索引创建流程

RestCreateIndexAction
--- 处理索引创建rest请求，创建CreateIndexRequest

TransportCreateIndexAction
--- 处理CreateIndexRequest请求

将请求转发给Master执行

MetaDataCreateIndexService::createIndex
--- 处理CreateIndexClusterStateUpdateRequest

ClusterService::submitStateUpdateTask
--- 提交IndexCreationTask执行(更新集群状态,创建分片)

IndexCreationTask::execute
--- 创建索引MetaData、更新ClusterState

创建对应的IndexService（验证索引各项设置是否正确）

创建新的IndexMetaData(mapping、alias、setting、pTerm...)

使用IndexMetaData创建新ClusterState

使用IndexMetaData创建新RoutingTable

AllocationService::reroute,为索引分配分片

ClusterService, 发布新的集群状态

Allocation: 分片分配

AllocationService::reroute
--- 分配分片

AllocationService::reroute

获取所有未分配分片

GatewayAllocator::allocateUnassigned

PrimaryShardAllocator

ReplicaShardAllocator

设置相关分片为状态为Initializing

只负责已存在分片的分配（一般用于集群启动）

```
private static boolean isResponsibleFor(final ShardRouting shard) {
    return shard.primary() // must be primary
            && shard.unassigned() // must be unassigned
            // only handle either an existing store or a snapshot recovery
            && (shard.recoverySource().getType() == RecoverySource.Type.EXISTING_STORE
                || shard.recoverySource().getType() == RecoverySource.Type.SNAPSHOT);
}
```

BalancedShardsAllocator::allocate

allocateUnassigned

为分片寻找最佳节点

moveShards

节点间迁移分片（分片无法继续存放在当前节点）

balance

节点分片再平衡

根据分片分配或迁移结果，构造新集群状态

Allocation: 处理未分配分片

Balancer::allocateUnassigned

判断当前分片是否可分配

```
Decision shardLevelDecision = allocation.deciders().canAllocate(shard, allocation);
```

遍历所有节点，计算分片节点权重

寻找出拥有最低权重的节点

判断分片能否分配给指定节点

```
Decision currentDecision = allocation.deciders().
    canAllocate(shard, node.getRoutingNode(), allocation);
```

更新分片状态为Initializing

计算分片级权重wShard：节点分片数+1-集群节点平均分片数

倾向于寻找拥有最少分片的节点（包含所有索引的分片）

计算索引级权重wIndex：节点索引分片数+1-集群节点索引平均分片数

倾向于寻找拥有最少索引分片的节点（只包含特定索引的分片）

计算节点权重：
(shardBalance*wShard + indexBalance*wIndex) / (shardBalance + indexBalance)

cluster.routing.allocation.balance.shard
cluster.routing.allocation.balance.index
使用权重因子来混合分片和索引级权重

总目标为寻找拥有最少分片的节点

两节点权重相同时

shard:1[r]

shardId=0

node:1
shard: 0[p]

shardIdMax1=0

node:2
shard: 2[p]

shardIdMax2=2

```
[(0,P,IDX1), (0,P,IDX2), (0,R,IDX1), (0,R,IDX1), (0,R,IDX2), (0,R,IDX2)]
round1 [(0,P,IDX1), (0,P,IDX2), (0,R,IDX1),              (0,R,IDX2)
round2                                  (0,R,IDX1),              (0,R,IDX2)]
```

```
||
  &&
    ||
      shardIdMax2 > shardId && shardIdMax1 > shardId
      shardIdMax2 < shardId && shardIdMax1 < shardId
    shardIdMax2 < shardIdMax1
  shardIdMax2 > shardIdMax1 && shardIdMax2 > shardId  && shardIdMax1 < shardId
```

Allocation: 重分配已启动的分片

Balancer::moveShards

遍历所有分片

判断分片是否需要迁移

获取sourceNode与targetNode

sourceNode::removeShard
targetNode::addShard

标记分片为relocating

分片需要处于STARTED状态

deciders表决该分片是否可以继续留在此节点上

```
Decision canRemain = allocation.deciders().canRemain(shardRouting, routingNode, allocation);
if (canRemain.type() != Decision.Type.NO) {
    return MoveDecision.stay(canRemain);
```

遍历除当前节点的所有节点（权重从小到大）

找出可分配该分片的节点，作为目标节点

```
Decision allocationDecision = allocation.deciders().canAllocate(shardRouting, target, allocation);
```

Allocation: 再平衡节点分片分配

目前没有pendingfetch任务、deciders表决可进行rebalance、节点数>1

cluster.routing.allocation.balance.threshold

按索引不平衡度降序排序，遍历索引

一个索引的不平衡度指使用此索引计算节点权重，最大节点权重与最小节点权重的差值

```
for (String index : buildWeightOrderedIndices()) {
    IndexMetaData indexMetaData = metaData.index(index);
```

获取索引相关节点（所在节点与可分配节点）

相关节点都被移到列表头部，用lowIdx与highIdx标记

```
int lowIdx = 0;
int highIdx = relevantNodes - 1;
```

重新计算相关节点权重（包含索引权重）

循环，直到尝试完所有最大权重差值节点对或者节点对差值<threshold

```
if (tryRelocateShard(minNode, maxNode, index, delta)) {
    weights[lowIdx] = sorter.weight(modelNodes[lowIdx]);
    weights[highIdx] = sorter.weight(modelNodes[highIdx]);
    sorter.sort(0, relevantNodes);
    lowIdx = 0;
    highIdx = relevantNodes - 1;
    continue;
}
```

尝试将分片从权重最大节点分配至权重最小节点

若权重差异减小，则执行此分配

节点按新权重排序

1、lowIdx++，直到末尾
2、lowIdx置0，highIdx--
3、lowIdx==highIdx==0
break

```
if (lowIdx < highIdx - 1) {
    lowIdx++;
} else if (lowIdx > 0) {
    lowIdx = 0;
    highIdx--;
} else {
    break;
}
```

若权重差异未减小，更新lowIdx与highIdx，更新节点对，继续尝试

Recovery: 分片恢复

IndicesClusterStateService::applyClusterState
--- 处理集群状态改变，做出索引级响应

获取最新集群状态

updateFailedShardsCache

删除被删除的索引分片

删除被删除索引的记录

更新索引MetaData

创建新索引

创建或更新分片

EMPTY_STORE, EXISTING_STORE(主分片本地恢复)

LOCAL_SHARDS（从当前节点其他分片恢复）

SNAPSHOT（从快照恢复）

PEER（副本从远程分片恢复）

根据recoveryType恢复分片

IndexShard::startRecovery

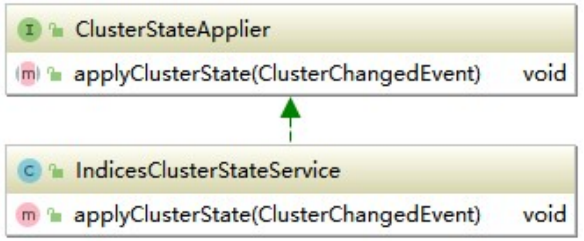
IndicesService::createShard

若分片不存在，createShard

创建一个LuceneDirectory

若分片存在，updateShard

分片恢复完毕后，集群状态更新，会调用此方法将分片状态更改为STARTED



Recovery: 分片恢复EMPTY_STORE

IndexShard::startRecovery::recoverFromStore
--- 分片恢复

标记分片状态为RECOVERING

StoreRecovery::recoverFromStore

INIT（默认状态，校验分片状态，判断是否为主分片）

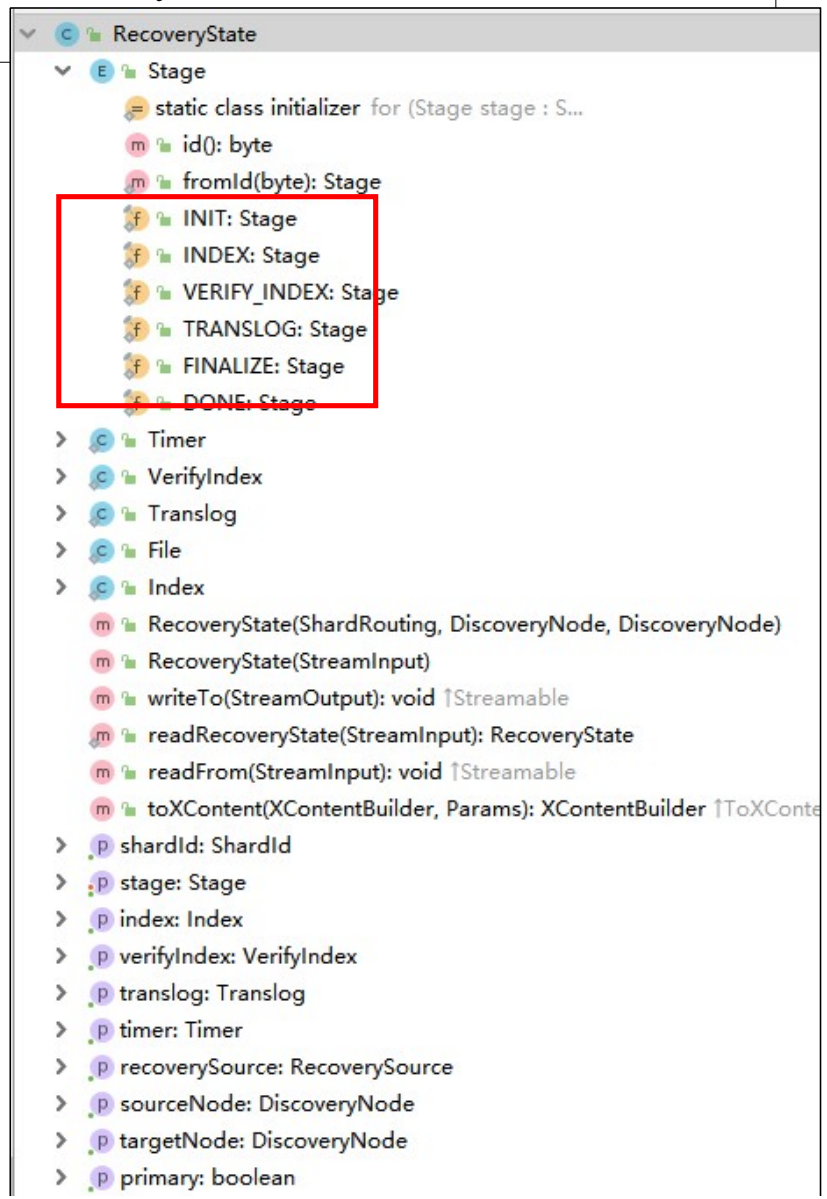
INDEX（读取SegmentInfo，获取version，更新索引version）

创建空索引

```
metadataLock.writeLock().lock();
try (IndexWriter writer = newEmptyIndexWriter(directory, luceneVersion)) {
    final Map<String, String> map = new HashMap<>();
    map.put(Engine.HISTORY_UUID_KEY, UUIDs.randomBase64UUID());
    map.put(SequenceNumbers.LOCAL_CHECKPOINT_KEY, Long.toString(SequenceNumbers.NO_OPS_PERFORMED));
    map.put(SequenceNumbers.MAX_SEQ_NO, Long.toString(SequenceNumbers.NO_OPS_PERFORMED));
    map.put(Engine.MAX_UNSAFE_AUTO_ID_TIMESTAMP_COMMIT_ID, "-1");
    updateCommitData(writer, map);
} finally {
    metadataLock.writeLock().unlock();
```

创建Translog,uuid写入commitData

```
final String translogUUID = Translog.createEmptyTranslog(
    indexShard.shardPath().resolveTranslog(), SequenceNumb
    indexShard.getPendingPrimaryTerm());
store.associateIndexWithNewTranslog(translogUUID);
writeEmptyRetentionLeasesFile(indexShard);
```



IndexShard::openEngineAndRecoverFromTranslog

IndexShard::innerOpenEngineAndTranslog

VERIFY_INDEX
（index.shard.check_on_startup）

检查索引分片是否损坏

TRANSLOG

从SegmentInfo中读出translogUUID

InternalEngine::recoverFromTranslog

从SegmentInfo中读出最后一次提交的Translog generation

从最后一次提交的generation开始生成Translog Snapshot

应用Operations

Lucene writer commit

refresh

IndexShard::openEngineAndRecoverFromTranslog
--- 索引校验，translog恢复

```
while ((operation = snapshot.next()) != null) {
    try {
        logger.trace( message: "[translog] recover op {}", operation);
        Engine.Result result = applyTranslogOperation(engine, operation, origin);
        switch (result.getResultType()) {
            case FAILURE:
                throw result.getFailure();
            case MAPPING_UPDATE_REQUIRED:
                throw new IllegalArgumentException("unexpected mapping update: " +
            case SUCCESS:
                break;
            default:
                throw new AssertionError( detailMessage: "Unknown result type [" +
        }

        opsRecovered++;
        onOperationRecovered.run();
    } catch (Exception e) {
```

```
switch (operation.opType()) {
    case INDEX:
        final Translog.Index index = (Translog.Index) operation;
        // we set canHaveDuplicates to true all the time such that we de-optimize the translog case and ensure that all
        // autoGeneratedID docs that are coming from the primary are updated correctly.
        result = applyIndexOperation(engine, index.seqNo(), index.primaryTerm(), index.version(),
            versionType, UNASSIGNED_SEQ_NO, ifPrimaryTerm: 0, index.getAutoGeneratedIdTimestamp(), isRetry: true, origin,
            new SourceToParse(shardId.getIndexName(), index.type(), index.id(), index.source(),
                XContentHelper.xContentType(index.source()), index.routing()));
        break;
```

文档解析

准备Index

执行Index

IndexShard::finalizeRecovery

FINALIZE

refresh

```
public void finalizeRecovery() {
    recoveryState().setStage(RecoveryState.Stage.FINALIZE);
    Engine engine = getEngine();
    engine.refresh( source: "recovery_finalization");
    engine.config().setEnableGcDeletes(true);
}
```

IndexShard::postRecovery

DONE

refresh

更改分片状态为POST_RECOVERY

RecoveryListener::onRecoveryDone

ShardStateAction::shardStarted

send cluster/shard/started rpc to master

IndexShard::finalizeRecovery
--- 更新恢复状态，refresh

IndexShard::postRecovery
--- 更新恢复状态，refresh

ShardStateAction::shardStarted
--- 向master节点发送分片启动请求

Recovery: 分片恢复PEER

IndexShard::startRecovery
--- 副本分片从主分片恢复

PeerRecoveryTargetService::startRecovery
--- 开始副本恢复，定义恢复阶段的RPC handler

标记分片状态为RECOVERING

PeerRecoveryTargetService::startRecovery

INIT（默认状态，校验分片状态，判断是否为主分片）

PeerRecoveryTargetService::doRecovery

INDEX（将主分片Lucene数据复制到副本分片）

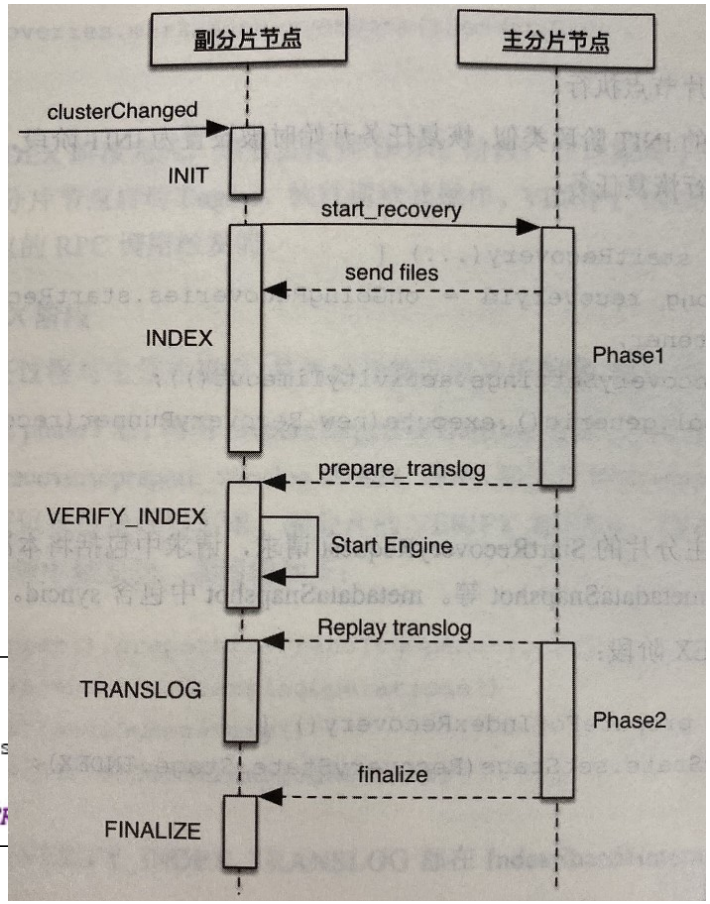
```
transportService.submitRequest(request.sourceNode(), PeerRecoverySourceService.Actions.START_RECOVERY, request,
    new TransportResponseHandler<RecoveryResponse>() {
```

主分片发送Lucene数据与Translog

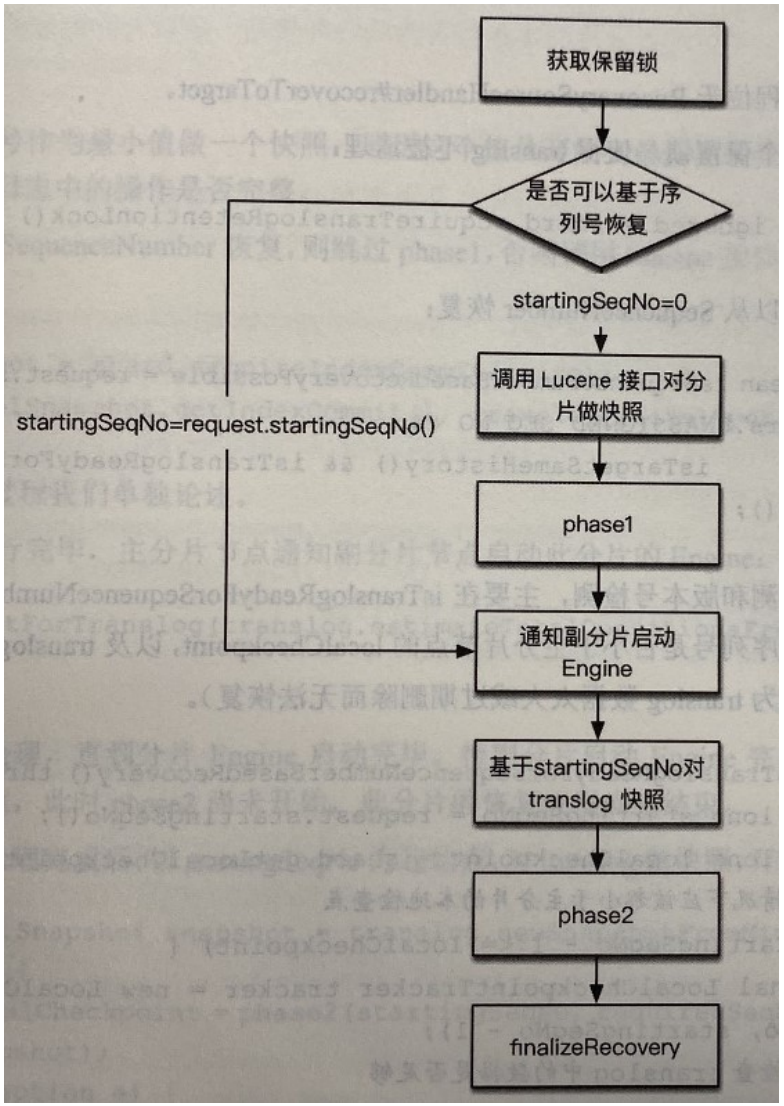
主分片通过RPC控制VERIFY_INDEX、TRANSLOG、FINALIZE阶段

副本分片回放Translog

```
transportService.registerRequestHandler(Actions.PREPARE_TRANSLOG, ThreadPool.Names.GENERIC,
        RecoveryPrepareForTranslogOperationsRequest::new, new PrepareForTranslogOperationsRequestHandler());
transportService.registerRequestHandler(Actions.TRANSLOG_OPS, RecoveryTranslogOperationsRequest::new, ThreadPool.Names
        new TranslogOperationsRequestHandler());
transportService.registerRequestHandler(Actions.FINALIZE, RecoveryFinalizeRecoveryRequest::new, ThreadPool.Names.GENE
```



分片恢复PEER:主分片流程



PeerRecoverySourceService
---StartRecoveryTransportRequestHandler
--- 开始主分片控制副本恢复流程

RecoverySourceHandler::recoverToTarget
--- 副本分片从主分片恢复

若可以基于seqNum恢复（seqNo小于主分片localCheckPoint且保留的Translog足够用于数据恢复）

发送Translog

基于Translog恢复

若无法基于seqNum恢复

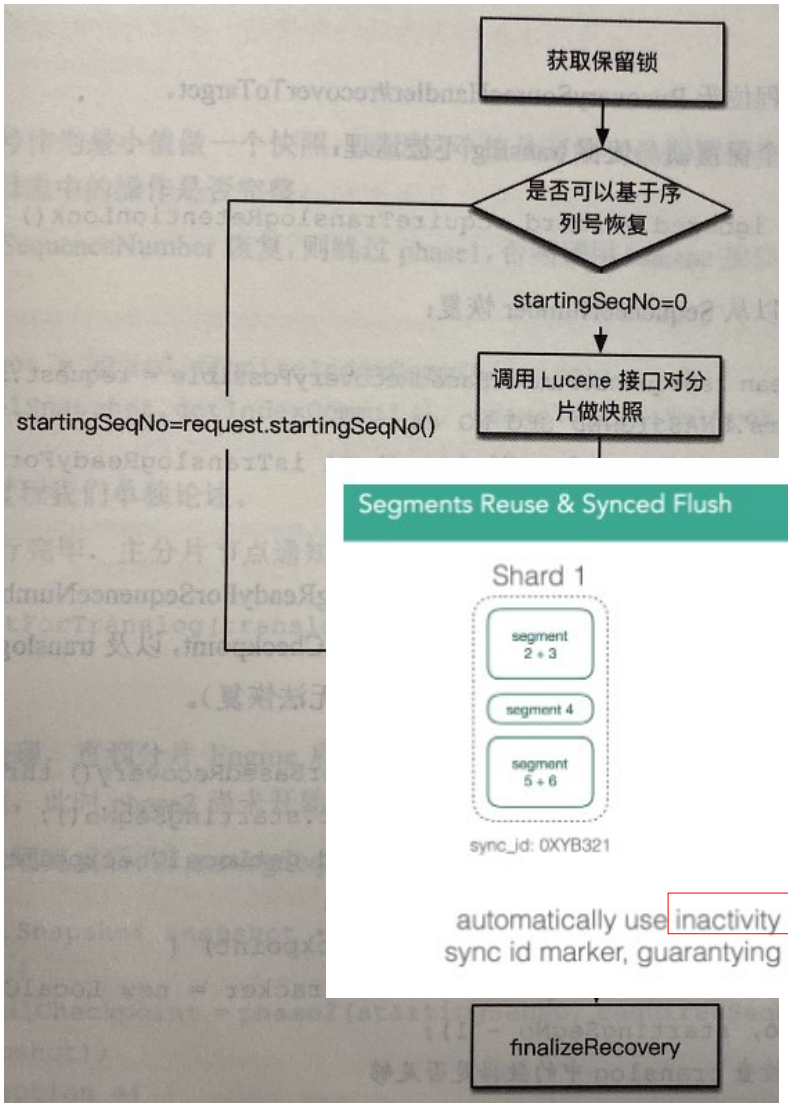
syncId比对（syncFlush）

发送Lucene快照

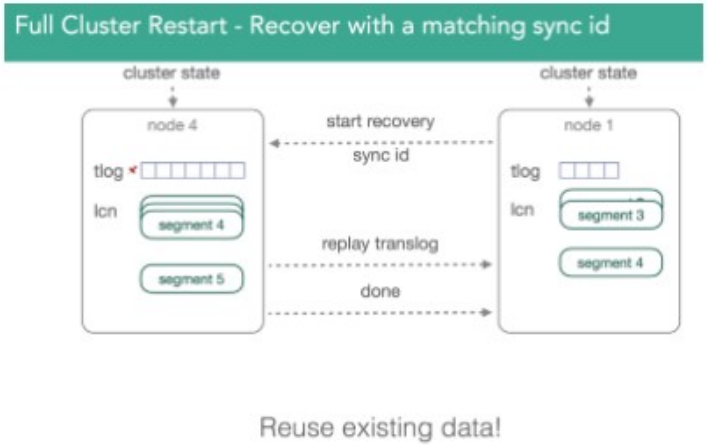
分片恢复PEER:主分片流程

PeerRecoverySourceService
---StartRecoveryTransportRequestHandler
--- 开始主分片控制副本恢复流程

RecoverySourceHandler::recoverToTarget
--- 副本分片从主分片恢复



若可以基于seqNum恢复（seqNo小于主分片localCheckPoint且保留的Translog足够用于数据恢复）



分片恢复PEER:Lucene快照

Lucene快照包含最后一次提交点的信息，以及全部Segment文件，是对已刷盘数据的完整快照。

```
private static final class CommitPoint extends IndexCommit {
    private String segmentsFileName;
    private final Collection<String> files;
    private final Directory dir;
    private final long generation;
    private final Map<String, String> userData;
    private final int segmentCount;
```

```
public IndexCommitRef acquireLastIndexCommit(final boolean flushFirst) throws EngineException {
    // we have to flush outside of the readlock otherwise we might have a problem upgrading
    // the to a write lock when we fail the engine in this operation
    if (flushFirst) {
        logger.trace("start flush for snapshot");
        flush( force: false,  waitIfOngoing: true);
        logger.trace("finish flush for snapshot");
    }
    final IndexCommit lastCommit = combinedDeletionPolicy.acquireIndexCommit( acquiri
    return new Engine.IndexCommitRef(lastCommit,  () -> releaseIndexCommit(lastCommit
}
```

```
org.apache.lucene.index 1 usage
  SnapshotDeletionPolicy 1 usage
    onCommit(List<? extends IndexCommit>) 1 usage
        73 primary.onCommit(wrapCommits(commits));
```

```
@Override
public synchronized void onCommit(List<? extends IndexCommit> commits) throws IOException {
    final int keptPosition = indexOfKeptCommits(commits, globalCheckpointSupplier.getAsLong());
    lastCommit = commits.get(commits.size() - 1);
    safeCommit = commits.get(keptPosition);
    for (int i = 0; i < keptPosition; i++) {
        if (snapshottedCommits.containsKey(commits.get(i)) == false) {
            deleteCommit(commits.get(i));
        }
    }
    updateRetentionPolicy();
}
```