🤔👀HDFS 底层交互原理

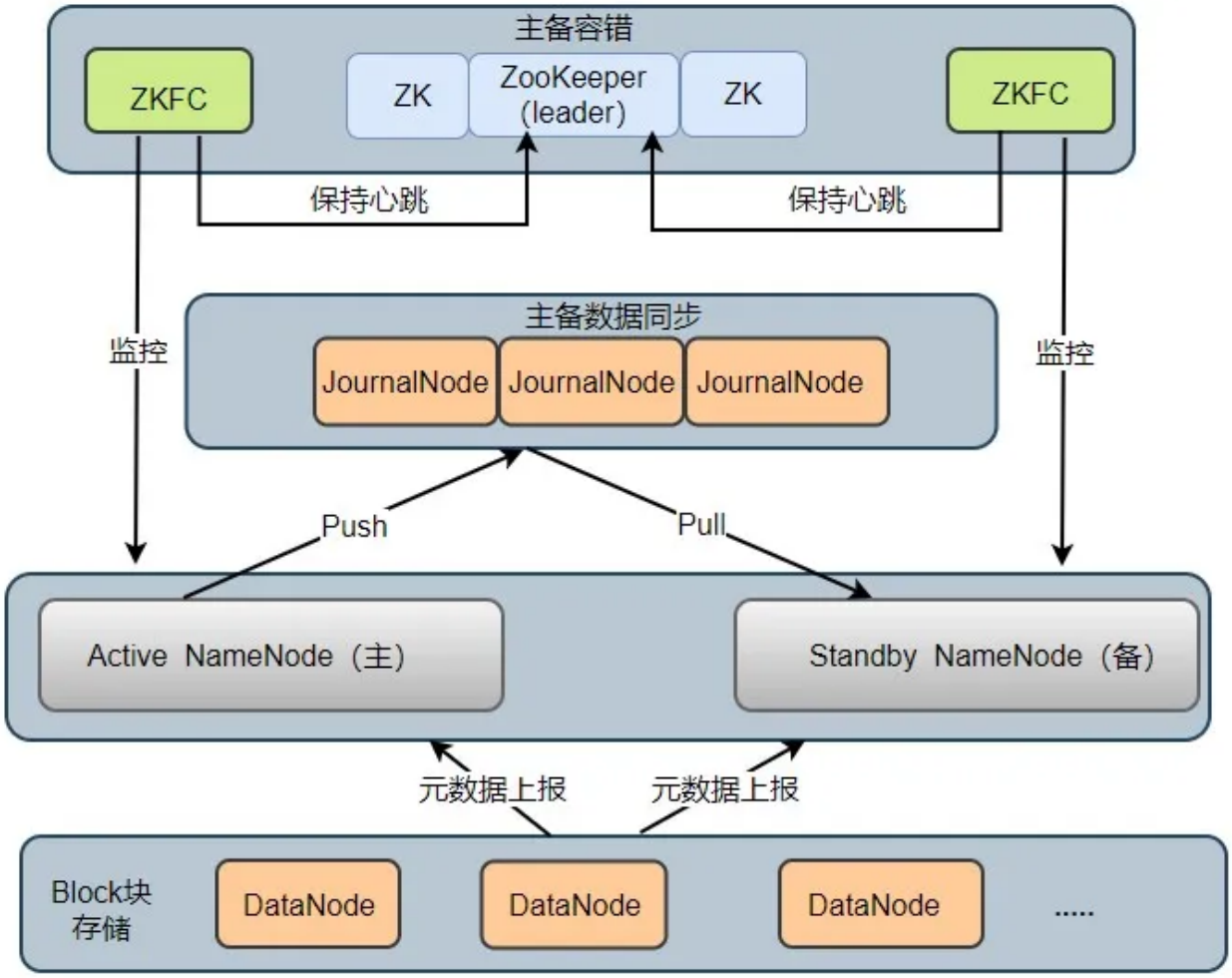
HDFS全称是 Hadoop Distribute File System，是 Hadoop最重要的组件之一，也被称为分步式存储之王。

1.HA 架构组成

*1.1HA*架构模型

在 HDFS 1.X 时，NameNode 是 HDFS 集群中可能发生单点故障的节点，集群中只有一个 NameNode，一旦 NameNode 宕机，整个集群将处于不可用的状态。

在 HDFS 2.X 时，HDFS 提出了高可用(High Availability, HA)的方案，解决了 HDFS 1.X 时的单点问题。在一个 HA 集群中，会配置两个 NameNode，一个是 Active NameNode（主），一个是 Stadby NameNode（备）。主节点负责执行所有修改命名空间的操作，备节点则执行同步操作，以保证与主节点命名空间的一致性。HA 架构模型如下图所示：



HA 集群中所包含的进程的职责各不相同。为了使得主节点和备用节点的状态一致，采用了 Quorum Journal Manger（QJM）方案解决了主备节点共享存储问题，如图 JournalNode 进程，下面依次介绍各个进程在架构中所起的作用：

- `Active NameNode`:它负责执行整个文件系统中命名空间的所有操作；维护着数据的元数据，包括文件名、副本数、文件的 BlockId 以及 Block 块所对应的节点信息；另外还接受 Client 端读写请求和 DataNode 汇报 Block 信息。
- `Standby NameNode`:它是 Active NameNode 的备用节点，一旦主节点宕机，备用节点会切换成主节点对外提供服务。它主要是监听 JournalNode Cluster 上 editlog 变化，以保证当前命名空间尽可能的与主节点同步。任意时刻，HA 集群只有一台 Active NameNode，另一个节点为 Standby NameNode。
- `JournalNode Cluster`: 用于主备节点间共享 editlog 日志文件的共享存储系统。负责存储 editlog 日志文件， 当 Active NameNode 执行了修改命名空间的操作时，它会定期将执行的操作记录在 editlog 中，并写入 JournalNode Cluster 中。Standby NameNode 会一直监听 JournalNode Cluster 上 editlog 的变化，如果发现 editlog 有改动，备用节点会读取 JournalNode 上的 editlog 并与自己当前的命名空间合并，从而实现了主备节点的数据一致性。

注意：QJM 方案是基于 Paxos 算法实现的，集群由 2N + 1 JouranlNode 进程组成，最多可以容忍 N 台 JournalNode 宕机，宕机数大于 N 台，这个算法就失效了！

- `ZKFailoverController`: ZKFC 以独立进程运行，每个 ZKFC 都监控自己负责的 NameNode，它可以实现 NameNode 自动故障切换：即当主节点异常，监控主节点的 ZKFC 则会断开与 ZooKeeper 的连接，释放分步式锁，监控备用节点的 ZKFC 进程会去获取锁，同时把备用 NameNode 切换成 主 NameNode。
- `ZooKeeper`: 为 ZKFC 进程实现自动故障转移提供统一协调服务。通过 ZooKeeper 中 Watcher 监听机制，通知 ZKFC 异常NameNode 下线；保证同一时刻只有一个主节点。

- `DataNode`：DataNode 是实际存储文件 Block 块的地方，一个 Block 块包含两个文件：一个是数据本身，一个是元数据（数据块长度、块数据的校验和、以及时间戳），DataNode 启动后会向 NameNode 注册，每 6 小时同时向主备两个 NameNode 上报所有的块信息，每 3 秒同时向主备两个 NameNode 发送一次心跳。

  DataNode 向 NameNode 汇报当前块信息的时间间隔，默认 6 小时，其配置参数名如下：

```
<property>
 <name>dfs.blockreport.intervalMsec</name>
 <value>21600000</value>
 <description>Determines block reporting interval in
milliseconds.</description>
</property>
```

*1.2HA*主备故障切换流程

HA 集群刚启动时，两个 NameNode 节点状态均为 Standby，之后两个 NameNode 节点启动 ZKFC 进程后会去 ZooKeeper 集群抢占分步式锁，成功获取分步式锁，ZooKeeper 会创建一个临时节点，成功抢占分步式锁的 NameNode 会成为 Active NameNode，ZKFC 便会实时监控自己的 NameNode。

HDFS 提供了两种 HA 状态切换方式：一种是管理员手动通过`DFSHAAdmin -faieover`执行状态切换；另一种则是自动切换。下面分别从两种情况分析故障的切换流程：

1. 主 NameNdoe 宕机后，备用 NameNode 如何升级为主节点?

   当主 NameNode 宕机后，对应的 ZKFC 进程检测到 NameNode 状态，便向 ZooKeeper 发生删除锁的命令，锁删除后，则触发一个事件回调备用 NameNode 上的 ZKFC

   ZKFC 得到消息后先去 ZooKeeper 争夺创建锁，锁创建完成后会检测原先的主 NameNode 是否真的挂掉（有可能由于网络延迟，心跳延迟），挂掉则升级备用 NameNode 为主节点，没挂掉则将原先的主节点降级为备用节点，将自己对应的 NameNode 升级为主节点。

2. 主 NameNode 上的 ZKFC 进程挂掉，主 NameNode 没挂，如何切换?

   ZKFC 挂掉后，ZKFC 和 ZooKeeper 之间 TCP 链接会随之断开，session 也会随之消失，锁被删除，触发一个事件回调备用 NameNode ZKFC，ZKFC 得到消息后会先去 ZooKeeper 争夺创建锁，锁创建完成后也会检测原先的主 NameNode 是否真的挂掉，挂掉则升级 备用 NameNode 为主节点，没挂掉则将主节点降级为备用节点，将自己对应的 NameNode 升级为主节点。

1.3Block、*packet*及*chunk* 概念

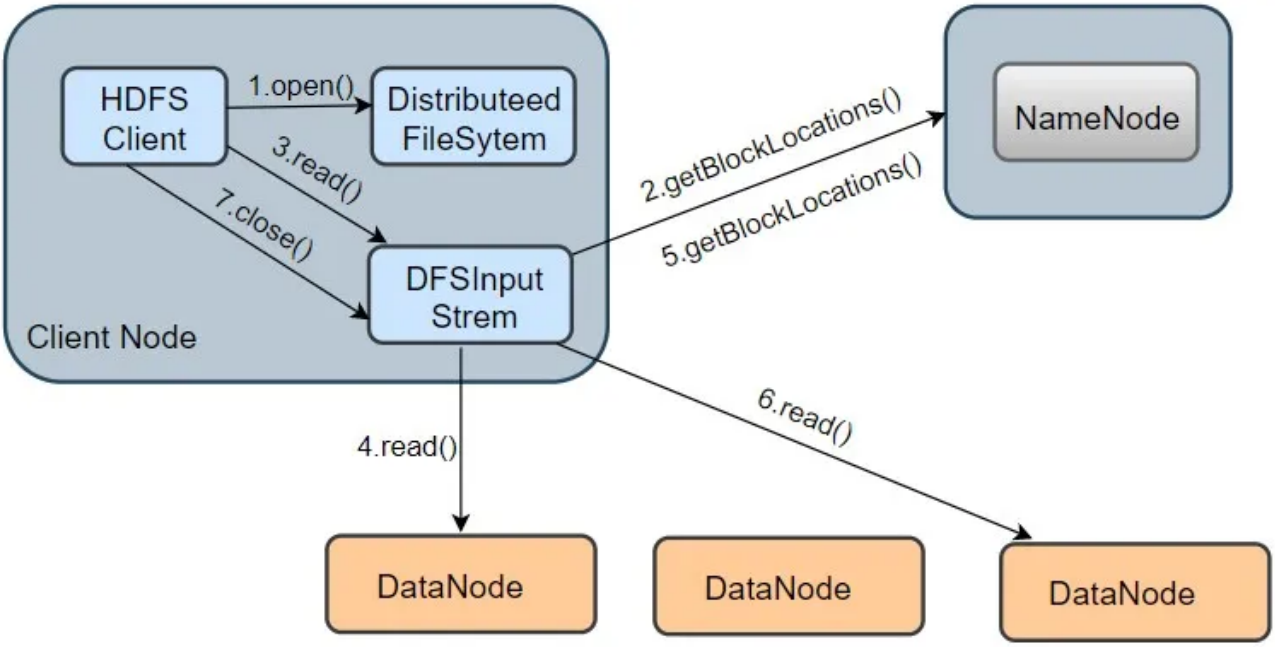
在 HDFS 中，文件存储是按照数据块（Block）为单位进行存储的，在读写数据时，`DFSOutputStream`使用 Packet 类来封装一个数据包。每个 Packet 包含了若干个 chunk 和对应的 checksum。

- **Block:** HDFS 上的文件都是分块存储的，即把一个文件物理划分为一个 Block 块存储。Hadoop 2.X/3.X 默认块大小为 128 M，1.X 为 64M.
- **Packet:** 是 Client 端向 DataNode 或 DataNode 的 Pipline 之间传输数据的基本单位，默认 64 KB
- **Chunk:** Chunk 是最小的单位，它是 Client 向 DataNode 或 DataNode PipLine 之间进行数据校验的基本单位，默认 512 Byte ，因为用作校验，所以每个 Chunk 需要带有 4 Byte 的校验位，实际上每个 Chunk 写入 Packtet 的大小为

516 Byte。

2.源码级读写流程

2.1HDFS 读流程



我们以从 HDFS 读取一个 information.txt 文件为例，其读取流程如上图所示，分为以下几个步骤：

1. **打开 information.txt 文件**：首先客户端调用 `DistributedFileSystem.open()` 方法打开文件，这个方法在底层会调用`DFSclient.open()` 方法，该方法会返回一个 HdfsDataInputStream 对象用于读取数据块。但实际上真正读取数据的是 DFSInputStream ，而 HdfsDataInputStream 是 DFSInputStream 的装饰类（`new HdfsDataInputStream（DFSInputStream）`）。
2. **从 NameNode 获取存储 information.txt 文件数据块的 DataNode 地址**：即获取组成 information.txt block 块信息。在构造输出流 DFSInputStream 时，会通过调用 `getBlockLocations()` 方法向 NameNode 节点获取组成 information.txt 的 block 的位置信息，并且 block 的位置信息是按照与客户端的距离远近排好序。
3. **连接 DataNode 读取数据块：** 客户端通过调用 `DFSInputStream.read()` 方法，连接到离客户端最近的一个 DataNode 读取 Block 块，数据会以数据包（packet）为单位从 DataNode 通过流式接口传到客户端，直到一个数据块读取完成；`DFSInputStream`会再次调用 getBlockLocations() 方法，获取下一个最优节点上的数据块位置。
4. **直到所有文件读取完成，调用 close() 方法，关闭输入流，释放资源。**

从上述流程可知，整个过程最主要涉及到 `open()`、`read()`两个方法（**其它方法都是在这两个方法的调用链中调用，如`getBlockLocations()`**），下面依次介绍这2个方法的实现。

> 注：本文是以 hadoop-3.1.3 源码为基础！

- **open（）方法**

事实上，在调用 `DistributedFileSystem.open()`方法时，底层调用的是 `DFSClient.open()`方法打开文件，并构造 `DFSInputStream` 输入流对象。

```
public DFSInputStream open(String src, int buffersize, boolean verifyChecksum)
      throws IOException {
    //检查DFSClicent 的运行状况
    checkOpen();
// 从 namenode 获取 block 位置信息，并存到 LocatedBlocks 对象中,最终传给 DFSInputStream 的构造方法
    try (TraceScope ignored = newPathTraceScope("newDFSInputStream", src)) {
      LocatedBlocks locatedBlocks = getLocatedBlocks(src, 0);
      //调用 openInternal 方法，获取输入流
      return openInternal(locatedBlocks, src, verifyChecksum);
    }
  }
```

- 整个 open()方法分为两部分：
- - 第一部分是，调用 checkOpen()方法检查 DFSClient 的运行状况，调用getLocateBlocks()方法，获取 block 的位置消息
  - 第二部分是，调用openInternal()方法，获取输入流。
- **openInternal( )方法**

```
private DFSInputStream openInternal(LocatedBlocks locatedBlocks, String src,
      boolean verifyChecksum) throws IOException {
    if (locatedBlocks != null) {
      //获取纠删码策略，纠删码是 Hadoop 3.x 的新特性，默认不启用纠删码策略
      ErasureCodingPolicy ecPolicy = locatedBlocks.getErasureCodingPolicy();
      if (ecPolicy != null) {
        //如果用户指定了纠删码策略，将返回一个 DFSStripedInputStream 对象
        //DFSStripedInputStream 会将数据逻辑字节范围的请求转换为存储在 DataNode 上的内部块
        return new DFSStripedInputStream(this, src, verifyChecksum, ecPolicy,
            locatedBlocks);
      }
     //如果未指定纠删码策略，调用 DFSInputStream 的构造方法，并且返回该 DFSInputStream 的对象
      return new DFSInputStream(this, src, verifyChecksum, locatedBlocks);
    } else {
      throw new IOException("Cannot open filename " + src);
    }
  }
```

- **DFSInputStream 构造方法**

```
DFSInputStream(DFSClient dfsClient, String src, boolean verifyChecksum,
      LocatedBlocks locatedBlocks) throws IOException {
    this.dfsClient = dfsClient;
    this.verifyChecksum = verifyChecksum;
    this.src = src;
    synchronized (infoLock) {
      this.cachingStrategy = dfsClient.getDefaultReadCachingStrategy();
    }
    this.locatedBlocks = locatedBlocks;
    //调用 openInfo 方法，参数：refreshLocatedBlocks，是否要更新 locateBlocks 属性。
    openInfo(false);
  }
```

- 构造方法做了2件事：

  第一部分是初始化 DFSInputStream 属性，其中 verifyChecksum 含义是：读取数据时是否进行校验，cachingStrategy，指的是缓存策略。

  第二部分，调用 openInfo()方法。

> 思考：为甚么要更新最后一个数据块长度?
>
> 因为可能会有这种情况出现，当客户端在读取文件时，最后一个文件块可能还在构建的状态（正在被写入），Datanode 还未上报最后一个文件块，那么 namenode 所保存的数据块长度有可能小于 Datanode实际存储的数据块长度，所以需要与 Datanode 通信以确认最后一个数据块的真实长度。

获取到 DFSInputStream 流对象后，并且得到了文件的所有 Block 块的位置信息，接下来调用`read()`方法，从 DataNode 读取数据块。

> 注：在`openInfo()` 方法

- - 在 `openInfp()` 中，会从 namenode 获取当前正在读取文件的最后一个数据块的长度 `lastBlockBeingWrittenLength`，如果返回的最后一个数据块的长度为 -1，这是一种特殊情况：即集群刚重启，DataNode 可能还没有向 NN 进行完整的数据块汇报，这时部分数据块位置信息还获取不到，也获取不到这些块的长度，则默认会重试 3 次，默认每次等待 4 秒，重新去获取文件对应的数据块的位置信息以及最后数据块长度；如果最后一个数据块的长度不为 -1，则表明，最后一个数据块已经是完整状态。

- **read()方法**

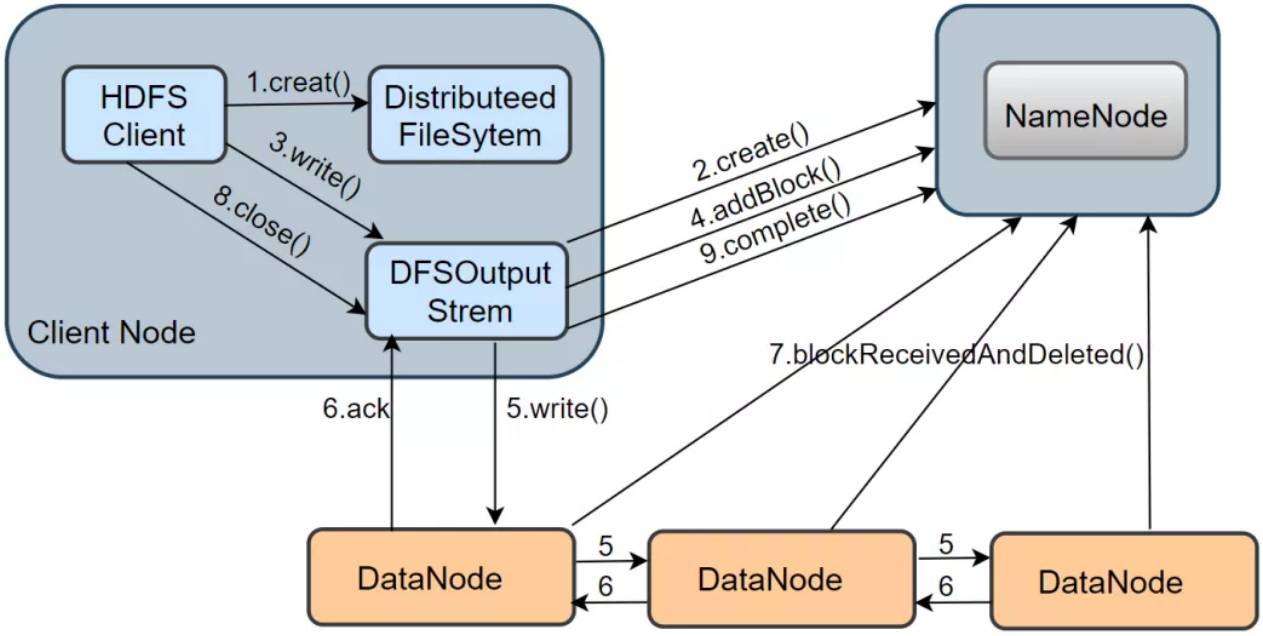
```
public synchronized int read(@Nonnull final byte buf[], int off, int len)
      throws IOException {
    //验证输入的参数是否可用
    validatePositionedReadArgs(pos, buf, off, len);
    if (len == 0) {
      return 0;
    }
    //构造字节数组作为容器
    ReaderStrategy byteArrayReader =
        new ByteArrayStrategy(buf, off, len, readStatistics, dfsClient);
    //调用 readWithStrategy()方法读取数据
    return readWithStrategy(byteArrayReader);
  }
```

- 当用户代码调用`read()`方法时，其底层调用的是 `DFSInputStream.read()`方法。该方法从输入流的 off 位置开始读取，读取 len 个字节，然后存入 buf 字节数组中。源码中构造了一个 ByteArrayStrategy 对象，该对象封装了 5 个属性，分别是：字节数组 buf，读取到的字节存入该字节数组；off，读取的偏移量；len，将要读取的目标长度；readStatistics，统计计数器，客户端。最后通过调用 `readWithStrategy()`方法去读取文件数据块的数据。

> 总结：HDFS 读取一个文件，调用流程如下：（中间涉及到的部分方法未列出）
>
> usercode 调用 `open()` ---> `DistributedFileSystem.open()` ---> `DFSClient.open()` ---> 返回一个 DFSInputStream 对象给 DistributedFileSystem ---> `new hdfsDataInputStream(DFSInputStream)` 并返回给用户；
>
> usercode 调用 `read()` ---> 底层`DFSIputStream.read()` ---> `readWithStrategy(bytArrayReader)`

2.2HDFS 写流程

介绍完 HDFS 读的流程，接下来看看一个文件的写操作的实现。从下图中可以看出，HDFS 写流程涉及的方法比较多，过程也比较复杂。



1.在 namenode 创建文件： 当 client 写一个新文件时，首先会调用 `DistributeedFileSytem.creat()` 方法，`DistributeFileSystem` 是客户端创建的一个对象，在收到 creat 命令之后，`DistributeFileSystem` 通过 RPC 与 NameNode 通信，让它在文件系统的 namespace 创建一个独立的新文件；namenode 会先确认文件是否存在以及客户端是否有权限，确认成功后，会返回一个 HdfsDataOutputStream 对象，与读流程类似，这个对象底层包装了一个 DFSOutputStream 对象，它才是写数据的真正执行者。

2.建立数据流 pipeline 管道： 客户端得到一个输出流对象，还需要通过调用 `ClientProtocol.addBlock()`向 namenode 申请新的空数据块，addBlock( ) 会返回一个 LocateBlock 对象，该对象保存了可写入的 DataNode 的信息，并构成一个 pipeline，默认是有三个 DataNode 组成。

3.通过数据流管道写数据: 当 `DFSOutputStream`调用 `write()`方法把数据写入时，数据会先被缓存在一个缓冲区中，写入的数据会被切分成多个数据包，每当达到一个数据包长度（默认65536字节）时，

`DFSOutputStream` 会构造一个 Packet 对象保存这个要发送的数据包；新构造的 Packet 对象会被放到 `DFSOutputStream`维护的 dataQueue 队列中，`DataStreamer` 线程会从 dataQueue 队列中取出 Packet 对象，通过底层 IO 流发送到 pipeline 中的第一个 DataNode，然后继续将所有的包转到第二个 DataNode 中，以此类推。发送完毕后，

这个 Packet 会被移出 dataQueue，放入 `DFSOutputStream` 维护的确认队列 ackQueue 中，该队列等待下游 DataNode 的写入确认。当一个包已经被 pipeline 中所有的 DataNode 确认了写入磁盘成功，这个数据包才会从确认队列中移除。

4.关闭输入流并提交文件： 当客户端完成了整个文件中所有的数据块的写操作之后，会调用 close() 方法关闭输出流，客户端还会调用 ClientProtoclo.complete( ) 方法通知 NameNode 提交这个文件中的所有数据块，

NameNode 还会确认该文件的备份数是否满足要求。对于 DataNode 而言，它会调用 blockReceivedAndDelete() 方法向 NameNode 汇报，NameNode 会更新内存中的数据块与数据节点的对应关系。

从上述流程来看，整个写流程主要涉及到了 creat()、write()这些方法，下面着重介绍下这两个方法的实现。当调用 DistributeedFileSytem.creat() 方法时，其底层调用的其实是 DFSClient.create()方法，其源码如下：

- **create( )方法**

```
public DFSOutputStream create(String src, FsPermission permission,
          EnumSet<CreateFlag> flag, boolean createParent,
                    short replication,long blockSize,
                    Progressable progress, int buffersize,
                    ChecksumOpt checksumOpt,
                    InetSocketAddress[] favoredNodes,
                    String ecPolicyName) throws IOException {
    //检查客户端是否已经打开
    checkOpen();
    final FsPermission masked = applyUMask(permission);
    LOG.debug("{}: masked={}", src, masked);
    //调用 DFSOutputStream.newStreamForCreate()创建输出流对象
    final DFSOutputStream result = DFSOutputStream.newStreamForCreate(this,
        src, masked, flag, createParent, replication, blockSize, progress,
        dfsClientConf.createChecksum(checksumOpt),
        getFavoredNodesStr(favoredNodes), ecPolicyName);
    //获取 HDFS 文件的租约
    beginFileLease(result.getFileId(), result);
    return result;
  }
```

- DistributeFileSystem.create()在底层会调用 DFSClient.create()方法。该方法主要完成三件事：

> 租约：指的是租约持有者在规定时间内获得该文件权限（写文件权限）的合同

- - 第一，检查客户端是否已经打开
  - 第二，调用静态的 newStreamForCreate() 方法，通过 RPC 与 NameNode 通信创建新文件，并构建出 DFSOutputStream流
  - 第三，执行 beginFileLease() 方法，获取新J建文件的租约
- **newStreamForCreate() 方法**

```
static DFSOutputStream newStreamForCreate(DFSClient dfsClient, String src,
      FsPermission masked, EnumSet<CreateFlag> flag, boolean createParent,
      short replication, long blockSize, Progressable progress,
      DataChecksum checksum, String[] favoredNodes, String ecPolicyName)
      throws IOException {

    try (TraceScope ignored =
             dfsClient.newPathTraceScope("newStreamForCreate", src)) {
      HdfsFileStatus stat = null;

    // 如果发生异常，并且异常为 RetryStartFileException ，便重新调用create()方法，重试次数为 10
      boolean shouldRetry = true;
      //重试次数为 10
      int retryCount = CREATE_RETRY_COUNT;
      while (shouldRetry) {
        shouldRetry = false;
        try {
           //调用 ClientProtocol.create() 方法，在命名空间中创建 HDFS 文件
          stat = dfsClient.namenode.create(src, masked, dfsClient.clientName,
              new EnumSetWritable<>(flag), createParent, replication,
```

```
                blockSize, SUPPORTED_CRYPTO_VERSIONS, ecPolicyName);
            break;
          } catch (RemoteException re) {
            IOException e = re.unwrapRemoteException(AccessControlException.class,
                //....此处省略了部分异常类型
                UnknownCryptoProtocolVersionException.class);
            if (e instanceof RetryStartFileException) {
                //如果发生异常，判断异常是否为 RetryStartFileException
              if (retryCount > 0) {
                shouldRetry = true;
                retryCount--;
              } else {
                throw new IOException("Too many retries because of encryption" +
                    " zone operations", e);
              }
            } else {
              throw e;
            }
          }
        }
        Preconditions.checkNotNull(stat, "HdfsFileStatus should not be null!");
        final DFSOutputStream out;
        if(stat.getErasureCodingPolicy() != null) {
        //如果用户指定了纠删码策略，将创建一个 DFSStripedOutputStream 对象
          out = new DFSStripedOutputStream(dfsClient, src, stat,
              flag, progress, checksum, favoredNodes);
        } else {
          //如果没指定纠删码策略，调用构造方法创建一个 DFSOutputStream 对象
          out = new DFSOutputStream(dfsClient, src, stat,
              flag, progress, checksum, favoredNodes, true);
        }
        //启动输出流对象的 Datastreamer 线程
        out.start();
        return out;
      }
    }
```

- newStreamForCreate()方法总共涉及三个部分：

  当构建完 DFSOutputStream 输出流时，客户端调用 **write()** 方法把数据包写入 dataQueue 队列，在将数据包发送到 DataNode 之前，DataStreamer会向 NameNode 申请分配一个新的数据块

  然后建立写这个数据块的数据流管道（pipeline），之后DataStreamer 会从 dataQueue 队列取出数据包，通过 pipeline 依次发送给各个 DataNode。每个数据包（packet）都有对应的序列号，当一个数据块中所有的数据包都发送完毕，

  并且都得到了 ack 消息确认后，Datastreamer会将当前数据块的 pipeline 关闭。通过不断循环上述过程，直到该文件（一个文件会被切分为多个 Block）的所有数据块都写完成。

- - 调用 ClientProtocol.create()方法，创建文件，如果发生异常为 RetryStartFileException ，则默认重试10次
  - 调用 DFSStripedOutputStream 或 DFSOutputStream 构造方法，构造输出流对象
  - 启动 Datastreamer 线程，Datastreamer 是 DFSOutputStream 中的一个内部类，负责构建 pipeline 管道，并将数据包发送到 pipeline 中的第一个 DataNode
- **writeChunk()方法**

```
protected synchronized void writeChunk(ByteBuffer buffer, int len,
      byte[] checksum, int ckoff, int cklen) throws IOException {
    writeChunkPrepare(len, ckoff, cklen);

    //将当前校验数据、校验块写入数据包中
    currentPacket.writeChecksum(checksum, ckoff, cklen);
```

```
    currentPacket.writeData(buffer, len);
    currentPacket.incNumChunks();
    getStreamer().incBytesCurBlock(len);

    // 如果当前数据包已经满了，或者写满了一个数据块，则将当前数据包放入发送队列中
    if (currentPacket.getNumChunks() == currentPacket.getMaxChunks() ||
            getStreamer().getBytesCurBlock() == blockSize) {
      enqueueCurrentPacketFull();
    }
  }
```

最终写数据调用都的是 `writeChunk ()`方法，其会首先调用 `checkChunkPrepare()`构造一个 Packet 对象保存数据包，

然后调用`writeCheckSum()`和`writeData()`方法，将校验块数据和校验和写入 Packet 对象中。

当 Packet 对象写满时（每个数据包都可以写入 maxChunks 个校验块），则调用 `enqueueCurrentPacketFull()`方法，将当前的 Packet 对象放入 dataQueue 队列中，等待 DataStreamer 线程的处理。

如果当前数据块中的所有数据都已经发送完毕，则发送一个空数据包标识所有数据已经发送完毕。

3.HDFS 如何保证可用性?

在 1.1 节中已经阐述了 HDFS 的高可用的架构，分别涉及到 `NameNode`，`DataNode`,`Journalnode`,`ZKFC`等组件。所以，在谈及 HDFS 如何保证可用性，要从多个方面去回答。

- 在 Hadoop 2.X 时，主备 NameNode 节点通过 JournalNode 的数据同步，来保证数据一致性，2个 ZKFC 进程负责各自的 NameNode 健康监控，从而实现了 NameNode 的高可用。Hadoop 3.X 时，NameNode 数量可以大于等于 2。

- 对于 `JournalNode` 来讲，也是分布式的，保证了可用性。因为有选举机制，所以 JournalNode 个数 一般都为 2N+1 个。在 主`NameNode`向 `JournalNode`写入 editlog 文件时，当有一半以上的（≥N+1） `JournalNode`返回写操作成功时即认为该次写成功。所以 `JournalNode`集群能容忍最多 N 台节点宕掉，如果多于 N 台机器挂掉，服务才不可用。

- ZKFC 主要辅助 ZooKeeper 做 Namenode 的健康监控，能够保证故障自动转移，它是部署在两台 NameNode 节点上的独立的进程。此外，ZooKeeper 集群也是一个独立的分布式系统，它通过 Zab 协议来保证数据一致，和主备节点的选举切换等机制来保证可用性。

- `DataNode` 节点主要负责存储数据，通过 3 副本策略来保证数据的完整性，所以其最大可容忍 2 台 `DataNode` 挂掉，同时 `NameNode` 会保证副本的数量。

- 最后，关于数据的可用性保证，HDFS 提供了**数据完整性**校验的机制。当客户端创建文件时，它会计算每个文件的数据块的`checknums`，也就是校验和，并存储在 NameNode 中。当客户端去读取文件时，会验证从 DataNode

接收的数据块的校验和，如果校验和不一致，说明该数据块已经损坏，此时客户端会选择从其它 DataNode 获取该数据块的可用副本。