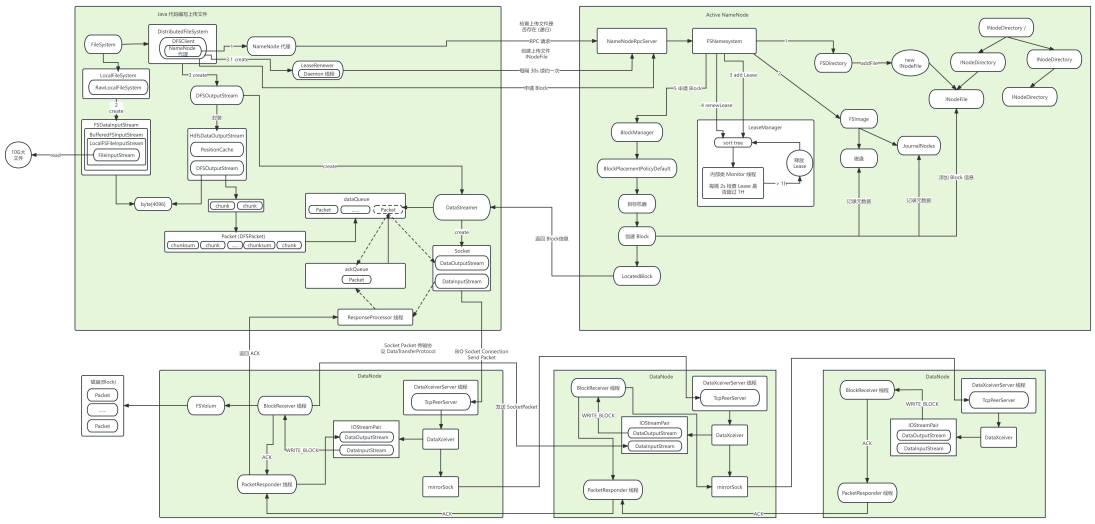
hadoop-hdfs 文件上传源码分析

一 案例

```
public class UploadMain {

    public static void main(String[] args) throws Exception {

        System.setProperty("HADOOP_USER_NAME", "tanbs");

        Configuration conf = new Configuration();

        // DistributedFileSystem (初始化操作在元数据管理源码分析讲过  这里不在累赘)
        FileSystem fileSystem = FileSystem.get(conf);

        fileSystem.copyFromLocalFile(new Path("./LICENSE.txt"), new Path("/mkdir/"));

    }

}
```

二 文件上传源码分析



2.1 DistributedFileSystem 的初始化

备注：参考元数据管理源码分析

2.1.1 DFSClient 初始化

备注：DFSClient 在创建以及初始化过程中获取 NameNode 的代理

2.2 执行上传

```
fileSystem.copyFromLocalFile(
// 本地上传文件路径
new Path("./LICENSE.txt"),
// HDFS 文件路径
 new Path("/mkdir/"));
```

```
/**
     * The src file is on the local disk.  Add it to the filesystem at
     * the given dst name.
     * delSrc indicates if the source should be removed
     *
     * @param delSrc    whether to delete the src
     * @param overwrite whether to overwrite an existing file
     * @param src       path
     * @param dst       path
     * @throws IOException IO failure
     */
    public void copyFromLocalFile(boolean delSrc, boolean overwrite,
                                  Path src, Path dst)
            throws IOException {
        // 创建 FileSystem 的入参 conf
        Configuration conf = getConf();
        // 往下追
        FileUtil.copy(
                // 获取本地 LocalFileSystem (跟获取 DistributedFileSystem 类似)
                getLocal(conf),
                src,
                this,
                dst,
                delSrc,
                overwrite,
```

```
                    conf);
    }
```

2.2.1 创建并初始化 LocalFileSystem (本地文件系统)

2.2.1.1 创建 LocalFileSystem

```
public LocalFileSystem() {
        // 往下追
        this(
                // 创建 RawLocalFileSystem
                new RawLocalFileSystem()
        );
    }
```

2.2.1.2 初始化 LocalFileSystem

```
@Override
    public void initialize(URI name, Configuration conf) throws IOException {
        if (fs.getConf() == null) {
            fs.initialize(name, conf);
        }
        String scheme = name.getScheme();
        if (!scheme.equals(fs.getUri().getScheme())) {
            swapScheme = scheme;
        }
    }
```

2.2.2 调用 FileUtil.copy()

```
/**
     * Copy files between FileSystems.
     */
    public static boolean copy(FileSystem srcFS, Path src,
                               FileSystem dstFS, Path dst,
                               boolean deleteSource,
```

```
                                    boolean overwrite,
                                    Configuration conf) throws IOException {
        // 调用 FilterFileSystem.getFileStatus() 最终调用 RawLocalFileSystem.getFileStatus()
        // 返回 src 上传文件的描述信息 (文件长度、是否为目录、副本数、Block 大小)
        FileStatus fileStatus = srcFS.getFileStatus(src);

        // 往下追
        return copy(srcFS, fileStatus, dstFS, dst, deleteSource, overwrite, conf);
    }
```

```
/**
     * Copy files between FileSystems.
     */
    public static boolean copy(FileSystem srcFS, FileStatus srcStatus,
                                    FileSystem dstFS, Path dst,
                                    boolean deleteSource,
                                    boolean overwrite,
                                    Configuration conf) throws IOException {
        // 获取上传文件的路径
        Path src = srcStatus.getPath();
        // 检查上传目标文件是否已经存在 (发送 Rpc 请求 NameNode 获取 dst 的描述信息再判断上传文件是否存在还是发送 RPC 请求)
        // 最终返回 dst = dst/src
        dst = checkDest(src.getName(), dstFS, dst, overwrite);

        // 如果 src path 是否目录情况下
        if (srcStatus.isDirectory()) {
            checkDependencies(srcFS, src, dstFS, dst);
            if (!dstFS.mkdirs(dst)) {
                return false;
            }
            FileStatus contents[] = srcFS.listStatus(src);
            for (int i = 0; i < contents.length; i++) {
                copy(srcFS, contents[i], dstFS,
                        new Path(dst, contents[i].getPath().getName()),
                        deleteSource, overwrite, conf);
            }
        } else {
            // 如果 src path 是文件
            InputStream in = null;
            OutputStream out = null;
            try {
                // 打开 src path 输入流
                // 最终返回 FSDataInputStream (继承 DataInputStream)
```

```
                    // FSDataInputStream 构造函数传入 BufferedFSInputStream( 继承 BufferedInputStream)
                    // BufferedFSInputStream 构造函数传入 LocalFSFileInputStream(extends FSInputStream extends InputStream )
                    // LocalFSFileInputStream 构造函数传入 Java IO 的 FileInputStream(src)
                    in = srcFS.open(src);

                    // 创建 HDFS 的输出流
                    // 包装 DFSOutputStream 返回 HdfsDataOutputStream
                    out = dstFS.create(dst, overwrite);

                    // 执行流拷贝
                    IOUtils.copyBytes(in, out, conf, true);
                } catch (IOException e) {
                    IOUtils.closeStream(out);
                    IOUtils.closeStream(in);
                    throw e;
                }
            }
            if (deleteSource) {
                return srcFS.delete(src, true);
            } else {
                return true;
            }

        }
```

2.2.2.1 检查上传目标文件是否已经存在 (发送 RPC 请求)

```
private static Path checkDest(String srcName, FileSystem dstFS, Path dst,
                                        boolean overwrite) throws IOException {
            FileStatus sdst;
            try {
                // 发送 Rpc 请求 NameNode 获取 dst 的描述信息
                sdst = dstFS.getFileStatus(dst);
            } catch (FileNotFoundException e) {
                sdst = null;
            }
            if (null != sdst) {
                if (sdst.isDirectory()) {
                    if (null == srcName) {
                        throw new PathIsDirectoryException(dst.toString());
```

```
                    }
                    // 递归
                    return checkDest(null, dstFS, new Path(dst, srcName), overwrite);
                } else if (!overwrite) {
                    throw new PathExistsException(dst.toString(),
                            "Target " + dst + " already exists");
                }
            }
            return dst;
        }
```

```
    /**
     * Returns the stat information about the file.
     *
     * @throws FileNotFoundException if the file does not exist.
     */
    @Override
    public FileStatus getFileStatus(Path f) throws IOException {
        statistics.incrementReadOps(1);
        storageStatistics.incrementOpCounter(OpType.GET_FILE_STATUS);
        Path absF = fixRelativePart(f);
        return new FileSystemLinkResolver<FileStatus>() {
            @Override
            public FileStatus doCall(final Path p) throws IOException {
                // 往下追
                HdfsFileStatus fi = dfs.getFileInfo(getPathName(p));
                if (fi != null) {
                    return fi.makeQualified(getUri(), p);
                } else {
                    throw new FileNotFoundException("File does not exist: " + p);
                }
            }

            @Override
            public FileStatus next(final FileSystem fs, final Path p)
                    throws IOException {
                return fs.getFileStatus(p);
            }
        }.resolve(this, absF);
    }
```

```
/**
     * Get the file info for a specific file or directory.
     *
```

```
     * @param src The string representation of the path to the file
     * @return object containing information regarding the file
     * or null if file not found
     * @see ClientProtocol#getFileInfo(String) for description of exceptions
     */
    public HdfsFileStatus getFileInfo(String src) throws IOException {
        checkOpen();
        try (TraceScope ignored = newPathTraceScope("getFileInfo", src)) {
            // 发送 Rpc 请求 NameNode 返回 src 的描述信息
            return namenode.getFileInfo(src);
        } catch (RemoteException re) {
            throw re.unwrapRemoteException(AccessControlException.class,
                    FileNotFoundException.class,
                    UnresolvedPathException.class);
        }
    }
```

2.2.2.2 打开本地文件系统上传文件输入流

```
/**
     * Opens an FSDataInputStream at the indicated Path.
     *
     * @param f the file to open
     * @throws IOException IO failure
     */
    public FSDataInputStream open(Path f) throws IOException {
        // 往下追（最终调用 RawLocalFileSystem.open())
        return open(f, getConf().getInt(IO_FILE_BUFFER_SIZE_KEY,
                IO_FILE_BUFFER_SIZE_DEFAULT));
    }
```

```
@Override
    public FSDataInputStream open(Path f, int bufferSize) throws IOException {
        getFileStatus(f);
        // 创建 FSDataInputStream
        return new FSDataInputStream(
                // 创建 BufferedFSInputStream
                new BufferedFSInputStream(
                        // 创建 LocalFSFileInputStream
                        new LocalFSFileInputStream(f),
                        // 4096
```

```
                                bufferSize));
    }
```

2.2.2.3 创建 HDFS 文件系统输出流

```
/**
     * Create an FSDataOutputStream at the indicated Path.
     *
     * @param f                the file to create
     * @param overwrite if a file with this name already exists, then if true,
     *                                the file will be overwritten, and if false an exception will be thrown.
     * @throws IOException IO failure
     */
    public FSDataOutputStream create(Path f, boolean overwrite)
            throws IOException {
        // 往下追
        return create(
                // 上传目标文件路径
                f,
                // true
                overwrite,
                // 4096
                getConf().getInt(IO_FILE_BUFFER_SIZE_KEY,
                        IO_FILE_BUFFER_SIZE_DEFAULT),
                // 3
                getDefaultReplication(f),
                // 128 MB
                getDefaultBlockSize(f));
    }
```

```
/**
     * Create an FSDataOutputStream at the indicated Path.
     *
     * @param f                 the file name to open
     * @param overwrite      if a file with this name already exists, then if true,
     *                                the file will be overwritten, and if false an error will be thrown.
     * @param bufferSize    the size of the buffer to be used.
     * @param replication required block replication for the file.
     * @throws IOException IO failure
     */
    public FSDataOutputStream create(Path f,
```

```
                                            boolean overwrite,
                                            int bufferSize,
                                            short replication,
                                            long blockSize) throws IOException {
        // 往下追
        return create(f, overwrite, bufferSize, replication, blockSize, null);
    }
```

```
/**
     * Create an FSDataOutputStream at the indicated Path with write-progress
     * reporting.
     *
     * @param f                 the file name to open
     * @param overwrite      if a file with this name already exists, then if true,
     *                          the file will be overwritten, and if false an error will be thrown.
     * @param bufferSize    the size of the buffer to be used.
     * @param replication required block replication for the file.
     * @throws IOException IO failure
     */
    public FSDataOutputStream create(Path f,
                                            boolean overwrite,
                                            int bufferSize,
                                            short replication,
                                            long blockSize,
                                            Progressable progress
    ) throws IOException {
        // 往下追 (最终调用 DistributedFileSystem.create())
        return this.create(
                f,
                FsCreateModes.applyUMask(
                        FsPermission.getFileDefault(),
                        FsPermission.getUMask(getConf())
                ),
                overwrite, bufferSize, replication, blockSize, progress);
    }
```

```
@Override
    public FSDataOutputStream create(Path f, FsPermission permission,
                                            boolean overwrite, int bufferSize, short replication,
long blockSize,
                                            Progressable progress) throws IOException {
        // 往下追
        return this.create(f, permission,
                // overwrite = true
```

```
                overwrite ? EnumSet.of(CreateFlag.CREATE, CreateFlag.OVERWRITE)
                        : EnumSet.of(CreateFlag.CREATE), bufferSize, replication,
                blockSize, progress, null);
    }
```

```
@Override
    public FSDataOutputStream create(final Path f, final FsPermission permission,
                                     final EnumSet<CreateFlag> cflags, final int
bufferSize,
                                     final short replication, final long blockSize,
                                     final Progressable progress, final ChecksumOpt
checksumOpt)
            throws IOException {
        statistics.incrementWriteOps(1);
        storageStatistics.incrementOpCounter(OpType.CREATE);

        // 上传文件到 HDFS 的绝对路径
        Path absF = fixRelativePart(f);

        return new FileSystemLinkResolver<FSDataOutputStream>() {
            @Override
            public FSDataOutputStream doCall(final Path p) throws IOException {
                // 创建 DFSOutputStream
                final DFSOutputStream dfsos = dfs.create(
                        // 上传文件绝对路径的 path 比如 /opt/app/hadoop/demo.txt
                        getPathName(p),
                        permission,
                        cflags, replication, blockSize, progress, bufferSize,
                        checksumOpt);

                // 包装 DFSOutputStream 返回 HdfsDataOutputStream
                return dfs.createWrappedOutputStream(dfsos, statistics);
            }

            @Override
            public FSDataOutputStream next(final FileSystem fs, final Path p)
                    throws IOException {
                return fs.create(p, permission, cflags, bufferSize,
                        replication, blockSize, progress, checksumOpt);
            }
        }.resolve(this, absF);
    }
```

2.2.2.3.1 创建 DFSOutputStream 并启动

```
 /**
       * Call {@link #create(String, FsPermission, EnumSet, boolean, short,
       * long, Progressable, int, ChecksumOpt)} with <code>createParent</code>
       * set to true.
       */
      public DFSOutputStream create(String src, FsPermission permission,
                                    EnumSet<CreateFlag> flag, short replication, long
blockSize,
                                    Progressable progress, int buffersize, ChecksumOpt
checksumOpt)
                  throws IOException {
            // 往下追
            return create(src, permission, flag, true,
                        replication, blockSize, progress, buffersize, checksumOpt, null);
      }
```

```
/**
       * Same as {@link #create(String, FsPermission, EnumSet, boolean, short, long,
       * Progressable, int, ChecksumOpt)} with the addition of favoredNodes that is
       * a hint to where the namenode should place the file blocks.
       * The favored nodes hint is not persisted in HDFS. Hence it may be honored
       * at the creation time only. HDFS could move the blocks during balancing or
       * replication, to move the blocks from favored nodes. A value of null means
       * no favored nodes for this create
       */
      public DFSOutputStream create(String src, FsPermission permission,
                                    EnumSet<CreateFlag> flag, boolean createParent, short
replication,
                                    long blockSize, Progressable progress, int buffersize,
                                    ChecksumOpt checksumOpt, InetSocketAddress[]
favoredNodes)
                  throws IOException {
            // 往下追
            return create(src, permission, flag, createParent, replication, blockSize,
                        progress, buffersize, checksumOpt, favoredNodes, null);
      }
```

```
 /**
       * Same as {@link #create(String, FsPermission, EnumSet, boolean, short, long,
       * Progressable, int, ChecksumOpt, InetSocketAddress[])} with the addition of
       * ecPolicyName that is used to specify a specific erasure coding policy
       * instead of inheriting any policy from this new file's parent directory.
       * This policy will be persisted in HDFS. A value of null means inheriting
```

```
     * parent groups' whatever policy.
     */
    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 {
        checkOpen();
        final FsPermission masked = applyUMask(permission);
        LOG.debug("{}: masked={}", src, masked);

        // 创建 DFSOutputStream
        final DFSOutputStream result = DFSOutputStream.newStreamForCreate(this,
                src, masked, flag, createParent, replication, blockSize, progress,
                dfsClientConf.createChecksum(checksumOpt),
                // null
                getFavoredNodesStr(favoredNodes),
                // null
                ecPolicyName);

        // 开始 Client 与 NameNode LeaseManager 进行契约
        beginFileLease(result.getFileId(), result);
        return result;
    }
```

```
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;

            // Retry the create if we get a RetryStartFileException up to a maximum
            // number of times
            boolean shouldRetry = true;
            int retryCount = CREATE_RETRY_COUNT;
            while (shouldRetry) {
```

```
                    shouldRetry = false;
                    try {
                        // 发送 RPC 请求 NameNode 创建文件上传路径 (调用 NameNodeRpcServer.create())
                        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,
                                DSQuotaExceededException.class,
                                QuotaByStorageTypeExceededException.class,
                                FileAlreadyExistsException.class,
                                FileNotFoundException.class,
                                ParentNotDirectoryException.class,
                                NSQuotaExceededException.class,
                                RetryStartFileException.class,
                                SafeModeException.class,
                                UnresolvedPathException.class,
                                SnapshotAccessControlException.class,
                                UnknownCryptoProtocolVersionException.class);
                        if (e instanceof 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) {
                    // 纠删码相关
                    out = new DFSStripedOutputStream(dfsClient, src, stat,
                            flag, progress, checksum, favoredNodes);
                } else {
                    // 创建 DFSOutputStream
```

```
                    out = new DFSOutputStream(dfsClient, src, stat,
                            flag, progress, checksum, favoredNodes, true);
                }
                // 启动 DFSOutputStream
                out.start();
                return out;
            }
        }
```

2.2.2.3.1.1 发送 RPC 请求 NameNode 创建文件上传路径 (调用 NameNodeRpcServer.create())

```
@Override // ClientProtocol
    public HdfsFileStatus create(String src, FsPermission masked,
                                 String clientName, EnumSetWritable<CreateFlag> flag,
                                 boolean createParent, short replication, long blockSize,
                                 CryptoProtocolVersion[] supportedVersions, String
ecPolicyName)
            throws IOException {
        checkNNStartup();

        // 获取 RPC Client 地址
        String clientMachine = getClientMachine();

        if (stateChangeLog.isDebugEnabled()) {
            stateChangeLog.debug("*DIR* NameNode.create: file "
                    + src + " for " + clientName + " at " + clientMachine);
        }
        if (!checkPathLength(src)) {
            throw new IOException("create: Pathname too long.  Limit "
                    + MAX_PATH_LENGTH + " characters, " + MAX_PATH_DEPTH + " levels.");
        }

        // 检查是否可以操作 NameNode 的元数据 (NameNode Standby 情况下报错)
        namesystem.checkOperation(OperationCategory.WRITE);

        // 重试相关
        CacheEntryWithPayload cacheEntry = RetryCache.waitForCompletion(retryCache, null);
        if (cacheEntry != null && cacheEntry.isSuccess()) {
            return (HdfsFileStatus) cacheEntry.getPayload();
        }

        HdfsFileStatus status = null;
```

```
        try {
            // 权限相关
            PermissionStatus perm = new PermissionStatus(getRemoteUser()
                    .getShortUserName(), null, masked);
            // 往下追
            status = namesystem.startFile(src, perm, clientName, clientMachine,
                    flag.get(), createParent, replication, blockSize, supportedVersions,
                    ecPolicyName, cacheEntry != null);
        } finally {
            RetryCache.setState(cacheEntry, status != null, status);
        }

        metrics.incrFilesCreated();
        metrics.incrCreateFileOps();
        return status;
    }
```

```
/**
     * Create a new file entry in the namespace.
     * <p>
     * For description of parameters and exceptions thrown see
     * {@link ClientProtocol#create}, except it returns valid file status upon
     * success
     */
    HdfsFileStatus startFile(String src, PermissionStatus permissions,
                             String holder, String clientMachine, EnumSet<CreateFlag>
flag,
                             boolean createParent, short replication, long blockSize,
                             CryptoProtocolVersion[] supportedVersions, String
ecPolicyName,
                             boolean logRetryCache) throws IOException {

        HdfsFileStatus status;
        try {
            // 往下追
            status = startFileInt(src, permissions, holder, clientMachine, flag,
                    createParent, replication, blockSize, supportedVersions, ecPolicyName,
                    logRetryCache);
        } catch (AccessControlException e) {
            logAuditEvent(false, "create", src);
            throw e;
        }
        logAuditEvent(true, "create", src, status);
        return status;
```

```
    }
```

```
private HdfsFileStatus startFileInt(String src,
                                    PermissionStatus permissions, String holder,
String clientMachine,
                                    EnumSet<CreateFlag> flag, boolean
createParent, short replication,
                                    long blockSize, CryptoProtocolVersion[]
supportedVersions,
                                    String ecPolicyName, boolean logRetryCache)
throws IOException {
        if (NameNode.stateChangeLog.isDebugEnabled()) {
            StringBuilder builder = new StringBuilder();
            builder.append("DIR* NameSystem.startFile: src=").append(src)
                    .append(", holder=").append(holder)
                    .append(", clientMachine=").append(clientMachine)
                    .append(", createParent=").append(createParent)
                    .append(", replication=").append(replication)
                    .append(", createFlag=").append(flag)
                    .append(", blockSize=").append(blockSize)
                    .append(", supportedVersions=")
                    .append(Arrays.toString(supportedVersions));
            NameNode.stateChangeLog.debug(builder.toString());
        }
        if (!DFSUtil.isValidName(src) ||
                FSDirectory.isExactReservedName(src) ||
                (FSDirectory.isReservedName(src)
                        && !FSDirectory.isReservedRawName(src)
                        && !FSDirectory.isReservedInodesName(src))) {
            throw new InvalidPathException(src);
        }

        // flag = [CREATE, OVERWRITE]
        boolean shouldReplicate = flag.contains(CreateFlag.SHOULD_REPLICATE);
        if (shouldReplicate &&
                (!org.apache.commons.lang.StringUtils.isEmpty(ecPolicyName))) {
            throw new HadoopIllegalArgumentException("SHOULD_REPLICATE flag and " +
                    "ecPolicyName are exclusive parameters. Set both is not allowed!");
        }

        INodesInPath iip = null;
        boolean skipSync = true; // until we do something that might create edits
        HdfsFileStatus stat = null;
        BlocksMapUpdateInfo toRemoveBlocks = null;
```

```
        // 再次检查当前 NameNode 是否处于 Active (Standby 情况下报错)
        checkOperation(OperationCategory.WRITE);

        // 用户权限相关
        final FSPermissionChecker pc = getPermissionChecker();
        writeLock();
        try {
            checkOperation(OperationCategory.WRITE);
            checkNameNodeSafeMode("Cannot create file" + src);

            // 往下追 (解析 src)
            iip = FSDirWriteFileOp.resolvePathForStartFile(
                    dir, pc, src, flag, createParent);

            if (blockSize < minBlockSize) {
                throw new IOException("Specified block size is less than configured" +
                        " minimum value (" +
DFSConfigKeys.DFS_NAMENODE_MIN_BLOCK_SIZE_KEY
                        + "): " + blockSize + " < " + minBlockSize);
            }

            if (shouldReplicate) {
                blockManager.verifyReplication(src, replication, clientMachine);
            } else {
                final ErasureCodingPolicy ecPolicy = FSDirErasureCodingOp
                        .getErasureCodingPolicy(this, ecPolicyName, iip);
                if (ecPolicy != null && (!ecPolicy.isReplicationPolicy())) {
                    checkErasureCodingSupported("createWithEC");
                    if (blockSize < ecPolicy.getCellSize()) {
                        throw new IOException("Specified block size (" + blockSize
                                + ") is less than the cell size (" + ecPolicy.getCellSize()
                                + ") of the erasure coding policy (" + ecPolicy + ").");
                    }
                } else {
                    blockManager.verifyReplication(src, replication, clientMachine);
                }
            }

            FileEncryptionInfo feInfo = null;
            if (!iip.isRaw() && provider != null) {
                EncryptionKeyInfo ezInfo = FSDirEncryptionZoneOp.getEncryptionKeyInfo(
                        this, iip, supportedVersions);
                // if the path has an encryption zone, the lock was released while
```

```
                    // generating the EDEK.    re-resolve the path to ensure the namesystem
                    // and/or EZ has not mutated
                    if (ezInfo != null) {
                        checkOperation(OperationCategory.WRITE);
                        iip = FSDirWriteFileOp.resolvePathForStartFile(
                                dir, pc, iip.getPath(), flag, createParent);
                        feInfo = FSDirEncryptionZoneOp.getFileEncryptionInfo(
                                dir, iip, ezInfo);
                    }
                }

                skipSync = false; // following might generate edits
                toRemoveBlocks = new BlocksMapUpdateInfo();
                dir.writeLock();
                try {
                    // 往下追 (添加元数据)
                    stat = FSDirWriteFileOp.startFile(this, iip, permissions, holder,
                            clientMachine, flag, createParent, replication, blockSize, feInfo,
                            toRemoveBlocks, shouldReplicate, ecPolicyName, logRetryCache);
                } catch (IOException e) {
                    skipSync = e instanceof StandbyException;
                    throw e;
                } finally {
                    dir.writeUnlock();
                }
            } finally {
                writeUnlock("create");
                // There might be transactions logged while trying to recover the lease.
                // They need to be sync'ed even when an exception was thrown.
                if (!skipSync) {
                    getEditLog().logSync();
                    if (toRemoveBlocks != null) {
                        removeBlocks(toRemoveBlocks);
                        toRemoveBlocks.clear();
                    }
                }
            }

            return stat;
        }
```

```
/**
     * Create a new file or overwrite an existing file<br>
     * <p>
```

```
     * Once the file is create the client then allocates a new block with the next
     * call using {@link ClientProtocol#addBlock}.
     * <p>
     * For description of parameters and exceptions thrown see
     * {@link ClientProtocol#create}
     */
    static HdfsFileStatus startFile(
            FSNamesystem fsn, INodesInPath iip,
            PermissionStatus permissions, String holder, String clientMachine,
            EnumSet<CreateFlag> flag, boolean createParent,
            short replication, long blockSize,
            FileEncryptionInfo feInfo, INode.BlocksMapUpdateInfo toRemoveBlocks,
            boolean shouldReplicate, String ecPolicyName, boolean logRetryEntry)
            throws IOException {
        assert fsn.hasWriteLock();
        // true
        boolean overwrite = flag.contains(CreateFlag.OVERWRITE);
        // false
        boolean isLazyPersist = flag.contains(CreateFlag.LAZY_PERSIST);

        // src 绝对路径
        final String src = iip.getPath();
        // 获取内存管理元数据 FSDirectory
        FSDirectory fsd = fsn.getFSDirectory();

        if (iip.getLastINode() != null) {
            if (overwrite) {
                List<INode> toRemoveINodes = new ChunkedArrayList<>();
                List<Long> toRemoveUCFiles = new ChunkedArrayList<>();
                long ret = FSDirDeleteOp.delete(fsd, iip, toRemoveBlocks,
                        toRemoveINodes, toRemoveUCFiles, now());
                if (ret >= 0) {
                    iip = INodesInPath.replace(iip, iip.length() - 1, null);
                    FSDirDeleteOp.incrDeletedFileCount(ret);
                    fsn.removeLeasesAndINodes(toRemoveUCFiles, toRemoveINodes, true);
                }
            } else {
                // If lease soft limit time is expired, recover the lease
                fsn.recoverLeaseInternal(FSNamesystem.RecoverLeaseOp.CREATE_FILE, iip,
                        src, holder, clientMachine, false);
                throw new FileAlreadyExistsException(src + " for client " +
                        clientMachine + " already exists");
            }
        }
```

```
        // 限制元数据创建操作
        fsn.checkFsObjectLimit();

        INodeFile newNode = null;
        // 创建 src 的 父节点
        // For example, path="/foo/bar/spam", "/foo" is an existing directory,
        // "/foo/bar" is not existing yet, the function will create directory bar.
        INodesInPath parent =
                FSDirMkdirOp.createAncestorDirectories(fsd, iip, permissions);
        if (parent != null) {
            // 添加元数据 INodeFile
            iip = addFile(fsd, parent, iip.getLastLocalName(), permissions,
                    replication, blockSize, holder, clientMachine, shouldReplicate,
                    ecPolicyName);
            newNode = iip != null ? iip.getLastINode().asFile() : null;
        }
        if (newNode == null) {
            throw new IOException("Unable to add " + src + " to namespace");
        }
        // 添加 Client 契约
        fsn.leaseManager.addLease(
                newNode.getFileUnderConstructionFeature().getClientName(),
                newNode.getId());
        if (feInfo != null) {
            FSDirEncryptionZoneOp.setFileEncryptionInfo(fsd, iip, feInfo,
                    XAttrSetFlag.CREATE);
        }
        setNewINodeStoragePolicy(fsd.getBlockManager(), iip, isLazyPersist);

        // 往 EditLog 添加一天元数据记录 AddOp
        fsd.getEditLog().logOpenFile(src, newNode, overwrite, logRetryEntry);

        if (NameNode.stateChangeLog.isDebugEnabled()) {
            NameNode.stateChangeLog.debug("DIR* NameSystem.startFile: added " +
                    src + " inode " + newNode.getId() + " " + holder);
        }
        // 返回创建 INodeFile 的相关信息
        return FSDirStatAndListingOp.getFileInfo(fsd, iip, false, false);
    }
```

2.2.2.3.1.1.1 添加元数据 INodeFile

```
/**
```

```
   * Add the given filename to the fs.
   *
   * @return the new INodesInPath instance that contains the new INode
   */
  private static INodesInPath addFile(
              FSDirectory fsd, INodesInPath existing, byte[] localName,
              PermissionStatus permissions, short replication, long preferredBlockSize,
              String clientName, String clientMachine, boolean shouldReplicate,
              String ecPolicyName) throws IOException {

        Preconditions.checkNotNull(existing);
        long modTime = now();
        INodesInPath newiip;
        fsd.writeLock();
        try {
              boolean isStriped = false;
              ErasureCodingPolicy ecPolicy = null;
              if (!shouldReplicate) {
                    ecPolicy = FSDirErasureCodingOp.getErasureCodingPolicy(
                                fsd.getFSNamesystem(), ecPolicyName, existing);
                    if (ecPolicy != null && (!ecPolicy.isReplicationPolicy())) {
                          isStriped = true;
                    }
              }
              // CONTIGUOUS
              final BlockType blockType = isStriped ?
                          BlockType.STRIPED : BlockType.CONTIGUOUS;
              final Short replicationFactor = (!isStriped ? replication : null);
              final Byte ecPolicyID = (isStriped ? ecPolicy.getId() : null);
              // 创建 INodeFile
              INodeFile newNode = newINodeFile(fsd.allocateNewInodeId(), permissions,
                          modTime, modTime, replicationFactor, ecPolicyID, preferredBlockSize,
                          blockType);
              newNode.setLocalName(localName);
              newNode.toUnderConstruction(clientName, clientMachine);
              // 添加 INodeFile
              newiip = fsd.addINode(existing, newNode, permissions.getPermission());
        } finally {
              fsd.writeUnlock();
        }
        if (newiip == null) {
              NameNode.stateChangeLog.info("DIR* addFile: failed to add " +
                          existing.getPath() + "/" + DFSUtil.bytes2String(localName));
              return null;
```

```
            }

            if (NameNode.stateChangeLog.isDebugEnabled()) {
                NameNode.stateChangeLog.debug("DIR* addFile: " +
                        DFSUtil.bytes2String(localName) + " is added");
            }
            return newiip;
        }
```

```
/**
     * Add the given child to the namespace.
     * @param existing the INodesInPath containing all the ancestral INodes
     * @param child the new INode to add
     * @param modes create modes
     * @return a new INodesInPath instance containing the new child INode. Null
     * if the adding fails.
     * @throws QuotaExceededException is thrown if it violates quota limit
     */
    INodesInPath addINode(INodesInPath existing, INode child,
                          FsPermission modes)
            throws QuotaExceededException, UnresolvedLinkException {
        cacheName(child);
        writeLock();
        try {
            // 往下追
            return addLastINode(existing, child, modes, true);
        } finally {
            writeUnlock();
        }
    }
```

```
/**
     * Add a child to the end of the path specified by INodesInPath.
     * @param existing the INodesInPath containing all the ancestral INodes
     * @param inode the new INode to add
     * @param modes create modes
     * @param checkQuota whether to check quota
     * @return an INodesInPath instance containing the new INode
     */
    @VisibleForTesting
    public INodesInPath addLastINode(INodesInPath existing, INode inode,
                                     FsPermission modes, boolean checkQuota) throws
QuotaExceededException {
        assert existing.getLastINode() != null &&
```

```
                existing.getLastINode().isDirectory();

        final int pos = existing.length();
        // Disallow creation of /.reserved. This may be created when loading
        // editlog/fsimage during upgrade since /.reserved was a valid name in older
        // release. This may also be called when a user tries to create a file
        // or directory /.reserved.
        if (pos == 1 && existing.getINode(0) == rootDir && isReservedName(inode)) {
            throw new HadoopIllegalArgumentException(
                    "File name \"" + inode.getLocalName() + "\" is reserved and cannot "
                            + "be created. If this is during upgrade change the name of the
"
                            + "existing file or directory to another name before upgrading
"
                            + "to the new release.");
        }
        // 父目录
        final INodeDirectory parent = existing.getINode(pos - 1).asDirectory();
        // The filesystem limits are not really quotas, so this check may appear
        // odd. It's because a rename operation deletes the src, tries to add
        // to the dest, if that fails, re-adds the src from whence it came.
        // The rename code disables the quota when it's restoring to the
        // original location because a quota violation would cause the the item
        // to go "poof".    The fs limits must be bypassed for the same reason.
        if (checkQuota) {
            final String parentPath = existing.getPath();
            verifyMaxComponentLength(inode.getLocalNameBytes(), parentPath);
            verifyMaxDirItems(parent, parentPath);
        }
        // always verify inode name
        verifyINodeName(inode.getLocalNameBytes());

        final QuotaCounts counts = inode
                .computeQuotaUsage(getBlockStoragePolicySuite(),
                        parent.getStoragePolicyID(), false, Snapshot.CURRENT_STATE_ID);
        updateCount(existing, pos, counts, checkQuota);

        boolean isRename = (inode.getParent() != null);
        // 往父目录添加子目录
        final boolean added = parent.addChild(inode, true,
                existing.getLatestSnapshotId());
        if (!added) {
            updateCountNoQuotaCheck(existing, pos, counts.negation());
            return null;
```

```
            } else {
                if (!isRename) {
                    copyINodeDefaultAcl(inode, modes);
                }
                // 缓存 Inode
                addToInodeMap(inode);
            }
            // 返回 Inode 信息
            return INodesInPath.append(existing, inode, inode.getLocalNameBytes());
        }
```

2.2.2.3.1.1.2 添加 Client 契约 (调用 LeaseManager.addLease())

```
/**
     * Adds (or re-adds) the lease for the specified file.
     */
    synchronized Lease addLease(
                // 客户端名字
                String holder,
                long inodeId) {
            Lease lease = getLease(holder);
            if (lease == null) {
                // 创建并添加 Lease
                lease = new Lease(holder);
                leases.put(holder, lease);
                // 排序 tree
                sortedLeases.add(lease);
            } else {
                renewLease(lease);
            }
            leasesById.put(inodeId, lease);
            lease.files.add(inodeId);
            return lease;
        }
```

2.2.2.3.1.1.3 往 EditLog 添加一天元数据记录 AddOp (调用 FSEditLog.logOpenFile)

```
/**
     * Add open lease record to edit log.
     * Records the block locations of the last block.
     */
```

```
    public void logOpenFile(String path, INodeFile newNode, boolean overwrite,
                                    boolean toLogRpcIds) {
        Preconditions.checkArgument(newNode.isUnderConstruction());
        PermissionStatus permissions = newNode.getPermissionStatus();
        AddOp op = AddOp.getInstance(cache.get())
                .setInodeId(newNode.getId())
                .setPath(path)
                .setReplication(newNode.getFileReplication())
                .setModificationTime(newNode.getModificationTime())
                .setAccessTime(newNode.getAccessTime())
                .setBlockSize(newNode.getPreferredBlockSize())
                .setBlocks(newNode.getBlocks())
                .setPermissionStatus(permissions)
                .setClientName(newNode.getFileUnderConstructionFeature().getClientName(
))
                .setClientMachine(
                        newNode.getFileUnderConstructionFeature().getClientMachine())
                .setOverwrite(overwrite)
                .setStoragePolicyId(newNode.getLocalStoragePolicyID())
                .setErasureCodingPolicyId(newNode.getErasureCodingPolicyID());

        AclFeature f = newNode.getAclFeature();
        if (f != null) {
            op.setAclEntries(AclStorage.readINodeLogicalAcl(newNode));
        }

        XAttrFeature x = newNode.getXAttrFeature();
        if (x != null) {
            op.setXAttrs(x.getXAttrs());
        }

        logRpcIds(op, toLogRpcIds);
        // 往下追 (参考 HDFS 元数据管理源码分析)
        logEdit(op);
    }
```

2.2.2.3.1.2 创建 DFSOutputStream

```
/**
     * Construct a new output stream for creating a file.
     */
    protected DFSOutputStream(DFSClient dfsClient, String src,
                                    HdfsFileStatus stat, EnumSet<CreateFlag> flag, Progressable
progress,
```

```
                                        DataChecksum checksum, String[] favoredNodes, boolean
createStreamer) {
            // 往下追
            this(dfsClient, src, flag, progress, stat, checksum);
            // false
            this.shouldSyncBlock = flag.contains(CreateFlag.SYNC_BLOCK);

            // 计算 Packet chunk 大小
            computePacketChunkSize(dfsClient.getConf().getWritePacketSize(),
                        bytesPerChecksum);

            if (createStreamer) {
                  // 创建 DataStreamer
                  streamer = new DataStreamer(stat, null, dfsClient, src, progress,
                              checksum, cachingStrategy, byteArrayManager, favoredNodes,
                              addBlockFlags);
            }
      }
```

2.2.2.3.1.2.1 创建 DataStreamer

```
    /**
      * construction with tracing info
      */
     DataStreamer(HdfsFileStatus stat, ExtendedBlock block, DFSClient dfsClient,
                        String src, Progressable progress, DataChecksum checksum,
                        AtomicReference<CachingStrategy> cachingStrategy,
                        ByteArrayManager byteArrayManage, String[] favoredNodes,
                        EnumSet<AddBlockFlag> flags) {
            // 往下追
            this(stat, block, dfsClient, src, progress, checksum, cachingStrategy,
                        byteArrayManage, false, favoredNodes, flags);
            // Block 构建状态
            stage = BlockConstructionStage.PIPELINE_SETUP_CREATE;
      }
```

```
private DataStreamer(HdfsFileStatus stat, ExtendedBlock block,
                              DFSClient dfsClient, String src,
                              Progressable progress, DataChecksum checksum,
                              AtomicReference<CachingStrategy> cachingStrategy,
                              ByteArrayManager byteArrayManage,
                              boolean isAppend, String[] favoredNodes,
                              EnumSet<AddBlockFlag> flags) {
            // 创建 BlockToWrite
```

```
            this.block = new BlockToWrite(block);
            this.dfsClient = dfsClient;
            this.src = src;
            this.progress = progress;
            this.stat = stat;
            this.checksum4WriteBlock = checksum;
            this.cachingStrategy = cachingStrategy;
            this.byteArrayManager = byteArrayManage;
            this.isLazyPersistFile = isLazyPersist(stat);
            this.isAppend = isAppend;
            this.favoredNodes = favoredNodes;
            final DfsClientConf conf = dfsClient.getConf();
            this.dfsclientSlowLogThresholdMs = conf.getSlowIoWarningThresholdMs();
            this.excludedNodes = initExcludedNodes(conf.getExcludedNodesCacheExpiry());
            this.errorState = new ErrorState(conf.getDatanodeRestartTimeout());
            this.addBlockFlags = flags;
        }
```

2.2.2.3.1.2.2 启动 DFSOutputStream

```
        protected synchronized void start() {
            // 启动 DataStreamer (调用 DataStreamer.run())
            getStreamer().start();
        }
```

2.2.2.3.1.2.2.1 启动 DataStreamer 线程 (调用 DataStreamer.run() 阻塞等待 Package)

```
/*
         * streamer thread is the only thread that opens streams to datanode,
         * and closes them. Any error recovery is also done by this thread.
         */
        @Override
        public void run() {
            long lastPacket = Time.monotonicNow();
            TraceScope scope = null;
            while (!streamerClosed && dfsClient.clientRunning) {
                // if the Responder encountered an error, shutdown Responder
                if (errorState.hasError()) {
                    closeResponder();
                }

                DFSPacket one;
                try {
```

```
// process datanode IO errors if any
boolean doSleep = processDatanodeOrExternalError();

final int halfSocketTimeout = dfsClient.getConf().getSocketTimeout() / 2;
synchronized (dataQueue) {
    // wait for a packet to be sent.
    long now = Time.monotonicNow();
    while ((!shouldStop() && dataQueue.size() == 0 &&
            (stage != BlockConstructionStage.DATA_STREAMING ||
                    now - lastPacket < halfSocketTimeout)) || doSleep) {
        long timeout = halfSocketTimeout - (now - lastPacket);
        timeout = timeout <= 0 ? 1000 : timeout;
        timeout = (stage == BlockConstructionStage.DATA_STREAMING) ?
                timeout : 1000;
        try {
            dataQueue.wait(timeout);
        } catch (InterruptedException e) {
            LOG.warn("Caught exception", e);
        }
        doSleep = false;
        now = Time.monotonicNow();
    }
    if (shouldStop()) {
        continue;
    }
    // get packet to be sent.
    if (dataQueue.isEmpty()) {
        one = createHeartbeatPacket();
    } else {
        try {
            backOffIfNecessary();
        } catch (InterruptedException e) {
            LOG.warn("Caught exception", e);
        }
        //  拉取一个  Package
        one = dataQueue.getFirst(); // regular data packet

        SpanId[] parents = one.getTraceParents();
        if (parents.length > 0) {
            scope = dfsClient.getTracer().
                    newScope("dataStreamer", parents[0]);
            scope.getSpan().setParents(parents);
        }
    }
```

```
}

// get new block from namenode.
if (LOG.isDebugEnabled()) {
    LOG.debug("stage=" + stage + ", " + this);
}
// 一开始 stage = PIPELINE_SETUP_CREATE
if (stage == BlockConstructionStage.PIPELINE_SETUP_CREATE) {
    LOG.debug("Allocating new block: {}", this);
    setPipeline(nextBlockOutputStream());
    initDataStreaming();
} else if (stage == BlockConstructionStage.PIPELINE_SETUP_APPEND) {
    LOG.debug("Append to block {}", block);
    setupPipelineForAppendOrRecovery();
    if (streamerClosed) {
        continue;
    }
    initDataStreaming();
}

long lastByteOffsetInBlock = one.getLastByteOffsetBlock();
if (lastByteOffsetInBlock > stat.getBlockSize()) {
    throw new IOException("BlockSize " + stat.getBlockSize() +
            " < lastByteOffsetInBlock, " + this + ", " + one);
}

if (one.isLastPacketInBlock()) {
    // wait for all data packets have been successfully acked
    synchronized (dataQueue) {
        while (!shouldStop() && ackQueue.size() != 0) {
            try {
                // wait for acks to arrive from datanodes
                dataQueue.wait(1000);
            } catch (InterruptedException e) {
                LOG.warn("Caught exception", e);
            }
        }
    }
    if (shouldStop()) {
        continue;
    }
    stage = BlockConstructionStage.PIPELINE_CLOSE;
}
```

```
// send the packet
SpanId spanId = SpanId.INVALID;
synchronized (dataQueue) {
    // move packet from dataQueue to ackQueue
    if (!one.isHeartbeatPacket()) {
        if (scope != null) {
            spanId = scope.getSpanId();
            scope.detach();
            one.setTraceScope(scope);
        }
        scope = null;
        dataQueue.removeFirst();
        ackQueue.addLast(one);
        packetSendTime.put(one.getSeqno(), Time.monotonicNow());
        dataQueue.notifyAll();
    }
}

LOG.debug("{} sending {}", this, one);

// write out data to remote datanode
try (TraceScope ignored = dfsClient.getTracer().
        newScope("DataStreamer#writeTo", spanId)) {
    one.writeTo(blockStream);
    blockStream.flush();
} catch (IOException e) {
    // HDFS-3398 treat primary DN is down since client is unable to
    // write to primary DN. If a failed or restarting node has already
    // been recorded by the responder, the following call will have no
    // effect. Pipeline recovery can handle only one node error at a
    // time. If the primary node fails again during the recovery, it
    // will be taken out then.
    errorState.markFirstNodeIfNotMarked();
    throw e;
}
lastPacket = Time.monotonicNow();

// update bytesSent
long tmpBytesSent = one.getLastByteOffsetBlock();
if (bytesSent < tmpBytesSent) {
    bytesSent = tmpBytesSent;
}

if (shouldStop()) {
```

```
                        continue;
                    }

                    // Is this block full?
                    if (one.isLastPacketInBlock()) {
                        // wait for the close packet has been acked
                        synchronized (dataQueue) {
                            while (!shouldStop() && ackQueue.size() != 0) {
                                dataQueue.wait(1000);// wait for acks to arrive from
datanodes
                            }
                        }
                        if (shouldStop()) {
                            continue;
                        }

                        endBlock();
                    }
                    if (progress != null) {
                        progress.progress();
                    }

                    // This is used by unit test to trigger race conditions.
                    if (artificialSlowdown != 0 && dfsClient.clientRunning) {
                        Thread.sleep(artificialSlowdown);
                    }
                } catch (Throwable e) {
                    // Log warning if there was a real error.
                    if (!errorState.isRestartingNode()) {
                        // Since their messages are descriptive enough, do not always
                        // log a verbose stack-trace WARN for quota exceptions.
                        if (e instanceof QuotaExceededException) {
                            LOG.debug("DataStreamer Quota Exception", e);
                        } else {
                            LOG.warn("DataStreamer Exception", e);
                        }
                    }
                    lastException.set(e);
                    assert !(e instanceof NullPointerException);
                    errorState.setInternalError();
                    if (!errorState.isNodeMarked()) {
                        // Not a datanode issue
                        streamerClosed = true;
                    }
```

```
                } finally {
                    if (scope != null) {
                        scope.close();
                        scope = null;
                    }
                }
            }
            closeInternal();
        }
```

2.2.2.3.2 开始 Client 与 NameNode LeaseManager 进行契约

```
/**
     * Get a lease and start automatic renewal
     */
    private void beginFileLease(final long inodeId, final DFSOutputStream out)
            throws IOException {
        synchronized (filesBeingWritten) {
            // 更新上一次契约时间
            putFileBeingWritten(inodeId, out);
            // 创建 LeaseRenewer
            getLeaseRenewer()
                    // 往下追 (启动 LeaseRenewer 内部线程与 NameNode 的 LeaseManager 进行定期契约)
                    .put(this);
        }
    }
```

2.2.2.3.2.1 创建 LeaseRenewer

```
 /**
     * Return the lease renewer instance. The renewer thread won't start
     * until the first output stream is created. The same instance will
     * be returned until all output streams are closed.
     */
    public LeaseRenewer getLeaseRenewer() {
        // 获取 LeaseRenewer
        return LeaseRenewer.getInstance(
                namenodeUri != null ? namenodeUri.getAuthority() : "null", ugi, this);
    }
```

```
/** Get a {@link LeaseRenewer} instance */
    public static LeaseRenewer getInstance(final String authority,
                                           final UserGroupInformation ugi, final
DFSClient dfsc) {
        // 创建 LeaseRenewer
        final LeaseRenewer r = Factory.INSTANCE.get(authority, ugi);
        r.addClient(dfsc);
        return r;
    }
```

```
private LeaseRenewer(Factory.Key factorykey) {
        this.factorykey = factorykey;
        // 计算续约时间 (默认 30s 执行一次续约)
        unsyncSetGraceSleepPeriod(leaseRenewerGraceDefault);

        if (LOG.isTraceEnabled()) {
            instantiationTrace = StringUtils.stringifyException(
                    new Throwable("TRACE"));
        } else {
            instantiationTrace = null;
        }
    }
```

2.2.2.3.2.2 启动 LeaseRenewer 内部线程与 NameNode 的 LeaseManager 进行定期契约 (调用 LeaseRenewer.run())

```
public synchronized void put(final DFSClient dfsc) {
        if (dfsc.isClientRunning()) {
            if (!isRunning() || isRenewerExpired()) {
                //start a new deamon with a new id.
                final int id = ++currentId;
                // 创建线程
                daemon = new Daemon(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            if (LOG.isDebugEnabled()) {
                                LOG.debug("Lease renewer daemon for " + clientsString()
                                        + " with renew id " + id + " started");
                            }
                            // 往下追
                            LeaseRenewer.this.run(id);
```

```
                                } catch (InterruptedException e) {
                                        LOG.debug("LeaseRenewer is interrupted.", e);
                                } finally {
                                        synchronized (LeaseRenewer.this) {
                                                Factory.INSTANCE.remove(LeaseRenewer.this);
                                        }
                                        if (LOG.isDebugEnabled()) {
                                                LOG.debug("Lease renewer daemon for " + clientsString()
                                                                + " with renew id " + id + " exited");
                                        }
                                }
                        }

                        @Override
                        public String toString() {
                                return String.valueOf(LeaseRenewer.this);
                        }
                });
                // 启动线程
                daemon.start();
        }
        emptyTime = Long.MAX_VALUE;
    }
}
```

```
/**
     * Periodically check in with the namenode and renew all the leases
     * when the lease period is half over.
     */
    private void run(final int id) throws InterruptedException {
        for (long lastRenewed = Time.monotonicNow(); !Thread.interrupted();
                Thread.sleep(getSleepPeriod())) {
            final long elapsed = Time.monotonicNow() - lastRenewed;
            if (elapsed >=
                    // 30s
                    getRenewalTime()
            ) {
                try {
                    // 执行契约
                    renew();
                    if (LOG.isDebugEnabled()) {
                        LOG.debug("Lease renewer daemon for " + clientsString()
                                + " with renew id " + id + " executed");
                    }
```

```
                    lastRenewed = Time.monotonicNow();
                } catch (SocketTimeoutException ie) {
                    LOG.warn("Failed to renew lease for " + clientsString() + " for "
                            + (elapsed / 1000) + " seconds.  Aborting ...", ie);
                    List<DFSClient> dfsclientsCopy;
                    synchronized (this) {
                        DFSClientFaultInjector.get().delayWhenRenewLeaseTimeout();
                        dfsclientsCopy = new ArrayList<>(dfsclients);
                        dfsclients.clear();
                        //Expire the current LeaseRenewer thread.
                        emptyTime = 0;
                        Factory.INSTANCE.remove(LeaseRenewer.this);
                    }
                    for (DFSClient dfsClient : dfsclientsCopy) {
                        dfsClient.closeAllFilesBeingWritten(true);
                    }
                    break;
                } catch (IOException ie) {
                    LOG.warn("Failed to renew lease for " + clientsString() + " for "
                            + (elapsed / 1000) + " seconds.  Will retry shortly ...", ie);
                }
            }

            synchronized (this) {
                if (id != currentId || isRenewerExpired()) {
                    if (LOG.isDebugEnabled()) {
                        if (id != currentId) {
                            LOG.debug("Lease renewer daemon for " + clientsString()
                                    + " with renew id " + id + " is not current");
                        } else {
                            LOG.debug("Lease renewer daemon for " + clientsString()
                                    + " with renew id " + id + " expired");
                        }
                    }
                    //no longer the current daemon or expired
                    return;
                }

                // if no clients are in running state or there is no more clients
                // registered with this renewer, stop the daemon after the grace
                // period.
                if (!clientsRunning() && emptyTime == Long.MAX_VALUE) {
                    emptyTime = Time.monotonicNow();
                }
```

```
                }
            }
        }
```

```
private void renew() throws IOException {
            final List<DFSClient> copies;
            synchronized (this) {
                copies = new ArrayList<>(dfsclients);
            }
            //sort the client names for finding out repeated names.
            Collections.sort(copies, new Comparator<DFSClient>() {
                @Override
                public int compare(final DFSClient left, final DFSClient right) {
                    return left.getClientName().compareTo(right.getClientName());
                }
            });
            String previousName = "";
            for (final DFSClient c : copies) {
                //skip if current client name is the same as the previous name.
                if (!c.getClientName().equals(previousName)) {
                    // 续约
                    if (!c.renewLease()) {
                        LOG.debug("Did not renew lease for client {}", c);
                        continue;
                    }
                    previousName = c.getClientName();
                    LOG.debug("Lease renewed for client {}", previousName);
                }
            }
        }
```

```
 /**
       * Renew leases.
       *
       * @return true if lease was renewed. May return false if this
       * client has been closed or has no files open.
       **/
      public boolean renewLease() throws IOException {
          if (clientRunning && !isFilesBeingWrittenEmpty()) {
              try {
                  // 发送 RPC 进行续约
                  namenode.renewLease(clientName);
                  // 更新上一次续约时间
```

```
                    updateLastLeaseRenewal();
                    return true;
                } catch (IOException e) {
                    // Abort if the lease has already expired.
                    final long elapsed = Time.monotonicNow() - getLastLeaseRenewal();
                    if (elapsed > HdfsConstants.LEASE_HARDLIMIT_PERIOD) {
                        LOG.warn("Failed to renew lease for " + clientName + " for "
                                + (elapsed / 1000) + " seconds (>= hard-limit ="
                                + (HdfsConstants.LEASE_HARDLIMIT_PERIOD / 1000) + "
seconds.) "
                                + "Closing all files being written ...", e);
                        closeAllFilesBeingWritten(true);
                    } else {
                        // Let the lease renewer handle it and retry.
                        throw e;
                    }
                }
            }
            return false;
        }
```

2.2.2.3.2.2.1 发送 RPC 进行续约 (调用 NameNodeRpcServer.renewLease)

```
    @Override // ClientProtocol
    public void renewLease(String clientName) throws IOException {
        checkNNStartup();
        // Client 发送 Rpc 进行续约
        namesystem.renewLease(clientName);
    }
```

```
/**
     * Renew the lease(s) held by the given client
     */
    void renewLease(String holder) throws IOException {
        checkOperation(OperationCategory.WRITE);
        readLock();
        try {
            checkOperation(OperationCategory.WRITE);
            checkNameNodeSafeMode("Cannot renew lease for " + holder);
            // 续约
            leaseManager.renewLease(holder);
        } finally {
            readUnlock("renewLease");
        }
```

```
    }
```

```
/**
     * Renew the lease(s) held by the given client
     */
    synchronized void renewLease(String holder) {
        // 根据 holder = Client 名称获取对应的 Lease
        renewLease(getLease(holder));
    }
```

```
    synchronized void renewLease(Lease lease) {
        if (lease != null) {
            // 先移除上一次续约 Lease
            sortedLeases.remove(lease);
            // 更新 Lease 对应的时间
            lease.renew();
            // 在添加续约
            sortedLeases.add(lease);
        }
    }
```

2.2.2.3.2.2.2 NameNode LeaseManager 续约监控线程(调用 LeaseManager 内部类 Monitor.run())

```
/**********************************************************
     * Monitor checks for leases that have expired,
     * and disposes of them.
     **********************************************************/
    class Monitor implements Runnable {
        final String name = getClass().getSimpleName();

        /** Check leases periodically. */
        @Override
        public void run() {
            for (; shouldRunMonitor && fsnamesystem.isRunning(); ) {
                boolean needSync = false;
                try {
                    fsnamesystem.writeLockInterruptibly();
                    try {
                        if (!fsnamesystem.isInSafeMode()) {
                            // 检查 Lease 是否过期
                            needSync = checkLeases();
```

```
                    }
                } finally {
                    fsnamesystem.writeUnlock("leaseManager");
                    // lease reassignments should to be sync'ed.
                    if (needSync) {
                        fsnamesystem.getEditLog().logSync();
                    }
                }

                // 每隔 2s 检查
                Thread.sleep(fsnamesystem.getLeaseRecheckIntervalMs());
            } catch (InterruptedException ie) {
                LOG.debug("{} is interrupted", name, ie);
            } catch (Throwable e) {
                LOG.warn("Unexpected throwable: ", e);
            }
        }
    }
}
```

```
/** Check the leases beginning from the oldest.
     *    @return true is sync is needed.
     */
    @VisibleForTesting
    synchronized boolean checkLeases() {
        boolean needSync = false;
        assert fsnamesystem.hasWriteLock();

        long start = monotonicNow();

        while (!sortedLeases.isEmpty() &&
                // 判断第一个 Lease 是否过期 (超过 1h)
                sortedLeases.first().expiredHardLimit()
                && !isMaxLockHoldToReleaseLease(start)) {

            // 获取第一个过期 Lease
            Lease leaseToCheck = sortedLeases.first();
            LOG.info("{} has expired hard limit", leaseToCheck);

            final List<Long> removing = new ArrayList<>();
            // need to create a copy of the oldest lease files, because
            // internalReleaseLease() removes files corresponding to empty files,
            // i.e. it needs to modify the collection being iterated over
```

```
                // causing ConcurrentModificationException
                Collection<Long> files = leaseToCheck.getFiles();
                Long[] leaseINodeIds = files.toArray(new Long[files.size()]);
                FSDirectory fsd = fsnamesystem.getFSDirectory();
                String p = null;
                String newHolder = getInternalLeaseHolder();
                for (Long id : leaseINodeIds) {
                    try {
                        INodesInPath iip = INodesInPath.fromINode(fsd.getInode(id));
                        p = iip.getPath();
                        // Sanity check to make sure the path is correct
                        if (!p.startsWith("/")) {
                            throw new IOException("Invalid path in the lease " + p);
                        }
                        final INodeFile lastINode = iip.getLastINode().asFile();
                        if (fsnamesystem.isFileDeleted(lastINode)) {
                            // INode referred by the lease could have been deleted.
                            removeLease(lastINode.getId());
                            continue;
                        }
                        boolean completed = false;
                        try {
                            completed = fsnamesystem.internalReleaseLease(
                                    leaseToCheck, p, iip, newHolder);
                        } catch (IOException e) {
                            LOG.warn("Cannot release the path {} in the lease {}. It will be "
                                    + "retried.", p, leaseToCheck, e);
                            continue;
                        }
                        if (LOG.isDebugEnabled()) {
                            if (completed) {
                                LOG.debug("Lease recovery for inode {} is complete. File
closed"
                                        + ".", id);
                            } else {
                                LOG.debug("Started block recovery {} lease {}", p,
leaseToCheck);
                            }
                        }
                        // If a lease recovery happened, we need to sync later.
                        if (!needSync && !completed) {
                            needSync = true;
                        }
                    } catch (IOException e) {
```

```
                        LOG.warn("Removing lease with an invalid path: {},{}", p,
                                leaseToCheck, e);
                        removing.add(id);
                    }
                    if (isMaxLockHoldToReleaseLease(start)) {
                        LOG.debug("Breaking out of checkLeases after {} ms.",
                                fsnamesystem.getMaxLockHoldToReleaseLeaseMs());
                        break;
                    }
                }

                for (Long id : removing) {
                    removeLease(leaseToCheck, id);
                }
            }

            return needSync;
        }
```

2.2.2.3.2.3 包装 DFSOutputStream 成 HdfsDataOutputStream

```
/**
     * Wraps the stream in a CryptoOutputStream if the underlying file is
     * encrypted.
     */
    public HdfsDataOutputStream createWrappedOutputStream(DFSOutputStream dfsos,
                                                           FileSystem.Statistics
statistics) throws IOException {
        // 往下追
        return createWrappedOutputStream(dfsos, statistics, 0);
    }
```

```
 /**
     * Wraps the stream in a CryptoOutputStream if the underlying file is
     * encrypted.
     */
    public HdfsDataOutputStream createWrappedOutputStream(DFSOutputStream dfsos,
                                                           FileSystem.Statistics
statistics, long startPos) throws IOException {
        final FileEncryptionInfo feInfo = dfsos.getFileEncryptionInfo();
        if (feInfo != null) {
            // File is encrypted, wrap the stream in a crypto stream.
            // Currently only one version, so no special logic based on the version #
```

```
                HdfsKMSUtil.getCryptoProtocolVersion(feInfo);
                final CryptoCodec codec = HdfsKMSUtil.getCryptoCodec(conf, feInfo);
                KeyVersion decrypted;
                try (TraceScope ignored = tracer.newScope("decryptEDEK")) {
                    decrypted = HdfsKMSUtil.decryptEncryptedDataEncryptionKey(feInfo,
                            getKeyProvider());
                }
                final CryptoOutputStream cryptoOut =
                        new CryptoOutputStream(dfsos, codec,
                                decrypted.getMaterial(), feInfo.getIV(), startPos);
                return new HdfsDataOutputStream(cryptoOut, statistics, startPos);
            } else {
                // No FileEncryptionInfo present so no encryption.
                // 创建 HdfsDataOutputStream
                return new HdfsDataOutputStream(dfsos, statistics, startPos);
            }
        }
```

2.2.2.3.2.3.1 创建 HdfsDataOutputStream

```
    public HdfsDataOutputStream(DFSOutputStream out, FileSystem.Statistics stats,
        long startPosition) throws IOException {
      // 往下追
      super(out, stats, startPosition);
    }
```

```
    public FSDataOutputStream(OutputStream out, FileSystem.Statistics stats,
                              long startPosition) {
        // 往下追
        super(
                // 创建 PositionCache
                new PositionCache(out, stats, startPosition)
        );
        // out = DFSOutputStream
        wrappedStream = out;
    }
```

2.2.2.3.2.3.1.1 创建 PositionCache

```
PositionCache(OutputStream out, FileSystem.Statistics stats, long pos) {
                // 往下追 (out = DFSOutputStream)
                super(out);
```

```
                statistics = stats;
                position = pos;
            }
```

2.2.2.4 执行流拷贝（调用 IOUtils.copyBytes)

```
/**
     * Copies from one stream to another.
     *
     * @param in InputStream to read from
     * @param out OutputStream to write to
     * @param conf the Configuration object
     * @param close whether or not close the InputStream and
     * OutputStream at the end. The streams are closed in the finally clause.
     */
    public static void copyBytes(InputStream in, OutputStream out, Configuration conf, boolean close)
            throws IOException {
        // 往下追
        copyBytes(
                // FSDataInputStream
                in,
                // HdfsDataOutputStream
                out,
                // 4096
                conf.getInt(IO_FILE_BUFFER_SIZE_KEY, IO_FILE_BUFFER_SIZE_DEFAULT),
                // true
                close);
    }
```

```
/**
     * Copies from one stream to another.
     *
     * @param in            InputStrem to read from
     * @param out           OutputStream to write to
     * @param buffSize the size of the buffer
     * @param close       whether or not close the InputStream and
     *                              OutputStream at the end. The streams are closed in the finally clause.
     */
    public static void copyBytes(InputStream in, OutputStream out,
                                 int buffSize, boolean close)
```

```
                throws IOException {
            try {
                // 往下追
                copyBytes(in, out, buffSize);
                if (close) {
                    out.close();
                    out = null;
                    in.close();
                    in = null;
                }
            } finally {
                if (close) {
                    closeStream(out);
                    closeStream(in);
                }
            }
        }
```

```
/**
     * Copies from one stream to another.
     *
     * @param in          InputStrem to read from
     * @param out         OutputStream to write to
     * @param buffSize the size of the buffer
     */
    public static void copyBytes(InputStream in, OutputStream out, int buffSize)
                throws IOException {
        PrintStream ps = out instanceof PrintStream ? (PrintStream) out : null;
        // 标准的 Java IO 编程
        // 4096
        byte[] buf = new byte[buffSize];
        int bytesRead = in.read(buf);
        while (bytesRead >= 0) {
            // 最终调用 FSOutputSummer.write()
            // out = HdfsDataOutputStream extends FSDataOutputStream extends DataOutputStream extends FilterOutputStream implements DataOutput
            // 先调用 DataOutput.write() 接着调用 FilterOutputStream.write() 然后调用 DataOutputStream.write()
            // 在然后调用 FSDataOutputStream 入参 PositionCache.write() 本质调用 DFSOutputStream
            // 但是 DFSOutputStream 没有 write 调用其父类 FSOutputSummer.write()
            out.write(buf, 0, bytesRead);
            if ((ps != null) && ps.checkError()) {
                throw new IOException("Unable to write to output stream.");
```

```
                }
                bytesRead = in.read(buf);
            }
        }
```

```
/**
     * Writes <code>len</code> bytes from the specified byte array
     * starting at offset <code>off</code> and generate a checksum for
     * each data chunk.
     *
     * <p> This method stores bytes from the given array into this
     * stream's buffer before it gets checksumed. The buffer gets checksumed
     * and flushed to the underlying output stream when all data
     * in a checksum chunk are in the buffer.  If the buffer is empty and
     * requested length is at least as large as the size of next checksum chunk
     * size, this method will checksum and write the chunk directly
     * to the underlying output stream.  Thus it avoids unnecessary data copy.
     *
     * @param b      the data.
     * @param off the start offset in the data.
     * @param len the number of bytes to write.
     * @throws IOException if an I/O error occurs.
     */
    @Override
    public synchronized void write(byte b[], int off, int len)
                throws IOException {

        checkClosed();

        if (off < 0 || len < 0 || off > b.length - len) {
            throw new ArrayIndexOutOfBoundsException();
        }

        // write1()
        for (int n = 0; n < len; n += write1(b, off + n, len - n)) {
        }
    }
```

```
/**
     * Write a portion of an array, flushing to the underlying
     * stream at most once if necessary.
     */
    private int write1(byte b[], int off, int len) throws IOException {
```

```
            // 写校验信息
            if (count == 0 && len >= buf.length) {
                // local buffer is empty and user buffer size >= local buffer size, so
                // simply checksum the user buffer and send it directly to the underlying
                // stream
                final int length = buf.length;
                writeChecksumChunks(b, off, length);
                return length;
            }

            // copy user data to local buffer
            // 读取数据到 local buffer 中
            int bytesToCopy = buf.length - count;
            bytesToCopy = Math.min(len, bytesToCopy);
            System.arraycopy(b, off, buf, count, bytesToCopy);
            count += bytesToCopy;
            if (count == buf.length) {
                // local buffer is full
                // 执行 buffer 的 flush
                flushBuffer();
            }
            return bytesToCopy;
        }
```

```
/* Forces any buffered output bytes to be checksumed and written out to
         * the underlying output stream.
         */
        protected synchronized void flushBuffer() throws IOException {
            // 往下追
            flushBuffer(false, true);
        }
```

```
/* Forces buffered output bytes to be checksummed and written out to
         * the underlying output stream. If there is a trailing partial chunk in the
         * buffer,
         * 1) flushPartial tells us whether to flush that chunk
         * 2) if flushPartial is true, keep tells us whether to keep that chunk in the
         * buffer (if flushPartial is false, it is always kept in the buffer)
         *
         * Returns the number of bytes that were flushed but are still left in the
         * buffer (can only be non-zero if keep is true).
         */
        protected synchronized int flushBuffer(boolean keep,
                                               boolean flushPartial) throws IOException {
```

```
            int bufLen = count;
            int partialLen = bufLen % sum.getBytesPerChecksum();
            int lenToFlush = flushPartial ? bufLen : bufLen - partialLen;
            if (lenToFlush != 0) {
                // 构造 packet 加入到 DataStreamer 的 dataQueue 队列中
                writeChecksumChunks(buf, 0, lenToFlush);
                if (!flushPartial || keep) {
                    count = partialLen;
                    System.arraycopy(buf, bufLen - count, buf, 0, count);
                } else {
                    count = 0;
                }
            }

            // total bytes left minus unflushed bytes left
            return count - (bufLen - lenToFlush);
        }
```

```
/**
     * Generate checksums for the given data chunks and output chunks & checksums
     * to the underlying output stream.
     */
    private void writeChecksumChunks(byte b[], int off, int len)
            throws IOException {
        // 重点 1： 计算出来校验和信息
        sum.calculateChunkedSums(b, off, len, checksum, 0);
        TraceScope scope = createWriteTraceScope();
        try {
            // 重点 2： 不停的写 chunk 构造一个完成的 Packet 加入到 DataStreamer 中的 dataQueue 队列中
            for (int i = 0; i < len; i += sum.getBytesPerChecksum()) {
                int chunkLen = Math.min(sum.getBytesPerChecksum(), len - i);
                int ckOffset = i / sum.getBytesPerChecksum() * getChecksumSize();
                // 往下追
                //  writeChunk() 方法由该类的子类 DFSOutputStream 来实现
                writeChunk(b, off + i, chunkLen, checksum, ckOffset,
                        getChecksumSize());
            }
        } finally {
            if (scope != null) {
                scope.close();
            }
        }
    }
```

2.2.2.4.1 创建 Packet 并 Packet 满了加入 dataQueue 队列并唤醒 DataStreamer 线程 (调用 DFSOutputStream.writeChunk())

```
// @see FSOutputSummer#writeChunk()
    @Override
    protected synchronized void writeChunk(byte[] b, int offset, int len,
                                           byte[] checksum, int ckoff, int cklen) throws
IOException {
        // 如果 currentPacket = null， 则生成一个 Packet (DFSPacket) 对象
        writeChunkPrepare(len, ckoff, cklen);

        // 写 checksum 4 个字节
        currentPacket.writeChecksum(checksum, ckoff, cklen);
        // 写数据 512B
        currentPacket.writeData(b, offset, len);
        // chunk 计数 +1
        currentPacket.incNumChunks();
        // 记录 offset
        getStreamer().incBytesCurBlock(len);

        // If packet is full, enqueue it for transmission
        // 1、达到最大 chunk 数量
        // 2、达到 blocksize 大小 证明这个 block 的数据写完 是最后一个 packet
        if (currentPacket.getNumChunks() == currentPacket.getMaxChunks() ||
                getStreamer().getBytesCurBlock() == blockSize) {
            // 往下追
            enqueueCurrentPacketFull();
        }
    }
```

2.2.2.4.1.1 创建 Packet (DFSPacket)

```
private synchronized void writeChunkPrepare(int buflen,
                                            int ckoff, int cklen) throws
IOException {
        dfsClient.checkOpen();
        checkClosed();

        if (buflen > bytesPerChecksum) {
            throw new IOException("writeChunk() buffer size is " + buflen +
                    " is larger than supported bytesPerChecksum " +
```

```
                    bytesPerChecksum);
            }
            if (cklen != 0 && cklen != getChecksumSize()) {
                throw new IOException("writeChunk() checksum size is supposed to be " +
                        getChecksumSize() + " but found to be " + cklen);
            }

            if (currentPacket == null) {
                // 创建 Package (DFSPacket)
                currentPacket = createPacket(packetSize,
                        chunksPerPacket, getStreamer()
                        .getBytesCurBlock(), getStreamer().getAndIncCurrentSeqno(), false);
                DFSClient.LOG.debug("WriteChunk allocating new packet seqno={},"
                                + " src={}, packetSize={}, chunksPerPacket={},
bytesCurBlock={}",
                        currentPacket.getSeqno(), src, packetSize, chunksPerPacket,
                        getStreamer().getBytesCurBlock() + ", " + this);
            }
        }
```

```
/**
     * Use {@link ByteArrayManager} to create buffer for non-heartbeat packets.
     */
    protected DFSPacket createPacket(int packetSize, int chunksPerPkt,
                                     long offsetInBlock, long seqno, boolean
lastPacketInBlock)
            throws InterruptedIOException {
        final byte[] buf;
        final int bufferSize = PacketHeader.PKT_MAX_HEADER_LEN + packetSize;

        try {
            buf = byteArrayManager.newByteArray(bufferSize);
        } catch (InterruptedException ie) {
            final InterruptedIOException iioe = new InterruptedIOException(
                    "seqno=" + seqno);
            iioe.initCause(ie);
            throw iioe;
        }

        // 创建 DFSPacket
        return new DFSPacket(buf, chunksPerPkt, offsetInBlock, seqno,
                getChecksumSize(), lastPacketInBlock);
    }
```

```
/**
     * Create a new packet.
     *
     * @param buf the buffer storing data and checksums
     * @param chunksPerPkt maximum number of chunks per packet.
     * @param offsetInBlock offset in bytes into the HDFS block.
     * @param seqno the sequence number of this packet
     * @param checksumSize the size of checksum
     * @param lastPacketInBlock if this is the last packet
     */
    public DFSPacket(byte[] buf, int chunksPerPkt, long offsetInBlock, long seqno,
                                int checksumSize, boolean lastPacketInBlock) {
        this.lastPacketInBlock = lastPacketInBlock;
        this.numChunks = 0;
        this.offsetInBlock = offsetInBlock;
        this.seqno = seqno;

        this.buf = buf;

        checksumStart = PacketHeader.PKT_MAX_HEADER_LEN;
        checksumPos = checksumStart;
        dataStart = checksumStart + (chunksPerPkt * checksumSize);
        dataPos = dataStart;
        maxChunks = chunksPerPkt;
    }
```

2.2.2.4.1.2 Packet 满了加入 dataQueue 队列并唤醒 DataStreamer 线程

```
    synchronized void enqueueCurrentPacketFull() throws IOException {
        LOG.debug("enqueue full {}, src={}, bytesCurBlock={}, blockSize={},"
                                + " appendChunk={}, {}", currentPacket, src, getStreamer()
                                .getBytesCurBlock(), blockSize, getStreamer().getAppendChunk(),
                        getStreamer());
        // 将构建好的数据 Packet 加入到 DataStreamer 的 dataQueue 队列中
        enqueueCurrentPacket();

        adjustChunkBoundary();
        endBlock();
    }
```

```
    void enqueueCurrentPacket() throws IOException {
        // 往下追
        getStreamer().waitAndQueuePacket(currentPacket);
```

```
            currentPacket = null;
        }
```

```
/**
         * wait for space of dataQueue and queue the packet
         *
         * @param packet the DFSPacket to be queued
         * @throws IOException
         */
        void waitAndQueuePacket(DFSPacket packet) throws IOException {
            synchronized (dataQueue) {
                try {
                    // If queue is full, then wait till we have enough space
                    boolean firstWait = true;
                    try {
                        while (!streamerClosed && dataQueue.size() + ackQueue.size() >
                                dfsClient.getConf().getWriteMaxPackets()) {
                            if (firstWait) {
                                Span span = Tracer.getCurrentSpan();
                                if (span != null) {
                                    span.addTimelineAnnotation("dataQueue.wait");
                                }
                                firstWait = false;
                            }
                            try {
                                dataQueue.wait();
                            } catch (InterruptedException e) {
                                // If we get interrupted while waiting to queue data, we still
need to get rid
                                // of the current packet. This is because we have an invariant
that if
                                // currentPacket gets full, it will get queued before the next
writeChunk.
                                //
                                // Rather than wait around for space in the queue, we should
instead try to
                                // return to the caller as soon as possible, even though we
slightly overrun
                                // the MAX_PACKETS length.
                                Thread.currentThread().interrupt();
                                break;
                            }
                        }
                    } finally {
```

```
                        Span span = Tracer.getCurrentSpan();
                        if ((span != null) && (!firstWait)) {
                            span.addTimelineAnnotation("end.wait");
                        }
                    }
                    checkClosed();
                    // 往下追
                    queuePacket(packet);
                } catch (ClosedChannelException ignored) {
                }
            }
        }
```

```
/**
     * Put a packet to the data queue
     *
     * @param packet the packet to be put into the data queued
     */
    void queuePacket(DFSPacket packet) {
        synchronized (dataQueue) {
            if (packet == null) return;
            packet.addTraceParent(Tracer.getCurrentSpanId());
            // 添加 Packet
            dataQueue.addLast(packet);
            // 更新上一次 Packet NO
            lastQueuedSeqno = packet.getSeqno();
            LOG.debug("Queued {}, {}", packet, this);
            // 唤醒
            dataQueue.notifyAll();
        }
    }
```

2.2.2.4.2 DataStreamer 线程被唤醒 (调用其 run())

```
/*
     * streamer thread is the only thread that opens streams to datanode,
     * and closes them. Any error recovery is also done by this thread.
     */
    @Override
    public void run() {
        long lastPacket = Time.monotonicNow();
        TraceScope scope = null;
```

```
while (!streamerClosed && dfsClient.clientRunning) {
    // if the Responder encountered an error, shutdown Responder
    if (errorState.hasError()) {
        closeResponder();
    }

    DFSPacket one;
    try {
        // process datanode IO errors if any
        boolean doSleep = processDatanodeOrExternalError();

        final int halfSocketTimeout = dfsClient.getConf().getSocketTimeout() / 2;
        synchronized (dataQueue) {
            // wait for a packet to be sent.
            long now = Time.monotonicNow();
            while ((!shouldStop() && dataQueue.size() == 0 &&
                    (stage != BlockConstructionStage.DATA_STREAMING ||
                            now - lastPacket < halfSocketTimeout)) || doSleep) {
                long timeout = halfSocketTimeout - (now - lastPacket);
                timeout = timeout <= 0 ? 1000 : timeout;
                timeout = (stage == BlockConstructionStage.DATA_STREAMING) ?
                        timeout : 1000;
                try {
                    dataQueue.wait(timeout);
                } catch (InterruptedException e) {
                    LOG.warn("Caught exception", e);
                }
                doSleep = false;
                now = Time.monotonicNow();
            }
            if (shouldStop()) {
                continue;
            }
            // get packet to be sent.
            if (dataQueue.isEmpty()) {
                one = createHeartbeatPacket();
            } else {
                try {
                    backOffIfNecessary();
                } catch (InterruptedException e) {
                    LOG.warn("Caught exception", e);
                }
                //  拉取一个  Package
                one = dataQueue.getFirst(); // regular data packet
```

```
                    SpanId[] parents = one.getTraceParents();
                    if (parents.length > 0) {
                        scope = dfsClient.getTracer().
                                newScope("dataStreamer", parents[0]);
                        scope.getSpan().setParents(parents);
                    }
                }
            }

            // get new block from namenode.
            if (LOG.isDebugEnabled()) {
                LOG.debug("stage=" + stage + ", " + this);
            }
            // 一开始 stage = PIPELINE_SETUP_CREATE
            if (stage == BlockConstructionStage.PIPELINE_SETUP_CREATE) {
                LOG.debug("Allocating new block: {}", this);

                // 设置 pipeline 信息
                setPipeline(
                        // 向 NameNode 申请 Block 并根据 Block 地址连接第一个 DataNode

                        // Socket pipeline 在里面建立
                        nextBlockOutputStream()
                );

                // 初始化数据流
                initDataStreaming();
            } else if (stage == BlockConstructionStage.PIPELINE_SETUP_APPEND) {
                LOG.debug("Append to block {}", block);
                setupPipelineForAppendOrRecovery();
                if (streamerClosed) {
                    continue;
                }
                initDataStreaming();
            }

            long lastByteOffsetInBlock = one.getLastByteOffsetBlock();
            if (lastByteOffsetInBlock > stat.getBlockSize()) {
                throw new IOException("BlockSize " + stat.getBlockSize() +
                        " < lastByteOffsetInBlock, " + this + ", " + one);
            }

            if (one.isLastPacketInBlock()) {
```

```
                    // wait for all data packets have been successfully acked
                    synchronized (dataQueue) {
                        while (!shouldStop() && ackQueue.size() != 0) {
                            try {
                                // wait for acks to arrive from datanodes
                                dataQueue.wait(1000);
                            } catch (InterruptedException e) {
                                LOG.warn("Caught exception", e);
                            }
                        }
                    }
                    if (shouldStop()) {
                        continue;
                    }
                    stage = BlockConstructionStage.PIPELINE_CLOSE;
                }

                // send the packet
                SpanId spanId = SpanId.INVALID;
                synchronized (dataQueue) {
                    // move packet from dataQueue to ackQueue
                    if (!one.isHeartbeatPacket()) {
                        if (scope != null) {
                            spanId = scope.getSpanId();
                            scope.detach();
                            one.setTraceScope(scope);
                        }
                        scope = null;
                        // 从 dataQueue 移除 Packet 到 ackQueue
                        dataQueue.removeFirst();
                        ackQueue.addLast(one);

                        // 添加要发送 packet 的 ID 以及当前时间戳
                        packetSendTime.put(one.getSeqno(), Time.monotonicNow());
                        dataQueue.notifyAll();
                    }
                }

                LOG.debug("{} sending {}", this, one);

                // write out data to remote datanode
                try (TraceScope ignored = dfsClient.getTracer().
                        newScope("DataStreamer#writeTo", spanId)) {
                    // 将 Packet 写到 blockStream (blockStream 就是与 DataNode
```

```
Socket 连接的输出流)
                    one.writeTo(blockStream);
                    // 刷新
                    blockStream.flush();
                } catch (IOException e) {
                    // HDFS-3398 treat primary DN is down since client is unable to
                    // write to primary DN. If a failed or restarting node has already
                    // been recorded by the responder, the following call will have no
                    // effect. Pipeline recovery can handle only one node error at a
                    // time. If the primary node fails again during the recovery, it
                    // will be taken out then.
                    errorState.markFirstNodeIfNotMarked();
                    throw e;
                }
                lastPacket = Time.monotonicNow();

                // update bytesSent
                long tmpBytesSent = one.getLastByteOffsetBlock();
                if (bytesSent < tmpBytesSent) {
                    bytesSent = tmpBytesSent;
                }

                if (shouldStop()) {
                    continue;
                }

                // Is this block full?
                if (one.isLastPacketInBlock()) {
                    // wait for the close packet has been acked
                    synchronized (dataQueue) {
                        while (!shouldStop() && ackQueue.size() != 0) {
                            dataQueue.wait(1000);// wait for acks to arrive from
datanodes
                        }
                    }
                    if (shouldStop()) {
                        continue;
                    }

                    endBlock();
                }
                if (progress != null) {
                    progress.progress();
                }
```

```
                    // This is used by unit test to trigger race conditions.
                    if (artificialSlowdown != 0 && dfsClient.clientRunning) {
                        Thread.sleep(artificialSlowdown);
                    }
                } catch (Throwable e) {
                    // Log warning if there was a real error.
                    if (!errorState.isRestartingNode()) {
                        // Since their messages are descriptive enough, do not always
                        // log a verbose stack-trace WARN for quota exceptions.
                        if (e instanceof QuotaExceededException) {
                            LOG.debug("DataStreamer Quota Exception", e);
                        } else {
                            LOG.warn("DataStreamer Exception", e);
                        }
                    }
                    lastException.set(e);
                    assert !(e instanceof NullPointerException);
                    errorState.setInternalError();
                    if (!errorState.isNodeMarked()) {
                        // Not a datanode issue
                        streamerClosed = true;
                    }
                } finally {
                    if (scope != null) {
                        scope.close();
                        scope = null;
                    }
                }
            }
            closeInternal();
        }
```

2.2.2.4.2.1 向 NameNode 申请 Block 并根据 Block 地址连接第一个 DataNode

```
    /**
     * Open a DataStreamer to a DataNode so that it can be written to.
     * This happens when a file is created and each time a new block is allocated.
     * Must get block ID and the IDs of the destinations from the namenode.
     * Returns the list of target datanodes.
     */
    protected LocatedBlock nextBlockOutputStream() throws IOException {
```

```
LocatedBlock lb;
DatanodeInfo[] nodes;
StorageType[] nextStorageTypes;
String[] nextStorageIDs;

// 申请 Block 重试次数 默认 3
int count = dfsClient.getConf().getNumBlockWriteRetry();

boolean success;
final ExtendedBlock oldBlock = block.getCurrentBlock();
do {
    errorState.resetInternalError();
    lastException.clear();

    DatanodeInfo[] excluded = getExcludedNodes();
    // 向 NameNode 申请 Block
    lb = locateFollowingBlock(
            excluded.length > 0 ? excluded : null, oldBlock);
    block.setCurrentBlock(lb.getBlock());
    block.setNumBytes(0);

    bytesSent = 0;
    accessToken = lb.getBlockToken();
    nodes = lb.getLocations();
    nextStorageTypes = lb.getStorageTypes();
    nextStorageIDs = lb.getStorageIDs();

    // Connect to first DataNode in the list.
    // 连接第一个 DataNode
    success = createBlockOutputStream(nodes, nextStorageTypes, nextStorageIDs,
            0L, false);

    // true
    if (!success) {
        LOG.warn("Abandoning " + block);
        dfsClient.namenode.abandonBlock(block.getCurrentBlock(),
                stat.getFileId(), src, dfsClient.clientName);
        block.setCurrentBlock(null);
        final DatanodeInfo badNode = nodes[errorState.getBadNodeIndex()];
        LOG.warn("Excluding datanode " + badNode);
        excludedNodes.put(badNode, badNode);
    }
} while (!success && --count >= 0);
```

```
        if (!success) {
            throw new IOException("Unable to create new block.");
        }
        return lb;
    }
```

2.2.2.4.2.1.1 向 NameNode 申请 Block（调用 NameNodeRpcServer.addBlock())

```
    private LocatedBlock locateFollowingBlock(DatanodeInfo[] excluded,
                                              ExtendedBlock oldBlock) throws
IOException {
        // 往下追
        return DFSOutputStream.addBlock(excluded, dfsClient, src, oldBlock,
                stat.getFileId(), favoredNodes, addBlockFlags);
    }
```

```
static LocatedBlock addBlock(DatanodeInfo[] excludedNodes,
                             DFSClient dfsClient, String src, ExtendedBlock prevBlock,
long fileId,
                             String[] favoredNodes, EnumSet<AddBlockFlag>
allocFlags)
            throws IOException {
        final DfsClientConf conf = dfsClient.getConf();
        int retries = conf.getNumBlockWriteLocateFollowingRetry();
        long sleeptime = conf.getBlockWriteLocateFollowingInitialDelayMs();
        long localstart = Time.monotonicNow();
        while (true) {
            try {
                // 发送 RPC 请求申请 Block
                return dfsClient.namenode.addBlock(src, dfsClient.clientName, prevBlock,
                        excludedNodes, fileId, favoredNodes, allocFlags);
            } catch (RemoteException e) {
                IOException ue = e.unwrapRemoteException(FileNotFoundException.class,
                        AccessControlException.class,
                        NSQuotaExceededException.class,
                        DSQuotaExceededException.class,
                        QuotaByStorageTypeExceededException.class,
                        UnresolvedPathException.class);
                if (ue != e) {
                    throw ue; // no need to retry these exceptions
                }
                if (NotReplicatedYetException.class.getName()
```

```
                    .equals(e.getClassName())) {
                if (retries == 0) {
                    throw e;
                } else {
                    --retries;
                    LOG.info("Exception while adding a block", e);
                    long elapsed = Time.monotonicNow() - localstart;
                    if (elapsed > 5000) {
                        LOG.info("Waiting for replication for " + (elapsed / 1000)
                                + " seconds");
                    }
                    try {
                        LOG.warn("NotReplicatedYetException sleeping " + src
                                + " retries left " + retries);
                        Thread.sleep(sleeptime);
                        sleeptime *= 2;
                    } catch (InterruptedException ie) {
                        LOG.warn("Caught exception", ie);
                    }
                }
            } else {
                throw e;
            }
        }
    }
```

```
@Override
    public LocatedBlock addBlock(String src, String clientName,
                                 ExtendedBlock previous, DatanodeInfo[] excludedNodes,
long fileId,
                                 String[] favoredNodes, EnumSet<AddBlockFlag>
addBlockFlags)
            throws IOException {
        checkNNStartup();
        // 申请 Block
        LocatedBlock locatedBlock = namesystem.getAdditionalBlock(src, fileId,
                clientName, previous, excludedNodes, favoredNodes, addBlockFlags);
        if (locatedBlock != null) {
            metrics.incrAddBlockOps();
        }
        return locatedBlock;
    }
```

```
/**
     * The client would like to obtain an additional block for the indicated
     * filename (which is being written-to).    Return an array that consists
     * of the block, plus a set of machines.    The first on this list should
     * be where the client writes data.    Subsequent items in the list must
     * be provided in the connection to the first datanode.
     * <p>
     * Make sure the previous blocks have been reported by datanodes and
     * are replicated.    Will return an empty 2-elt array if we want the
     * client to "try again later".
     */
    LocatedBlock getAdditionalBlock(
            String src, long fileId, String clientName, ExtendedBlock previous,
            DatanodeInfo[] excludedNodes, String[] favoredNodes,
            EnumSet<AddBlockFlag> flags) throws IOException {
        final String operationName = "getAdditionalBlock";
        NameNode.stateChangeLog.debug("BLOCK* getAdditionalBlock: {}    inodeId {}" +
                " for {}", src, fileId, clientName);

        LocatedBlock[] onRetryBlock = new LocatedBlock[1];
        FSDirWriteFileOp.ValidateAddBlockResult r;
        checkOperation(OperationCategory.READ);
        final FSPermissionChecker pc = getPermissionChecker();
        readLock();
        try {
            checkOperation(OperationCategory.READ);
            // 申请 Block 校验相关
            r = FSDirWriteFileOp.validateAddBlock(this, pc, src, fileId, clientName,
                    previous, onRetryBlock);
        } finally {
            readUnlock(operationName);
        }

        if (r == null) {
            assert onRetryBlock[0] != null : "Retry block is null";
            // This is a retry. Just return the last block.
            return onRetryBlock[0];
        }

        // 选择目标 Block 对应的 DataNode 地址
        DatanodeStorageInfo[] targets = FSDirWriteFileOp.chooseTargetForNewBlock(
                blockManager, src, excludedNodes, favoredNodes, flags, r);

        checkOperation(OperationCategory.WRITE);
```

```
            writeLock();
            LocatedBlock lb;
            try {
                checkOperation(OperationCategory.WRITE);
                // 将选择 DatanodeStorageInfo 封装成 LocatedBlock 并更新内存元数据 INodeFile 对应的 Block 信息
                // 以及磁盘数据与 JournalNodes
                lb = FSDirWriteFileOp.storeAllocatedBlock(
                        this, src, fileId, clientName, previous, targets);
            } finally {
                writeUnlock(operationName);
            }
            getEditLog().logSync();
            return lb;
        }
```

```
static DatanodeStorageInfo[] chooseTargetForNewBlock(
                BlockManager bm, String src, DatanodeInfo[] excludedNodes,
                String[] favoredNodes, EnumSet<AddBlockFlag> flags,
                ValidateAddBlockResult r) throws IOException {
            Node clientNode = null;

            boolean ignoreClientLocality = (flags != null
                    && flags.contains(AddBlockFlag.IGNORE_CLIENT_LOCALITY));

            // If client locality is ignored, clientNode remains 'null' to indicate
            if (!ignoreClientLocality) {
                clientNode = bm.getDatanodeManager().getDatanodeByHost(r.clientMachine);
                if (clientNode == null) {
                    clientNode = getClientNode(bm, r.clientMachine);
                }
            }

            // 黑名单
            Set<Node> excludedNodesSet =
                    (excludedNodes == null) ? new HashSet<>()
                            : new HashSet<>(Arrays.asList(excludedNodes));

            // 白名单
            List<String> favoredNodesList =
                    (favoredNodes == null) ? Collections.emptyList()
                            : Arrays.asList(favoredNodes);

            // choose targets for the new block to be allocated.
```

```
            // 往下追
            return bm.chooseTarget4NewBlock(src, r.numTargets, clientNode,
                    excludedNodesSet, r.blockSize,
                    favoredNodesList, r.storagePolicyID,
                    r.blockType, r.ecPolicy, flags);
        }
```

```
 /**
         * Choose target datanodes for creating a new block.
         *
         * @throws IOException if the number of targets < minimum replication.
         * @see BlockPlacementPolicy#chooseTarget(String, int, Node,
         * Set, long, List, BlockStoragePolicy, EnumSet)
         */
        public DatanodeStorageInfo[] chooseTarget4NewBlock(final String src,
                                                           final int numOfReplicas, final
Node client,
                                                           final                Set<Node>
excludedNodes,
                                                           final long blocksize,
                                                           final                List<String>
favoredNodes,
                                                           final byte storagePolicyID,
                                                           final BlockType blockType,
                                                           final       ErasureCodingPolicy
ecPolicy,
                                                           final
EnumSet<AddBlockFlag> flags) throws IOException {
            // 获取白名单对应的 DatanodeDescriptor
            List<DatanodeDescriptor> favoredDatanodeDescriptors =
                    getDatanodeDescriptors(favoredNodes);

            final BlockStoragePolicy storagePolicy =
                    storagePolicySuite.getPolicy(storagePolicyID);
            final BlockPlacementPolicy blockplacement =
                    placementPolicies.getPolicy(blockType);

            // 选择 DataNode
            final DatanodeStorageInfo[] targets = blockplacement.chooseTarget(src,
                    numOfReplicas, client, excludedNodes, blocksize,
                    favoredDatanodeDescriptors, storagePolicy, flags);

            final String errorMessage = "File %s could only be written to %d of " +
```

```
                    "the %d %s. There are %d datanode(s) running and %s "
                    + "node(s) are excluded in this operation.";
            if (blockType == BlockType.CONTIGUOUS && targets.length < minReplication) {
                throw new IOException(String.format(errorMessage, src,
                        targets.length, minReplication, "minReplication nodes",
                        getDatanodeManager().getNetworkTopology().getNumOfLeaves(),
                        (excludedNodes == null ? "no" : excludedNodes.size())));
            } else if (blockType == BlockType.STRIPED &&
                    targets.length < ecPolicy.getNumDataUnits()) {
                throw new IOException(
                        String.format(errorMessage, src, targets.length,
                                ecPolicy.getNumDataUnits(),
                                String.format("required nodes for %s", ecPolicy.getName()),

getDatanodeManager().getNetworkTopology().getNumOfLeaves(),
                                (excludedNodes == null ? "no" : excludedNodes.size())));
            }
            return targets;
        }
```

2.2.2.4.2.1.2 连接第一个 DataNode (根据申请 Block 信息)

```
// connects to the first datanode in the pipeline
        // Returns true if success, otherwise return failure.
        //
        boolean createBlockOutputStream(DatanodeInfo[] nodes,
                                        StorageType[]       nodeStorageTypes,       String[]
nodeStorageIDs,
                                        long newGS, boolean recoveryFlag) {
            if (nodes.length == 0) {
                LOG.info("nodes are empty for write pipeline of " + block);
                return false;
            }
            String firstBadLink = "";
            boolean checkRestart = false;
            if (LOG.isDebugEnabled()) {
                LOG.debug("pipeline = " + Arrays.toString(nodes) + ", " + this);
            }

            // persist blocks on namenode on next flush
            persistBlocks.set(true);

            int refetchEncryptionKey = 1;
```

```
while (true) {
    boolean result = false;
    DataOutputStream out = null;
    try {
        assert null == s : "Previous socket unclosed";
        assert null == blockReplyStream : "Previous blockReplyStream unclosed";

        // 创建与第一个 DataNode 的 socket 连接
        s = createSocketForPipeline(nodes[0], nodes.length, dfsClient);

        long writeTimeout = dfsClient.getDatanodeWriteTimeout(nodes.length);
        long readTimeout = dfsClient.getDatanodeReadTimeout(nodes.length);

        // 获取 Socket 的输出输入流
        OutputStream unbufOut = NetUtils.getOutputStream(s, writeTimeout);
        InputStream unbufIn = NetUtils.getInputStream(s, readTimeout);

        IOStreamPair saslStreams = dfsClient.saslClient.socketSend(s,
                unbufOut, unbufIn, dfsClient, accessToken, nodes[0]);
        unbufOut = saslStreams.out;
        unbufIn = saslStreams.in;

        // 封装 Socket 的输出流为 DataOutputStream
        out = new DataOutputStream(new BufferedOutputStream(unbufOut,
                DFSUtilClient.getSmallBufferSize(dfsClient.getConfiguration())));

        // 封装 Socket 的输入流为 DataInputStream
        blockReplyStream = new DataInputStream(unbufIn);

        //
        // Xmit header info to datanode
        //

        BlockConstructionStage bcs = recoveryFlag ?
                stage.getRecoveryStage() : stage;

        // We cannot change the block length in 'block' as it counts the number
        // of bytes ack'ed.
        ExtendedBlock blockCopy = block.getCurrentBlock();
        blockCopy.setNumBytes(stat.getBlockSize());

        boolean[] targetPinnings = getPinnings(nodes);

        // send the request
```

```
                    // 创建 Sender
                    new Sender(out)
                            // 发送请求写 Block
                            .writeBlock(blockCopy, nodeStorageTypes[0], accessToken,
                                    dfsClient.clientName, nodes, nodeStorageTypes, null, bcs,
                                    nodes.length, block.getNumBytes(), bytesSent, newGS,
                                    checksum4WriteBlock,                cachingStrategy.get(),
isLazyPersistFile,
                                    (targetPinnings     !=     null     &&     targetPinnings[0]),
targetPinnings,
                                    nodeStorageIDs[0], nodeStorageIDs);

                    // receive ack for connect
                    BlockOpResponseProto resp = BlockOpResponseProto.parseFrom(
                            PBHelperClient.vintPrefixed(blockReplyStream));
                    Status pipelineStatus = resp.getStatus();
                    firstBadLink = resp.getFirstBadLink();

                    // Got an restart OOB ack.
                    // If a node is already restarting, this status is not likely from
                    // the same node. If it is from a different node, it is not
                    // from the local datanode. Thus it is safe to treat this as a
                    // regular node error.
                    if (PipelineAck.isRestartOOBStatus(pipelineStatus) &&
                            !errorState.isRestartingNode()) {
                        checkRestart = true;
                        throw new IOException("A datanode is restarting.");
                    }

                    String logInfo = "ack with firstBadLink as " + firstBadLink;
                    DataTransferProtoUtil.checkBlockOpStatus(resp, logInfo);

                    assert null == blockStream : "Previous blockStream unclosed";
                    // 赋值
                    blockStream = out;
                    result = true; // success
                    errorState.resetInternalError();
                    lastException.clear();
                    // remove all restarting nodes from failed nodes list
                    failed.removeAll(restartingNodes);
                    restartingNodes.clear();
                } catch (IOException ie) {
                    if (!errorState.isRestartingNode()) {
                        LOG.info("Exception in createBlockOutputStream " + this, ie);
```

```
                }
                if (ie instanceof InvalidEncryptionKeyException &&
                        refetchEncryptionKey > 0) {
                    LOG.info("Will fetch a new encryption key and retry, "
                            + "encryption key was invalid when connecting to "
                            + nodes[0] + " : " + ie);
                    // The encryption key used is invalid.
                    refetchEncryptionKey--;
                    dfsClient.clearDataEncryptionKey();
                    // Don't close the socket/exclude this node just yet. Try again with
                    // a new encryption key.
                    continue;
                }

                // find the datanode that matches
                if (firstBadLink.length() != 0) {
                    for (int i = 0; i < nodes.length; i++) {
                        // NB: Unconditionally using the xfer addr w/o hostname
                        if (firstBadLink.equals(nodes[i].getXferAddr())) {
                            errorState.setBadNodeIndex(i);
                            break;
                        }
                    }
                } else {
                    assert !checkRestart;
                    errorState.setBadNodeIndex(0);
                }

                final int i = errorState.getBadNodeIndex();
                // Check whether there is a restart worth waiting for.
                if (checkRestart) {
                    errorState.initRestartingNode(i,
                            "Datanode " + i + " is restarting: " + nodes[i],
                            shouldWaitForRestart(i));
                }
                errorState.setInternalError();
                lastException.set(ie);
                result = false;    // error
            } finally {
                if (!result) {
                    IOUtils.closeSocket(s);
                    s = null;
                    IOUtils.closeStream(out);
                    IOUtils.closeStream(blockReplyStream);
```

```
                        blockReplyStream = null;
                    }
                }
                return result;
            }
        }
```

2.2.2.4.2.1.2.1 创建与第一个 DataNode 的 socket 连接

```
/**
     * Create a socket for a write pipeline
     *
     * @param first    the first datanode
     * @param length the pipeline length
     * @param client client
     * @return the socket connected to the first datanode
     */
    static Socket createSocketForPipeline(final DatanodeInfo first,
                                          final int length, final DFSClient client) throws
IOException {
        final DfsClientConf conf = client.getConf();
        final String dnAddr = first.getXferAddr(conf.isConnectToDnViaHostname());
        LOG.debug("Connecting to datanode {}", dnAddr);

        // DataNode 地址
        final InetSocketAddress isa = NetUtils.createSocketAddr(dnAddr);
        // 创建 Socket
        final Socket sock = client.socketFactory.createSocket();

        final int timeout = client.getDatanodeReadTimeout(length);
        // 连接 DataNode (本质连接 DataNode 中 DataXceiverServer[run()] 的
TcpPeerServer)
        NetUtils.connect(sock, isa, client.getRandomLocalInterfaceAddr(),
                conf.getSocketTimeout());
        sock.setTcpNoDelay(conf.getDataTransferTcpNoDelay());
        sock.setSoTimeout(timeout);
        sock.setKeepAlive(true);
        if (conf.getSocketSendBufferSize() > 0) {
            sock.setSendBufferSize(conf.getSocketSendBufferSize());
        }
        LOG.debug("Send buf size {}", sock.getSendBufferSize());
        return sock;
    }
```

2.2.2.4.2.1.2.2 DataNode 接收 Socket 连接 (调用 DataXceiverServer.run())

```
@Override
    public void run() {
        Peer peer = null;
        while (datanode.shouldRun && !datanode.shutdownForUpgrade) {
            try {
                // TcpPeerServer 等待接收 SocketChannel 连接 (阻塞)
                peer = peerServer.accept();

                // Make sure the xceiver count is not exceeded
                int curXceiverCount = datanode.getXceiverCount();
                // 4096
                if (curXceiverCount > maxXceiverCount) {
                    throw new IOException("Xceiver count " + curXceiverCount
                            + " exceeds the limit of concurrent xcievers: "
                            + maxXceiverCount);
                }

                // 将 DataXceiverServer 的 TcpPeerServer TCP-Server 接收到一个 Socket
                // 封装成 DataXceiver 线程 并调用其 run() 进行接收数据处理
                new Daemon(datanode.threadGroup,
                        DataXceiver.create(peer, datanode, this))
                        .start();
            } catch (SocketTimeoutException ignored) {
                // wake up to see if should continue to run
            } catch (AsynchronousCloseException ace) {
                // another thread closed our listener socket - that's expected during
shutdown,
                // but not in other circumstances
                if (datanode.shouldRun && !datanode.shutdownForUpgrade) {
                    LOG.warn(datanode.getDisplayName() + ":DataXceiverServer: ", ace);
                }
            } catch (IOException ie) {
                IOUtils.cleanup(null, peer);
                LOG.warn(datanode.getDisplayName() + ":DataXceiverServer: ", ie);
            } catch (OutOfMemoryError ie) {
                IOUtils.cleanup(null, peer);
                // DataNode can run out of memory if there is too many transfers.
                // Log the event, Sleep for 30 seconds, other transfers may complete by
                // then.
                LOG.error("DataNode is out of memory. Will retry in 30 seconds.", ie);
                try {
```

```
                        Thread.sleep(30 * 1000);
                    } catch (InterruptedException e) {
                        // ignore
                    }
                } catch (Throwable te) {
                    LOG.error(datanode.getDisplayName()
                            + ":DataXceiverServer: Exiting due to: ", te);
                    datanode.shouldRun = false;
                }
            }

            // Close the server to stop reception of more requests.
            try {
                peerServer.close();
                closed = true;
            } catch (IOException ie) {
                LOG.warn(datanode.getDisplayName()
                        + " :DataXceiverServer: close exception", ie);
            }

            // if in restart prep stage, notify peers before closing them.
            if (datanode.shutdownForUpgrade) {
                restartNotifyPeers();
                // Each thread needs some time to process it. If a thread needs
                // to send an OOB message to the client, but blocked on network for
                // long time, we need to force its termination.
                LOG.info("Shutting down DataXceiverServer before restart");
                // Allow roughly up to 2 seconds.
                for (int i = 0; getNumPeers() > 0 && i < 10; i++) {
                    try {
                        Thread.sleep(200);
                    } catch (InterruptedException e) {
                        // ignore
                    }
                }
            }
            // Close all peers.
            closeAllPeers();
        }
```

2.2.2.4.2.1.2.3 创建 Sender (发送请求写一个请求并后续往 out 输入 Packet)

```
    /** Create a sender for DataTransferProtocol with a output stream. */
```

```
  public Sender(final DataOutputStream out) {
    // out = DataOutputStream
    this.out = out;
  }
```

```
@Override
  public void writeBlock(final ExtendedBlock blk,
        final StorageType storageType,
        final Token<BlockTokenIdentifier> blockToken,
        final String clientName,
        final DatanodeInfo[] targets,
        final StorageType[] targetStorageTypes,
        final DatanodeInfo source,
        final BlockConstructionStage stage,
        final int pipelineSize,
        final long minBytesRcvd,
        final long maxBytesRcvd,
        final long latestGenerationStamp,
        DataChecksum requestedChecksum,
        final CachingStrategy cachingStrategy,
        final boolean allowLazyPersist,
        final boolean pinning,
        final boolean[] targetPinnings,
        final String storageId,
        final String[] targetStorageIds) throws IOException {
    ClientOperationHeaderProto header = DataTransferProtoUtil.buildClientHeader(
        blk, clientName, blockToken);

    ChecksumProto checksumProto =
        DataTransferProtoUtil.toProto(requestedChecksum);

    // 构建发送 Block 请求对象
    OpWriteBlockProto.Builder proto = OpWriteBlockProto.newBuilder()
        .setHeader(header)
        .setStorageType(PBHelperClient.convertStorageType(storageType))
        .addAllTargets(PBHelperClient.convert(targets, 1))
        .addAllTargetStorageTypes(
// 剪切 targets [0,1,2] -> [1,2]
            PBHelperClient.convertStorageTypes(targetStorageTypes, 1))
        .setStage(toProto(stage))
        .setPipelineSize(pipelineSize)
        .setMinBytesRcvd(minBytesRcvd)
        .setMaxBytesRcvd(maxBytesRcvd)
        .setLatestGenerationStamp(latestGenerationStamp)
```

```
            .setRequestedChecksum(checksumProto)
            .setCachingStrategy(getCachingStrategy(cachingStrategy))
            .setAllowLazyPersist(allowLazyPersist)
            .setPinning(pinning)
            .addAllTargetPinnings(PBHelperClient.convert(targetPinnings, 1))
            .addAllTargetStorageIds(PBHelperClient.convert(targetStorageIds, 1));
    if (source != null) {
        proto.setSource(PBHelperClient.convertDatanodeInfo(source));
    }
    if (storageId != null) {
        proto.setStorageId(storageId);
    }

    // 往下追
    send(out, Op.WRITE_BLOCK, proto.build());
  }
```

```
  private static void send(final DataOutputStream out, final Op opcode,
        final Message proto) throws IOException {
    LOG.trace("Sending DataTransferOp {}: {}",
          proto.getClass().getSimpleName(), proto);
    op(out, opcode);
    proto.writeDelimitedTo(out);
    out.flush();
  }
```

2.2.2.4.2.2 初始化数据流

```
/**
     * Initialize for data streaming
     */
    private void initDataStreaming() {
        this.setName("DataStreamer for file " + src +
                " block " + block);
        if (LOG.isDebugEnabled()) {
            LOG.debug("nodes {} storageTypes {} storageIDs {}",
                    Arrays.toString(nodes),
                    Arrays.toString(storageTypes),
                    Arrays.toString(storageIDs));
        }
        // 创建 ResponseProcessor 线程并启动
        response = new ResponseProcessor(nodes);
        response.start();
```

```
            // stage = DATA_STREAMING
            stage = BlockConstructionStage.DATA_STREAMING;
        }
```

2.2.2.4.2.2.1 创建 ResponseProcessor 线程并启动 (调用其 run())

```
private class ResponseProcessor extends Daemon {

            private volatile boolean responderClosed = false;
            private DatanodeInfo[] targets = null;
            private boolean isLastPacketInBlock = false;

            ResponseProcessor(DatanodeInfo[] targets) {
                  this.targets = targets;
            }

            @Override
            public void run() {

                  setName("ResponseProcessor for block " + block);
                  PipelineAck ack = new PipelineAck();

                  TraceScope scope = null;
                  while (!responderClosed && dfsClient.clientRunning && !isLastPacketInBlock) {
                        // process responses from datanodes.
                        try {
                              // read an ack from the pipeline
                              // blockReplyStream 就是与第一个 DataNode 建立 Socket 连接获取
的输入流
                              ack.readFields(blockReplyStream);
                              if (ack.getSeqno() != DFSPacket.HEART_BEAT_SEQNO) {
                                    Long begin = packetSendTime.get(ack.getSeqno());
                                    if (begin != null) {
                                          long duration = Time.monotonicNow() - begin;
                                          if (duration > dfsclientSlowLogThresholdMs) {
                                                LOG.info("Slow ReadProcessor read fields for block " +
block
                                                            + " took " + duration + "ms (threshold="
                                                            + dfsclientSlowLogThresholdMs + "ms); ack: " +
ack
                                                            + ", targets: " + Arrays.asList(targets));
                                          }
                                    }
```

```
                        }

                        if (LOG.isDebugEnabled()) {
                            LOG.debug("DFSClient {}", ack);
                        }

                        long seqno = ack.getSeqno();
                        // processes response status from datanodes.
                        ArrayList<DatanodeInfo> congestedNodesFromAck = new ArrayList<>();
                        for (int i = ack.getNumOfReplies() - 1; i >= 0 && dfsClient.clientRunning;
i--) {
                            final Status reply = PipelineAck.getStatusFromHeader(ack
                                    .getHeaderFlag(i));
                            if (PipelineAck.getECNFromHeader(ack.getHeaderFlag(i)) ==
                                    PipelineAck.ECN.CONGESTED) {
                                congestedNodesFromAck.add(targets[i]);
                            }
                            // Restart will not be treated differently unless it is
                            // the local node or the only one in the pipeline.
                            if (PipelineAck.isRestartOOBStatus(reply)) {
                                final String message = "Datanode " + i + " is restarting: "
                                        + targets[i];
                                errorState.initRestartingNode(i, message,
                                        shouldWaitForRestart(i));
                                throw new IOException(message);
                            }
                            // node error
                            if (reply != SUCCESS) {
                                errorState.setBadNodeIndex(i); // mark bad datanode
                                throw new IOException("Bad response " + reply +
                                        " for " + block + " from datanode " + targets[i]);
                            }
                        }

                        if (!congestedNodesFromAck.isEmpty()) {
                            synchronized (congestedNodes) {
                                congestedNodes.clear();
                                congestedNodes.addAll(congestedNodesFromAck);
                            }
                        } else {
                            synchronized (congestedNodes) {
                                congestedNodes.clear();
                                lastCongestionBackoffTime = 0;
                            }
```

```
                    }

                    assert seqno != PipelineAck.UNKOWN_SEQNO :
                            "Ack for unknown seqno should be a failed ack: " + ack;
                    if (seqno == DFSPacket.HEART_BEAT_SEQNO) {    // a heartbeat ack
                        continue;
                    }

                    // a success ack for a data packet
                    // 成功发送一个 Packet 返回 ack 从而删除 ackQueue 对应的
Packet
                    DFSPacket one;
                    synchronized (dataQueue) {
                        one = ackQueue.getFirst();
                    }
                    if (one.getSeqno() != seqno) {
                        throw new IOException("ResponseProcessor: Expecting seqno " +
                                " for block " + block +
                                one.getSeqno() + " but received " + seqno);
                    }
                    isLastPacketInBlock = one.isLastPacketInBlock();

                    // Fail the packet write for testing in order to force a
                    // pipeline recovery.
                    if (DFSClientFaultInjector.get().failPacket() &&
                            isLastPacketInBlock) {
                        failPacket = true;
                        throw new IOException(
                                "Failing the last packet for testing.");
                    }

                    // update bytesAcked
                    block.setNumBytes(one.getLastByteOffsetBlock());

                    synchronized (dataQueue) {
                        scope = one.getTraceScope();
                        if (scope != null) {
                            scope.reattach();
                            one.setTraceScope(null);
                        }
                        lastAckedSeqno = seqno;
                        pipelineRecoveryCount = 0;
                        ackQueue.removeFirst();
                        packetSendTime.remove(seqno);
```

```
                                dataQueue.notifyAll();

                                one.releaseBuffer(byteArrayManager);
                            }
                        } catch (Exception e) {
                            if (!responderClosed) {
                                lastException.set(e);
                                errorState.setInternalError();
                                errorState.markFirstNodeIfNotMarked();
                                synchronized (dataQueue) {
                                    dataQueue.notifyAll();
                                }
                                if (!errorState.isRestartingNode()) {
                                    LOG.warn("Exception for " + block, e);
                                }
                                responderClosed = true;
                            }
                        } finally {
                            if (scope != null) {
                                scope.close();
                            }
                            scope = null;
                        }
                    }
                }

                void close() {
                    responderClosed = true;
                    this.interrupt();
                }
            }
```

2.2.2.4.2.2.2 Packet 写到 blockStream (blockStream 就是与 DataNode Socket 连接的输出流)

```
// move packet from dataQueue to ackQueue
                            if (!one.isHeartbeatPacket()) {
                                if (scope != null) {
                                    spanId = scope.getSpanId();
                                    scope.detach();
                                    one.setTraceScope(scope);
                                }
                                scope = null;
```

```
                        // 从 dataQueue 移除 Packet 到 ackQueue
                        dataQueue.removeFirst();
                        ackQueue.addLast(one);

                        // 添加要发送 packet 的 ID 以及当前时间戳
                        packetSendTime.put(one.getSeqno(), Time.monotonicNow());
                        dataQueue.notifyAll();
                    }

 // write out data to remote datanode
                try (TraceScope ignored = dfsClient.getTracer().
                        newScope("DataStreamer#writeTo", spanId)) {
                    // 将 Packet 写到 blockStream (blockStream 就是与 DataNode
Socket 连接的输出流)
                    one.writeTo(blockStream);
                    // 刷新
                    blockStream.flush();
```

2.2.2.4.3 Client 与 DataNode 建立 Socket pipeline

入口：DataXceiverServer.run()

```
@Override
    public void run() {
        Peer peer = null;
        while (datanode.shouldRun && !datanode.shutdownForUpgrade) {
            try {
                // TcpPeerServer 等待接收 SocketChannel 连接 (阻塞)
                peer = peerServer.accept();

                // Make sure the xceiver count is not exceeded
                int curXceiverCount = datanode.getXceiverCount();
                // 4096
                if (curXceiverCount > maxXceiverCount) {
                    throw new IOException("Xceiver count " + curXceiverCount
                            + " exceeds the limit of concurrent xcievers: "
                            + maxXceiverCount);
                }

                // 将 DataXceiverServer 的 TcpPeerServer TCP-Server 接收到一个 Socket
                // 封装成 DataXceiver 线程 并调用其 run() 进行接收数据处理
                new Daemon(datanode.threadGroup,
                        DataXceiver.create(peer, datanode, this))
                        .start();
            } catch (SocketTimeoutException ignored) {
```

```
                    // wake up to see if should continue to run
                } catch (AsynchronousCloseException ace) {
                    // another thread closed our listener socket - that's expected during shutdown,
                    // but not in other circumstances
                    if (datanode.shouldRun && !datanode.shutdownForUpgrade) {
                        LOG.warn(datanode.getDisplayName() + ":DataXceiverServer: ", ace);
                    }
                } catch (IOException ie) {
                    IOUtils.cleanup(null, peer);
                    LOG.warn(datanode.getDisplayName() + ":DataXceiverServer: ", ie);
                } catch (OutOfMemoryError ie) {
                    IOUtils.cleanup(null, peer);
                    // DataNode can run out of memory if there is too many transfers.
                    // Log the event, Sleep for 30 seconds, other transfers may complete by
                    // then.
                    LOG.error("DataNode is out of memory. Will retry in 30 seconds.", ie);
                    try {
                        Thread.sleep(30 * 1000);
                    } catch (InterruptedException e) {
                        // ignore
                    }
                } catch (Throwable te) {
                    LOG.error(datanode.getDisplayName()
                            + ":DataXceiverServer: Exiting due to: ", te);
                    datanode.shouldRun = false;
                }
            }

            // Close the server to stop reception of more requests.
            try {
                peerServer.close();
                closed = true;
            } catch (IOException ie) {
                LOG.warn(datanode.getDisplayName()
                        + " :DataXceiverServer: close exception", ie);
            }

            // if in restart prep stage, notify peers before closing them.
            if (datanode.shutdownForUpgrade) {
                restartNotifyPeers();
                // Each thread needs some time to process it. If a thread needs
                // to send an OOB message to the client, but blocked on network for
                // long time, we need to force its termination.
```

```
                LOG.info("Shutting down DataXceiverServer before restart");
                // Allow roughly up to 2 seconds.
                for (int i = 0; getNumPeers() > 0 && i < 10; i++) {
                    try {
                        Thread.sleep(200);
                    } catch (InterruptedException e) {
                        // ignore
                    }
                }
            }
            // Close all peers.
            closeAllPeers();
        }
```

```
/**
     * Read/write data from/to the DataXceiverServer.
     */
    @Override
    public void run() {
        int opsProcessed = 0;
        Op op = null;

        try {
            synchronized (this) {
                xceiver = Thread.currentThread();
            }

            // 缓存
            dataXceiverServer.addPeer(peer, Thread.currentThread(), this);
            peer.setWriteTimeout(datanode.getDnConf().socketWriteTimeout);

            InputStream input = socketIn;
            try {
                // 创建 IOStreamPair(in,out)
                IOStreamPair saslStreams = datanode.saslServer.receive(peer, socketOut,
                        socketIn, datanode.getXferAddress().getPort(),
                        datanode.getDatanodeId());
                // 封装 Socket 的输入流为 BufferedInputStream
                input = new BufferedInputStream(saslStreams.in,
                        smallBufferSize);
                // 赋值 Socket 的输出流为 socketOut
                socketOut = saslStreams.out;
            } catch (InvalidMagicNumberException imne) {
```

```
                    if (imne.isHandshake4Encryption()) {
                        LOG.info("Failed to read expected encryption handshake from client " +
                                "at {}. Perhaps the client " +
                                "is running an older version of Hadoop which does not support
" +
                                "encryption", peer.getRemoteAddressString(), imne);
                    } else {
                        LOG.info("Failed to read expected SASL data transfer protection " +
                                        "handshake from client at {}" +
                                        ". Perhaps the client is running an older version of
Hadoop " +
                                        "which does not support SASL data transfer
protection",
                                peer.getRemoteAddressString(), imne);
                    }
                    return;
                }

                // 调用父类
                super.initialize(
                        // 再次封装输入流为 DataInputStream
                        new DataInputStream(input)
                );

                // We process requests in a loop, and stay around for a short timeout.
                // This optimistic behaviour allows the other end to reuse connections.
                // Setting keepalive timeout to 0 disable this behavior.
                do {
                    updateCurrentThreadName("Waiting for operation #" + (opsProcessed + 1));

                    try {
                        if (opsProcessed != 0) {
                            assert dnConf.socketKeepaliveTimeout > 0;
                            peer.setReadTimeout(dnConf.socketKeepaliveTimeout);
                        } else {
                            peer.setReadTimeout(dnConf.socketTimeout);
                        }
                        // 读取数据类型（WRITE_BLOCK (80))
                        op = readOp();
                    } catch (InterruptedIOException ignored) {
                        // Time out while we wait for client rpc
                        break;
                    } catch (EOFException | ClosedChannelException e) {
                        // Since we optimistically expect the next op, it's quite normal to
```

```
                    // get EOF here.
                    LOG.debug("Cached {} closing after {} ops.    " +
                            "This message is usually benign.", peer, opsProcessed);
                    break;
                } catch (IOException err) {
                    incrDatanodeNetworkErrors();
                    throw err;
                }

                // restore normal timeout
                if (opsProcessed != 0) {
                    peer.setReadTimeout(dnConf.socketTimeout);
                }

                opStartTime = monotonicNow();

                //  处理  WRITE_BLOCK
                processOp(op);

                ++opsProcessed;
            } while ((peer != null) &&
                    (!peer.isClosed() && dnConf.socketKeepaliveTimeout > 0));
        } catch (Throwable t) {
            String s = datanode.getDisplayName() + ":DataXceiver error processing "
                    + ((op == null) ? "unknown" : op.name()) + " operation "
                    + " src: " + remoteAddress + " dst: " + localAddress;
            if (op == Op.WRITE_BLOCK && t instanceof ReplicaAlreadyExistsException) {
                // For WRITE_BLOCK, it is okay if the replica already exists since
                // client and replication may write the same block to the same datanode
                // at the same time.
                if (LOG.isTraceEnabled()) {
                    LOG.trace(s, t);
                } else {
                    LOG.info("{}; {}", s, t.toString());
                }
            } else if (op == Op.READ_BLOCK && t instanceof SocketTimeoutException) {
                String s1 =
                        "Likely the client has stopped reading, disconnecting it";
                s1 += " (" + s + ")";
                if (LOG.isTraceEnabled()) {
                    LOG.trace(s1, t);
                } else {
                    LOG.info("{}; {}", s1, t.toString());
                }
```

```
                } else if (t instanceof InvalidToken ||
                        t.getCause() instanceof InvalidToken) {
                    // The InvalidToken exception has already been logged in
                    // checkAccess() method and this is not a server error.
                    LOG.trace(s, t);
                } else {
                    LOG.error(s, t);
                }
            } finally {
                collectThreadLocalStates();
                LOG.debug("{}:Number of active connections is: {}",
                        datanode.getDisplayName(), datanode.getXceiverCount());
                updateCurrentThreadName("Cleaning up");
                if (peer != null) {
                    dataXceiverServer.closePeer(peer);
                    IOUtils.closeStream(in);
                }
            }
        }
```

2.2.2.4.3.1 创建 IOStreamPair(in,out)

```
/**
     * Receives SASL negotiation from a peer on behalf of a server.
     *
     * @param peer connection peer
     * @param underlyingOut connection output stream
     * @param underlyingIn connection input stream
     * @param xferPort data transfer port of DataNode accepting connection
     * @param datanodeId ID of DataNode accepting connection
     * @return new pair of streams, wrapped after SASL negotiation
     * @throws IOException for any error
     */
    public IOStreamPair receive(Peer peer, OutputStream underlyingOut,
        InputStream underlyingIn, int xferPort, DatanodeID datanodeId)
        throws IOException {
      if (dnConf.getEncryptDataTransfer()) {
        LOG.debug(
          "SASL server doing encrypted handshake for peer = {}, datanodeId = {}",
          peer, datanodeId);
        return getEncryptedStreams(peer, underlyingOut, underlyingIn);
      } else if (!UserGroupInformation.isSecurityEnabled()) {
        LOG.debug(
          "SASL server skipping handshake in unsecured configuration for "
```

```
            + "peer = {}, datanodeId = {}", peer, datanodeId);
        return new IOStreamPair(underlyingIn, underlyingOut);
      } else if (SecurityUtil.isPrivilegedPort(xferPort)) {
        LOG.debug(
            "SASL server skipping handshake in secured configuration for "
            + "peer = {}, datanodeId = {}", peer, datanodeId);
        return new IOStreamPair(underlyingIn, underlyingOut);
      } else if (dnConf.getSaslPropsResolver() != null) {
        LOG.debug(
            "SASL server doing general handshake for peer = {}, datanodeId = {}",
            peer, datanodeId);
        // 创建 IOStreamPair
        return getSaslStreams(peer, underlyingOut, underlyingIn);
      } else if (dnConf.getIgnoreSecurePortsForTesting()) {
        // It's a secured cluster using non-privileged ports, but no SASL.  The
        // only way this can happen is if the DataNode has
        // ignore.secure.ports.for.testing configured, so this is a rare edge case.
        LOG.debug(
            "SASL server skipping handshake in secured configuration with no SASL "
            + "protection configured for peer = {}, datanodeId = {}",
            peer, datanodeId);
        return new IOStreamPair(underlyingIn, underlyingOut);
      } else {
        // The error message here intentionally does not mention
        // ignore.secure.ports.for.testing.  That's intended for dev use only.
        // This code path is not expected to execute ever, because DataNode startup
        // checks for invalid configuration and aborts.
        throw new IOException(String.format("Cannot create a secured " +
            "connection if DataNode listens on unprivileged port (%d) and no " +
            "protection is defined in configuration property %s.",
            datanodeId.getXferPort(), DFS_DATA_TRANSFER_PROTECTION_KEY));
      }
    }
```

```
  /**
   * Receives SASL negotiation for general-purpose handshake.
   *
   * @param peer connection peer
   * @param underlyingOut connection output stream
   * @param underlyingIn connection input stream
   * @return new pair of streams, wrapped after SASL negotiation
   * @throws IOException for any error
   */
```

```
private IOStreamPair getSaslStreams(Peer peer, OutputStream underlyingOut,
      InputStream underlyingIn) throws IOException {
    if (peer.hasSecureChannel() ||
        dnConf.getTrustedChannelResolver().isTrusted(getPeerAddress(peer))) {
      // 创建 IOStreamPair
      return new IOStreamPair(underlyingIn, underlyingOut);
    }
```

```
public IOStreamPair(InputStream in, OutputStream out) {
    // socket 的输入输出流
    this.in = in;
    this.out = out;
  }
```

2.2.2.4.3.2 读取数据类型（WRITE_BLOCK (80))

```
    /** Read an Op.  It also checks protocol version. */
    protected final Op readOp() throws IOException {
        // 读取版本号 28 (可以从 Client Sender.writeBlock() 查看)
        final short version = in.readShort();
        if (version != DataTransferProtocol.DATA_TRANSFER_VERSION) {
            throw new IOException("Version Mismatch (Expected: " +
                    DataTransferProtocol.DATA_TRANSFER_VERSION +
                    ", Received: " + version + " )");
        }
        // 读取一个字节判断是什么类型的数据 返回 WRITE_BLOCK (80) -> (可以从 Client Sender.writeBlock() 查看)
        return Op.read(in);
    }
```

2.2.2.4.3.3 处理 WRITE_BLOCK

```
/** Process op by the corresponding method. */
    protected final void processOp(Op op) throws IOException {
        switch (op) {
            case READ_BLOCK:
                opReadBlock();
                break;
            case WRITE_BLOCK:
                // 往下追
                opWriteBlock(in);
                break;
            case REPLACE_BLOCK:
```

```
                    opReplaceBlock(in);
                    break;
                case COPY_BLOCK:
                    opCopyBlock(in);
                    break;
                case BLOCK_CHECKSUM:
                    opBlockChecksum(in);
                    break;
                case BLOCK_GROUP_CHECKSUM:
                    opStripedBlockChecksum(in);
                    break;
                case TRANSFER_BLOCK:
                    opTransferBlock(in);
                    break;
                case REQUEST_SHORT_CIRCUIT_FDS:
                    opRequestShortCircuitFds(in);
                    break;
                case RELEASE_SHORT_CIRCUIT_FDS:
                    opReleaseShortCircuitFds(in);
                    break;
                case REQUEST_SHORT_CIRCUIT_SHM:
                    opRequestShortCircuitShm(in);
                    break;
                default:
                    throw new IOException("Unknown op " + op + " in data stream");
            }
        }
```

```
/** Receive OP_WRITE_BLOCK */
    private void opWriteBlock(DataInputStream in) throws IOException {
        // 读取 Client 发送数据 (第一条数据 还不是 Packet)
        final OpWriteBlockProto proto = OpWriteBlockProto.parseFrom(vintPrefixed(in));
        // 其他 DataNode 连接信息
        final DatanodeInfo[] targets = PBHelperClient.convert(proto.getTargetsList());
        TraceScope traceScope = continueTraceSpan(proto.getHeader(),
                proto.getClass().getSimpleName());
        try {
            // 往下追 (调用 DataXceiver.writeBlock)

writeBlock(PBHelperClient.convert(proto.getHeader().getBaseHeader().getBlock()),
                    PBHelperClient.convertStorageType(proto.getStorageType()),
                    PBHelperClient.convert(proto.getHeader().getBaseHeader().getToken()),
                    proto.getHeader().getClientName(),
                    targets,
```

```
                        PBHelperClient.convertStorageTypes(proto.getTargetStorageTypesList(),
targets.length),
                        PBHelperClient.convert(proto.getSource()),
                        fromProto(proto.getStage()),
                        proto.getPipelineSize(),
                        proto.getMinBytesRcvd(), proto.getMaxBytesRcvd(),
                        proto.getLatestGenerationStamp(),
                        fromProto(proto.getRequestedChecksum()),
                        (proto.hasCachingStrategy() ?
                                getCachingStrategy(proto.getCachingStrategy()) :
                                CachingStrategy.newDefaultStrategy()),
                        (proto.hasAllowLazyPersist() ? proto.getAllowLazyPersist() : false),
                        (proto.hasPinning() ? proto.getPinning() : false),
                        (PBHelperClient.convertBooleanList(proto.getTargetPinningsList())),
                        proto.getStorageId(),
                        proto.getTargetStorageIdsList().toArray(new String[0]));
            } finally {
                if (traceScope != null) traceScope.close();
            }
        }
```

```
@Override
    public void writeBlock(final ExtendedBlock block,
                           final StorageType storageType,
                           final Token<BlockTokenIdentifier> blockToken,
                           final String clientname,
                           final DatanodeInfo[] targets,
                           final StorageType[] targetStorageTypes,
                           final DatanodeInfo srcDataNode,
                           final BlockConstructionStage stage,
                           final int pipelineSize,
                           final long minBytesRcvd,
                           final long maxBytesRcvd,
                           final long latestGenerationStamp,
                           DataChecksum requestedChecksum,
                           CachingStrategy cachingStrategy,
                           boolean allowLazyPersist,
                           final boolean pinning,
                           final boolean[] targetPinnings,
                           final String storageId,
                           final String[] targetStorageIds) throws IOException {

        previousOpClientName = clientname;
        updateCurrentThreadName("Receiving block " + block);
```

```
final boolean isDatanode = clientname.length() == 0;
final boolean isClient = !isDatanode;
final boolean isTransfer = stage == BlockConstructionStage.TRANSFER_RBW
        || stage == BlockConstructionStage.TRANSFER_FINALIZED;
allowLazyPersist = allowLazyPersist &&
        (dnConf.getAllowNonLocalLazyPersist() || peer.isLocal());
long size = 0;

// reply to upstream datanode or client
// 获取当前 Socket 的输出流 封装成 DataOutputStream
final DataOutputStream replyOut = getBufferedOutputStream();

int nst = targetStorageTypes.length;
StorageType[] storageTypes = new StorageType[nst + 1];
storageTypes[0] = storageType;
if (targetStorageTypes.length > 0) {
    System.arraycopy(targetStorageTypes, 0, storageTypes, 1, nst);
}

// To support older clients, we don't pass in empty storageIds
final int nsi = targetStorageIds.length;
final String[] storageIds;
if (nsi > 0) {
    storageIds = new String[nsi + 1];
    storageIds[0] = storageId;
    if (targetStorageTypes.length > 0) {
        System.arraycopy(targetStorageIds, 0, storageIds, 1, nsi);
    }
} else {
    storageIds = new String[0];
}
checkAccess(replyOut, isClient, block, blockToken, Op.WRITE_BLOCK,
        BlockTokenIdentifier.AccessMode.WRITE,
        storageTypes, storageIds);

// check single target for transfer-RBW/Finalized
if (isTransfer && targets.length > 0) {
    throw new IOException(stage + " does not support multiple targets "
            + Arrays.asList(targets));
}

if (LOG.isDebugEnabled()) {
    LOG.debug("opWriteBlock: stage={}, clientname={}\n    " +
                    "block    ={}, newGs={}, bytesRcvd=[{}, {}]\n    " +
```

```
                    "targets={}; pipelineSize={}, srcDataNode={}, pinning={}",
                stage, clientname, block, latestGenerationStamp, minBytesRcvd,
                maxBytesRcvd, Arrays.asList(targets), pipelineSize, srcDataNode,
                pinning);
        LOG.debug("isDatanode={}, isClient={}, isTransfer={}",
                isDatanode, isClient, isTransfer);
        LOG.debug("writeBlock receive buf size {} tcp no delay {}",
                peer.getReceiveBufferSize(), peer.getTcpNoDelay());
    }

    // We later mutate block's generation stamp and length, but we need to
    // forward the original version of the block to downstream mirrors, so
    // make a copy here.
    final ExtendedBlock originalBlock = new ExtendedBlock(block);
    if (block.getNumBytes() == 0) {
        block.setNumBytes(dataXceiverServer.estimateBlockSize);
    }

    LOG.info("Receiving {} src: {} dest: {}",
            block, remoteAddress, localAddress);

    // 连接下一个 DataNode 信息
    DataOutputStream mirrorOut = null;    // stream to next target
    DataInputStream mirrorIn = null;      // reply from next target
    Socket mirrorSock = null;             // socket to next target
    String mirrorNode = null;             // the name:port of next target

    String firstBadLink = "";             // first datanode that failed in connection setup
    Status mirrorInStatus = SUCCESS;
    final String storageUuid;
    final boolean isOnTransientStorage;
    try {
        final Replica replica;
        if (isDatanode ||
                stage != BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
            // open a block receiver
            // 打开 Block Receiver
            setCurrentBlockReceiver(
                    // 创建 BlockReceiver
                    getBlockReceiver(block, storageType, in,
                    peer.getRemoteAddressString(),
                    peer.getLocalAddressString(),
                    stage, latestGenerationStamp, minBytesRcvd, maxBytesRcvd,
                    clientname, srcDataNode, datanode, requestedChecksum,
```

```
                        cachingStrategy, allowLazyPersist, pinning, storageId));
                replica = blockReceiver.getReplica();
            } else {
                replica = datanode.data.recoverClose(
                        block, latestGenerationStamp, minBytesRcvd);
            }
            storageUuid = replica.getStorageUuid();
            isOnTransientStorage = replica.isOnTransientStorage();

            //
            // Connect to downstream machine, if appropriate
            //
            // 如果当前 DataNode 还需要连接下一个 DataNode 建立 Socket pipeline 通道
            if (targets.length > 0) {
                InetSocketAddress mirrorTarget = null;
                // Connect to backup machine
                // 连接 Socket pipeline 的下一个 DataNode
                mirrorNode = targets[0].getXferAddr(connectToDnViaHostname);
                LOG.debug("Connecting to datanode {}", mirrorNode);
                mirrorTarget = NetUtils.createSocketAddr(mirrorNode);
                // 获取下一个 DataNode 的 Socket 连接
                mirrorSock = datanode.newSocket();
                try {

                    DataNodeFaultInjector.get().failMirrorConnection();

                    int timeoutValue = dnConf.socketTimeout +
                            (HdfsConstants.READ_TIMEOUT_EXTENSION * targets.length);
                    int writeTimeout = dnConf.socketWriteTimeout +
                            (HdfsConstants.WRITE_TIMEOUT_EXTENSION * targets.length);
                    // 执行 Socket Connection
                    NetUtils.connect(mirrorSock, mirrorTarget, timeoutValue);
                    mirrorSock.setTcpNoDelay(dnConf.getDataTransferServerTcpNoDelay());
                    mirrorSock.setSoTimeout(timeoutValue);
                    mirrorSock.setKeepAlive(true);
                    if (dnConf.getTransferSocketSendBufferSize() > 0) {
                        mirrorSock.setSendBufferSize(
                                dnConf.getTransferSocketSendBufferSize());
                    }

                    // 获取下一个 Socket 的输入输出流
                    OutputStream unbufMirrorOut = NetUtils.getOutputStream(mirrorSock,
```

```
                        writeTimeout);
                InputStream unbufMirrorIn = NetUtils.getInputStream(mirrorSock);

                DataEncryptionKeyFactory keyFactory =
                        datanode.getDataEncryptionKeyFactoryForBlock(block);
                SecretKey secretKey = null;
                if (dnConf.overwriteDownstreamDerivedQOP) {
                    String bpid = block.getBlockPoolId();
                    BlockKey blockKey = datanode.blockPoolTokenSecretManager
                            .get(bpid).getCurrentKey();
                    secretKey = blockKey.getKey();
                }
                IOStreamPair saslStreams = datanode.saslClient.socketSend(
                        mirrorSock, unbufMirrorOut, unbufMirrorIn, keyFactory,
                        blockToken, targets[0], secretKey);
                unbufMirrorOut = saslStreams.out;
                unbufMirrorIn = saslStreams.in;
                // 封装下一个 Socket 的输入输出流
                mirrorOut = new DataOutputStream(new
BufferedOutputStream(unbufMirrorOut,
                        smallBufferSize));
                mirrorIn = new DataInputStream(unbufMirrorIn);

                String targetStorageId = null;
                if (targetStorageIds.length > 0) {
                    // Older clients may not have provided any targetStorageIds
                    targetStorageId = targetStorageIds[0];
                }
                if (targetPinnings != null && targetPinnings.length > 0) {
                    // 也是发送一个 Block (打开) 请求 (根据 Client 端相同 如果
下一个 DataNode 还需要连接再下一个 DataNode 的 Socket 请求类似即可)
                    new Sender(mirrorOut).writeBlock(originalBlock,
targetStorageTypes[0],
                            blockToken, clientname, targets, targetStorageTypes,
                            srcDataNode, stage, pipelineSize, minBytesRcvd,
maxBytesRcvd,
                            latestGenerationStamp, requestedChecksum,
cachingStrategy,
                            allowLazyPersist, targetPinnings[0], targetPinnings,
                            targetStorageId, targetStorageIds);
                } else {
                    new Sender(mirrorOut).writeBlock(originalBlock,
targetStorageTypes[0],
                            blockToken, clientname, targets, targetStorageTypes,
```

```
                                    srcDataNode, stage, pipelineSize, minBytesRcvd,
maxBytesRcvd,
                                    latestGenerationStamp, requestedChecksum,
cachingStrategy,
                                    allowLazyPersist, false, targetPinnings,
                                    targetStorageId, targetStorageIds);
                        }

                        mirrorOut.flush();

                        DataNodeFaultInjector.get().writeBlockAfterFlush();

                        // read connect ack (only for clients, not for replication req)
                        if (isClient) {
                            BlockOpResponseProto connectAck =

BlockOpResponseProto.parseFrom(PBHelperClient.vintPrefixed(mirrorIn));
                            mirrorInStatus = connectAck.getStatus();
                            firstBadLink = connectAck.getFirstBadLink();
                            if (mirrorInStatus != SUCCESS) {
                                LOG.debug("Datanode {} got response for connect" +
                                                "ack from downstream datanode with
firstbadlink as {}",
                                        targets.length, firstBadLink);
                            }
                        }

                    } catch (IOException e) {
                        if (isClient) {
                            BlockOpResponseProto.newBuilder()
                                    .setStatus(ERROR)
                                    // NB: Unconditionally using the xfer addr w/o hostname
                                    .setFirstBadLink(targets[0].getXferAddr())
                                    .build()
                                    .writeDelimitedTo(replyOut);
                            replyOut.flush();
                        }
                        IOUtils.closeStream(mirrorOut);
                        mirrorOut = null;
                        IOUtils.closeStream(mirrorIn);
                        mirrorIn = null;
                        IOUtils.closeSocket(mirrorSock);
                        mirrorSock = null;
                        if (isClient) {
```

```
                    LOG.error("{}:Exception transfering block {} to mirror {}",
                            datanode, block, mirrorNode, e);
                    throw e;
                } else {
                    LOG.info("{}:Exception transfering {} to mirror {}- continuing " +
                            "without the mirror", datanode, block, mirrorNode, e);
                    incrDatanodeNetworkErrors();
                }
            }
        }

        // send connect-ack to source for clients and not transfer-RBW/Finalized
        if (isClient && !isTransfer) {
            if (mirrorInStatus != SUCCESS) {
                LOG.debug("Datanode {} forwarding connect ack to upstream " +
                        "firstbadlink is {}", targets.length, firstBadLink);
            }
            BlockOpResponseProto.newBuilder()
                    .setStatus(mirrorInStatus)
                    .setFirstBadLink(firstBadLink)
                    .build()
                    .writeDelimitedTo(replyOut);
            replyOut.flush();
        }

        // receive the block and mirror to the next target
        // DataNode 接收 Block (Packet) 并发送给下一个 DataNode
        if (blockReceiver != null) {
            String mirrorAddr = (mirrorSock == null) ? null : mirrorNode;
            // 往下追
            blockReceiver.receiveBlock(mirrorOut, mirrorIn, replyOut,
                    mirrorAddr, null, targets, false);

            // send close-ack for transfer-RBW/Finalized
            if (isTransfer) {
                LOG.trace("TRANSFER: send close-ack");
                writeResponse(SUCCESS, null, replyOut);
            }
        }

        // update its generation stamp
        if (isClient &&
                stage == BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
            block.setGenerationStamp(latestGenerationStamp);
```

```
                    block.setNumBytes(minBytesRcvd);
                }

                // if this write is for a replication request or recovering
                // a failed close for client, then confirm block. For other client-writes,
                // the block is finalized in the PacketResponder.
                if (isDatanode ||
                        stage == BlockConstructionStage.PIPELINE_CLOSE_RECOVERY) {
                    datanode.closeBlock(block, null, storageUuid, isOnTransientStorage);
                    LOG.info("Received {} src: {} dest: {} of size {}",
                            block, remoteAddress, localAddress, block.getNumBytes());
                }

                if (isClient) {
                    size = block.getNumBytes();
                }
            } catch (IOException ioe) {
                LOG.info("opWriteBlock {} received exception {}",
                        block, ioe.toString());
                incrDatanodeNetworkErrors();
                throw ioe;
            } finally {
                // close all opened streams
                IOUtils.closeStream(mirrorOut);
                IOUtils.closeStream(mirrorIn);
                IOUtils.closeStream(replyOut);
                IOUtils.closeSocket(mirrorSock);
                IOUtils.closeStream(blockReceiver);
                setCurrentBlockReceiver(null);
            }

            //update metrics
            datanode.getMetrics().addWriteBlockOp(elapsed());
            datanode.getMetrics().incrWritesFromClient(peer.isLocal(), size);
        }
```

2.2.2.4.3.3.1 创建 BlockReceiver

```
/**
     * Separated for testing.
     */
    @VisibleForTesting
    BlockReceiver getBlockReceiver(
```

```
                final ExtendedBlock block, final StorageType storageType,
                final DataInputStream in,
                final String inAddr, final String myAddr,
                final BlockConstructionStage stage,
                final long newGs, final long minBytesRcvd, final long maxBytesRcvd,
                final String clientname, final DatanodeInfo srcDataNode,
                final DataNode dn, DataChecksum requestedChecksum,
                CachingStrategy cachingStrategy,
                final boolean allowLazyPersist,
                final boolean pinning,
                final String storageId) throws IOException {
        // 创建  BlockReceiver
        return new BlockReceiver(block, storageType, in,
                        inAddr, myAddr, stage, newGs, minBytesRcvd, maxBytesRcvd,
                        clientname, srcDataNode, dn, requestedChecksum,
                        cachingStrategy, allowLazyPersist, pinning, storageId);
    }
```

```
BlockReceiver(final ExtendedBlock block, final StorageType storageType,
                        final DataInputStream in,
                        final String inAddr, final String myAddr,
                        final BlockConstructionStage stage,
                        final long newGs, final long minBytesRcvd, final long maxBytesRcvd,
                        final String clientname, final DatanodeInfo srcDataNode,
                        final DataNode datanode, DataChecksum requestedChecksum,
                        CachingStrategy cachingStrategy,
                        final boolean allowLazyPersist,
                        final boolean pinning,
                        final String storageId) throws IOException {
        try {
            this.block = block;
            this.in = in;
            this.inAddr = inAddr;
            this.myAddr = myAddr;
            this.srcDataNode = srcDataNode;
            this.datanode = datanode;

            this.clientname = clientname;
            this.isDatanode = clientname.length() == 0;
            this.isClient = !this.isDatanode;
            this.restartBudget = datanode.getDnConf().restartReplicaExpiry;
            this.datanodeSlowLogThresholdMs =
                    datanode.getDnConf().getSlowIoWarningThresholdMs();
```

```
                // For replaceBlock() calls response should be sent to avoid socketTimeout
                // at clients. So sending with the interval of 0.5 * socketTimeout
                final long readTimeout = datanode.getDnConf().socketTimeout;
                this.responseInterval = (long) (readTimeout * 0.5);
                //for datanode, we have
                //1: clientName.length() == 0, and
                //2: stage == null or PIPELINE_SETUP_CREATE
                this.stage = stage;
                this.isTransfer = stage == BlockConstructionStage.TRANSFER_RBW
                        || stage == BlockConstructionStage.TRANSFER_FINALIZED;

                this.pinning = pinning;
                this.lastSentTime = Time.monotonicNow();
                // Downstream will timeout in readTimeout on receiving the next packet.
                // If there is no data traffic, a heartbeat packet is sent at
                // the interval of 0.5*readTimeout. Here, we set 0.9*readTimeout to be
                // the threshold for detecting congestion.
                this.maxSendIdleTime = (long) (readTimeout * 0.9);
                if (LOG.isDebugEnabled()) {
                    LOG.debug(getClass().getSimpleName() + ": " + block
                            + "\n storageType=" + storageType + ", inAddr=" + inAddr
                            + ", myAddr=" + myAddr + "\n stage=" + stage + ", newGs=" +
newGs
                            + ", minBytesRcvd=" + minBytesRcvd
                            + ", maxBytesRcvd=" + maxBytesRcvd + "\n clientname=" +
clientname
                            + ", srcDataNode=" + srcDataNode
                            + ", datanode=" + datanode.getDisplayName()
                            + "\n requestedChecksum=" + requestedChecksum
                            + "\n cachingStrategy=" + cachingStrategy
                            + "\n allowLazyPersist=" + allowLazyPersist + ", pinning=" + pinning
                            + ", isClient=" + isClient + ", isDatanode=" + isDatanode
                            + ", responseInterval=" + responseInterval
                            + ", storageID=" + (storageId != null ? storageId : "null")
                    );
                }

                //
                // Open local disk out
                //
                if (isDatanode) { //replication or move
                    replicaHandler =
                            datanode.data.createTemporary(storageType, storageId, block,
false);
```

```
                } else {
                    switch (stage) {
                        // 一开始来到这里
                        case PIPELINE_SETUP_CREATE:
                            // 调用 FsDatasetImpl.createRbw() 创建 ReplicaHandler
                            replicaHandler = datanode.data.createRbw(storageType, storageId,
                                    block, allowLazyPersist);
                            datanode.notifyNamenodeReceivingBlock(
                                    block, replicaHandler.getReplica().getStorageUuid());
                            break;
                        case PIPELINE_SETUP_STREAMING_RECOVERY:
                            replicaHandler = datanode.data.recoverRbw(
                                    block, newGs, minBytesRcvd, maxBytesRcvd);
                            block.setGenerationStamp(newGs);
                            break;
                        case PIPELINE_SETUP_APPEND:
                            replicaHandler = datanode.data.append(block, newGs,
minBytesRcvd);
                            block.setGenerationStamp(newGs);
                            datanode.notifyNamenodeReceivingBlock(
                                    block, replicaHandler.getReplica().getStorageUuid());
                            break;
                        case PIPELINE_SETUP_APPEND_RECOVERY:
                            replicaHandler = datanode.data.recoverAppend(block, newGs,
minBytesRcvd);
                            block.setGenerationStamp(newGs);
                            datanode.notifyNamenodeReceivingBlock(
                                    block, replicaHandler.getReplica().getStorageUuid());
                            break;
                        case TRANSFER_RBW:
                        case TRANSFER_FINALIZED:
                            // this is a transfer destination
                            replicaHandler = datanode.data.createTemporary(storageType,
storageId,
                                    block, isTransfer);
                            break;
                        default:
                            throw new IOException("Unsupported stage " + stage +
                                    " while receiving block " + block + " from " + inAddr);
                    }
                }
                replicaInfo = replicaHandler.getReplica();
                this.dropCacheBehindWrites = (cachingStrategy.getDropBehind() == null) ?
                        datanode.getDnConf().dropCacheBehindWrites :
```

```
                    cachingStrategy.getDropBehind();
            this.syncBehindWrites = datanode.getDnConf().syncBehindWrites;
            this.syncBehindWritesInBackground = datanode.getDnConf().
                    syncBehindWritesInBackground;

            final boolean isCreate = isDatanode || isTransfer
                    || stage == BlockConstructionStage.PIPELINE_SETUP_CREATE;
            streams = replicaInfo.createStreams(isCreate, requestedChecksum);
            assert streams != null : "null streams!";

            // read checksum meta information
            this.clientChecksum = requestedChecksum;
            this.diskChecksum = streams.getChecksum();
            this.needsChecksumTranslation = !clientChecksum.equals(diskChecksum);
            this.bytesPerChecksum = diskChecksum.getBytesPerChecksum();
            this.checksumSize = diskChecksum.getChecksumSize();

            this.checksumOut = new DataOutputStream(new BufferedOutputStream(
                    streams.getChecksumOut(), DFSUtilClient.getSmallBufferSize(
                    datanode.getConf())));
            // write data chunk header if creating a new replica
            if (isCreate) {
                BlockMetadataHeader.writeHeader(checksumOut, diskChecksum);
            }
        } catch (ReplicaAlreadyExistsException bae) {
            throw bae;
        } catch (ReplicaNotFoundException bne) {
            throw bne;
        } catch (IOException ioe) {
            if (replicaInfo != null) {
                replicaInfo.releaseAllBytesReserved();
            }
            IOUtils.closeStream(this);
            cleanupBlock();

            // check if there is a disk error
            IOException cause = DatanodeUtil.getCauseIfDiskError(ioe);
            DataNode.LOG
                    .warn("IOException in BlockReceiver constructor :" + ioe.getMessage()
                            + (cause == null ? "" : ". Cause is "), cause);
            if (cause != null) {
                ioe = cause;
                // Volume error check moved to FileIoProvider
            }
```

```
                throw ioe;
            }
        }
```

2.2.2.4.3.3.1.1 创建 ReplicaHandler (调用 FsDatasetImpl.createRbw())

```
@Override // FsDatasetSpi
    public ReplicaHandler createRbw(
            StorageType storageType, String storageId, ExtendedBlock b,
            boolean allowLazyPersist) throws IOException {
        try (AutoCloseableLock lock = datasetLock.acquire()) {
            ReplicaInfo replicaInfo = volumeMap.get(b.getBlockPoolId(),
                    b.getBlockId());
            if (replicaInfo != null) {
                throw new ReplicaAlreadyExistsException("Block " + b +
                        " already exists in state " + replicaInfo.getState() +
                        " and thus cannot be created.");
            }
            // create a new block
            FsVolumeReference ref = null;

            // Use ramdisk only if block size is a multiple of OS page size.
            // This simplifies reservation for partially used replicas
            // significantly.
            if (allowLazyPersist &&
                    lazyWriter != null &&
                    b.getNumBytes() % cacheManager.getOsPageSize() == 0 &&
                    reserveLockedMemory(b.getNumBytes())) {
                try {
                    // First try to place the block on a transient volume.
                    ref = volumes.getNextTransientVolume(b.getNumBytes());
                    datanode.getMetrics().incrRamDiskBlocksWrite();
                } catch (DiskOutOfSpaceException de) {
                    // Ignore the exception since we just fall back to persistent storage.
                    LOG.warn("Insufficient space for placing the block on a transient "
                            + "volume, fall back to persistent storage: "
                            + de.getMessage());
                } finally {
                    if (ref == null) {
                        cacheManager.release(b.getNumBytes());
                    }
                }
```

```
        }

        if (ref == null) {
            ref = volumes.getNextVolume(storageType, storageId, b.getNumBytes());
        }

        FsVolumeImpl v = (FsVolumeImpl) ref.getVolume();
        // create an rbw file to hold block in the designated volume

        if (allowLazyPersist && !v.isTransientStorage()) {
            datanode.getMetrics().incrRamDiskBlocksWriteFallback();
        }

        ReplicaInPipeline newReplicaInfo;
        try {
            newReplicaInfo = v.createRbw(b);
            if (newReplicaInfo.getReplicaInfo().getState() != ReplicaState.RBW) {
                throw new IOException("CreateRBW returned a replica of state "
                        + newReplicaInfo.getReplicaInfo().getState()
                        + " for block " + b.getBlockId());
            }
        } catch (IOException e) {
            IOUtils.cleanup(null, ref);
            throw e;
        }

        volumeMap.add(b.getBlockPoolId(), newReplicaInfo.getReplicaInfo());
        // 创建 ReplicaHandler
        return new ReplicaHandler(newReplicaInfo, ref);
    }
}
```

```
public ReplicaHandler(
        ReplicaInPipeline replica, FsVolumeReference reference) {
    this.replica = replica;
    this.volumeReference = reference;
}
```

2.2.2.4.3.3.2 当前 DataNode 还需要连接下一个 DataNode 建立 Socket pipeline 通道

```
// 如果当前 DataNode 还需要连接下一个 DataNode 建立 Socket pipeline 通道
```

```
                if (targets.length > 0) {
                    InetSocketAddress mirrorTarget = null;
                    // Connect to backup machine
                    // 连接 Socket pipeline 的下一个 DataNode
                    mirrorNode = targets[0].getXferAddr(connectToDnViaHostname);
                    LOG.debug("Connecting to datanode {}", mirrorNode);
                    mirrorTarget = NetUtils.createSocketAddr(mirrorNode);
                    // 获取下一个 DataNode 的 Socket 连接
                    mirrorSock = datanode.newSocket();
                    try {

                        DataNodeFaultInjector.get().failMirrorConnection();

                        int timeoutValue = dnConf.socketTimeout +
                                (HdfsConstants.READ_TIMEOUT_EXTENSION * targets.length);
                        int writeTimeout = dnConf.socketWriteTimeout +
                                (HdfsConstants.WRITE_TIMEOUT_EXTENSION                    *
targets.length);
                        // 执行 Socket Connection
                        NetUtils.connect(mirrorSock, mirrorTarget, timeoutValue);
                        mirrorSock.setTcpNoDelay(dnConf.getDataTransferServerTcpNoDelay());
                        mirrorSock.setSoTimeout(timeoutValue);
                        mirrorSock.setKeepAlive(true);
                        if (dnConf.getTransferSocketSendBufferSize() > 0) {
                            mirrorSock.setSendBufferSize(
                                    dnConf.getTransferSocketSendBufferSize());
                        }

                        // 获取下一个 Socket 的输入输出流
                        OutputStream unbufMirrorOut = NetUtils.getOutputStream(mirrorSock,
                                writeTimeout);
                        InputStream unbufMirrorIn = NetUtils.getInputStream(mirrorSock);

                        DataEncryptionKeyFactory keyFactory =
                                datanode.getDataEncryptionKeyFactoryForBlock(block);
                        SecretKey secretKey = null;
                        if (dnConf.overwriteDownstreamDerivedQOP) {
                            String bpid = block.getBlockPoolId();
                            BlockKey blockKey = datanode.blockPoolTokenSecretManager
                                    .get(bpid).getCurrentKey();
                            secretKey = blockKey.getKey();
                        }
                        IOStreamPair saslStreams = datanode.saslClient.socketSend(
                                mirrorSock, unbufMirrorOut, unbufMirrorIn, keyFactory,
```

```
                        blockToken, targets[0], secretKey);
                unbufMirrorOut = saslStreams.out;
                unbufMirrorIn = saslStreams.in;
                // 封装下一个 Socket 的输入输出流
                mirrorOut = new DataOutputStream(new
BufferedOutputStream(unbufMirrorOut,
                        smallBufferSize));
                mirrorIn = new DataInputStream(unbufMirrorIn);

                String targetStorageId = null;
                if (targetStorageIds.length > 0) {
                    // Older clients may not have provided any targetStorageIds
                    targetStorageId = targetStorageIds[0];
                }
                if (targetPinnings != null && targetPinnings.length > 0) {
                    // 也是发送一个 Block (打开) 请求 (根据 Client 端相同 如果
下一个 DataNode 还需要连接再下一个 DataNode 的 Socket 请求类似即可)
                    new Sender(mirrorOut).writeBlock(originalBlock,
targetStorageTypes[0],
                            blockToken, clientname, targets, targetStorageTypes,
                            srcDataNode, stage, pipelineSize, minBytesRcvd,
maxBytesRcvd,
                            latestGenerationStamp, requestedChecksum,
cachingStrategy,
                            allowLazyPersist, targetPinnings[0], targetPinnings,
                            targetStorageId, targetStorageIds);
                } else {
                    new Sender(mirrorOut).writeBlock(originalBlock,
targetStorageTypes[0],
                            blockToken, clientname, targets, targetStorageTypes,
                            srcDataNode, stage, pipelineSize, minBytesRcvd,
maxBytesRcvd,
                            latestGenerationStamp, requestedChecksum,
cachingStrategy,
                            allowLazyPersist, false, targetPinnings,
                            targetStorageId, targetStorageIds);
                }

                mirrorOut.flush();

                DataNodeFaultInjector.get().writeBlockAfterFlush();

                // read connect ack (only for clients, not for replication req)
                if (isClient) {
```

```
                                BlockOpResponseProto connectAck =

BlockOpResponseProto.parseFrom(PBHelperClient.vintPrefixed(mirrorIn));
                                mirrorInStatus = connectAck.getStatus();
                                firstBadLink = connectAck.getFirstBadLink();
                                if (mirrorInStatus != SUCCESS) {
                                    LOG.debug("Datanode {} got response for connect" +
                                                    "ack     from  downstream  datanode  with
firstbadlink as {}",
                                                targets.length, firstBadLink);
                                }
                            }
```

2.2.2.4.3.3.3 DataNode 接收 Block (Packet) 并发送给下一个 DataNode

```
void receiveBlock(
                DataOutputStream mirrOut, // output to next datanode
                DataInputStream mirrIn,     // input from next datanode
                DataOutputStream replyOut,    // output to previous datanode
                String mirrAddr, DataTransferThrottler throttlerArg,
                DatanodeInfo[] downstreams,
                boolean isReplaceBlock) throws IOException {

            syncOnClose = datanode.getDnConf().syncOnClose;
            dirSyncOnFinalize = syncOnClose;
            boolean responderClosed = false;
            mirrorOut = mirrOut;
            mirrorAddr = mirrAddr;
            initPerfMonitoring(downstreams);
            throttler = throttlerArg;

            this.replyOut = replyOut;
            this.isReplaceBlock = isReplaceBlock;

            try {
                if (isClient && !isTransfer) {
                    // 创建 PacketResponder 线程并启动
                    responder = new Daemon(datanode.threadGroup,
                                new PacketResponder(replyOut, mirrIn, downstreams));
                    responder.start(); // start thread to processes responses
                }

                // 持续接收 Block packet 直到 Block 最后一个 packet
```

```
            while (receivePacket() >= 0) { /* Receive until the last packet */ }

            // wait for all outstanding packet responses. And then
            // indicate responder to gracefully shutdown.
            // Mark that responder has been closed for future processing
            if (responder != null) {
                ((PacketResponder) responder.getRunnable()).close();
                responderClosed = true;
            }

            // If this write is for a replication or transfer-RBW/Finalized,
            // then finalize block or convert temporary to RBW.
            // For client-writes, the block is finalized in the PacketResponder.
            if (isDatanode || isTransfer) {
                // Hold a volume reference to finalize block.
                try (ReplicaHandler handler = claimReplicaHandler()) {
                    // close the block/crc files
                    close();
                    block.setNumBytes(replicaInfo.getNumBytes());

                    if (stage == BlockConstructionStage.TRANSFER_RBW) {
                        // for TRANSFER_RBW, convert temporary to RBW
                        datanode.data.convertTemporaryToRbw(block);
                    } else {
                        // for isDatnode or TRANSFER_FINALIZED
                        // Finalize the block.
                        datanode.data.finalizeBlock(block, dirSyncOnFinalize);
                    }
                }
                datanode.metrics.incrBlocksWritten();
            }

        } catch (IOException ioe) {
            replicaInfo.releaseAllBytesReserved();
            if (datanode.isRestarting()) {
                // Do not throw if shutting down for restart. Otherwise, it will cause
                // premature termination of responder.
                LOG.info("Shutting down for restart (" + block + ").");
            } else {
                LOG.info("Exception for " + block, ioe);
                throw ioe;
            }
        } finally {
            // Clear the previous interrupt state of this thread.
```

```
            Thread.interrupted();

            // If a shutdown for restart was initiated, upstream needs to be notified.
            // There is no need to do anything special if the responder was closed
            // normally.
            if (!responderClosed) { // Data transfer was not complete.
                if (responder != null) {
                    // In case this datanode is shutting down for quick restart,
                    // send a special ack upstream.
                    if (datanode.isRestarting() && isClient && !isTransfer) {
                        try (Writer out = new OutputStreamWriter(
                                replicaInfo.createRestartMetaStream(), "UTF-8")) {
                            // write out the current time.
                            out.write(Long.toString(Time.now() + restartBudget));
                            out.flush();
                        } catch (IOException ioe) {
                            // The worst case is not recovering this RBW replica.
                            // Client will fall back to regular pipeline recovery.
                        } finally {
                            IOUtils.closeStream(streams.getDataOut());
                        }
                        try {
                            // Even if the connection is closed after the ack packet is
                            // flushed, the client can react to the connection closure
                            // first. Insert a delay to lower the chance of client
                            // missing the OOB ack.
                            Thread.sleep(1000);
                        } catch (InterruptedException ie) {
                            // It is already going down. Ignore this.
                        }
                    }
                    responder.interrupt();
                }
                IOUtils.closeStream(this);
                cleanupBlock();
            }
            if (responder != null) {
                try {
                    responder.interrupt();
                    // join() on the responder should timeout a bit earlier than the
                    // configured deadline. Otherwise, the join() on this thread will
                    // likely timeout as well.
                    long joinTimeout = datanode.getDnConf().getXceiverStopTimeout();
                    joinTimeout = joinTimeout > 1 ? joinTimeout * 8 / 10 : joinTimeout;
```

```
                        responder.join(joinTimeout);
                        if (responder.isAlive()) {
                            String msg = "Join on responder thread " + responder
                                    + " timed out";
                            LOG.warn(msg + "\n" + StringUtils.getStackTrace(responder));
                            throw new IOException(msg);
                        }
                    } catch (InterruptedException e) {
                        responder.interrupt();
                        // do not throw if shutting down for restart.
                        if (!datanode.isRestarting()) {
                            throw new IOException("Interrupted receiveBlock");
                        }
                    }
                    responder = null;
                }
            }
        }
```

2.2.2.4.3.3.3.1 创建 PacketResponder 线程并启动

```
  PacketResponder(final DataOutputStream upstreamOut,
                            final DataInputStream downstreamIn, final DatanodeInfo[]
downstreams) {
                // 下游 socket in
                this.downstreamIn = downstreamIn;

                // 上游 Socket out
                this.upstreamOut = upstreamOut;

                this.type = downstreams == null ? PacketResponderType.NON_PIPELINE
                        : downstreams.length == 0 ? PacketResponderType.LAST_IN_PIPELINE
                        : PacketResponderType.HAS_DOWNSTREAM_IN_PIPELINE;

                final StringBuilder b = new StringBuilder(getClass().getSimpleName())
                        .append(": ").append(block).append(", type=").append(type);
                if (type == PacketResponderType.HAS_DOWNSTREAM_IN_PIPELINE) {
                    b.append(", downstreams=").append(downstreams.length)
                            .append(":").append(Arrays.asList(downstreams));
                }
                this.myString = b.toString();
            }
```

```
 /**
             * Thread to process incoming acks.
             * @see java.lang.Runnable#run()
             */
            @Override
            public void run() {
                boolean lastPacketInBlock = false;
                final long startTime = ClientTraceLog.isInfoEnabled() ? System.nanoTime() : 0;
                while (isRunning() && !lastPacketInBlock) {
                    long totalAckTimeNanos = 0;
                    boolean isInterrupted = false;
                    try {
                        Packet pkt = null;
                        long expected = -2;
                        PipelineAck ack = new PipelineAck();
                        long seqno = PipelineAck.UNKOWN_SEQNO;
                        long ackRecvNanoTime = 0;
                        try {
                            if (type != PacketResponderType.LAST_IN_PIPELINE
&& !mirrorError) {
                                DataNodeFaultInjector.get().failPipeline(replicaInfo,
mirrorAddr);
                                // read an ack from downstream datanode
                                // 读取下游 socket in ack (也即下一个 DataNode 上报
ack)
                                ack.readFields(downstreamIn);
                                ackRecvNanoTime = System.nanoTime();
                                if (LOG.isDebugEnabled()) {
                                    LOG.debug(myString + " got " + ack);
                                }
                                // Process an OOB ACK.
                                Status oobStatus = ack.getOOBStatus();
                                if (oobStatus != null) {
                                    LOG.info("Relaying an out of band ack of type " +
oobStatus);
                                    sendAckUpstream(ack, PipelineAck.UNKOWN_SEQNO, 0L,
0L,

PipelineAck.combineHeader(datanode.getECN(),
                                                                Status.SUCCESS));
                                    continue;
                                }
                                seqno = ack.getSeqno();
                            }
```

```
                                if (seqno != PipelineAck.UNKOWN_SEQNO
                                        || type == PacketResponderType.LAST_IN_PIPELINE) {
                                    pkt = waitForAckHead(seqno);
                                    if (!isRunning()) {
                                        break;
                                    }
                                    expected = pkt.seqno;
                                    if (type == PacketResponderType.HAS_DOWNSTREAM_IN_PIPELINE
                                            && seqno != expected) {
                                        throw new IOException(myString + "seqno: expected=" + expected
                                                + ", received=" + seqno);
                                    }
                                    if (type == PacketResponderType.HAS_DOWNSTREAM_IN_PIPELINE) {
                                        // The total ack time includes the ack times of downstream
                                        // nodes.
                                        // The value is 0 if this responder doesn't have a downstream
                                        // DN in the pipeline.
                                        totalAckTimeNanos = ackRecvNanoTime - pkt.ackEnqueueNanoTime;
                                        // Report the elapsed time from ack send to ack receive minus
                                        // the downstream ack time.
                                        long ackTimeNanos = totalAckTimeNanos
                                                - ack.getDownstreamAckTimeNanos();
                                        if (ackTimeNanos < 0) {
                                            if (LOG.isDebugEnabled()) {
                                                LOG.debug("Calculated invalid ack time: " + ackTimeNanos
                                                        + "ns.");
                                            }
                                        } else {
                                            datanode.metrics.addPacketAckRoundTripTimeNanos(ackTimeNanos);
                                        }
                                    }
                                    lastPacketInBlock = pkt.lastPacketInBlock;
                                }
                            } catch (InterruptedException ine) {
                                isInterrupted = true;
```

```
                        } catch (IOException ioe) {
                            if (Thread.interrupted()) {
                                isInterrupted = true;
                            } else if (ioe instanceof EOFException && !packetSentInTime()) {
                                // The downstream error was caused by upstream including
this
                                // node not sending packet in time. Let the upstream
determine
                                // who is at fault.  If the immediate upstream node thinks it
                                // has sent a packet in time, this node will be reported as bad.
                                // Otherwise, the upstream node will propagate the error up
by
                                // closing the connection.
                                LOG.warn("The downstream error might be due to congestion
in " +
                                                "upstream including this node. Propagating
the error: ",
                                        ioe);
                                throw ioe;
                            } else {
                                // continue to run even if can not read from mirror
                                // notify client of the error
                                // and wait for the client to shut down the pipeline
                                mirrorError = true;
                                LOG.info(myString, ioe);
                            }
                        }

                        if (Thread.interrupted() || isInterrupted) {
                            /*
                             * The receiver thread cancelled this thread. We could also check
                             * any other status updates from the receiver thread (e.g. if it is
                             * ok to write to replyOut). It is prudent to not send any more
                             * status back to the client because this datanode has a problem.
                             * The upstream datanode will detect that this datanode is bad,
and
                             * rightly so.
                             *
                             * The receiver thread can also interrupt this thread for sending
                             * an out-of-band response upstream.
                             */
                            LOG.info(myString + ": Thread is interrupted.");
                            running = false;
                            continue;
```

```
                    }

                    if (lastPacketInBlock) {
                        // Finalize the block and close the block file
                        finalizeBlock(startTime);
                    }

                    Status myStatus = pkt != null ? pkt.ackStatus : Status.SUCCESS;
                    // 发送 ack 给上游
                    sendAckUpstream(ack, expected, totalAckTimeNanos,
                            (pkt != null ? pkt.offsetInBlock : 0),
                            PipelineAck.combineHeader(datanode.getECN(), myStatus));
                    if (pkt != null) {
                        // remove the packet from the ack queue
                        removeAckHead();
                    }
                } catch (IOException e) {
                    LOG.warn("IOException in BlockReceiver.run(): ", e);
                    if (running) {
                        // Volume error check moved to FileIoProvider
                        LOG.info(myString, e);
                        running = false;
                        if (!Thread.interrupted()) { // failure not caused by interruption
                            receiverThread.interrupt();
                        }
                    }
                } catch (Throwable e) {
                    if (running) {
                        LOG.info(myString, e);
                        running = false;
                        receiverThread.interrupt();
                    }
                }
            }
            LOG.info(myString + " terminating");
        }
```

2.2.2.4.3.3.3.2 持续接收 Block packet 直到 Block 最后一个 packet (ack 添加到队列以便返回给上游)

```
/**
     * Receives and processes a packet. It can contain many chunks.
     * returns the number of data bytes that the packet has.
```

```
     */
    private int receivePacket() throws IOException {
        // read the next packet
        //  读取下一个  Packet
        packetReceiver.receiveNextPacket(in);

        PacketHeader header = packetReceiver.getHeader();
        if (LOG.isDebugEnabled()) {
            LOG.debug("Receiving one packet for block " + block +
                    ": " + header);
        }

        // Sanity check the header
        if (header.getOffsetInBlock() > replicaInfo.getNumBytes()) {
            throw new IOException("Received an out-of-sequence packet for " + block +
                    "from " + inAddr + " at offset " + header.getOffsetInBlock() +
                    ". Expecting packet starting at " + replicaInfo.getNumBytes());
        }
        if (header.getDataLen() < 0) {
            throw new IOException("Got wrong length during writeBlock(" + block +
                    ") from " + inAddr + " at offset " +
                    header.getOffsetInBlock() + ": " +
                    header.getDataLen());
        }

        long offsetInBlock = header.getOffsetInBlock();
        long seqno = header.getSeqno();
        boolean lastPacketInBlock = header.isLastPacketInBlock();
        final int len = header.getDataLen();
        boolean syncBlock = header.getSyncBlock();

        // avoid double sync'ing on close
        if (syncBlock && lastPacketInBlock) {
            this.syncOnClose = false;
            // sync directory for finalize irrespective of syncOnClose config since
            // sync is requested.
            this.dirSyncOnFinalize = true;
        }

        // update received bytes
        final long firstByteInBlock = offsetInBlock;
        offsetInBlock += len;
        if (replicaInfo.getNumBytes() < offsetInBlock) {
            replicaInfo.setNumBytes(offsetInBlock);
```

```
        }

        // put in queue for pending acks, unless sync was requested
        if (responder != null && !syncBlock && !shouldVerifyChecksum()) {
            // 将接收的 packet ack 添加到 responder 队列中
            ((PacketResponder) responder.getRunnable()).enqueue(seqno,
                    lastPacketInBlock, offsetInBlock, Status.SUCCESS);
        }

        // Drop heartbeat for testing.
        if (seqno < 0 && len == 0 &&
                DataNodeFaultInjector.get().dropHeartbeatPacket()) {
            return 0;
        }

        //First write the packet to the mirror:
        if (mirrorOut != null && !mirrorError) {
            try {
                long begin = Time.monotonicNow();
                // For testing. Normally no-op.
                DataNodeFaultInjector.get().stopSendingPacketDownstream(mirrorAddr);

                // 将接收到的 Packet 发送给下游
                packetReceiver.mirrorPacketTo(mirrorOut);
                // 刷写 socket
                mirrorOut.flush();

                long now = Time.monotonicNow();
                setLastSentTime(now);
                long duration = now - begin;
                DataNodeFaultInjector.get().logDelaySendingPacketDownstream(
                        mirrorAddr,
                        duration);
                trackSendPacketToLastNodeInPipeline(duration);
                if (duration > datanodeSlowLogThresholdMs && LOG.isWarnEnabled()) {
                    LOG.warn("Slow BlockReceiver write packet to mirror took " + duration
                            + "ms (threshold=" + datanodeSlowLogThresholdMs + "ms), "
                            + "downstream DNs=" + Arrays.toString(downstreamDNs)
                            + ", blockId=" + replicaInfo.getBlockId());
                }
            } catch (IOException e) {
                handleMirrorOutError(e);
            }
        }
```

```
            ByteBuffer dataBuf = packetReceiver.getDataSlice();
            ByteBuffer checksumBuf = packetReceiver.getChecksumSlice();

            if (lastPacketInBlock || len == 0) {
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Receiving an empty packet or the end of the block " + block);
                }
                // sync block if requested
                if (syncBlock) {
                    flushOrSync(true);
                }
            } else {
                final int checksumLen = diskChecksum.getChecksumSize(len);
                final int checksumReceivedLen = checksumBuf.capacity();

                if (checksumReceivedLen > 0 && checksumReceivedLen != checksumLen) {
                    throw new IOException("Invalid checksum length: received length is "
                            + checksumReceivedLen + " but expected length is " +
checksumLen);
                }

                if (checksumReceivedLen > 0 && shouldVerifyChecksum()) {
                    try {
                        verifyChunks(dataBuf, checksumBuf);
                    } catch (IOException ioe) {
                        // checksum error detected locally. there is no reason to continue.
                        if (responder != null) {
                            try {
                                ((PacketResponder) responder.getRunnable()).enqueue(seqno,
                                        lastPacketInBlock, offsetInBlock,
                                        Status.ERROR_CHECKSUM);
                                // Wait until the responder sends back the response
                                // and interrupt this thread.
                                Thread.sleep(3000);
                            } catch (InterruptedException e) {
                            }
                        }
                        throw new IOException("Terminating due to a checksum error." + ioe);
                    }

                    if (needsChecksumTranslation) {
                        // overwrite the checksums in the packet buffer with the
                        // appropriate polynomial for the disk storage.
```

```
                        translateChunks(dataBuf, checksumBuf);
                    }
                }

                if (checksumReceivedLen == 0 && !streams.isTransientStorage()) {
                    // checksum is missing, need to calculate it
                    checksumBuf = ByteBuffer.allocate(checksumLen);
                    diskChecksum.calculateChunkedSums(dataBuf, checksumBuf);
                }

                // by this point, the data in the buffer uses the disk checksum

                final boolean shouldNotWriteChecksum = checksumReceivedLen == 0
                        && streams.isTransientStorage();
                try {
                    long onDiskLen = replicaInfo.getBytesOnDisk();
                    if (onDiskLen < offsetInBlock) {
                        // Normally the beginning of an incoming packet is aligned with the
                        // existing data on disk. If the beginning packet data offset is not
                        // checksum chunk aligned, the end of packet will not go beyond the
                        // next chunk boundary.
                        // When a failure-recovery is involved, the client state and the
                        // the datanode state may not exactly agree. I.e. the client may
                        // resend part of data that is already on disk. Correct number of
                        // bytes should be skipped when writing the data and checksum
                        // buffers out to disk.
                        long partialChunkSizeOnDisk = onDiskLen % bytesPerChecksum;
                        long lastChunkBoundary = onDiskLen - partialChunkSizeOnDisk;
                        boolean alignedOnDisk = partialChunkSizeOnDisk == 0;
                        boolean alignedInPacket = firstByteInBlock % bytesPerChecksum == 0;

                        // If the end of the on-disk data is not chunk-aligned, the last
                        // checksum needs to be overwritten.
                        boolean overwriteLastCrc = !alignedOnDisk
&& !shouldNotWriteChecksum;
                        // If the starting offset of the packat data is at the last chunk
                        // boundary of the data on disk, the partial checksum recalculation
                        // can be skipped and the checksum supplied by the client can be used
                        // instead. This reduces disk reads and cpu load.
                        boolean doCrcRecalc = overwriteLastCrc &&
                                (lastChunkBoundary != firstByteInBlock);

                        // If this is a partial chunk, then verify that this is the only
                        // chunk in the packet. If the starting offset is not chunk
```

```
                    // aligned, the packet should terminate at or before the next
                    // chunk boundary.
                    if (!alignedInPacket && len > bytesPerChecksum) {
                        throw new IOException("Unexpected packet data length for "
                                + block + " from " + inAddr + ": a partial chunk must be "
                                + " sent in an individual packet (data length = " + len
                                + " > bytesPerChecksum = " + bytesPerChecksum + ")");
                    }

                    // If the last portion of the block file is not a full chunk,
                    // then read in pre-existing partial data chunk and recalculate
                    // the checksum so that the checksum calculation can continue
                    // from the right state. If the client provided the checksum for
                    // the whole chunk, this is not necessary.
                    Checksum partialCrc = null;
                    if (doCrcRecalc) {
                        if (LOG.isDebugEnabled()) {
                            LOG.debug("receivePacket for " + block
                                    + ": previous write did not end at the chunk
boundary."
                                    + " onDiskLen=" + onDiskLen);
                        }
                        long offsetInChecksum = BlockMetadataHeader.getHeaderSize() +
                                onDiskLen / bytesPerChecksum * checksumSize;
                        partialCrc = computePartialChunkCrc(onDiskLen,
offsetInChecksum);
                    }

                    // The data buffer position where write will begin. If the packet
                    // data and on-disk data have no overlap, this will not be at the
                    // beginning of the buffer.
                    int startByteToDisk = (int) (onDiskLen - firstByteInBlock)
                            + dataBuf.arrayOffset() + dataBuf.position();

                    // Actual number of data bytes to write.
                    int numBytesToDisk = (int) (offsetInBlock - onDiskLen);

                    // Write data to disk.
                    // 将接收到 Packet 写入磁盘
                    long begin = Time.monotonicNow();
                    streams.writeDataToDisk(dataBuf.array(),
                            startByteToDisk, numBytesToDisk);
                    long duration = Time.monotonicNow() - begin;
```

```
                        if (duration > datanodeSlowLogThresholdMs && LOG.isWarnEnabled()) {
                            LOG.warn("Slow BlockReceiver write data to disk cost:" + duration
                                    + "ms (threshold=" + datanodeSlowLogThresholdMs +
"ms), "
                                    + "volume=" + getVolumeBaseUri()
                                    + ", blockId=" + replicaInfo.getBlockId());
                        }

                        if (duration > maxWriteToDiskMs) {
                            maxWriteToDiskMs = duration;
                        }

                        final byte[] lastCrc;
                        if (shouldNotWriteChecksum) {
                            lastCrc = null;
                        } else {
                            int skip = 0;
                            byte[] crcBytes = null;

                            // First, prepare to overwrite the partial crc at the end.
                            if (overwriteLastCrc) { // not chunk-aligned on disk
                                // prepare to overwrite last checksum
                                adjustCrcFilePosition();
                            }

                            // The CRC was recalculated for the last partial chunk. Update the
                            // CRC by reading the rest of the chunk, then write it out.
                            if (doCrcRecalc) {
                                // Calculate new crc for this chunk.
                                int bytesToReadForRecalc =
                                        (int) (bytesPerChecksum - partialChunkSizeOnDisk);
                                if (numBytesToDisk < bytesToReadForRecalc) {
                                    bytesToReadForRecalc = numBytesToDisk;
                                }

                                partialCrc.update(dataBuf.array(), startByteToDisk,
                                        bytesToReadForRecalc);
                                byte[] buf = FSOutputSummer.convertToByteStream(partialCrc,
                                        checksumSize);
                                crcBytes = copyLastChunkChecksum(buf, checksumSize,
buf.length);
                                checksumOut.write(buf);
                                if (LOG.isDebugEnabled()) {
                                    LOG.debug("Writing out partial crc for data len " + len +
```

```
                                        ", skip=" + skip);
                    }
                    skip++; //  For the partial chunk that was just read.
                }

                // Determine how many checksums need to be skipped up to the last
                // boundary. The checksum after the boundary was already counted
                // above. Only count the number of checksums skipped up to the
                // boundary here.
                long skippedDataBytes = lastChunkBoundary - firstByteInBlock;

                if (skippedDataBytes > 0) {
                    skip += (int) (skippedDataBytes / bytesPerChecksum) +
                            ((skippedDataBytes % bytesPerChecksum == 0) ? 0 : 1);
                }
                skip *= checksumSize; // Convert to number of bytes

                // write the rest of checksum
                final int offset = checksumBuf.arrayOffset() +
                        checksumBuf.position() + skip;
                final int end = offset + checksumLen - skip;
                // If offset >= end, there is no more checksum to write.
                // I.e. a partial chunk checksum rewrite happened and there is no
                // more to write after that.
                if (offset >= end && doCrcRecalc) {
                    lastCrc = crcBytes;
                } else {
                    final int remainingBytes = checksumLen - skip;
                    lastCrc = copyLastChunkChecksum(checksumBuf.array(),
                            checksumSize, end);
                    checksumOut.write(checksumBuf.array(), offset, remainingBytes);
                }
            }

            /// flush entire packet, sync if requested
            flushOrSync(syncBlock);

            replicaInfo.setLastChecksumAndDataLen(offsetInBlock, lastCrc);

            datanode.metrics.incrBytesWritten(len);
```

```
                        datanode.metrics.incrTotalWriteTime(duration);

                        manageWriterOsCache(offsetInBlock);
                    }
                } catch (IOException iex) {
                    // Volume error check moved to FileIoProvider
                    throw iex;
                }
            }

            // if sync was requested, put in queue for pending acks here
            // (after the fsync finished)
            if (responder != null && (syncBlock || shouldVerifyChecksum())) {
                ((PacketResponder) responder.getRunnable()).enqueue(seqno,
                        lastPacketInBlock, offsetInBlock, Status.SUCCESS);
            }

            /*
             * Send in-progress responses for the replaceBlock() calls back to caller to
             * avoid timeouts due to balancer throttling. HDFS-6247
             */
            if (isReplaceBlock
                    && (Time.monotonicNow() - lastResponseTime > responseInterval)) {
                BlockOpResponseProto.Builder response = BlockOpResponseProto.newBuilder()
                        .setStatus(Status.IN_PROGRESS);
                response.build().writeDelimitedTo(replyOut);
                replyOut.flush();

                lastResponseTime = Time.monotonicNow();
            }

            if (throttler != null) { // throttle I/O
                throttler.throttle(len);
            }

            return lastPacketInBlock ? -1 : len;
        }
```