前言

本教程主要帮您解决以下几个问题：

- HDFS 是用来解决什么问题的？怎么解决的?
- 如何在命令行下操作 HDFS ?
- 如何使用 java api 来操作 HDFS ?
- 了解了基本思路和操作方法后，想知道 HDFS 读写数据的具体流程是怎么样的

学习并实践完成后，可以对 HDFS 有比较清晰的认识，并可以进行熟练操作，为后续学习 hadoop 体系打好基础

具体内容结构可以查看目录

1. HDFS 基本原理

HDFS（Hadoop Distribute File System）是一个分布式文件系统，是 Hadoop 的重要成员

文件系统的问题

文件系统是操作系统提供的磁盘空间管理服务，只需要我们指定把文件放到哪儿，从哪个路径读取文件句可以了，不用关心文件在磁盘上是如何存放的

当文件所需空间大于本机磁盘空间时，如何处理呢?

一是加磁盘，但加到一定程度就有限制了

二是加机器，用远程共享目录的方式提供网络化的存储，这种方式可以理解为分布式文件系统的雏形，可以把不同文件放入不同的机器中，空间不足了可以继续加机器，突破了存储空间的限制

但这个方式有多个问题

（1）单机负载可能极高

例如某个文件是热门，很多用户经常读取这个文件，就使此文件所在机器的访问压力极高

（2）数据不安全

如果某个文件所在的机器出现故障，这个文件就不能访问了，可靠性很差

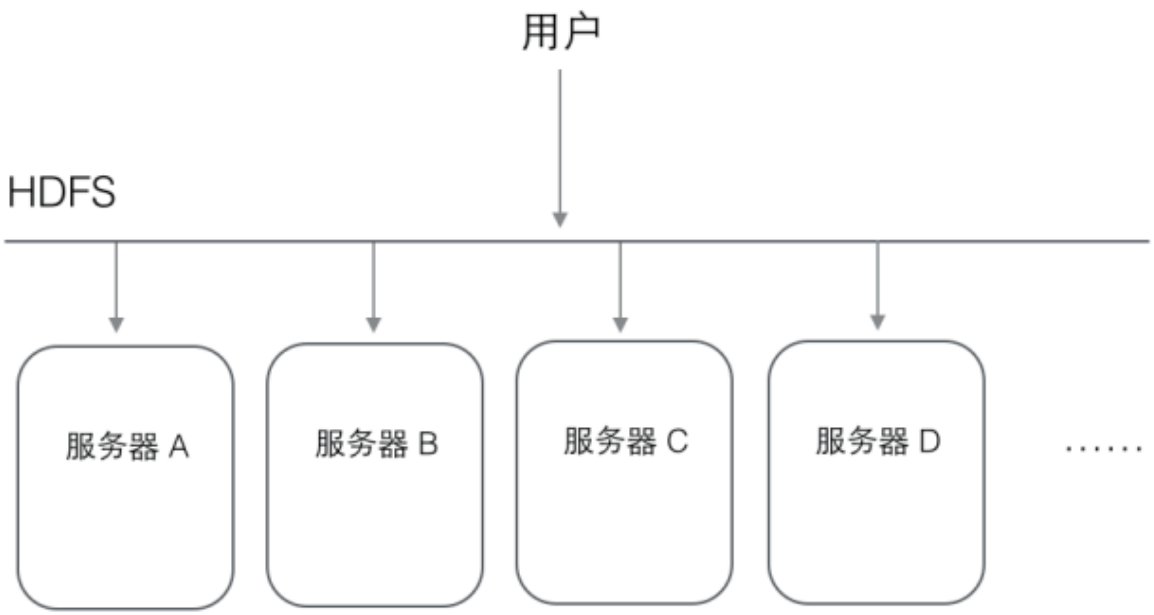
（3）文件整理困难

例如想把一些文件的存储位置进行调整，就需要看目标机器的空间是否够用，并且需要自己维护文件位置，如果机器非常多，操作就极为复杂

HDFS的解决思路

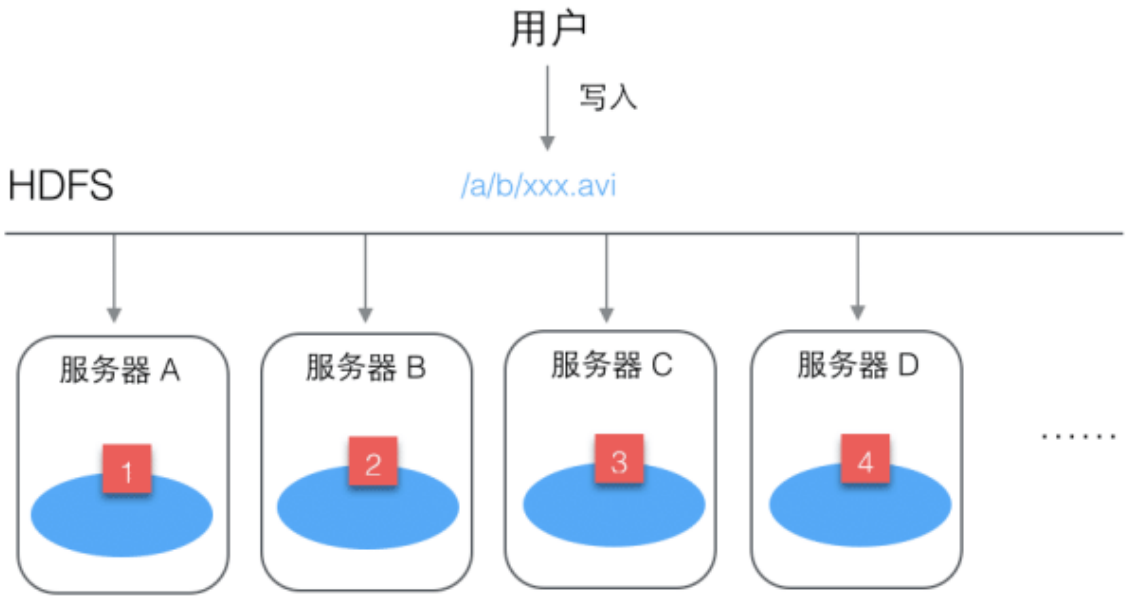
HDFS是个抽象层，底层依赖很多独立的服务器，对外提供统一的文件管理功能，对于用户来讲，感觉就像在操作一台机器，感受不到HDFS下面的多台服务器

例如用户访问HDFS中的 `/a/b/c.mpg` 这个文件，HDFS负责从底层相应服务器中读取，然后返回给用户，这样用户只需和HDFS打交道，不关心这个文件是怎么存储的



例如用户需要保存一个文件 `/a/b/xxx.avi`

HDFS首先会把这个文件进行分割，例如分为4块，然后分别放到不同服务器上

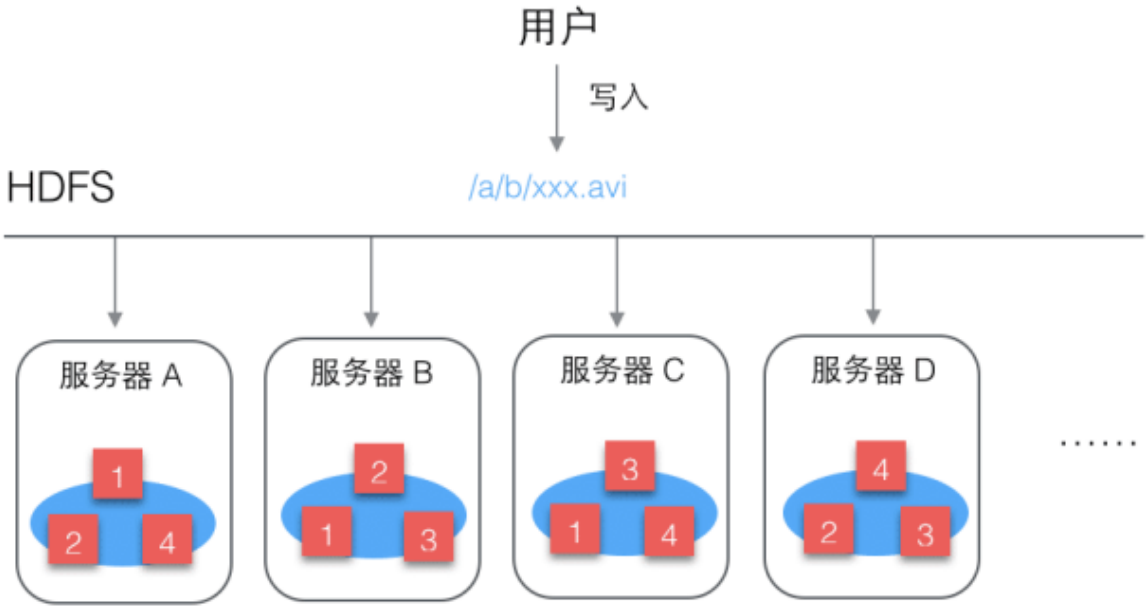


这样做有个好处，不怕文件太大，并且读文件的压力不会全都集中在一台服务器上

但如果某台服务器坏了，文件就读不全了

HDFS为保证文件可靠性，会把每个文件块进行多个备份

- 块1 : A B C
- 块2 : A B D
- 块3 : B C D
- 块4 : A C D



这样文件的可靠性就大大增强了，即使某个服务器坏了，也可以完整读取文件

同时还带来一个很大的好处，就是增加了文件的并发访问能力，比如多个用户读取这个文件时，都要读块1，HDFS可以根据服务器的繁忙程度，选择从哪台服务器读块1

元数据的管理

HDFS中存了哪些文件?

文件被分成了哪些块?

每个块被放在哪台服务器上?

……

这些都叫做元数据，这些元数据被抽象为一个目录树，记录了这些复杂的对应关系

这些元数据由一个单独的模块进行管理，这个模块叫做 `NameNode`

存放文件块的真实服务器叫做 `DataNode`

所以用户访问HDFS的过程可以理解为：

> 用户 -> HDFS -> NameNode -> DataNode

HDFS 优点

（1）容量可以线性扩展

（2）有副本机制，存储可靠性高，吞吐量增大

（3）有了NameNode后，用户访问文件只需指定HDFS上的路径

2. HDFS 实践

经过上面介绍，可以对 HDFS 有个基本的了解，下面开始进行实际操作，在实践中更好的认识 HDFS

2.1 安装实践环境

您可以选择`自己搭建环境`，也可以使用`打包好的 hadoop 环境`（版本 2.7.3）

这个hadoop环境实际上是一个虚机镜像，所以需要安装 virtualbox 虚拟机、vagrant 镜像管理工具，和我做的 hadoop 镜像，然后用这个镜像启动虚机就可以了，下面是具体操作步骤：

1）安装 virtualbox

下载地址

```
https://www.virtualbox.org/wiki/Downloads
```

2）安装 vagrant

因为官网下载较慢，我上传到了云盘

windows版

```
链接: https://pan.baidu.com/s/1pKKQGHl 密码: eykr
```

Mac版

```
链接: https://pan.baidu.com/s/1slts9yt 密码: aig4
```

安装完成后，在命令行终端下就可以使用 vagrant 命令

3）下载 hadoop 镜像

```
链接: https://pan.baidu.com/s/1bpaisnd 密码: pn6c
```

4）启动

加载 hadoop 镜像

```
vagrant box add {自定义镜像名称} {镜像所在路径}
```

例如您想命名为`hadoop`，镜像下载后的路径为`d:\hadoop.box`，加载命令就是这样：

```
vagrant box add hadoop d:\hadoop.box
```

创建工作目录，例如 `d:\hdfstest`

进入此目录，初始化

```
cd d:\hdfstest
vagrant init hadoop
```

启动虚机

```
vagrant up
```

启动完成后，就可以使用SSH客户端登录虚机了

```
IP  127.0.0.1
端口 2222
用户名 root
密码 vagrant
```

登录后使用命令 `ifconfig` 查看本虚机的IP（如 192.168.31.239），可以使用此IP和端口22登录了

```
IP  192.168.31.239
端口 22
用户名 root
密码 vagrant
```

hadoop 服务器环境搭建完成

2.2 Shell 命令行操作

登录 hadoop 服务器后，先启动 hdfs，执行命令：

```
start-dfs.sh
```

- 查看帮助

```
hdfs dfs –help
```

- 显示目录信息

```
hdfs dfs –ls /
```

-ls 后面是要查看的目录路径

- 创建目录

创建目录 `/test`

```
hdfs dfs –mkdir /test
```

一次创建多级目录 `/aa/bb`

```
hdfs dfs –mkdir –p /aa/bb
```

- 上传文件

形式

```
hdfs dfs –put {本地路径} {hdfs中的路径}
```

示例（先创建好一个测试文件 mytest.txt，内容随意，然后上传到 /test）

```
hadoop  fs  –put  ~/mytest.txt  /test
```

- 显示文件内容

```
hdfs dfs –cat /test/mytest.txt
```

- 下载文件

```
hdfs dfs –get /test/mytest.txt ./mytest2.txt
```

- 合并下载

先创建2个测试文件（log.access, log.error），内容随意，使用 `-put` 上传到 `/test` 目录下

```
hdfs dfs -put log.* /test
```

然后把2个log文件合并下载到一个文件中

```
hdfs dfs -getmerge /test/log.* ./log
```

查看本地 log 文件内容，应该包含 log.access 与 log.error 两个文件的内容

- 复制

从hdfs的一个路径拷贝hdfs的另一个路径

```
hdfs dfs -cp /test/mytest.txt /aa/mytest.txt.2
```

验证

```
hdfs dfs -ls /aa
```

- 移动文件

```
hdfs dfs -mv /aa/mytest.txt.2 /aa/bb
```

验证

```
hdfs dfs -ls /aa/bb
```

应列出 mytest.txt.2

- 删除

```
hdfs dfs -rm -r /aa/bb/mytest.txt.2
```

使用 **-r** 参数可以一次删除多级目录

验证

```
hdfs dfs -ls /aa/bb
```

应为空

- 修改文件权限

与linux文件系统中的用法一样，修改文件所属权限

```
-chgrp
-chmod
-chown
```

示例

```
hdfs dfs -chmod 666 /test/mytest.txt
hdfs dfs -chown someuser:somegrp /test/mytest.txt
```

- 统计文件系统的可用空间

```
hdfs dfs -df -h /
```

- 统计文件夹的大小

```
hdfs dfs -du -s -h /test
```

2.3 Java api 操作

2.3.1 环境配置

因为需要在本机连接 hadoop 虚机服务器，所以需要配置 hadoop，使其可以被外部访问

先登录 hadoop 虚机服务器，然后：

1）查看本机IP

```
ip address
```

例如IP为：192.168.31.239

2）修改文件：

```
vi /usr/local/hadoop-2.7.3/etc/hadoop/core-site.xml
```

找到下面内容

```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
</property>
```

把其中的 `localhost:9000` 修改为本机IP `192.168.31.239:9000`

3）重新启动 hdfs

```
# 停止
stop-dfs.sh
# 启动
start-dfs.sh
```

2.3.2 搭建开发环境

1）新建项目目录 `hdfstest`

2）在项目目录下创建 `pom.xml`

内容：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```
  <groupId>demo.hdfs</groupId>
  <artifactId>hdfstest</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>hdfstest</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEn
coding>
  </properties>

  <dependencies>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.5.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.5.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-client</artifactId>
            <version>2.5.1</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

3）创建源码目录 `src/main/java`

现在项目目录结构

```
├── pom.xml
├── src
│   └── main
```

```
|       └── java
```

2.3.3 示例代码

2.3.3.1 查看文件列表 ls

1）新建文件 `src/main/java/Ls.java`

> 列出 / 下的文件列表，及递归获取所有文件

```
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.BlockLocation;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.LocatedFileStatus;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.RemoteIterator;

public class Ls {
    public static void main(String[] args) throws Exception {
        String uri = "hdfs://192.168.31.239:9000/"; // 根据自己环
境修改
        Configuration config = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), config,
 "root");

        // 方法1：直接获取
        FileStatus[] listStatus = fs.listStatus(new Path("/"));
        for (FileStatus file : listStatus) {
            System.out.println("[" + (file.isFile() ? "file" :
"dir") + "]  " + file.getPath().getName());
        }

        // 方法2：使用迭代器，并递归获取所有子文件夹下的文件
        RemoteIterator<LocatedFileStatus> listFiles = fs.listFi
les(new Path("/"), true);

        while (listFiles.hasNext()) {
            System.out.println("============================");
            LocatedFileStatus fileStatus = listFiles.next();
            System.out.println("块大小 :" + fileStatus.getBlockS
ize());
            System.out.println("所属  :" + fileStatus.getOwner()
```

```
);
            System.out.println("备份数 :" + fileStatus.getReplic
ation());
            System.out.println("权限  :" + fileStatus.getPermiss
ion());
            System.out.println("名称  :" + fileStatus.getPath().
getName());
            System.out.println("----------块信息----------");
            BlockLocation[] blockLocations = fileStatus.getBloc
kLocations();
            for (BlockLocation b : blockLocations) {
                System.out.println("块起始偏移量  : " + b.getOffs
et());
                System.out.println("块长度     :" + b.getLength(
));
                // 块所在的datanode节点
                String[] datanodes = b.getHosts();
                for (String dn : datanodes) {
                    System.out.println("datanode    :" + dn);
                }
            }

        }
    }
}
```

2）编译执行

```
mvn compile
mvn exec:java -Dexec.mainClass="Ls" -Dexec.cleanupDaemonThreads
=false
```

2.3.3.2 创建目录 mkdir

> 在 hdfs 中创建目录 /mkdir/a/b

1）新建文件 `src/main/java/Mkdir.java`

```
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
```

```
public class Mkdir {
    public static void main(String[] args) throws Exception {
        String uri = "hdfs://192.168.31.239:9000/"; // 根据自己环
境修改
        Configuration config = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), config,
 "root");
        boolean mkdirs = fs.mkdirs(new Path("/mkdir/a/b"));
        System.out.println("【创建目录结果】"+mkdirs);
    }
}
```

2）编译执行

```
mvn compile
mvn exec:java -Dexec.mainClass="Mkdir" -Dexec.cleanupDaemonThre
ads=false
```

3）在服务器中使用 hdfs 命令验证

```
hdfs dfs -ls /mkdir
```

2.3.3.3 上传文件 put

> 在当前项目目录下新建测试文件，上传到 hdfs 中的 /mkdir

1）在项目目录下创建测试文件 `testfile.txt`，内容随意

2）新建文件 `src/main/java/Put.java`

```
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class Put {
    public static void main(String[] args) throws Exception {
        String uri = "hdfs://192.168.31.239:9000/"; // 根据自己环
境修改
        Configuration config = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), config,
```

```
 "root");
        String localpath = "testfile.txt";
        fs.copyFromLocalFile(new Path(localpath), new Path("/mk
dir"));
        fs.close();
    }
}
```

3）编译执行

```
mvn compile
mvn exec:java -Dexec.mainClass="Put" -Dexec.cleanupDaemonThread
s=false
```

4）在服务器中使用 hdfs 命令验证

```
hdfs dfs -ls /mkdir
hdfs dfs -cat /mkdir/testfile.txt
```

2.3.3.4 下载文件 get

1）新建文件 `src/main/java/Get.java`

> 把 hdfs 中 /mkdir/testfile.txt 下载到当前项目目录下

```
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class Get {
    public static void main(String[] args) throws Exception {
        String uri = "hdfs://192.168.31.239:9000/"; // 根据自己环
境修改
        Configuration config = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), config,
 "root");
        String localpath = "testfile2.txt";
        fs.copyToLocalFile(new Path("/mkdir/testfile.txt"), new
 Path(localpath));
    }
```

```
}
```

2）编译执行

```
mvn compile
mvn exec:java -Dexec.mainClass="Get" -Dexec.cleanupDaemonThread
s=false
```

3）查看项目目录下是否存在 testfile2.txt 及其内容

2.3.3.5 删除文件 delete

> 删除 hdfs 上之前上传的 /mkdir/testfile.txt

1）新建文件 `src/main/java/Del.java`

```
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class Del {
    public static void main(String[] args) throws Exception {
        String uri = "hdfs://192.168.31.239:9000/"; // 根据自己环
境修改
        Configuration config = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), config,
 "root");
        boolean ret = fs.delete(new Path("/mkdir/testfile.txt")
, true);
        System.out.println("【删除结果】"+ret);
    }
}
```

2）编译执行

```
mvn compile
mvn exec:java -Dexec.mainClass="Del" -Dexec.cleanupDaemonThread
s=false
```

3）在服务器中使用 hdfs 命令验证，检查 testfile.txt 是否被删除

```
hdfs dfs -ls /mkdir
```

2.3.3.6 重命名 rename

> 把 hdfs 中的 /mkdir/a 重命名为 /mkdir/a2

1）新建文件 `src/main/java/Rename.java`

```
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class Rename {
    public static void main(String[] args) throws Exception {
        String uri = "hdfs://192.168.31.239:9000/"; // 根据自己环
境修改
        Configuration config = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), config,
 "root");
        fs.rename(new Path("/mkdir/a"), new Path("/mkdir/a2"));
    }
}
```

2）编译执行

```
mvn compile
mvn exec:java -Dexec.mainClass="Rename" -Dexec.cleanupDaemonThr
eads=false
```

3）在服务器中使用 hdfs 命令验证

```
hdfs dfs -ls /mkdir
```

2.3.3.7 流方式读取文件部分内容

> 上传一个文本文件，然后使用流方式读取部分内容保存到当前项目目录

1）在服务器中创建一个测试文件 test.txt，内容：

```
123456789abcdefghijklmn
```

上传到 hdfs

```
hdfs dfs -put test.txt /
```

2）在本地项目中新建文件 `src/main/java/StreamGet.java`

```
import java.io.FileOutputStream;
import java.net.URI;

import org.apache.commons.io.IOUtils;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class StreamGet {
    public static void main(String[] args) throws Exception {
        String uri = "hdfs://192.168.31.239:9000/"; // 根据自己环
境修改
        Configuration config = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), config,
 "root");

        FSDataInputStream inputStream = fs.open(new Path("/test
.txt"));
        inputStream.seek(5); //指定读取的开始位置
        FileOutputStream outputStream = new FileOutputStream("t
est.txt.part2");
        IOUtils.copy(inputStream, outputStream);
    }
}
```

2）编译执行

```
mvn compile
mvn exec:java -Dexec.mainClass="StreamGet" -Dexec.cleanupDaemon
Threads=false
```

3) 执行后查看项目目录下的 test.txt.part2，内容应为：

```
6789abcdefghijklmn
```

前面的 `12345` 已经被略过

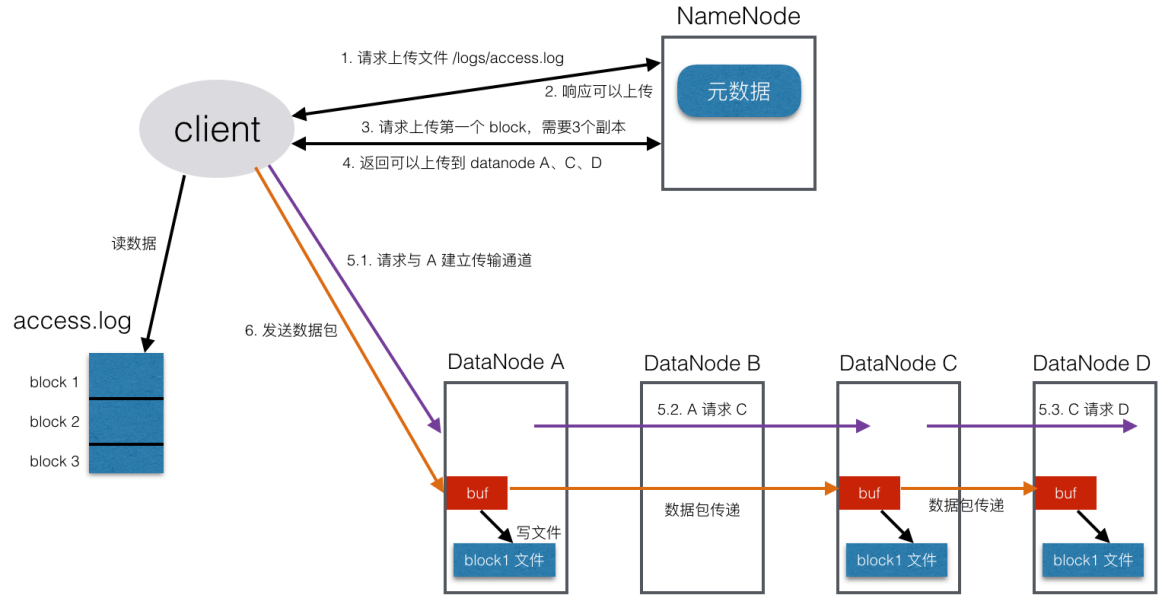
3. 深入了解

3.1 写入机制

向 HDFS 中写入文件时，是按照块儿为单位的，client 会根据配置中设置的块儿的大小把目标文件切为多块，例如文件是300M，配置中块大小值为128M，那么就分为3块儿

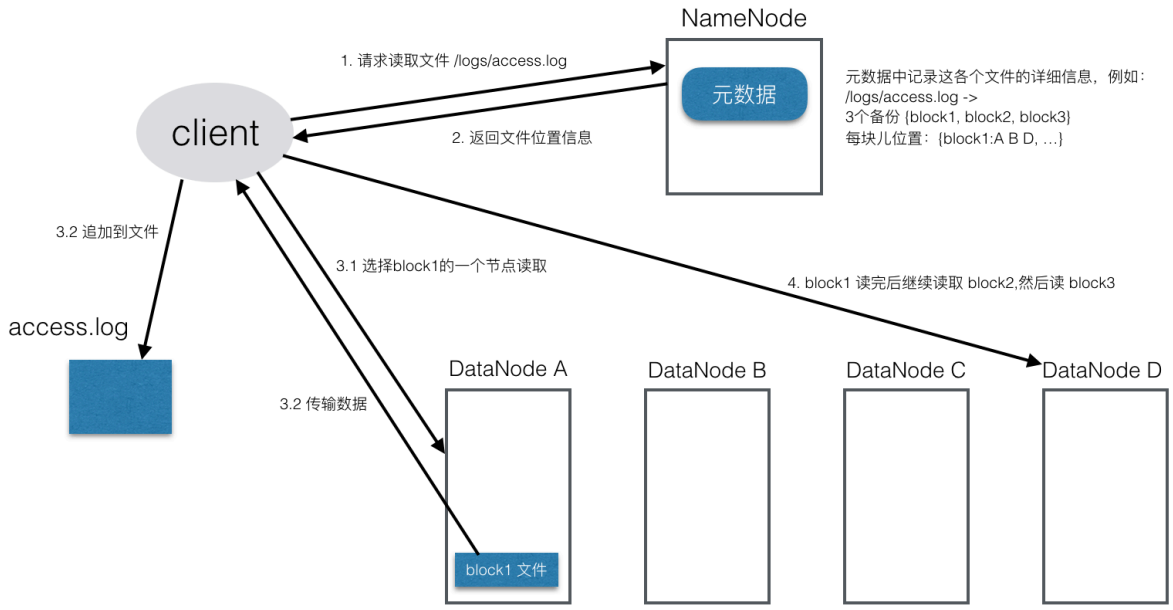
具体写入流程：

1. client 向 namenode 发请求，说想要上传文件
2. namenode 会检查目标文件是否存在、父目录是否存在，检查没有问题后返回确认信息
3. client 再发请求，问第一个 block 应该传到哪些 datanode 上
4. namenode 经过衡量，返回 3 个可用的 datanode（A,B,C）
5. client 与 A 建立连接，A 与 B 建立连接，B 与 C 建立连接，形成一个 pipeline
6. 传输管道建立完成后，client 开始向 A 发送数据包，此数据包会经过管道依次传递到 B 和 C
7. 当第一个 block 的数据都传完以后，client 再向 namenode 请求第二个 block 上传到哪些 datenode，然后建立传输管道发送数据
8. 就这样，直到 client 把文件全部上传完成



3.2 读取机制

1. client 把要读取的文件路径发给 namenode，查询元数据，找到文件块所在的datanode服务器
2. client 直到了文件包含哪几块儿、每一块儿在哪些 datanode 上，就选择那些离自己近的 datanode（在同一机房，如果有多个离着近的，就随机选择），请求建立socket流
3. 从 datanode 获取数据
4. client 接收数据包，先本地缓存，然后写入目标文件
5. 直到文件读取完成



3.3 NameNode 机制

通过对hdfs读写流程的了解，可以发现 namenode 是一个很重要的部分，它记录着整个hdfs系统的元数据，这些元数据是需要持久化的，要保存到文件中

namenode 还要承受巨大的访问量，client 读写文件时都需要请求 namenode，写文件时要修改元数据，读文件时要查询元数据

为了提高效率，namenode 便将元数据加载到内存中，每次修改时，直接修改内存，而不是直接修改文件，同时会记录下操作日志，供后期修改文件时使用

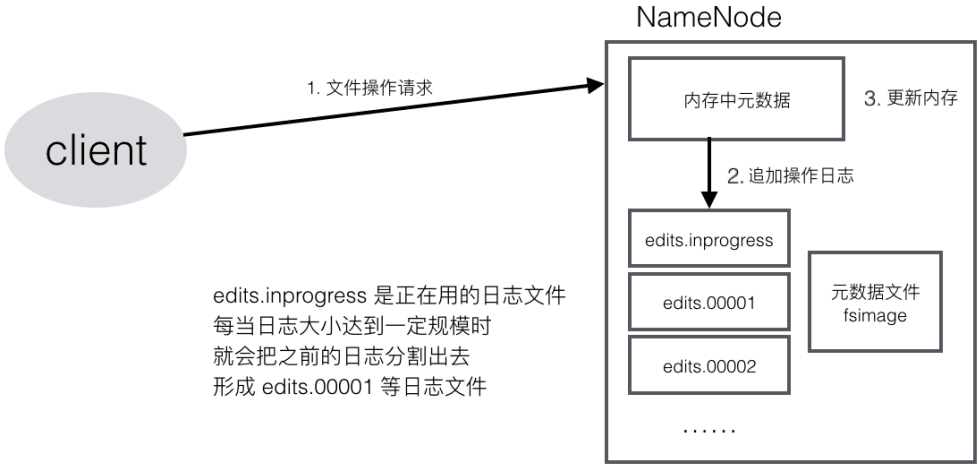
这样，namenode 对数据的管理就涉及到了3种存储形式：

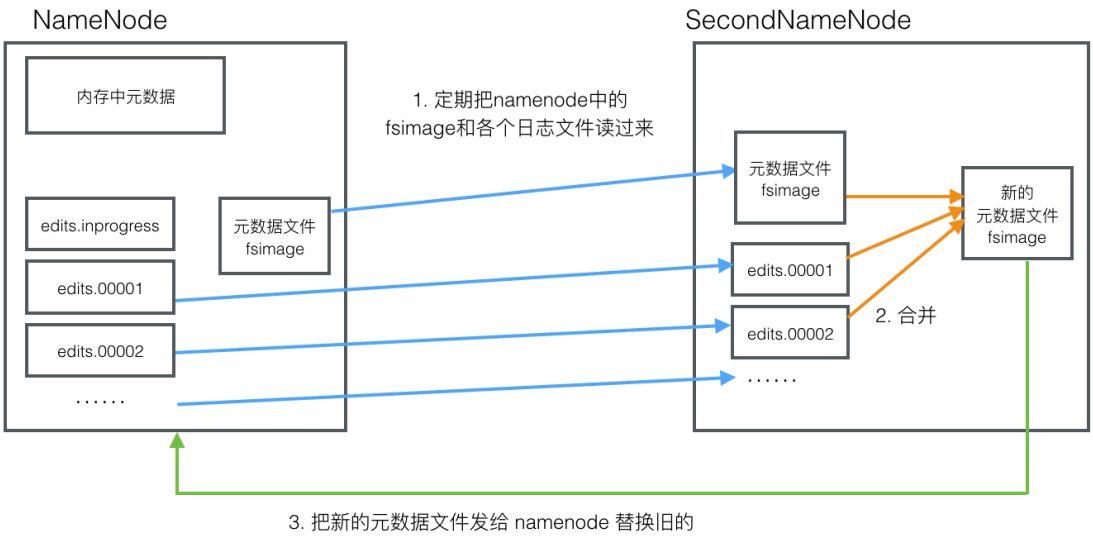
1. 内存数据
2. 元数据文件
3. 操作日志文件

namenode 需要定期对元数据文件和日志文件进行整合，以保证文件中数据是新的，但这个过程很消耗性能，namenode 需要快速的响应 client 的大量请求，很难去完成文件整合操作，这时就引入了一个小助手 `secondnamenode`

secondnamenode 会定期从 namenode 中下载元数据文件和操作日志，进行整合，形成新的数据文件，然后传回 namenode，并替换掉之前的旧文件

secondnamenode 是 namenode 的好帮手，替 namenode 完成了这个重体力活儿，并且还可以作为 namenode 的一个防灾备份，当 namenode 数据丢失时，secondnamenode 上有最近一次整理好的数据文件，可以传给 namenode 进行加载，这样可以保证最少的数据丢失





4. 小结

HDFS 的基础内容介绍完了，希望可以帮助您快速熟悉 HDFS 的思路和使用方式

本内容出自公众号 `性能与架构` （yogoup），如有批评与建议（例如 内容有误、感觉有什么不足的地方、改进建议等），欢迎发送消息