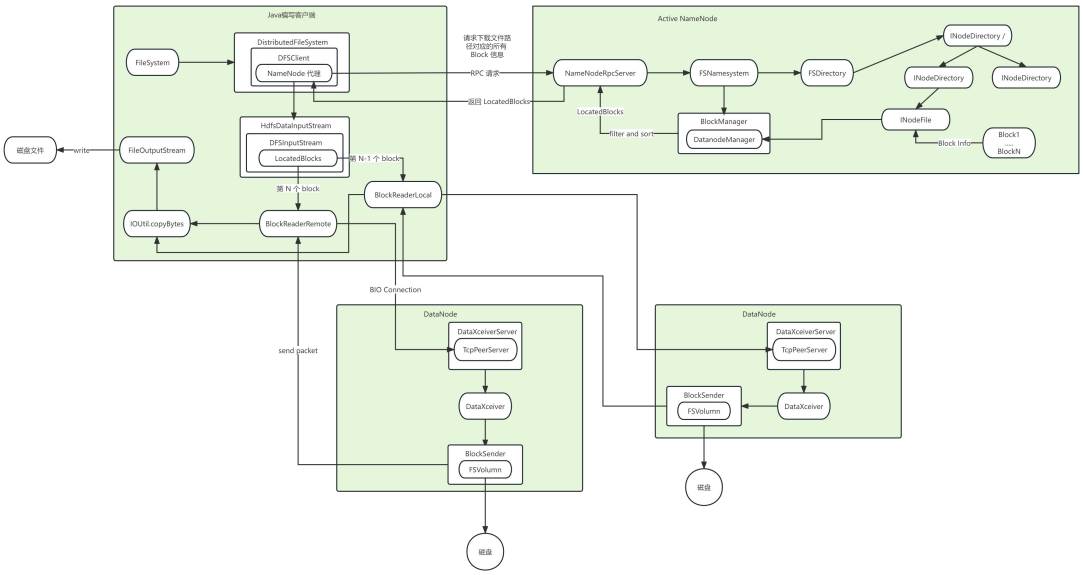
hadoop-hdfs 文件下载源码分析

一 案例

```
public class DownLoadMain {

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

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

		Configuration conf = new Configuration();

		FileSystem fileSystem = FileSystem.get(conf);

		FSDataInputStream fis = fileSystem.open(new Path("/mkdir/LICENSE.txt"));

		FileOutputStream fos = new FileOutputStream("./LICENSE.txt");
		IOUtils.copyBytes(fis, fos, 1024, true);

		fos.close();
		fis.close();
		fileSystem.close();

	}

}
```

二 文件下载源码分析



2.1 创建 HDFSDataInputStream (继承 FSDataInputStream)

```
FSDataInputStream fis = fileSystem.open(new Path("/mkdir/LICENSE.txt"));
```

```
/**
     * 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())
        // 请求 HDFS 下载文件返回 HdfsDataInputStream (最终调用 DistributedFileSystem.open())
        return open(f, getConf().getInt(IO_FILE_BUFFER_SIZE_KEY,
                IO_FILE_BUFFER_SIZE_DEFAULT));
    }
```

```
@Override
    public FSDataInputStream open(
            // 请求 HDFS 下载文件路径
            Path f,
            // 默认 4096
            final int bufferSize)
```

```
            throws IOException {

        statistics.incrementReadOps(1);
        storageStatistics.incrementOpCounter(OpType.OPEN);

        // 还是返回请求 HDFS 下载文件路径
        Path absF = fixRelativePart(f);
        return new FileSystemLinkResolver<FSDataInputStream>() {
            @Override
            public FSDataInputStream doCall(final Path p) throws IOException {
                // 创建并返回 DFSInputStream
                final DFSInputStream dfsis =
                        dfs.open(
                                // 请求下载 HDFS 文件路径 比如 /opt/app/LICENSE.txt
                                getPathName(p),
                                bufferSize, verifyChecksum);
                // 包装 DFSInputStream 成 HdfsDataInputStream
                return dfs.createWrappedInputStream(dfsis);
            }

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

```
/**
     * Create an input stream that obtains a nodelist from the
     * namenode, and then reads from all the right places.  Creates
     * inner subclass of InputStream that does the right out-of-band
     * work.
     */
    public DFSInputStream open(String src, int buffersize, boolean verifyChecksum)
            throws IOException {
        checkOpen();
        //    Get block info from namenode
        try (TraceScope ignored = newPathTraceScope("newDFSInputStream", src)) {
            // 获取请求下载 HDFS 文件 Block 信息 (发送 RPC 请求)
            LocatedBlocks locatedBlocks = getLocatedBlocks(src, 0);
            // 往下追 创建并返回 DFSInputStream
            return openInternal(locatedBlocks, src, verifyChecksum);
        }
```

```
    }
```

2.1.1 获取请求下载 HDFS 文件 Block 信息 (发送 RPC 请求)

```
    public LocatedBlocks getLocatedBlocks(String src, long start)
            throws IOException {
        // 往下追
        return getLocatedBlocks(
                // /opt/app/LICENSE.txt
                src,
                // 0
                start,
                // 10 * 128 MB
                dfsClientConf.getPrefetchSize());
    }
```

```
/*
     * This is just a wrapper around callGetBlockLocations, but non-static so that
     * we can stub it out for tests.
     */
    @VisibleForTesting
    public LocatedBlocks getLocatedBlocks(String src, long start, long length)
            throws IOException {
        try (TraceScope ignored = newPathTraceScope("getBlockLocations", src)) {
            // 往下追
            return callGetBlockLocations(namenode, src, start, length);
        }
    }
```

```
    /**
     * @see ClientProtocol#getBlockLocations(String, long, long)
     */
    static LocatedBlocks callGetBlockLocations(ClientProtocol namenode,
                                               String src, long start, long length)
            throws IOException {
        try {
            // 调用 NameNodeRpcServer.getBlockLocations()
            return namenode.getBlockLocations(src, start, length);
        } catch (RemoteException re) {
            throw re.unwrapRemoteException(AccessControlException.class,
                    FileNotFoundException.class,
```

```
                    UnresolvedPathException.class);
        }
    }
```

2.1.1.1 调用 NameNodeRpcServer.getBlockLocations()

```
@Override // ClientProtocol
    public LocatedBlocks getBlockLocations(String src,
                                           long offset,
                                           long length)
            throws IOException {
        checkNNStartup();
        metrics.incrGetBlockLocations();
        // 往下追
        return namesystem.getBlockLocations(getClientMachine(), src, offset, length);
    }
```

```
/**
     * Get block locations within the specified range.
     *
     * @see ClientProtocol#getBlockLocations(String, long, long)
     */
    LocatedBlocks getBlockLocations(String clientMachine, String srcArg,
                                    long offset, long length) throws IOException {
        final String operationName = "open";
        checkOperation(OperationCategory.READ);
        GetBlockLocationsResult res = null;
        final FSPermissionChecker pc = getPermissionChecker();
        readLock();
        try {
            checkOperation(OperationCategory.READ);
            // 获取请求下载文件路径的所有 Block 信息
            res = FSDirStatAndListingOp.getBlockLocations(
                    dir, pc, srcArg, offset, length, true);

            if (isInSafeMode()) {
                for (LocatedBlock b : res.blocks.getLocatedBlocks()) {
                    // if safemode & no block locations yet then throw safemodeException
                    if ((b.getLocations() == null) || (b.getLocations().length == 0)) {
                        SafeModeException se = newSafemodeException(
                                "Zero blocklocations for " + srcArg);
                        if (haEnabled && haContext != null &&
```

```
                                        (haContext.getState().getServiceState() == ACTIVE ||
                                                haContext.getState().getServiceState()          ==
OBSERVER)) {
                                    throw new RetriableException(se);
                                } else {
                                    throw se;
                                }
                            }
                        }
                    } else if (haEnabled && haContext != null &&
                            haContext.getState().getServiceState() == OBSERVER) {
                        for (LocatedBlock b : res.blocks.getLocatedBlocks()) {
                            if (b.getLocations() == null || b.getLocations().length == 0) {
                                throw  new  ObserverRetryOnActiveException("Zero  blocklocations
for "
                                        + srcArg);
                            }
                        }
                    }
                } catch (AccessControlException e) {
                    logAuditEvent(false, operationName, srcArg);
                    throw e;
                } finally {
                    readUnlock(operationName);
                }

                logAuditEvent(true, operationName, srcArg);

                if (!isInSafeMode() && res.updateAccessTime()) {
                    String src = srcArg;
                    checkOperation(OperationCategory.WRITE);
                    writeLock();
                    final long now = now();
                    try {
                        checkOperation(OperationCategory.WRITE);
                        /**
                          * Resolve the path again and update the atime only when the file
                          * exists.
                          *
                          * XXX: Races can still occur even after resolving the path again.
                          * For example:
                          *
                          * <ul>
                          *     <li>Get the block location for "/a/b"</li>
```

```
                    *      <li>Rename "/a/b" to "/c/b"</li>
                    *      <li>The second resolution still points to "/a/b", which is
                    *      wrong.</li>
                    * </ul>
                    *
                    * The behavior is incorrect but consistent with the one before
                    * HDFS-7463. A better fix is to change the edit log of SetTime to
                    * use inode id instead of a path.
                    */
                   final INodesInPath iip = dir.resolvePath(pc, srcArg, DirOp.READ);
                   src = iip.getPath();

                   INode inode = iip.getLastINode();
                   boolean updateAccessTime = inode != null &&
                           now > inode.getAccessTime() + dir.getAccessTimePrecision();
                   if (!isInSafeMode() && updateAccessTime) {
                       boolean changed = FSDirAttrOp.setTimes(dir, iip, -1, now, false);
                       if (changed) {
                           getEditLog().logTimes(src, -1, now);
                       }
                   }
               } catch (Throwable e) {
                   LOG.warn("Failed to update the access time of " + src, e);
               } finally {
                   writeUnlock(operationName);
               }
           }

           LocatedBlocks blocks = res.blocks;
           // 排序 (根据客户端机器进行排序 比如剔除下线的 DataNode、本地短路下载文件就不会走网络)
           sortLocatedBlocks(clientMachine, blocks);
           return blocks;
       }
```

2.1.1.1.1 获取请求下载文件路径的所有 Block 信息

```
/**
     * Get block locations within the specified range.
     *
     * @throws IOException
     * @see ClientProtocol#getBlockLocations(String, long, long)
     */
    static GetBlockLocationsResult getBlockLocations(
```

```
        FSDirectory fsd, FSPermissionChecker pc, String src, long offset,
        long length, boolean needBlockToken) throws IOException {
    Preconditions.checkArgument(offset >= 0,
            "Negative offset is not supported. File: " + src);
    Preconditions.checkArgument(length >= 0,
            "Negative length is not supported. File: " + src);
    BlockManager bm = fsd.getBlockManager();
    fsd.readLock();
    try {
        // 解析请求下载文件路径
        final INodesInPath iip = fsd.resolvePath(pc, src, DirOp.READ);
        src = iip.getPath();
        // 根据请求下载文件路径获取对应的 INodeFile
        final INodeFile inode = INodeFile.valueOf(iip.getLastINode(), src);

        if (fsd.isPermissionEnabled()) {
            fsd.checkPathAccess(pc, iip, FsAction.READ);
            fsd.checkUnreadableBySuperuser(pc, iip);
        }

        final long fileSize = iip.isSnapshot()
                ? inode.computeFileSize(iip.getPathSnapshotId())
                : inode.computeFileSizeNotIncludingLastUcBlock();

        boolean isUc = inode.isUnderConstruction();
        if (iip.isSnapshot()) {
            // if src indicates a snapshot file, we need to make sure the returned
            // blocks do not exceed the size of the snapshot file.
            length = Math.min(length, fileSize - offset);
            isUc = false;
        }

        final FileEncryptionInfo feInfo =
                FSDirEncryptionZoneOp.getFileEncryptionInfo(fsd, iip);
        final ErasureCodingPolicy ecPolicy = FSDirErasureCodingOp.
                unprotectedGetErasureCodingPolicy(fsd.getFSNamesystem(), iip);

        // 封装请求下载文件对应的所有 Block 信息 (LocatedBlock)
        final LocatedBlocks blocks = bm.createLocatedBlocks(
                // 获取请求下载文件路径对应的 Block 信息
                inode.getBlocks(iip.getPathSnapshotId()),
                fileSize, isUc, offset,
                length, needBlockToken, iip.isSnapshot(), feInfo, ecPolicy);
```

```
                final long now = now();
                boolean updateAccessTime = fsd.isAccessTimeSupported()
                        && !iip.isSnapshot()
                        && now > inode.getAccessTime() + fsd.getAccessTimePrecision();
                // 封装
                return new GetBlockLocationsResult(updateAccessTime, blocks);
            } finally {
                fsd.readUnlock();
            }
        }
```

```
/**
     * Create a LocatedBlocks.
     */
    public LocatedBlocks createLocatedBlocks(final BlockInfo[] blocks,
                                             final long fileSizeExcludeBlocksUnderConstruction,
                                             final boolean isFileUnderConstruction, final long offset,
                                             final long length, final boolean needBlockToken,
                                             final boolean inSnapshot, FileEncryptionInfo feInfo,
                                             ErasureCodingPolicy ecPolicy)
                throws IOException {
            assert namesystem.hasReadLock();
            if (blocks == null) {
                return null;
            } else if (blocks.length == 0) {
                return new LocatedBlocks(0, isFileUnderConstruction,
                        Collections.<LocatedBlock>emptyList(), null, false, feInfo, ecPolicy);
            } else {
                if (LOG.isDebugEnabled()) {
                    LOG.debug("blocks = {}", java.util.Arrays.asList(blocks));
                }
                final AccessMode mode = needBlockToken ? BlockTokenIdentifier.AccessMode.READ : null;

                LocatedBlockBuilder locatedBlocks = providedStorageMap
                        .newLocatedBlocks(Integer.MAX_VALUE)
                        .fileLength(fileSizeExcludeBlocksUnderConstruction)
                        .lastUC(isFileUnderConstruction)
                        .encryption(feInfo)
                        .erasureCoding(ecPolicy);
```

```
            // 请求下载文件路径的所有 Block DataNode 地址信息添加到 locatedBlocks
            createLocatedBlockList(locatedBlocks, blocks, offset, length, mode);
            if (!inSnapshot) {
                final BlockInfo last = blocks[blocks.length - 1];
                final long lastPos = last.isComplete() ?
                        fileSizeExcludeBlocksUnderConstruction - last.getNumBytes()
                        : fileSizeExcludeBlocksUnderConstruction;

                // 添加最后一个 Block DataNode 信息
                locatedBlocks
                        .lastBlock(createLocatedBlock(locatedBlocks, last, lastPos, mode))
                        .lastComplete(last.isComplete());
            } else {
                locatedBlocks
                        .lastBlock(createLocatedBlock(locatedBlocks, blocks,
                                fileSizeExcludeBlocksUnderConstruction, mode))
                        .lastComplete(true);
            }
            // 构建
            LocatedBlocks locations = locatedBlocks.build();
            // Set caching information for the located blocks.
            CacheManager cm = namesystem.getCacheManager();
            if (cm != null) {
                cm.setCachedLocations(locations);
            }
            // 返回
            return locations;
        }
    }
```

```
private void createLocatedBlockList(
            LocatedBlockBuilder locatedBlocks,
            final BlockInfo[] blocks,
            // 0
            final long offset,
            // 10 * 128 MB
            final long length,
            final AccessMode mode) throws IOException {
        int curBlk;
        long curPos = 0, blkSize = 0;
        // Block 个数
        int nrBlocks = (blocks[0].getNumBytes() == 0) ? 0 : blocks.length;
```

```
        for (curBlk = 0; curBlk < nrBlocks; curBlk++) {
            blkSize = blocks[curBlk].getNumBytes();
            assert blkSize > 0 : "Block of size 0";
            if (curPos + blkSize > offset) {
                break;
            }
            curPos += blkSize;
        }

        if (nrBlocks > 0 && curBlk == nrBlocks)      // offset >= end of file
            return;

        // 0 + 128 MB
        long endOff = offset + length;
        do {
            // 遍历所有的 Block 信息
            // 添加一个 Block 对应的地址 LocatedBlock
            locatedBlocks.addBlock(
                    // 返回请求下载文件路径的所有 Block DataNode 地址信息
                    createLocatedBlock(locatedBlocks, blocks[curBlk], curPos, mode));
            curPos += blocks[curBlk].getNumBytes();
            curBlk++;
        } while (curPos < endOff
                && curBlk < blocks.length
                && !locatedBlocks.isBlockMax());
    }
```

```
    private LocatedBlock createLocatedBlock(LocatedBlockBuilder locatedBlocks,
                                            final BlockInfo blk, final long pos, final
AccessMode mode)
            throws IOException {
        // 往下追 创建 LocatedBlock
        final LocatedBlock lb = createLocatedBlock(locatedBlocks, blk, pos);
        if (mode != null) {
            setBlockToken(lb, mode);
        }
        return lb;
    }
```

```
/**
     * @return a LocatedBlock for the given block
     */
    private LocatedBlock createLocatedBlock(LocatedBlockBuilder locatedBlocks,
                                            final BlockInfo blk, final long pos) throws
```

```
IOException {
            if (!blk.isComplete()) {
                  final BlockUnderConstructionFeature uc = blk.getUnderConstructionFeature();
                  if (blk.isStriped()) {
                        final DatanodeStorageInfo[] storages = uc.getExpectedStorageLocations();
                        final ExtendedBlock eb = new ExtendedBlock(getBlockPoolId(),
                                    blk);
                        return newLocatedStripedBlock(eb, storages, uc.getBlockIndices(), pos,
                                    false);
                  } else {
                        final DatanodeStorageInfo[] storages = uc.getExpectedStorageLocations();
                        final ExtendedBlock eb = new ExtendedBlock(getBlockPoolId(),
                                    blk);
                        return null == locatedBlocks
                                    ? newLocatedBlock(eb, storages, pos, false)
                                    : locatedBlocks.newLocatedBlock(eb, storages, pos, false);
                  }
            }

            // get block locations
            // 获取 Block 副本 DataNode 地址 默认 3
            NumberReplicas numReplicas = countNodes(blk);
            final int numCorruptNodes = numReplicas.corruptReplicas();
            final int numCorruptReplicas = corruptReplicas.numCorruptReplicas(blk);
            if (numCorruptNodes != numCorruptReplicas) {
                  LOG.warn("Inconsistent number of corrupt replicas for {}"
                              + " blockMap has {} but corrupt replicas map has {}",
                        blk, numCorruptNodes, numCorruptReplicas);
            }

            final int numNodes = blocksMap.numNodes(blk);
            final boolean isCorrupt;
            if (blk.isStriped()) {
                  BlockInfoStriped sblk = (BlockInfoStriped) blk;
                  isCorrupt = numCorruptReplicas != 0 &&
                        numReplicas.liveReplicas() < sblk.getRealDataBlockNum();
            } else {
                  // true
                  isCorrupt = numCorruptReplicas != 0 && numCorruptReplicas == numNodes;
            }
            // 3
            int numMachines = isCorrupt ? numNodes : numNodes - numCorruptReplicas;
            numMachines -= numReplicas.maintenanceNotForReadReplicas();
            // 创建 对应 Block 副本 DataNode 存储信息数组
```

```
        DatanodeStorageInfo[] machines = new DatanodeStorageInfo[numMachines];
        // null
        final byte[] blockIndices = blk.isStriped() ? new byte[numMachines] : null;
        int j = 0, i = 0;
        if (numMachines > 0) {
            final boolean noCorrupt = (numCorruptReplicas == 0);
            for (DatanodeStorageInfo storage : blocksMap.getStorages(blk)) {
                if (storage.getState() != State.FAILED) {
                    final DatanodeDescriptor d = storage.getDatanodeDescriptor();
                    // Don't pick IN_MAINTENANCE or dead ENTERING_MAINTENANCE
states.
                    if (d.isInMaintenance()
                            || (d.isEnteringMaintenance() && !d.isAlive())) {
                        continue;
                    }

                    //   Block 没有副本损坏
                    if (noCorrupt) {
                        // 赋值
                        machines[j++] = storage;
                        i = setBlockIndices(blk, blockIndices, i, storage);
                    } else {
                        final boolean replicaCorrupt = isReplicaCorrupt(blk, d);
                        if (isCorrupt || !replicaCorrupt) {
                            machines[j++] = storage;
                            i = setBlockIndices(blk, blockIndices, i, storage);
                        }
                    }
                }
            }
        }

        if (j < machines.length) {
            machines = Arrays.copyOf(machines, j);
        }

        assert j == machines.length :
                "isCorrupt: " + isCorrupt +
                        " numMachines: " + numMachines +
                        " numNodes: " + numNodes +
                        " numCorrupt: " + numCorruptNodes +
                        " numCorruptRepls: " + numCorruptReplicas;
        final ExtendedBlock eb = new ExtendedBlock(getBlockPoolId(), blk);
        return blockIndices == null
```

```
                    ? null == locatedBlocks ? newLocatedBlock(eb, machines, pos, isCorrupt)
                    // 创建 LocatedBlock
                    : locatedBlocks.newLocatedBlock(eb, machines, pos, isCorrupt)
                    : newLocatedStripedBlock(eb, machines, blockIndices, pos, isCorrupt);
    }
```

2.1.1.1.2 排序 LocatedBlocks (LocatedBlocks 保存当前下载文件路径的所有 LocatedBlock 信息)

```
private void sortLocatedBlocks(String clientMachine, LocatedBlocks blocks) {
        if (blocks != null) {
            List<LocatedBlock> blkList = blocks.getLocatedBlocks();
            if (blkList == null || blkList.size() == 0) {
                // simply return, block list is empty
                return;
            }

            // 根据客户端机器进行排序
            blockManager.getDatanodeManager().sortLocatedBlocks(clientMachine,
                    blkList);

            // lastBlock is not part of getLocatedBlocks(), might need to sort it too
            LocatedBlock lastBlock = blocks.getLastLocatedBlock();
            if (lastBlock != null) {
                ArrayList<LocatedBlock> lastBlockList = Lists.newArrayList(lastBlock);
                blockManager.getDatanodeManager().sortLocatedBlocks(clientMachine,
                        lastBlockList);
            }
        }
    }
```

```
/**
     * Sort the non-striped located blocks by the distance to the target host.
     * <p>
     * For striped blocks, it will only move decommissioned/stale nodes to the
     * bottom. For example, assume we have storage list:
     * d0, d1, d2, d3, d4, d5, d6, d7, d8, d9
     * mapping to block indices:
     * 0, 1, 2, 3, 4, 5, 6, 7, 8, 2
     * <p>
     * Here the internal block b2 is duplicated, locating in d2 and d9. If d2 is
     * a decommissioning node then should switch d2 and d9 in the storage list.
     * After sorting locations, will update corresponding block indices
```

```
     * and block tokens.
     */
    public void sortLocatedBlocks(final String targetHost,
                                  final List<LocatedBlock> locatedBlocks) {
        // 默认  avoidStaleDataNodesForRead = false
        Comparator<DatanodeInfo> comparator = avoidStaleDataNodesForRead ?
                new DFSUtil.ServiceAndStaleComparator(staleInterval) :
                // 返回
                new DFSUtil.ServiceComparator();
        // sort located block
        for (LocatedBlock lb : locatedBlocks) {
            if (lb.isStriped()) {
                sortLocatedStripedBlock(lb, comparator);
            } else {
                // 往下追
                sortLocatedBlock(lb, targetHost, comparator);
            }
        }
    }
```

```
/**
     * Move decommissioned/stale datanodes to the bottom. Also, sort nodes by
     * network distance.
     *
     * @param lb               located block
     * @param targetHost target host
     * @param comparator dn comparator
     */
    private void sortLocatedBlock(final LocatedBlock lb, String targetHost,
                                  Comparator<DatanodeInfo> comparator) {
        // As it is possible for the separation of node manager and datanode,
        // here we should get node but not datanode only .
        boolean nonDatanodeReader = false;
        Node client = getDatanodeByHost(targetHost);
        if (client == null) {
            nonDatanodeReader = true;
            List<String> hosts = new ArrayList<>(1);
            hosts.add(targetHost);
            List<String> resolvedHosts = dnsToSwitchMapping.resolve(hosts);
            if (resolvedHosts != null && !resolvedHosts.isEmpty()) {
                String rName = resolvedHosts.get(0);
                if (rName != null) {
                    client = new NodeBase(rName + NodeBase.PATH_SEPARATOR_STR +
                            targetHost);
```

```
                    }
                } else {
                    LOG.error("Node Resolution failed. Please make sure that rack " +
                            "awareness scripts are functional.");
                }
            }

            DatanodeInfo[] di = lb.getLocations();
            // Move decommissioned/stale datanodes to the bottom
            //  移除已经下线的  DataNode
            Arrays.sort(di, comparator);

            // Sort nodes by network distance only for located blocks
            int lastActiveIndex = di.length - 1;
            while (lastActiveIndex > 0 && isInactive(di[lastActiveIndex])) {
                --lastActiveIndex;
            }
            int activeLen = lastActiveIndex + 1;
            if (nonDatanodeReader) {
                //  网络拓扑排序
                networktopology.sortByDistanceUsingNetworkLocation(client,
                        lb.getLocations(), activeLen);
            } else {
                networktopology.sortByDistance(client, lb.getLocations(), activeLen);
            }
            // move PROVIDED storage to the end to prefer local replicas.
            lb.moveProvidedToEnd(activeLen);
            // must update cache since we modified locations array
            lb.updateCachedStorageInfo();
        }
```

2.1.1.2 请求下载文件的所有 Block 在线的 DataNode 地址信息封装成 LocatedBlocks 返回给客户端

```
// 获取请求下载 HDFS 文件 Block 信息 (发送 RPC 请求)
                LocatedBlocks locatedBlocks = getLocatedBlocks(src, 0);
```

2.1.2 创建并返回 DFSInputStream

```
    private DFSInputStream openInternal(LocatedBlocks locatedBlocks, String src,
                                        boolean verifyChecksum) throws IOException {
        if (locatedBlocks != null) {
            ErasureCodingPolicy ecPolicy = locatedBlocks.getErasureCodingPolicy();
            if (ecPolicy != null) {
                return new DFSStripedInputStream(this, src, verifyChecksum, ecPolicy,
                        locatedBlocks);
            }
            // 创建 DFSInputStream
            return new DFSInputStream(this, src, verifyChecksum, locatedBlocks);
        } else {
            throw new IOException("Cannot open filename " + src);
        }
    }
```

```
  DFSInputStream(DFSClient dfsClient, String src, boolean verifyChecksum,
      LocatedBlocks locatedBlocks) throws IOException {
    this.dfsClient = dfsClient;
    this.verifyChecksum = verifyChecksum;
    this.src = src;
    synchronized (infoLock) {
      this.cachingStrategy = dfsClient.getDefaultReadCachingStrategy();
    }
    // Block 信息
    this.locatedBlocks = locatedBlocks;
    openInfo(false);
  }
```

2.1.3 包装 DFSInputStream 成 HdfsDataInputStream

```
/**
     * Wraps the stream in a CryptoInputStream if the underlying file is
     * encrypted.
     */
    public HdfsDataInputStream createWrappedInputStream(DFSInputStream dfsis)
            throws IOException {
        FileEncryptionInfo feInfo = dfsis.getFileEncryptionInfo();
        if (feInfo != null) {
            CryptoInputStream cryptoIn;
```

```
                try (TraceScope ignored = getTracer().newScope("decryptEDEK")) {
                    cryptoIn = HdfsKMSUtil.createWrappedInputStream(dfsis,
                            getKeyProvider(), feInfo, getConfiguration());
                }
                return new HdfsDataInputStream(cryptoIn);
            } else {
                // No FileEncryptionInfo so no encryption.
                //  创建  HdfsDataInputStream
                return new HdfsDataInputStream(dfsis);
            }
        }
```

```
    public HdfsDataInputStream(DFSInputStream in) {
        // in =    DFSInputStream
        super(in);
    }
```

```
        public FSDataInputStream(InputStream in) {
            //  往下追
            super(
                    // in = DFSInputStream
                    in
            );
            if (!(in instanceof Seekable) || !(in instanceof PositionedReadable)) {
                throw new IllegalArgumentException(
                        "In is not an instance of Seekable or PositionedReadable");
            }
        }
```

2.2 创建 FileOutputStream (Java IO 编程)

```
    FileOutputStream fos = new FileOutputStream("./LICENSE.txt");
```

2.3 流拷贝

```
    IOUtils.copyBytes(fis, fos, 1024, true);
```

```
/**
     * 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];
        // 最终调用 DFSInputStream.read
        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()

                // 写到本地磁盘
                out.write(buf, 0, bytesRead);
                if ((ps != null) && ps.checkError()) {
                        throw new IOException("Unable to write to output stream.");
                }
                // 继续读取
                bytesRead = in.read(buf);
        }
    }
```

```
    @Override
    public synchronized int read() throws IOException {
        if (oneByteBuf == null) {
            // 赋值
            oneByteBuf = new byte[1];
        }
        // 读取一个字节
        int ret = read(oneByteBuf, 0, 1);
        return (ret <= 0) ? -1 : (oneByteBuf[0] & 0xff);
    }
```

```
    /**
     * Read the entire buffer.
     */
    @Override
    public synchronized int read(@Nonnull final byte buf[], int off, int len)
                throws IOException {
        validatePositionedReadArgs(pos, buf, off, len);
        if (len == 0) {
            return 0;
        }
        // 创建 ByteArrayStrategy
        ReaderStrategy byteArrayReader =
                    new ByteArrayStrategy(buf, off, len, readStatistics, dfsClient);
        // 往下追
        return readWithStrategy(byteArrayReader);
    }
```

```
protected synchronized int readWithStrategy(ReaderStrategy strategy)
        throws IOException {
    dfsClient.checkOpen();
    if (closed.get()) {
        throw new IOException("Stream closed");
    }

    // 1
    int len = strategy.getTargetLength();
    CorruptedBlocks corruptedBlocks = new CorruptedBlocks();
    failures = 0;
    // 0 <  请求下载文件  Block  个数  比如  3
    if (pos < getFileLength()) {
        int retries = 2;
        while (retries > 0) {
            try {
                // currentNode can be left as null if previous read had a checksum
                // error on the same block. See HDFS-3067
                if (pos > blockEnd || currentNode == null) {
                    //  读取第一个  Block  对应的  DataNode
                    //  读取第二个  Block  对应的  DataNode
                    //  读取第 N 个  Block  对应的  DataNode
                    currentNode = blockSeekTo(pos);
                }
                int realLen = (int) Math.min(len, (blockEnd - pos + 1L));
                synchronized (infoLock) {
                    if (locatedBlocks.isLastBlockComplete()) {
                        realLen = (int) Math.min(realLen,
                                locatedBlocks.getFileLength() - pos);
                    }
                }
                //  读取数据
                int result = readBuffer(strategy, realLen, corruptedBlocks);

                if (result >= 0) {
                    pos += result;
                } else {
                    // got a EOS from reader though we expect more data on it.
                    throw new IOException("Unexpected EOS from the reader");
                }
                updateReadStatistics(readStatistics, result, blockReader);

dfsClient.updateFileSystemReadStats(blockReader.getNetworkDistance(),
```

```
                                result);
                        return result;
                    } catch (ChecksumException ce) {
                        throw ce;
                    } catch (IOException e) {
                        checkInterrupted(e);
                        if (retries == 1) {
                            DFSClient.LOG.warn("DFS Read", e);
                        }
                        blockEnd = -1;
                        if (currentNode != null) {
                            addToDeadNodes(currentNode);
                        }
                        if (--retries == 0) {
                            throw e;
                        }
                    } finally {
                        // Check if need to report block replicas corruption either read
                        // was successful or ChecksumException occurred.
                        reportCheckSumFailure(corruptedBlocks,
                                getCurrentBlockLocationsLength(), false);
                    }
                }
            }
            return -1;
        }
```

2.3.1 读取第 N 个 Block 对应的 DataNode

```
/**
     * Open a DataInputStream to a DataNode so that it can be read from.
     * We get block ID and the IDs of the destinations at startup, from the namenode.
     */
    private synchronized DatanodeInfo blockSeekTo(long target)
            throws IOException {
        if (target >= getFileLength()) {
            throw new IOException("Attempted to read past end of file");
        }

        // Will be getting a new BlockReader.
        closeCurrentBlockReaders();
```

```
//
// Connect to best DataNode for desired Block, with potential offset
//
DatanodeInfo chosenNode;
int refetchToken = 1; // only need to get a new access token once
int refetchEncryptionKey = 1; // only need to get a new encryption key once

boolean connectFailedOnce = false;

while (true) {
    //
    // Compute desired block
    //
    // 获取第 N 个 Block 对应的 LocatedBlock
    LocatedBlock targetBlock = getBlockAt(target);

    // update current position
    this.pos = target;
    this.blockEnd = targetBlock.getStartOffset() +
            targetBlock.getBlockSize() - 1;
    this.currentLocatedBlock = targetBlock;

    long offsetIntoBlock = target - targetBlock.getStartOffset();

    // 选择第 pos 个 Block 对应的 DataNode 信息
    DNAddrPair retval = chooseDataNode(targetBlock, null);
    // 选择 DataNode
    chosenNode = retval.info;
    // DataNode 地址
    InetSocketAddress targetAddr = retval.addr;
    StorageType storageType = retval.storageType;
    // Latest block if refreshed by chooseDatanode()
    targetBlock = retval.block;

    try {
        // 获取 BlockReaderRemote
        blockReader = getBlockReader(targetBlock, offsetIntoBlock,
                targetBlock.getBlockSize() - offsetIntoBlock, targetAddr,
                storageType, chosenNode);
        if (connectFailedOnce) {
            DFSClient.LOG.info("Successfully connected to " + targetAddr +
                    " for " + targetBlock.getBlock());
        }
        // 返回存储 Block DataNode 信息
```

```
                    return chosenNode;
                } catch (IOException ex) {
                    checkInterrupted(ex);
                    if (ex instanceof InvalidEncryptionKeyException && refetchEncryptionKey > 0)
{
                        DFSClient.LOG.info("Will fetch a new encryption key and retry, "
                                + "encryption key was invalid when connecting to " +
targetAddr
                                + " : " + ex);
                        // The encryption key used is invalid.
                        refetchEncryptionKey--;
                        dfsClient.clearDataEncryptionKey();
                    } else if (refetchToken > 0 && tokenRefetchNeeded(ex, targetAddr)) {
                        refetchToken--;
                        fetchBlockAt(target);
                    } else {
                        connectFailedOnce = true;
                        DFSClient.LOG.warn("Failed to connect to {} for block {}, " +
                                "add to deadNodes and continue. ", targetAddr,
                                targetBlock.getBlock(), ex);
                        // Put chosen node into dead list, continue
                        addToDeadNodes(chosenNode);
                    }
                }
            }
        }
```

```
    private DNAddrPair chooseDataNode(LocatedBlock block,
                                      Collection<DatanodeInfo> ignoredNodes) throws
IOException {
        // 往下追
        return chooseDataNode(block, ignoredNodes, true);
    }
```

```
/**
     * Choose datanode to read from.
     *
     * @param block                 Block to choose datanode addr from
     * @param ignoredNodes          Ignored nodes inside.
     * @param refetchIfRequired Whether to refetch if no nodes to chose
     *                              from.
     * @return Returns chosen DNAddrPair; Can be null if refetchIfRequired is
     * false.
     */
```

```
    private DNAddrPair chooseDataNode(LocatedBlock block,
                                      Collection<DatanodeInfo> ignoredNodes, boolean
refetchIfRequired)
            throws IOException {
        while (true) {
            // 选择
            DNAddrPair result = getBestNodeDNAddrPair(block, ignoredNodes);
            if (result != null) {
                return result;
            } else if (refetchIfRequired) {
                block = refetchLocations(block, ignoredNodes);
            } else {
                return null;
            }
        }
    }
```

```
 /**
     * Get the best node from which to stream the data.
     * @param block LocatedBlock, containing nodes in priority order.
     * @param ignoredNodes Do not choose nodes in this array (may be null)
     * @return The DNAddrPair of the best node. Null if no node can be chosen.
     */
    protected DNAddrPair getBestNodeDNAddrPair(LocatedBlock block,
                                               Collection<DatanodeInfo>
ignoredNodes) {
        // 获取 Block 对应的所有 DataNode 信息
        DatanodeInfo[] nodes = block.getLocations();
        StorageType[] storageTypes = block.getStorageTypes();
        DatanodeInfo chosenNode = null;
        StorageType storageType = null;
        if (nodes != null) {
            for (int i = 0; i < nodes.length; i++) {
                if (!deadNodes.containsKey(nodes[i])
                        && (ignoredNodes == null || !ignoredNodes.contains(nodes[i]))) {
                    // 一般情况下直接返回第一个
                    chosenNode = nodes[i];
                    // Storage types are ordered to correspond with nodes, so use the same
                    // index to get storage type.
                    if (storageTypes != null && i < storageTypes.length) {
                        storageType = storageTypes[i];
                    }
                    break;
                }
```

```
                }
            }
            if (chosenNode == null) {
                reportLostBlock(block, ignoredNodes);
                return null;
            }

            // 封装选择 DataNode 信息
            final String dnAddr =
                    chosenNode.getXferAddr(dfsClient.getConf().isConnectToDnViaHostname());
            DFSClient.LOG.debug("Connecting to datanode {}", dnAddr);
            InetSocketAddress targetAddr = NetUtils.createSocketAddr(dnAddr);
            // 创建 DNAddrPair 并返回
            return new DNAddrPair(chosenNode, targetAddr, storageType, block);
        }
```

2.3.1.1 获取 BlockReaderRemote

```
protected BlockReader getBlockReader(LocatedBlock targetBlock,
                                     long offsetInBlock, long length,
InetSocketAddress targetAddr,
                                     StorageType storageType, DatanodeInfo
datanode) throws IOException {
            ExtendedBlock blk = targetBlock.getBlock();
            Token<BlockTokenIdentifier> accessToken = targetBlock.getBlockToken();
            CachingStrategy curCachingStrategy;
            boolean shortCircuitForbidden;
            synchronized (infoLock) {
                curCachingStrategy = cachingStrategy;
                shortCircuitForbidden = shortCircuitForbidden();
            }
            // 返回 BlockReaderRemote
            return new BlockReaderFactory(dfsClient.getConf()).
                    setInetSocketAddress(targetAddr).
                    setRemotePeerFactory(dfsClient).
                    setDatanodeInfo(datanode).
                    setStorageType(storageType).
                    setFileName(src).
                    setBlock(blk).
                    setBlockToken(accessToken).
                    setStartOffset(offsetInBlock).
                    setVerifyChecksum(verifyChecksum).
```

```
                setClientName(dfsClient.clientName).
                setLength(length).
                setCachingStrategy(curCachingStrategy).
                setAllowShortCircuitLocalReads(!shortCircuitForbidden).
                setClientCacheContext(dfsClient.getClientContext()).
                setUserGroupInformation(dfsClient.ugi).
                setConfiguration(dfsClient.getConfiguration()).
                build();
    }
```

```
/**
   * Build a BlockReader with the given options.
   *
   * This function will do the best it can to create a block reader that meets
   * all of our requirements.  We prefer short-circuit block readers
   * (BlockReaderLocal and BlockReaderLocalLegacy) over remote ones, since the
   * former avoid the overhead of socket communication.  If short-circuit is
   * unavailable, our next fallback is data transfer over UNIX domain sockets,
   * if dfs.client.domain.socket.data.traffic has been enabled.  If that doesn't
   * work, we will try to create a remote block reader that operates over TCP
   * sockets.
   *
   * There are a few caches that are important here.
   *
   * The ShortCircuitCache stores file descriptor objects which have been passed
   * from the DataNode.
   *
   * The DomainSocketFactory stores information about UNIX domain socket paths
   * that we not been able to use in the past, so that we don't waste time
   * retrying them over and over.  (Like all the caches, it does have a timeout,
   * though.)
   *
   * The PeerCache stores peers that we have used in the past.  If we can reuse
   * one of these peers, we avoid the overhead of re-opening a socket.  However,
   * if the socket has been timed out on the remote end, our attempt to reuse
   * the socket may end with an IOException.  For that reason, we limit our
   * attempts at socket reuse to dfs.client.cached.conn.retry times.  After
   * that, we create new sockets.  This avoids the problem where a thread tries
   * to talk to a peer that it hasn't talked to in a while, and has to clean out
   * every entry in a socket cache full of stale entries.
   *
   * @return The new BlockReader.  We will not return null.
   *
   * @throws InvalidToken
```

```
   *                   If the block token was invalid.
   *             InvalidEncryptionKeyException
   *                   If the encryption key was invalid.
   *             Other IOException
   *                   If there was another problem.
   */
  public BlockReader build() throws IOException {
    Preconditions.checkNotNull(configuration);
    Preconditions
        .checkState(length >= 0, "Length must be set to a non-negative value");
    BlockReader reader = tryToCreateExternalBlockReader();
    if (reader != null) {
      return reader;
    }
    final ShortCircuitConf scConf = conf.getShortCircuitConf();
    try {
      // 判断是否为本地短路读取  Block (默认情况下不开启)
      if (scConf.isShortCircuitLocalReads() && allowShortCircuitLocalReads) {
        if (clientContext.getUseLegacyBlockReaderLocal()) {
          reader = getLegacyBlockReaderLocal();
          if (reader != null) {
            LOG.trace("{}: returning new legacy block reader local.", this);
            return reader;
          }
        } else {
          reader = getBlockReaderLocal();
          if (reader != null) {
            LOG.trace("{}: returning new block reader local.", this);
            return reader;
          }
        }
      }
      if (scConf.isDomainSocketDataTraffic()) {
        reader = getRemoteBlockReaderFromDomain();
        if (reader != null) {
          LOG.trace("{}: returning new remote block reader using UNIX domain "
              + "socket on {}", this, pathInfo.getPath());
          return reader;
        }
      }
    } catch (IOException e) {
      LOG.debug("Block read failed. Getting remote block reader using TCP", e);
    }
    Preconditions.checkState(!DFSInputStream.tcpReadsDisabledForTesting,
```

```
            "TCP reads were disabled for testing, but we failed to " +
            "do a non-TCP read.");
      // 从远程获取 Block 数据 通过 TCP
      return getRemoteBlockReaderFromTcp();
    }
```

```
/**
     * Get a BlockReaderRemote that communicates over a TCP socket.
     *
     * @return The new BlockReader.    We will not return null, but instead throw
     *                an exception if this fails.
     *
     * @throws InvalidToken
     *                    If the block token was invalid.
     *                InvalidEncryptionKeyException
     *                    If the encryption key was invalid.
     *                Other IOException
     *                    If there was another problem.
     */
    private BlockReader getRemoteBlockReaderFromTcp() throws IOException {
      LOG.trace("{}: trying to create a remote block reader from a TCP socket",
          this);
      BlockReader blockReader = null;
      while (true) {
        BlockReaderPeer curPeer = null;
        Peer peer = null;
        try {
          // 创建 BlockReaderPeer
          curPeer = nextTcpPeer();
          if (curPeer.fromCache) remainingCacheTries--;
          // BasicInetPeer
          peer = curPeer.peer;
          // 发送一个请求给 DataNode 读取对应的 Block 数据
          // 返回 BlockReaderRemote
          blockReader = getRemoteBlockReader(peer);
          return blockReader;
        } catch (IOException ioe) {
          if (isSecurityException(ioe)) {
            LOG.trace("{}: got security exception while constructing a remote "
                + "block reader from {}", this, peer, ioe);
            throw ioe;
          }
          if ((curPeer != null) && curPeer.fromCache) {
            // Handle an I/O error we got when using a cached peer.    These are
```

```
                // considered less serious, because the underlying socket may be
                // stale.
                LOG.debug("Closed potentially stale remote peer {}", peer, ioe);
              } else {
                // Handle an I/O error we got when using a newly created peer.
                LOG.warn("I/O error constructing remote block reader.", ioe);
                throw ioe;
              }
            } finally {
              if (blockReader == null) {
                IOUtilsClient.cleanup(LOG, peer);
              }
            }
          }
        }
```

```
  @SuppressWarnings("deprecation")
  private BlockReader getRemoteBlockReader(Peer peer) throws IOException {
    int networkDistance = clientContext.getNetworkDistance(datanode);
    // 往下追
    return BlockReaderRemote.newBlockReader(
        fileName, block, token, startOffset, length,
        verifyChecksum, clientName, peer, datanode,
        clientContext.getPeerCache(), cachingStrategy,
        networkDistance);
  }
```

```
  /**
   * Create a new BlockReader specifically to satisfy a read.
   * This method also sends the OP_READ_BLOCK request.
   *
   * @param file  File location
   * @param block  The block object
   * @param blockToken  The block token for security
   * @param startOffset  The read offset, relative to block head
   * @param len  The number of bytes to read
   * @param verifyChecksum  Whether to verify checksum
   * @param clientName  Client name
   * @param peer  The Peer to use
   * @param datanodeID  The DatanodeID this peer is connected to
   * @return New BlockReader instance, or null on error.
   */
  public static BlockReader newBlockReader(String file,
      ExtendedBlock block,
```

```
    Token<BlockTokenIdentifier> blockToken,
    long startOffset, long len,
    boolean verifyChecksum,
    String clientName,
    Peer peer, DatanodeID datanodeID,
    PeerCache peerCache,
    CachingStrategy cachingStrategy,
    int networkDistance) throws IOException {
// in and out will be closed when sock is closed (by the caller)
// 获取 Client 与 DataNode 连接的输出流
final DataOutputStream out = new DataOutputStream(new BufferedOutputStream(
        peer.getOutputStream()));
// 发送一个请求读取 Block 数据
new Sender(out).readBlock(block, blockToken, clientName, startOffset, len,
        verifyChecksum, cachingStrategy);

//
// Get bytes in block
//
// 获取输入流
DataInputStream in = new DataInputStream(peer.getInputStream());

BlockOpResponseProto status = BlockOpResponseProto.parseFrom(
        PBHelperClient.vintPrefixed(in));
checkSuccess(status, peer, block, file);
ReadOpChecksumInfoProto checksumInfo =
        status.getReadOpChecksumInfo();
DataChecksum checksum = DataTransferProtoUtil.fromProto(
        checksumInfo.getChecksum());
//Warning when we get CHECKSUM_NULL?

// Read the first chunk offset.
long firstChunkOffset = checksumInfo.getChunkOffset();

if ( firstChunkOffset < 0 || firstChunkOffset > startOffset ||
        firstChunkOffset <= (startOffset - checksum.getBytesPerChecksum())) {
    throw new IOException("BlockReader: error in first chunk offset (" +
            firstChunkOffset + ") startOffset is " +
            startOffset + " for file " + file);
}

// 创建 BlockReaderRemote
return new BlockReaderRemote(file, block.getBlockId(), checksum,
        verifyChecksum, startOffset, firstChunkOffset, len, peer, datanodeID,
```

```
            peerCache, networkDistance);
    }
```

```
protected BlockReaderRemote(String file, long blockId,
                                        DataChecksum checksum, boolean verifyChecksum,
                                        long startOffset, long firstChunkOffset,
                                        long bytesToRead, Peer peer,
                                        DatanodeID datanodeID, PeerCache peerCache,
                                        int networkDistance) {
      // Path is used only for printing block and file information in debug
      this.peer = peer;
      this.datanodeID = datanodeID;
      this.in = peer.getInputStreamChannel();
      this.checksum = checksum;
      this.verifyChecksum = verifyChecksum;
      this.startOffset = Math.max( startOffset, 0 );
      this.filename = file;
      this.peerCache = peerCache;
      this.blockId = blockId;

      // The total number of bytes that we need to transfer from the DN is
      // the amount that the user wants (bytesToRead), plus the padding at
      // the beginning in order to chunk-align. Note that the DN may elect
      // to send more than this amount if the read starts/ends mid-chunk.
      this.bytesNeededToFinish = bytesToRead + (startOffset - firstChunkOffset);
      bytesPerChecksum = this.checksum.getBytesPerChecksum();
      checksumSize = this.checksum.getChecksumSize();
      this.networkDistance = networkDistance;
    }
```

2.3.1.1.1 连接 DataNode (创建 BlockReaderPeer)

```
/**
     * Get the next TCP-based peer-- either from the cache or by creating it.
     *
     * @return the next Peer, or null if we could not construct one.
     *
     * @throws IOException    If there was an error while constructing the peer
     *                        (such as an InvalidEncryptionKeyException)
     */
    private BlockReaderPeer nextTcpPeer() throws IOException {
      if (remainingCacheTries > 0) {
        Peer peer = clientContext.getPeerCache().get(datanode, false);
        if (peer != null) {
```

```
            LOG.trace("nextTcpPeer: reusing existing peer {}", peer);
            return new BlockReaderPeer(peer, true);
          }
        }
        try {
          // 连接 DataNode 的 DataXceiverServer 的 run() 里面的 TcpPeerServer
          Peer peer = remotePeerFactory.newConnectedPeer(inetSocketAddress, token,
              datanode);
          LOG.trace("nextTcpPeer: created newConnectedPeer {}", peer);
          // 创建 BlockReaderPeer
          return new BlockReaderPeer(peer, false);
        } catch (IOException e) {
          LOG.trace("nextTcpPeer: failed to create newConnectedPeer connected to"
              + "{}", datanode);
          throw e;
        }
      }
```

2.3.2 读取数据

```
/* This is a used by regular read() and handles ChecksumExceptions.
       * name readBuffer() is chosen to imply similarity to readBuffer() in
       * ChecksumFileSystem
       */
      private synchronized int readBuffer(ReaderStrategy reader, int len,
                                          CorruptedBlocks corruptedBlocks)
              throws IOException {
          IOException ioe;

          /* we retry current node only once. So this is set to true only here.
           * Intention is to handle one common case of an error that is not a
           * failure on datanode or client : when DataNode closes the connection
           * since client is idle. If there are other cases of "non-errors" then
           * then a datanode might be retried by setting this to true again.
           */
          boolean retryCurrentNode = true;

          while (true) {
              // retry as many times as seekToNewSource allows.
              try {
                  // 读取数据 调用 ByteArrayStrategy.readFromBlock()
                  return reader.readFromBlock(blockReader, len);
```

```
            } catch (ChecksumException ce) {
                DFSClient.LOG.warn("Found Checksum error for "
                        + getCurrentBlock() + " from " + currentNode
                        + " at " + ce.getPos());
                ioe = ce;
                retryCurrentNode = false;
                // we want to remember which block replicas we have tried
                corruptedBlocks.addCorruptedBlock(getCurrentBlock(), currentNode);
            } catch (IOException e) {
                if (!retryCurrentNode) {
                    DFSClient.LOG.warn("Exception while reading from "
                            + getCurrentBlock() + " of " + src + " from "
                            + currentNode, e);
                }
                ioe = e;
            }
            boolean sourceFound;
            if (retryCurrentNode) {
                /* possibly retry the same node so that transient errors don't
                 * result in application level failures (e.g. Datanode could have
                 * closed the connection because the client is idle for too long).
                 */
                sourceFound = seekToBlockSource(pos);
            } else {
                addToDeadNodes(currentNode);
                sourceFound = seekToNewSource(pos);
            }
            if (!sourceFound) {
                throw ioe;
            }
            retryCurrentNode = false;
        }
    }
```

```
    @Override
    public int readFromBlock(BlockReader blockReader,
                             int length) throws IOException {
        // blockReader = BlockReaderRemote
        // 读取数据
        int nRead = blockReader.read(readBuf, offset, length);
        if (nRead > 0) {
            offset += nRead;
        }
```

```
            return nRead;
        }
```

```
@Override
    public synchronized int read(byte[] buf, int off, int len)
            throws IOException {
        boolean logTraceEnabled = LOG.isTraceEnabled();
        UUID randomId = null;
        if (logTraceEnabled) {
            randomId = UUID.randomUUID();
            LOG.trace("Starting read #{} file {} from datanode {}",
                  randomId, filename, datanodeID.getHostName());
        }

        if (curDataSlice == null ||
              curDataSlice.remaining() == 0 && bytesNeededToFinish > 0) {

            // 读取下一个 Packet
            readNextPacket();
        }

        if (logTraceEnabled) {
            LOG.trace("Finishing read #{}", randomId);
        }

        if (curDataSlice.remaining() == 0) {
            // we're at EOF now
            return -1;
        }

        int nRead = Math.min(curDataSlice.remaining(), len);
        curDataSlice.get(buf, off, nRead);

        return nRead;
    }
```

```
private void readNextPacket() throws IOException {
        //Read packet headers.
        // 读取 Packet Header
        packetReceiver.receiveNextPacket(in);

        PacketHeader curHeader = packetReceiver.getHeader();
        curDataSlice = packetReceiver.getDataSlice();
        assert curDataSlice.capacity() == curHeader.getDataLen();
```

```
LOG.trace("DFSClient readNextPacket got header {}", curHeader);

// Sanity check the lengths
if (!curHeader.sanityCheck(lastSeqNo)) {
   throw new IOException("BlockReader: error in packet header " +
         curHeader);
}

if (curHeader.getDataLen() > 0) {
   int chunks = 1 + (curHeader.getDataLen() - 1) / bytesPerChecksum;
   int checksumsLen = chunks * checksumSize;

   assert packetReceiver.getChecksumSlice().capacity() == checksumsLen :
         "checksum slice capacity=" +
               packetReceiver.getChecksumSlice().capacity() +
               " checksumsLen=" + checksumsLen;

   lastSeqNo = curHeader.getSeqno();
   if (verifyChecksum && curDataSlice.remaining() > 0) {
      // N.B.: the checksum error offset reported here is actually
      // relative to the start of the block, not the start of the file.
      // This is slightly misleading, but preserves the behavior from
      // the older BlockReader.
      checksum.verifyChunkedSums(curDataSlice,
            packetReceiver.getChecksumSlice(),
            filename, curHeader.getOffsetInBlock());
   }
   bytesNeededToFinish -= curHeader.getDataLen();
}

// First packet will include some data prior to the first byte
// the user requested. Skip it.
if (curHeader.getOffsetInBlock() < startOffset) {
   int newPos = (int) (startOffset - curHeader.getOffsetInBlock());
   curDataSlice.position(newPos);
}

// If we've now satisfied the whole client read, read one last packet
// header, which should be empty
if (bytesNeededToFinish <= 0) {
   readTrailingEmptyPacket();
   if (verifyChecksum) {
      sendReadResult(Status.CHECKSUM_OK);
```

```
        } else {
          sendReadResult(Status.SUCCESS);
        }
      }
    }
```

```
/**
     * Reads all of the data for the next packet into the appropriate buffers.
     * <p>
     * The data slice and checksum slice members will be set to point to the
     * user data and corresponding checksums. The header will be parsed and
     * set.
     */
    public void receiveNextPacket(ReadableByteChannel in) throws IOException {
        // 往下追
        doRead(in, null);
    }
```

```
private void doRead(ReadableByteChannel ch, InputStream in)
                throws IOException {
        // Each packet looks like:
        //     PLEN       HLEN          HEADER        CHECKSUMS    DATA
        //     32-bit    16-bit     <protobuf>    <variable length>
        //
        // PLEN:          Payload length
        //                  = length(PLEN) + length(CHECKSUMS) + length(DATA)
        //                  This length includes its own encoded length in
        //                  the sum for historical reasons.
        //
        // HLEN:          Header length
        //                  = length(HEADER)
        //
        // HEADER:       the actual packet header fields, encoded in protobuf
        // CHECKSUMS: the crcs for the data chunk. May be missing if
        //                  checksums were not requested
        // DATA           the actual block data
        Preconditions.checkState(curHeader == null || !curHeader.isLastPacketInBlock());

        curPacketBuf.clear();
        curPacketBuf.limit(PacketHeader.PKT_LENGTHS_LEN);
        // 读取 packet 到 curPacketBuf = ByteBuf
        doReadFully(ch, in, curPacketBuf);
        curPacketBuf.flip();
```

```
int payloadLen = curPacketBuf.getInt();

if (payloadLen < Ints.BYTES) {
      // The "payload length" includes its own length. Therefore it
      // should never be less than 4 bytes
      throw new IOException("Invalid payload length " +
                  payloadLen);
}
int dataPlusChecksumLen = payloadLen - Ints.BYTES;
int headerLen = curPacketBuf.getShort();
if (headerLen < 0) {
      throw new IOException("Invalid header length " + headerLen);
}

LOG.trace("readNextPacket: dataPlusChecksumLen={}, headerLen={}",
            dataPlusChecksumLen, headerLen);

// Sanity check the buffer size so we don't allocate too much memory
// and OOME.
int totalLen = payloadLen + headerLen;
if (totalLen < 0 || totalLen > MAX_PACKET_SIZE) {
      throw new IOException("Incorrect value for packet payload size: " +
                  payloadLen);
}

// Make sure we have space for the whole packet, and
// read it.
// 确保有足够的空间存放整个 packet
reallocPacketBuf(PacketHeader.PKT_LENGTHS_LEN +
            dataPlusChecksumLen + headerLen);

curPacketBuf.clear();
curPacketBuf.position(PacketHeader.PKT_LENGTHS_LEN);
curPacketBuf.limit(PacketHeader.PKT_LENGTHS_LEN +
            dataPlusChecksumLen + headerLen);

// 读取数据 到 curPacketBuf
doReadFully(ch, in, curPacketBuf);

curPacketBuf.flip();
curPacketBuf.position(PacketHeader.PKT_LENGTHS_LEN);

// Extract the header from the front of the buffer (after the length prefixes)
// 读取头部信息
```

```
            byte[] headerBuf = new byte[headerLen];
            curPacketBuf.get(headerBuf);
            if (curHeader == null) {
                curHeader = new PacketHeader();
            }
            curHeader.setFieldsFromData(payloadLen, headerBuf);

            // Compute the sub-slices of the packet
            int checksumLen = dataPlusChecksumLen - curHeader.getDataLen();
            if (checksumLen < 0) {
                throw new IOException("Invalid packet: data length in packet header " +
                        "exceeds data length received. dataPlusChecksumLen=" +
                        dataPlusChecksumLen + " header: " + curHeader);
            }

            // 往下追
            reslicePacket(headerLen, checksumLen, curHeader.getDataLen());
        }
```

2.3.3 DataNode 处理 Client 请求下载 Block

入口：DataXceiverServer.run() 最终调用 DataXceiver.run()

```
/**
     * 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)）
                    // 读取数据类型（READ_BLOCK (81)）
                    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
                // 处理 READ_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);
            }
        }
    }
```

```
/** Process op by the corresponding method. */
    protected final void processOp(Op op) throws IOException {
        switch (op) {
            case READ_BLOCK:
                // 读取 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_READ_BLOCK */
    private void opReadBlock() throws IOException {
        OpReadBlockProto proto = OpReadBlockProto.parseFrom(vintPrefixed(in));
        TraceScope traceScope = continueTraceSpan(proto.getHeader(),
                proto.getClass().getSimpleName());
        try {
            // 往下追 (调用 DataXceiver.readBlock())
            readBlock(PBHelperClient.convert(proto.getHeader().getBaseHeader().getBlock()),
                    PBHelperClient.convert(proto.getHeader().getBaseHeader().getToken()),
                    proto.getHeader().getClientName(),
                    proto.getOffset(),
                    proto.getLen(),
                    proto.getSendChecksums(),
                    (proto.hasCachingStrategy() ?
                            getCachingStrategy(proto.getCachingStrategy()) :
                            CachingStrategy.newDefaultStrategy()));
        } finally {
```

```
            if (traceScope != null) traceScope.close();
        }
    }
```

```
@Override
    public void readBlock(final ExtendedBlock block,
                          final Token<BlockTokenIdentifier> blockToken,
                          final String clientName,
                          final long blockOffset,
                          final long length,
                          final boolean sendChecksum,
                          final CachingStrategy cachingStrategy) throws IOException {
        previousOpClientName = clientName;
        long read = 0;
        updateCurrentThreadName("Sending block " + block);
        // socket output stream
        OutputStream baseStream = getOutputStream();
        // 封装 Socket output stream 的输出流为 DataOutputStream
        DataOutputStream out = getBufferedOutputStream();
        checkAccess(out, true, block, blockToken, Op.READ_BLOCK,
                BlockTokenIdentifier.AccessMode.READ);

        // send the block
        // 发送 Block 对象 BlockSender
        BlockSender blockSender = null;
        DatanodeRegistration dnR =
                datanode.getDNRegistrationForBP(block.getBlockPoolId());
        final String clientTraceFmt =
                clientName.length() > 0 && ClientTraceLog.isInfoEnabled()
                        ?       String.format(DN_CLIENTTRACE_FORMAT,       localAddress,
remoteAddress,
                        "%d", "HDFS_READ", clientName, "%d",
                        dnR.getDatanodeUuid(), block, "%d")
                        : dnR + " Served block " + block + " to " +
                        remoteAddress;

        try {
            try {
                // 创建 BlockSender
                blockSender = new BlockSender(block, blockOffset, length,
                        true, false, sendChecksum, datanode, clientTraceFmt,
                        cachingStrategy);
            } catch (IOException e) {
```

```
                String msg = "opReadBlock " + block + " received exception " + e;
                LOG.info(msg);
                sendResponse(ERROR, msg);
                throw e;
            }

            // send op status
            writeSuccessWithChecksumInfo(blockSender, new
DataOutputStream(getOutputStream()));

            long beginRead = Time.monotonicNow();
            // 发送 Block
            read = blockSender.sendBlock(out, baseStream, null); // send data
            long duration = Time.monotonicNow() - beginRead;
            if (blockSender.didSendEntireByteRange()) {
                // If we sent the entire range, then we should expect the client
                // to respond with a Status enum.
                try {
                    ClientReadStatusProto stat = ClientReadStatusProto.parseFrom(
                            PBHelperClient.vintPrefixed(in));
                    if (!stat.hasStatus()) {
                        LOG.warn("Client {} did not send a valid status code " +
                                "after reading. Will close connection.",
                                peer.getRemoteAddressString());
                        IOUtils.closeStream(out);
                    }
                } catch (IOException ioe) {
                    LOG.debug("Error reading client status response. Will close
connection.", ioe);
                    IOUtils.closeStream(out);
                    incrDatanodeNetworkErrors();
                }
            } else {
                IOUtils.closeStream(out);
            }
            datanode.metrics.incrBytesRead((int) read);
            datanode.metrics.incrBlocksRead();
            datanode.metrics.incrTotalReadTime(duration);
        } catch (SocketException ignored) {
            LOG.trace("{}:Ignoring exception while serving {} to {}",
                    dnR, block, remoteAddress, ignored);
            // Its ok for remote side to close the connection anytime.
            datanode.metrics.incrBlocksRead();
            IOUtils.closeStream(out);
```

```
            } catch (IOException ioe) {
                /* What exactly should we do here?
                 * Earlier version shutdown() datanode if there is disk error.
                 */
                if (!(ioe instanceof SocketTimeoutException)) {
                    LOG.warn("{}:Got exception while serving {} to {}",
                            dnR, block, remoteAddress, ioe);
                    incrDatanodeNetworkErrors();
                }
                // Normally the client reports a bad block to the NN. However if the
                // meta file is corrupt or an disk error occurs (EIO), then the client
                // never gets a chance to do validation, and hence will never report
                // the block as bad. For some classes of IO exception, the DN should
                // report the block as bad, via the handleBadBlock() method
                datanode.handleBadBlock(block, ioe, false);
                throw ioe;
            } finally {
                IOUtils.closeStream(blockSender);
            }

            //update metrics
            datanode.metrics.addReadBlockOp(elapsed());
            datanode.metrics.incrReadsFromClient(peer.isLocal(), read);
        }
```

2.3.3.1 创建 BlockSender

```
/**
     * Constructor
     *
     * @param block Block that is being read
     * @param startOffset starting offset to read from
     * @param length length of data to read
     * @param corruptChecksumOk if true, corrupt checksum is okay
     * @param verifyChecksum verify checksum while reading the data
     * @param sendChecksum send checksum to client.
     * @param datanode datanode from which the block is being read
     * @param clientTraceFmt format string used to print client trace logs
     * @throws IOException
     */
    BlockSender(ExtendedBlock block, long startOffset, long length,
                boolean corruptChecksumOk, boolean verifyChecksum,
```

```
                boolean sendChecksum, DataNode datanode, String clientTraceFmt,
                CachingStrategy cachingStrategy)
    throws IOException {
  InputStream blockIn = null;
  DataInputStream checksumIn = null;
  FsVolumeReference volumeRef = null;
  this.fileIoProvider = datanode.getFileIoProvider();
  try {
    this.block = block;
    this.corruptChecksumOk = corruptChecksumOk;
    this.verifyChecksum = verifyChecksum;
    this.clientTraceFmt = clientTraceFmt;

    /*
     * If the client asked for the cache to be dropped behind all reads,
     * we honor that.    Otherwise, we use the DataNode defaults.
     * When using DataNode defaults, we use a heuristic where we only
     * drop the cache for large reads.
     */
    if (cachingStrategy.getDropBehind() == null) {
      this.dropCacheBehindAllReads = false;
      this.dropCacheBehindLargeReads =
          datanode.getDnConf().dropCacheBehindReads;
    } else {
      this.dropCacheBehindAllReads =
          this.dropCacheBehindLargeReads =
              cachingStrategy.getDropBehind().booleanValue();
    }
    /*
     * Similarly, if readahead was explicitly requested, we always do it.
     * Otherwise, we read ahead based on the DataNode settings, and only
     * when the reads are large.
     */
    if (cachingStrategy.getReadahead() == null) {
      this.alwaysReadahead = false;
      this.readaheadLength = datanode.getDnConf().readaheadLength;
    } else {
      this.alwaysReadahead = true;
      this.readaheadLength = cachingStrategy.getReadahead().longValue();
    }
    this.datanode = datanode;

    if (verifyChecksum) {
      // To simplify implementation, callers may not specify verification
```

```
    // without sending.
    Preconditions.checkArgument(sendChecksum,
          "If verifying checksum, currently must also send it.");
  }

  // if there is a append write happening right after the BlockSender
  // is constructed, the last partial checksum maybe overwritten by the
  // append, the BlockSender need to use the partial checksum before
  // the append write.
  ChunkChecksum chunkChecksum = null;
  final long replicaVisibleLength;
  try(AutoCloseableLock lock = datanode.data.acquireDatasetLock()) {
    replica = getReplica(block, datanode);
    replicaVisibleLength = replica.getVisibleLength();
  }
  if (replica.getState() == ReplicaState.RBW) {
    final ReplicaInPipeline rbw = (ReplicaInPipeline) replica;
    waitForMinLength(rbw, startOffset + length);
    chunkChecksum = rbw.getLastChecksumAndDataLen();
  }
  if (replica instanceof FinalizedReplica) {
    chunkChecksum = getPartialChunkChecksumForFinalized(
          (FinalizedReplica)replica);
  }

  if (replica.getGenerationStamp() < block.getGenerationStamp()) {
    throw new IOException("Replica gen stamp < block genstamp, block="
          + block + ", replica=" + replica);
  } else if (replica.getGenerationStamp() > block.getGenerationStamp()) {
    if (DataNode.LOG.isDebugEnabled()) {
      DataNode.LOG.debug("Bumping up the client provided"
            + " block's genstamp to latest " + replica.getGenerationStamp()
            + " for block " + block);
    }
    block.setGenerationStamp(replica.getGenerationStamp());
  }
  if (replicaVisibleLength < 0) {
    throw new IOException("Replica is not readable, block="
          + block + ", replica=" + replica);
  }
  if (DataNode.LOG.isDebugEnabled()) {
    DataNode.LOG.debug("block=" + block + ", replica=" + replica);
  }
```

```
// transferToFully() fails on 32 bit platforms for block sizes >= 2GB,
// use normal transfer in those cases
this.transferToAllowed = datanode.getDnConf().transferToAllowed &&
    (!is32Bit || length <= Integer.MAX_VALUE);

// Obtain a reference before reading data
//  获取  Block FSVolume  磁盘存储路径
volumeRef = datanode.data.getVolume(block).obtainReference();

/*
  * (corruptChecksumOK, meta_file_exist): operation
  * True,      True: will verify checksum
  * True,    False: No verify, e.g., need to read data from a corrupted file
  * False,    True: will verify checksum
  * False, False: throws IOException file not found
  */
DataChecksum csum = null;
if (verifyChecksum || sendChecksum) {
    LengthInputStream metaIn = null;
    boolean keepMetaInOpen = false;
    try {
        DataNodeFaultInjector.get().throwTooManyOpenFiles();
        metaIn = datanode.data.getMetaDataInputStream(block);
        if (!corruptChecksumOk || metaIn != null) {
            if (metaIn == null) {
                //need checksum but meta-data not found
                throw new FileNotFoundException("Meta-data not found for " +
                        block);
            }

            // The meta file will contain only the header if the NULL checksum
            // type was used, or if the replica was written to transient storage.
            // Also, when only header portion of a data packet was transferred
            // and then pipeline breaks, the meta file can contain only the
            // header and 0 byte in the block data file.
            // Checksum verification is not performed for replicas on transient
            // storage.    The header is important for determining the checksum
            // type later when lazy persistence copies the block to non-transient
            // storage and computes the checksum.
            int expectedHeaderSize = BlockMetadataHeader.getHeaderSize();
            if (!replica.isOnTransientStorage() &&
                    metaIn.getLength() >= expectedHeaderSize) {
                checksumIn = new DataInputStream(new BufferedInputStream(
                        metaIn, IO_FILE_BUFFER_SIZE));
```

```
            csum = BlockMetadataHeader.readDataChecksum(checksumIn, block);
            keepMetaInOpen = true;
          } else if (!replica.isOnTransientStorage() &&
              metaIn.getLength() < expectedHeaderSize) {
            LOG.warn("The meta file length {} is less than the expected " +
                "header length {}, indicating the meta file is corrupt",
                metaIn.getLength(), expectedHeaderSize);
            throw new CorruptMetaHeaderException("The meta file length "+
                metaIn.getLength()+" is less than the expected length "+
                expectedHeaderSize);
          }
        } else {
          LOG.warn("Could not find metadata file for " + block);
        }
      } catch (FileNotFoundException e) {
        if ((e.getMessage() != null) && !(e.getMessage()
            .contains("Too many open files"))) {
          // The replica is on its volume map but not on disk
          datanode
              .notifyNamenodeDeletedBlock(block, replica.getStorageUuid());
          datanode.data.invalidate(block.getBlockPoolId(),
              new Block[] {block.getLocalBlock()});
        }
        throw e;
      } finally {
        if (!keepMetaInOpen) {
          IOUtils.closeStream(metaIn);
        }
      }
    }
    if (csum == null) {
      csum = DataChecksum.newDataChecksum(DataChecksum.Type.NULL,
          (int)CHUNK_SIZE);
    }

    /*
     * If chunkSize is very large, then the metadata file is mostly
     * corrupted. For now just truncate bytesPerchecksum to blockLength.
     */
    int size = csum.getBytesPerChecksum();
    if (size > 10*1024*1024 && size > replicaVisibleLength) {
      csum = DataChecksum.newDataChecksum(csum.getChecksumType(),
          Math.max((int)replicaVisibleLength, 10*1024*1024));
```

```
      size = csum.getBytesPerChecksum();
    }
    chunkSize = size;
    checksum = csum;
    checksumSize = checksum.getChecksumSize();
    length = length < 0 ? replicaVisibleLength : length;

    // end is either last byte on disk or the length for which we have a
    // checksum
    long end = chunkChecksum != null ? chunkChecksum.getDataLength()
        : replica.getBytesOnDisk();
    if (startOffset < 0 || startOffset > end
        || (length + startOffset) > end) {
      String msg = " Offset " + startOffset + " and length " + length
      + " don't match block " + block + " ( blockLen " + end + " )";
      LOG.warn(datanode.getDNRegistrationForBP(block.getBlockPoolId()) +
          ":sendBlock() : " + msg);
      throw new IOException(msg);
    }

    // Ensure read offset is position at the beginning of chunk
    offset = startOffset - (startOffset % chunkSize);
    if (length >= 0) {
      // Ensure endOffset points to end of chunk.
      long tmpLen = startOffset + length;
      if (tmpLen % chunkSize != 0) {
        tmpLen += (chunkSize - tmpLen % chunkSize);
      }
      if (tmpLen < end) {
        // will use on-disk checksum here since the end is a stable chunk
        end = tmpLen;
      } else if (chunkChecksum != null) {
        // last chunk is changing. flag that we need to use in-memory checksum
        this.lastChunkChecksum = chunkChecksum;
      }
    }
    endOffset = end;

    // seek to the right offsets
    if (offset > 0 && checksumIn != null) {
      long checksumSkip = (offset / chunkSize) * checksumSize;
      // note blockInStream is seeked when created below
      if (checksumSkip > 0) {
        // Should we use seek() for checksum file as well?
```

```
                IOUtils.skipFully(checksumIn, checksumSkip);
            }
        }
        seqno = 0;

        if (DataNode.LOG.isDebugEnabled()) {
            DataNode.LOG.debug("replica=" + replica);
        }
        // 打开 Block 输入流
        blockIn = datanode.data.getBlockInputStream(block, offset); // seek to offset
        ris = new ReplicaInputStreams(
                blockIn, checksumIn, volumeRef, fileIoProvider);
    } catch (IOException ioe) {
        IOUtils.closeStream(this);
        org.apache.commons.io.IOUtils.closeQuietly(blockIn);
        org.apache.commons.io.IOUtils.closeQuietly(checksumIn);
        throw ioe;
    }
}
```

2.3.3.2 发送 Packet

```
/**
     * sendBlock() is used to read block and its metadata and stream the data to
     * either a client or to another datanode.
     *
     * @param out    stream to which the block is written to
     * @param baseStream optional. if non-null, <code>out</code> is assumed to
     *                   be a wrapper over this stream. This enables optimizations for
     *                   sending the data, e.g.
     *                   {@link SocketOutputStream#transferToFully(FileChannel,
     *                   long, int)}.
     * @param throttler for sending data.
     * @return total bytes read, including checksum data.
     */
    long sendBlock(DataOutputStream out, OutputStream baseStream,
                   DataTransferThrottler throttler) throws IOException {
        final TraceScope scope = datanode.getTracer().
                newScope("sendBlock_" + block.getBlockId());
        try {
            // 往下追
```

```
            return doSendBlock(out, baseStream, throttler);
        } finally {
            scope.close();
        }
    }
```

```
private long doSendBlock(DataOutputStream out, OutputStream baseStream,
                DataTransferThrottler throttler) throws IOException {
        if (out == null) {
            throw new IOException( "out stream is null" );
        }
        initialOffset = offset;
        long totalRead = 0;
        OutputStream streamForSendChunks = out;

        lastCacheDropOffset = initialOffset;

        if (isLongRead() && ris.getDataInFd() != null) {
            // Advise that this file descriptor will be accessed sequentially.
            ris.dropCacheBehindReads(block.getBlockName(), 0, 0,
                    POSIX_FADV_SEQUENTIAL);
        }

        // Trigger readahead of beginning of file if configured.
        manageOsCache();

        final long startTime = ClientTraceLog.isDebugEnabled() ? System.nanoTime() : 0;
        try {
            int maxChunksPerPacket;
            int pktBufSize = PacketHeader.PKT_MAX_HEADER_LEN;
            boolean transferTo = transferToAllowed && !verifyChecksum
                    && baseStream instanceof SocketOutputStream
                    && ris.getDataIn() instanceof FileInputStream;
            if (transferTo) {
                FileChannel fileChannel =
                        ((FileInputStream)ris.getDataIn()).getChannel();
                blockInPosition = fileChannel.position();
                streamForSendChunks = baseStream;
                maxChunksPerPacket = numberOfChunks(TRANSFERTO_BUFFER_SIZE);

                // Smaller packet size to only hold checksum when doing transferTo
                pktBufSize += checksumSize * maxChunksPerPacket;
            } else {
                maxChunksPerPacket = Math.max(1,
```

```
                numberOfChunks(IO_FILE_BUFFER_SIZE));
            // Packet size includes both checksum and data
            pktBufSize += (chunkSize + checksumSize) * maxChunksPerPacket;
        }

        ByteBuffer pktBuf = ByteBuffer.allocate(pktBufSize);

        while (endOffset > offset && !Thread.currentThread().isInterrupted()) {
            manageOsCache();
            // 发送 Packet
            long len = sendPacket(pktBuf, maxChunksPerPacket, streamForSendChunks,
                transferTo, throttler);
            offset += len;
            totalRead += len + (numberOfChunks(len) * checksumSize);
            seqno++;
        }
        // If this thread was interrupted, then it did not send the full block.
        if (!Thread.currentThread().isInterrupted()) {
            try {
                // send an empty packet to mark the end of the block
                sendPacket(pktBuf, maxChunksPerPacket, streamForSendChunks, transferTo,
                    throttler);
                out.flush();
            } catch (IOException e) { //socket error
                throw ioeToSocketException(e);
            }

            sentEntireByteRange = true;
        }
    } finally {
        if ((clientTraceFmt != null) && ClientTraceLog.isDebugEnabled()) {
            final long endTime = System.nanoTime();
            ClientTraceLog.debug(String.format(clientTraceFmt, totalRead,
                initialOffset, endTime - startTime));
        }
        close();
    }
    return totalRead;
}
```

```
/**
     * Sends a packet with up to maxChunks chunks of data.
     *
     * @param pkt buffer used for writing packet data
```

```
   * @param maxChunks maximum number of chunks to send
   * @param out stream to send data to
   * @param transferTo use transferTo to send data
   * @param throttler used for throttling data transfer bandwidth
   */
  private int sendPacket(ByteBuffer pkt, int maxChunks, OutputStream out,
      boolean transferTo, DataTransferThrottler throttler) throws IOException {
    int dataLen = (int) Math.min(endOffset - offset,
                                 (chunkSize * (long) maxChunks));

    int numChunks = numberOfChunks(dataLen); // Number of chunks be sent in the packet
    int checksumDataLen = numChunks * checksumSize;
    int packetLen = dataLen + checksumDataLen + 4;
    boolean lastDataPacket = offset + dataLen == endOffset && dataLen > 0;

    // The packet buffer is organized as follows:
    // _______HHHHCCCCD?D?D?D?
    //              ^     ^
    //              |     \ checksumOff
    //              \ headerOff
    // _ padding, since the header is variable-length
    // H = header and length prefixes
    // C = checksums
    // D? = data, if transferTo is false.

    int headerLen = writePacketHeader(pkt, dataLen, packetLen);

    // Per above, the header doesn't start at the beginning of the
    // buffer
    int headerOff = pkt.position() - headerLen;

    int checksumOff = pkt.position();
    byte[] buf = pkt.array();

    if (checksumSize > 0 && ris.getChecksumIn() != null) {
      readChecksum(buf, checksumOff, checksumDataLen);

      // write in progress that we need to use to get last checksum
      if (lastDataPacket && lastChunkChecksum != null) {
        int start = checksumOff + checksumDataLen - checksumSize;
        byte[] updatedChecksum = lastChunkChecksum.getChecksum();
        if (updatedChecksum != null) {
          System.arraycopy(updatedChecksum, 0, buf, start, checksumSize);
        }
```

```
        }
    }

    int dataOff = checksumOff + checksumDataLen;
    if (!transferTo) { // normal transfer
        try {
            // 读取 Packet
            ris.readDataFully(buf, dataOff, dataLen);
        } catch (IOException ioe) {
            if (ioe.getMessage().startsWith(EIO_ERROR)) {
                throw new DiskFileCorruptException("A disk IO error occurred", ioe);
            }
            throw ioe;
        }

        if (verifyChecksum) {
            verifyChecksum(buf, dataOff, dataLen, numChunks, checksumOff);
        }
    }

    try {
        if (transferTo) {
            // Socket 输出流
            SocketOutputStream sockOut = (SocketOutputStream)out;
            // First write header and checksums
            // 发送数据头部
            sockOut.write(buf, headerOff, dataOff - headerOff);

            // no need to flush since we know out is not a buffered stream
            FileChannel fileCh = ((FileInputStream)ris.getDataIn()).getChannel();
            LongWritable waitTime = new LongWritable();
            LongWritable transferTime = new LongWritable();
            fileIoProvider.transferToSocketFully(
                    ris.getVolumeRef().getVolume(), sockOut, fileCh, blockInPosition,
                    dataLen, waitTime, transferTime);
            datanode.metrics.addSendDataPacketBlockedOnNetworkNanos(waitTime.get());
            datanode.metrics.addSendDataPacketTransferNanos(transferTime.get());
            blockInPosition += dataLen;
        } else {
            // normal transfer
            // 正常情况下传输
            out.write(buf, headerOff, dataOff + dataLen - headerOff);
        }
    } catch (IOException e) {
```

```
        if (e instanceof SocketTimeoutException) {
          /*
           * writing to client timed out.    This happens if the client reads
           * part of a block and then decides not to read the rest (but leaves
           * the socket open).
           *
           * Reporting of this case is done in DataXceiver#run
           */
        } else {
          /* Exception while writing to the client. Connection closure from
           * the other end is mostly the case and we do not care much about
           * it. But other things can go wrong, especially in transferTo(),
           * which we do not want to ignore.
           *
           * The message parsing below should not be considered as a good
           * coding example. NEVER do it to drive a program logic. NEVER.
           * It was done here because the NIO throws an IOException for EPIPE.
           */
          String ioem = e.getMessage();
          /*
           * If we got an EIO when reading files or transferTo the client socket,
           * it's very likely caused by bad disk track or other file corruptions.
           */
          if (ioem.startsWith(EIO_ERROR)) {
            throw new DiskFileCorruptException("A disk IO error occurred", e);
          }
          if (!ioem.startsWith("Broken pipe") && !ioem.startsWith("Connection reset")) {
            LOG.error("BlockSender.sendChunks() exception: ", e);
            datanode.getBlockScanner().markSuspectBlock(
                ris.getVolumeRef().getVolume().getStorageID(),
                block);
          }
        }
        throw ioeToSocketException(e);
      }

      if (throttler != null) { // rebalancing so throttle
        throttler.throttle(packetLen);
      }

      return dataLen;
    }
```