如何把开源项目用好，很大程度上是由学习路径决定的：

a. fork下来，起一个demo，上一个测试环境，遇到问题再去社区提问或找些实践文章；

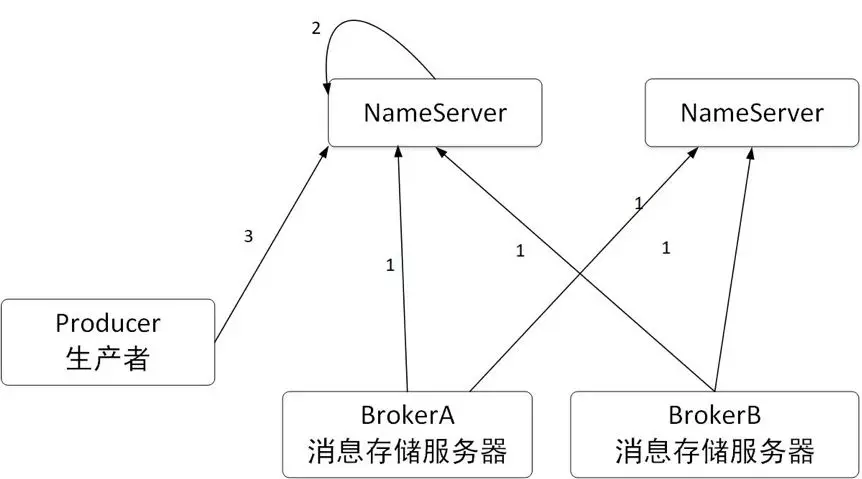
b. 把官方文档通读一遍，理解下产品、特点和应用场景；

c. 先看一遍源代码，理解清楚其中的代码逻辑；

d. 看源代码太费劲，找本社区推荐的书系统的梳理下；

本文来自 Apache RocketMQ 的资深用户丁威，他和 MyCat 的核心开发者周继锋合著了《RocketMQ技术内幕：架构设计与实现原理》一书，目的是希望用图解的方式梳理 RocketMQ的核心原理，包括 RocketMQ Topic 的路由注册与剔除机制、消息发送高可用设计、消息存储文件设计、并发消息拉取与消息消费流程、主从同步(HA)、事务消息基本实现原理等，帮助开发者在使用 RocketMQ 的同时，还能对其核心原理了然于心。

**Topic 的路由机制**



介绍路由注册机制之前，先简单看下 RocketMQ 的整体架构：

Producer：消息生产者，用于向消息服务器发送消息；

NameServer：路由注册中心；

Broker：消息存储服务器；

Consumer：消息消费者，该流程图中未涉及；

*联通性：*

A. NameServer 之间互不通信，无法感知对方的存在。

B. Producer 生产者与 NameServer 集群中的一台服务器建立长连接，并持有整个 NameServer 集群的列表。

C. Broker 服务会与每台 NameServer 保持长连接。

*Topic路由注册与剔除流程：*

A. Broker 每30s向 NameServer 发送心跳包，心跳包中包含主题的路由信息（主题的读写队列数、操作权限等），NameServer 会通过 HashMap 更新 Topic 的路由信息，并记录最后一次收到 Broker 的时间戳。

B. NameServer 以每10s的频率清除已宕机的 Broker，NameServer 认为 Broker 宕机的依据是如果当前系统时间戳减去最后一次收到 Broker 心跳包的时间戳大于120s。

C. 消息生产者以每30s的频率去拉取主题的路由信息，即消息生产者并不会立即感知 Broker 服务器的新增与删除。

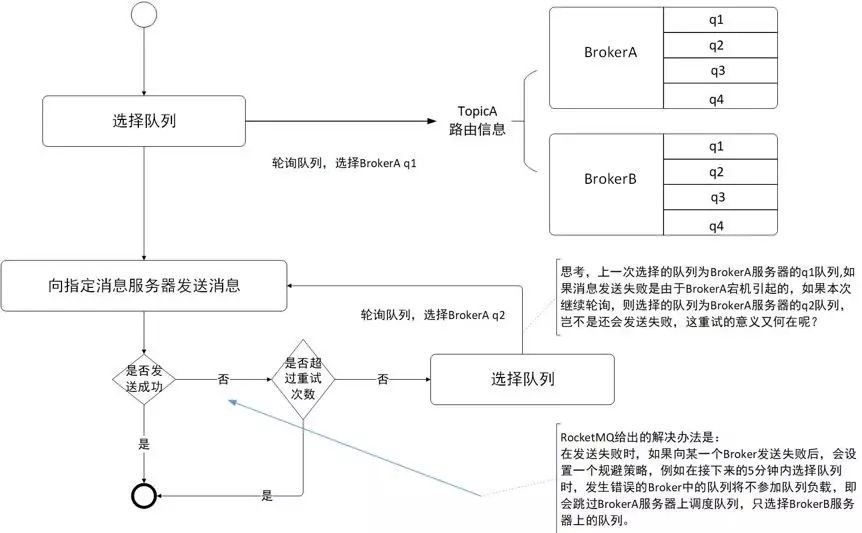
*该部分涉及到的编程技巧：*

A. 基于长连接的编程模型、心跳包。

B. 多线程编程，读写锁经典使用场景。

思考：由于消息生产者无法实时感知 Broker 服务器的宕机，那消息发送的高可用性如何保证呢？

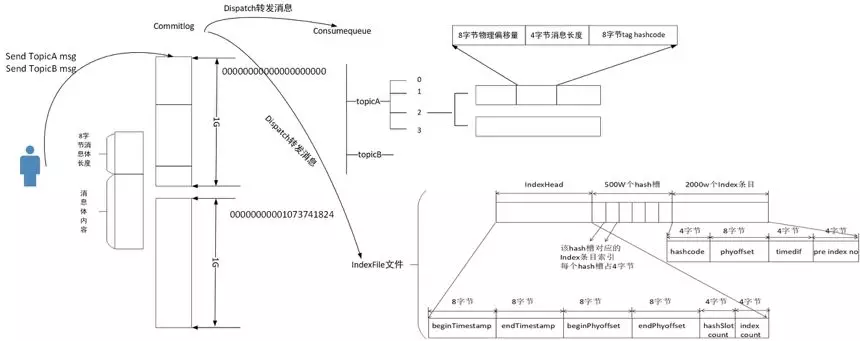
**消息发送高可用设计**



消息发送队列负载默认采用轮询机制，消息发送时默认选择重试机制来保证消息发送的高可用。当 Broker 宕机后，虽然消息发送者无法第一时间感知 Broker 宕机，但是当消息发送者向 Broker 发送消息返回异常后，生产者会在接下来一定时间内，例如5分钟内不会再次选择该 Broker上的队列，这样就规避了发生故障的 Broker，结合重试机制，巧妙实现消息发送的高可用。

**消息存储文件设计**

RocketMQ 存储设计主要包含 CommitLog 文件、ConsumeQueue 文件和 IndexFile 文件。



*CommitLog 文件：*

消息存储文件，所有主题的消息随着到达 Broker 的顺序写入 CommitLog 文件，每个文件默认为1G，文件的命名也及其巧妙，使用该存储在消息文件中的第一个全局偏移量来命名文件，这样的设计主要是方便根据消息的物理偏移量，快速定位到消息所在的物理文件。RocketMQCommitLog 文件使用顺序写，极大提高了文件的写性能。

*ConsumeQueue 文件：*

消息消费队列文件，是 CommitLog 文件的基于 Topic 的索引文件，主要用于消费者根据 Topic消费消息，其组织方式为 /topic/queue，同一个队列中存在多个文件，ConsumeQueue 设计极具技巧性，其每个条目使用固定长度（8字节 CommitLog 物理偏移量、4字节消息长度、8字节 Tag HashCode），这里不是存储 tag 的原始字符串，而是存储 HashCode，目的就是确保每个条目的长度固定，可以使用访问类似数组下标的方式来快速定位条目，极大的提高了 ConsumeQueue文件的读取性能，试想一下，消息消费者根据 Topic、消息消费进度(ConsumeQueue 逻辑偏移量)，即第几个 ConsumeQueue 条目，这样根据消费进度去访问消息的方法为使用逻辑偏移量logicOffset\* 20即可找到该条目的起始偏移量( ConsumeQueue 文件中的偏移量)，然后读取该偏移量后20个字节即得到了一个条目，无需遍历 ConsumeQueue 文件。

*IndexFile 文件：*

基于物理磁盘文件实现 Hash 索引。其文件由40字节的文件头、500W个 Hash 槽，每个 Hash 槽为4个字节，最后由2000万个 Index 条目，每个条目由20个字节构成，分别为4字节的索引key的 HashCode、8字节消息物理偏移量、4字节时间戳、4字节的前一个Index条目( Hash 冲突的链表结构)。

*存储文件部分的编程技巧：*

A. 内存映射文件编程技巧。

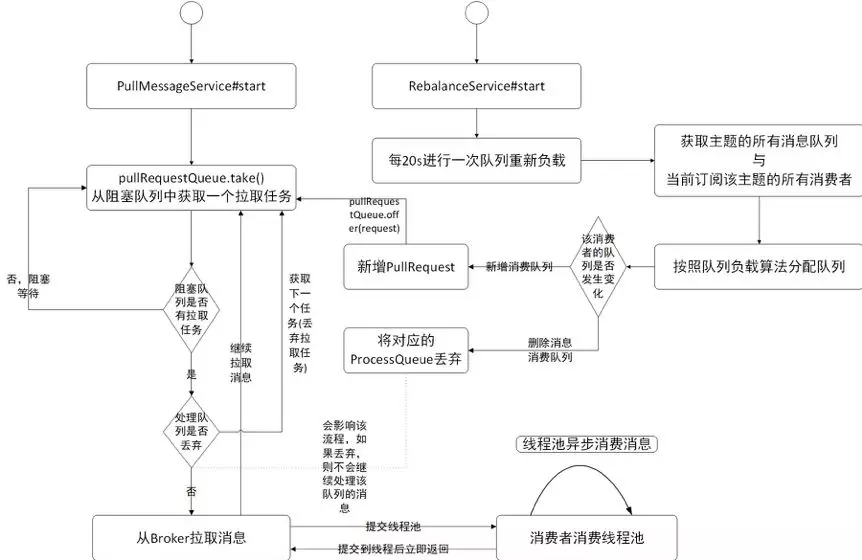
B. 内存锁定技术。

C. 基于文件的Hash索引实现技巧。

D. 多线程协作技巧。

E. 异步刷盘机制实现

**并发消息拉取和消息消费流程**



消息消费通常涉及到消息队列负载、消息拉取、消息过滤、消息消费（处理消息）、消费进度反馈等方面。并发消息拉取与消息消费流程如图所示：

注：下面有关消息消费阐述的相关观点主要基于集群消费模式下的并发消费机制。

*消息队列负载：*

集群内(同一消费组)内的消费者共同承担主题下所有消息的消费，即一条消息只能被集群中一个消费者消费。RocketMQ的队列负载原则是一个消费者可以承担同一主题下的多个消息消费队列，但同一个消息消费队列同一时间只允许被分配给一个消费者。

*RebalaceService 线程：*

其职责是负责消息消费队列的负载，默认以20s的间隔按照队列负载算法进行队列分配，如果此次分配到的队列与上一次分配的队列不相同，则需要触发消息队列的更新操作：

A. 如果是新分配的队列，则创建 PullReqeust 对象(拉取消息任务)，添加到 PullMessageService 线程内部的阻塞队列 pullRequestQueue 中。如果该队列中存在拉取任务，则 PullMessageService 会向 Broker 拉取消息。

B. 如果是上次分配但本次未分配的队列，将其处理队列 ProcessQueue 的状态设置为丢弃，然后 PullMessageService 线程在根据 PullRequest 拉取消息时首先会判断 ProcessQueue 队列的状态，如果是已丢弃状态，则直接丢弃 PullRequest 对象，停止拉取该队列中的消息，否则向Broker 拉取消息，拉取到一批消息后，提交到一个处理线程池，然后继续将 PullRequest 对象添加到 pullRequestQueue，即很快就会再次触发对该消息消费队列的再次拉取，这也是 RocketMQ 实现 PUSH 模式的本质。

消费者消费线程池处理完一条消息时，消费者需要向 Broker 汇报消费的进度，以防消息重复消费。这样当消费者重启后，指示消费者应该从哪条消息开始消费。并发消费模式下，由于多线程消费的缘故，提交到线程池消费的消息默认情况下无法保证消息消费的顺序。

例如，线程池正在消费偏移量为1,2,3的消息，并不保证偏移量为1的消息先消费完成，如果消息的处理完成顺序为3,1,2，使用消息完成的顺序去更新消息消费进度显然是有问题的，有可能会造成消息丢失，故RocketMQ的消息消费进度反馈策略是每一条消息处理完成后，并不是用消息自身的偏移量去更新消息消费进度，而是使用处理队列中最小的偏移量去更新，在此例中，如果是消息3的消息先处理完成，则会使用偏移量为1去更新消息消费进度。当然这种处理保证了不丢消息，但却带来了另外一个问题，消息有可能会重复消息。

在 PUSH 模式下，PullMessageService 拉取完一批消息后，将消息提交到线程池后会“马不蹄停”去拉下一批消息，如果此时消息消费线程池处理速度很慢，处理队列中的消息会越积越多，占用的内存也随之飙升，最终引发内存溢出，更加不能接受的消息消费进度并不会向前推进，因为只要该处理队列中偏移量最小的消息未处理完成，整个消息消费进度则无法向前推进，如果消费端重启，又得重复拉取消息并造成大量消息重复消费。RocketMQ 解决该问题的策略是引入消费端的限流机制。

*RocketMQ 消息消费端的限流的两个维度：*

A. 消息堆积数量

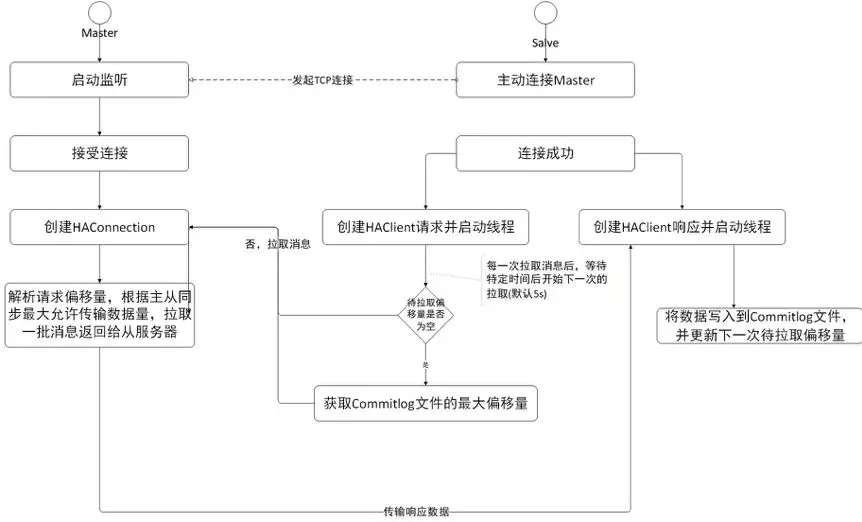
如果消息消费处理队列中的消息条数超过1000条会触发消费端的流控，其具体做法是放弃本次拉取动作，并且延迟50ms后将放入该拉取任务放入到pullRequestQueue中，每1000次流控会打印一次消费端流控日志。

B. 消息堆积大小

如果处理队列中堆积的消息总内存大小超过100M，同样触发一次流控。

注：上述只需满足条件之一就会触发一次流控。

**主从同步（HA）**



RocketMQ 的主从同步机制如下：

A. 首先启动Master并在指定端口监听；

B. 客户端启动，主动连接Master，建立TCP连接；

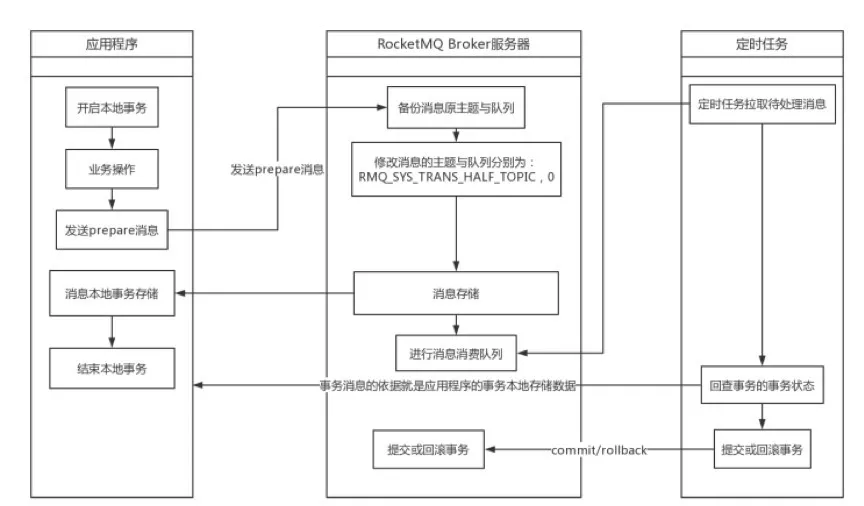
C. 客户端以每隔5s的间隔时间向服务端拉取消息，如果是第一次拉取的话，先获取本地commitlog文件中最大的偏移量，以该偏移量向服务端拉取消息；

D. 服务端解析请求，并返回一批数据给客户端；

E. 客户端收到一批消息后，将消息写入本地commitlog文件中，然后向Master汇报拉取进度，并更新下一次待拉取偏移量；

F. 然后重复第3步；

**事务消息**



RocketMQ事务消息的实现原理是类似基于二阶段提交与事务状态回查来实现的。事务消息的发送只支持同步方式，其实现的关键点包括：

A. 在应用程序端，在一个本地事务中，通过发送消息API向Broker发送Prepare状态的消息，收到消息服务器返回成功后执行事件回调函数，在事件函数的职责就是记录该消息的事务状态，通常采用消息发送本地事务表，即往本地事务表中插入一条记录，如果业务处理成功，则消息本地事务中会存在相关记录；如果本地事务执行失败而导致事务回滚，此时本地事务状态中不存在该消息的事务状态。

B. 消息服务端收到Prepare的消息时，如何保证消息不会被消费端立即处理呢？原来消息服务端收到Prepare状态的消息，会先备份原消息的主题与队列，然后变更主题为：RMQ\_SYS\_TRANS\_OP\_HALF\_TOPIC，队列为0。

C. 消息服务端会开启一个专门的线程，以每60s的频率从RMQ\_SYS\_TRANS\_OP\_HALF\_TOPIC中拉取一批消息，进行事务状态的回查，其实现原理是根据消息所属的消息生产者组名随机获取一个生产者，向其询问该消息对应的本地事务是否成功，如果本地事务成功（该部分是由业务提供的事务回查监听器来实现），则消息服务端执行提交动作；如果事务状态返回失败，则消息服务端执行回滚动作；如果事务状态未知，则不做处理，待下一次定时任务触发再检查。默认如果连续5次回查都无法得到确切的事务状态，则执行回滚动作。

以上只是 RocketMQ 所有核心的一部分，在文章的结尾处，我想再分享一下我学习 RocketMQ的一些心得：

A. 通读 RocketMQ 官方文档，从全局上了解 RocketMQ。

B. 在IDE工具中搭建 RocketMQ 调试环境，启动 NameServer、Broker 服务器，并重点关注源码的 example 包，运行一个快速入门示例。

C. 根据功能模块进行学习，例如消息发送、消息存储、消息消费，同时注意不要发散，例如在学习消息发送相关的流程时，遇到消息存储后，可暂时不去理会消息存储相关的细节，先一笔带过，待学完消息发送后，再去重点学习其他分支，例如存储、刷盘，主从同步等。