FSDataInputStream**和**FSDataOutputStream**：这两个类是HDFS中的输入输出流。分别通过FileSystem的**open**方法和**create**方法获得。

HDFS Java API

- 如何获得一个**DistributedFileSystem**对象
- 通过静态工程方法**get**获得：

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

  - 通过URI的scheme来决定返回一个什么样的文件系统对象，如hdfs://localhost/user/tom/quangle.txt作为URI参数会返回**DistributedFileSystem**对象

HDFS Java API：如何从HDFS读数据

- 首先通过静态工场方法**get**获得**DistributedFileSystem**对象
- 利用**DistributedFileSystem**对象的**open**方法获得**FSDataInputStream**对象

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

```
public class FileSystemCat {
    public static void main(String[] args) throws Exception {
        String uri = args[0]; // hdfs://localhost/user/tom/quangle.txt
        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);
        }
    }
}
```

会获得DistributedFileSystem对象

会获得FSDataInputStream对象

HDFS Java API：如何从HDFS读数据

- The program runs as follows:

```
% 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.

HDFS Java API：如何向HDFS写数据

- 首先通过静态工程方法**get**获得**DistributedFileSystem**对象
- 利用**DistributedFileSystem**对象的**create**方法得**FSDataOutputStream**对象

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

```
public class FileCopy{
    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));
        IOUtils.copyBytes(in, out, 4096, true);
    }
}
```

会获得DistributedFileSystem对象

会获得FSDataOutputStream对象

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

HDFS Java API：如何创建目录

- **FileSystem**也提供了创建HDFS目录的方法
- 首先通过静态工程方法**get**获得**DistributedFileSystem**对象
- 利用**DistributedFileSystem**对象的**mkdirs**方法创建目录

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

- 这个方法会按照客户要求创建未存在的父目录。如果包括包括父目录在内的目录创建成功，方法返回true
- 一般不需要特别创建一个目录，因为调用create方法写文件时会自动创建文件的所在目录

- /root/user/gxw/dir1

HDFS Java API：如何查询文件系统

- JAVA API也提供了文件系统的基本查询系统，通过这个接口，可以查询文件系统元数据和目录结构
- 文件系统元数据都被封装在**FileStatus**类里
- **FileSystem**的**getFileStatus**提供了获取目录或文件的状态对象方法

```
FileStatus  getFileStatus( Path f) throws IOException
```

HDFS Java API：如何查询文件系统

- JAVA API也提供了文件系统的基本查询系统，通过这个接口，可以查询文件系统元数据和目录结构
- **FileSystem的listStatus**提供了列出文件目录内容的方法

```
FileStatus[]  listStatus( Path f) throws IOException

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

FileStatus[]  listStatus( Path[] files) throws IOException

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

- 当传入参数是一个文件时，listStatus方法返回长度为1的FileStatus对象数组；当传入参数是目录时，它会返回包含0个或多个FileStatus对象的数组
- PathFilter是用来过滤满足正则表达式的文件或目录
- 4个版本的重载方法，支持输入路径数组，可以一次对多个目录进行查询，并将查询结果放到一个FileStatus对象数组

HDFS Java API：显示文件元数据信息

```
public class ShowFileStatus {
        public static void main(String[] args) throws Exception{
            String uri = args[0];
            Configuration conf = new Configuration();
            FileSystem fs = FileSystem.get(new URI(uri), conf);
            Path path = new Path(uri);
            FileStatus stat = fs.getFileStatus(path);
            System.out.println("Full path:" + stat.getPath().toUri().getPath());
            if(stat.isDir())
                System.out.println(" is directory");
            else
                System.out.println(" is file");
            System.out.println("Length:" + stat.getLen());
            System.out.println("Last modified time:" + stat.getModificationTime());
            System.out.println("Replication:" + stat.getReplication());
            System.out.println("Block size:" + stat.getBlockSize());
            System.out.println("Owner:" + stat.getOwner());
            System.out.println("Group:" + stat.getGroup());
            System.out.println("Permission:" + stat.getPermission());
        }
}
```

```
hadoop jar ch02.jar ch02.ShowFileStatus /user/hadoop/input/core-site.xml
```

HDFS Java API：列出目录内容

```
public class ListStatus {

        public static void main(String[] args) throws Exception{
                String uri = args[0];
                Configuration conf = new Configuration();
                FileSystem fs = FileSystem.get(new URI(uri), conf);
                Path path = new Path(uri);
                FileStatus[] status = fs.listStatus(path);
                Path[] listedPaths = FileUtil.stat2Paths(status);
                for(Path p : listedPaths){
                        System.out.println(p);
                }
        }
}
```

hadoop jar ch02.jar ch02.ListStatus /user/hadoop/input

HDFS Java API：通过通配符实现目录筛选

- **FileSystem的globStatus提供了利用通配符筛选目录或文件**

FileStatus[] globStatus(Path pathPattern) throws IOException

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

通配符	名称	匹配功能
*	星号	匹配0个或多个字符
?	问号	匹配一个字符
[ab]	字符类别	匹配{a，b}中的一个字符
[^ab]	非此字符类别	匹配不属于{a，b}中的一个字符
[a-b]	字符范围	匹配在{a,b}范围内（字典序）的字符，包括a，b
[^a-b]	非此字符范围	匹配不在{a,b}范围内（字典序）的字符，包括a，b
{a,b}	或选择	匹配表达式a或者b
\c	转义字符	转义元字符

HDFS Java API：利用通配符

```
public class ListStatusWithPattern {

        public static void main(String[] args) throws Exception{
            String uriPattern = args[0];
            Configuration conf = new Configuration();
            FileSystem fs = FileSystem.get(new URI(uriPattern), conf);
            Path pathPattern = new Path(uriPattern);
            FileStatus[] status = fs.globStatus(pathPattern);
            Path[] listedPaths = FileUtil.stat2Paths(status);
            for(Path p : listedPaths){
                    System.out.println(p);
            }
        }
}
```

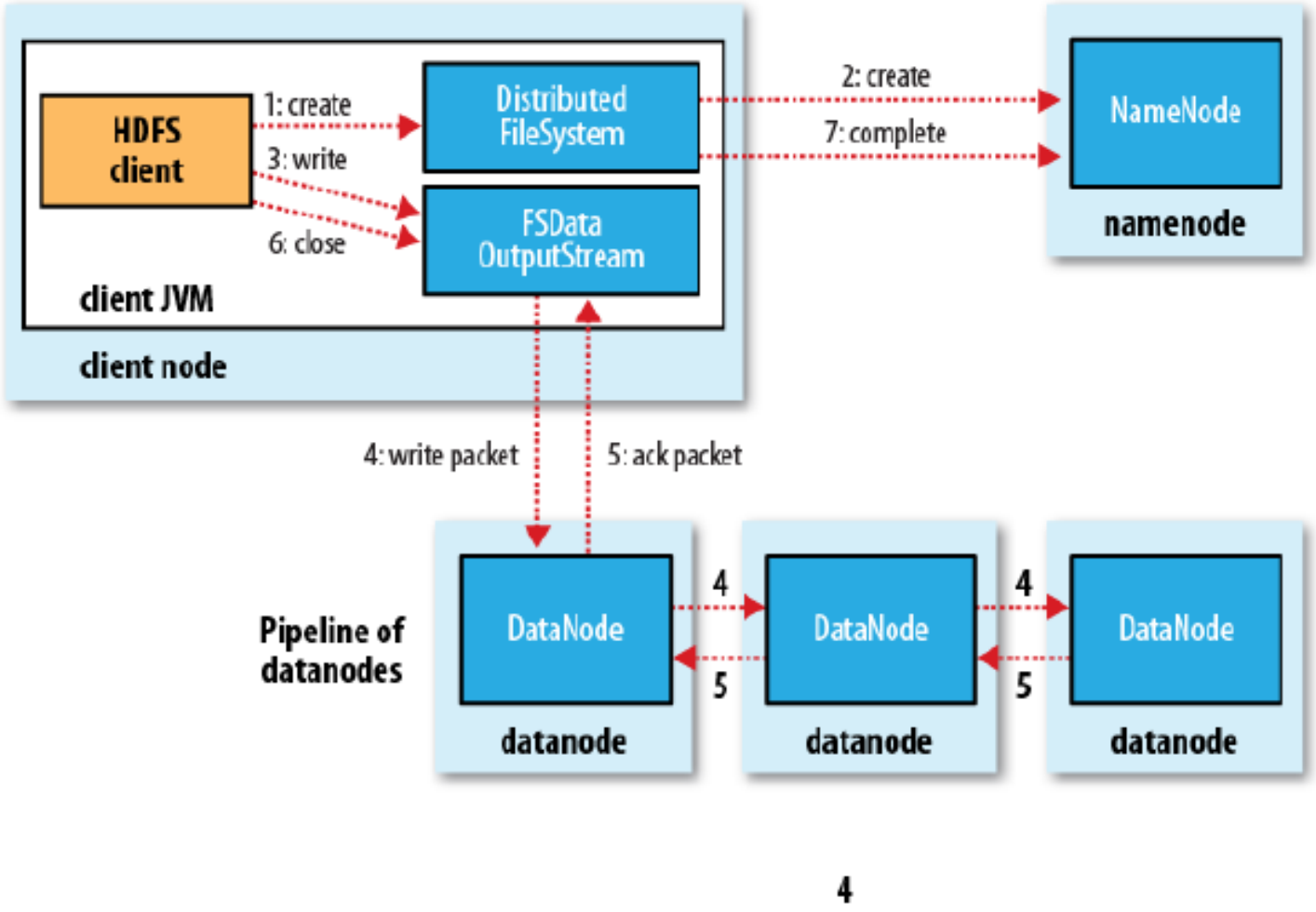
hadoop jar ch02.jar ch02.ListStatusWithPattern /user/hadoop/input/*.xml

HDFS Java API： PathFilter

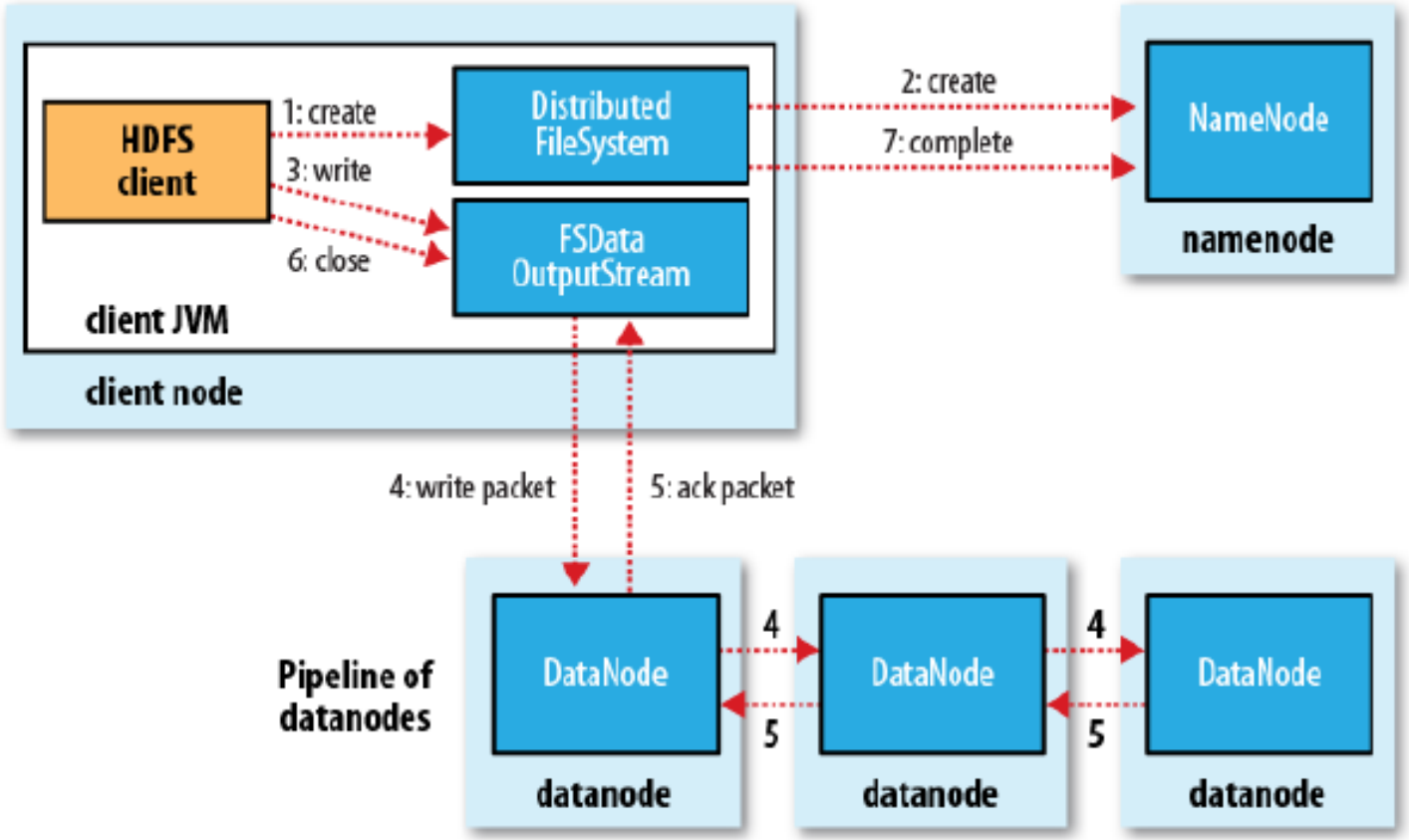
```
//Hadoop API 定义了接口PathFilter
package org.apache.hadoop.fs;
public interface PathFilter{
        boolean accept(Path path); //如果接受该path，则返回true
}
```

```
//现在实现一个排除匹配正则表达式的目录的PathFilter
public class RegexExcludePathFilter implements PathFilter{
        private final String regex;
        public RegexExcludePathFilter(String regext){
                this.regex = regex;
        }
        public boolean accept(Path path){
                return !path.toString().match(regex);
        }
}
```

HDFS如何写文件？

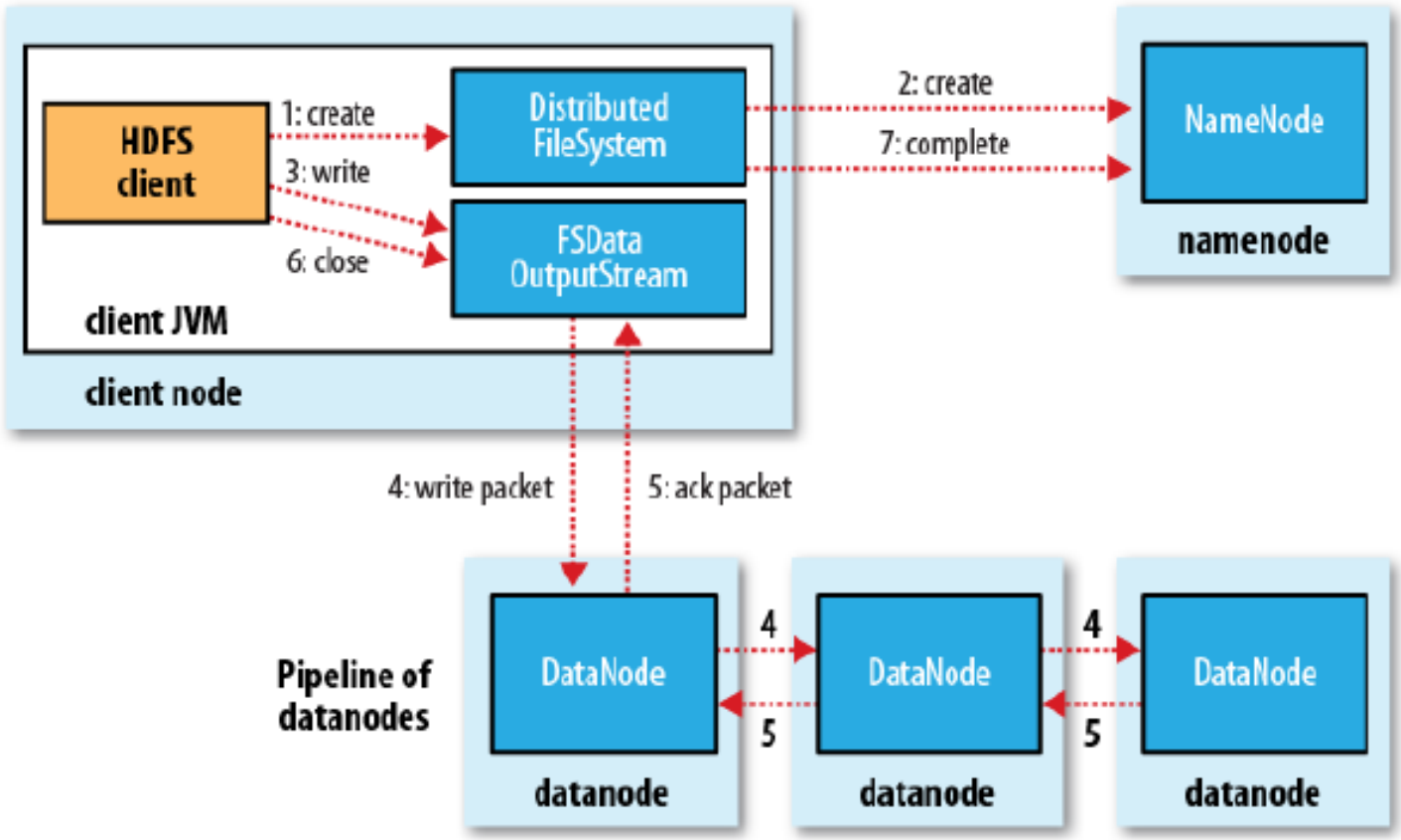

HDFS client
1: create
3: write
6: close
Distributed FileSystem
FSData OutputStream
client JVM
client node
2: create
7: complete
NameNode
namenode
4: write packet
5: ack packet
Pipeline of datanodes
DataNode
datanode
4
5
DataNode
datanode
4
5
DataNode
datanode

HDFS如何写文件？



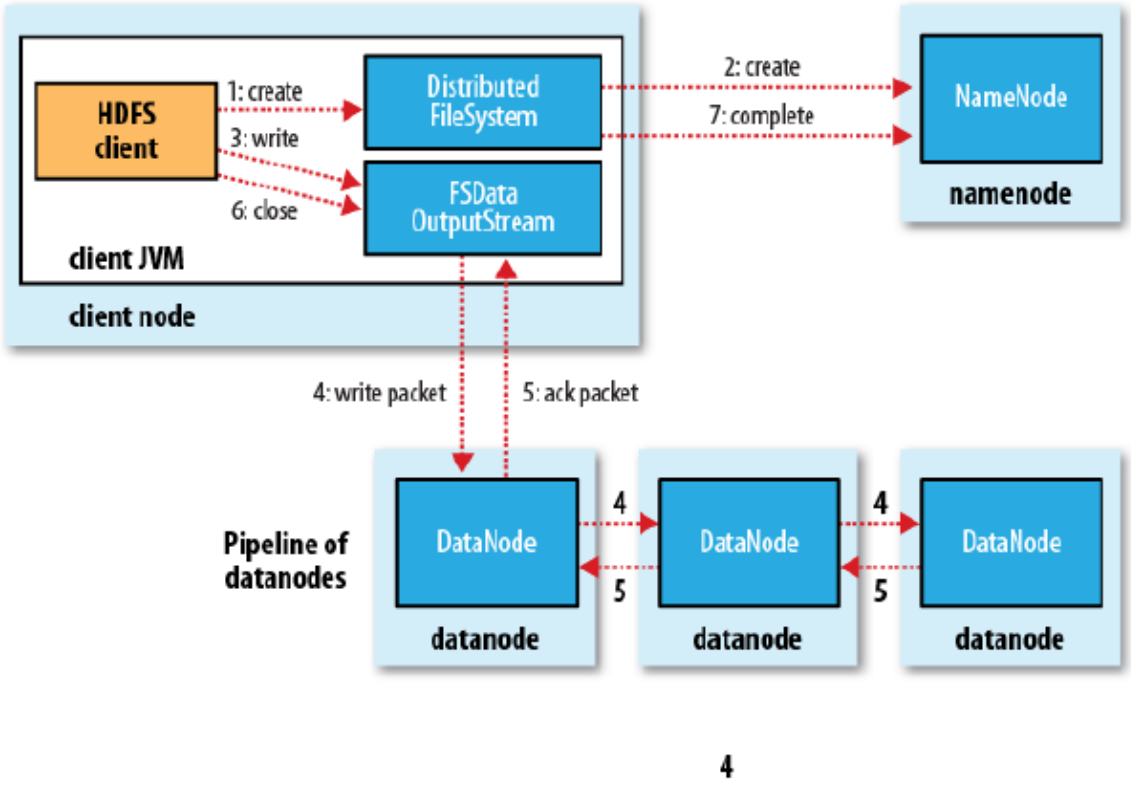
Step1:客户调用DistributedFileSystem对象的create()方法创建一个文件

HDFS如何写文件？



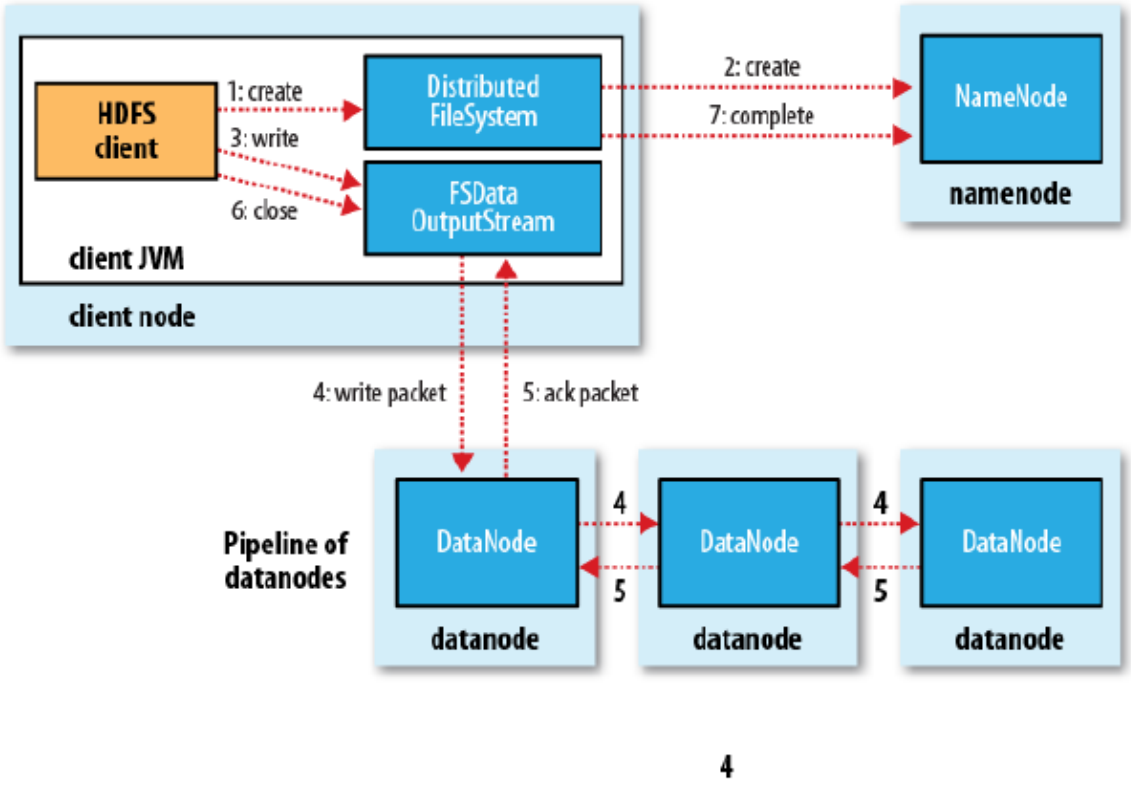
Step 2:DistributedFileSystem 通过向namenode发出RPC调用，在namenode的文件系统命名空间中创建一个新文件，此时没有block和datanode与该文件关联。Namenode会进行各种验证（如用户权限），如果验证成功namenode会创建一个新文件的记录，同时DistributedFileSystem 对象会返回FSDataOutputStream给客户以让客户写入数据。 FSDataOutputStream内部封装了一个 DFSOutputStream对象, 由这个对象来处理namenode与datanode之间的通信

HDFS如何写文件？



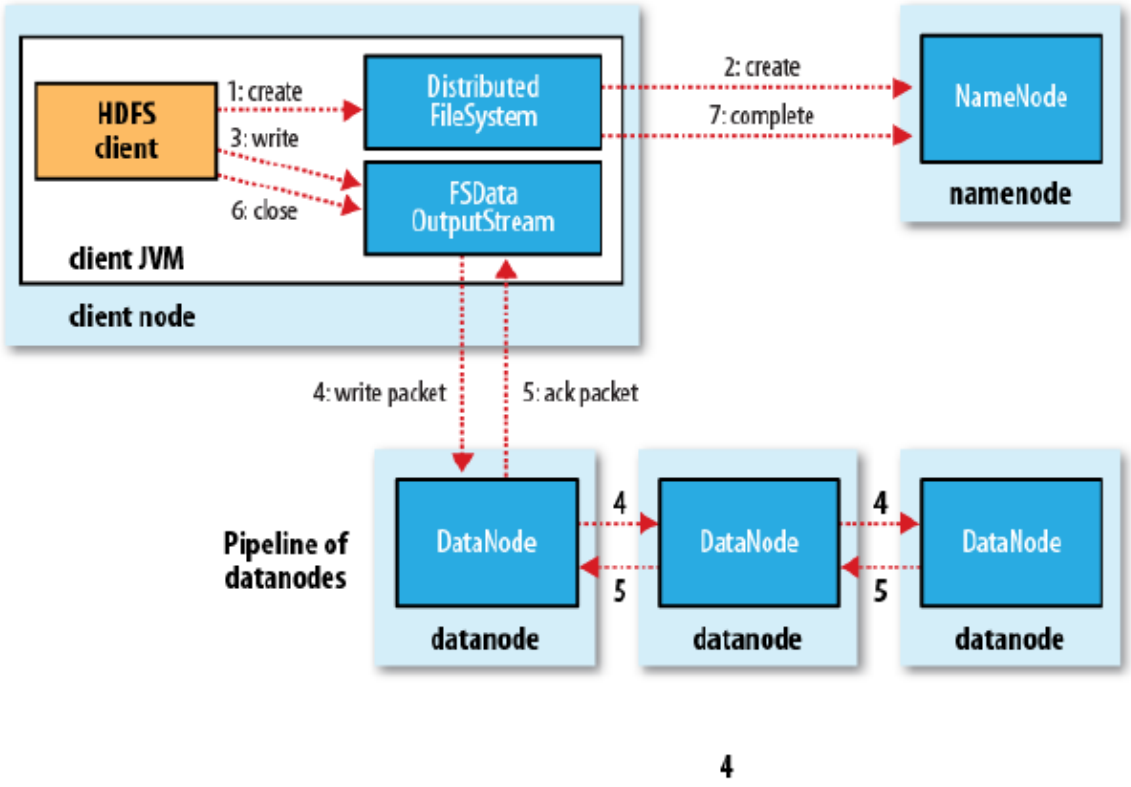
Step 3当客户写入数据时，DFSOutputStream会将文件分成packets，然后写入内部队列，称为*data queue*. 数据队列由DataStreamer处理， DataStreamer的作用是请求namenode为packets挑选合适的datanode列表来存放数据包对应的blocks的复本。被挑选的datanode列表形成了pipeline，假设复本数为3，则pipeline里有3个datenode。

HDFS如何写文件？



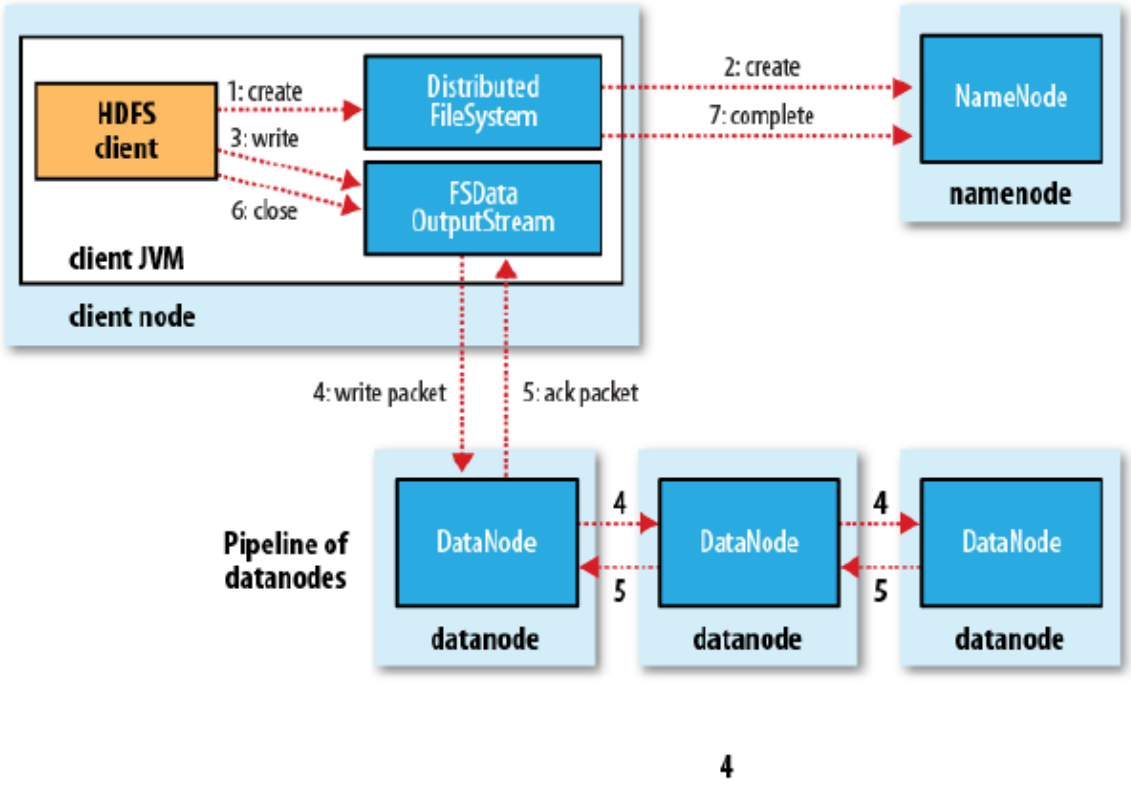
Step 4: DataStreamer 将packets以流的形式发送给pipeline里的第1个datanode，第1个datenode会存储这个包，并将这个包转发到pipeline里的第2个datanode。类似地，第2个datanode存储完这个包后将这个包转发给第3个datanode。

HDFS如何写文件？



Step 5: DFSOutputStream同时也维护了另一个内部packets队列，用来等待pipeline里datanode的确认信息，这个队列叫*ack queue*。 当所有datanode都返回了一个packet的确认信息，这个packet才会从*ack queue*中删除。

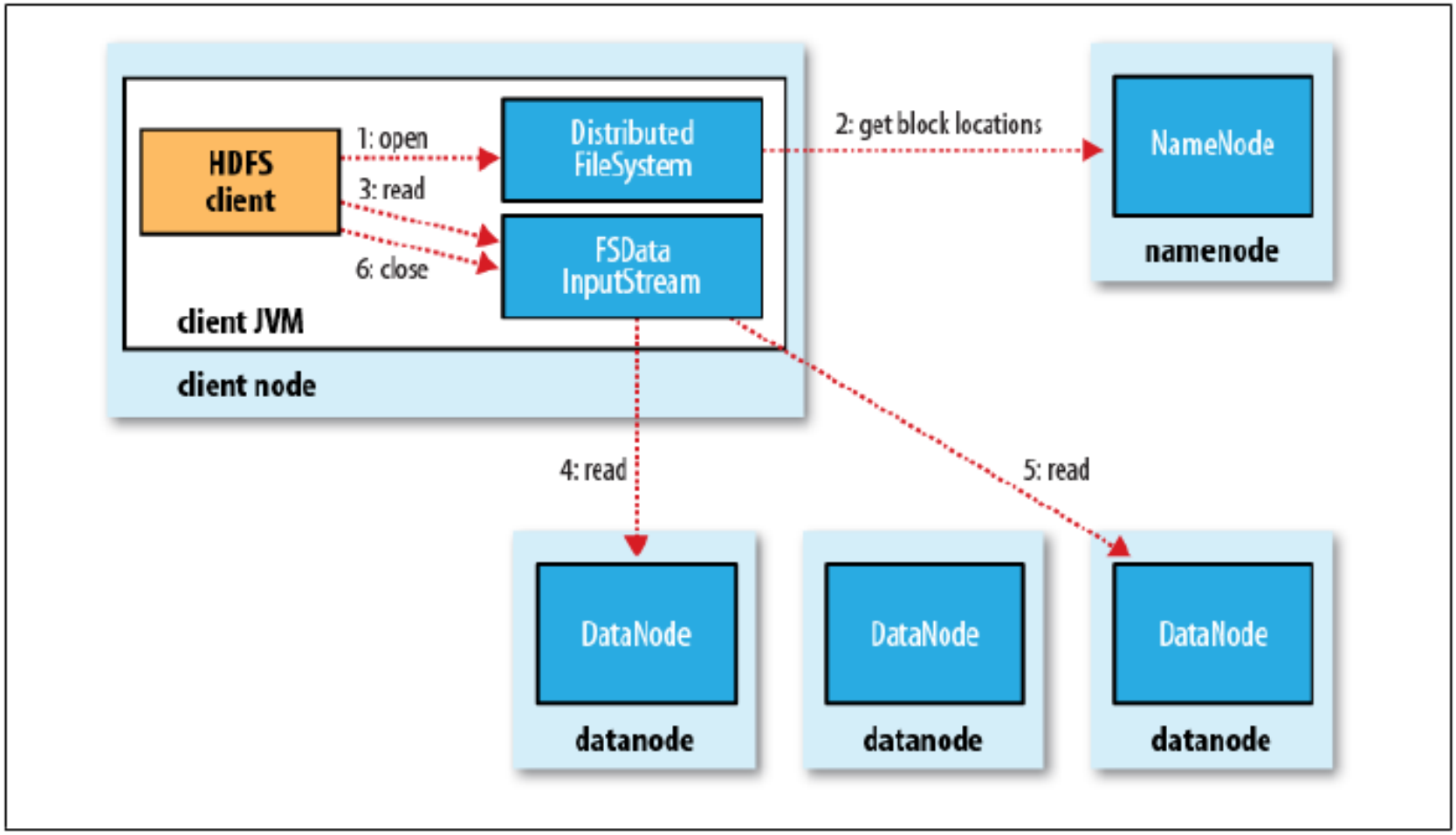
HDFS如何写文件？



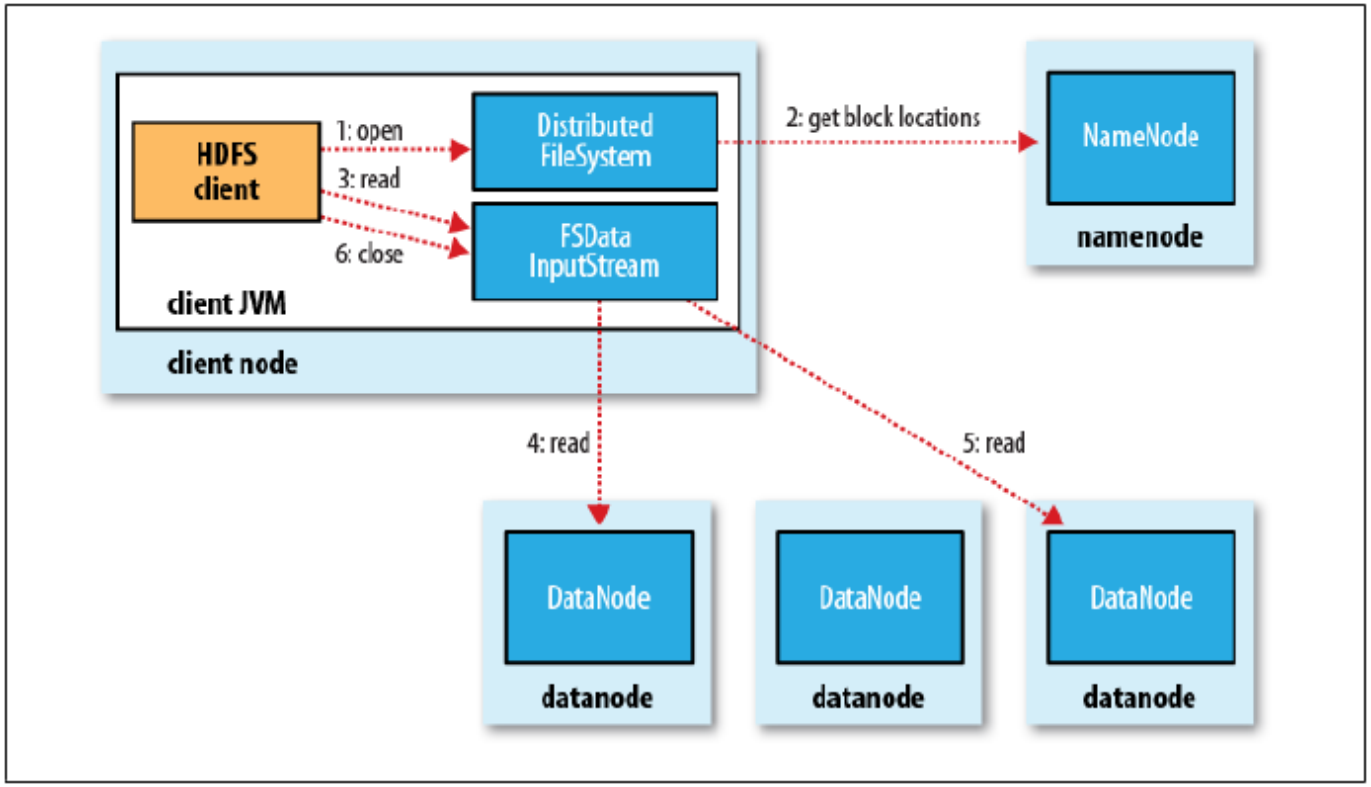
Step 6: 当客户端完成数据写入操作后，会调用FSDataOutputStream流的close方法。这个操作会将所有剩余的packets放入到datanode pipeline，并等待这些packets的确认信息。

Step7 ： 当收到所有剩余packets的确认信息后，客户通知namenode文件写入完毕。

HDFS如何读文件？

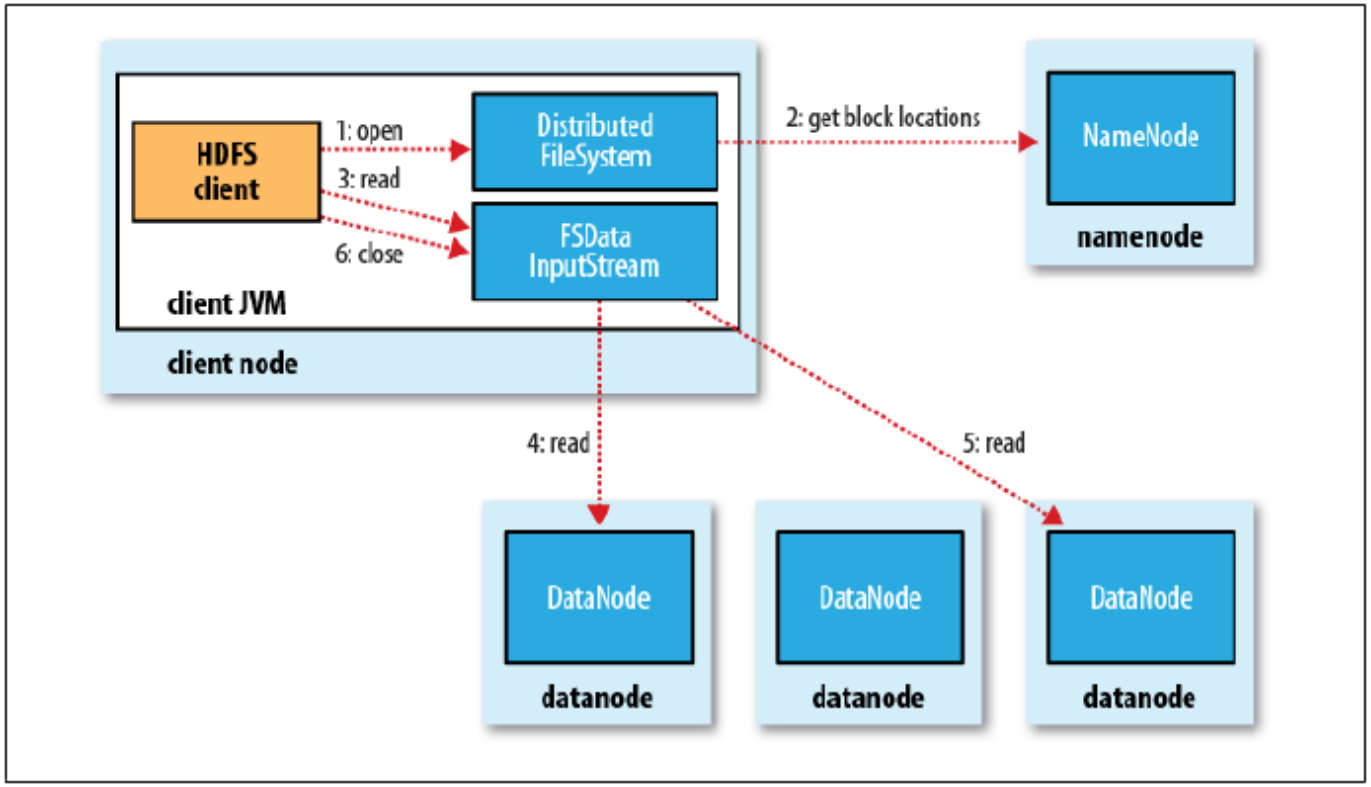

HDFS client
1: open
3: read
6: close
Distributed FileSystem
FSData InputStream
client JVM
client node
2: get block locations
NameNode
namenode
4: read
5: read
DataNode
datanode
DataNode
datanode
DataNode
datanode

HDFS如何读文件？



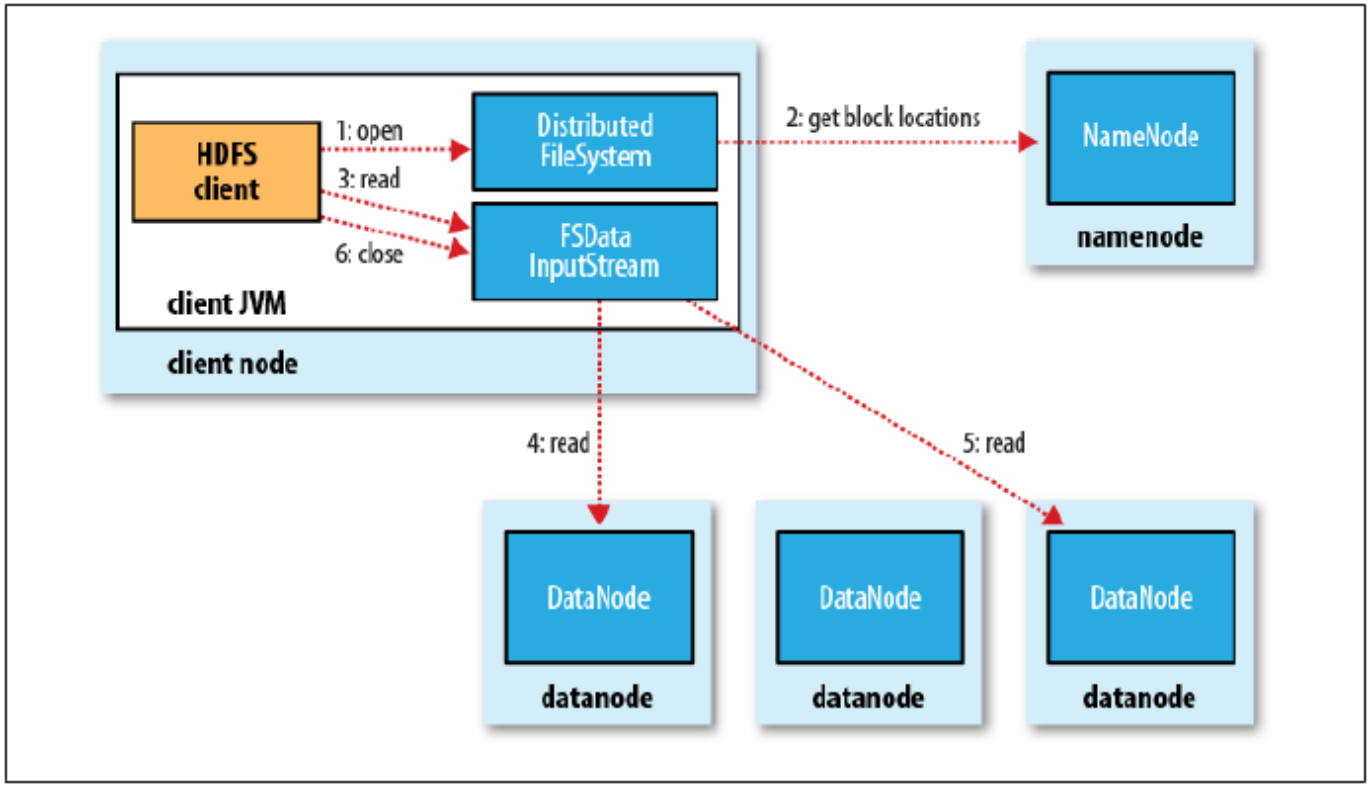
Step 1：客户调用FileSystem对象的open方法打开需要读取的文件，对于HDFS，FileSystem对象是DistributedFileSystem的一个实例

HDFS如何读文件？



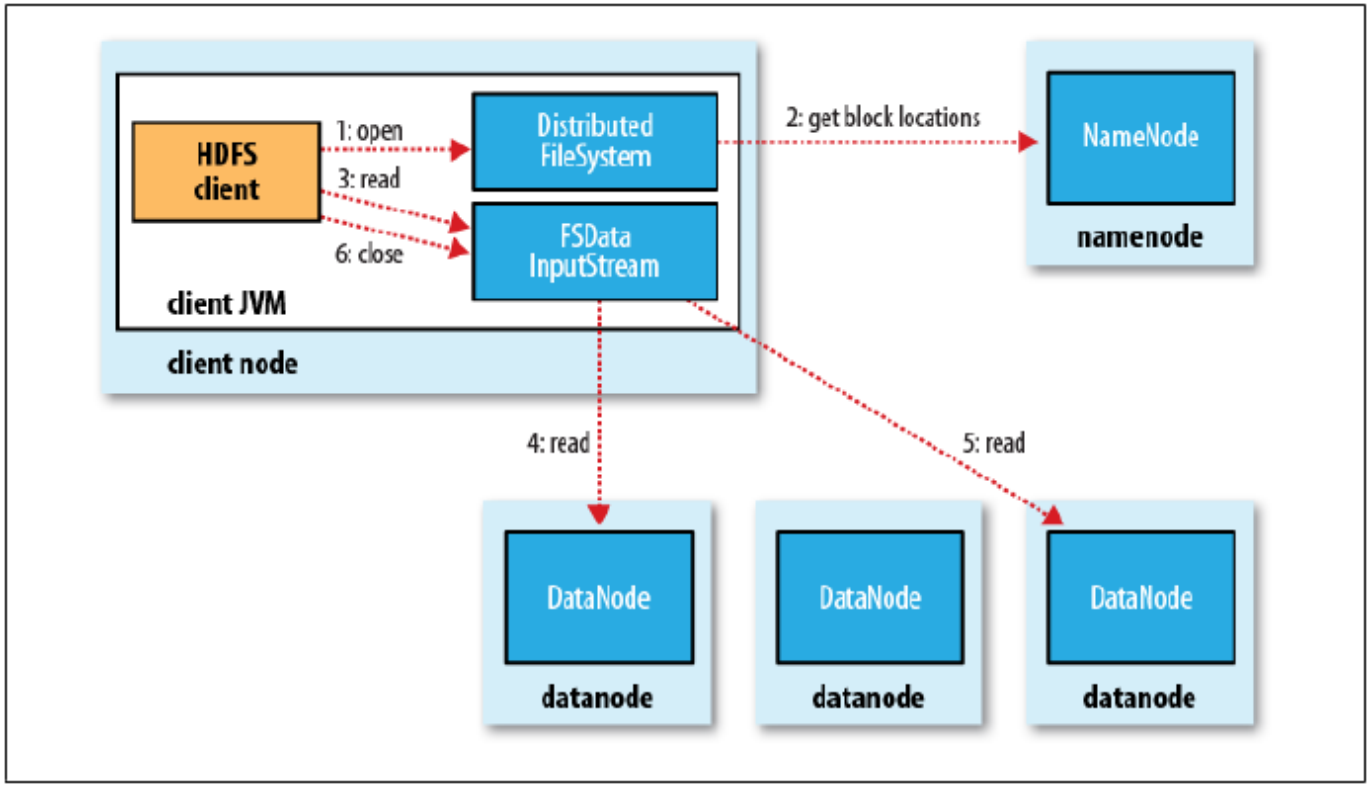
Step 2: DistributedFileSystem通过RPC请求namenode来确定文件块（blocks）的位置。Namenode会返回每个block对应的datanode的地址。

HDFS如何读文件？



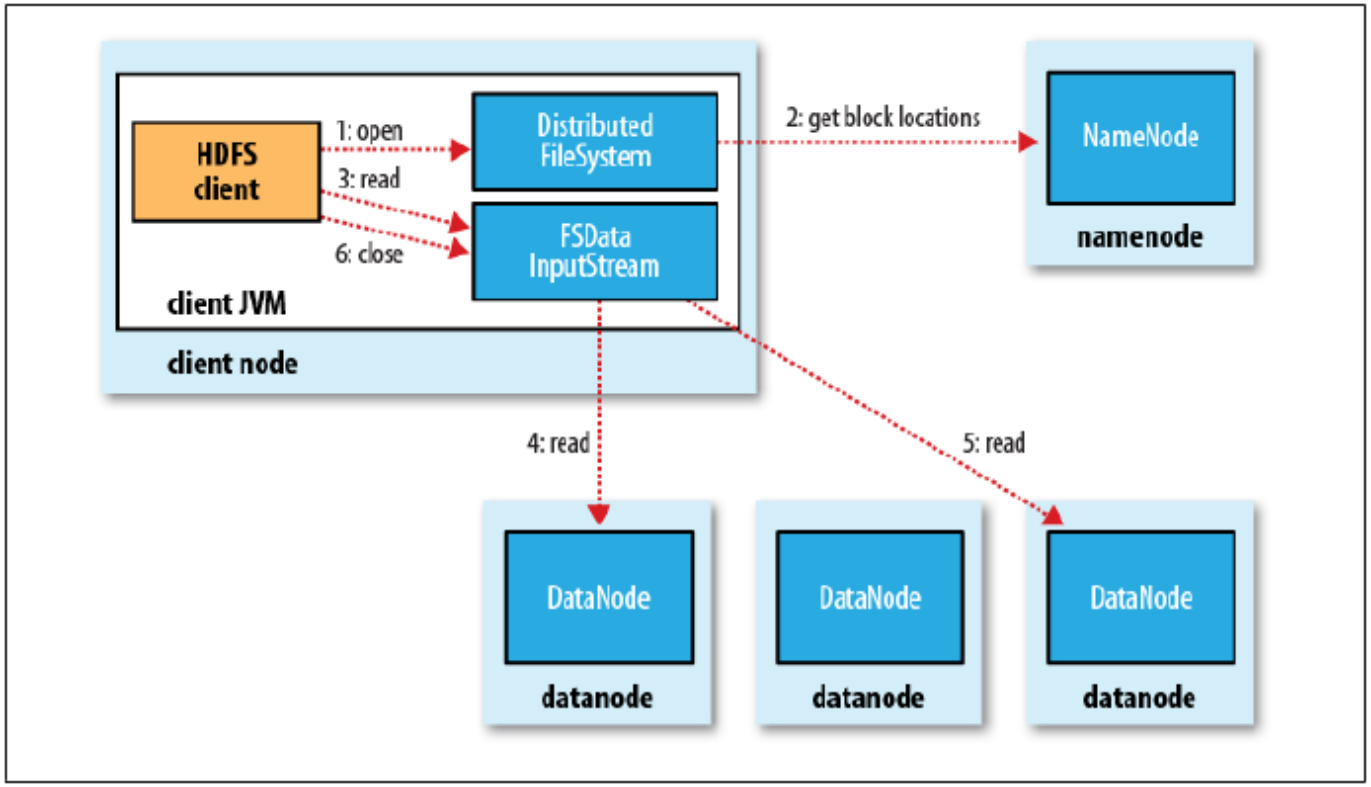
Step 3: DistributedFileSystem的open方法会返回一个FSDataInputStream对象给客户， FSDataInputStream是支持文件定位（file seeks）的输入流，客户利用这个对象读取文件数据。 FSDataInputStream封装了一个DFSInputStream对象，这个对象用来管理namenode和datanode之间的IO。客户调用FSDataInputStream的read方法开始读取数据。

HDFS如何读文件？



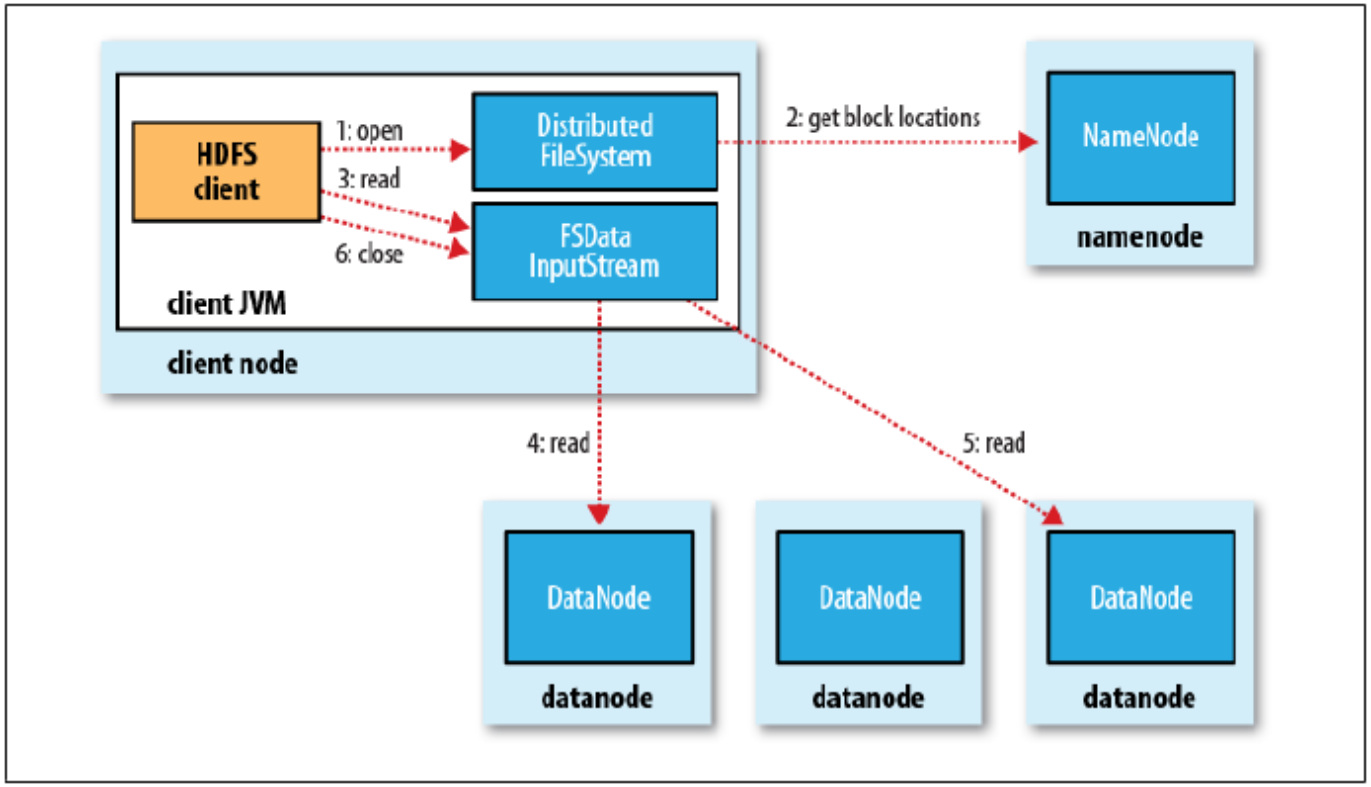
Step 4: DFSInputStream对象内部保存了文件开始部分若干blocks对应的datanode的地址，首先它会连接包含文件第一个block的最近的datanode，并将数据流从datanode读回客户。客户会重复地调用流的read方法直到整个block的数据全部读完。

HDFS如何读文件？



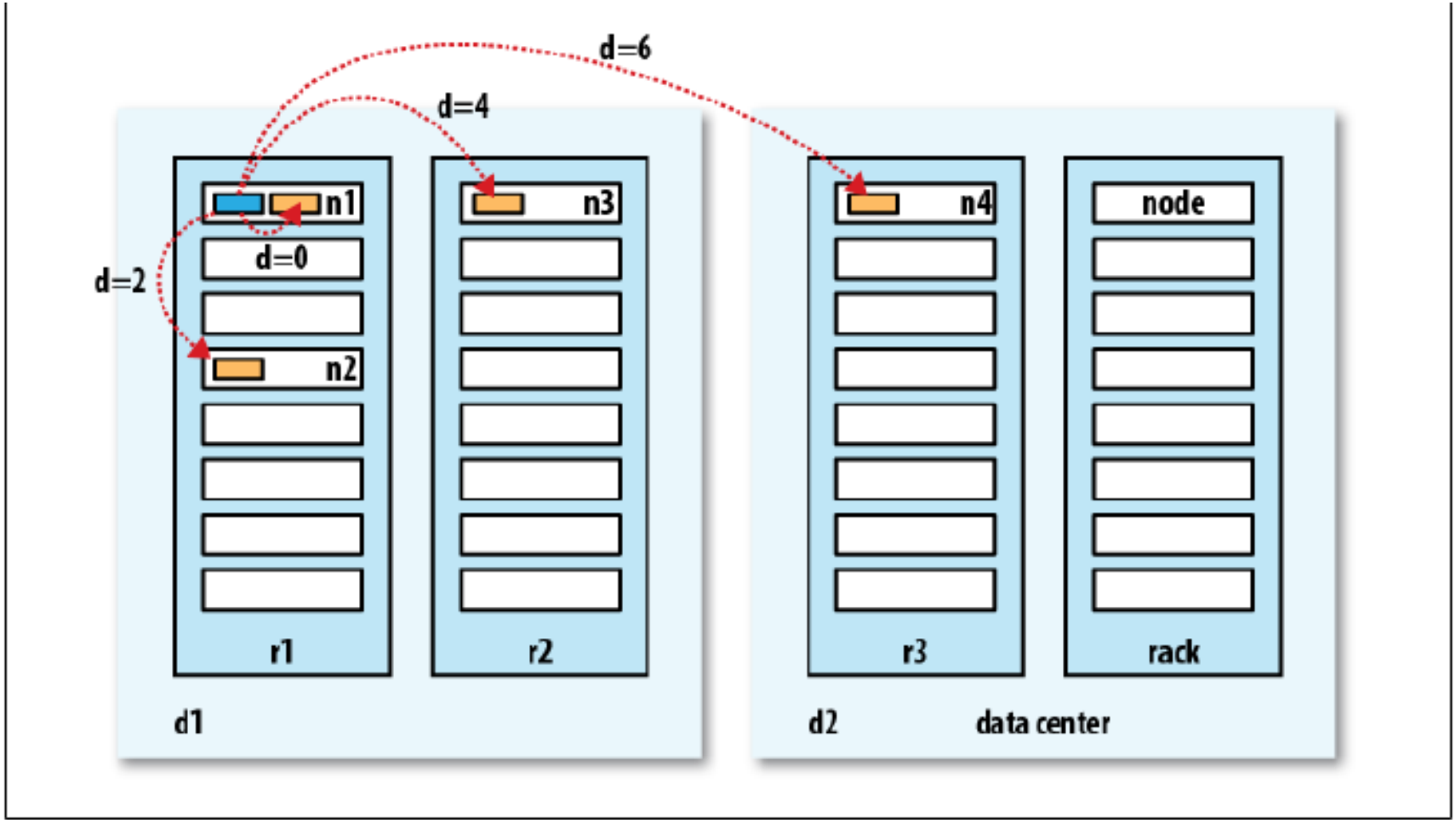
Step 5: 当第一个block 的数据读取完毕后， DFSInputStream会关闭和datanode的连接，并查找存储下一个block的最近的datanode并读取。

HDFS如何读文件？



Step 6: blocks会被依次读取，这个过程中DFSInputStream 会不断连接新的datanode。DFSInputStream也会请求namenode获取下一批blocks对应的datanode地址。当客户完成数据读取后，调用FSDataInputStream的close方法关闭流。

Network distance in Hadoop



What does it mean for two nodes in a local network to be "close" to each other?

Network distance in Hadoop

- Hadoop takes a simple approach in which the network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor.
- The idea is that the bandwidth available for each of the following scenarios becomes progressively less:
  - Processes on the same node
  - Different nodes on the same rack
  - Nodes on different racks in the same data center
  - Nodes in different data centers

Network distance in Hadoop

- For example, imagine a node n1 on rack r1 in data center d1. This can be represented as /d1/r1/n1. Using this notation, here are the distances for the four scenarios:
  - distance(/d1/r1/n1, /d1/r1/n1) = 0 (processes on the same node)
  - distance(/d1/r1/n1, /d1/r1/n2) = 2 (different nodes on the same rack)
  - distance(/d1/r1/n1, /d1/r2/n3) = 4 (nodes on different racks in the same data center)
  - distance(/d1/r1/n1, /d2/r3/n4) = 6 (nodes in different data centers)

HDFS 一致性模型

- 文件系统一致性模型描述了文件读写的可见性，HDFS牺牲了一些POSIX的需求来补偿性能，所以有些操作可能和传统的文件系统不同
- 当创建一个文件时，它在文件系统的命名空间中是可见的，代码如下：

```
Path p = new Path(“/p”);
fs.create(p);
assertThat(fs.exists(p), is(true));
```

- 但是对这个文件的任何写操作不保证是可见的，即使在数据流已经刷新的情况下，文件的长度在很长时间也会显示为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));
```

HDFS 一致性模型

- HDFS提供了一个同步函数，这个方法是FSDataOutputStream类的sync()函数。当sync()函数返回成功时，HDFS就可以保证此时写入的文件数据文件是一致的并且对于所有用户是可见的

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

- 在HDFS中关闭一个文件也隐式地执行了sync( )函数

```
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())));
```

Hadoop的配置示例:core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
   <property>
      <name>fs.default.name</name>
      <value>hdfs://192.168.141.200:9000</value>
    </property>
   <property>
      <name>hadoop.tmp.dir</name>
      <value>/home/hadoop/HadoopClusterTest/tmpdir</value>
   </property>
</configuration>
```

指定文件系统的URI shema，namenodeIP地址，端口号，以后这个前缀就可以省略

指定临时文件目录

HDFS 系统的配置示例:hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
            <name>dfs.permissions</name>
            <value>false</value>
    </property>
    <property>
            <name>dfs.replication</name>
            <value>2</value>
    </property>
    <property>
            <name>dfs.name.dir</name>
            <value>/home/hadoop/HadoopClusterTest/tmpdir/hdfs/name</value>
    </property>
    <property>
            <name>dfs.data.dir</name>
            <value>/home/hadoop/HadoopClusterTest/tmpdir/hdfs/data</value>
    </property>
</configuration>
```

指定是否开启文件系统的权限，设置成false是为了Eclipse插件使用

指定文件系统复本数量

namenode保存分布式文件系统元数据（edit log和filesystem image）的本地目录列表

datenode保存blocks数据的本地目录列表

HDFS 系统的配置示例:maped-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
        <name>mapred.job.tracker</name>
        <value>192.168.141.200:9001</value>
  </property>
  <property>
        <name>mapred.child.java.opts</name>
        <value>-Xmx512m</value>
  </property>
  <property>
        <name>mapred.tasktracker.map.tasks.maximum</name>
        <value>6</value>
  </property>
  <property>
        <name>mapred.tasktracker.reduce.tasks.maximum</name>
        <value>2</value>
  </property>
</configuration>
```

设置jobtracker的ip地址和端口号

设置tasktracker能使用的最大内存

map task最大数

reduce task最大数

Hadoop的配置示例:masters和slaves文件

masters文件

192.168.141.200

slaves文件

192.168.141.201

HDFS 文件系统的结构

- Namenode文件系统结构
- 最新格式化的Namenode会创建以下的目录结构

```
${dfs.name.dir}/current/VERSION
                      /edits
                      /fsimages
                      /fstime
```

  - ${dfs.name.dir}是在hdfs-site.xml中定义的属性值，这个值指定了在namenode保存分布式文件系统元数据（edit log和filesystem image）的本地目录列表，每个元数据文件在本地目录列表中的每个目录都有备份
  - edits文件记录了对文件系统的编辑日志,二进制文件
  - fstime记录了每次检查点的时间，二进制文件

HDFS 文件系统的结构

- Namenode文件系统结构
- 最新格式化的Namenode会创建以下的目录结构

```
${dfs.name.dir}/current/VERSION
                      /edits
                      /fsimages
                      /fstime
```

- fsimage是文件系统镜像文件，记录以序列化格式存储的文件系统目录和文件inodes，每个inode表示一个文件或目录的元数据信息。fsimage文件是文件系统元数据的持久性检查点（persistent checkpoint），二进制文件
- 注意fsimage不保存datanode与block之间的映射关系。namenode是将这种映射关系维护在内存里：每当datanode加入集群时，namenode向datanode索取块列表以建立映射关系。Namenode还周期性地查询datanode以保证它拥有最新的块映射

HDFS 文件系统的结构

- namenode的VERSION文件包含了正在运行的HDFS的版本信息
  - namespaceID：文件系统的唯一标识符，任何datanode在注册到datanode前都不知道namespaceID值，因此namenode可以利用该属性鉴别新的datanode
  - cTime：标记了namenode存储系统的创建时间，对于刚刚格式化的存储系统，这个属性值为0；但文件系统升级后，该值会更新到新的时间戳
  - storageType：说明该存储目录包含的是namenode的数据结构
  - layoutVersion：描述HDFS持久化数据结构（布局）的版本，只要布局变更，版本号便递减；当布局变更时，HDFS需要升级

```
#Tue Mar 10 19:21:36 GMT 2009
namespaceID=134368441
cTime=0
storageType=NAME_NODE
layoutVersion=-18
```

HDFS 文件系统镜像和编辑日志

- 文件系统客户端执行写操作时，这些操作首先被记录到编辑日志中
  - namenode是在内存中维护文件系统的元数据
  - 当编辑日志被修改时，内存中相关元数据信息也同步更新
  - 内存中的元数据可以支持客户端的读请求
- fsimage文件是文件系统元数据的一个持久性检查点
- 是最近一次checkpoint内存中元数据的磁盘镜像
  - 并非每一个写操作都会更新这个文件，因为fsimage是一个大型文件（可能高达几个GB），如果频繁地执行写操作 会使得系统运行极为缓慢
  - 但如果namenode发生故障，可以先把fsimage文件载入内存重构最近checkpoint的元数据，再执行编辑日志中记录的各项操作。namenode在启动阶段就是这么做的

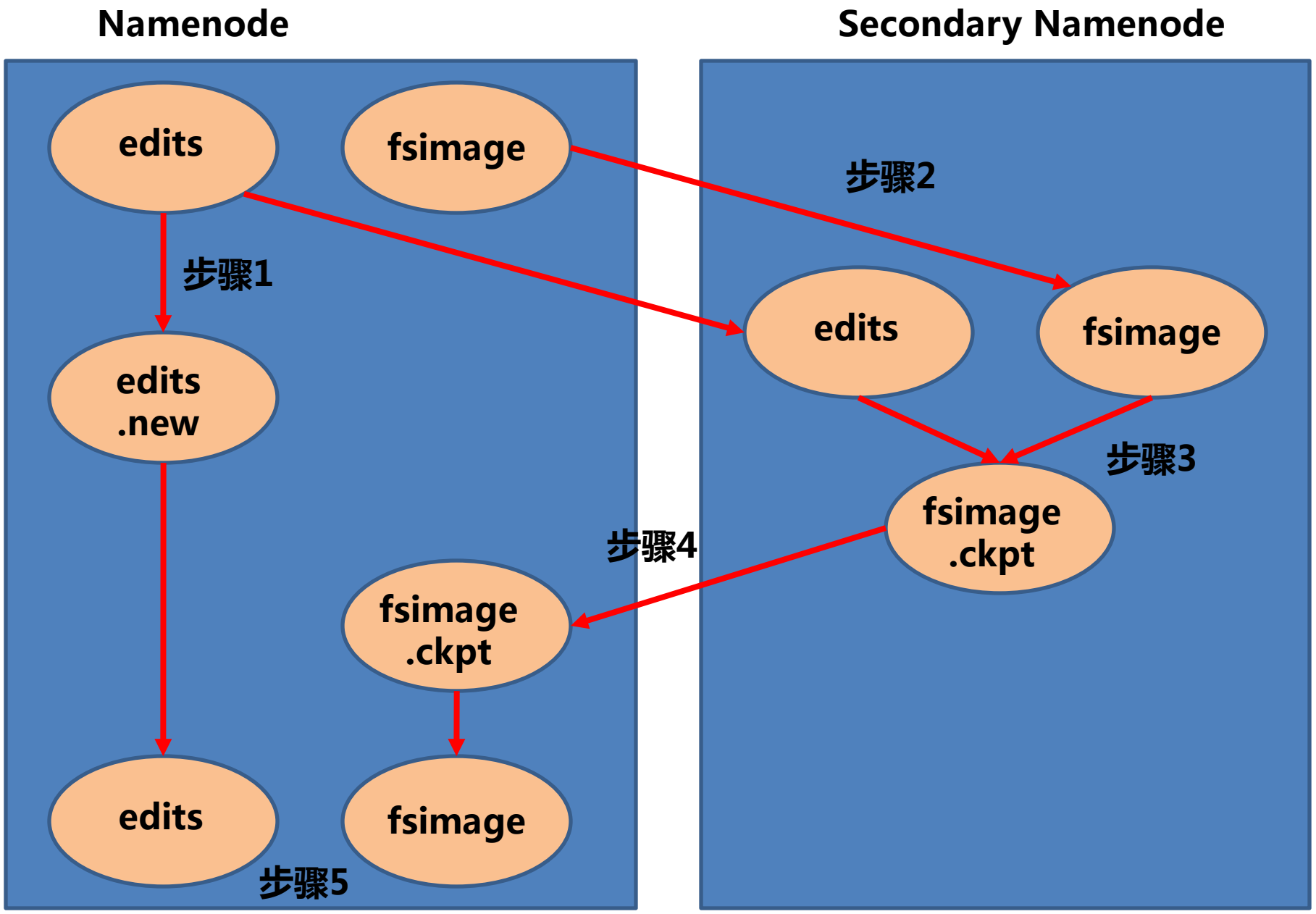
editlog记录了上一次checkpoint后的写操作

fsimage + editlog = 最新的元数据信息

HDFS 文件系统检查点的创建

- 随着用户不断地写操作，editlog会不断增长。虽然这对namenode运行没有影响，但将editlog中的操作与fsimage合并将会很慢（如namenode重新启动时），这段时间将使文件系统处于离线状态
- 解决方案是运行辅助namenode，为主namenode内存中的元数据创建检查点
  - 1 辅助namenode请求主namenode停止使用旧editlog文件，暂时将新的 写操作记录写到一个新editlog文件里
  - 2 辅助namenode从主namenode获取fsimage和editlog文件（采用HTTP GET）
  - 3 辅助namenode将fsimage载入内存，逐一执行editlog文件中的操作，创建新 的fsimage文件
  - 4 辅助namenode将新的 fsimage文件发送回主namenode（采用HTTP POST）
  - 5主namenode用从辅助namenode接收的fsimage文件替换旧的fsimage文件；用步骤1产生的新editlog文件替换旧的editlog文件；同时还更新fstime文件来记录检查点创建的时间
  - 最终，主namenode拥有最新的 fsimage文件和更小的editlog文件（该文件记录了上述过程中收到的新的用户写请求）

HDFS 文件系统检查点的创建

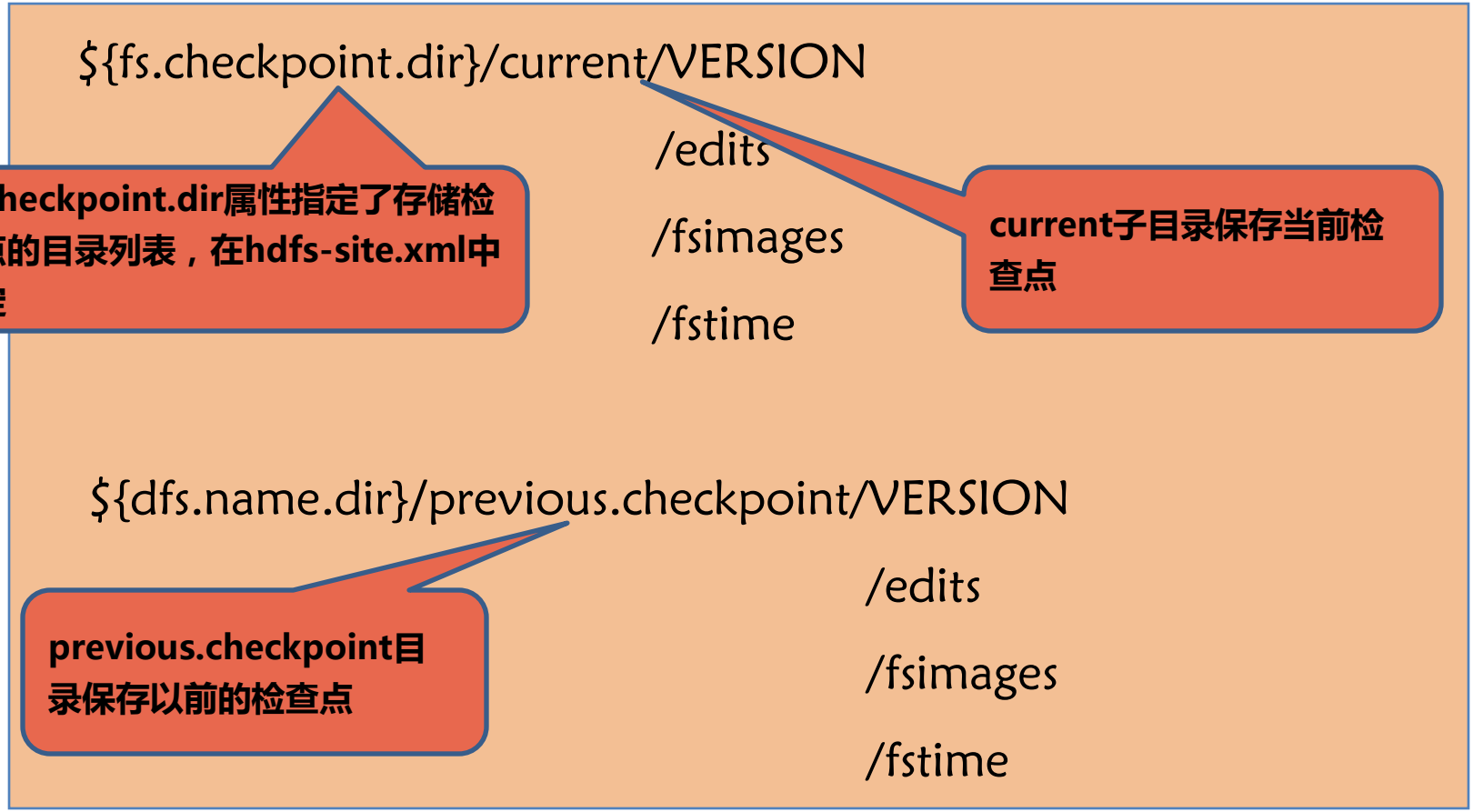


HDFS 文件系统创建检查点的触发条件

- 创建检查点的触发条件受两个配置参数控制
  - 通常情况下，辅助namenode每隔一个小时(通过fs.checkpoint.period属性设置，以秒为单位)创建检查点
  - 当编辑日志大小达到64MB(由fs.checkpoint.size属性设置，以字节为单位)创建检查点。系统每隔五分钟检查一次编辑日志大小

HDFS 文件系统的结构

- 辅助namenode文件系统结构



HDFS 文件系统的结构

- datanode文件系统结构
- datanode的文件目录是启动时自己创建的,不需要格式化

```
${dfs.data.dir}/current/VERSION
                       /blk_<id_1>
                       / blk_<id_1>.meta
                       …
                       /blk_<id_64>
                       /blk_<id_64>.meta
                       /subdir0/
                       …
                       /subdir63/
```

  - 除了VERSION文件外,其它文件都有blk前缀,包括二类:HDFS块元数据文件（.meta）和块数据文件，元数据文件包括头部（含版本、类型）和该块内各区段的一系列校验和
  - 当目录中数据块已经存储64个（通过dfs.datanode.numblocks）数据块时，就创建一个子目录，这样的好处是即使文件中的块非常多，也只需要访问少数几个目录级别就可以获取数据，避免很多文件放在一个目录中
  - 如果dfs.data.dir属性指定了存储数据的多个目录，数据块会以round-robin方式写到各个目录中

HDFS 文件系统的结构

- datenode的VERSION文件包含了以下信息
  - namespaceID 、 cTime 、 layoutVersion 与 namenode 的 值 相 同 ，namespaceID是datanode首次访问namenode时从namenode获得的
  - storageType：说明该存储目录包含的是datanode的数据结构
  - 各个datanode的storageID都不相同（但对于存储目录来说是相同的），namenode可以利用这个属性来识别datanode

```
#Tue Mar 10 21:32:31 GMT 2009
namespaceID=134368441
storageID=DS-547717739-172.16.85.1-50010-1236720751627
cTime=0
storageType=DATA_NODE
layoutVersion=-18
```

节点失效是常态

- DataNode中的磁盘挂了怎么办？
- DataNode所在机器挂了怎么办？
- NameNode挂了怎么办？

DataNode的磁盘挂了怎么办？

- DataNode正常服务
  - 坏掉的磁盘上的数据尽快通知NameNode

DataNode所在机器挂了怎么办？

- 问：NameNode怎么知道DataNode挂掉了？
- 答：datanode每3秒钟向namenode发送心跳，如果10分钟datanode没有向namenode发送心跳，则namenode认为该datanode已经dead，namenode将取出该datanode上对应的block，对其进行复制。

NameNode挂了怎么办？

- 持久化元数据
  - 操作日志（edit log）
    - 记录文件创建，删除，修改文件属性等操作
  - fsimage
    - 包含完整的命名空间
    - File -> Block的映射关系
    - 文件的属性（ACL, quota, 修改时间等）

NameNode挂了怎么办？

- Secondary NameNode
  - 将NameNode的fsimage与edit log从Namenode复制到临时目录
  - 将fsimage同edit log合并，并产生新的fsimage（减少启动时间）
  - 将产生的新的fsimage上传给NameNode
  - 清除NameNode中的edit log
  - [注]： Secondary NameNode仅仅对NameNode中元数据提供冷备方案

使用归档文件（Archive）

- HDFS存储小文件效率较低，因为一个文件存贮在一个block中，block的元数据由namenode维护在内存中，因此大量的小文件会吃掉namenode大量内存
  - 注意小文件不会占据更多的磁盘空间。例如1MB大小的文件用128MB的block来存储，只会占用1MB空间
- Hadoop归档文件，或HAR文件，是将多个文件高效打包到block中的归档方法，这样可以减少namenode消耗的内存同时仍然允许用户透明地访问这些文件

使用归档文件（Archive）

- 利用Hadoop的archive命令，可以将一个文档集合打包成归档文件(archive file)。这个命令实际上就是一个MapRedcue Job来并行地处理这些文件，因此需要一个Hadoop集群来运行这个命令
  - 假设需要归档的文件及目录如下

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

  - 现在运行archive命令

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

    - -archiveName选项指定归档文件名称 ,例子中为 files.har
    - -p 指定要归档目录的父目录/my
    - 第三个参数files为要归档的文件目录，用相对目录
    - 最后一个参数为har文件的输出目录

使用归档文件（Archive）

- 现在看看归档文件内容

```
% hadoop fs -ls /my
Found 2 items
drwxr-xr-x - tom supergroup 0 2009-04-09 19:13 /my/files
drwxr-xr-x - tom supergroup 0 2009-04-09 19:13 /my/files.har
```

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

- HAR文件由二个索引文件和part文件集合组成（这个例子里只有一个part文件），part文件里包含了被连接在一起的大量原始文件的内容，索引文件可以帮助我们找到每个被归档的原始文件（起始位置和长度）。

使用归档文件（Archive）

- 当使用har URI scheme访问HAR文件时，将会使用HAR文件系统（构建在底层文件系统之上，这里底层文件系统是HDFS），这些细节对应用都是透明的
- 以下命令可以递归地列出achive中的文件

```
% hadoop fs -lsr har:///my/files.har
drw-r--r-- -          tom supergroup 0 2009-04-09 19:13 /my/files.har/my
drw-r--r-- -          tom supergroup 0 2009-04-09 19:13 /my/files.har/my/files
-rw-r--r-- 10         tom supergroup 1 2009-04-09 19:13 /my/files.har/my/files/a
drw-r--r-- -          tom supergroup 0 2009-04-09 19:13 /my/files.har/my/files/dir
-rw-r--r-- 10         tom supergroup 1 2009-04-09 19:13 /my/files.har/my/files/dir/b
```

  - 上面的例子是使用HAR文件系统，因为URI Scheme是har

使用归档文件（Archive）

- 如果要删除HAR文件，需要使用递归格式进行删除，因为对基础文件系统来说，HAR文件是个目录

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