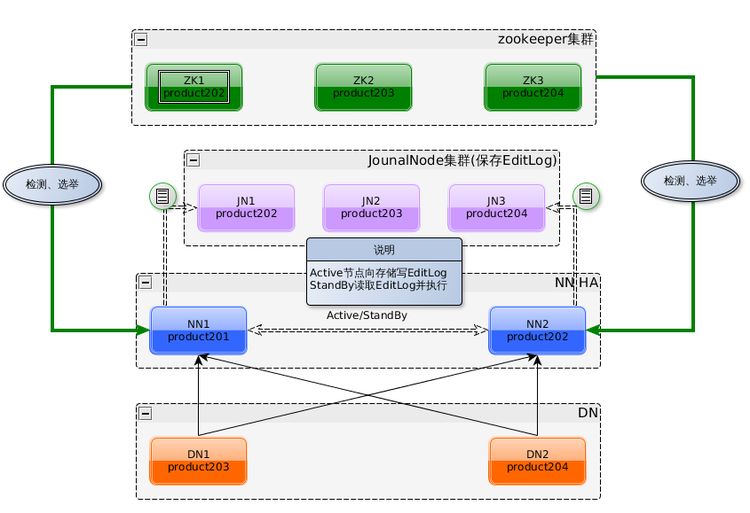
Hadoop JournalNode实现元数据共享

Hadoop HDFS的高可用性原理如下图所示：



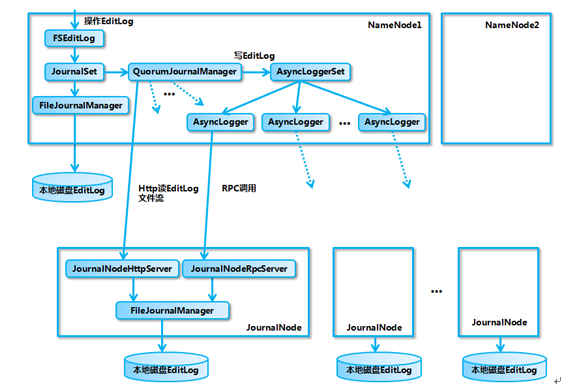
在启用高可用时，启动状态为Active和Standby的两个NameNode，两者之间形成互备，其中只有Active NameNode对外提供读写服务（两个NN同时对外提供服务时可能会造成脑裂），这两者之间通过ZKFailoverController实现主备选举和切换。

Active和Standby NameNode之间要保证元数据一致，实现的方式是使用共享存储系统：NFS或者使用Quorum Journal Nodes。QJM是基于Paxos算法的HDFS HA方案，基本原理是启动2N+1台JN存储Edit log，每次写操作有大于(N+1)写入成功则认为该次写成功，数据不会丢失；该算法能容忍最多N台机器挂掉，否则该算法失效。

在Active NN上的任何修改操作，都会通过记录Log的形式保存到至少半数以上的JN上，Standby NN检测到JN中log的变化并同步到自己的目录镜像树。当发生故障时，Active的NN挂掉，Standby NN在成为Active NN之前会读取所有的JN中修改日志，这样就可以保证与挂掉的NN目录镜像树一致，然后无缝接替它的职责，维护来自客户端请求，从而达到一个高可用的目录。

# 基于QJM的HA执行流程

基于QJM的共享存储系统主要用于保存EditLog（不保存FSImage文件，FSImage文件还是保存在NN的本地磁盘上。基本思想是来自于Paxos算法，启动多个JournalNode节点组成集群来存储EditLog。NameNode写入EditLog的过程如下图所示：



NN写EditLog的时候，除了向本地磁盘写入EditLog之外，也会并行向JournalNode集群之中的每一个JournalNode发送写请求，只要大多数JN节点返回成功就认为EditLog写入成功。如果有2N+1台JN，那么根据大多数原则，最多可以容忍有N台JN节点挂掉。

涉及到的概念如下：

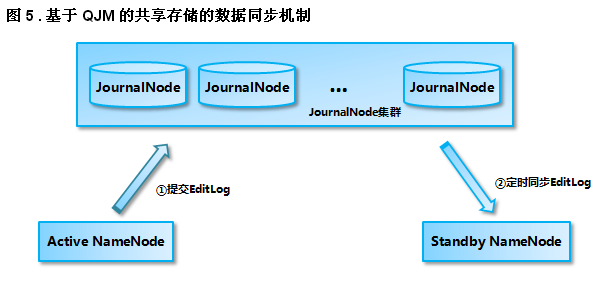
* FSEditLog，这个类封装了对EditLog的所有操作，是NN对EditLog的所有操作入口
* JournalSet，这个类封装了对本地磁盘和JN集群上的EditLog操作，内部包含了两类：FileJournalManager，实现对本地磁盘上的EditLog操作；QuorumJournalManager，实现对JN集群上共享目录的EditLog操作。FSEditLog只会调用JournalSet的相关方法，而不会直接使用FJM和QJM
* FileJournalManager，封装了对本地磁盘上的Edit文件操作，不仅NN向本地磁盘写入EditLog使用FJM，JournalNode向本地磁盘写入EditLog的时候也复用FJM的代码和逻辑
* QuorumJournalManager，封装了对JN集群的EditLog操作，根据URI创建负责与JN集群的通信类AsyncLoggerSet。对于读操作，QJM则是通过Http接口从JN上的Journal-

NodeHttpServer读取EditLog数据

* AsyncLoggerSet，内部包含了与JN集群进行通信的AsyncLogger列表，每一个AsyncLogger会对应一个JournalNode列表，另外AsyncLoggerSet也包含了用于等待大多数JN返回结果的工具类方法给QJM使用
* AsyncLogger，具体的实现类是IPCLoggerChannel，其执行方法调用的时候，会把调用提交到一个单线程线程池中，由线程池来负责向对应的JN的JournalNodeRpcServer发送RPC请求
* JournalNodeRpcServer，运行在JN进程中的RPC服务，接收NameNode端AsyncLogger的RPC请求
* JournalNodeHttpServer，运行在JN节点进程中的Http服务，用于接收处于Standby状态的NN和其他JN同步的EditLog文件流请求

# 数据同步机制分析

Active NN和Standby NN使用JN集群来进行数据同步的过程如下图所示：



Active NN首先把EditLog提交到JN集群，然后Standby NN再从JN集群定时同步EditLog

## 2.1 Active NN提交Edit Log到JN集群

当处于Active状态的NN调用FSEditLog类的logSync方法来提交EditLog的时候，会通过JournalSet同时向本地磁盘目录和JN集群上的共享存储目录写入EditLog

*protected void logSync(long mytxid) {*

*try {*

*EditLogOutputStream logStream = null;*

*......*

*editLogStream.setReadyToFlush();*

*logStream = editLogStream;*

*......*

*try {*

*if (logStream != null) {*

*logStream.flush();*

*}*

*} .....*

*}*

其中JournalSet与EditLogOutputStream相关的类如下所示



基于QJM相关的类为QuorumOutputStream和QuorumJournalManager。FSEditLog中的logSync调用QuorumOutputStream#flush将文件系统的操作写入JN节点，其中调用时使用了Java闭包技术，通过JournalSetOutputStream来封装，如下所示：

*protected void flushAndSync(final boolean durable) throws IOException {*

*mapJournalsAndReportErrors(new JournalClosure() {*

*@Override*

*public void apply(JournalAndStream jas) throws IOException {*

*if (jas.isActive()) {*

*jas.getCurrentStream().flushAndSync(durable);*

*}*

*}*

*}, "flushAndSync");*

*}*

执行时调用JournalSet#mapJournalsAndReportError来处理flushAndSync操作时遇到的异常信息

*private void mapJournalsAndReportErrors(*

*JournalClosure closure, String status) throws IOException{*

*List<JournalAndStream> badJAS = Lists.newLinkedList();*

*for (JournalAndStream jas : journals) {*

*try {*

*closure.apply(jas);*

*} catch (Throwable t) {*

*if (jas.isRequired()) {*

*final String msg = "Error: " + status + " failed for required journal ("*

*+ jas + ")";*

*LOG.fatal(msg, t);*

*//写入异常时，关闭Journal Stream，不再尝试写入，*

*//JN基于Paxos算法，满足大多数写入成功，则不会抛出异常*

*abortAllJournals();*

*//NN执行时，写入JN EditLog失败，则认为该NN不可用，主动退出*

*terminate(1, msg);*

*} else {*

*LOG.error("Error: " + status + " failed for (journal " + jas + ")", t);*

*badJAS.add(jas);*

*}*

*} …..}}*

NN向JournalNode集群写入Edit Log，是通过并行调用每一个JN的QjournalProtocol RPC接口的journal方法实现的，如果对大多数JournalNode的journal方法调用成功，则认为提交EditLog成功，否则NameNode就认为这次提交的EditLog失败。提交EditLog失败会导致Active NN关闭JournalSet之后退出进程，留待处于Standby状态的NN接管之后进行数据恢复，其相关类如下图所示:



其执行方法为：IPCLoggerChannel#sendEdits

*public ListenableFuture<Void> sendEdits(*

*final long segmentTxId, final long firstTxnId,*

*final int numTxns, final byte[] data) {*

*ListenableFuture<Void> ret = null;*

*try {*

*ret = singleThreadExecutor.submit(new Callable<Void>() {*

*@Override*

*public Void call() throws IOException {*

*try { //通过rpc，写入Edit Log*

*getProxy().journal(createReqInfo(),*

*segmentTxId, firstTxnId, numTxns, data);*

*} catch (IOException e) { //写入失败，抛出异常*

*QuorumJournalManager.LOG.warn(*

*"Remote journal " + IPCLoggerChannel.this + " failed to " +*

*"write txns " + firstTxnId + "-" + (firstTxnId + numTxns - 1) +*

*". Will try to write to this JN again after the next " +*

*"log roll.", e);*

*synchronized (IPCLoggerChannel.this) {*

*outOfSync = true;*

*}*

*throw e;*

*} ......*

*}*

抛出的异常会被捕捉到，判断是否大多数都成功，否则抛出异常由mapJournalsAnd

ReportErrors处理。QuorumOutputStream#flushAndSync中的核心代码：

*QuorumCall<AsyncLogger, Void> qcall = loggers.sendEdits(*

*segmentTxId, firstTxToFlush,*

*numReadyTxns, data);*

*loggers.waitForWriteQuorum(qcall, writeTimeoutMs, "sendEdits");*

*//该出等待调用输出，并抛出异常， 2N+1 JN 集群，当<N时异常*

*AsyncLoggerSet#waitForWriteQuorum(q: QuorumCall<AsyncLogger,V>)*

*<V> Map<AsyncLogger, V> waitForWriteQuorum(QuorumCall<AsyncLogger, V> q,*

*int timeoutMs, String operationName) throws IOException {*

*int majority = getMajoritySize(); //* *loggers.size() / 2 + 1，成功的阈值*

*try {*

*q.waitFor(*

*loggers.size(), // either all respond ，所有的JN*

*majority, // or we get a majority successes*

*majority, // or we get a majority failures,*

*timeoutMs, operationName);*

*} catch (Exception) //抛出异常*

*if (q.countSuccesses() < majority) {*

*//写入成功的节点数小于大多数阈值，则表示写入失败*

*q.rethrowException("Got too many exceptions to achieve quorum size " +*

*getMajorityString());*

*}*

*return q.getResults(); }*

QuorumCall#waitFor，等待调用结果，并判断是否成功，其中timeoutMS配置参数为：

*dfs.qjournal.write-txns.timeout.ms*

*public synchronized void waitFor(*

*int minResponses, int minSuccesses, int maxExceptions,*

*int millis, String operationName)*

*throws InterruptedException, TimeoutException {*

*while (true) {*

*//下面是调用成功节点数超过majority，则成功返回*

*if (minResponses > 0 && countResponses() >= minResponses) return;*

*if (minSuccesses > 0 && countSuccesses() >= minSuccesses) return;*

*if (maxExceptions >= 0 && countExceptions() > maxExceptions) return;*

*long now = timer.monotonicNow();*

*if (now > nextLogTime) {*

*nextLogTime = now + WAIT\_PROGRESS\_INTERVAL\_MILLIS;*

*}*

*long rem = et - now;*

*if (rem <= 0) {*

*// Increase timeout if a full GC occurred after restarting stopWatch*

*long timeoutIncrease = getQuorumTimeoutIncreaseMillis(0, millis);*

*if (timeoutIncrease > 0) {*

*et += timeoutIncrease;*

*} else {*

*throw new TimeoutException(); //等待响应，超时*

*}*

*}*

*restartQuorumStopWatch();*

*rem = Math.min(rem, nextLogTime - now);*

*rem = Math.max(rem, 1);*

*wait(rem);*

*// Increase timeout if a full GC occurred after restarting stopWatch*

*long timeoutIncrease = getQuorumTimeoutIncreaseMillis(-rem, millis);*

*if (timeoutIncrease > 0) {*

*et += timeoutIncrease;*

*}*

*}*

*}*

Active NN提交EditLog到JN集群的过程实际上是同步阻塞的（见waitForWriteQuorum），但是不需要所有的JN都调用成功，只要大多数JN调用成功就可以了。如果无法形成大多数（majority），那么认为提交EditLog失败，NN停止服务退出进程。

对应到分布式系统的CAP原理，虽然采用了Paxos的大多数思想对C(consistency，一致性)和A(availability，可用性)进行折中，但是还是可以认为NN选择一致性而放弃A，符合NN对数据一致性的要求。

## 2.2 Standby NN从JN集群中同步EditLog

当NN进入Standby状态之后，会启动一个EditLogTailer线程，这个线程会定期调用EditLogTailer的doTailEdits方法从JN集群中同步EditLog，然后把同步的EdtLog回放到内存之中的文件系统进行之上（并不会同时把EditLog写入到本地磁盘上）。

*void startStandbyServices(final Configuration conf) throws IOException {*

*......*

*editLogTailer = new EditLogTailer(this, conf);*

*editLogTailer.start();*

*......*

*}*

EditLogTailer类成员变量如下：



EditLogTailer周期性的调用doWork，获取EditLog，如下：

*private void doWork() {*

*while (shouldRun) {*

*try {*

*//Standby没有更多的transactions，则触发Active NN的rollEditLog操作*

*if (tooLongSinceLastLoad() &&lastRollTriggerTxId < lastLoadedTxnId) {*

*triggerActiveLogRoll();*

*}....*

*try {*

*doTailEdits();*

*} .....*

*}*

核心方法doTailEdits

*void doTailEdits() throws IOException, InterruptedException {*

*try {*

*FSImage image = namesystem.getFSImage();*

*long lastTxnId = image.getLastAppliedTxId();*

*Collection<EditLogInputStream> streams;*

*try {*

*streams = editLog.selectInputStreams(lastTxnId + 1, 0,*

*null, inProgressOk, true);*

*} ....*

*long editsLoaded = 0;*

*try {*

*editsLoaded = image.loadEdits(streams, namesystem);*

*} ...}*

采用EdiltLogInputStream，根据url从JournalNodeHttpServer中获取Edit Log。

# 数据恢复机制分析

处于Standby状态的NN转换为Active状态时，JN集群中各个JN上的EditLog就可能会处于不一致状态，所以首先要做的事情就是让JN集群各个节点上的EditLog恢复一致。然后Standby NN从JN集群上补齐落后的Edit Log，至此就可以安全的对外提供服务。

*void startActiveServices() throws IOException {*

*startingActiveService = true;*

*try {*

*FSEditLog editLog = getFSImage().getEditLog();*

*if (!editLog.isOpenForWrite()) {*

*.....*

*editLog.recoverUnclosedStreams();*

*...*

*long nextTxId = getFSImage().getLastAppliedTxId() + 1;*

*LOG.info("Will take over writing edit logs at txnid " +*

*nextTxId);*

*editLog.setNextTxId(nextTxId);*

*....}*

JN集群上EditLog达成一致的过程就是一致性算法Paxos的典型应用场景，可以看做是Single Instance Paxos算法的一个实现，在达成一致的过程中，Active NN和JN集群之间的交互流程如下面两个图所示：



阶段1：Active NN从JN中获取相关信息

在Active NN中使用Epoch来标识每一次生命周期，每发生一次NN的主备切换，Epoch就会加1，这是一种fencing机制，防止脑裂的情况。

**步骤1：从JN中获取当前Epoch(getJournalState)**

Active NN首先向JN集群发送getJouralState RPC访问，JN会返回自己保存的最近的Epoch(lastPromisedEpoch)



JN通过JournalNodeRpcServer#getJournalState返回结果。

*public GetJournalStateResponseProto getJournalState(String journalId,*

*String nameServiceId) throws IOException {*

*long epoch = jn.getOrCreateJournal(journalId,*

*nameServiceId).getLastPromisedEpoch();*

*return GetJournalStateResponseProto.newBuilder()*

*.setLastPromisedEpoch(epoch)*

*.setHttpPort(jn.getBoundHttpAddress().getPort())*

*.setFromURL(jn.getHttpServerURI()).build();*

*}*

响应中主要包括lastPromisedEpoch和jn的URI。

**2）New Epoch**

NN收到大多数的JN返回的Epoch后，在其中选择最大的一个加1作为当前新的Epoch，然后向各个JN发送newEpoch RPC请求。

详细过程见QuorumJournalManager#createNewUniqueEpoch

*Map<AsyncLogger, NewEpochResponseProto> createNewUniqueEpoch()*

*throws IOException {*

*Map<AsyncLogger, GetJournalStateResponseProto> lastPromises =*

*loggers.waitForWriteQuorum(loggers.getJournalState(),*

*getJournalStateTimeoutMs, "getJournalState()");*

*long maxPromised = Long.MIN\_VALUE;*

*for (GetJournalStateResponseProto resp : lastPromises.values()) {*

*maxPromised = Math.max(maxPromised, resp.getLastPromisedEpoch());*

*}*

*assert maxPromised >= 0;*

*long myEpoch = maxPromised + 1;*

*Map<AsyncLogger, NewEpochResponseProto> resps =*

*loggers.waitForWriteQuorum(loggers.newEpoch(nsInfo, myEpoch),*

*newEpochTimeoutMs, "newEpoch(" + myEpoch + ")");*

*loggers.setEpoch(myEpoch);*

*return resps;*

*}*

*public ListenableFuture<NewEpochResponseProto> newEpoch(*

*final long epoch) {*

*return singleThreadExecutor.submit(new Callable<NewEpochResponseProto>() {*

*@Override*

*public NewEpochResponseProto call() throws IOException {*

*return getProxy().newEpoch(journalId, nameServiceId, nsInfo, epoch);*

*}*

*});*

*}*

**3) JN返回startTxId**

JN在收到信的Epoch后，会检查这个新的Epoch是否比它本地保存的lastPromisedEpoch大，如果大的话就把lastPromisedEpoch更新为新的Epoch，并且向NN返回它自己的本地磁盘上的最新的EditLogSegment的起始事务id，为后面的数据恢复过程做好准备，如果小于或等于的话就向NN返回错误。

*synchronized NewEpochResponseProto newEpoch(*

*NamespaceInfo nsInfo, long epoch) throws IOException {*

*.....*

*if (epoch <= getLastPromisedEpoch()) {*

*throw new IOException("Proposed epoch " + epoch + " <= last promise " +*

*getLastPromisedEpoch());*

*}*

*....*

*updateLastPromisedEpoch(epoch);*

*abortCurSegment();*

*NewEpochResponseProto.Builder builder =*

*NewEpochResponseProto.newBuilder();*

*EditLogFile latestFile = scanStorageForLatestEdits();*

*if (latestFile != null) {*

*builder.setLastSegmentTxId(latestFile.getFirstTxId());*

*}*

*return builder.build();*

*}*

NN收到大多数JN对newEpoch的成功响应之后，就会认为生成新的Epoch成功。在生成新的Epoch之后，每次NN在向JN集群提交EditLog的时候，都会把这个Epoch作为参数传递过去，如果传过来的epoch值比它自己保存的lastPromisedEpoch小的话，那么这次写操作会被拒绝。一旦大多数JN拒绝这次操作，那么这次写操作就失败。如果原来的Active NN恢复正常之后再向JN写EditLog，那么因为它的Epoch肯定比新生成的Epoch小，并且大多数的JN都接受了新生成的Epoch，所以拒绝写入的JN数目至少是大多数，这样原来的Active NN写EditLog就肯定失败，失败之后这个NN进程就会直接退出，实现了对原来的Active NN隔离。

4）选择要恢复的EditLog Segment的id

JN在处理完newEpoch RPC之后，会向NN返回它自己的本地磁盘上最新的一个EditLog Segment的起始事务id，这个事务id实际上也作为这个EditLog Segment的id。NN会在这些id之中选择一个最大的id作为要进行数据恢复的EditLog Segment的id

*long mostRecentSegmentTxId = Long.MIN\_VALUE;*

*for (NewEpochResponseProto r : resps.values()) {*

*if (r.hasLastSegmentTxId()) {*

*mostRecentSegmentTxId = Math.max(mostRecentSegmentTxId,*

*r.getLastSegmentTxId());*

*}*

*}*

至此Active NN已经从JN中获取了要达到分布式一致性的mostRecentSegmentTxId，JN根据这个txId达到一致。如下图所示：



阶段2：JN达到分布式一致

QuorumJournalManager#recoverUnclosedSegment的执行如下：

*private void recoverUnclosedSegment(long segmentTxId) throws IOException {*

*QuorumCall<AsyncLogger,PrepareRecoveryResponseProto> prepare =*

*loggers.prepareRecovery(segmentTxId);*

*Map<AsyncLogger, PrepareRecoveryResponseProto> prepareResponses=*

*loggers.waitForWriteQuorum(prepare, prepareRecoveryTimeoutMs,*

*"prepareRecovery(" + segmentTxId + ")");*

*.....*

*//根据prepareRecovery返回结果，选择同步的基准数据源，选择选择Epoch比较高或者事务数更多的Segment*

*Entry<AsyncLogger, PrepareRecoveryResponseProto> bestEntry = Collections.max(*

*prepareResponses.entrySet(), SegmentRecoveryComparator.INSTANCE);*

*AsyncLogger bestLogger = bestEntry.getKey();*

*PrepareRecoveryResponseProto bestResponse = bestEntry.getValue();*

*......*

*SegmentStateProto logToSync = bestResponse.getSegmentState();*

*assert segmentTxId == logToSync.getStartTxId();*

*//获取要基准数据源的url*

*URL syncFromUrl = bestLogger.buildURLToFetchLogs(segmentTxId);*

*//各JN 调用recovery流程*

*QuorumCall<AsyncLogger,Void> accept = loggers.acceptRecovery(logToSync, syncFromUrl);*

*loggers.waitForWriteQuorum(accept, acceptRecoveryTimeoutMs,*

*"acceptRecovery(" + TextFormat.shortDebugString(logToSync) + ")");*

*//等待同步完成*

*QuorumCall<AsyncLogger, Void> finalize =*

*loggers.finalizeLogSegment(logToSync.getStartTxId(), logToSync.getEndTxId());*

*loggers.waitForWriteQuorum(finalize, finalizeSegmentTimeoutMs,*

*String.format("finalizeLogSegment(%s-%s)",*

*logToSync.getStartTxId(),*

*logToSync.getEndTxId()));*

*}*

5）prepareRecovery RPC

NN接下来向JN集群发送prepareRecovery RPC请求，请求的参数就是选出的EditLog Segment的id。JN收到请求后返回本地磁盘上这个Segment的起始事务id、结束事务id(in-progress或finalized)。

并且这里产生新的Epoch作为Paxos算法的提案编号（proposal Number），只要大多数的JN的prepareRecovery RPC调用成功返回，NN就认为成功。调用如下：

*public synchronized PrepareRecoveryResponseProto prepareRecovery(*

*RequestInfo reqInfo, long segmentTxId) throws IOException {*

*......*

*PrepareRecoveryResponseProto.Builder builder =*

*PrepareRecoveryResponseProto.newBuilder();*

*PersistedRecoveryPaxosData previouslyAccepted = getPersistedPaxosData(segmentTxId);*

*completeHalfDoneAcceptRecovery(previouslyAccepted);*

*SegmentStateProto segInfo = getSegmentInfo(segmentTxId);*

*if (previouslyAccepted != null && !hasFinalizedSegment) {*

*SegmentStateProto acceptedState = previouslyAccepted.getSegmentState();*

*builder.setAcceptedInEpoch(previouslyAccepted.getAcceptedInEpoch())*

*.setSegmentState(previouslyAccepted.getSegmentState());*

*} else {*

*if (segInfo != null) {*

*builder.setSegmentState(segInfo);*

*}*

*}*

*builder.setLastWriterEpoch(lastWriterEpoch.get());*

*if (committedTxnId.get() != HdfsServerConstants.INVALID\_TXID) {*

*builder.setLastCommittedTxId(committedTxnId.get());*

*}*

*PrepareRecoveryResponseProto resp = builder.build();*

*return resp;*

*}*

根据prepareRecovery的响应后，选择进行同步的基准数据源

*Entry<AsyncLogger, PrepareRecoveryResponseProto> bestEntry = Collections.max(*

*prepareResponses.entrySet(), SegmentRecoveryComparator.INSTANCE);*

*AsyncLogger bestLogger = bestEntry.getKey();*

*PrepareRecoveryResponseProto bestResponse = bestEntry.getValue();*

*......*

*SegmentStateProto logToSync = bestResponse.getSegmentState();*

*assert segmentTxId == logToSync.getStartTxId();*

*//获取要基准数据源的url*

*URL syncFromUrl = bestLogger.buildURLToFetchLogs(segmentTxId);*

6）acceptRecovery RPC

向JN集群发送acceptRecovery RPC请求

*QuorumCall<AsyncLogger,Void> accept = loggers.acceptRecovery(logToSync, syncFromUrl);*

*loggers.waitForWriteQuorum(accept, acceptRecoveryTimeoutMs,*

*"acceptRecovery(" + TextFormat.shortDebugString(logToSync) + ")");*

将选定的基准数据源作为参数，JN接收到acceptRecovery RPC请求后，从基准数据源JN的JournalNodeHttpServer上下载EditLog Segment，将本地的EditLog Segment替换为下载的EditLog Segment，如下所示：

*public void acceptRecovery(RequestInfo reqInfo, SegmentStateProto log,*

*URL fromUrl) throws IOException {*

*jn.getOrCreateJournal(reqInfo.getJournalId(), reqInfo.getNameServiceId())*

*.acceptRecovery(reqInfo, log, fromUrl);*

*}*

Journal执行的同步工作，如下所示：

*public synchronized void acceptRecovery(RequestInfo reqInfo,*

*SegmentStateProto segment, URL fromUrl) throws IOException {*

*long segmentTxId = segment.getStartTxId();*

*PersistedRecoveryPaxosData oldData = getPersistedPaxosData(segmentTxId);*

*PersistedRecoveryPaxosData newData = PersistedRecoveryPaxosData.newBuilder()*

*.setAcceptedInEpoch(reqInfo.getEpoch())*

*.setSegmentState(segment)*

*.build();*

*.....*

*File syncedFile = null;*

*SegmentStateProto currentSegment = getSegmentInfo(segmentTxId);*

*if (currentSegment == null ||*

*currentSegment.getEndTxId() != segment.getEndTxId()) {*

*updateHighestWrittenTxId(Math.max(segment.getEndTxId(), highestWrittenTxId));*

*} else {*

*syncedFile = syncLog(reqInfo, segment, fromUrl);*

*}*

*.....*

*}*

7）finalizeLogSegment RPC

NN确认大多数的JN的Editlog Segment已经从基准数据源进行了同步，接下来NN向JN集群发送finalizeLogSegment RPC请求，JN接收到请求后，将对应的EditLog Segment 从in-progress状态转换为finalized状态，实际上就是将文件名从edit\_inprogress\_{startTxid}重命名为edits\_{startTxid}-${endTxid}

*public synchronized void finalizeLogSegment(RequestInfo reqInfo, long startTxId,*

*long endTxId) throws IOException {*

*....*

*FileJournalManager.EditLogFile elf = fjm.getLogFile(startTxId);*

*...*

*if (elf.isInProgress()) {*

*if (needsValidation) {*

*LOG.info("Validating log segment " + elf.getFile() + " about to be " +*

*"finalized");*

*elf.scanLog(Long.MAX\_VALUE, false);*

*checkSync(elf.getLastTxId() == endTxId,*

*"Trying to finalize in-progress log segment %s to end at " +*

*"txid %s but log %s on disk only contains up to txid %s",*

*startTxId, endTxId, elf.getFile(), elf.getLastTxId());*

*}*

*fjm.finalizeLogSegment(startTxId, endTxId);*

*} ...*

*}*

*synchronized public void finalizeLogSegment(long firstTxId, long lastTxId)*

*throws IOException {*

*File inprogressFile = NNStorage.getInProgressEditsFile(sd, firstTxId);*

*File dstFile = NNStorage.getFinalizedEditsFile(*

*sd, firstTxId, lastTxId);*

*NativeIO.renameTo(inprogressFile, dstFile);*

*....*

*}*

只要大多数JN的finalizeLogSegment RPC调用成功返回，则NN就认为成功，此时可以保证JN集群大多数JN节点已经处于一致的状态，这样NN才能安全地从JN集群上补齐落后的EditLog数据。

http://www.cnblogs.com/f1194361820/p/6856292.html

https://www.edureka.co/blog/how-to-set-up-hadoop-cluster-with-hdfs-high-availability/

要分析的问题：

1. Epoch计数原理
2. Potential GetGroup可能触发的问题
3. Paxos算法分析 ，估计与2 Epoch相关

目前的问题：

