阿里云开发者社区 ALIBABA CLOUD DEVELOPER COMMUNITY | 阿里巴巴云原生 | RocketMQ

Apache RocketMQ 源码解析

藏经阁

阿里云开发者电子书系列

扫一扫加入作者公众号
中间件兴趣圈

扫一扫关注
RocketMQ 官微

扫一扫关注【阿里巴巴云原生】公众号
获取第一手技术干货

阿里云开发者“藏经阁”
海量免费电子书下载

作者简介

作者简介

丁威，《 RocketMQ 技术内幕 》作者，RocketMQ 官方社区优秀布道师，荣获 CSDN2020 博客之星亚军；担任中通快递研发中心资深架构师，维护『 中间件兴趣圈 』公众号，主打成体系剖析 Java 主流中间件，尝试从源码分析、架构设计、实战、故障分析等维度深刻揭晓中间件技术，已覆盖 RocketMQ、Dubbo、Sentienl、Kafka、Canal、MyCat、ElasticJob、ElasticSearch 等。

推荐人及推荐序

推荐人

杜恒，Apache RocketMQ PMC Member/committer，Linux OpenMessaging TSC Member，目前负责 RocketMQ 专有云商业化以及开源技术生态构建。具有多年分布式系统、中间件研究及工程经验。目前对分布式中间件、K8s、微服务、物联网、Serverless 感兴趣。

推荐序

Apache RocketMQ 作为一款高吞吐，抗万亿消息堆积的云原生消息平台，目前已经被国内 75% 以上互联网、金融等公司所采用，逐渐成为企业 IT 架构的核心基础设施。

丁威老师作为资深架构师，在分布式架构、存储方面功底深厚，目前在企业内部负责着日均千亿级消息流转的 RocketMQ 集群。本书不仅由浅入深的介绍了 RocketMQ 的架构与实现，而且包含了多年线上超大规模集群开发运维经验的总结，通过本书不仅能够掌握分布式消息平台的设计原理，对线上疑难问题排查分析、性能调优与架构设计也大有帮助。

目录

2.1 RocketMQ DLedger 多副本即主从切换专栏回顾(源码阅读技巧篇)

RocketMQ DLedger 多副本即主从切换专栏总共包含 9 篇文章，时间跨度大概为 2 个月的时间，笔者觉得授人以鱼不如授人以渔，借以这个系列来展示该系列的创作始末，展示笔者阅读源码的技巧。

首先在下决心研读 RocketMQ DLedger 多副本(主从切换)的源码之前，首先还是要通过官方的分享、百度等途径对该功能进行一些基本的了解。

我们了解到 RocketMQ 在 4.5.0 之前提供了主从同步功能，即当主节点宕机后，消费端可以继续从从节点上消费消息，但无法继续向该复制组发送消息。RocketMQ 4.5.0 版本引入了多副本机制，即 DLedger，支持主从切换，即当一个复制组内的主节点宕机后，会在该复制组内触发重新选主，选主完成后即可继续提供消息写功能。同时还了解到 rocketmq 主从切换是基于 raft 协议的。

raft 协议是何许人也，我猜想大部分读者对这个名词并不陌生，但像笔者一样只是听过其大体作用但并未详细学习的应该也不在少数，故我觉得看 RocketMQ DLedger 多副本即主从切换之前应该重点了解 raft 协议。

一、RocketMQ 多副本前置篇：初探 raft 协议

本文主要根据 raft 官方提供的动画来学习了解 raft 协议，从本文基本得知了 raft 协议主要包含两个重要部分：选主 以及 日志复制。在了解了 raft 协议的选主、日志复制的基本实现后，然后就可以步入到 RocketMQ DLedger 多副本即主从切换的源码研究了，以探究大神是如何实现 raft 协议的。同时在了解到了 raft 协议的选主部分内容后，自己也可以简单的思考，如果自己去实现 raft 协议，应该要实现哪些关键点，当时我的思考如下：

1.3 思考如何实现Raft选主

1. 节点状态
   需要引入3中节点状态：Follower(跟随者)、Candidate(候选者)，投票的触发点，Leader(主节点)。
2. 进入投票状态的计时器
   Follower、Candidate 两个状态时，需要维护一个计时器，每次定时时间从150ms-300ms之间进行随机，即每个节点的每次的计时过期不一样，Follower状态时，计时器到点后，触发一轮投票。节点在收到投票请求、Leader 的心跳请求并作出响应后需要重置定时器。
3. 投票轮次Team
   Candidate 状态的节点，每发起一轮投票，Term 加一；Term的存储。
4. 投票机制
   每一轮一个节点只能为一个节点投赞成票，例如节点A中维护的轮次为3，并且已经为节点B投了赞成票，如果收到其他节点，投票轮次为3，则会投反对票，如果收到轮次为4的节点，是又可以投赞成票的。
5. 成为Leader的条件
   必须得到集群中节点的大多数，即超过半数，例如如果集群中有3个节点，则必须得到两票，如果其中一台服务器宕机，剩下的两个节点，还能进行选主吗？答案是可以的，因为可以得到2票，超过初始集群中3的一半，所以通常集群中的机器各位尽量为计数，因为4台的可用性与3台的一样。

这样在看源码时更加有针对性，不至于在阅读源码过程中“迷失”。

二、源码分析 RocketMQ DLedger 多副本之 Leader 选主

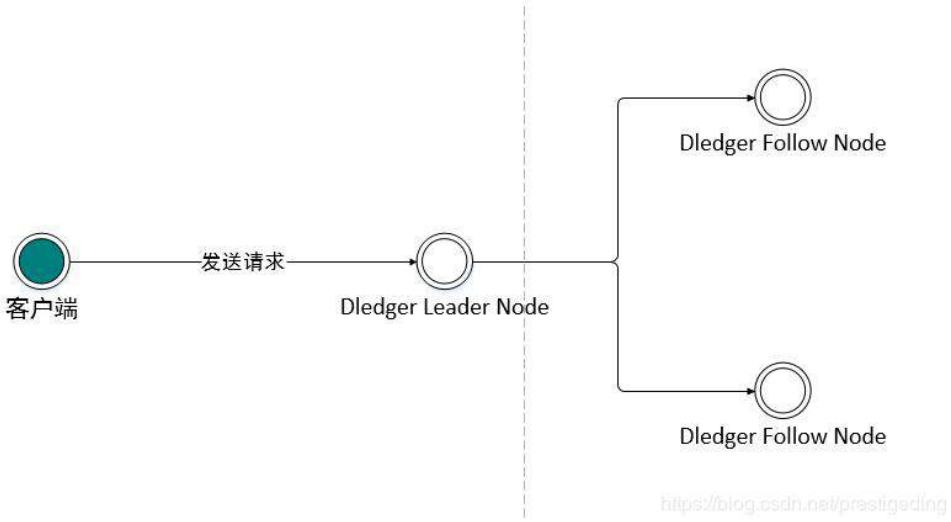
本文按照上一篇的思路，重点对 DLedgerLeaderElector 的实现进行了详细分析，特别是其内部的状态机流转，最后也给出一张流程图对选主过程进行一个简单的梳理与总结。

温馨提示：如果在阅读源码的过程中一时无法理解，可以允许其提供的单元测试，DEBUG 一下，可以起到拨云见雾之效。

三、源码分析 RocketMQ DLedger 多副本存储实现

在学习完 DLedger 选主实现后，接下来将重点突破 raft 协议的另外一个部分：日志复制。因为日志复制将涉及到存储，故在学习日志复制之前，先来看一下 DLedger 与存储相关的设计，例如 DLedger 日志条目的存储协议、日志在服务器的组织等关系，这部分类比 RocketMQ commitlog 等的存储。

四、源码分析 RocketMQ DLedger(多副本) 之日志追加流程



在学习完 DLedger 多副本即主从切换 日志存储后,我们将正式进入到日志复制部分,从上图我们可以简单了解，日志复制其实包含两个比较大的阶段，第一阶段是指主节点(Leader)接受客户端请求后，将数据先存储到主服务器中，然后再将数据转发到它的所有从节点。故本篇文章中的关注第一阶段：日志追加。

五、源码分析 RocketMQ DLedger(多副本) 之日志复制(传播)

本文继续关注日志复制的第二个阶段，包含主节点日志转发、从节点接收日志、主节点对日志转发进行仲裁,即需要实现只有超过集群半数节点都存储成功才认为该消息已成功提交，才会对客户端承偌消息发送成功。

六、基于 raft 协议的 RocketMQ DLedger 多副本日志复制设计原理

源码解读 raft 协议的日志复制部分毕竟比较枯燥，故本文梳理了 3 张流程图，并对日志的实现要点做一个总结，以此来介绍 rocketmq Dledger 多副本即主从切换部分的 raft 协议的解读。

七、RocketMQ 整合 DLedger(多副本)即主从切换实现平滑升级的设计技巧

前面 6 篇文章都聚焦在 raft 协议的选主与日志复制。从本节开始将介绍 rocketmq 主从切换的实现细节，基于 raft 协议已经可以实现主节点的选主与日志复制，主从切换的另外一个核心就是主从切换后元数据的同步，例如 topic、消费组订阅信息、消息消费进度等。另外主从切换是 rocketmq 4.5.0 版本才引入的，如果从老版本升级到 4.5.0，直接兼容原先的消息是重中之中，故本文将详细剖析其设计要点。

八、源码分析 RocketMQ DLedger 多副本即主从切换实现原理

从设计上理解了平滑升级的技巧，本篇就从源码角度剖析主从切换的实现要点，即重点关注元数据的同步（特别是消息消费进度的同步）。

九、RocketMQ DLedger 多副本即主从切换实战

经过前面 8 篇文章的铺垫，我相信大家对 DLedger 的实现原理有了一个全新的认识，本篇作为该系列的收官之作，介绍如何从主从同步集群平滑升级到 DLedger，即主从切换版本，并对功能进行验证。

整体总结一下就是首先从整体上认识其核心要点，然后逐步展开，逐步分解形成一篇一篇的文章，在遇到看不懂的时候，可以 debug 官方提供的单元测试用例。

如果本文对大家有所帮助的话，麻烦帮忙点个【在看】，谢谢。

温馨提示：本专栏是《RocketMQ 技术内幕》作者倾力打造的又一个精彩系列，也是《RocketMQ 技术内幕》第二版的原始素材。

2.2 源码分析 RocketMQ ACL

有关 RocketMQ ACL 的使用请查看上一篇《RocketMQ ACL 使用指南》，本文从源码的角度，分析一下 RocketMQ ACL 的实现原理。

备注：RocketMQ 在 4.4.0 时引入了 ACL 机制，本文代码基于 RocketMQ4.5.0 版本。

根据 RocketMQ ACL 使用手册，我们应该首先看一下 Broker 服务器在开启 ACL 机制时如何加载配置文件，并如何工作的。

一、BrokerController#initialAcl

Broker 端 ACL 的入口代码为：BrokerController#initialAcl

```
    private void initialAcl() {
        if (!this.brokerConfig.isAclEnable()) {                                    // @1
            log.info("The broker dose not enable acl");
            return;
        }

        List<AccessValidator> accessValidators = ServiceProvider.load(ServiceProvider.ACL_VALIDATOR_ID, AccessValidator.class);    // @2
        if (accessValidators == null || accessValidators.isEmpty()) {
            log.info("The broker dose not load the AccessValidator");
            return;
        }

        for (AccessValidator accessValidator: accessValidators) {                  // @3
            final AccessValidator validator = accessValidator;
            this.registerServerRPCHook(new RPCHook() {

    @Override
                public void doBeforeRequest(String remoteAddr, RemotingCommand request) {
                    //Do not catch the exception
```

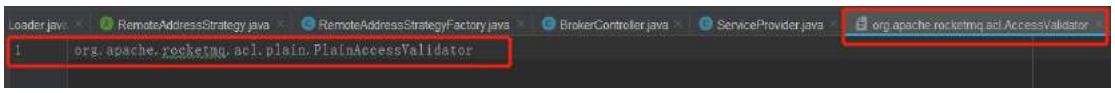
```
                    validator.validate(validator.parse(request, remoteAddr));
            // @4
                }

                @Override
                public void doAfterResponse(String remoteAddr, RemotingCommand requ
est, RemotingCommand response) {
                }
            });
        }
    }
```

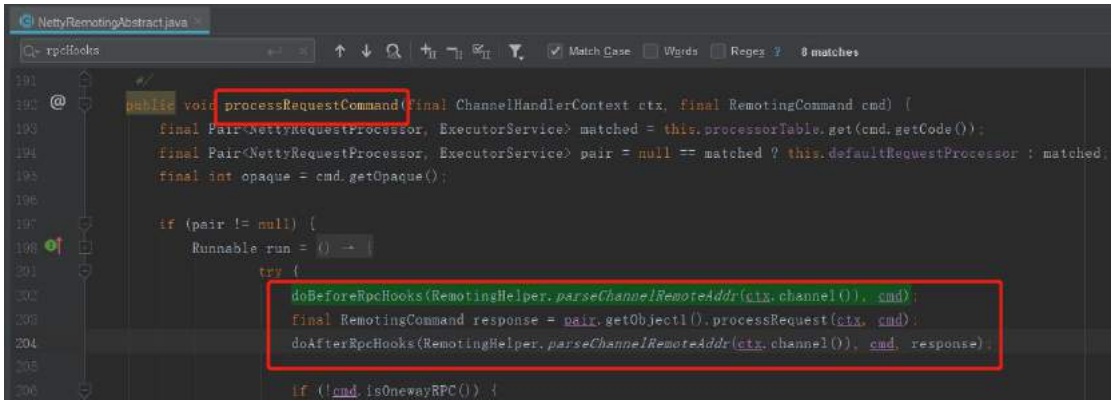
本方法的实现共 4 个关键点。

代码@1：首先判断 Broker 是否开启了 acl，通过配置参数 aclEnable 指定，默认为 false。

代码@2：使用类似 SPI 机制，加载配置的 AccessValidator,该方法返回一个列表，其实现逻辑时读取 META-INF/service/org.apache.rocketmq.acl.AccessValidator 文件中配置的访问验证器，默认配置内容如下：



代码@3：遍历配置的访问验证器(AccessValidator),并向 Broker 处理服务器注册钩子函数，RPCHook 的 doBeforeRequest 方法会在服务端接收到请求，将其请求解码后，执行处理请求之前被调用;RPCHook 的 doAfterResponse 方法会在处理完请求后，将结果返回之前被调用，其调用如图所示：

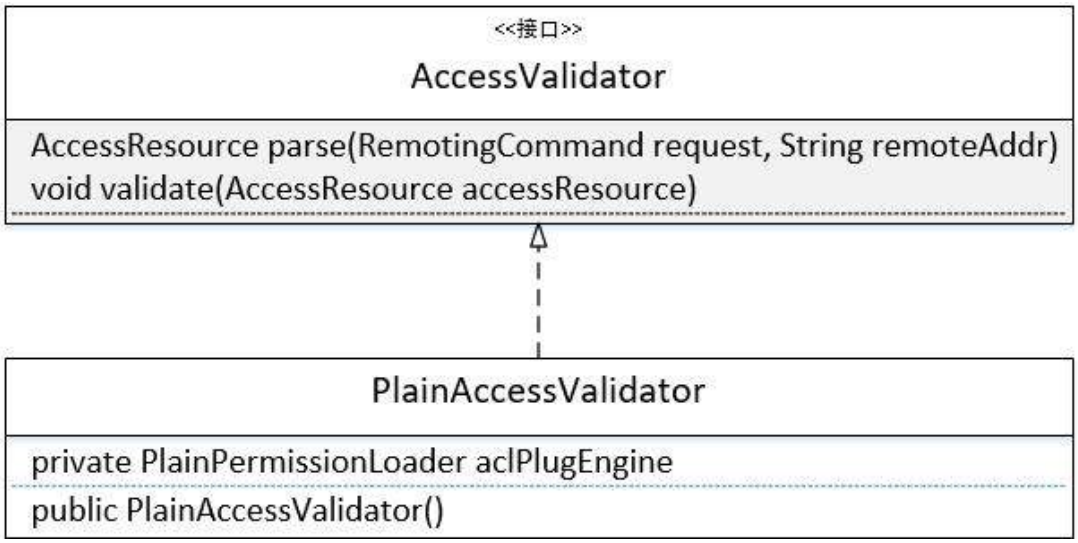


代码@4：在 RPCHook#doBeforeRequest 方法中调用 AccessValidator#validate，在真实处理命令之前，先执行 ACL 的验证逻辑，如果拥有该操作的执行权限，则放行，否则抛出 AclException。

接下来，我们将重点放到 Broker 默认实现的访问验证器：PlainAccessValidator。

二、PlainAccessValidator

1. 类图



AccessValidator

访问验证器接口，主要定义两个接口。

1. AccessResource parse(RemotingCommand request, String remoteAddr)
   从请求头中解析本次请求对应的访问资源，即本次请求需要的访问权限。

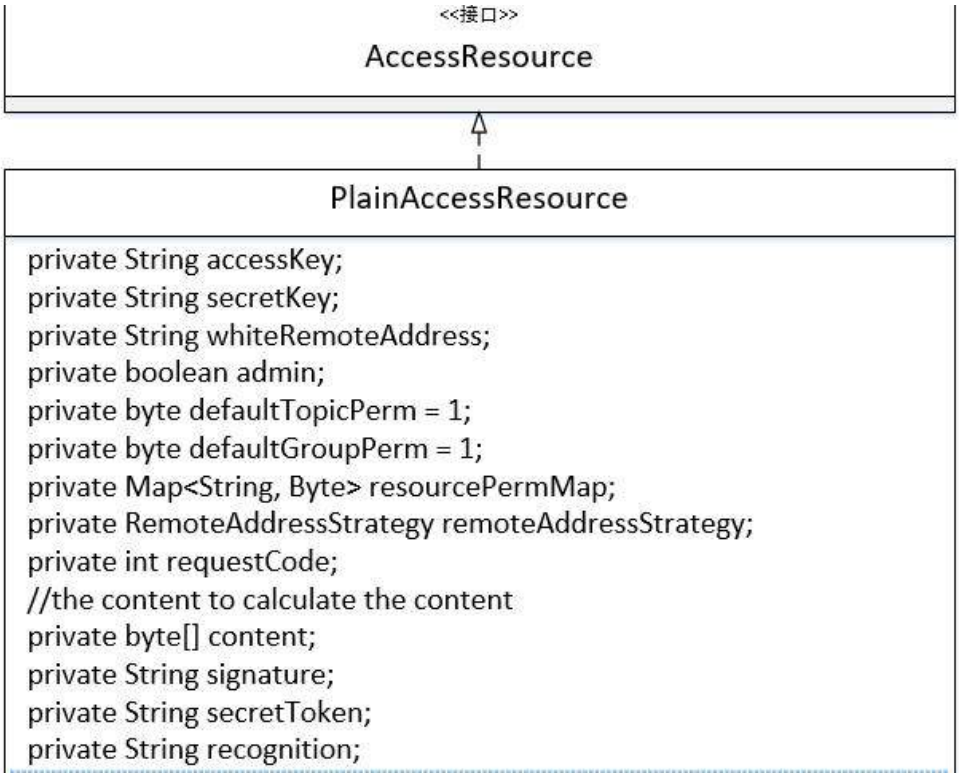
2. void validate(AccessResource accessResource)
   根据本次需要访问的权限，与请求用户拥有的权限进行对比验证，判断是拥有权限，如果没有访问该操作的权限，则抛出异常，否则放行。

PlainAccessValidator

RocketMQ 默认提供的基于 yml 配置格式的访问验证器。

接下来我们重点看一下 PlainAccessValidator 的 parse 方法与 validate 方法的实现细节。在讲解该方法之前，我们首先认识一下 RocketMQ 封装访问资源的 PlainAccessResource。

2. PlainAccessResource 类图



我们对其属性一一做个介绍：

- private String accessKey
  访问 Key，用户名。

- private String secretKey
  用户密码。

- private String whiteRemoteAddress
  远程 IP 地址白名单。

- private boolean admin
  是否是管理员角色。

- private byte defaultTopicPerm = 1
  默认 topic 访问权限，即如果没有配置 topic 的权限，则 Topic 默认的访问权限为 1，表示为 DENY。

- private byte defaultGroupPerm = 1
  默认的消费组访问权限，默认为 DENY。

- private Map<String, Byte> resourcePermMap
  资源需要的访问权限映射表。

- private RemoteAddressStrategy remoteAddressStrategy
  远程 IP 地址验证策略。

- private int requestCode
  当前请求的 requestCode。

- private byte[] content
  请求头与请求体的内容。

- private String signature
  签名字符串，这是通常的套路，在客户端时，首先将请求参数排序，然后使用 secretKey 生成签名字符串，服务端重复这个步骤，然后对比签名字符串，如果相同，则认为登录成功，否则失败。

- private String secretToken
  密钥 token。

- private String recognition
  目前作用未知，代码中目前未被使用。

2. 构造方法

```
public PlainAccessValidator() {
    aclPlugEngine = new PlainPermissionLoader();
}
```

构造函数，直接创建 PlainPermissionLoader 对象，从命名上来看，应该是触发 acl 规则的加载，即解析 plain_acl.yml，接下来会重点探讨，即 acl 启动流程之配置文件的解析。

3. parse 方法

该方法的作用就是从请求命令中解析出本次访问所需要的访问权限，最终构建 AccessResource 对象，为后续的校验权限做准备。

```
PlainAccessResource accessResource = new PlainAccessResource();
if (remoteAddr != null && remoteAddr.contains(":")) {
    accessResource.setWhiteRemoteAddress(remoteAddr.split(":")[0]);
} else {
    accessResource.setWhiteRemoteAddress(remoteAddr);
}
```

Step1：首先创建 PlainAccessResource，从远程地址中提取出远程访问 IP 地址。

```
if (request.getExtFields() == null) {
    throw new AclException("request's extFields value is null");
}
accessResource.setRequestCode(request.getCode());
accessResource.setAccessKey(request.getExtFields().get(SessionCredentials.ACCESS
_KEY));
accessResource.setSignature(request.getExtFields().get(SessionCredentials.SIGNATU
RE));
accessResource.setSecretToken(request.getExtFields().get(SessionCredentials.SECUR
ITY_TOKEN));
```

Step2：如果请求头中的扩展字段为空，则抛出异常，如果不为空，则从请求头中读取 requestCode、accessKey(请求用户名)、签名字符串(signature)、secretToken。

```
    try {
                switch (request.getCode()) {
                    case RequestCode.SEND_MESSAGE:
                        accessResource.addResourceAndPerm(request.getExtFields().get("
topic"), Permission.PUB);
                        break;
                    case RequestCode.SEND_MESSAGE_V2:
                        accessResource.addResourceAndPerm(request.getExtFields().get("
b"), Permission.PUB);
                        break;
                    case RequestCode.CONSUMER_SEND_MSG_BACK:
                        accessResource.addResourceAndPerm(request.getExtFields().get("
originTopic"), Permission.PUB);
                        accessResource.addResourceAndPerm(getRetryTopic(request.get
ExtFields().get("group")), Permission.SUB);
                        break;
                    case RequestCode.PULL_MESSAGE:
                        accessResource.addResourceAndPerm(request.getExtFields().get("
topic"), Permission.SUB);
                        accessResource.addResourceAndPerm(getRetryTopic(request.get
ExtFields().get("consumerGroup")), Permission.SUB);
                        break;
                    case RequestCode.QUERY_MESSAGE:
          accessResource.addResourceAndPerm(request.getExtFields().get("topic"), Permissi
on.SUB);
                        break;
                    case RequestCode.HEART_BEAT:
                        HeartbeatData heartbeatData = HeartbeatData.decode(request.get
Body(), HeartbeatData.class);
                        for (ConsumerData data : heartbeatData.getConsumerDataSet()) {
                            accessResource.addResourceAndPerm(getRetryTopic(data.get
GroupName()), Permission.SUB);
                            for (SubscriptionData subscriptionData : data.getSubscriptionD
ataSet()) {
                                accessResource.addResourceAndPerm(subscriptionData.g
etTopic(), Permission.SUB);
                            }
                        }
                        break;
                    case RequestCode.UNREGISTER_CLIENT:
                        final UnregisterClientRequestHeader unregisterClientRequestHead
er =
                            (UnregisterClientRequestHeader) request
```

```
                                .decodeCommandCustomHeader(UnregisterClientRequestH
eader.class);
                        accessResource.addResourceAndPerm(getRetryTopic(unregisterCli
entRequestHeader.getConsumerGroup()), Permission.SUB);
                        break;
                    case RequestCode.GET_CONSUMER_LIST_BY_GROUP:
                        final GetConsumerListByGroupRequestHeader getConsumerListBy
GroupRequestHeader =
                            (GetConsumerListByGroupRequestHeader) request
                                .decodeCommandCustomHeader(GetConsumerListByGrou
pRequestHeader.class);
                        accessResource.addResourceAndPerm(getRetryTopic(getConsum
erListByGroupRequestHeader.getConsumerGroup()), Permission.SUB);
                        break;
                    case RequestCode.UPDATE_CONSUMER_OFFSET:
                        final UpdateConsumerOffsetRequestHeader updateConsumerOffse
tRequestHeader =
                            (UpdateConsumerOffsetRequestHeader) request
        .decodeCommandCustomHeader(UpdateConsumerOffsetRequestHeader.class);
                        accessResource.addResourceAndPerm(getRetryTopic(updateCons
umerOffsetRequestHeader.getConsumerGroup()), Permission.SUB);
                        accessResource.addResourceAndPerm(updateConsumerOffsetReq
uestHeader.getTopic(), Permission.SUB);
                        break;
                    default:
                        break;

                }
            } catch (Throwable t) {
                throw new AclException(t.getMessage(), t);
            }
```

Step3：根据请求命令，设置本次请求需要拥有的权限，上述代码比较简单，就是从请求中得出本次操作的 Topic、消息组名称，为了方便区分 topic 与消费组，消费组使用消费者对应的重试主题，当成资源的 Key，从这里也可以看出，当前版本需要进行 ACL 权限验证的请求命令如下：

```
SEND_MESSAGE
SEND_MESSAGE_V2
CONSUMER_SEND_MSG_BACK
PULL_MESSAGE
```

```
    QUERY_MESSAGE
    HEART_BEAT
    UNREGISTER_CLIENT
    GET_CONSUMER_LIST_BY_GROUP
    UPDATE_CONSUMER_OFFSET

    // Content
    SortedMap<String, String> map = new TreeMap<String, String>();
    for (Map.Entry<String, String> entry : request.getExtFields().entrySet()) {
        if (!SessionCredentials.SIGNATURE.equals(entry.getKey())) {
            map.put(entry.getKey(), entry.getValue());
        }
    }
    accessResource.setContent(AclUtils.combineRequestContent(request, map));
    return accessResource;
```

Step4：对扩展字段进行排序，便于生成签名字符串，然后将扩展字段与请求体(body)写入 content 字段。完成从请求头中解析出本次请求需要验证的权限。

4. validate 方法

```
    public void validate(AccessResource accessResource) {
        aclPlugEngine.validate((PlainAccessResource) accessResource);
    }
```

验证权限，即根据本次请求需要的权限与当前用户所拥有的权限进行对比，如果符合，则正常执行；否则抛出 AclException。

为了揭开配置文件的解析与验证，我们将目光投入到 PlainPermissionLoader。

三、PlainPermissionLoader

该类的主要职责：加载权限，即解析 acl 主要配置文件 plain_acl.yml。

1. 类图

PlainPermissionLoader
private static final String DEFAULT_PLAIN_ACL_FILE = "conf/plain_acl.yml" private String fileName = System.getProperty("rocketmq.acl.plain.file", DEFAULT_PLAIN_ACL_FILE) private Map<String/** AccessKey **/, PlainAccessResource> plainAccessResourceMap private List<RemoteAddressStrategy> globalWhiteRemoteAddressStrategy private RemoteAddressStrategyFactory remoteAddressStrategyFactory = new RemoteAddressStrategyFactory() private boolean isWatchStart
public PlainPermissionLoader() public void load() public void validate(PlainAccessResource plainAccessResource)

下面对其核心属性与核心方法一一介绍：

- DEFAULT_PLAIN_ACL_FILE

  默认 acl 配置文件名称，默认值为 conf/plain_acl.yml。

- String fileName

  acl 配置文件名称，默认为 DEFAULT_PLAIN_ACL_FILE ,可以通过系统参数-Drocketmq.acl.plain.file=fileName 指定。

- Map<String, PlainAccessResource> plainAccessResourceMap

  解析出来的权限配置映射表，以用户名为键。

- RemoteAddressStrategyFactory remoteAddressStrategyFactory

  远程 IP 解析策略工厂，用于解析白名单 IP 地址。

- boolean isWatchStart

  是否开启了文件监听，即自动监听 plain_acl.yml 文件，一旦该文件改变，可在不重启服务器的情况下自动生效。

- public PlainPermissionLoader()

  构造方法。

- public void load()

  加载配置文件。

- public void validate(PlainAccessResource plainAccessResource)

  验证是否有权限访问待访问资源。

下面重点来探讨：

2. PlainPermissionLoader 构造方法

```
public PlainPermissionLoader() {
    load();
    watch();
}
```

在构造方法中调用 load 与 watch 方法。

3. load

```
Map<String, PlainAccessResource> plainAccessResourceMap = new HashMap<>();
List<RemoteAddressStrategy> globalWhiteRemoteAddressStrategy = new ArrayList<>
();
String path = fileHome + File.separator + fileName;
JSONObject plainAclConfData = AclUtils.getYamlDataObject(path,JSONObject.class);
```

Step1：初始化 plainAccessResourceMap(用户配置的访问资源，即权限容器)、globalWhiteRemoteAddressStrategy：全局 IP 白名单访问策略。配置文件，默认为${ROCKETMQ_HOME}/conf/plain_acl.yml。

```
JSONArray globalWhiteRemoteAddressesList = plainAclConfData.getJSONArray("globa
lWhiteRemoteAddresses");
if (globalWhiteRemoteAddressesList != null && !globalWhiteRemoteAddressesList.isE
mpty()) {
    for (int i = 0; i < globalWhiteRemoteAddressesList.size(); i++) {
        globalWhiteRemoteAddressStrategy.add(remoteAddressStrategyFactory.
        getRemoteAddressStrategy(globalWhiteRemoteAddressesList.getString(i)));
    }
}
```

Step2：globalWhiteRemoteAddresses：全局白名单，类型为数组。根据配置的规则，使用 remoteAddressStrategyFactory 获取一个访问策略，下文会重点介绍其配置规则。

```
    JSONArray accounts = plainAclConfData.getJSONArray("accounts");
    if (accounts != null && !accounts.isEmpty()) {
        List<PlainAccessConfig> plainAccessConfigList = accounts.toJavaList(PlainAccess
Config.class);
        for (PlainAccessConfig plainAccessConfig : plainAccessConfigList) {
            PlainAccessResource plainAccessResource = buildPlainAccessResource(plain
AccessConfig);
            plainAccessResourceMap.put(plainAccessResource.getAccessKey(),plainAcces
sResource);
        }
    }
    this.globalWhiteRemoteAddressStrategy = globalWhiteRemoteAddressStrategy;
    this.plainAccessResourceMap = plainAccessResourceMap;
```

Step3：解析 plain_acl.yml 文件中的另外一个根元素 accounts，用户定义的权限信息。从 PlainAccessConfig 的定义来看，accounts 标签下支持如下标签：

- accessKey
- secretKey
- whiteRemoteAddress
- admin
- defaultTopicPerm
- defaultGroupPerm
- topicPerms
- groupPerms

上述标签的说明，请参考：《RocketMQ ACL 使用指南》。具体的解析过程比较容易，就不再细说。

load 方法主要完成 acl 配置文件的解析，将用户定义的权限加载到内存中。

4. watch

```
    private void watch() {
        try {
            String watchFilePath = fileHome + fileName;
```

```
    FileWatchService fileWatchService = new FileWatchService(new String[] {watchFilePat
h}, new FileWatchService.Listener() {
                        @Override
                        public void onChanged(String path) {
                            log.info("The plain acl yml changed, reload the context");
                            load();
                        }
                });
                fileWatchService.start();
                log.info("Succeed to start AclWatcherService");
                this.isWatchStart = true;
            } catch (Exception e) {
                log.error("Failed to start AclWatcherService", e);
            }
    }
```

监听器，默认以 500ms 的频率判断文件的内容是否变化。在文件内容发生变化后调用 load()方法，重新加载配置文件。那 FileWatchService 是如何判断两个文件的内容发生了变化呢?

```
    FileWatchService#hash
    private String hash(String filePath) throws IOException, NoSuchAlgorithmException {
        Path path = Paths.get(filePath);
        md.update(Files.readAllBytes(path));
        byte[] hash = md.digest();
        return UtilAll.bytes2string(hash);
    }
```

获取文件 md5 签名来做对比，这里为什么不在启动时先记录上一次文件的修改时间，然后先判断其修改时间是否变化，再判断其内容是否真正发生变化。

5. validate

```
    // Check the global white remote addr
    for (RemoteAddressStrategy remoteAddressStrategy : globalWhiteRemoteAddressStra
tegy) {
        if (remoteAddressStrategy.match(plainAccessResource)) {
            return;
        }
    }
```

Step1：首先使用全局白名单对资源进行验证，只要一个规则匹配，则返回，表示认证成功。

```
    if (plainAccessResource.getAccessKey() == null) {
        throw new AclException(String.format("No accessKey is configured"));
    }
    if (!plainAccessResourceMap.containsKey(plainAccessResource.getAccessKey())) {
        throw new AclException(String.format("No acl config for %s", plainAccessResourc
e.getAccessKey()));
    }
```

Step2：如果请求信息中，没有设置用户名，则抛出未配置 AccessKey 异常；如果 Broker 中并为配置该用户的配置信息，则抛出 AclException。

```
    // Check the white addr for accesskey
    PlainAccessResource ownedAccess = plainAccessResourceMap.get(plainAccessReso
urce.getAccessKey());
    if (ownedAccess.getRemoteAddressStrategy().match(plainAccessResource)) {
        return;
    }
```

Step3：如果用户配置的白名单与待访问资源规则匹配的话，则直接发认证通过。

```
    // Check the signature
    String signature = AclUtils.calSignature(plainAccessResource.getContent(), ownedAcc
ess.getSecretKey());
    if (!signature.equals(plainAccessResource.getSignature())) {
        throw new AclException(String.format("Check signature failed for accessKey=%s",
 plainAccessResource.getAccessKey()));
    }
```

Step4：验证签名。

```
    checkPerm(plainAccessResource, ownedAccess);
```

Step5：调用 checkPerm 方法，验证需要的权限与拥有的权限是否匹配。

checkPerm

```
    if (Permission.needAdminPerm(needCheckedAccess.getRequestCode()) && !ownedAc
cess.isAdmin()) {
        throw new AclException(String.format("Need admin permission for request code
=%d, but accessKey=%s is not", needCheckedAccess.getRequestCode(), ownedAccess.
getAccessKey()));}
```

Step6：如果当前的请求命令属于必须是 Admin 用户才能访问的权限，并且当前用户并不是管理员角色，则抛出异常，如下命令需要 admin 角色才能进行的操作：


```
    Map<String, Byte> needCheckedPermMap = needCheckedAccess.getResourcePerm
Map();
    Map<String, Byte> ownedPermMap = ownedAccess.getResourcePermMap();
    if (needCheckedPermMap == null) {
        // If the needCheckedPermMap is null,then return
        return;
    }
    if (ownedPermMap == null && ownedAccess.isAdmin()) {
        // If the ownedPermMap is null and it is an admin user, then return
        return;
    }
```

Step7：如果该请求不需要进行权限验证，则通过认证，如果当前用户的角色是管理员，并且没有配置用户权限，则认证通过，返回。

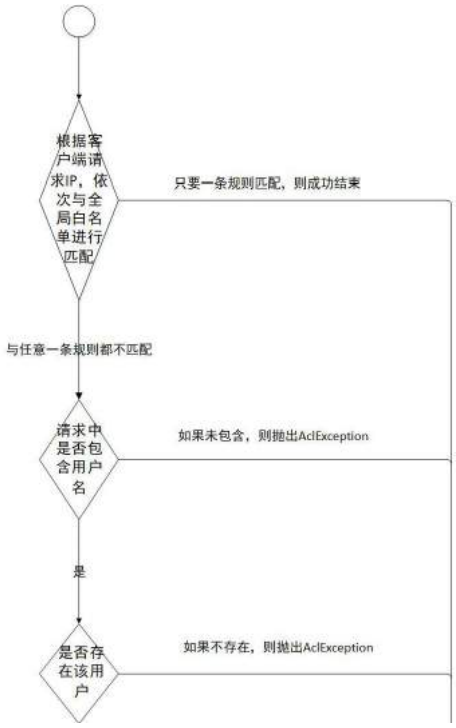
```
    for (Map.Entry<String, Byte> needCheckedEntry : needCheckedPermMap.entrySet())
{
        String resource = needCheckedEntry.getKey();
        Byte neededPerm = needCheckedEntry.getValue();
```

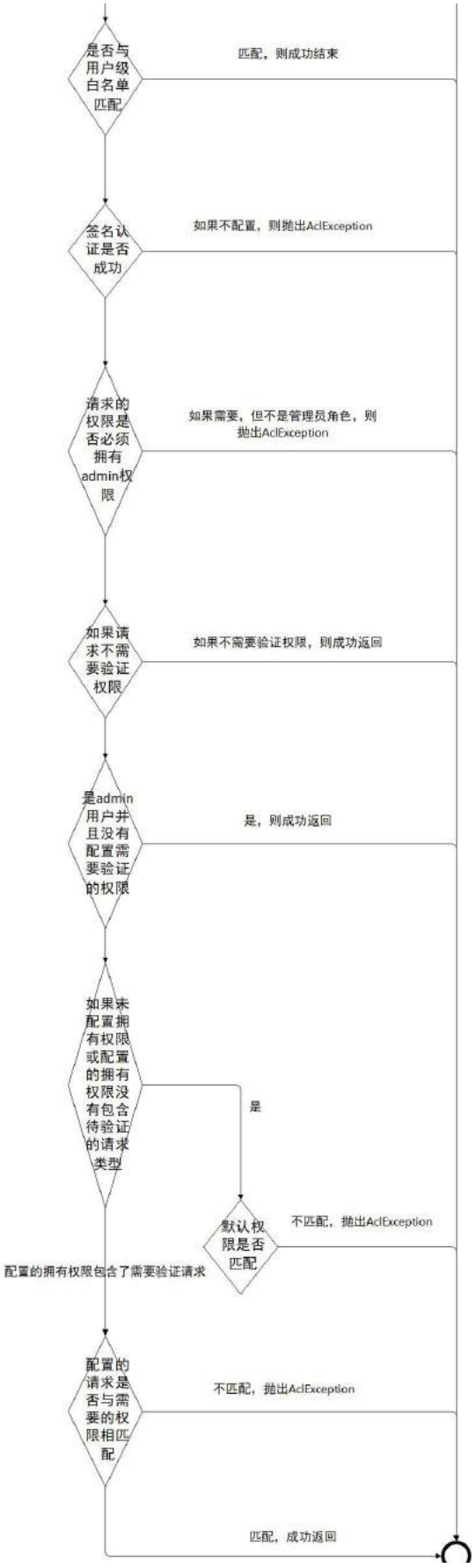
```
        boolean isGroup = PlainAccessResource.isRetryTopic(resource);

        if (ownedPermMap == null || !ownedPermMap.containsKey(resource)) {
            // Check the default perm
            byte ownedPerm = isGroup ? ownedAccess.getDefaultGroupPerm() : ownedA
ccess.getDefaultTopicPerm();
            if (!Permission.checkPermission(neededPerm, ownedPerm)) {
                throw new AclException(String.format("No default permission for %s", Plai
nAccessResource.printStr(resource, isGroup)));
            }
            continue;
        }
        if (!Permission.checkPermission(neededPerm, ownedPermMap.get(resource))) {
            throw new AclException(String.format("No default permission for %s", PlainAc
cessResource.printStr(resource, isGroup)));
        }
    }
```

Step8：遍历需要权限与拥有的权限进行对比，如果配置对应的权限，则判断是否匹配；如果未配置权限，则判断默认权限时是否允许，不允许，则抛出 AclException。

验证逻辑就介绍到这里了，下面给出其匹配流程图：





上述阐述了从 Broker 服务器启动、加载 acl 配置文件流程、动态监听配置文件、服务端权限验证流程，接下来我们看一下客户端关于 ACL 需要处理的事情。

四、AclClientRPCHook

回顾一下，我们引入 ACL 机制后，客户端的代码示例如下：

其在创建 DefaultMQProducer 时，注册 AclClientRPCHook 钩子，会在向服务端发送远程命令前后执行其钩子函数，接下来我们重点分析一下 AclClientRPCHook。

1. doBeforeRequest

```
    public void doBeforeRequest(String remoteAddr, RemotingCommand request) {
        byte[] total = AclUtils.combineRequestContent(request,
                parseRequestContent(request, sessionCredentials.getAccessKey(), session
Credentials.getSecurityToken()));    // @1
        String signature = AclUtils.calSignature(total, sessionCredentials.getSecretKey());
                                                    // @2
        request.addExtField(SIGNATURE, signature);
                                                                // @3
        request.addExtField(ACCESS_KEY, sessionCredentials.getAccessKey());
        // The SecurityToken value is unneccessary,user can choose this one.
        if (sessionCredentials.getSecurityToken() != null) {
            request.addExtField(SECURITY_TOKEN, sessionCredentials.getSecurityToken
());
        }}
```

代码@1：将 Request 请求参数进行排序，并加入 accessKey。

代码@2：对排好序的请参数，使用用户配置的密码生成签名，并最近到扩展字段 Signature，然后服务端也会按照相同的算法生成 Signature，如果相同，则表示签名验证成功(类似于实现登录的效果)。

代码@3：将 Signature、AccessKey 等加入到请求头的扩展字段中，服务端拿到这些元数据，结合请求头中的信息，根据配置的权限，进行权限校验。

关于 ACL 客户端生成签名是一种通用套路，就不在细讲了。

源码分析 ACL 的实现就介绍到这里了，下文将介绍 RocketMQ 消息轨迹的使用与实现原理分析。

2.3 源码分析 RocketMQ 消息轨迹

本文沿着《RocketMQ 消息轨迹-设计篇》的思路，从如下 3 个方面对其源码进行解读：

一、发送消息轨迹

二、消息轨迹格式

三、存储消息轨迹数据

一、发送消息轨迹流程

首先我们来看一下在消息发送端如何启用消息轨迹，示例代码如下：

```
public class TraceProducer {
    public static void main(String[] args) throws MQClientException, InterruptedExcept
ion {
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupNam
e",true);     // @1
        producer.setNamesrvAddr("127.0.0.1:9876");
        producer.start();
        for (int i = 0; i < 10; i++)
            try {
                {
                    Message msg = new Message("TopicTest",
                        "TagA",
                        "OrderID188",
                        "Hello world".getBytes(RemotingHelper.DEFAULT_CHARSET));
                    SendResult sendResult = producer.send(msg);
                    System.out.printf("%s%n", sendResult);
                }

            } catch (Exception e) {
                e.printStackTrace();
            }
        producer.shutdown();
    }
}
```

从上述代码可以看出其关键点是在创建 DefaultMQProducer 时指定开启消息轨迹跟踪。我们不妨浏览一下 DefaultMQProducer 与启用消息轨迹相关的构造函数：

```
public DefaultMQProducer(final String producerGroup, boolean enableMsgTrace)
public DefaultMQProducer(final String producerGroup, boolean enableMsgTrace, final String customizedTraceTopic)
```

参数如下：

- String producerGroup
  生产者所属组名。

- boolean enableMsgTrace
  是否开启跟踪消息轨迹，默认为 false。

- String customizedTraceTopic
  如果开启消息轨迹跟踪，用来存储消息轨迹数据所属的主题名称，默认为：RMQ_SYS_TRACE_TOPIC。

1. DefaultMQProducer 构造函数

```
public DefaultMQProducer(final String producerGroup, RPCHook rpcHook, boolean enableMsgTrace,final String customizedTraceTopic) {        // @1
        this.producerGroup = producerGroup;
        defaultMQProducerImpl = new DefaultMQProducerImpl(this, rpcHook);
        //if client open the message trace feature
        if (enableMsgTrace) {

                                        // @2
            try {
                AsyncTraceDispatcher dispatcher = new AsyncTraceDispatcher(customizedTraceTopic, rpcHook);
                dispatcher.setHostProducer(this.getDefaultMQProducerImpl());
                traceDispatcher = dispatcher;
                this.getDefaultMQProducerImpl().registerSendMessageHook(
                    new SendMessageTraceHookImpl(traceDispatcher));
```

```
                                // @3
                } catch (Throwable e) {
                        log.error("system mqtrace hook init failed ,maybe can't send msg trace d
ata");
                }
        }
    }
```

代码@1：首先介绍一下其局部变量。

- String producerGroup
  生产者所属组。

- RPCHook rpcHook
  生产者发送钩子函数。

- boolean enableMsgTrace
  是否开启消息轨迹跟踪。

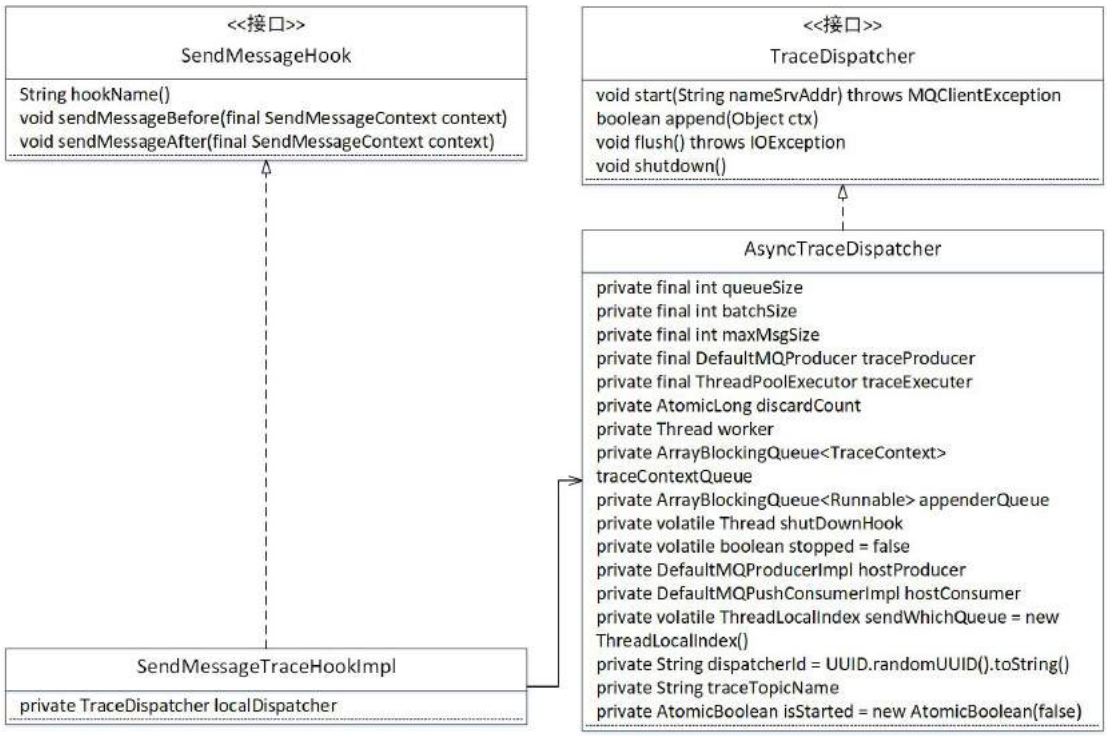
- String customizedTraceTopic
  定制用于存储消息轨迹的数据。

代码@2：用来构建 AsyncTraceDispatcher，看其名：异步转发消息轨迹数据，稍后重点关注。

代码@3：构建 SendMessageTraceHookImpl 对象，并使用 AsyncTraceDispatcher 用来异步转发。

2. SendMessageTraceHookImpl 钩子函数

SendMessageTraceHookImpl 类图



1）SendMessageHook

消息发送钩子函数，用于在消息发送之前、发送之后执行一定的业务逻辑，是记录消息轨迹的最佳扩展点。

2）raceDispatcher

消息轨迹转发处理器，其默认实现类 AsyncTraceDispatcher，异步实现消息轨迹数据的发送。下面对其属性做一个简单的介绍：

- int queueSize

  异步转发，队列长度，默认为 2048，当前版本不能修改。

- int batchSize

  批量消息条数，消息轨迹一次消息发送请求包含的数据条数，默认为 100，当前版本不能修改。

- int maxMsgSize

  消息轨迹一次发送的最大消息大小，默认为 128K，当前版本不能修改。

- DefaultMQProducer traceProducer
  用来发送消息轨迹的消息发送者。

- ThreadPoolExecutor traceExecuter
  线程池，用来异步执行消息发送。

- AtomicLong discardCount
  记录丢弃的消息个数。

- Thread worker
  woker 线程，主要负责从追加队列中获取一批待发送的消息轨迹数据，提交到线程池中执行。

- ArrayBlockingQueue<TraceContext> traceContextQueue
  消息轨迹 TraceContext 队列，用来存放待发送到服务端的消息。

- ArrayBlockingQueue<Runnable> appenderQueue
  线程池内部队列，默认长度 1024。

- DefaultMQPushConsumerImpl hostConsumer
  消费者信息，记录消息消费时的轨迹信息。

- String traceTopicName
  用于跟踪消息轨迹的 topic 名称。

源码分析 SendMessageTraceHookImpl

sendMessageBefore：

```
    public void sendMessageBefore(SendMessageContext context) {
        //if it is message trace data,then it doesn't recorded
        if (context == null || context.getMessage().getTopic().startsWith(((AsyncTraceDispa
tcher) localDispatcher).getTraceTopicName())) {    // @1
            return;
```

```
        }
        //build the context content of TuxeTraceContext
        TraceContext tuxeContext = new TraceContext();
        tuxeContext.setTraceBeans(new ArrayList<TraceBean>(1));
        context.setMqTraceContext(tuxeContext);
        tuxeContext.setTraceType(TraceType.Pub);
        tuxeContext.setGroupName(context.getProducerGroup());

// @2
        //build the data bean object of message trace
        TraceBean traceBean = new TraceBean();

            // @3
        traceBean.setTopic(context.getMessage().getTopic());
        traceBean.setTags(context.getMessage().getTags());
        traceBean.setKeys(context.getMessage().getKeys());
        traceBean.setStoreHost(context.getBrokerAddr());
        traceBean.setBodyLength(context.getMessage().getBody().length);
        traceBean.setMsgType(context.getMsgType());
        tuxeContext.getTraceBeans().add(traceBean);
    }
```

代码@1：如果 topic 主题为消息轨迹的 Topic，直接返回。

代码@2：在消息发送上下文中，设置用来跟踪消息轨迹的上下环境，里面主要包含一个 TraceBean 集合、追踪类型（TraceType.Pub）与生产者所属的组。

代码@3：构建一条跟踪消息，用 TraceBean 来表示，记录原消息的 topic、tags、keys、发送到 broker 地址、消息体长度等消息。

从上文看出，sendMessageBefore 主要的用途就是在消息发送的时候，先准备一部分消息跟踪日志，存储在发送上下文环境中，此时并不会发送消息轨迹数据。

sendMessageAfter：

```
    public void sendMessageAfter(SendMessageContext context) {
        //if it is message trace data,then it doesn't recorded
```

```
        if (context == null || context.getMessage().getTopic().startsWith(((AsyncTraceDispa
tcher) localDispatcher).getTraceTopicName())       // @1
            || context.getMqTraceContext() == null) {
            return;
        }
        if (context.getSendResult() == null) {
            return;
        }

        if (context.getSendResult().getRegionId() == null
            || !context.getSendResult().isTraceOn()) {
            // if switch is false,skip it
            return;
        }

        TraceContext tuxeContext = (TraceContext) context.getMqTraceContext();
        TraceBean traceBean = tuxeContext.getTraceBeans().get(0);
                                                                    // @2
        int costTime = (int) ((System.currentTimeMillis() - tuxeContext.getTimeStamp()) /
tuxeContext.getTraceBeans().size());       // @3
        tuxeContext.setCostTime(costTime);
                                                                        // @
4
        if (context.getSendResult().getSendStatus().equals(SendStatus.SEND_OK)) {

            tuxeContext.setSuccess(true);
        } else {
            tuxeContext.setSuccess(false);
        }
        tuxeContext.setRegionId(context.getSendResult().getRegionId());

        traceBean.setMsgId(context.getSendResult().getMsgId());
        traceBean.setOffsetMsgId(context.getSendResult().getOffsetMsgId());
        traceBean.setStoreTime(tuxeContext.getTimeStamp() + costTime / 2);
        localDispatcher.append(tuxeContext);
                                                                    // @5
    }
```

代码@1：如果 topic 主题为消息轨迹的 Topic，直接返回。

代码@2：从 MqTraceContext 中获取跟踪的 TraceBean，虽然设计成 List 结构体，但在消息发送场景，这里的数据永远只有一条，及时是批量发送也不例外。

代码@3：获取消息发送到收到响应结果的耗时。

代码@4：设置 costTime(耗时)、success(是否发送成功)、regionId(发送到 broker 所在的分区)、msgId(消息 ID，全局唯一)、offsetMsgId(消息物理偏移量，如果是批量消息，则是最后一条消息的物理偏移量)、storeTime，这里使用的是(客户端发送时间 + 二分之一的耗时)来表示消息的存储时间，这里是一个估值。

代码@5：将需要跟踪的信息通过 TraceDispatcher 转发到 Broker 服务器。其代码如下：

```
public boolean append(final Object ctx) {
    boolean result = traceContextQueue.offer((TraceContext) ctx);
    if (!result) {
        log.info("buffer full" + discardCount.incrementAndGet() + " ,context is " + ctx);
    }
    return result;
}
```

这里一个非常关键的点是 offer 方法的使用，当队列无法容纳新的元素时会立即返回 false，并不会阻塞。

接下来将目光转向 TraceDispatcher 的实现。

3. TraceDispatcher 实现原理

TraceDispatcher，用于客户端消息轨迹数据转发到 Broker，其默认实现类：AsyncTraceDispatcher。

TraceDispatcher 构造函数

```
public AsyncTraceDispatcher(String traceTopicName, RPCHook rpcHook) throws MQ
ClientException {
```


```
        // queueSize is greater than or equal to the n power of 2 of value
        this.queueSize = 2048;
        this.batchSize = 100;
        this.maxMsgSize = 128000;
        this.discardCount = new AtomicLong(0L);
        this.traceContextQueue = new ArrayBlockingQueue<TraceContext>(1024);
        this.appenderQueue = new ArrayBlockingQueue<Runnable>(queueSize);
        if (!UtilAll.isBlank(traceTopicName)) {
            this.traceTopicName = traceTopicName;
        } else {
            this.traceTopicName = MixAll.RMQ_SYS_TRACE_TOPIC;
        }                           // @1
        this.traceExecuter = new ThreadPoolExecutor(// :
            10, //
            20, //
            1000 * 60, //
            TimeUnit.MILLISECONDS, //
            this.appenderQueue, //
            new ThreadFactoryImpl("MQTraceSendThread_"));
        traceProducer = getAndCreateTraceProducer(rpcHook);        // @2
    }
```

代码@1：初始化核心属性，该版本这些值都是“固化”的，用户无法修改。

- queueSize

  队列长度，默认为 2048，异步线程池能够积压的消息轨迹数量。

- batchSize

  一次向 Broker 批量发送的消息条数，默认为 100.

- maxMsgSize

  向 Broker 汇报消息轨迹时，消息体的总大小不能超过该值，默认为 128k。

- discardCount

  整个运行过程中，丢弃的消息轨迹数据，这里要说明一点的是，如果消息 TPS 发送过大，异步转发线程处理不过来时，会主动丢弃消息轨迹数据。

- traceContextQueue

traceContext 积压队列，客户端(消息发送、消息消费者)在收到处理结果后，将消息轨迹提交到噶队列中，则会立即返回。

- appenderQueue

提交到 Broker 线程池中队列。

- traceTopicName

用于接收消息轨迹的 Topic，默认为 RMQ_SYS_TRANS_HALF_TOPIC。

- traceExecuter

用于发送到 Broker 服务的异步线程池，核心线程数默认为 10，最大线程池为 20，队列堆积长度 2048，线程名称：MQTraceSendThread_。、

- traceProducer

发送消息轨迹的 Producer。

代码@2：调用 getAndCreateTraceProducer 方法创建用于发送消息轨迹的 Producer(消息发送者)，下面详细介绍一下其实现。

getAndCreateTraceProducer 详解

```
    private DefaultMQProducer getAndCreateTraceProducer(RPCHook rpcHook) {
            DefaultMQProducer traceProducerInstance = this.traceProducer;
            if (traceProducerInstance == null) {   //@1
                traceProducerInstance = new DefaultMQProducer(rpcHook);
                traceProducerInstance.setProducerGroup(TraceConstants.GROUP_NAM
E);
                traceProducerInstance.setSendMsgTimeout(5000);
                traceProducerInstance.setVipChannelEnabled(false);
                // The max size of message is 128K
                traceProducerInstance.setMaxMessageSize(maxMsgSize - 10 * 1000);
            }
            return traceProducerInstance;
        }
```

代码@1：如果还未建立发送者，则创建用于发送消息轨迹的消息发送者，其 GroupName 为：_INNER_TRACE_PRODUCER，消息发送超时时间 5s，最大允许发送消息大小 118K。

start

```
    public void start(String nameSrvAddr) throws MQClientException {
        if (isStarted.compareAndSet(false, true)) {        // @1
            traceProducer.setNamesrvAddr(nameSrvAddr);
            traceProducer.setInstanceName(TRACE_INSTANCE_NAME + "_" + nameSrv
Addr);
            traceProducer.start();
        }
        this.worker = new Thread(new AsyncRunnable(), "MQ-AsyncTraceDispatcher-Thr
ead-" + dispatcherId);     // @2
        this.worker.setDaemon(true);
        this.worker.start();

        this.registerShutDownHook();
    }
```

开始启动，其调用的时机为启动 DefaultMQProducer 时，如果启用跟踪消息轨迹，则调用之。

代码@1：如果用于发送消息轨迹的发送者没有启动，则设置 nameserver 地址，并启动着。

代码@2：启动一个线程，用于执行 AsyncRunnable 任务，接下来将重点介绍。

AsyncRunnable

```
    class AsyncRunnable implements Runnable {
             private boolean stopped;
     public void run() {
            while (!stopped) {
                List<TraceContext> contexts = new ArrayList<TraceContext>(batchSize);
```

```
        // @1
                for (int i = 0; i < batchSize; i++) {
                    TraceContext context = null;
                    try {
                        //get trace data element from blocking Queue — traceContextQue
ue
                        context = traceContextQueue.poll(5, TimeUnit.MILLISECONDS);
    // @2
                    } catch (InterruptedException e) {
                    }
                    if (context != null) {
                        contexts.add(context);
                    } else {
                        break;
                    }
                }
                if (contexts.size() > 0) {
                            :
                    AsyncAppenderRequest request = new AsyncAppenderRequest(conte
xts);  // @3
                    traceExecuter.submit(request);

                } else if (AsyncTraceDispatcher.this.stopped) {
                    this.stopped = true;
                }
            }
        }
    }
```

代码@1：构建待提交消息跟踪 Bean，每次最多发送 batchSize，默认为 100 条。

代码@2：从 traceContextQueue 中取出一个待提交的 TraceContext，设置超时时间为 5s，即如何该队列中没有待提交的 TraceContext，则最多等待 5s。

代码@3：向线程池中提交任务 AsyncAppenderRequest。

AsyncAppenderRequest#sendTraceData

```
    public void sendTraceData(List<TraceContext> contextList) {
        Map<String, List<TraceTransferBean>> transBeanMap = new HashMap<String, Li
st<TraceTransferBean>>();
        for (TraceContext context : contextList) {          //@1
            if (context.getTraceBeans().isEmpty()) {
                continue;
            }
            // Topic value corresponding to original message entity content
            String topic = context.getTraceBeans().get(0).getTopic();       // @2
            // Use  original message entity's topic as key
            String key = topic;
            List<TraceTransferBean> transBeanList = transBeanMap.get(key);
            if (transBeanList == null) {
                transBeanList = new ArrayList<TraceTransferBean>();
                transBeanMap.put(key, transBeanList);
            }
            TraceTransferBean traceData = TraceDataEncoder.encoderFromContextBean
(context);     // @3
            transBeanList.add(traceData);
        }
        for (Map.Entry<String, List<TraceTransferBean>> entry : transBeanMap.entrySet())
 {         // @4
            flushData(entry.getValue());
        }
    }
```

代码@1：遍历收集的消息轨迹数据。

代码@2：获取存储消息轨迹的 Topic。

代码@3：对 TraceContext 进行编码，这里是消息轨迹的传输数据，稍后对其详细看一下，了解其上传的格式。

代码@4：将编码后的数据发送到 Broker 服务器。

二、消息轨迹格式

TraceDataEncoder#encoderFromContextBean

根据消息轨迹跟踪类型，其格式会有一些不一样，下面分别来介绍其合适。

1）PUB(消息发送)

```
case Pub: {
    TraceBean bean = ctx.getTraceBeans().get(0);
    //append the content of context and traceBean to transferBean's TransData
    sb.append(ctx.getTraceType()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(ctx.getTimeStamp()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(ctx.getRegionId()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(ctx.getGroupName()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getTopic()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getMsgId()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getTags()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getKeys()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getStoreHost()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getBodyLength()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(ctx.getCostTime()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getMsgType().ordinal()).append(TraceConstants.CONTENT_SPLIT
OR)//
      .append(bean.getOffsetMsgId()).append(TraceConstants.CONTENT_SPLITOR)//
     .append(ctx.isSuccess()).append(TraceConstants.FIELD_SPLITOR);
}
```

消息轨迹数据的协议使用字符串拼接，字段的分隔符号为 1，整个数据以 2 结尾，感觉这个设计还是有点“不可思议”，为什么不直接使用 json 协议呢?

2）SubBefore(消息消费之前)

```
for (TraceBean bean : ctx.getTraceBeans()) {
    sb.append(ctx.getTraceType()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(ctx.getTimeStamp()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(ctx.getRegionId()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(ctx.getGroupName()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(ctx.getRequestId()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getMsgId()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getRetryTimes()).append(TraceConstants.CONTENT_SPLITOR)//
      .append(bean.getKeys()).append(TraceConstants.FIELD_SPLITOR);//
    } }
```

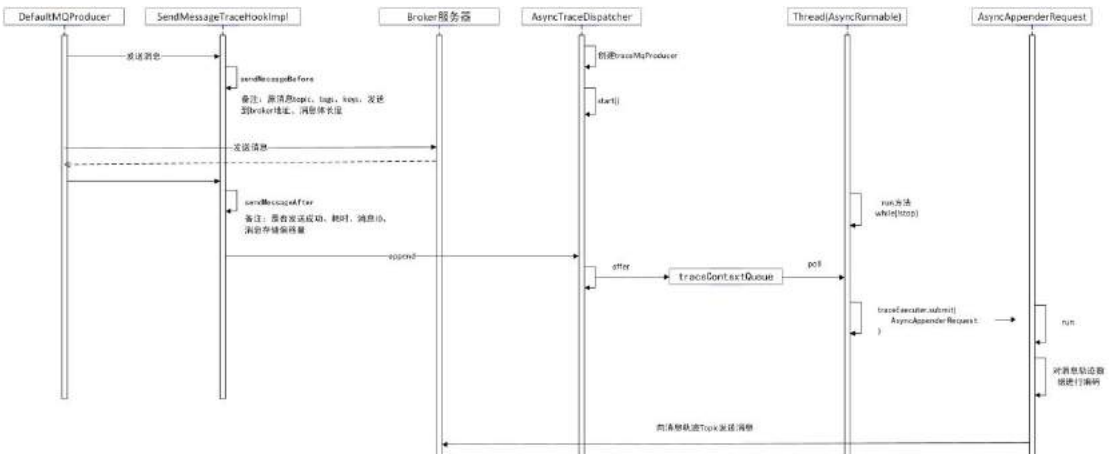
轨迹就是按照上述顺序拼接而成，各个字段使用 1 分隔，每一条记录使用 2 结尾。

3）SubAfter（消息消费后）

```
case SubAfter: {
    for (TraceBean bean : ctx.getTraceBeans()) {
        sb.append(ctx.getTraceType()).append(TraceConstants.CONTENT_SPLITOR)//
          .append(ctx.getRequestId()).append(TraceConstants.CONTENT_SPLITOR)//
          .append(bean.getMsgId()).append(TraceConstants.CONTENT_SPLITOR)//
          .append(ctx.getCostTime()).append(TraceConstants.CONTENT_SPLITOR)//
          .append(ctx.isSuccess()).append(TraceConstants.CONTENT_SPLITOR)//
          .append(bean.getKeys()).append(TraceConstants.CONTENT_SPLITOR)//
          .append(ctx.getContextCode()).append(TraceConstants.FIELD_SPLITOR);
        }
    }
}
```

格式编码一样，就不重复多说。

经过上面的源码跟踪，消息发送端的消息轨迹跟踪流程、消息轨迹数据编码协议就清晰了，接下来我们使用一张序列图来结束本部分的讲解。



其实行文至此，只关注了消息发送的消息轨迹跟踪，消息消费的轨迹跟踪又是如何呢？其实现原理其实是一样的，就是在消息消费前后执行特定的钩子函数，其实现类为 ConsumeMessageTraceHookImpl，由于其实现与消息发送的思路类似，故就不详细介绍了。

三、消息轨迹数据如何存储

其实从上面的分析，我们已经得知，RocketMQ 的消息轨迹数据存储在到 Broker 上，那消息轨迹的主题名如何指定？其路由信息又怎么分配才好呢？是每台 Broker 上都创建还是只在其中某台上创建呢？RocketMQ 支持系统默认与自定义消息轨迹的主题。

1. 使用系统默认的主题名称

RocketMQ 默认的消息轨迹主题为：RMQ_SYS_TRACE_TOPIC，那该 Topic 需要手工创建吗？其路由信息呢？

```
{
    if (this.brokerController.getBrokerConfig().isTraceTopicEnable()) {     // @1
        String topic = this.brokerController.getBrokerConfig().getMsgTraceTopicName
();
        TopicConfig topicConfig = new TopicConfig(topic);
        this.systemTopicList.add(topic);
        topicConfig.setReadQueueNums(1);
 // @2
        topicConfig.setWriteQueueNums(1);
        this.topicConfigTable.put(topicConfig.getTopicName(), topicConfig);
    }
}
```

上述代码出自 TopicConfigManager 的构造函数，在 Broker 启动的时候会创建 topicConfigManager 对象，用来管理 topic 的路由信息。

代码@1：如果 Broker 开启了消息轨迹跟踪(traceTopicEnable=true)时，会自动创建默认消息轨迹的 topic 路由信息，注意其读写队列数为 1。

2. 用户自定义消息轨迹主题

在创建消息发送者、消息消费者时，可以显示的指定消息轨迹的 Topic，例如：

```
public DefaultMQProducer(final String producerGroup, RPCHook rpcHook, boolean enableMsgTrace,final String customizedTraceTopic)

public DefaultMQPushConsumer(final String consumerGroup, RPCHook rpcHook,
        AllocateMessageQueueStrategy allocateMessageQueueStrategy, boolean enableMsgTrace, final String customizedTraceTopic)
```

通过 customizedTraceTopic 来指定消息轨迹 Topic。

温馨提示：通常在生产环境上，将不会开启自动创建主题，故需要 RocketMQ 运维管理人员提前创建好 Topic。

好了，本文就介绍到这里了，本文详细介绍了 RocktMQ 消息轨迹的实现原理，下一篇，我们将进入到多副本的学习中。

2.4 RocketMQ 多副本前置篇：初探 raft 协议

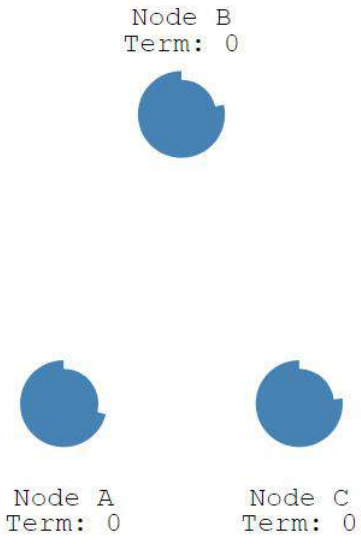
原来 raft 协议这么简单。

Raft 协议解决分布式领域解决一致性的又一著名协议，主要包含 Leader 选举、日志复制两个部分。

本文根据 raft 官方给出的 raft 动画进行学习，其动画展示地址：
http://thesecretlivesofdata.com/raft/

一、Leader 选举

1. 一轮投票中，只有一个节点发起投票的情况



Raft 协议中节点有 3 种状态（角色）：

- Follower

跟随者。

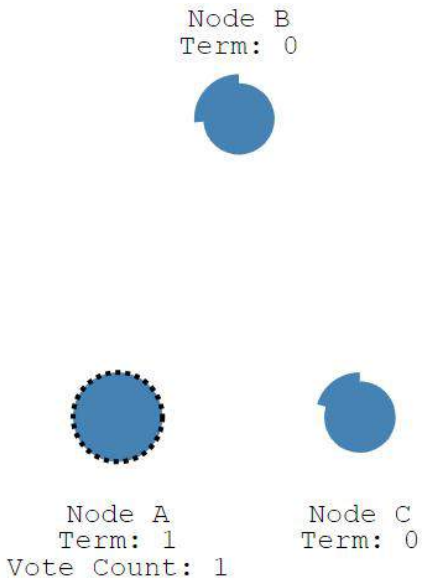
- Candidate

  候选者。

- Leader

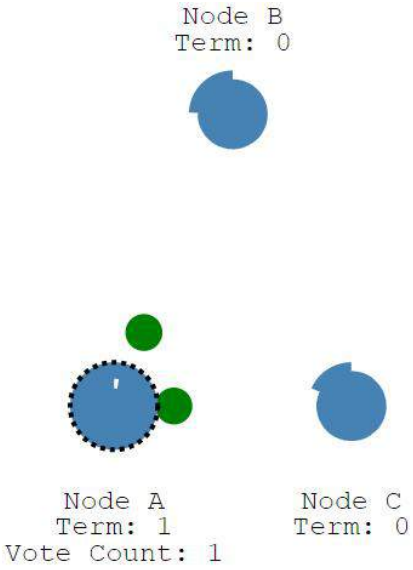
  领导者(Leader)，通常我们所说的的主节点。

首先 3 个节点初始状态为 Follower，每个节点会有一个超时时间(定时器)，其时间设置为 150ms~300ms 之间的随机值。当定时器到期后，节点状态从 Follower 变成 Candidate，如下图所示：

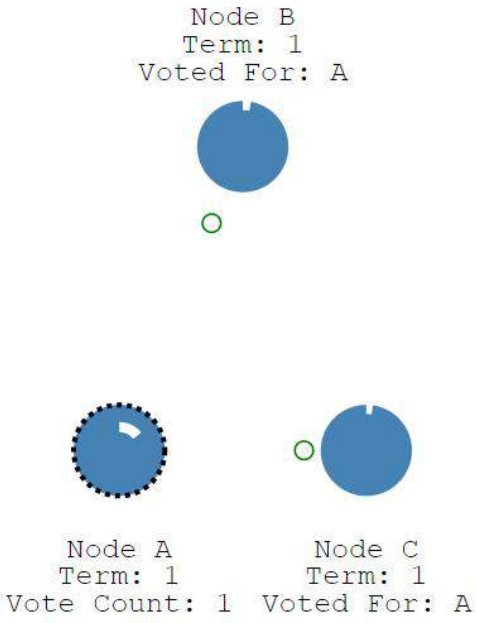


通常情况下，三个节点中会有一个节点定时器率先到期，节点状态变为 Candidate，候选者状态下的节点会发起选举投票。我们先来考虑只有一个节点变为 Candidate 时是如何进行选主的。

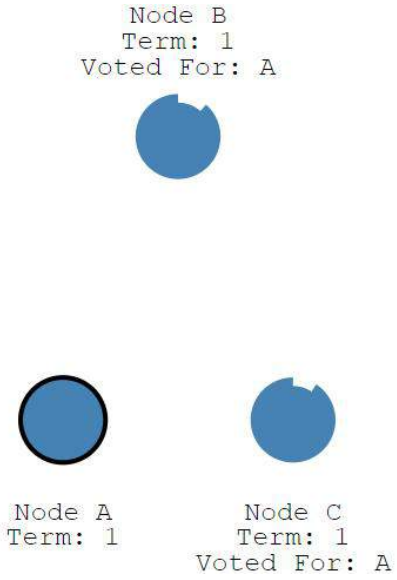
当节点状态为 Candidate，将发起一轮投票，由于是第一轮投票，设置本轮投票轮次为 1，并首先为自己投上一票，正如上图所示的 NodeA 节点，Team 为 1，Vote Count 为 1.



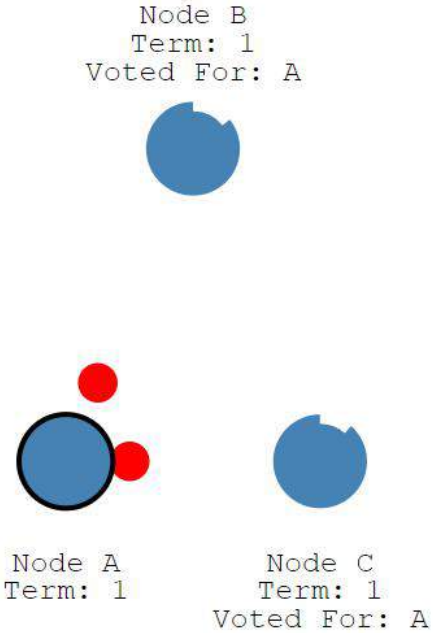
当一个节点的定时器超时后，首先为自己投上一票，然后向该组内其他的节点发起投票(用拉票更加合适)，发送投票请求。



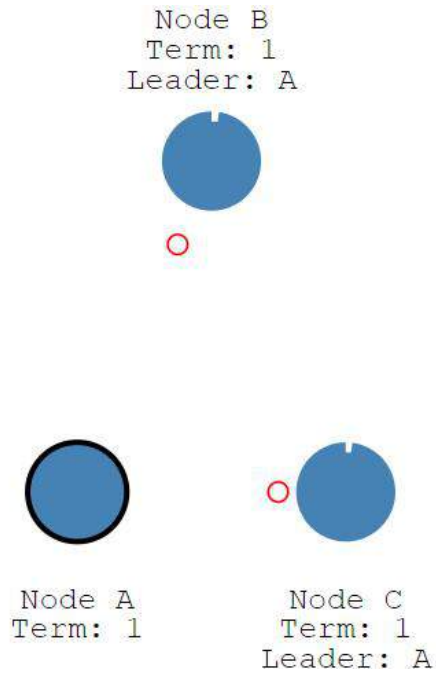
当集群内的节点收到投票请求外，如果本轮未进行过投票，则赞同，否则反对，然后将结果返回，并重置定时器。



当节点 A 收到的赞同票大于一半时，则升级为该集群的 Leader，然后定时向集群内的其他节点发送心跳，已确定自己的领导地位，正如下图所示。

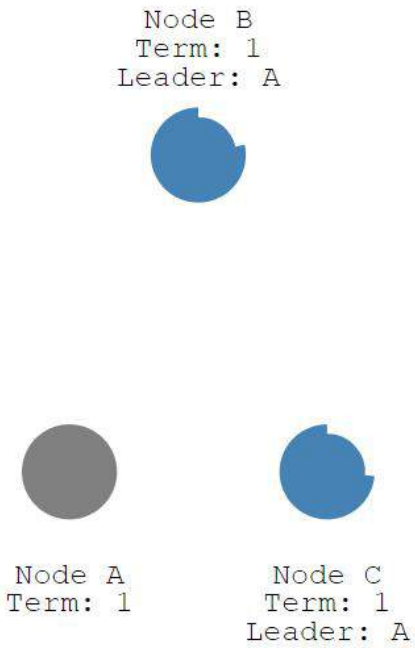


Node A，集群中的 Leader 正在向其他节点发送心跳包。

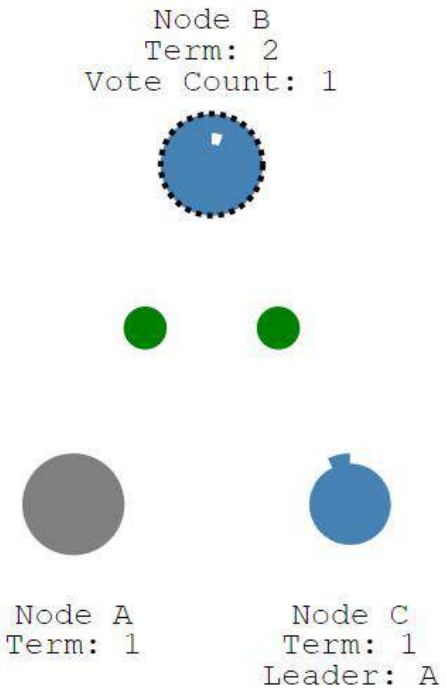


节点在收到 Leader 的心跳包后，返回响应结果，并重置自身的定时器，如果 Flower 状态的节点在记时时间超时内没有收到 Leader 的心跳包，就会从 Flower 节点变成 Candidate,该节点就会发起下一轮投票。

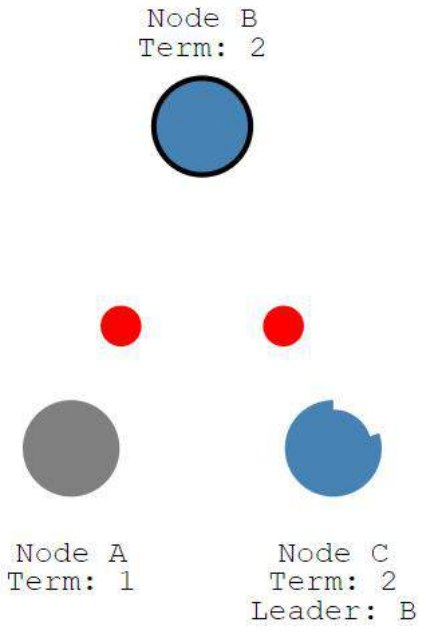
例如 NodeA 节点宕机，停止向它的从发送心跳，我们来看一下集群如何重新选主。



如果主节点宕机，则停止向集群内的节点发送心跳包。随着定时器的到期，节点 B 的先节点 C 变成 Candidate，则向集群内的节点发起投票，如下图所示。



节点 B，首先将投票轮次设置为 2，然后首先为自己投上一篇，然后向其他节点发起投票请求。



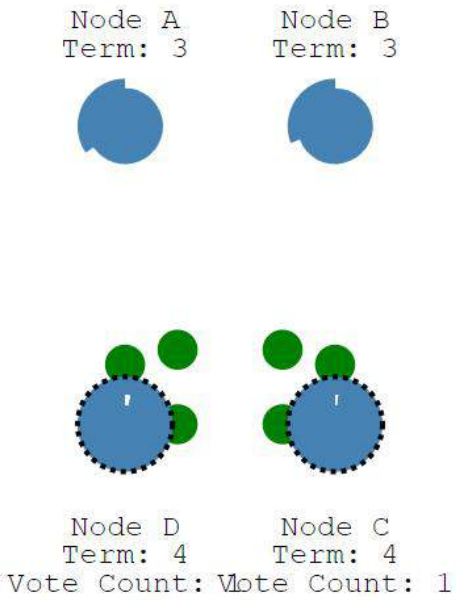
节点 C 收到请求，由于其投票轮次大于自己的投票轮次，并该轮次并未投票，投出赞成票并返回结果，然后重置计时器。节点 B 将顺理成章的成为新的 Leader 并定时发送心跳包。

3 个节点的选主就介绍到这里了，也许有网友会说，虽然各个节点的定时器是随机的，但也有可能同一时间，或一个节点在未收到另一个节点发起的投票请求之前变成 Candidate，即在一轮投票过程中，有大于 1 个的节点状态都是 Candidate，那该如何选主呢?

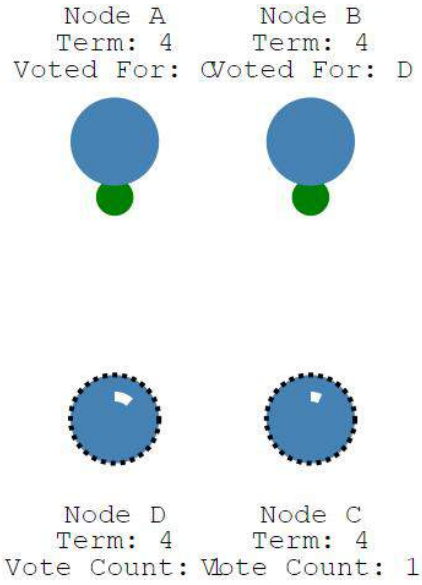
下面以 4 个节点的集群为例，来阐述上述这种情况情况下，如何进行选主。

2. 一轮投票中，超过一个节点发起投票的情况

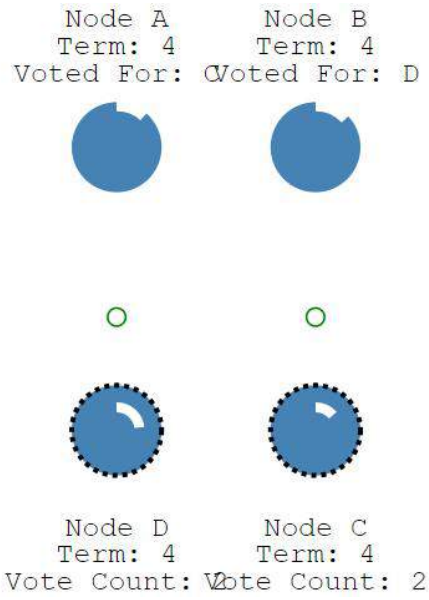
首先同时有两个节点进入 Candidate 状态，并开始新的一轮投票，当前投票编号为 4，首先先为自己投上一票，然后向集群中的其他节点发起投票，如下图所示：



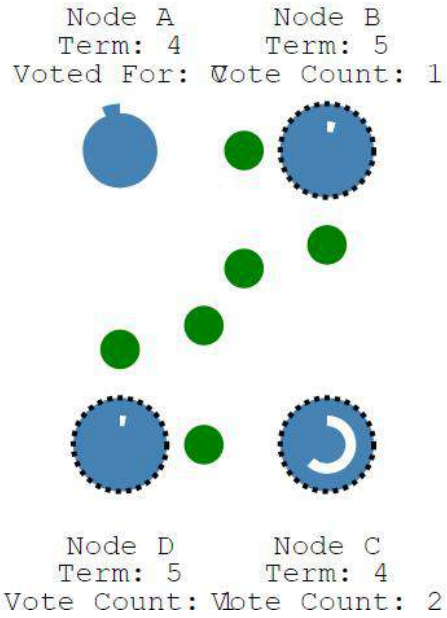
然后各个节点收到投票请求，如下所示，进行投票：



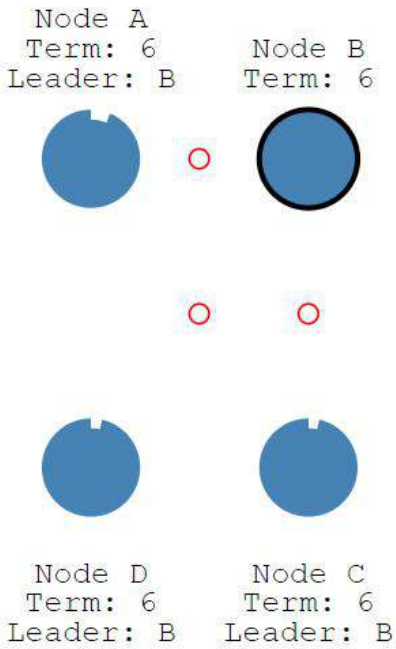
首先节点 C、D 在收到 D、C 节点的投票请求时，都会返回不同意，因为在本轮投票中，已经各自为自己投了一票，按照上图，节点 A 同意 C 节点、节点 B 同意 D 节点，那此时 C、D 都只获的两票，当然如果 A,B 都认为 C 或 D 成为主节点，则选择就可以结束了，上图显示，C、D 都只获的 2 票，未超过半数，无法成为主节点，那接下来会发生什么呢？请看下图：



此时 A,B,C,D 的定时器各自在倒计时，当节点成为 Candidate 时，或自身状态本身是 Candidate 并且定时器触发后，发起一轮新的投票，图中是节点 B、节点 D 同时发起了新的一轮投票。



投票结果如下：节点 A,节点 C 同意节点 B 成为 leader，但由于 BD 都发起了第 5 轮投票，最终的投票轮次更新为 6，如图所示：



关于 Raft 协议的选主就介绍到这里了，接下来我们来思考一下，如果自己 Raft 协议，至少要考虑哪些问题，为下一篇源码阅读 Dleger(RocketMQ 多副本)模块提供一些思路。

3. 思考如何实现 Raft 选主

- 节点状态

需要引入 3 中节点状态：Follower(跟随者)、Candidate(候选者)，投票的触发点，Leader(主节点)。

- 选择定时器

Follower、Candidate 两个状态时，需要维护一个定时器，每次定时时间从 150ms-300ms 直接进行随机，即每个节点的定时过期不一样，Follower 状态时，定时器到点后，触发一轮投票。节点在收到投票请求、Leader 的心跳请求并作出响应后，需要重置定时器。

- 投票轮次 Team

Candidate 状态的节点，每发起一轮投票，Team 加一。

- 投票机制

每一轮一个节点只能为一个节点投赞成票，例如节点 A 中维护的轮次为 3，并且已经为节点 B 投了赞成票，如果收到其他节点，投票轮次为 3，则会投反对票，如果收到轮次为 4 的节点，是又可以投赞成票的。

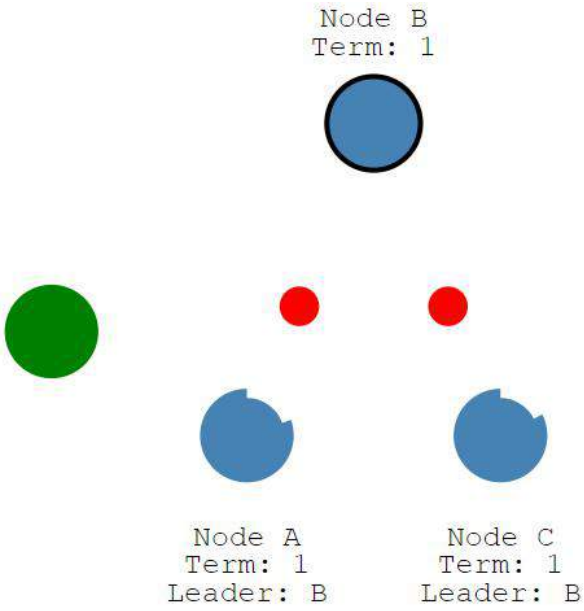
- 成为 Leader 的条件

必须得到集群中初始数量的大多数，例如如果集群中有 3 台集群，则必须得到两票，如果其中一台服务器宕机，剩下的两个节点，还能进行选主吗？答案是可以的，因为可以得到 2 票，超过初始集群中 3 的一半，所以通常集群中的机器各位尽量为计数，因为 4 台的可用性与 3 台的一样。

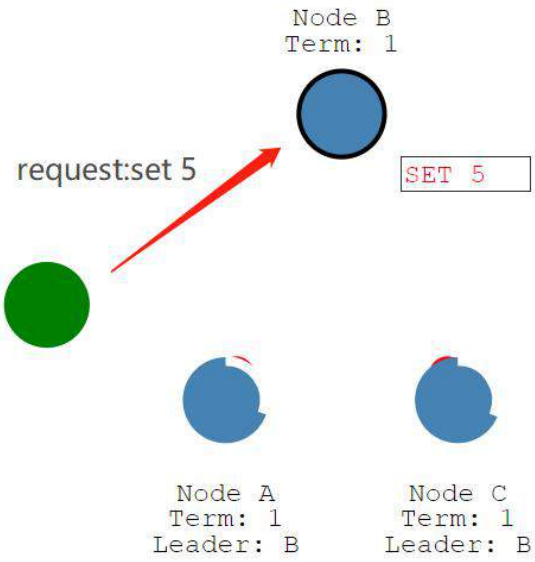
温馨提示：上述结论只是我的一些思考，我们可以带着上述思考，进入到 Dleger 的学习中，下一篇将从源码分析的角度来学习大神是如何实现 Raft 协议的 Leader 选主的，让我们一起期待吧。

二、日志复制

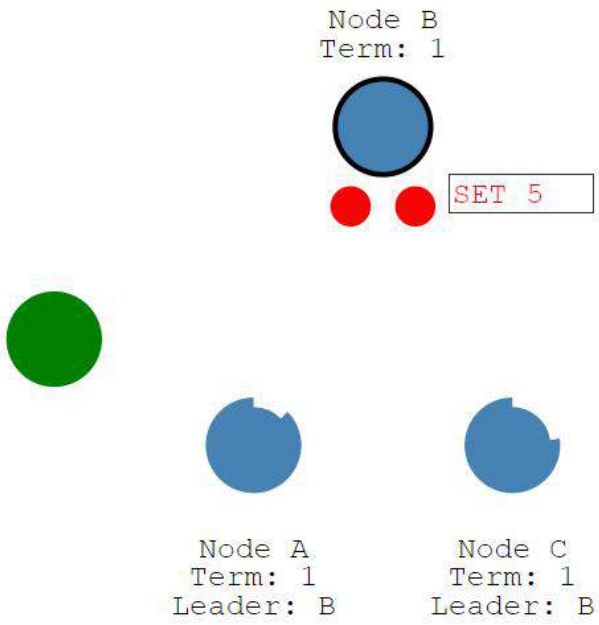
完成集群内的选主工作后，客户端向主节点发送请求，由主节点负责数据的复制，使集群内的数据保持一致性，初始状态如下图所示：



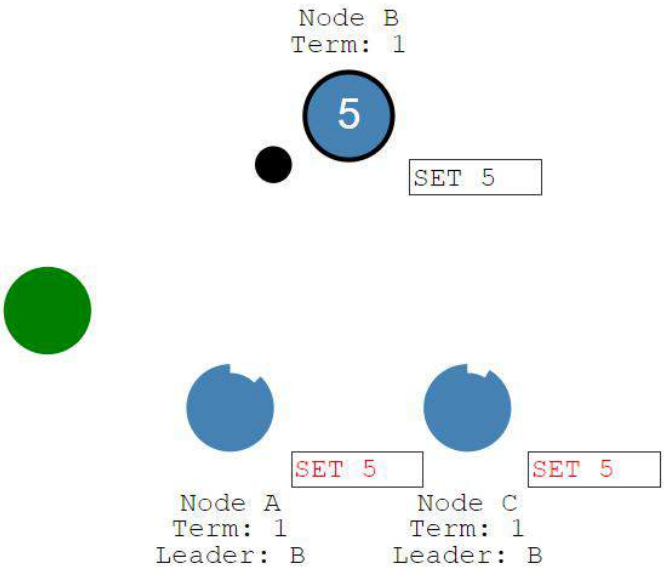
客户端向主节点发起请求，例如 set 5，将数据更新为 5，如下图所示：



主节点收到客户端请求后，将数据追加到 Leader 的日志中(但未提交)，然后在下一个心跳包中将日志转发到集群内从节点，如下图所示：



从节点收到 Leader 的日志后，追加到从节点的日志文件中，并返回确认 ACK。Leader 收到从节点的确认信息后，向客户端发送确认信息。



上述的日志复制比较简单，是由于只考虑正常的情况，如果中间发生异常，该如何保证数据一致性呢?

1. 如果 Leader 节点向从节点广播日志时，其中某个从节点发送故障宕机，该如何处理呢?
2. 日志在什么环节进行提交呢? Leader 节点在收到客户端的数据变更请求后，首先追加到主节点的日志文件中，然后广播到从节点，从节点收到日志信息，是提交日志后返回 ACK，还是什么时候提交呢?
3. 日志如何保证唯一。
4. 如何处理网络出现分区。

我相信读者朋友肯定还有更多的疑问，本文不打算来回答上述疑问，而是带着这些问题进入到 RocketMQ 多副本的学习中，通过源码分析 RocketMQ DLedger 的实现后，再来重新总结 raft 协议。

2.5 源码分析 RocketMQ 多副本之 Leader 选主

本文关键字：RocketMQ、 多副本、 DLedger、 leader 选举、 主从切换

本文将按照《RocketMQ 多副本前置篇：初探 raft 协议》的思路来学习 RocketMQ 选主逻辑。首先先回顾一下关于 Leader 的一些思考：

- 节点状态

需要引入 3 中节点状态：Follower(跟随者)、Candidate(候选者)，投票的触发点，Leader(主节点)。

- 选举计时器

Follower、Candidate 两个状态时，需要维护一个定时器，每次定时时间从 150ms-300ms 直接进行随机，即每个节点的定时过期不一样，Follower 状态时，定时器到点后，触发一轮投票。节点在收到投票请求、Leader 的心跳请求并作出响应后，需要重置定时器。

- 投票轮次 Team

Candidate 状态的节点，每发起一轮投票，Team 加一。

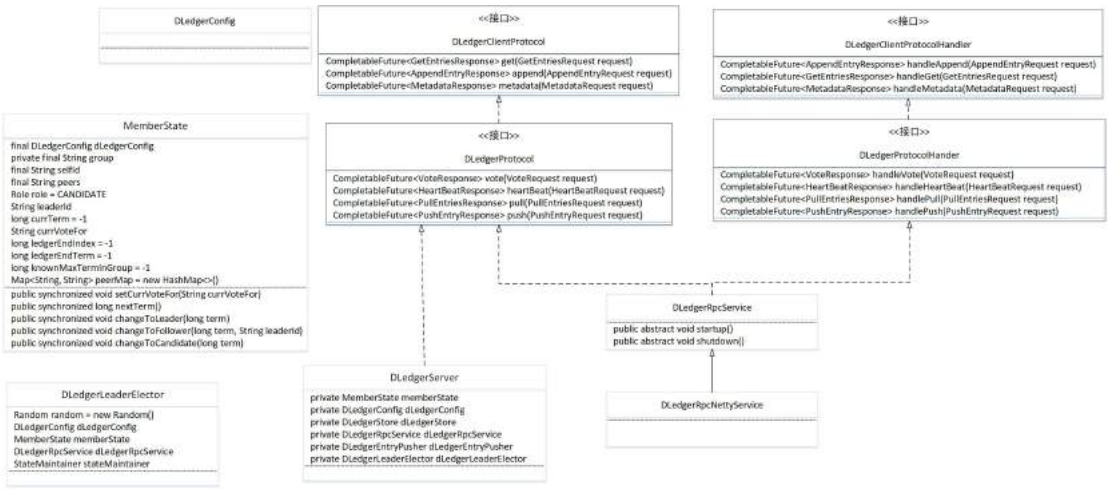
- 投票机制

每一轮一个节点只能为一个节点投赞成票，例如节点 A 中维护的轮次为 3，并且已经为节点 B 投了赞成票，如果收到其他节点，投票轮次为 3，则会投反对票，如果收到轮次为 4 的节点，是又可以投赞成票的。

- 成为 Leader 的条件

必须得到集群中初始数量的大多数，例如如果集群中有 3 台集群，则必须得到两票，如果其中一台服务器宕机，剩下的两个节点，还能进行选主吗？答案是可以的，因为可以得到 2 票，超过初始集群中 3 的一半，所以通常集群中的机器各位尽量为计数，因为 4 台的可用性与 3 台的一样。

一、DLedger 关于选主的核心类图



1. DLedgerConfig

多副本模块相关的配置信息，例如集群节点信息。

2. MemberState

节点状态机，即 raft 协议中的 follower、candidate、leader 三种状态的状态机实现。

3. raft 协议相关

DLedgerClientProtocol

DLedger 客户端协议，主要定义如下三个方法，在后面的日志复制部分会重点阐述。

- CompletableFuture<GetEntriesResponse> get(GetEntriesRequest request)
  客户端从服务器获取日志条目（获取数据）

- CompletableFuture<AppendEntryResponse> append(AppendEntryRequest request)
  客户端向服务器追加日志（存储数据）

- CompletableFuture<MetadataResponse> metadata(MetadataRequest request)
  获取元数据。

DLedgerProtocol

DLedger 服务端协议，主要定义如下四个方法。

- CompletableFuture<VoteResponse> vote(VoteRequest request)
  发起投票请求。

- CompletableFuture<HeartBeatResponse> heartBeat(HeartBeatRequest request)
  Leader 向从节点发送心跳包。

- CompletableFuture<PullEntriesResponse> pull(PullEntriesRequest request)
  拉取日志条目，在日志复制部分会详细介绍。

- CompletableFuture<PushEntryResponse> push(PushEntryRequest request)
  推送日志条件，在日志复制部分会详细介绍。

协议处理 Handler

DLedgerClientProtocolHandler、DLedgerProtocolHander 协议处理器。

4. DLedgerRpcService

DLedger Server(节点)之间的网络通信，默认基于 Netty 实现，其实现类为：DLedgerRpcNettyService。

5. DLedgerLeaderElector

Leader 选举实现器。

6. DLedgerServer

Dledger Server，Dledger 节点的封装类。

接下来将从 DLedgerLeaderElector 开始剖析 DLedger 是如何实现 Leader 选举的。（基于 raft 协议）。

二、源码分析 Leader 选举

1. DLedgerLeaderElector 类图

我们先一一来介绍其属性的含义：

- Random random
  随机数生成器，对应 raft 协议中选举超时时间是一随机数。

- DLedgerConfig dLedgerConfig
  配置参数。

- MemberState memberState
  节点状态机。

- DLedgerRpcService dLedgerRpcService
  rpc 服务，实现向集群内的节点发送心跳包、投票的 RPC 实现。

- long lastLeaderHeartBeatTime
  上次收到心跳包的时间戳。

- long lastSendHeartBeatTime
  上次发送心跳包的时间戳。

- long lastSuccHeartBeatTime
  上次成功收到心跳包的时间戳。

- int heartBeatTimeIntervalMs
  一个心跳包的周期，默认为 2s。

- int maxHeartBeatLeak
  允许最大的 N 个心跳周期内未收到心跳包，状态为 Follower 的节点只有超过 maxHeartBeatLeak * heartBeatTimeIntervalMs 的时间内未收到主节点的心跳包，才会重新进入 Candidate 状态，重新下一轮的选举。

- long nextTimeToRequestVote
  发送下一个心跳包的时间戳。

- boolean needIncreaseTermImmediately
  是否应该立即发起投票。

- int minVoteIntervalMs
  最小的发送投票间隔时间，默认为 300ms。

- int maxVoteIntervalMs
  最大的发送投票的间隔，默认为 1000ms。

- List<RoleChangeHandler> roleChangeHandlers
  注册的节点状态处理器，通过 addRoleChangeHandler 方法添加。

- long lastVoteCost
  上一次投票的开销。

- StateMaintainer stateMaintainer
  状态机管理器。

2. 启动选举状态管理器

通过 DLedgerLeaderElector 的 startup 方法启动状态管理机，代码如下：

```
DLedgerLeaderElector#startup
public void startup() {
    stateMaintainer.start();    // @1
    for (RoleChangeHandler roleChangeHandler : roleChangeHandlers) {    // @2
        roleChangeHandler.startup();
    }
}
```

代码@1：启动状态维护管理器。

代码@2：遍历状态改变监听器并启动它，可通过 DLedgerLeaderElector 的 addRoleChangeHandler 方法增加状态变化监听器。

其中的是启动状态管理器线程，其 run 方法实现：

```
public void run() {
    while (running.get()) {
        try {
            doWork();
        } catch (Throwable t) {
            if (logger != null) {
                logger.error("Unexpected Error in running {} ", getName(), t);
            }
        }
    }
    latch.countDown();
}
```

从上面来看，主要是循环调用 doWork 方法，接下来重点看其 doWork 的实现：

```
public void doWork() {
    try {
        if (DLedgerLeaderElector.this.dLedgerConfig.isEnableLeaderElector()) {    // @1
            DLedgerLeaderElector.this.refreshIntervals(dLedgerConfig);
// @2
            DLedgerLeaderElector.this.maintainState();
      // @3
        }
        sleep(10);
                                // @4
    } catch (Throwable t) {
        DLedgerLeaderElector.logger.error("Error in heartbeat", t);
    }
}
```

代码@1：如果该节点参与 Leader 选举，则首先调用@2 重置定时器，然后维护状态，进行投票选举等流程，是接下来重点需要剖析的。

代码@4：没执行一次选主，休息 10ms。

```
DLedgerLeaderElector#maintainState
private void maintainState() throws Exception {
    if (memberState.isLeader()) {
        maintainAsLeader();
    } else if (memberState.isFollower()) {
        maintainAsFollower();
    } else {
        maintainAsCandidate();
    }
}
```

根据当前的状态机状态，执行对应的操作，从 raft 协议中可知，总共存在 3 种状态：

- leader

领导者，主节点，该状态下，需要定时向从节点发送心跳包，用来传播数据、确保其领导地位。

- follower

从节点，该状态下，会开启定时器，尝试进入到 candidate 状态，以便发起投票选举，同时一旦收到主节点的心跳包，则重置定时器。

- candidate

候选者，该状态下的节点会发起投票，尝试选择自己为主节点，选举成功后，不会存在该状态下的节点。

我们在继续往下看之前，需要知道 memberState 的初始值是什么？我们追溯到创建 MemberState 的地方，发现其初始状态为 CANDIDATE。那我们接下从 maintainAsCandidate 方法开始跟进。

温馨提示：在 raft 协议中，节点的状态默认为 follower，DLedger 的实现从 candidate 开始，一开始，集群内的所有节点都会尝试发起投票，这样第一轮要达成选举几乎不太可能。

3. 选举状态机状态流转

整个状态机的驱动，由线程反复执行 maintainState 方法。下面重点来分析其状态的驱动。

maintainAsCandidate 方法

```
DLedgerLeaderElector#maintainAsCandidate
if (System.currentTimeMillis() < nextTimeToRequestVote && !needIncreaseTermImme
diately) {
    return;
}
long term;
long ledgerEndTerm;
long ledgerEndIndex;
```

Step1：首先先介绍几个变量的含义：

- nextTimeToRequestVote

下一次发发起的投票的时间，如果当前时间小于该值，说明计时器未过期，此时无需发起投票。

- needIncreaseTermImmediately

是否应该立即发起投票。如果为 true，则忽略计时器，该值默认为 false，当收到从主节点的心跳包并且当前状态机的轮次大于主节点的轮次，说明集群中 Leader 的投票轮次小于从几点的轮次，应该立即发起新的投票。

- term

投票轮次。

- ledgerEndTerm

Leader 节点当前的投票轮次。

- ledgerEndIndex

  当前日志的最大序列，即下一条日志的开始 index，在日志复制部分会详细介绍。

```
DLedgerLeaderElector#maintainAsCandidate
synchronized (memberState) {
    if (!memberState.isCandidate()) {
        return;
    }
    if (lastParseResult == VoteResponse.ParseResult.WAIT_TO_VOTE_NEXT || need
IncreaseTermImmediately) {
        long prevTerm = memberState.currTerm();
        term = memberState.nextTerm();
        logger.info("{}_[INCREASE_TERM] from {} to {}", memberState.getSelfId(), pre
vTerm, term);
        lastParseResult = VoteResponse.ParseResult.WAIT_TO_REVOTE;
    } else {
        term = memberState.currTerm();
    }
    ledgerEndIndex = memberState.getLedgerEndIndex();
    ledgerEndTerm = memberState.getLedgerEndTerm();
}
```

Step2：初始化 team、ledgerEndIndex 、ledgerEndTerm 属性，其实现关键点如下：

- 如果上一次的投票结果为待下一次投票或应该立即开启投票，并且根据当前状态机获取下一轮的投票轮次，稍后会着重讲解一下状态机轮次的维护机制。
- 如果上一次的投票结果不是 WAIT_TO_VOTE_NEXT(等待下一轮投票)，则投票轮次依然为状态机内部维护的轮次。

```
DLedgerLeaderElector#maintainAsCandidate
if (needIncreaseTermImmediately) {
    nextTimeToRequestVote = getNextTimeToRequestVote();
    needIncreaseTermImmediately = false;
    return;
}
```

Step3：如果 needIncreaseTermImmediately 为 true，则重置该标记位为 false，并重新设置下一次投票超时时间，其实现代码如下：

```
    private long getNextTimeToRequestVote() {
        return System.currentTimeMillis() + lastVoteCost + minVoteIntervalMs + random.nextInt(maxVoteIntervalMs - minVoteIntervalMs);
    }
```

下一次倒计时：当前时间戳 + 上次投票的开销 + 最小投票间隔(300ms) + （1000- 300 ）之间的随机值。

```
    final List<CompletableFuture<VoteResponse>> quorumVoteResponses = voteForQuorumResponses(term, ledgerEndTerm, ledgerEndIndex);
```

Step4：向集群内的其他节点发起投票请，并返回投票结果列表，稍后会重点分析其投票过程。可以预见，接下来就是根据各投票结果进行仲裁。

```
    final AtomicLong knownMaxTermInGroup = new AtomicLong(-1);
    final AtomicInteger allNum = new AtomicInteger(0);
    final AtomicInteger validNum = new AtomicInteger(0);
    final AtomicInteger acceptedNum = new AtomicInteger(0);
    final AtomicInteger notReadyTermNum = new AtomicInteger(0);
    final AtomicInteger biggerLedgerNum = new AtomicInteger(0);
    final AtomicBoolean alreadyHasLeader = new AtomicBoolean(false);
```

Step5：在进行投票结果仲裁之前，先来介绍几个局部变量的含义：

- knownMaxTermInGroup
  已知的最大投票轮次。

- allNum
  所有投票票数。

- validNum
  有效投票数。

- acceptedNum

  获得的投票数。

  notReadyTermNum

未准备投票的节点数量，如果对端节点的投票轮次小于发起投票的轮次，则认为对端未准备好，对端节点使用本次的轮次进入 Candidate 状态。

- biggerLedgerNum

  发起投票的节点的 ledgerEndTerm 小于对端节点的个数。

- alreadyHasLeader

  是否已经存在 Leader。

```
for (CompletableFuture<VoteResponse> future : quorumVoteResponses) {
    // 省略部分代码
}
Step5：遍历投票结果，收集投票结果，接下来重点看其内部实现。

if (x.getVoteResult() != VoteResponse.RESULT.UNKNOWN) {
    validNum.incrementAndGet();
}
```

Step6：如果投票结果不是 UNKNOW，则有效投票数量增 1。

```
synchronized (knownMaxTermInGroup) {
    switch (x.getVoteResult()) {
        case ACCEPT:
            acceptedNum.incrementAndGet();
            break;
        case REJECT_ALREADY_VOTED:
            break;
        case REJECT_ALREADY_HAS_LEADER:
            alreadyHasLeader.compareAndSet(false, true);
            break;
        case REJECT_TERM_SMALL_THAN_LEDGER:
        case REJECT_EXPIRED_VOTE_TERM:
            if (x.getTerm() > knownMaxTermInGroup.get()) {
```

```
                    knownMaxTermInGroup.set(x.getTerm());
                }
                break;
            case REJECT_EXPIRED_LEDGER_TERM:
            case REJECT_SMALL_LEDGER_END_INDEX:
                biggerLedgerNum.incrementAndGet();
                break;
            case REJECT_TERM_NOT_READY:
                notReadyTermNum.incrementAndGet();
                break;
            default:
                break;
        }
    }
```

Step7：统计投票结构，几个关键点如下：

- ACCEPT

赞成票，acceptedNum 加一，只有得到的赞成票超过集群节点数量的一半才能成为 Leader。

- REJECT_ALREADY_VOTED

拒绝票，原因是已经投了其他节点的票。

- REJECT_ALREADY_HAS_LEADER

拒绝票，原因是因为集群中已经存在 Leaer 了。alreadyHasLeader 设置为 true，无需在判断其他投票结果了，结束本轮投票。

- REJECT_TERM_SMALL_THAN_LEDGER

拒绝票，如果自己维护的 term 小于远端维护的 ledgerEndTerm，则返回该结果，如果对端的 team 大于自己的 team，需要记录对端最大的投票轮次，以便更新自己的投票轮次。

- REJECT_EXPIRED_VOTE_TERM

拒绝票，如果自己维护的 term 小于远端维护的 term，更新自己维护的投票轮次。

- REJECT_EXPIRED_LEDGER_TERM

拒绝票，如果自己维护的 ledgerTerm 小于对端维护的 ledgerTerm，则返回该结果。如果是此种情况，增加计数器 biggerLedgerNum 的值。

- REJECT_SMALL_LEDGER_END_INDEX

拒绝票，如果对端的 ledgerTeam 与自己维护的 ledgerTeam 相等，但是自己维护的 dedgerEndIndex 小于对端维护的值，返回该值，增加 biggerLedgerNum 计数器的值。

- REJECT_TERM_NOT_READY

拒绝票，对端的投票轮次小于自己的 team，则认为对端还未准备好投票，对端使用自己的投票轮次，是自己进入到 Candidate 状态。

```
    try {
        voteLatch.await(3000 + random.nextInt(maxVoteIntervalMs), TimeUnit.MILLISECO
NDS);
    } catch (Throwable ignore) {
    }
```

Step8：等待收集投票结果，并设置超时时间。

```
    lastVoteCost = DLedgerUtils.elapsed(startVoteTimeMs);
    VoteResponse.ParseResult parseResult;
    if (knownMaxTermInGroup.get() > term) {
        parseResult = VoteResponse.ParseResult.WAIT_TO_VOTE_NEXT;
        nextTimeToRequestVote = getNextTimeToRequestVote();
        changeRoleToCandidate(knownMaxTermInGroup.get());
    } else if (alreadyHasLeader.get()) {
        parseResult = VoteResponse.ParseResult.WAIT_TO_VOTE_NEXT;
        nextTimeToRequestVote = getNextTimeToRequestVote() + heartBeatTimeInterval
Ms * maxHeartBeatLeak;
    } else if (!memberState.isQuorum(validNum.get())) {
        parseResult = VoteResponse.ParseResult.WAIT_TO_REVOTE;
        nextTimeToRequestVote = getNextTimeToRequestVote();
    } else if (memberState.isQuorum(acceptedNum.get())) {
        parseResult = VoteResponse.ParseResult.PASSED;
    } else if (memberState.isQuorum(acceptedNum.get() + notReadyTermNum.get())) {
        parseResult = VoteResponse.ParseResult.REVOTE_IMMEDIATELY;
    } else if (memberState.isQuorum(acceptedNum.get() + biggerLedgerNum.get())) {
```

```
            parseResult = VoteResponse.ParseResult.WAIT_TO_REVOTE;
            nextTimeToRequestVote = getNextTimeToRequestVote();
        } else {
            parseResult = VoteResponse.ParseResult.WAIT_TO_VOTE_NEXT;
            nextTimeToRequestVote = getNextTimeToRequestVote();
        }
```

Step9：根据收集的投票结果判断是否能成为 Leader。

温馨提示：在讲解关键点之前，我们先定义先将（当前时间戳 + 上次投票的开销 + 最小投票间隔(300ms) + （1000- 300 ）之间的随机值）定义为“ 1 个常规计时器”。

其关键点如下：

- 如果对端的投票轮次大于发起投票的节点，则该节点使用对端的轮次，重新进入到 Candidate 状态，并且重置投票计时器，其值为“1 个常规计时器”
- 如果已经存在 Leader，该节点重新进入到 Candidate,并重置定时器，该定时器的时间： “1 个常规计时器” + heartBeatTimeIntervalMs * maxHeartBeatLeak ，其中 heartBeatTimeIntervalMs 为一次心跳间隔时间，
- maxHeartBeatLeak 为 允许最大丢失的心跳包，即如果 Flower 节点在多少个心跳周期内未收到心跳包，则认为 Leader 已下线。
- 如果收到的有效票数未超过半数，则重置计时器为“ 1 个常规计时器”，然后等待重新投票，注意状态为 WAIT_TO_REVOTE，该状态下的特征是下次投票时不增加投票轮次。
- 如果得到的赞同票超过半数，则成为 Leader。
- 如果得到的赞成票加上未准备投票的节点数超过半数，则应该立即发起投票，故其结果为 REVOTE_IMMEDIATELY。
- 如果得到的赞成票加上对端维护的 ledgerEndIndex 超过半数，则重置计时器，继续本轮次的选举。
- 其他情况，开启下一轮投票。

```
    if (parseResult == VoteResponse.ParseResult.PASSED) {
        logger.info("[{}] [VOTE_RESULT] has been elected to be the leader in term {}",
memberState.getSelfId(), term);
        changeRoleToLeader(term);
    }
```

Step10：如果投票成功，则状态机状态设置为 Leader，然后状态管理在驱动状态时会调用 DLedgerLeaderElector#maintainState 时，将进入到 maintainAsLeader 方法。

maintainAsLeader 方法

经过 maintainAsCandidate 投票选举后，被其他节点选举成为领导后，会执行该方法，其他节点的状态还是 Candidate，并在计时器过期后，又尝试去发起选举。接下来重点分析成为 Leader 节点后，该节点会做些什么？

```
DLedgerLeaderElector#maintainAsLeader
private void maintainAsLeader() throws Exception {
    if (DLedgerUtils.elapsed(lastSendHeartBeatTime) > heartBeatTimeIntervalMs) {  //
@1
        long term;
        String leaderId;
        synchronized (memberState) {
            if (!memberState.isLeader()) {       // @2
                //stop sending
                return;
            }
            term = memberState.currTerm();
            leaderId = memberState.getLeaderId();
            lastSendHeartBeatTime = System.currentTimeMillis();      // @3
        }
        sendHeartbeats(term, leaderId);      // @4
    }
}
```

代码@1：首先判断上一次发送心跳的时间与当前时间的差值是否大于心跳包发送间隔，如果超过，则说明需要发送心跳包。

代码@2：如果当前不是 leader 节点，则直接返回，主要是为了二次判断。

代码@3：重置心跳包发送计时器。

代码@4：向集群内的所有节点发送心跳包，稍后会详细介绍心跳包的发送。

maintainAsFollower 方法

当 Candidate 状态的节点在收到主节点发送的心跳包后，会将状态变更为 follower，那我们先来看一下在 follower 状态下，节点会做些什么事情?

```
    private void maintainAsFollower() {
        if (DLedgerUtils.elapsed(lastLeaderHeartBeatTime) > 2 * heartBeatTimeIntervalMs)
{
            synchronized (memberState) {
                if (memberState.isFollower() && (DLedgerUtils.elapsed(lastLeaderHeartBe
atTime) > maxHeartBeatLeak * heartBeatTimeIntervalMs)) {
                    logger.info("[{}][HeartBeatTimeOut] lastLeaderHeartBeatTime: {} heartB
eatTimeIntervalMs: {} lastLeader={}", memberState.getSelfId(), new Timestamp(lastLeader
HeartBeatTime), heartBeatTimeIntervalMs, memberState.getLeaderId());
                    changeRoleToCandidate(memberState.currTerm());
                }
            }
        }
    }
```

如果 maxHeartBeatLeak（默认为 3)个心跳包周期内未收到心跳，则将状态变更为 Candidate。

状态机的驱动就介绍到这里，在上面的流程中，其实我们忽略了两个重要的过程，一个是发起投票请求与投票请求响应、发送心跳包与心跳包响应，那我们接下来将重点介绍这两个过程。

4. 投票与投票请求

节点的状态为 Candidate 时会向集群内的其他节点发起投票请求(个人觉得理解为拉票更好)，向对方询问是否愿意选举我为 Leader，对端节点会根据自己的情况对其投赞成票、拒绝票，如果是拒绝票，还会给出拒绝原因，具体由 voteForQuorumResponses、handleVote 这两个方法来实现，接下来我们分别对这两个方法进行详细分析。

voteForQuorumResponses

发起投票请求。

```
    private List<CompletableFuture<VoteResponse>> voteForQuorumResponses(long ter
m, long ledgerEndTerm,
        long ledgerEndIndex) throws Exception {    // @1
        List<CompletableFuture<VoteResponse>> responses = new ArrayList<>();
        for (String id : memberState.getPeerMap().keySet()) {                    // @2
            VoteRequest voteRequest = new VoteRequest();                         // @3 start
            voteRequest.setGroup(memberState.getGroup());
            voteRequest.setLedgerEndIndex(ledgerEndIndex);
            voteRequest.setLedgerEndTerm(ledgerEndTerm);
            voteRequest.setLeaderId(memberState.getSelfId());
            voteRequest.setTerm(term);
            voteRequest.setRemoteId(id);
            CompletableFuture<VoteResponse> voteResponse;                // @3 end
            if (memberState.getSelfId().equals(id)) {                            // @4
                voteResponse = handleVote(voteRequest, true);
            } else {
                //async
                voteResponse = dLedgerRpcService.vote(voteRequest);  // @5
            }
            responses.add(voteResponse);
        }
        return responses;
    }
```

代码@1：首先先解释一下参数的含义：

- long term

  发起投票的节点当前的投票轮次。

- long ledgerEndTerm

  发起投票节点维护的已知的最大投票轮次。

- long ledgerEndIndex
  发起投票节点维护的已知的最大日志条目索引。

代码@2：遍历集群内的节点集合，准备异步发起投票请求。这个集合在启动的时候指定，不能修改。

代码@3：构建投票请求。

代码@4：如果是发送给自己的，则直接调用 handleVote 进行投票请求响应，如果是发送给集群内的其他节点，则通过网络发送投票请求，对端节点调用各自的 handleVote 对集群进行响应。

接下来重点关注 handleVote 方法，重点探讨其投票处理逻辑。

handleVote 方法

由于 handleVote 方法会并发被调用，因为可能同时收到多个节点的投票请求，故本方法都被 synchronized 方法包含，锁定的对象为状态机 memberState 对象。

```
    if (!memberState.isPeerMember(request.getLeaderId())) {
        logger.warn("[BUG] [HandleVote] remoteId={} is an unknown member", request.getLeaderId());
        return CompletableFuture.completedFuture(newVoteResponse(request).term(memberState.currTerm()).voteResult(VoteResponse.RESULT.REJECT_UNKNOWN_LEADER));
    }
    if (!self && memberState.getSelfId().equals(request.getLeaderId())) {
        logger.warn("[BUG] [HandleVote] selfId={} but remoteId={}", memberState.getSelfId(), request.getLeaderId());
        return CompletableFuture.completedFuture(new VoteResponse(request).term(memberState.currTerm()).voteResult(VoteResponse.RESULT.REJECT_UNEXPECTED_LEADER));
    }
```

Step1：为了逻辑的完整性对其请求进行检验，除非有 BUG 存在，否则是不会出现上述问题的。

```
    if (request.getTerm() < memberState.currTerm()) {      // @1
        return CompletableFuture.completedFuture(new VoteResponse(request).term(mem
berState.currTerm()).voteResult(VoteResponse.RESULT.REJECT_EXPIRED_VOTE_TER
M));
    } else if (request.getTerm() == memberState.currTerm()) {    // @2
        if (memberState.currVoteFor() == null) {
            //let it go
        } else if (memberState.currVoteFor().equals(request.getLeaderId())) {
             //repeat just let it go
        } else {
            if (memberState.getLeaderId() != null) {
                return CompletableFuture.completedFuture(new VoteResponse(request).t
erm(memberState.currTerm()).voteResult(VoteResponse.RESULT.REJECT_ALREADY__H
AS_LEADER));
            } else {
                return CompletableFuture.completedFuture(new VoteResponse(reques
t).term(memberState.currTerm()).voteResult(VoteResponse.RESULT.REJECT_ALREADY_
VOTED));
            }
        }
    } else {                // @3
        //stepped down by larger term
        changeRoleToCandidate(request.getTerm());
        needIncreaseTermImmediately = true;
        //only can handleVote when the term is consistent
        return CompletableFuture.completedFuture(new VoteResponse(request).term(mem
berState.currTerm()).voteResult(VoteResponse.RESULT.REJECT_TERM_NOT_READY));
    }
```

Step2：判断发起节点、响应节点维护的 team 进行投票“仲裁”，分如下几种情况讨论：

- 如果发起投票节点的 term 小于当前节点的 term
  - 此种情况下投拒绝票，也就是说在 raft 协议的世界中，谁的 term 越大，越有话语权。
- 如果发起投票节点的 term 等于当前节点的 term
- 如果两者的 term 相等，说明两者都处在同一个投票轮次中，地位平等，接下来看该节点是否已经投过票。

- 如果未投票、或已投票给请求节点，则继续后面的逻辑（请看 step3）。
- 如果该节点已存在的 Leader 节点，则拒绝并告知已存在 Leader 节点。
- 如果该节点还未有 Leader 节点，但已经投了其他节点的票，则拒绝请求节点，并告知已投票。
- 如果发起投票节点的 term 大于当前节点的 term。
  - 拒绝请求节点的投票请求，并告知自身还未准备投票，自身会使用请求节点的投票轮次立即进入到 Candidate 状态。

```
    if (request.getLedgerEndTerm() < memberState.getLedgerEndTerm()) {
        return CompletableFuture.completedFuture(new VoteResponse(request).term(mem
berState.currTerm()).voteResult(VoteResponse.RESULT.REJECT_EXPIRED_LEDGER_TE
RM));
    } else if (request.getLedgerEndTerm() == memberState.getLedgerEndTerm() && requ
est.getLedgerEndIndex() < memberState.getLedgerEndIndex()) {
        return CompletableFuture.completedFuture(new VoteResponse(request).term(mem
berState.currTerm()).voteResult(VoteResponse.RESULT.REJECT_SMALL_LEDGER_END
_INDEX));
    }

    if (request.getTerm() < memberState.getLedgerEndTerm()) {
        return CompletableFuture.completedFuture(new VoteResponse(request).term(mem
berState.getLedgerEndTerm()).voteResult(VoteResponse.RESULT.REJECT_TERM_SMAL
L_THAN_LEDGER));
    }
```

Step3：判断请求节点的 ledgerEndTerm 与当前节点的 ledgerEndTerm，这里主要是判断日志的复制进度。

- 如果请求节点的 ledgerEndTerm 小于当前节点的 ledgerEndTerm 则拒绝，其原因是请求节点的日志复制进度比当前节点低，这种情况是不能成为主节点的。
- 如果 ledgerEndTerm 相等，但是 ledgerEndIndex 比当前节点小，则拒绝，原因与上一条相同。
- 如果请求的 term 小于 ledgerEndTerm 以同样的理由拒绝。

```
    memberState.setCurrVoteFor(request.getLeaderId());
    return CompletableFuture.completedFuture(new VoteResponse(request).term(member
State.currTerm()).voteResult(VoteResponse.RESULT.ACCEPT));
```

Step4：经过层层条件帅选，将宝贵的赞成票投给请求节点。

经过几轮投票，最终一个节点能成功被推举出来，选为主节点。主节点为了维持其领导地位，需要定时向从节点发送心跳包，接下来我们重点看一下心跳包的发送与响应。

5. 心跳包与心跳包响应

sendHeartbeats

Step1：遍历集群中的节点，异步发送心跳包。

```
    CompletableFuture<HeartBeatResponse> future = dLedgerRpcService.heartBeat(hea
rtBeatRequest);
        future.whenComplete((HeartBeatResponse x, Throwable ex) -> {
            try {

                if (ex != null) {
                    throw ex;
                }
                switch (DLedgerResponseCode.valueOf(x.getCode())) {
                    case SUCCESS:
                        succNum.incrementAndGet();
                        break;
                    case EXPIRED_TERM:
                        maxTerm.set(x.getTerm());
                        break;
                    case INCONSISTENT_LEADER:
                        inconsistLeader.compareAndSet(false, true);
                        break;
                    case TERM_NOT_READY:
                        notReadyNum.incrementAndGet();
                        break;
                    default:
```

```
                    break;
                }
                if (memberState.isQuorum(succNum.get())
                    || memberState.isQuorum(succNum.get() + notReadyNum.get())) {
                    beatLatch.countDown();
                }
            } catch (Throwable t) {
                logger.error("Parse heartbeat response failed", t);
            } finally {
                allNum.incrementAndGet();
                if (allNum.get() == memberState.peerSize()) {
                    beatLatch.countDown();
                }
            }
        });
    }
```

Step2：统计心跳包发送响应结果，关键点如下：

- SUCCESS
  心跳包成功响应。

- EXPIRED_TERM
  主节点的投票 term 小于从节点的投票轮次。

- INCONSISTENT_LEADER
  从节点已经有了新的主节点。

- TERM_NOT_READY
  从节点未准备好。

这些响应值，我们在处理心跳包时重点探讨。

```
beatLatch.await(heartBeatTimeIntervalMs, TimeUnit.MILLISECONDS);
if (memberState.isQuorum(succNum.get())) {    // @1
    lastSuccHeartBeatTime = System.currentTimeMillis();
```

```
    } else {
        logger.info("[{}] Parse heartbeat responses in cost={} term={} allNum={} succNum
={} notReadyNum={} inconsistLeader={} maxTerm={} peerSize={} lastSuccHeartBeatTime=
{}",
                memberState.getSelfId(), DLedgerUtils.elapsed(startHeartbeatTimeMs),
 term, allNum.get(), succNum.get(), notReadyNum.get(), inconsistLeader.get(), maxTerm.g
et(), memberState.peerSize(), new Timestamp(lastSuccHeartBeatTime));
        if (memberState.isQuorum(succNum.get() + notReadyNum.get())) {     // @2
            lastSendHeartBeatTime = -1;
        } else if (maxTerm.get() > term) {
         // @3
            changeRoleToCandidate(maxTerm.get());
        } else if (inconsistLeader.get()) {
         // @4
            changeRoleToCandidate(term);
        } else if (DLedgerUtils.elapsed(lastSuccHeartBeatTime) > maxHeartBeatLeak * he
artBeatTimeIntervalMs) {
            changeRoleToCandidate(term);
        }
    }
```

对收集的响应结果做仲裁，其实现关键点：

- 如果成功的票数大于进群内的半数，则表示集群状态正常，正常按照心跳包间隔发送心跳包(见代码@1)。
- 如果成功的票数加上未准备的投票的节点数量超过集群内的半数，则立即发送心跳包(见代码@2)。
- 如果从节点的投票轮次比主节点的大，则使用从节点的投票轮次，或从节点已经有了另外的主节点，节点状态从 Leader 转换为 Candidate。

思考一下：主节点的投票轮次在什么情况下会比从节点小，从节点为什么会有另外的主节点了？

接下来我们重点看一下心跳包的处理逻辑。

handleHeartBeat

```
    if (request.getTerm() < memberState.currTerm()) {
        return CompletableFuture.completedFuture(new HeartBeatResponse().term(memb
erState.currTerm()).code(DLedgerResponseCode.EXPIRED_TERM.getCode()));
    } else if (request.getTerm() == memberState.currTerm()) {
        if (request.getLeaderId().equals(memberState.getLeaderId())) {
            lastLeaderHeartBeatTime = System.currentTimeMillis();
            return CompletableFuture.completedFuture(new HeartBeatResponse());
        }
    }
```

Step1：如果主节点的 term 小于 从节点的 term，发送反馈给主节点，告知主节点的 term 已过时；如果投票轮次相同，并且发送心跳包的节点是该节点的主节点，则返回成功。

下面重点讨论主节点的 term 大于从节点的情况。

```
    synchronized (memberState) {
        if (request.getTerm() < memberState.currTerm()) {    // @1
            return CompletableFuture.completedFuture(new HeartBeatResponse().term(me
mberState.currTerm()).code(DLedgerResponseCode.EXPIRED_TERM.getCode()));
        } else if (request.getTerm() == memberState.currTerm()) {  // @2
            if (memberState.getLeaderId() == null) {
                changeRoleToFollower(request.getTerm(), request.getLeaderId());
                return CompletableFuture.completedFuture(new HeartBeatResponse());
            } else if (request.getLeaderId().equals(memberState.getLeaderId())) {
                lastLeaderHeartBeatTime = System.currentTimeMillis();
                return CompletableFuture.completedFuture(new HeartBeatResponse());
            } else {
                //this should not happen, but if happened
                logger.error("[{}][BUG] currTerm {} has leader {}, but received leader {}",
memberState.getSelfId(), memberState.currTerm(), memberState.getLeaderId(), request.ge
tLeaderId());
                return CompletableFuture.completedFuture(new HeartBeatResponse().cod
e(DLedgerResponseCode.INCONSISTENT_LEADER.getCode()));
            }
        } else {
```

```
                //To make it simple, for larger term, do not change to follower immediately
                //first change to candidate, and notify the state-maintainer thread
                changeRoleToCandidate(request.getTerm());
                needIncreaseTermImmediately = true;
                //TOOD notify
                return CompletableFuture.completedFuture(new HeartBeatResponse().code(DL
edgerResponseCode.TERM_NOT_READY.getCode()));
            }
        }
```

Step2：加锁来处理（这里更多的是从节点第一次收到主节点的心跳包）

代码@1：如果主节的投票轮次小于当前投票轮次，则返回主节点投票轮次过期。

代码@2：如果投票轮次相同。

如果当前节点的主节点字段为空，则使用主节点的 ID，并返回成功。

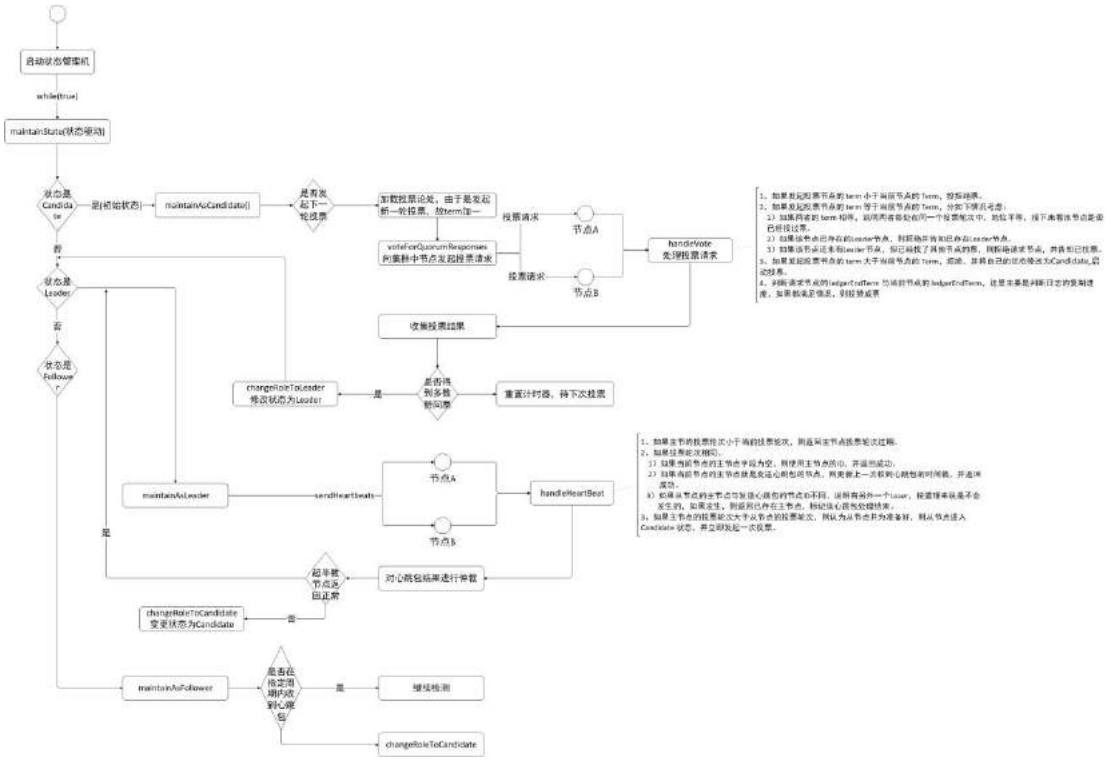
如果当前节点的主节点就是发送心跳包的节点，则更新上一次收到心跳包的时间戳，并返回成功。

如果从节点的主节点与发送心跳包的节点 ID 不同，说明有另外一个 Leaer，按道理来说是不会发送的，如果发生，则返回已存在主节点，标记该心跳包处理结束。

代码@3：如果主节点的投票轮次大于从节点的投票轮次，则认为从节点并为准备好，则从节点进入 Candidate 状态，并立即发起一次投票。

心跳包的处理就介绍到这里。

RocketMQ 多副本之 Leader 选举的源码分析就介绍到这里了，为了加强对源码的理解，先梳理流程图如下：



本文就介绍到这里了，如果对您有一定的帮助，麻烦帮忙点点在看，您的认可是我持续创作的最大动力。

2.6 源码分析 RocketMQ DLedger(多副本) 之日志追加流程

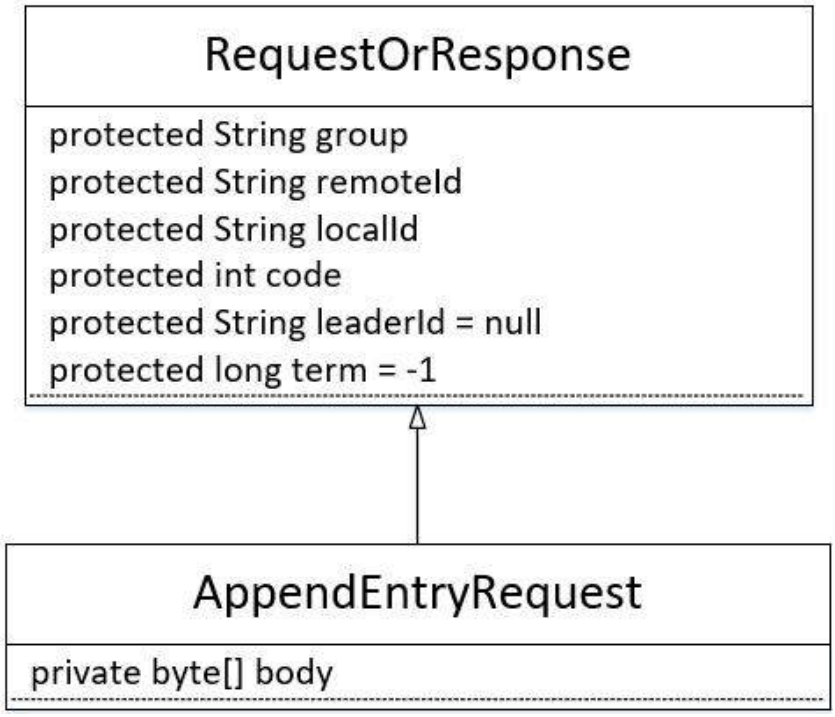
上一篇我们详细分析了 Dledger 是如何实现 raft 协议的选主部分的，本文将详细分析日志复制的实现。

根据 raft 协议可知，当整个集群完成 Leader 选主后，集群中的主节点就可以接受客户端的请求，而集群中的从节点只负责从主节点同步数据，而不会处理读写请求，与 M-S 结构的读写分离有着巨大的区别。

有了前篇文章的基础，本文将直接从 Leader 处理客户端请求入口开始，其入口为：DLedgerServer 的 handleAppend 方法开始讲起。

一、日志复制基本流程

在正式分析 RocketMQ DLedger 多副本复制之前，我们首先来了解客户端发送日志的请求协议字段，其类图如下所示：



我们先一一介绍各个字段的含义：

- String group
  该集群所属组名。

- String remoteId
  请求目的节点 ID。

- String localId
  节点 ID。

- int code
  请求响应字段，表示返回响应码。

- String leaderId = null
  集群中的 Leader Id。

- long term
  集群当前的选举轮次。

- byte[] body
  待发送的数据。

接下来我们就正式开始日志请求的学习。

```
    DLedgerServer#handleAppend
    PreConditions.check(memberState.getSelfId().equals(request.getRemoteId()), DLedger
ResponseCode.UNKNOWN_MEMBER, "%s != %s", request.getRemoteId(), memberState.
getSelfId());
    reConditions.check(memberState.getGroup().equals(request.getGroup()), DLedgerResp
onseCode.UNKNOWN_GROUP, "%s != %s", request.getGroup(), memberState.getGroup
());
    PreConditions.check(memberState.isLeader(), DLedgerResponseCode.NOT_LEADER);
```

Step1：首先验证请求的合理性：

- 如果请求的节点 ID 不是当前处理节点，则抛出异常。
- 如果请求的集群不是当前节点所在的集群，则抛出异常。
- 如果当前节点不是主节点，则抛出异常。

```
DLedgerServer#handleAppend
long currTerm = memberState.currTerm();
if (dLedgerEntryPusher.isPendingFull(currTerm)) {  // @1
    AppendEntryResponse appendEntryResponse = new AppendEntryResponse();
    appendEntryResponse.setGroup(memberState.getGroup());
    appendEntryResponse.setCode(DLedgerResponseCode.LEADER_PENDING_FUL
L.getCode());
    appendEntryResponse.setTerm(currTerm);
    appendEntryResponse.setLeaderId(memberState.getSelfId());
    return AppendFuture.newCompletedFuture(-1, appendEntryResponse);
} else {   // @2
    DLedgerEntry dLedgerEntry = new DLedgerEntry();
    dLedgerEntry.setBody(request.getBody());
    DLedgerEntry resEntry = dLedgerStore.appendAsLeader(dLedgerEntry);
    return dLedgerEntryPusher.waitAck(resEntry);
}
```

Step2：如果预处理队列已经满了，则拒绝客户端请求，返回 LEADER_PENDING_FULL 错误码；如果未满，将请求封装成 DledgerEntry，则调用 dLedgerStore 方法追加日志，并且通过使用 dLedgerEntryPusher 的 waitAck 方法同步等待副本节点的复制响应，并最终将结果返回给调用方法。

代码@1：如果 dLedgerEntryPusher 的 push 队列已满，则返回追加一次，其错误码为 LEADER_PENDING_FULL。

代码@2：追加消息到 Leader 服务器，并向从节点广播，在指定时间内如果未收到从节点的确认，则认为追加失败。

接下来就按照上述三个要点进行展开：

- 判断 Push 队列是否已满
- Leader 节点存储消息
- 主节点等待从节点复制 ACK

1. 如何判断 Push 队列是否已满

```
DLedgerEntryPusher#isPendingFull
public boolean isPendingFull(long currTerm) {
    checkTermForPendingMap(currTerm, "isPendingFull");       // @1
    return pendingAppendResponsesByTerm.get(currTerm).size() > dLedgerConfig.get
MaxPendingRequestsNum(); // @2
}
```

主要分两个步骤：

代码@1：检查当前投票轮次是否在 PendingMap 中，如果不在，则初始化，其结构为：Map<Long/* 投票轮次*/, ConcurrentMap<Long, TimeoutFuture<AppendEntryResponse>>>。

代码@2：检测当前等待从节点返回结果的个数是否超过其最大请求数量，可通过 maxPendingRequestsNum 配置，该值默认为：10000。

上述逻辑比较简单，但疑问随着而来，ConcurrentMap<Long, TimeoutFuture<AppendEntryResponse>> 中的数据是从何而来的呢？我们不妨接着往下看。

2. Leader 节点存储数据

Leader 节点的数据存储主要由 DLedgerStore 的 appendAsLeader 方法实现。DLedger 分别实现了基于内存、基于文件的存储实现，本文重点关注基于文件的存储实现，其实现类为：DLedgerMmapFileStore。

下面重点来分析一下数据存储流程。

```
DLedgerMmapFileStore#appendAsLeader
PreConditions.check(memberState.isLeader(), DLedgerResponseCode.NOT_LEADER);
PreConditions.check(!isDiskFull, DLedgerResponseCode.DISK_FULL);
```

Step1：首先判断是否可以追加数据，其判断依据主要是如下两点：

- 当前节点的状态是否是 Leader，如果不是，则抛出异常。
- 当前磁盘是否已满，其判断依据是 DLedger 的根目录 或 数据文件目录的使用率超过了允许使用的最大值，默认值为 85%。

```
ByteBuffer dataBuffer = localEntryBuffer.get();
ByteBuffer indexBuffer = localIndexBuffer.get();
```

Step2：从本地线程变量获取一个数据与索引 buffer。其中用于存储数据的 ByteBuffer，其容量固定为 4M ，索引的 ByteBuffer 为两个索引条目的长度，固定为 64 个字节。

```
DLedgerEntryCoder.encode(entry, dataBuffer);
public static void encode(DLedgerEntry entry, ByteBuffer byteBuffer) {
    byteBuffer.clear();
    int size = entry.computSizeInBytes();
    //always put magic on the first position
    byteBuffer.putInt(entry.getMagic());
    byteBuffer.putInt(size);
    byteBuffer.putLong(entry.getIndex());
    byteBuffer.putLong(entry.getTerm());
    byteBuffer.putLong(entry.getPos());
    byteBuffer.putInt(entry.getChannel());
    byteBuffer.putInt(entry.getChainCrc());
    byteBuffer.putInt(entry.getBodyCrc());
    byteBuffer.putInt(entry.getBody().length);
    byteBuffer.put(entry.getBody());
    byteBuffer.flip();
}
```

Step3：将 DLedgerEntry，即将数据写入到 ByteBuffer 中，从这里看出，每一次写入会调用 ByteBuffer 的 clear 方法，将数据清空，从这里可以看出，每一次数据追加，只能存储 4M 的数据。

```
DLedgerMmapFileStore#appendAsLeader
synchronized (memberState) {
    PreConditions.check(memberState.isLeader(), DLedgerResponseCode.NOT_LEADER, null);
// ... 省略代码
}
```

Step4：锁定状态机，并再一次检测节点的状态是否是 Leader 节点。

```
DLedgerMmapFileStore#appendAsLeader
long nextIndex = ledgerEndIndex + 1;
entry.setIndex(nextIndex);
entry.setTerm(memberState.currTerm());
entry.setMagic(CURRENT_MAGIC);
DLedgerEntryCoder.setIndexTerm(dataBuffer, nextIndex, memberState.currTerm(), CURRENT_MAGIC);
```

Step5：为当前日志条目设置序号，即 entryIndex 与 entryTerm (投票轮次)。并将魔数、entryIndex、entryTerm 等写入到 bytebuffer 中。

```
DLedgerMmapFileStore#appendAsLeader
long prePos = dataFileList.preAppend(dataBuffer.remaining());
entry.setPos(prePos);
PreConditions.check(prePos != -1, DLedgerResponseCode.DISK_ERROR, null);
DLedgerEntryCoder.setPos(dataBuffer, prePos);
```

Step6：计算新的消息的起始偏移量，关于 dataFileList 的 preAppend 后续详细介绍其实现，然后将该偏移量写入日志的 bytebuffer 中。

```
DLedgerMmapFileStore#appendAsLeader
for (AppendHook writeHook : appendHooks) {
    writeHook.doHook(entry, dataBuffer.slice(), DLedgerEntry.BODY_OFFSET);
}
```

Step7：执行钩子函数。

```
DLedgerMmapFileStore#appendAsLeader
long dataPos = dataFileList.append(dataBuffer.array(), 0, dataBuffer.remaining());
PreConditions.check(dataPos != -1, DLedgerResponseCode.DISK_ERROR, null);
PreConditions.check(dataPos == prePos, DLedgerResponseCode.DISK_ERROR, null);
```

Step8：将数据追加到 pagecache 中。该方法稍后详细介绍。

```
DLedgerMmapFileStore#appendAsLeader
DLedgerEntryCoder.encodeIndex(dataPos, entrySize, CURRENT_MAGIC, nextIndex, memberState.currTerm(), indexBuffer);
long indexPos = indexFileList.append(indexBuffer.array(), 0, indexBuffer.remaining(), false);
PreConditions.check(indexPos == entry.getIndex() * INDEX_UNIT_SIZE, DLedgerResponseCode.DISK_ERROR, null);
```

Step9：构建条目索引并将索引数据追加到 pagecache。

```
DLedgerMmapFileStore#appendAsLeader
ledgerEndIndex++;
ledgerEndTerm = memberState.currTerm();
if (ledgerBeginIndex == -1) {
    ledgerBeginIndex = ledgerEndIndex;
}
updateLedgerEndIndexAndTerm();
```

Step10：ledgerEndeIndex 加一（下一个条目）的序号。并设置 leader 节点的状态机的 ledgerEndIndex 与 ledgerEndTerm。

Leader 节点数据追加就介绍到这里，稍后会重点介绍与存储相关方法的实现细节。

3. 主节点等待从节点复制 ACK

其实现入口为 dLedgerEntryPusher 的 waitAck 方法。

```
DLedgerEntryPusher#waitAck
public CompletableFuture<AppendEntryResponse> waitAck(DLedgerEntry entry) {
    updatePeerWaterMark(entry.getTerm(), memberState.getSelfId(), entry.getIndex());
// @1
    if (memberState.getPeerMap().size() == 1) {
                        // @2
        AppendEntryResponse response = new AppendEntryResponse();
        response.setGroup(memberState.getGroup());
```

```
                response.setLeaderId(memberState.getSelfId());
                response.setIndex(entry.getIndex());
                response.setTerm(entry.getTerm());
                response.setPos(entry.getPos());
                return AppendFuture.newCompletedFuture(entry.getPos(), response);
            } else {
                checkTermForPendingMap(entry.getTerm(), "waitAck");

                AppendFuture<AppendEntryResponse> future = new AppendFuture<>(dLedge
rConfig.getMaxWaitAckTimeMs()); // @3
                future.setPos(entry.getPos());
                CompletableFuture<AppendEntryResponse> old = pendingAppendResponses
ByTerm.get(entry.getTerm()).put(entry.getIndex(), future);       // @4
                if (old != null) {
                    logger.warn("[MONITOR] get old wait at index={}", entry.getIndex());
                }
                wakeUpDispatchers();                                                    // @5
                return future;
            }
        }
```

代码@1：更新当前节点的 push 水位线。

代码@2：如果集群的节点个数为 1，无需转发，直接返回成功结果。

代码@3：构建 append 响应 Future 并设置超时时间，默认值为：2500 ms，可以通过 maxWaitAckTimeMs 配置改变其默认值。

代码@4：将构建的 Future 放入等待结果集合中。

代码@5：唤醒 Entry 转发线程，即将主节点中的数据 push 到各个从节点。

接下来分别对上述几个关键点进行解读。

updatePeerWaterMark 方法

```
DLedgerEntryPusher#updatePeerWaterMark
private void updatePeerWaterMark(long term, String peerId, long index) {     // 代码@1
    synchronized (peerWaterMarksByTerm) {
        checkTermForWaterMark(term, "updatePeerWaterMark");                    // 代码@2
        if (peerWaterMarksByTerm.get(term).get(peerId) < index) {              // 代码@3
            peerWaterMarksByTerm.get(term).put(peerId, index);
        }
    }
}
```

代码@1：先来简单介绍该方法的两个参数：

- long term
  当前的投票轮次。

- String peerId
  当前节点的 ID。

- long index
  当前追加数据的序号。

代码@2：初始化 peerWaterMarksByTerm 数据结构，其结果为 < Long /** term */, Map< String /** peerId */, Long /** entry index*/>。

代码@3：如果 peerWaterMarksByTerm 存储的 index 小于当前数据的 index，则更新。

wakeUpDispatchers 详解

```
DLedgerEntryPusher#updatePeerWaterMark
public void wakeUpDispatchers() {
    for (EntryDispatcher dispatcher : dispatcherMap.values()) {
        dispatcher.wakeup();
    } }
```

该方法主要就是遍历转发器并唤醒。本方法的核心关键就是 EntryDispatcher，在详细介绍它之前我们先来看一下该集合的初始化。

```
DLedgerEntryPusher 构造方法
for (String peer : memberState.getPeerMap().keySet()) {
    if (!peer.equals(memberState.getSelfId())) {
        dispatcherMap.put(peer, new EntryDispatcher(peer, logger));
    }
}
```

原来在构建 DLedgerEntryPusher 时会为每一个子节点创建一 EntryDispatcher 对象。

显然，日志的复制由 DLedgerEntryPusher 来实现。由于篇幅的原因，该部分内容将在下篇文章中继续。

上面在讲解 Leader 追加日志时并没有详细分析其实现，为了知识体系的完备，接下来我们来分析一下其核心实现。

二、日志存储实现详情

本节主要对 MmapFileList 的 preAppend 与 append 方法进行详细讲解。

备注：存储部分的设计请查阅笔者的博客：https://blog.csdn.net/prestigeding/article/details/100177780 ，MmapFileList 对标 RocketMQ 的 MappedFileQueue。

1. MmapFileList 的 preAppend 详解

该方法最终会调用两个参数的 preAppend 方法，故我们直接来看两个参数的 preAppend 方法。

```
    MmapFileList#preAppend
    public long preAppend(int len, boolean useBlank) {                    // @1
        MmapFile mappedFile = getLastMappedFile();                         // @2 start
        if (null == mappedFile || mappedFile.isFull()) {
            mappedFile = getLastMappedFile(0);
        }
        if (null == mappedFile) {
            logger.error("Create mapped file for {}", storePath);
            return -1;
        }
         // @2 end
        int blank = useBlank ? MIN_BLANK_LEN : 0;
        if (len + blank > mappedFile.getFileSize() - mappedFile.getWrotePosition()) {    //
@3
            if (blank < MIN_BLANK_LEN) {
                logger.error("Blank {} should ge {}", blank, MIN_BLANK_LEN);
                return -1;
            } else {
                ByteBuffer byteBuffer = ByteBuffer.allocate(mappedFile.getFileSize() - ma
ppedFile.getWrotePosition());       // @4
                byteBuffer.putInt(BLANK_MAGIC_CODE);
                                                                           // @5
                byteBuffer.putInt(mappedFile.getFileSize() - mappedFile.getWrotePosition
());                                                    // @6
                if (mappedFile.appendMessage(byteBuffer.array())) {
                                                                           // @7
                    //need to set the wrote position
                    mappedFile.setWrotePosition(mappedFile.getFileSize());
                } else {
                    logger.error("Append blank error for {}", storePath);
                    return -1;
                }
                mappedFile = getLastMappedFile(0);
                if (null == mappedFile) {
                    logger.error("Create mapped file for {}", storePath);
                    return -1;
                }
            }
        }
        return mappedFile.getFileFromOffset() + mappedFile.getWrotePosition();// @8
    }
```


代码@1：首先介绍其参数的含义：

- int len

需要申请的长度。

- boolean useBlank

是否需要填充，默认为 true。

代码@2：获取最后一个文件，即获取当前正在写的文件。

代码@3：如果需要申请的资源超过了当前文件可写字节时，需要处理的逻辑。代码@4-@7 都是其处理逻辑。

代码@4：申请一个当前文件剩余字节的大小的 bytebuffer。

代码@5：先写入魔数。

代码@6：写入字节长度，等于当前文件剩余的总大小。

代码@7：写入空字节，代码@4-@7 的用意就是写一条空 Entry，填入魔数与 size，方便解析。

代码@8：如果当前文件足以容纳待写入的日志，则直接返回其物理偏移量。

经过上述代码解读，我们很容易得出该方法的作用，就是返回待写入日志的起始物理偏移量。

2. MmapFileList 的 append 详解

最终会调用 4 个参数的 append 方法，其代码如下：

```
MmapFileList#append
public long append(byte[] data, int pos, int len, boolean useBlank) {   // @1
    if (preAppend(len, useBlank) == -1) {
      return -1;
    }
    MmapFile mappedFile = getLastMappedFile();                          // @2
    long currPosition = mappedFile.getFileFromOffset() + mappedFile.getWrotePosition
();    // @3
    if (!mappedFile.appendMessage(data, pos, len)) {                    // @4
        logger.error("Append error for {}", storePath);
        return -1;
    }
    return currPosition;
}
```

代码@1：首先介绍一下各个参数：

- byte[] data
  待写入的数据，即待追加的日志。

- int pos
  从 data 字节数组哪个位置开始读取。

- int len
  待写入的字节数量。

- boolean useBlank
  是否使用填充，默认为 true。

代码@2：获取最后一个文件，即当前可写的文件。

代码@3：获取当前写入指针。

代码@4：追加消息。

最后我们再来看一下 appendMessage，具体的消息追加实现逻辑。

```
DefaultMmapFile#appendMessage
public boolean appendMessage(final byte[] data, final int offset, final int length) {
    int currentPos = this.wrotePosition.get();

    if ((currentPos + length) <= this.fileSize) {
        ByteBuffer byteBuffer = this.mappedByteBuffer.slice(); // @1
        byteBuffer.position(currentPos);
        byteBuffer.put(data, offset, length);
        this.wrotePosition.addAndGet(length);
        return true;
    }
    return false;
}
```

该方法我主要是想突出一下写入的方式是 mappedByteBuffer，是通过 FileChannel 的 map 方法创建，即我们常说的 PageCache，即消息追加首先是写入到 pageCache 中。

本文详细介绍了 Leader 节点处理客户端消息追加请求的前面两个步骤，即判断 Push 队列是否已满 与 Leader 节点存储消息。考虑到篇幅的问题，各个节点的数据同步将在下一篇文章中详细介绍。

在进入下一篇的文章学习之前，我们不妨思考一下如下问题：

如果主节点追加成功 (写入到 PageCache)，但同步到从节点过程失败或此时主节点宕机，集群中的数据如何保证一致性?

2.7 源码分析 RocketMQ DLedger(多副本) 之日志复制(传播)

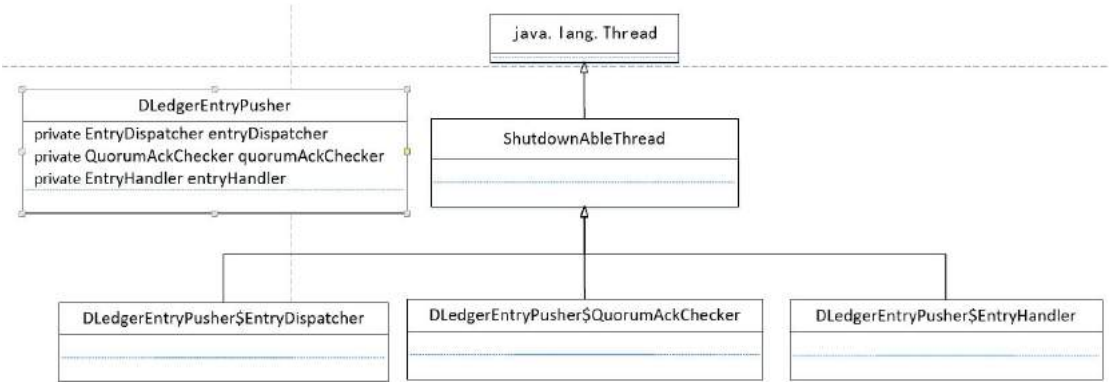
本文紧接着 源码分析 RocketMQ DLedger(多副本) 之日志追加流程 ，继续 Leader 处理客户端 append 的请求流程中最至关重要的一环：日志复制。

DLedger 多副本由 DLedgerEntryPusher 实现，接下来将对其进行详细介绍。

温馨提示：由于本篇幅较长，为了更好的理解其实现，大家可以带着如下疑问来通读本篇文章：

- raft 协议中有一个非常重要的概念：已提交日志序号，该如何实现。
- 客户端向 DLedger 集群发送一条日志，必须得到集群中大多数节点的认可才能被认为写入成功。
- raft 协议中追加、提交两个动作如何实现。

DLedgerEntryPusher 类图：



主要由如下 4 个类构成：

- DLedgerEntryPusher
- DLedger 日志转发与处理核心类，该内会启动如下 3 个对象，其分别对应一个线程。
- EntryHandler
- 日志接收处理线程，当节点为从节点时激活。

- QuorumAckChecker

  日志追加 ACK 投票处理线程，当前节点为主节点时激活。

- EntryDispatcher

  日志转发线程，当前节点为主节点时追加。

接下来我们将详细介绍上述 4 个类，从而揭晓日志复制的核心实现原理。

一、DLedgerEntryPusher

1. 核心类图

DLedgerEntryPusher
private DLedgerConfig dLedgerConfig private DLedgerStore dLedgerStore private final MemberState memberState private DLedgerRpcService dLedgerRpcService private Map<Long, ConcurrentMap<String, Long>> peerWaterMarksByTerm = new ConcurrentHashMap<>() private Map<Long, ConcurrentMap<Long, TimeoutFuture<AppendEntryResponse>>> pendingAppendResponsesByTerm private EntryHandler entryHandler = new EntryHandler(logger) private QuorumAckChecker quorumAckChecker = new QuorumAckChecker(logger) private Map<String, EntryDispatcher> dispatcherMap = new HashMap<>()
public DLedgerEntryPusher(DLedgerConfig c, MemberState state, DLedgerStore store, DLedgerRpcService rpc) public void startup() public void shutdown() public CompletableFuture<PushEntryResponse> handlePush(PushEntryRequest request) public boolean isPendingFull(long currTerm) public CompletableFuture<AppendEntryResponse> waitAck(DLedgerEntry entry) public void wakeUpDispatchers()

DLedger 多副本日志推送的核心实现类，里面会创建 EntryDispatcher、QuorumAckChecker、EntryHandler 三个核心线程。其核心属性如下：

- DLedgerConfig dLedgerConfig

  多副本相关配置。

- DLedgerStore dLedgerStore

  存储实现类。

- MemberState memberState

  节点状态机。

- DLedgerRpcService dLedgerRpcService
  RPC 服务实现类，用于集群内的其他节点进行网络通讯。

- Map<Long, ConcurrentMap<String, Long>> peerWaterMarksByTerm
  每个节点基于投票轮次的当前水位线标记。键值为投票轮次，值为 ConcurrentMap <String/** 节点 id*/, Long/** 节点对应的日志序号*/>。

- Map<Long, ConcurrentMap<Long, TimeoutFuture<AppendEntryRespons e>>> pendingAppendResponsesByTerm
  用于存放追加请求的响应结果(Future 模式)。

- EntryHandler entryHandler
  从节点上开启的线程，用于接收主节点的 push 请求(append、commit、append)。

- QuorumAckChecker quorumAckChecker
  主节点上的追加请求投票器。

- Map<String, EntryDispatcher> dispatcherMap
  主节点日志请求转发器，向从节点复制消息等。

接下来介绍一下其核心方法的实现。

2. 构造方法

```
    public DLedgerEntryPusher(DLedgerConfig dLedgerConfig, MemberState memberStat
e, DLedgerStore dLedgerStore,
        DLedgerRpcService dLedgerRpcService) {
        this.dLedgerConfig = dLedgerConfig;
        this.memberState = memberState;
        this.dLedgerStore = dLedgerStore;
        this.dLedgerRpcService = dLedgerRpcService;
        for (String peer : memberState.getPeerMap().keySet()) {
            if (!peer.equals(memberState.getSelfId())) {
                dispatcherMap.put(peer, new EntryDispatcher(peer, logger));
            }
        } }
```

构造方法的重点是会根据集群内的节点，依次构建对应的 EntryDispatcher 对象。

3. startup

```
DLedgerEntryPusher#startup
public void startup() {
    entryHandler.start();
    quorumAckChecker.start();
    for (EntryDispatcher dispatcher : dispatcherMap.values()) {
        dispatcher.start();
    }
}
```

依次启动 EntryHandler、QuorumAckChecker 与 EntryDispatcher 线程。

备注:DLedgerEntryPusher 的其他核心方法在详细分析其日志复制原理的过程中会一一介绍。

DLedger 可关闭的线程封装，与 RocketMQ 中的 ServiceThread(服务类)线程的实现有异曲同工之妙，主要是实现线程无限循环中的在阻塞、唤醒等通用实现。各个子类只需关注各自具体业务的实现，实现抽象方法 doWork 即可。

接下来将从 EntryDispatcher、QuorumAckChecker、EntryHandler 来阐述 RocketMQ DLedger(多副本)的实现原理。

二、EntryDispatcher 详解

1. 核心类图

DLedgerEntryPusher$EntryDispatcher
private AtomicReference<PushEntryRequest.Type> type = new AtomicReference<>(PushEntryRequest.Type.COMPARE private long lastPushCommitTimeMs = -1 private String peerId private long compareIndex = -1 private long writeIndex = -1 private int maxPendingSize = 1000 private long term = -1 private String leaderId = null private long lastCheckLeakTimeMs = System.currentTimeMillis() private ConcurrentMap<Long, Long> pendingMap = new ConcurrentHashMap<>() private Quota quota = new Quota(dLedgerConfig.getPeerPushQuota())
public EntryDispatcher(String peerId, Logger logger) public void doWork()

其核心属性如下：

- AtomicReference<PushEntryRequest.Type> type = new AtomicReference <>(PushEntryRequest.Type.COMPARE)

向从节点发送命令的类型，可选值：PushEntryRequest.Type.COMPARE、TRUNCATE、APPEND、COMMIT，下面详细说明。

- long lastPushCommitTimeMs = -1

上一次发送提交类型的时间戳。

- String peerId

目标节点 ID。

- long compareIndex = -1

已完成比较的日志序号。

- long writeIndex = -1

已写入的日志序号。

- int maxPendingSize = 1000

允许的最大挂起日志数量。

- long term = -1
  Leader 节点当前的投票轮次。

- String leaderId = null
  Leader 节点 ID。

- long lastCheckLeakTimeMs = System.currentTimeMillis()
  上次检测泄漏的时间，所谓的泄漏，就是看挂起的日志请求数量是否查过了 maxPendingSize 。

- ConcurrentMap<Long, Long> pendingMap = new ConcurrentHashMap<>()
  记录日志的挂起时间，key：日志的序列(entryIndex)，value：挂起时间戳。

- Quota quota = new Quota(dLedgerConfig.getPeerPushQuota())
  配额。

2. Push 请求类型

DLedger 主节点向从从节点复制日志总共定义了 4 类请求类型，其枚举类型为 PushEntryRequest.Type，其值分别为 COMPARE、TRUNCATE、APPEND、COMMIT。

- COMPARE
  如果 Leader 发生变化，新的 Leader 需要与他的从节点的日志条目进行比较，以便截断从节点多余的数据。

- TRUNCATE
  如果 Leader 通过索引完成日志对比，则 Leader 将发送 TRUNCATE 给它的从节点。

- APPEND
  将日志条目追加到从节点。

- COMMIT

通常，leader 会将提交的索引附加到 append 请求，但是如果 append 请求很少且分散，leader 将发送一个单独的请求来通知从节点提交的索引。

对主从节点的请求类型有了一个初步的认识后，我们将从 EntryDispatcher 的业务处理入口 doWork 方法开始讲解。

3. doWork 方法详解

```
    public void doWork() {
        try {
            if (!checkAndFreshState()) {                                    // @1
                waitForRunning(1);
                return;
            }

            if (type.get() == PushEntryRequest.Type.APPEND) {    // @2
                doAppend();
            } else {
                doCompare();                                                //
@3
            }
            waitForRunning(1);
        } catch (Throwable t) {
            DLedgerEntryPusher.logger.error("[Push-{}]Error in {} writeIndex={} compareIn
dex={}", peerId, getName(), writeIndex, compareIndex, t);
            DLedgerUtils.sleep(500);
        }
    }
```

代码@1：检查状态，是否可以继续发送 append 或 compare。

代码@2：如果推送类型为 APPEND，主节点向从节点传播消息请求。

代码@3：主节点向从节点发送对比数据差异请求(当一个新节点被选举成为主节点时，往往这是第一步) 。

checkAndFreshState 详解

```
EntryDispatcher#checkAndFreshState
private boolean checkAndFreshState() {
    if (!memberState.isLeader()) {       // @1
        return false;
    }
    if (term != memberState.currTerm() || leaderId == null || !leaderId.equals(memberS
tate.getLeaderId())) {       // @2
        synchronized (memberState) {
            if (!memberState.isLeader()) {
                return false;
            }
            PreConditions.check(memberState.getSelfId().equals(memberState.getLead
erId()), DLedgerResponseCode.UNKNOWN);
            term = memberState.currTerm();
            leaderId = memberState.getSelfId();
            changeState(-1, PushEntryRequest.Type.COMPARE);
        }
    }
    return true;
}
```

代码@1：如果节点的状态不是主节点，则直接返回 false。则结束 本次 doWork 方法。因为只有主节点才需要向从节点转发日志。

代码@2：如果当前节点状态是主节点，但当前的投票轮次与状态机轮次或 leaderId 还未设置，或 leaderId 与状态机的 leaderId 不相等，这种情况通常是集群触发了重新选举，设置其 term、leaderId 与状态机同步，即将发送 COMPARE 请求。

接下来看一下 changeState（改变状态）。

```
private synchronized void changeState(long index, PushEntryRequest.Type target) {
    logger.info("[Push-{}]Change state from {} to {} at {}", peerId, type.get(), target, in
dex);
    switch (target) {
        case APPEND:         // @1
```

```
                compareIndex = -1;
                updatePeerWaterMark(term, peerId, index);
                quorumAckChecker.wakeup();
                writeIndex = index + 1;
                break;
            case COMPARE:     // @2
                if (this.type.compareAndSet(PushEntryRequest.Type.APPEND, PushEntryR
equest.Type.COMPARE)) {
                    compareIndex = -1;
                    pendingMap.clear();
                }
                break;
            case TRUNCATE:       // @3
                compareIndex = -1;
                break;
            default:
                break;
        }
        type.set(target);
    }
```

代码@1：如果将目标类型设置为 append，则重置 compareIndex ，并设置 writeIndex 为当前 index 加1。

代码@2：如果将目标类型设置为 COMPARE，则重置 compareIndex 为负一，接下将向各个从节点发送 COMPARE 请求类似，并清除已挂起的请求。

代码@3：如果将目标类型设置为 TRUNCATE，则重置 compareIndex 为负一。

接下来具体来看一下 APPEND、COMPARE、TRUNCATE 等请求。

append 请求详解

```
EntryDispatcher#doAppend
private void doAppend() throws Exception {
    while (true) {
        if (!checkAndFreshState()) {
```

```
    // @1
                    break;
                }
                if (type.get() != PushEntryRequest.Type.APPEND) {           // @2
                    break;
                }
                if (writeIndex > dLedgerStore.getLedgerEndIndex()) {      // @3
                    doCommit();
                    doCheckAppendResponse();
                    break;
                }
                if (pendingMap.size() >= maxPendingSize || (DLedgerUtils.elapsed(lastCheckL
eakTimeMs) > 1000)) {       // @4
                    long peerWaterMark = getPeerWaterMark(term, peerId);
                    for (Long index : pendingMap.keySet()) {
                        if (index < peerWaterMark) {
                            pendingMap.remove(index);
                        }
                    }
                    lastCheckLeakTimeMs = System.currentTimeMillis();
                }
                if (pendingMap.size() >= maxPendingSize) {      // @5
                    doCheckAppendResponse();
                    break;
                }
                doAppendInner(writeIndex);                                  // @6
                writeIndex++;
            }
        }
```

代码@1：检查状态，已经在上面详细介绍。

代码@2：如果请求类型不为 APPEND，则退出，结束本轮 doWork 方法执行。

代码@3：writeIndex 表示当前追加到从该节点的序号，通常情况下主节点向从节点发送 append 请求时，会附带主节点的已提交指针，但如何 append 请求发不那么频繁，writeIndex 大于 leaderEndIndex 时（由于 pending 请求超过其 pending 请求的队列长度（默认为 1w），时，会阻止数据的追加，此时有可能出现 writeIndex 大于 leaderEndIndex 的情况，此时单独发送 COMMIT 请求。

代码@4：检测 pendingMap(挂起的请求数量)是否发送泄漏，即挂起队列中容量是否超过允许的最大挂起阀值。获取当前节点关于本轮次的当前水位线(已成功 append 请求的日志序号)，如果发现正在挂起请求的日志序号小于水位线，则丢弃。

代码@5：如果挂起的请求（等待从节点追加结果）大于 maxPendingSize 时，检查并追加一次 append 请求。

代码@6：具体的追加请求。

doCommit 发送提交请求

```
    EntryDispatcher#doCommit
    private void doCommit() throws Exception {
        if (DLedgerUtils.elapsed(lastPushCommitTimeMs) > 1000) {    // @1
            PushEntryRequest request = buildPushRequest(null, PushEntryRequest.Type.
COMMIT);    // @2
            //Ignore the results
            dLedgerRpcService.push(request);
                                                // @3
            lastPushCommitTimeMs = System.currentTimeMillis();
        }
    }
```

代码@1：如果上一次单独发送 commit 的请求时间与当前时间相隔低于 1s，放弃本次提交请求。

代码@2：构建提交请求。

代码@3：通过网络向从节点发送 commit 请求。

接下来先了解一下如何构建 commit 请求包。

```
    EntryDispatcher#buildPushRequest
    private PushEntryRequest buildPushRequest(DLedgerEntry entry, PushEntryRequest.
Type target) {
        PushEntryRequest request = new PushEntryRequest();
```

```
        request.setGroup(memberState.getGroup());
        request.setRemoteId(peerId);
        request.setLeaderId(leaderId);
        request.setTerm(term);
        request.setEntry(entry);
        request.setType(target);
        request.setCommitIndex(dLedgerStore.getCommittedIndex());
        return request;
    }
```

提交包请求字段主要包含如下字段：DLedger 节点所属组、从节点 id、主节点 id，当前投票轮次、日志内容、请求类型与 committedIndex(主节点已提交日志序号)。

doCheckAppendResponse 检查并追加请求

```
    EntryDispatcher#doCheckAppendResponse
    private void doCheckAppendResponse() throws Exception {
        long peerWaterMark = getPeerWaterMark(term, peerId);    // @1
        Long sendTimeMs = pendingMap.get(peerWaterMark + 1);
        if (sendTimeMs != null && System.currentTimeMillis() - sendTimeMs > dLedgerC
onfig.getMaxPushTimeOutMs()) { // @2
            logger.warn("[Push-{}]Retry to push entry at {}", peerId, peerWaterMark + 1);
            doAppendInner(peerWaterMark + 1);
        }
    }
```

该方法的作用是检查 append 请求是否超时，其关键实现如下：

- 获取已成功 append 的序号。
- 从挂起的请求队列中获取下一条的发送时间，如果不为空并去超过了 append 的超时时间，则再重新发送 append 请求，最大超时时间默认为 1s，可以通过 maxPush TimeOutMs 来改变默认值。

doAppendInner 追加请求

向从节点发送 append 请求。

```
    EntryDispatcher#doAppendInner
    private void doAppendInner(long index) throws Exception {
        DLedgerEntry entry = dLedgerStore.get(index);    // @1
        PreConditions.check(entry != null, DLedgerResponseCode.UNKNOWN, "writeIndex
=%d", index);
        checkQuotaAndWait(entry);                                    // @2
        PushEntryRequest request = buildPushRequest(entry, PushEntryRequest.Type.AP
PEND);    // @3
        CompletableFuture<PushEntryResponse> responseFuture = dLedgerRpcService.p
ush(request);    // @4
        pendingMap.put(index, System.currentTimeMillis());
                                            // @5
        responseFuture.whenComplete((x, ex) -> {
            try {
                PreConditions.check(ex == null, DLedgerResponseCode.UNKNOWN);
                DLedgerResponseCode responseCode = DLedgerResponseCode.valueOf
(x.getCode());
                switch (responseCode) {
                    case SUCCESS:
                                                        // @6
                        pendingMap.remove(x.getIndex());
                        updatePeerWaterMark(x.getTerm(), peerId, x.getIndex());
                        quorumAckChecker.wakeup();
                        break;
                    case INCONSISTENT_STATE:
                                                // @7
                        logger.info("[Push-{}]Get INCONSISTENT_STATE when push inde
x={} term={}", peerId, x.getIndex(), x.getTerm());
                        changeState(-1, PushEntryRequest.Type.COMPARE);
                        break;
                    default:
                        logger.warn("[Push-{}]Get error response code {} {}", peerId, resp
onseCode, x.baseInfo());
                        break;
                }
            } catch (Throwable t) {
                logger.error("", t);
            }
        });
        lastPushCommitTimeMs = System.currentTimeMillis();
    }
```

代码@1：首先根据序号查询出日志。

代码@2：检测配额，如果超过配额，会进行一定的限流，其关键实现点：

- 首先触发条件：append 挂起请求数已超过最大允许挂起数；基于文件存储并主从差异超过 300m，可通过 peerPushThrottlePoint 配置。
- 每秒追加的日志超过 20m(可通过 peerPushQuota 配置)，则会 sleep 1s 中后再追加。

代码@3：构建 PUSH 请求日志。

代码@4：通过 Netty 发送网络请求到从节点，从节点收到请求会进行处理(本文并不会探讨与网络相关的实现细节)。

代码@5：用 pendingMap 记录待追加的日志的发送时间，用于发送端判断是否超时的一个依据。

代码@6：请求成功的处理逻辑，其关键实现点如下：

- 移除 pendingMap 中的关于该日志的发送超时时间。
- 更新已成功追加的日志序号(按投票轮次组织，并且每个从服务器一个键值对)。
- 唤醒 quorumAckChecker 线程(主要用于仲裁 append 结果)，后续会详细介绍。

代码@7：Push 请求出现状态不一致情况，将发送 COMPARE 请求，来对比主从节点的数据是否一致。

日志转发 append 追加请求类型就介绍到这里了，接下来我们继续探讨另一个请求类型 compare。

compare 请求详解

COMPARE 类型的请求有 doCompare 方法发送，首先该方法运行在 while (true) 中，故在查阅下面代码时，要注意其退出循环的条件。

```
EntryDispatcher#doCompare
if (!checkAndFreshState()) {
    break;
}
if (type.get() != PushEntryRequest.Type.COMPARE
    && type.get() != PushEntryRequest.Type.TRUNCATE) {
    break;
}
if (compareIndex == -1 && dLedgerStore.getLedgerEndIndex() == -1) {
    break;
}
```

Step1：验证是否执行，有几个关键点如下：

- 判断是否是主节点，如果不是主节点，则直接跳出。
- 如果是请求类型不是 COMPARE 或 TRUNCATE 请求，则直接跳出。
- 如果已比较索引 和 ledgerEndIndex 都为 -1 ，表示一个新的 DLedger 集群，则直接跳出。

```
EntryDispatcher#doCompare
if (compareIndex == -1) {
    compareIndex = dLedgerStore.getLedgerEndIndex();
    logger.info("[Push-{}][DoCompare] compareIndex=-1 means start to compare", peerId);
} else if (compareIndex > dLedgerStore.getLedgerEndIndex() || compareIndex < dLedgerStore.getLedgerBeginIndex()) {
    logger.info("[Push-{}][DoCompare] compareIndex={} out of range {}-{}", peerId, compareIndex, dLedgerStore.getLedgerBeginIndex(), dLedgerStore.getLedgerEndIndex());
    compareIndex = dLedgerStore.getLedgerEndIndex();
}
```

Step2：如果 compareIndex 为 -1 或 compareIndex 不在有效范围内，则重置待比较序列号为当前已已存储的最大日志序号：ledgerEndIndex。

```
DLedgerEntry entry = dLedgerStore.get(compareIndex);
PreConditions.check(entry != null, DLedgerResponseCode.INTERNAL_ERROR, "compareIndex=%d", compareIndex);
```

```
    PushEntryRequest request = buildPushRequest(entry, PushEntryRequest.Type.COMP
ARE);
    CompletableFuture<PushEntryResponse> responseFuture = dLedgerRpcService.push
(request);
    PushEntryResponse response = responseFuture.get(3, TimeUnit.SECONDS);
```

Step3：根据序号查询到日志，并向从节点发起 COMPARE 请求，其超时时间为3s。

```
    EntryDispatcher#doCompare
    long truncateIndex = -1;
    if (response.getCode() == DLedgerResponseCode.SUCCESS.getCode()) {    // @1
        if (compareIndex == response.getEndIndex()) {
            changeState(compareIndex, PushEntryRequest.Type.APPEND);
            break;
        } else {
            truncateIndex = compareIndex;
        }

    } else if (response.getEndIndex() < dLedgerStore.getLedgerBeginIndex()
            || response.getBeginIndex() > dLedgerStore.getLedgerEndIndex()) {     // @2
        truncateIndex = dLedgerStore.getLedgerBeginIndex();
    } else if (compareIndex < response.getBeginIndex()) {
    // @3
        truncateIndex = dLedgerStore.getLedgerBeginIndex();
    } else if (compareIndex > response.getEndIndex()) {
    // @4
        compareIndex = response.getEndIndex();
    } else {
                                    // @5
     compareIndex--;
    }

    if (compareIndex < dLedgerStore.getLedgerBeginIndex()) {                                    //
@6
        truncateIndex = dLedgerStore.getLedgerBeginIndex();
    }
```

Step4：根据响应结果计算需要截断的日志序号，其主要实现关键点如下：

代码@1：如果两者的日志序号相同，则无需截断，下次将直接先从节点发送 append 请求；否则将 truncateIndex 设置为响应结果中的 endIndex。

代码@2：如果从节点存储的最大日志序号小于主节点的最小序号，或者从节点的最小日志序号大于主节点的最大日志序号，即两者不相交，这通常发生在从节点崩溃很长一段时间，而主节点删除了过期的条目时。truncateIndex 设置为主节点的 ledgerBeginIndex，即主节点目前最小的偏移量。

代码@3：如果已比较的日志序号小于从节点的开始日志序号，很可能是从节点磁盘发送损耗，从主节点最小日志序号开始同步。

代码@4：如果已比较的日志序号大于从节点的最大日志序号，则已比较索引设置为从节点最大的日志序号，触发数据的继续同步。

代码@5：如果已比较的日志序号大于从节点的开始日志序号，但小于从节点的最大日志序号，则待比较索引减一。

代码@6：如果比较出来的日志序号小于主节点的最小日志需要，则设置为主节点的最小序号。

```
    if (truncateIndex != -1) {
        changeState(truncateIndex, PushEntryRequest.Type.TRUNCATE);
        doTruncate(truncateIndex);
        break;
    }
```

Step5：如果比较出来的日志序号不等于 -1 ，则向从节点发送 TRUNCATE 请求。

doTruncate 详解

```
    private void doTruncate(long truncateIndex) throws Exception {
        PreConditions.check(type.get() == PushEntryRequest.Type.TRUNCATE, DLedgerResponseCode.UNKNOWN);
```

```
        DLedgerEntry truncateEntry = dLedgerStore.get(truncateIndex);
        PreConditions.check(truncateEntry != null, DLedgerResponseCode.UNKNOWN);
        logger.info("[Push-{}]Will push data to truncate truncateIndex={} pos={}", peerId, tr
uncateIndex, truncateEntry.getPos());
        PushEntryRequest truncateRequest = buildPushRequest(truncateEntry, PushEntry
Request.Type.TRUNCATE);
        PushEntryResponse truncateResponse = dLedgerRpcService.push(truncateReque
st).get(3, TimeUnit.SECONDS);
        PreConditions.check(truncateResponse != null, DLedgerResponseCode.UNKNOW
N, "truncateIndex=%d", truncateIndex);
        PreConditions.check(truncateResponse.getCode() == DLedgerResponseCode.SUC
CESS.getCode(), DLedgerResponseCode.valueOf(truncateResponse.getCode()), "truncatel
ndex=%d", truncateIndex);
        lastPushCommitTimeMs = System.currentTimeMillis();
        changeState(truncateIndex, PushEntryRequest.Type.APPEND);
    }
```

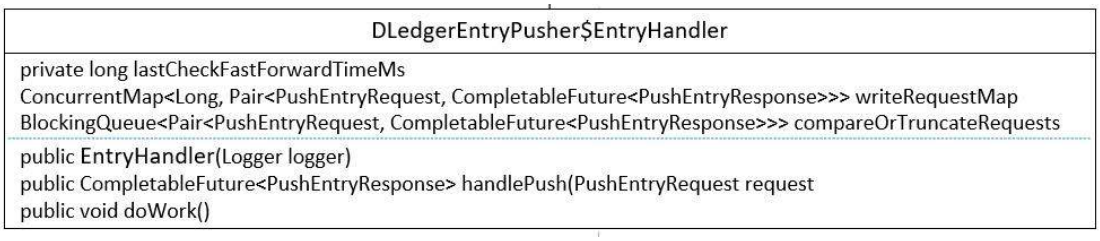
该方法主要就是构建 truncate 请求到从节点。

关于服务端的消息复制转发就介绍到这里了，主节点负责向从服务器 PUSH 请求，从节点自然而然的要处理这些请求，接下来我们就按照主节点发送的请求，来具体分析一下从节点是如何响应的。

三、EntryHandler 详解

EntryHandler 同样是一个线程，当节点状态为从节点时激活。

1. 核心类图



其核心属性如下:

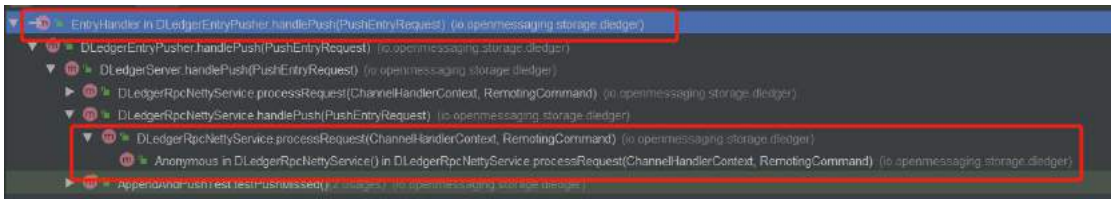
- long lastCheckFastForwardTimeMs
  上一次检查主服务器是否有 push 消息的时间戳。

- ConcurrentMap<Long, Pair<PushEntryRequest, CompletableFuture<PushEntryResponse>>> writeRequestMap
  append 请求处理队列。

- BlockingQueue<Pair<PushEntryRequest, CompletableFuture<PushEntryResponse>>> compareOrTruncateRequests
  COMMIT、COMPARE、TRUNCATE 相关请求

2. handlePush

从上文得知，主节点会主动向从节点传播日志，从节点会通过网络接受到请求数据进行处理，其调用链如图所示:



最终会调用 EntryHandler 的 handlePush 方法。

```
    EntryHandler#handlePush
    public CompletableFuture<PushEntryResponse> handlePush(PushEntryRequest reque
st) throws Exception {
        //The timeout should smaller than the remoting layer's request timeout
        CompletableFuture<PushEntryResponse> future = new TimeoutFuture<>(1000);
    // @1
        switch (request.getType()) {
            case APPEND:
                                            // @2
```

```
        PreConditions.check(request.getEntry() != null, DLedgerResponseCode.UNEXPECTED_ARGUMENT);
                    long index = request.getEntry().getIndex();
                    Pair<PushEntryRequest, CompletableFuture<PushEntryResponse>> old = writeRequestMap.putIfAbsent(index, new Pair<>(request, future));
                    if (old != null) {
                        logger.warn("[MONITOR]The index {} has already existed with {} and curr is {}", index, old.getKey().baseInfo(), request.baseInfo());
                        future.complete(buildResponse(request, DLedgerResponseCode.REPEATED_PUSH.getCode()));
                    }
                    break;
                case COMMIT:
                                                    // @3
                    compareOrTruncateRequests.put(new Pair<>(request, future));
                    break;
                case COMPARE:
                case TRUNCATE:
                                                  // @4
                    PreConditions.check(request.getEntry() != null, DLedgerResponseCode.UNEXPECTED_ARGUMENT);
                    writeRequestMap.clear();
                    compareOrTruncateRequests.put(new Pair<>(request, future));
                    break;
                default:
                    logger.error("[BUG]Unknown type {} from {}", request.getType(), request.baseInfo());
                    future.complete(buildResponse(request, DLedgerResponseCode.UNEXPECTED_ARGUMENT.getCode()));
                    break;
            }
            return future;
        }
```

从几点处理主节点的 push 请求，其实现关键点如下：

代码@1：首先构建一个响应结果 Future，默认超时时间 1s。

代码@2：如果是 APPEND 请求，放入到 writeRequestMap 集合中，如果已存在该数据结构，说明主节点重复推送，构建返回结果，其状态码为 REPEATED_PUSH。放入到 writeRequestMap 中，由 doWork 方法定时去处理待写入的请求。

代码@3：如果是提交请求， 将请求存入 compareOrTruncateRequests 请求处理中，由 doWork 方法异步处理。

代码@4：如果是 COMPARE 或 TRUNCATE 请求，将待写入队列 writeRequestMap 清空，并将请求放入 compareOrTruncateRequests 请求队列中，由 doWork 方法异步处理。

接下来，我们重点来分析 doWork 方法的实现。

3. doWork 方法详解

```
EntryHandler#doWork
public void doWork() {
    try {
        if (!memberState.isFollower()) {       // @1
            waitForRunning(1);
            return;
        }
        if (compareOrTruncateRequests.peek() != null) {     // @2
            Pair<PushEntryRequest, CompletableFuture<PushEntryResponse>> pair =
compareOrTruncateRequests.poll();
            PreConditions.check(pair != null, DLedgerResponseCode.UNKNOWN);
            switch (pair.getKey().getType()) {
                case TRUNCATE:
                    handleDoTruncate(pair.getKey().getEntry().getIndex(), pair.getKey(),
pair.getValue());
                    break;
                case COMPARE:
                    handleDoCompare(pair.getKey().getEntry().getIndex(), pair.getKey(),
pair.getValue());
                    break;
                case COMMIT:
```

```
                handleDoCommit(pair.getKey().getCommitIndex(), pair.getKey(), pair.getValue());
                    break;
                default:
                    break;
            }
        } else { // @3
            long nextIndex = dLedgerStore.getLedgerEndIndex() + 1;
            Pair<PushEntryRequest, CompletableFuture<PushEntryResponse>> pair =
writeRequestMap.remove(nextIndex);
            if (pair == null) {
                checkAbnormalFuture(dLedgerStore.getLedgerEndIndex());
                waitForRunning(1);
                return;
            }
            PushEntryRequest request = pair.getKey();
            handleDoAppend(nextIndex, request, pair.getValue());
        }
    } catch (Throwable t) {
        DLedgerEntryPusher.logger.error("Error in {}", getName(), t);
        DLedgerUtils.sleep(100);
    }
}
```

代码@1：如果当前节点的状态不是从节点，则跳出。

代码@2：如果 compareOrTruncateRequests 队列不为空，说明有 COMMIT、COMPARE、TRUNCATE 等请求，这类请求优先处理。值得注意的是这里使用是 peek、poll 等非阻塞方法，然后根据请求的类型，调用对应的方法。稍后详细介绍。

代码@3：如果只有 append 类请求，则根据当前节点最大的消息序号，尝试从 writeRequestMap 容器中，获取下一个消息复制请求(ledgerEndIndex + 1) 为 key 去查找。如不为空，则执行 doAppend 请求，如果为空，则调用 checkAbnormalFuture 来处理异常情况。

接下来我们来重点分析各个处理细节。

handleDoCommit

处理提交请求，其处理比较简单，就是调用 DLedgerStore 的 updateCommittedIndex 更新其已提交偏移量，故我们还是具体看一下 DLedgerStore 的 updateCommittedIndex 方法。

```
    DLedgerMmapFileStore#updateCommittedIndex
    public void updateCommittedIndex(long term, long newCommittedIndex) {    // @1
        if (newCommittedIndex == -1
                || ledgerEndIndex == -1
                || term < memberState.currTerm()
                || newCommittedIndex == this.committedIndex) {
    // @2
                return;
        }
        if (newCommittedIndex < this.committedIndex
                || newCommittedIndex < this.ledgerBeginIndex) {
   // @3
            logger.warn("[MONITOR]Skip update committed index for new={} < old={} or new={} < beginIndex={}", newCommittedIndex, this.committedIndex, newCommittedIndex, this.ledgerBeginIndex);
            return;
        }
        long endIndex = ledgerEndIndex;
        if (newCommittedIndex > endIndex) {
         // @4
                //If the node fall behind too much, the committedIndex will be larger than enIndex.
            newCommittedIndex = endIndex;
        }
        DLedgerEntry dLedgerEntry = get(newCommittedIndex);                    // @5
        PreConditions.check(dLedgerEntry != null, DLedgerResponseCode.DISK_ERROR);
        this.committedIndex = newCommittedIndex;
        this.committedPos = dLedgerEntry.getPos() + dLedgerEntry.getSize();       // @6
    }
```

代码@1：首先介绍一下方法的参数：

- long term
  主节点当前的投票轮次。

- long newCommittedIndex:
  主节点发送日志复制请求时的已提交日志序号。

代码@2：如果待更新提交序号为 -1 或 投票轮次小于从节点的投票轮次或主节点投票轮次等于从节点的已提交序号，则直接忽略本次提交动作。

代码@3：如果主节点的已提交日志序号小于从节点的已提交日志序号或待提交序号小于当前节点的最小有效日志序号，则输出警告日志[MONITOR]，并忽略本次提交动作。

代码@4：如果从节点落后主节点太多，则重置 提交索引为从节点当前最大有效日志序号。

代码@5：尝试根据待提交序号从从节点查找数据，如果数据不存在，则抛出 DISK_ERROR 错误。

黛米@6：更新 commitedIndex、committedPos 两个指针，DledgerStore 会定时将已提交指针刷入 checkpoint 文件，达到持久化 commitedIndex 指针的目的。

handleDoCompare

处理主节点发送过来的 COMPARE 请求，其实现也比较简单，最终调用 buildResponse 方法构造响应结果。

```
EntryHandler#buildResponse
private PushEntryResponse buildResponse(PushEntryRequest request, int code) {
    PushEntryResponse response = new PushEntryResponse();
    response.setGroup(request.getGroup());
    response.setCode(code);
    response.setTerm(request.getTerm());
    if (request.getType() != PushEntryRequest.Type.COMMIT) {
        response.setIndex(request.getEntry().getIndex());
    }
```

```
        response.setBeginIndex(dLedgerStore.getLedgerBeginIndex());
        response.setEndIndex(dLedgerStore.getLedgerEndIndex());
        return response;
    }
```

主要也是返回当前从几点的 ledgerBeginIndex、ledgerEndIndex 以及投票轮次，供主节点进行判断比较。

handleDoTruncate

handleDoTruncate 方法实现比较简单，删除从节点上 truncateIndex 日志序号之后的所有日志，具体调用 dLedgerStore 的 truncate 方法，由于其存储与 RocketMQ 的存储设计基本类似故本文就不在详细介绍，简单介绍其实现要点：根据日志序号，去定位到日志文件，如果命中具体的文件，则修改相应的读写指针、刷盘指针等，并将所在在物理文件之后的所有文件删除。大家如有兴趣，可以查阅笔者的《RocketMQ 技术内幕》第 4 章：RocketMQ 存储相关内容。

handleDoAppend

```
    private void handleDoAppend(long writeIndex, PushEntryRequest request,
        CompletableFuture<PushEntryResponse> future) {
        try {
            PreConditions.check(writeIndex == request.getEntry().getIndex(), DLedgerResp
onseCode.INCONSISTENT_STATE);
            DLedgerEntry entry = dLedgerStore.appendAsFollower(request.getEntry(), req
uest.getTerm(), request.getLeaderId());
            PreConditions.check(entry.getIndex() == writeIndex, DLedgerResponseCode.IN
CONSISTENT_STATE);
            future.complete(buildResponse(request, DLedgerResponseCode.SUCCESS.get
Code()));
            dLedgerStore.updateCommittedIndex(request.getTerm(), request.getCommitInd
ex());
        } catch (Throwable t) {
            logger.error("[HandleDoWrite] writeIndex={}", writeIndex, t);
            future.complete(buildResponse(request, DLedgerResponseCode.INCONSISTE
NT_STATE.getCode()));
        }
    }
```

其实现也比较简单，调用 DLedgerStore 的 appendAsFollower 方法进行日志的追加，与 appendAsLeader 在日志存储部分相同，只是从节点无需再转发日志。

checkAbnormalFuture

该方法是本节的重点，doWork 的从服务器存储的最大有效日志序号(ledgerEndIndex) + 1 序号，尝试从待写请求中获取不到对应的请求时调用，这种情况也很常见，例如主节点并么有将最新的数据 PUSH 给从节点。接下来我们详细来看看该方法的实现细节。

```
EntryHandler#checkAbnormalFuture
if (DLedgerUtils.elapsed(lastCheckFastForwardTimeMs) < 1000) {
    return;
}
lastCheckFastForwardTimeMs  = System.currentTimeMillis();
if (writeRequestMap.isEmpty()) {
    return;
}
```

Step1：如果上一次检查的时间距现在不到 1s，则跳出；如果当前没有积压的 append 请求，同样跳出，因为可以同样明确的判断出主节点还未推送日志。

```
EntryHandler#checkAbnormalFuture
for (Pair<PushEntryRequest, CompletableFuture<PushEntryResponse>> pair : writeRequestMap.values()) {
    long index = pair.getKey().getEntry().getIndex();                    // @1
    //Fall behind
    if (index <= endIndex) {                                                    // @2
        try {
            DLedgerEntry local = dLedgerStore.get(index);
            PreConditions.check(pair.getKey().getEntry().equals(local), DLedgerResponseCode.INCONSISTENT_STATE);
            pair.getValue().complete(buildResponse(pair.getKey(), DLedgerResponseCode.SUCCESS.getCode()));
            logger.warn("[PushFallBehind]The leader pushed an entry index={} smaller than current ledgerEndIndex={}, maybe the last ack is missed", index, endIndex);
        } catch (Throwable t) {
```

```
        logger.error("[PushFallBehind]The leader pushed an entry index={} smaller than cur
rent ledgerEndIndex={}, maybe the last ack is missed", index, endIndex, t);
                    pair.getValue().complete(buildResponse(pair.getKey(), DLedgerResponseC
ode.INCONSISTENT_STATE.getCode()));
                }
                writeRequestMap.remove(index);
                continue;
            }
            //Just OK
            if (index ==  endIndex + 1) {     // @3
                //The next entry is coming, just return
                return;
            }
            //Fast forward
            TimeoutFuture<PushEntryResponse> future  = (TimeoutFuture<PushEntryRespon
se>) pair.getValue();     // @4
            if (!future.isTimeOut()) {
                continue;
            }
            if (index < minFastForwardIndex) {
                                                                    // @5
                minFastForwardIndex = index;
            }
        }
```

Step2：遍历当前待写入的日志追加请求(主服务器推送过来的日志复制请求)，找到需要快速快进的的索引。其关键实现点如下：

代码@1：首先获取待写入日志的序号。

代码@2：如果待写入的日志序号小于从节点已追加的日志(endIndex)，并且日志的确已存储在从节点，则返回成功，并输出警告日志【PushFallBehind】，继续监测下一条待写入日志。

代码@3：如果待写入 index 等于 endIndex + 1，则结束循环，因为下一条日志消息已经在待写入队列中，即将写入。

代码@4：如果待写入 index 大于 endIndex + 1，并且未超时，则直接检查下一条待写入日志。

代码@5：如果待写入 index 大于 endIndex + 1，并且已经超时，则记录该索引，使用 minFastForwardIndex 存储。

```
EntryHandler#checkAbnormalFuture
if (minFastForwardIndex == Long.MAX_VALUE) {
    return;
}
Pair<PushEntryRequest, CompletableFuture<PushEntryResponse>> pair = writeRequ
estMap.get(minFastForwardIndex);
if (pair == null) {
    return;
}
```

Step3：如果未找到需要快速失败的日志序号或 writeRequestMap 中未找到其请求，则直接结束检测。

```
EntryHandler#checkAbnormalFuture
logger.warn("[PushFastForward] ledgerEndIndex={} entryIndex={}", endIndex, minFastF
orwardIndex);
pair.getValue().complete(buildResponse(pair.getKey(), DLedgerResponseCode.INCONS
ISTENT_STATE.getCode()));
```

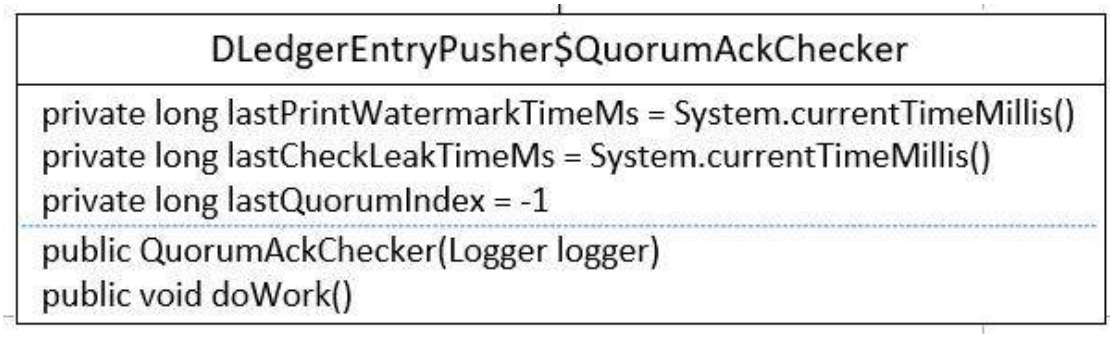
Step4：则向主节点报告从节点已经与主节点发生了数据不一致，从节点并没有写入序号 minFastForwardIndex 的日志。如果主节点收到此种响应，将会停止日志转发，转而向各个从节点发送 COMPARE 请求，从而使数据恢复一致。

行为至此，已经详细介绍了主服务器向从服务器发送请求，从服务做出响应，那接下来就来看一下，服务端收到响应结果后的处理，我们要知道主节点会向它所有的从节点传播日志，主节点需要在指定时间内收到超过集群一半节点的确认，才能认为日志写入成功，那我们接下来看一下其实现过程。

四、QuorumAckChecker

日志复制投票器，一个日志写请求只有得到集群内的的大多数节点的响应，日志才会被提交。

1. 类图



其核心属性如下：

- long lastPrintWatermarkTimeMs
  上次打印水位线的时间戳，单位为毫秒。

- long lastCheckLeakTimeMs
  上次检测泄漏的时间戳，单位为毫秒。

- long lastQuorumIndex
  已投票仲裁的日志序号。

2. doWork 详解

```
    QuorumAckChecker#doWork
    if (DLedgerUtils.elapsed(lastPrintWatermarkTimeMs) > 3000) {
        logger.info("[{}][{}] term={} ledgerBegin={} ledgerEnd={} committed={} watermarks={}
",
                memberState.getSelfId(), memberState.getRole(), memberState.currTerm(),
 dLedgerStore.getLedgerBeginIndex(), dLedgerStore.getLedgerEndIndex(), dLedgerStore.g
etCommittedIndex(), JSON.toJSONString(peerWaterMarksByTerm));
```

```
        lastPrintWatermarkTimeMs = System.currentTimeMillis();
    }
```

Step1：如果离上一次打印 watermak 的时间超过 3s，则打印一下当前的 term、ledgerBegin、ledgerEnd、committed、peerWaterMarksByTerm 这些数据日志。

```
    QuorumAckChecker#doWork
    if (!memberState.isLeader()) {    // @2
        waitForRunning(1);
        return;
    }
```

Step2：如果当前节点不是主节点，直接返回，不作为。

```
    QuorumAckChecker#doWork
    if (pendingAppendResponsesByTerm.size() > 1) {    // @1
        for (Long term : pendingAppendResponsesByTerm.keySet()) {
            if (term == currTerm) {
                continue;
            }
            for (Map.Entry<Long, TimeoutFuture<AppendEntryResponse>> futureEntry : pendingAppendResponsesByTerm.get(term).entrySet()) {
                AppendEntryResponse response = new AppendEntryResponse();
                response.setGroup(memberState.getGroup());
                response.setIndex(futureEntry.getKey());
                response.setCode(DLedgerResponseCode.TERM_CHANGED.getCode());
                response.setLeaderId(memberState.getLeaderId());
                logger.info("[TermChange] Will clear the pending response index={} for term changed from {} to {}", futureEntry.getKey(), term, currTerm);
                futureEntry.getValue().complete(response);
            }
            pendingAppendResponsesByTerm.remove(term);
        }
    }
    if (peerWaterMarksByTerm.size() > 1) {
        for (Long term : peerWaterMarksByTerm.keySet()) {
            if (term == currTerm) {
                continue;
            }
            logger.info("[TermChange] Will clear the watermarks for term changed from {} to {}", term, currTerm);
            peerWaterMarksByTerm.remove(term);
        }}
```

Step3：清理 pendingAppendResponsesByTerm、peerWaterMarksByTerm 中本次投票轮次的数据，避免一些不必要的内存使用。

```
Map<String, Long> peerWaterMarks = peerWaterMarksByTerm.get(currTerm);
long quorumIndex = -1;
for (Long index : peerWaterMarks.values()) {  // @1
    int num = 0;
    for (Long another : peerWaterMarks.values()) {  // @2
        if (another >= index) {
            num++;
        }
    }
    if (memberState.isQuorum(num) && index > quorumIndex) {  // @3
        quorumIndex = index;
    }
}
dLedgerStore.updateCommittedIndex(currTerm, quorumIndex);  // @4
```

Step4：根据各个从节点反馈的进度，进行仲裁，确定已提交序号。为了加深对这段代码的理解，再来啰嗦一下 peerWaterMarks 的作用，存储的是各个从节点当前已成功追加的日志序号。例如一个三节点的 DLedger 集群，peerWaterMarks 数据存储大概如下：

```
{
 “dledger_group_01_0” : 100,
"dledger_group_01_1" : 101,
}
```

其中 dledger_group_01_0 为从节点 1 的 ID，当前已复制的序号为 100，而 dledger_group_01_1 为节点 2 的 ID，当前已复制的序号为 101。再加上主节点，如何确定可提交序号呢?

代码@1:首先遍历 peerWaterMarks 的 value 集合,即上述示例中的 {100, 101}，用临时变量 index 来表示待投票的日志序号，需要集群内超过半数的节点的已复制序号超过该值，则该日志能被确认提交。

代码@2：遍历 peerWaterMarks 中的所有已提交序号，与当前值进行比较，如果节点的已提交序号大于等于待投票的日志序号(index)，num 加一，表示投赞成票。

代码@3：对 index 进行仲裁，如果超过半数 并且 index 大于 quorumIndex，更新 quorumIndex 的值为 index。quorumIndex 经过遍历的，得出当前最大的可提交日志序号。

代码@4：更新 committedIndex 索引，方便 DLedgerStore 定时将 committedIndex 写入 checkpoint 中。

```
    ConcurrentMap<Long, TimeoutFuture<AppendEntryResponse>> responses = pending
AppendResponsesByTerm.get(currTerm);
    boolean needCheck = false;
    int ackNum = 0;
    if (quorumIndex >= 0) {
        for (Long i = quorumIndex; i >= 0; i--) {  // @1
            try {
                CompletableFuture<AppendEntryResponse> future = responses.remove(i);
  // @2
                if (future == null) {
                                    // @3
                    needCheck = lastQuorumIndex != -1 && lastQuorumIndex != quoruml
ndex && i != lastQuorumIndex;
                    break;
                } else if (!future.isDone()) {
                                // @4
                    AppendEntryResponse response = new AppendEntryResponse();
                    response.setGroup(memberState.getGroup());
                    response.setTerm(currTerm);
                    response.setIndex(i);
                    response.setLeaderId(memberState.getSelfId());
                    response.setPos(((AppendFuture) future).getPos());
                    future.complete(response);
                }
                ackNum++;
                                    // @5
            } catch (Throwable t) {
                logger.error("Error in ack to index={} term={}", i, currTerm, t);
            }
        }
    }
```

Step5：处理 quorumIndex 之前的挂起请求，需要发送响应到客户端,其实现步骤：

代码@1：从 quorumIndex 开始处理，没处理一条，该序号减一，直到大于 0 或主动退出，请看后面的退出逻辑。

代码@2：responses 中移除该日志条目的挂起请求。

代码@3：如果未找到挂起请求，说明前面挂起的请求已经全部处理完毕，准备退出，退出之前再 设置 needCheck 的值，其依据如下(三个条件必须同时满足)：

- 最后一次仲裁的日志序号不等于-1。
- 并且最后一次不等于本次新仲裁的日志序号。
- 最后一次仲裁的日志序号不等于最后一次仲裁的日志。正常情况一下，条件一、条件二通常为 true，但这一条大概率会返回 false。
  - 即 needCheck 的含义是是否需要检查请求泄漏。

代码@4：向客户端返回结果。

代码@5：ackNum，表示本次确认的数量。

```
    if (ackNum == 0) {
        for (long i = quorumIndex + 1; i < Integer.MAX_VALUE; i++) {
            TimeoutFuture<AppendEntryResponse> future = responses.get(i);
            if (future == null) {
                break;
            } else if (future.isTimeOut()) {
                AppendEntryResponse response = new AppendEntryResponse();
                response.setGroup(memberState.getGroup());
                response.setCode(DLedgerResponseCode.WAIT_QUORUM_ACK_TIMEO
UT.getCode());
                response.setTerm(currTerm);
                response.setIndex(i);
                response.setLeaderId(memberState.getSelfId());
                future.complete(response);
            } else {
                break;
            }
        }
        waitForRunning(1);
    }
```

Step6：如果本次确认的个数为 0，则尝试去判断超过该仲裁序号的请求，是否已经超时，如果已超时，则返回超时响应结果。

```
    if (DLedgerUtils.elapsed(lastCheckLeakTimeMs) > 1000 || needCheck) {
        updatePeerWaterMark(currTerm, memberState.getSelfId(), dLedgerStore.getLedge
rEndIndex());
        for (Map.Entry<Long, TimeoutFuture<AppendEntryResponse>> futureEntry : resp
onses.entrySet()) {
            if (futureEntry.getKey() < quorumIndex) {
                AppendEntryResponse response = new AppendEntryResponse();
                response.setGroup(memberState.getGroup());
                response.setTerm(currTerm);
                response.setIndex(futureEntry.getKey());
                response.setLeaderId(memberState.getSelfId());
                response.setPos(((AppendFuture) futureEntry.getValue()).getPos());
                futureEntry.getValue().complete(response);
                responses.remove(futureEntry.getKey());
            }
        }
        lastCheckLeakTimeMs = System.currentTimeMillis();
    }
```

Step7：检查是否发送泄漏。其判断泄漏的依据是如果挂起的请求的日志序号小于已提交的序号，则移除。

Step8：一次日志仲裁就结束了，最后更新 lastQuorumIndex 为本次仲裁的的新的提交值。

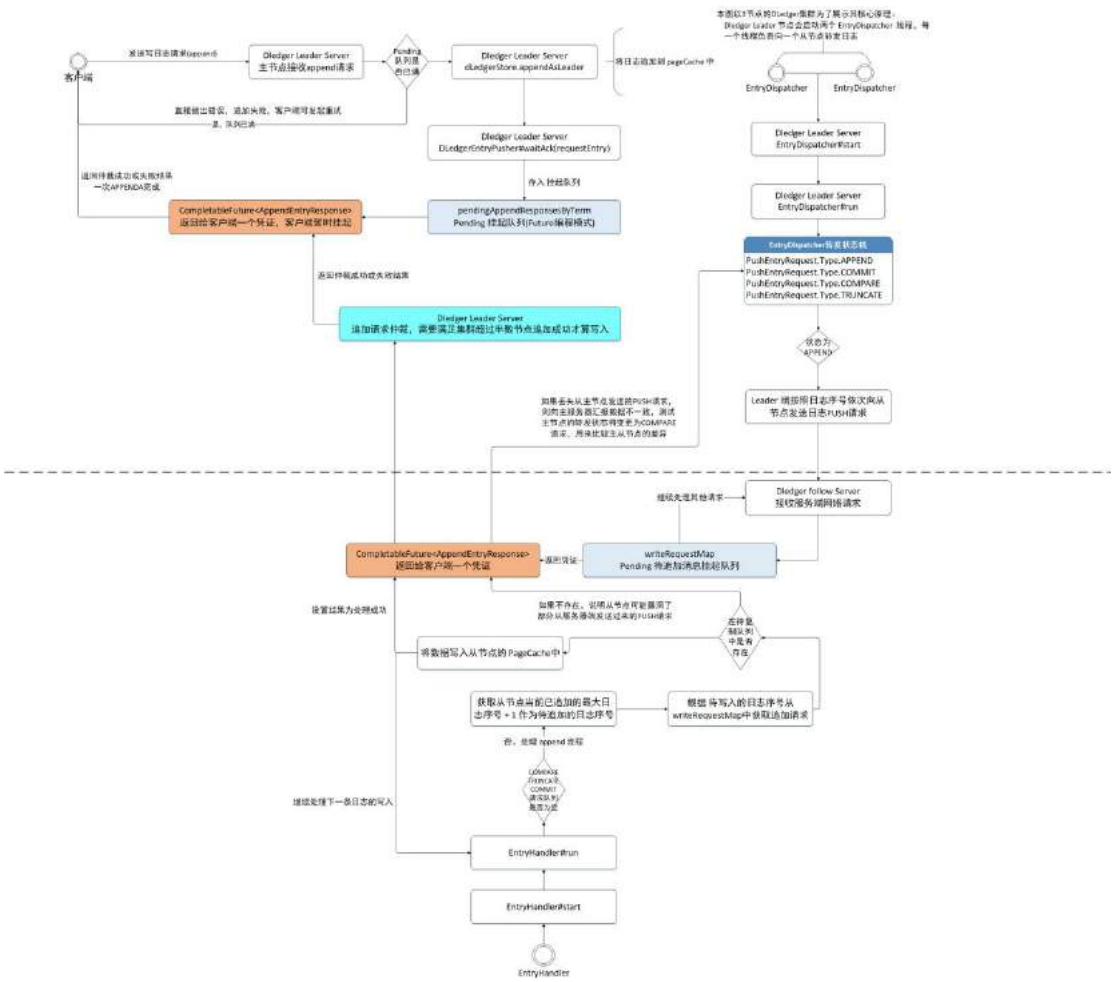
关于 DLedger 的日志复制部分就介绍到这里了。本文篇幅较长，看到这里的各位亲爱的读者朋友们，麻烦点个在看，谢谢。

2.8 基于 raft 协议的 RocketMQ DLedger 多副本日志复制实现原理

经过前面 3 篇源码分析 RocketMQ DLedger 多副本实现机制，可能有不少读者朋友们觉得源码阅读较为枯燥，看的有点云里雾里，本篇将首先梳理一下 RocketMQ DLedger 多副本的实现流程图，然后思考在异常情况下如何保证数据一致性。

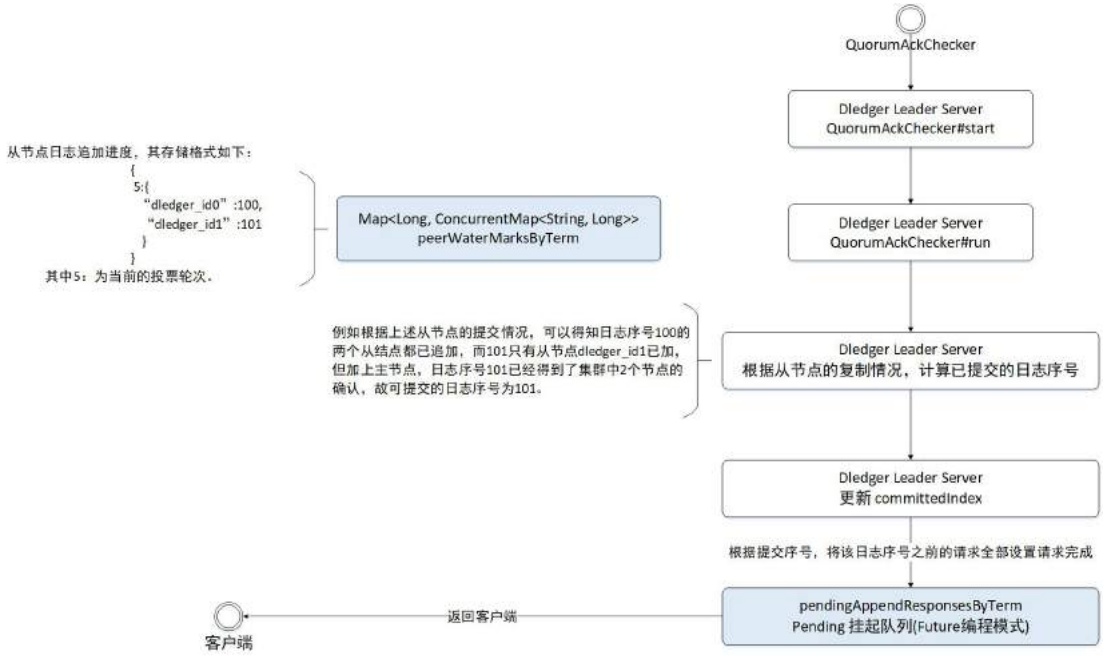
一、RocketMQ DLedger 多副本日志复制流程图

1. RocketMQ DLedger 日志复制(append) 请求流程图



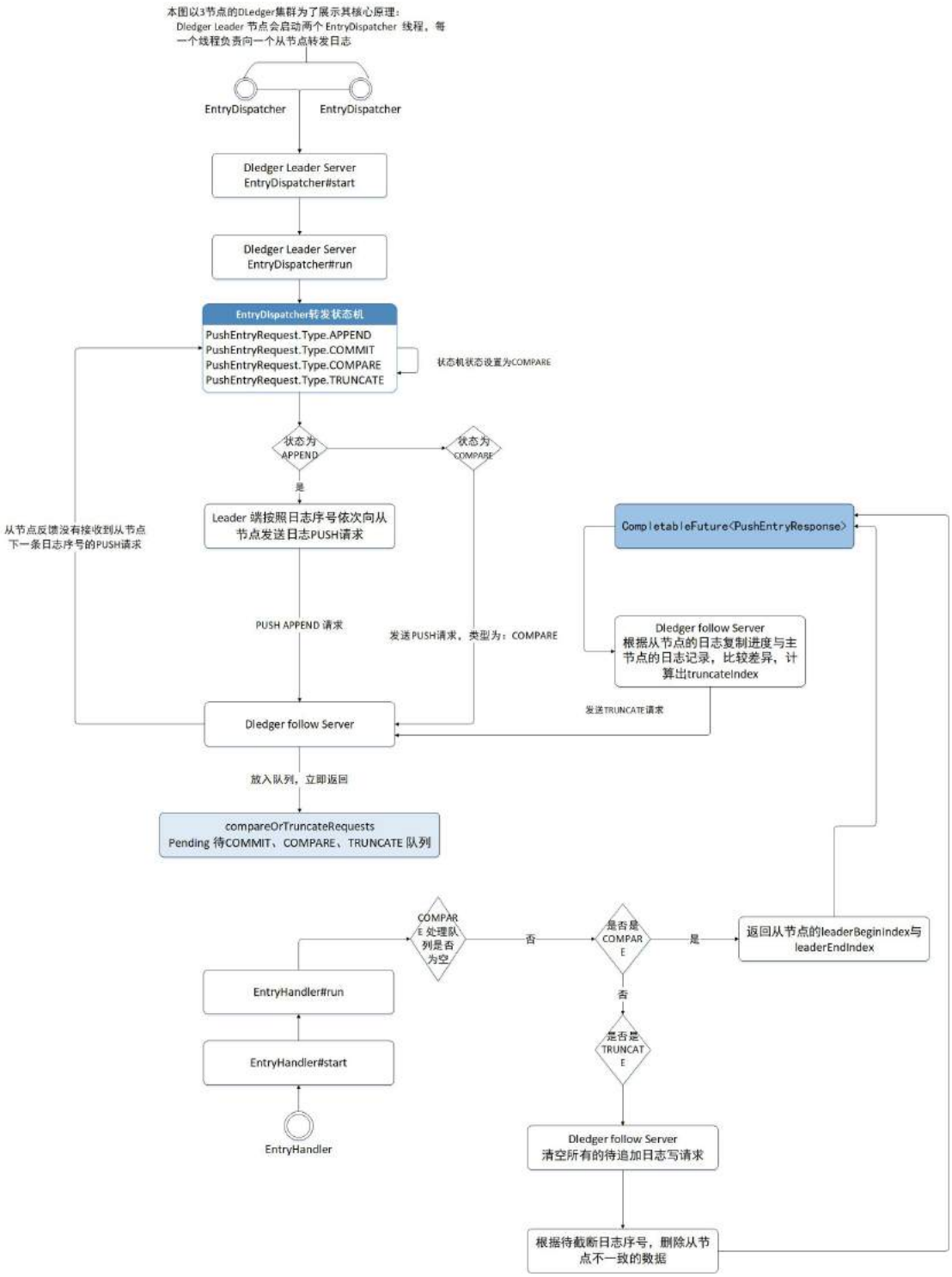
该图的详细分析，请参考文章：

《RocketMQ DLedger 日志仲裁流程图》



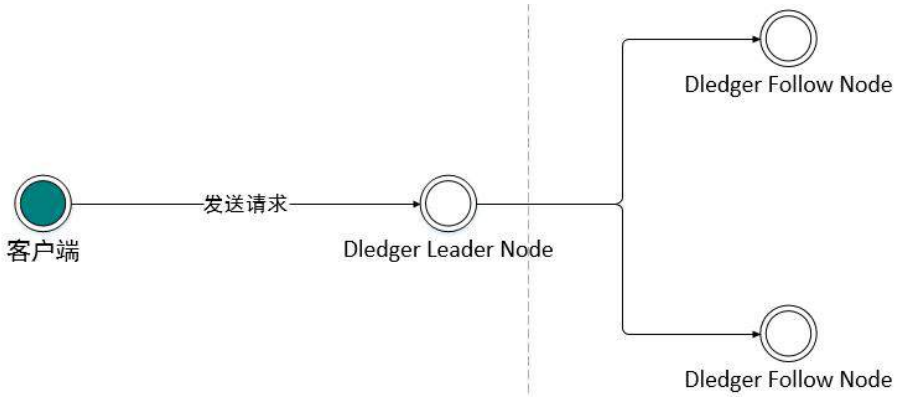
该图的详细分析，请参考文章：

《RocketMQ DLedger 多副本日志》



该图的详细分析，请参考文章：

《RocketMQ DLedger 多副本日志复制实现要点》



上图就是一个简易的日志复制的模型：图中客户端向 DLedger 集群发起一个写请求，集群中的 Leader 节点来处理写请求，首先数据先存入 Leader 节点，然后需要广播给它的所有从节点，从节点接收到 Leader 节点的数据推送对数据进行存储，然后向主节点汇报存储的结果，Leader 节点会对该日志的存储结果进行仲裁，如果超过集群数量的一半都成功存储了该数据，主节点则向客户端返回写入成功，否则向客户端写入写入失败。

接下来我们将来探讨日志复制的核心设计要点。

1. 日志编号

为了方便对日志进行管理与辨别，raft 协议为一条一条的消息进行编号，每一条消息达到主节点时会生成一个全局唯一的递增号，这样可以根据日志序号来快速的判断数据在主从复制过程中数据是否一致，在 DLedger 的实现中对应 DLedgerMemoryStore 中的 ledgerBeginIndex、ledgerEndIndex，分别表示当前节点最小的日志序号与最大的日志序号，下一条日志的序号为 ledgerEndIndex + 1 。

与日志序号还与一个概念绑定的比较紧密，即当前的投票轮次。

2. 追加与提交机制

请思考如下问题，Leader 节点收到客户端的数据写入请求后，通过解析请求，提取数据部分，构建日志对象，并生成日志序号，用 seq 表示，然后存储到 Leader 节点中，然后将日志广播(推送)到其从节点，由于这个过程中存在网络时延，如果此时客户端向主节点查询 seq 的日志，由于日志已经存储在 Leader 节点中了，如果直接返回给客户端显然是有问题的，那该如何来避免这种情况的发生呢?

为了解决上述问题，DLedger 的实现(应该也是 raft 协议的一部分)引入了已提交指针(committedIndex)。即当主节点收到客户端请求时，首先先将数据存储，但此时数据是未提交的，此过程可以称之为追加，此时客户端无法访问，只有当集群内超过半数的节点都将日志追加完成后，才会更新 committedIndex 指针，得以是数据能否客户端访问。

一条日志要能被提交的充分必要条件是日志得到了集群内超过半数节点成功追加，才能被认为已提交。

3. 日志一致性如何保证

从上文得知，一个拥有 3 个节点的 DLedger 集群，只要主节点和其中一个从节点成功追加日志，则认为已提交，客户端即可通过主节点访问。由于部分数据存在延迟，在 DLedger 的实现中，读写请求都将由 Leader 节点来负责。那落后的从节点如何再次跟上集群的步骤呢?

要重新跟上主节点的日志记录，首先要知道的是如何判断从节点已丢失数据呢?

DLedger 的实现思路是，DLedger 会按照日志序号向从节点源源不断的转发日志，从节点接收后将这些待追加的数据放入一个待写队列中。关键中的关键：从节点并不是从挂起队列中处理一个一个的追加请求，而是首先查阅从节点当前已追加的最大日志序号，用 ledgerEndIndex 表示，然后尝试追加 (ledgerEndIndex + 1)的日志，用该序号从代写队列中查找，如果该队列不为空，并且没有 (ledgerEndIndex + 1)的日志条目，说明从节点未接收到这条日志，发生了数据缺失。然后从节点在响应主节点 append 的请求时会告知数据不一致，然后主节点的日志转发线程其状态会变更为 COMPARE，将向该从节点发送 COMPARE 命令，用来比较主从节点的数据差异，根据比较的差异重新从主节点同步数据或删除从节点上多余的数据，最终达到一致。于此同时，主节点也会对 PUSH 超时推送的消息发起重推，尽最大可能帮助从节点及时更新到主节点的数据。

更多问题，欢迎大家留言与我一起探讨。

2.9 源码分析 RocketMQ DLedger 多副本存储实现

RocketMQ DLedger 的存储实现思路与 RocketMQ 的存储实现思路相似，本文就不再从源码角度详细剖析其实现，只是点出其实现关键点。我们不妨简单回顾一下 CommitLog 文件、ConsumeQueue 文件设计思想。

其文件组成形式如下：

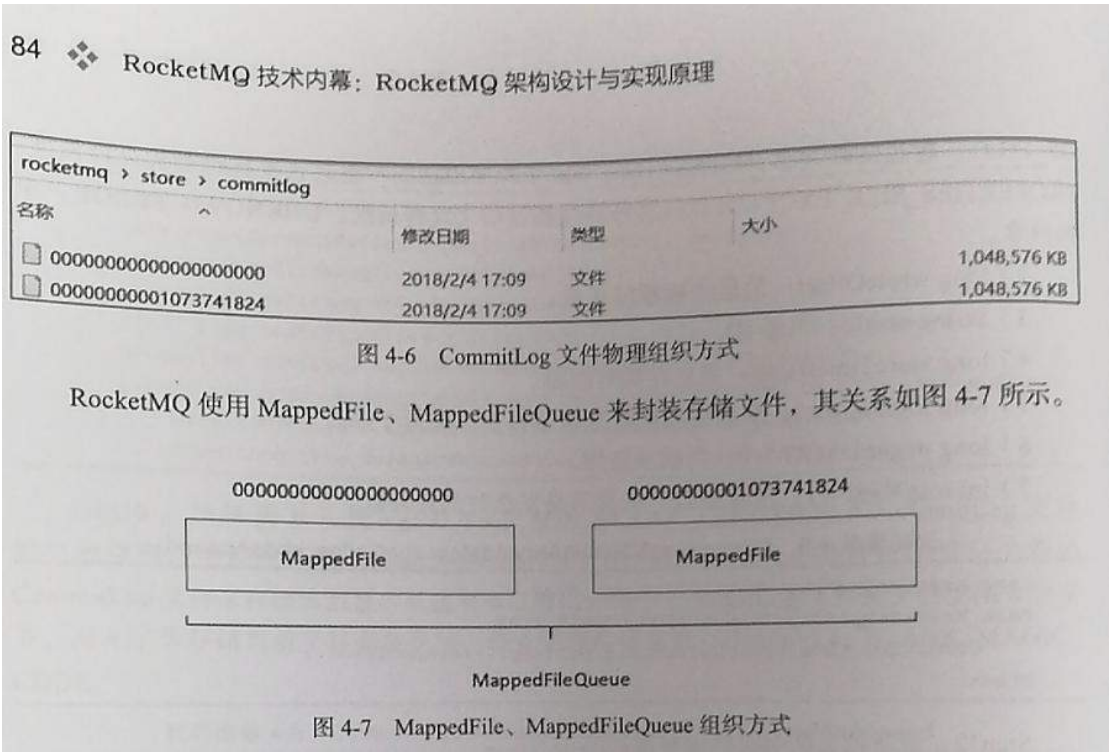


图 4-6 CommitLog 文件物理组织方式

RocketMQ 使用 MappedFile、MappedFileQueue 来封装存储文件，其关系如图 4-7 所示。

图 4-7 MappedFile、MappedFileQueue 组织方式

正如上图所示，多个 commitlog 文件组成一个逻辑上的连续文件，使用 MappedFileQueue 表示，单个 commitlog 文件使用 MappedFile 表示。

温馨提示：如果想详细了解 RocketMQ 关于存储部分的讲解，可以关注笔者的《RocketMQ 技术内幕》一书。

一、DLedger 存储相关类图

1. DLedgerStore

存储抽象类，定义如下核心方法：

- public abstract DLedgerEntry appendAsLeader(DLedgerEntry entry)
  向主节点追加日志(数据)。

- public abstract DLedgerEntry appendAsFollower(DLedgerEntry entry, long leaderTerm, String leaderId)
  向从节点同步日志。

- public abstract DLedgerEntry get(Long index)
  根据日志下标查找日志。

- public abstract long getCommittedIndex()
  获取已提交的下标。

- public abstract long getLedgerEndTerm()
  获取 Leader 当前最大的投票轮次。

- public abstract long getLedgerEndIndex()
  获取 Leader 下一条日志写入的下标。

- public abstract long getLedgerBeginIndex()
  获取 Leader 第一条消息的下标。

- public void updateCommittedIndex(long term, long committedIndex)
  更新 commitedIndex 的值，为空实现，由具体的存储子类实现。

- protected void updateLedgerEndIndexAndTerm()
  更新 Leader 维护的 ledgerEndIndex 和 ledgerEndTerm 。

- public void flush()
  刷写，空方法，由具体子类实现。

- public long truncate(DLedgerEntry entry, long leaderTerm, String leaderId)
  删除日志，空方法，由具体子类实现。

- public void startup()
  启动存储管理器，空方法，由具体子类实现。

- public void shutdown()
  关闭存储管理器，空方法，由具体子类实现。

2. DLedgerMemoryStore

Dledger 基于内存实现的日志存储。

3. DLedgerMmapFileStore

基于文件内存映射机制的存储实现。其核心属性如下：

- long ledgerBeginIndex = -1
  日志的起始索引，默认为 -1。

- long ledgerEndIndex = -1
  下一条日志下标，默认为 -1。

- long committedIndex = -1
  已提交的日志索引。

- long ledgerEndTerm
  当前最大的投票轮次。

- DLedgerConfig dLedgerConfig
  DLedger 的配置信息。

- MemberState memberState
  状态机。

- MmapFileList dataFileList
  日志文件(数据文件)的内存映射 Queue。

- MmapFileList indexFileList
  索引文件的内存映射文件集合。（可对标 RocketMQ MappedFIleQueue）。

- ThreadLocal<ByteBuffer> localIndexBuffer
  ThreadLocal<ByteBuffer> localEntryBuffer

- FlushDataService flushDataService
  数据文件刷盘线程。

- CleanSpaceService cleanSpaceService
  清除过期日志文件线程。

- boolean isDiskFull = false
  磁盘是否已满。

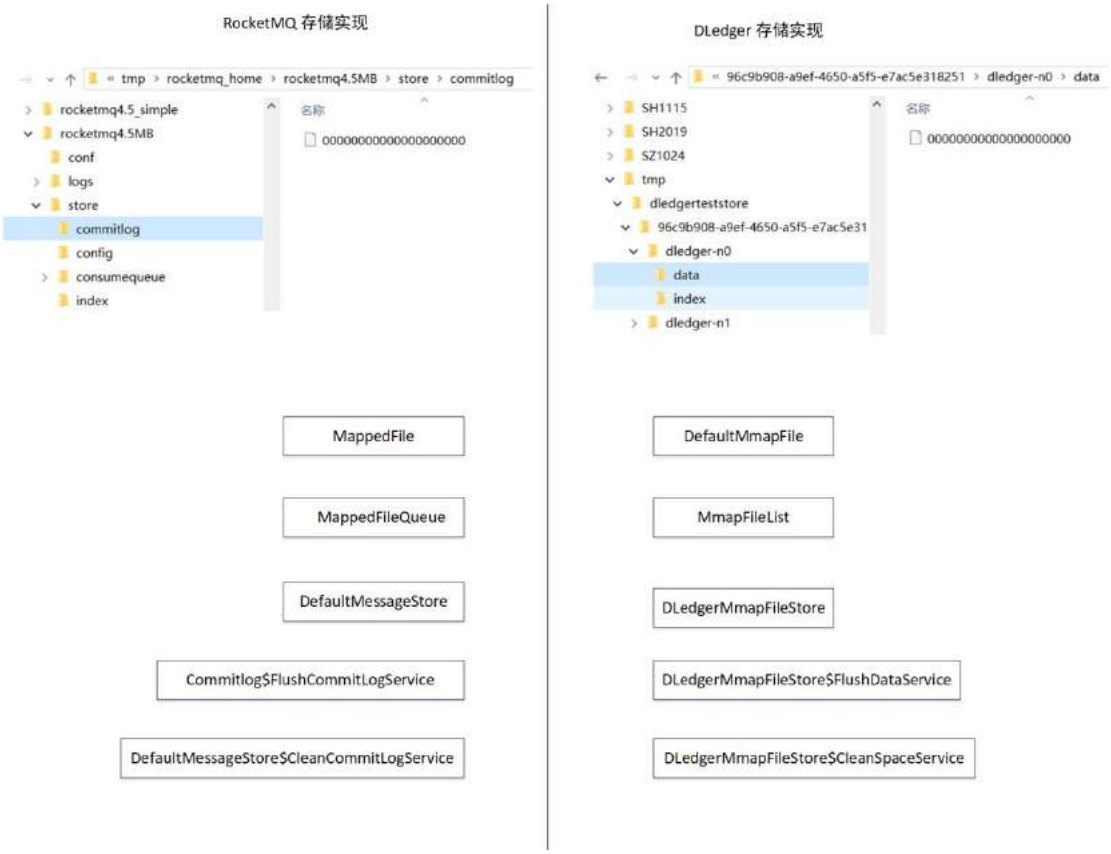
- long lastCheckPointTimeMs
  上一次检测点（时间戳）。

- AtomicBoolean hasLoaded
  是否已经加载，主要用来避免重复加载(初始化)日志文件。

- AtomicBoolean hasRecovered
  是否已恢复。

二、DLedger 存储 对标 RocketMQ 存储

存储部分主要包含存储映射文件、消息存储格式、刷盘、文件加载与文件恢复、过期文件删除等，由于这些内容在 RocketMQ 存储部分都已详细介绍，故本文点到为止，其对应的参考映射如下：



在 RocketMQ 中使用 MappedFile 来表示一个物理文件，而在 DLedger 中使用 DefaultMmapFIle 来表示一个物理文件。

在 RocketMQ 中使用 MappedFile 来表示多个物理文件(逻辑上连续)，而在 DLedger 中则使用 MmapFileList。

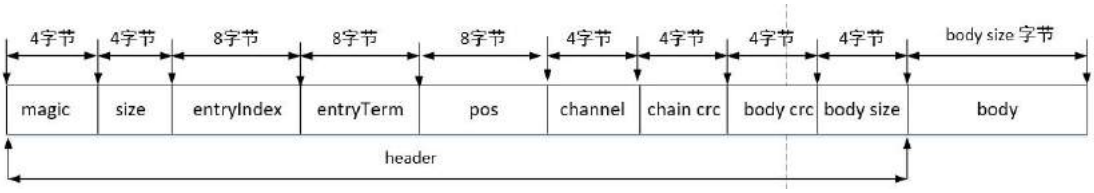
在 RocketMQ 中使用 DefaultMessageStore 来封装存储逻辑，而在 DLedger 中则使用 DLedgerMmapFileStore 来封装存储逻辑。

在 RocketMQ 中使用 Commitlog$FlushCommitLogService 来实现commitlog 文件的刷盘，而在 DLedger 中则使用 DLedgerMmapFileStore$FlushDataService 来实现文件刷盘。

在 RocketMQ 中使用 DefaultMessageStore$CleanCommitlogService 来实现 commitlog 过期文件的删除，而 DLedger 中则使用 DLedgerMmapFileStore$CleanSpaceService 来实现。

由于其实现原理相同，上述部分已经在《RocketMQ 技术内幕》第 4 章中详细剖析，故这里就不重复分析了。

三、DLedger 数据存储格式



存储格式字段的含义如下：

- magic

  魔数，4 字节。

- size

  条目总长度，包含 Header(协议头) + 消息体，占 4 字节。

- entryIndex

  当前条目的 index，占 8 字节。

- entryTerm

  当前条目所属的 投票轮次，占 8 字节。

- pos

  该条目的物理偏移量，类似于 commitlog 文件的物理偏移量，占 8 字节。

- channel

  保留字段，当前版本未使用，占 4 字节。

- chain crc

  当前版本未使用，占 4 字节。

- body crc

  body 的 CRC 校验和，用来区分数据是否损坏，占 4 字节。

- body size

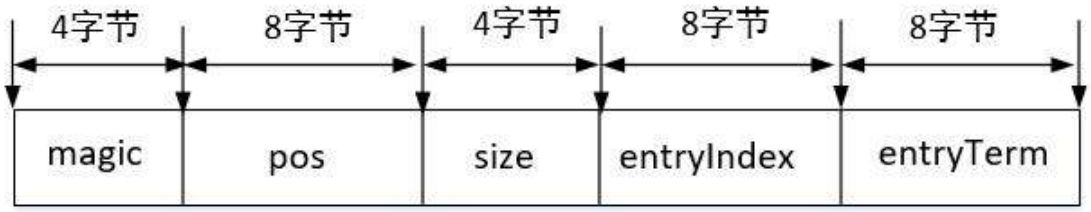
  用来存储 body 的长度，占 4 个字节。

- body

  具体消息的内容。

源码参考点：DLedgerMmapFileStore#recover、DLedgerEntry、DLedgerEntryCoder。

四、DLedger 索引存储格式



即一个索引条目占 32 个字节。

五、思考

DLedger 存储相关就介绍到这里，为了与大家增加互动，特提出如下两个思考题，欢迎与作者互动，这些问题将在该系列的后面文章专题探讨。

- DLedger 如果整合 RocketMQ 中的 commitlog 文件，使之支持多副本?
- 从老版本如何升级到新版本，需要考虑哪些因素呢?

2.10 源码分析 RocketMQ 整合 DLedger(多副本)实现平滑升级的设计技巧

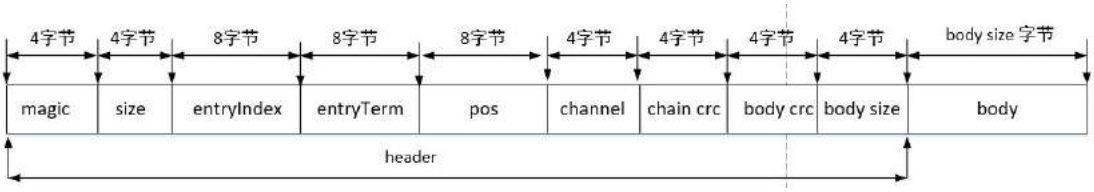
源码分析 RocketMQ DLedger 多副本系列已经进行到第 8 篇了，前面的章节主要是介绍了基于 raft 协议的选主与日志复制，从本篇开始将开始关注如何将 DLedger 应用到 RocketMQ 中。

摘要：详细分析了 RocketMQ DLedger 多副本(主从切换) 是如何整合到 RocketMQ 中，本文的行文思路首先结合已掌握的 DLedger 多副本相关的知识初步思考其实现思路，然后从 Broker 启动流程、DLedgerCommitlog 核心类的讲解，再从消息发送(追加)与消息查找来进一步探讨 DLedger 是如何支持平滑升级的。

一、阅读源码之前的思考

RocketMQ 的消息存储文件主要包括 commitlog 文件、consumequeue 文件与 Index 文件。commitlog 文件存储全量的消息，consumequeue、index 文件都是基于 commitlog 文件构建的。要使用 DLedger 来实现消息存储的一致性，应该关键是要实现 commitlog 文件的一致性，即 DLedger 要整合的对象应该是 commitlog 文件，即只需保证 raft 协议的复制组内各个节点的 commitlog 文件一致即可。

我们知道使用文件存储消息都会基于一定的存储格式，rocketmq 的 commitlog 一个条目就包含魔数、消息长度，消息属性、消息体等，而我们再来回顾一下 DLedger 日志的存储格式：



DLedger 要整合 commitlog 文件，是不是可以把 rocketmq 消息，即一个个 commitlog 条目整体当成 DLedger 的 body 字段即可。

还等什么，跟我一起来看源码吧！！！别急，再抛一个问题，DLedger 整合 RocketMQ commitlog，能不能做到平滑升级？

带着这些思考和问题，一起来探究 DLedger 是如何整合 RocketMQ 的。

二、从 Broker 启动流程看 DLedger

温馨提示：本文不会详细介绍 Broker 端的启动流程，只会点出在启动过程中与 DLedger 相关的代码，如想详细了解 Broker 的启动流程，建议关注笔者的《RocketMQ 技术内幕》一书。

Broker 涉及到 DLedger 相关关键点如下：

1. 构建 DefaultMessageStore

```
DefaultMessageStore 构造方法
if(messageStoreConfig.isEnableDLegerCommitLog()) {   // @1
    this.commitLog = new DLedgerCommitLog(this);
 else {
    this.commitLog = new CommitLog(this);                    // @2
}
```

代码@1：如果开启 DLedger ，commitlog 的实现类为 DLedgerCommitLog，也是本文需要关注的关键所在。

代码@2：如果未开启 DLedger，则使用旧版的 Commitlog 实现类。

2. 增加节点状态变更事件监听器

```
BrokerController#initialize
if (messageStoreConfig.isEnableDLegerCommitLog()) {
    DLedgerRoleChangeHandler roleChangeHandler = new DLedgerRoleChangeHandler(this, (DefaultMessageStore) messageStore);
    ((DLedgerCommitLog)((DefaultMessageStore) messageStore).getCommitLog()).getdLedgerServer().getdLedgerLeaderElector().addRoleChangeHandler(roleChangeHandler);
}
```

主要调用 LedgerLeaderElector 的 addRoleChanneHandler 方法增加 节点角色变更事件监听器，DLedgerRoleChangeHandler 是实现主从切换的另外一个关键点。

3. 调用 DefaultMessageStore 的 load 方法

```
DefaultMessageStore#load
// load Commit Log
result = result && this.commitLog.load();    // @1
// load Consume Queue
result = result && this.loadConsumeQueue();
if (result) {
    this.storeCheckpoint =  new StoreCheckpoint(StorePathConfigHelper.getStoreCheckpoint(this.messageStoreConfig.getStorePathRootDir()));
    this.indexService.load(lastExitOK);
    this.recover(lastExitOK);                                    // @2
    log.info("load over, and the max phy offset = {}", this.getMaxPhyOffset());
}
```

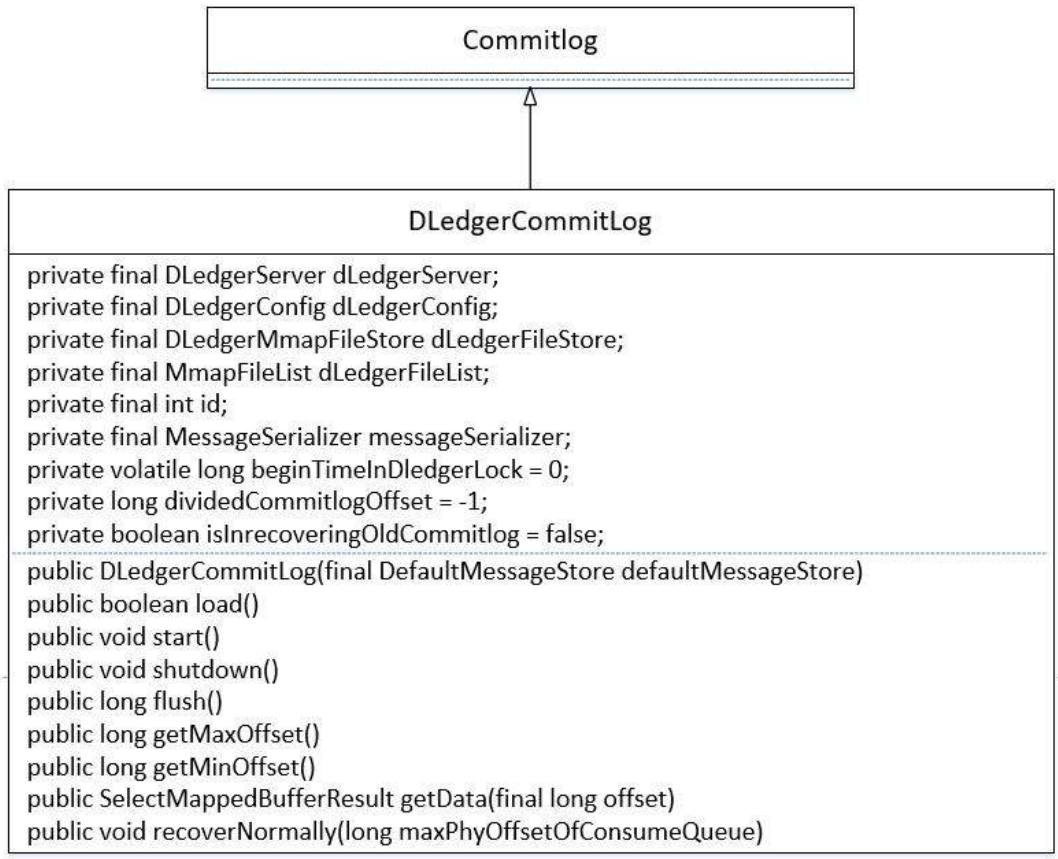
代码@1、@2 最终都是委托 commitlog 对象来执行，这里的关键又是如果开启了 DLedger，则最终调用的是 DLedgerCommitLog。

经过上面的铺垫，主角 DLedgerCommitLog “闪亮登场“了。

三、DLedgerCommitLog 详解

温馨提示：由于 Commitlog 的绝大部分方法都已经在《RocketMQ 技术内幕》一书中详细介绍了，并且 DLedgerCommitLog 的实现原理与 Commitlog 文件的实现原理类同，本文会一笔带过关于存储部分的实现细节。

1. 核心类图



DLedgerCommitlog 继承自 Commitlog。让我们一一来看一下它的核心属性：

- DLedgerServer dLedgerServer

  基于 raft 协议实现的集群内的一个节点，用 DLedgerServer 实例表示。

- DLedgerConfig dLedgerConfig

  DLedger 的配置信息。

- DLedgerMmapFileStore dLedgerFileStore

  DLedger 基于文件映射的存储实现。

- MmapFileList dLedgerFileList
  DLedger 所管理的存储文件集合，对比 RocketMQ 中的 MappedFileQueue。

- int id
  节点 ID，0 表示主节点，非 0 表示从节点

- MessageSerializer messageSerializer
  消息序列器。

- long beginTimeInDledgerLock = 0
  用于记录 消息追加的时耗(日志追加所持有锁时间)。

- long dividedCommitlogOffset = -1
  记录的旧 commitlog 文件中的最大偏移量，如果访问的偏移量大于它，则访问 dledger 管理的文件。

- boolean isInrecoveringOldCommitlog = false
  是否正在恢复旧的 commitlog 文件。

接下来我们将详细介绍 DLedgerCommitlog 各个核心方法及其实现要点。

2. 构造方法

```
public DLedgerCommitLog(final DefaultMessageStore defaultMessageStore) {
    super(defaultMessageStore);                    // @1
    dLedgerConfig = new DLedgerConfig();
    dLedgerConfig.setEnableDiskForceClean(defaultMessageStore.getMessageStoreConfig().isCleanFileForciblyEnable());
    dLedgerConfig.setStoreType(DLedgerConfig.FILE);
    dLedgerConfig.setSelfId(defaultMessageStore.getMessageStoreConfig().getdLegerSelfId());
    dLedgerConfig.setGroup(defaultMessageStore.getMessageStoreConfig().getdLegerGroup());
    dLedgerConfig.setPeers(defaultMessageStore.getMessageStoreConfig().getdLegerPeers());
```

```
        dLedgerConfig.setStoreBaseDir(defaultMessageStore.getMessageStoreConfig().get
StorePathRootDir());
        dLedgerConfig.setMappedFileSizeForEntryData(defaultMessageStore.getMessageS
toreConfig().getMapedFileSizeCommitLog());
        dLedgerConfig.setDeleteWhen(defaultMessageStore.getMessageStoreConfig().getD
eleteWhen());
        dLedgerConfig.setFileReservedHours(defaultMessageStore.getMessageStoreConfig
().getFileReservedTime() + 1);
        id = Integer.valueOf(dLedgerConfig.getSelfId().substring(1)) + 1;                // @2
        dLedgerServer = new DLedgerServer(dLedgerConfig);                                //
 @3
        dLedgerFileStore = (DLedgerMmapFileStore) dLedgerServer.getdLedgerStore();
        DLedgerMmapFileStore.AppendHook appendHook = (entry, buffer, bodyOffset) ->
 {
            assert bodyOffset == DLedgerEntry.BODY_OFFSET;
            buffer.position(buffer.position() + bodyOffset + MessageDecoder.PHY_PO
S_POSITION);
            buffer.putLong(entry.getPos() + bodyOffset);
        };
        dLedgerFileStore.addAppendHook(appendHook);    // @4
        dLedgerFileList = dLedgerFileStore.getDataFileList();
        this.messageSerializer = new MessageSerializer(defaultMessageStore.getMessage
StoreConfig().getMaxMessageSize());    // @5
    }
```

代码@1：调用父类 即 CommitLog 的构造函数，加载 ${ROCKETMQ_HOME}/store/comitlog 下的 commitlog 文件，以便兼容升级 DLedger 的消息。我们稍微看一下 CommitLog 的构造函数：

代码@2：构建 DLedgerConfig 相关配置属性，其主要属性如下：

- enableDiskForceClean

是否强制删除文件，取自 broker 配置属性 cleanFileForciblyEnable，默认为 true 。

- storeType

DLedger 存储类型，固定为 基于文件的存储模式。

dLegerSelfId

leader 节点的 id 名称，示例配置：n0，其配置要求第二个字符后必须是数字。

- dLegerGroup

DLeger group 的名称，建议与 broker 配置属性 brokerName 保持一致。

- dLegerPeers

DLeger Group 中所有的节点信息，其配置示例 n0-127.0.0.1:40911;n1-127.0.0.1:40912;n2-127.0.0.1:40913。多个节点使用分号隔开。

- storeBaseDir

设置 DLedger 的日志文件的根目录，取自 borker 配件文件中的 storePathRootDir ，即 RocketMQ 的数据存储根路径。

- mappedFileSizeForEntryData

设置 DLedger 的单个日志文件的大小，取自 broker 配置文件中的 mapedFileSizeCommitLog，即与 commitlog 文件的单个文件大小一致。

- deleteWhen

DLedger 日志文件的删除时间，取自 broker 配置文件中的 deleteWhen，默认为凌晨 4 点。

- fileReservedHours

DLedger 日志文件保留时长，取自 broker 配置文件中的 fileReservedHours，默认为 72h。

代码@3：根据 DLedger 配置信息创建 DLedgerServer，即创建 DLedger 集群节点，集群内各个节点启动后，就会触发选主。

代码@4：构建 appendHook 追加钩子函数，这是兼容 Commitlog 文件很关键的一步，后面会详细介绍其作用。

代码@5：构建消息序列化。

根据上述的流程图，构建好 DefaultMessageStore 实现后，就是调用其 load 方法，在启用 DLedger 机制后，会依次调用 DLedgerCommitlog 的 load、recover 方法。

3. load

```
public boolean load() {
    boolean result = super.load();
    if (!result) {
        return false;
    }
    return true;
}
```

DLedgerCommitLog 的 laod 方法实现比较简单，就是调用 其父类 Commitlog 的 load 方法，即这里也是为了启用 DLedger 时能够兼容以前的消息。

4. recover

在 Broker 启动时会加载 commitlog、consumequeue 等文件，需要恢复其相关是数据结构，特别是与写入、刷盘、提交等指针，其具体调用 recover 方法。

```
DLedgerCommitLog#recover
public void recoverNormally(long maxPhyOffsetOfConsumeQueue) {  // @1
    recover(maxPhyOffsetOfConsumeQueue);
}
```

首先会先恢复 consumequeue，得出 consumequeue 中记录的最大有效物理偏移量，然后根据该物理偏移量进行恢复。

接下来看一下该方法的处理流程与关键点。

```
DLedgerCommitLog#recover
dLedgerFileStore.load();
```

Step1：加载 DLedger 相关的存储文件，并一一构建对应的 MmapFile，其初始化三个重要的指针 wrotePosition、flushedPosition、committedPosition 三个指针为文件的大小。

```
DLedgerCommitLog#recover
if (dLedgerFileList.getMappedFiles().size() > 0) {
    dLedgerFileStore.recover();    // @1
    dividedCommitlogOffset = dLedgerFileList.getFirstMappedFile().getFileFromOffset();
 // @2
    MappedFile mappedFile = this.mappedFileQueue.getLastMappedFile();
    if (mappedFile != null) {
                                        // @3
        disableDeleteDledger();
    }
    long maxPhyOffset = dLedgerFileList.getMaxWrotePosition();
    // Clear ConsumeQueue redundant data
    if (maxPhyOffsetOfConsumeQueue >= maxPhyOffset) {        // @4
        log.warn("[TruncateCQ]maxPhyOffsetOfConsumeQueue({}) >= processOffset
({}), truncate dirty logic files", maxPhyOffsetOfConsumeQueue, maxPhyOffset);
        this.defaultMessageStore.truncateDirtyLogicFiles(maxPhyOffset);
    }
    return;
}
```

Step2：如果已存在 DLedger 的数据文件，则只需要恢复 DLedger 相关数据文建，因为在加载旧的 commitlog 文件时已经将其重要的数据指针设置为最大值。其关键实现点如下：

首先调用 DLedger 文件存储实现类 DLedgerFileStore 的 recover 方法，恢复管辖的 MMapFile 对象(一个文件对应一个 MMapFile 实例)的相关指针，其实现方法与 RocketMQ 的 DefaultMessageStore 的恢复过程类似。

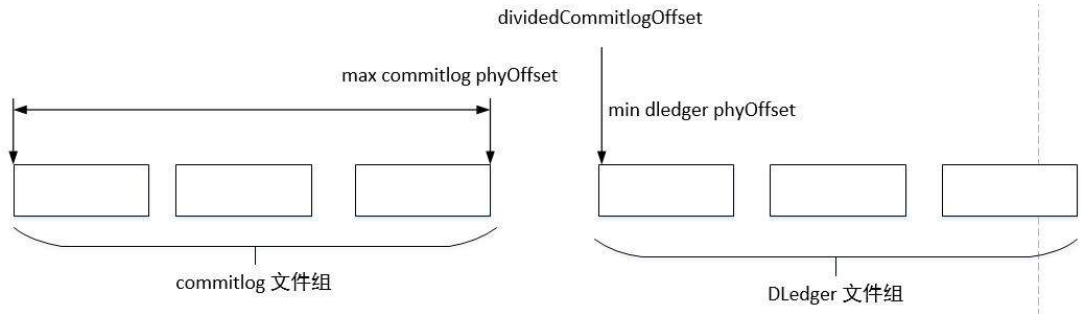
设置 dividedCommitlogOffset 的值为 DLedger 中所有物理文件的最小偏移量。操作消息的物理偏移量小于该值，则从 commitlog 文件中查找；物理偏移量大于等于该值的话则从 DLedger 相关的文件中查找消息。

如果存在旧的 commitlog 文件，则禁止删除 DLedger 文件，其具体做法就是禁止强制删除文件，并将文件的有效存储时间设置为 10 年。

如果 consumequeue 中存储的最大物理偏移量大于 DLedger 中最大的物理偏移量，则删除多余的 consumequeue 文件。

温馨提示：为什么当存在 commitlog 文件的情况下，不能删除 DLedger 相关的日志文件呢?

因为在此种情况下，如果 DLedger 中的物理文件有删除，则物理偏移量会断层。



正常情况下， maxCommitlogPhyOffset 与 dividedCommitlogOffset 是连续的，这样非常方便是访问 commitlog 还是 访问 DLedger ，但如果 DLedger 部分文件删除后，这两个值就变的不连续，就会造成中间的文件空洞，无法被连续访问。

```
DLedgerCommitLog#recover
isInrecoveringOldCommitlog = true;
super.recoverNormally(maxPhyOffsetOfConsumeQueue);
isInrecoveringOldCommitlog = false;
```


Step3：如果启用了 DLedger 并且是初次启动(还未生成 DLedger 相关的日志文件)，则需要恢复 旧的 commitlog 文件。

```
DLedgerCommitLog#recover
MappedFile mappedFile = this.mappedFileQueue.getLastMappedFile();
if (mappedFile == null) {                // @1
    return;
}
ByteBuffer byteBuffer =  mappedFile.sliceByteBuffer();
byteBuffer.position(mappedFile.getWrotePosition());
boolean needWriteMagicCode = true;
// 1 TOTAL SIZE
byteBuffer.getInt(); //size
int magicCode = byteBuffer.getInt();
if (magicCode == CommitLog.BLANK_MAGIC_CODE) {    // @2
    needWriteMagicCode = false;
} else {
    log.info("Recover old commitlog found a illegal magic code={}", magicCode);
}
dLedgerConfig.setEnableDiskForceClean(false);
dividedCommitlogOffset = mappedFile.getFileFromOffset() + mappedFile.getFileSize();
// @3
log.info("Recover old commitlog needWriteMagicCode={} pos={} file={} dividedCommitl
ogOffset={}", needWriteMagicCode, mappedFile.getFileFromOffset() + mappedFile.getWrot
ePosition(), mappedFile.getFileName(), dividedCommitlogOffset);
if (needWriteMagicCode) {  // @4
    byteBuffer.position(mappedFile.getWrotePosition());
    byteBuffer.putInt(mappedFile.getFileSize() - mappedFile.getWrotePosition());
    byteBuffer.putInt(BLANK_MAGIC_CODE);
    mappedFile.flush(0);
}
mappedFile.setWrotePosition(mappedFile.getFileSize());    // @5
mappedFile.setCommittedPosition(mappedFile.getFileSize());
mappedFile.setFlushedPosition(mappedFile.getFileSize());
dLedgerFileList.getLastMappedFile(dividedCommitlogOffset);
log.info("Will set the initial commitlog offset={} for dledger", dividedCommitlogOffset);
}
```

Step4：如果存在旧的 commitlog 文件，需要将最后的文件剩余部分全部填充，即不再接受新的数据写入，新的数据全部写入到 DLedger 的数据文件中。其关键实现点如下：

- 尝试查找最后一个 commitlog 文件，如果未找到，则结束。
- 从最后一个文件的最后写入点(原 commitlog 文件的 待写入位点)尝试去查找写入的魔数，如果存在魔数并等于 CommitLog.BLANK_MAGIC_CODE，则无需再写入魔数，在升级 DLedger 第一次启动时，魔数为空，故需要写入魔数。
- 初始化 dividedCommitlogOffset ，等于最后一个文件的起始偏移量加上文件的大小，即该指针指向最后一个文件的结束位置。
- 将 最后一个 commitlog 未写满的数据全部写入，其方法为 设置消息体的 size 与魔数即可。
- 设置最后一个文件的 wrotePosition、flushedPosition、committedPosition 为文件的大小，同样有意味者最后一个文件已经写满，下一条消息将写入 DLedger 中。

在启用 DLedger 机制时 Broker 的启动流程就介绍到这里了，相信大家已经了解 DLedger 在整合 RocketMQ 上做的努力，接下来我们从消息追加、消息读取两个方面再来探讨 DLedger 是如何无缝整合 RocketMQ 的，实现平滑升级的。

四、从消息追加看 DLedger 整合 RocketMQ 如何实现无缝兼容

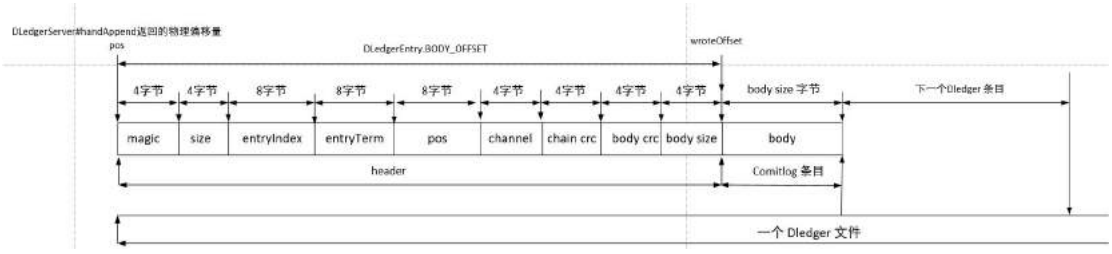
温馨提示：本节同样也不会详细介绍整个消息追加(存储流程)，只是要点出与 DLedger(多副本、主从切换)相关的核心关键点。如果想详细了解消息追加的流程，可以阅读笔者所著的《RocketMQ 技术内幕》一书。

```
    DLedgerCommitLog#putMessage
    AppendEntryRequest request = new AppendEntryRequest();
    request.setGroup(dLedgerConfig.getGroup());
    request.setRemoteId(dLedgerServer.getMemberState().getSelfId());
    request.setBody(encodeResult.data);
    dledgerFuture = (AppendFuture<AppendEntryResponse>) dLedgerServer.handleAppe
nd(request);
    if (dledgerFuture.getPos() == -1) {
        return new PutMessageResult(PutMessageStatus.OS_PAGECACHE_BUSY, new
AppendMessageResult(AppendMessageStatus.UNKNOWN_ERROR));
    }
```

关键点一：消息追加时，则不再写入到原先的 commitlog 文件中，而是调用 DLedgerServer 的 handleAppend 进行消息追加，该方法会有集群内的 Leader 节点负责消息追加以及在消息复制，只有超过集群内的半数节点成功写入消息后，才会返回写入成功。如果追加成功，将会返回本次追加成功后的起始偏移量，即 pos 属性，即类似于 rocketmq 中 commitlog 的偏移量，即物理偏移量。

```
DLedgerCommitLog#putMessage
long wroteOffset =  dledgerFuture.getPos() + DLedgerEntry.BODY_OFFSET;
ByteBuffer buffer = ByteBuffer.allocate(MessageDecoder.MSG_ID_LENGTH);
String msgId = MessageDecoder.createMessageId(buffer, msg.getStoreHostBytes(), w
roteOffset);
eclipseTimeInLock = this.defaultMessageStore.getSystemClock().now() - beginTimeIn
DledgerLock;
appendResult = new AppendMessageResult(AppendMessageStatus.PUT_OK, wroteO
ffset, encodeResult.data.length, msgId, System.currentTimeMillis(), queueOffset, eclipseTi
meInLock);
```

关键点二：根据 DLedger 的起始偏移量计算真正的消息的物理偏移量，从开头部分得知，DLedger 自身有其存储协议，其 body 字段存储真实的消息，即 commitlog 条目的存储结构，返回给客户端的消息偏移量为 body 字段的开始偏移量，即通过 putMessage 返回的物理偏移量与不使用 Dledger 方式返回的物理偏移量的含义是一样的，即从开偏移量开始，可以正确读取消息，这样 DLedger 完美的兼容了 RocketMQ Commitlog。关于 pos 以及 wroteOffset 的图解如下：



五、从消息读取看 DLedger 整合 RocketMQ 如何实现无缝兼容

```
    DLedgerCommitLog#getMessage
    public SelectMappedBufferResult getMessage(final long offset, final int size) {
        if (offset < dividedCommitlogOffset) {    // @1
            return super.getMessage(offset, size);
        }
        int mappedFileSize = this.dLedgerServer.getdLedgerConfig().getMappedFileSizeFor
EntryData();
        MmapFile mappedFile = this.dLedgerFileList.findMappedFileByOffset(offset, offset
== 0);    // @2
        if (mappedFile != null) {
            int pos = (int) (offset % mappedFileSize);
            return   convertSbr(mappedFile.selectMappedBuffer(pos, size));
                        // @3
        }
        return null;
    }
```

消息查找比较简单，因为返回给客户端消息，转发给 consumequeue 的消息物理偏移量并不是 DLedger 条目的偏移量，而是真实消息的起始偏移量。其实现关键点如下：

- 如果查找的物理偏移量小于 dividedCommitlogOffset，则从原先的 commitlog 文件中查找。
- 然后根据物理偏移量按照二分方找到具体的物理文件。
- 对物理偏移量取模，得出在该物理文件中中的绝对偏移量，进行消息查找即可，因为只有知道其物理偏移量，从该处先将消息的长度读取出来，然后即可读出一条完整的消息。

六、总结

根据上面详细的介绍，我想读者朋友们应该不难得出如下结论：

- DLedger 在整合时，使用 DLedger 条目包裹 RocketMQ 中的 commitlog 条目，即在 DLedger 条目的 body 字段来存储整条 commitlog 条目。
- 引入 dividedCommitlogOffset 变量，表示物理偏移量小于该值的消息存在于旧的 commitlog 文件中，实现 升级 DLedger 集群后能访问到旧的数据。

- 新 DLedger 集群启动后，会将最后一个 commitlog 填充，即新的数据不会再写入到 原先的 commitlog 文件。
- 消息追加到 DLedger 数据日志文件中，返回的偏移量不是 DLedger 条目的起始偏移量，而是 DLedger 条目中 body 字段的起始偏移量，即真实消息的起始偏移量，保证消息物理偏移量的语义与 RocketMQ Commitlog 一样。

RocketMQ 整合 DLedger(多副本)实现平滑升级的设计技巧就介绍到这里了。

如果本文对您有一定的帮助话，麻烦帮忙点个赞，非常感谢。

2.11 源码分析 RocketMQ DLedger 多副本即主从切换实现原理

DLedger 基于 raft 协议，故天然支持主从切换，即主节点(Leader)发送故障，会重新触发选主，在集群内再选举出新的主节点。

RocketMQ 中主从同步，从节点不仅会从主节点同步数据，也会同步元数据，包含 topic 路由信息、消费进度、延迟队列处理队列、消费组订阅配置等信息。那主从切换后元数据如何同步呢？特别是主从切换过程中，对消息消费有多大的影响，会丢失消息吗？

主从同步相关问答可先阅读该文章：
https://blog.csdn.net/prestigeding/article/details/93672079

温馨提示：本文假设大家已经对 RocketMQ4.5 版本之前的主从同步实现有一定的了解，这部分内容在《RocketMQ 技术内幕》一书中有详细的介绍，大家也可以参考
https://blog.csdn.net/prestigeding/article/details/93672079

一、BrokerController 中与主从相关的方法详解

本节先对 BrokerController 中与主从切换相关的方法。

1. startProcessorByHa

```
BrokerController#startProcessorByHa
private void startProcessorByHa(BrokerRole role) {
    if (BrokerRole.SLAVE != role) {
        if (this.transactionalMessageCheckService != null) {
            this.transactionalMessageCheckService.start();
        }
    }
}
```

感觉该方法的取名较为随意，该方法的作用是开启事务状态回查处理器，即当节点为主节点时，开启对应的事务状态回查处理器，对 PREPARE 状态的消息发起事务状态回查请求。

2. shutdownProcessorByHa

```
BrokerController#shutdownProcessorByHa
private void shutdownProcessorByHa() {
    if (this.transactionalMessageCheckService != null) {
        this.transactionalMessageCheckService.shutdown(true);
    }
}
```

关闭事务状态回查处理器，当节点从主节点变更为从节点后，该方法被调用。

3. handleSlaveSynchronize

```
BrokerController#handleSlaveSynchronize
private void handleSlaveSynchronize(BrokerRole role) {
    if (role == BrokerRole.SLAVE) {    // @1
        if (null != slaveSyncFuture) {
            slaveSyncFuture.cancel(false);
        }
        this.slaveSynchronize.setMasterAddr(null);    //
        slaveSyncFuture = this.scheduledExecutorService.scheduleAtFixedRate(new R
unnable() {
            @Override
            public void run() {
                try {
                    BrokerController.this.slaveSynchronize.syncAll();
                } catch (Throwable e) {
                    log.error("ScheduledTask SlaveSynchronize syncAll error.", e);
                }
            }
        }, 1000 * 3, 1000 * 10, TimeUnit.MILLISECONDS);
    } else {  // @2
        //handle the slave synchronise
```

```
            if (null != slaveSyncFuture) {
                slaveSyncFuture.cancel(false);
            }
            this.slaveSynchronize.setMasterAddr(null);
        }
    }
```

该方法的主要作用是处理从节点的元数据同步，即从节点向主节点主动同步 topic 的路由信息、消费进度、延迟队列处理队列、消费组订阅配置等信息。

代码@1：如果当前节点的角色为从节点：

- 如果上次同步的 future 不为空，则首先先取消。
- 然后设置 slaveSynchronize 的 master 地址为空。不知大家是否与笔者一样，有一个疑问，从节点的时候，如果将 master 地址设置为空，那如何同步元数据，那这个值会在什么时候设置呢?
- 开启定时同步任务，每 10s 从主节点同步一次元数据。

代码@2：如果当前节点的角色为主节点，则取消定时同步任务并设置 master 的地址为空。

4. changeToSlave

```
    BrokerController#changeToSlave
    public void changeToSlave(int brokerId) {
        log.info("Begin to change to slave brokerName={} brokerId={}", brokerConfig.getBr
okerName(), brokerId);
        //change the role
        brokerConfig.setBrokerId(brokerId == 0 ? 1 : brokerId); //TO DO check          // @
1
        messageStoreConfig.setBrokerRole(BrokerRole.SLAVE);
 // @2
        //handle the scheduled service
        try {
            this.messageStore.handleScheduleMessageService(BrokerRole.SLAVE);      //
@3
        } catch (Throwable t) {
```

```
            log.error("[MONITOR] handleScheduleMessageService failed when changing t
o slave", t);
        }
        //handle the transactional service
        try {
            this.shutdownProcessorByHa();
                    //  @4
        } catch (Throwable t) {
            log.error("[MONITOR] shutdownProcessorByHa failed when changing to slave
", t);
        }
        //handle the slave synchronise
        handleSlaveSynchronize(BrokerRole.SLAVE);
         // @5
        try {
            this.registerBrokerAll(true, true, brokerConfig.isForceRegister());                    //
 @6
        } catch (Throwable ignored) {
        }
        log.info("Finish to change to slave brokerName={} brokerId={}", brokerConfig.getBr
okerName(), brokerId);
    }
```

Broker 状态变更为从节点。其关键实现如下：

- 设置 brokerId，如果 broker 的 id 为 0，则设置为 1，这里在使用的时候，注意规划好集群内节点的 brokerId。
- 设置 broker 的状态为 BrokerRole.SLAVE。
- 如果是从节点，则关闭定时调度线程(处理 RocketMQ 延迟队列)，如果是主节点，则启动该线程。
- 关闭事务状态回查处理器。
- 从节点需要启动元数据同步处理器，即启动 SlaveSynchronize 定时从主服务器同步元数据。
- 立即向集群内所有的 nameserver 告知 broker 信息状态的变更。

5. changeToMaster

```
    BrokerController#changeToMaster
    public void changeToMaster(BrokerRole role) {
        if (role == BrokerRole.SLAVE) {
            return;
        }
        log.info("Begin to change to master brokerName={}", brokerConfig.getBrokerName
());
        //handle the slave synchronise
        handleSlaveSynchronize(role);    // @1
        //handle the scheduled service
        try {
            this.messageStore.handleScheduleMessageService(role);        // @2
        } catch (Throwable t) {
            log.error("[MONITOR] handleScheduleMessageService failed when changing t
o master", t);
        }
        //handle the transactional service
        try {
            this.startProcessorByHa(BrokerRole.SYNC_MASTER);             // @3
        } catch (Throwable t) {
            log.error("[MONITOR] startProcessorByHa failed when changing to master",
t);
        }
        //if the operations above are totally successful, we change to master
        brokerConfig.setBrokerId(0); //TO DO check                                    // @4
        messageStoreConfig.setBrokerRole(role);
        try {
            this.registerBrokerAll(true, true, brokerConfig.isForceRegister()); // @5
        } catch (Throwable ignored) {
        }
        log.info("Finish to change to master brokerName={}", brokerConfig.getBrokerName
());
    }
```

该方法是 Broker 角色从从节点变更为主节点的处理逻辑，其实现要点如下：

- 关闭元数据同步器，因为主节点无需同步。
- 开启定时任务处理线程。
- 开启事务状态回查处理线程。
- 设置 brokerId 为 0。
- 向 nameserver 立即发送心跳包以便告知 broker 服务器当前最新的状态。

主从节点状态变更的核心方法就介绍到这里了，接下来看看如何触发主从切换。

二、如何触发主从切换

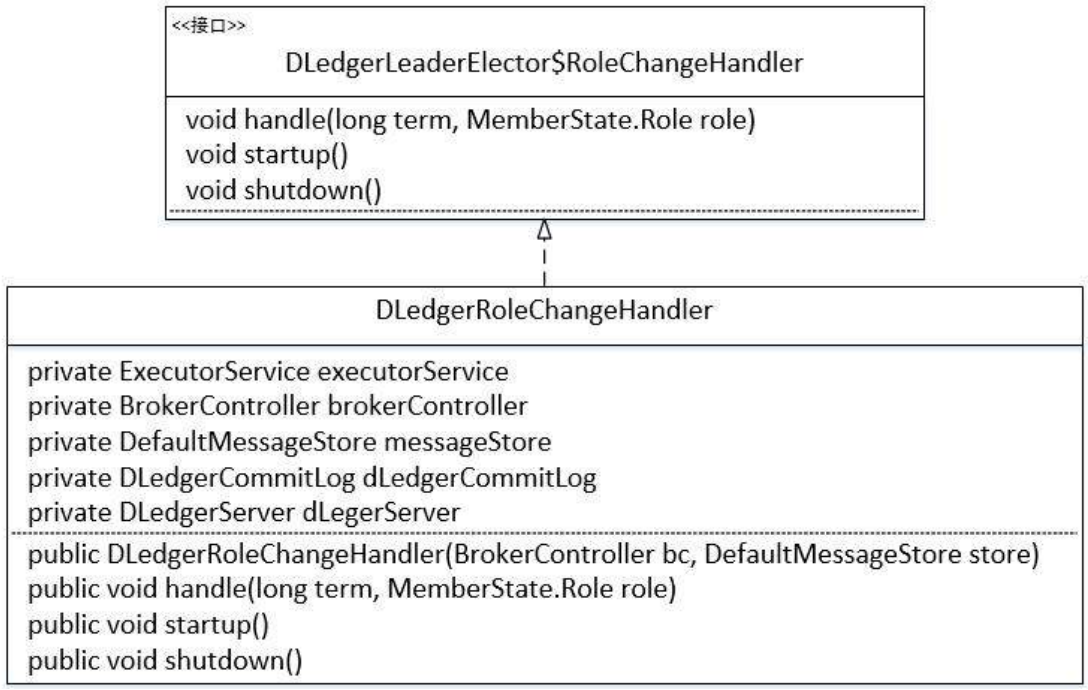
从前面的文章我们可以得知，RocketMQ DLedger 是基于 raft 协议实现的，在该协议中就实现了主节点的选举与主节点失效后集群会自动进行重新选举，经过协商投票产生新的主节点，从而实现高可用。

```
BrokerController#initialize
if (messageStoreConfig.isEnableDLegerCommitLog()) {
    DLedgerRoleChangeHandler roleChangeHandler = new DLedgerRoleChangeHandler(this, (DefaultMessageStore) messageStore);
    ((DLedgerCommitLog)((DefaultMessageStore) messageStore).getCommitLog()).getdLedgerServer().getdLedgerLeaderElector().addRoleChangeHandler(roleChangeHandler);
}
```

上述代码片段截取自 BrokerController 的 initialize 方法，我们可以得知在 Broker 启动时，如果开启了 多副本机制，即 enableDLedgerCommitLog 参数设置为 true，会为 集群节点选主器添加 roleChangeHandler 事件处理器，即节点发送变更后的事件处理器。

接下来我们将重点探讨 DLedgerRoleChangeHandler 。

1. 类图



DLedgerRoleChangeHandler 继承自 RoleChangeHandler，即节点状态发生变更后的事件处理器。上述的属性都很简单，在这里就重点介绍一下 ExecutorService executorService，事件处理线程池，但只会开启一个线程，故事件将一个一个按顺序执行。

接下来我们来重点看一下 handle 方法的执行。

2. handle 主从状态切换处理逻辑

```
    DLedgerRoleChangeHandler#handle
    public void handle(long term, MemberState.Role role) {
        Runnable runnable = new Runnable() {
            public void run() {
                long start = System.currentTimeMillis();
                try {
                    boolean succ = true;
                    log.info("Begin handling broker role change term={} role={} currStoreR
ole={}", term, role, messageStore.getMessageStoreConfig().getBrokerRole());
                    switch (role) {
                        case CANDIDATE:

    // @1
```

```
                    if (messageStore.getMessageStoreConfig().getBrokerRole() !=
BrokerRole.SLAVE) {
                        brokerController.changeToSlave(dLedgerCommitLog.getId
());
                    }
                    break;
                case FOLLOWER:              // @2
                    brokerController.changeToSlave(dLedgerCommitLog.getId());
                    break;
                case LEADER:                // @3
                    while (true) {
                        if (!dLegerServer.getMemberState().isLeader()) {
                            succ = false;
                            break;
                        }
                        if (dLegerServer.getdLedgerStore().getLedgerEndIndex() =
= -1) {
                            break;
                        }
                        if (dLegerServer.getdLedgerStore().getLedgerEndIndex() =
= dLegerServer.getdLedgerStore().getCommittedIndex()
                            && messageStore.dispatchBehindBytes() == 0) {
                            break;
                        }
                        Thread.sleep(100);
                    }
                    if (succ) {
                        messageStore.recoverTopicQueueTable();
                        brokerController.changeToMaster(BrokerRole.SYNC_MAST
ER);
                    }
                    break;
                default:
                    break;
            }
            log.info("Finish handling broker role change succ={} term={} role={} cu
rrStoreRole={} cost={}", succ, term, role, messageStore.getMessageStoreConfig().getBrok
erRole(), DLedgerUtils.elapsed(start));
        } catch (Throwable t) {
            log.info("[MONITOR]Failed handling broker role change term={} role={}
 currStoreRole={} cost={}", term, role, messageStore.getMessageStoreConfig().getBrokerR
ole(), DLedgerUtils.elapsed(start), t);
        }
```

```
            }
        };
        executorService.submit(runnable);
    }
```

代码@1：如果当前节点状态机状态为 CANDIDATE，表示正在发起 Leader 节点，如果该服务器的角色不是 SLAVE 的话，需要将状态切换为 SLAVE。

代码@2：如果当前节点状态机状态为 FOLLOWER，broker 节点将转换为 从节点。

代码@3：如果当前节点状态机状态为 Leader，说明该节点被选举为 Leader，在切换到 Master 节点之前，首先需要等待当前节点追加的数据都已经被提交后才可以将状态变更为 Master，其关键实现如下：

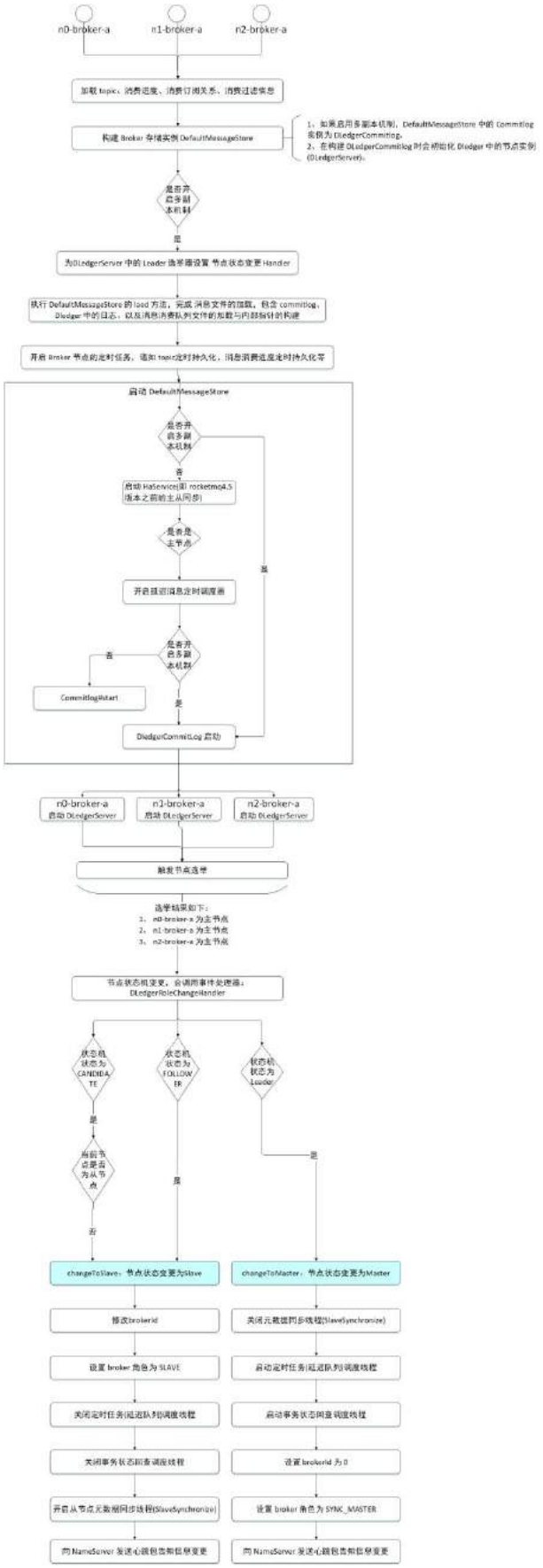
- 如果 ledgerEndIndex 为 -1，表示当前节点还未又数据转发，直接跳出循环，无需等待。
- 如果 ledgerEndIndex 不为 -1 ，则必须等待数据都已提交，即 ledgerEndIndex 与 committedIndex 相等。
- 并且需要等待 commitlog 日志全部已转发到 consumequeue 中，即 ReputMessageService 中的 reputFromOffset 与 commitlog 的 maxOffset 相等。
- 等待上述条件满足后，即可以进行状态的变更，需要恢复 ConsumeQueue，维护每一个 queue 对应的 maxOffset，然后将 broker 角色转变为 master。

经过上面的步骤，就能实时完成 broker 主节点的自动切换。由于单从代码的角度来看主从切换不够直观，下面我将给出主从切换的流程图。

3. 主从切换流程图

由于从源码的角度或许不够直观，故本节给出其流程图。

温馨提示：该流程图的前半部分在 源码分析 RocketMQ 整合 DLedger(多副本)实现平滑升级的设计技巧 该文中有所阐述。



三、主从切换若干问题思考

我相信经过上面的讲解，大家应该对主从切换的实现原理有了一个比较清晰的理解，我更相信读者朋友们会抛出一个疑问，主从切换会不会丢失消息，消息消费进度是否会丢失而导致重复消费呢?

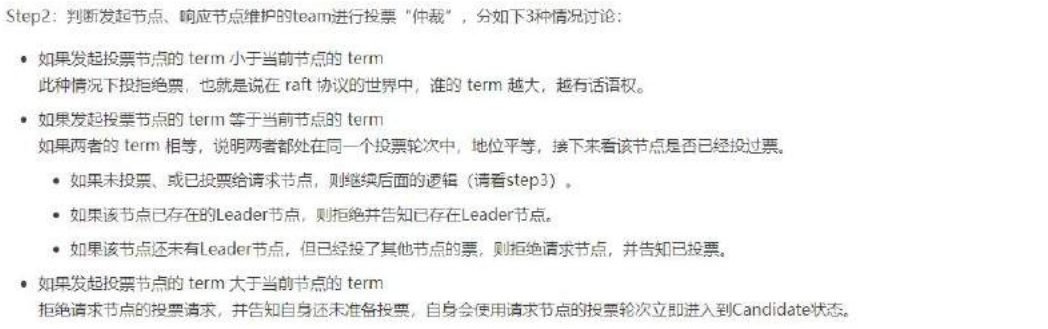
温馨提示：由于个人水平有限，该部分可能存在错误，希望能与大家多多讨论交流。

1. 消息消费进度是否存在丢失风险

首先，由于 RocketMQ 元数据，当然也包含消息消费进度的同步是采用的从服务器定时向主服务器拉取进行更新，存在时延，引入 DLedger 机制，也并不保证其一致性，DLedger 只保证 commitlog 文件的一致性。当主节点宕机后，各个从节点并不会完成同步了消息消费进度，于此同时，消息消费继续，此时消费者会继续从从节点拉取消息进行消费，但汇报的从节点并不一定会成为新的主节点，故消费进度在 broker 端存在丢失的可能性。当然并不是一定会丢失，因为消息消费端只要不重启，消息消费进度会存储在内存中。综合所述，消息消费进度在 broker 端会有丢失的可能性，存在重复消费的可能性，不过问题不大，因为 RocketMQ 本身也不承若不会重复消费。

2. 消息是否存在丢失风险

消息会不会丢失的关键在于，日志复制进度的从节点是否可以被选举为主节点，如果在一个集群中，从节点的复制进度落后与从主节点，但当主节点宕机后，如果该从节点被选举成为新的主节点，那这将是一个灾难，将会丢失数据。关于一个节点是否给另外一个节点投赞成票的逻辑在 https://blog.csdn.net/prestigeding/article/details/99697323 的 2.4.2 handleVote 方法中已详细介绍，在这里我以截图的方式再展示其核心点：


Step2：判断发起节点、响应节点维护的team进行投票“仲裁”，分如下3种情况讨论：

- 如果发起投票节点的 term 小于当前节点的 term
  此种情况下投拒绝票，也就是说在 raft 协议的世界中，谁的 term 越大，越有话语权。
- 如果发起投票节点的 term 等于当前节点的 term
  如果两者的 term 相等，说明两者都处在同一个投票轮次中，地位平等，接下来看该节点是否已经投过票。
  - 如果未投票、或已投票给请求节点，则继续后面的逻辑（请看step3）。
  - 如果该节点已存在的Leader节点，则拒绝并告知已存在Leader节点。
  - 如果该节点还未有Leader节点，但已经投了其他节点的票，则拒绝请求节点，并告知已投票。
- 如果发起投票节点的 term 大于当前节点的 term
  拒绝请求节点的投票请求，并告知自身还未准备投票，自身会使用请求节点的投票轮次立即进入到Candidate状态。

Step3：判断请求节点的 ledgerEndTerm 与当前节点的 ledgerEndTerm，这里主要是判断日志的复制进度。

- 如果请求节点的 ledgerEndTerm 小于当前节点的 ledgerEndTerm 则拒绝，其原因是请求节点的日志复制进度比当前节点低，这种情况是不能成为主节点的。
- 如果 ledgerEndTerm 相等，但是 ledgerEndIndex 比当前节点小，则拒绝，原因与上一条相同。
- 如果请求的 term 小于 ledgerEndTerm 以同样的理由拒绝。

从上面可以得知，如果发起投票节点的复制进度比自己小的话，会投拒绝票。其

```
lastVoteCost = DLedgerUtils.elapsed(startVoteTimeMs);
VoteResponse.ParseResult parseResult;
if (knownMaxTermInGroup.get() > term) {
    parseResult = VoteResponse.ParseResult.WAIT_TO_VOTE_NEXT;
    nextTimeToRequestVote = getNextTimeToRequestVote();
    changeRoleToCandidate(knownMaxTermInGroup.get());
} else if (alreadyHasLeader.get()) {
    parseResult = VoteResponse.ParseResult.WAIT_TO_VOTE_NEXT;
    nextTimeToRequestVote = getNextTimeToRequestVote() + heartBeatTimeIntervalMs * maxHeartBeatLeak;
} else if (!memberState.isQuorum(validNum.get())) {
    parseResult = VoteResponse.ParseResult.WAIT_TO_REVOTE;
    nextTimeToRequestVote = getNextTimeToRequestVote();
} else if (memberState.isQuorum(acceptedNum.get())) {
    parseResult = VoteResponse.ParseResult.PASSED;
} else if (memberState.isQuorum(acceptedNum.get() + notReadyTermNum.get())) {
    parseResult = VoteResponse.ParseResult.REVOTE_IMMEDIATELY;
} else if (memberState.isQuorum(acceptedNum.get() + biggerLedgerNum.get())) {
    parseResult = VoteResponse.ParseResult.WAIT_TO_REVOTE;
    nextTimeToRequestVote = getNextTimeToRequestVote();
} else {
    parseResult = VoteResponse.ParseResult.WAIT_TO_VOTE_NEXT;
    nextTimeToRequestVote = getNextTimeToRequestVote();
}
```

其关键点如下：

- 如果对端的投票轮次大于发起投票的节点，则该节点使用对端的轮次，重新进入到Candidate状态，并且重置投票计时器，其值为“1个常规计时器”
- 如果已经存在Leader，该节点重新进入到Candidate,并重置定时器，该定时器的时间：“1个常规计时器” + heartBeatTimeIntervalMs * maxHeartBeatLeak，其中 heartBeatTimeIntervalMs 为一次心跳间隔时间，maxHeartBeatLeak 为 允许最大丢失的心跳包，即如果Flower节点在多少个心跳周期内未收到心跳包，则认为Leader已下线。
- 如果收到的有效票数未超过半数，则重置计时器为“1个常规计时器”，然后等待重新投票，注意状态为WAIT_TO_REVOTE，该状态下的特征是下次投票时不增加投票轮次。
- 如果得到的赞同票超过半数，则成为Leader。
- 如果得到的赞成票加上未准备投票的节点数超过半数，则应该立即发起投票，故其结果为REVOTE_IMMEDIATELY。
- 如果得到的赞成票加上对端维护的ledgerEndIndex超过半数，则重置计时器，继续本轮次的选举。
- 其他情况，开启下一轮投票。

必须得到集群内超过半数节点认可，即最终选举出来的主节点的当前复制进度一定是比绝大多数的从节点要大，并且也会等于承诺给客户端的已提交偏移量。故得出的结论是不会丢消息。

本文的介绍就到此为止了，最后抛出一个思考题与大家相互交流学习，也算是对 DLedger 多副本即主从切换一个总结回顾。答案我会以留言的方式或在下一篇文章中给出。

四、思考题

例如一个集群内有 5 个节点的 DLedgr 集群。

Leader Node: n0-broker-a
folloer Node: n1-broker-a,n2-broker-a,n3-broker-a,n4-broker-a

从节点的负责进度可能不一致，例如：

n1-broker-a 复制进度为 100
n2-broker-a 复制进度为 120
n3-broker-a 复制进度为 90
n4-broker-a 负载进度为 90

如果此时 n0-broker-a 节点宕机，触发选主，如果 n1 率先发起投票，由于 n1, 的复制进度大于 n3,n4，再加上自己一票，是有可能成为 leader 的，此时消息会丢失吗? 为什么?

欢迎大家以留言的方式进行交流，也可以加我微信号 dingwpmz 与我进行探讨，最后如果这篇文章对大家有所帮助的话，麻烦点下【在看】谢谢大家。

扫一扫加入作者公众号
中间件兴趣圈

扫一扫关注
RocketMQ 官微

扫一扫关注【阿里巴巴云原生】公众号
获取第一手技术干货

阿里云开发者“藏经阁”
海量免费电子书下载