2019

Hadoop 文件系统分析报告

软件体系架构 HADOOP 项目文件系统源代码分析

葛临雪 黄佳乐 李佳哲

目录

1. Hadoop 概述

Hadoop 是由 Apache 基金会所开发的分布式系统基础架构。它使用户在不了解分布式底层细节的情况下开发分布式程序，充分利用 Hadoop 集群进行高速运算和存储

Hadoop 的出现解决了大数据领域的两大问题：大数据存储和大数据分析，也就是 Hadoop 的两大核心：HDFS 和 MapReduce。

（1） HDFS(Hadoop Distributed File System) 是可扩展、高容错性、高性能的分布式文件系统。

（2） MapReduce 是实现了基于集群的高性能并行计算的分布式计算框架

下图展示了大数据处理领域的技术架构，Hadoop 的 HDFS 负责文件存储层的分布式文件存储，MapReduce 则负责编程模型层的编程模型：

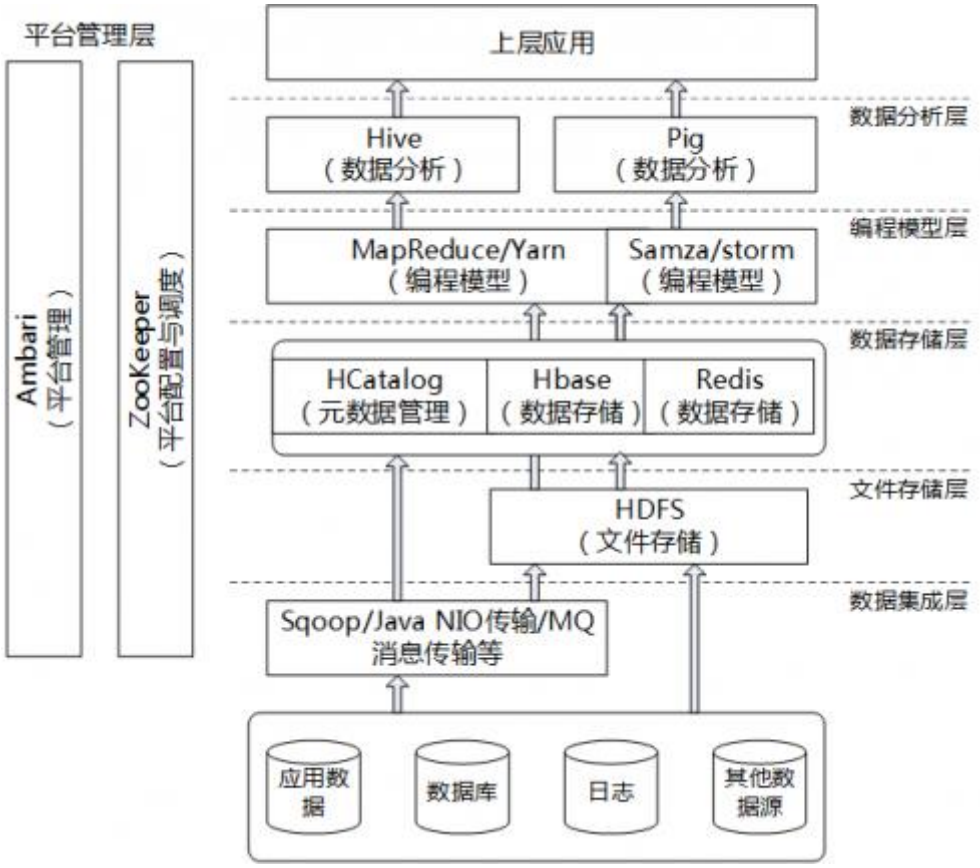

图 1-1 Hadoop 技术架构

接下里，我们以 Hadoop-2.9.2 为例，对 Hadoop 项目文件系统的源代码进行分析，给出源代码的详细类图和核心过程分析，并分析 Hadoop 文件系统的软件体系架构。

2. Hadoop 文件系统概述

Hadoop 中提供了抽象文件系统的概念，支持多个文件系统，HDFS 只是其中的一个实现。Hadoop 抽象文件系统，从某种角度上来说，扮演着 Linux 中 VFS (Virtual File System 虚拟文件系统) 的角色。这种面向接口的设计模式使得 Hadoop 的文件系统抽象程度高，扩展性很强。

在 Hadoop 中，org.apache.hadoop.fs 包提供了一个抽象文件系统的 API。该包下有 50 多个类：

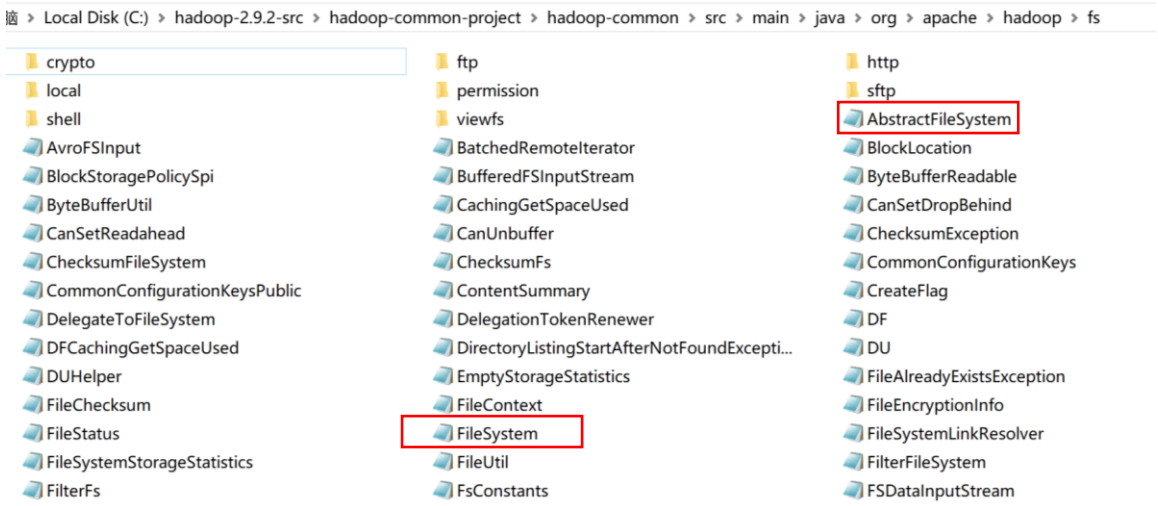

图 2-1 fs 包总览

其中包括 FileSytem 抽象类和 AbstractFileSyetem 抽象类作为抽象文件系统的的基类，提供了基本的抽象操作。其中 FileSystem 类是 0.21 版本之前唯一的基类，但在 0.21 中，出现了 AbstractFileSystem，该类似乎用来取代 FileSystem 的部分功能。

FileSystem 抽象类定义了一组分布式文件系统和通用的 IO 组件接口，hdfs、ftp、kfs、s3 等都是通过基类实现的具体的文件系统。

2.1 类层次结构

Hadoop 的文件抽象类从不同的文件系统中抽取了共同的操作，如打开文件，创建文件，删除文件，获取文件信息等。而 FileSystem 抽象类和 AbstractFileSystem 抽象类则定义了这些基本功能，然后通过派生，实现各个不同的文件系统。由此，我们可以得到两个类继承的层次结构，如图 2-1 和图 2-2 所示：

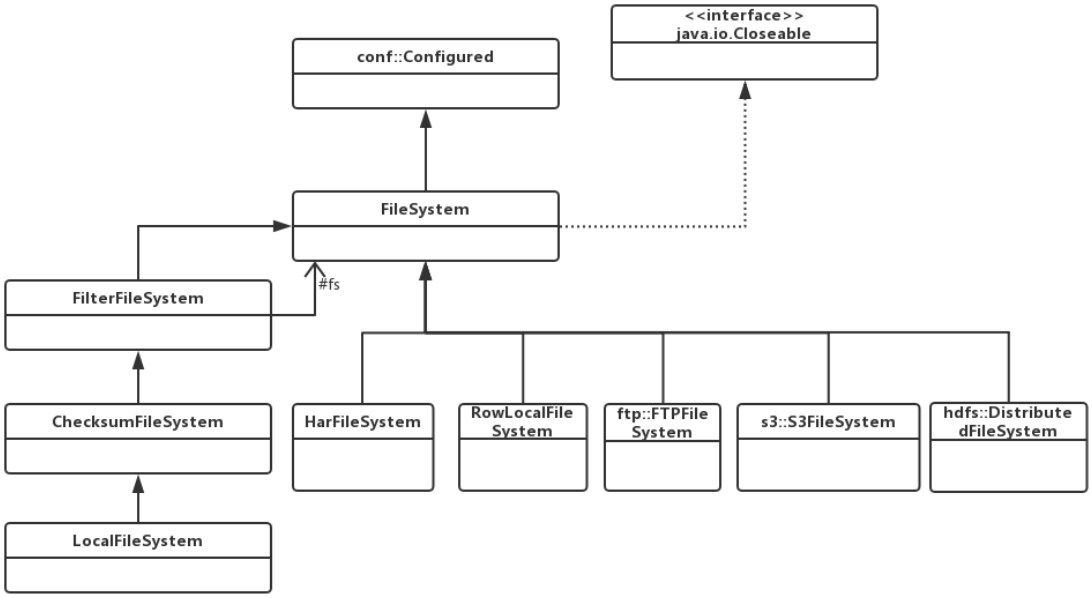

图 2-2 FileSystem 抽象类

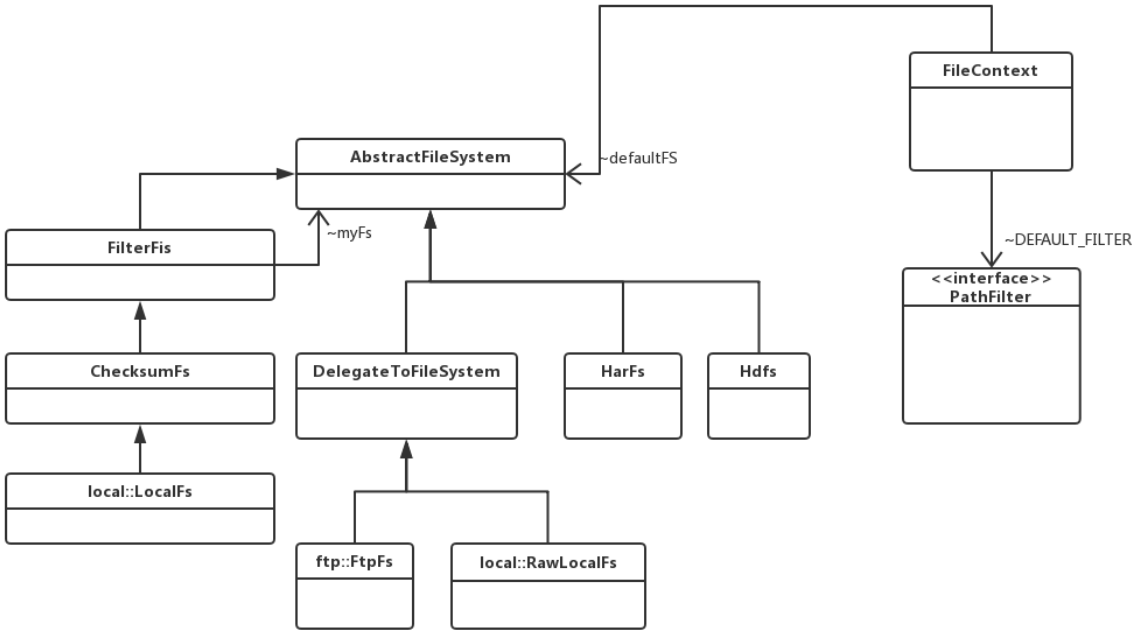

图 2-3 AbstractFileSystem 抽象类

从类的继承层次图中，我们发现这两个类层次结构十分相似。

除 FilterFileSystem 之外，FileSystem 的直接子类都是具体的文件系统。它们的具体描述如表 2-1：

表 2-1 Hadoop 具体文件系统

文件系统	Java 实现	描述
Local	fs.LocalFileSystem	支持有客户端校验和本地文件系统
HDFS	hdfs.DistributionFileSystem	Hadoop 的分布式文件系统
HFTP	hdfs.HftpFileSystem	支持通过 HTTP 的方式以**只读**的方式访问 HDFS
HAR	fs.HarFileSystem	在 Hadoop 文件系统上，对文件进行归档
FTP	fs.ftp.FtpFileSystem	由 FTP 服务器支持的文件系统
s3(local)	fs.s3native.NativeS3FileSystem	基于 Amazon 的文件系统
s3(block)	fs.s3.NativeS3FileSystem	基于 Amazon 的文件系统，以块格式存储

在平时的使用中，Hadoop 的使用者可以分为两类：**应用程序编写者**及**文件系统实现者**，在 Hadoop-0.21 之前，FileSystem 抽象类一般作为文件系统的基类，一方面向应用程序编写者提供使用 Hadoop 文件系统的接口；一方面向文件系统实现者提供一个文件系统的接口。但在 Hadoop-0.21 版本之后，出现了 FileContext 抽象类和 AbstractFileSystem 抽象类，通过这两个抽象类，可以将原本集中在 FileSystem 类中的功能区分来，使各个类之间耦合度降低，让使用者更方便的在应用程序中使用多个文件系统。FileContext 类用于向应用程序编写提供使用 Hadoop 文件系统的接口，而 FileSystem 类则由文件系统实现者使用。在当前版本下，AbstractFileSystem 还没有取代 FileSystem 对文件系统实现者提供接口。

2.2 抽象文件系统 FileSystem

对 FileSystem 进行分析，其中定义了许多关于处理文件和目录相关事务的函数，对相关操作函数总结如表 2-2：

表 2-2 FileSystem 抽象文件系统操作

Hadoop 的 FileSystem	Java 操作	描述
FileSystem.open FileSystem.create FileSystem.append	URL.openStream	打开一个文件
FileSystem.getFileStatus FileSystem.get*	File.get*	获取文件/目录的属性
FileSystem.set*	File.set*	改变文件的属性
FileSystem.createNewFile	File.createNewFile	创建一个文件
FileSystem.delete	File.delete	从文件系统中删除一个文件
FileSystem.rename	File.renameTo	更改文件/目录名
FileSystem.mkdirs	File.mkdir	在给定目录下创建一个子目录
FileSystem.delete	File.delete	从一个目录中删除一个空的子目录
FileSystem.listStatus	File.list	读取一个目录下的项目
FileSystem.getWorkingDirectory		返回当前工作目录
FileSystem.setWorkingDirectory		更改当前工作目录
FSDataInputStream.read	InputSteam.read	读取文件中的数据
FSDataOutputStream.write	OutputSteam.write	向文件写入数据
FSDataInputStream.close FSDataOutputStream.close	InputSteam.close OutputSteam.close	关闭一个文件
FSDataInputStream.seek	RandomAccessFile.seek	改变文件读写位置

通过 FileSystem.getFileStatus()方法，Hadoop 抽象文件系统可以一次获得文件/目录的所有属性，这些属性被保存在 FileStatus 类中：

```
34 public class FileStatus implements Writable, Comparable<FileStatus> {
35
36   private Path path;
37   private long length;
38   private boolean isdir;
39   private short block_replication;
40   private long blocksize;
41   private long modification_time;
42   private long access_time;
43   private FsPermission permission;
44   private String owner;
45   private String group;
46   private Path symlink;
```

图 2-4 FileStatus 类代码片段

FileStatus 类保存了文件路径、文件长度、是否为目录、文件副本数（HDFS）、块大小（HDFS）、修改时间、访问时间、许可信息、文件所有者等一系列关于文件/目录的属性。同样，FileStatus 类实现了 Writable 接口，这说明 FileStatus 可以被序列化后在网络上传输，同时一次性将文件的所有属性读出并返回到客户端，这样可以减少在分布式系统中进行网络传输的次数。

对于一个具体的文件系统而言，它继承了 FileSystem 抽象类，为了使文件系统生效，具体的文件系统需要实现一些抽象方法。对 FileSystem 中的抽象方法总结如表 2-3。

表 2-3 FileSystem 抽象方法汇总

抽象方法	描述
public abstract URI getUri();	获取文件系统 URI
public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException;	为读打开一个文件，并返回一个输入流

public abstract FSDataOutputStream create(Path f, FsPermission permission, boolean overwrite, int bufferSize, short replication,long blockSize, Progressable progress) throws IOException;	创建一个文件，并返回一个输出流
public abstract FSDataOutputStream append(Path f, int bufferSize, Progressable progress) throws IOException;	在一个已经存在的文件中追加数据
public abstract Boolean rename(Path src, Path dst) throws IOException;	修改文件名或目录名
public abstract Boolean delete(Path f) throws IOException; public abstract Boolean delete(Path f, boolean recursive) throws IOException;	删除文件
public abstract FileStatus[] ListStatus(Path f) throws FileNotFoundException, IOException;	如果 Path 是一个目录，读取一个目录下所有的项目和项目属性；如果 Path 是一个文件，获取文件属性
public abstract void setWorkingDirectory(Path new_dir);	设置当前工作目录
public abstract Path getWorkingDirectory ();	获取当前工作目录
public abstract Boolean mkdirs(Path f, FsPermission permission) throw IOException;	创建一个目录
public abstract FileStatus getFileStatus(Path f) throws IOException;	获取文件或目录的属性

2.3 IO 输入输出流

Hadoop 抽象文件系统和 Java 类似，都使用流机制（Stream）进行文件的读写。

用于读写数据流的抽象类分别是：FSDataInputStream 和 FSDataOutputStream，两个抽象类的继承结构如图 2-5 和图 2-6：

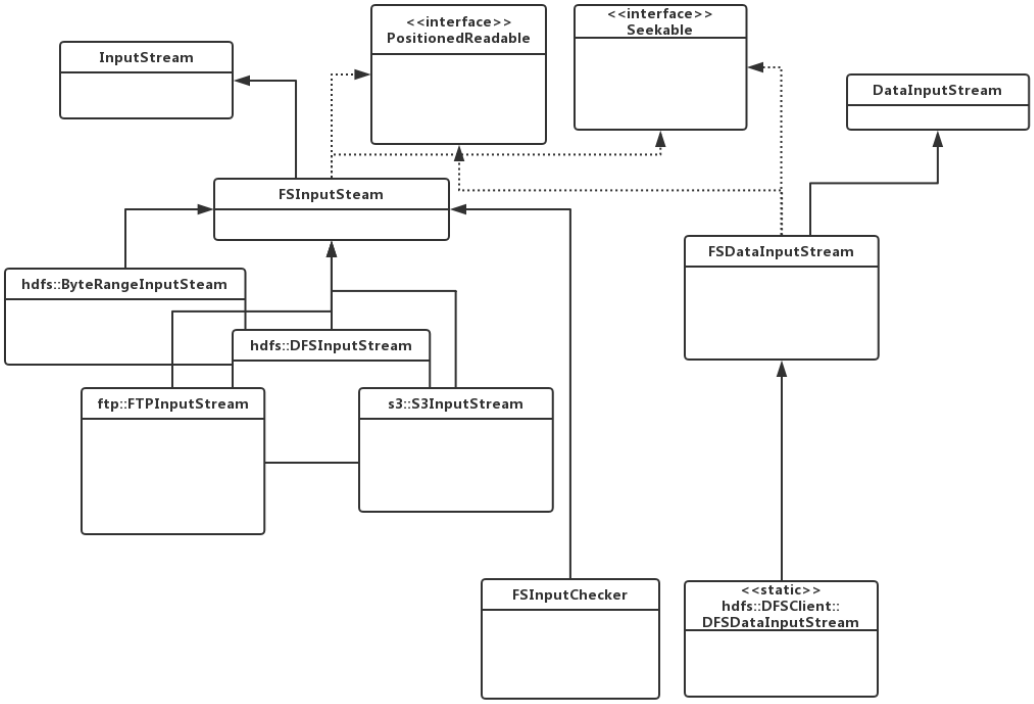

图 2-5 输入流类图

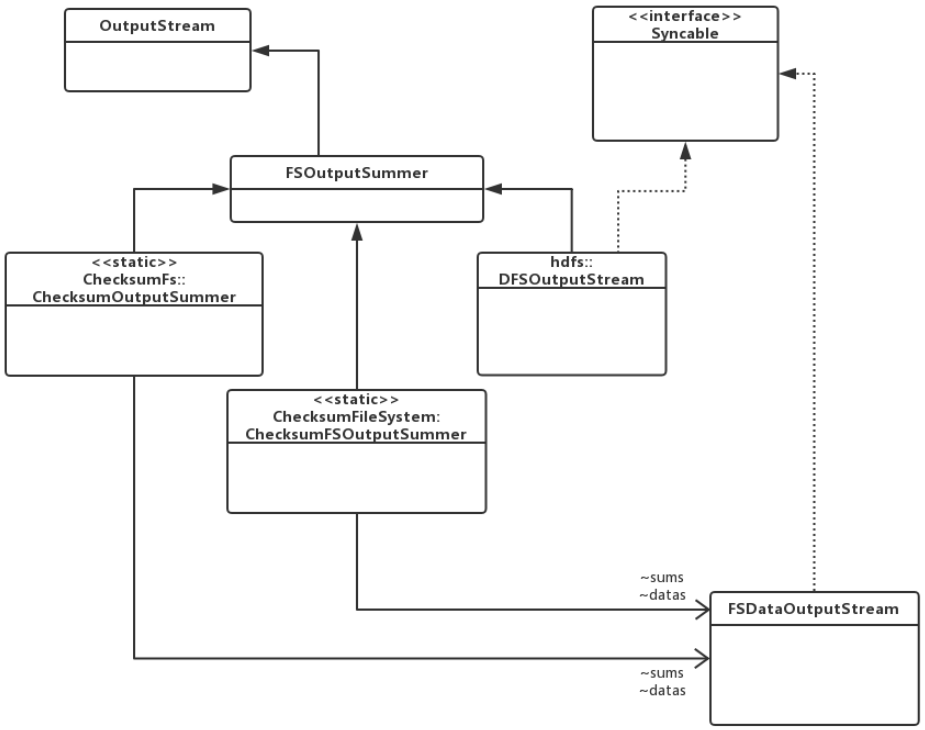

图 2-6 输出流类图

了解一点 Java 的人都知道，Java 的 IO 类被分割为输入输出两个部分。通过继承 InputStream()（输入流）和 OutputStream()（输出流）来实现对字节或字节流的操作字节。InputStream()通过使用隐式的记录指针来表示当前正准备从那个字节开始读取，读取之后，记录指针自定向向后移动；而 OutputStream 同样使用隐式指针记录即将存放字节的位置，输出之后，记录指针自动向后移动。

在 Hadoop 中，FSInputStream、FSDataInputStream 和 FSDataOutputStream 的作用与 InputStream、DataInputStream 和 DataOutputStream 在 Java IO 中的作用类似。

2.3.1 FSInputSystem

FSInputStream 实现了接口 Seekable 和 PositionedReadable。提供了基本的读取一个输入流的操作，比如定位 seek 和读取到特定的 buffer 的操作。FSInputStream 在原有的 InputStream 的基础上添加了 getPos()方法，同时可以通过 seek 方法定位指定的偏移量处。

各个不同的文件系统可以通过派生该类，重写相应的抽象方法，实现不同的功能。

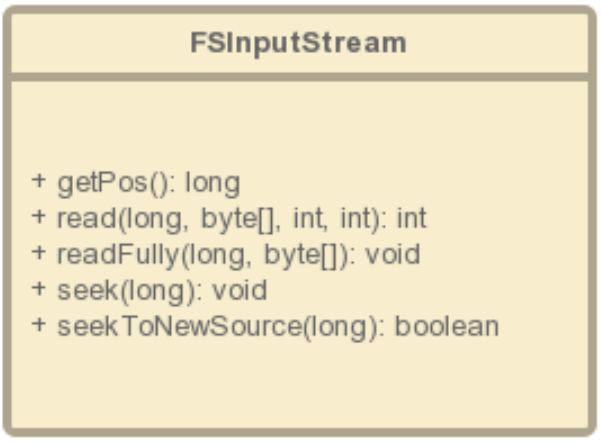

图 2-7 FSInputStream 类

2.3.2 FSOutputSummer

KFSOutputStream、S3OutputStream 和 FSOutputStream 类都是从抽象类 OutputStream 派生的。KFSOutputStream 和 S3OutputStream 都是具体类，他们重写了 OutputStream 中的抽象方法。而 FSOutputSummer 是抽象类，它实现了 OutputStream 中的 write()方法，并提供了计算校验和。FSOutputSummer 类中的属性 buf 数组用来缓存写入的数据，当数据超过 buf.length 个数时，才会调用 flushBuffer()实现真正的写入数据的功能。

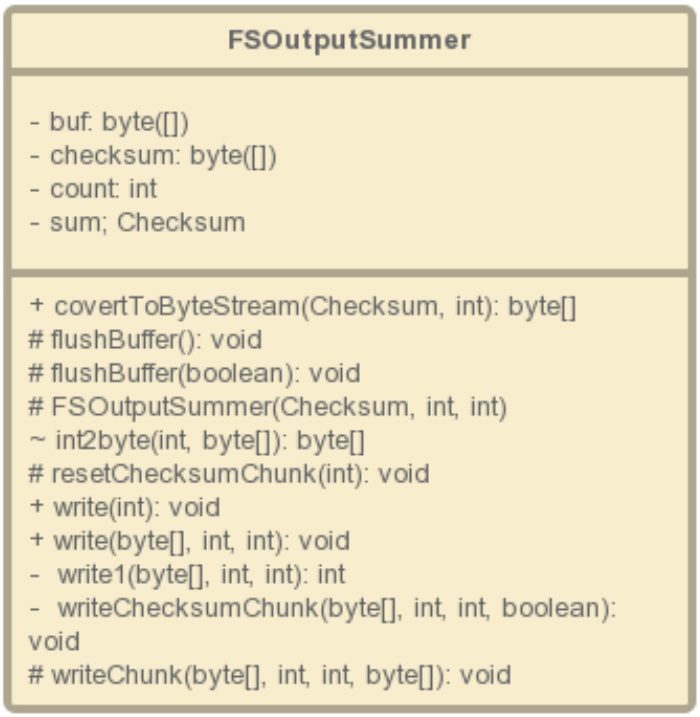

图 2-8 FSOutputSummer 类

2.3.3 FSDataInputSystem

FSDataInputStream 继承于 DataInputStream 类，实现了接口 Seekable 和 PositionedReadable，提供了随机访问的功能。实现了将 FSInputStream 包装在 DataInputStream 中并通过 BufferedInputStream 缓冲输入的实用程序。FSDataInputStream 通过实现接口，使 Hadoop 中的文件输入流具有流式搜索和流式定位读取的功能。

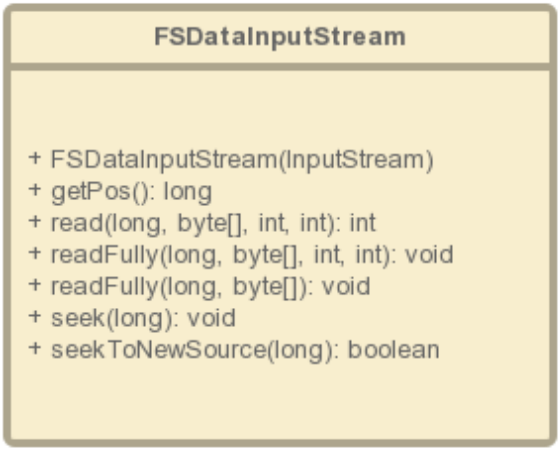

图 2-9 FSDataInputStream 类

2.3.4 FSDataOutputSystem

FSDataOutputStream 没有实现接口 Seekable 和 PositionedReadable，而是实现了 Syncable 接口，它实现了将 OutputStream 包装在 DataOutputStream 之中，并且不允许除文件尾部的其他位置的写入。

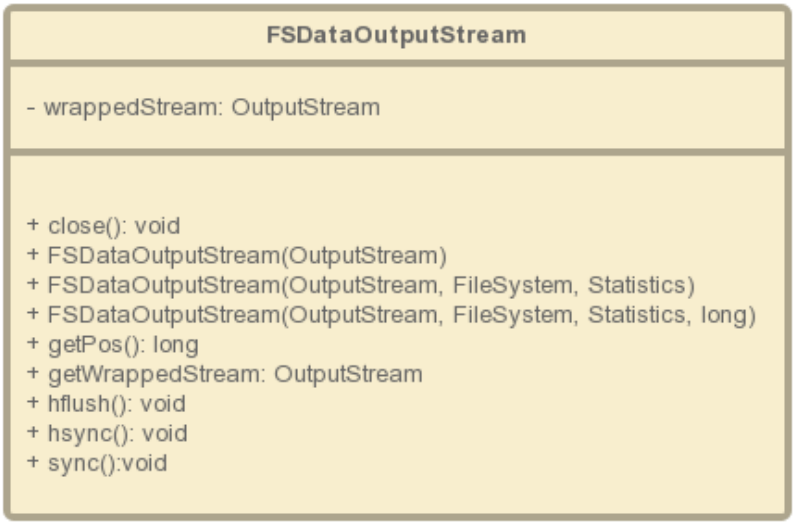

图 2-10 FSDataOutputStream 类

3. HDFS

HDFS（Hadoop Distributed File System），作为 Google File System（GFS）的实现，是 Hadoop 项目的核心子项目，是分布式计算中数据存储管理的基础，是基于流数据模式访问和处理超大文件的需求而开发的，可以运行于廉价的商用服务器上。它所具有的高容错、高可靠性、高可扩展性、高获得性、高吞吐率等特征为海量数据提供了不怕故障的存储，为超大数据集（Large Data Set）的应用处理带来了很多便利。

3.1 特点与架构

HDFS 具有高容错性、可部署在低廉的硬件上、高吞吐量、适用于海量数据的可靠性存储和数据归档等特点：

- 自动快速检测底层的硬件错误
- 以流式数据访问存储超大文件，提高数据吞吐量
- 一次写入、多次读取的文件访问模式，保证数据一致性
- 可以在廉价的商用硬件上实现集群扩展

HDFS 采用主-从（master/slave）架构。一个 HDFS 集群是由一个 NameNode 和一定数目的 DataNode 组成。NameNode 是一个中心服务器，负责管理文件系统的名字空间（namespace）以及客户端对文件的访问。集群中的一个节点具有一个 DataNode，负责管理其所在节点上的存储。

从内部看，一个文件被分为一个或多个数据块，这些块存储在一组 DataNode 上。NameNode 执行 namespace 内文件系统的操作，比如打开、关闭、重命名文件或目录操作。NameNode 同时负责确定数据块到具体的 DataNode 节点的映射。DataNode 负责处理文件系统客户端的读写请求，并且在 NameNode 的统一调度下进行数据块的创建、删除和复制。

图 3-1-1 和图 3-2-1 展示了 HDFS 的架构信息：

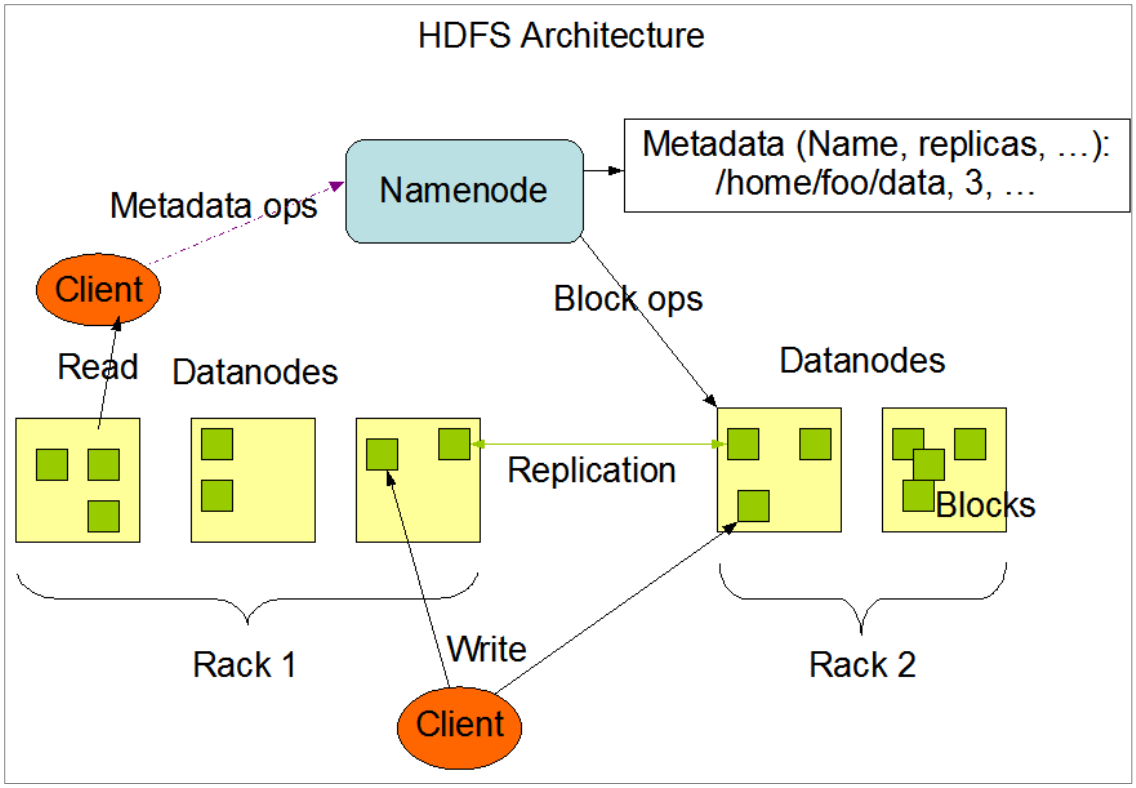

图 3-1-1 HDFS 架构图

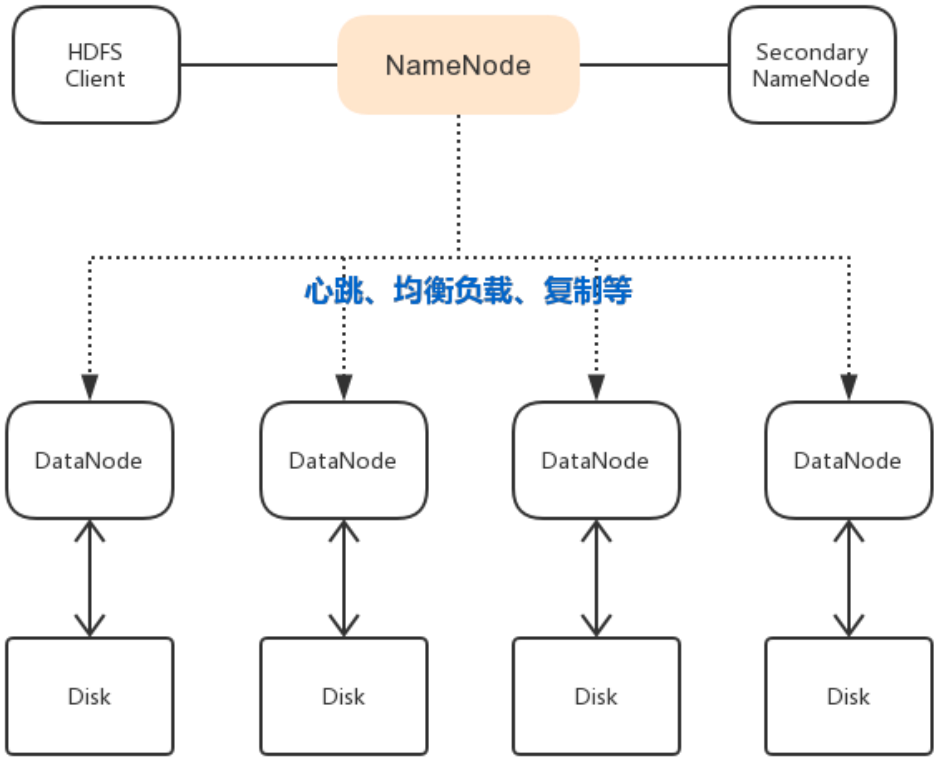

图 3-1-2 hdfs 架构图

HDFS 的架构主要分为 HDFS Client、NameNode、DataNode 和 Secondary NameNode，下面来分别简单介绍一下这四个组成部分：

1） Client（客户端）

- 文件切分。在文件上传 HDFS 的时候，Client 将文件切分成 Block 进行存储
- 与 NameNode 进行交互，获取文件位置
- 与 DataNode 进行交互，读取或写入数据
- 通过命令来访问并管理 HDFS

2） NameNode（Master）

- 管理 HDFS 的 namespace
- 管理数据块（block）的映射信息
- 配置副本策略
- 处理客户端读写请求

3） DataNode（Slave）

- 存储实际数据块
- 执行数据块读/写操作

4） Secondary NameNode

Secondary NameNode 并不是 NameNode 的热备份（当 NameNode 挂掉时，并不能马上替换 NameNode 并提供服务）

- 辅助 NameNode，分担其工作量
- 定期合并 fsimage（元数据镜像文件（文件系统的目录树））与 fsedits（元数据的操作日志（针对文件系统做的修改操作记录）），并推送给 NameNode
- 在紧急情况下，可辅助恢复 NameNode

分析 HDFS 的架构，我们可以发现很多主从架构的特点：并行计算（提升计算性能）、容错处理（提升计算可靠性）、计算精度（提升计算精确度），但同样，这样的架构会使设备孤立，没有共享的状态。而且主-从通信中的延迟是一个问题，导致 HDFS 不支持低延时数据访问。

接下来我们将分析 HDFS 的主从架构是怎样在 HDFS 的具体功能中体现的：

3.2 数据存储

HDFS 的目标之一是实现即使在出现故障时也能可靠地存储数据，HDFS 将数据块（block）作为一个存储单元，每个数据块的默认存储单位是 128M。和普通文件系统相同的点是，HDFS 中的文件也是被分成一个个数据块存储在 DataNode 下，但不同点是，HDFS 中，若文件大小小于一个数据块大小，就不需要占用整个数据块的存储空间。

3.2.1 异构存储

3.2.1.1 概述

异构存储作为 HDFS 的数据存储方式，其第一阶段是将数据节点存储模型从单个存储更改为存储集合，每个存储对应于物理存储介质，根据不同的数据类型采取不同的存储策略，也对应了不同的存储类型。这样就可以达到对不同的数据更好的实现它们的用途，也可以根据各个存储介质读写特性的不同发挥各自的优势，提高 HDFS 文件系统的性能和可靠性。譬如针对冷数据，采用像磁盘这样容量大、读写性能不高的介质存储；而对于热数据，则可用读写速度快、容量小的固态硬盘来存储。也就是说，HDFS 异构存储的实现使得我们不需要搭建独立的集群来存放不同类型的数据，在一套集群内即可实现数据的存储管理。

3.2.1.2 异构存储类型

异构存储添加了存储类型，分别为 ARCHIVE,DISK,SSD 和 RAM_DISK。其中：

- ARCHIVE: 高存储密度（PB 级存储），但计算能力很小，可用于支持存档存储、解决数据量的容量扩增问题
- DISK: 磁盘存储，默认存储类型
- SSD: 固态硬盘，读写速度快，容量小
- RAM_DISK: 内存存储，用于支持在内存中写入单个副本文件

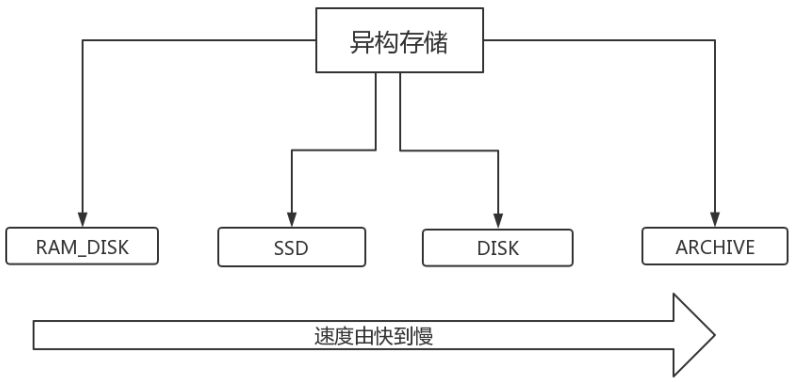

图 3-2-1 异构存储类型

```
35 public enum StorageType {
36   // sorted by the speed of the storage types, from fast to slow
37   RAM_DISK(true),
38   SSD(false),
39   DISK(false),
40   ARCHIVE(false);
41
42   private final boolean isTransient;
43
44   public static final StorageType DEFAULT = DISK;
45
46   public static final StorageType[] EMPTY_ARRAY = {};
47
48   private static final StorageType[] VALUES = values();
49
50   StorageType(boolean isTransient) {
51     this.isTransient = isTransient;
52   }
```

图 3-2-2 StorageType 类代码片段

在 StorageType 类中，对四种存储类型的选择进行了赋值，主要是根据内存存储的持续化写入功能进行划分。TRUE 和 FALSE 代表的是此类存储类型是否有 transient 特性，也就是非持久化。可见，只有内存存储 RAM_DISK 是非持久化的。在 HDFS 中，StorageType 的设定是十分重要的。

3.2.1.3 异构存储策略

异构存储引入了新的存储策略概念，以允许根据存储策略将文件存储在不同的存储类型中。存储策略共有 6 种，分别为：

- Hot：用于存储和计算，用于处理的数据将保留在此策略中。当块 hot 时，所有的副本都存储在 DISK 中。
- Cold：仅适用于计算量有限的存储，不再使用的数据或需要存档的数据将从热存储移动到冷存储中。当块 cold 时，所有副本存储在 ARCHIVE 中。
- Warm：数据部分热和部分冷。当一个块是 Warm 时，它的一些副本存储在 DISK 中，其余副本存储在 ARCHIVE 中。
- ALL_SSD：用于在 SSD 中存储所有副本。
- One_SSD：用于在 SSD 中存储其中一个副本，剩余副本存储在 DISK 中
- Lazy_Persist：用于在内存中写入具有单个副本的块，副本首先写在 RAM_DISK 中，然后将它懒惰的保存在 DISK 中。

表 3-4-1 典型的存储策略表

政策 ID	政策名称	块存放的存储类型列表	用于文件创建的回退存储类型列表	用于复制的回退存储类型列表
15	Lazy_Persist	RAM_DISK:1, DISK:n-1	DISK	DISK
12	All_SSD	SSD:n	DISK	DISK
10	One_SSD	SSD:1, DISK:n-1	SSD, DISK	SSD, DISK
7	Hot(default)	DISK:n	<none>	ARCHIVE
5	Warm	DISK:1, ARCHIVE:n-1	ARCHIVE, DISK	ARCHIVE, DISK
2	Cold	ARCHIVE:n	<none>	<none>

在异构存储选择数据存储类型且空间足够时，根据上述表项#3 中指定的存储类型列表存储块副本，而当#3 中的某些存储类型空间不足时，#4 和#5 中指定的回退存储类型列表分别用于替换文件创建和复制的空间存储类型。

3.2.1.4 异构存储的流程

异构存储的基本原理如图 3-2-3 所示，基本步骤分为以下 3 步：

- DataNode 通过心跳汇报自身数据存储目录的 StorageType 给 NameNode
- 随后 NamaNode 进行汇总并更新集群内各个节点的存储类型情况
- 待存储文件根据自身设定的存储策略信息向 NameNode 请求拥有此类型存储介质的 DataNode 作为候选节点

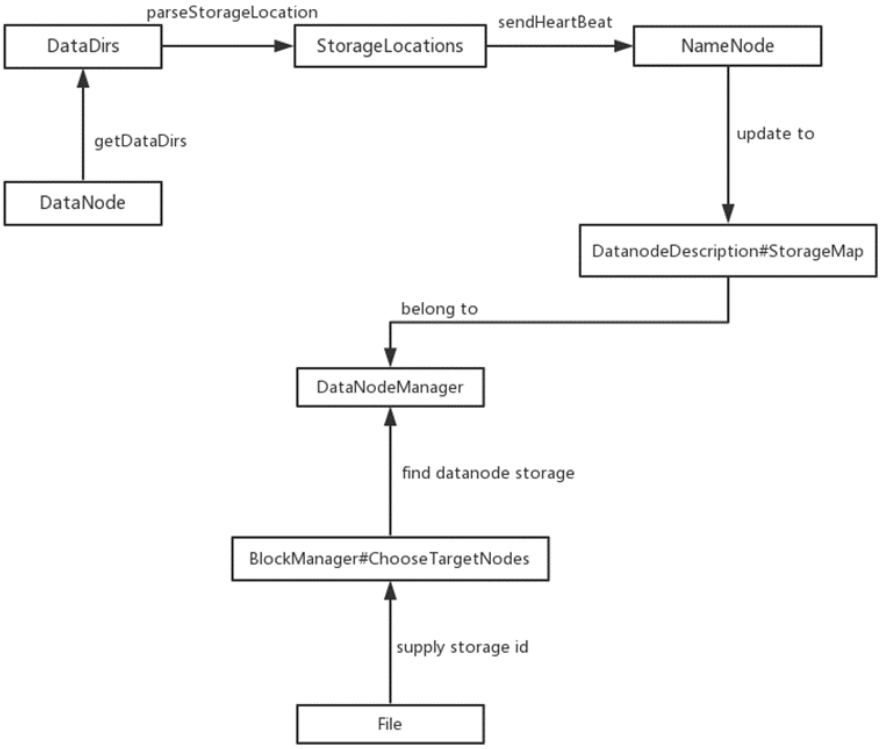

图 3-2-3 异构存储原理流程图

1) DataNode 目录解析和心跳汇报

首先是数据目录的解析与心跳汇报过程。在 FsDatasetImpl 类的构造函数中对 dataDir 进行存储目录的解析，生成了 StorageType 的 List 列表，该函数调用了 DataNode 类中的 getStorageLocation 方法来实现此功能。

```
,
FsDatasetImpl(DataNode datanode, DataStorage storage, Configuration conf
    ) throws IOException {
  this.fsRunning = true;
  this.datanode = datanode;
  this.dataStorage = storage;
  this.conf = conf;
  this.smallBufferSize = DFSUtilClient.getSmallBufferSize(conf);
  this.datasetLock = new AutoCloseableLock(
      new InstrumentedLock(getClass().getName(), LOG,
        new ReentrantLock(true),
        conf.getTimeDuration(
          DFSConfigKeys.DFS_LOCK_SUPPRESS_WARNING_INTERVAL_KEY,
          DFSConfigKeys.DFS_LOCK_SUPPRESS_WARNING_INTERVAL_DEFAULT,
          TimeUnit.MILLISECONDS),
        300));
  this.datasetLockCondition = datasetLock.newCondition();

  // The number of volumes required for operation is the total number
  // of volumes minus the number of failed volumes we can tolerate.
  volFailuresTolerated = datanode.getDnConf().getVolFailuresTolerated();

  Collection<StorageLocation> dataLocations = DataNode.getStorageLocations(conf);
  List<VolumeFailureInfo> volumeFailureInfos = getInitialVolumeFailureInfos(
      dataLocations, storage);
```

图 3-2-4 FsDatasetImpl 类的构造方法——实现数据目录解析功能

DataNode 节点调用 getStorageLocation 方法采用正则匹配的方式将数据存储目录对应的字符串解析成 StorageLocation 对象；

```
public static StorageLocation parse(String rawLocation)
    throws IOException, SecurityException {
  Matcher matcher = regex.matcher(rawLocation);
  StorageType storageType = StorageType.DEFAULT;
  String location = rawLocation;

  if (matcher.matches()) {
    String classString = matcher.group(1);
    location = matcher.group(2).trim();
    if (!classString.isEmpty()) {
      storageType =
          StorageType.valueOf(StringUtils.toUpperCase(classString));
    }
  }

  return new StorageLocation(storageType, new Path(location).toUri());
}
```

图 3-2-5 StorageLocation 类中的 parse 方法——解析数据存储目录对应的字符串

而后 FsDatasetImpl 类再将数据存储目录标识符和数据存储目录对应的 DataNodeStorage 实例加入 StorageMap

```
private void addVolume(Collection<StorageLocation> dataLocations,
    Storage.StorageDirectory sd) throws IOException {
  final File dir = sd.getCurrentDir();
  final StorageType storageType =
      getStorageTypeFromLocations(dataLocations, sd.getRoot());
```

图 3-2-6 FsDatasetImpl 类的 addVolume 方法——将 DataNodeStorage 实例加入 StorageMap

DataNode 将 StorageMap 存储的信息组织成 StorageReport 实例，而后调用 sendHeartBeat 方法获取数据目录存储类型的心跳汇报信息，并将心跳信息发送给 NameNode

```
public StorageReport[] getStorageReports(String bpid)
    throws IOException {
  List<StorageReport> reports;
  synchronized (statsLock) {
    List<FsVolumeImpl> curVolumes = volumes.getVolumes();
    reports = new ArrayList<>(curVolumes.size());
```

图 3-2-7 从 StorageMap 中获取心跳汇报信息

```
HeartbeatResponse sendHeartBeat(boolean requestBlockReportLease)
    throws IOException {
  scheduler.scheduleNextHeartbeat();
  StorageReport[] reports =
      dn.getFSDataset().getStorageReports(bpos.getBlockPoolId());
  if (LOG.isDebugEnabled()) {
    LOG.debug("Sending heartbeat with " + reports.length +
              " storage reports from service actor: " + this);
  }

  final long now = monotonicNow();
```

图 3-2-8 获取数据目录存储类型的心跳汇报信息，发送给 NameNode

2) 更新心跳信息

第二阶段是心跳处理和更新过程，前者主要是在 DataNodeManager 类的 handleHeartBeat 中进行，后者则用 heartbeatManager 的 updateHeartbeat 方法实现，具体代码实现分别如图 3-4-9 和 3-4-10 所示：

```
public DatanodeCommand[] handleHeartbeat(DatanodeRegistration nodeReg,
    StorageReport[] reports, final String blockPoolId,
    long cacheCapacity, long cacheUsed, int xceiverCount,
    int maxTransfers, int failedVolumes,
    VolumeFailureSummary volumeFailureSummary,
    @Nonnull SlowPeerReports slowPeers,
    @Nonnull SlowDiskReports slowDisks) throws IOException {
  final DatanodeDescriptor nodeinfo;
  try {
    nodeinfo = getDatanode(nodeReg);
  } catch (UnregisteredNodeException e) {
    return new DatanodeCommand[]{RegisterCommand.REGISTER};
  }

  // Check if this datanode should actually be shutdown instead.
  if (nodeinfo != null && nodeinfo.isDisallowed()) {
    setDatanodeDead(nodeinfo);
    throw new DisallowedDatanodeException(nodeinfo);
  }

  if (nodeinfo == null || !nodeinfo.isRegistered()) {
    return new DatanodeCommand[]{RegisterCommand.REGISTER};
  }
  heartbeatManager.updateHeartbeat(nodeinfo, reports, cacheCapacity,
      cacheUsed, xceiverCount, failedVolumes, volumeFailureSummary);
```

图 3-2-9 handleheartBeat 方法——处理 DataNode 的心跳信息

```
synchronized void updateHeartbeat(final DatanodeDescriptor node,
    StorageReport[] reports, long cacheCapacity, long cacheUsed,
    int xceiverCount, int failedVolumes,
    VolumeFailureSummary volumeFailureSummary) {
  stats.subtract(node);
  node.updateHeartbeat(reports, cacheCapacity, cacheUsed,
    xceiverCount, failedVolumes, volumeFailureSummary);
  stats.add(node);
}
```

图 3-2-10 更新心跳信息

最终在 Heartbeatmanager 中会调用到 DatanodeDesciptor 对象的 updateHeartbeatState 方法来不断更新 sotrage 中的信息。

3) 目标存储介质类型节点的请求

各个 DataNode 心跳信息都更新完毕后，有目标存储介质需求的待复制文件快就会向 NameNode 请求 DataNode，这部分处理在 FSNamesystem 的 getAdditionBlock 中进行：

```
2542  LocatedBlock getAdditionalBlock(
2543      String src, long fileId, String clientName, ExtendedBlock previous,
2544      DatanodeInfo[] excludedNodes, String[] favoredNodes,
2545      EnumSet<AddBlockFlag> flags) throws IOException {
2546    final String operationName = "getAdditionalBlock";
2547    NameNode.stateChangeLog.debug("BLOCK* getAdditionalBlock: {} inodeId {}" +
2548        " for {}", src, fileId, clientName);
2549
2550    LocatedBlock[] onRetryBlock = new LocatedBlock[1];
2551    FSDirWriteFileOp.ValidateAddBlockResult r;
2552    FSPermissionChecker pc = getPermissionChecker();
2553    checkOperation(OperationCategory.READ);
2554    readLock();
2555    try {
2556      checkOperation(OperationCategory.READ);
2557      r = FSDirWriteFileOp.validateAddBlock(this, pc, src, fileId, clientName,
2558                                            previous, onRetryBlock);
2559    } finally {
2560      readUnlock(operationName);
2561    }
2562
2563    if (r == null) {
2564      assert onRetryBlock[0] != null : "Retry block is null";
2565      // This is a retry. Just return the last block.
2566      return onRetryBlock[0];
2567    }
2568
2569    DatanodeStorageInfo[] targets = FSDirWriteFileOp.chooseTargetForNewBlock(
2570        blockManager, src, excludedNodes, favoredNodes, flags, r);
2571
```

图 3-2-11 请求符合目标存储介质类型的 DataNode 节点

目标存储节点信息就被保存到了具体某块的信息中，这里的 target 类型为 DatanodeStorageInfo,代表的是 DataNode 中的一个 DataDir 存储目录，而 blockManager 再根据给定的候选 DatanodeStorageInfo 存储目录和存储策略来选择出目标节点。

4) 缺陷与不足

目前 HDFS 还不能对文件目录存储策略变更做出自动的数据迁移，需要用户在执行了相应策略 setStoragePolicy 之后，额外执行 hdfs-mover 命令做文件目录的扫描，进行数据块的迁移。

3.2.2 内存存储

内存存储作为异构存储的一种方式，是其中最快的策略，对待即时使用消息和小文件的存储十分便利，在提高 HDFS 文件系统可用性的同时，也带来了极大的革新，不只局限于存储大文件，真正可以实现各种类型数据的海量存储从底层扩展了 HDFS 的数据存储方式。

3.2.2.1 内存存储原理

内存存储采用的是 LAZY PERSIST 策略，将机器的内存作为存储数据的载体，也就是说此时节点的内存也充当了一块“磁盘”。换个易理解的说法也就是，假设有个内存数据块队列，在队列头部不断有新增的数据块插入，就是待存储的块，因为资源有限，要把队列尾部的块，也就是早些时候的块持久化到磁盘中，然后才有空间腾出来存新的块，然后形成这样一个循环，新的块加入，老的块移除，保证了整体数据的更新。

HDFS 支持由 DataNode 管理的写入到堆栈内存的功能，DataNode 会异步的将数据从内存持久化至磁盘，从而在性能敏感的 IO Path 中移去昂贵的磁盘 IO 和校验，因此我们称之为 Lazy Persist。HDFS 尽可能保证在该策略下的持久性，但在副本还未持久化之磁盘时，节点重启的话，有可能发生罕见的数据遗失，这可以选择 Lazy Persist Writes 的策略来减少延迟，但会损失一定的持久性。

下面给出 Hadoop 官网提供的内存存储原理图：

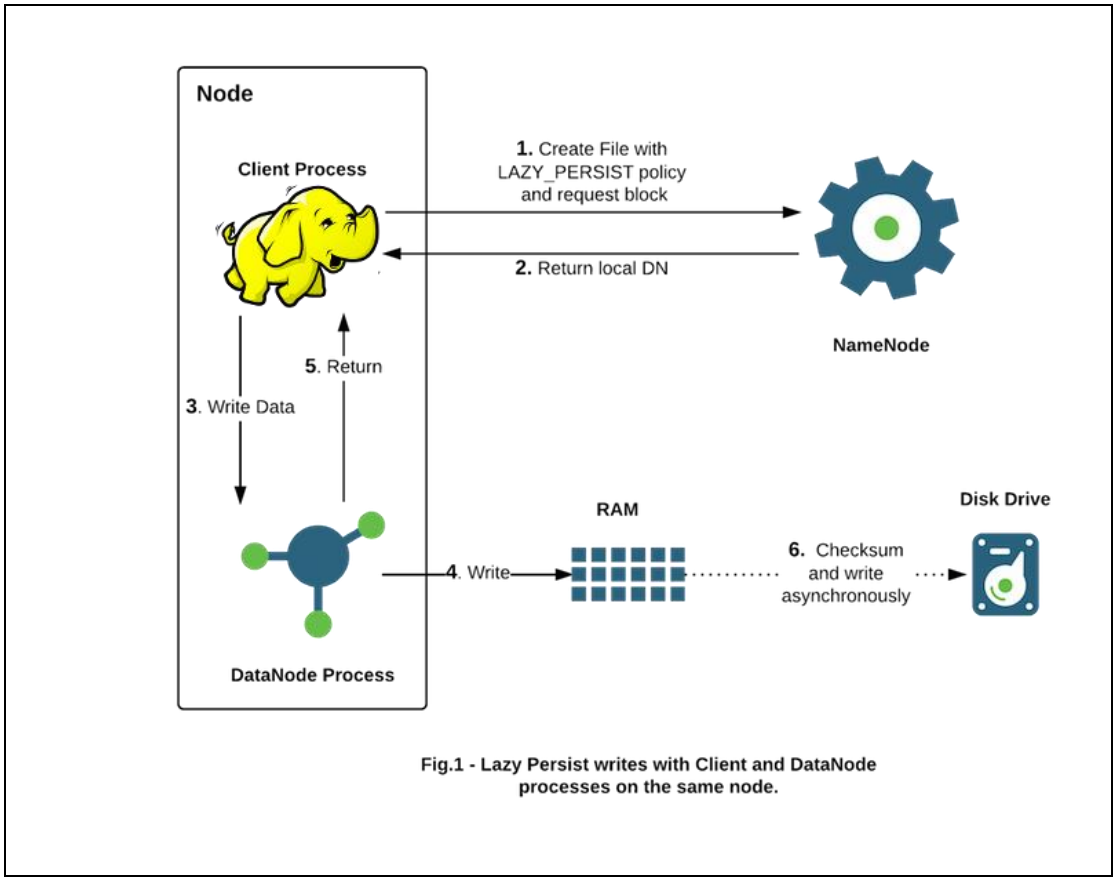

图 3-2-12 内存存储原理图

上图所展示的内存存储步骤大致可分为 3 步：

- 第一步，对目标文件目录设置 StoragePolicy 为 LAZY_PERSIST 内存存储策略
- 第二步，客户端进程向 NameNode 发起创建/写文件的请求
- 第三步，请求到具体的 DataNode，DataNode 会把这些数据块写入 RAM 内存中，同时启动异步线程服务将内存数据持久化到磁盘上。

内存的异步持久化存储，也就是其明显不同于其他介质存储数据的地方：数据不是立马落盘的，而是以懒惰、延时的方式来处理。

3.2.2.2 内存存储过程与分析

因为数据存储的同时会有另外一部分数据被异步的持久化，这就需要 FsDatasetImpl 来管理 DataNode 所有磁盘的读写数据。在 FsDatasetImpl 中，与内存存储相关的对象有 3 个：

- LazyWriter：一个线程服务，会不断循环地从数据块列表中取出数据块，加入到下一步的异步持久化线程池中去执行
- RamDiskAsyncLazyPersistService:异步持久化线程服务，针对每一个磁盘块设置一个对应的线程池，需要持久化到给定磁盘块的数据会被提交到对应的线程池中。每个线程池的最大线程数为 1.
- RamDiskReplicaLruTracker:副本块跟踪类，维护所有已持久化、未持久化的副本和总副本的数据信息。所以当一个副本被最终存储到内存中后，相应的副本所属队列信息会发生变更。

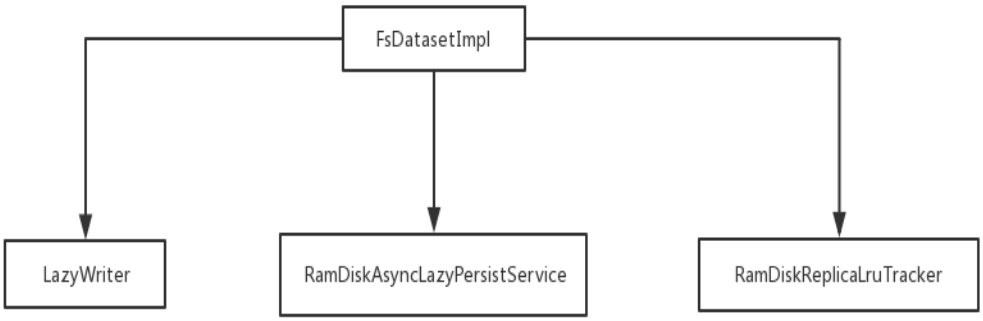

图 3-2-13 FsDatasetImpl 类涉及到内存存储的方法

在以上 3 者中，RamDiskReplicaLruTracker 扮演的是一个中间人的角色，在其内部维护了多个关系的数据块信息，主要就是以下 3 类：

```
/**
 * Map of blockpool ID to <map of blockID to ReplicaInfo>.
 */
Map<String, Map<Long, RamDiskReplicaLru>> replicaMaps;

/**
 * Queue of replicas that need to be written to disk.
 * Stale entries are GC'd by dequeueNextReplicaToPersist.
 */
Queue<RamDiskReplicaLru> replicasNotPersisted;

/**
 * Map of persisted replicas ordered by their last use times.
 */
TreeMultimap<Long, RamDiskReplicaLru> replicasPersisted;

RamDiskReplicaLruTracker() {
  replicaMaps = new HashMap<>();
  replicasNotPersisted = new LinkedList<>();
  replicasPersisted = TreeMultimap.create();
}
```

图 3-2-14 RamDiskReplicaLruTracker 中与内存存储有关的主要变量

RamDiskReplicaLruTracker 中的方法操作绝大多数与这 3 个变量的增删改动有关，这三者的关系图如下：

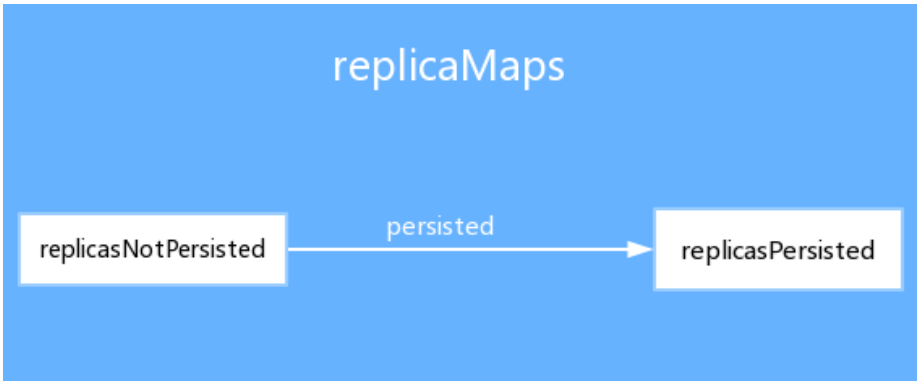

图 3-2-15

RAM 副本会存储在 replicaMaps 中，而 replicasNotPersisted 保存的是将会写入到磁盘的队列，replicasPersisted 保存的是已经被持久化的副本，并且按照上次使用的时间排序，这就涉及到后面将详述的 LRU 算法。

很显然，内存存储中的异步持久化操作步骤就是：当节点被设置为 LAZY_PERSIST 策略后，就会有新的副本块被存储到内存中，同时会加入到 replicaNotPersisted 队列中，然后 dequeueNextReplicaToPersist 方法再取出下一个将被持久化的副本块，进行写磁盘操作。该过程流程图如下：

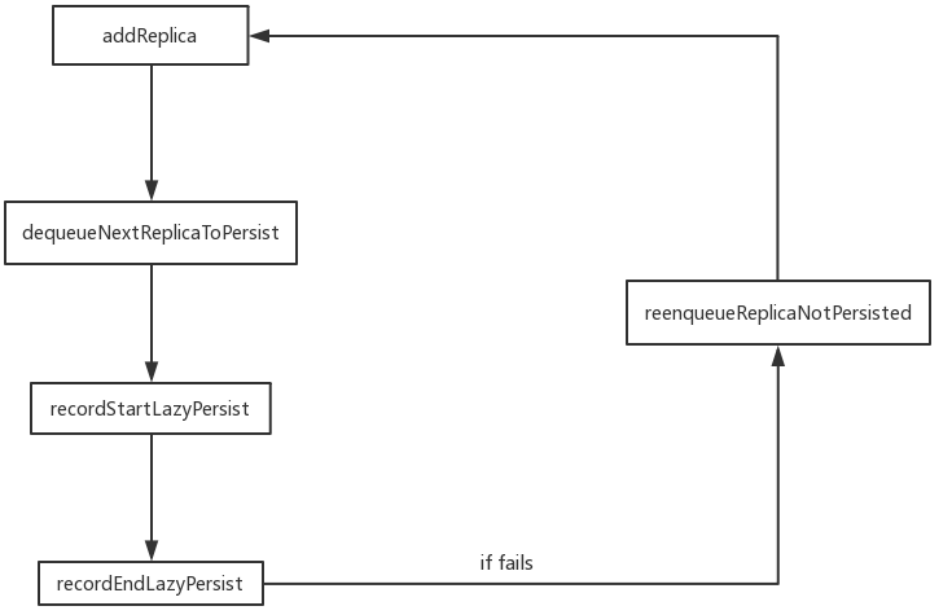

图 3-2-16 异步持久化流程图

这里涉及到一个内存节点空间不足时数据块替换的问题，在内存存储中是采用 LRU 算法来处理这一问题的。所谓 LRU 算法，也就是最近最少使用算法，使用这一算法，可以保证现有副本块的活跃度，会把最近很久没有访问过的副本块进行移除，具体的过程实现主要和 3 个方法相关：

- discardReplica：当某一副本被检测出不需要（已被删除或已损坏）的时候，可以从内存中移除、撤销。
- touch：意味着访问了一次某特定的副本块，会更新该副本块的 LastUsedTime
- getNextCandidateForEviction：在 DataNode 内存空间不足时，选择需要被移除的块，留出空间给新的副本块来存放，默认的是 LRU 策略。

```
private class RamDiskReplicaLru extends RamDiskReplica {
  long lastUsedTime;

  private RamDiskReplicaLru(String bpid, long blockId,
                            FsVolumeImpl ramDiskVolume,
                            long lockedBytesReserved) {
    super(bpid, blockId, ramDiskVolume, lockedBytesReserved);
  }
```

```
@Override
synchronized RamDiskReplicaLru getNextCandidateForEviction() {
  final Iterator<RamDiskReplicaLru> it = replicasPersisted.values().iterator();
  while (it.hasNext()) {
    final RamDiskReplicaLru ramDiskReplicaLru = it.next();
    it.remove();
```

图 3-2-17 内存存储中涉及到 LRU 算法的部分

通过原理和分析概念可以发现一个特别点，内存存储是根据已持久化的块的访问时间来进行筛选移除，而不是直接取内存中的块，最后是在内存中移除与候选块属于同一副本信息的块并释放内存空间。

介绍了 RamDiskReplicaLruTracker 类，接下来就看下剩下两个对象吧。

LazyWriter 是一个线程服务，它循环不断地从队列中取出待持久化的数据块，提交到异步持久化服务中去，其流程图可以归纳如下：

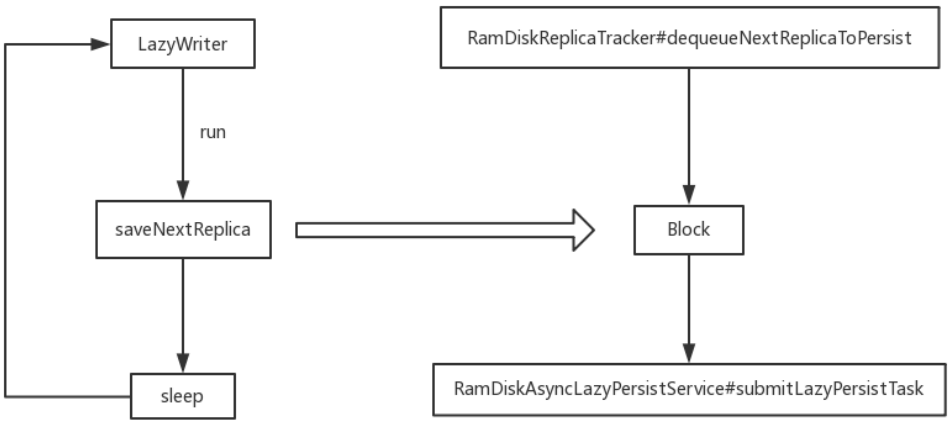

图 3-2-18 LazyWriter 流程图

Run:取出新的副本块并提交到异步服务中，返回是否提交成功布尔值；

saveNextReplica：从队列中取出新的待持久化的块，提交到异步服务器中去。

然后我们结合上述两个对象，就可以得到下面一个完整的流程（暂不考虑第三部分的内部执行逻辑）：

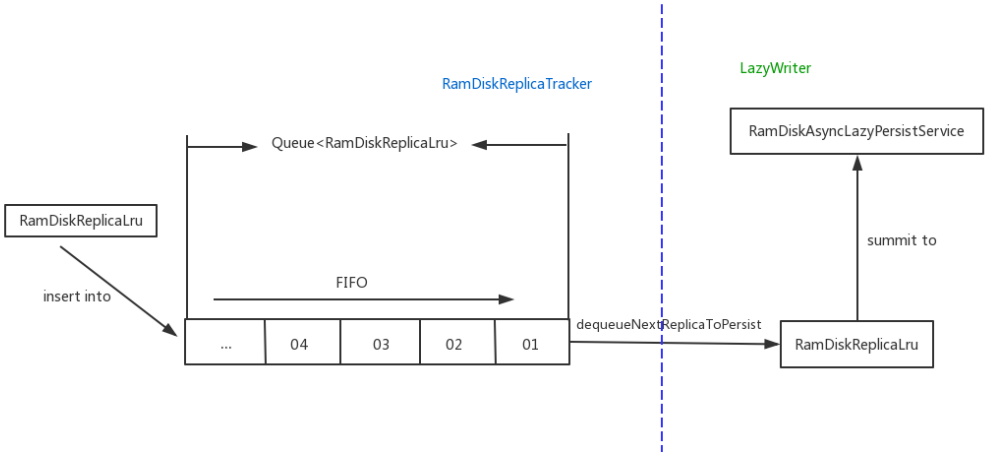

图 3-2-19

而异步服务的最后一部分内容 RamDiskAsyncLazyPersistService 相对比较简单，主要围绕 Volume 磁盘和 Executor 线程池。线程池列表定义如下：

```
class RamDiskAsyncLazyPersistService {
  public static final Log LOG = LogFactory.getLog(RamDiskAsyncLazyPersistService.class);

  // ThreadPool core pool size
  private static final int CORE_THREADS_PER_VOLUME = 1;
  // ThreadPool maximum pool size
  private static final int MAXIMUM_THREADS_PER_VOLUME = 1;
  // ThreadPool keep-alive time for threads over core pool size
  private static final long THREADS_KEEP_ALIVE_SECONDS = 60;

  private final DataNode datanode;
  private final Configuration conf;

  private final ThreadGroup threadGroup;
  private Map<File, ThreadPoolExecutor> executors
      = new HashMap<File, ThreadPoolExecutor>();
```

图 3-2-20 线程池列表

用红色圈出的 File 代表的是一个独立的磁盘所在目录，但我认为可以用 String 字符串替代，这样既可以减少存储空间，有简单明了。RamDiskAsyncLazyPersistService 总的结构图如下：

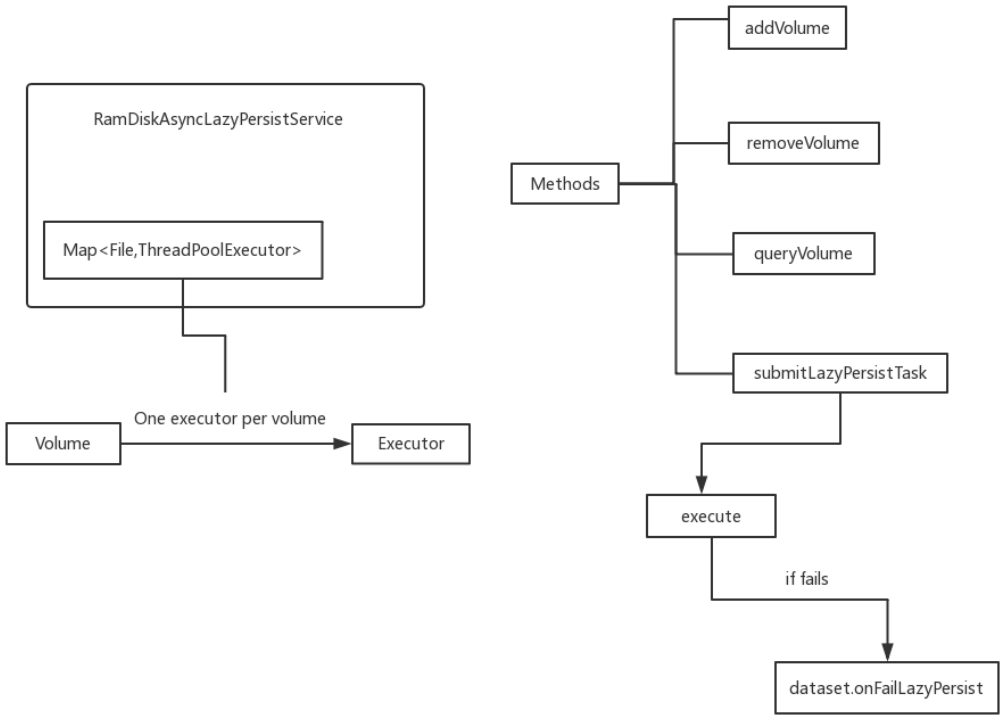

图 3-2-21 RamDiskAsyncLazyPersistService 结构图

以上便是内存存储的原理内容，要真正的使用，还需要进行具体的配置才行，这里就不多赘述了。总的来说，实现数据的持久化是内存存储的难点和重点，采用 LRU 策略后，可以减少延迟，达到懒持久的目的。

3.2.3 块存储

在 HDFS 异构存储方式中，除了内存存储之外，还有一类存储方式也很重要，就是 HDFS 的 Archival Storage。Archival Storage 指的是一种高密度的存储方式，以此解决集群数据规模增长带来的存储空间不足的问题。通常用于 Archival Storage 的节点不需要很好的计算性能，一般用于冷数据的存储。

块存储策略集合是 BlockStoragePolicySuite，在这个类内部定义了 6 种策略，这 6 中策略也就是异构存储的基本存储策略类型：LAZY_PERSIST, ALL_SSD, ONE_SSD, HOT, WARM, COLD, 他们的处理数据的速度依次减缓。因为它们的构造方法基本相同，这里就只截出 LAZY_PERSIST 部分源码以展示。

```
policies[lazyPersistId] = new BlockStoragePolicy(lazyPersistId,
    HdfsConstants.MEMORY_STORAGE_POLICY_NAME,
    new StorageType[]{StorageType.RAM_DISK, StorageType.DISK},
    new StorageType[]{StorageType.DISK},
    new StorageType[]{StorageType.DISK},
    true);    // Cannot be changed on regular files, but inherited.
```

图 3-2-22 源码中的存储策略构造

块存储的大致流程就是：首先在 BlockStoragePolicySuite 类中初始化构造六种存储策略类型，同时 NameNode 通过 INodeFile 获取 INode 文件目录和块存储策略的 ID 类型，并以此来决定具体的存储方式。

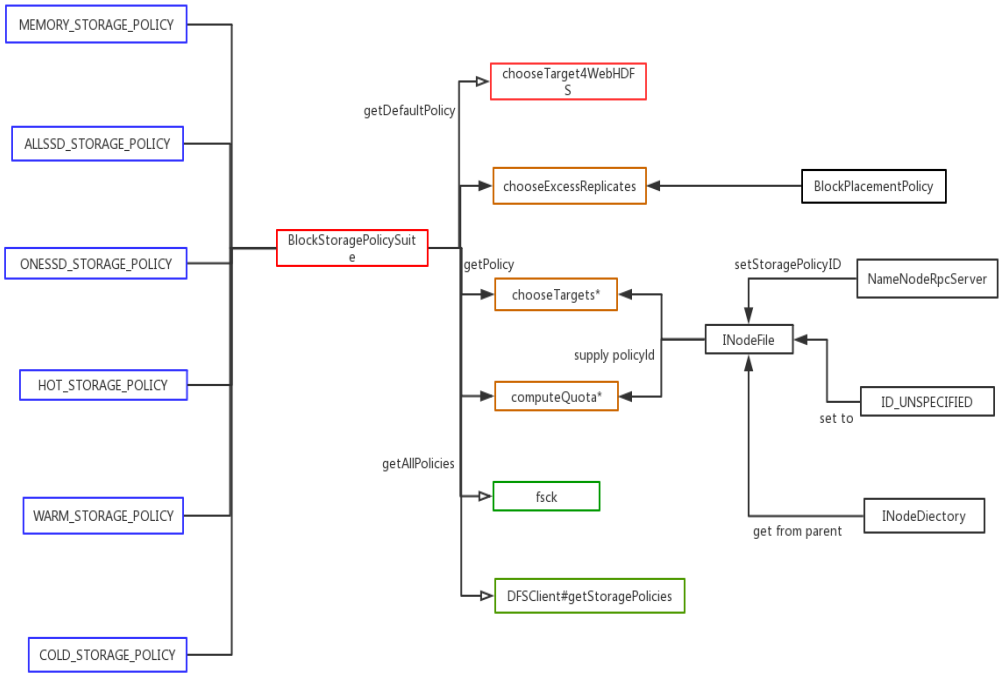

图 3-2-23 块存储的大致流程

3.2.4 其他

HDFS 虽然提高了数据存储的可靠性，但仍存在着一些不足。目前 HDFS 还不能对文件目录存储策略做出自动的数据迁移，需要用户在执行了相应策略 setStoragePolicy 之后，额外执行 hdfs –mover 命令做文件目录的扫描，进行数据块的迁移。这里所指的文件目录存储策略变更有：

- 原先未设置 StoragePolicy，后来进行了设置
- 原先设置了 A 策略，后来又设置了 B 策略

在内存存储中还有涉及到一个老师文档要求的问题：

> 本题目的主要内容是对目前开源领域著名的云计算平台 Hadoop 中底层的、文件系统的支持机制进行深入分析与理解，得出其设计理念，根据分析结果考虑何向 Hadoop 中加入对 Windows 文件系统的支持。最终给出对应的代码分析报告实验过程报告。

内存存储的原理是利用 Linux 操作系统本身存在虚拟内存盘的部分来模拟一个硬盘，也就是说 Linux 可以使用内核支持的机制来支持，而 Windows 系统并没有这块部分，所以不能直接使用内存存储，如果要使用，就需要安装相应的软件，如 VSuite Ramdisk 等等，然后这些软件再利用系统分配给它的内存空间来实现这种模拟。所以 Hadoop 可以在推出 Windows 版本是，在内存存储这块实现这类软件的功能，或者说，利用自身的内存空间来模拟硬盘，这样就可以实现对 Windows 文件系统的支持了。

3.3 数据管理

为了提高 HDFS 的可用性和高容错性，HDFS 采用了很多对数据管理的方式，其中，我们将从数据快照、数据复制，和集中缓存管理来介绍 HDFS 的主从架构，是怎样服务于 HDFS 的数据管理的。

3.3.1 数据快照

3.3.1.1 概述

HDFS 快照是文件系统的只读时间点副本。可以在文件系统的子树或整个文件系统上拍摄快照。快照的一些常见用例是数据备份，防止用户错误和灾难恢复。

快照不是简单的数据复制，而是只做差异的复制：这一原则在其他很多系统快照概念中都是遵守的,比如磁盘快照,也是不保存真实数据的.因为不保存实际的数据,所以快照的生成往往非常的迅速.在 HDFS 中,如果对其中一个目录比如/A 下创建一个快照,则快照文件中将会有与/A 目录下完全一样的子目录文件结构以及相应的属性信息,通过 fs -cat 也能看到里面的具体的文件内容,但是这并不意味着 snapshot 对此数据进行完全拷贝,这里遵循一原则,对于大多不变的数据,你所看到的数据其实是当前物理路径所指的内容,而发生变更的 INode 才是会被 snapshot 额外拷贝,其实是一个差异拷贝.HDFS 中只为每个 snapshot 快照只保存相对

当时快照创建时间点发生过变更的 INode 信息,只是"存不同". 然后获取快照信息时,根据 snapshotId 和当前没发生过变更的 INode 信息,进行对应恢复即可.

HDFS 快照的实施非常有效

- 快照的创建是即时的：成本是 O（1），不包括 inode 查找时间
- 仅当相对于快照进行修改时才使用附加内存：内存使用量为 O（M），其中 M 是已修改文件/目录的数量。
- 不复制 datanode 中的块：快照文件记录块列表和文件大小。没有数据复制。
- 快照不会对常规 HDFS 操作产生负面影响：以反向时间顺序记录修改，以便可以直接访问当前数据。通过从当前数据中减去修改来计算快照数据。

3.3.1.2 快照实现原理

实现上是通过在每个目标节点下面创建 snapshot 节点，后续任何子节点的变化都会同步记录到 snapshot 上。例如删除子节点下面的文件，并不是直接文件元信息以及数据删除，而是将他们移动到 snapshot 下面。这样后续还能够恢复回来。另外 snapshot 保存是一个完全的现场，不仅是删除的文件还能找到，新创建的文件也无法看到。后一种效果的实现是通过在 snapshot 中记录哪些文件是新创建的，查看列表的时候将这些文件排除在外。

在 HDFS 中 INode 表示一个节点，其中 INodeFile 表示文件，INodeDirectory 表示目录。INodeFileWithSnapshot 表示带有快照的文件，INodeDirectoryWithSnapshot 表示带有快照的目录，（INodeDirectorySnapshottable 表示可以创建快照的目录，INodeDirectoryWithSnapshot 不能创建新的快照，只能将目录的变化记录到现有的快照里面），相关的类结构如下

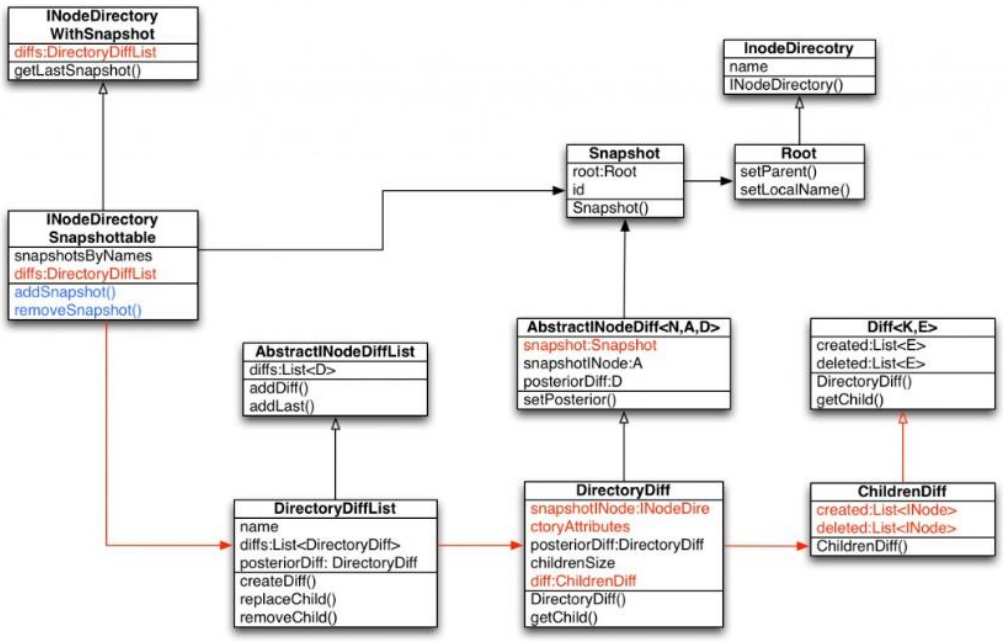

图 3-3-1 INodeDirectorySnapshottable 类结构

图中红线表示的是关键类的引用关系，其中最重要的是 DirectoryDiffList，里面保存了一些快照和当前目录的差别。每一个 DirectoryDiff 中包含快照以及儿子变化，是实现快照功能的核心。ChilderenDiff 中 created list 保存的是从快照时间之后新创建的节点，deteled list 保存的新删除的节点。snapshot 中的 root 节点保存了 snapshot 的 name，可以通过这个找到对应的快照。

对 Snapshottable 类介绍如下：

- 快照管理器管理多个快照目录
- 一个快照目录拥有多个快照文件
- 不允许创建出网状关系的快照目录
- 一旦将目录设置为 snapshottable，就可以在任何目录上拍摄快照
- 快照目录可以容纳 65,536 个同步快照，快照目录的数量没有限制
- 管理员可以将任何目录设置为快照
- 如果快照目录中有快照，则在删除所有快照之前，既不能删除也不能重命名目录。
- 目前不允许嵌套的快照目录。换句话说，如果其祖先/后代之一是快照目录，则无法将目录设置为 snapshottable。

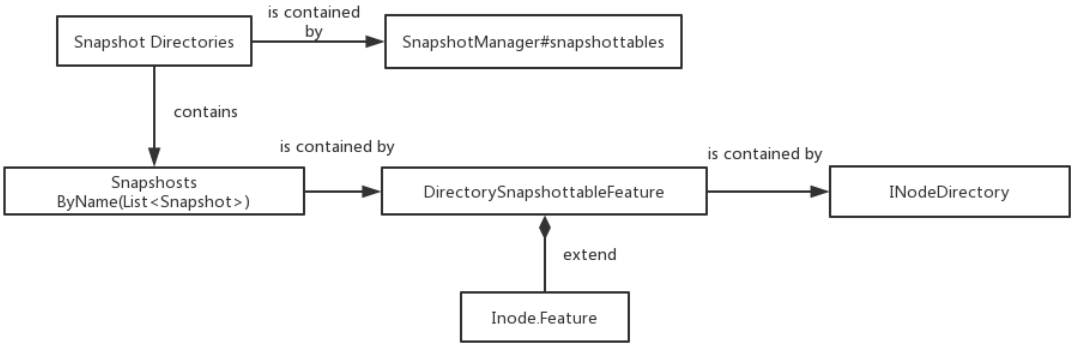

图 3-3-2 Snapshot 结构关系

3.3.1.3 其他

以 SnapshotManager 为一个处理中心，给出 snapshot 快照的调用过程。SnapshotManager 负责接收 snapshot 操作请求,继而调用相关类进行处理，这里的相关类就是 INodeDirectory 中的 Feature 继承类了，所以全部过程分为如下俩部分：

1） 上游请求的接收。如图：

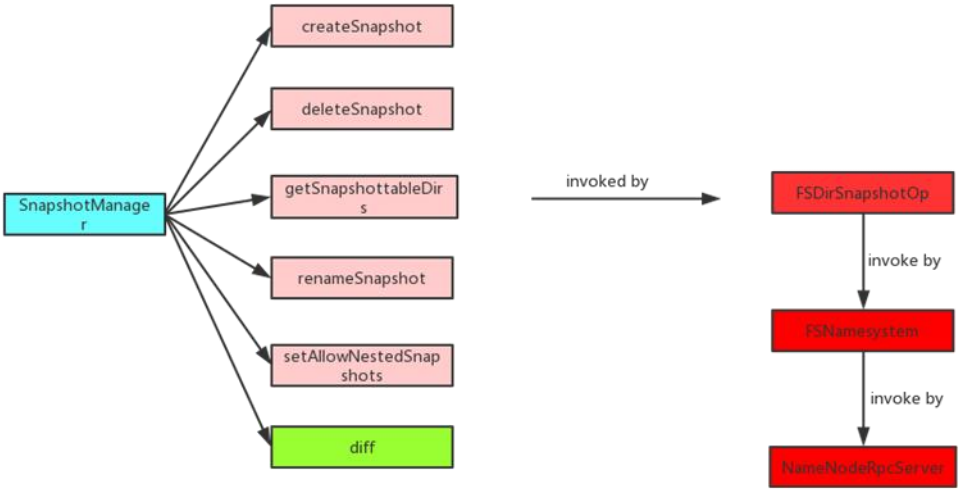

图 3-3-3 上游请求流程图

2） 请求的下游处理。如图：

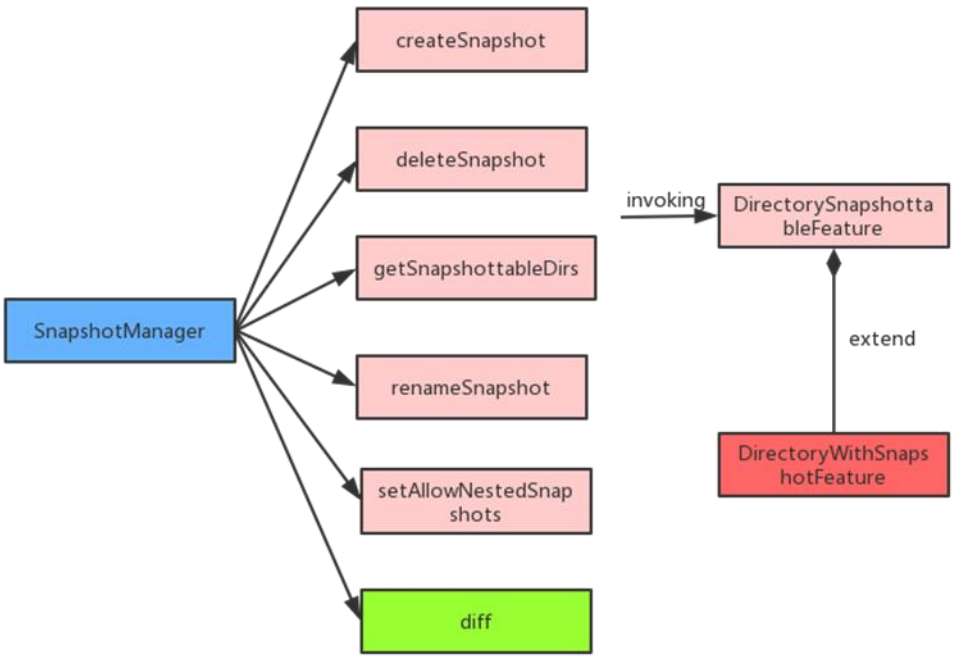

图 3-3-4 下游处理流程图

对 Snapshot 原理进行分析，提出以下两个问题：

1) Snapshot 快照如何生成的,如何能够做到元数据的完全一致的?快照是如何完全一致的反映出那一时刻的文件目录信息呢?

i. Snapshot 的创建操作是基于 hadoop fs 的-createSnapshot 命令触发的,需要传入 2 个参数,快照所在父目录名和快照名称.

ii. 每个 snapshot 都有对应自身目录下的 INode 信息列表，以 snapshotID 作为区分标志

iii. 通过将 diff 发生过变更的 INode 信息与原目录节点信息进行结合,然后返回一个新的子节点信息作为最终结果返回的.diff 中保留的 INode 就是当时快照创建时的 INode 信息.

iv. HDFS 中只为每个 snapshot 快照只保存相对当时快照创建时间点发生过变更的 INode 信息,只是"存不同". 然后获取快照信息时,根据 snapshotId 和当前没发生过变更的 INode 信息,进行对应恢复即可.

2) Snapshot 快照之间是如何做 diff 比较出不同的?

命令如下：

```
hdfs snapshotDiff <path> <fromSnapshot> <toSnapshot>
```

有 4 种变更类型：created，deleted，modified，renamed.

snapshotDiff 函数处理分为俩过程：

i. 生成 SnapshotDiffInfo 对象，此对象里面包含了源，目标快照间的发生改变的文件信息，对于同一目录,会有多个 dirDiff(dirDiff 指的是相对此 snapshot 发生改变的 INode),这些 dirDiff 被加入到了 DirDiffList 列表对象中,然后根据 snapshotId 作为下标索引进行获取.同样的在 File 文件中,也存在单个 snapshot 的 FileDiff 对象,以及 FileDiffList 列表项.

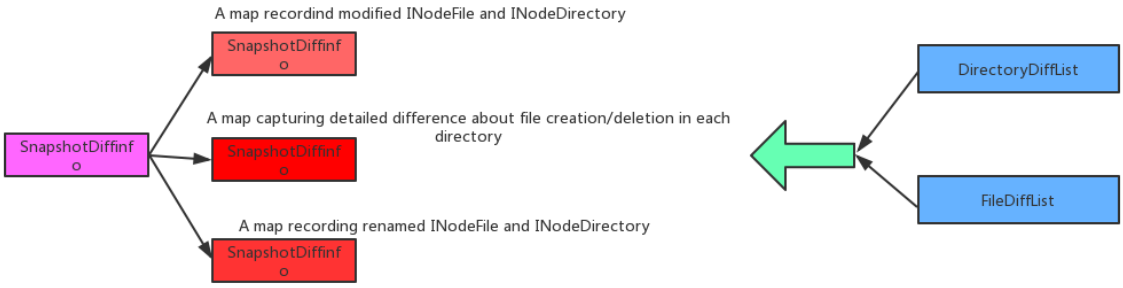

图 3-3-5 生成 SnapshotDiffInfo 对象过程

ii. 根据发生改变的文件目录信息生成 diff 报告。

3.3.2 数据复制

HDFS 设计成能可靠地在集群中大量机器之间存储大量的文件。它的存储方式是将每个文件分成一系列块，除了最后一个块之后的所有块都具备相同的大小，HDFS 中的文件是一次写的，并且任何时候都只有一个写操作,文件中除了最后一个块，其他块都有相同的大小。属于文件的块为了故障容错而被复制,应用程序可以指定文件的副本数。复制因子可以在文件创建时指定，并可以在之后修改。NameNode 做出有关块复制的所有决定，定期从集群中的每个 DataNode 接收 Heartbeat 和 Blockreport,Heartbeat 代表着 DataNode 正常运行，Blockreport 包含 DataNode 上所有块的表。

3.3.2.1 副本的放置策略

块副本存放位置的选择严重影响 HDFS 的可靠性和性能。副本存放位置的优化是 HDFS 区分于其他分布式文件系统的的特征，机架感知副本放置策略的目的是提高数据可靠性，可用性和网络带宽利用率。副本放置策略的当前实现是朝这个方向的第一次努力。大型 HDFS 实例在通常分布在多个机架上的计算机群集上运行。不同机架中两个节点之间的通信必须通过交换机。在大多数情况下，同一机架中的计算机之间的网络带宽大于不同机架中的计算机之间的网络带宽。

一个简单但非最优的策略是将复制品放在独特的机架上。这可以防止在整个机架出现故障时丢失数据，并允许在读取数据时使用来自多个机架的带宽。此策略在群集中均匀分布副本，这样可以轻松平衡组件故障的负载。但是，此策略会增加写入成本，因为写入需要将块传输到多个机架。

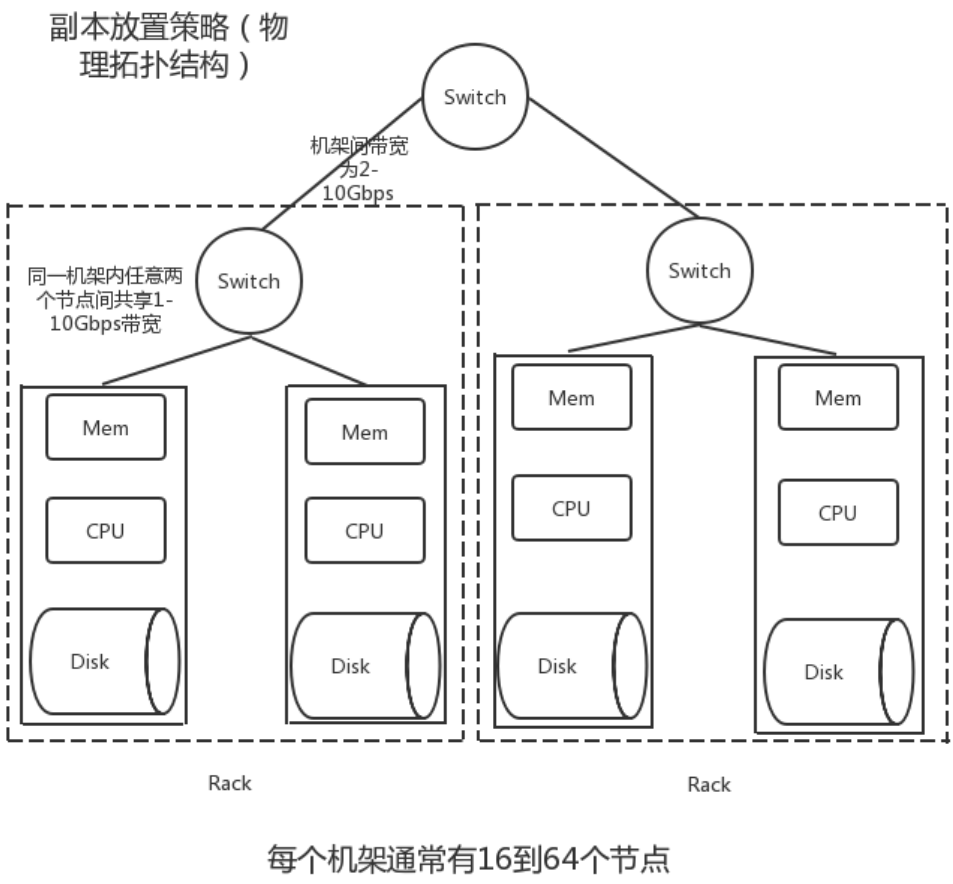

图 3-3-6 副本放置策略

对于常见情况，当复制因子等于 3 时，HDFS 的副本放置策略是将第一个副本放在本地节点，将第二个副本放到本地机架上的另外一个节点而将第三个副本放到不同机架上的节点。这种方式减少了机架间的写流量，从而提高了写的性能。机架故障的几率远小于节点故障。这种方式并不影响数据可靠性和可用性的限制，并且它确实减少了读操作的网络聚合带宽，因为文件块仅存在两个不同的机架， 而不是三个。文件的副本不是均匀地分布在机架当中，1/3 在同一个节点上，1/3 副本在同一个机架上，另外 1/3 均匀地分布在其他机架上。这种方式提高了写的性能，并且不影响数据的可靠性和读性能。

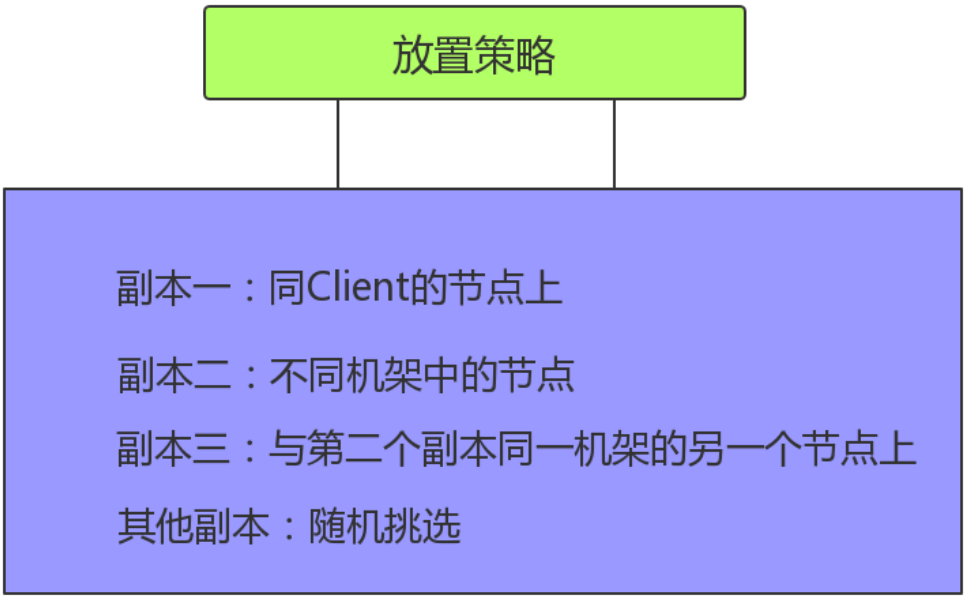

图 3-3-7 副本放置策略

如果复制因子大于 3 时，就随机确定第四个及以下副本的放置，主要遵循策略是每个机架的副本数量低于上限（基本上是（副本-1）/机架+2），**注意 NameNode 不允许 DataNode 具有同一块的多个副本，所以创建最大副本数是此时 DataNode 的总数。**

其中，策略选择的核心方法是 chooseTargets():

```
196 /*初始化操作*/    //如果需要的副本数为0或机器节点数量为0，返回空
197     if (numOfReplicas == 0 || clusterMap.getNumOfLeaves()==0) {
198       return DatanodeStorageInfo.EMPTY_ARRAY;
199     }
200       //创建移除名单列表集
201     if (excludedNodes == null) {
202       excludedNodes = new HashSet<Node>();
203     }
204      //计算每个机架所允许的最大副本数
205     int[] result = getMaxNodesPerRack(chosenStorage.size(), numOfReplicas);
206     numOfReplicas = result[0];
207     int maxNodesPerRack = result[1];
208 /*选择目标节点*/       //将所选节点加入到结果列表中，同时加入到移除列表中，意为已选择过的节点
209     final List<DatanodeStorageInfo> results = new ArrayList<DatanodeStorageInfo>(chosenStorage);
210     for (DatanodeStorageInfo storage : chosenStorage) {
211       //添加已选择的节点到移除列表中
212       addToExcludedNodes(storage.getDatanodeDescriptor(), excludedNodes);
213     }
214       //判断是否需要避免旧的、未更新的节点
215     boolean avoidStaleNodes = (stats != null
216         && stats.isAvoidingStaleDataNodesForWrite());
217       //选择numOfReplicas副本数的目标机器，并返回其中第一个节点
218     final Node localNode = chooseTarget(numOfReplicas, writer, excludedNodes,
219         blocksize, maxNodesPerRack, results, avoidStaleNodes, storagePolicy,
220         EnumSet.noneOf(StorageType.class), results.isEmpty());
221       //如果不想返回初始选中的目标节点，则进行移除
222     if (!returnChosenNodes) {
223       results.removeAll(chosenStorage);
224     }
225 /*对目标节点列表进行排序，形成pipeline*/
226     // 根据最短距离对目标节点列表进行排序，率成Pipeline
227     return getPipeline(
228         (writer != null && writer instanceof DatanodeDescriptor) ? writer
229             : localNode,
230         results.toArray(new DatanodeStorageInfo[results.size()]));
231   }
```

图 3-3-8 chooseTargets()方法代码片段

其中：

- 存储的存储类型必须是请求的存储类型
- 存储不能是 READ_ONLY（只读）
- 存储不能是坏的
- 存储所在节点不应该是已下线或下线中的节点
- 存储所在节点不应该是消息落后节点
- 节点内保证有足够的剩余空间能满足写块所要求的大小
- 要考虑节点 IO 负载繁忙程度
- 要满足同机架内最大副本数的限制

3.3.2.2 副本选择

为了最大限度地减少全局带宽消耗和读取延迟，HDFS 尝试满足最接近读取器的副本的读取请求。如果在与读取器节点相同的机架上存在副本，则该副本首选满足读取请求。如果 HDFS 群集跨越多个数据中心，则驻留在本地数据中心的副本优先于任何远程副本。

3.3.2.3 安全模式

在启动时，NameNode 进入一个名为 Safemode 的特殊状态。当 NameNode 处于 Safemode 状态时，不会发生数据块的复制。NameNode 从 DataNode 接收 Heartbeat 和 Blockreport 消息。Blockreport 包含 DataNode 托管的数据块列表。每个块都有指定的最小副本数。当使用 NameNode 检入该数据块的最小副本数时，会认为该块是安全复制的。在可配置百分比的安全复制数据块使用 NameNode 检入（再加上 30 秒）后，NameNode 退出 Safemode 状态。然后，它确定仍然具有少于指定数量的副本的数据块列表（如果有）。然后，NameNode 将这些块复制到其他 DataNode。

3.3.3 集中缓存管理

3.3.3.1 背景

Hadoop 设计之初借鉴 GFS/MapReduce 的思想：移动计算的成本远小于移动数据的成本。所以调度通常会尽可能将计算移动到拥有数据的节点上，在作业执行过程中，从 HDFS 角度看，计算和数据通常是同一个 DataNode 节点，即存在大量的本地读写。

但是 HDFS 最初实现时，并没有区分本地读和远程读，二者的实现方式完全一样，都是先由 DataNode 读取数据，然后通过 DFSClient 与 DataNode 之间的 Socket 管道进行数据交互。这样的实现方式很显然由于经过 DataNode 中转对数据读性能有一定的影响。

社区很早也关注到了这一问题，先后提出过两种方案来提升性能，分别是 HDFS-347 和 HDFS-2246。

HDFS-2246 是比较直接和自然的想法，既然 DFSClient 和 DataNode 在同一个节点上，当 DFSClient 对数据发出读请求后，DataNode 提供给 DFSClient 包括文件路径，偏移量和长度的三元组（path,offset,length）,DFSClient 拿到这些信息后直接从文件系统读取数据，从而绕过 DataNode 避免一次数据中转的过程。但是这个方案存在两个问题，首先，HDFS 需要为所有用户配置白名单，赋予其可读权限，当增加新用户需要更新白名单，维护不方便；其

次，当为用户赋权后，意味着用户拥有了访问所有数据的权限，相当于超级用户，从而导致数据存在安全漏洞。

HDFS-347 使用 UNIX 提供的 Unix Domain Socket 进程通讯机制实现了安全的本地短路读取。DFSClient 向 DataNode 请求数据时，DataNode 打开块文件和元数据文件，通过 Unix Domain Socket 将对应的文件描述符传给 DFSClient，而不再是路径、偏移量和长度等三元组。文件描述符是只读的，DFSClient 不能随意修改接收到的文件。同时由于 DFSClient 自身无法访问块所在的目录，也就不能访问未授权数据。

虽然本地短路读在性能上有了明显的提升，但是从全集群看，依然存在几个性能问题：

- DFSClient 向 DataNode 发起数据读请求后，DataNode 在 OS Buffer 对数据会进行 Cache，但是数据 Cache 的分布情况并没有显式暴漏给上层，对任务调度透明，造成 Cache 浪费。比如同一 Block 多个副本可能被 Cache 在多个存储这些副本的 DataNode OS Buffer，造成内存资源浪费。
- 由于 Cache 的分布对任务调度透明，一些低优先级任务的读请求有可能将高优先级任务正在使用的数据从 Cache 中淘汰出去，造成数据必须从磁盘读，增加读数据的开销从而影响任务的完成时间，甚至影响到关键生产任务 SLA.

3.3.3.2 概论

HDFS 中的集中式缓存管理是一种显式缓存机制，允许用户指定 HDFS 缓存的路径。NameNode 将与磁盘上具有所需块的 DataNode 进行通信，并指示它们将块缓存在堆外缓存中。

HDFS 中的集中式缓存管理具有许多显着优势。

- 用户可以指定常用数据或者高优先级任务对应的数据常驻内存，避免被淘汰 到磁盘。例如在数据仓库应用中事实表会频繁与其他表 JOIN，如果将这些事实表常驻内存，当 DataNode 内存使用紧张的时候也不会把这些数据淘汰出去，可以很好的实现了对于关键生产任务的 SLA 保障；
- 由 NameNode 统一进行缓存的集中管理，DFSClient 根据 Block 被 Cache 分布 情况调度任务，尽可能实现本地内存读，减少资源浪费；

- 明显提升读性能。当 DFSClient 要读取的数据被 Cache 在同一 DataNode 时， 可以通过 ZeroCopy（零拷贝）直接从内存读，略过磁盘 IO 和 checksum 校验等环节，从而提升读性能；
- 集中式缓存可以提高整体集群内存利用率。当依赖于每个 DataNode 上的 OS 缓冲区高速缓存时，重复读取块将导致块的所有 n 个副本被拉入缓冲区高速缓存。借助集中式高速缓存管理，用户可以明确地引脚只米的的 ñ 副本，节省了纳米内存。

3.3.3.3 架构及原理

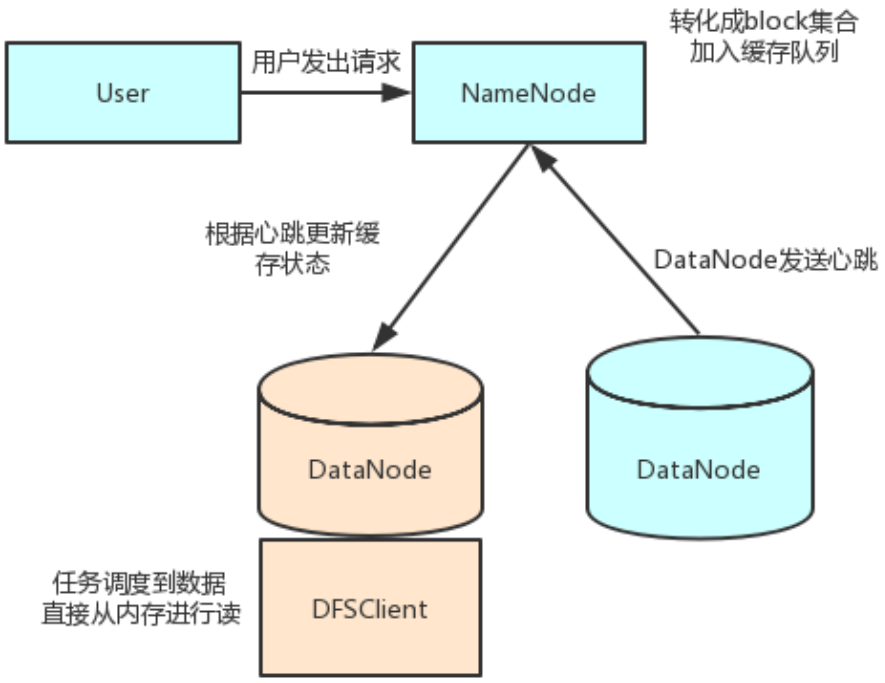

图 3-3-9 HDFS 缓存流程图

对 CacheDirective 缓存的详细流程

1) 用户通过调用客户端接口向 NameNode 发起对 CacheDirective（Directory/File）缓存请求
2) NameNode 接收到缓存请求后，将 CacheDirective 转换成需要缓存的 Block 集 合，并根据一定的策略并将其加入到缓存队列中

3) NameNode 接收到 DataNode 心跳后，将缓存 Block 的指令下发到 DataNode

4) DataNode 接收到 NameNode 发下的缓存指令后根据实际情况触发 Native JNI 进行系统调用，对 Block 数据进行实际缓存，缓存目前在文件和目录级别上完成

5) 此后 DataNode 定期（默认 10s）向 NameNode 进行缓存汇报，更新当前节点 的缓存状态

6) 上层调度尽可能将任务调度到数据所在的 DataNode，当客户端进行读数据请 求时，通过 DFSClient 直接从内存进行 ZeroCopy，从而显著提升性能

HDFS 集中式缓存的架构如图 3-3-9 所示，这里主要涉及到 NameNode 和 DataNode 两侧的管理和实现，NameNode 对数据缓存的统一集中管理，并根据策略调度合适的 DataNode 对具体的数据进行数据的缓存和置换，DataNode 根据 NameNode 的指令执行对数据的实际缓存和置换。

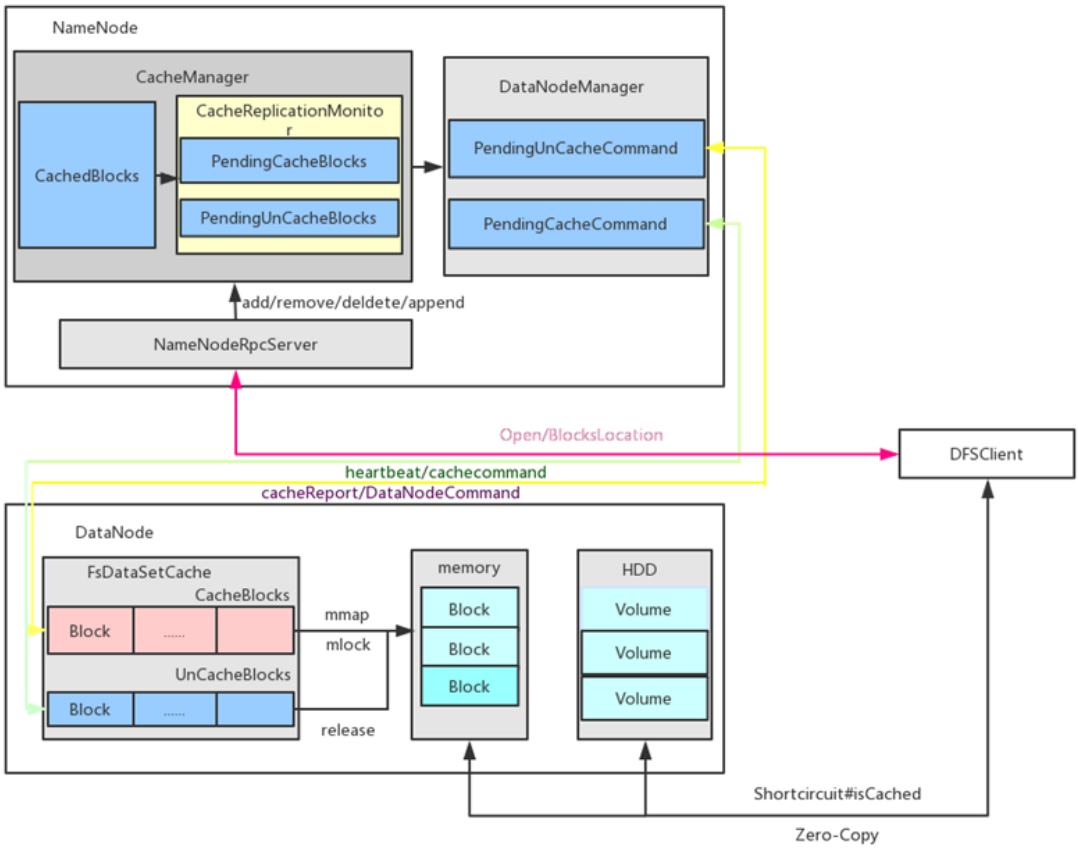

图 3-3-10 HDFS 集中缓存架构图

DataNode 是执行缓存和置换的具体执行者，具体来说即 cacheBlock 和 uncacheBlock 调用。FsDatasetImpl#FsDatasetCache 类是该操作的执行入口。

FsDatasetCache 的核心是称为 mappableBlockMap 的 HashMap，用于维护当前缓存的 Block 集合，其中 Key 为标记该 Block 的 ExtendedBlockId，为了能够实现与 Federation 的兼容，在 blockid 的基础上增加了 blockpoolid，这样多个 blockpool 的 block 不会因为 blockid 相同产生冲突；Value 是具体的缓存数据及当前的缓存状态。

private final HashMap<ExtendedBlockId, Value> mappableBlockMap = new HashMap <ExtendedBlockId, Value>();

注：生命周期状态

//缓存块缓存中状态

CACHING

//缓存块取消状态

CACHING_CANCELLED

//缓存块已缓存状态

CACHED

//缓存块取消缓存状态

UNCACHING

mappableBlockMap 更新只有 FsdatasetCache 提供的两个具体的函数入口：

synchronized void cacheBlock(long blockId, String bpid, String blockFileName, long length, long genstamp, Executor volumeExecutor)

synchronized void uncacheBlock(String bpid, long blockId)

对 mappableBlockMap 的并发控制实际上放在了 cacheBlock 和 uncacheBlock 两个方法上，虽然锁粒度比较大，但是并不会对并发读写带来影响。原因是：cacheBlock 在系统调用前构造空 Value 结构加入 mappableBlockMap 中，此时该 Value 维护的 Block 状态是 CACHING，之后将真正缓存数据的任务加入异步任务 CachingTask 去完成，所以锁很快会被释放，当处于 CACHING 状态的 Block 被访问的时候会退化到从 HDD 访问，异步任务 CachingTask 完成数据缓存后将其状态置为 CACHED；uncacheBlock 是同样原理。所以，虽然锁的粒度比较大，但是并不会阻塞后续的数据缓存任务，也不会对数据读写带来额外的开销。

CacheBlock、UnCacheBlock 场景触发：

1) cacheBlock:唯一的入口是从 ANN（HA Active NameNode）下发的指令

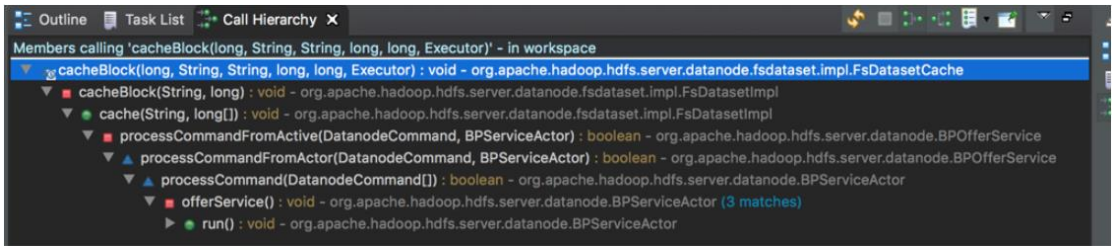

图 3-3-11 cacheBlock 函数触发图

2) uncacheBlock:与 cacheBlock 不同，uncacheBlock 存在三个入口：append，invalidate 和 uncache。其中 uncache 也是来自 ANN 下发的指令；append 和 invalidate 触发 uncacheBlock 的原因是：append 会导致数据发生变化，缓存失效需要清理后重新缓存，invalidate 来自删除操作，缓存同样失效需要清理。

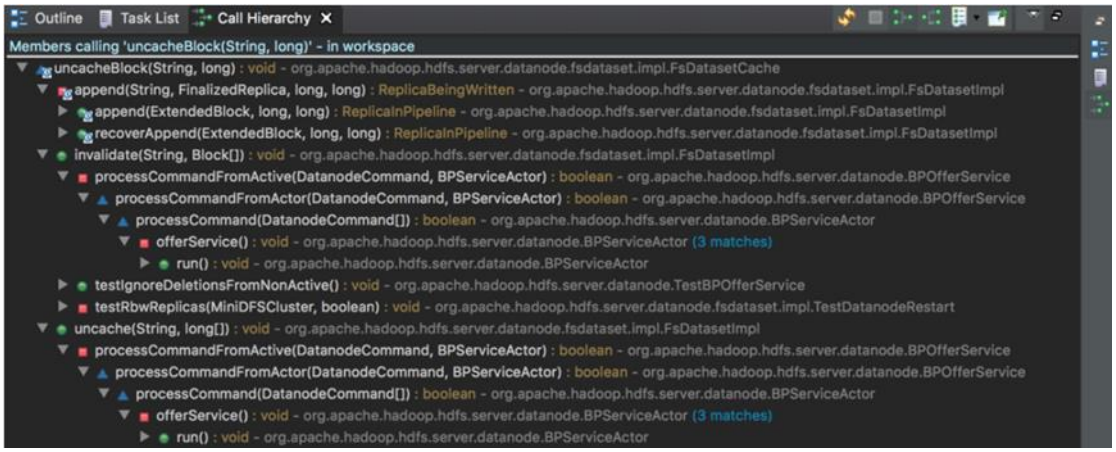

图 3-3-12 uncacheBlock 函数触发图

3.3.3.4 相关概念

- 缓存指令

一个缓存指令定义应该缓存的路径。路径可以是目录或文件。目录以非递归方式缓存，仅表示目录的第一级列表中的文件。

指令还指定其他参数，例如缓存复制因子和到期时间。复制因子指定要缓存的块副本数。如果多个缓存指令引用同一文件，则应用最大缓存复制因子。

到期时间在命令行中指定为生存时间（TTL），缓存指令到期后，NameNode 在进行缓存决策时不再考虑它

- 缓存池

一个高速缓存池是用来管理缓存指令组的管理实体。缓存池具有类似 UNIX 的权限，这些权限限制哪些用户和组可以访问池。写入权限允许用户向池中添加和删除缓存指令。读取权限允许用户列出池中的缓存指令以及其他元数据。执行权限未使用。

缓存池还用于资源管理。池可以强制执行最大限制，这限制了池中指令可以聚合聚合的字节数。通常，池限制的总和将近似等于为群集上的 HDFS 缓存保留的聚合内存量。缓存池还会跟踪大量统计信息，以帮助群集用户确定缓存的内容和缓存内容。

缓存池也可以实施最长的生存时间。这限制了添加到池中的指令的最大到期时间.

3.3.3.5 Cacheadmin 命令行界面

在命令行上，管理员和用户可以通过 hdfs cacheadmin 子命令与缓存池和指令进行交互。缓存指令由唯一的，不重复的 64 位整数 ID 标识。缓存池由唯一的字符串名称标识。

- 缓存指令命令
  - addDirectivere

用法：添加新的缓存指令

hdfs cacheadmin -addDirective -path <path> -pool <pool-name> [-force] [-replication <replication>] [-ttl <time-to-live>]

  - removeDirective

用法：删除缓存指令

hdfs cacheadmin -removeDirective <id>

  - removeDirectives

用法：删除具有指定路径的每个缓存指令

hdfs cacheadmin -removeDirectives <path>

  - listDirectives

用法：列出缓存指令

- 缓存池命令
  - addPool

    用法：添加新的缓存池
  - modifyPool

    用法：修改现有缓存池的元数据
  - removePool

    用法：删除缓存池，这同时解除了与池相关了路径
  - listPools

    用法：显示有关一个或多个缓存池的信息，例如名称，所有者，组，权限等
  - help

    用法：help,获取有关命令的详细帮助

3.4 数据访问

在对读写流程进行分析之前，首先对 HDFS 的数据模型进行介绍：

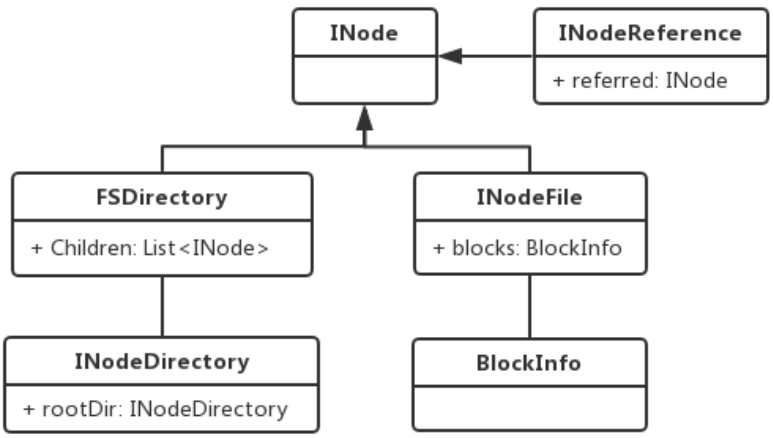

图 2-4-1 hdfs 数据模型

如上图所示，在 NameNode 中有一个唯一的 FSDirectory 类负责维护文件系统的节点关系。文件系统中的每个路径会被抽象为一个 INode 对象。在 FSDirectory 中有一个叫做 rootDir 的 INodeDirectory 类，继承自 INode 类，它代表着整个文件系统的根节点。

常用的 INode 节点有 INodeDirectory, INodeFile, INodeReference 三种。

- INodeDirectory 类代表着对目录对象的抽象，在类中有一个 List<INode> 对象 children 负责保存当前节点的子节点信息。
- INodeFile 类代表着对文件对象的抽象，对于一个大文件， Hdfs 可能将其拆分为多个小文件进行存储，在这里的 blocks 对象是一个数据对象，代表着小文件的具体存放位置信息。
- INodeReference 类可以理解成 Unix 系统中的硬链接。当文件系统中可能出现多个 path 地址对应同一个 INode 节点时，会构造出 INodeReference 对象。

对于 HDFS 而言，它的默认 FileSystem 抽象类的实现类是 DistributedFileSystem，在 DistributedFileSystem 内有 DFSClient 对象。这个对象利用了 RPC 通信机制，构造了一个 NameNode 的代理对象，负责同 NameNode 间进行 RPC 操作。

```
118 public class DistributedFileSystem extends FileSystem
119     implements KeyProviderTokenIssuer {
120   private Path workingDir;
121   private URI uri;
122   private String homeDirPrefix =
123       HdfsClientConfigKeys.DFS_USER_HOME_DIR_PREFIX_DEFAULT;
124
125   DFSClient dfs;
126   private boolean verifyChecksum = true;
127
128   private DFSOpsCountStatistics storageStatistics;
129
130   static{
131     HdfsConfiguration.init();
132   }
133
```

图 2-4-2 DistributedFileSystem 类代码片段

其中，RPC（Remote Procedure Call）又称作远程过程调用，它是一种从远程计算机程序上请求服务，而不需要了解底层网络通信技术的协议。RPC 协议底层会采用 UDP 网络协议来实现程序之间数据的传输。RPC 采用客户机/服务器模式来实现通信。服务请求程序是一个客户机，而服务提供程序是一个服务器。首先，客户机调用进程会发送一个有进程参数的调用信息到服务进程，然后等待应答消息。在服务端，进程保持睡眠状态直到调用信息到达

为止。当一个调用信息到达，服务器获得进程参数，计算结果，并将答复信息发送给客户机，然后等待下一个调用信息。最后，客户端调用进程会接受服务端发来的答复信息，获得计算结果，然后继续处理结果。这就是一个完整的 RPC 过程。

在 Hadoop 中，为了方便集群中各个组件的通信，它采用了 RPC 通信协议。同时，为了提高不同组件之间的通信效率以及组件自身的负载情况，Hadoop 在内部实现了一个基于 IPC 模型的 RPC。服务端把某些接口和接口中的方法暴露给客户端，客户端和服务端只需要实现这些接口中的方法就可以进行通信了。

3.4.1 文件读取

HDFS 文件读取的步骤图：

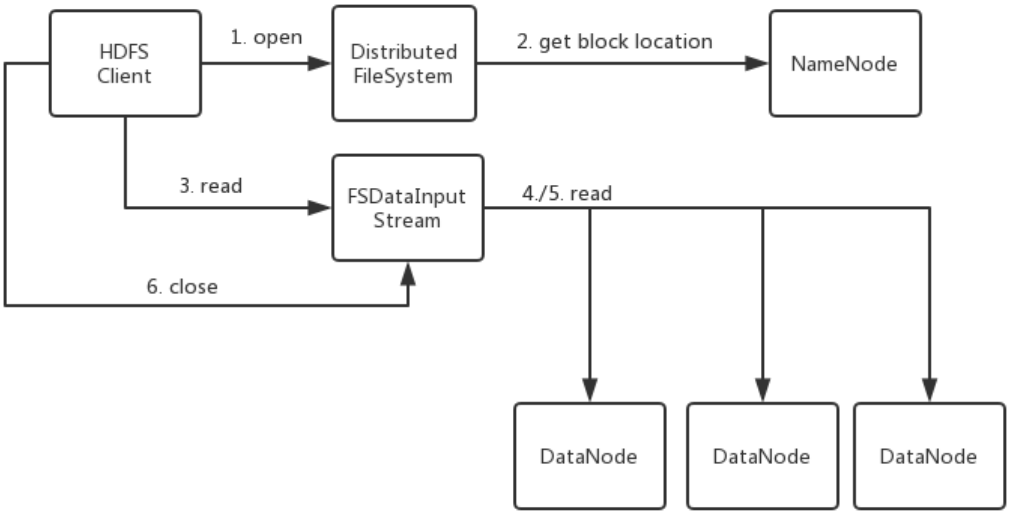

图 2-4-3 hdfs 文件读取步骤图

1） 客户端通过调用 DistributedFileSystem 的 open 方法

```
323  @Override
324  public FSDataInputStream open(Path f, final int bufferSize)
325      throws IOException {
326    statistics.incrementReadOps(1);
327    storageStatistics.incrementOpCounter(OpType.OPEN);
328    Path absF = fixRelativePart(f);
329    return new FileSystemLinkResolver<FSDataInputStream>() {
330      @Override
331      public FSDataInputStream doCall(final Path p) throws IOException {
332        final DFSInputStream dfsis =
333            dfs.open(getPathName(p), bufferSize, verifyChecksum);
334        return dfs.createWrappedInputStream(dfsis);
335      }
336      @Override
337      public FSDataInputStream next(final FileSystem fs, final Path p)
338          throws IOException {
339        return fs.open(p, bufferSize);
340      }
341    }.resolve(this, absF);
342  }
```

图 2-4-4 open 方法

2） DistributedFileSystem 通过 RPC(远程过程调用)获得存储文件的第一批 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） 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。

4） 数据从 DataNode 流向客户端。

5） 如果第一个 block 块的数据读完结束，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。

6） 当第一批 block 都读取完成，DFSInputStream 就会去 NameNode 拿下一批 blocks 的 location，然后继续读取，当所有的 block 块都读完，关闭掉所有的流。

3.4.2 文件写入

HDFS 文件写入流程图

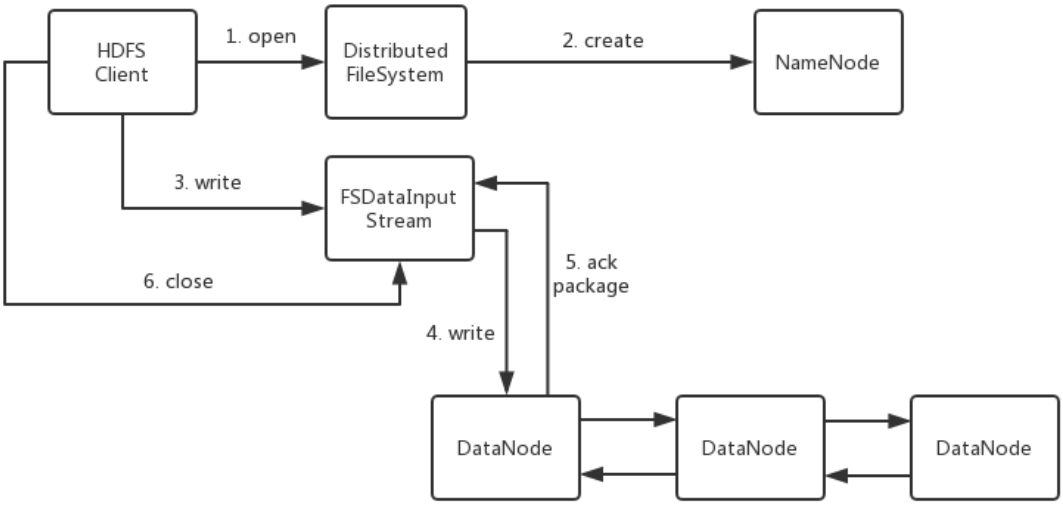

图 2-4-5 hdfs 文件写入流程图

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

```
414    @Override
415    public FSDataOutputStream create(Path f, FsPermission permission,
416        boolean overwrite, int bufferSize, short replication, long blockSize,
417        Progressable progress) throws IOException {
418      return this.create(f, permission,
419          overwrite ? EnumSet.of(CreateFlag.CREATE, CreateFlag.OVERWRITE)
420              : EnumSet.of(CreateFlag.CREATE), bufferSize, replication,
421          blockSize, progress, null);
422    }
```

图 3-4-6 create 方法

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） 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。

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。

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。

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 package 后，通知 DataNode 把文件标示为已完成。

在创建文件的操作中，有一个过程值得我们的关注，那就是分步（Staging）：

客户端创建文件的请求不会立即到达 NameNode 节点。实际上，HDFS 客户端会先缓存文件数据在本地临时文件中，当本地临时文件收集的数据块大小达到了数据块的大小时，客户端才会联系 NameNode。NameNode 将文件名插入到 HDFS 文件系统结构中，并为此数据块分配 block。然后 NameNode 将数据块所属的 DataNode 的 id 和 datablock 的 id 返回给客户端，客户端将本地临时文件中存储的数据写入到对应 DataNode 的 datablock 上。当文件关闭时，客户端将剩下没有 flush 的数据写到 datablock 上之后，再通知 NameNode 文件已经关闭了，在这个时候，NameNode 才会提交文件创建的操作到 EditLog 文件中。所以如果 NameNode 在文件关闭前挂掉了，那么这个文件信息就丢失了（虽然数据已经写到了 DataNode 上）。

在写入文件的操作中，同样有一个过程值得我们关注，那就是复制流水线（Replication Pipelining）

当客户端将数据写入 HDFS 文件时，首先将其数据写入本地缓冲区。假设 HDFS 文件的复制因子为 3（用户可指定）。当本地文件接收的数据量达到了一个数据块的大小，客户端就联系 NameNode，获得一个存储这个数据块的 DataNode 节点集合。集合中指定的 DataNode 节点存储数据块的副本。然后，客户端将数据块刷新 flush 到第一个 DataNode。第一个 DataNode 开始以小部分接收数据，将每个部分写入其本地存储库，并将该部分传输到

列表中的第二个 DataNode。第二个 DataNode 又开始接收数据块的每个部分，将该部分写入其存储库，然后将该部分刷新到第三个 DataNode。最后，第三个 DataNode 将数据写入本地存储库。因此，DataNode 可以在流水线中接收来自前一个数据的数据，并且同时将数据转发到流水线中的下一个数据。因此，数据从一个 DataNode 流水线到下一个。

HDFS 被设计成支持大文件存储，适合那些需要处理大规模数据集的应用（写入一次，多次读取）。客户端上传文件到 HDFS 文件系统时，并不会马上上传到 DataNode 节点上，而是先存储在客户端本地临时文件夹下面，当本地存储数据量的大小达到系统数据块大小时，客户端联系 NameNode 节点从 NameNode 节点（NameNode 节点将文件名存入他的文件系统中）获取数据块存储位置信息（包括副本数，存放的节点位置等信息），根据获取到的位置信息将本地临时数据块上传至对应的 DataNode 节点；当文件关闭时，客户端将剩余的数据量上传至 DataNode 节点，并告诉 NameNode 节点文件已关闭。NameNode 节点收到文件关闭的消息时将日志写入到日志记录中。

分析完 HDFS 对于文件的读取和写入过程后，发现 HDFS 中的文件 IO 操作主要是发生在 Client 和 DataNode 中。

NameNode 作为整个文件系统的 Master，负责管理整个文件系统的路径树，当需要新建文件或读取文件时，会从文件树中读取对应的路径节点的 Block 信息，发送回 Client 端。Client 通过从返回数据中得到的 DataNode 和 Block 信息，直接从 DataNode 中进行数据读取。

整个数据 IO 流程中，NameNode 只负责管理节点和 DataNode 的对应关系，涉及到 IO 操作的行为少，从而将整个文件传输压力从 NameNode 转移到了 DataNode 中。

4. 其他

4.1 HDFS 数据块

前面不断提到 HDFS 是将文件划分为多个分块作为独立的存储单元，且在 HDFS 中小于一个块大小的文件不会占据整个块的空间，而设置数据块的好处还有许多：

- 一个文件的大小可以大于任意 DataNode 的磁盘容量
- 方便对数据进行备份，提高容错能力
- 简化了存储子系统的设计

HDFS 中数据块的大小一般默认为 128M 或 64M，这是因为如果块的大小远小于 64M，数据块的数量就会增加，会增加硬盘寻道时间和 NameNode 的内存消耗；而远大于 128M，又会造成数据加载时间过长，造成监管时间难以把控，且一旦 Map 崩溃，系统恢复过程过长，同时处理的数据量越大，时间复杂度也越高。

关于块缓存机制在前面数据管理部分已有介绍，现在我们就来看看块丢失的问题。造成块丢失的原因有很多，比如误删了指定文件夹下的某些文件就会导致 HDFS 集群的元数据丢失。通过在文件系统上运行 fsck 指令，就可以查看是否有损坏的块、受影响的文件和块的具体信息。

4.2 HDFS 异常处理、

HDFS 常见的故障类型是 NameNode 故障、DataNode 故障和网络断裂。HDFS 对这三种故障的处理机制说明了它的可靠性。

对于 NameNode 故障，也就是 FsImage 和 EditLog 文件发生损坏，可以将 NameNode 配置为支持维护 FsImage 和 EditLog 的多个副本来解决。这样对 FsImage 和 EditLog 的更新都会使得每个 FsImage 和 EditLog 同步更新，虽然这回降低 NameNode 可以支持的每秒命名空间事务的速率，但是这种降低是可接受的。增加这种故障恢复能力的另一个选择是使用多个 NameNode 在 NFS 使用共享存储或分布式编辑日志。

DataNode 故障有两种，一种是从 DataNode 获取的数据块可能已经损坏，另一种是 DataNode 之间的数据不平衡。针对第一种问题，HDFS 客户端软件会对 HDFS 文件的内容进行校验和检查；对第二种问题，因为 HDFS 架构与数据重新平衡方案兼容，如果 DataNode

上的可用空间低于某个阈值，则 HDFS 可能会自动将数据从一个 DataNode 移动到另一个 DataNode；如果对特定文件的需求突然变高，也可以动态的创建其他副本并重新平衡群集中的其他数据。

而网络断裂就有可能导致 DataNode 的子集与 NameNode 失去连接。这可以由 NameNode 通过缺少 Heartbeat（心跳）消息检测出来。每个 DataNode 定期向 NameNode 发送 Heartbeat 消息，NameNode 会将最近没有 Heartbeat 的 DataNode 标记为已死，并不会将任何新的 IO 请求转发给它们。DataNode 死亡可能会导致某些块的复制因子低于其指定值，所以 NameNode 要不断跟踪需要复制的块，在必要时启动复制。这里的必要情况一般为 DataNode 可能已死、副本可能已损坏和 DataNode 上的硬盘可能会失败等。

4.3 HDFS 的缺点及改进策略

HDFS 虽然是一个不错的分布式文件系统，但仍存在着一些缺点。

一个是 HDFS 不大适合那些要求低延时访问的应用程序，因为 HDFS 是为大吞吐量数据而设计的，这需要以一定延时作为代价，而且 HDFS 是单 Master 的，所有对文件的请求都要经过它，所以必然会有存在延时，这也是主-从架构存在的问题。

要减少延时，可以使用缓存或者多 Master 的设计来降低客户端数据请求的压力，也可以通过修改 HDFS 系统内部设计，但这会牵扯到对 HDFS 大吞吐量的满足度，风险和难度较大。

再一个就是 HDFS 很难满足存储大量小文件。因为 NameNode 把元数据放在内存中，所以文件系统能容纳的文件数目是由 NameNode 的内存大小来决定的。一般来说，每个文件、文件夹和块都要占据 150 字节左右的空间，100 万个文件就至少需要 300M 的内存，虽然目前可以保证数百万文件的存储，但拓展到数亿，甚至数十亿的时候，对于目前的硬件水平难以实现。而且大量的小文件存储，也会增加线程的管理开销。

如果要让 HDFS 处理好小文件，可以借鉴 Hbase 的处理方法，利用 SequenceFile、MapFile、Har 等方式归档小文件，但要注意归档文件和原先小文件的映射关系。也可以模仿 Google，采用横向扩展，把几个 Hadoop 集群集合在一个虚拟服务器的后面，形成一个大的 Hadoop 集群。还有一个就是将单 Master 拓展为多 Master，这样也可以减轻低延时访问的压力。

最后一个缺点就是目前 Hadoop 只支持单用户写，不支持并发多用户写；可以用 append 操作在文件的末尾添加数据，但不支持在文件的任意位置进行修改等等。但这些特性的改进方案我们并没有想到，可能在以后的版本中会加入，但增加一部分特性的同时，也很可能降低 Hadoop 的效率。

4.4 其他问题

对于老师上课提出的问题：

如何将建立在 FileSystem 上的文件系统迁移到 AbstractFileSystem 上？

通过之前对源代码的阅读，我们可以发现 FileSystem 和 AbstractFileSystem 都是抽象类，且他们有许多同名的方法，比如 get、create 等。所以在进行迁移的时候，我们不需要将整个文件系统重写，而只需要对于 FileSystem 和 AbstractFileSystem 的不同方法进行重写，再对 AbstractFileSystem 里额外的抽象方法进行定义，即可实现自定义文件系统的迁移。

5. 总结及心得

葛临雪：

通过这次软件体系架构的作业，我对于 HDFS 文件系统有了更系统、更细致的了解，以前只是知道它属于 Hadoop 分布式文件系统，它的作用是什么，但现在知道了它是如何来进行数据存储的，包括它的数据存储类型、不同的存储方式和存储流程。

在整个数据的存储和交互过程中，NameNode 和 DataNode 占据了极重要的位置，所以我将大部分的时间花在理解这两者上，理解它们是什么以及它们彼此是如何通信的，基本就可以理解异构存储的基本原理。至于内存存储因为涉及到内存，所以还要知道数据块、线程池之间的关联，使用到的类大多与磁盘有关，关于这部分我的理解是内存存储提供了一个线程池以实现异步服务，防止异常时的数据丢失，比较浅层。

但在了解的过程中，我也发现了 HDFS 数据存储的一些缺点，比如说不能对文件目录存储策略做出自动的数据迁移，很难满足存储大量小文件的需求等等，但这些目前还没有很好的解决方案，毕竟如果要实现这些功能的话，HDFS 的效率和数据存储可靠性可能会受到一定影响。

除去理论知识的丰富，这次的大作业也让我感受到了思路清晰的重要性。在听了第一批同学的展示以后，我们根据老师给的评价，确定了主体思路是要从体系结构的角度来看待HDFS，所以刚开始的时候我们就决定分为几大模块功能来写，而不是从代码入手，去讲功能实现的具体细节，所以我们很幸运的没有像别的组一样陷入细节。确定好模块后，每个人就选择自己感兴趣的部分去自己找资料了解学习，最后汇总给演讲的同学，我认为这样的分工很好的提高了效率，也通过组员的进展来彼此激励，促进任务早些完成。

然后这次，因为准备软体大作业的同时我们还需要准备别的作业和考试，所以其实每个人都是抽空在做这个作业，大家都很忙很累，特别是负责演讲的同学，真的不容易，我很内疚没有替她分担更多。希望下次这种团队合作的作业，我可以多多为大家做点事。

最后，通过这次大作业，我学会了画出清晰的结构图，学会了文档的排版，学会了根据代码更好的理解理论，但仍然存在着许多不足，譬如文档写得不够通俗，不方便其他搭档的理解；效率不高，绘图缓慢。这些都是我今后需要改进的点，希望以后自己可以做的更好。

黄佳乐：

在这么一个对 hadoop 源码分析的过程中，我们小组分工明确，我的部分是 hdfs 的数据管理部分，以 HDFS 中的集中缓存管理，数据复制，和快照三个部分，结合源码分析出的类图，以及查阅官方文档上的流程图，架构图，详细分析了 HDFS 的数据管理模块的体系结构，在这个过程中，我经历了从不知道怎么下手写文档，因为虽然有了解过也使用过 HDFS，但是对于它的内部原理以前是没有涉及过的，于是就从官方文档上查阅，再从网上搜索别的博主对于这方面知识的认识与总结，多方面查找资料，去理解原来每个字都认识，但是连着一起就不认识了的原理，学习别人是怎么理解这个问题的，以及这个大问题应该怎么分成小问题来阐述。在这个过程中我总结自己的收获就是学习了一种学习能力吧。我们小组原本是打算写 AXIS2 的，但是由于我们配置所需环境出现了一些不可解决的问题，中途决定换个项目，因为问题总是会解决的，可是时间禁不住我们的磨。所以大家明智的决定换个主题完成。在这个过程中，还得感谢我的小组成员们，我的亲亲舍友们，在我一个人进度落后的情况下，并没有埋怨我，而是鼓励我，在问到我我负责部分的有些原理的时候，我自己及也迷糊的时候，大家也没有嫌弃我，而是给我时间再去捋一捋原理，再给大家转述这个原理。队长佳哲无疑是付出最多，最辛苦的，因为她要负责整个文档的汇总，要负责理解我们三个人

每部分的大概原理与机制，再上台做 ppt 展示，接受老师的询问，临雪也是和佳哲根据查阅资料商量讨论我们整个项目文档的分层与分工，创新归纳能力极好，我就听从她们俩的安排，努力完成分配给我的任务，跟上大家的脚步。谢谢大家！在这个过程中，收获颇多！

李佳哲：

通过对 Hadoop 文件系统源代码的分析，我们对 Hadoop 抽象文件系统有了最基本的认识，同时对分布式文件系统 HDFS 的架构有了一定的了解。课堂上老师讲解的可能是一些架构的概念，而通过大作业，我们对具体的架构在软件系统中的应用有了更直观的感受和理解，尤其是对于主-从架构的理解。之前可能只是了解了主从架构的模式和优点，但通过分析 HDFS 是怎样利用主从架构服务于它的功能，对主从架构的优缺点有了更清晰的认识。

在完成大作业的过程中，因为开始的比较晚，所以完成的压力很大。刚开始拿到这个大作业的时候，觉得无从下手，整个 Hadoop-2.9.2 的源代码底下包含着数十个文件夹，每个文件夹下又会延伸出更多的文件夹，如何有组织的阅读代码是我们首先要克服的问题。在阅读了老师给的 pdf 后，我们发现它只是在单纯的分析文件系统的源码，讨论具体的类和方法，并没有上升到架构和应用的层面，在经过讨论之后，我们觉得按照老师给的样本的框架完成并不能完全理解 HDFS，只是在理解代码，而没有从更宏观的角度理解问题。最终，在图书馆借阅了《Hadoop 源码分析》，阅读了 Hadoop 官方网站给出的架构和功能实例之后，我们决定按照 HDFS 的具体功能来组织和分解源代码，这样更能体现 HDFS 的架构，更利于理解，也不会陷入细节中。

经过查阅资料和讨论，最终决定从“数据存储”、“数据管理”、和“IO 操作”三个方面来组织文档，阅读代码。因为 HDFS 是一个分布式的文件系统，文件系统最重要的功能就是对数据的存储和操作，包括通过 IO 接口和外界进行数据交流。在阅读源码的过程中，我们发现主从架构完全是服务于这几个功能的，在主从架构的基础上，HDFS 实现了将文件分布式存储，并且引入了一系列额外的数据管理的方式（快照、数据复制）来克服由于主从架构导致的数据不一致、数据丢失等问题。而 HDFS 所具有的缺点，也是由于主从架构需要频繁访问主节点所导致的。

通过完成大作业，不仅让我对软件设计的架构有了更清晰的理解，同样对架构是怎样服务于软件功能有了更清楚的认识。

附录：成员分工

葛临雪 16130120103：HDFS 数据存储、异常处理模块

黄佳乐 16130120109：HDFS 数据管理模块

李佳哲 16130120112（组长）：Hadoop 抽象文件系统分析，HDFS IO 操作模块，ppt 汇报，文档整理