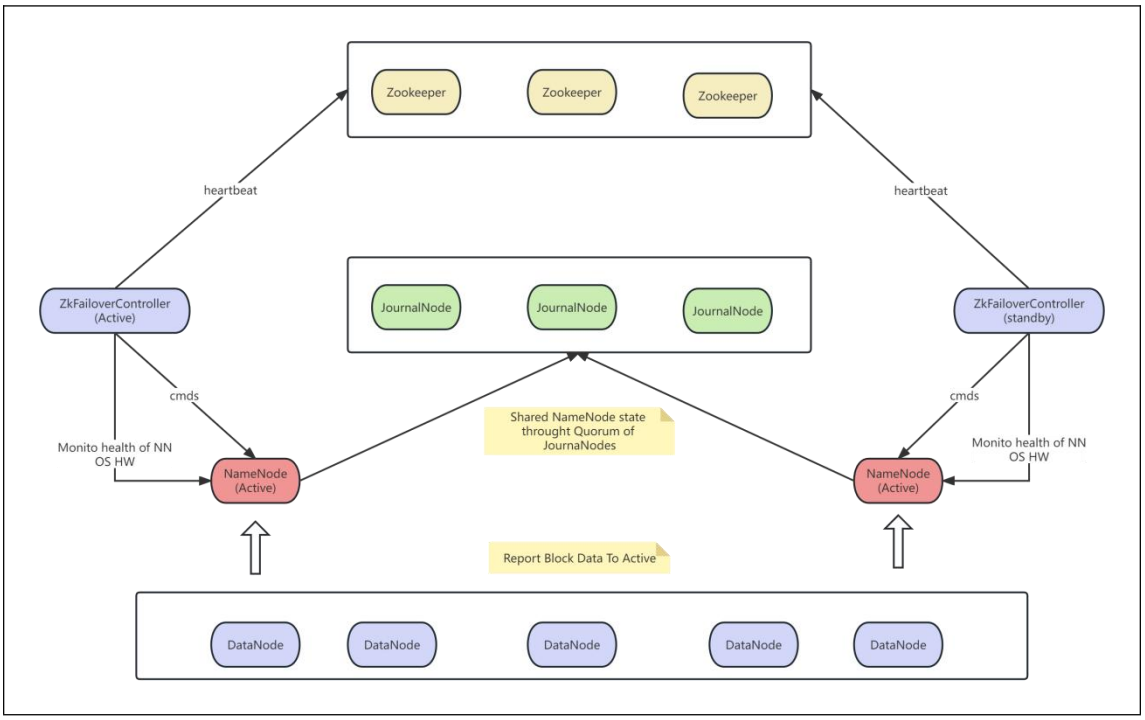
hadoop-hdfs 核心组件启动源码分析

一 hdfs 组件架构



二集群启动脚本分析

2.1 start-dfs.sh 脚本分析

```
# 启动 namenodes
echo "Starting namenodes on [${NAMENODES}]"
hadoop_uservar_su hdfs namenode "${HADOOP_HDFS_HOME}/bin/hdfs" \
        --workers \
        --config "${HADOOP_CONF_DIR}" \
        --hostnames "${NAMENODES}" \
        --daemon start \
        namenode ${nameStartOpt}
```

```
# 启动 datanodes
echo "Starting datanodes"
hadoop_uservar_su hdfs datanode "${HADOOP_HDFS_HOME}/bin/hdfs" \
```

```
        --workers \
        --config "${HADOOP_CONF_DIR}" \
        --daemon start \
        datanode ${dataStartOpt}
```

```
# 启动 journalnode
# quorumjournal nodes (if any)
JOURNAL_NODES=$("${HADOOP_HDFS_HOME}/bin/hdfs" getconf -journalNodes 2>&-)

if [[ "${#JOURNAL_NODES}" != 0 ]]; then
  echo "Starting journal nodes [${JOURNAL_NODES}]"

  hadoop_uservar_su hdfs journalnode "${HADOOP_HDFS_HOME}/bin/hdfs" \
      --workers \
      --config "${HADOOP_CONF_DIR}" \
      --hostnames "${JOURNAL_NODES}" \
      --daemon start \
      journalnode
    (( HADOOP_JUMBO_RETCOUNTER=HADOOP_JUMBO_RETCOUNTER + $? ))
fi
```

```
# 启动 zkfc
# ZK Failover controllers, if auto-HA is enabled
AUTOHA_ENABLED=$("${HADOOP_HDFS_HOME}/bin/hdfs" getconf -confKey
dfs.ha.automatic-failover.enabled | tr '[:upper:]' '[:lower:]')
if [[ "${AUTOHA_ENABLED}" = "true" ]]; then
  echo "Starting ZK Failover Controllers on NN hosts [${NAMENODES}]"

  hadoop_uservar_su hdfs zkfc "${HADOOP_HDFS_HOME}/bin/hdfs" \
      --workers \
      --config "${HADOOP_CONF_DIR}" \
      --hostnames "${NAMENODES}" \
      --daemon start \
      zkfc
  (( HADOOP_JUMBO_RETCOUNTER=HADOOP_JUMBO_RETCOUNTER + $? ))
fi
```

2.2 hdfs.sh 脚本分析

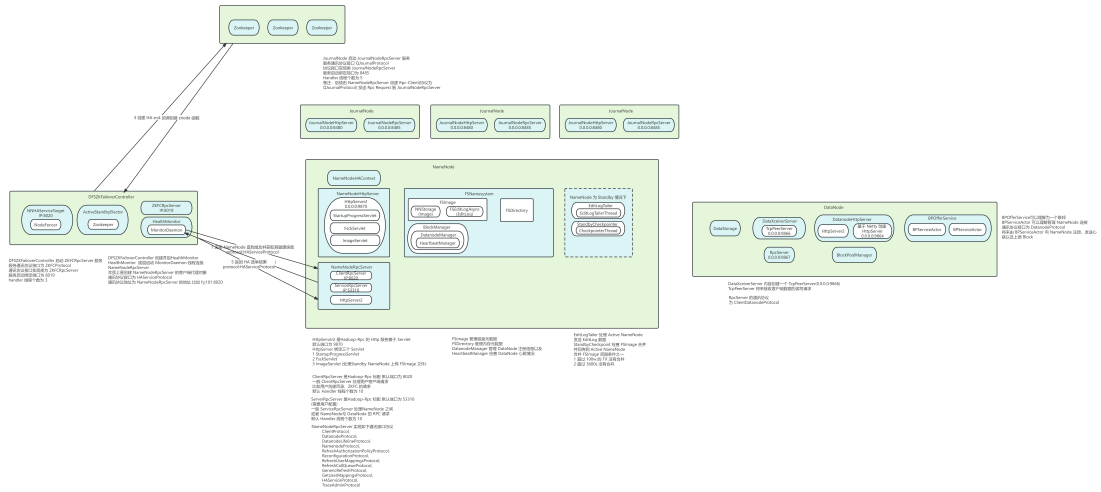
```
function hdfscmd_case
{
  subcmd=$1
```

```
  shift

  case ${subcmd} in
    ......
    # datanode 启动类
    datanode)
      HADOOP_SUBCMD_SUPPORTDAEMONIZATION="true"

HADOOP_SECURE_CLASSNAME="org.apache.hadoop.hdfs.server.datanode.SecureDataNodeStart
er"
      HADOOP_CLASSNAME='org.apache.hadoop.hdfs.server.datanode.DataNode'
      hadoop_deprecate_envvar HADOOP_SECURE_DN_PID_DIR HADOOP_SECURE_PID_DIR
      hadoop_deprecate_envvar HADOOP_SECURE_DN_LOG_DIR HADOOP_SECURE_LOG_DIR
    ;;
    ......
    # journalnode 启动类
    journalnode)
      HADOOP_SUBCMD_SUPPORTDAEMONIZATION="true"
      HADOOP_CLASSNAME='org.apache.hadoop.hdfs.qjournal.server.JournalNode'
    ;;
    ......
    # namenode 启动类
    namenode)
      HADOOP_SUBCMD_SUPPORTDAEMONIZATION="true"
      HADOOP_CLASSNAME='org.apache.hadoop.hdfs.server.namenode.NameNode'
      hadoop_add_param HADOOP_OPTS hdfs.audit.logger
"-Dhdfs.audit.logger=${HDFS_AUDIT_LOGGER}"
    ;;
    ......
    # zkfc 启动类
    zkfc)
      HADOOP_SUBCMD_SUPPORTDAEMONIZATION="true"
      HADOOP_CLASSNAME='org.apache.hadoop.hdfs.tools.DFSZKFailoverController'
    ;;
    *)
    ......
  esac
}
```

三 集群启动组件源码分析



3.1 JournalNode 启动源码分析

主入口类：org.apache.hadoop.hdfs.qjournal.server.JournalNode.main()

```
public static void main(String[] args) throws Exception {
        StringUtils.startupShutdownMessage(JournalNode.class, args, LOG);
        try {
            // 创建 JournalNode 对象并调用其 run()
            System.exit(ToolRunner.run(new JournalNode(), args));
        } catch (Throwable e) {
            LOG.error("Failed to start journalnode.", e);
            terminate(-1, e);
        }
    }
```

```
public static int run(Tool tool, String[] args)
            throws Exception {
        // 往下追
        return run(
                // 对于 JournalNode 默认值 null
                tool.getConf(),
                tool, args);
    }
```

```
public static int run(Configuration conf, Tool tool, String[] args)
            throws Exception {
        if (CallerContext.getCurrent() == null) {
            CallerContext ctx = new CallerContext.Builder("CLI").build();
```

```
                CallerContext.setCurrent(ctx);
            }

            // 对于 JournalNode 启动 conf 默认值 null
            if (conf == null) {
                // 创建 Configuration 对象 并将 core-default.xml core-site.xml 文件名添加到 defaultResources ArrayList 容器中
                conf = new Configuration();
            }
            // 解析入参 args
            GenericOptionsParser parser = new GenericOptionsParser(conf, args);
            // set the configuration back, so that Tool can configure itself
            // 对于 JournalNode 回设 Configuration 对象 底层加载四大文件 (hdfs-default.xml hdfs-site.xml core-default.xml core-site.xml)
            tool.setConf(conf);

            //get the args w/o generic hadoop args
            // 解析 main() 入参
            String[] toolArgs = parser.getRemainingArgs();
            // 对于 JournalNode 调用其 run()
            return tool.run(toolArgs);
        }
```

3.1.1 创建 JournalNode

```
    static {
        // 调用 HdfsConfiguration 静态代码块 添加 hdfs-default.xml hdfs-site.xml 文件名到 defaultResources 容器中
        HdfsConfiguration.init();
    }

    // 调用无参构造函数
```

org.apache.hadoop.hdfs.HdfsConfiguration

```
static {
    addDeprecatedKeys();

    // adds the default resources
    Configuration.addDefaultResource("hdfs-default.xml");
    Configuration.addDefaultResource("hdfs-site.xml");
  }
```

3.1.2 初始化 JournalNode

JournalNode.setConf()

```
@Override
    public void setConf(Configuration conf) {
        this.conf = conf;

        String journalNodeDir = null;
        Collection<String> nameserviceIds;

        // 从四大配置文件获取 key = dfs.internal.nameservices 的 value 默认为空
        // 如果在获取过程中发现没有加载四大文件 则先加载再获取
        nameserviceIds = conf.getTrimmedStringCollection(
                DFSConfigKeys.DFS_INTERNAL_NAMESERVICES_KEY);

        if (nameserviceIds.size() == 0) {
            // 获取 key = dfs.nameservices 的 value
            // 一般情况下 在 hdfs-site.xml 配置对应的值 比如 mycluster
            nameserviceIds = conf.getTrimmedStringCollection(
                    DFSConfigKeys.DFS_NAMESERVICES);
        }

        // if nameservicesIds size is less than 2, it means it is not a federated
        // setup
        if (nameserviceIds.size() < 2) {
            // Check in HA, if journal edit dir is set by appending with nameserviceId
            // 获取 journalNodeDir 元数据存储目录
            for (String nameService : nameserviceIds) {
                journalNodeDir =
conf.get(DFSConfigKeys.DFS_JOURNALNODE_EDITS_DIR_KEY +
                        "." + nameService);
            }
            if (journalNodeDir == null) {
                // 一般情况下 在 hdfs-site.xml 配置 key = dfs.journalnode.edits.dir
                // value = 存储目录 默认值 /tmp/hadoop/dfs/journalnode/
                journalNodeDir =
conf.get(DFSConfigKeys.DFS_JOURNALNODE_EDITS_DIR_KEY,
                        DFSConfigKeys.DFS_JOURNALNODE_EDITS_DIR_DEFAULT);
            }
            // 添加 journalNodeDir 文件路径到 localDir ArrayList 容器
            localDir.add(new File(journalNodeDir.trim()));
        }
```

```
        if (this.tracer == null) {
            this.tracer = new Tracer.Builder("JournalNode").
                    conf(TraceUtils.wrapHadoopConf("journalnode.htrace", conf)).
                    build();
        }
    }
```

3.1.3 启动 JournalNode

JournalNode.run()

```
@Override
    public int run(String[] args) throws Exception {
        // 启动 JournalNode
        start();

        // 阻塞等待线程完成
        return join();
    }
```

```
public void start() throws IOException {
        Preconditions.checkState(!isStarted(), "JN already running");

        try {

            // 检查 JournalNodeEditDir 并创建对应的目录
            for (File journalDir : localDir) {
                validateAndCreateJournalDir(journalDir);
            }

            DefaultMetricsSystem.initialize("JournalNode");
            JvmMetrics.create("JournalNode",
                    conf.get(DFSConfigKeys.DFS_METRICS_SESSION_ID_KEY),
                    DefaultMetricsSystem.instance());

            // 获取 JournalNode Rpc Server 地址
            // 从配置获取 key = dfs.journalnode.rpc-address 默认值 0.0.0.0:8485 (默认配置在 hdfs-default.xml)
            InetSocketAddress socAddr = JournalNodeRpcServer.getAddress(conf);

            SecurityUtil.login(conf, DFSConfigKeys.DFS_JOURNALNODE_KEYTAB_FILE_KEY,
                    DFSConfigKeys.DFS_JOURNALNODE_KERBEROS_PRINCIPAL_KEY,
                    socAddr.getHostName());
```

```
                registerJNMXBean();

                // 创建 JournalNodeHttpServer 并启动
                // 默认 dfs.journalnode.http-address = 0.0.0.0:8480
                httpServer = new JournalNodeHttpServer(conf, this,
                        getHttpServerBindAddress(conf));
                httpServer.start();

                httpServerURI = httpServer.getServerURI().toString();

                // 创建 JournalNodeRpcServer 并启动
                rpcServer = new JournalNodeRpcServer(conf, this);
                rpcServer.start();
            } catch (IOException ioe) {
                //Shutdown JournalNode of JournalNodeRpcServer fails to start
                LOG.error("Failed to start JournalNode.", ioe);
                this.stop(1);
                throw ioe;
            }
        }
```

3.1.3.1 创建 JournalNodeHttpServer 并启动

```
JournalNodeHttpServer(Configuration conf, JournalNode jn,
          InetSocketAddress bindAddress) {
      this.conf = conf;
      this.localJournalNode = jn;
      this.bindAddress = bindAddress;
    }
```

```
void start() throws IOException {
      final InetSocketAddress httpAddr = bindAddress;

      final String httpsAddrString = conf.get(
            DFSConfigKeys.DFS_JOURNALNODE_HTTPS_ADDRESS_KEY,
            DFSConfigKeys.DFS_JOURNALNODE_HTTPS_ADDRESS_DEFAULT);
      InetSocketAddress httpsAddr = NetUtils.createSocketAddr(httpsAddrString);

      if (httpsAddr != null) {
        // If DFS_JOURNALNODE_HTTPS_BIND_HOST_KEY exists then it overrides the
        // host name portion of DFS_NAMENODE_HTTPS_ADDRESS_KEY.
```

```
        final String bindHost =
            conf.getTrimmed(DFSConfigKeys.DFS_JOURNALNODE_HTTPS_BIND_HOST_KEY);
        if (bindHost != null && !bindHost.isEmpty()) {
          httpsAddr = new InetSocketAddress(bindHost, httpsAddr.getPort());
        }
      }

      HttpServer2.Builder builder = DFSUtil.httpServerTemplateForNNAndJN(conf,
          httpAddr, httpsAddr, "journal",
          DFSConfigKeys.DFS_JOURNALNODE_KERBEROS_INTERNAL_SPNEGO_PRINCIPAL_KEY,
          DFSConfigKeys.DFS_JOURNALNODE_KEYTAB_FILE_KEY);

      httpServer = builder.build();
      httpServer.setAttribute(JN_ATTRIBUTE_KEY, localJournalNode);
      httpServer.setAttribute(JspHelper.CURRENT_CONF, conf);
      httpServer.addInternalServlet("getJournal", "/getJournal",
          GetJournalEditServlet.class, true);
      httpServer.start();

      HttpConfig.Policy policy = DFSUtil.getHttpPolicy(conf);
      int connIdx = 0;
      if (policy.isHttpEnabled()) {
        httpAddress = httpServer.getConnectorAddress(connIdx++);
        conf.set(DFSConfigKeys.DFS_JOURNALNODE_HTTP_ADDRESS_KEY,
            NetUtils.getHostPortString(httpAddress));
      }

      if (policy.isHttpsEnabled()) {
        httpsAddress = httpServer.getConnectorAddress(connIdx);
        conf.set(DFSConfigKeys.DFS_JOURNALNODE_HTTPS_ADDRESS_KEY,
            NetUtils.getHostPortString(httpsAddress));
      }
    }
```

3.1.3.2 创建 JournalNodeRpcServer 并启动

```
JournalNodeRpcServer(Configuration conf, JournalNode jn) throws IOException {
        this.jn = jn;

        // 从 JournalNode 配置信息拷贝一份
        Configuration confCopy = new Configuration(conf);
```

```
        // Ensure that nagling doesn't kick in, which could cause latency issues.
        confCopy.setBoolean(
                CommonConfigurationKeysPublic.IPC_SERVER_TCPNODELAY_KEY,
                true);

        // 从配置文件获取 JournalNodeRpcServer 地址
        // 默认值 dfs.journalnode.rpc-address = 0.0.0.0:8485
        InetSocketAddress addr = getAddress(confCopy);
        String bindHost = conf.getTrimmed(DFS_JOURNALNODE_RPC_BIND_HOST_KEY, null);
        if (bindHost == null) {
            bindHost = addr.getHostName();
        }
        LOG.info("RPC server is binding to " + bindHost + ":" + addr.getPort());

        // 设置 Rpc Server 通讯协议 QJournalProtocol (QJournalProtocolPB 接口上注解获取)
        RPC.setProtocolEngine(confCopy, QJournalProtocolPB.class,
                ProtobufRpcEngine.class);
        // QJournalProtocol 协议接口的实现类对象 this = JournalNodeRpcServer
        QJournalProtocolServerSideTranslatorPB translator =
                new QJournalProtocolServerSideTranslatorPB(this);
        BlockingService service = QJournalProtocolService
                .newReflectiveBlockingService(translator);

        // hadoop-rpc 标配
        this.server = new RPC.Builder(confCopy)
                .setProtocol(QJournalProtocolPB.class)
                .setInstance(service)
                .setBindAddress(bindHost)
                .setPort(addr.getPort())
                // 配置 handler 线程个数 默认值 5
                .setNumHandlers(HANDLER_COUNT)
                .setVerbose(false)
                .build();


        // Adding InterQJournalProtocolPB to server
        // 添加 JournalNodeRpcServer 内部的通讯协议 InterQJournalProtocol( JournalNode 与 JournalNode 之间交互)
        InterQJournalProtocolServerSideTranslatorPB
                qJournalProtocolServerSideTranslatorPB = new
                InterQJournalProtocolServerSideTranslatorPB(this);
        BlockingService interQJournalProtocolService = InterQJournalProtocolService
                .newReflectiveBlockingService(qJournalProtocolServerSideTranslatorPB);
```

```
            DFSUtil.addPBProtocol(confCopy, InterQJournalProtocolPB.class,
                    interQJournalProtocolService, server);


            // set service-level authorization security policy
            if (confCopy.getBoolean(
                    CommonConfigurationKeys.HADOOP_SECURITY_AUTHORIZATION, false)) {
                server.refreshServiceAcl(confCopy, new HDFSPolicyProvider());
            }
            this.server.setTracer(jn.tracer);
        }
```

```
        void start() {
            // hadoop-rpc  标配启动
            this.server.start();
        }
```

3.2 DFSZKFailoverController 启动源码分析

主入口类：org.apache.hadoop.hdfs.tools.DFSZKFailoverController

```
public static void main(String args[])
                throws Exception {
            StringUtils.startupShutdownMessage(DFSZKFailoverController.class,
                    args, LOG);
            if (DFSUtil.parseHelpArgument(args,
                    ZKFailoverController.USAGE, System.out, true)) {
                System.exit(0);
            }

            // 解析入参
            GenericOptionsParser parser = new GenericOptionsParser(
                    // 创建  HdfsConfiguration (添加四大文件路径)
                    new HdfsConfiguration(),
                    args);
            try {
                // 创建  DFSZKFailoverController
                DFSZKFailoverController zkfc = DFSZKFailoverController.create(
                        parser.getConfiguration());
                // 运行  DFSZKFailoverController
                System.exit(zkfc.run(parser.getRemainingArgs()));
            } catch (Throwable t) {
```

```
                LOG.fatal("DFSZKFailOverController exiting due to earlier exception "
                        + t);
                terminate(1, t);
            }
        }
```

3.2.1 创建 DFSZKFailoverController

```
public static DFSZKFailoverController create(Configuration conf) {
        // 拷贝 Configuration 并加载四大配置文件
        Configuration localNNConf = DFSHAAdmin.addSecurityConfiguration(conf);
        // 一般情况下 在 hdfs-site.xml 配置对应的值 比如 mycluster
        String nsId = DFSUtil.getNamenodeNameServiceId(conf);

        // 从配置文件判断 NameNode 是否启动 HA 模式
        if (!HAUtil.isHAEnabled(localNNConf, nsId)) {
            throw new HadoopIllegalArgumentException(
                    "HA is not enabled for this namenode.");
        }
        // 判断当前节点是否启动 NameNode 进行判断是否应该启动 ZKFC
        // 如果当前节点需要启动 NameNode 返回值 比如 nn1 nn2
        String nnId = HAUtil.getNameNodeId(localNNConf, nsId);
        if (nnId == null) {
            String msg = "Could not get the namenode ID of this node. " +
                    "You may run zkfc on the node other than namenode.";
            throw new HadoopIllegalArgumentException(msg);
        }

        // 添加或者更新配置
        NameNode.initializeGenericKeys(localNNConf, nsId, nnId);
        DFSUtil.setGenericConf(localNNConf, nsId, nnId, ZKFC_CONF_KEYS);

        // 创建 NNHAServiceTarget
        NNHAServiceTarget localTarget = new NNHAServiceTarget(
                localNNConf, nsId, nnId);
        // 创建 DFSZKFailoverController
        return new DFSZKFailoverController(localNNConf, localTarget);
    }
```

3.2.1.1 创建 NNHAServiceTarget

```
public NNHAServiceTarget(
                Configuration conf,
                String nsId,
                String nnId) {
        Preconditions.checkNotNull(nnId);

        if (nsId == null) {
            nsId = DFSUtil.getOnlyNameServiceIdOrNull(conf);
            if (nsId == null) {
                String errorString = "Unable to determine the name service ID.";
                String[] dfsNames = conf.getStrings(DFS_NAMESERVICES);
                if ((dfsNames != null) && (dfsNames.length > 1)) {
                    errorString = "Unable to determine the name service ID. " +
                            "This is an HA configuration with multiple name services " +
                            "configured. " + DFS_NAMESERVICES + " is set to " +
                            Arrays.toString(dfsNames) + ". Please re-run with the -ns
option.";
                }
                throw new IllegalArgumentException(errorString);
            }
        }
        assert nsId != null;

        // Make a copy of the conf, and override configs based on the
        // target node -- not the node we happen to be running on.
        // 拷贝 Configuration
        HdfsConfiguration targetConf = new HdfsConfiguration(conf);
        NameNode.initializeGenericKeys(targetConf, nsId, nnId);

        // 获取 key = dfs.namenode.rpc-address.mycluster.nn1 对应的 value 比如
hj101:8020
        String serviceAddr =
                DFSUtil.getNamenodeServiceAddr(targetConf, nsId, nnId);
        if (serviceAddr == null) {
            throw new IllegalArgumentException(
                    "Unable to determine service address for namenode '" + nnId + "'");
        }
        // 创建 NameNode 的 InetSocketAddress(hj101,8020)
        this.addr = NetUtils.createSocketAddr(serviceAddr,
                HdfsClientConfigKeys.DFS_NAMENODE_RPC_PORT_DEFAULT);
```

```
        // 默认值 null
        String lifelineAddrStr =
                DFSUtil.getNamenodeLifelineAddr(targetConf, nsId, nnId);
        this.lifelineAddr = (lifelineAddrStr != null) ?
                NetUtils.createSocketAddr(lifelineAddrStr) : null;

        // 获取 key = dfs.ha.automatic-failover.enabled 对应的 value 默认值 false
        // 一般情况下 如果配置了 NameNode HA 在 hdfs-site.xml 需要配置为 true
        this.autoFailoverEnabled = targetConf.getBoolean(
                DFSConfigKeys.DFS_HA_AUTO_FAILOVER_ENABLED_KEY,
                DFSConfigKeys.DFS_HA_AUTO_FAILOVER_ENABLED_DEFAULT);
        if (autoFailoverEnabled) {
            // key = dfs.ha.zkfc.port value = 8019
            int port = DFSZKFailoverController.getZkfcPort(targetConf);
            if (port != 0) {
                // 创建 zkfcAddr = new InetSocketAddress(hj101,8019)
                setZkfcPort(port);
            }
        }

        try {
            // 创建 NodeFencer 对象
            this.fencer = NodeFencer.create(targetConf,
                    DFSConfigKeys.DFS_HA_FENCE_METHODS_KEY);
        } catch (BadFencingConfigurationException e) {
            this.fenceConfigError = e;
        }

        // nn1 nn2
        this.nnId = nnId;
        // mycluster
        this.nsId = nsId;
    }
```

3.2.1.2 创建 DFSZKFailoverController

```
private DFSZKFailoverController(Configuration conf,
                                NNHAServiceTarget localTarget) {
        // 往下追
        super(conf, localTarget);
        // NNHAServiceTarget
        this.localNNTarget = localTarget;
```

```
        // Setup ACLs
        adminAcl = new AccessControlList(
                conf.get(DFSConfigKeys.DFS_ADMIN, " "));
        LOG.info("Failover controller configured for NameNode " +
                localTarget);
    }
```

```
    protected ZKFailoverController(Configuration conf, HAServiceTarget localTarget) {
        // NNHAServiceTarget
        this.localTarget = localTarget;
        this.conf = conf;
    }
```

3.2.2 运行 DFSZKFailoverController

```
public int run(final String[] args) throws Exception {
        // 判断是否配置开启 auto-failover
        // 从 NNHAServiceTarget 的 autoFailoverEnabled 判断
        if (!localTarget.isAutoFailoverEnabled()) {
            LOG.error("Automatic failover is not enabled for " + localTarget + "." +
                    " Please ensure that automatic failover is enabled in the " +
                    "configuration before running the ZK failover controller.");
            return ERR_CODE_AUTO_FAILOVER_NOT_ENABLED;
        }

        loginAsFCUser();
        try {
            return SecurityUtil.doAsLoginUserOrFatal(new PrivilegedAction<Integer>() {
                @Override
                public Integer run() {
                    try {
                        // 往下追
                        return doRun(args);
                    } catch (Exception t) {
                        throw new RuntimeException(t);
                    } finally {
                        if (elector != null) {
                            elector.terminateConnection();
                        }
                    }
                }
            });
        } catch (RuntimeException rte) {
```

```
                throw (Exception) rte.getCause();
            }
        }
```

```
private int doRun(String[] args)
                throws Exception {
            try {
                // 初始化 ZK
                initZK();
            } catch (KeeperException ke) {
                LOG.error("Unable to start failover controller. Unable to connect "
                        + "to ZooKeeper quorum at " + zkQuorum + ". Please check the "
                        + "configured value for " + ZK_QUORUM_KEY + " and ensure that "
                        + "ZooKeeper is running.", ke);
                return ERR_CODE_NO_ZK;
            }
            try {
                if (args.length > 0) {
                    if ("-formatZK".equals(args[0])) {
                        boolean force = false;
                        boolean interactive = true;
                        for (int i = 1; i < args.length; i++) {
                            if ("-force".equals(args[i])) {
                                force = true;
                            } else if ("-nonInteractive".equals(args[i])) {
                                interactive = false;
                            } else {
                                badArg(args[i]);
                            }
                        }
                        // 如果是 -formatZK 操作
                        return formatZK(force, interactive);
                    } else {
                        badArg(args[0]);
                    }
                }
            } catch (Exception e) {
                LOG.error("The failover controller encounters runtime error", e);
                throw e;
            }

            if (!elector.parentZNodeExists()) {
```

```
                LOG.error("Unable to start failover controller. "
                        + "Parent znode does not exist.\n"
                        + "Run with -formatZK flag to initialize ZooKeeper.");
                return ERR_CODE_NO_PARENT_ZNODE;
            }

            try {
                localTarget.checkFencingConfigured();
            } catch (BadFencingConfigurationException e) {
                LOG.error("Fencing is not configured for " + localTarget + ".\n" +
                        "You must configure a fencing method before using automatic " +
                        "failover.", e);
                return ERR_CODE_NO_FENCER;
            }

            try {
                // 初始化 ZKFCRpcServer
                initRPC();
                // 初始化 HealthMonitor 并启动
                initHM();
                // 启动 ZKFCRpcServer
                startRPC();
                // 阻塞线程
                mainLoop();
            } catch (Exception e) {
                LOG.error("The failover controller encounters runtime error: ", e);
                throw e;
            } finally {
                rpcServer.stopAndJoin();

                elector.quitElection(true);
                healthMonitor.shutdown();
                healthMonitor.join();
            }
            return 0;
        }
```

3.2.2.1 初始化 ZK

```
private void initZK() throws HadoopIllegalArgumentException, IOException,
                KeeperException {
        // 获取 ZK 地址
```

```
        // 从配置获取 key = ha.zookeeper.quorum 对应的 value 比如如下
        // hj105:2181,hj106:2181,hj107:2181
        zkQuorum = conf.get(ZK_QUORUM_KEY);

        // 默认值 10 * 1000ms
        int zkTimeout = conf.getInt(ZK_SESSION_TIMEOUT_KEY,
                ZK_SESSION_TIMEOUT_DEFAULT);

        // Parse ACLs from configuration.
        // 默认值 world:anyone:rwcda
        String zkAclConf = conf.get(ZK_ACL_KEY, ZK_ACL_DEFAULT);
        zkAclConf = ZKUtil.resolveConfIndirection(zkAclConf);
        List<ACL> zkAcls = ZKUtil.parseACLs(zkAclConf);
        if (zkAcls.isEmpty()) {
            zkAcls = Ids.CREATOR_ALL_ACL;
        }

        // Parse authentication from configuration.
        List<ZKAuthInfo> zkAuths = SecurityUtil.getZKAuthInfos(conf, ZK_AUTH_KEY);

        // Sanity check configuration.
        Preconditions.checkArgument(zkQuorum != null,
                "Missing required configuration '%s' for ZooKeeper quorum",
                ZK_QUORUM_KEY);
        Preconditions.checkArgument(zkTimeout > 0,
                "Invalid ZK session timeout %s", zkTimeout);

        // 默认值 3
        int maxRetryNum = conf.getInt(
                CommonConfigurationKeys.HA_FC_ELECTOR_ZK_OP_RETRIES_KEY,
                CommonConfigurationKeys.HA_FC_ELECTOR_ZK_OP_RETRIES_DEFAULT);
        // 创建 ActiveStandbyElector
        elector = new ActiveStandbyElector(
                zkQuorum, zkTimeout,
                // /hadoop-ha/mycluster
                getParentZnode(),
                zkAcls, zkAuths,
                // 回调函数
                new ElectorCallbacks(),
                maxRetryNum);
    }
```

3.2.2.1.1 创建 ActiveStandbyElector

```
public ActiveStandbyElector(String zookeeperHostPorts,
                                    int zookeeperSessionTimeout, String parentZnodeName,
List<ACL> acl,
                                    List<ZKAuthInfo> authInfo, ActiveStandbyElectorCallback
app,
                                    int maxRetryNum) throws IOException,
HadoopIllegalArgumentException,
            KeeperException {
        // 往下追
        this(
                zookeeperHostPorts, zookeeperSessionTimeout,
                parentZnodeName, acl,
                authInfo,
                app,
                maxRetryNum, true);
    }
```

```
public ActiveStandbyElector(String zookeeperHostPorts,
                                    int zookeeperSessionTimeout, String parentZnodeName,
List<ACL> acl,
                                    List<ZKAuthInfo> authInfo, ActiveStandbyElectorCallback
app,
                                    int maxRetryNum, boolean failFast) throws IOException,
            HadoopIllegalArgumentException, KeeperException {
        if (app == null || acl == null || parentZnodeName == null
                || zookeeperHostPorts == null || zookeeperSessionTimeout <= 0) {
            throw new HadoopIllegalArgumentException("Invalid argument");
        }
        zkHostPort = zookeeperHostPorts;
        zkSessionTimeout = zookeeperSessionTimeout;
        zkAcl = acl;
        zkAuthInfo = authInfo;
        // ElectorCallbacks
        appClient = app;
        // /hadoop-ha/mycluster
        znodeWorkingDir = parentZnodeName;
        // /hadoop-ha/mycluster/ActiveStandbyElectorLock
        zkLockFilePath = znodeWorkingDir + "/" + LOCK_FILENAME;
        // /hadoop-ha/mycluster/ActiveBreadCrumb
        zkBreadCrumbPath = znodeWorkingDir + "/" + BREADCRUMB_FILENAME;
        this.maxRetryNum = maxRetryNum;

        // establish the ZK Connection for future API calls
```

```
            if (failFast) {
                // 创建与 ZK 连接
                createConnection();
            } else {
                reEstablishSession();
            }
        }
```

3.2.2.1.2 创建 ZK 连接

```
private void createConnection() throws IOException, KeeperException {
            if (zkClient != null) {
                try {
                    zkClient.close();
                } catch (InterruptedException e) {
                    throw new IOException("Interrupted while closing ZK",
                            e);
                }
                zkClient = null;
                watcher = null;
            }
            // 创建与 ZK 连接
            zkClient = connectToZooKeeper();
            if (LOG.isDebugEnabled()) {
                LOG.debug("Created new connection for " + this);
            }
        }
```

```
protected synchronized ZooKeeper connectToZooKeeper() throws IOException,
                KeeperException {

            // Unfortunately, the ZooKeeper constructor connects to ZooKeeper and
            // may trigger the Connected event immediately. So, if we register the
            // watcher after constructing ZooKeeper, we may miss that event. Instead,
            // we construct the watcher first, and have it block any events it receives
            // before we can set its ZooKeeper reference.
            // ZK watch 监听机制对象
            watcher = new WatcherWithClientRef();
            // 创建 ZooKeeper 连接
            ZooKeeper zk = createZooKeeper();
            watcher.setZooKeeperRef(zk);

            // Wait for the asynchronous success/failure. This may throw an exception
            // if we don't connect within the session timeout.
            // 阻塞等待与 ZK 连接
```

```
            watcher.waitForZKConnectionEvent(zkSessionTimeout);

            for (ZKAuthInfo auth : zkAuthInfo) {
                zk.addAuthInfo(auth.getScheme(), auth.getAuth());
            }
            return zk;
        }
```

```
private void waitForZKConnectionEvent(int connectionTimeoutMs)
                    throws KeeperException, IOException {
                try {
                    // 等待连接 ZK 成功
                    // 对于 zkfc 的 ActiveStandbyElector 连接 ZK 成功 则回调
WatcherWithClientRef.process()
                    // 进而 hasReceivedEvent.countdown()
                    if (!hasReceivedEvent.await(connectionTimeoutMs,
TimeUnit.MILLISECONDS)) {
                        LOG.error("Connection timed out: couldn't connect to ZooKeeper in " +
                                "{} milliseconds", connectionTimeoutMs);
                        zk.close();
                        throw KeeperException.create(Code.CONNECTIONLOSS);
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    throw new IOException(
                            "Interrupted when connecting to zookeeper server", e);
                }
            }
```

3.2.2.1.3 ZK 回调函数 (连接 ZK 成功)

```
/**
     * interface implementation of Zookeeper watch events (connection and node),
     * proxied by {@link WatcherWithClientRef}.
     */
    synchronized void processWatchEvent(ZooKeeper zk, WatchedEvent event) {
        // 获取 ZK 监听事件类型
        Event.EventType eventType = event.getType();

        if (isStaleClient(zk)) return;
        if (LOG.isDebugEnabled()) {
            LOG.debug("Watcher event type: " + eventType + " with state:"
                    + event.getState() + " for path:" + event.getPath()
                    + " connectionState: " + zkConnectionState
```

```
                    + " for " + this);
        }

        // 如果是 ZK 连接状态发生变化情况
        if (eventType == Event.EventType.None) {
            // the connection state has changed
            switch (event.getState()) {
                case SyncConnected:
                    LOG.info("Session connected.");
                    // if the listener was asked to move to safe state then it needs to
                    // be undone
                    ConnectionState prevConnectionState = zkConnectionState;
                    zkConnectionState = ConnectionState.CONNECTED;
                    if (prevConnectionState == ConnectionState.DISCONNECTED &&
                            wantToBeInElection) {
                        monitorActiveStatus();
                    }
                    break;
                case Disconnected:
                    LOG.info("Session disconnected. Entering neutral mode...");

                    // ask the app to move to safe state because zookeeper connection
                    // is not active and we dont know our state
                    zkConnectionState = ConnectionState.DISCONNECTED;
                    enterNeutralMode();
                    break;
                case Expired:
                    // the connection got terminated because of session timeout
                    // call listener to reconnect
                    LOG.info("Session expired. Entering neutral mode and rejoining...");
                    enterNeutralMode();
                    reJoinElection(0);
                    break;
                case SaslAuthenticated:
                    LOG.info("Successfully authenticated to ZooKeeper using SASL.");
                    break;
                default:
                    fatalError("Unexpected Zookeeper watch event state: "
                            + event.getState());
                    break;
            }

            return;
        }
```

```
            // a watch on lock path in zookeeper has fired. so something has changed on
            // the lock. ideally we should check that the path is the same as the lock
            // path but trusting zookeeper for now
            // 如果是监听锁路径 znode 发生变化情况
            String path = event.getPath();
            if (path != null) {
                switch (eventType) {
                    case NodeDeleted:
                        if (state == State.ACTIVE) {
                            enterNeutralMode();
                        }
                        joinElectionInternal();
                        break;
                    case NodeDataChanged:
                        monitorActiveStatus();
                        break;
                    default:
                        if (LOG.isDebugEnabled()) {
                            LOG.debug("Unexpected node event: " + eventType + " for path: " +
path);
                        }
                        monitorActiveStatus();
                }

                return;
            }

            // some unexpected error has occurred
            fatalError("Unexpected watch error from Zookeeper");
        }
```

3.2.2.2 初始化 ZKFCRpcServer

```
protected void initRPC() throws IOException {
            // (hj101:8019)
            InetSocketAddress bindAddr = getRpcAddressToBindTo();
            // 创建 ZKFCRpcServer
            rpcServer = new ZKFCRpcServer(conf, bindAddr, this, getPolicyProvider());
        }
```

3.2.2.2.1 创建 ZKFCRpcServer

```
ZKFCRpcServer(Configuration conf,
                        InetSocketAddress bindAddr,
                        ZKFailoverController zkfc,
                        PolicyProvider policy) throws IOException {
        // 赋值 zkfc
        this.zkfc = zkfc;

        // 设置 zkfc 的通讯协议 ZKFCProtocol (从 ZKFCProtocolPB 接口上注解获取)
        RPC.setProtocolEngine(conf, ZKFCProtocolPB.class,
                ProtobufRpcEngine.class);
        // ZKFCProtocolPB 协议接口实现类 this = ZKFCRpcServer
        ZKFCProtocolServerSideTranslatorPB translator =
                new ZKFCProtocolServerSideTranslatorPB(this);
        BlockingService service = ZKFCProtocolService
                .newReflectiveBlockingService(translator);

        // hadoop-rpc 的标配
        this.server = new RPC.Builder(conf)
                .setProtocol(ZKFCProtocolPB.class)
                .setInstance(service)
                .setBindAddress(bindAddr.getHostName())
                .setPort(bindAddr.getPort())
                // 设置 handler 线程个数 3
                .setNumHandlers(HANDLER_COUNT)
                .setVerbose(false).build();

        // set service-level authorization security policy
        if (conf.getBoolean(
                CommonConfigurationKeys.HADOOP_SECURITY_AUTHORIZATION, false)) {
            server.refreshServiceAcl(conf, policy);
        }

    }
```

3.2.2.3 初始化 HealthMonitor 并启动

```
private void initHM() {
        // 创建 HealthMonitor (底层创建 MonitorDaemon 线程)
        healthMonitor = new HealthMonitor(conf, localTarget);
        // HealthMonitor 添加健康检查回调 HealthCallbacks
        healthMonitor.addCallback(
```

```
                    // 创建 HealthCallbacks
                    new HealthCallbacks()
            );
            // HealthMonitor 添加服务状态回调 ServiceStateCallBacks
            healthMonitor.addServiceStateCallback(
                    // 创建 ServiceStateCallBacks
                    new ServiceStateCallBacks()
            );
            // 启动 HealthMonitor (底层启动 MonitorDaemon 线程去连接 NameNode )
            healthMonitor.start();
        }
```

3.2.2.3.1 启动 MonitorDaemon 线程

```
@Override
        public void run() {
            while (shouldRun) {
                try {
                    // 连接 NameNode 直到成功 否则重试
                    loopUntilConnected();
                    // 获取 NameNode 的健康状态
                    doHealthChecks();
                } catch (InterruptedException ie) {
                    Preconditions.checkState(!shouldRun,
                            "Interrupted but still supposed to run");
                }
            }
        }
```

3.2.2.3.2 连接 NameNode 服务直到成功

```
private void loopUntilConnected() throws InterruptedException {
        // zkfc 连接 NameNode
        tryConnect();
        // 如果连接不上 NameNode 每隔 1000ms 重试
        while (proxy == null) {
            Thread.sleep(connectRetryInterval);
            tryConnect();
        }
        assert proxy != null;
    }
```

```
private void tryConnect() {
```

```
            Preconditions.checkState(proxy == null);

            try {
                synchronized (this) {
                    // 返回 HAServiceProtocolClientSideTranslatorPB
                    // ZKFC 创建一个 RPC Client 连接 NameNode 协议接口为
HAServiceProtocol
                    proxy = createProxy();
                }
            } catch (IOException e) {
                LOG.warn("Could not connect to local service at " + targetToMonitor +
                        ": " + e.getMessage());
                proxy = null;
                enterState(State.SERVICE_NOT_RESPONDING);
            }
        }
```

```
/**
     * Connect to the service to be monitored. Stubbed out for easier testing.
     */
    protected HAServiceProtocol createProxy() throws IOException {
        // 返回 HAServiceProtocolClientSideTranslatorPB
        return targetToMonitor.getHealthMonitorProxy(conf, rpcTimeout);
    }
```

```
public HAServiceProtocol getHealthMonitorProxy(Configuration conf,
                                                int timeoutMs) throws
IOException {
            InetSocketAddress addr = getHealthMonitorAddress();
            if (addr == null) {
                // 获取本地 NameNode 地址 比如 hj101:8020
                addr = getAddress();
            }
            // 创建 HAServiceProtocolClientSideTranslatorPB
            return getProxyForAddress(conf, timeoutMs, addr);
        }
```

```
    private HAServiceProtocol getProxyForAddress(Configuration conf,
                                                 int timeoutMs, InetSocketAddress
addr) throws IOException {
            Configuration confCopy = new Configuration(conf);
            // Lower the timeout so we quickly fail to connect
            confCopy.setInt(
```

```
                    CommonConfigurationKeysPublic.IPC_CLIENT_CONNECT_MAX_RETRIES_KEY,
1);
            // 通过反射的方式创建 StandardSocketFactory 对象
            SocketFactory factory = NetUtils.getDefaultSocketFactory(confCopy);
            // 创建 HAServiceProtocolClientSideTranslatorPB
            return new HAServiceProtocolClientSideTranslatorPB(
                        addr,
                        confCopy, factory, timeoutMs);
      }
```

3.2.2.3.2.1 创建 HAServiceProtocolClientSideTranslatorPB

```
    public HAServiceProtocolClientSideTranslatorPB(
                InetSocketAddress addr, Configuration conf,
                SocketFactory socketFactory, int timeout) throws IOException {
            // 通讯协议接口 HAServiceProtocol (从 QJournalProtocolPB 接口上注解获取)
            RPC.setProtocolEngine(conf, HAServiceProtocolPB.class,
                        ProtobufRpcEngine.class);
            // hadoop-rpc client 端的标配操作
            // 获取 HAServiceProtocol 协议接口的代理
            rpcProxy = RPC.getProxy(HAServiceProtocolPB.class,
                        RPC.getProtocolVersion(HAServiceProtocolPB.class),
                        addr,
                        UserGroupInformation.getCurrentUser(), conf, socketFactory, timeout);
      }
```

3.2.2.3.3 获取 NameNode 的健康状态 (异常，没有连接 NameNode 成功)

```
private void doHealthChecks() throws InterruptedException {
            while (shouldRun) {
                  HAServiceStatus status = null;
                  boolean healthy = false;
                  try {
                        // 获取 NameNode 服务状态
                        status = proxy.getServiceStatus();
                        proxy.monitorHealth();
                        healthy = true;
                  } catch (Throwable t) {
                        if (isHealthCheckFailedException(t)) {
                              LOG.warn("Service health check failed for {}", targetToMonitor, t);
                              enterState(State.SERVICE_UNHEALTHY);
                        } else {
                              // 在没有 NameNode 启动成功之前 ZKFC 的 HealthMonitor 还没
```

```
有与 NameNode 连接成功
                        // 因此抛出连接异常
                        LOG.warn("Transport-level exception trying to monitor health of {}",
                                targetToMonitor, t);
                        // 停止代理
                        RPC.stopProxy(proxy);
                        proxy = null;
                        // ZKFC 的 HealthMonitor 进入 SERVICE_NOT_RESPONDING
                        enterState(State.SERVICE_NOT_RESPONDING);
                        Thread.sleep(sleepAfterDisconnectMillis);
                        return;
                    }
                }

                if (status != null) {
                    setLastServiceStatus(status);
                }
                if (healthy) {
                    enterState(State.SERVICE_HEALTHY);
                }

                Thread.sleep(checkIntervalMillis);
            }
        }
```

3.2.2.4 启动 ZKFCRpcServer

```
    protected void startRPC() throws IOException {
        // hadoop-rpc 启动标配操作
        rpcServer.start();
    }
```

3.2.3 等待 NameNode 启动进行通讯

备注：MonitorDaemon 线程一直重试连接 NameNode 直到连接成功
核心代码如下：

```
@Override
        public void run() {
            while (shouldRun) {
                try {
```

```
                    // 连接 NameNode 直到成功 否则重试
                    loopUntilConnected();
                    // 获取 NameNode 的健康状态
                    doHealthChecks();
                } catch (InterruptedException ie) {
                    Preconditions.checkState(!shouldRun,
                            "Interrupted but still supposed to run");
                }
            }
        }
    }
```

3.2.3.1 检查 NameNode 健康状态

```
private void doHealthChecks() throws InterruptedException {
        while (shouldRun) {
            HAServiceStatus status = null;
            boolean healthy = false;
            try {
                // 获取 NameNode 服务状态 (发送 Rpc Request NameNode NameNode 启动之后默认状态为 Standby)
                status = proxy.getServiceStatus();
                // 监控 NameNode 健康状态 (发送 Rpc Request NameNode)
                proxy.monitorHealth();
                // 如果上述两个 Rpc Request 请求成功 表示 ZKFC 正常与 NameNode 通讯
                healthy = true;
            } catch (Throwable t) {
                if (isHealthCheckFailedException(t)) {
                    LOG.warn("Service health check failed for {}", targetToMonitor, t);
                    enterState(State.SERVICE_UNHEALTHY);
                } else {
                    // 在没有 NameNode 启动成功之前 ZKFC 的 HealthMonitor 还没有与 NameNode 连接成功
                    // 因此抛出连接异常
                    LOG.warn("Transport-level exception trying to monitor health of {}",
                            targetToMonitor, t);
                    // 停止代理
                    RPC.stopProxy(proxy);
                    proxy = null;
                    // ZKFC 的 HealthMonitor 进入 SERVICE_NOT_RESPONDING
                    enterState(State.SERVICE_NOT_RESPONDING);
```

```
                    Thread.sleep(sleepAfterDisconnectMillis);
                    return;
                }
            }

            if (status != null) {
                // 更新 ZKFC 获取 NameNode 服务状态 (NameNode 正常启动之后 默认状态为 standby)
                setLastServiceStatus(status);
            }
            if (healthy) {
                // 表示 ZKFC 正常与 NameNode 通讯 ZKFC 进行 SERVICE_HEALTHY
                enterState(State.SERVICE_HEALTHY);
            }

            Thread.sleep(checkIntervalMillis);
        }
    }
```

3.2.3.2 ZKFC 进入 SERVICE_HEALTHY 状态回调 HealthCallbacks.enteredState()

```
/**
     * Callbacks from HealthMonitor
     */
    class HealthCallbacks implements HealthMonitor.Callback {
        @Override
        public void enteredState(HealthMonitor.State newState) {
            // 更新状态 (lastHealthState = State.SERVICE_HEALTHY)
            setLastHealthState(newState);
            // 检查状态
            recheckElectability();
        }
    }
```

```
    protected synchronized void setLastHealthState(HealthMonitor.State newState) {
        LOG.info("Local service " + localTarget +
                " entered state: " + newState);
        lastHealthState = newState;
    }
```

3.2.3.3 ZKFC 将监控的 NameNode 加入 HA 选举

```
/**
     * Check the current state of the service, and join the election
     * if it should be in the election.
     */
    private void recheckElectability() {
        // Maintain lock ordering of elector -> ZKFC
        synchronized (elector) {
            synchronized (this) {
                boolean healthy = lastHealthState == State.SERVICE_HEALTHY;

                long remainingDelay = delayJoiningUntilNanotime - System.nanoTime();
                if (remainingDelay > 0) {
                    if (healthy) {
                        LOG.info("Would have joined master election, but this node is " +
                                "prohibited from doing so for " +
                                TimeUnit.NANOSECONDS.toMillis(remainingDelay) + "
more ms");
                    }
                    scheduleRecheck(remainingDelay);
                    return;
                }

                switch (lastHealthState) {
                    case SERVICE_HEALTHY:
                        // ZKFC 正常与 NameNode 通讯 将该 NameNode 加入 HA 选
举
                        elector.joinElection(
                                // 构建 NameNode HA 选举 znode 锁节点数据 (byte
数据)
                                targetToData(localTarget)
                        );
                        if (quitElectionOnBadState) {
                            quitElectionOnBadState = false;
                        }
                        break;

                    case INITIALIZING:
                        LOG.info("Ensuring that " + localTarget + " does not " +
                                "participate in active master election");
```

```
                                elector.quitElection(false);
                                serviceState = HAServiceState.INITIALIZING;
                                break;

                            case SERVICE_UNHEALTHY:
                            case SERVICE_NOT_RESPONDING:
                                LOG.info("Quitting master election for " + localTarget +
                                        " and marking that fencing is necessary");
                                // NameNode 退出选举
                                elector.quitElection(true);
                                // 服务状态更新为 INITIALIZING
                                serviceState = HAServiceState.INITIALIZING;
                                break;

                            case HEALTH_MONITOR_FAILED:
                                fatalError("Health monitor failed!");
                                break;

                            default:
                                throw new IllegalArgumentException("Unhandled state:"
                                        + lastHealthState);
                        }
                    }
                }
            }
```

```
/**
     * To participate in election, the app will call joinElection. The result will
     * be notified by a callback on either the becomeActive or becomeStandby app
     * interfaces. <br/>
     * After this the elector will automatically monitor the leader status and
     * perform re-election if necessary<br/>
     * The app could potentially start off in standby mode and ignore the
     * becomeStandby call.
     *
     * @param data to be set by the app. non-null data must be set.
     * @throws HadoopIllegalArgumentException if valid data is not supplied
     */
    public synchronized void joinElection(byte[] data)
            throws HadoopIllegalArgumentException {

        if (data == null) {
            throw new HadoopIllegalArgumentException("data cannot be null");
        }
```

```
        if (wantToBeInElection) {
            LOG.info("Already in election. Not re-connecting.");
            return;
        }

        // NameNode HA znode 锁节点数据
        appData = new byte[data.length];
        System.arraycopy(data, 0, appData, 0, data.length);

        if (LOG.isDebugEnabled()) {
            LOG.debug("Attempting active election for " + this);
        }
        // 将 ZKFC 监控的 NameNode 加入 HA 选举
        joinElectionInternal();
    }
```

```
private void joinElectionInternal() {
        Preconditions.checkState(appData != null,
                "trying to join election without any app data");
        if (zkClient == null) {
            if (!reEstablishSession()) {
                fatalError("Failed to reEstablish connection with ZooKeeper");
                return;
            }
        }

        createRetryCount = 0;
        // 表示 ZKFC 对应的 NameNode 进入 HA 选举
        wantToBeInElection = true;
        // ZKFC 的 ZK Client 尝试创建 NameNode HA znode 锁临时节点
        createLockNodeAsync();
    }
```

```
    private void createLockNodeAsync() {
        // 往 ZK 创建 znode 临时锁节点(/hadoop-ha/mycluster/ActiveStandbyElectorLock)
        // 无论是否创建成功 将回调 ActiveStandbyElector.processResult(int,string,object,string)
        zkClient.create(zkLockFilePath, appData, zkAcl, CreateMode.EPHEMERAL,
                this, zkClient);
    }
```

3.2.3.4 ZKFC 创建 HA znode 锁临时节点回调函数

回调 ActiveStandbyElector.processResult(int,string,object,string) 方法

```
 /**
       * interface implementation of Zookeeper callback for create
       */
      @Override
      public synchronized void processResult(int rc, String path, Object ctx,
                                                    String name) {
            if (isStaleClient(ctx)) return;
            if (LOG.isDebugEnabled()) {
                  LOG.debug("CreateNode result: " + rc + " for path: " + path
                              + " connectionState: " + zkConnectionState +
                              "    for " + this);
            }

            Code code = Code.get(rc);
            // ZKFC 的 zk client 创建 NameNode HA znode 临时锁节点成功
            // 说明 ZKFC 监控的 NameNode 成为 active
            if (isSuccess(code)) {
                  // we successfully created the znode. we are the leader. start monitoring
                  // ZKFC 监控的 NameNode 成为 active
                  if (becomeActive()) {
                        // 进行监控 HA znode 锁临时节点 (底层还是创建 znode 锁临时节点)
                        monitorActiveStatus();
                  } else {
                        // 如果 NameNode 状态转为 active 状态失败 重新加入 HA znode 选
举
                        reJoinElectionAfterFailureToBecomeActive();
                  }
                  return;
            }

            // 如果 HA znode 锁临时节点已经存在
            if (isNodeExists(code)) {
                  if (createRetryCount == 0) {
                        // znode exists and we did not retry the operation. so a different
                        // instance has created it. become standby and monitor lock.
                        // ZKFC 监控的 NameNode 成为 standby
                        becomeStandby();
                  }
                  // if we had retried then the znode could have been created by our first
                  // attempt to the server (that we lost) and this node exists response is
```

```
                // for the second attempt. verify this case via ephemeral node owner. this
                // will happen on the callback for monitoring the lock.
                // 进行监控 HA znode 锁临时节点 (底层还是创建 znode 锁临时节点)
                monitorActiveStatus();
                return;
            }

            String errorMessage = "Received create error from Zookeeper. code:"
                    + code.toString() + " for path " + path;
            LOG.debug(errorMessage);

            // HA znode 节点既不是 ZkFC 创建 也不存在 可能出现 ZK 连接失败或者 ZK Session 超时需要重试
            // 创建 HA znode 锁临时节点 (默认重试三次)
            if (shouldRetry(code)) {
                if (createRetryCount < maxRetryNum) {
                    LOG.debug("Retrying createNode createRetryCount: " + createRetryCount);
                    ++createRetryCount;
                    createLockNodeAsync();
                    return;
                }
                errorMessage = errorMessage
                        + ". Not retrying further znode create connection errors.";
            } else if (isSessionExpired(code)) {
                // This isn't fatal - the client Watcher will re-join the election
                LOG.warn("Lock acquisition failed because session was lost");
                return;
            }

            fatalError(errorMessage);
        }
```

3.2.3.4.1 ZKFC 监控的 NameNode 成为 active

```
private boolean becomeActive() {
            assert wantToBeInElection;
            if (state == State.ACTIVE) {
                // already active
                // 如果 NameNode 已经是 active 状态直接返回
                return true;
            }
            try {
                // fence 机制 判断 znode(/hadoop-ha/mycluster/ActiveBreadCrumb) 是否存在数据
```

```
                Stat oldBreadcrumbStat = fenceOldActive();
                // 创建或者更新 znode(/hadoop-ha/mycluster/ActiveBreadCrumb)
                writeBreadCrumbNode(oldBreadcrumbStat);

                LOG.debug("Becoming active for {}", this);

                // ZKFC 监控的 NameNode 成为 active
                appClient.becomeActive();
                // ZKFC 状态变为 ACTIVE
                state = State.ACTIVE;
                return true;
            } catch (Exception e) {
                LOG.warn("Exception handling the winning of election", e);
                // Caller will handle quitting and rejoining the election.
                return false;
            }
        }
```

3.2.3.4.1.1 fence 机制

```
/**
     * If there is a breadcrumb node indicating that another node may need
     * fencing, try to fence that node.
     *
     * @return the Stat of the breadcrumb node that was read, or null
     * if no breadcrumb node existed
     */
    private Stat fenceOldActive() throws InterruptedException, KeeperException {
        final Stat stat = new Stat();
        byte[] data;
        LOG.info("Checking for any old active which needs to be fenced...");
        try {
            data = zkDoWithRetries(new ZKAction<byte[]>() {
                @Override
                public byte[] run() throws KeeperException, InterruptedException {
                    // 获取 znode(/hadoop-ha/mycluster/ActiveBreadCrumb) 节点数据
                    return zkClient.getData(zkBreadCrumbPath, false, stat);
                }
            });
        } catch (KeeperException ke) {
            if (isNodeDoesNotExist(ke.code())) {
                LOG.info("No old node to fence");
                return null;
            }
```

```
                // If we failed to read for any other reason, then likely we lost
                // our session, or we don't have permissions, etc. In any case,
                // we probably shouldn't become active, and failing the whole
                // thing is the best bet.
                throw ke;
            }

            LOG.info("Old node exists: {}", StringUtils.byteToHexString(data));
            if (Arrays.equals(data, appData)) {
                LOG.info("But old node has our own data, so don't need to fence it.");
            } else {
                // 底层将根据 data 数据解析哪个 NameNode 处于 active 需要将该
                // NameNode 由 active 转为 standby (根据 data 数据区获取 NameNode 的
host + port
                // 然后通过创建一个 Rpc Client 发送一个 Rpc Request 给该 NameNode
                // 最终调用 NameNodeRpcServer.transitionToStandby())
                appClient.fenceOldActive(data);
            }
            return stat;
        }
```

```
/**
     * Write the "ActiveBreadCrumb" node, indicating that this node may need
     * to be fenced on failover.
     *
     * @param oldBreadcrumbStat
     */
    private void writeBreadCrumbNode(Stat oldBreadcrumbStat)
            throws KeeperException, InterruptedException {
        Preconditions.checkState(appData != null, "no appdata");

        LOG.info("Writing znode {} to indicate that the local " +
                "node is the most recent active...", zkBreadCrumbPath);
        if (oldBreadcrumbStat == null) {
            // No previous active, just create the node
            // 如果之前没有 NameNode active 则创建
znode(/hadoop-ha/mycluster/ActiveBreadCrumb)
            createWithRetries(zkBreadCrumbPath, appData, zkAcl,
                    CreateMode.PERSISTENT);
        } else {
            // There was a previous active, update the node
            setDataWithRetries(zkBreadCrumbPath, appData,
oldBreadcrumbStat.getVersion());
```

```
        }
    }
```

3.2.3.4.1.2 创建 RpcClient 发送 Rpc Request 到 NameNode 让其成为 active

org.apache.hadoop.ha.ZKFailoverController.ElectorCallbacks.becomeActive()

```
@Override
        public void becomeActive() throws ServiceFailedException {
            ZKFailoverController.this.becomeActive();
        }
```

```
private synchronized void becomeActive() throws ServiceFailedException {
        LOG.info("Trying to make " + localTarget + " active...");
        try {
            // 调用 NameNodeRpcServer.transitionToActive()
            HAServiceProtocolHelper.transitionToActive(
                    // 获取 NameNode 的代理客户端 (通讯协议为 HAServiceProtocol)
                    localTarget.getProxy(conf,
FailoverController.getRpcTimeoutToNewActive(conf)),
                    // 创建 Rpc Request 请求参数 StateChangeRequestInfo
                    createReqInfo());
            String msg = "Successfully transitioned " + localTarget +
                    " to active state";
            LOG.info(msg);
            serviceState = HAServiceState.ACTIVE;
            recordActiveAttempt(new ActiveAttemptRecord(true, msg));

        } catch (Throwable t) {
            String msg = "Couldn't make " + localTarget + " active";
            LOG.error(msg, t);

            recordActiveAttempt(new ActiveAttemptRecord(false, msg + "\n" +
                    StringUtils.stringifyException(t)));

            if (t instanceof ServiceFailedException) {
                throw (ServiceFailedException) t;
            } else {
                throw new ServiceFailedException("Couldn't transition to active",
                        t);
            }
            /*
              * TODO:
```

```
                * we need to make sure that if we get fenced and then quickly restarted,
                * none of these calls will retry across the restart boundary
                * perhaps the solution is that, whenever the nn starts, it gets a unique
                * ID, and when we start becoming active, we record it, and then any future
                * calls use the same ID
                */

            }
        }
```

3.2.3.4.1.3 最终调用 NameNodeRpcServer.transitionToActive()

```
    @Override // HAServiceProtocol
    public synchronized void transitionToActive(StateChangeRequestInfo req)
            throws ServiceFailedException, AccessControlException, IOException {
        checkNNStartup();
        nn.checkHaStateChange(req);
        nn.transitionToActive();
    }
```

3.2.3.4.2 ZKFC 监控的 NameNode 成为 standby

```
    private void becomeStandby() {
            if (state != State.STANDBY) {
                LOG.debug("Becoming standby for {}", this);
                // ZKFC 监控 NameNode 的状态修改为  STANDBY
                state = State.STANDBY;
                //  往下追
                appClient.becomeStandby();
            }
        }
```

org.apache.hadoop.ha.ZKFailoverController.ElectorCallbacks

```
@Override
            public void becomeStandby() {
                ZKFailoverController.this.becomeStandby();
            }
```

3.2.3.4.2.1 创建 RpcClient 发送 Rpc Request 到 NameNode 让其成为 standby

```
private synchronized void becomeStandby() {
            LOG.info("ZK Election indicated that " + localTarget +
                    " should become standby");
```

```
            try {
                // 默认 5000ms
                int timeout = FailoverController.getGracefulFenceTimeout(conf);
                // 获取 NameNode 的代理客户端 (通讯协议为 HAServiceProtocol)
                localTarget.getProxy(conf, timeout)
                        // 调用 NameNodeRpcServer.transitionToStandby()
                        .transitionToStandby(
                                // 创建 Rpc Request 请求参数 StateChangeRequestInfo
                                createReqInfo()
                        );
                LOG.info("Successfully transitioned " + localTarget +
                        " to standby state");
            } catch (Exception e) {
                LOG.error("Couldn't transition " + localTarget + " to standby state",
                        e);
                // TODO handle this. It's a likely case since we probably got fenced
                // at the same time.
            }
            // 状态修改
            serviceState = HAServiceState.STANDBY;
        }
```

3.2.3.4.2.2 最终调用 NameNodeRpcServer.transitionToStandby

```
    @Override // HAServiceProtocol
    public synchronized void transitionToStandby(StateChangeRequestInfo req)
            throws ServiceFailedException, AccessControlException, IOException {
        checkNNStartup();
        nn.checkHaStateChange(req);
        nn.transitionToStandby();
    }
```

3.3 NameNode 启动源码分析

主入口类：org.apache.hadoop.hdfs.server.namenode.NameNode

```
public static void main(String argv[]) throws Exception {
            if (DFSUtil.parseHelpArgument(argv, NameNode.USAGE, System.out, true)) {
                System.exit(0);
            }

            try {
```

```
                StringUtils.startupShutdownMessage(NameNode.class, argv, LOG);
                // 创建并启动 NameNode
                NameNode namenode = createNameNode(argv, null);
                if (namenode != null) {
                    // NameNode 的 NameNodeRpcServer 的 clientRpcServer/serviceRpcServer 阻塞
                    namenode.join();
                }
            } catch (Throwable e) {
                LOG.error("Failed to start namenode.", e);
                terminate(1, e);
            }
        }
```

```
public static NameNode createNameNode(String argv[], Configuration conf)
                throws IOException {
            LOG.info("createNameNode " + Arrays.asList(argv));
            if (conf == null)
                // 创建配置 HdfsConfiguration 对象 (四大文件路径的添加)
                conf = new HdfsConfiguration();

            // Parse out some generic args into Configuration.
            // 解析 NameNode 启动入参
            GenericOptionsParser hParser = new GenericOptionsParser(conf, argv);
            argv = hParser.getRemainingArgs();

            // Parse the rest, NN specific args.
            // 根据启动入参判断启动模式 一般情况下返回 regular (常规的)
            StartupOption startOpt = parseArguments(argv);
            if (startOpt == null) {
                printUsage(System.err);
                return null;
            }

            // 往配置设置启动模式
            setStartupOption(conf, startOpt);

            boolean aborted = false;
            switch (startOpt) {
                case FORMAT:
                    aborted = format(conf, startOpt.getForceFormat(),
                            startOpt.getInteractiveFormat());
                    terminate(aborted ? 1 : 0);
                    return null; // avoid javac warning
```

```
            case GENCLUSTERID:
                System.err.println("Generating new cluster id:");
                System.out.println(NNStorage.newClusterID());
                terminate(0);
                return null;
            case ROLLBACK:
                aborted = doRollback(conf, true);
                terminate(aborted ? 1 : 0);
                return null; // avoid warning
            case BOOTSTRAPSTANDBY:
                String[] toolArgs = Arrays.copyOfRange(argv, 1, argv.length);
                int rc = BootstrapStandby.run(toolArgs, conf);
                terminate(rc);
                return null; // avoid warning
            case INITIALIZESHAREDEDITS:
                aborted = initializeSharedEdits(conf,
                        startOpt.getForceFormat(),
                        startOpt.getInteractiveFormat());
                terminate(aborted ? 1 : 0);
                return null; // avoid warning
            case BACKUP:
            case CHECKPOINT:
                NamenodeRole role = startOpt.toNodeRole();
                DefaultMetricsSystem.initialize(role.toString().replace(" ", ""));
                return new BackupNode(conf, role);
            case RECOVER:
                NameNode.doRecovery(startOpt, conf);
                return null;
            case METADATAVERSION:
                printMetadataVersion(conf);
                terminate(0);
                return null; // avoid javac warning
            case UPGRADEONLY:
                DefaultMetricsSystem.initialize("NameNode");
                new NameNode(conf);
                terminate(0);
                return null;
            default:
                DefaultMetricsSystem.initialize("NameNode");
                // 创建 NameNode
                return new NameNode(conf);
        }
    }
```

3.3.1 创建 NameNode

```
    public NameNode(Configuration conf) throws IOException {
        // 往下追
        this(conf, NamenodeRole.NAMENODE);
    }
```

```
protected NameNode(Configuration conf, NamenodeRole role)
        throws IOException {
    // 往下追 (设置 conf 属性)
    super(conf);

    this.tracer = new Tracer.Builder("NameNode").
            conf(TraceUtils.wrapHadoopConf(NAMENODE_HTRACE_PREFIX, conf)).
            build();
    this.tracerConfigurationManager =
            new TracerConfigurationManager(NAMENODE_HTRACE_PREFIX, conf);

    // 设置角色 默认值 NAMENODE
    this.role = role;

    // 从配置文件 (加载四大配置文件) 获取 NameNode HA 集群 ID 比如 mycluster
    String nsId = getNameServiceId(conf);
    // 获取配置集群当前节点的 NameNode ID 比如 nn1、nn2
    String namenodeId = HAUtil.getNameNodeId(conf, nsId);
    // mycluster
    clientNamenodeAddress = NameNodeUtils.getClientNamenodeAddress(
            conf, nsId);

    if (clientNamenodeAddress != null) {
        LOG.info("Clients should use {} to access"
                + " this namenode/service.", clientNamenodeAddress);
    }
    // 判断是否启动 HA 模式
    this.haEnabled = HAUtil.isHAEnabled(conf, nsId);

    // NameNode 启动默认状态为 STANDBY
    state = createHAState(
            // NameNode 启动模式一般为 StartupOption.REGULAR
            getStartupOption(conf)
    );

    this.allowStaleStandbyReads = HAUtil.shouldAllowStandbyReads(conf);
```

```
            // 创建 NameNodeHAContext
            this.haContext = createHAContext();
            try {
                // 往配置对象添加 HA ID
                // 比如 key = dfs.nameservice.id value = mycluster
                initializeGenericKeys(conf, nsId, namenodeId);

                // 初始化 NameNode
                initialize(getConf());
                try {
                    haContext.writeLock();
                    state.prepareToEnterState(haContext);

                    // NameNode 进入 Standby 状态
                    state.enterState(haContext);
                } finally {
                    haContext.writeUnlock();
                }
            } catch (IOException e) {
                this.stopAtException(e);
                throw e;
            } catch (HadoopIllegalArgumentException e) {
                this.stopAtException(e);
                throw e;
            }
            this.started.set(true);
        }
```

3.3.2 创建 NameNodeHAContext

```
    protected HAContext createHAContext() {
        return new NameNodeHAContext();
    }
```

3.3.3 配置对象添加/更新指定 KV

```
/**
     * In federation configuration is set for a set of
     * namenode and secondary namenode/backup/checkpointer, which are
     * grouped under a logical nameservice ID. The configuration keys specific
```

```
     * to them have suffix set to configured nameserviceId.
     * <p>
     * This method copies the value from specific key of format key.nameserviceId
     * to key, to set up the generic configuration. Once this is done, only
     * generic version of the configuration is read in rest of the code, for
     * backward compatibility and simpler code changes.
     *
     * @param conf                 Configuration object to lookup specific key and to set the value
     *                             to the key passed. Note the conf object is modified
     * @param nameserviceId name service Id (to distinguish federated NNs)
     * @param namenodeId       the namenode ID (to distinguish HA NNs)
     * @see DFSUtil#setGenericConf(Configuration, String, String, String...)
     */
    public static void initializeGenericKeys(Configuration conf,
                                                        String nameserviceId, String namenodeId)
{
        if ((nameserviceId != null && !nameserviceId.isEmpty()) ||
                (namenodeId != null && !namenodeId.isEmpty())) {
            if (nameserviceId != null) {
                // 往配置添加 key = dfs.nameservice.id value = mycluster
                conf.set(DFS_NAMESERVICE_ID, nameserviceId);
            }
            if (namenodeId != null) {
                // 往配置添加 key= dfs.ha.namenode.id value = nn1 (或者 nn2)
                conf.set(DFS_HA_NAMENODE_ID_KEY, namenodeId);
            }

            // 添加各种key拼接key = ${NAMENODE_SPECIFIC_KEYS.nameserviceId.namenodeId}
            // 比如 key = dfs.namenode.rpc-address = hj101:8020
            DFSUtil.setGenericConf(conf, nameserviceId, namenodeId,
                    NAMENODE_SPECIFIC_KEYS);
            // 同上
            DFSUtil.setGenericConf(conf, nameserviceId, null,
                    NAMESERVICE_SPECIFIC_KEYS);
        }

        // If the RPC address is set use it to (re-)configure the default FS
        // 覆盖 fs.defaultFS = hdfs://mycluster 为 fs.defaultFS = hdfs://hj101:8020
        if (conf.get(DFS_NAMENODE_RPC_ADDRESS_KEY) != null) {
            URI defaultUri = URI.create(HdfsConstants.HDFS_URI_SCHEME + "://"
                    + conf.get(DFS_NAMENODE_RPC_ADDRESS_KEY));
            conf.set(FS_DEFAULT_NAME_KEY, defaultUri.toString());
            if (LOG.isDebugEnabled()) {
```

```
                    LOG.debug("Setting " + FS_DEFAULT_NAME_KEY + " to " +
defaultUri.toString());
                }
            }
        }
```

3.3.4 初始化 NameNode

```
/**
     * Initialize name-node.
     *
     * @param conf the configuration
     */
    protected void initialize(Configuration conf) throws IOException {
        if (conf.get(HADOOP_USER_GROUP_METRICS_PERCENTILES_INTERVALS) == null) {
            String intervals = conf.get(DFS_METRICS_PERCENTILES_INTERVALS_KEY);
            if (intervals != null) {
                conf.set(HADOOP_USER_GROUP_METRICS_PERCENTILES_INTERVALS,
                        intervals);
            }
        }

        UserGroupInformation.setConfiguration(conf);
        loginAsNameNodeUser(conf);

        NameNode.initMetrics(conf, this.getRole());
        StartupProgressMetrics.register(startupProgress);

        // JVM 监控
        pauseMonitor = new JvmPauseMonitor();
        pauseMonitor.init(conf);
        pauseMonitor.start();
        metrics.getJvmMetrics().setPauseMonitor(pauseMonitor);

        if (NamenodeRole.NAMENODE == role) {
            // 启动 HttpServer
            startHttpServer(conf);
        }

        // 创建 FSNamesystem 并加载元数据
        loadNamesystem(conf);
```

```
        startAliasMapServerIfNecessary(conf);

        // 创建 NameNodeRpcServer
        rpcServer = createRpcServer(conf);

        initReconfigurableBackoffKey();

        if (clientNamenodeAddress == null) {
            // This is expected for MiniDFSCluster. Set it now using
            // the RPC server's bind address.
            clientNamenodeAddress =
                    NetUtils.getHostPortString(getNameNodeAddress());
            LOG.info("Clients are to use " + clientNamenodeAddress + " to access"
                    + " this namenode/service.");
        }

        // HttpServer2 设置 NameNodeRpcServer 的 clientRpcAddress 地址 (hj101:8020)
        // HttpServer2 设置 FsNamesystem 的 FSImage
        if (NamenodeRole.NAMENODE == role) {
            httpServer.setNameNodeAddress(getNameNodeAddress());
            httpServer.setFSImage(getFSImage());
        }

        // 启动公共组件
        startCommonServices(conf);

        startMetricsLogger(conf);
    }
```

3.3.4.1 启动 HttpServer

```
private void startHttpServer(final Configuration conf) throws IOException {
        // 创建 NameNodeHttpServer
        httpServer = new NameNodeHttpServer(conf,
                this,
                // 从配置获取 key = dfs.namenode.http-address 对应的 value 默认值
0.0.0.0:9870
                getHttpServerBindAddress(conf));
        // 启动 NameNodeHttpServer
        httpServer.start();
        httpServer.setStartupProgress(startupProgress);
    }
```

3.3.4.1.1 创建 NameNodeHttpServer

```
    NameNodeHttpServer(Configuration conf, NameNode nn,
                       InetSocketAddress bindAddress) {
        this.conf = conf;
        this.nn = nn;
        this.bindAddress = bindAddress;
    }
```

3.3.4.1.2 启动 NameNodeHttpServer

```
/**
     * @see DFSUtil#getHttpPolicy(org.apache.hadoop.conf.Configuration)
     * for information related to the different configuration options and
     * Http Policy is decided.
     */
    void start() throws IOException {
        HttpConfig.Policy policy = DFSUtil.getHttpPolicy(conf);
        // 0.0.0.0
        final String infoHost = bindAddress.getHostName();

        final InetSocketAddress httpAddr = bindAddress;
        // 0.0.0.0:9871
        final String httpsAddrString = conf.getTrimmed(
                DFSConfigKeys.DFS_NAMENODE_HTTPS_ADDRESS_KEY,
                DFSConfigKeys.DFS_NAMENODE_HTTPS_ADDRESS_DEFAULT);
        InetSocketAddress httpsAddr = NetUtils.createSocketAddr(httpsAddrString);

        if (httpsAddr != null) {
            // If DFS_NAMENODE_HTTPS_BIND_HOST_KEY exists then it overrides the
            // host name portion of DFS_NAMENODE_HTTPS_ADDRESS_KEY.
            final String bindHost =

conf.getTrimmed(DFSConfigKeys.DFS_NAMENODE_HTTPS_BIND_HOST_KEY);
            if (bindHost != null && !bindHost.isEmpty()) {
                httpsAddr = new InetSocketAddress(bindHost, httpsAddr.getPort());
            }
        }

        // 构建 HttpServer2
        HttpServer2.Builder builder = DFSUtil.httpServerTemplateForNNAndJN(conf,
                httpAddr, httpsAddr, "hdfs",
```

```
DFSConfigKeys.DFS_NAMENODE_KERBEROS_INTERNAL_SPNEGO_PRINCIPAL_KEY,
                DFSConfigKeys.DFS_NAMENODE_KEYTAB_FILE_KEY);
        final boolean xFrameEnabled = conf.getBoolean(
                DFSConfigKeys.DFS_XFRAME_OPTION_ENABLED,
                DFSConfigKeys.DFS_XFRAME_OPTION_ENABLED_DEFAULT);
        final String xFrameOptionValue = conf.getTrimmed(
                DFSConfigKeys.DFS_XFRAME_OPTION_VALUE,
                DFSConfigKeys.DFS_XFRAME_OPTION_VALUE_DEFAULT);
        builder.configureXFrame(xFrameEnabled).setXFrameOption(xFrameOptionValue);
        httpServer = builder.build();

        if (policy.isHttpsEnabled()) {
            // assume same ssl port for all datanodes
            InetSocketAddress                    datanodeSslPort                    =
NetUtils.createSocketAddr(conf.getTrimmed(
                    DFSConfigKeys.DFS_DATANODE_HTTPS_ADDRESS_KEY, infoHost + ":"
                            + DFSConfigKeys.DFS_DATANODE_HTTPS_DEFAULT_PORT));
            httpServer.setAttribute(DFSConfigKeys.DFS_DATANODE_HTTPS_PORT_KEY,
                    datanodeSslPort.getPort());
        }

        // 初始化 NameNode Web (HttpServer2 也即访问: IP:9870)
        initWebHdfs(conf, bindAddress.getHostName(), httpServer,
                NamenodeWebHdfsMethods.class.getPackage().getName());

        httpServer.setAttribute(NAMENODE_ATTRIBUTE_KEY, nn);
        httpServer.setAttribute(JspHelper.CURRENT_CONF, conf);

        // HttpServer2 绑定 Servlets (核心三个 Servlet)
        setupServlets(httpServer, conf);

        // 启动 HttpServer2
        httpServer.start();

        int connIdx = 0;
        if (policy.isHttpEnabled()) {
            httpAddress = httpServer.getConnectorAddress(connIdx++);
            conf.set(DFSConfigKeys.DFS_NAMENODE_HTTP_ADDRESS_KEY,
                    NetUtils.getHostPortString(httpAddress));
        }

        if (policy.isHttpsEnabled()) {
            httpsAddress = httpServer.getConnectorAddress(connIdx);
            conf.set(DFSConfigKeys.DFS_NAMENODE_HTTPS_ADDRESS_KEY,
```

```
                    NetUtils.getHostPortString(httpsAddress));
        }
    }
```

3.3.4.1.2.1 构建 HttpServer2

```
/**
     * Return a HttpServer.Builder that the journalnode / namenode / secondary
     * namenode can use to initialize their HTTP / HTTPS server.
     */
    public static HttpServer2.Builder httpServerTemplateForNNAndJN(
            Configuration conf, final InetSocketAddress httpAddr,
            final InetSocketAddress httpsAddr, String name, String spnegoUserNameKey,
            String spnegoKeytabFileKey) throws IOException {
        HttpConfig.Policy policy = getHttpPolicy(conf);

        // 构建 HttpServer2
        HttpServer2.Builder builder = new HttpServer2.Builder().setName(name)
                .setConf(conf).setACL(new AccessControlList(conf.get(DFS_ADMIN, " ")))
                .setSecurityEnabled(UserGroupInformation.isSecurityEnabled())
                .setUsernameConfKey(spnegoUserNameKey)
                .setKeytabConfKey(getSpnegoKeytabKey(conf, spnegoKeytabFileKey));

        // initialize the webserver for uploading/downloading files.
        if (UserGroupInformation.isSecurityEnabled()) {
            LOG.info("Starting web server as: "
                    + SecurityUtil.getServerPrincipal(conf.get(spnegoUserNameKey),
                    httpAddr.getHostName()));
        }

        if (policy.isHttpEnabled()) {
            if (httpAddr.getPort() == 0) {
                builder.setFindPort(true);
            }

            // 0.0.0.0:9870
            URI uri = URI.create("http://" + NetUtils.getHostPortString(httpAddr));
            builder.addEndpoint(uri);
            LOG.info("Starting Web-server for " + name + " at: " + uri);
        }

        if (policy.isHttpsEnabled() && httpsAddr != null) {
            Configuration sslConf = loadSslConfiguration(conf);
            loadSslConfToHttpServerBuilder(builder, sslConf);
```

```
            if (httpsAddr.getPort() == 0) {
                builder.setFindPort(true);
            }

            // 0.0.0.0:9871
            URI uri = URI.create("https://" + NetUtils.getHostPortString(httpsAddr));
            builder.addEndpoint(uri);
            LOG.info("Starting Web-server for " + name + " at: " + uri);
        }
        return builder;
    }
```

3.3.4.1.2.2 HttpServer2 绑定 Servlets

```
    private static void setupServlets(HttpServer2 httpServer, Configuration conf) {
        // NameNode 的 HttpServer2 绑定 Servlets
        httpServer.addInternalServlet("startupProgress",
                StartupProgressServlet.PATH_SPEC, StartupProgressServlet.class);
        httpServer.addInternalServlet("fsck", "/fsck", FsckServlet.class,
                true);
        httpServer.addInternalServlet("imagetransfer", ImageServlet.PATH_SPEC,
                ImageServlet.class, true);
    }
```

3.3.4.1.2.3 启动 HttpServer2

```
/**
   * Start the server. Does not wait for the server to start.
   */
  public void start() throws IOException {
    try {
      try {
        openListeners();
        webServer.start();
      } catch (IOException ex) {
        LOG.info("HttpServer.start() threw a non Bind IOException", ex);
        throw ex;
      } catch (MultiException ex) {
        LOG.info("HttpServer.start() threw a MultiException", ex);
        throw ex;
      }
```

```
        // Make sure there is no handler failures.
        Handler[] hs = webServer.getHandlers();
        for (Handler handler : hs) {
          if (handler.isFailed()) {
            throw new IOException(
                "Problem in starting http server. Server handlers failed");
          }
        }
        // Make sure there are no errors initializing the context.
        Throwable unavailableException = webAppContext.getUnavailableException();
        if (unavailableException != null) {
          // Have to stop the webserver, or else its non-daemon threads
          // will hang forever.
          webServer.stop();
          throw new IOException("Unable to initialize WebAppContext",
              unavailableException);
        }
      } catch (IOException e) {
        throw e;
      } catch (InterruptedException e) {
        throw (IOException) new InterruptedIOException(
            "Interrupted while starting HTTP server").initCause(e);
      } catch (Exception e) {
        throw new IOException("Problem starting http server", e);
      }
    }
```

3.3.4.2 创建 FSNamesystem 并加载元数据

```
    protected void loadNamesystem(Configuration conf) throws IOException {
        // 从磁盘加载元数据 封装成 FsNamesystem
        this.namesystem = FSNamesystem.loadFromDisk(conf);
    }
```

```
  /**
     * Instantiates an FSNamesystem loaded from the image and edits
     * directories specified in the passed Configuration.
     *
     * @param conf the Configuration which specifies the storage directories
     *             from which to load
     * @return an FSNamesystem which contains the loaded namespace
     * @throws IOException if loading fails
```

```
     */
    static FSNamesystem loadFromDisk(Configuration conf) throws IOException {

        // 检查元数据配置路径
        checkConfiguration(conf);

        // 创建 FSImage
        FSImage fsImage = new FSImage(conf,
                // 从配置获取 key = dfs.namenode.name.dir
                // 一般在 hdfs-site.xml 配置 比如 /opt/app/hadoop-3.1.3/data/name
                //  file:///opt/app/hadoop-3.1.3/data
                FSNamesystem.getNamespaceDirs(conf),
                // 从配置中获取 key = dfs.namenode.shared.edits.dir
                //  一 般 在 hdfs-site.xml 配 置 比 如 qjournal://hj101:8485;hj102:8485;hj103:8485/mycluster
                //  还 有 默 认 的 key = dfs.namenode.edits.dir value = file:///tmp/hadoop/dfs/name
                FSNamesystem.getNamespaceEditsDirs(conf));

        // 创建 FSNamesystem
        FSNamesystem namesystem = new FSNamesystem(conf, fsImage, false);
        // StartupOption.REGULAR
        StartupOption startOpt = NameNode.getStartupOption(conf);
        if (startOpt == StartupOption.RECOVER) {
            namesystem.setSafeMode(SafeModeAction.SAFEMODE_ENTER);
        }

        long loadStart = monotonicNow();
        try {
            // 从磁盘加载元数据
            namesystem.loadFSImage(startOpt);
        } catch (IOException ioe) {
            LOG.warn("Encountered exception loading fsimage", ioe);
            fsImage.close();
            throw ioe;
        }
        long timeTakenToLoadFSImage = monotonicNow() - loadStart;
        LOG.info("Finished loading FSImage in " + timeTakenToLoadFSImage + " msecs");
        NameNodeMetrics nnMetrics = NameNode.getNameNodeMetrics();
        if (nnMetrics != null) {
            nnMetrics.setFsImageLoadTime((int) timeTakenToLoadFSImage);
        }
        // HdfsFileStatus
        namesystem.getFSDirectory().createReservedStatuses(namesystem.getCTime());
```

```
            return namesystem;
        }
```

3.3.4.2.1 检查元数据配置路径

```
/**
         * Check the supplied configuration for correctness.
         *
         * @param conf Supplies the configuration to validate.
         * @throws IOException                          if the configuration could not be queried.
         * @throws IllegalArgumentException if the configuration is invalid.
         */
        private static void checkConfiguration(Configuration conf)
                throws IOException {

            // 从配置中获取 key=dfs.namenode.name.dir 对应的 value
            // 一般在 hdfs-site.xml 文件配置
            final Collection<URI> namespaceDirs =
                    FSNamesystem.getNamespaceDirs(conf);

            // 1 从配置中获取 key = dfs.namenode.shared.edits.dir
            //    1.1    一    般    在    hdfs-site.xml    配    置    比    如
qjournal://hj101:8485;hj102:8485;hj103:8485/mycluster
            // 2  默认返回 key = dfs.namenode.edits.dir value = file:///tmp/hadoop/dfs/name
            final Collection<URI> editsDirs =
                    FSNamesystem.getNamespaceEditsDirs(conf);

            // 1 file:///tmp/hadoop/dfs/name
            // 2 qjournal://hj101:8485;hj102:8485;hj103:8485/mycluster
            final Collection<URI> requiredEditsDirs =
                    FSNamesystem.getRequiredNamespaceEditsDirs(conf);

            // qjournal://hj101:8485;hj102:8485;hj103:8485/mycluster
            final Collection<URI> sharedEditsDirs =
                    FSNamesystem.getSharedEditsDirs(conf);

            for (URI u : requiredEditsDirs) {
                if (u.toString().compareTo(
                        DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_DEFAULT) == 0) {
                    continue;
                }

                // Each required directory must also be in editsDirs or in
                // sharedEditsDirs.
```

```
                if (!editsDirs.contains(u) &&
                        !sharedEditsDirs.contains(u)) {
                    throw new IllegalArgumentException("Required edits directory " + u
                            + " not found: "
                            + DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_KEY + "=" + editsDirs
+ "; "
                            + DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_REQUIRED_KEY
                            + "=" + requiredEditsDirs + "; "
                            + DFSConfigKeys.DFS_NAMENODE_SHARED_EDITS_DIR_KEY
                            + "=" + sharedEditsDirs);
                }
            }

            if (namespaceDirs.size() == 1) {
                LOG.warn("Only one image storage directory ("
                        + DFS_NAMENODE_NAME_DIR_KEY + ") configured. Beware of data
loss"
                        + " due to lack of redundant storage directories!");
            }

            if (editsDirs.size() == 1) {
                LOG.warn("Only one namespace edits storage directory ("
                        + DFS_NAMENODE_EDITS_DIR_KEY + ") configured. Beware of data loss"
                        + " due to lack of redundant storage directories!");
            }
        }
```

3.3.4.2.2 创建 FSImage

```
/**
     * Construct the FSImage. Set the default checkpoint directories.
     *
     * Setup storage and initialize the edit log.
     *
     * @param conf Configuration
     * @param imageDirs Directories the image can be stored in.
     * @param editsDirs Directories the editlog can be stored in.
     * @throws IOException if directories are invalid.
     */
    protected FSImage(Configuration conf,
                        Collection<URI> imageDirs,
                        List<URI> editsDirs)
            throws IOException {
        this.conf = conf;
```

```
    // 创建 NNStorage
    storage = new NNStorage(conf, imageDirs, editsDirs);

    // false
    if(conf.getBoolean(DFSConfigKeys.DFS_NAMENODE_NAME_DIR_RESTORE_KEY,
                       DFSConfigKeys.DFS_NAMENODE_NAME_DIR_RESTORE_DEFAULT)) {
      storage.setRestoreFailedStorage(true);
    }

    // 创建 FSEditLogAsync
    this.editLog = FSEditLog.newInstance(conf, storage, editsDirs);

    // 创建 NNStorageRetentionManager
    archivalManager = new NNStorageRetentionManager(conf, storage, editLog);
  }
```

3.3.4.2.2.1 创建 NNStorage

```
/**
     * Construct the NNStorage.
     * @param conf Namenode configuration.
     * @param imageDirs Directories the image can be stored in.
     * @param editsDirs Directories the editlog can be stored in.
     * @throws IOException if any directories are inaccessible.
     */
    public NNStorage(Configuration conf,
                     Collection<URI> imageDirs,
                     Collection<URI> editsDirs)
        throws IOException {
      // 往下追
      super(NodeType.NAME_NODE);

      // this may modify the editsDirs, so copy before passing in
      // 设置元数据存储目录
      setStorageDirectories(
          // 1
          imageDirs,
          // 2
          Lists.newArrayList(editsDirs),
          // 1
          FSNamesystem.getSharedEditsDirs(conf));

      //Update NameDirSize metric value after NN start
```

```
        // 更新 NameNode 元数据存储目录大小 以便监控
        updateNameDirSize();
    }
```

```
/**
     * Set the storage directories which will be used. This should only ever be
     * called from inside NNStorage. However, it needs to remain package private
     * for testing, as StorageDirectories need to be reinitialised after using
     * Mockito.spy() on this class, as Mockito doesn't work well with inner
     * classes, such as StorageDirectory in this case.
     *
     * Synchronized due to initialization of storageDirs and removedStorageDirs.
     *
     * @param fsNameDirs Locations to store images.
     * @param fsEditsDirs Locations to store edit logs.
     * @throws IOException
     */
    @VisibleForTesting
    synchronized void setStorageDirectories(Collection<URI> fsNameDirs,
                                            Collection<URI> fsEditsDirs,
                                            Collection<URI> sharedEditsDirs)
            throws IOException {
        // 清空集合
        getStorageDirs().clear();
        this.removedStorageDirs.clear();

        // Add all name dirs with appropriate NameNodeDirType
        for (URI dirName : fsNameDirs) {
            checkSchemeConsistency(dirName);
            boolean isAlsoEdits = false;
            for (URI editsDirName : fsEditsDirs) {
                if (editsDirName.compareTo(dirName) == 0) {
                    isAlsoEdits = true;
                    fsEditsDirs.remove(editsDirName);
                    break;
                }
            }
            // 返回 IMAGE_AND_EDITS
            NameNodeDirType dirType = (isAlsoEdits) ?
                    NameNodeDirType.IMAGE_AND_EDITS :
                    NameNodeDirType.IMAGE;

            // Add to the list of storage directories, only if the
            // URI is of type file://
```

```
                if (dirName.getScheme().compareTo("file") == 0) {
                    // 添加元数据存储目录
                    this.addStorageDir(
                            // 创建 StorageDirectory
                            // 一般路径是配置在 hdfs-site.xml key = dfs.namenode.name.dir
                            // value 比如 /opt/app/hadoop-3.1.3/data/name
                            new StorageDirectory(new File(dirName.getPath()), dirType,
                            sharedEditsDirs.contains(dirName)));  // Don't lock the dir if it's
shared.
                }
            }

            // Add edits dirs if they are different from name dirs
            for (URI dirName : fsEditsDirs) {
                checkSchemeConsistency(dirName);
                // Add to the list of storage directories, only if the
                // URI is of type file://
                // 添加元数据存储目录 -> file:///tmp/hadoop/dfs/name
                if (dirName.getScheme().compareTo("file") == 0) {
                    this.addStorageDir(new StorageDirectory(new File(dirName.getPath()),
                            NameNodeDirType.EDITS, sharedEditsDirs.contains(dirName)));
                }
            }
        }
```

3.3.4.2.2.2 创建 FSEditLogAsync

```
static FSEditLog newInstance(Configuration conf, NNStorage storage,
                                            List<URI> editsDirs) {
            // true
            boolean asyncEditLogging = conf.getBoolean(
                    DFSConfigKeys.DFS_NAMENODE_EDITS_ASYNC_LOGGING,
                    DFSConfigKeys.DFS_NAMENODE_EDITS_ASYNC_LOGGING_DEFAULT);
            LOG.info("Edit logging is async:" + asyncEditLogging);

            return asyncEditLogging
                    // 创建 FSEditLogAsync
                    ? new FSEditLogAsync(conf, storage, editsDirs)
                    : new FSEditLog(conf, storage, editsDirs);
        }
```

```
    FSEditLogAsync(Configuration conf, NNStorage storage, List<URI> editsDirs) {
        // 往下追
        super(conf, storage, editsDirs);
```

```
        // op instances cannot be shared due to queuing for background thread.
        cache.disableCache();
    }
```

```
/**
         * Constructor for FSEditLog. Underlying journals are constructed, but
         * no streams are opened until open() is called.
         *
         * @param conf           The namenode configuration
         * @param storage      Storage object used by namenode
         * @param editsDirs List of journals to use
         */
        FSEditLog(Configuration conf, NNStorage storage, List<URI> editsDirs) {
            isSyncRunning = false;

            this.conf = conf;
            this.storage = storage;
            metrics = NameNode.getNameNodeMetrics();
            lastPrintTime = monotonicNow();

            // If this list is empty, an error will be thrown on first use
            // of the editlog, as no journals will exist
            // 1 qjournal://hj101:8485;hj102:8485;hj103:8485/mycluster
            // 2 file:///tmp/hadoop/dfs/name
            this.editsDirs = Lists.newArrayList(editsDirs);

            // 从配置中获取 key = dfs.namenode.shared.edits.dir
            // 一般在hdfs-site.xml配置比如qjournal://hj101:8485;hj102:8485;hj103:8485/mycluster
            this.sharedEditsDirs = FSNamesystem.getSharedEditsDirs(conf);
        }
```

3.3.4.2.2.3 创建 NNStorageRetentionManager

```
    public NNStorageRetentionManager(Configuration conf, NNStorage storage,
            LogsPurgeable purgeableLogs) {
        // 往下追
        this(conf, storage, purgeableLogs, new DeletionStoragePurger());
    }
```

3.3.4.2.3 创建 FSNamesystem

```
/**
```

```
   * Create an FSNamesystem associated with the specified image.
   * <p>
   * Note that this does not load any data off of disk -- if you would
   * like that behavior, use {@link #loadFromDisk(Configuration)}
   *
   * @param conf                     configuration
   * @param fsImage                  The FSImage to associate with
   * @param ignoreRetryCache Whether or not should ignore the retry cache setup
   *                                 step. For Secondary NN this should be set to true.
   * @throws IOException on bad configuration
   */
  FSNamesystem(Configuration conf, FSImage fsImage, boolean ignoreRetryCache)
          throws IOException {
      provider = DFSUtil.createKeyProviderCryptoExtension(conf);
      LOG.info("KeyProvider: " + provider);

      if (conf.getBoolean(DFS_NAMENODE_AUDIT_LOG_ASYNC_KEY,
              DFS_NAMENODE_AUDIT_LOG_ASYNC_DEFAULT)) {
          LOG.info("Enabling async auditlog");
          enableAsyncAuditLog();
      }

      fsLock = new FSNamesystemLock(conf, detailedLockHoldTimeMetrics);
      cond = fsLock.newWriteLockCondition();
      cpLock = new ReentrantLock();

      // fsImage = FSImage (管理 NameNode 磁盘元数据)
      this.fsImage = fsImage;
      try {
          // 资源监控检查间隔 key = dfs.namenode.resource.check.interval
          // 默认值 5000
          resourceRecheckInterval = conf.getLong(
                  DFS_NAMENODE_RESOURCE_CHECK_INTERVAL_KEY,
                  DFS_NAMENODE_RESOURCE_CHECK_INTERVAL_DEFAULT);

          // 获取当前启动 NameNode 用户信息
          this.fsOwner = UserGroupInformation.getCurrentUser();

          // 获取 NameNode DFS 超级用户组 key = dfs.permissions.superusergroup
          // 默认值 supergroup
          this.supergroup = conf.get(DFS_PERMISSIONS_SUPERUSERGROUP_KEY,
                  DFS_PERMISSIONS_SUPERUSERGROUP_DEFAULT);

          // key = dfs.permissions.enabled value = true
```

```
            this.isPermissionEnabled = conf.getBoolean(DFS_PERMISSIONS_ENABLED_KEY,
                    DFS_PERMISSIONS_ENABLED_DEFAULT);

            // key = dfs.namenode.snapshotdiff.listing.limit value = 1000
            this.snapshotDiffReportLimit =
                    conf.getInt(DFS_NAMENODE_SNAPSHOT_DIFF_LISTING_LIMIT,
                            DFS_NAMENODE_SNAPSHOT_DIFF_LISTING_LIMIT_DEFAULT);

            LOG.info("fsOwner                = " + fsOwner);
            LOG.info("supergroup             = " + supergroup);
            LOG.info("isPermissionEnabled = " + isPermissionEnabled);

            // block allocation has to be persisted in HA using a shared edits directory
            // so that the standby has up-to-date namespace information
            // mycluster
            nameserviceId = DFSUtil.getNamenodeNameServiceId(conf);
            // true
            this.haEnabled = HAUtil.isHAEnabled(conf, nameserviceId);

            // Sanity check the HA-related config.
            if (nameserviceId != null) {
                LOG.info("Determined nameservice ID: " + nameserviceId);
            }
            LOG.info("HA Enabled: " + haEnabled);
            if (!haEnabled && HAUtil.usesSharedEditsDir(conf)) {
                LOG.warn("Configured NNs:\n" + DFSUtil.nnAddressesAsString(conf));
                throw new IOException("Invalid configuration: a shared edits dir " +
                        "must not be specified if HA is not enabled.");
            }

            // block manager needs the haEnabled initialized
            //  创建  BlockManager
            this.blockManager = new BlockManager(this, haEnabled, conf);
            this.datanodeStatistics                                                    =
blockManager.getDatanodeManager().getDatanodeStatistics();

            // Get the checksum type from config
            String checksumTypeStr = conf.get(DFS_CHECKSUM_TYPE_KEY,
                    DFS_CHECKSUM_TYPE_DEFAULT);
            DataChecksum.Type checksumType;
            try {
                checksumType = DataChecksum.Type.valueOf(checksumTypeStr);
            } catch (IllegalArgumentException iae) {
                throw new IOException("Invalid checksum type in "
```

```
                                + DFS_CHECKSUM_TYPE_KEY + ": " + checksumTypeStr);
                }

                // 创建 FsServerDefaults
                this.serverDefaults = new FsServerDefaults(
                                conf.getLongBytes(DFS_BLOCK_SIZE_KEY, DFS_BLOCK_SIZE_DEFAULT),
                                conf.getInt(DFS_BYTES_PER_CHECKSUM_KEY,
DFS_BYTES_PER_CHECKSUM_DEFAULT),
                                conf.getInt(DFS_CLIENT_WRITE_PACKET_SIZE_KEY,
DFS_CLIENT_WRITE_PACKET_SIZE_DEFAULT),
                                (short) conf.getInt(DFS_REPLICATION_KEY, DFS_REPLICATION_DEFAULT),
                                conf.getInt(IO_FILE_BUFFER_SIZE_KEY, IO_FILE_BUFFER_SIZE_DEFAULT),
                                conf.getBoolean(DFS_ENCRYPT_DATA_TRANSFER_KEY,
DFS_ENCRYPT_DATA_TRANSFER_DEFAULT),
                                conf.getLong(FS_TRASH_INTERVAL_KEY, FS_TRASH_INTERVAL_DEFAULT),
                                checksumType,
                                conf.getTrimmed(

CommonConfigurationKeysPublic.HADOOP_SECURITY_KEY_PROVIDER_PATH,
                                                ""),
                                blockManager.getStoragePolicySuite().getDefaultPolicy().getId());

                this.maxFsObjects = conf.getLong(DFS_NAMENODE_MAX_OBJECTS_KEY,
                                DFS_NAMENODE_MAX_OBJECTS_DEFAULT);

                this.minBlockSize                                                                =
conf.getLong(DFSConfigKeys.DFS_NAMENODE_MIN_BLOCK_SIZE_KEY,
                                DFSConfigKeys.DFS_NAMENODE_MIN_BLOCK_SIZE_DEFAULT);
                this.maxBlocksPerFile                                                            =
conf.getLong(DFSConfigKeys.DFS_NAMENODE_MAX_BLOCKS_PER_FILE_KEY,
                                DFSConfigKeys.DFS_NAMENODE_MAX_BLOCKS_PER_FILE_DEFAULT);
                this.numCommittedAllowed = conf.getInt(

DFSConfigKeys.DFS_NAMENODE_FILE_CLOSE_NUM_COMMITTED_ALLOWED_KEY,

DFSConfigKeys.DFS_NAMENODE_FILE_CLOSE_NUM_COMMITTED_ALLOWED_DEFAULT);

                this.maxCorruptFileBlocksReturn = conf.getInt(

DFSConfigKeys.DFS_NAMENODE_MAX_CORRUPT_FILE_BLOCKS_RETURNED_KEY,

DFSConfigKeys.DFS_NAMENODE_MAX_CORRUPT_FILE_BLOCKS_RETURNED_DEFAULT);

                this.dtpReplaceDatanodeOnFailure = ReplaceDatanodeOnFailure.get(conf);
```

```
                this.standbyShouldCheckpoint = conf.getBoolean(
                        DFS_HA_STANDBY_CHECKPOINTS_KEY,
DFS_HA_STANDBY_CHECKPOINTS_DEFAULT);
                // # edit autoroll threshold is a multiple of the checkpoint threshold
// edit  默认滚动阈值  0.5 * 100 0000
                this.editLogRollerThreshold = (long)
                        (conf.getFloat(

DFS_NAMENODE_EDIT_LOG_AUTOROLL_MULTIPLIER_THRESHOLD,

DFS_NAMENODE_EDIT_LOG_AUTOROLL_MULTIPLIER_THRESHOLD_DEFAULT) *
                                        conf.getLong(
                                                DFS_NAMENODE_CHECKPOINT_TXNS_KEY,
                                                DFS_NAMENODE_CHECKPOINT_TXNS_DEFAULT));
// edit  默认滚动间隔    5 * 60 * 1000
                this.editLogRollerInterval = conf.getInt(
                        DFS_NAMENODE_EDIT_LOG_AUTOROLL_CHECK_INTERVAL_MS,

DFS_NAMENODE_EDIT_LOG_AUTOROLL_CHECK_INTERVAL_MS_DEFAULT);

                this.lazyPersistFileScrubIntervalSec = conf.getInt(
                        DFS_NAMENODE_LAZY_PERSIST_FILE_SCRUB_INTERVAL_SEC,
                        DFS_NAMENODE_LAZY_PERSIST_FILE_SCRUB_INTERVAL_SEC_DEFAULT);

                if (this.lazyPersistFileScrubIntervalSec < 0) {
                      throw new IllegalArgumentException(
                                DFS_NAMENODE_LAZY_PERSIST_FILE_SCRUB_INTERVAL_SEC
                                        + " must be zero (for disable) or greater than zero.");
                }

                this.edekCacheLoaderDelay = conf.getInt(

DFSConfigKeys.DFS_NAMENODE_EDEKCACHELOADER_INITIAL_DELAY_MS_KEY,

DFSConfigKeys.DFS_NAMENODE_EDEKCACHELOADER_INITIAL_DELAY_MS_DEFAULT);
                this.edekCacheLoaderInterval = conf.getInt(

DFSConfigKeys.DFS_NAMENODE_EDEKCACHELOADER_INTERVAL_MS_KEY,

DFSConfigKeys.DFS_NAMENODE_EDEKCACHELOADER_INTERVAL_MS_DEFAULT);

                this.leaseRecheckIntervalMs = conf.getLong(
                        DFS_NAMENODE_LEASE_RECHECK_INTERVAL_MS_KEY,
```

```
                    DFS_NAMENODE_LEASE_RECHECK_INTERVAL_MS_DEFAULT);
            this.maxLockHoldToReleaseLeaseMs = conf.getLong(
                    DFS_NAMENODE_MAX_LOCK_HOLD_TO_RELEASE_LEASE_MS_KEY,

DFS_NAMENODE_MAX_LOCK_HOLD_TO_RELEASE_LEASE_MS_DEFAULT);

            // For testing purposes, allow the DT secret manager to be started regardless
            // of whether security is enabled.
            alwaysUseDelegationTokensForTests = conf.getBoolean(
                    DFS_NAMENODE_DELEGATION_TOKEN_ALWAYS_USE_KEY,
                    DFS_NAMENODE_DELEGATION_TOKEN_ALWAYS_USE_DEFAULT);

            this.dtSecretManager = createDelegationTokenSecretManager(conf);

            // 创建 FSDirectory (管理 NameNode 内存元数据)
            this.dir = new FSDirectory(this, conf);
            // 创建 SnapshotManager
            this.snapshotManager = new SnapshotManager(conf, dir);
            // 创建 CacheManager
            this.cacheManager = new CacheManager(this, conf, blockManager);
            // Init ErasureCodingPolicyManager instance.
            ErasureCodingPolicyManager.getInstance().init(conf);
            this.topConf = new TopConf(conf);
            this.auditLoggers = initAuditLoggers(conf);
            this.isDefaultAuditLogger = auditLoggers.size() == 1 &&
                    auditLoggers.get(0) instanceof DefaultAuditLogger;
            this.retryCache = ignoreRetryCache ? null : initRetryCache(conf);
            Class<? extends INodeAttributeProvider> klass = conf.getClass(
                    DFS_NAMENODE_INODE_ATTRIBUTES_PROVIDER_KEY,
                    null, INodeAttributeProvider.class);
            if (klass != null) {
                inodeAttributeProvider = ReflectionUtils.newInstance(klass, conf);
                LOG.info("Using INode attribute provider: " + klass.getName());
            }
            this.maxListOpenFilesResponses = conf.getInt(
                    DFSConfigKeys.DFS_NAMENODE_LIST_OPENFILES_NUM_RESPONSES,

DFSConfigKeys.DFS_NAMENODE_LIST_OPENFILES_NUM_RESPONSES_DEFAULT
            );
            Preconditions.checkArgument(maxListOpenFilesResponses > 0,
                    DFSConfigKeys.DFS_NAMENODE_LIST_OPENFILES_NUM_RESPONSES +
                            " must be a positive integer."
            );
```

```
                this.blockDeletionIncrement = conf.getInt(
                        DFSConfigKeys.DFS_NAMENODE_BLOCK_DELETION_INCREMENT_KEY,

DFSConfigKeys.DFS_NAMENODE_BLOCK_DELETION_INCREMENT_DEFAULT);
                Preconditions.checkArgument(blockDeletionIncrement > 0,
                        DFSConfigKeys.DFS_NAMENODE_BLOCK_DELETION_INCREMENT_KEY +
                                " must be a positive integer.");
            } catch (IOException e) {
                LOG.error(getClass().getSimpleName() + " initialization failed.", e);
                close();
                throw e;
            } catch (RuntimeException re) {
                LOG.error(getClass().getSimpleName() + " initialization failed.", re);
                close();
                throw re;
            }
        }
```

3.3.4.2.3.1 创建 BlockManager

```
public BlockManager(final Namesystem namesystem, boolean haEnabled,
                                final Configuration conf) throws IOException {
            // FSNamesystem
            this.namesystem = namesystem;

            // 创建 DatanodeManager
            datanodeManager = new DatanodeManager(this, namesystem, conf);

            // 从 DatanodeManager 获取 HeartbeatManager
            heartbeatManager = datanodeManager.getHeartbeatManager();

            // 创建 BlockIdManager
            this.blockIdManager = new BlockIdManager(this);

            blocksPerPostponedRescan = (int) Math.min(Integer.MAX_VALUE,
                    datanodeManager.getBlocksPerPostponedMisreplicatedBlocksRescan());
            rescannedMisreplicatedBlocks =
                    new ArrayList<Block>(blocksPerPostponedRescan);
            startupDelayBlockDeletionInMs = conf.getLong(

DFSConfigKeys.DFS_NAMENODE_STARTUP_DELAY_BLOCK_DELETION_SEC_KEY,

DFSConfigKeys.DFS_NAMENODE_STARTUP_DELAY_BLOCK_DELETION_SEC_DEFAULT) * 1000L;
            invalidateBlocks = new InvalidateBlocks(
```

```
                datanodeManager.getBlockInvalidateLimit(),
                startupDelayBlockDeletionInMs,
                blockIdManager);

        // Compute the map capacity by allocating 2% of total memory
        blocksMap = new BlocksMap(
                LightWeightGSet.computeCapacity(2.0, "BlocksMap"));
        placementPolicies = new BlockPlacementPolicies(
                conf, datanodeManager.getFSClusterStats(),
                datanodeManager.getNetworkTopology(),
                datanodeManager.getHost2DatanodeMap());
        storagePolicySuite = BlockStoragePolicySuite.createDefaultSuite();
        pendingReconstruction = new PendingReconstructionBlocks(conf.getInt(

DFSConfigKeys.DFS_NAMENODE_RECONSTRUCTION_PENDING_TIMEOUT_SEC_KEY,

DFSConfigKeys.DFS_NAMENODE_RECONSTRUCTION_PENDING_TIMEOUT_SEC_DEFAULT)
                * 1000L);

        blockTokenSecretManager = createBlockTokenSecretManager(conf);

        providedStorageMap = new ProvidedStorageMap(namesystem, this, conf);

        this.maxCorruptFilesReturned = conf.getInt(
                DFSConfigKeys.DFS_DEFAULT_MAX_CORRUPT_FILES_RETURNED_KEY,
                DFSConfigKeys.DFS_DEFAULT_MAX_CORRUPT_FILES_RETURNED);
        this.defaultReplication = conf.getInt(DFSConfigKeys.DFS_REPLICATION_KEY,
                DFSConfigKeys.DFS_REPLICATION_DEFAULT);

        final int maxR = conf.getInt(DFSConfigKeys.DFS_REPLICATION_MAX_KEY,
                DFSConfigKeys.DFS_REPLICATION_MAX_DEFAULT);
        final int minR = conf.getInt(DFSConfigKeys.DFS_NAMENODE_REPLICATION_MIN_KEY,
                DFSConfigKeys.DFS_NAMENODE_REPLICATION_MIN_DEFAULT);
        if (minR <= 0)
            throw new IOException("Unexpected configuration parameters: "
                    + DFSConfigKeys.DFS_NAMENODE_REPLICATION_MIN_KEY
                    + " = " + minR + " <= 0");
        if (maxR > Short.MAX_VALUE)
            throw new IOException("Unexpected configuration parameters: "
                    + DFSConfigKeys.DFS_REPLICATION_MAX_KEY
                    + " = " + maxR + " > " + Short.MAX_VALUE);
        if (minR > maxR)
            throw new IOException("Unexpected configuration parameters: "
                    + DFSConfigKeys.DFS_NAMENODE_REPLICATION_MIN_KEY
```

```
                            + " = " + minR + " > "
                            + DFSConfigKeys.DFS_REPLICATION_MAX_KEY
                            + " = " + maxR);
            this.minReplication = (short) minR;
            this.maxReplication = (short) maxR;

            this.maxReplicationStreams =

conf.getInt(DFSConfigKeys.DFS_NAMENODE_REPLICATION_MAX_STREAMS_KEY,

DFSConfigKeys.DFS_NAMENODE_REPLICATION_MAX_STREAMS_DEFAULT);
            this.replicationStreamsHardLimit =
                    conf.getInt(

DFSConfigKeys.DFS_NAMENODE_REPLICATION_STREAMS_HARD_LIMIT_KEY,

DFSConfigKeys.DFS_NAMENODE_REPLICATION_STREAMS_HARD_LIMIT_DEFAULT);
            this.blocksInvalidateWorkPct = DFSUtil.getInvalidateWorkPctPerIteration(conf);
            this.blocksReplWorkMultiplier = DFSUtil.getReplWorkMultiplier(conf);

            this.redundancyRecheckIntervalMs = conf.getTimeDuration(
                    DFSConfigKeys.DFS_NAMENODE_REDUNDANCY_INTERVAL_SECONDS_KEY,

DFSConfigKeys.DFS_NAMENODE_REDUNDANCY_INTERVAL_SECONDS_DEFAULT,
                    TimeUnit.SECONDS, TimeUnit.MILLISECONDS);

            this.storageInfoDefragmentInterval =
                    conf.getLong(

DFSConfigKeys.DFS_NAMENODE_STORAGEINFO_DEFRAGMENT_INTERVAL_MS_KEY,

DFSConfigKeys.DFS_NAMENODE_STORAGEINFO_DEFRAGMENT_INTERVAL_MS_DEFAULT);
            this.storageInfoDefragmentTimeout =
                    conf.getLong(

DFSConfigKeys.DFS_NAMENODE_STORAGEINFO_DEFRAGMENT_TIMEOUT_MS_KEY,

DFSConfigKeys.DFS_NAMENODE_STORAGEINFO_DEFRAGMENT_TIMEOUT_MS_DEFAULT);
            this.storageInfoDefragmentRatio =
                    conf.getDouble(

DFSConfigKeys.DFS_NAMENODE_STORAGEINFO_DEFRAGMENT_RATIO_KEY,

DFSConfigKeys.DFS_NAMENODE_STORAGEINFO_DEFRAGMENT_RATIO_DEFAULT);
```

```
        this.encryptDataTransfer =
                conf.getBoolean(DFSConfigKeys.DFS_ENCRYPT_DATA_TRANSFER_KEY,
                        DFSConfigKeys.DFS_ENCRYPT_DATA_TRANSFER_DEFAULT);

        this.maxNumBlocksToLog =
                conf.getLong(DFSConfigKeys.DFS_MAX_NUM_BLOCKS_TO_LOG_KEY,
                        DFSConfigKeys.DFS_MAX_NUM_BLOCKS_TO_LOG_DEFAULT);
        this.numBlocksPerIteration = conf.getInt(
                DFSConfigKeys.DFS_BLOCK_MISREPLICATION_PROCESSING_LIMIT,
                DFSConfigKeys.DFS_BLOCK_MISREPLICATION_PROCESSING_LIMIT_DEFAULT);

        final int minMaintenanceR = conf.getInt(
                DFSConfigKeys.DFS_NAMENODE_MAINTENANCE_REPLICATION_MIN_KEY,

DFSConfigKeys.DFS_NAMENODE_MAINTENANCE_REPLICATION_MIN_DEFAULT);

        if (minMaintenanceR < 0) {
            throw new IOException("Unexpected configuration parameters: "
                    +
DFSConfigKeys.DFS_NAMENODE_MAINTENANCE_REPLICATION_MIN_KEY
                    + " = " + minMaintenanceR + " < 0");
        }
        if (minMaintenanceR > defaultReplication) {
            throw new IOException("Unexpected configuration parameters: "
                    +
DFSConfigKeys.DFS_NAMENODE_MAINTENANCE_REPLICATION_MIN_KEY
                    + " = " + minMaintenanceR + " > "
                    + DFSConfigKeys.DFS_REPLICATION_KEY
                    + " = " + defaultReplication);
        }
        this.minReplicationToBeInMaintenance = (short) minMaintenanceR;

        // 3s
        long heartbeatIntervalSecs = conf.getTimeDuration(
                DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_KEY,
                DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_DEFAULT, TimeUnit.SECONDS);
        // 30s
        long blockRecoveryTimeout = getBlockRecoveryTimeout(heartbeatIntervalSecs);

        // 创建 PendingRecoveryBlocks
        pendingRecoveryBlocks = new PendingRecoveryBlocks(blockRecoveryTimeout);

        // 创建 BlockReportLeaseManager
```

```
        this.blockReportLeaseManager = new BlockReportLeaseManager(conf);

        // 创建 BlockManagerSafeMode
        bmSafeMode = new BlockManagerSafeMode(this, namesystem, haEnabled, conf);

        // 1024
        int queueSize = conf.getInt(
                DFSConfigKeys.DFS_NAMENODE_BLOCKREPORT_QUEUE_SIZE_KEY,
                DFSConfigKeys.DFS_NAMENODE_BLOCKREPORT_QUEUE_SIZE_DEFAULT);

        // Block Report 线程
        blockReportThread = new BlockReportProcessingThread(queueSize);

        LOG.info("defaultReplication             = {}", defaultReplication);
        LOG.info("maxReplication                 = {}", maxReplication);
        LOG.info("minReplication                 = {}", minReplication);
        LOG.info("maxReplicationStreams          = {}", maxReplicationStreams);
        LOG.info("redundancyRecheckInterval    = {}ms", redundancyRecheckIntervalMs);
        LOG.info("encryptDataTransfer            = {}", encryptDataTransfer);
        LOG.info("maxNumBlocksToLog              = {}", maxNumBlocksToLog);
    }
```

3.3.4.2.3.1.1 创建 DatanodeManager

```
DatanodeManager(final BlockManager blockManager, final Namesystem namesystem,
                        final Configuration conf) throws IOException {

        // FSNamesystem
        this.namesystem = namesystem;
        // BlockManager
        this.blockManager = blockManager;

        // true
        this.useDfsNetworkTopology = conf.getBoolean(
                DFSConfigKeys.DFS_USE_DFS_NETWORK_TOPOLOGY_KEY,
                DFSConfigKeys.DFS_USE_DFS_NETWORK_TOPOLOGY_DEFAULT);
        if (useDfsNetworkTopology) {
            // DFSNetworkTopology
            networktopology = DFSNetworkTopology.getInstance(conf);
        } else {
            networktopology = NetworkTopology.getInstance(conf);
        }

        // 创建 HeartbeatManager
        this.heartbeatManager = new HeartbeatManager(namesystem,
```

```
                    blockManager, conf);

            // 创建 DatanodeAdminManager
            this.datanodeAdminManager = new DatanodeAdminManager(namesystem,
                    blockManager, heartbeatManager);

            // 创建 newFSClusterStats
            this.fsClusterStats = newFSClusterStats();

            // false
            this.dataNodePeerStatsEnabled = conf.getBoolean(
                    DFSConfigKeys.DFS_DATANODE_PEER_STATS_ENABLED_KEY,
                    DFSConfigKeys.DFS_DATANODE_PEER_STATS_ENABLED_DEFAULT);
            // false
            this.dataNodeDiskStatsEnabled = Util.isDiskStatsEnabled(conf.getInt(

DFSConfigKeys.DFS_DATANODE_FILEIO_PROFILING_SAMPLING_PERCENTAGE_KEY,
                    DFSConfigKeys.

DFS_DATANODE_FILEIO_PROFILING_SAMPLING_PERCENTAGE_DEFAULT));

            // 创建 Timer
            final Timer timer = new Timer();

            // null
            this.slowPeerTracker = dataNodePeerStatsEnabled ?
                    new SlowPeerTracker(conf, timer) : null;
            // null
            this.slowDiskTracker = dataNodeDiskStatsEnabled ?
                    new SlowDiskTracker(conf, timer) : null;

            // 9866
            this.defaultXferPort = NetUtils.createSocketAddr(
                    conf.getTrimmed(DFSConfigKeys.DFS_DATANODE_ADDRESS_KEY,
                            DFSConfigKeys.DFS_DATANODE_ADDRESS_DEFAULT)).getPort();
            // 9864
            this.defaultInfoPort = NetUtils.createSocketAddr(
                    conf.getTrimmed(DFSConfigKeys.DFS_DATANODE_HTTP_ADDRESS_KEY,

DFSConfigKeys.DFS_DATANODE_HTTP_ADDRESS_DEFAULT)).getPort();
            // 9865
            this.defaultInfoSecurePort = NetUtils.createSocketAddr(
                    conf.getTrimmed(DFSConfigKeys.DFS_DATANODE_HTTPS_ADDRESS_KEY,
```

```
DFSConfigKeys.DFS_DATANODE_HTTPS_ADDRESS_DEFAULT)).getPort();
        // 9867
        this.defaultIpcPort = NetUtils.createSocketAddr(
                conf.getTrimmed(DFSConfigKeys.DFS_DATANODE_IPC_ADDRESS_KEY,
                        DFSConfigKeys.DFS_DATANODE_IPC_ADDRESS_DEFAULT)).getPort();
        // HostFileManager
        this.hostConfigManager = ReflectionUtils.newInstance(

conf.getClass(DFSConfigKeys.DFS_NAMENODE_HOSTS_PROVIDER_CLASSNAME_KEY,
                        HostFileManager.class, HostConfigManager.class), conf);
        try {
            this.hostConfigManager.refresh();
        } catch (IOException e) {
            LOG.error("error reading hosts files: ", e);
        }

        // ScriptBasedMapping
        this.dnsToSwitchMapping = ReflectionUtils.newInstance(

conf.getClass(DFSConfigKeys.NET_TOPOLOGY_NODE_SWITCH_MAPPING_IMPL_KEY,
                        ScriptBasedMapping.class, DNSToSwitchMapping.class), conf);

        // false
        this.rejectUnresolvedTopologyDN = conf.getBoolean(
                DFSConfigKeys.DFS_REJECT_UNRESOLVED_DN_TOPOLOGY_MAPPING_KEY,

DFSConfigKeys.DFS_REJECT_UNRESOLVED_DN_TOPOLOGY_MAPPING_DEFAULT);

        // If the dns to switch mapping supports cache, resolve network
        // locations of those hosts in the include list and store the mapping
        // in the cache; so future calls to resolve will be fast.
        if (dnsToSwitchMapping instanceof CachedDNSToSwitchMapping) {
            final ArrayList<String> locations = new ArrayList<>();
            for (InetSocketAddress addr : hostConfigManager.getIncludes()) {
                locations.add(addr.getAddress().getHostAddress());
            }
            dnsToSwitchMapping.resolve(locations);
        }

        // 3s = 3000ms
        heartbeatIntervalSeconds = conf.getTimeDuration(
                DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_KEY,
                DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_DEFAULT, TimeUnit.SECONDS);
        // 5m = 5 * 60 * 1000ms
```

```
        heartbeatRecheckInterval = conf.getInt(
                DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_KEY,

DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_DEFAULT); // 5 minutes
        // 630s
        this.heartbeatExpireInterval = 2 * heartbeatRecheckInterval
                + 10 * 1000 * heartbeatIntervalSeconds;

        // Effected block invalidate limit is the bigger value between
        // value configured in hdfs-site.xml, and 20 * HB interval.
        // 1000
        final int configuredBlockInvalidateLimit = conf.getInt(
                DFSConfigKeys.DFS_BLOCK_INVALIDATE_LIMIT_KEY,
                DFSConfigKeys.DFS_BLOCK_INVALIDATE_LIMIT_DEFAULT);
        final int countedBlockInvalidateLimit = 20 * (int) (heartbeatIntervalSeconds);
        this.blockInvalidateLimit = Math.max(countedBlockInvalidateLimit,
                configuredBlockInvalidateLimit);
        LOG.info(DFSConfigKeys.DFS_BLOCK_INVALIDATE_LIMIT_KEY
                + ": configured=" + configuredBlockInvalidateLimit
                + ", counted=" + countedBlockInvalidateLimit
                + ", effected=" + blockInvalidateLimit);

        this.checkIpHostnameInRegistration = conf.getBoolean(

DFSConfigKeys.DFS_NAMENODE_DATANODE_REGISTRATION_IP_HOSTNAME_CHECK_KEY,

DFSConfigKeys.DFS_NAMENODE_DATANODE_REGISTRATION_IP_HOSTNAME_CHECK_DEFAULT);

LOG.info(DFSConfigKeys.DFS_NAMENODE_DATANODE_REGISTRATION_IP_HOSTNAME_CHECK_K
EY
                + "=" + checkIpHostnameInRegistration);

        this.avoidStaleDataNodesForRead = conf.getBoolean(
                DFSConfigKeys.DFS_NAMENODE_AVOID_STALE_DATANODE_FOR_READ_KEY,

DFSConfigKeys.DFS_NAMENODE_AVOID_STALE_DATANODE_FOR_READ_DEFAULT);
        this.avoidStaleDataNodesForWrite = conf.getBoolean(

DFSConfigKeys.DFS_NAMENODE_AVOID_STALE_DATANODE_FOR_WRITE_KEY,

DFSConfigKeys.DFS_NAMENODE_AVOID_STALE_DATANODE_FOR_WRITE_DEFAULT);
        this.staleInterval = getStaleIntervalFromConf(conf, heartbeatExpireInterval);
        this.ratioUseStaleDataNodesForWrite = conf.getFloat(
```

```
DFSConfigKeys.DFS_NAMENODE_USE_STALE_DATANODE_FOR_WRITE_RATIO_KEY,

DFSConfigKeys.DFS_NAMENODE_USE_STALE_DATANODE_FOR_WRITE_RATIO_DEFAULT);
            Preconditions.checkArgument(
                        (ratioUseStaleDataNodesForWrite > 0 &&
                                    ratioUseStaleDataNodesForWrite <= 1.0f),

DFSConfigKeys.DFS_NAMENODE_USE_STALE_DATANODE_FOR_WRITE_RATIO_KEY +
                                    " = '" + ratioUseStaleDataNodesForWrite + "' is invalid. " +
                                    "It should be a positive non-zero float value, not greater than
1.0f.");
            this.timeBetweenResendingCachingDirectivesMs = conf.getLong(

DFSConfigKeys.DFS_NAMENODE_PATH_BASED_CACHE_RETRY_INTERVAL_MS,

DFSConfigKeys.DFS_NAMENODE_PATH_BASED_CACHE_RETRY_INTERVAL_MS_DEFAULT);
            this.blocksPerPostponedMisreplicatedBlocksRescan = conf.getLong(

DFSConfigKeys.DFS_NAMENODE_BLOCKS_PER_POSTPONEDBLOCKS_RESCAN_KEY,

DFSConfigKeys.DFS_NAMENODE_BLOCKS_PER_POSTPONEDBLOCKS_RESCAN_KEY_DEFAULT);
      }
```

3.3.4.2.3.1.1.1 创建 HeartbeatManager

```
HeartbeatManager(final Namesystem namesystem,
                        final BlockManager blockManager, final Configuration conf) {
            // FSNamesystem
            this.namesystem = namesystem;
            // BlockManager
            this.blockManager = blockManager;

            // false
            boolean avoidStaleDataNodesForWrite = conf.getBoolean(

DFSConfigKeys.DFS_NAMENODE_AVOID_STALE_DATANODE_FOR_WRITE_KEY,

DFSConfigKeys.DFS_NAMENODE_AVOID_STALE_DATANODE_FOR_WRITE_DEFAULT);
            // 5 min
            long recheckInterval = conf.getInt(
                        DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_KEY,

DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_DEFAULT); // 5 min
            // 30 s
```

```
            long staleInterval = conf.getLong(
                    DFSConfigKeys.DFS_NAMENODE_STALE_DATANODE_INTERVAL_KEY,
                    DFSConfigKeys.DFS_NAMENODE_STALE_DATANODE_INTERVAL_DEFAULT);//
30s

            if (avoidStaleDataNodesForWrite && staleInterval < recheckInterval) {
                this.heartbeatRecheckInterval = staleInterval;
                LOG.info("Setting heartbeat recheck interval to " + staleInterval
                        +                 "                since                 "                 +
DFSConfigKeys.DFS_NAMENODE_STALE_DATANODE_INTERVAL_KEY
                        + " is less than "
                        +
DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_KEY);
            } else {
                // 5 min
                this.heartbeatRecheckInterval = recheckInterval;
            }
        }
```

3.3.4.2.3.1.1.2 创建 DatanodeAdminManager

```
DatanodeAdminManager(final Namesystem namesystem,
        final BlockManager blockManager, final HeartbeatManager hbManager) {
    // FSNamesystem
    this.namesystem = namesystem;
    // BlockManager
    this.blockManager = blockManager;
    // HeartbeatManager
    this.hbManager = hbManager;

    //  定时线程池
    executor = Executors.newScheduledThreadPool(1,
        new ThreadFactoryBuilder().setNameFormat("DatanodeAdminMonitor-%d")
            .setDaemon(true).build());

    outOfServiceNodeBlocks = new TreeMap<>();
    pendingNodes = new ArrayDeque<>();
  }
```

3.3.4.2.3.1.2 从 DatanodeManager 获取 HeartbeatManager

```
  /** @return the heartbeat manager. */
```

```
    HeartbeatManager getHeartbeatManager() {
        return heartbeatManager;
    }
```

3.3.4.2.3.1.3 创建 PendingRecoveryBlocks

```
    PendingRecoveryBlocks(long timeout) {
        // 30s
        this.recoveryTimeoutInterval = timeout;
    }
```

3.3.4.2.3.1.4 创建 BlockReportLeaseManager

```
BlockReportLeaseManager(Configuration conf) {
        // 往下追
        this(conf.getInt(

DFSConfigKeys.DFS_NAMENODE_MAX_FULL_BLOCK_REPORT_LEASES,

DFSConfigKeys.DFS_NAMENODE_MAX_FULL_BLOCK_REPORT_LEASES_DEFAULT),
                conf.getLong(

DFSConfigKeys.DFS_NAMENODE_FULL_BLOCK_REPORT_LEASE_LENGTH_MS,

DFSConfigKeys.DFS_NAMENODE_FULL_BLOCK_REPORT_LEASE_LENGTH_MS_DEFAULT));
    }
```

```
BlockReportLeaseManager(int maxPending, long leaseExpiryMs) {
        Preconditions.checkArgument(maxPending >= 1,
                "Cannot set the maximum number of block report leases to a " +
                        "value less than 1.");
        // 6
        this.maxPending = maxPending;
        Preconditions.checkArgument(leaseExpiryMs >= 1,
                "Cannot set full block report lease expiry period to a value " +
                        "less than 1.");
        // 5L * 60L * 1000L
        this.leaseExpiryMs = leaseExpiryMs;
    }
```

3.3.4.2.3.1.5 创建 BlockManagerSafeMode

```
BlockManagerSafeMode(BlockManager blockManager, Namesystem namesystem,
                                boolean haEnabled, Configuration conf) {
        // BlockManager
        this.blockManager = blockManager;
```

```
            // FSNamesystem
            this.namesystem = namesystem;
            // true
            this.haEnabled = haEnabled;
            // 0.999
            this.threshold = conf.getFloat(DFS_NAMENODE_SAFEMODE_THRESHOLD_PCT_KEY,
                        DFS_NAMENODE_SAFEMODE_THRESHOLD_PCT_DEFAULT);

            if (this.threshold > 1.0) {
                  LOG.warn("The threshold value shouldn't be greater than 1, " +
                              "threshold: {}", threshold);
            }
            // 0
            this.datanodeThreshold = conf.getInt(
                        DFS_NAMENODE_SAFEMODE_MIN_DATANODES_KEY,
                        DFS_NAMENODE_SAFEMODE_MIN_DATANODES_DEFAULT);
            // 1
            int minReplication =
                        conf.getInt(DFSConfigKeys.DFS_NAMENODE_REPLICATION_MIN_KEY,
                                    DFSConfigKeys.DFS_NAMENODE_REPLICATION_MIN_DEFAULT);
            // DFS_NAMENODE_SAFEMODE_REPLICATION_MIN_KEY is an expert level setting,
            // setting this lower than the min replication is not recommended
            // and/or dangerous for production setups.
            // When it's unset, safeReplication will use dfs.namenode.replication.min
            // 1
            this.safeReplication =

conf.getInt(DFSConfigKeys.DFS_NAMENODE_SAFEMODE_REPLICATION_MIN_KEY,
                                    minReplication);
            // default to safe mode threshold (i.e., don't populate queues before
            // leaving safe mode)
            // 0.999
            this.replQueueThreshold =
                        conf.getFloat(DFS_NAMENODE_REPL_QUEUE_THRESHOLD_PCT_KEY,
                                    (float) threshold);
            // 30 * 1000ms
            this.extension = conf.getTimeDuration(DFS_NAMENODE_SAFEMODE_EXTENSION_KEY,
                        DFS_NAMENODE_SAFEMODE_EXTENSION_DEFAULT,
                        MILLISECONDS);

            this.inRollBack = isInRollBackMode(NameNode.getStartupOption(conf));

            LOG.info("{} = {}", DFS_NAMENODE_SAFEMODE_THRESHOLD_PCT_KEY, threshold);
            LOG.info("{} = {}", DFS_NAMENODE_SAFEMODE_MIN_DATANODES_KEY,
```

```
                datanodeThreshold);
        LOG.info("{} = {}", DFS_NAMENODE_SAFEMODE_EXTENSION_KEY, extension);
    }
```

3.3.4.2.3.1.6 创建 BlockReportProcessingThread 线程

```
private class BlockReportProcessingThread extends Thread {
        private static final long MAX_LOCK_HOLD_MS = 4;
        private long lastFull = 0;

        private final BlockingQueue<Runnable> queue;

        BlockReportProcessingThread(int size) {
            super("Block report processor");
            // 1024
            queue = new ArrayBlockingQueue<>(size);
            setDaemon(true);
        }

        @Override
        public void run() {
            try {
                processQueue();
            } catch (Throwable t) {
                ExitUtil.terminate(1,
                        getName() + " encountered fatal exception: " + t);
            }
        }
```

3.3.4.2.3.2 创建 FSDirectory

```
FSDirectory(FSNamesystem ns, Configuration conf) throws IOException {
        // used to enable/disable the use of expanded string tables.
        SerialNumberManager.initialize(conf);

        this.dirLock = new ReentrantReadWriteLock(true); // fair

        // 创建 INodeId
        this.inodeId = new INodeId();

        // 创建 NameNode 管理内存元数据根目录 INodeDirectory
        rootDir = createRoot(ns);

        inodeMap = INodeMap.newInstance(rootDir);
```

```
            // true
            this.isPermissionEnabled = conf.getBoolean(
                        DFSConfigKeys.DFS_PERMISSIONS_ENABLED_KEY,
                        DFSConfigKeys.DFS_PERMISSIONS_ENABLED_DEFAULT);
            //  当前用户
            this.fsOwnerShortUserName =
                        UserGroupInformation.getCurrentUser().getShortUserName();
            // supergroup
            this.supergroup = conf.get(
                        DFSConfigKeys.DFS_PERMISSIONS_SUPERUSERGROUP_KEY,
                        DFSConfigKeys.DFS_PERMISSIONS_SUPERUSERGROUP_DEFAULT);
            // false
            this.aclsEnabled = conf.getBoolean(
                        DFSConfigKeys.DFS_NAMENODE_ACLS_ENABLED_KEY,
                        DFSConfigKeys.DFS_NAMENODE_ACLS_ENABLED_DEFAULT);

            LOG.info("ACLs enabled? " + aclsEnabled);
            this.posixAclInheritanceEnabled = conf.getBoolean(
                        DFSConfigKeys.DFS_NAMENODE_POSIX_ACL_INHERITANCE_ENABLED_KEY,

DFSConfigKeys.DFS_NAMENODE_POSIX_ACL_INHERITANCE_ENABLED_DEFAULT);
            LOG.info("POSIX ACL inheritance enabled? " + posixAclInheritanceEnabled);
            this.xattrsEnabled = conf.getBoolean(
                        DFSConfigKeys.DFS_NAMENODE_XATTRS_ENABLED_KEY,
                        DFSConfigKeys.DFS_NAMENODE_XATTRS_ENABLED_DEFAULT);
            LOG.info("XAttrs enabled? " + xattrsEnabled);
            this.xattrMaxSize = conf.getInt(
                        DFSConfigKeys.DFS_NAMENODE_MAX_XATTR_SIZE_KEY,
                        DFSConfigKeys.DFS_NAMENODE_MAX_XATTR_SIZE_DEFAULT);
            Preconditions.checkArgument(xattrMaxSize > 0,
                        "The maximum size of an xattr should be > 0: (%s).",
                        DFSConfigKeys.DFS_NAMENODE_MAX_XATTR_SIZE_KEY);
            Preconditions.checkArgument(xattrMaxSize <=
                                    DFSConfigKeys.DFS_NAMENODE_MAX_XATTR_SIZE_HARD_LIMIT,
                        "The maximum size of an xattr should be <= maximum size"
                                    + "hard limit "
DFSConfigKeys.DFS_NAMENODE_MAX_XATTR_SIZE_HARD_LIMIT
                                    + ": (%s).",
DFSConfigKeys.DFS_NAMENODE_MAX_XATTR_SIZE_KEY);

            this.accessTimePrecision = conf.getLong(
                        DFS_NAMENODE_ACCESSTIME_PRECISION_KEY,
                        DFS_NAMENODE_ACCESSTIME_PRECISION_DEFAULT);
```

```
this.storagePolicyEnabled =
        conf.getBoolean(DFS_STORAGE_POLICY_ENABLED_KEY,
                DFS_STORAGE_POLICY_ENABLED_DEFAULT);

this.quotaByStorageTypeEnabled =
        conf.getBoolean(DFS_QUOTA_BY_STORAGETYPE_ENABLED_KEY,
                DFS_QUOTA_BY_STORAGETYPE_ENABLED_DEFAULT);

int configuredLimit = conf.getInt(
        DFSConfigKeys.DFS_LIST_LIMIT, DFSConfigKeys.DFS_LIST_LIMIT_DEFAULT);
this.lsLimit = configuredLimit > 0 ?
        configuredLimit : DFSConfigKeys.DFS_LIST_LIMIT_DEFAULT;
this.contentCountLimit = conf.getInt(
        DFSConfigKeys.DFS_CONTENT_SUMMARY_LIMIT_KEY,
        DFSConfigKeys.DFS_CONTENT_SUMMARY_LIMIT_DEFAULT);
this.contentSleepMicroSec = conf.getLong(
        DFSConfigKeys.DFS_CONTENT_SUMMARY_SLEEP_MICROSEC_KEY,
        DFSConfigKeys.DFS_CONTENT_SUMMARY_SLEEP_MICROSEC_DEFAULT);

// filesystem limits
this.maxComponentLength = conf.getInt(
        DFSConfigKeys.DFS_NAMENODE_MAX_COMPONENT_LENGTH_KEY,
        DFSConfigKeys.DFS_NAMENODE_MAX_COMPONENT_LENGTH_DEFAULT);
this.maxDirItems = conf.getInt(
        DFSConfigKeys.DFS_NAMENODE_MAX_DIRECTORY_ITEMS_KEY,
        DFSConfigKeys.DFS_NAMENODE_MAX_DIRECTORY_ITEMS_DEFAULT);
this.inodeXAttrsLimit = conf.getInt(
        DFSConfigKeys.DFS_NAMENODE_MAX_XATTRS_PER_INODE_KEY,
        DFSConfigKeys.DFS_NAMENODE_MAX_XATTRS_PER_INODE_DEFAULT);

this.protectedDirectories = parseProtectedDirectories(conf);

Preconditions.checkArgument(this.inodeXAttrsLimit >= 0,
        "Cannot set a negative limit on the number of xattrs per inode (%s).",
        DFSConfigKeys.DFS_NAMENODE_MAX_XATTRS_PER_INODE_KEY);
// We need a maximum maximum because by default, PB limits message sizes
// to 64MB. This means we can only store approximately 6.7 million entries
// per directory, but let's use 6.4 million for some safety.
final int MAX_DIR_ITEMS = 64 * 100 * 1000;
Preconditions.checkArgument(
        maxDirItems > 0 && maxDirItems <= MAX_DIR_ITEMS, "Cannot set "
                + DFSConfigKeys.DFS_NAMENODE_MAX_DIRECTORY_ITEMS_KEY
                + " to a value less than 1 or greater than " + MAX_DIR_ITEMS);
```

```
            // 10
            int threshold = conf.getInt(
                        DFSConfigKeys.DFS_NAMENODE_NAME_CACHE_THRESHOLD_KEY,
                        DFSConfigKeys.DFS_NAMENODE_NAME_CACHE_THRESHOLD_DEFAULT);
            NameNode.LOG.info("Caching file names occurring more than " + threshold
                        + " times");

            // 创建 NameCache
            nameCache = new NameCache<ByteArray>(threshold);
            // FSNamesystem
            namesystem = ns;
            // 从 FSNamesystem 属性 FSImage 中获取 FSEditLogAsync
            this.editLog = ns.getEditLog();

            ezManager = new EncryptionZoneManager(this, conf);

            this.quotaInitThreads = conf.getInt(
                        DFSConfigKeys.DFS_NAMENODE_QUOTA_INIT_THREADS_KEY,
                        DFSConfigKeys.DFS_NAMENODE_QUOTA_INIT_THREADS_DEFAULT);

            initUsersToBypassExtProvider(conf);
      }
```

3.3.4.2.3.3 创建 SnapshotManager

```
public SnapshotManager(final Configuration conf, final FSDirectory fsdir) {
      // FSDirectory
      this.fsdir = fsdir;

      // false
      this.captureOpenFiles = conf.getBoolean(
            DFS_NAMENODE_SNAPSHOT_CAPTURE_OPENFILES,
            DFS_NAMENODE_SNAPSHOT_CAPTURE_OPENFILES_DEFAULT);
      // false
      this.skipCaptureAccessTimeOnlyChange = conf.getBoolean(
            DFS_NAMENODE_SNAPSHOT_SKIP_CAPTURE_ACCESSTIME_ONLY_CHANGE,
            DFS_NAMENODE_SNAPSHOT_SKIP_CAPTURE_ACCESSTIME_ONLY_CHANGE_DEFAULT);
      // true
      this.snapshotDiffAllowSnapRootDescendant = conf.getBoolean(
            DFSConfigKeys.DFS_NAMENODE_SNAPSHOT_DIFF_ALLOW_SNAP_ROOT_DESCENDANT,
            DFSConfigKeys.

DFS_NAMENODE_SNAPSHOT_DIFF_ALLOW_SNAP_ROOT_DESCENDANT_DEFAULT);
      // 65535
      this.maxSnapshotLimit = conf.getInt(
```

```
            DFSConfigKeys.DFS_NAMENODE_SNAPSHOT_MAX_LIMIT,
            DFSConfigKeys.DFS_NAMENODE_SNAPSHOT_MAX_LIMIT_DEFAULT);
        LOG.info("Loaded config captureOpenFiles: " + captureOpenFiles
            + ", skipCaptureAccessTimeOnlyChange: "
            + skipCaptureAccessTimeOnlyChange
            + ", snapshotDiffAllowSnapRootDescendant: "
            + snapshotDiffAllowSnapRootDescendant
            + ", maxSnapshotLimit: "
            + maxSnapshotLimit);

        // 0
        final int maxLevels = conf.getInt(
            DFSConfigKeys.DFS_NAMENODE_SNAPSHOT_SKIPLIST_MAX_LEVELS,
            DFSConfigKeys.DFS_NAMENODE_SNAPSHOT_SKIPLIST_MAX_SKIP_LEVELS_DEFAULT);

        // 10
        final int skipInterval = conf.getInt(
            DFSConfigKeys.DFS_NAMENODE_SNAPSHOT_SKIPLIST_SKIP_INTERVAL,
            DFSConfigKeys.DFS_NAMENODE_SNAPSHOT_SKIPLIST_SKIP_INTERVAL_DEFAULT);

        DirectoryDiffListFactory.init(skipInterval, maxLevels, LOG);
    }
```

3.3.4.2.3.4 创建 CacheManager

```
CacheManager(FSNamesystem namesystem, Configuration conf,
                    BlockManager blockManager) {
        // FSNamesystem
        this.namesystem = namesystem;
        // BlockManager
        this.blockManager = blockManager;

        this.nextDirectiveId = 1;
        // 100
        this.maxListCachePoolsResponses = conf.getInt(
                DFS_NAMENODE_LIST_CACHE_POOLS_NUM_RESPONSES,
                DFS_NAMENODE_LIST_CACHE_POOLS_NUM_RESPONSES_DEFAULT);
        // 100
        this.maxListCacheDirectivesNumResponses = conf.getInt(
                DFS_NAMENODE_LIST_CACHE_DIRECTIVES_NUM_RESPONSES,
                DFS_NAMENODE_LIST_CACHE_DIRECTIVES_NUM_RESPONSES_DEFAULT);
        // 30 * 1000
        scanIntervalMs = conf.getLong(
                DFS_NAMENODE_PATH_BASED_CACHE_REFRESH_INTERVAL_MS,
                DFS_NAMENODE_PATH_BASED_CACHE_REFRESH_INTERVAL_MS_DEFAULT);
```

```
            // 0.25
            float cachedBlocksPercent = conf.getFloat(
                        DFS_NAMENODE_PATH_BASED_CACHE_BLOCK_MAP_ALLOCATION_PERCENT,

DFS_NAMENODE_PATH_BASED_CACHE_BLOCK_MAP_ALLOCATION_PERCENT_DEFAULT);
            if (cachedBlocksPercent < MIN_CACHED_BLOCKS_PERCENT) {
                LOG.info("Using minimum value {} for {}", MIN_CACHED_BLOCKS_PERCENT,

DFS_NAMENODE_PATH_BASED_CACHE_BLOCK_MAP_ALLOCATION_PERCENT);
                cachedBlocksPercent = MIN_CACHED_BLOCKS_PERCENT;
            }
            // 创建 LightWeightGSet
            this.cachedBlocks = new LightWeightGSet<CachedBlock, CachedBlock>(
                        LightWeightGSet.computeCapacity(cachedBlocksPercent,
                                    "cachedBlocks"));

        }
```

3.3.4.2.4 从磁盘加载元数据到内存

代码入口：org.apache.hadoop.hdfs.server.namenode.FSNamesystem.loadFSImage()

```
private void loadFSImage(StartupOption startOpt) throws IOException {
            // 获取 FSImage
            final FSImage fsImage = getFSImage();

            // format before starting up if requested
            if (startOpt == StartupOption.FORMAT) {

                fsImage.format(this, fsImage.getStorage().determineClusterId());// reuse current
id

                startOpt = StartupOption.REGULAR;
            }
            boolean success = false;
            writeLock();
            try {
                // We shouldn't be calling saveNamespace if we've come up in standby state.
                MetaRecoveryContext recovery = startOpt.createRecoveryContext();
                // 加载元数据
                final boolean staleImage
                            = fsImage.recoverTransitionRead(startOpt, this, recovery);
                if (RollingUpgradeStartupOption.ROLLBACK.matches(startOpt)) {
                    rollingUpgradeInfo = null;
                }
                final boolean needToSave = staleImage && !haEnabled && !isRollingUpgrade();
```

```
                LOG.info("Need to save fs image? " + needToSave
                        + " (staleImage=" + staleImage + ", haEnabled=" + haEnabled
                        + ", isRollingUpgrade=" + isRollingUpgrade() + ")");
                if (needToSave) {
                    // 可能 NameNode 启动需要将之前的 FSimage + Editlig 合并输出新的
FSImage
                    fsImage.saveNamespace(this);
                } else {
                    // No need to save, so mark the phase done.
                    StartupProgress prog = NameNode.getStartupProgress();
                    prog.beginPhase(Phase.SAVING_CHECKPOINT);
                    prog.endPhase(Phase.SAVING_CHECKPOINT);
                }
                // This will start a new log segment and write to the seen_txid file, so
                // we shouldn't do it when coming up in standby state
                if (!haEnabled || (haEnabled && startOpt == StartupOption.UPGRADE)
                        || (haEnabled && startOpt == StartupOption.UPGRADEONLY)) {
                    fsImage.openEditLogForWrite(getEffectiveLayoutVersion());
                }
                success = true;
            } finally {
                if (!success) {
                    fsImage.close();
                }
                writeUnlock("loadFSImage", true);
            }
            // 加载元数据完成
            imageLoadComplete();
        }
```

```
/**
     * Analyze storage directories.
     * Recover from previous transitions if required.
     * Perform fs state transition if necessary depending on the namespace info.
     * Read storage info.
     *
     * @throws IOException
     * @return true if the image needs to be saved or false otherwise
     */
    boolean recoverTransitionRead(StartupOption startOpt, FSNamesystem target,
            MetaRecoveryContext recovery)
            throws IOException {
        assert startOpt != StartupOption.FORMAT :
```

```
    "NameNode formatting should be performed before reading the image";

// 获取 image dirs
Collection<URI> imageDirs = storage.getImageDirectories();
// 获取 EditLog dirs
Collection<URI> editsDirs = editLog.getEditURIs();

// none of the data dirs exist
if((imageDirs.size() == 0 || editsDirs.size() == 0)
                                  && startOpt != StartupOption.IMPORT)
   throw new IOException(
         "All specified directories are not accessible or do not exist.");

// 1. For each data directory calculate its state and
// check whether all is consistent before transitioning.
Map<StorageDirectory, StorageState> dataDirStates =
               new HashMap<StorageDirectory, StorageState>();
boolean isFormatted = recoverStorageDirs(startOpt, storage, dataDirStates);

if (LOG.isTraceEnabled()) {
   LOG.trace("Data dir states:\n  " +
      Joiner.on("\n  ").withKeyValueSeparator(": ")
      .join(dataDirStates));
}

if (!isFormatted && startOpt != StartupOption.ROLLBACK
                           && startOpt != StartupOption.IMPORT) {
   throw new IOException("NameNode is not formatted.");
}


int layoutVersion = storage.getLayoutVersion();
if (startOpt == StartupOption.METADATAVERSION) {
   System.out.println("HDFS Image Version: " + layoutVersion);
   System.out.println("Software format version: " +
      HdfsServerConstants.NAMENODE_LAYOUT_VERSION);
   return false;
}

if (layoutVersion < Storage.LAST_PRE_UPGRADE_LAYOUT_VERSION) {
   NNStorage.checkVersionUpgradable(storage.getLayoutVersion());
}
if (startOpt != StartupOption.UPGRADE
      && startOpt != StartupOption.UPGRADEONLY
```

```
        && !RollingUpgradeStartupOption.STARTED.matches(startOpt)
        && layoutVersion < Storage.LAST_PRE_UPGRADE_LAYOUT_VERSION
        && layoutVersion != HdfsServerConstants.NAMENODE_LAYOUT_VERSION) {
      throw new IOException(
          "\nFile system image contains an old layout version "
          + storage.getLayoutVersion() + ".\nAn upgrade to version "
          + HdfsServerConstants.NAMENODE_LAYOUT_VERSION + " is required.\n"
          + "Please restart NameNode with the \""
          + RollingUpgradeStartupOption.STARTED.getOptionString()
          + "\" option if a rolling upgrade is already started;"
          + " or restart NameNode with the \""
          + StartupOption.UPGRADE.getName() + "\" option to start"
          + " a new upgrade.");
    }

    storage.processStartupOptionsForUpgrade(startOpt, layoutVersion);

    // 2. Format unformatted dirs.
    for (Iterator<StorageDirectory> it = storage.dirIterator(); it.hasNext();) {
      StorageDirectory sd = it.next();
      StorageState curState = dataDirStates.get(sd);
      switch(curState) {
      case NON_EXISTENT:
        throw new IOException(StorageState.NON_EXISTENT +
                                    " state cannot be here");
      case NOT_FORMATTED:
        // Create a dir structure, but not the VERSION file. The presence of
        // VERSION is checked in the inspector's needToSave() method and
        // saveNamespace is triggered if it is absent. This will bring
        // the storage state uptodate along with a new VERSION file.
        // If HA is enabled, NNs start up as standby so saveNamespace is not
        // triggered.
        LOG.info("Storage directory " + sd.getRoot() + " is not formatted.");
        LOG.info("Formatting ...");
        sd.clearDirectory(); // create empty currrent dir
        // For non-HA, no further action is needed here, as saveNamespace will
        // take care of the rest.
        if (!target.isHaEnabled()) {
          continue;
        }
        // If HA is enabled, save the dirs to create a version file later when
        // a checkpoint image is saved.
        if (newDirs == null) {
          newDirs = new HashSet<StorageDirectory>();
```

```
            }
            newDirs.add(sd);
            break;
          default:
            break;
          }
        }

        // 3. Do transitions
        switch(startOpt) {
        case UPGRADE:
        case UPGRADEONLY:
          doUpgrade(target);
          return false; // upgrade saved image already
        case IMPORT:
          doImportCheckpoint(target);
          return false; // import checkpoint saved image already
        case ROLLBACK:
          throw new AssertionError("Rollback is now a standalone command, " +
              "NameNode should not be starting with this option.");
        case REGULAR:
        default:
          // just load the image
        }

        // 从磁盘加载元数据到内存
        return loadFSImage(target, startOpt, recovery);
      }
```

```
/**
   * Choose latest image from one of the directories,
   * load it and merge with the edits.
   *
   * Saving and loading fsimage should never trigger symlink resolution.
   * The paths that are persisted do not have *intermediate* symlinks
   * because intermediate symlinks are resolved at the time files,
   * directories, and symlinks are created. All paths accessed while
   * loading or saving fsimage should therefore only see symlinks as
   * the final path component, and the functions called below do not
   * resolve symlinks that are the final path component.
   *
   * @return whether the image should be saved
   * @throws IOException
```

```
 */
private boolean loadFSImage(FSNamesystem target, StartupOption startOpt,
      MetaRecoveryContext recovery)
      throws IOException {
   final boolean rollingRollback
         = RollingUpgradeStartupOption.ROLLBACK.matches(startOpt);
   final EnumSet<NameNodeFile> nnfs;
   if (rollingRollback) {
      // if it is rollback of rolling upgrade, only load from the rollback image
      nnfs = EnumSet.of(NameNodeFile.IMAGE_ROLLBACK);
   } else {
      // otherwise we can load from both IMAGE and IMAGE_ROLLBACK
      nnfs = EnumSet.of(NameNodeFile.IMAGE, NameNodeFile.IMAGE_ROLLBACK);
   }
   final FSImageStorageInspector inspector = storage
         .readAndInspectDirs(nnfs, startOpt);

   isUpgradeFinalized = inspector.isUpgradeFinalized();
   // 获取 FSImage 文件
   List<FSImageFile> imageFiles = inspector.getLatestImages();

   StartupProgress prog = NameNode.getStartupProgress();
   prog.beginPhase(Phase.LOADING_FSIMAGE);
   File phaseFile = imageFiles.get(0).getFile();
   prog.setFile(Phase.LOADING_FSIMAGE, phaseFile.getAbsolutePath());
   prog.setSize(Phase.LOADING_FSIMAGE, phaseFile.length());
   boolean needToSave = inspector.needToSave();

   Iterable<EditLogInputStream> editStreams = null;

   // 初始化 EditLog (NameNode 本地和 JournalNode)
   initEditLog(startOpt);

   if (NameNodeLayoutVersion.supports(
         LayoutVersion.Feature.TXID_BASED_LAYOUT, getLayoutVersion())) {
      // If we're open for write, we're either non-HA or we're the active NN, so
      // we better be able to load all the edits. If we're the standby NN, it's
      // OK to not be able to read all of edits right now.
      // In the meanwhile, for HA upgrade, we will still write editlog thus need
      // this toAtLeastTxId to be set to the max-seen txid
      // For rollback in rolling upgrade, we need to set the toAtLeastTxId to
      // the txid right before the upgrade marker.
      long toAtLeastTxId = editLog.isOpenForWrite() ? inspector
            .getMaxSeenTxId() : 0;
```

```
    if (rollingRollback) {
      // note that the first image in imageFiles is the special checkpoint
      // for the rolling upgrade
      toAtLeastTxId = imageFiles.get(0).getCheckpointTxId() + 2;
    }
    editStreams = editLog.selectInputStreams(
        imageFiles.get(0).getCheckpointTxId() + 1,
        toAtLeastTxId, recovery, false);
  } else {
    editStreams = FSImagePreTransactionalStorageInspector
      .getEditLogStreams(storage);
  }
  int maxOpSize = conf.getInt(DFSConfigKeys.DFS_NAMENODE_MAX_OP_SIZE_KEY,
      DFSConfigKeys.DFS_NAMENODE_MAX_OP_SIZE_DEFAULT);
  for (EditLogInputStream elis : editStreams) {
    elis.setMaxOpSize(maxOpSize);
  }

  for (EditLogInputStream l : editStreams) {
    LOG.debug("Planning to load edit log stream: " + l);
  }
  if (!editStreams.iterator().hasNext()) {
    LOG.info("No edit log streams selected.");
  }

  FSImageFile imageFile = null;
  for (int i = 0; i < imageFiles.size(); i++) {
    try {
      imageFile = imageFiles.get(i);
      // 加载 fsimage
      loadFSImageFile(target, recovery, imageFile, startOpt);
      break;
    } catch (IllegalReservedPathException ie) {
      throw new IOException("Failed to load image from " + imageFile,
          ie);
    } catch (Exception e) {
      LOG.error("Failed to load image from " + imageFile, e);
      target.clear();
      imageFile = null;
    }
  }
  // Failed to load any images, error out
  if (imageFile == null) {
    FSEditLog.closeAllStreams(editStreams);
```

```
            throw new IOException("Failed to load FSImage file, see error(s) " +
                "above for more info.");
        }
        prog.endPhase(Phase.LOADING_FSIMAGE);

        if (!rollingRollback) {
          // 加载 editlog
          long txnsAdvanced = loadEdits(editStreams, target, Long.MAX_VALUE,
              startOpt, recovery);
          needToSave |= needsResaveBasedOnStaleCheckpoint(imageFile.getFile(),
              txnsAdvanced);
        } else {
          // Trigger the rollback for rolling upgrade. Here lastAppliedTxId equals
          // to the last txid in rollback fsimage.
          rollingRollback(lastAppliedTxId + 1, imageFiles.get(0).getCheckpointTxId());
          needToSave = false;
        }
        editLog.setNextTxId(lastAppliedTxId + 1);
        return needToSave;
      }
```

3.3.4.2.4.1 初始化 EditLog (NameNode 本地和 JournalNode)

```
public void initEditLog(StartupOption startOpt) throws IOException {
    Preconditions.checkState(getNamespaceID() != 0,
        "Must know namespace ID before initting edit log");
    String nameserviceId = DFSUtil.getNamenodeNameServiceId(conf);
    if (!HAUtil.isHAEnabled(conf, nameserviceId)) {
      // If this NN is not HA
      editLog.initJournalsForWrite();
      editLog.recoverUnclosedStreams();
    } else if (HAUtil.isHAEnabled(conf, nameserviceId)
        && (startOpt == StartupOption.UPGRADE
            || startOpt == StartupOption.UPGRADEONLY
            || RollingUpgradeStartupOption.ROLLBACK.matches(startOpt))) {
      // This NN is HA, but we're doing an upgrade or a rollback of rolling
      // upgrade so init the edit log for write.
      editLog.initJournalsForWrite();
      if (startOpt == StartupOption.UPGRADE
          || startOpt == StartupOption.UPGRADEONLY) {
        long sharedLogCTime = editLog.getSharedLogCTime();
        if (this.storage.getCTime() < sharedLogCTime) {
          throw new IOException("It looks like the shared log is already " +
              "being upgraded but this NN has not been upgraded yet. You " +
```

```
                    "should restart this NameNode with the '" +
                    StartupOption.BOOTSTRAPSTANDBY.getName() + "' option to bring " +
                    "this NN in sync with the other.");
            }
          }
          editLog.recoverUnclosedStreams();
        } else {
          // This NN is HA and we're not doing an upgrade.
          // 初始化 Editlog 输出流 (包括 本地 editLog 以及 JournalNode)
          editLog.initSharedJournalsForRead();
        }
      }
```

```
public synchronized void initSharedJournalsForRead() {
        if (state == State.OPEN_FOR_READING) {
            LOG.warn("Initializing shared journals for READ, already open for READ",
                    new Exception());
            return;
        }
        Preconditions.checkState(state == State.UNINITIALIZED ||
                state == State.CLOSED);

        // 初始化 JournalNode 连接信息
        initJournals(
                // 从配置中获取 key = dfs.namenode.shared.edits.dir
                // 一 般 在 hdfs-site.xml 配 置 比 如
qjournal://hj101:8485;hj102:8485;hj103:8485/mycluster
                this.sharedEditsDirs
        );

        state = State.OPEN_FOR_READING;
    }
```

```
private synchronized void initJournals(List<URI> dirs) {
        // 1
        int minimumRedundantJournals = conf.getInt(
                DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_KEY,
                DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_DEFAULT);

        synchronized (journalSetLock) {
            // 创建 JournalSet
            journalSet = new JournalSet(minimumRedundantJournals);

            for (URI u : dirs) {
```

```
                    // true
                    boolean required = FSNamesystem.getRequiredNamespaceEditsDirs(conf)
                            .contains(u);
                    // false
                    if (u.getScheme().equals(NNStorage.LOCAL_URI_SCHEME)) {
                        StorageDirectory sd = storage.getStorageDirectory(u);
                        if (sd != null) {
                            // 创建 FileJournalManager (NameNode 本地 EditLog)
                            journalSet.add(new FileJournalManager(conf, sd, storage),
                                    required, sharedEditsDirs.contains(u));
                        }
                    } else {
                        //
                        journalSet.add(
                                // 创建 QuorumJournalManager (JournalNode)
                                createJournal(u),
                                // true
                                required,
                                // true
                                sharedEditsDirs.contains(u));
                    }
                }
            }

            if (journalSet.isEmpty()) {
                LOG.error("No edits directories configured!");
            }
        }
```

3.3.4.2.4.1.1 创建 FileJournalManager (NameNode 本地 EditLog)

```
    void add(JournalManager j, boolean required, boolean shared) {
        // FileJournalManager (NameNode 本地 EditLog)
        // QuorumJournalManager (JournalNode 的 EditLog)
        JournalAndStream jas = new JournalAndStream(j, required, shared);
        journals.add(jas);
    }
```

3.3.4.2.4.1.2 创建 QuorumJournalManager (JournalNode)

```
void add(JournalManager j, boolean required, boolean shared) {
        // FileJournalManager (NameNode 本地 EditLog)
        // QuorumJournalManager (JournalNode 的 EditLog)
        JournalAndStream jas = new JournalAndStream(j, required, shared);
```

```
            journals.add(jas);
        }
```

3.3.4.2.4.2 加载 FSImage

```
void loadFSImageFile(FSNamesystem target, MetaRecoveryContext recovery,
            FSImageFile imageFile, StartupOption startupOption) throws IOException {
        LOG.info("Planning to load image: " + imageFile);
        StorageDirectory sdForProperties = imageFile.sd;
        storage.readProperties(sdForProperties, startupOption);

        if (NameNodeLayoutVersion.supports(
                LayoutVersion.Feature.TXID_BASED_LAYOUT, getLayoutVersion())) {
            // For txid-based layout, we should have a .md5 file
            // next to the image file
            boolean isRollingRollback = RollingUpgradeStartupOption.ROLLBACK
                    .matches(startupOption);
            loadFSImage(imageFile.getFile(), target, recovery, isRollingRollback);
        } else if (NameNodeLayoutVersion.supports(
                LayoutVersion.Feature.FSIMAGE_CHECKSUM, getLayoutVersion())) {
            // In 0.22, we have the checksum stored in the VERSION file.
            String md5 = storage.getDeprecatedProperty(
                    NNStorage.DEPRECATED_MESSAGE_DIGEST_PROPERTY);
            if (md5 == null) {
                throw new InconsistentFSStateException(sdForProperties.getRoot(),
                        "Message digest property " +
                        NNStorage.DEPRECATED_MESSAGE_DIGEST_PROPERTY +
                        " not set for storage directory " + sdForProperties.getRoot());
            }
            // 执行
            loadFSImage(imageFile.getFile(), new MD5Hash(md5), target, recovery,
                    false);
        } else {
            // We don't have any record of the md5sum
            loadFSImage(imageFile.getFile(), null, target, recovery, false);
        }
    }
```

3.3.4.2.4.3 加载 EditLog

```
public long loadEdits(Iterable<EditLogInputStream> editStreams,
            FSNamesystem target, long maxTxnsToRead,
            StartupOption startOpt, MetaRecoveryContext recovery)
            throws IOException {
        LOG.debug("About to load edits:\n    " + Joiner.on("\n    ").join(editStreams));
```

```
StartupProgress prog = NameNode.getStartupProgress();
prog.beginPhase(Phase.LOADING_EDITS);

long prevLastAppliedTxId = lastAppliedTxId;
long remainingReadTxns = maxTxnsToRead;
try {
  // 创建 FSEditLogLoader
  FSEditLogLoader loader = new FSEditLogLoader(target, lastAppliedTxId);

  // Load latest edits
  for (EditLogInputStream editIn : editStreams) {
    LogAction logAction = loadEditLogHelper.record();
    if (logAction.shouldLog()) {
      String logSuppressed = "";
      if (logAction.getCount() > 1) {
        logSuppressed = "; suppressed logging for " +
            (logAction.getCount() - 1) + " edit reads";
      }
      LOG.info("Reading " + editIn + " expecting start txid #" +
          (lastAppliedTxId + 1) + logSuppressed);
    }
    try {
      // 执行
      remainingReadTxns -= loader.loadFSEdits(editIn, lastAppliedTxId + 1,
              remainingReadTxns, startOpt, recovery);
    } finally {
      // Update lastAppliedTxId even in case of error, since some ops may
      // have been successfully applied before the error.
      lastAppliedTxId = loader.getLastAppliedTxId();
    }
    // If we are in recovery mode, we may have skipped over some txids.
    if (editIn.getLastTxId() != HdfsServerConstants.INVALID_TXID
        && recovery != null) {
      lastAppliedTxId = editIn.getLastTxId();
    }
    if (remainingReadTxns <= 0) {
      break;
    }
  }
} finally {
  FSEditLog.closeAllStreams(editStreams);
}
prog.endPhase(Phase.LOADING_EDITS);
return lastAppliedTxId - prevLastAppliedTxId;
```

```
    }
```

3.3.4.3 创建 NameNodeRpcServer

```
/**
     * Create the RPC server implementation. Used as an extension point for the
     * BackupNode.
     */
    protected NameNodeRpcServer createRpcServer(Configuration conf)
            throws IOException {
        // 创建 NameNodeRpcServer
        return new NameNodeRpcServer(conf, this);
    }
```

NameNodeRpcServer 实现如下协议

```
NameNodeRpcServer implements NamenodeProtocols
  extends ClientProtocol,
          DatanodeProtocol,
          DatanodeLifelineProtocol,
          NamenodeProtocol,
          RefreshAuthorizationPolicyProtocol,
          ReconfigurationProtocol,
          RefreshUserMappingsProtocol,
          RefreshCallQueueProtocol,
          GenericRefreshProtocol,
          GetUserMappingsProtocol,
          HAServiceProtocol,
          TraceAdminProtocol {
}
```

```
public NameNodeRpcServer(Configuration conf, NameNode nn)
            throws IOException {
        // NameNode
        this.nn = nn;
        // FSNamesystem
        this.namesystem = nn.getNamesystem();

        this.retryCache = namesystem.getRetryCache();
        this.metrics = NameNode.getNameNodeMetrics();

        // NameNodeRpcServer ClientRpcServer 服务 Handler 个数设置项 默认值 10
        // key = dfs.namenode.handler.count
```

```
int handlerCount =
        conf.getInt(DFS_NAMENODE_HANDLER_COUNT_KEY,
                DFS_NAMENODE_HANDLER_COUNT_DEFAULT);

// 注册 Rpc 协议 ClientProtocol (NameNodeRpcServer 实现该接口)
// 客户端 (编写用户代码上传/下载文件) 与 NameNode 通讯协议
RPC.setProtocolEngine(conf, ClientNamenodeProtocolPB.class,
        ProtobufRpcEngine.class);
ClientNamenodeProtocolServerSideTranslatorPB
        clientProtocolServerTranslator =
        new ClientNamenodeProtocolServerSideTranslatorPB(this);
BlockingService clientNNPbService = ClientNamenodeProtocol.
        newReflectiveBlockingService(clientProtocolServerTranslator);

// 一次 RPC Request 数据包最大长度
// key = ipc.maximum.data.length = ipc.maximum.data.length
int maxDataLength = conf.getInt(IPC_MAXIMUM_DATA_LENGTH,
        IPC_MAXIMUM_DATA_LENGTH_DEFAULT);

// NameNodeRpcServer 实现 DatanodeProtocol 接口协议
// 该接口协议与 DataNode 通讯
// Protocol that a DFS datanode uses to communicate with the NameNode
// It's used to upload current load information and block reports
DatanodeProtocolServerSideTranslatorPB dnProtoPbTranslator =
        new DatanodeProtocolServerSideTranslatorPB(this, maxDataLength);
BlockingService dnProtoPbService = DatanodeProtocolService
        .newReflectiveBlockingService(dnProtoPbTranslator);

// Protocol used by a DataNode to send lifeline messages to a NameNode.
DatanodeLifelineProtocolServerSideTranslatorPB lifelineProtoPbTranslator =
        new DatanodeLifelineProtocolServerSideTranslatorPB(this);
BlockingService lifelineProtoPbService = DatanodeLifelineProtocolService
        .newReflectiveBlockingService(lifelineProtoPbTranslator);

// Protocol that a secondary NameNode uses to communicate with the NameNode
// It's used to get part of the name node state
NamenodeProtocolServerSideTranslatorPB namenodeProtocolXlator =
        new NamenodeProtocolServerSideTranslatorPB(this);
BlockingService NNPbService = NamenodeProtocolService
        .newReflectiveBlockingService(namenodeProtocolXlator);

RefreshAuthorizationPolicyProtocolServerSideTranslatorPB refreshAuthPolicyXlator =
        new RefreshAuthorizationPolicyProtocolServerSideTranslatorPB(this);
BlockingService refreshAuthService = RefreshAuthorizationPolicyProtocolService
```

```
                .newReflectiveBlockingService(refreshAuthPolicyXlator);

        RefreshUserMappingsProtocolServerSideTranslatorPB refreshUserMappingXlator =
                new RefreshUserMappingsProtocolServerSideTranslatorPB(this);
        BlockingService refreshUserMappingService = RefreshUserMappingsProtocolService
                .newReflectiveBlockingService(refreshUserMappingXlator);

        RefreshCallQueueProtocolServerSideTranslatorPB refreshCallQueueXlator =
                new RefreshCallQueueProtocolServerSideTranslatorPB(this);
        BlockingService refreshCallQueueService = RefreshCallQueueProtocolService
                .newReflectiveBlockingService(refreshCallQueueXlator);

        GenericRefreshProtocolServerSideTranslatorPB genericRefreshXlator =
                new GenericRefreshProtocolServerSideTranslatorPB(this);
        BlockingService genericRefreshService = GenericRefreshProtocolService
                .newReflectiveBlockingService(genericRefreshXlator);

        GetUserMappingsProtocolServerSideTranslatorPB getUserMappingXlator =
                new GetUserMappingsProtocolServerSideTranslatorPB(this);
        BlockingService getUserMappingService = GetUserMappingsProtocolService
                .newReflectiveBlockingService(getUserMappingXlator);

        // 与 ZKFC 通讯协议接口
        // Protocol interface that provides High Availability related primitives to
        // monitor and fail-over the service.
        HAServiceProtocolServerSideTranslatorPB haServiceProtocolXlator =
                new HAServiceProtocolServerSideTranslatorPB(this);
        BlockingService haPbService = HAServiceProtocolService
                .newReflectiveBlockingService(haServiceProtocolXlator);

        ReconfigurationProtocolServerSideTranslatorPB reconfigurationProtocolXlator
                = new ReconfigurationProtocolServerSideTranslatorPB(this);
        BlockingService reconfigurationPbService = ReconfigurationProtocolService
                .newReflectiveBlockingService(reconfigurationProtocolXlator);

        TraceAdminProtocolServerSideTranslatorPB traceAdminXlator =
                new TraceAdminProtocolServerSideTranslatorPB(this);
        BlockingService traceAdminService = TraceAdminService
                .newReflectiveBlockingService(traceAdminXlator);

        // 默认返回 null（因为没有在 hdfs-site.xml 配置 key = dfs.namenode.servicerpc-address.mycluster.nn1 value = hj101:53310)
        // 如果配置了 返回 hj101:53310
        InetSocketAddress serviceRpcAddr = nn.getServiceRpcServerAddress(conf);
```

```
        if (serviceRpcAddr != null) {
            String bindHost = nn.getServiceRpcServerBindHost(conf);
            if (bindHost == null) {
                bindHost = serviceRpcAddr.getHostName();
            }
            LOG.info("Service RPC server is binding to " + bindHost + ":" +
                    serviceRpcAddr.getPort());

            // ServiceRpcServer Handler 线程个数 默认值 10
            int serviceHandlerCount =
                    conf.getInt(DFS_NAMENODE_SERVICE_HANDLER_COUNT_KEY,
                            DFS_NAMENODE_SERVICE_HANDLER_COUNT_DEFAULT);

            // 创建 ServiceRpcServer 并绑定一堆协议
            serviceRpcServer = new RPC.Builder(conf)
                    .setProtocol(

org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
                    .setInstance(clientNNPbService)
                    .setBindAddress(bindHost)
                    .setPort(serviceRpcAddr.getPort())
                    .setNumHandlers(serviceHandlerCount)
                    .setVerbose(false)
                    .setSecretManager(namesystem.getDelegationTokenSecretManager())
                    .build();

            // Add all the RPC protocols that the namenode implements
            DFSUtil.addPBProtocol(conf, HAServiceProtocolPB.class, haPbService,
                    serviceRpcServer);
            DFSUtil.addPBProtocol(conf, ReconfigurationProtocolPB.class,
                    reconfigurationPbService, serviceRpcServer);
            DFSUtil.addPBProtocol(conf, NamenodeProtocolPB.class, NNPbService,
                    serviceRpcServer);
            DFSUtil.addPBProtocol(conf, DatanodeProtocolPB.class, dnProtoPbService,
                    serviceRpcServer);
            DFSUtil.addPBProtocol(conf, RefreshAuthorizationPolicyProtocolPB.class,
                    refreshAuthService, serviceRpcServer);
            DFSUtil.addPBProtocol(conf, RefreshUserMappingsProtocolPB.class,
                    refreshUserMappingService, serviceRpcServer);
            // We support Refreshing call queue here in case the client RPC queue is full
            DFSUtil.addPBProtocol(conf, RefreshCallQueueProtocolPB.class,
                    refreshCallQueueService, serviceRpcServer);
            DFSUtil.addPBProtocol(conf, GenericRefreshProtocolPB.class,
                    genericRefreshService, serviceRpcServer);
```

```
            DFSUtil.addPBProtocol(conf, GetUserMappingsProtocolPB.class,
                    getUserMappingService, serviceRpcServer);
            DFSUtil.addPBProtocol(conf, TraceAdminProtocolPB.class,
                    traceAdminService, serviceRpcServer);

            // Update the address with the correct port
            InetSocketAddress listenAddr = serviceRpcServer.getListenerAddress();
            serviceRPCAddress = new InetSocketAddress(
                    serviceRpcAddr.getHostName(), listenAddr.getPort());

            // set key = dfs.namenode.servicerpc-address    value = hj101:53310
            nn.setRpcServiceServerAddress(conf, serviceRPCAddress);
        } else {
            serviceRpcServer = null;
            serviceRPCAddress = null;
        }

        // 类似与 ServiceRpcServer
        // 默认返回 null 如果在 hdfs-site.xml 配置
dfs.namenode.lifeline.rpc-address.mycluster.nn1 = xxx
        InetSocketAddress lifelineRpcAddr = nn.getLifelineRpcServerAddress(conf);
        if (lifelineRpcAddr != null) {
            RPC.setProtocolEngine(conf, HAServiceProtocolPB.class,
                    ProtobufRpcEngine.class);
            String bindHost = nn.getLifelineRpcServerBindHost(conf);
            if (bindHost == null) {
                bindHost = lifelineRpcAddr.getHostName();
            }
            LOG.info("Lifeline RPC server is binding to {}:{}", bindHost,
                    lifelineRpcAddr.getPort());

            int lifelineHandlerCount = conf.getInt(
                    DFS_NAMENODE_LIFELINE_HANDLER_COUNT_KEY, 0);
            if (lifelineHandlerCount <= 0) {
                float lifelineHandlerRatio = conf.getFloat(
                        DFS_NAMENODE_LIFELINE_HANDLER_RATIO_KEY,
                        DFS_NAMENODE_LIFELINE_HANDLER_RATIO_DEFAULT);
                lifelineHandlerCount = Math.max(
                        (int) (handlerCount * lifelineHandlerRatio), 1);
            }
            lifelineRpcServer = new RPC.Builder(conf)
                    .setProtocol(HAServiceProtocolPB.class)
                    .setInstance(haPbService)
                    .setBindAddress(bindHost)
```

```
                .setPort(lifelineRpcAddr.getPort())
                .setNumHandlers(lifelineHandlerCount)
                .setVerbose(false)
                .setSecretManager(namesystem.getDelegationTokenSecretManager())
                .build();

        DFSUtil.addPBProtocol(conf, DatanodeLifelineProtocolPB.class,
                lifelineProtoPbService, lifelineRpcServer);

        // Update the address with the correct port
        InetSocketAddress listenAddr = lifelineRpcServer.getListenerAddress();
        lifelineRPCAddress = new InetSocketAddress(lifelineRpcAddr.getHostName(),
                listenAddr.getPort());
        nn.setRpcLifelineServerAddress(conf, lifelineRPCAddress);
    } else {
        lifelineRpcServer = null;
        lifelineRPCAddress = null;
    }

    // 返回 hj101:8020
    InetSocketAddress rpcAddr = nn.getRpcServerAddress(conf);
    // null
    String bindHost = nn.getRpcServerBindHost(conf);
    if (bindHost == null) {
        bindHost = rpcAddr.getHostName();
    }
    // RPC server is binding to hj101:8020
    LOG.info("RPC server is binding to " + bindHost + ":" + rpcAddr.getPort());

    // 创建 RPC Server 并绑定一推协议
    clientRpcServer = new RPC.Builder(conf)
            .setProtocol(

org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
            .setInstance(clientNNPbService)
            .setBindAddress(bindHost)
            .setPort(rpcAddr.getPort())
            .setNumHandlers(handlerCount)
            .setVerbose(false)
            .setSecretManager(namesystem.getDelegationTokenSecretManager())
            .setAlignmentContext(new GlobalStateIdContext(namesystem))
            .build();

    // Add all the RPC protocols that the namenode implements
```

```
            DFSUtil.addPBProtocol(conf, HAServiceProtocolPB.class, haPbService,
                    clientRpcServer);
            DFSUtil.addPBProtocol(conf, ReconfigurationProtocolPB.class,
                    reconfigurationPbService, clientRpcServer);
            DFSUtil.addPBProtocol(conf, NamenodeProtocolPB.class, NNPbService,
                    clientRpcServer);
            DFSUtil.addPBProtocol(conf, DatanodeProtocolPB.class, dnProtoPbService,
                    clientRpcServer);
            DFSUtil.addPBProtocol(conf, RefreshAuthorizationPolicyProtocolPB.class,
                    refreshAuthService, clientRpcServer);
            DFSUtil.addPBProtocol(conf, RefreshUserMappingsProtocolPB.class,
                    refreshUserMappingService, clientRpcServer);
            DFSUtil.addPBProtocol(conf, RefreshCallQueueProtocolPB.class,
                    refreshCallQueueService, clientRpcServer);
            DFSUtil.addPBProtocol(conf, GenericRefreshProtocolPB.class,
                    genericRefreshService, clientRpcServer);
            DFSUtil.addPBProtocol(conf, GetUserMappingsProtocolPB.class,
                    getUserMappingService, clientRpcServer);
            DFSUtil.addPBProtocol(conf, TraceAdminProtocolPB.class,
                    traceAdminService, clientRpcServer);

            // set service-level authorization security policy
            if (serviceAuthEnabled = conf.getBoolean(
                                    CommonConfigurationKeys.HADOOP_SECURITY_AUTHORIZATION,
false)) {
                clientRpcServer.refreshServiceAcl(conf, new HDFSPolicyProvider());
                if (serviceRpcServer != null) {
                    serviceRpcServer.refreshServiceAcl(conf, new HDFSPolicyProvider());
                }
                if (lifelineRpcServer != null) {
                    lifelineRpcServer.refreshServiceAcl(conf, new HDFSPolicyProvider());
                }
            }

            // The rpc-server port can be ephemeral... ensure we have the correct info
            InetSocketAddress listenAddr = clientRpcServer.getListenerAddress();
            clientRpcAddress = new InetSocketAddress(
                    rpcAddr.getHostName(), listenAddr.getPort());
            // set fs.defaultFS = hdfs://hj101:8020
            nn.setRpcServerAddress(conf, clientRpcAddress);

            minimumDataNodeVersion = conf.get(

DFSConfigKeys.DFS_NAMENODE_MIN_SUPPORTED_DATANODE_VERSION_KEY,
```

```
DFSConfigKeys.DFS_NAMENODE_MIN_SUPPORTED_DATANODE_VERSION_DEFAULT);

        defaultECPolicyName = conf.getTrimmed(
                DFSConfigKeys.DFS_NAMENODE_EC_SYSTEM_DEFAULT_POLICY,
                DFSConfigKeys.DFS_NAMENODE_EC_SYSTEM_DEFAULT_POLICY_DEFAULT);

        // Set terse exception whose stack trace won't be logged
        clientRpcServer.addTerseExceptions(SafeModeException.class,
                FileNotFoundException.class,
                HadoopIllegalArgumentException.class,
                FileAlreadyExistsException.class,
                InvalidPathException.class,
                ParentNotDirectoryException.class,
                UnresolvedLinkException.class,
                AlreadyBeingCreatedException.class,
                QuotaExceededException.class,
                RecoveryInProgressException.class,
                AccessControlException.class,
                InvalidToken.class,
                LeaseExpiredException.class,
                NSQuotaExceededException.class,
                DSQuotaExceededException.class,
                QuotaByStorageTypeExceededException.class,
                AclException.class,
                FSLimitException.PathComponentTooLongException.class,
                FSLimitException.MaxDirectoryItemsExceededException.class);

        clientRpcServer.addSuppressedLoggingExceptions(StandbyException.class,
                UnresolvedPathException.class);

        clientRpcServer.setTracer(nn.tracer);
        if (serviceRpcServer != null) {
            serviceRpcServer.setTracer(nn.tracer);
        }
        if (lifelineRpcServer != null) {
            lifelineRpcServer.setTracer(nn.tracer);
        }
        int[] auxiliaryPorts =
                conf.getInts(DFS_NAMENODE_RPC_ADDRESS_AUXILIARY_KEY);
        if (auxiliaryPorts != null && auxiliaryPorts.length != 0) {
            for (int auxiliaryPort : auxiliaryPorts) {
                this.clientRpcServer.addAuxiliaryListener(auxiliaryPort);
            }
```

```
        }
    }
```

3.3.4.3.1 创建 ServiceRpcServer 并绑定一堆协议

```
```

3.3.4.3.2 创建 ClientRpcServer 并绑定一推协议

```
```

3.3.4.4 启动公共组件

```
/**
     * Start the services common to active and standby states
     */
    private void startCommonServices(Configuration conf) throws IOException {

        // 启动 FSNamesystem 公共服务
        namesystem.startCommonServices(conf, haContext);

        registerNNSMXBean();
        if (NamenodeRole.NAMENODE != role) {
            startHttpServer(conf);
            httpServer.setNameNodeAddress(getNameNodeAddress());
            httpServer.setFSImage(getFSImage());
        }

        // 启动 NameNodeRpcServer (clientRpcServer 或者 serviceRpcServer)
        rpcServer.start();
        try {
            plugins = conf.getInstances(DFS_NAMENODE_PLUGINS_KEY,
                    ServicePlugin.class);
        } catch (RuntimeException e) {
            String pluginsValue = conf.get(DFS_NAMENODE_PLUGINS_KEY);
            LOG.error("Unable to load NameNode plugins. Specified list of plugins: " +
                    pluginsValue, e);
            throw e;
        }
        for (ServicePlugin p : plugins) {
            try {
                p.start(this);
            } catch (Throwable t) {
                LOG.warn("ServicePlugin " + p + " could not be started", t);
            }
```

```
        }
        // NameNode RPC up at: hj101/192.168.2.101:8020
        LOG.info(getRole() + " RPC up at: " + getNameNodeAddress());

        if (rpcServer.getServiceRpcAddress() != null) {
            LOG.info(getRole() + " service RPC up at: "
                    + rpcServer.getServiceRpcAddress());
        }
    }
```

3.3.4.4.1 启动 FSNamesystem 公共服务

```
/**
     * Start services common to both active and standby states
     */
    void startCommonServices(Configuration conf, HAContext haContext) throws IOException {
        this.registerMBean(); // register the MBean for the FSNamesystemState
        writeLock();
        this.haContext = haContext;
        try {
            // 创建资源检查对象 NameNodeResourceChecker
            nnResourceChecker = new NameNodeResourceChecker(conf);
            // 判断 NameNode 元数据存储目录磁盘空间是否大于 100 MB
            checkAvailableResources();

            assert !blockManager.isPopulatingReplQueues();
            StartupProgress prog = NameNode.getStartupProgress();
            prog.beginPhase(Phase.SAFEMODE);

            // 根据 NameNode 元数据计算 completeBlock 个数
            long completeBlocksTotal = getCompleteBlocksTotal();

            prog.setTotal(Phase.SAFEMODE, STEP_AWAITING_REPORTED_BLOCKS,
                    completeBlocksTotal);

            // 激活 BlockManager
            blockManager.activate(conf, completeBlocksTotal);
        } finally {
            writeUnlock("startCommonServices");
        }

        registerMXBean();
        DefaultMetricsSystem.instance().register(this);
        if (inodeAttributeProvider != null) {
            inodeAttributeProvider.start();
```

```
                dir.setINodeAttributeProvider(inodeAttributeProvider);
            }
            snapshotManager.registerMXBean();

            // hj101:53310
            InetSocketAddress serviceAddress = NameNode.getServiceAddress(conf, true);
            this.nameNodeHostName = (serviceAddress != null) ?
                    serviceAddress.getHostName() : "";
        }
```

3.3.4.4.1.1 创建资源检查对象 NameNodeResourceChecker

```
/**
         * Create a NameNodeResourceChecker, which will check the edits dirs and any
         * additional dirs to check set in <code>conf</code>.
         */
        public NameNodeResourceChecker(Configuration conf) throws IOException {
            this.conf = conf;

            // 初始化卷 (NameNode 元数据存储目录)
            volumes = new HashMap<String, CheckedVolume>();

            // 默认 100MB
            duReserved = conf.getLong(DFSConfigKeys.DFS_NAMENODE_DU_RESERVED_KEY,
                    DFSConfigKeys.DFS_NAMENODE_DU_RESERVED_DEFAULT);

            // key = dfs.namenode.resource.checked.volumes 默认值空
            Collection<URI> extraCheckedVolumes = Util.stringCollectionAsURIs(conf
                    .getTrimmedStringCollection(DFSConfigKeys.DFS_NAMENODE_CHECKED_VOL
UMES_KEY));

            // file:///tmp/hadoop/dfs/name
            Collection<URI> localEditDirs = Collections2.filter(
                    // 1 qjournal://hj101:8485;hj102:8485;hj103:8485/mycluster
                    // 2 file:///tmp/hadoop/dfs/name
                    FSNamesystem.getNamespaceEditsDirs(conf),
                    new Predicate<URI>() {
                        @Override
                        public boolean apply(URI input) {
                            if (input.getScheme().equals(NNStorage.LOCAL_URI_SCHEME)) {
                                return true;
                            }
                            return false;
                        }
                    });
```

```
        // Add all the local edits dirs, marking some as required if they are
        // configured as such.
        for (URI editsDirToCheck : localEditDirs) {
            // 往 volumes 添加检查目录
            addDirToCheck(editsDirToCheck,
                    // true
                    FSNamesystem.getRequiredNamespaceEditsDirs(conf).contains(
                            editsDirToCheck));
        }

        // All extra checked volumes are marked "required"
        for (URI extraDirToCheck : extraCheckedVolumes) {
            addDirToCheck(extraDirToCheck, true);
        }

        // 1
        minimumRedundantVolumes = conf.getInt(
                DFSConfigKeys.DFS_NAMENODE_CHECKED_VOLUMES_MINIMUM_KEY,
                DFSConfigKeys.DFS_NAMENODE_CHECKED_VOLUMES_MINIMUM_DEFAULT);
    }
```

```
/**
     * Add the volume of the passed-in directory to the list of volumes to check.
     * If <code>required</code> is true, and this volume is already present, but
     * is marked redundant, it will be marked required. If the volume is already
     * present but marked required then this method is a no-op.
     *
     * @param directoryToCheck The directory whose volume will be checked for available
space.
     */
    private void addDirToCheck(URI directoryToCheck, boolean required)
            throws IOException {
        File dir = new File(directoryToCheck.getPath());
        if (!dir.exists()) {
            throw new IOException("Missing directory " + dir.getAbsolutePath());
        }

        CheckedVolume newVolume = new CheckedVolume(dir, required);
        CheckedVolume volume = volumes.get(newVolume.getVolume());
        if (volume == null || !volume.isRequired()) {
            volumes.put(newVolume.getVolume(), newVolume);
        }
    }
```

3.3.4.4.1.2 判断 NameNode 元数据存储目录磁盘空间是否大于 100 MB

```
/**
     * Perform resource checks and cache the results.
     */
    void checkAvailableResources() {
        long resourceCheckTime = monotonicNow();
        Preconditions.checkState(nnResourceChecker != null,
                "nnResourceChecker not initialized");
        // 判断 NameNode 元数据存储目录磁盘空间是否大于 100 MB
        hasResourcesAvailable = nnResourceChecker.hasAvailableDiskSpace();
        resourceCheckTime = monotonicNow() - resourceCheckTime;
        NameNode.getNameNodeMetrics().addResourceCheckTime(resourceCheckTime);
    }
```

```
    /**
     * Return true if disk space is available on at least one of the configured
     * redundant volumes, and all of the configured required volumes.
     *
     * @return True if the configured amount of disk space is available on at
     * least one redundant volume and all of the required volumes, false
     * otherwise.
     */
    public boolean hasAvailableDiskSpace() {
        // 往下追
        return NameNodeResourcePolicy.areResourcesAvailable(volumes.values(),
                minimumRedundantVolumes);
    }
```

```
/**
   * Return true if and only if there are sufficient NN
   * resources to continue logging edits.
   *
   * @param resources the collection of resources to check.
   * @param minimumRedundantResources the minimum number of redundant resources
   *            required to continue operation.
   * @return true if and only if there are sufficient NN resources to
   *             continue logging edits.
   */
  static boolean areResourcesAvailable(
      Collection<? extends CheckableNameNodeResource> resources,
      int minimumRedundantResources) {

    // TODO: workaround:
    // - during startup, if there are no edits dirs on disk, then there is
```

```
        // a call to areResourcesAvailable() with no dirs at all, which was
        // previously causing the NN to enter safemode
        if (resources.isEmpty()) {
          return true;
        }

        int requiredResourceCount = 0;
        int redundantResourceCount = 0;
        int disabledRedundantResourceCount = 0;
        // 遍历 NamNode 元数据存储目录
        for (CheckableNameNodeResource resource : resources) {
          if (!resource.isRequired()) {
            redundantResourceCount++;
            if (!resource.isResourceAvailable()) {
              disabledRedundantResourceCount++;
            }
          } else {
            requiredResourceCount++;
            // 判断 NamNode 元数据存储目录存储空间是否大于等于 100 MB
            if (!resource.isResourceAvailable()) {
              // Short circuit - a required resource is not available.
              return false;
            }
          }
        }

        if (redundantResourceCount == 0) {
          // If there are no redundant resources, return true if there are any
          // required resources available.
          return requiredResourceCount > 0;
        } else {
          return redundantResourceCount - disabledRedundantResourceCount >=
              minimumRedundantResources;
        }
      }
    }
```

3.3.4.4.1.3 根据 NameNode 元数据计算 completeBlock 个数

```
    /**
     * Get the total number of COMPLETE blocks in the system.
     * For safe mode only complete blocks are counted.
     * This is invoked only during NN startup and checkpointing.
     */
    public long getCompleteBlocksTotal() {
```

```
            // Calculate number of blocks under construction
            long numUCBlocks = 0;
            readLock();
            try {
                // 获取 ConstructionBlock 个数
                numUCBlocks = leaseManager.getNumUnderConstructionBlocks();
                // CompleteBlock = BlocksTotal - ConstructionBlock
                return getBlocksTotal() - numUCBlocks;
            } finally {
                readUnlock("getCompleteBlocksTotal");
            }
        }
```

```
/**
     * This method iterates through all the leases and counts the number of blocks
     * which are not COMPLETE. The FSNamesystem read lock MUST be held before
     * calling this method.
     */
    synchronized long getNumUnderConstructionBlocks() {
      assert this.fsnamesystem.hasReadLock() : "The FSNamesystem read lock wasn't"
        + "acquired before counting under construction blocks";
      long numUCBlocks = 0;
      // 遍历文件
      for (Long id : getINodeIdWithLeases()) {
        // 从 FSDirectory 获取 Inode 表示一个文件
        INode inode = fsnamesystem.getFSDirectory().getInode(id);
        if (inode == null) {
          // The inode could have been deleted after getINodeIdWithLeases() is
          // called, check here, and ignore it if so
          LOG.warn("Failed to find inode {} in getNumUnderConstructionBlocks().",
              id);
          continue;
        }
        final INodeFile cons = inode.asFile();
        // 判断该文件是否处于 UnderConstruction
        if (!cons.isUnderConstruction()) {
          LOG.warn("The file {} is not under construction but has lease.",
              cons.getFullPathName());
          continue;
        }
        // 获取该文件的 Block 个数
        BlockInfo[] blocks = cons.getBlocks();
        if(blocks == null) {
```

```
            continue;
        }
        for(BlockInfo b : blocks) {
            if(!b.isComplete()) {
                // 该文件的其中一个 Block 处于 UnderConstruction
                numUCBlocks++;
            }
        }
    }
    LOG.info("Number of blocks under construction: {}", numUCBlocks);
    return numUCBlocks;
  }
```

```
 /**
     * Get the total number of blocks in the system.
     */
    @Override // FSNamesystemMBean
    @Metric
    public long getBlocksTotal() {
        return blockManager.getTotalBlocks();
    }
```

3.3.4.4.1.4 激活 BlockManager

```
public void activate(Configuration conf, long blockTotal) {

        // 启动 PendingReconstructionMonitor 线程
        pendingReconstruction.start();

        // 激活 DataNodeManager
        datanodeManager.activate(conf);

        // 启动 RedundancyMonitor 线程
        this.redundancyThread.setName("RedundancyMonitor");
        this.redundancyThread.start();

        // 启动 StorageInfoDefragmenter 线程
        storageInfoDefragmenterThread.setName("StorageInfoMonitor");
        storageInfoDefragmenterThread.start();

        // 启动 BlockReportProcessingThread 线程
        this.blockReportThread.start();

        mxBeanName = MBeans.register("NameNode", "BlockStats", this);
```

```
        // Initialize the safe mode information
        // 检查 DataNode 的 Block 上报情况 可能进入安全模式
        bmSafeMode.activate(blockTotal);
    }
```

3.3.4.4.1.4.1 启动 PendingReconstructionMonitor 线程

```
/*
   * A periodic thread that scans for blocks that never finished
   * their reconstruction request.
   */
  class PendingReconstructionMonitor implements Runnable {
    @Override
    public void run() {
      while (fsRunning) {
        long period = Math.min(DEFAULT_RECHECK_INTERVAL, timeout);
        try {
          pendingReconstructionCheck();
          Thread.sleep(period);
        } catch (InterruptedException ie) {
          LOG.debug("PendingReconstructionMonitor thread is interrupted.", ie);
        }
      }
    }
```

3.3.4.4.1.4.2 激活 DataNodeManager

```
    void activate(final Configuration conf) {
        // start the DataNode admin monitor thread
        datanodeAdminManager.activate(conf);

        // 激活 HeartbeatManager
        heartbeatManager.activate();
    }
```

3.3.4.4.1.4.2.1 启动 DataNode 管理者线程

org.apache.hadoop.hdfs.server.blockmanagement.DatanodeAdminManager

```
/**
   * Start the DataNode admin monitor thread.
   * @param conf
   */
  void activate(Configuration conf) {
    final int intervalSecs = (int) conf.getTimeDuration(
        DFSConfigKeys.DFS_NAMENODE_DECOMMISSION_INTERVAL_KEY,
```

```
            DFSConfigKeys.DFS_NAMENODE_DECOMMISSION_INTERVAL_DEFAULT,
            TimeUnit.SECONDS);
        checkArgument(intervalSecs >= 0, "Cannot set a negative " +
            "value for " + DFSConfigKeys.DFS_NAMENODE_DECOMMISSION_INTERVAL_KEY);

        int blocksPerInterval = conf.getInt(
            DFSConfigKeys.DFS_NAMENODE_DECOMMISSION_BLOCKS_PER_INTERVAL_KEY,
            DFSConfigKeys.DFS_NAMENODE_DECOMMISSION_BLOCKS_PER_INTERVAL_DEFAULT);

        final String deprecatedKey =
            "dfs.namenode.decommission.nodes.per.interval";
        final String strNodes = conf.get(deprecatedKey);
        if (strNodes != null) {
          LOG.warn("Deprecated configuration key {} will be ignored.",
              deprecatedKey);
          LOG.warn("Please update your configuration to use {} instead.",
              DFSConfigKeys.DFS_NAMENODE_DECOMMISSION_BLOCKS_PER_INTERVAL_KEY);
        }

        checkArgument(blocksPerInterval > 0,
            "Must set a positive value for "
            + DFSConfigKeys.DFS_NAMENODE_DECOMMISSION_BLOCKS_PER_INTERVAL_KEY);

        final int maxConcurrentTrackedNodes = conf.getInt(

DFSConfigKeys.DFS_NAMENODE_DECOMMISSION_MAX_CONCURRENT_TRACKED_NODES,
            DFSConfigKeys
                .DFS_NAMENODE_DECOMMISSION_MAX_CONCURRENT_TRACKED_NODES_DEFA
ULT);
        checkArgument(maxConcurrentTrackedNodes >= 0, "Cannot set a negative " +
            "value for "
            +
DFSConfigKeys.DFS_NAMENODE_DECOMMISSION_MAX_CONCURRENT_TRACKED_NODES);

        // 定时调度 Monitor 线程
        monitor = new Monitor(blocksPerInterval, maxConcurrentTrackedNodes);
        executor.scheduleAtFixedRate(monitor, intervalSecs, intervalSecs,
            TimeUnit.SECONDS);

        LOG.debug("Activating DatanodeAdminManager with interval {} seconds, " +
                "{} max blocks per interval, " +
                "{} max concurrently tracked nodes.", intervalSecs,
            blocksPerInterval, maxConcurrentTrackedNodes);
    }
```

```
@Override
    public void run() {
      LOG.debug("DatanodeAdminMonitor is running.");
      if (!namesystem.isRunning()) {
        LOG.info("Namesystem is not running, skipping " +
            "decommissioning/maintenance checks.");
        return;
      }
      // Reset the checked count at beginning of each iteration
      numBlocksChecked = 0;
      numBlocksCheckedPerLock = 0;
      numNodesChecked = 0;
      // Check decommission or maintenance progress.
      namesystem.writeLock();
      try {
        processPendingNodes();
        check();
      } catch (Exception e) {
        LOG.warn("DatanodeAdminMonitor caught exception when processing node.",
            e);
      } finally {
        namesystem.writeUnlock();
      }
      if (numBlocksChecked + numNodesChecked > 0) {
        LOG.info("Checked {} blocks and {} nodes this tick", numBlocksChecked,
            numNodesChecked);
      }
    }
```

```
/**
     * Pop datanodes off the pending list and into decomNodeBlocks,
     * subject to the maxConcurrentTrackedNodes limit.
     */
    private void processPendingNodes() {
      while (!pendingNodes.isEmpty() &&
          (maxConcurrentTrackedNodes == 0 ||
          outOfServiceNodeBlocks.size() < maxConcurrentTrackedNodes)) {
        outOfServiceNodeBlocks.put(pendingNodes.poll(), null);
      }
    }
```

3.3.4.4.1.4.2.2 激活 HeartbeatManager 并检查 DataNode 的心跳

```
    void activate() {
```

```
        // 启动 DataNode 心跳检查服务 也即调用 Monitor 线程的 run()
        heartbeatThread.start();
    }
```

org.apache.hadoop.hdfs.server.blockmanagement.HeartbeatManager

```
/**
     * Periodically check heartbeat and update block key
     */
    private class Monitor implements Runnable {
        private long lastHeartbeatCheck;
        private long lastBlockKeyUpdate;

        @Override
        public void run() {
            while (namesystem.isRunning()) {
                // 重设 StopWatch
                restartHeartbeatStopWatch();
                try {
                    final long now = Time.monotonicNow();
                    // 0 + 30s (每隔 30s 执行 heartbeatCheck)
                    if (lastHeartbeatCheck + heartbeatRecheckInterval < now) {
                        heartbeatCheck();
                        lastHeartbeatCheck = now;
                    }
                    if (blockManager.shouldUpdateBlockKey(now - lastBlockKeyUpdate)) {
                        synchronized (HeartbeatManager.this) {
                            for (DatanodeDescriptor d : datanodes) {
                                d.setNeedKeyUpdate(true);
                            }
                        }
                        lastBlockKeyUpdate = now;
                    }
                } catch (Exception e) {
                    LOG.error("Exception while checking heartbeat", e);
                }
                try {
                    Thread.sleep(5000);  // 5 seconds
                } catch (InterruptedException ignored) {
                }
                // avoid declaring nodes dead for another cycle if a GC pause lasts
                // longer than the node recheck interval
                if (shouldAbortHeartbeatCheck(-5000)) {
                    LOG.warn("Skipping next heartbeat scan due to excessive pause");
                    lastHeartbeatCheck = Time.monotonicNow();
```

```
                    }
                }
            }
        }
    }
```

```
    /**
     * Check if there are any expired heartbeats, and if so,
     * whether any blocks have to be re-replicated.
     * While removing dead datanodes, make sure that only one datanode is marked
     * dead at a time within the synchronized section. Otherwise, a cascading
     * effect causes more datanodes to be declared dead.
     * Check if there are any failed storage and if so,
     * Remove all the blocks on the storage. It also covers the following less
     * common scenarios. After DatanodeStorage is marked FAILED, it is still
     * possible to receive IBR for this storage.
     * 1) DN could deliver IBR for failed storage due to its implementation.
     * a) DN queues a pending IBR request.
     * b) The storage of the block fails.
     * c) DN first sends HB, NN will mark the storage FAILED.
     * d) DN then sends the pending IBR request.
     * 2) SBN processes block request from pendingDNMessages.
     * It is possible to have messages in pendingDNMessages that refer
     * to some failed storage.
     * a) SBN receives a IBR and put it in pendingDNMessages.
     * b) The storage of the block fails.
     * c) Edit log replay get the IBR from pendingDNMessages.
     * Alternatively, we can resolve these scenarios with the following approaches.
     * A. Make sure DN don't deliver IBR for failed storage.
     * B. Remove all blocks in PendingDataNodeMessages for the failed storage
     * when we remove all blocks from BlocksMap for that storage.
     */
    @VisibleForTesting
    void heartbeatCheck() {
        // 获取 DatanodeManager
        final DatanodeManager dm = blockManager.getDatanodeManager();
        // It's OK to check safe mode w/o taking the lock here, we re-check
        // for safe mode after taking the lock before removing a datanode.
        if (namesystem.isInStartupSafeMode()) {
            return;
        }
        boolean allAlive = false;
        while (!allAlive) {
            // locate the first dead node.
```

```
                DatanodeDescriptor dead = null;

                // locate the first failed storage that isn't on a dead node.
                DatanodeStorageInfo failedStorage = null;

                // check the number of stale nodes
                int numOfStaleNodes = 0;
                int numOfStaleStorages = 0;
                synchronized (this) {
                    // 遍历每个 DataNode
                    for (DatanodeDescriptor d : datanodes) {
                        // check if an excessive GC pause has occurred
                        if (shouldAbortHeartbeatCheck(0)) {
                            return;
                        }
                        if (dead == null &&
                                // 判断上一次 DataNode 心跳事件是否小于当前时间 -
630s
                                // 如果是 认为该 DataNode 死了
                                dm.isDatanodeDead(d)
                        ) {
                            // 心跳超时计数器加 1
                            stats.incrExpiredHeartbeats();
                            dead = d;
                        }
                        if (d.isStale(dm.getStaleInterval())) {
                            numOfStaleNodes++;
                        }
                        DatanodeStorageInfo[] storageInfos = d.getStorageInfos();
                        for (DatanodeStorageInfo storageInfo : storageInfos) {
                            if (storageInfo.areBlockContentsStale()) {
                                numOfStaleStorages++;
                            }

                            if (failedStorage == null &&
                                    storageInfo.areBlocksOnFailedStorage() &&
                                    d != dead) {
                                failedStorage = storageInfo;
                            }
                        }

                    }

                    // Set the number of stale nodes in the DatanodeManager
```

```
                dm.setNumStaleNodes(numOfStaleNodes);
                dm.setNumStaleStorages(numOfStaleStorages);
            }

            allAlive = dead == null && failedStorage == null;
            if (!allAlive && namesystem.isInStartupSafeMode()) {
                return;
            }
            if (dead != null) {
                // acquire the fsnamesystem lock, and then remove the dead node.
                namesystem.writeLock();
                try {
                    // 移除死掉的 DataNode
                    dm.removeDeadDatanode(dead, !dead.isMaintenance());
                } finally {
                    namesystem.writeUnlock();
                }
            }
            if (failedStorage != null) {
                // acquire the fsnamesystem lock, and remove blocks on the storage.
                namesystem.writeLock();
                try {
                    blockManager.removeBlocksAssociatedTo(failedStorage);
                } finally {
                    namesystem.writeUnlock();
                }
            }
        }
    }
```

```
  /** Is the datanode dead? */
    boolean isDatanodeDead(DatanodeDescriptor node) {
        return (node.getLastUpdateMonotonic() <
                (monotonicNow() - heartbeatExpireInterval));
    }
```

```
/** Remove a dead datanode. */
    void removeDeadDatanode(final DatanodeID nodeID,
                            boolean removeBlocksFromBlockMap) {
        DatanodeDescriptor d;
        try {
            // 获取 DatanodeDescriptor
            d = getDatanode(nodeID);
        } catch (IOException e) {
```

```
                d = null;
            }
            if (d != null &&
                    // 二次判断是否真的挂了
                    isDatanodeDead(d)
            ) {
                NameNode.stateChangeLog.info(
                        "BLOCK* removeDeadDatanode: lost heartbeat from " + d
                                + ", removeBlocksFromBlockMap " +
removeBlocksFromBlockMap);
                // 移除挂了的 DataNode
                removeDatanode(d, removeBlocksFromBlockMap);
            }
        }
```

3.3.4.4.1.4.3 启动 RedundancyMonitor 线程

```
/**
     * Periodically calls computeBlockRecoveryWork().
     */
    private class RedundancyMonitor implements Runnable {

        @Override
        public void run() {
            while (namesystem.isRunning()) {
                try {
                    // Process recovery work only when active NN is out of safe mode.
                    if (isPopulatingReplQueues()) {
                        computeDatanodeWork();
                        processPendingReconstructions();
                        rescanPostponedMisreplicatedBlocks();
                    }
                    TimeUnit.MILLISECONDS.sleep(redundancyRecheckIntervalMs);
                } catch (Throwable t) {
                    if (!namesystem.isRunning()) {
                        LOG.info("Stopping RedundancyMonitor.");
                        if (!(t instanceof InterruptedException)) {
                            LOG.info("RedundancyMonitor received an exception"
                                    + " while shutting down.", t);
                        }
                        break;
                    } else if (!checkNSRunning && t instanceof InterruptedException) {
                        LOG.info("Stopping RedundancyMonitor for testing.");
                        break;
                    }
```

```
                    LOG.error("RedundancyMonitor thread received Runtime exception. ",
                            t);
                    terminate(1, t);
                }
            }
        }
    }
```

3.3.4.4.1.4.4 启动 StorageInfoDefragmenter 线程

```
/**
     * Runnable that monitors the fragmentation of the StorageInfo TreeSet and
     * compacts it when it falls under a certain threshold.
     */
    private class StorageInfoDefragmenter implements Runnable {

        @Override
        public void run() {
            while (namesystem.isRunning()) {
                try {
                    // Check storage efficiency only when active NN is out of safe mode.
                    if (isPopulatingReplQueues()) {
                        scanAndCompactStorages();
                    }
                    Thread.sleep(storageInfoDefragmentInterval);
                } catch (Throwable t) {
                    if (!namesystem.isRunning()) {
                        LOG.info("Stopping thread.");
                        if (!(t instanceof InterruptedException)) {
                            LOG.info("Received an exception while shutting down.", t);
                        }
                        break;
                    } else if (!checkNSRunning && t instanceof InterruptedException) {
                        LOG.info("Stopping for testing.");
                        break;
                    }
                    LOG.error("Thread received Runtime exception.", t);
                    terminate(1, t);
                }
            }
        }
```

3.3.4.4.1.4.5 启动 BlockReportProcessingThread 线程

```
private class BlockReportProcessingThread extends Thread {
        private static final long MAX_LOCK_HOLD_MS = 4;
        private long lastFull = 0;

        private final BlockingQueue<Runnable> queue;

        BlockReportProcessingThread(int size) {
            super("Block report processor");
            // 1024
            queue = new ArrayBlockingQueue<>(size);
            setDaemon(true);
        }

        @Override
        public void run() {
            try {
                processQueue();
            } catch (Throwable t) {
                ExitUtil.terminate(1,
                        getName() + " encountered fatal exception: " + t);
            }
        }
```

```
private void processQueue() {
            while (namesystem.isRunning()) {
                NameNodeMetrics metrics = NameNode.getNameNodeMetrics();
                try {
                    Runnable action = queue.take();
                    // batch as many operations in the write lock until the queue
                    // runs dry, or the max lock hold is reached.
                    int processed = 0;
                    namesystem.writeLock();
                    metrics.setBlockOpsQueued(queue.size() + 1);
                    try {
                        long start = Time.monotonicNow();
                        do {
                            processed++;
                            action.run();
                            if (Time.monotonicNow() - start > MAX_LOCK_HOLD_MS) {
                                break;
                            }
                            action = queue.poll();
                        } while (action != null);
```

```
                        } finally {
                            namesystem.writeUnlock();
                            metrics.addBlockOpsBatched(processed - 1);
                        }
                    } catch (InterruptedException e) {
                        // ignore unless thread was specifically interrupted.
                        if (Thread.interrupted()) {
                            break;
                        }
                    }
                }
                queue.clear();
            }
```

3.3.4.4.1.4.6 检查 DataNode 的 Block 上报情况 可能进入安全模式

```
/**
     * Initialize the safe mode information.
     *
     * @param total initial total blocks
     */
    void activate(long total) {
        assert namesystem.hasWriteLock();
        assert status == BMSafeModeStatus.OFF;

        startTime = monotonicNow();
        // total = CompleteBlocks 个数 假设 1000
        setBlockTotal(total);

        // 判断离开安全模式
        if (areThresholdsMet()) {
            boolean exitResult = leaveSafeMode(false);
            Preconditions.checkState(exitResult, "Failed to leave safe mode.");
        } else {
            // enter safe mode
            // 进入安全模式
            status = BMSafeModeStatus.PENDING_THRESHOLD;

            initializeReplQueuesIfNecessary();
            reportStatus("STATE* Safe mode ON.", true);
            lastStatusReport = monotonicNow();
        }
    }
```

```
 /**
```

```
     * Set total number of blocks.
     */
    void setBlockTotal(long total) {
        assert namesystem.hasWriteLock();
        synchronized (this) {
            // 1000
            this.blockTotal = total;
            // 1000 * 0.999 = 999
            this.blockThreshold = (long) (total * threshold);
        }
        // 1000 * 0.999 = 999
        this.blockReplQueueThreshold = (long) (total * replQueueThreshold);
    }
```

```
/**
     * @return true if both block and datanode threshold are met else false.
     */
    private boolean areThresholdsMet() {
        assert namesystem.hasWriteLock();
        // Calculating the number of live datanodes is time-consuming
        // in large clusters. Skip it when datanodeThreshold is zero.
        int datanodeNum = 0;
        // 0
        // 总是返回 false
        if (datanodeThreshold > 0) {
            datanodeNum = blockManager.getDatanodeManager().getNumLiveDataNodes();
        }
        synchronized (this) {
            return
                    // blockSafe 等待 DataNode 上传 Block
                    // 如果 DataNode 还没有启动 blockSafe = 0 则返回 false
                    // 直到 DataNode 启动并上报其 Block 个数总和大于等于
blockThreshold (CompleteBlock)
                    blockSafe >= blockThreshold
                            // 总是返回 true
                            && datanodeNum >= datanodeThreshold;
        }
    }
```

3.3.4.4.2 启动 NameNodeRpcServer (clientRpcServer 或者 serviceRpcServer)

```
/**
     * Start client and service RPC servers.
     */
    void start() {
        // hadoop-rpc 标配启动
        clientRpcServer.start();
        if (serviceRpcServer != null) {
            // hadoop-rpc 标配启动
            serviceRpcServer.start();
        }
        if (lifelineRpcServer != null) {
            lifelineRpcServer.start();
        }
    }
```

3.3.5 NameNode 进入 Standby 状态

org.apache.hadoop.hdfs.server.namenode.ha.StandbyState

```
    @Override
    public void enterState(HAContext context) throws ServiceFailedException {
        try {
            // NameNode 进入 Standby 状态 启动 Standby 对应的服务
            context.startStandbyServices();
        } catch (IOException e) {
            throw new ServiceFailedException("Failed to start standby services", e);
        }
    }
```

3.3.5.1 启动 Standby 对应的服务

```
    @Override
        public void startStandbyServices() throws IOException {
            try {
                // NameNode 为 Standby 状态下 启动 FSNamesystem
                namesystem.startStandbyServices(getConf(),
                        state == NameNode.OBSERVER_STATE);
            } catch (Throwable t) {
                doImmediateShutdown(t);
            }
        }
```

3.3.5.1.1 NameNode 为 Standby 状态下 启动 FSNamesystem 相关服务

```
/**
     * Start services required in standby or observer state
     *
     * @throws IOException
     */
    void startStandbyServices(final Configuration conf, boolean isObserver)
            throws IOException {
        LOG.info("Starting services required for " +
                (isObserver ? "observer" : "standby") + " state");

        // !false = true
        if (!getFSImage().editLog.isOpenForRead()) {
            // During startup, we're already open for read.
            // 初始化 JournalNode 连接信息
            getFSImage().editLog.initSharedJournalsForRead();
        }
        blockManager.setPostponeBlocksFromFuture(true);

        // Disable quota checks while in standby.
        dir.disableQuotaChecks();

        // 创建 EditLogTailer
        editLogTailer = new EditLogTailer(this, conf);
        // 启动 EditLogTailerThread 线程
        editLogTailer.start();

        if (!isObserver && standbyShouldCheckpoint) {
            // 创建 StandbyCheckpointer
            standbyCheckpointer = new StandbyCheckpointer(conf, this);
            // 启动 CheckpointerThread 线程
            standbyCheckpointer.start();
        }
    }
```

3.3.5.1.1.1 初始化 JournalNode 连接信息

```
public synchronized void initSharedJournalsForRead() {
        if (state == State.OPEN_FOR_READING) {
            LOG.warn("Initializing shared journals for READ, already open for READ",
                    new Exception());
            return;
        }
```

```
        Preconditions.checkState(state == State.UNINITIALIZED ||
                state == State.CLOSED);

        // 初始化 JournalNode 连接信息
        initJournals(
                // 从配置中获取 key = dfs.namenode.shared.edits.dir
                // 一 般 在 hdfs-site.xml 配 置 比 如
qjournal://hj101:8485;hj102:8485;hj103:8485/mycluster
                this.sharedEditsDirs
        );

        state = State.OPEN_FOR_READING;
    }
```

```
private synchronized void initJournals(List<URI> dirs) {
        // 1
        int minimumRedundantJournals = conf.getInt(
                DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_KEY,
                DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_DEFAULT);

        synchronized (journalSetLock) {
            // 创建 JournalSet
            journalSet = new JournalSet(minimumRedundantJournals);

            for (URI u : dirs) {
                // true
                boolean required = FSNamesystem.getRequiredNamespaceEditsDirs(conf)
                        .contains(u);
                // false
                if (u.getScheme().equals(NNStorage.LOCAL_URI_SCHEME)) {
                    StorageDirectory sd = storage.getStorageDirectory(u);
                    if (sd != null) {
                        journalSet.add(new FileJournalManager(conf, sd, storage),
                                required, sharedEditsDirs.contains(u));
                    }
                } else {
                    //
                    journalSet.add(
                            // 创建 QuorumJournalManager
                            createJournal(u),
                            // true
                            required,
                            // true
                            sharedEditsDirs.contains(u));
```

```
                    }
                }
            }

            if (journalSet.isEmpty()) {
                LOG.error("No edits directories configured!");
            }
        }
```

3.3.5.1.1.1.1 创建 QuorumJournalManager

```
/**
     * Construct a custom journal manager.
     * The class to construct is taken from the configuration.
     *
     * @param uri Uri to construct
     * @return The constructed journal manager
     * @throws IllegalArgumentException if no class is configured for uri
     */
    @VisibleForTesting
    JournalManager createJournal(URI uri) {
        // 获取 org.apache.hadoop.hdfs.qjournal.client.QuorumJournalManager.class
        Class<? extends JournalManager> clazz
                = getJournalClass(conf, uri.getScheme());

        try {
            Constructor<? extends JournalManager> cons
                    = clazz.getConstructor(
                    Configuration.class,
                    URI.class,
                    NamespaceInfo.class,
                    String.class);
            // mycluster
            String nameServiceId = conf.get(DFSConfigKeys.DFS_NAMESERVICE_ID);
            // 调用 QuorumJournalManager 的构造函数 (四个参数的构造函数)
            return cons.newInstance(conf, uri, storage.getNamespaceInfo(),
                    nameServiceId);
        } catch (NoSuchMethodException ne) {
            try {
                Constructor<? extends JournalManager> cons
                        = clazz.getConstructor(Configuration.class, URI.class,
                        NamespaceInfo.class);
                return cons.newInstance(conf, uri, storage.getNamespaceInfo());
            } catch (Exception e) {
```

```
                    throw new IllegalArgumentException("Unable to construct journal, "
                            + uri, e);
                }
            } catch (Exception e) {
                throw new IllegalArgumentException("Unable to construct journal, "
                        + uri, e);
            }
        }
```

3.3.5.1.1.1.2 初始化 QuorumJournalManager

```
// 创建 QuorumJournalManager
    public QuorumJournalManager(Configuration conf,
                                URI uri,
                                NamespaceInfo nsInfo,
                                String nameServiceId) throws IOException {
        // 往下追
        this(conf, uri, nsInfo, nameServiceId, IPCLoggerChannel.FACTORY);
    }
```

```
QuorumJournalManager(Configuration conf,
                                URI uri, NamespaceInfo nsInfo, String nameServiceId,
                                AsyncLogger.Factory loggerFactory) throws IOException {
        Preconditions.checkArgument(conf != null, "must be configured");

        this.conf = conf;
        this.uri = uri;
        this.nsInfo = nsInfo;
        this.nameServiceId = nameServiceId;

        // 创建 AsyncLoggerSet
        this.loggers = new AsyncLoggerSet(
                // 解析 JournalNode 连接地址 创建 IPCLoggerChannel
                createLoggers(loggerFactory)
        );

        this.maxTxnsPerRpc =
                conf.getInt(QJM_RPC_MAX_TXNS_KEY, QJM_RPC_MAX_TXNS_DEFAULT);
        Preconditions.checkArgument(maxTxnsPerRpc > 0,
                "Must specify %s greater than 0!", QJM_RPC_MAX_TXNS_KEY);
        this.inProgressTailingEnabled = conf.getBoolean(
                DFSConfigKeys.DFS_HA_TAILEDITS_INPROGRESS_KEY,
                DFSConfigKeys.DFS_HA_TAILEDITS_INPROGRESS_DEFAULT);
        // Configure timeouts.
        this.startSegmentTimeoutMs = conf.getInt(
```

```
                DFSConfigKeys.DFS_QJOURNAL_START_SEGMENT_TIMEOUT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_START_SEGMENT_TIMEOUT_DEFAULT);
        this.prepareRecoveryTimeoutMs = conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_PREPARE_RECOVERY_TIMEOUT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_PREPARE_RECOVERY_TIMEOUT_DEFAULT);
        this.acceptRecoveryTimeoutMs = conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_ACCEPT_RECOVERY_TIMEOUT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_ACCEPT_RECOVERY_TIMEOUT_DEFAULT);
        this.finalizeSegmentTimeoutMs = conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_FINALIZE_SEGMENT_TIMEOUT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_FINALIZE_SEGMENT_TIMEOUT_DEFAULT);
        this.selectInputStreamsTimeoutMs = conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_SELECT_INPUT_STREAMS_TIMEOUT_KEY,

DFSConfigKeys.DFS_QJOURNAL_SELECT_INPUT_STREAMS_TIMEOUT_DEFAULT);
        this.getJournalStateTimeoutMs = conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_GET_JOURNAL_STATE_TIMEOUT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_GET_JOURNAL_STATE_TIMEOUT_DEFAULT);
        this.newEpochTimeoutMs = conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_NEW_EPOCH_TIMEOUT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_NEW_EPOCH_TIMEOUT_DEFAULT);
        this.writeTxnsTimeoutMs = conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_WRITE_TXNS_TIMEOUT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_WRITE_TXNS_TIMEOUT_DEFAULT);
        this.timeoutMs = (int) conf.getTimeDuration(DFSConfigKeys
                        .DFS_QJM_OPERATIONS_TIMEOUT,
                DFSConfigKeys.DFS_QJM_OPERATIONS_TIMEOUT_DEFAULT, TimeUnit
                        .MILLISECONDS);

        int connectTimeoutMs = conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_HTTP_OPEN_TIMEOUT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_HTTP_OPEN_TIMEOUT_DEFAULT);
        int readTimeoutMs = conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_HTTP_READ_TIMEOUT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_HTTP_READ_TIMEOUT_DEFAULT);
        this.connectionFactory = URLConnectionFactory
                .newDefaultURLConnectionFactory(connectTimeoutMs,       readTimeoutMs,
conf);

        // 设置发送元数据给 JournalNode 的缓存区大小 默认 512KB
        setOutputBufferCapacity(OUTPUT_BUFFER_CAPACITY_DEFAULT);
    }
```

3.3.5.1.1.1.2.1 创建 AsyncLoggerSet 并初始化 JournalNode 连接信息 IPCLoggerChannel

```
    protected List<AsyncLogger> createLoggers(
            AsyncLogger.Factory factory) throws IOException {
        // 往下追
        return createLoggers(conf, uri, nsInfo, factory, nameServiceId);
    }
```

```
static List<AsyncLogger> createLoggers(Configuration conf,
                                       URI uri,
                                       NamespaceInfo nsInfo,
                                       AsyncLogger.Factory factory,
                                       String nameServiceId)
        throws IOException {
    List<AsyncLogger> ret = Lists.newArrayList();
    // 解析 JournalNode 地址 也即解析 hj101:8485;hj102:8485;hj103:8485/mycluster
    // 1 -> hj101:8485
    // 2 -> hj102:8485
    // 3 -> hj103:8485
    List<InetSocketAddress> addrs = Util.getAddressesList(uri);
    if (addrs.size() % 2 == 0) {
        LOG.warn("Quorum journal URI '" + uri + "' has an even number " +
                "of Journal Nodes specified. This is not recommended!");
    }

    // 返回 mycluster
    String jid = parseJournalId(uri);

    for (InetSocketAddress addr : addrs) {
        ret.add(
                // 创建 IPCLoggerChannel
                factory.createLogger(conf, nsInfo, jid, nameServiceId, addr)
        );
    }
    return ret;
}
```

```
static final Factory FACTORY = new AsyncLogger.Factory() {
    @Override
    public AsyncLogger createLogger(Configuration conf,
                                    NamespaceInfo nsInfo,
                                    String journalId,
                                    String nameServiceId,
```

```
                                                    InetSocketAddress addr) {
                // 创建 IPCLoggerChannel
                return new IPCLoggerChannel(conf, nsInfo, journalId, nameServiceId, addr);
            }
        };
```

```
public IPCLoggerChannel(Configuration conf,
                                    NamespaceInfo nsInfo,
                                    String journalId,
                                    String nameServiceId,
                                    InetSocketAddress addr) {
        this.conf = conf;
        this.nsInfo = nsInfo;
        // mycluster
        this.journalId = journalId;
        // mycluster
        this.nameServiceId = nameServiceId;
        // hj101:8485
        this.addr = addr;
        // 10
        this.queueSizeLimitBytes = 1024 * 1024 * conf.getInt(
                DFSConfigKeys.DFS_QJOURNAL_QUEUE_SIZE_LIMIT_KEY,
                DFSConfigKeys.DFS_QJOURNAL_QUEUE_SIZE_LIMIT_DEFAULT);

        // 初始化两个线程池
        singleThreadExecutor = MoreExecutors.listeningDecorator(
                createSingleThreadExecutor());
        parallelExecutor = MoreExecutors.listeningDecorator(
                createParallelExecutor());

        metrics = IPCLoggerChannelMetrics.create(this);
    }
```

3.3.5.1.1.2 创建 EditLogTailer 并启动 EditLogTailerThread 线程

```
public EditLogTailer(FSNamesystem namesystem, Configuration conf) {
        // 创建 EditLogTailerThread 线程
        this.tailerThread = new EditLogTailerThread();

        this.conf = conf;
        this.namesystem = namesystem;
        // 从 FSImage 获取 FSEditLogAsync
        this.editLog = namesystem.getEditLog();

        lastLoadTimeMs = monotonicNow();
```

```
            lastRollTimeMs = monotonicNow();

            // 120s
            logRollPeriodMs = conf.getTimeDuration(
                    DFSConfigKeys.DFS_HA_LOGROLL_PERIOD_KEY,
                    DFSConfigKeys.DFS_HA_LOGROLL_PERIOD_DEFAULT,
                    TimeUnit.SECONDS,
                    TimeUnit.MILLISECONDS);
            List<RemoteNameNodeInfo> nns = Collections.emptyList();
            if (logRollPeriodMs >= 0) {
                try {
                    // 另一个 NameNode (hj102:8020) 连接信息
                    nns = RemoteNameNodeInfo.getRemoteNameNodes(conf);
                } catch (IOException e) {
                    throw new IllegalArgumentException("Remote NameNodes not correctly configured!", e);
                }

                // 解析另一个 NameNode(hj102:8020) 连接信息
                for (RemoteNameNodeInfo info : nns) {
                    // overwrite the socket address, if we need to
                    InetSocketAddress ipc = NameNode.getServiceAddress(info.getConfiguration(), true);
                    // sanity check the ipc address
                    Preconditions.checkArgument(ipc.getPort() > 0,
                            "Active NameNode must have an IPC port configured. " + "Got address '%s'", ipc);
                    info.setIpcAddress(ipc);
                }

                LOG.info("Will roll logs on active node every " +
                        (logRollPeriodMs / 1000) + " seconds.");
            } else {
                LOG.info("Not going to trigger log rolls on active node because " +
                        DFSConfigKeys.DFS_HA_LOGROLL_PERIOD_KEY + " is negative.");
            }

            sleepTimeMs = conf.getTimeDuration(
                    DFSConfigKeys.DFS_HA_TAILEDITS_PERIOD_KEY,
                    DFSConfigKeys.DFS_HA_TAILEDITS_PERIOD_DEFAULT,
                    TimeUnit.SECONDS, TimeUnit.MILLISECONDS);
            long maxSleepTimeMsTemp = conf.getTimeDuration(
                    DFSConfigKeys.DFS_HA_TAILEDITS_PERIOD_BACKOFF_MAX_KEY,
                    DFSConfigKeys.DFS_HA_TAILEDITS_PERIOD_BACKOFF_MAX_DEFAULT,
```

```
                    TimeUnit.SECONDS, TimeUnit.MILLISECONDS);
            if (maxSleepTimeMsTemp > 0 && maxSleepTimeMsTemp < sleepTimeMs) {
                LOG.warn(DFSConfigKeys.DFS_HA_TAILEDITS_PERIOD_BACKOFF_MAX_KEY
                        + " was configured to be " + maxSleepTimeMsTemp
                        + " ms, but this is less than "
                        + DFSConfigKeys.DFS_HA_TAILEDITS_PERIOD_KEY
                        + ". Disabling backoff when tailing edit logs.");
                maxSleepTimeMs = 0;
            } else {
                maxSleepTimeMs = maxSleepTimeMsTemp;
            }

            // 60s
            rollEditsTimeoutMs = conf.getTimeDuration(
                    DFSConfigKeys.DFS_HA_TAILEDITS_ROLLEDITS_TIMEOUT_KEY,
                    DFSConfigKeys.DFS_HA_TAILEDITS_ROLLEDITS_TIMEOUT_DEFAULT,
                    TimeUnit.SECONDS, TimeUnit.MILLISECONDS);

            rollEditsRpcExecutor = Executors.newSingleThreadExecutor(
                    new ThreadFactoryBuilder().setDaemon(true).build());

            // 3
            maxRetries                                                                    =
conf.getInt(DFSConfigKeys.DFS_HA_TAILEDITS_ALL_NAMESNODES_RETRY_KEY,
                    DFSConfigKeys.DFS_HA_TAILEDITS_ALL_NAMESNODES_RETRY_DEFAULT);
            if (maxRetries <= 0) {
                LOG.error("Specified a non-positive number of retries for the number of retries for
the " +
                        "namenode connection when manipulating the edit log (" +
                        DFSConfigKeys.DFS_HA_TAILEDITS_ALL_NAMESNODES_RETRY_KEY + "),
setting to default: " +

DFSConfigKeys.DFS_HA_TAILEDITS_ALL_NAMESNODES_RETRY_DEFAULT);
                maxRetries                                                                =
DFSConfigKeys.DFS_HA_TAILEDITS_ALL_NAMESNODES_RETRY_DEFAULT;
            }

            // false
            inProgressOk = conf.getBoolean(
                    DFSConfigKeys.DFS_HA_TAILEDITS_INPROGRESS_KEY,
                    DFSConfigKeys.DFS_HA_TAILEDITS_INPROGRESS_DEFAULT);

            this.maxTxnsPerLock = conf.getLong(
                    DFS_HA_TAILEDITS_MAX_TXNS_PER_LOCK_KEY,
```

```
                    DFS_HA_TAILEDITS_MAX_TXNS_PER_LOCK_DEFAULT);

            nnCount = nns.size();
            // setup the iterator to endlessly loop the nns
            this.nnLookup = Iterators.cycle(nns);

            LOG.debug("logRollPeriodMs=" + logRollPeriodMs +
                    " sleepTime=" + sleepTimeMs);
        }
```

```
/**
         * The thread which does the actual work of tailing edits journals and
         * applying the transactions to the FSNS.
         */
        private class EditLogTailerThread extends Thread {
            private volatile boolean shouldRun = true;

            private EditLogTailerThread() {
                super("Edit log tailer");
            }

            private void setShouldRun(boolean shouldRun) {
                this.shouldRun = shouldRun;
            }

            @Override
            public void run() {
                SecurityUtil.doAsLoginUserOrFatal(
                        new PrivilegedAction<Object>() {
                            @Override
                            public Object run() {
                                // 执行
                                doWork();
                                return null;
                            }
                        });
            }
```

3.3.5.1.1.3 创建 StandbyCheckpointer 并启动 CheckpointerThread 线程

```
public StandbyCheckpointer(Configuration conf, FSNamesystem ns)
                throws IOException {
        this.namesystem = ns;
        this.conf = conf;

        // 创建 CheckpointConf
        this.checkpointConf = new CheckpointConf(conf);

        // 创建 CheckpointerThread 线程
        this.thread = new CheckpointerThread();

        this.uploadThreadFactory = new ThreadFactoryBuilder().setDaemon(true)
                .setNameFormat("TransferFsImageUpload-%d").build();

        // 设置 NameNode 地址
        setNameNodeAddresses(conf);
    }
```

```
private class CheckpointerThread extends Thread {
        private volatile boolean shouldRun = true;
        private volatile long preventCheckpointsUntil = 0;

        private CheckpointerThread() {
            super("Standby State Checkpointer");
        }

        private void setShouldRun(boolean shouldRun) {
            this.shouldRun = shouldRun;
        }

        @Override
        public void run() {
            // We have to make sure we're logged in as far as JAAS
            // is concerned, in order to use kerberized SSL properly.
            SecurityUtil.doAsLoginUserOrFatal(
                    new PrivilegedAction<Object>() {
                        @Override
                        public Object run() {
                            // 执行
                            doWork();
                            return null;
                        }
                    });
```

```
        }
```

3.3.6 DataNode 往 NameNode 注册

入口类：NameNodeRpcServe.registerDatanode()

```
@Override // DatanodeProtocol
    public DatanodeRegistration registerDatanode(DatanodeRegistration nodeReg)
            throws IOException {
        checkNNStartup();
        verifySoftwareVersion(nodeReg);
        // 注册 DataNode 信息
        namesystem.registerDatanode(nodeReg);
        return nodeReg;
    }
```

```
/**
     * Register Datanode.
     * <p>
     * The purpose of registration is to identify whether the new datanode
     * serves a new data storage, and will report new data block copies,
     * which the namenode was not aware of; or the datanode is a replacement
     * node for the data storage that was previously served by a different
     * or the same (in terms of host:port) datanode.
     * The data storages are distinguished by their storageIDs. When a new
     * data storage is reported the namenode issues a new unique storageID.
     * <p>
     * Finally, the namenode returns its namespaceID as the registrationID
     * for the datanodes.
     * namespaceID is a persistent attribute of the name space.
     * The registrationID is checked every time the datanode is communicating
     * with the namenode.
     * Datanodes with inappropriate registrationID are rejected.
     * If the namenode stops, and then restarts it can restore its
     * namespaceID and will continue serving the datanodes that has previously
     * registered with the namenode without restarting the whole cluster.
     *
     * @see org.apache.hadoop.hdfs.server.datanode.DataNode
     */
    void registerDatanode(DatanodeRegistration nodeReg) throws IOException {
        writeLock();
        try {
            // BlockManager 添加 DataNode 信息
```

```
                blockManager.registerDatanode(nodeReg);
            } finally {
                writeUnlock("registerDatanode");
            }
        }
```

3.3.6.1 BlockManager 添加 DataNode 信息

```
    public void registerDatanode(DatanodeRegistration nodeReg)
            throws IOException {
        assert namesystem.hasWriteLock();
        // DatanodeManager 添加 DataNode 注册信息
        datanodeManager.registerDatanode(nodeReg);
        // 检查安全模式
        bmSafeMode.checkSafeMode();
    }
```

3.3.6.1.1 DatanodeManager 添加 DataNode 注册信息

```
/**
     * Register the given datanode with the namenode. NB: the given
     * registration is mutated and given back to the datanode.
     *
     * @param nodeReg the datanode registration
     * @throws DisallowedDatanodeException if the registration request is
     *       denied because the datanode does not match includes/excludes
     * @throws UnresolvedTopologyException if the registration request is
     *       denied because resolving datanode network location fails.
     */
    public void registerDatanode(DatanodeRegistration nodeReg)
            throws DisallowedDatanodeException, UnresolvedTopologyException {
        // DataNode Rpc Client 地址
        InetAddress dnAddress = Server.getRemoteIp();
        if (dnAddress != null) {
            // Mostly called inside an RPC, update ip and peer hostname
            String hostname = dnAddress.getHostName();
            String ip = dnAddress.getHostAddress();
            if (checkIpHostnameInRegistration && !isNameResolved(dnAddress)) {
                // Reject registration of unresolved datanode to prevent performance
                // impact of repetitive DNS lookups later.
                final String message = "hostname cannot be resolved (ip="
                        + ip + ", hostname=" + hostname + ")";
                LOG.warn("Unresolved datanode registration: " + message);
                throw new DisallowedDatanodeException(nodeReg, message);
```

```
            }
            // update node registration with the ip and hostname from rpc request
            nodeReg.setIpAddr(ip);
            nodeReg.setPeerHostName(hostname);
        }

        try {
            nodeReg.setExportedKeys(blockManager.getBlockKeys());

            // Checks if the node is not on the hosts list.    If it is not, then
            // it will be disallowed from registering.
            if (!hostConfigManager.isIncluded(nodeReg)) {
                throw new DisallowedDatanodeException(nodeReg);
            }

            // BLOCK* registerDatanode: from DatanodeRegistration(192.168.2.102:9866,
            // datanodeUuid=6099a2c7-afc2-4b65-9f0e-9f174b13050f, infoPort=9864,
            // infoSecurePort=0, ipcPort=9867, storageInfo=lv=-57;
            // cid=CID-46886fa9-7e22-40a1-b909-935ee379f640;nsid=1229485002;
            // c=1650950314011) storage 6099a2c7-afc2-4b65-9f0e-9f174b13050f
            NameNode.stateChangeLog.info("BLOCK* registerDatanode: from "
                    + nodeReg + " storage " + nodeReg.getDatanodeUuid());

            //  从注册信息中获取  DatanodeDescriptor
            // null
            DatanodeDescriptor nodeS = getDatanode(nodeReg.getDatanodeUuid());
            // null
            DatanodeDescriptor nodeN = host2DatanodeMap.getDatanodeByXferAddr(
                    nodeReg.getIpAddr(), nodeReg.getXferPort());

            if (nodeN != null && nodeN != nodeS) {
                NameNode.LOG.info("BLOCK* registerDatanode: " + nodeN);
                // nodeN previously served a different data storage,
                // which is not served by anybody anymore.
                removeDatanode(nodeN);
                // physically remove node from datanodeMap
                wipeDatanode(nodeN);
                nodeN = null;
            }

            if (nodeS != null) {
                if (nodeN == nodeS) {
                    // The same datanode has been just restarted to serve the same data
                    // storage. We do not need to remove old data blocks, the delta will
```

```
                        // be calculated on the next block report from the datanode
                        if (NameNode.stateChangeLog.isDebugEnabled()) {
                            NameNode.stateChangeLog.debug("BLOCK* registerDatanode: "
                                    + "node restarted.");
                        }
                    } else {
                        // nodeS is found
             /* The registering datanode is a replacement node for the existing
                data storage, which from now on will be served by a new node.
                If this message repeats, both nodes might have same storageID
                by (insanely rare) random chance. User needs to restart one of the
                nodes with its data cleared (or user can just remove the StorageID
                value in "VERSION" file under the data directory of the datanode,
                but this is might not work if VERSION file format has changed
            */
                        NameNode.stateChangeLog.info("BLOCK* registerDatanode: " + nodeS
                                + " is replaced by " + nodeReg + " with the same storageID "
                                + nodeReg.getDatanodeUuid());
                    }

                    boolean success = false;
                    try {
                        // update cluster map
                        getNetworkTopology().remove(nodeS);
                        if (shouldCountVersion(nodeS)) {
                            decrementVersionCount(nodeS.getSoftwareVersion());
                        }
                        nodeS.updateRegInfo(nodeReg);

                        nodeS.setSoftwareVersion(nodeReg.getSoftwareVersion());
                        nodeS.setDisallowed(false); // Node is in the include list

                        // resolve network location
                        if (this.rejectUnresolvedTopologyDN) {
                            nodeS.setNetworkLocation(resolveNetworkLocation(nodeS));

nodeS.setDependentHostNames(getNetworkDependencies(nodeS));
                        } else {
                            nodeS.setNetworkLocation(

resolveNetworkLocationWithFallBackToDefaultLocation(nodeS));
                            nodeS.setDependentHostNames(
                                    getNetworkDependenciesWithDefault(nodeS));
                        }
```

```
                        getNetworkTopology().add(nodeS);
                        resolveUpgradeDomain(nodeS);

                        // also treat the registration message as a heartbeat
                        heartbeatManager.register(nodeS);
                        incrementVersionCount(nodeS.getSoftwareVersion());
                        startAdminOperationIfNecessary(nodeS);
                        success = true;
                    } finally {
                        if (!success) {
                            removeDatanode(nodeS);
                            wipeDatanode(nodeS);
                            countSoftwareVersions();
                        }
                    }
                    return;
                }

                // 创建 DatanodeDescriptor
                DatanodeDescriptor nodeDescr
                        = new DatanodeDescriptor(nodeReg, NetworkTopology.DEFAULT_RACK);
                boolean success = false;
                try {
                    // resolve network location
                    if (this.rejectUnresolvedTopologyDN) {
                        nodeDescr.setNetworkLocation(resolveNetworkLocation(nodeDescr));

nodeDescr.setDependentHostNames(getNetworkDependencies(nodeDescr));
                    } else {
                        nodeDescr.setNetworkLocation(

resolveNetworkLocationWithFallBackToDefaultLocation(nodeDescr));
                        nodeDescr.setDependentHostNames(
                                getNetworkDependenciesWithDefault(nodeDescr));
                    }
                    nodeDescr.setSoftwareVersion(nodeReg.getSoftwareVersion());
                    resolveUpgradeDomain(nodeDescr);

                    // register new datanode
                    // 注册 DataNode
                    addDatanode(nodeDescr);
                    // 往 BlockReportLeaseManager 注册 DataNode
                    blockManager.getBlockReportLeaseManager()
                            .register(nodeDescr);
```

```
                // also treat the registration message as a heartbeat
                // no need to update its timestamp
                // because its is done when the descriptor is created
                //  往心跳管理者添加  DataNode
                heartbeatManager.addDatanode(nodeDescr);
                heartbeatManager.updateDnStat(nodeDescr);

                incrementVersionCount(nodeReg.getSoftwareVersion());
                startAdminOperationIfNecessary(nodeDescr);
                success = true;
            } finally {
                if (!success) {
                    removeDatanode(nodeDescr);
                    wipeDatanode(nodeDescr);
                    countSoftwareVersions();
                }
            }
        } catch (InvalidTopologyException e) {
            // If the network location is invalid, clear the cached mappings
            // so that we have a chance to re-add this DataNode with the
            // correct network location later.
            List<String> invalidNodeNames = new ArrayList<>(3);
            // clear cache for nodes in IP or Hostname
            invalidNodeNames.add(nodeReg.getIpAddr());
            invalidNodeNames.add(nodeReg.getHostName());
            invalidNodeNames.add(nodeReg.getPeerHostName());
            dnsToSwitchMapping.reloadCachedMappings(invalidNodeNames);
            throw e;
        }
    }
```

3.3.6.1.1.1 往 BlockReportLeaseManager 注册 DataNode

```
    public synchronized void register(DatanodeDescriptor dn) {
        //  注册  DataNode
        registerNode(dn);
    }
```

```
    public synchronized void register(DatanodeDescriptor dn) {
        //  注册  DataNode
        registerNode(dn);
    }
```

```
private synchronized NodeData registerNode(DatanodeDescriptor dn) {
        if (nodes.containsKey(dn.getDatanodeUuid())) {
            LOG.info("Can't register DN {} because it is already registered.",
                    dn.getDatanodeUuid());
            return null;
        }
        NodeData node = new NodeData(dn.getDatanodeUuid());
        // 添加 NodeData
        deferredHead.addToBeginning(node);
        nodes.put(dn.getDatanodeUuid(), node);
        //2023-03-25 11:25:39,249 INFO
org.apache.hadoop.hdfs.server.blockmanagement.BlockReportLeaseManager:
        // Registered DN 01263ed0-7bb5-4dec-845b-97a8d2d5d052 (192.168.2.101:9866)
        LOG.info("Registered DN {} ({}).", dn.getDatanodeUuid(), dn.getXferAddr());
        return node;
    }
```

3.3.6.1.2 往心跳管理者 HeartbeatManager 添加 DataNode

```
    synchronized void addDatanode(final DatanodeDescriptor d) {
        // update in-service node count
        datanodes.add(d);
        d.setAlive(true);
    }
```

3.3.7 DataNode 向 NameNode 发送心跳

入口类：NameNodeRpcServer.sendHeartbeat()

```
@Override // DatanodeProtocol
    public HeartbeatResponse sendHeartbeat(DatanodeRegistration nodeReg,
                                           StorageReport[] report, long
dnCacheCapacity, long dnCacheUsed,
                                           int xmitsInProgress, int xceiverCount,
                                           int failedVolumes, VolumeFailureSummary
volumeFailureSummary,
                                           boolean requestFullBlockReportLease,
                                           @Nonnull SlowPeerReports slowPeers,
                                           @Nonnull SlowDiskReports slowDisks)
throws IOException {
        checkNNStartup();
        verifyRequest(nodeReg);
        // FSNamesystem 处理 DataNode 发送过来的心跳
```

```
            return namesystem.handleHeartbeat(nodeReg, report,
                    dnCacheCapacity, dnCacheUsed, xceiverCount, xmitsInProgress,
                    failedVolumes, volumeFailureSummary, requestFullBlockReportLease,
                    slowPeers, slowDisks);
        }
```

```
/**
         * The given node has reported in.  This method should:
         * 1) Record the heartbeat, so the datanode isn't timed out
         * 2) Adjust usage stats for future block allocation
         * <p>
         * If a substantial amount of time passed since the last datanode
         * heartbeat then request an immediate block report.
         *
         * @return an array of datanode commands
         * @throws IOException
         */
        HeartbeatResponse handleHeartbeat(DatanodeRegistration nodeReg,
                                          StorageReport[] reports, long cacheCapacity, long
cacheUsed,
                                          int xceiverCount, int xmitsInProgress, int
failedVolumes,
                                          VolumeFailureSummary volumeFailureSummary,
                                          boolean requestFullBlockReportLease,
                                          @Nonnull SlowPeerReports slowPeers,
                                          @Nonnull SlowDiskReports slowDisks) throws
IOException {
            readLock();
            try {
                //get datanode commands
                final int maxTransfer = blockManager.getMaxReplicationStreams()
                        - xmitsInProgress;

                // BlockManager 的 DatanodeManager 处理 DataNode 发送过来的心跳并返
回处理结果
                DatanodeCommand[] cmds = blockManager.getDatanodeManager()
                        .handleHeartbeat(
                        nodeReg, reports, getBlockPoolId(), cacheCapacity, cacheUsed,
                        xceiverCount, maxTransfer, failedVolumes, volumeFailureSummary,
                        slowPeers, slowDisks);
                long blockReportLeaseId = 0;

                if (requestFullBlockReportLease) {
                    blockReportLeaseId = blockManager.requestBlockReportLeaseId(nodeReg);
```

```
                }
                //create ha status
                final NNHAStatusHeartbeat haState = new NNHAStatusHeartbeat(
                        haContext.getState().getServiceState(),
                        getFSImage().getCorrectLastAppliedOrWrittenTxId());

                return new HeartbeatResponse(cmds, haState, rollingUpgradeInfo,
                        blockReportLeaseId);
            } finally {
                readUnlock("handleHeartbeat");
            }
        }
```

3.3.7.1 BlockManager 的 DatanodeManager 处理 DataNode 发送过来的心跳并返回处理结果 (DataNode 上报 Block 命令)

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

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

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

            // If we are in safemode, do not send back any recovery / replication
            // requests. Don't even drain the existing queue of work.
            if (namesystem.isInSafeMode()) {
                return new DatanodeCommand[0];
            }

            // block recovery command
            final BlockRecoveryCommand brCommand = getBlockRecoveryCommand(blockPoolId,
                    nodeinfo);
            if (brCommand != null) {
                return new DatanodeCommand[]{brCommand};
            }

            final List<DatanodeCommand> cmds = new ArrayList<>();
            // Allocate _approximately_ maxTransfers pending tasks to DataNode.
            // NN chooses pending tasks based on the ratio between the lengths of
            // replication and erasure-coded block queues.
            int totalReplicateBlocks = nodeinfo.getNumberOfReplicateBlocks();
            int totalECBlocks = nodeinfo.getNumberOfBlocksToBeErasureCoded();
            int totalBlocks = totalReplicateBlocks + totalECBlocks;
            if (totalBlocks > 0) {
                int numReplicationTasks = (int) Math.ceil(
                        (double) (totalReplicateBlocks * maxTransfers) / totalBlocks);
                int numECTasks = (int) Math.ceil(
                        (double) (totalECBlocks * maxTransfers) / totalBlocks);

                if (LOG.isDebugEnabled()) {
                    LOG.debug("Pending replication tasks: " + numReplicationTasks
                            + " erasure-coded tasks: " + numECTasks);
                }
                // check pending replication tasks
                List<BlockTargetPair> pendingList = nodeinfo.getReplicationCommand(
                        numReplicationTasks);
                if (pendingList != null && !pendingList.isEmpty()) {
                    cmds.add(new                BlockCommand(DatanodeProtocol.DNA_TRANSFER,
blockPoolId,
                            pendingList));
```

```
        }
        // check pending erasure coding tasks
        List<BlockECReconstructionInfo> pendingECList = nodeinfo
                .getErasureCodeCommand(numECTasks);
        if (pendingECList != null && !pendingECList.isEmpty()) {
            cmds.add(new BlockECReconstructionCommand(
                    DNA_ERASURE_CODING_RECONSTRUCTION, pendingECList));
        }
    }

    // check block invalidation
    Block[] blks = nodeinfo.getInvalidateBlocks(blockInvalidateLimit);
    if (blks != null) {
        cmds.add(new BlockCommand(DatanodeProtocol.DNA_INVALIDATE, blockPoolId,
                blks));
    }
    // cache commands
    addCacheCommands(blockPoolId, nodeinfo, cmds);
    // key update command
    blockManager.addKeyUpdateCommand(cmds, nodeinfo);

    // check for balancer bandwidth update
    if (nodeinfo.getBalancerBandwidth() > 0) {
        cmds.add(new BalancerBandwidthCommand(nodeinfo.getBalancerBandwidth()));
        // set back to 0 to indicate that datanode has been sent the new value
        nodeinfo.setBalancerBandwidth(0);
    }

    if (slowPeerTracker != null) {
        final Map<String, Double> slowPeersMap = slowPeers.getSlowPeers();
        if (!slowPeersMap.isEmpty()) {
            if (LOG.isDebugEnabled()) {
                LOG.debug("DataNode " + nodeReg + " reported slow peers: " +
                        slowPeersMap);
            }
            for (String slowNodeId : slowPeersMap.keySet()) {
                slowPeerTracker.addReport(slowNodeId, nodeReg.getIpcAddr(false));
            }
        }
    }

    if (slowDiskTracker != null) {
        if (!slowDisks.getSlowDisks().isEmpty()) {
            if (LOG.isDebugEnabled()) {
```

```
                        LOG.debug("DataNode " + nodeReg + " reported slow disks: " +
                                slowDisks.getSlowDisks());
                    }
                    slowDiskTracker.addSlowDiskReport(nodeReg.getIpcAddr(false), slowDisks);
                }
            }

            if (!cmds.isEmpty()) {
                return cmds.toArray(new DatanodeCommand[cmds.size()]);
            }

            return new DatanodeCommand[0];
        }
```

3.3.7.1.1 HeartbeatManager 更新心跳

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

3.3.7.1.1.1 BlockManager 更新心跳

```
void updateHeartbeat(DatanodeDescriptor node, StorageReport[] reports,
                     long cacheCapacity, long cacheUsed, int xceiverCount, int failedVolumes,
                     VolumeFailureSummary volumeFailureSummary) {

        for (StorageReport report : reports) {
            providedStorageMap.updateStorage(node, report.getStorage());
        }
        // 更新心跳
        node.updateHeartbeat(reports, cacheCapacity, cacheUsed, xceiverCount,
                failedVolumes, volumeFailureSummary);
    }
```

3.3.7.1.1.1.1 DatanodeDescriptor 更新心跳

```
/**
   * Updates stats from datanode heartbeat.
   */
  void updateHeartbeat(StorageReport[] reports, long cacheCapacity,
        long cacheUsed, int xceiverCount, int volFailures,
        VolumeFailureSummary volumeFailureSummary) {
    // 更新心跳
    updateHeartbeatState(reports, cacheCapacity, cacheUsed, xceiverCount,
          volFailures, volumeFailureSummary);
    heartbeatedSinceRegistration = true;
  }
```

```
  /**
   * process datanode heartbeat or stats initialization.
   */
  void updateHeartbeatState(StorageReport[] reports, long cacheCapacity,
        long cacheUsed, int xceiverCount, int volFailures,
        VolumeFailureSummary volumeFailureSummary) {
    updateStorageStats(reports, cacheCapacity, cacheUsed, xceiverCount,
          volFailures, volumeFailureSummary);
    setLastUpdate(Time.now());
    setLastUpdateMonotonic(Time.monotonicNow());
    rollBlocksScheduled(getLastUpdateMonotonic());
  }
```

```
private void updateStorageStats(StorageReport[] reports, long cacheCapacity,
        long cacheUsed, int xceiverCount, int volFailures,
        VolumeFailureSummary volumeFailureSummary) {
    long totalCapacity = 0;
    long totalRemaining = 0;
    long totalBlockPoolUsed = 0;
    long totalDfsUsed = 0;
    long totalNonDfsUsed = 0;
    Set<DatanodeStorageInfo> failedStorageInfos = null;

    // Decide if we should check for any missing StorageReport and mark it as
    // failed. There are different scenarios.
    // 1. When DN is running, a storage failed. Given the current DN
    //      implementation doesn't add recovered storage back to its storage list
    //      until DN restart, we can assume volFailures won't decrease
    //      during the current DN registration session.
    //      When volumeFailures == this.volumeFailures, it implies there is no
    //      state change. No need to check for failed storage. This is an
```

```
//       optimization.    Recent versions of the DataNode report a
//       VolumeFailureSummary containing the date/time of the last volume
//       failure.    If that's available, then we check that instead for greater
//       accuracy.
// 2. After DN restarts, volFailures might not increase and it is possible
//       we still have new failed storage. For example, admins reduce
//       available storages in configuration. Another corner case
//       is the failed volumes might change after restart; a) there
//       is one good storage A, one restored good storage B, so there is
//       one element in storageReports and that is A. b) A failed. c) Before
//       DN sends HB to NN to indicate A has failed, DN restarts. d) After DN
//       restarts, storageReports has one element which is B.
final boolean checkFailedStorages;
if (volumeFailureSummary != null && this.volumeFailureSummary != null) {
   checkFailedStorages = volumeFailureSummary.getLastVolumeFailureDate() >
         this.volumeFailureSummary.getLastVolumeFailureDate();
} else {
   checkFailedStorages = (volFailures > this.volumeFailures) ||
         !heartbeatedSinceRegistration;
}

if (checkFailedStorages) {
   if (this.volumeFailures != volFailures) {
      LOG.info("Number of failed storages changes from {} to {}",
            this.volumeFailures, volFailures);
   }
   synchronized (storageMap) {
      failedStorageInfos =
            new HashSet<>(storageMap.values());
   }
}

setCacheCapacity(cacheCapacity);
setCacheUsed(cacheUsed);
setXceiverCount(xceiverCount);
this.volumeFailures = volFailures;
this.volumeFailureSummary = volumeFailureSummary;
for (StorageReport report : reports) {

   DatanodeStorageInfo storage =
         storageMap.get(report.getStorage().getStorageID());
   if (checkFailedStorages) {
      failedStorageInfos.remove(storage);
   }
```

```
        // DatanodeStorageInfo 接收心跳
        storage.receivedHeartbeat(report);
        // skip accounting for capacity of PROVIDED storages!
        if (StorageType.PROVIDED.equals(storage.getStorageType())) {
          continue;
        }

        totalCapacity += report.getCapacity();
        totalRemaining += report.getRemaining();
        totalBlockPoolUsed += report.getBlockPoolUsed();
        totalDfsUsed += report.getDfsUsed();
        totalNonDfsUsed += report.getNonDfsUsed();
      }

      // Update total metrics for the node.
      setCapacity(totalCapacity);
      setRemaining(totalRemaining);
      setBlockPoolUsed(totalBlockPoolUsed);
      setDfsUsed(totalDfsUsed);
      setNonDfsUsed(totalNonDfsUsed);
      if (checkFailedStorages) {
        updateFailedStorage(failedStorageInfos);
      }
      long storageMapSize;
      synchronized (storageMap) {
        storageMapSize = storageMap.size();
      }
      if (storageMapSize != reports.length) {
        pruneStorageMap(reports);
      }
    }
```

```
  /** Sets time when this information was accurate. */
  public void setLastUpdate(long lastUpdate) {
    this.lastUpdate = lastUpdate;
  }
```

3.3.7.1.2 心跳结果返回 DataNode 上报 Block 命令

```
// block recovery command
          // 心跳返回 DataNode 需要上报 Block 命令
          final BlockRecoveryCommand brCommand = getBlockRecoveryCommand(blockPoolId,
                  nodeinfo);
          if (brCommand != null) {
```

```
            return new DatanodeCommand[]{brCommand};
        }
```

3.3.8 处理 DataNode 上报 Block 请求

入口类：NameNodeRpcServer.blockReport()

```
@Override // DatanodeProtocol
    public DatanodeCommand blockReport(final DatanodeRegistration nodeReg,
                                       String poolId, final StorageBlockReport[] reports,
                                       final BlockReportContext context) throws
IOException {
        checkNNStartup();
        verifyRequest(nodeReg);
        if (blockStateChangeLog.isDebugEnabled()) {
            blockStateChangeLog.debug("*BLOCK* NameNode.blockReport: "
                    + "from " + nodeReg + ", reports.length=" + reports.length);
        }
        final BlockManager bm = namesystem.getBlockManager();
        boolean noStaleStorages = false;
        try {
            if (bm.checkBlockReportLease(context, nodeReg)) {
                for (int r = 0; r < reports.length; r++) {
                    final BlockListAsLongs blocks = reports[r].getBlocks();
                    //
                    // BlockManager.processReport accumulates information of prior calls
                    // for the same node and storage, so the value returned by the last
                    // call of this loop is the final updated value for noStaleStorage.
                    //
                    final int index = r;
                    noStaleStorages = bm.runBlockOp(new Callable<Boolean>() {
                        @Override
                        public Boolean call() throws IOException {
                            // 处理上报 Block
                            return bm.processReport(nodeReg,
reports[index].getStorage(),
                                    blocks, context);
                        }
                    });
                }
            }
        } catch (UnregisteredNodeException une) {
            LOG.debug("Datanode {} is attempting to report but not register yet.",
                    nodeReg);
```

```
            return RegisterCommand.REGISTER;
        }
        bm.removeBRLeaseIfNeeded(nodeReg, context);

        BlockManagerFaultInjector.getInstance().
                incomingBlockReportRpc(nodeReg, context);

        if (nn.getFSImage().isUpgradeFinalized() &&
                !namesystem.isRollingUpgrade() &&
                nn.isActiveState() &&
                noStaleStorages) {
            return new FinalizeCommand(poolId);
        }

        return null;
    }
```

3.3.9 NameNode 成为 Active

入口类：NameNodeRpcServer.transitionToActive()

```
@Override // HAServiceProtocol
    public synchronized void transitionToActive(StateChangeRequestInfo req)
            throws ServiceFailedException, AccessControlException, IOException {
        checkNNStartup();
        nn.checkHaStateChange(req);
        // NameNode 成为 Active
        nn.transitionToActive();
    }
```

```
synchronized void transitionToActive()
            throws ServiceFailedException, AccessControlException {
        namesystem.checkSuperuserPrivilege();
        if (!haEnabled) {
            throw new ServiceFailedException("HA for namenode is not enabled");
        }
        if (state == OBSERVER_STATE) {
            throw new ServiceFailedException(
                    "Cannot transition from '" + OBSERVER_STATE + "' to '" +
                            ACTIVE_STATE + "'");
        }
        // 设置 ACTIVE_STATE
        state.setState(haContext, ACTIVE_STATE);
```

```
    }
```

```
ACTIVE_STATE = new ActiveState();
```

```
@Override
    public void setState(HAContext context, HAState s) throws ServiceFailedException {
        if (s == NameNode.ACTIVE_STATE ||
                (!isObserver && s == NameNode.OBSERVER_STATE) ||
                (isObserver && s == NameNode.STANDBY_STATE)) {
            // 设置 ACTIVE_STATE
            setStateInternal(context, s);
            return;
        }
        super.setState(context, s);
    }
```

```
 /**
   * Internal method to move from the existing state to a new state.
   * @param context HA context
   * @param s new state
   * @throws ServiceFailedException on failure to transition to new state.
   */
  protected final void setStateInternal(final HAContext context, final HAState s)
      throws ServiceFailedException {
    // 取消 StandbyCheckpointer
    prepareToExitState(context);

    s.prepareToEnterState(context);
    context.writeLock();
    try {
      // 停止 Standby 服务
      exitState(context);
      // 设置状态 ActiveState
      context.setState(s);
      // 进入 ActiveState
      s.enterState(context);
      // 更新 HA Active 时间
      s.updateLastHATransitionTime();
    } finally {
      context.writeUnlock();
    }
  }
```

3.3.9.1 取消 StandbyCheckpointer

```
@Override
    public void prepareToExitState(HAContext context) throws ServiceFailedException {
        // 停止 StandbyCheckpointer
        context.prepareToStopStandbyServices();
    }
```

```
        @Override
        public void prepareToStopStandbyServices() throws ServiceFailedException {
            try {
                // 停止 StandbyCheckpointer
                namesystem.prepareToStopStandbyServices();
            } catch (Throwable t) {
                doImmediateShutdown(t);
            }
        }
```

```
/**
     * Called while the NN is in Standby state, but just about to be
     * asked to enter Active state. This cancels any checkpoints
     * currently being taken.
     */
    void prepareToStopStandbyServices() throws ServiceFailedException {
        if (standbyCheckpointer != null) {
            //
            standbyCheckpointer.cancelAndPreventCheckpoints(
                    "About to leave standby state");
        }
    }
```

```
**
     * Cancel any checkpoint that's currently being made,
     * and prevent any new checkpoints from starting for the next
     * minute or so.
     */
    public void cancelAndPreventCheckpoints(String msg) throws ServiceFailedException {
        synchronized (cancelLock) {
            // The checkpointer thread takes this lock and checks if checkpointing is
            // postponed.
            thread.preventCheckpointsFor(PREVENT_AFTER_CANCEL_MS);
```

```
                // Before beginning a checkpoint, the checkpointer thread
                // takes this lock, and creates a canceler object.
                // If the canceler is non-null, then a checkpoint is in
                // progress and we need to cancel it. If it's null, then
                // the operation has not started, meaning that the above
                // time-based prevention will take effect.
                if (canceler != null) {
                    canceler.cancel(msg);
                }
            }
        }
```

3.3.9.2 退出 Standby 相关服务

```
@Override
    public void exitState(HAContext context) throws ServiceFailedException {
        try {
            //
            context.stopStandbyServices();
        } catch (IOException e) {
            throw new ServiceFailedException("Failed to stop standby services", e);
        }
    }
```

```
        @Override
        public void stopStandbyServices() throws IOException {
            try {
                if (namesystem != null) {
                    namesystem.stopStandbyServices();
                }
            } catch (Throwable t) {
                doImmediateShutdown(t);
            }
        }
```

```
 /**
     * Stop services required in standby state
     */
    void stopStandbyServices() throws IOException {
        LOG.info("Stopping services started for standby state");
        if (standbyCheckpointer != null) {
            // 停止 CheckpointerThread 线程
```

```
                standbyCheckpointer.stop();
            }
            if (editLogTailer != null) {
                // 停止 EditLogTailerThread 线程
                editLogTailer.stop();
            }
            if (dir != null && getFSImage() != null && getFSImage().editLog != null) {
                //
                getFSImage().editLog.close();
            }
        }
```

3.3.9.2.1 停止 CheckpointerThread 线程

```
public void stop() throws IOException {
        cancelAndPreventCheckpoints("Stopping checkpointer");
        thread.setShouldRun(false);
        thread.interrupt();
        try {
            thread.join();
        } catch (InterruptedException e) {
            LOG.warn("Edit log tailer thread exited with an exception");
            throw new IOException(e);
        }
    }
```

3.3.9.2.2 停止 EditLogTailerThread 线程

```
public void stop() throws IOException {
        tailerThread.setShouldRun(false);
        tailerThread.interrupt();
        try {
            tailerThread.join();
        } catch (InterruptedException e) {
            LOG.warn("Edit log tailer thread exited with an exception");
            throw new IOException(e);
        } finally {
            rollEditsRpcExecutor.shutdown();
        }
    }
```

3.3.9.3 设置状态 ActiveState

```
    protected class NameNodeHAContext implements HAContext {
```

```
        @Override
        public void setState(HAState s) {
            // ActiveState
            state = s;
        }
```

3.3.9.4 进入 ActiveState 并启动相关服务

```
@Override
  public void enterState(HAContext context) throws ServiceFailedException {
    try {
      // 启动 ActiveState 服务
      context.startActiveServices();
    } catch (IOException e) {
      throw new ServiceFailedException("Failed to start active services", e);
    }
  }
```

```
@Override
        public void startActiveServices() throws IOException {
            try {
                // 启动 ActiveState 服务
                namesystem.startActiveServices();
                startTrashEmptier(getConf());
            } catch (Throwable t) {
                doImmediateShutdown(t);
            }
        }
```

3.3.9.4.1 启动 ActiveState 服务

```
/**
     * Start services required in active state
     *
     * @throws IOException
     */
    void startActiveServices() throws IOException {
        startingActiveService = true;
        LOG.info("Starting services required for active state");
        writeLock();
        try {
            // 从 FSImage 获取 FSEditLogAsync
            FSEditLog editLog = getFSImage().getEditLog();
```

```
            if (!editLog.isOpenForWrite()) {
                // During startup, we're already open for write during initialization.
                // 初始化 JournalNode 连接 (以便后续发送 RPC Request 给 JournalNode)
                editLog.initJournalsForWrite();
                // May need to recover
                editLog.recoverUnclosedStreams();

                LOG.info("Catching up to latest edits from old active before " +
                        "taking over writer role in edits logs");
                editLogTailer.catchupDuringFailover();

                blockManager.setPostponeBlocksFromFuture(false);
                blockManager.getDatanodeManager().markAllDatanodesStale();
                blockManager.clearQueues();
                blockManager.processAllPendingDNMessages();

                // Only need to re-process the queue, If not in SafeMode.
                if (!isInSafeMode()) {
                    LOG.info("Reprocessing replication and invalidation queues");
                    blockManager.initializeReplQueues();
                }

                if (LOG.isDebugEnabled()) {
                    LOG.debug("NameNode metadata after re-processing " +
                            "replication and invalidation queues during failover:\n" +
                            metaSaveAsString());
                }

                long nextTxId = getFSImage().getLastAppliedTxId() + 1;
                LOG.info("Will take over writing edit logs at txnid " +
                        nextTxId);
                // 设置下一次 TXID
                editLog.setNextTxId(nextTxId);

                // open write
                getFSImage().editLog.openForWrite(getEffectiveLayoutVersion());
            }

            // Initialize the quota.
            dir.updateCountForQuota();
            // Enable quota checks.
            dir.enableQuotaChecks();
            dir.ezManager.startReencryptThreads();
```

```
            if (haEnabled) {
                // Renew all of the leases before becoming active.
                // This is because, while we were in standby mode,
                // the leases weren't getting renewed on this NN.
                // Give them all a fresh start here.
                leaseManager.renewAllLeases();
            }
            leaseManager.startMonitor();
            startSecretManagerIfNecessary();

            //ResourceMonitor required only at ActiveNN. See HDFS-2914
            this.nnrmthread = new Daemon(new NameNodeResourceMonitor());
            nnrmthread.start();

            nnEditLogRoller = new Daemon(new NameNodeEditLogRoller(
                    editLogRollerThreshold, editLogRollerInterval));
            nnEditLogRoller.start();

            if (lazyPersistFileScrubIntervalSec > 0) {
                lazyPersistFileScrubber = new Daemon(new LazyPersistFileScrubber(
                        lazyPersistFileScrubIntervalSec));
                lazyPersistFileScrubber.start();
            } else {
                LOG.warn("Lazy persist file scrubber is disabled,"
                        + " configured scrub interval is zero.");
            }

            cacheManager.startMonitorThread();
            blockManager.getDatanodeManager().setShouldSendCachingCommands(true);
            if (provider != null) {
                edekCacheLoader = Executors.newSingleThreadExecutor(
                        new ThreadFactoryBuilder().setDaemon(true)
                                .setNameFormat("Warm Up EDEK Cache Thread #%d")
                                .build());
                FSDirEncryptionZoneOp.warmUpEdekCache(edekCacheLoader, dir,
                        edekCacheLoaderDelay, edekCacheLoaderInterval);
            }
        } finally {
            startingActiveService = false;
            blockManager.checkSafeMode();
            writeUnlock("startActiveServices");
        }
    }
```

3.3.9.4.1.1 初始化 JournalNode 连接 (以便后续发送 RPC Request 给 JournalNode)

```
public synchronized void initJournalsForWrite() {
    Preconditions.checkState(state == State.UNINITIALIZED ||
            state == State.CLOSED, "Unexpected state: %s", state);

    // 初始化 JournalNode 连接
    initJournals(this.editsDirs);
    state = State.BETWEEN_LOG_SEGMENTS;
}
```

```
private synchronized void initJournals(List<URI> dirs) {
    // 1
    int minimumRedundantJournals = conf.getInt(
            DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_KEY,
            DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_DEFAULT);

    synchronized (journalSetLock) {
        // 创建 JournalSet
        journalSet = new JournalSet(minimumRedundantJournals);

        for (URI u : dirs) {
            // true
            boolean required = FSNamesystem.getRequiredNamespaceEditsDirs(conf)
                    .contains(u);
            // false
            if (u.getScheme().equals(NNStorage.LOCAL_URI_SCHEME)) {
                StorageDirectory sd = storage.getStorageDirectory(u);
                if (sd != null) {
                    // 创建 FileJournalManager
                    journalSet.add(new FileJournalManager(conf, sd, storage),
                            required, sharedEditsDirs.contains(u));
                }
            } else {
                //
                journalSet.add(
                        // 创建 QuorumJournalManager
                        createJournal(u),
                        // true
                        required,
                        // true
                        sharedEditsDirs.contains(u));
            }
```

```
                }
            }

            if (journalSet.isEmpty()) {
                LOG.error("No edits directories configured!");
            }
        }
```

3.3.9.4.1.2 LeaseManager 开始监控 EditLog

```
    void startMonitor() {
        Preconditions.checkState(lmthread == null,
                "Lease Monitor already running");
        shouldRunMonitor = true;
        // 创建 Monitor 线程并启动
        lmthread = new Daemon(new Monitor());
        lmthread.start();
    }
```

3.3.9.4.1.2.1 创建 Monitor 线程并启动

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

        /** Check leases periodically. */
        @Override
        public void run() {
            for(; shouldRunMonitor && fsnamesystem.isRunning(); ) {
                boolean needSync = false;
                try {
                    fsnamesystem.writeLockInterruptibly();
                    try {
                        if (!fsnamesystem.isInSafeMode()) {
                            needSync = checkLeases();
                        }
                    } finally {
                        fsnamesystem.writeUnlock("leaseManager");
                        // lease reassignments should to be sync'ed.
                        if (needSync) {
```

```
                    fsnamesystem.getEditLog().logSync();
                }
            }

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

3.3.9.4.1.3 创建 NameNodeResourceMonitor 线程并启动

```
/**
     * Periodically calls hasAvailableResources of NameNodeResourceChecker, and if
     * there are found to be insufficient resources available, causes the NN to
     * enter safe mode. If resources are later found to have returned to
     * acceptable levels, this daemon will cause the NN to exit safe mode.
     */
    class NameNodeResourceMonitor implements Runnable {
        boolean shouldNNRmRun = true;

        @Override
        public void run() {
            try {
                while (fsRunning && shouldNNRmRun) {
                    checkAvailableResources();
                    if (!nameNodeHasResourcesAvailable()) {
                        String lowResourcesMsg = "NameNode low on available disk space.
";
                        if (!isInSafeMode()) {
                            LOG.warn(lowResourcesMsg + "Entering safe mode.");
                        } else {
                            LOG.warn(lowResourcesMsg + "Already in safe mode.");
                        }
                        enterSafeMode(true);
                    }
                    try {
                        Thread.sleep(resourceRecheckInterval);
                    } catch (InterruptedException ie) {
                        // Deliberately ignore
                    }
```

```
                    }
                } catch (Exception e) {
                    FSNamesystem.LOG.error("Exception in NameNodeResourceMonitor: ", e);
                }
            }
```

3.3.9.4.1.4 创建 NameNodeEditLogRoller 线程并启动

```
class NameNodeEditLogRoller implements Runnable {

        private boolean shouldRun = true;
        private final long rollThreshold;
        private final long sleepIntervalMs;

        public NameNodeEditLogRoller(long rollThreshold, int sleepIntervalMs) {
            // edit 默认滚动阈值  0.5 * 100 0000
            this.rollThreshold = rollThreshold;
            // edit 默认滚动间隔   5 * 60 * 1000
            this.sleepIntervalMs = sleepIntervalMs;
        }

        @Override
        public void run() {
            while (fsRunning && shouldRun) {
                try {
                    long numEdits = getCorrectTransactionsSinceLastLogRoll();
                    if (numEdits > rollThreshold) {
                        FSNamesystem.LOG.info("NameNode rolling its own edit log
because"
                                + " number of edits in open segment exceeds threshold of
"
                                + rollThreshold);
                        rollEditLog();
                    }
                } catch (Exception e) {
                    FSNamesystem.LOG.error("Swallowing exception in "
                            + NameNodeEditLogRoller.class.getSimpleName() + ":", e);
                }
                try {
                    Thread.sleep(sleepIntervalMs);
                } catch (InterruptedException e) {
                    FSNamesystem.LOG.info(NameNodeEditLogRoller.class.getSimpleName()
                            + " was interrupted, exiting");
```

```
                    break;
                }
            }
        }
```

3.3.9.4.1.5 更新 HA Active 时间

```
private void updateLastHATransitionTime() {
    lastHATransitionTime = Time.now();
}
```

3.3.10 NameNode 成为 Standby

入口类：NameNodeRpcServer.transitionToStandby

```
@Override // HAServiceProtocol
    public synchronized void transitionToStandby(StateChangeRequestInfo req)
            throws ServiceFailedException, AccessControlException, IOException {
        checkNNStartup();
        nn.checkHaStateChange(req);
        // NameNode 进入 Standby 状态
        nn.transitionToStandby();
    }
```

```
synchronized void transitionToStandby()
            throws ServiceFailedException, AccessControlException {
        namesystem.checkSuperuserPrivilege();
        if (!haEnabled) {
            throw new ServiceFailedException("HA for namenode is not enabled");
        }
        // 进入 StandbyState 状态
        state.setState(haContext, STANDBY_STATE);
    }
```

```
@Override
    public void setState(HAContext context, HAState s) throws ServiceFailedException {
        if (s == NameNode.ACTIVE_STATE ||
                (!isObserver && s == NameNode.OBSERVER_STATE) ||
                (isObserver && s == NameNode.STANDBY_STATE)) {
            // 设置 ACTIVE_STATE
            setStateInternal(context, s);
            return;
        }
        // nothing to do
```

```
            super.setState(context, s);
        }
```

3.4 DataNode 启动源码分析

入口类：org.apache.hadoop.hdfs.server.datanode.DataNode

```
    public static void main(String args[]) {
        if (DFSUtil.parseHelpArgument(args, DataNode.USAGE, System.out, true)) {
            System.exit(0);
        }

        // 启动 DataNode
        secureMain(args, null);
    }
```

```
public static void secureMain(String args[], SecureResources resources) {
        int errorCode = 0;
        try {
            StringUtils.startupShutdownMessage(DataNode.class, args, LOG);
            // 创建 DataNode 并启动
            DataNode datanode = createDataNode(args, null, resources);
            if (datanode != null) {
                // 阻塞等待
                datanode.join();
            } else {
                errorCode = 1;
            }
        } catch (Throwable e) {
            LOG.error("Exception in secureMain", e);
            terminate(1, e);
        } finally {
            // We need to terminate the process here because either shutdown was called
            // or some disk related conditions like volumes tolerated or volumes required
            // condition was not met. Also, In secure mode, control will go to Jsvc
            // and Datanode process hangs if it does not exit.
            LOG.warn("Exiting Datanode");
            terminate(errorCode);
        }
    }
```

```
 /**
     * Instantiate & Start a single datanode daemon and wait for it to finish.
```

```
     * If this thread is specifically interrupted, it will stop waiting.
     */
    @VisibleForTesting
    @InterfaceAudience.Private
    public static DataNode createDataNode(String args[], Configuration conf,
                                           SecureResources resources) throws
IOException {
        // 初始化 DataNode
        DataNode dn = instantiateDataNode(args, conf, resources);
        if (dn != null) {
            // 启动 DataNode
            dn.runDatanodeDaemon();
        }
        return dn;
    }
```

3.4.1 初始化 DataNode

```
/**
     * Instantiate a single datanode object, along with its secure resources.
     * This must be run by invoking{@link DataNode#runDatanodeDaemon()}
     * subsequently.
     */
    public static DataNode instantiateDataNode(String args[], Configuration conf,
                                               SecureResources resources) throws
IOException {
        if (conf == null)
            // 创建 HdfsConfiguration
            conf = new HdfsConfiguration();

        if (args != null) {
            // parse generic hadoop options
            // 解析入参
            GenericOptionsParser hParser = new GenericOptionsParser(conf, args);
            args = hParser.getRemainingArgs();
        }

        if (!parseArguments(args, conf)) {
            printUsage(System.err);
            return null;
        }
```

```
        // 将 DataNode 存储数据的目录封装成 StorageLocation
        Collection<StorageLocation> dataLocations = getStorageLocations(conf);

        UserGroupInformation.setConfiguration(conf);
        SecurityUtil.login(conf, DFS_DATANODE_KEYTAB_FILE_KEY,
                DFS_DATANODE_KERBEROS_PRINCIPAL_KEY, getHostName(conf));

        // 实例化 DataNode
        return makeInstance(dataLocations, conf, resources);
    }
```

3.4.1.1 DataNode 存储数据的目录封装成 StorageLocation

```
public static List<StorageLocation> getStorageLocations(Configuration conf) {
        // 从配置中获取 key = dfs.datanode.data.dir 对应的 value
        // 一般情况下在 hdfs-site.xml 配置对应的值 比如 /opt/app/hadoop-3.1.3/data/data
        Collection<String> rawLocations =
                conf.getTrimmedStringCollection(DFS_DATANODE_DATA_DIR_KEY);
        List<StorageLocation> locations =
                new ArrayList<StorageLocation>(rawLocations.size());

        for (String locationString : rawLocations) {
            final StorageLocation location;
            try {
                // 将 DataNode 存储数据的目录封装成 StorageLocation
                location = StorageLocation.parse(locationString);
            } catch (IOException | SecurityException ioe) {
                LOG.error("Failed to initialize storage directory {}." +
                        "Exception details: {}", locationString, ioe.toString());
                // Ignore the exception.
                continue;
            }

            // 添加
            locations.add(location);
        }

        return locations;
    }
```

3.4.1.2 实例化 DataNode (构造函数)

```
/**
     * Make an instance of DataNode after ensuring that at least one of the
     * given data directories (and their parent directories, if necessary)
     * can be created.
     *
     * @param dataDirs    List of directories, where the new DataNode instance should
     *                    keep its files.
     * @param conf        Configuration instance to use.
     * @param resources Secure resources needed to run under Kerberos
     * @return DataNode instance for given list of data dirs and conf, or null if
     * no directory from this directory list can be created.
     * @throws IOException
     */
    static DataNode makeInstance(Collection<StorageLocation> dataDirs,
                                 Configuration conf, SecureResources resources) throws
IOException {
        List<StorageLocation> locations;

        // 创建 StorageLocationChecker 检查 DataNode 数据存储目录
        StorageLocationChecker storageLocationChecker =
                new StorageLocationChecker(conf, new Timer());
        try {
            // 检查 DataNode 数据存储目录
            locations = storageLocationChecker.check(conf, dataDirs);
        } catch (InterruptedException ie) {
            throw new IOException("Failed to instantiate DataNode", ie);
        }
        DefaultMetricsSystem.initialize("DataNode");

        assert locations.size() > 0 : "number of data directories should be > 0";

        // 创建 DataNode 对象
        return new DataNode(conf, locations, storageLocationChecker, resources);
    }
```

```
/**
     * Create the DataNode given a configuration, an array of dataDirs,
     * and a namenode proxy.
     */
    DataNode(final Configuration conf,
```

```
                final List<StorageLocation> dataDirs,
                final StorageLocationChecker storageLocationChecker,
                final SecureResources resources) throws IOException {
        // 往下追
        super(conf);

        this.tracer = createTracer(conf);
        this.tracerConfigurationManager =
                new TracerConfigurationManager(DATANODE_HTRACE_PREFIX, conf);

        // 创建 FileIoProvider
        this.fileIoProvider = new FileIoProvider(conf, this);
        // 创建 BlockScanner
        this.blockScanner = new BlockScanner(this);

        this.lastDiskErrorCheck = 0;
        this.maxNumberOfBlocksToLog = conf.getLong(DFS_MAX_NUM_BLOCKS_TO_LOG_KEY,
                DFS_MAX_NUM_BLOCKS_TO_LOG_DEFAULT);

        this.usersWithLocalPathAccess = Arrays.asList(

conf.getTrimmedStrings(DFSConfigKeys.DFS_BLOCK_LOCAL_PATH_ACCESS_USER_KEY));
        this.connectToDnViaHostname = conf.getBoolean(
                DFSConfigKeys.DFS_DATANODE_USE_DN_HOSTNAME,
                DFSConfigKeys.DFS_DATANODE_USE_DN_HOSTNAME_DEFAULT);
        this.supergroup = conf.get(DFSConfigKeys.DFS_PERMISSIONS_SUPERUSERGROUP_KEY,
                DFSConfigKeys.DFS_PERMISSIONS_SUPERUSERGROUP_DEFAULT);
        this.isPermissionEnabled = conf.getBoolean(
                DFSConfigKeys.DFS_PERMISSIONS_ENABLED_KEY,
                DFSConfigKeys.DFS_PERMISSIONS_ENABLED_DEFAULT);
        this.pipelineSupportECN = conf.getBoolean(
                DFSConfigKeys.DFS_PIPELINE_ECN_ENABLED,
                DFSConfigKeys.DFS_PIPELINE_ECN_ENABLED_DEFAULT);

        confVersion = "core-" +
                conf.get("hadoop.common.configuration.version", "UNSPECIFIED") +
                ",hdfs-" +
                conf.get("hadoop.hdfs.configuration.version", "UNSPECIFIED");

        // 创建 DatasetVolumeChecker
        this.volumeChecker = new DatasetVolumeChecker(conf, new Timer());

        // Determine whether we should try to pass file descriptors to clients.
        if (conf.getBoolean(HdfsClientConfigKeys.Read.ShortCircuit.KEY,
```

```
                    HdfsClientConfigKeys.Read.ShortCircuit.DEFAULT)) {
                String reason = DomainSocket.getLoadingFailureReason();
                if (reason != null) {
                    LOG.warn("File descriptor passing is disabled because {}", reason);
                    this.fileDescriptorPassingDisabledReason = reason;
                } else {
                    LOG.info("File descriptor passing is enabled.");
                    this.fileDescriptorPassingDisabledReason = null;
                }
            } else {
                this.fileDescriptorPassingDisabledReason =
                        "File descriptor passing was not configured.";
                LOG.debug(this.fileDescriptorPassingDisabledReason);
            }

            // 通过反射的方式创建 StandardSocketFactory 对象
            this.socketFactory = NetUtils.getDefaultSocketFactory(conf);

            try {
                // 返回 DataNode 本地 IP 地址 比如 192.168.2.103 -> hj103
                hostName = getHostName(conf);

                // Configured hostname is hj101
                LOG.info("Configured hostname is {}", hostName);
                // 启动 DataNode
                startDataNode(dataDirs, resources);
            } catch (IOException ie) {
                shutdown();
                throw ie;
            }
            final int dncCacheMaxSize =
                    conf.getInt(DFS_DATANODE_NETWORK_COUNTS_CACHE_MAX_SIZE_KEY,

DFS_DATANODE_NETWORK_COUNTS_CACHE_MAX_SIZE_DEFAULT);
            datanodeNetworkCounts =
                    CacheBuilder.newBuilder()
                            .maximumSize(dncCacheMaxSize)
                            .build(new CacheLoader<String, Map<String, Long>>() {
                                @Override
                                public Map<String, Long> load(String key) throws Exception {
                                    final Map<String, Long> ret = new HashMap<String,
Long>();

                                    ret.put("networkErrors", 0L);
                                    return ret;
```

```
                                    }
                                });

        initOOBTimeout();
        this.storageLocationChecker = storageLocationChecker;
    }
```

3.4.1.2.1 创建 DatasetVolumeChecker

```
 /**
     * @param conf    Configuration object.
     * @param timer {@link Timer} object used for throttling checks.
     */
    public DatasetVolumeChecker(Configuration conf, Timer timer)
            throws DiskErrorException {
        // 10m = 10 * 60 * 1000
        maxAllowedTimeForCheckMs = conf.getTimeDuration(
                DFS_DATANODE_DISK_CHECK_TIMEOUT_KEY,
                DFS_DATANODE_DISK_CHECK_TIMEOUT_DEFAULT,
                TimeUnit.MILLISECONDS);

        if (maxAllowedTimeForCheckMs <= 0) {
            throw new DiskErrorException("Invalid value configured for "
                    + DFS_DATANODE_DISK_CHECK_TIMEOUT_KEY + " - "
                    + maxAllowedTimeForCheckMs + " (should be > 0)");
        }

        this.timer = timer;

        // 0
        maxVolumeFailuresTolerated = conf.getInt(
                DFS_DATANODE_FAILED_VOLUMES_TOLERATED_KEY,
                DFS_DATANODE_FAILED_VOLUMES_TOLERATED_DEFAULT);

        minDiskCheckGapMs = conf.getTimeDuration(
                DFSConfigKeys.DFS_DATANODE_DISK_CHECK_MIN_GAP_KEY,
                DFSConfigKeys.DFS_DATANODE_DISK_CHECK_MIN_GAP_DEFAULT,
                TimeUnit.MILLISECONDS);

        if (minDiskCheckGapMs < 0) {
            throw new DiskErrorException("Invalid value configured for "
                    + DFS_DATANODE_DISK_CHECK_MIN_GAP_KEY + " - "
                    + minDiskCheckGapMs + " (should be >= 0)");
        }
```

```
            // 10m = 10 * 60 * 1000
            diskCheckTimeout = conf.getTimeDuration(
                        DFSConfigKeys.DFS_DATANODE_DISK_CHECK_TIMEOUT_KEY,
                        DFSConfigKeys.DFS_DATANODE_DISK_CHECK_TIMEOUT_DEFAULT,
                        TimeUnit.MILLISECONDS);

            if (diskCheckTimeout < 0) {
                  throw new DiskErrorException("Invalid value configured for "
                              + DFS_DATANODE_DISK_CHECK_TIMEOUT_KEY + " - "
                              + diskCheckTimeout + " (should be >= 0)");
            }

            lastAllVolumesCheck = timer.monotonicNow() - minDiskCheckGapMs;

            if                              (maxVolumeFailuresTolerated                              <
DataNode.MAX_VOLUME_FAILURE_TOLERATED_LIMIT) {
                  throw new DiskErrorException("Invalid value configured for "
                              + DFS_DATANODE_FAILED_VOLUMES_TOLERATED_KEY + " - "
                              + maxVolumeFailuresTolerated + " "
                              + DataNode.MAX_VOLUME_FAILURES_TOLERATED_MSG);
            }

            // 创建 ThrottledAsyncChecker
            delegateChecker = new ThrottledAsyncChecker<>(
                        timer, minDiskCheckGapMs, diskCheckTimeout,
                        Executors.newCachedThreadPool(
                                    new ThreadFactoryBuilder()
                                                .setNameFormat("DataNode DiskChecker thread %d")
                                                .setDaemon(true)
                                                .build()));

            checkVolumeResultHandlerExecutorService = Executors.newCachedThreadPool(
                        new ThreadFactoryBuilder()
                                    .setNameFormat("VolumeCheck ResultHandler thread %d")
                                    .setDaemon(true)
                                    .build());
      }
```

3.4.1.2.2 启动 DataNode

```
/**
       * This method starts the data node with the specified conf.
       * <p>
       * If conf's CONFIG_PROPERTY_SIMULATED property is set
```

```
     * then a simulated storage based data node is created.
     *
     * @param dataDirectories - only for a non-simulated storage data node
     * @throws IOException
     */
    void startDataNode(List<StorageLocation> dataDirectories,
                       SecureResources resources
    ) throws IOException {

        // settings global for all BPs in the Data Node
        // null
        this.secureResources = resources;
        synchronized (this) {
            // DataNode  数据存储目录
            this.dataDirs = dataDirectories;
        }
        this.dnConf = new DNConf(this);
        checkSecureConfig(dnConf, getConf(), resources);

        if (dnConf.maxLockedMemory > 0) {
            if (!NativeIO.POSIX.getCacheManipulator().verifyCanMlock()) {
                throw new RuntimeException(String.format(
                        "Cannot start datanode because the configured max locked
memory" +
                                " size (%s) is greater than zero and native code is not
available.",
                        DFS_DATANODE_MAX_LOCKED_MEMORY_KEY));
            }
            if (Path.WINDOWS) {
                NativeIO.Windows.extendWorkingSetSize(dnConf.maxLockedMemory);
            } else {
                long ulimit = NativeIO.POSIX.getCacheManipulator().getMemlockLimit();
                if (dnConf.maxLockedMemory > ulimit) {
                    throw new RuntimeException(String.format(
                            "Cannot start datanode because the configured max locked
memory" +
                                    " size (%s) of %d bytes is more than the datanode's
available" +
                                    " RLIMIT_MEMLOCK ulimit of %d bytes.",
                            DFS_DATANODE_MAX_LOCKED_MEMORY_KEY,
                            dnConf.maxLockedMemory,
                            ulimit));
                }
            }
```

```
        }
        LOG.info("Starting DataNode with maxLockedMemory = {}",
                dnConf.maxLockedMemory);

        int volFailuresTolerated = dnConf.getVolFailuresTolerated();
        int volsConfigured = dnConf.getVolsConfigured();

        if (volFailuresTolerated < MAX_VOLUME_FAILURE_TOLERATED_LIMIT
                || volFailuresTolerated >= volsConfigured) {
            throw new DiskErrorException("Invalid value configured for "
                    + "dfs.datanode.failed.volumes.tolerated - " + volFailuresTolerated
                    + ". Value configured is either less than -1 or >= "
                    + "to the number of configured volumes (" + volsConfigured + ").");
        }

        // 创建 DataStorage
        storage = new DataStorage();

        // global DN settings
        registerMXBean();

        // 初始化 DataXceiverServer
        initDataXceiver();

        // 创建 DatanodeHttpServer 并启动
        startInfoServer();

        pauseMonitor = new JvmPauseMonitor();
        pauseMonitor.init(getConf());
        pauseMonitor.start();

        // BlockPoolTokenSecretManager is required to create ipc server.
        // 创建 BlockPoolTokenSecretManager
        this.blockPoolTokenSecretManager = new BlockPoolTokenSecretManager();

        // Login is done by now. Set the DN user name.
        dnUserName = UserGroupInformation.getCurrentUser().getUserName();
        LOG.info("dnUserName = {}", dnUserName);
        LOG.info("supergroup = {}", supergroup);

        // 初始化 RpcServer
        initIpcServer();

        metrics = DataNodeMetrics.create(getConf(), getDisplayName());
```

```
            peerMetrics = dnConf.peerStatsEnabled ?
                    DataNodePeerMetrics.create(getDisplayName(), getConf()) : null;
            metrics.getJvmMetrics().setPauseMonitor(pauseMonitor);

            // 创建 ErasureCodingWorker
            ecWorker = new ErasureCodingWorker(getConf(), this);
            // 创建 BlockRecoveryWorker
            blockRecoveryWorker = new BlockRecoveryWorker(this);

            // 创建 BlockPoolManager
            blockPoolManager = new BlockPoolManager(this);
            // 刷新 NameNode 地址
            blockPoolManager.refreshNamenodes(getConf());

            // Create the ReadaheadPool from the DataNode context so we can
            // exit without having to explicitly shutdown its thread pool.
            readaheadPool = ReadaheadPool.getInstance();
            saslClient = new SaslDataTransferClient(dnConf.getConf(),
                    dnConf.saslPropsResolver, dnConf.trustedChannelResolver);
            saslServer = new SaslDataTransferServer(dnConf, blockPoolTokenSecretManager);
            startMetricsLogger();

            if (dnConf.diskStatsEnabled) {
                diskMetrics = new DataNodeDiskMetrics(this,
                        dnConf.outliersReportIntervalMs);
            }
        }
```

3.4.1.2.2.1 创建 DataStorage

```
    DataStorage() {
        // 往下追
        super(NodeType.DATA_NODE);
        trashEnabledBpids = Collections.newSetFromMap(
            new ConcurrentHashMap<String, Boolean>());
    }
```

```
    /**
     * Create empty storage info of the specified type
     */
    protected Storage(NodeType type) {
        // 往下追
        super(type);
    }
```

```
    public StorageInfo(NodeType type) {
        // 往下追
        this(0, 0, "", 0L, type);
    }
```

```
    public StorageInfo(int layoutV, int nsID, String cid, long cT, NodeType type) {
        // keep on
        layoutVersion = layoutV;
        clusterID = cid;
        namespaceID = nsID;
        cTime = cT;
        storageType = type;
    }
```

3.4.1.2.2.2 初始化 DataXceiverServer

```
private void initDataXceiver() throws IOException {
        // find free port or use privileged port provided
        TcpPeerServer tcpPeerServer;
        if (secureResources != null) {
            tcpPeerServer = new TcpPeerServer(secureResources);
        } else {
            // 128
            int backlogLength = getConf().getInt(

CommonConfigurationKeysPublic.IPC_SERVER_LISTEN_QUEUE_SIZE_KEY,

CommonConfigurationKeysPublic.IPC_SERVER_LISTEN_QUEUE_SIZE_DEFAULT);

            // 创建 TcpPeerServer (TCP-Server)
            tcpPeerServer = new TcpPeerServer(
                    dnConf.socketWriteTimeout,
                    // 默认 0.0.0.0:9866
                    DataNode.getStreamingAddr(getConf()),
                    backlogLength);
        }
        if (dnConf.getTransferSocketRecvBufferSize() > 0) {
            tcpPeerServer.setReceiveBufferSize(
                    dnConf.getTransferSocketRecvBufferSize());
        }
        // Opened streaming server at /0.0.0.0:9866
        streamingAddr = tcpPeerServer.getStreamingAddr();
        LOG.info("Opened streaming server at {}", streamingAddr);
```

```
            // 创建 ThreadGroup
            this.threadGroup = new ThreadGroup("dataXceiverServer");
            // 创建 DataXceiverServer (实现 Runnable 接口)
            xserver = new DataXceiverServer(tcpPeerServer, getConf(), this);
            // 封装 DataXceiverServer 为 Daemon 对象
            this.dataXceiverServer = new Daemon(threadGroup, xserver);
            this.threadGroup.setDaemon(true); // auto destroy when empty

            if (getConf().getBoolean(
                        HdfsClientConfigKeys.Read.ShortCircuit.KEY,
                        HdfsClientConfigKeys.Read.ShortCircuit.DEFAULT) ||
                        getConf().getBoolean(

HdfsClientConfigKeys.DFS_CLIENT_DOMAIN_SOCKET_DATA_TRAFFIC,
                                    HdfsClientConfigKeys
                                            .DFS_CLIENT_DOMAIN_SOCKET_DATA_TRAFFIC_DEFAULT
)) {
                    DomainPeerServer domainPeerServer =
                            getDomainPeerServer(getConf(), streamingAddr.getPort());
                    if (domainPeerServer != null) {
                        this.localDataXceiverServer = new Daemon(threadGroup,
                                new DataXceiverServer(domainPeerServer, getConf(), this));
                        LOG.info("Listening on UNIX domain socket: {}",
                                domainPeerServer.getBindPath());
                    }
                }
            }
            // 创建 ShortCircuitRegistry
            this.shortCircuitRegistry = new ShortCircuitRegistry(getConf());
        }
```

3.4.1.2.2.2.1 创建 TcpPeerServer (TCP-Server)

```
    /**
     * Create a non-secure TcpPeerServer.
     *
     * @param socketWriteTimeout    The Socket write timeout in ms.
     * @param bindAddr                 The address to bind to.
     * @param backlogLength            The length of the tcp accept backlog
     * @throws IOException
     */
    public TcpPeerServer(int socketWriteTimeout,
                         InetSocketAddress bindAddr,
                         int backlogLength) throws IOException {
        this.serverSocket = (socketWriteTimeout > 0) ?
                // 创建 ServerSocketChannel.open().socket()
```

```
                    ServerSocketChannel.open().socket() : new ServerSocket();
            // ServerSocketChannel 绑定地址 (默认 0.0.0.0:9866)
            Server.bind(serverSocket, bindAddr, backlogLength);
        }
```

3.4.1.2.2.2 创建 DataXceiverServer (实现 Runnable 接口)

```
DataXceiverServer(PeerServer peerServer, Configuration conf,
                                DataNode datanode) {
            // TcpPeerServer (0.0.0.0:9866)
            this.peerServer = peerServer;
            this.datanode = datanode;

            // 4096
            this.maxXceiverCount =
                    conf.getInt(DFSConfigKeys.DFS_DATANODE_MAX_RECEIVER_THREADS_KEY,

DFSConfigKeys.DFS_DATANODE_MAX_RECEIVER_THREADS_DEFAULT);

            // 128MB
            this.estimateBlockSize = conf.getLongBytes(DFSConfigKeys.DFS_BLOCK_SIZE_KEY,
                    DFSConfigKeys.DFS_BLOCK_SIZE_DEFAULT);

            //set up parameter for cluster balancing
            // 创建 BlockBalanceThrottler
            this.balanceThrottler = new BlockBalanceThrottler(
                    // 10MB

conf.getLongBytes(DFSConfigKeys.DFS_DATANODE_BALANCE_BANDWIDTHPERSEC_KEY,

DFSConfigKeys.DFS_DATANODE_BALANCE_BANDWIDTHPERSEC_DEFAULT),
                    // 50

conf.getInt(DFSConfigKeys.DFS_DATANODE_BALANCE_MAX_NUM_CONCURRENT_MOVES_KEY,

DFSConfigKeys.DFS_DATANODE_BALANCE_MAX_NUM_CONCURRENT_MOVES_DEFAULT));
        }
```

3.4.1.2.2.3 创建 DatanodeHttpServer 并启动

```
/**
         * @see DFSUtil#getHttpPolicy(org.apache.hadoop.conf.Configuration)
         * for information related to the different configuration options and
         * Http Policy is decided.
         */
```

```
    private void startInfoServer()
            throws IOException {
        // SecureDataNodeStarter will bind the privileged port to the channel if
        // the DN is started by JSVC, pass it along.
        // null
        ServerSocketChannel httpServerChannel = secureResources != null ?
                secureResources.getHttpServerChannel() : null;

        // 创建 DatanodeHttpServer
        httpServer = new DatanodeHttpServer(getConf(), this, httpServerChannel);
        // 启动 DatanodeHttpServer
        httpServer.start();

        if (httpServer.getHttpAddress() != null) {
            infoPort = httpServer.getHttpAddress().getPort();
        }
        if (httpServer.getHttpsAddress() != null) {
            infoSecurePort = httpServer.getHttpsAddress().getPort();
        }
    }
```

3.4.1.2.2.3.1 创建 DatanodeHttpServer

```
public DatanodeHttpServer(final Configuration conf,
                          final DataNode datanode,
                          final ServerSocketChannel externalHttpChannel)
            throws IOException {
        // 创建 RestCsrfPreventionFilter
        this.restCsrfPreventionFilter = createRestCsrfPreventionFilter(conf);
        this.conf = conf;

        Configuration confForInfoServer = new Configuration(conf);
        // 4
        confForInfoServer.setInt(HttpServer2.HTTP_MAX_THREADS_KEY,
                HTTP_MAX_THREADS);
        // 1
        confForInfoServer.setInt(HttpServer2.HTTP_SELECTOR_COUNT_KEY,
                HTTP_SELECTOR_THREADS);
        // 1
        confForInfoServer.setInt(HttpServer2.HTTP_ACCEPTOR_COUNT_KEY,
                HTTP_ACCEPTOR_THREADS);
        int proxyPort =
                confForInfoServer.getInt(DFS_DATANODE_HTTP_INTERNAL_PROXY_PORT, 0);
```

```
// 创建 HttpServer2
HttpServer2.Builder builder = new HttpServer2.Builder()
        .setName("datanode")
        .setConf(confForInfoServer)
        .setACL(new AccessControlList(conf.get(DFS_ADMIN, " ")))
        .hostName(getHostnameForSpnegoPrincipal(confForInfoServer))
        .addEndpoint(URI.create("http://localhost:" + proxyPort))
        .setFindPort(true);

// true
final boolean xFrameEnabled = conf.getBoolean(
        DFSConfigKeys.DFS_XFRAME_OPTION_ENABLED,
        DFSConfigKeys.DFS_XFRAME_OPTION_ENABLED_DEFAULT);

final String xFrameOptionValue = conf.getTrimmed(
        DFSConfigKeys.DFS_XFRAME_OPTION_VALUE,
        DFSConfigKeys.DFS_XFRAME_OPTION_VALUE_DEFAULT);

builder.configureXFrame(xFrameEnabled).setXFrameOption(xFrameOptionValue);

// 构建 HttpServer2
this.infoServer = builder.build();

this.infoServer.setAttribute(HttpServer2.CONF_CONTEXT_ATTRIBUTE, conf);
this.infoServer.setAttribute("datanode", datanode);
this.infoServer.setAttribute(JspHelper.CURRENT_CONF, conf);

// HttpServer2 添加 Servlet
this.infoServer.addServlet(null, "/blockScannerReport",
        BlockScanner.Servlet.class);

DataNodeUGIProvider.init(conf);

// 启动 HttpServer2
this.infoServer.start();

final InetSocketAddress jettyAddr = infoServer.getConnectorAddress(0);

this.confForCreate = new Configuration(conf);
confForCreate.set(FsPermission.UMASK_LABEL, "000");

// Netty NIO 相关组件
this.bossGroup = new NioEventLoopGroup();
this.workerGroup = new NioEventLoopGroup();
```

```
        this.externalHttpChannel = externalHttpChannel;
        HttpConfig.Policy policy = DFSUtil.getHttpPolicy(conf);

        if (policy.isHttpEnabled()) {
            // 基于 Netty 创建 Http Server 服务
            this.httpServer = new ServerBootstrap().group(bossGroup, workerGroup)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline p = ch.pipeline();
                            p.addLast(new HttpRequestDecoder(),
                                    new HttpResponseEncoder());
                            if (restCsrfPreventionFilter != null) {
                                p.addLast(new RestCsrfPreventionFilterHandler(
                                        restCsrfPreventionFilter));
                            }
                            p.addLast(
                                    new ChunkedWriteHandler(),
                                    new URLDispatcher(jettyAddr, conf, confForCreate));
                        }
                    });

            // 设置参数
            this.httpServer.childOption(
                    ChannelOption.WRITE_BUFFER_HIGH_WATER_MARK,
                    conf.getInt(
                            DFSConfigKeys.DFS_WEBHDFS_NETTY_HIGH_WATERMARK,

DFSConfigKeys.DFS_WEBHDFS_NETTY_HIGH_WATERMARK_DEFAULT));
            this.httpServer.childOption(
                    ChannelOption.WRITE_BUFFER_LOW_WATER_MARK,
                    conf.getInt(
                            DFSConfigKeys.DFS_WEBHDFS_NETTY_LOW_WATERMARK,

DFSConfigKeys.DFS_WEBHDFS_NETTY_LOW_WATERMARK_DEFAULT));

            if (externalHttpChannel == null) {
                // 绑定 NioServerSocketChannel
                httpServer.channel(NioServerSocketChannel.class);
            } else {
                httpServer.channelFactory(new ChannelFactory<NioServerSocketChannel>() {
                    @Override
                    public NioServerSocketChannel newChannel() {
```

```
                                return new NioServerSocketChannel(externalHttpChannel) {
                                    // The channel has been bounded externally via JSVC,
                                    // thus bind() becomes a no-op.
                                    @Override
                                    protected void doBind(SocketAddress localAddress) throws
Exception {
                                    }
                                };
                            }
                        });
                }
            } else {
                this.httpServer = null;
            }

            if (policy.isHttpsEnabled()) {
                this.sslFactory = new SSLFactory(SSLFactory.Mode.SERVER, conf);
                try {
                    sslFactory.init();
                } catch (GeneralSecurityException e) {
                    throw new IOException(e);
                }
                this.httpsServer = new ServerBootstrap().group(bossGroup, workerGroup)
                        .channel(NioServerSocketChannel.class)
                        .childHandler(new ChannelInitializer<SocketChannel>() {
                            @Override
                            protected void initChannel(SocketChannel ch) throws Exception {
                                ChannelPipeline p = ch.pipeline();
                                p.addLast(
                                        new SslHandler(sslFactory.createSSLEngine()),
                                        new HttpRequestDecoder(),
                                        new HttpResponseEncoder());
                                if (restCsrfPreventionFilter != null) {
                                    p.addLast(new RestCsrfPreventionFilterHandler(
                                            restCsrfPreventionFilter));
                                }
                                p.addLast(
                                        new ChunkedWriteHandler(),
                                        new URLDispatcher(jettyAddr, conf, confForCreate));
                            }
                        });
            } else {
                this.httpsServer = null;
                this.sslFactory = null;
```

```
        }
    }
```

3.4.1.2.2.3.1.1 创建 HttpServer2 并启动

```
```

3.4.1.2.2.3.1.2 基于 Netty 创建 Http Server 服务

```
```

3.4.1.2.2.3.2 启动 DatanodeHttpServer (基于 Netty 创建的 Http-Server)

```
public void start() throws IOException {
        if (httpServer != null) {
            InetSocketAddress infoAddr = DataNode.getInfoAddr(conf);
            // 绑定 0.0.0.0:9864
            ChannelFuture f = httpServer.bind(infoAddr);
            try {
                f.syncUninterruptibly();
            } catch (Throwable e) {
                if (e instanceof BindException) {
                    throw NetUtils.wrapException(null, 0, infoAddr.getHostName(),
                            infoAddr.getPort(), (SocketException) e);
                } else {
                    throw e;
                }
            }
            httpAddress = (InetSocketAddress) f.channel().localAddress();
            // Listening HTTP traffic on /0.0.0.0:9864
            LOG.info("Listening HTTP traffic on " + httpAddress);
        }

        if (httpsServer != null) {
            InetSocketAddress secInfoSocAddr =
                    NetUtils.createSocketAddr(conf.getTrimmed(
                            DFS_DATANODE_HTTPS_ADDRESS_KEY,
                            DFS_DATANODE_HTTPS_ADDRESS_DEFAULT));
            ChannelFuture f = httpsServer.bind(secInfoSocAddr);

            try {
                f.syncUninterruptibly();
            } catch (Throwable e) {
                if (e instanceof BindException) {
                    throw NetUtils.wrapException(null, 0, secInfoSocAddr.getHostName(),
                            secInfoSocAddr.getPort(), (SocketException) e);
```

```
                    } else {
                        throw e;
                    }
                }
                httpsAddress = (InetSocketAddress) f.channel().localAddress();
                LOG.info("Listening HTTPS traffic on " + httpsAddress);
            }
        }
```

3.4.1.2.2.4 创建 BlockPoolTokenSecretManager

```

```

3.4.1.2.2.5 始化 RpcServer 并绑定通讯协议

```
private void initIpcServer() throws IOException {
        // 默认 0.0.0.0：9867
        InetSocketAddress ipcAddr = NetUtils.createSocketAddr(
                getConf().getTrimmed(DFS_DATANODE_IPC_ADDRESS_KEY));

        // Add all the RPC protocols that the Datanode implements
        // 设置 ClientDatanodeProtocol 通讯协议
        RPC.setProtocolEngine(getConf(), ClientDatanodeProtocolPB.class,
                ProtobufRpcEngine.class);
        ClientDatanodeProtocolServerSideTranslatorPB clientDatanodeProtocolXlator =
                new ClientDatanodeProtocolServerSideTranslatorPB(this);
        BlockingService service = ClientDatanodeProtocolService
                .newReflectiveBlockingService(clientDatanodeProtocolXlator);

        // 构建 RPC
        ipcServer = new RPC.Builder(getConf())
                .setProtocol(ClientDatanodeProtocolPB.class)
                .setInstance(service)
                .setBindAddress(ipcAddr.getHostName())
                .setPort(ipcAddr.getPort())
                .setNumHandlers(
                        // 默认 Handler 个数 10
                        getConf().getInt(DFS_DATANODE_HANDLER_COUNT_KEY,

DFS_DATANODE_HANDLER_COUNT_DEFAULT)).setVerbose(false)
                .setSecretManager(blockPoolTokenSecretManager).build();

        // 绑定通讯协议
        ReconfigurationProtocolServerSideTranslatorPB reconfigurationProtocolXlator
                = new ReconfigurationProtocolServerSideTranslatorPB(this);
```

```
        service = ReconfigurationProtocolService
                .newReflectiveBlockingService(reconfigurationProtocolXlator);
        DFSUtil.addPBProtocol(getConf(), ReconfigurationProtocolPB.class, service,
                ipcServer);

        InterDatanodeProtocolServerSideTranslatorPB interDatanodeProtocolXlator =
                new InterDatanodeProtocolServerSideTranslatorPB(this);
        service = InterDatanodeProtocolService
                .newReflectiveBlockingService(interDatanodeProtocolXlator);
        DFSUtil.addPBProtocol(getConf(), InterDatanodeProtocolPB.class, service,
                ipcServer);

        TraceAdminProtocolServerSideTranslatorPB traceAdminXlator =
                new TraceAdminProtocolServerSideTranslatorPB(this);
        BlockingService traceAdminService = TraceAdminService
                .newReflectiveBlockingService(traceAdminXlator);
        DFSUtil.addPBProtocol(
                getConf(),
                TraceAdminProtocolPB.class,
                traceAdminService,
                ipcServer);

        // Opened IPC server at /0.0.0.0:9867
        LOG.info("Opened IPC server at {}", ipcServer.getListenerAddress());

        // set service-level authorization security policy
        if (getConf().getBoolean(
                CommonConfigurationKeys.HADOOP_SECURITY_AUTHORIZATION, false)) {
            ipcServer.refreshServiceAcl(getConf(), new HDFSPolicyProvider());
        }
    }
```

3.4.1.2.2.6 创建 ErasureCodingWorker

```
public ErasureCodingWorker(Configuration conf, DataNode datanode) {
    this.datanode = datanode;
    this.conf = conf;
    this.xmitWeight = conf.getFloat(
        DFSConfigKeys.DFS_DN_EC_RECONSTRUCTION_XMITS_WEIGHT_KEY,
        DFSConfigKeys.DFS_DN_EC_RECONSTRUCTION_XMITS_WEIGHT_DEFAULT
    );
    Preconditions.checkArgument(this.xmitWeight >= 0,
        "Invalid value configured for " +
            DFSConfigKeys.DFS_DN_EC_RECONSTRUCTION_XMITS_WEIGHT_KEY +
            ", it can not be negative value (" + this.xmitWeight + ").");
```

```
    initializeStripedReadThreadPool();
    initializeStripedBlkReconstructionThreadPool(conf.getInt(
        DFSConfigKeys.DFS_DN_EC_RECONSTRUCTION_THREADS_KEY,
        DFSConfigKeys.DFS_DN_EC_RECONSTRUCTION_THREADS_DEFAULT));
  }
```

3.4.1.2.2.7 创建 BlockRecoveryWorker

```
  BlockRecoveryWorker(DataNode datanode) {
    this.datanode = datanode;
    conf = datanode.getConf();
    dnConf = datanode.getDnConf();
  }
```

3.4.1.2.2.8 创建 BlockPoolManager

```
   BlockPoolManager(DataNode dn) {
        this.dn = dn;
    }
```

3.4.1.2.2.9 刷新 NameNode 地址

```
void refreshNamenodes(Configuration conf)
            throws IOException {
        LOG.info("Refresh request received for nameservices: " +
                conf.get(DFSConfigKeys.DFS_NAMESERVICES));

        Map<String, Map<String, InetSocketAddress>> newAddressMap = null;
        Map<String, Map<String, InetSocketAddress>> newLifelineAddressMap = null;

        try {
            // mycluster -> {nn1 -> hj101:53310, nn2 -> hj102:53310}
            newAddressMap =
                    DFSUtil.getNNServiceRpcAddressesForCluster(conf);
            // null
            newLifelineAddressMap =
                    DFSUtil.getNNLifelineRpcAddressesForCluster(conf);
        } catch (IOException ioe) {
            LOG.warn("Unable to get NameNode addresses.");
        }

        if (newAddressMap == null || newAddressMap.isEmpty()) {
            throw new IOException("No services to connect, missing NameNode " +
                    "address.");
```

```
            }

            synchronized (refreshNamenodesLock) {
                // 刷新 NameNode 地址
                doRefreshNamenodes(newAddressMap, newLifelineAddressMap);
            }
        }
```

```
private void doRefreshNamenodes(
                // mycluster -> {nn1 -> hj101:53310, nn2 -> hj102:53310}
                Map<String, Map<String, InetSocketAddress>> addrMap,
                // null
                Map<String, Map<String, InetSocketAddress>> lifelineAddrMap)
                throws IOException {
            assert Thread.holdsLock(refreshNamenodesLock);

            Set<String> toRefresh = Sets.newLinkedHashSet();
            Set<String> toAdd = Sets.newLinkedHashSet();
            Set<String> toRemove;

            synchronized (this) {
                // Step 1. For each of the new nameservices, figure out whether
                // it's an update of the set of NNs for an existing NS,
                // or an entirely new nameservice.
                for (String nameserviceId : addrMap.keySet()) {
                    if (bpByNameserviceId.containsKey(nameserviceId)) {
                        toRefresh.add(nameserviceId);
                    } else {
                        // add mycluster
                        toAdd.add(nameserviceId);
                    }
                }

                // Step 2. Any nameservices we currently have but are no longer present
                // need to be removed.
                toRemove = Sets.newHashSet(Sets.difference(
                        bpByNameserviceId.keySet(), addrMap.keySet()));

                assert toRefresh.size() + toAdd.size() ==
                        addrMap.size() :
                        "toAdd: " + Joiner.on(",").useForNull("<default>").join(toAdd) +
                                "                                toRemove:                "                +
Joiner.on(",").useForNull("<default>").join(toRemove) +
                                "                                toRefresh:                "                +
```

```
Joiner.on(",").useForNull("<default>").join(toRefresh);


                // Step 3. Start new nameservices
                if (!toAdd.isEmpty()) {
                    // Starting BPOfferServices for nameservices: mycluster
                    LOG.info("Starting BPOfferServices for nameservices: " +
                            Joiner.on(",").useForNull("<default>").join(toAdd));

                    // [mycluster]
                    for (String nsToAdd : toAdd) {
                        // {nn1 -> hj101:53310, nn2 -> hj102:53310}
                        Map<String, InetSocketAddress> nnIdToAddr = addrMap.get(nsToAdd);
                        // null
                        Map<String, InetSocketAddress> nnIdToLifelineAddr =
                                lifelineAddrMap.get(nsToAdd);

                        ArrayList<InetSocketAddress> addrs =
                                Lists.newArrayListWithCapacity(nnIdToAddr.size());
                        ArrayList<InetSocketAddress> lifelineAddrs =
                                Lists.newArrayListWithCapacity(nnIdToAddr.size());

                        for (String nnId : nnIdToAddr.keySet()) {
                            // add hj101:53310
                            // add hj102:53310
                            addrs.add(nnIdToAddr.get(nnId));

                            lifelineAddrs.add(nnIdToLifelineAddr != null ?
                                    nnIdToLifelineAddr.get(nnId) : null);
                        }
                        // 创建 BPOfferService
                        // 一个 BPOfferService 就是一个联邦 也即一个 NameNode HA 组
合
                        // 比如一般配置 NameNode HA 的 nameservice 为 mycluster
                        // mycluster 也即 NameNode 的 HA 组合 存在两个 NameNode
(nn1,nn2)
                        // nn1 或者 nn2 就是 NameNode 的地址 也即 BPServiceActor
                        BPOfferService bpos = createBPOS(nsToAdd, addrs, lifelineAddrs);
                        bpByNameserviceId.put(nsToAdd, bpos);
                        offerServices.add(bpos);
                    }
                }

                // 启动 BPOfferServices 的 BPServiceActor 去跟 NameNode 进行连接并向
```

NameNode 注册

```
                startAll();
            }

            // Step 4. Shut down old nameservices. This happens outside
            // of the synchronized(this) lock since they need to call
            // back to .remove() from another thread
            if (!toRemove.isEmpty()) {
                LOG.info("Stopping BPOfferServices for nameservices: " +
                        Joiner.on(",").useForNull("<default>").join(toRemove));

                for (String nsToRemove : toRemove) {
                    BPOfferService bpos = bpByNameserviceId.get(nsToRemove);
                    bpos.stop();
                    bpos.join();
                    // they will call remove on their own
                }
            }

            // Step 5. Update nameservices whose NN list has changed
            if (!toRefresh.isEmpty()) {
                LOG.info("Refreshing list of NNs for nameservices: " +
                        Joiner.on(",").useForNull("<default>").join(toRefresh));

                for (String nsToRefresh : toRefresh) {
                    BPOfferService bpos = bpByNameserviceId.get(nsToRefresh);
                    Map<String, InetSocketAddress> nnIdToAddr = addrMap.get(nsToRefresh);
                    Map<String, InetSocketAddress> nnIdToLifelineAddr =
                            lifelineAddrMap.get(nsToRefresh);
                    ArrayList<InetSocketAddress> addrs =
                            Lists.newArrayListWithCapacity(nnIdToAddr.size());
                    ArrayList<InetSocketAddress> lifelineAddrs =
                            Lists.newArrayListWithCapacity(nnIdToAddr.size());
                    for (String nnId : nnIdToAddr.keySet()) {
                        addrs.add(nnIdToAddr.get(nnId));
                        lifelineAddrs.add(nnIdToLifelineAddr != null ?
                                nnIdToLifelineAddr.get(nnId) : null);
                    }
                    try {
                        UserGroupInformation.getLoginUser()
                                .doAs(new PrivilegedExceptionAction<Object>() {
                                    @Override
                                    public Object run() throws Exception {
                                        bpos.refreshNNList(addrs, lifelineAddrs);
```

```
                                        return null;
                                    }
                                });
                    } catch (InterruptedException ex) {
                        IOException ioe = new IOException();
                        ioe.initCause(ex.getCause());
                        throw ioe;
                    }
                }
            }
        }
```

3.4.1.2.2.9.1 创建 BPOfferService

```
/**
     * Extracted out for test purposes.
     */
    protected BPOfferService createBPOS(
            final String nameserviceId,
            List<InetSocketAddress> nnAddrs,
            List<InetSocketAddress> lifelineNnAddrs) {
        // 创建 BPOfferService
        return new BPOfferService(nameserviceId, nnAddrs, lifelineNnAddrs, dn);
    }
```

```
BPOfferService(
            // mycluster
            final String nameserviceId,
            // [hj101:53310, hj102:53310]
            List<InetSocketAddress> nnAddrs,
            // null
            List<InetSocketAddress> lifelineNnAddrs,
            DataNode dn) {
        Preconditions.checkArgument(!nnAddrs.isEmpty(),
                "Must pass at least one NN.");
        Preconditions.checkArgument(nnAddrs.size() == lifelineNnAddrs.size(),
                "Must pass same number of NN addresses and lifeline addresses.");
        this.nameserviceId = nameserviceId;
        this.dn = dn;

        for (int i = 0; i < nnAddrs.size(); ++i) {
            // 创建 BPServiceActor(hj101:53310) BPServerActor(hj102:53310)
            this.bpServices.add(new BPServiceActor(nnAddrs.get(i),
```

```
                    lifelineNnAddrs.get(i), this));
            }
        }
```

3.4.1.2.2.9.1.1 创建 BPServiceActor

```
BPServiceActor(InetSocketAddress nnAddr, InetSocketAddress lifelineNnAddr,
                    BPOfferService bpos) {
        // BPOfferService
        this.bpos = bpos;
        // DataNode
        this.dn = bpos.getDataNode();
        // hj101:53310
        this.nnAddr = nnAddr;
        // null
        this.lifelineSender = lifelineNnAddr != null ?
                new LifelineSender(lifelineNnAddr) : null;
        // null
        this.initialRegistrationComplete = lifelineNnAddr != null ?
                new CountDownLatch(1) : null;

        this.dnConf = dn.getDnConf();
        // 创建 IncrementalBlockReportManager
        this.ibrManager = new IncrementalBlockReportManager(
                dnConf.ibrInterval,
                dn.getMetrics());

        prevBlockReportId = ThreadLocalRandom.current().nextLong();
        fullBlockReportLeaseId = 0;
        scheduler = new Scheduler(dnConf.heartBeatInterval,
                dnConf.getLifelineIntervalMs(), dnConf.blockReportInterval,
                dnConf.outliersReportIntervalMs);
        // get the value of maxDataLength.
        this.maxDataLength = dnConf.getMaxDataLength();
    }
```

3.4.1.2.2.9.2 启动 BPOfferServices 的 BPServiceActor 去跟 NameNode 进行连接并向 NameNode 注册

```
synchronized void startAll() throws IOException {
        try {
            UserGroupInformation.getLoginUser().doAs(
                    new PrivilegedExceptionAction<Object>() {
```

```
                                @Override
                                public Object run() throws Exception {
                                    for (BPOfferService bpos : offerServices) {
                                        // 启动 BPOfferService
                                        bpos.start();
                                    }
                                    return null;
                                }
                            });
        } catch (InterruptedException ex) {
            IOException ioe = new IOException();
            ioe.initCause(ex.getCause());
            throw ioe;
        }
    }
```

```
    //This must be called only by blockPoolManager
    void start() {
        for (BPServiceActor actor : bpServices) {
            // 启动 BPServiceActor
            actor.start();
        }
    }
```

3.4.1.2.2.9.2.1 启动 BPOfferService 并调用其 run()

```
//This must be called only by BPOfferService
    void start() {
        if ((bpThread != null) && (bpThread.isAlive())) {
            //Thread is started already
            return;
        }
        // 创建线程
        bpThread = new Thread(this);
        bpThread.setDaemon(true); // needed for JUnit testing
        // 启动线程 调用 run()
        bpThread.start();

        if (lifelineSender != null) {
            lifelineSender.start();
        }
    }
```

3.4.1.2.2.9.2.2 DataNode 向 NameNode 注册以及维持心跳 (查看 3.4.2.4 章节)

```
/**
     * No matter what kind of exception we get, keep retrying to offerService().
     * That's the loop that connects to the NameNode and provides basic DataNode
     * functionality.
     *
     * Only stop when "shouldRun" or "shouldServiceRun" is turned off, which can
     * happen either at shutdown or due to refreshNamenodes.
     */
    @Override
    public void run() {
        LOG.info(this + " starting to offer service");

        try {
            while (true) {
                // init stuff
                try {
                    // setup storage
                    // DataNode 连接 NameNode 并向 NameNode 注册 DataNode 信
息
                    connectToNNAndHandshake();
                    break;
                } catch (IOException ioe) {
                    // Initial handshake, storage recovery or registration failed
                    runningState = RunningState.INIT_FAILED;
                    if (shouldRetryInit()) {
                        // Retry until all namenode's of BPOS failed initialization
                        LOG.error("Initialization failed for " + this + " "
                                + ioe.getLocalizedMessage());
                        sleepAndLogInterrupts(5000, "initializing");
                    } else {
                        runningState = RunningState.FAILED;
                        LOG.error("Initialization failed for " + this + ". Exiting. ", ioe);
                        return;
                    }
                }
            }

            runningState = RunningState.RUNNING;
            if (initialRegistrationComplete != null) {
                initialRegistrationComplete.countDown();
            }

            while (shouldRun()) {
```

```
                try {
                    // DataNode 与 NameNode 维持心跳
                    offerService();
                } catch (Exception ex) {
                    LOG.error("Exception in BPOfferService for " + this, ex);
                    sleepAndLogInterrupts(5000, "offering service");
                }
            }
            runningState = RunningState.EXITED;
        } catch (Throwable ex) {
            LOG.warn("Unexpected exception in block pool " + this, ex);
            runningState = RunningState.FAILED;
        } finally {
            LOG.warn("Ending block pool service for: " + this);
            cleanUp();
        }
    }
```

3.4.2 启动 DataNode

```
/**
     * Start a single datanode daemon and wait for it to finish.
     * If this thread is specifically interrupted, it will stop waiting.
     */
    public void runDatanodeDaemon() throws IOException {
        // 启动 BlockPoolManager
        blockPoolManager.startAll();

        // start dataXceiveServer
        // 启动 DataXceiverServer
        dataXceiverServer.start();

        if (localDataXceiverServer != null) {
            localDataXceiverServer.start();
        }
        ipcServer.setTracer(tracer);
        // 启动 RpcServer
        ipcServer.start();

        startPlugins(getConf());
    }
```

3.4.2.1 启动 BlockPoolManager

```
synchronized void startAll() throws IOException {
        try {
            UserGroupInformation.getLoginUser().doAs(
                    new PrivilegedExceptionAction<Object>() {
                        @Override
                        public Object run() throws Exception {
                            for (BPOfferService bpos : offerServices) {
                                // 启动 BPOfferService (前面已经启动过了)
                                bpos.start();
                            }
                            return null;
                        }
                    });
        } catch (InterruptedException ex) {
            IOException ioe = new IOException();
            ioe.initCause(ex.getCause());
            throw ioe;
        }
    }
```

3.4.2.2 启动 DataXceiverServer 调用其 run()

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

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

                // 将 DataXceiverServer 的 TcpPeerServer TCP-Server 接收到一个 Socket
```

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

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

            // if in restart prep stage, notify peers before closing them.
```

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

3.4.2.2.1 TcpPeerServer 等待接收 SocketChannel 连接 (阻塞)

```
    @Override
    public Peer accept() throws IOException, SocketTimeoutException {
        // serverSocket 绑定默认为
        Peer peer = DFSUtilClient.peerFromSocket(serverSocket.accept());
        return peer;
    }
```

```
public static Peer peerFromSocket(Socket socket)
            throws IOException {
        Peer peer;
        boolean success = false;
        try {
            // TCP_NODELAY is crucial here because of bad interactions between
            // Nagle's Algorithm and Delayed ACKs. With connection keepalive
            // between the client and DN, the conversation looks like:
            //     1. Client -> DN: Read block X
            //     2. DN -> Client: data for block X
            //     3. Client -> DN: Status OK (successful read)
            //     4. Client -> DN: Read block Y
            // The fact that step #3 and #4 are both in the client->DN direction
            // triggers Nagling. If the DN is using delayed ACKs, this results
            // in a delay of 40ms or more.
            //
            // TCP_NODELAY disables nagling and thus avoids this performance
```

```
                // disaster.
                socket.setTcpNoDelay(true);
                // 接收到 SocketChannel
                SocketChannel channel = socket.getChannel();
                if (channel == null) {
                    // 创建 BasicInetPeer
                    peer = new BasicInetPeer(socket);
                } else {
                    peer = new NioInetPeer(socket);
                }
                success = true;
                return peer;
            } finally {
                if (!success) {
                    // peer is always null so no need to call peer.close().
                    socket.close();
                }
            }
        }
```

3.4.2.2.1.1 将接收的 SocketChannel 封装成 BasicInetPeer

```
public BasicInetPeer(Socket socket) throws IOException {
    // SocketChannel
    this.socket = socket;
    // SocketChannel 的 output 输出流
    this.out = socket.getOutputStream();
    // SocketChannel 的 input 输入流
    this.in = socket.getInputStream();
    // 判断是否为本地短路
    this.isLocal = socket.getInetAddress().equals(socket.getLocalAddress());
}
```

3.4.2.2.2 收到一个 Socket (BasicInetPeer) 封装成 DataXceiver 线程 并调用其 run() 进行接收数据处理

```
public static DataXceiver create(Peer peer, DataNode dn,
        DataXceiverServer dataXceiverServer) throws IOException {
    // 创建 DataXceiver
    return new DataXceiver(peer, dn, dataXceiverServer);
}
```

```
private DataXceiver(Peer peer, DataNode datanode,
        DataXceiverServer dataXceiverServer) throws IOException {
```

```
        super(datanode.getTracer());
        this.peer = peer;
        this.dnConf = datanode.getDnConf();
        this.socketIn = peer.getInputStream();
        this.socketOut = peer.getOutputStream();
        this.datanode = datanode;
        this.dataXceiverServer = dataXceiverServer;
        this.connectToDnViaHostname = datanode.getDnConf().connectToDnViaHostname;
        this.ioFileBufferSize = DFSUtilClient.getIoFileBufferSize(datanode.getConf());
        this.smallBufferSize = DFSUtilClient.getSmallBufferSize(datanode.getConf());
        remoteAddress = peer.getRemoteAddressString();
        final int colonIdx = remoteAddress.indexOf(':');
        remoteAddressWithoutPort =
              (colonIdx < 0) ? remoteAddress : remoteAddress.substring(0, colonIdx);
        localAddress = peer.getLocalAddressString();

        LOG.debug("Number of active connections is: {}",
              datanode.getXceiverCount());
    }
```

3.4.2.2.2.1 调用 DataXceiver.run() 进行数据处理

```
```

3.4.2.3 启动 RpcServer (0.0.0.0:9867)

```
/**
     * Starts the service.  Must be called before any calls will be handled.
     */
    public synchronized void start() {
        // 启动 Responder 线程 (本质调用 Responder run())
        responder.start();
        // 启动 Listener 线程 (本质调用 Listener run())
        listener.start();

        if (auxiliaryListenerMap != null && auxiliaryListenerMap.size() > 0) {
            for (Listener newListener : auxiliaryListenerMap.values()) {
                newListener.start();
            }
        }

        // 创建 Handler 线程数组 默认大小 10
        handlers = new Handler[handlerCount];
```

```
            // 初始化 Handler 线程并启动
            for (int i = 0; i < handlerCount; i++) {
                handlers[i] = new Handler(i);
                handlers[i].start();
            }
        }
```

3.4.2.4 DataNode 向 NameNode 注册以及心跳维持

入口类：org.apache.hadoop.hdfs.server.datanode.BPServiceActor.run()

3.4.2.4.1 DataNode 向 NameNode 注册

connectToNNAndHandshake()

```
private void connectToNNAndHandshake() throws IOException {
            // get NN proxy
            // 创建 DatanodeProtocolClientSideTranslatorPB (通讯协议接口为 DatanodeProtocol) 并获取 NameNode 代理
            bpNamenode = dn.connectToNN(nnAddr);

            // First phase of the handshake with NN - get the namespace
            // info.
            // 发送一个 Rpc 请求获取 NameNode 信息 (阻塞直到获取成功)
            NamespaceInfo nsInfo = retrieveNamespaceInfo();

            // Verify that this matches the other NN in this HA pair.
            // This also initializes our block pool in the DN if we are
            // the first NN connection for this BP.
            // 更新 NameNode 信息并加载 DataNode 数据存储信息
            bpos.verifyAndSetNamespaceInfo(this, nsInfo);

            /* set thread name again to include NamespaceInfo when it's available. */
            this.bpThread.setName(formatThreadName("heartbeating", nnAddr));

            // Second phase of the handshake with the NN.
            // 往 NameNode 注册 DataNode
            register(nsInfo);
        }
```

3.4.2.4.1.1 获取 NameNode 代理 (通讯协议接口为 DatanodeProtocol)

```
    /**
     * Connect to the NN. This is separated out for easier testing.
     */
```

```
    DatanodeProtocolClientSideTranslatorPB connectToNN(
            InetSocketAddress nnAddr) throws IOException {
        // 创建 DatanodeProtocolClientSideTranslatorPB
        return new DatanodeProtocolClientSideTranslatorPB(nnAddr, getConf());
    }
```

```
public DatanodeProtocolClientSideTranslatorPB(InetSocketAddress nameNodeAddr,
                                              Configuration conf) throws
IOException {
        // 设置通讯协议为 DatanodeProtocol
        RPC.setProtocolEngine(conf, DatanodeProtocolPB.class,
                ProtobufRpcEngine.class);
        UserGroupInformation ugi = UserGroupInformation.getCurrentUser();

        // 创建 NameNode 代理对象
        rpcProxy = createNamenode(nameNodeAddr, conf, ugi);
    }
```

```
private static DatanodeProtocolPB createNamenode(
            InetSocketAddress nameNodeAddr, Configuration conf,
            UserGroupInformation ugi) throws IOException {
        // 获取 DatanodeProtocol 代理对象
        return RPC.getProxy(DatanodeProtocolPB.class,
                RPC.getProtocolVersion(DatanodeProtocolPB.class),
                nameNodeAddr,
                ugi,
                conf,
                NetUtils.getSocketFactory(conf, DatanodeProtocolPB.class)
        );
    }
```

3.4.2.4.1.2 发送一个 Rpc 请求获取 NameNode 信息 (阻塞直到获取成功)

```
/**
     * Perform the first part of the handshake with the NameNode.
     * This calls <code>versionRequest</code> to determine the NN's
     * namespace and version info. It automatically retries until
     * the NN responds or the DN is shutting down.
     *
     * @return the NamespaceInfo
     */
    @VisibleForTesting
    NamespaceInfo retrieveNamespaceInfo() throws IOException {
        NamespaceInfo nsInfo = null;
        while (shouldRun()) {
```

```
                try {
                    // 发送 Rpc Request 获取 NameNode 信息 返回 NamespaceInfo
                    // 调用 NameNodeRpcServer.versionRequest()
                    nsInfo = bpNamenode.versionRequest();
                    LOG.debug(this + " received versionRequest response: " + nsInfo);
                    break;
                } catch (SocketTimeoutException e) {    // namenode is busy
                    LOG.warn("Problem connecting to server: " + nnAddr);
                } catch (IOException e) {    // namenode is not available
                    LOG.warn("Problem connecting to server: " + nnAddr);
                }

                // try again in a second
                sleepAndLogInterrupts(5000, "requesting version info from NN");
            }

            if (nsInfo != null) {
                checkNNVersion(nsInfo);
            } else {
                throw new IOException("DN shut down before block pool connected");
            }
            return nsInfo;
        }
```

3.4.2.4.1.2.1 调用 NameNodeRpcServer.versionRequest()

```
```

3.4.2.4.1.3 更新 NameNode 信息并加载 DataNode 数据存储信息

```
/**
     * Called by the BPServiceActors when they handshake to a NN.
     * If this is the first NN connection, this sets the namespace info
     * for this BPOfferService. If it's a connection to a new NN, it
     * verifies that this namespace matches (eg to prevent a misconfiguration
     * where a StandbyNode from a different cluster is specified)
     */
    void verifyAndSetNamespaceInfo(BPServiceActor actor, NamespaceInfo nsInfo)
            throws IOException {
        writeLock();

        if (nsInfo.getState() == HAServiceState.ACTIVE
                && bpServiceToActive == null) {
            LOG.info("Acknowledging ACTIVE Namenode during handshake" + actor);
            bpServiceToActive = actor;
```

```
            }

            try {
                // DataNode 更新 NameNode 信息
                if (setNamespaceInfo(nsInfo) == null) {
                    boolean success = false;

                    // Now that we know the namespace ID, etc, we can pass this to the DN.
                    // The DN can now initialize its local storage if we are the
                    // first BP to handshake, etc.
                    try {
                        // 初始化 DataNode BlockPoolManager
                        dn.initBlockPool(this);
                        success = true;
                    } finally {
                        if (!success) {
                            // The datanode failed to initialize the BP. We need to reset
                            // the namespace info so that other BPService actors still have
                            // a chance to set it, and re-initialize the datanode.
                            setNamespaceInfo(null);
                        }
                    }
                }
            } finally {
                writeUnlock();
            }
        }
```

3.4.2.4.1.4 往 NameNode 注册 DataNode

```
/**
     * Register one bp with the corresponding NameNode
     * <p>
     * The bpDatanode needs to register with the namenode on startup in order
     * 1) to report which storage it is serving now and
     * 2) to receive a registrationID
     *
     * issued by the namenode to recognize registered datanodes.
     *
     * @param nsInfo current NamespaceInfo
     * @see FSNamesystem#registerDatanode(DatanodeRegistration)
     * @throws IOException
     */
    void register(NamespaceInfo nsInfo) throws IOException {
        // The handshake() phase loaded the block pool storage
```

```
            // off disk - so update the bpRegistration object from that info
            // 创建 DataNode 注册 DatanodeRegistration
            DatanodeRegistration newBpRegistration = bpos.createRegistration();

            // (Datanode Uuid 01263ed0-7bb5-4dec-845b-97a8d2d5d052) service to
hj101/192.168.2.101:53310 beginning handshake with NN
            LOG.info(this + " beginning handshake with NN");

            while (shouldRun()) {
                try {
                    // Use returned registration from namenode with updated fields
                    // DataNode 向 NameNode 注册并返回注册结果信息
                    // 调用 NameNodeRpcServer 的 registerDatanode()
                    newBpRegistration = bpNamenode.registerDatanode(newBpRegistration);
                    newBpRegistration.setNamespaceInfo(nsInfo);
                    bpRegistration = newBpRegistration;
                    break;
                } catch (EOFException e) {  // namenode might have just restarted
                    LOG.info("Problem connecting to server: " + nnAddr + " :"
                            + e.getLocalizedMessage());
                    sleepAndLogInterrupts(1000, "connecting to server");
                } catch (SocketTimeoutException e) {  // namenode is busy
                    LOG.info("Problem connecting to server: " + nnAddr);
                    sleepAndLogInterrupts(1000, "connecting to server");
                }
            }

            // Block pool <registering> (Datanode Uuid unassigned) service to
            // hj101/192.168.2.101:53310 starting to offer service
            LOG.info("Block pool " + this + " successfully registered with NN");
            bpos.registrationSucceeded(this, bpRegistration);

            // reset lease id whenever registered to NN.
            // ask for a new lease id at the next heartbeat.
            fullBlockReportLeaseId = 0;

            // random short delay - helps scatter the BR from all DNs
            // 定时上报 Block 调度器更新上报时间
            scheduler.scheduleBlockReport(dnConf.initialBlockReportDelayMs);
        }
```

3.4.2.4.1.4.1 调用 NameNodeRpcServer 的 registerDatanode()

```
```

3.4.2.4.2 DataNode 与 NameNode 维持心跳

offerService();

```
/**
     * Main loop for each BP thread. Run until shutdown,
     * forever calling remote NameNode functions.
     */
    private void offerService() throws Exception {
        LOG.info("For namenode " + nnAddr + " using"
                + " BLOCKREPORT_INTERVAL of " + dnConf.blockReportInterval + "msec"
                + " CACHEREPORT_INTERVAL of " + dnConf.cacheReportInterval + "msec"
                + " Initial delay: " + dnConf.initialBlockReportDelayMs + "msec"
                + "; heartBeatInterval=" + dnConf.heartBeatInterval
                + (lifelineSender != null ?
                "; lifelineIntervalMs=" + dnConf.getLifelineIntervalMs() : ""));

        //
        // Now loop for a long time....
        //
        while (shouldRun()) {
            try {
                DataNodeFaultInjector.get().startOfferService();
                final long startTime = scheduler.monotonicNow();

                //
                // Every so often, send heartbeat or block-report
                // 每隔 3s 发送心跳
                final boolean sendHeartbeat = scheduler.isHeartbeatDue(startTime);
                HeartbeatResponse resp = null;
                if (sendHeartbeat) {
                    //
                    // All heartbeat messages include following info:
                    // -- Datanode name
                    // -- data transfer port
                    // -- Total capacity
                    // -- Bytes remaining
                    //
                    boolean requestBlockReportLease = (fullBlockReportLeaseId == 0) &&
                            scheduler.isBlockReportDue(startTime);
                    if (!dn.areHeartbeatsDisabledForTests()) {
                        // 发送心跳返回结果
                        resp = sendHeartBeat(requestBlockReportLease);
                        assert resp != null;
                        if (resp.getFullBlockReportLeaseId() != 0) {
                            if (fullBlockReportLeaseId != 0) {
```

```
                                    LOG.warn(nnAddr + " sent back a full block report lease "
+
                                            "ID of 0x" +

Long.toHexString(resp.getFullBlockReportLeaseId()) +
                                            ", but we already have a lease ID of 0x" +
                                            Long.toHexString(fullBlockReportLeaseId) + ". "
+
                                            "Overwriting old lease ID.");
                                }
                                fullBlockReportLeaseId = resp.getFullBlockReportLeaseId();
                            }
                            dn.getMetrics().addHeartbeat(scheduler.monotonicNow()           -
startTime);

                            // If the state of this NN has changed (eg STANDBY->ACTIVE)
                            // then let the BPOfferService update itself.
                            //
                            // Important that this happens before processCommand below,
                            // since the first heartbeat to a new active might have commands
                            // that we should actually process.
                            bpos.updateActorStatesFromHeartbeat(
                                    this, resp.getNameNodeHaState());
                            state = resp.getNameNodeHaState().getState();

                            if (state == HAServiceState.ACTIVE) {
                                handleRollingUpgradeStatus(resp);
                            }

                            long startProcessCommands = monotonicNow();
                            // 判断 DataNode 发送心跳是否返回执行命令
                            if (!processCommand(resp.getCommands()))
                                continue;
                            long endProcessCommands = monotonicNow();
                            if (endProcessCommands - startProcessCommands > 2000) {
                                LOG.info("Took       "       +       (endProcessCommands       -
startProcessCommands)
                                        + "ms to process " + resp.getCommands().length
                                        + " commands from NN");
                            }
                        }
                    }
                    if (!dn.areIBRDisabledForTests() &&
                            (ibrManager.sendImmediately() || sendHeartbeat)) {
```

```
                    ibrManager.sendIBRs(bpNamenode, bpRegistration,
                            bpos.getBlockPoolId());
                }

                List<DatanodeCommand> cmds = null;
                boolean forceFullBr =
                      scheduler.forceFullBlockReport.getAndSet(false);
                if (forceFullBr) {
                    LOG.info("Forcing a full block report to " + nnAddr);
                }
                if ((fullBlockReportLeaseId != 0) || forceFullBr) {
                    cmds = blockReport(fullBlockReportLeaseId);
                    fullBlockReportLeaseId = 0;
                }
                processCommand(cmds == null ? null : cmds.toArray(new
DatanodeCommand[cmds.size()]));

                if (!dn.areCacheReportsDisabledForTests()) {
                    DatanodeCommand cmd = cacheReport();
                    processCommand(new DatanodeCommand[]{cmd});
                }

                if (sendHeartbeat) {
                    dn.getMetrics().addHeartbeatTotal(
                            scheduler.monotonicNow() - startTime);
                }

                // There is no work to do;  sleep until hearbeat timer elapses,
                // or work arrives, and then iterate again.
                ibrManager.waitTillNextIBR(scheduler.getHeartbeatWaitTime());
            } catch (RemoteException re) {
                String reClass = re.getClassName();
                if (UnregisteredNodeException.class.getName().equals(reClass) ||
                        DisallowedDatanodeException.class.getName().equals(reClass) ||
                        IncorrectVersionException.class.getName().equals(reClass)) {
                    LOG.warn(this + " is shutting down", re);
                    shouldServiceRun = false;
                    return;
                }
                LOG.warn("RemoteException in offerService", re);
                sleepAfterException();
            } catch (IOException e) {
                LOG.warn("IOException in offerService", e);
                sleepAfterException();
```

```
                } finally {
                    DataNodeFaultInjector.get().endOfferService();
                }
                processQueueMessages();
            } // while (shouldRun())
        } // offerService
```

3.4.2.4.2.1 发送心跳并调用 NameNodeRpcServer.sendHeartbeat()

```
HeartbeatResponse sendHeartBeat(boolean requestBlockReportLease)
                throws IOException {
            // 更新下一次心跳时间
            scheduler.scheduleNextHeartbeat();
            // 获取 DataNode Block 信息
            StorageReport[] reports =
                    dn.getFSDataset().getStorageReports(bpos.getBlockPoolId());
            if (LOG.isDebugEnabled()) {
                LOG.debug("Sending heartbeat with " + reports.length +
                        " storage reports from service actor: " + this);
            }

            // 更新下一次心跳时间
            final long now = monotonicNow();
            scheduler.updateLastHeartbeatTime(now);

            VolumeFailureSummary volumeFailureSummary = dn.getFSDataset()
                    .getVolumeFailureSummary();
            int numFailedVolumes = volumeFailureSummary != null ?
                    volumeFailureSummary.getFailedStorageLocations().length : 0;
            final boolean outliersReportDue = scheduler.isOutliersReportDue(now);
            final SlowPeerReports slowPeers =
                    outliersReportDue && dn.getPeerMetrics() != null ?
                            SlowPeerReports.create(dn.getPeerMetrics().getOutliers()) :
                            SlowPeerReports.EMPTY_REPORT;
            final SlowDiskReports slowDisks =
                    outliersReportDue && dn.getDiskMetrics() != null ?

SlowDiskReports.create(dn.getDiskMetrics().getDiskOutliersStats()) :
                            SlowDiskReports.EMPTY_REPORT;

            // 发送心跳 调用 NameNodeRpcServer.sendHeartbeat()
            HeartbeatResponse response = bpNamenode.sendHeartbeat(bpRegistration,
                    reports,
                    dn.getFSDataset().getCacheCapacity(),
                    dn.getFSDataset().getCacheUsed(),
```

```
                    dn.getXmitsInProgress(),
                    dn.getXceiverCount(),
                    numFailedVolumes,
                    volumeFailureSummary,
                    requestBlockReportLease,
                    slowPeers,
                    slowDisks);

        if (outliersReportDue) {
            // If the report was due and successfully sent, schedule the next one.
            scheduler.scheduleNextOutlierReport();
        }
        return response;
    }
```

3.4.2.4.2.1.1 调用 NameNodeRpcServer.sendHeartbeat()

```
```

3.4.2.4.2.2 心跳结果返回是否存在执行命令 (存在则执行)

```
```

3.4.2.4.2.2.1 处理 NameNode 心跳返回结果命令-上报 Block

```
/**
     * Report the list blocks to the Namenode
     * @return DatanodeCommands returned by the NN. May be null.
     * @throws IOException
     */
    List<DatanodeCommand> blockReport(long fullBrLeaseId) throws IOException {
        final ArrayList<DatanodeCommand> cmds = new ArrayList<DatanodeCommand>();

        // Flush any block information that precedes the block report. Otherwise
        // we have a chance that we will miss the delHint information
        // or we will report an RBW replica after the BlockReport already reports
        // a FINALIZED one.
        ibrManager.sendIBRs(bpNamenode, bpRegistration,
                bpos.getBlockPoolId());

        long brCreateStartTime = monotonicNow();
        Map<DatanodeStorage, BlockListAsLongs> perVolumeBlockLists =
                dn.getFSDataset().getBlockReports(bpos.getBlockPoolId());

        // Convert the reports to the format expected by the NN.
        int i = 0;
```

```
            int totalBlockCount = 0;
            StorageBlockReport reports[] =
                    new StorageBlockReport[perVolumeBlockLists.size()];

            for (Map.Entry<DatanodeStorage, BlockListAsLongs> kvPair :
perVolumeBlockLists.entrySet()) {
                BlockListAsLongs blockList = kvPair.getValue();
                reports[i++] = new StorageBlockReport(kvPair.getKey(), blockList);
                totalBlockCount += blockList.getNumberOfBlocks();
            }

            // Send the reports to the NN.
            int numReportsSent = 0;
            int numRPCs = 0;
            boolean success = false;
            long brSendStartTime = monotonicNow();
            long reportId = generateUniqueBlockReportId();
            boolean useBlocksBuffer =
                    bpRegistration.getNamespaceInfo().isCapabilitySupported(
                            NamespaceInfo.Capability.STORAGE_BLOCK_REPORT_BUFFERS);
            blockReportSizes.clear();
            try {
                if (totalBlockCount < dnConf.blockReportSplitThreshold) {
                    // Below split threshold, send all reports in a single message.
                    DatanodeCommand cmd = bpNamenode.blockReport(
                            bpRegistration, bpos.getBlockPoolId(), reports,
                            new BlockReportContext(1, 0, reportId, fullBrLeaseId, true));
                    blockReportSizes.add(
                            calculateBlockReportPBSize(useBlocksBuffer, reports));
                    numRPCs = 1;
                    numReportsSent = reports.length;
                    if (cmd != null) {
                        cmds.add(cmd);
                    }
                } else {
                    // Send one block report per message.
                    for (int r = 0; r < reports.length; r++) {
                        StorageBlockReport singleReport[] = {reports[r]};
                        //  上报  Block to NameNode
                        DatanodeCommand cmd = bpNamenode.blockReport(
                                bpRegistration, bpos.getBlockPoolId(), singleReport,
                                new BlockReportContext(reports.length, r, reportId,
                                        fullBrLeaseId, true));
                        blockReportSizes.add(
```

```
                                    calculateBlockReportPBSize(useBlocksBuffer, singleReport));
                            numReportsSent++;
                            numRPCs++;
                            if (cmd != null) {
                                cmds.add(cmd);
                            }
                        }
                    }
                    success = true;
                } finally {
                    // Log the block report processing stats from Datanode perspective
                    long brSendCost = monotonicNow() - brSendStartTime;
                    long brCreateCost = brSendStartTime - brCreateStartTime;
                    dn.getMetrics().addBlockReport(brSendCost);
                    final int nCmds = cmds.size();
                    LOG.info((success ? "S" : "Uns") +
                            "uccessfully sent block report 0x" +
                            Long.toHexString(reportId) + ",  containing " + reports.length +
                            " storage report(s), of which we sent " + numReportsSent + "." +
                            " The reports had " + totalBlockCount +
                            " total blocks and used " + numRPCs +
                            " RPC(s). This took " + brCreateCost +
                            " msec to generate and " + brSendCost +
                            " msecs for RPC and NN processing." +
                            " Got back " +
                            ((nCmds == 0) ? "no commands" :
                                    ((nCmds == 1) ? "one command: " + cmds.get(0) :
                                            (nCmds + " commands: " + Joiner.on("; 
").join(cmds)))) +
                            ".");
                }
                scheduler.updateLastBlockReportTime(monotonicNow());
                scheduler.scheduleNextBlockReport();
                return cmds.size() == 0 ? null : cmds;
            }
```

3.4.2.4.2.2.1.1 调用 NameNodeRpcServer.blockReport()

```
@Override // DatanodeProtocol
    public DatanodeCommand blockReport(final DatanodeRegistration nodeReg,
                                       String poolId, final StorageBlockReport[] reports,
                                       final BlockReportContext context) throws 
IOException {
        checkNNStartup();
```

```
        verifyRequest(nodeReg);
        if (blockStateChangeLog.isDebugEnabled()) {
            blockStateChangeLog.debug("*BLOCK* NameNode.blockReport: "
                    + "from " + nodeReg + ", reports.length=" + reports.length);
        }
        final BlockManager bm = namesystem.getBlockManager();
        boolean noStaleStorages = false;
        try {
            if (bm.checkBlockReportLease(context, nodeReg)) {
                for (int r = 0; r < reports.length; r++) {
                    final BlockListAsLongs blocks = reports[r].getBlocks();
                    //
                    // BlockManager.processReport accumulates information of prior calls
                    // for the same node and storage, so the value returned by the last
                    // call of this loop is the final updated value for noStaleStorage.
                    //
                    final int index = r;
                    noStaleStorages = bm.runBlockOp(new Callable<Boolean>() {
                        @Override
                        public Boolean call() throws IOException {
                            return bm.processReport(nodeReg,
reports[index].getStorage(),
                                    blocks, context);
                        }
                    });
                }
            }
        } catch (UnregisteredNodeException une) {
            LOG.debug("Datanode {} is attempting to report but not register yet.",
                    nodeReg);
            return RegisterCommand.REGISTER;
        }
        bm.removeBRLeaseIfNeeded(nodeReg, context);

        BlockManagerFaultInjector.getInstance().
                incomingBlockReportRpc(nodeReg, context);

        if (nn.getFSImage().isUpgradeFinalized() &&
                !namesystem.isRollingUpgrade() &&
                nn.isActiveState() &&
                noStaleStorages) {
            return new FinalizeCommand(poolId);
        }
```

```
        return null;
    }
```