translog分享

大纲

translog的介绍

translog的底层结构

translog的删除机制

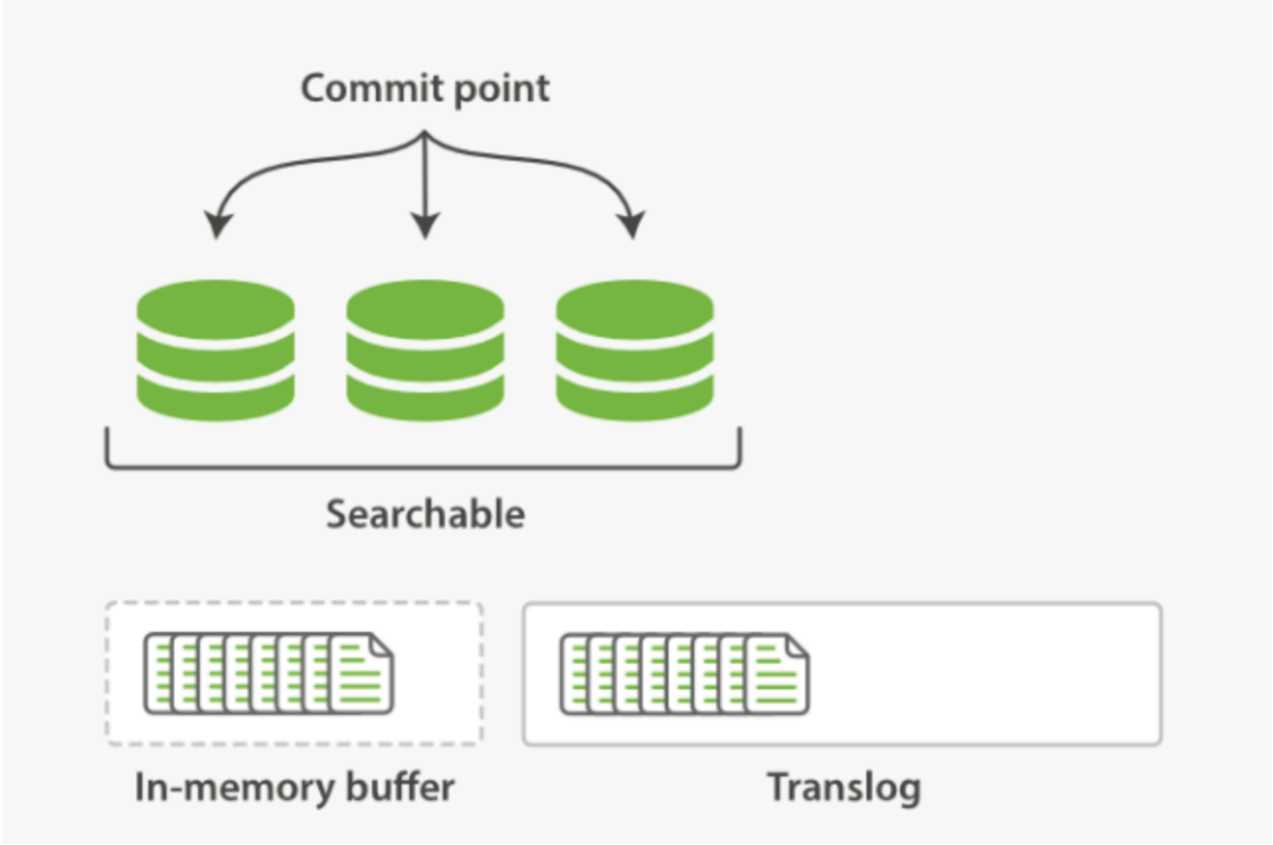
副本shard恢复

简介

- Elasticsearch采取的机制是将数据添加到lucene，lucene内部会维护一个数据缓冲区，此时数据都是不可搜索的。每隔一段时间（默认为1秒），Elasticsearch会执行一次refresh操作：lucene中所有的缓存数据都被写入到一个新的Segment，清空缓存数据。此时数据就可以被搜索。当然，每次执行refresh操作都会生成一个新的Segment文件，这样一来Segment文件有大有小，相当碎片化。Elasticsearch内部会开启一个线程将小的Segment合并（Merge）成大的Segment，减少碎片化，降低文件打开数，提升IO性能。
- 不过这样也带来一个问题。数据写入缓冲区中，没有及时保存到磁盘中，一旦发生程序崩溃或者服务器宕机，数据就会发生丢失。为了保证可靠性，Elasticsearch引入了Translog（事务日志）。每次数据被添加或者删除，都要在Translog中添加一条记录。这样一旦发生崩溃，数据可以从Translog中恢复。

translog简介

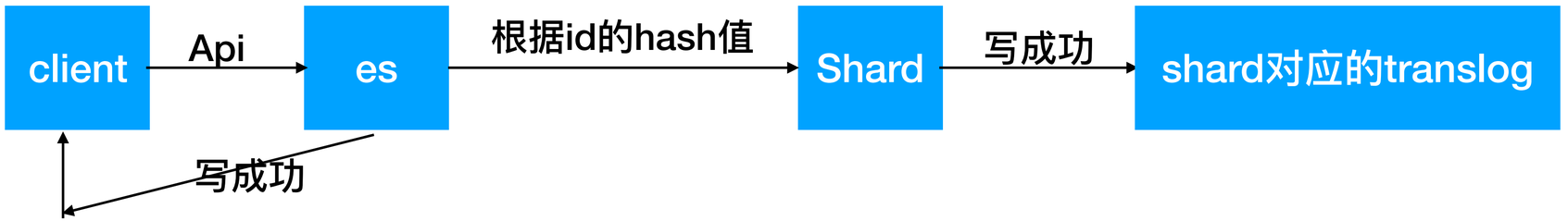
translog与内存buffer关系



通俗来讲当机器宕机，内存中的数据还未刷入磁盘，translog的作用就是恢复这部分数据。

translog简介

写数据流程



这里可以思考下为什么先写shard，后写translog?

translog简介

es中translog配置

- index.translog.flush_threshold_size　　默认到512mb，就会触发flush
- index.translog.durability　　同步或异步，默认同步，就是实时刷盘
- index.translog.sync_interval　　当异步时，刷磁盘的间隔

当我们能容许极端情况下丢失一些数据时，建议将translog改为异步，
这样可以极大的提升效率

translog底层文件

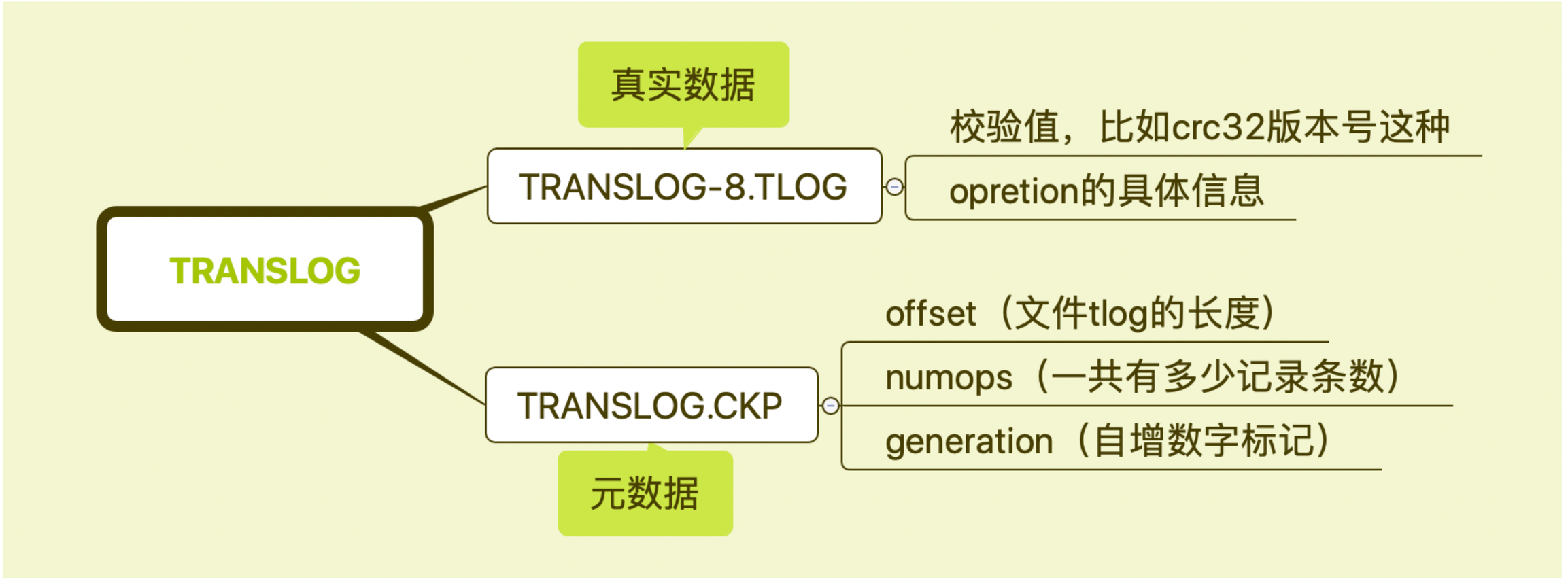
translog文件

```
-rw-r--r--  1 huafei  staff  48  9  9 11:00 translog-14.ckp
-rw-r--r--  1 huafei  staff  43  9  7 11:34 translog-14.tlog
-rw-r--r--  1 huafei  staff  48  9  2 19:22 translog-2.ckp
-rw-r--r--  1 huafei  staff  43  8 23 17:24 translog-2.tlog
-rw-r--r--  1 huafei  staff  48  9  2 19:55 translog-3.ckp
-rw-r--r--  1 huafei  staff  43  9  2 19:22 translog-3.tlog
-rw-r--r--  1 huafei  staff  43 12 24 14:06 translog-36.tlog
-rw-r--r--  1 huafei  staff  48  9  2 20:53 translog-4.ckp
-rw-r--r--  1 huafei  staff  43  9  2 19:55 translog-4.tlog
-rw-r--r--  1 huafei  staff  48  9  3 09:37 translog-5.ckp
-rw-r--r--  1 huafei  staff  43  9  2 20:53 translog-5.tlog
-rw-r--r--  1 huafei  staff  48  9  3 10:03 translog-6.ckp
-rw-r--r--  1 huafei  staff  43  9  3 09:37 translog-6.tlog
-rw-r--r--  1 huafei  staff  48  9  3 10:29 translog-7.ckp
-rw-r--r--  1 huafei  staff  43  9  3 10:03 translog-7.tlog
-rw-r--r--  1 huafei  staff  48  9  3 14:02 translog-8.ckp
-rw-r--r--  1 huafei  staff  43  9  3 10:29 translog-8.tlog
-rw-r--r--  1 huafei  staff  48 12 24 14:06 translog.ckp
```

主要是2种文件，ckp和tlog。比如上面当前的ckp就是translog.ckp，tlog就是translog-36.tlog，其他都是历史的。后面会详细讲两者的结构和里面的数据格式以及历史的数据何时触发删除。

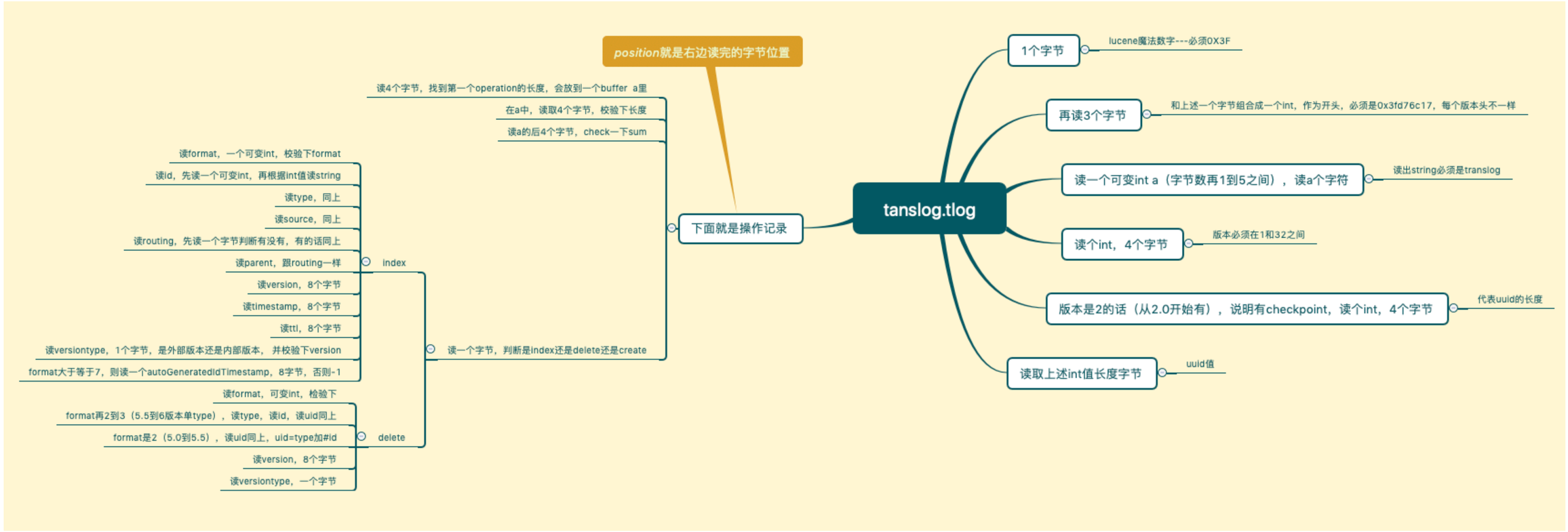
translog底层文件

translog文件



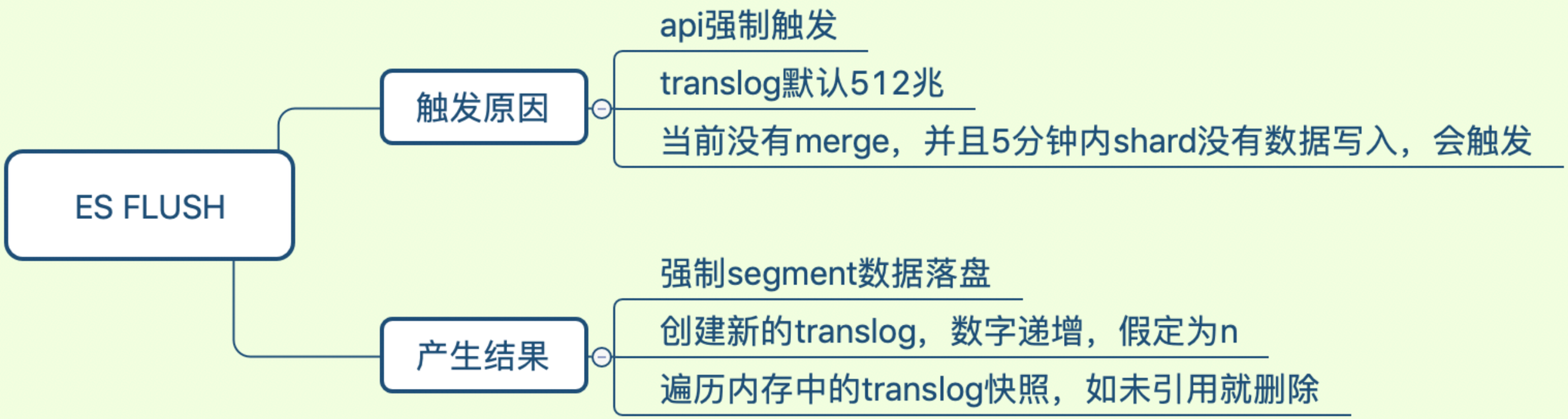
translog底层文件

translog.tlog的信息



translog删除机制

flush原因以及产生的结果



translog删除机制

删除translog代码，截取部分

```
    try (ReleasableLock ignored = writeLock.acquire()) {
        if (closed.get()) {
            // we're shutdown potentially on some tragic event - don't delete anything
            return;
        }
        //从引用的快照中获取最小的generation
        long minReferencedGen = outstandingViews.stream().mapToLong(Translog.View::minTr
        //跟当前比，取最小的
        minReferencedGen = Math.min(lastCommittedTranslogFileGeneration, minReferencedGe
        final long finalMinReferencedGen = minReferencedGen;

        //遍历readers中包含的translog，将小于generation的reader放入list中
        List<TranslogReader> unreferenced = readers.stream().filter(r -> r.getGeneration
        for (final TranslogReader unreferencedReader : unreferenced) {
            Path translogPath = unreferencedReader.path();
            logger.trace("delete translog file - not referenced and not current anymore
            //close reader
```

副本shard恢复

副本

es的一个索引很多个主shard，每个主shard都会有对应的副本shard（可以理解为主shard的一份拷贝）

副本的作用

有利

1. 高可用，主shard挂了，副本shard顶上去
2. 分担查询的压力，查询轮训分配到主副shard

缺点

1. 降低写入的性能，主写成功后，再写副，都成功后返回client成功
2. 主副极端情况下会有不一致的情况，缺少主副的数据校验

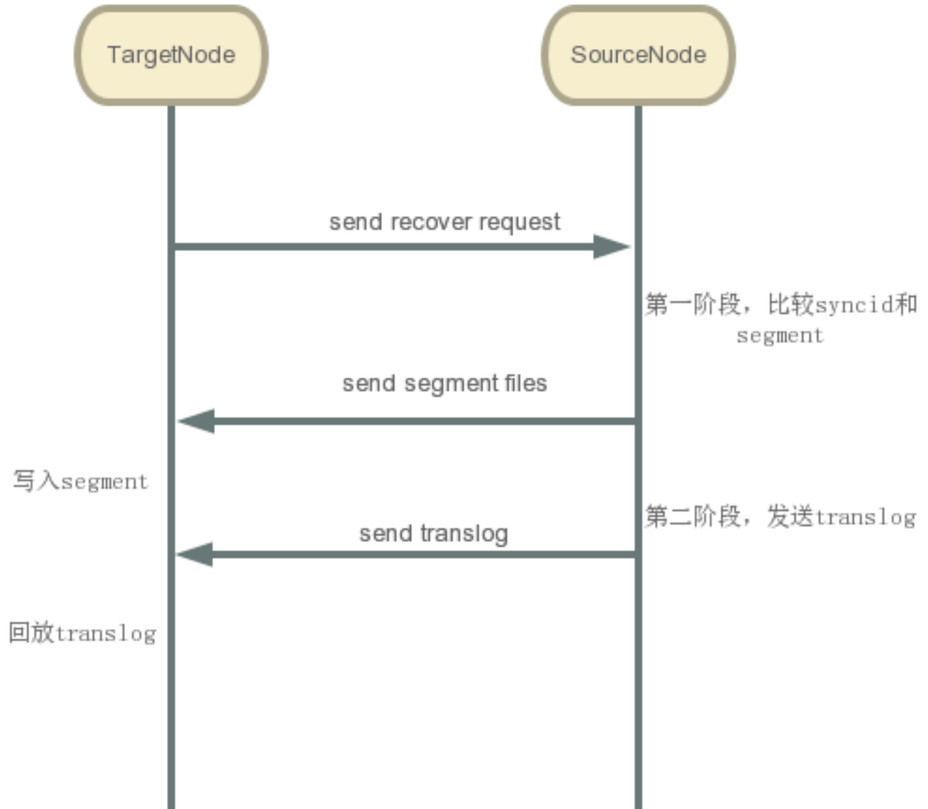
副本需要恢复的场景

1. 机器重启，shard需要重新分配
2. shard迁移，从一个节点到另一个节点
3. 副本有问题，需要剔除重新分配

副本shard恢复

分片恢复总体流程

ES副本分片恢复主要涉及恢复的目标节点和源节点，目标节点即故障恢复的节点，源节点为提供恢复的节点。目标节点向源节点发送分片恢复请求，源节点接收到请求后主要分两阶段来处理。第一阶段，对需要恢复的shard创建snapshot，然后根据请求中的metadata对比如果 syncid 相同且 doc 数量相同则跳过，否则对比shard的segment文件差异，将有差异的segment文件发送给target node。第二阶段，为了保证target node数据的完整性，需要将本地的translog发送给target node，且对接收到的translog进行回放。整体流程如下图所示。



副本shard恢复

发送请求

```
... ...
metadataSnapshot = recoveryTarget.indexShard().snapshotStoreMetadata();
... ...
// 创建recovery quest
request = new StartRecoveryRequest(recoveryTarget.shardId(),
recoveryTarget.indexShard().routingEntry().allocationId().getId(), recoveryTarget.sourceNode(),
clusterService.localNode(), metadataSnapshot, recoveryTarget.state().getPrimary(), recoveryTarget.recoveryId());
... ...
// 向源节点发送请求，请求恢复
cancellableThreads.execute(() -> responseHolder.set(
        transportService.submitRequest(request.sourceNode(), PeerRecoverySourceService.Actions.START_RECOVERY, request,
                new FutureTransportResponseHandler<RecoveryResponse>() {
                    @Override
                    public RecoveryResponse newInstance() {
                        return new RecoveryResponse();
                    }
                }).txGet()));
```

副本shard恢复

源节点接收到请求处理

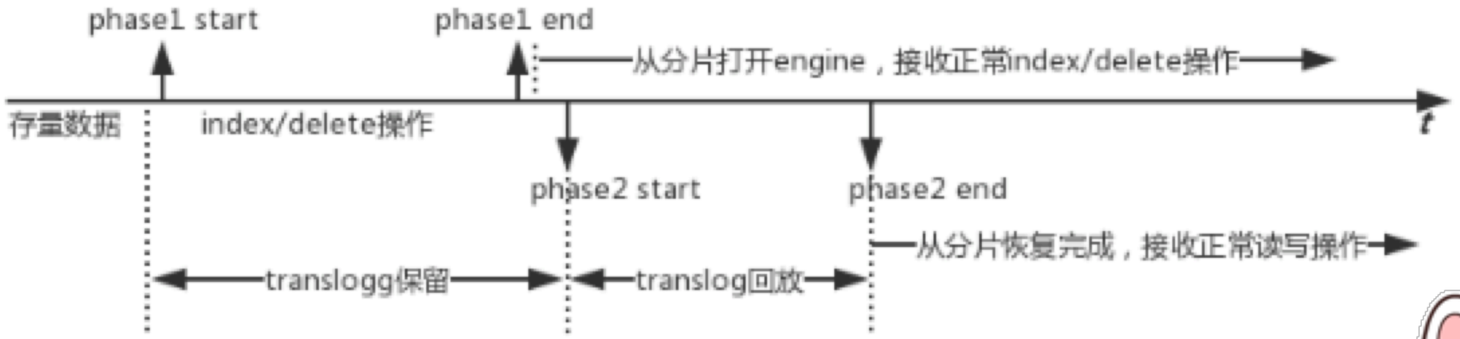
```
public RecoveryResponse recoverToTarget() throws IOException { // 恢复分为两阶段
    try (Translog.View translogView = shard.acquireTranslogView()) {
        final IndexCommit phase1Snapshot;
        try {
            phase1Snapshot = shard.acquireIndexCommit(false);
        } catch (Exception e) {
            IOUtils.closeWhileHandlingException(translogView);
            throw new RecoveryEngineException(shard.shardId(), 1, "Snapshot failed", e);
        }
        try {
            phase1(phase1Snapshot, translogView);  // 第一阶段，比较syncid和segment,然后得出有差异的部分，主动将数据推送给请求方
        } catch (Exception e) {
            throw new RecoveryEngineException(shard.shardId(), 1, "phase1 failed", e);
        } finally {
            try {
                shard.releaseIndexCommit(phase1Snapshot);
            } catch (IOException ex) {
                logger.warn("releasing snapshot caused exception", ex);
            }
        }
        // engine was just started at the end of phase 1
        if (shard.state() == IndexShardState.RELOCATED) {
            throw new IndexShardRelocatedException(request.shardId());
        }
        try {
            phase2(translogView.snapshot()); // 第二阶段，发送translog
        } catch (Exception e) {
            throw new RecoveryEngineException(shard.shardId(), 2, "phase2 failed", e);
        }
        finalizeRecovery();
    }
    return response;
}
```

副本shard恢复

目标节点恢复

1. 接受第一阶段的segment，落盘
2. 打开引擎，正常接受index和update操作
3. 接受translog的回放

（2和3是并行的）



这里可以思考下如何保证完整性和一致性，会有数据丢失的可能吗?

副本shard恢复

- 完整性

首先，phase1阶段，保证了存量的历史数据可以恢复到从分片。phase1阶段完成后，从分片引擎打开，可以正常处理index、delete请求，而translog覆盖完了整个phase1阶段，因此在phase1阶段中的index/delete操作都将被记录下来，在phase2阶段进行translog回放时，副本分片正常的index和delete操作和translog是并行执行的，这就保证了恢复开始之前的数据、恢复中的数据都会完整的写入到副本分片，保证了数据的完整性。

- 一致性

由于phase1阶段完成后，从分片便可正常处理写入操作，而此时从分片的写入和phase2阶段的translog回放时并行执行的，如果translog的回放慢于正常的写入操作，那么可能会导致老的数据后写入，造成数据不一致。ES为了保证数据的一致性在进行写入操作时，会比较当前写入的版本和lucene文档版本号，如果当前版本更小，说明是旧数据则不会将文档写入lucene。

有一种情况我觉得会不一致，比如业务方正常过来的delete数据,如果此时这个数据还在translog中（还未回放到副本中），是不是副本那边会删除不了，最终导致主shard和副本shard不一致，副本数据会多（跟刚遇到ump的一个索引很像，副本shard和主不一致，副本多）-------后面有时间验证下这个，会不会有这种情况发生

利用translog来同步

- 完整性

首先，phase1阶段，保证了存量的历史数据可以恢复到从分片。phase1阶段完成后，从分片引擎打开，可以正常处理index、delete请求，而translog覆盖完了整个phase1阶段，因此在phase1阶段中的index/delete操作都将被记录下来，在phase2阶段进行translog回放时，副本分片正常的index和delete操作和translog是并行执行的，这就保证了恢复开始之前的数据、恢复中的数据都会完整的写入到副本分片，保证了数据的完整性。

- 一致性

由于phase1阶段完成后，从分片便可正常处理写入操作，而此时从分片的写入和phase2阶段的translog回放时并行执行的，如果translog的回放慢于正常的写入操作，那么可能会导致老的数据后写入，造成数据不一致。ES为了保证数据的一致性在进行写入操作时，会比较当前写入的版本和lucene文档版本号，如果当前版本更小，说明是旧数据则不会将文档写入lucene。

有一种情况我觉得会不一致，比如业务方正常过来的delete数据,如果此时这个数据还在translog中（还未回放到副本中），是不是副本那边会删除不了，最终导致主shard和副本shard不一致，副本数据会多（跟刚遇到ump的一个索引很像，副本shard和主不一致，副本多）-------后面有时间验证下这个，会不会有这种情况发生

下期分享预告

部分lucene底层文件的结构

translog介绍

重建索引的实现

倒排

分词插件

分词器