前言

作者BB

大家好呀，我是 yes，是这本消息队列核心知识点电子书的作者，电子书的全部内容整理自我的个人公众号「**yes的练级攻略**」里的关于消息队列的文章。

公众号里面还有别的系列，比如 JVM 、MySQL 、Spring 等等，这个日后也会整理出 PDF，可以微信搜索「**yes的练级攻略**」关注我的公众号，进行一波更进，内容都是首发公众号的。

这本 PDF，我称之为 0.1 版本，**主要涵盖了关于消息队列的大部分核心知识点**，涉及的消息队列有 RocketMQ、Kafka。

对于里面的大多数内容都进行了源码层面的解析，做到真正的源码级别理解，大家不要怕会被源码劝退了，源码剖析之后我都进行了图解，照着图来理解，非常舒服~

所以，如果你想要深入消息队列，这本 PDF 很适合你；如果你想在面试官前面装一下，这本 PDF 也很适合你；如果你个人想深入源码，但是不知道如何入手，这本 PDF 还是很适合你。

当然，这是这本 PDF 的第一个版本，也算是内测版本，如有错误可以通过公众号联系我，期待你的指正和探讨。

再说一下关于面试方向的问题。不吹牛，这本 PDF 的深度对于大厂面试题来说绝对够了，你读完就知道啦~

还有，这本书的内容不像教科书那般连贯，因为是公众号文章合成的，但是我还是建议从前往后读，因为我个人对内容的先后还是做了排版的。

本 PDF 大纲：

- 从消息队列常见面试题入手来解析消息队列
- 如何设计一个消息队列？
- 消息队列设计成推消息还是拉消息？RocketMQ和Kafka是怎么做的？
- 消息队列之事务消息？RocketMQ和Kafka是怎么做的？
- 比 RocketMQ 更好的事务消息实现是什么？
- Kafka的索引设计有什么亮点？
- Kafka日志段如何读写解析？
- Kafka控制器事件处理全流程解析
- Kafka请求处理全流程解析
- Kafka为什么要抛弃Zookeeper？
- 进阶必看的 RocketMQ，这次一网打尽
- Kafka和RocketMQ底层存储揭秘，为什么能这么快？
- 未完待更新

从消息队列常见面试题入手来解析消息队列

今儿咱们就来盘一盘大方向上的消息队列有哪些核心注意点。

核心点有很多，为了更**贴合实际场景**，我从常见的面试问题入手：

- 如何保证消息不丢失？
- 如果处理重复消息？

- 如何保证消息的有序性？
- 如果处理消息堆积？

当然在剖析这几个问题之前需要简单的介绍下**什么是消息队列**，消息队列常见的一些**基本术语和概念。**

接下来进入正文。

什么是消息队列

来看看维基百科怎么说的，顺带学学英语这波不亏：

> In computer science, message queues and mailboxes are software-engineering components typically used for inter-process communication (IPC), or for inter-thread communication within the same process. They use a queue for messaging – the passing of control or of content. Group communication systems provide similar kinds of functionality.

翻译一下：在计算机科学领域，消息队列和邮箱都是软件工程组件，通常用于进程间或同一进程内的线程通信。它们通过队列来传递消息-传递控制信息或内容，群组通信系统提供类似的功能。

简单的概括下上面的定义：**消息队列就是一个使用队列来通信的组件。**

上面的定义没有错，但就现在而言我们日常所说的**消息队列常常指代的是消息中间件**，它的存在不仅仅只是为了通信这个问题。

为什么需要消息队列

从本质上来说是因为互联网的快速发展，**业务不断扩张**，促使技术架构需要不断的演进。

从以前的单体架构到现在的微服务架构，成百上千的服务之间相互调用和依赖。从互联网初期一个服务器上有 100 个在线用户已经很了不得，到现在坐拥10亿日活的微信。我们需要有一个「东西」来解耦服务之间的关系、控制资源合理合时的使用以及缓冲流量洪峰等等。

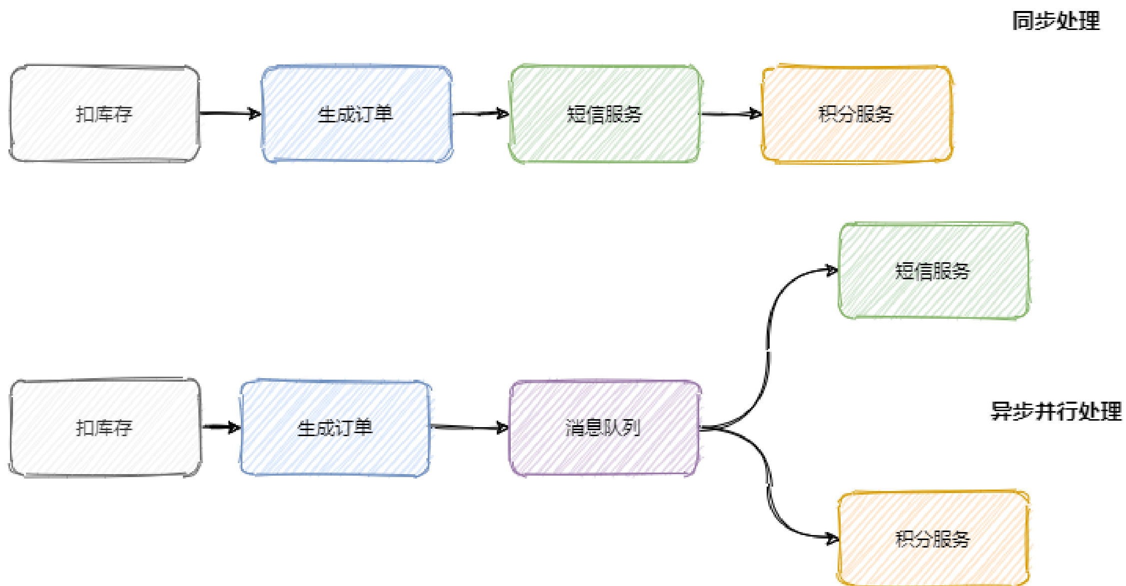
消息队列就应运而生了。它常用来实现：**异步处理、服务解耦、流量控制**。

异步处理

随着公司的发展你可能会发现你项目的**请求链路越来越长**，例如刚开始的电商项目，可以就是粗暴的扣库存、下单。慢慢地又加上积分服务、短信服务等。这一路同步调用下来客户可能等急了，这时候就是消息队列登场的好时机。

调用链路长、响应就慢了，并且相对于扣库存和下单，积分和短信没必要这么的 "及时"。因此只需要在下单结束那个流程，扔个消息到消息队列中就可以直接返回响应了。而且积分服务和短信服务可以并行的消费这条消息。

可以看出消息队列可以**减少请求的等待，还能让服务异步并发处理，提升系统总体性能。**

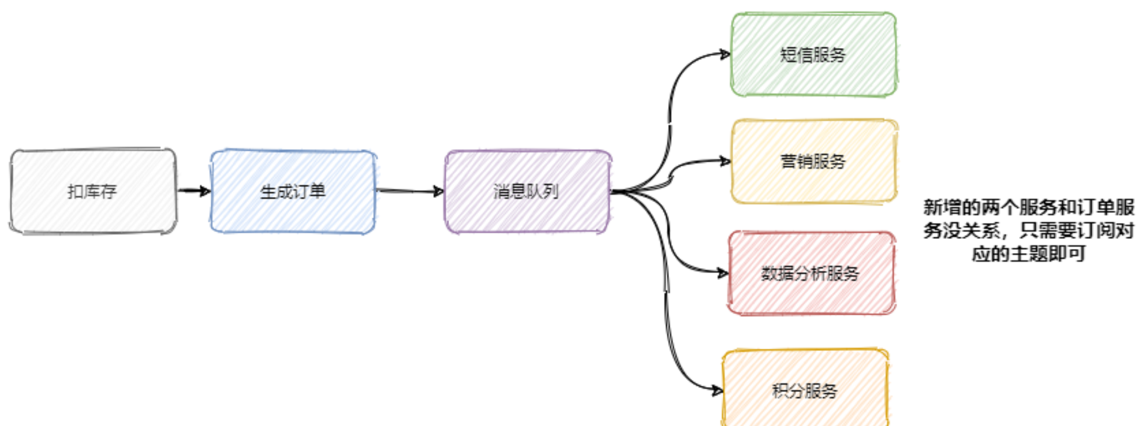


服务解耦

上面我们说到加了积分服务和短信服务，这时候可能又要来个营销服务，之后领导又说想做个大数据，又来个数据分析服务等等。

可以发现订单的下游系统在不断的扩充，为了迎合这些下游系统订单服务需要经常地修改，任何一个下游系统接口的变更可能都会影响到订单服务，这订单服务组可疯了，**真·「核心」项目组。**

所以一般会选用消息队列来解决系统之间耦合的问题，订单服务把订单相关消息塞到消息队列中，下游系统谁要谁就订阅这个主题。这样订单服务就解放啦！



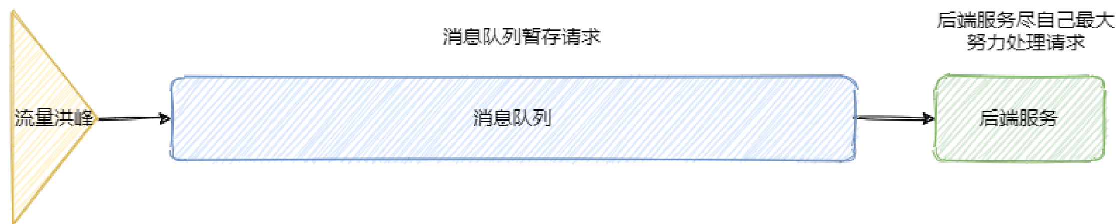
流量控制

想必大家都听过「削峰填谷」，后端服务相对而言都是比较「弱」的，因为业务较重，处理时间较长。像一些例如秒杀活动爆发式流量打过来可能就顶不住了。因此需要引入一个中间件来做缓冲，消息队列再适合不过了。

网关的请求先放入消息队列中，后端服务尽自己最大能力去消息队列中消费请求。超时的请求可以直接返回错误。

当然还有一些服务特别是某些后台任务，不需要及时地响应，并且业务处理复杂且流程长，那么过来的请求先放入消息队列中，后端服务按照自己的节奏处理。这也是很 nice 的。

上面两种情况分别对应着生产者生产过快和消费者消费过慢两种情况，消息队列都能在其中发挥很好的缓冲效果。



注意

引入消息队列固然有以上的好处，但是多引入一个中间件系统的稳定性就下降一层，运维的难度抬高一层。因此要**权衡利弊**，**系统是演进的**。

消息队列基本概念

消息队列有两种模型：**队列模型**和**发布/订阅模型**。

队列模型

生产者往某个队列里面发送消息，一个队列可以存储多个生产者的消息，一个队列也可以有多个消费者，
但是消费者之间是竞争关系，即每条消息只能被一个消费者消费。

发布/订阅模型

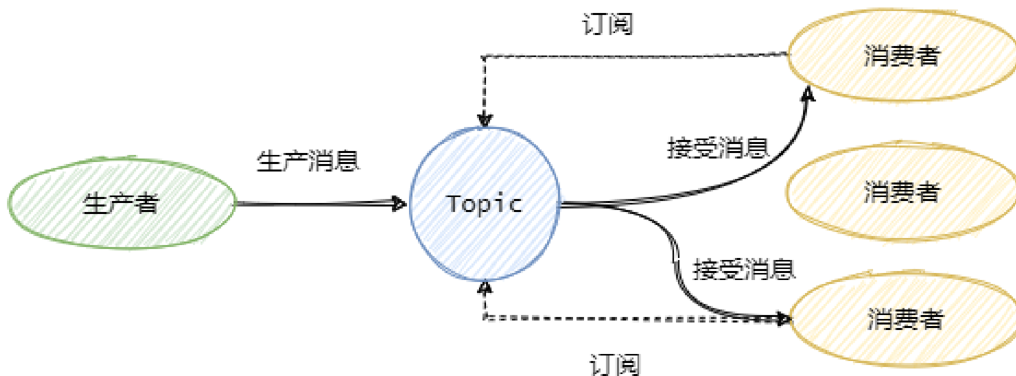
为了解决一条消息能被多个消费者消费的问题，发布/订阅模型就来了。该模型是将消息发往一个 `Topic` 即主题中，所有订阅了这个 `Topic` 的订阅者都能消费这条消息。

其实可以这么理解，发布/订阅模型等于我们都加入了一个群聊中，我发一条消息，加入了这个群聊的人都能收到这条消息。
那么队列模型就是一对一聊天，我发给你的消息，只能在你的聊天窗口弹出，是不可能弹出到别人的聊天窗口中的。

讲到这有人说，那我一对一聊天对每个人都发同样的消息不就也实现了一条消息被多个人消费了嘛。

是的，通过多队列全量存储相同的消息，即数据的冗余可以实现一条消息被多个消费者消费。`RabbitMQ` 就是采用队列模型，通过 `Exchange` 模块来将消息发送至多个队列，解决一条消息需要被多个消费者消费问题。

这里还能看到假设群聊里除我之外只有一个人，那么此时的发布/订阅模型和队列模型其实就一样了。



小结一下

队列模型每条消息只能被一个消费者消费，而发布/订阅模型就是为让一条消息可以被多个消费者消费而生的，当然队列模型也可以通过消息全量存储至多个队列来解决一条消息被多个消费者消费问题，但是会有数据的冗余。

发布/订阅模型兼容队列模型，即只有一个消费者的情况下和队列模型基本一致。

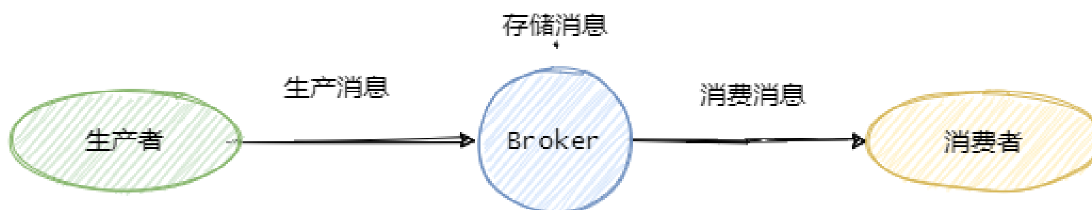
`RabbitMQ` 采用队列模型，`RocketMQ` 和 `Kafka` 采用发布/订阅模型。

接下来的内容都基于发布/订阅模型。

常用术语

一般我们称发送消息方为生产者 `Producer`，接受消费消息方为消费者 `Consumer`，消息队列服务端为 `Broker`。

消息从 `Producer` 发往 `Broker`，`Broker` 将消息存储至本地，然后 `Consumer` 从 `Broker` 拉取消息，或者 `Broker` 推送消息至 `Consumer`，最后消费。



为了提高并发度，往往**发布/订阅模型**还会引入**队列**或者**分区**的概念。即消息是发往一个主题下的某个队列或者某个分区中。`RocketMQ` 中叫队列，`Kafka` 叫分区，本质一样。

例如某个主题下有 5 个队列，那么这个主题的并发度就提高为 5 ，同时可以有 5 个消费者**并行消费**该主题的消息。一般可以采用轮询或者 `key hash` 取余等策略来将同一个主题的消息分配到不同的队列中。

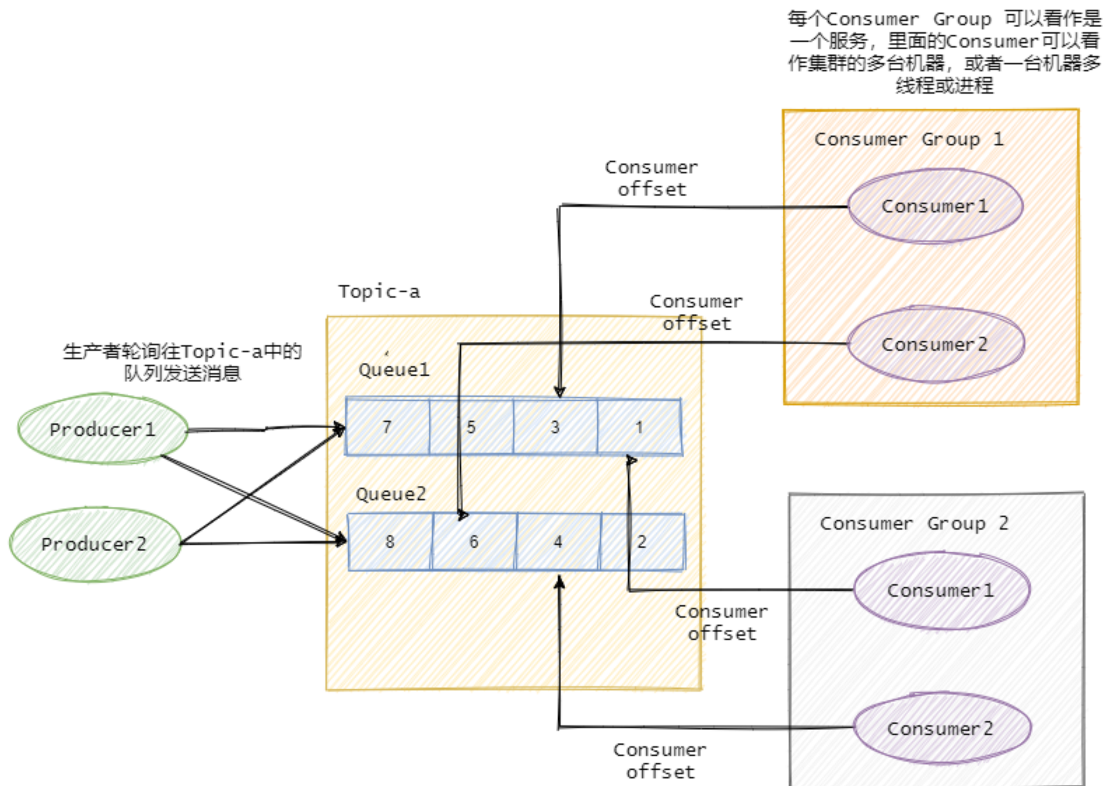
与之对应的消费者一般都有组的概念 `Consumer Group`，即消费者都是属于某个消费组的。一条消息会发往多个订阅了这个主题的消费组。

假设现在有两个消费组分别是`Group 1`和`Group 2`，它们都订阅了`Topic-a`。此时有一条消息发往`Topic-a`，那么这两个消费组都能接收到这条消息。

然后这条消息实际是写入`Topic`某个队列中，消费组中的某个消费者对应消费一个队列的消息。

在物理上除了副本拷贝之外，一条消息在`Broker`中只会有一份，每个消费组会有自己的`offset`即消费点位来标识消费到的位置。在消费点位之前的消息表明已经消费过了。当然这个`offset`是队列级别的。每个消费组都会维护订阅的`Topic`下的每个队列的`offset`。

来个图看看应该就很清晰了。

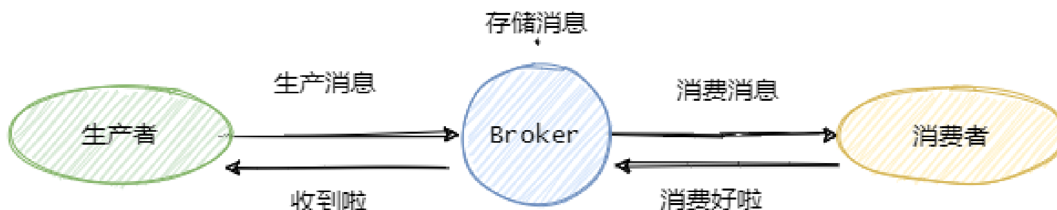


基本上熟悉了消息队列常见的术语和一些概念之后，咱们再来看看消息队列常见的核心面试点。

如何保证消息不丢失

就我们市面上常见的消息队列而言，只要**配置得当**，我们的消息就不会丢。

先来看看这个图，



可以看到一共有三个阶段，分别是**生产消息、存储消息和消费消息**。我们从这三个阶段分别入手来看看如何确保消息不会丢失。

生产消息

生产者发送消息至`Broker`，需要处理`Broker`的响应，不论是同步还是异步发送消息，同步和异步回调都需要做好`try-catch`，妥善的处理响应，如果`Broker`返回写入失败等错误消息，需要重试发送。当多次发送失败需要作报警，日志记录等。

这样就能保证在生产消息阶段消息不会丢失。

存储消息

存储消息阶段需要在**消息刷盘之后**再给生产者响应，假设消息写入缓存中就返回响应，那么机器突然断电这消息就没了，而生产者以为已经发送成功了。

如果`Broker`是集群部署，有多副本机制，即消息不仅仅要写入当前`Broker`,还需要写入副本机中。那配置成至少写入两台机子后再给生产者响应。这样基本上就能保证存储的可靠了。一台挂了还有一台还在呢（假如怕两台都挂了..那就再多些）。

那假如来个地震机房机子都挂了呢？emmmmmm...大公司基本上都有异地多活。

那要是这几个地都地震了呢？emmmmmm...这时候还是先关心关心人吧。

消费消息

这里经常会有同学犯错，有些同学当消费者拿到消息之后直接存入内存队列中就直接返回给`Broker`消费成功，这是不对的。

你需要考虑拿到消息放在内存之后消费者就宕机了怎么办。所以我们应该在**消费者真正执行完业务逻辑之后，再发送给`Broker`消费成功**，这才是真正的消费了。

所以只要我们在消息业务逻辑处理完成之后再给`Broker`响应，那么消费阶段消息就不会丢失。

小结一下

可以看出，保证消息的可靠性需要**三方配合**。

`生产者`需要处理好`Broker`的响应，出错情况下利用重试、报警等手段。

`Broker`需要控制响应的时机，单机情况下是消息刷盘后返回响应，集群多副本情况下，即发送至两个副本及以上的情况下再返回响应。

`消费者`需要在执行完真正的业务逻辑之后再返回响应给`Broker`。

但是要注意**消息可靠性增强了，性能就下降了**，等待消息刷盘、多副本同步后返回都会影响性能。因此还是看业务，例如日志的传输可能丢那么一两条关系不大，因此没必要等消息刷盘再响应。

如果处理重复消息

我们先来看看能不能避免消息的重复。

假设我们发送消息，就管发，不管`Broker`的响应，那么我们发往`Broker`是不会重复的。

但是一般情况我们是不允许这样的，这样消息就完全不可靠了，我们的基本需求是消息至少得发到`Broker`上，那就得等`Broker`的响应，那么就可能存在`Broker`已经写入了，当时响应由于网络原因生产者没有收到，然后生产者又重发了一次，此时消息就重复了。

再看消费者消费的时候，假设我们消费者拿到消息消费了，业务逻辑已经走完了，事务提交了，此时需要更新`Consumer offset`了，然后这个消费者挂了，另一个消费者顶上，此时`Consumer offset`还没更新，于是又拿到刚才那条消息，业务又被执行了一遍。于是消息又重复了。

可以看到正常业务而言**消息重复是不可避免的**，因此我们只能从**另一个角度**来解决重复消息的问题。

关键点就是**幂等**。既然我们不能防止重复消息的产生，那么我们只能在业务上处理重复消息所带来的影响。

幂等处理重复消息

幂等是数学上的概念，我们就理解为同样的参数多次调用同一个接口和调用一次产生的结果是一致的。

例如这条 SQL
`update t1 set money = 150 where id = 1 and money = 100;` 执行多少遍`money`都是150，这就叫幂等。

因此需要改造业务处理逻辑，使得在重复消息的情况下也不会影响最终的结果。

可以通过上面我那条 SQL 一样，做了个**前置条件判断**，即`money = 100`情况，并且直接修改，更通用的是做个`version`即版本号控制，对比消息中的版本号和数据库中的版本号。

或者通过**数据库的约束例如唯一键**，例如`insert into update on duplicate key...`。

或者**记录关键的key**，比如处理订单这种，记录订单ID，假如有重复的消息过来，先判断下这个ID是否已经被处理过了，如果没处理再进行下一步。当然也可以用全局唯一ID等等。

基本上就这么几个套路，**真正应用到实际中还是得看具体业务细节**。

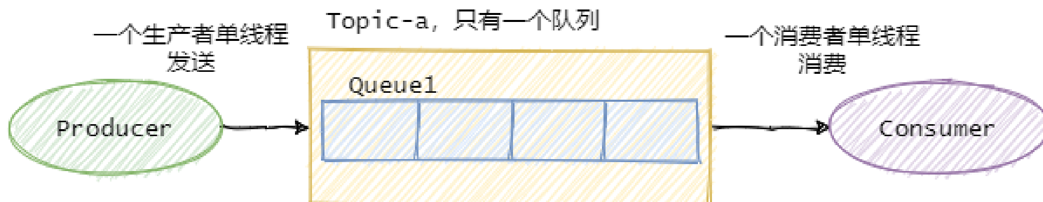
如何保证消息的有序性

有序性分：**全局有序和部分有序。**

全局有序

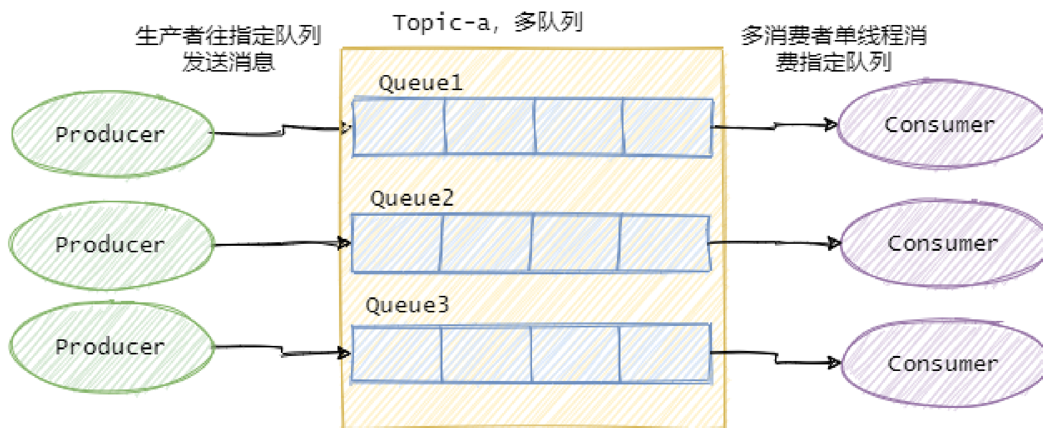
如果要保证消息的全局有序，首先只能由一个生产者往`Topic`发送消息，并且一个`Topic`内部只能有一个队列（分区）。消费者也必须是单线程消费这个队列。这样的消息就是全局有序的！

不过一般情况下我们都不需要全局有序，即使是同步`MySQL Binlog`也只需要保证单表消息有序即可。



部分有序

因此绝大部分的有序需求是部分有序，部分有序我们就可以将`Topic`内部划分成我们需要的队列数，把消息通过特定的策略发往固定的队列中，然后每个队列对应一个单线程处理的消费者。这样即完成了部分有序的需求，又可以通过队列数量的并发来提高消息处理效率。



图中我画了多个生产者，一个生产者也可以，只要同类消息发往指定的队列即可。

如果处理消息堆积

消息的堆积往往是因为**生产者的生产速度与消费者的消费速度不匹配。**有可能是因为消息消费失败反复重试造成的，也有可能就是消费者消费能力弱，渐渐地消息就积压了。

因此我们需要**先定位消费慢的原因**，如果是`bug`则处理 `bug` ，如果是因为本身消费能力较弱，我们可以优化下消费逻辑，比如之前是一条一条消息消费处理的，这次我们批量处理，比如数据库的插入，一条一条插和批量插效率是不一样的。

假如逻辑我们已经都优化了，但还是慢，那就得考虑水平扩容了，增加`Topic`的队列数和消费者数量，**注意队列数一定要增加**，不然新增加的消费者是没东西消费的。**一个Topic中，一个队列只会分配给一个消费者。**

当然你消费者内部是单线程还是多线程消费那看具体场景。不过要注意上面提高的消息丢失的问题，如果你是将接受到的消息写入**内存队列**之后，然后就返回响应给`Broker`，然后多线程向内存队列消费消息，假设此时消费者宕机了，内存队列里面还未消费的消息也就丢了。

最后

上面的几个问题都是我们在使用消息队列的时候经常能遇到的问题，并且也是面试关于消息队列方面的核心考点。今天没有深入具体消息队列的细节，但是套路就是这么个套路，大方向上搞明白很关键

如何设计一个消息队列？

这种设计类问题想必大家都不陌生，面试时或多或少都能碰到。

比如如何写一个线程池？如何写一个 HashMap ？如何写一个 RPC 框架等等，当然这里的写不是真的叫你用代码写出来，只是说说设计理念，整体架构。

这个面试题来自于一个读者的字节面试经历，我会从面试技巧和消息中间件的设计两个方面阐述。

我觉得重点在于面试技巧，因为它通用。

两种极端的情况

大多数同学遇到这种问题会出现两种极端的情况：

- 第一种：一脸懵逼，两眼无神，不知从何说起，万般思绪，都化作一声叹息。
- 第二种：夸夸其谈，像是口中架起了一把加特林，哒哒哒哒哒哒哒哒，还冒着蓝火。

第一种不用说了，好一点的面试官可能会引导你，会问一些提示性的问题，一步一步地带你渐入佳境，当然你要是胸中无点滴，那还是没救的，场面就异常地尴尬。

第二种会把面试官整蒙了，或许你真的懂很多，很多细节也都清晰，但是你不能一股脑儿的都抛出来，这会显得你抓不住重点。

面试官也是人

这点其实很关键，很多把面试官当成一个莫得感情的提问机器人，觉得他无所不能可以完全 get 到你的点，殊不知你引以为傲的细节回答，他可能觉得你在说蛇皮。

是人就会有感情，就需要交流，好的面试官会把控整体进度，从拉家常开始，让场子热起来再一步一步的深挖。

当然也有一些面试官比较弱，这时候就需要你来**特意地流出一点空白，来让面试官涂鸦**，让面试官感觉你这人就很舒服，你这波就稳了。

当然即使面对着把控全场的面试官你也得主动出击，每个人都有自己的擅长点，你需要引导面试官来询问你的长处。

正确的回答姿势

正确的回答姿势是 BFS（广度优先搜索） 而不是 DFS （深度优先搜索），什么意思呢？

就是我们需要**先从大局上讲出需要设计的东西的重点，然后再等待面试官的继续提问，深挖。**

我们需要**揣摩面试官的心理**，从他的提问可以看出他想要知道的重点是哪个方向的。

比如就拿 HashMap 来说，你**简单的**把获取、写入、冲突处理、扩容啥的都说了，然后等待面试官接下来的提问，有可能会往线程安全方面深入，也有可能会往扩容方向再挖，比如引出 Redis 的 hash 扩容等等。

所以说给面试官留提问的机会，抓住他的喜好或者说熟知的方向回答，这样如果你答得好，相互之间谈的来，面试官会对你高度认可。

而且在**说各设计要点的时候也要注意停顿，要留机会给面试官插话，让面试官充分参与你的设计。**

还是拿 HashMap 作为例子，比如你说了获取、写入、冲突之后稍作停顿，这时候大概率面试官还会问还有吗？让面试官有参与感，**让他感觉经过他的引导这个设计才逐步地完善。**

当然如果不问也没事，你停顿下继续说就行。

让面试成为一场技术交流，这是面试的最高境界，相信面试完了之后双方都会有意犹未尽的感觉，惺惺相惜就是这么来的。

但是这种场景也不是这么容易碰到的，首先你和面试官得有相同方向的喜好，比如你对 JVM 有很深入的研究，而面试官对存储方面有很深入的研究，JVM 懂的不深，这样就碰不出火花了。

所以说会有很多人碰到这么个情况：我面这个公司一面挂，另一家公司面面超神，这都是很正常的。

当然你要是说你全能，那当我没说。

小结一下面试技巧

首先要正确的看待面试官，你和面试官是同等的，不要一来就低声下气的。

其次回答问题需要抓住重点，不要一股脑儿的把你知道的都说了，要留白待面试官提问。

要把控面试的节奏，往自己熟知的方向上引。

如何写个消息中间件

接下来咱们再看看如何写个消息中间件。

首先我们需要明确地提出消息中间件的几个重要角色，分别是生产者、消费者、Broker、注册中心。

简述下消息中间件数据流转过程，无非就是生产者生成消息，发送至 Broker，Broker 可以暂缓消息，然后消费者再从 Broker 获取消息，用于消费。

而注册中心用于服务的发现包括：Broker 的发现、生产者的发现、消费者的发现，当然还包括下线，可以说服务的高可用离不开注册中心。

然后开始简述实现要点，可以同通信讲起：各模块的通信可以基于 Netty 然后自定义协议来实现，注册中心可以利用 zookeeper、consul、eureka、nacos 等等，也可以像 RocketMQ 自己实现简单的 namesrv （这一句话就都是关键词）。

为了考虑扩容和整体的性能，采用分布式的思想，像 Kafka 一样采取分区理念，一个 Topic 分为多个 partition，并且为保证数据可靠性，采取多副本存储，即 Leader 和 follower，根据性能和数据可靠的权衡提供异步和同步的刷盘存储。

并且利用选举算法保证 Leader 挂了之后 follower 可以顶上，保证消息队列的高可用。

也同样为了提高消息队列的可靠性利用本地文件系统来存储消息，并且采用顺序写的方式来提高性能。

可根据消息队列的特性利用内存映射、零拷贝进一步的提升性能，还可利用像 Kafka 这种批处理思想提高整体的吞吐。

至此就差不多了，该说的要点说的都差不多了，面试官心里已经想，这人好像有点东西。

之后可以深挖的点就很多了，比如提到的 Netty，各种注册中心就能问很多，比如各注册中心之间的选型对比等。

你还提到了选举算法，所以可能会问 Bully 算法、Raft 算法、ZAB 算法等等。

你还提到了分区，可能会问这个分区和 RocketMQ 的队列有什么不同啊？具体分区要怎么实现？

然后你提到顺序写，可能会问为什么要顺序写啊？你说的内存映射和零拷贝又是什么啊？那你知道 RocketMQ 和 Kafka 用了哪个吗？

当然还有可能问各种细节，比如消息的写入如何存储、消息的索引如何生成等等，来深挖看你有没有看过消息中间件的源码。

可以问的还很多，这篇文章我也不可能每个点都延伸开说，**这些知识点还是得靠大家日积月累和平日的多加思考。**

当然日后的文章可以写一写今天提到的一些点，比如 Netty、选举算法啊，多种注册中心对比啊啥的。

面试官想问的是什么

再回到这个面试题，其实面试官想问的就是大方向上的设计，包括整体的架构、数据的流转和一些特性的把握，**所以对于这个问题他想听到的就是那些重点，而不是那些细节。**

而继续的深挖取决于你回答这个问题时提出的各个关键词，对于面试官自身而言熟悉的词一抓到，他就已经知道下一步要问你什么了。

所以**在回答面试官的时候不仅要 get 到他的点，还得为之后的回答铺路**，不会说的点不要提，擅长的点多提提。

最后

之前我已经提到了，这篇文章的重点其实不在于如何回答写一个消息中间件，而在于面试的技巧。

因为面试题千千万，而技巧掌握了那么千千万的面试题都适用。

我还想提一下关于面试的一些个人看法，我个人是面试驱动学习型选手，我学习的动力就是面试，我享受面试官问我啥我都嘴角一翘微微一笑的那种不羁。

但是我不提倡那种纯粹背面试题的做法，学习是一个日积月累的过程，就像我每篇文末说的，从一点点到亿点点，又像我每篇开头都会提的，每个时代，都不会亏待会学习的人。

我的面试驱动不仅仅是说为了面试而学习，还要以面试场景来学习，什么意思呢？

学任何一种东西，都模拟一个面试官在你前面，让他从各种角度向你提问，驱动你全方位的理解一个知识点，这才是我说的面试驱动学习型选手。

所以如果你看过我之前的文章会发现我经常会提出为什么呢，然后再作答。

还有一点要注意，**动手能力，这很关键。**

Talk is cheap, show me the code。

消息队列设计成推消息还是拉消息？RocketMQ和Kafka是怎么做的？

今天我们就来谈一谈消息队列的推拉模式，这也是一个面试热点，例如你在简历里面写了 RocketMQ，基本上会问你 RocketMQ 采用的是推模式还是拉模式啊？是拉模式？不是有 PushConsumer 吗？

今天我们就来谈谈推拉模式，并且再来看看 RocketMQ 和 Kafka 是如何做的。

推拉模式

首先明确一下推拉模式到底是在讨论消息队列的哪一个步骤，一般而言我们在谈论**推拉模式的时候指的是 Comsumer 和 Broker 之间的交互。**

默认的认为 Producer 与 Broker 之间就是推的方式，即 Producer 将消息推送给 Broker，而不是 Broker 主动去拉取消息。

想象一下，如果需要 Broker 去拉取消息，那么 Producer 就必须在本地通过日志的形式保存消息来等待 Broker 的拉取，如果有很多生产者的话，那么消息的可靠性不仅仅靠 Broker 自身，还需要靠成百上千的 Producer。

Broker 还能靠多副本等机制来保证消息的存储可靠，而成百上千的 Producer 可靠性就有点难办了，所以默认的 Producer 都是推消息给 Broker。

所以说有些情况分布式好，而有些时候还是集中管理好。

推模式

推模式指的是消息从 Broker 推向 Consumer，即 Consumer 被动的接收消息，由 Broker 来主导消息的发送。

我们来想一下推模式有什么好处？

消息实时性高， Broker 接受完消息之后可以立马推送给 Consumer。

对于消费者使用来说更简单，简单啊就等着，反正有消息来了就会推过来。

推模式有什么缺点？

推送速率难以适应消费速率，推模式的目标就是以最快的速度推送消息，当生产者往 Broker 发送消息的速率大于消费者消费消息的速率时，随着时间的增长消费者那边可能就“爆仓”了，因为根本消费不过来啊。当推送速率过快就像 DDos 攻击一样消费者就傻了。

并且不同的消费者的消费速率还不一样，身为 Broker 很难平衡每个消费者的推送速率，如果要实现自适应的推送速率那就需要在推送的时候消费者告诉 Broker ，我不行了你推慢点吧，然后 Broker 需要维护每个消费者的状态进行推送速率的变更。

这其实就增加了 Broker 自身的复杂度。

所以说推模式难以根据消费者的状态控制推送速率，适用于消息量不大、消费能力强要求实时性高的情况下。

拉模式

拉模式指的是 Consumer 主动向 Broker 请求拉取消息，即 Broker 被动的发送消息给 Consumer。

我们来想一下拉模式有什么好处?

拉模式主动权就在消费者身上了，**消费者可以根据自身的情况来发起拉取消息的请求。**假设当前消费者觉得自己消费不过来了，它可以根据一定的策略停止拉取，或者间隔拉取都行。

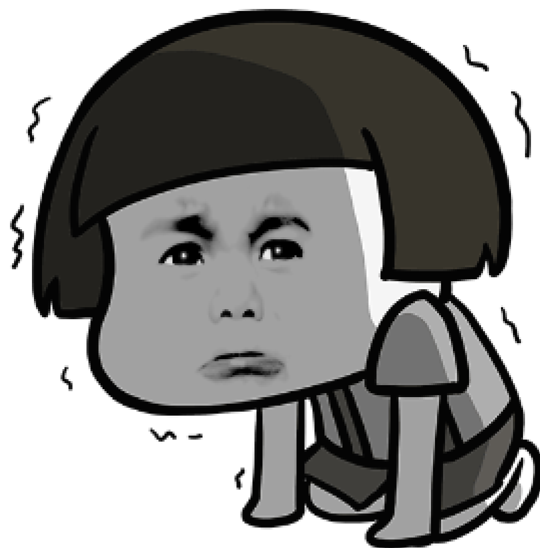
拉模式下 Broker 就相对轻松了，它只管存生产者发来的消息，至于消费的时候自然由消费者主动发起，来一个请求就给它消息呗，从哪开始拿消息，拿多少消费者都告诉它，它就是一个没有感情的工具人，消费者要是没来取也不关它的事。

拉模式可以更合适的进行消息的批量发送，基于推模式可以来一个消息就推送，也可以缓存一些消息之后再推送，但是推送的时候其实不知道消费者到底能不能一次性处理这么多消息。而拉模式就更加合理，它可以参考消费者请求的信息来决定缓存多少消息之后批量发送。

拉模式有什么缺点?

消息延迟，毕竟是消费者去拉取消息，但是消费者怎么知道消息到了呢？所以它只能不断地拉取，但是又不能很频繁地请求，太频繁了就变成消费者在攻击 Broker 了。因此需要降低请求的频率，比如隔个 2 秒请求一次，你看着消息就很有可能延迟 2 秒了。

消息忙请求，忙请求就是比如消息隔了几个小时才有，那么在几个小时之内消费者的请求都是无效的，在做无用功。



那到底是推还是拉

可以看到推模式和拉模式各有优缺点，到底该如何选择呢？

RocketMQ 和 Kafka 都选择了拉模式，当然业界也有基于推模式的消息队列如 ActiveMQ。

我个人觉得拉模式更加的合适，因为现在的消息队列都有持久化消息的需求，也就是说本身它就有个存储功能，它的使命就是接受消息，保存好消息使得消费者可以消费消息即可。

而消费者各种各样，身为 Broker 不应该有依赖于消费者的倾向，我已经为你保存好消息了，你要就来拿好了。

虽说一般而言 Broker 不会成为瓶颈，因为消费端有业务消耗比较慢，但是 Broker 毕竟是一个中心点，能轻量就尽量轻量。

那么竟然 RocketMQ 和 Kafka 都选择了拉模式，它们就不怕拉模式的缺点么？ 怕，所以它们操作了一波，减轻了拉模式的缺点。

长轮询

RocketMQ 和 Kafka 都是利用“长轮询”来实现拉模式，我们就来看看它们是如何操作的。

为了简单化，下面我把消息不满足本次拉取的条数啊、总大小啊等等都统一描述成还没有消息，反正都是不满足条件。

RocketMQ 中的长轮询

RocketMQ 中的 PushConsumer 其实是披着拉模式的方法，**只是看起来像推模式而已。**

因为 RocketMQ 在被背后偷偷的帮我们去 Broker 请求数据了。

后台会有个 RebalanceService 线程，这个线程会根据 topic 的队列数量和当前消费组的消费者个数做负载均衡，每个队列产生的 pullRequest 放入阻塞队列 pullRequestQueue 中。然后又有个 PullMessageService 线程不断的从阻塞队列 pullRequestQueue 中获取 pullRequest，然后通过网络请求 broker，这样实现的准实时拉取消息。

这一部分代码我不截了，就是这么个事儿，稍后会用图来展示。

然后 Broker 的 PullMessageProcessor 里面的 processRequest 方法是用来处理拉消息请求的，有消息就直接返回，如果没有消息怎么办呢？我们来看一下代码。

```
case ResponseCode.PULL_NOT_FOUND:
        // 如果没找到消息，并且 broker允许挂起消息并且请求允许挂起消息
    if (brokerAllowSuspend && hasSuspendFlag) {
        long pollingTimeMills = suspendTimeoutMillisLong;
        if (!this.brokerController.getBrokerConfig().isLongPollingEnable()) {
            pollingTimeMills = this.brokerController.getBrokerConfig().getShortPollingTimeMills();
        }

        String topic = requestHeader.getTopic();
        long offset = requestHeader.getQueueOffset();
        int queueId = requestHeader.getQueueId();
        PullRequest pullRequest = new PullRequest(request, channel, pollingTimeMills,
            this.brokerController.getMessageStore().now(), offset, subscriptionData, messageFilter);
        //调用 PullRequestHoldService ，将请求挂起
        this.brokerController.getPullRequestHoldService().suspendPullRequest(topic, queueId,
pullRequest);
        response = null;
        break;
    }
```

我们再来看下 suspendPullRequest 方法做了什么。

```java
public void suspendPullRequest(final String topic, final int queueId, final PullRequest pullRequest) {
    String key = this.buildKey(topic, queueId); //通过主题和队列id做个key
    ManyPullRequest mpr = this.pullRequestTable.get(key);
    if (null == mpr) {
        mpr = new ManyPullRequest();
        ManyPullRequest prev = this.pullRequestTable.putIfAbsent(key, mpr);
        if (prev != null) {
            mpr = prev;
        }
    }

    mpr.addPullRequest(pullRequest); //将拉请求加进去
}

//实际就是加到这个list 里面

private final ArrayList<PullRequest> pullRequestList = new ArrayList<>();
```

而 PullRequestHoldService 这个线程会每 5 秒从 pullRequestTable 取PullRequest请求，然后看看待拉取消息请求的偏移量是否小于当前消费队列最大偏移量，如果条件成立则说明有新消息了，则会调用 notifyMessageArriving ，最终调用 PullMessageProcessor 的 executeRequestWhenWakeup() 方法重新尝试处理这个消息的请求，也就是再来一次，整个长轮询的时间默认 30 秒。

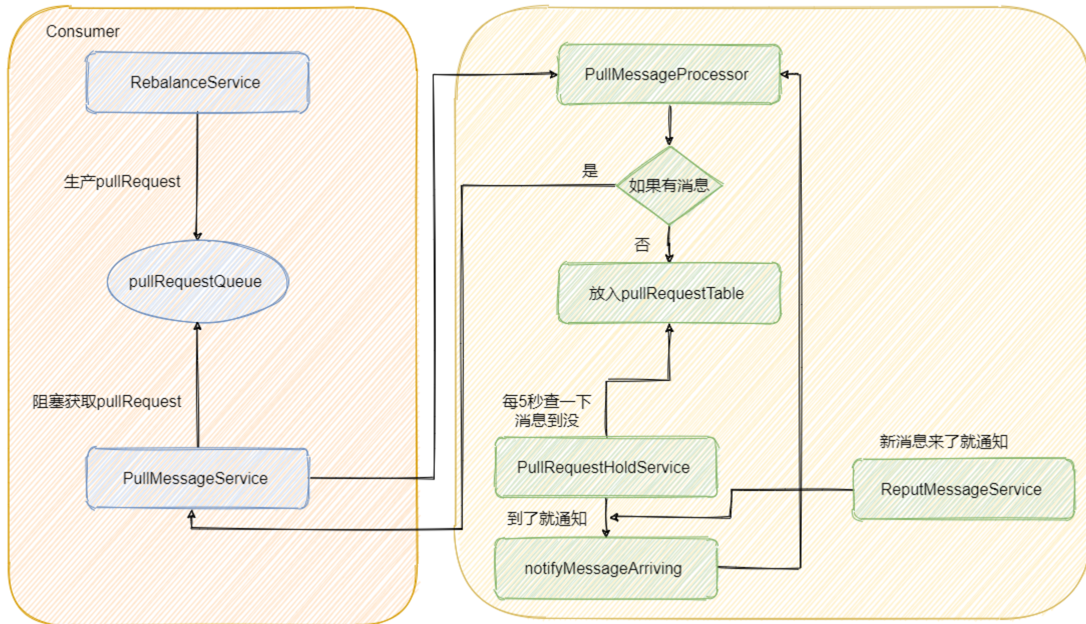
```java
public void run() {
    while (!this.isStopped()) {
        try {
            if (this.brokerController.getBrokerConfig().isLongPollingEnable()) {
                this.waitForRunning(5 * 1000); // 5秒
            } else {
                this.waitForRunning(this.brokerController.getBrokerConfig().getShortPollingTimeMills());
            }

            long beginLockTimestamp = this.systemClock.now();
            this.checkHoldRequest(); //这个动作就是去 pullRequestTable 查看请求
            long costTime = this.systemClock.now() - beginLockTimestamp;
            if (costTime > 5 * 1000) {
                log.info("[NOTIFYME] check hold request cost {} ms.", costTime);
            }
        } catch (Throwable e) {
            log.warn(this.getServiceName() + " service has exception. ", e);
        }
    }
}

private void checkHoldRequest() {
    for (String key : this.pullRequestTable.keySet()) {
        String[] kArray = key.split(TOPIC_QUEUEID_SEPARATOR);
        if (2 == kArray.length) {
            String topic = kArray[0];
            int queueId = Integer.parseInt(kArray[1]);
            //拿当前队列最大偏移量
            final long offset = this.brokerController.getMessageStore().getMaxOffsetInQueue(topic,
queueId);
            try {
                this.notifyMessageArriving(topic, queueId, offset); //通知数据是否到达
            } catch (Throwable e) {
                log.error("check hold request failed. topic={}, queueId={}", topic, queueId, e);
            }
        }
    }
}

 notifyMessageArriving 最终调用下面的方法

 this.brokerController.getPullMessageProcessor().executeRequestWhenWakeup()

 而 executeRequestWhenWakeup 最终调用下面的方法

 final RemotingCommand response = PullMessageProcessor.this.processRequest(channel, request, false);
```

简单的说就是 5 秒会检查一次消息时候到了，如果到了则调用 processRequest 再处理一次。这好像不太实时啊？ 5秒？

别急，还有个 ReputMessageService 线程，这个线程用来不断地从 commitLog 中解析数据并分发请求，构建出 ConsumeQueue 和 IndexFile 两种类型的数据，**并且也会有唤醒请求的操作，来弥补每 5s 一次这么慢的延迟**

代码我就不截了，就是消息写入并且会调用 pullRequestHoldService#notifyMessageArriving。

最后我再来画个图，描述一下整个流程。



Kafka 中的长轮询

像 Kafka 在拉请求中有参数，可以使得消费者请求在 “长轮询” 中阻塞等待。

简单的说就是消费者去 Broker 拉消息，定义了一个超时时间，也就是说消费者去请求消息，如果有的话马上返回消息，如果没有的话消费者等着直到超时，然后再次发起拉消息请求。

并且 Broker 也得配合，如果消费者请求过来，有消息肯定马上返回，没有消息那就建立一个延迟操作，等条件满足了再返回。

我们来简单的看一下源码，为了突出重点，我会删减一些代码。

先来看消费者端的代码。

```
public ConsumerRecords<K, V> poll(final Duration timeout) {
    return poll(time.timer(timeout), true);
}

private ConsumerRecords<K, V> poll(final Timer timer, final boolean includeMetadataInTimeout) {
  .......
  try {
      // poll for new data until the timeout expires
      do {
          final Map<TopicPartition, List<ConsumerRecord<K, V>>> records = pollForFetches(timer);
          if (!records.isEmpty()) {
              if (fetcher.sendFetches() > 0 || client.hasPendingRequests()) {
                  client.transmitSends();
              }
              return this.interceptors.onConsume(new ConsumerRecords<>(records));
          }
      } while (timer.notExpired());

      return ConsumerRecords.empty();
    }
  .......
}
```

上面那个 poll 接口想必大家都很熟悉，其实从注解直接就知道了确实是等待数据的到来或者超时，我们再简单的往下看。

```
private Map<TopicPartition, List<ConsumerRecord<K, V>>> pollForFetches(Timer timer) {
    long pollTimeout = coordinator == null ? timer.remainingMs() :
            Math.min(coordinator.timeToNextPoll(timer.currentTimeMs()), timer.remainingMs());

    // if data is available already, return it immediately
    final Map<TopicPartition, List<ConsumerRecord<K, V>>> records = fetcher.fetchedRecords();
    if (!records.isEmpty()) { //如果已经有消息直接返回
        return records;
    }

    // send any new fetches (won't resend pending fetches)
    fetcher.sendFetches(); //只是构造请求还未发送

    Timer pollTimer = time.timer(pollTimeout);
    client.poll(pollTimer, () -> { //真正的发送了
        return !fetcher.hasAvailableFetches();
    });

    return fetcher.fetchedRecords();
}
```

我们再来看下最终 client.poll 调用的是什么。

```
public List<ClientResponse> poll(long timeout, long now) {
    ensureActive();

    if (!abortedSends.isEmpty()) {
        .....
    }

    long metadataTimeout = metadataUpdater.maybeUpdate(now);
    try {
        this.selector.poll(Utils.min(timeout, metadataTimeout, defaultRequestTimeoutMs));
    } catch (IOException e) {
        log.error("Unexpected error during I/O", e);
    }

    // process completed actions
    long updatedNow = this.time.milliseconds();
    List<ClientResponse> responses = new ArrayList<>();
    handleCompletedSends(responses, updatedNow);
    handleCompletedReceives(responses, updatedNow);
    handleDisconnections(responses, updatedNow);
    handleConnections();
    handleInitiateApiVersionRequests(updatedNow);
    handleTimedOutRequests(responses, updatedNow);
    completeResponses(responses);

    return responses;
}
```

最后**调用的就是 Kafka 包装过的 selector，而最终会调用 Java nio 的 select(timeout)。**

现在消费者端的代码已经清晰了，**我们再来看看 Broker 如何做的。**

Broker 处理所有请求的入口其实我在之前的文章介绍过，就在 KafkaApis.scala 文件的 handle 方法下，这次的主角就是 handleFetchRequest 。

```
/**
 * Top-level method that handles all requests and multiplexes to the right api
 */
def handle(request: RequestChannel.Request): Unit = {
  try {
    request.header.apiKey match {
      case ApiKeys.PRODUCE => handleProduceRequest(request)
      case ApiKeys.FETCH => handleFetchRequest(request)
      ............
```

这个方法进来，我截取最重要的部分。

```
//根据请求和配置，确定最大拉多少，最小拉多少
val fetchMaxBytes = Math.min(Math.min(fetchRequest.maxBytes, config.fetchMaxBytes), maxQuotaWindowBytes)
val fetchMinBytes = Math.min(fetchRequest.minBytes, fetchMaxBytes)

replicaManager.fetchMessages(
  fetchRequest.maxWait.toLong, //最大等待时间
  fetchRequest.replicaId,
  fetchMinBytes,
  fetchMaxBytes,
  versionId <= 2,
  interesting,
  replicationQuota(fetchRequest),
  processResponseCallback,
  fetchRequest.isolationLevel,
  clientMetadata)
```

下面的图片就是 fetchMessages 方法内部实现，源码给的注释已经很清晰了，大家放大图片看下即可。

```
    // respond immediately if 1) fetch request does not want to wait
    //                        2) fetch request does not require any data
    //                        3) has enough data to respond
    //                        4) some error happens while reading data
    if (timeout <= 0 || fetchInfos.isEmpty || bytesReadable >= fetchMinBytes || errorReadingData) {
      responseCallback(fetchPartitionData) //如果符合以上注释直接返回数据
    } else {
      //构建延迟操作
      val delayedFetch = new DelayedFetch(timeout, fetchMetadata, this, quota, clientMetadata,
        responseCallback)

      // create a list of (topic, partition) pairs to use as keys for this delayed fetch operation
      val delayedFetchKeys = fetchPartitionStatus.map { case (tp, _) => TopicPartitionOperationKey(tp) }
      //放入延迟拉取的炼狱中!
      delayedFetchPurgatory.tryCompleteElseWatch(delayedFetch, delayedFetchKeys)
    }
```

这个炼狱名字取得很有趣，简单的说就是利用我之前文章提到的时间轮，来执行定时任务，例如这里是 `delayedFetchPurgatory`，专门用来处理延迟拉取操作。

我们先简单想一下，这个延迟操作都需要实现哪些方法，首先构建的延迟操作需要有检查机制，来查看消息是否已经到了，然后呢还得有个消息到了之后该执行的方法，还需要有执行完毕之后该干啥的方法，当然还得有个超时之后得干啥的方法。

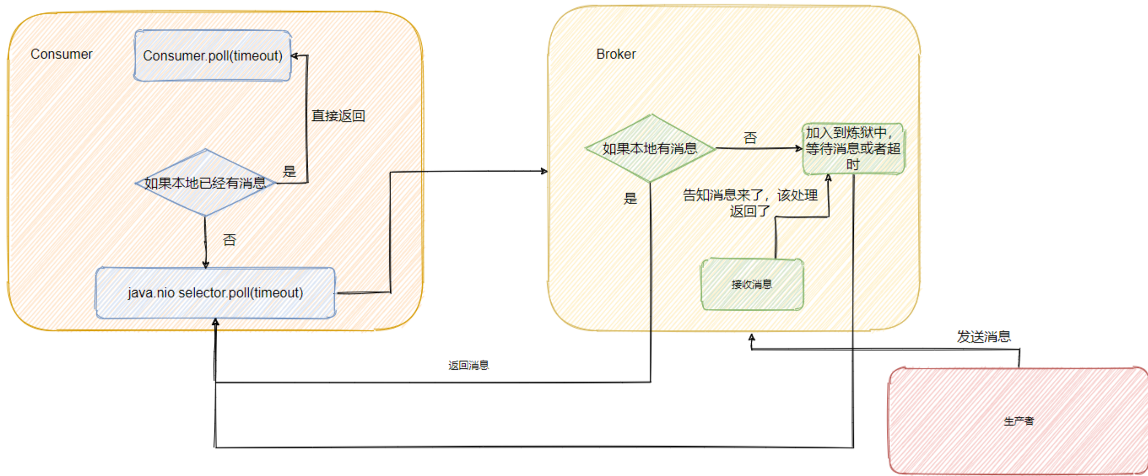
这几个方法其实对应的就是代码里的 DelayedFetch ，这个类继承了 DelayedOperation 内部有：

- isCompleted 检查条件是否满足的方法
- tryComplete 条件满足之后执行的方法
- onComplete 执行完毕之后调用的方法
- onExpiration 过期之后需要执行的方法

判断是否过期就是由时间轮来推动判断的，但是总不能等过期的时候再去看消息到了没吧？

这里 Kafka 和 RocketMQ 的机制一样，也会在消息写入的时候提醒这些延迟请求消息来了，具体代码我不贴了， 在 ReplicaManager#appendRecords 方法内部再深入个两方法可以看到。

不过虽说代码不贴，图还是要画一下的。



小结一下

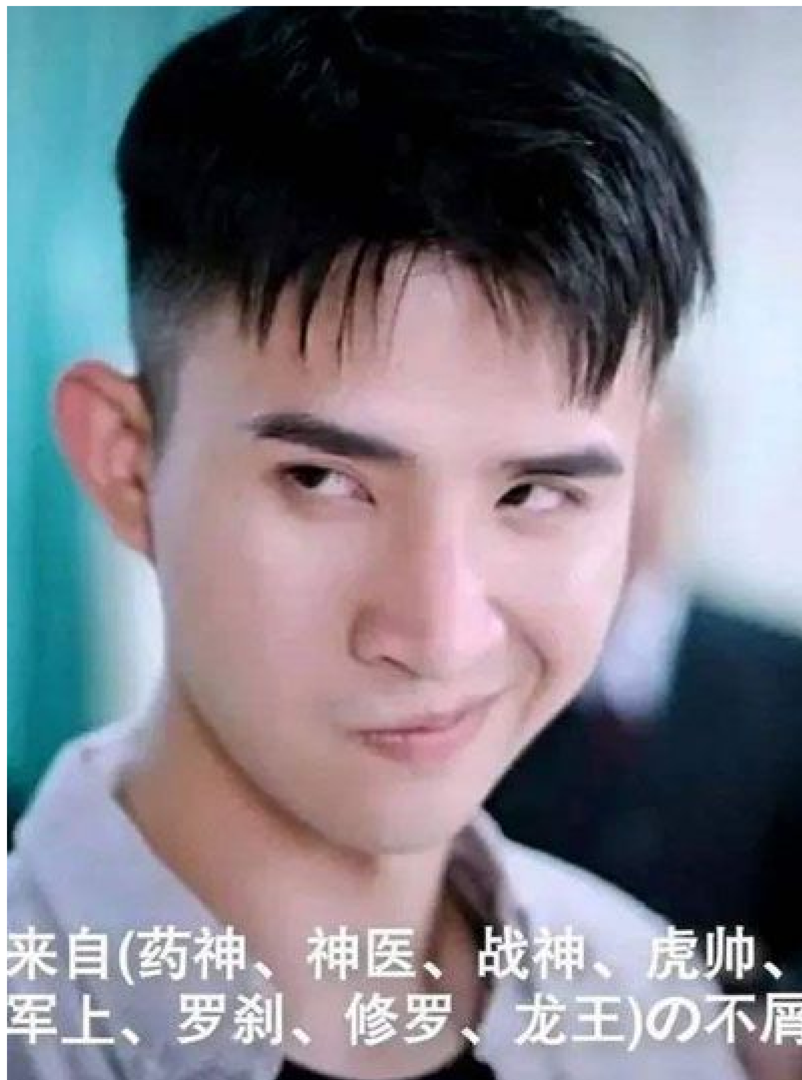
可以看到 RocketMQ 和 Kafka 都是采用“长轮询”的机制，具体的做法都是通过消费者等待消息，当有消息的时候 Broker 会直接返回消息，如果没有消息都会采取延迟处理的策略，并且为了保证消息的及时性，在对应队列或者分区有新消息到来的时候都会提醒消息来了，及时返回消息。

一句话说就是消费者和 Broker 相互配合，拉取消息请求不满足条件的时候 hold 住，避免了多次频繁的拉取动作，当消息一到就提醒返回。

最后

总的而言推拉模式各有优劣，而我个人觉得一般情况下拉模式更适合于消息队列。

看了这篇文章相信之后面试官问你推还是拉？建议给他个歪嘴笑。



消息队列之事务消息？RocketMQ和Kafka是怎么做的？

今天我们来谈一谈消息队列的事务消息，一说起事务相信大家都不陌生，脑海里蹦出来的就是 ACID。

通常我们理解的事务就是为了一些更新操作要么都成功，要么都失败，不会有中间状态的产生，而 ACID 是一个严格的事务实现的定义，不过在单体系统时候一般都不会严格的遵循 ACID 的约束来实现事务，更别说分布式系统了。

分布式系统往往只能妥协到最终一致性，保证数据最终的完整性和一致性，主要原因就是实力不允许…因为可用性为王。

而且要保证完全版的事务实现代价很大，你想想要维护这么多系统的数据，不允许有中间状态数据可以被读取，所有的操作必须不可分割，这意味着一个事务的执行是阻塞的，资源是被长时间锁定的。

在高并发情况下资源被长时间的占用，就是致命的伤害，举一个有味道的例子，如厕高峰期，好了懂得都懂。

对了， ACID 是什么还不太清楚的同学，赶紧去查一查，这里我就不展开说了。

分布式事务

那说到分布式事务，常见的有 2PC、TCC 和事务消息，这篇文章重点就是事务消息，不过 2PC 和 TCC 我稍微提一下。

2PC

2PC 就是二阶段提交，分别有协调者和参与者两个角色，二阶段分别是准备阶段和提交阶段。

准备阶段就是协调者向各参与者发送准备命令，这个阶段参与者除了事务的提交啥都做了，而提交阶段就是协调者看看各个参与者准备阶段都 o 不 ok，如果有 ok 那么就向各个参与者发送提交命令，如果有一个不 ok 那么就发送回滚命令。

这里的重点就是 **2PC 只适用于数据库层面的事务**，什么意思呢？就是你想在数据库里面写一条数据同时又要上传一张图片，这两个操作 2PC 无法保证两个操作满足事务的约束。

而且 2PC 是一种**强一致性**的分布式事务，它是**同步阻塞**的，即在接收到提交或回滚命令之前，所有参与者都是互相等待，特别是执行完准备阶段的时候，此时的资源都是锁定的状态，假如有一个参与者卡了很久，其他参与者都得等它，**产生长时间资源锁定状态下的阻塞**。

总体而言效率低，并且存在**单点故障**问题，协调者是就是那个单点，并且在极端条件下存在**数据不一致**的风险，例如某个参与者未收到提交命令，此时宕机了，恢复之后数据是回滚的，而其他参与者其实都已经执行了提交事务的命令了。

TCC

TCC 能保证业务层面的事务，也就是说它不仅仅是数据库层面，上面的上传图片这种操作它也能做。

TCC 分为三个阶段 try - confirm - cancel，简单的说就是每个业务都需要有这三个方法，先都执行 try 方法，这一阶段不会做真正的业务操作，只是先占个坑，什么意思呢？比如打算加 10 个积分，那先在预添加字段加上这 10 积分，这个时候用户账上的积分其实是没有增加的。

然后如果都 try 成功了那么就执行 confirm 方法，大家都来做真正的业务操作，如果有一个 try 失败了那么大家都执行 cancel 操作，来撤回刚才的修改。

可以看到 **TCC 其实对业务的耦合性很大**，因为业务上需要做一定的改造才能完成这三个方法，这其实就是 TCC 的缺点，**并且 confirm 和 cancel 操作要注意幂等**，因为到执行这两步的时候没有退路，是务必要完成的，因此需要有重试机制，所以需要保证方法幂等。

事务消息

事务消息就是今天文章的主角了，它**主要是适用于异步更新的场景，并且对数据实时性要求不高的地方。**

它的目的是为了**解决消息生产者与消息消费者的数据一致性问题。**

比如你点外卖，我们先选了炸鸡加入购物车，又选了瓶可乐，然后下单，付完款这个流程就结束了。

而购物车里面的数据就很适合用消息通知异步删除，因为一般而言我们下完单不会再去点开这个店家的菜单，而且就算点开了购物车里还有这些菜品也没有关系，影响不大。

我们希望的就是下单成功之后购物车的菜品最终会被删除，所以要点就是**下单和发消息这两个步骤要么都成功要么都失败。**

RocketMQ 事务消息

我们先来看一下 RocketMQ 是如何实现事务消息的。

RocketMQ 的事务消息也可以被认为是一个两阶段提交，简单的说就是在事务开始的时候会先发送一个半消息给 Broker。

半消息的意思就是这个消息此时对 Consumer 是不可见的，而且也不是存在真正要发送的队列中，而是一个特殊队列。

发送完半消息之后再执行本地事务，再根据本地事务的执行结果来决定是向 Broker 发送提交消息，还是发送回滚消息。

此时有人说这一步发送提交或者回滚消息失败了怎么办？

影响不大，Broker 会定时的向 Producer 来反查这个事务是否成功，具体的就是 Producer 需要暴露一个接口，通过这个接口 Broker 可以得知事务到底有没有执行成功，没成功就返回未知，因为有可能事务还在执行，会进行多次查询。

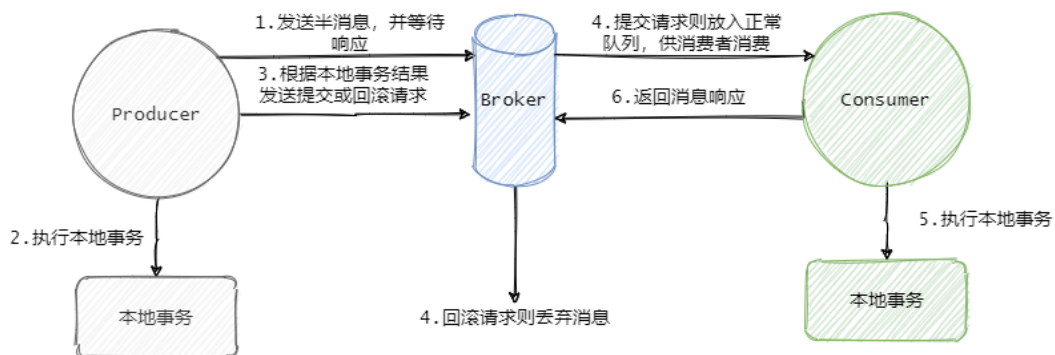
如果成功那么就将半消息恢复到正常要发送的队列中，这样消费者就可以消费这条消息了。

我们再来简单的看下如何使用，我根据官网示例代码简化了下。

```java
public class TransactionProducer {
  public static void main(String[] args) throws MQClientException, InterruptedException {
      TransactionListener transactionListener = new TransactionListenerImpl();//这个下面会展示具体内容
      TransactionMQProducer producer = new TransactionMQProducer("yes"); //创建TransactionMQProducer
      ExecutorService executorService = new ThreadPoolExecutor(...);

      producer.setExecutorService(executorService);
      producer.setTransactionListener(transactionListener);    构建 producer
      producer.start(); //开启 producer

      SendResult sendResult = producer.sendMessageInTransaction(msg, null);//发送事务消息
      System.out.printf("%s%n", sendResult);

  }
}

public class TransactionListenerImpl implements TransactionListener {
    //类似于订单id
    private AtomicInteger transactionIndex = new AtomicInteger(0);
    //保存本地事务的结果
    private ConcurrentHashMap<String, Integer> localTrans = new ConcurrentHashMap<>();

    @Override
    //执行本地事务的方法
    public LocalTransactionState executeLocalTransaction(Message msg, Object arg) {
        int value = transactionIndex.getAndIncrement();
        int status = value % 3;                                  执行本地事务的方法
        localTrans.put(msg.getTransactionId(), status);
        return LocalTransactionState.UNKNOW;
    }

    @Override
    //给 Broker 使用的反查事务结果的方法
    public LocalTransactionState checkLocalTransaction(MessageExt msg) {
        Integer status = localTrans.get(msg.getTransactionId());
        if (null != status) {                                    反查事务结果的方法
            switch (status) {
                case 0:
                    return LocalTransactionState.UNKNOW;
                case 1:
                    return LocalTransactionState.COMMIT_MESSAGE;
                case 2:
                    return LocalTransactionState.ROLLBACK_MESSAGE;
            }
        }
        return LocalTransactionState.COMMIT_MESSAGE;
    }
}
```

可以看到使用起来还是很简便直观的，无非就是多加个反查事务结果的方法，然后把本地事务执行的过程写在 TransationListener 里面。

至此 RocketMQ 事务消息大致的流程已经清晰了，我们画一张整体的流程图来过一遍，其实到第四步这个消息要么就是正常的消息，要么就是抛弃什么都不存在，此时这个事务消息已经结束它的生命周期了。



RocketMQ 事务消息源码分析

然后我们再从源码的角度来看看到底是怎么做的，首先我们看下 `sendMessageInTransaction` 方法，方法有点长，不过没有关系结构还是很清晰的。

```
public TransactionSendResult sendMessageInTransaction(final Message msg,
    final LocalTransactionExecuter localTransactionExecuter, final Object arg)
    throws MQClientException {
    ......
    SendResult sendResult = null;
    // 标记此时的消息，为半消息
    MessageAccessor.putProperty(msg, MessageConst.PROPERTY_TRANSACTION_PREPARED, "true");
    MessageAccessor.putProperty(msg, MessageConst.PROPERTY_PRODUCER_GROUP, this.defaultMQProducer.getProducerGroup());
    try {
        sendResult = this.send(msg); //发送半消息
    } catch (Exception e) {
        throw new MQClientException("send message Exception", e);
    }

    LocalTransactionState localTransactionState = LocalTransactionState.UNKNOW;
    Throwable localException = null;
    switch (sendResult.getSendStatus()) {
        case SEND_OK: { //如果半消息发送成功
            try {
                if (sendResult.getTransactionId() != null) {
                    msg.putUserProperty("__transactionId__", sendResult.getTransactionId());
                }
                String transactionId = msg.getProperty(MessageConst.PROPERTY_UNIQ_CLIENT_MESSAGE_ID_KEYIDX);
                if (null != transactionId && !"".equals(transactionId)) {
                    msg.setTransactionId(transactionId);
                }
                // 执行本地事务，这个localTransactionExecuter已经废弃了
                if (null != localTransactionExecuter) {
                    localTransactionState = localTransactionExecuter.executeLocalTransactionBranch(msg, arg);
                } else if (transactionListener != null) { //所以应该使用这个
                    log.debug("Used new transaction API");
                    localTransactionState = transactionListener.executeLocalTransaction(msg, arg);
                }
                if (null == localTransactionState) {
                    localTransactionState = LocalTransactionState.UNKNOW;
                }
                ....
            } catch (Throwable e) {
                log.info("executeLocalTransactionBranch exception", e);
                log.info(msg.toString());
                localException = e;
            }
        }
        break;
         //如果半消息发送失败，发送回滚消息请求
        case FLUSH_DISK_TIMEOUT:
        case FLUSH_SLAVE_TIMEOUT:
        case SLAVE_NOT_AVAILABLE:
            localTransactionState = LocalTransactionState.ROLLBACK_MESSAGE;
            break;
        default:
            break;
    }

    try {
        //向 Broker 发送本地事务结果，这是一个 oneway 的发送，即单向的不会等待 Broker 的响应
        this.endTransaction(sendResult, localTransactionState, localException);
    } catch (Exception e) {
        log.warn("local transaction execute " + localTransactionState + ", but end broker transaction failed", e);
    }

    TransactionSendResult transactionSendResult = new TransactionSendResult();
    transactionSendResult.setSendStatus(sendResult.getSendStatus());
    transactionSendResult.setMessageQueue(sendResult.getMessageQueue());
    transactionSendResult.setMsgId(sendResult.getMsgId());
    transactionSendResult.setQueueOffset(sendResult.getQueueOffset());
    transactionSendResult.setTransactionId(sendResult.getTransactionId());
    transactionSendResult.setLocalTransactionState(localTransactionState);

    return transactionSendResult;
}
```

构建半消息，发送至 Broker

执行本地事务

单向发送事务结果给 Broker

组装此时事务结果返回

流程也就是我们上面分析的，将消息塞入一些属性，标明此时这个消息还是半消息，然后发送至 Broker，然后执行本地事务，然后将本地事务的执行状态发送给 Broker ，我们现在**再来看下 Broker 到底是怎么处理这个消息的。**

在 Broker 的 SendMessageProcessor#sendMessage 中会处理这个半消息请求，因为今天主要分析的是事务消息，所以其他流程不做分析，我大致的说一下原理。

简单的说就是 sendMessage 中查到接受来的消息的属性里面 `MessageConst.PROPERTY_TRANSACTION_PREPARED` 是 true ，那么可以得知这个消息是事务消息，然后再判断一下这条消息是否超过最大消费次数，是否要延迟，Broker 是否接受事务消息等操作后，将这条消息真正的 topic 和队列存入属性中，然后重置消息的 topic 为 `RMQ_SYS_TRANS_HALF_TOPIC` ，并且队列是 0 的队列中，使得消费者无法读取这个消息。

以上就是整体处理半消息的流程，我们来看一下源码。

```
String traFlag = oriProps.get(MessageConst.PROPERTY_TRANSACTION_PREPARED); //获取这个属性值
if (traFlag != null && Boolean.parseBoolean(traFlag)
    && !(msgInner.getReconsumeTimes() > 0 && msgInner.getDelayTimeLevel() > 0)) { //各种判断
    if (this.brokerController.getBrokerConfig().isRejectTransactionMessage()) {//如果broker不接受事务消息，就返回NO_PERMISSION
        response.setCode(ResponseCode.NO_PERMISSION);
        response.setRemark(
            "the broker[" + this.brokerController.getBrokerConfig().getBrokerIP1()
                + "] sending transaction message is forbidden");
        return response;
    }
    putMessageResult = this.brokerController.getTransactionalMessageService().prepareMessage(msgInner); //这里就是修改操作了
} else {
    putMessageResult = this.brokerController.getMessageStore().putMessage(msgInner); //条件不满足就正常写入
}

//prepareMessage 实际是调用 transactionalMessageBridge.putHalfMessage(messageInner)，然后里面的 parseHalfMessageInner 方法是修改
的逻辑，我们就直接来看修改的方法。

private MessageExtBrokerInner parseHalfMessageInner(MessageExtBrokerInner msgInner) {
    MessageAccessor.putProperty(msgInner, MessageConst.PROPERTY_REAL_TOPIC, msgInner.getTopic()); //把真的 topic 塞入属性中
    MessageAccessor.putProperty(msgInner, MessageConst.PROPERTY_REAL_QUEUE_ID,
        String.valueOf(msgInner.getQueueId())); //把真的队列id塞入属性中
    msgInner.setSysFlag(
        MessageSysFlag.resetTransactionValue(msgInner.getSysFlag(), MessageSysFlag.TRANSACTION_NOT_TYPE));
    msgInner.setTopic(TransactionalMessageUtil.buildHalfTopic()); //将topic设置成 RMQ_SYS_TRANS_HALF_TOPIC
    msgInner.setQueueId(0); //队列 0
    msgInner.setPropertiesString(MessageDecoder.messageProperties2String(msgInner.getProperties()));
    return msgInner;
}
```

就是来了波狸猫换太子，其实延时消息也是这么实现的，最终将换了皮的消息入盘。

Broker 处理提交或者回滚消息的处理方法是 `EndTransactionProcessor#processRequest`，我们来看一看它做了什么操作。

```
switch (requestHeader.getCommitOrRollback()) {
    case MessageSysFlag.TRANSACTION_NOT_TYPE: {
        LOGGER.warn("...");
        return null;
    }

    case MessageSysFlag.TRANSACTION_COMMIT_TYPE: {
        break;
    }

    case MessageSysFlag.TRANSACTION_ROLLBACK_TYPE: {
        LOGGER.warn("...");
        break;
    }
    default:
        return null;
}

OperationResult result = new OperationResult();
if (MessageSysFlag.TRANSACTION_COMMIT_TYPE == requestHeader.getCommitOrRollback()) {
    //获取这条消息物理偏移量
    result = this.brokerController.getTransactionalMessageService().commitMessage(requestHeader);

  if (result.getResponseCode() == ResponseCode.SUCCESS) {
      RemotingCommand res = checkPrepareMessage(result.getPrepareMessage(), requestHeader); //做一些判断

      if (res.getCode() == ResponseCode.SUCCESS) {
          MessageExtBrokerInner msgInner = endMessageTransaction(result.getPrepareMessage()); //将消息的topic和queue 换回来
          .....
          RemotingCommand sendResult = sendFinalMessage(msgInner); //将消息写入真正的主题队列中
          if (sendResult.getCode() == ResponseCode.SUCCESS) {
              //写入成功，那么就删除这条半消息，实际上不是删除，而是将其加入了一个 half_op 队列，表示处理过了。
              this.brokerController.getTransactionalMessageService().deletePrepareMessage(result.getPrepareMessage());
          }
          return sendResult;
      }
      return res;
  }
} else if (MessageSysFlag.TRANSACTION_ROLLBACK_TYPE == requestHeader.getCommitOrRollback()) {
  result = this.brokerController.getTransactionalMessageService().rollbackMessage(requestHeader);
  if (result.getResponseCode() == ResponseCode.SUCCESS) {
      RemotingCommand res = checkPrepareMessage(result.getPrepareMessage(), requestHeader);
      if (res.getCode() == ResponseCode.SUCCESS) {
          this.brokerController.getTransactionalMessageService().deletePrepareMessage(result.getPrepareMessage());
      }
      return res;
  }
}
  response.setCode(result.getResponseCode());
  response.setRemark(result.getResponseRemark());
  return response;
```

先通过请求头判断类型，如果是no_type 不作处理，如果是提交则break，回滚则记录个日志

提交事务，则将消息topic和queue组装回来，然后写入commitlog中供消费者消费，此时的删除半小时是假的，实际上是将消息再写入hlaf_op队列表示处理过，到时候依据这个来判重

回滚操作和提交操作的差别就在于没有拼装消息写入commitlog 这一步

可以看到，如果是提交事务就是把皮再换回来写入真正的topic所属的队列中，供消费者消费，如果是回滚则是将半消息记录到一个 half_op 主题下，到时候后台服务扫描半消息的时候就依据其来判断这个消息已经处理过了。

那个后台服务就是 `TransactionalMessageCheckService` 服务，它会定时的扫描半消息队列，去请求反查接口看看事务成功了没，具体执行的就是`TransactionalMessageServiceImpl#check` 方法。

我大致说一下流程，这一步骤其实涉及到的代码很多，我就不贴代码了，有兴趣的同学自行了解。不过我相信用语言也是能说清楚的。

首先取半消息 topic 即`RMQ_SYS_TRANS_HALF_TOPIC`下的所有队列，如果还记得上面内容的话，就知道半消息写入的队列是 id 是 0 的这个队列，然后取出这个队列对应的 half_op 主题下的队列，即 `RMQ_SYS_TRANS_OP_HALF_TOPIC` 主题下的队列。

这个 half_op 主要是为了记录这个事务消息已经被处理过，也就是说已经得知此事务消息是提交的还是回滚的消息会被记录在 half_op 中。

然后调用 `fillOpRemoveMap` 方法，从 half_op 取一批已经处理过的消息来去重，将那些没有记录在 half_op 里面的半消息调用 `putBackHalfMsgQueue` 又写入了 commitlog 中，然后发送事务反查请求，这个反查请求也是 oneWay，即不会等待响应。当然此时的半消息队列的消费 offset 也会推进。

然后**producer**中的 ClientRemotingProcessor#processRequest 会处理这个请求，会把任务扔到 TransactionMQProducer 的线程池中进行，最终会调用上面我们发消息时候定义的 `checkLocalTransactionState` 方法，然后将事务状态发送给 Broker，也是用 oneWay 的方式。

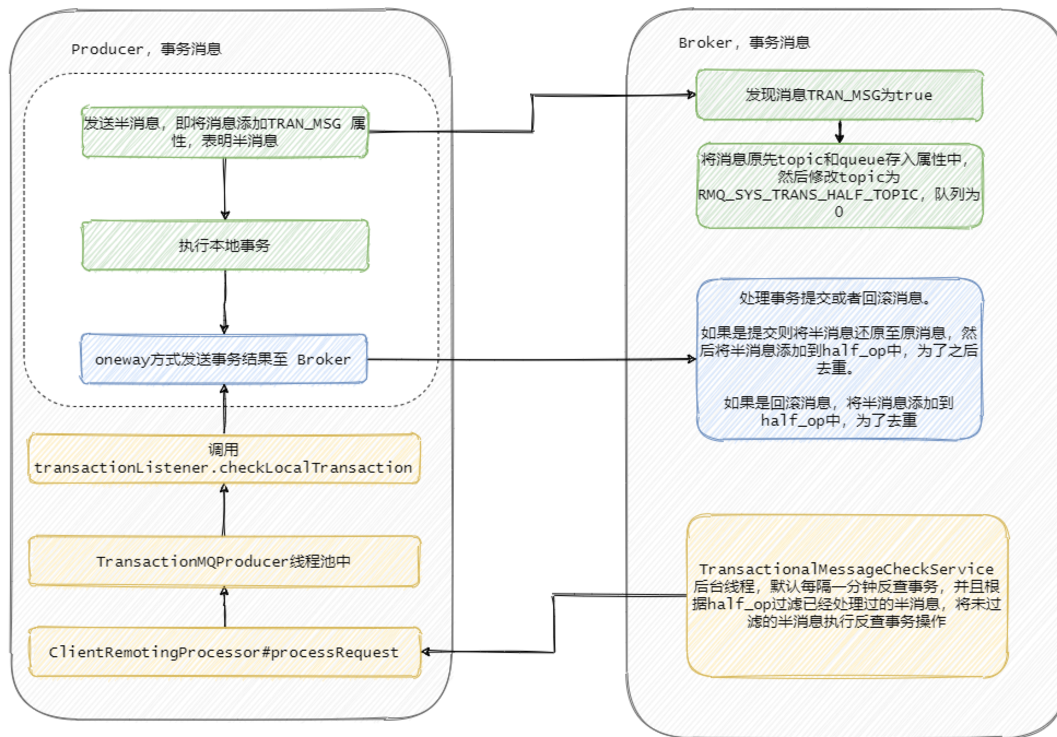
看到这里相信大家会有一些疑问，比如为什么要有个 half_op ，为什么半消息处理了还要再写入 commitlog 中别急听我一一道来。

首先 **RocketMQ 的设计就是顺序追加写入，所以说不会更改已经入盘的消息**，那事务消息又需要更新反查的次数，超过一定反查失败就判定事务回滚。

因此每一次要反查的时候就将以前的半消息再入盘一次，并且往前推进消费进度。而 half_op 又会记录每一次反查的结果，不论是提交还是回滚都会记录，因此下一次还循环到处理此半消息的时候，可以从 half_op 得知此事务已经结束了，因此就被过滤掉不需要处理了。

如果得到的反查的结果是 UNKNOW，那 half_op 中也不会记录此结果，因此还能再次反查，并且更新反查次数。

到现在整个流程已经清晰了，我再画个图总结一下 Broker 的事务处理流程。



Kafka 事务消息

Kafka 的事务消息和 RocketMQ 的事务消息又不一样了，RocketMQ 解决的是本地事务的执行和发消息这两个动作满足事务的约束。

而 Kafka 事务消息则是用在一次事务中需要发送多个消息的情况，保证多个消息之间的事务约束，即多条消息要么都发送成功，要么都发送失败，就像下面代码所演示的。

```
producer.initTransactions();// 初始化事务
try {
      producer.beginTransaction(); //开始事务
      producer.send(message1); //发送消息1
      producer.send(message2); //发送消息2
      producer.send(message3); //发送消息3
      producer.commitTransaction(); //提交消息
} catch (KafkaException e) {
    producer.abortTransaction(); //回滚消息
}
```

保证的是消息1、2、3要么都发送成功，要么都发送失败

Kafka 的事务基本上是配合其幂等机制来实现 Exactly Once 语义的，所以说 Kafka 的事务消息不是我们想的那种事务消息，RocketMQ 的才是。

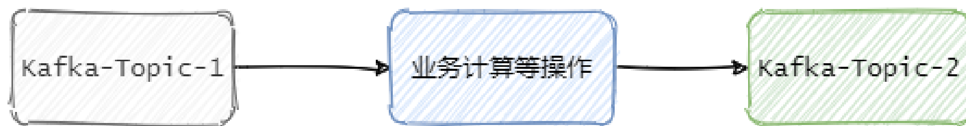
讲到这我就想扯一下了，说到这个 Exactly Once 其实不太清楚的同学很容易会误解。

我们知道消息可靠性有三种，分别是最多一次、恰好一次、最少一次，之前在消息队列连环问的文章我已经提到了基本上我们都是用最少一次然后配合消费者端的幂等来实现恰好一次。

消息恰好被消费一次当然我们所有人追求的，但是之前文章我已经从各方面已经分析过了，基本上难以达到。

而 Kafka 竟说它能实现 Exactly Once？这么牛啤吗？这其实是 Kafka 的一个噱头，你要说他错，他还真没错，你要说他对但是他实现的 Exactly Once 不是你心中想的那个 Exactly Once。

它的恰好一次只能存在一种场景，就是从 **Kafka 作为消息源，然后做了一番操作之后，再写入 Kafka 中**。



那他是如何实现恰好一次的？就是通过幂等，和我们在业务上实现的一样通过一个唯一 Id， 然后记录下来，如果已经记录过了就不写入，这样来保证恰好一次。

所以说 **Kafka 实现的是在特定场景下的恰好一次，不是我们所想的利用 Kafka 来发送消息，那么这条消息只会恰巧被消费一次。**

这其实和 Redis 说他实现事务了一样，也不是我们心想的事务。

所以开源软件说啥啥特性开发出来了，我们一味的相信，因此其往往都是残血的或者在特殊的场景下才能满足，不要被误导了，不能相信表面上的描述，还得详细的看看文档或者源码。

不过从另一个角度看也无可厚非，作为一个开源软件肯定是想更多的人用，我也没说谎呀，我文档上写的很清楚的，这标题也没骗人吧？

确实，比如你点进震惊xxxx标题的文章，人家也没骗你啥，他自己确实震惊的呢。

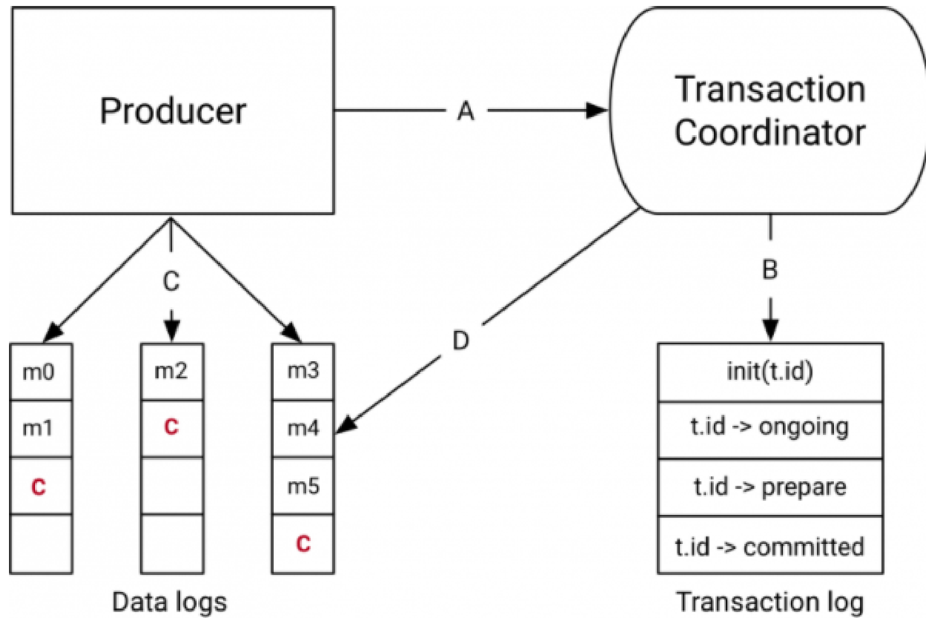
再回来谈 Kafka 的事务消息，所以说这个事务消息不是我们想要的那个事务消息，其实不是今天的主题了，不过我还是简单的说一下。

Kafka 的事务有事务协调者角色，事务协调者其实就是 Broker 的一部分。

在开始事务的时候，生产者会向事务协调者发起请求表示事务开启，事务协调者会将这个消息记录到特殊的日志-事务日志中，然后生产者再发送真正想要发送的消息，这里 Kafka 和 RocketMQ 处理不一样，Kafka 会像对待正常消息一样处理这些事务消息，**由消费端来过滤这个消息。**

然后发送完毕之后生产者会向事务协调者发送提交或者回滚请求，由事务协调者来进行两阶段提交，如果是提交那么会先执行预提交，即把事务的状态置为预提交然后写入事务日志，然后再向所有事务有关的分区写入一条类似事务结束的消息，这样消费端消费到这个消息的时候就知道事务好了，可以把消息放出来了。

最后协调者会向事务日志中再记一条事务结束信息，至此 Kafka 事务就完成了，我拿 confluent.io 上的图来总结一下这个流程。



最后

至此我们已经知道了 RocketMQ 和 Kakfa 的事务消息全流程，可以看到 RocketMQ 的事务消息才是我们想要的，当然你要是用的流式计算那么 Kakfa 的事务消息也是你想要的。

比 RocketMQ 更好的事务消息实现是什么？

先抛出的一个问题：一个事务涉及 mysql 和 mq，到底哪个写入成功重要？

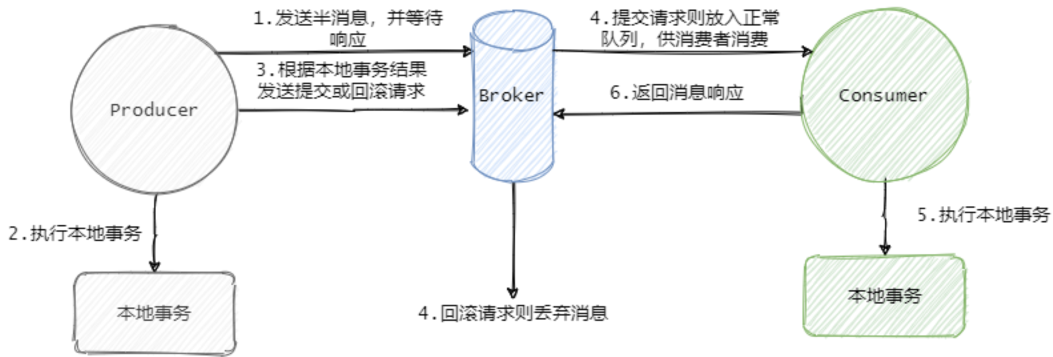
假如线上 mq 集群网络故障，导致发消息失败，即使 mysql 还是活着的，但是无法进行事务。

所以其实这个问题问的是：mysql 和 mq 之间的写入顺序。

在 RocketMQ 中，事务消息的实现方案是先发半消息（半消息对消费者不可见），待半消息发送成功之后，才能执行本地事务，等本地事务执行成功之后，再向 Broker 发送请求将半消息转成正常消息，这样消费者就可以消费此消息。

这种顺序等于先得成功写入 mq，然后再写入数据库，这样的模式会出现一个问题：**即 mq 集群挂了，事务就无法继续进行了，等于整个应用无法正常执行了。**

看一下我之前画的 RocketMQ 事务消息流程图:



第一步是需要等待半消息的响应，如果响应失败就无法执行本地事务。

看下伪代码，可能更清晰:

```
result = sendHalfMsg();//发送半消息
if(result.success) {
  执行本地事务
} else {
  回滚此次事务
}
```

具体 RocketMQ 事务细节看我这篇：[RocketMQ与Kafka之事务消息]

所以，先写 mq 后写 mysql 就会发生 mysql 还好好的，但是 mq 挂了事务就无法正常执行的情况。

那 qmq 怎么做的呢?

> PS: QMQ是去哪儿网内部广泛使用的消息中间件，自2012年诞生以来在去哪儿网所有业务场景中广泛的应用，包括跟交易息息相关的订单场景； 也包括报价搜索等高吞吐量场景。目前在公司内部日常消息qps在60W左右，生产上承载将近4W+消息topic，消息的端到端延迟可以控制在10ms以内。

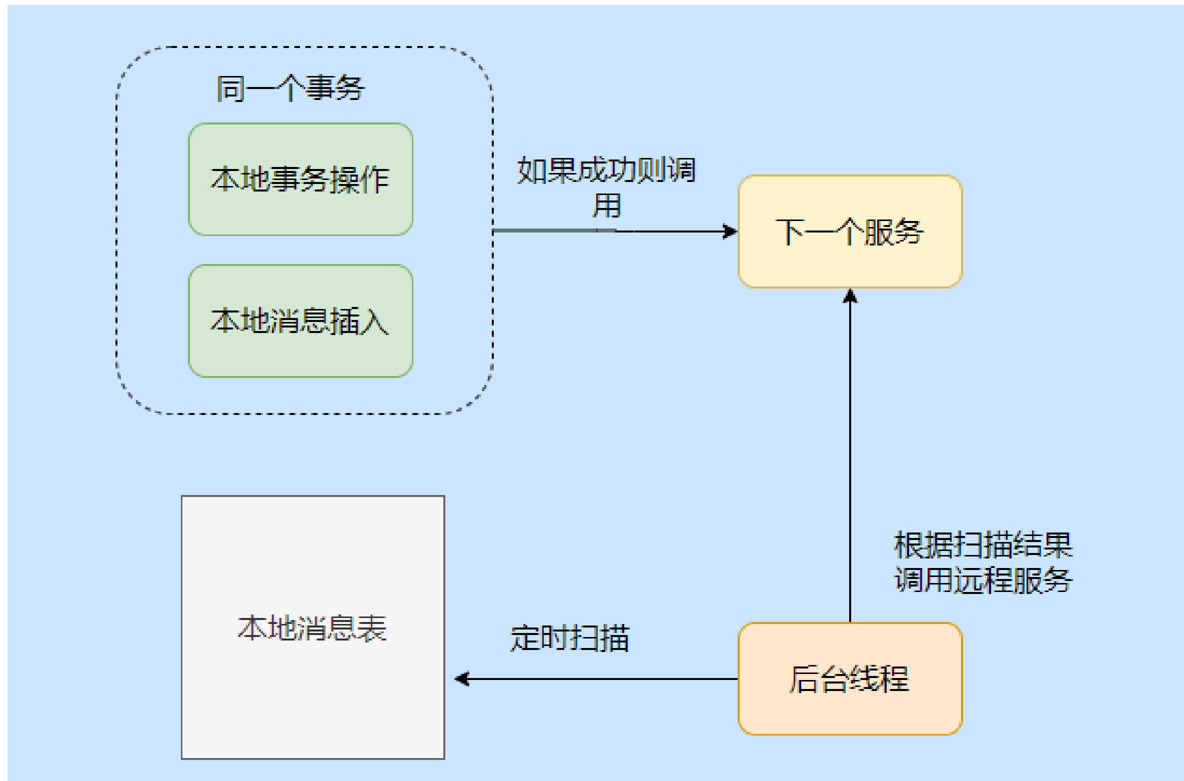
在说 qmq 的事务消息之前，先来说下本地消息表这个分布式事务实现方式。

本地消息就是利用了关系型数据库的事务能力，会在数据库中存放一张本地事务消息表，在进行本地事务操作中加入了本地消息表的插入，即**将业务的执行和将消息放入到消息表中的操作放在同一个事务中提交。**

这样本地事务执行成功的话，消息肯定也插入成功，然后再调用其他服务，如果其他服务调用成功就修改这条本地消息的状态。

如果失败也不要紧，会有一个后台线程扫描，发现这些状态的消息，会一直调用相应的服务，一般会设置重试的次数，如果一直不行则特殊记录，待人工介入处理。

可以看到，本地事务消息表还是很简单的，也是一种最大努力通知的思想。



在理解本地消息表之后，我们再来看一下 qmq 的事务消息是如何设计的。

首先，想要用 qmq 的事务消息，需要在数据库中建一张表，就是如下这样的表：

```sql
CREATE DATABASE `qmq_produce`;
CREATE TABLE `qmq_msg_queue` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `content` longtext NOT NULL,
  `status` smallint(6) NOT NULL DEFAULT '0' COMMENT '消息状态',
  `error` int unsigned NOT NULL DEFAULT '0' COMMENT '错误次数',
  `create_time` datetime NOT NULL COMMENT '创建时间',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='记录业务系统消息';
```

是不是有本地消息表那味儿了？

没错核心思想就是本地消息表！利用关系型数据库的事务能力，将业务的写入和消息表的写入融在一个事务中，这样业务成功则消息表肯定写入成功。

然后在事务提交之后，立刻发送事务消息，如果发送成功，则删除本地消息表中的记录，来看一下伪代码的实现，应该就很清晰了：

```java
@Transactional // 在一个事务中
public void yes(){
    Order order = buildOrder();
    orderDao.insert(order);
    Message message = buildMessage(order);
    messageDao.insert(message);
    //异步，在事务提交后执行
    triggerAfterTransactionCommit(()->{
        messageClient.send(message);
        messageDao.delete(message);
    });
}
```

当然，这是我剖开来写的实现思路，qmq的使用没这么麻烦，直接在 sendMessage 里把上面的逻辑都包装好了，所以使用起来直接就是一个发送消息：

```
@Transactional // 在一个事务中
public void yes(){
    Order order = buildOrder();
    orderDao.insert(order);
    //封装插入消息、发送消息、删除消息的逻辑
    producer.sendMessage(buildMessage(order));
}
```

如果消息发送失败，也就是比如 mq 集群挂了，并不会影响事务的执行，业务的执行和事务消息的插入都已经成功了，那此时待消息已经安安静静的在消息库里等着，后台能会有一个补偿任务，会将这些消息捞出来重新发送，直到发送成功。

想必，现在你应该对 qmq 的事务消息流程应该很清晰了，它的顺序就属于先写数据库，再发mq，即使 mq 集群挂了，也不会影响事务的进行，不会导致应用无法正常执行了。

这里可能有人会问，那如果 mysql 挂了呢?

我只能说数据库都挂了，那就都没了，别想啥别的了。

再来看 RocketMQ 和 QMQ

至此，想必你已经清楚 RocketMQ 和 QMQ 事务消息的区别，我们再来盘下 QMQ 事务消息更优的原因。

RocketMQ 只支持单事务消息，也就是无法在一个事务内发送多种事务消息。

而 QMQ 可以在一次事务中发多个消息，来看下伪代码：

```
@Transactional // 在一个事务中
public void yes(){
    Order order = buildOrder();
    orderDao.insert(order);
    producer.sendMessage(buildMessageA(order));
    producer.sendMessage(buildMessageB(order));
    producer.sendMessage(buildMessageC(order));
}
```

这样的实现就比 RocketMQ 灵活多了。

然后 RocketMQ 事务消息的实现还需要提供一个反查机制，因为RocketMQ 事务消息的提交是 oneway 的发送方式，有可能 Broker 没有接收到事务提交的消息，所以 Broker 会定时去生产者那边查看事务是否已经执行完成，因此生产者需要保存本地事务执行结果，简单的就是用一个 map 保存，让 Broker 可以通过消息的事务 id 查找到事务执行的结果。

```
public class TransactionListenerImpl implements TransactionListener {
    //类似于订单id
    private AtomicInteger transactionIndex = new AtomicInteger(0);
    //保存本地事务的结果
    private ConcurrentHashMap<String, Integer> localTrans = new ConcurrentHashMap<>();

    @Override
    //执行本地事务的方法
    public LocalTransactionState executeLocalTransaction(Message msg, Object arg) {
        int value = transactionIndex.getAndIncrement();
        int status = value % 3;                                  执行本地事务的方法
        localTrans.put(msg.getTransactionId(), status);
        return LocalTransactionState.UNKNOW;
    }

    @Override
    //给 Broker 使用的反查事务结果的方法
    public LocalTransactionState checkLocalTransaction(MessageExt msg) {
        Integer status = localTrans.get(msg.getTransactionId());
        if (null != status) {                                    反查事务结果的方法
            switch (status) {
                case 0:
                    return LocalTransactionState.UNKNOW;
                case 1:
                    return LocalTransactionState.COMMIT_MESSAGE;
                case 2:
                    return LocalTransactionState.ROLLBACK_MESSAGE;
            }
        }
        return LocalTransactionState.COMMIT_MESSAGE;
    }
}
```

如果还要考虑发送事务消息的生产者挂了，那么 Broker 会找同个生产组的其他生产者来查询事务结果，所以这个存储还得提出来放到第三方，而不是本地内存保存。

因此，RocketMQ 得多维护一个本地事务执行结果，是稍微有点麻烦的。

当然，QMQ 还得建表呢，不过按照 QMQ 说的：如果公司方便的话，可以直接合并进DBA的初始化数据库的自动化流程中，这样就透明了。

还有一点 RocketMQ 的 api 不太友好，改造有点大，之后的迁移不太方便。

贴一下完整的使用 RocketMQ 事务消息的代码：

```
public class TransactionProducer {
  public static void main(String[] args) throws MQClientException, InterruptedException {
      TransactionListener transactionListener = new TransactionListenerImpl();//这个下面会展示具体内容
      TransactionMQProducer producer = new TransactionMQProducer("yes"); //创建TransactionMQProducer
      ExecutorService executorService = new ThreadPoolExecutor(...);

      producer.setExecutorService(executorService);
      producer.setTransactionListener(transactionListener);    构建 producer
      producer.start(); //开启 producer

      SendResult sendResult = producer.sendMessageInTransaction(msg, null);//发送事务消息
      System.out.printf("%s%n", sendResult);

  }
}

public class TransactionListenerImpl implements TransactionListener {
    //类似于订单id
    private AtomicInteger transactionIndex = new AtomicInteger(0);
    //保存本地事务的结果
    private ConcurrentHashMap<String, Integer> localTrans = new ConcurrentHashMap<>();

    @Override
    //执行本地事务的方法
    public LocalTransactionState executeLocalTransaction(Message msg, Object arg) {
        int value = transactionIndex.getAndIncrement();
        int status = value % 3;                                          执行本地事务的方法
        localTrans.put(msg.getTransactionId(), status);
        return LocalTransactionState.UNKNOW;
    }

    @Override
    //给 Broker 使用的反查事务结果的方法
    public LocalTransactionState checkLocalTransaction(MessageExt msg) {
        Integer status = localTrans.get(msg.getTransactionId());
        if (null != status) {                                            反查事务结果的方法
            switch (status) {
                case 0:
                    return LocalTransactionState.UNKNOW;
                case 1:
                    return LocalTransactionState.COMMIT_MESSAGE;
                case 2:
                    return LocalTransactionState.ROLLBACK_MESSAGE;
            }
        }
        return LocalTransactionState.COMMIT_MESSAGE;
    }
}
```

可以看到，如果想要搞事务消息，首先新建 transationMQproducer，然后再新建一个 transcationListenerImpl，再覆盖 listener 执行事务的方法和回查事务的方法，等于你得把业务逻辑实现在 transcationListenerImpl 内部，这和我们平日里在 service 里面实现事务的差距就有点大了。

而 QMQ 提供了内置 Spring 事务的方式，所以就直接在 service 实现就行了。

```
@Transactional // 在一个事务中
public void yes(){
    Order order = buildOrder();
    orderDao.insert(order);
    producer.sendMessage(buildMessageA(order));
}
```

这就很贴合我们平日的使用方式了，这样对业务的改造很小，并且迁移也很方便。

最后

暂时就分析这么多了，对 QMQ 有兴趣的同学可以再自己研究一下，包括 QMQ 的消息模型多了个 pull log，**便于解决 consumer 的动态扩容缩容问题**，这也是比 RocketMQ 更灵活的一个地方。

当然，多了个中间层，效率应该会有所降低，这个我还没试验过。

还有 QMQ 的 Exactly once 消费等等，有机会之后再写一篇盘一盘。

Kafka 系列

Kafka的索引设计有什么亮点?

其实这篇文章只是从Kafka索引入手，来讲述算法在工程上基于场景的灵活运用。单单是因为看源码的时候有感而写之。

索引的重要性

索引对于我们来说并不陌生，每一本书籍的目录就是索引在现实生活中的应用。通过寥寥几页纸就得以让我等快速查找需要的内容。冗余了几页纸，缩短了查阅的时间。**空间和时间上的互换**，包含着宇宙的哲学。

工程领域上数据库的索引更是不可或缺，没有索引很难想象如此庞大的数据该如何检索。

明确了索引的重要性，咱再来看看索引在Kafka里是如何实现的。

索引在Kafka中的实践

首先Kafka的索引是**稀疏索引**，这样可以避免索引文件占用过多的内存，从而可以**在内存中保存更多的索引**。对应的就是Broker 端参数`log.index.interval.bytes` 值，默认4KB，即4KB的消息建一条索引。

Kafka中有三大类索引：位移索引、时间戳索引和已中止事务索引。分别对应了.index、.timeindex、.txnindex文件。

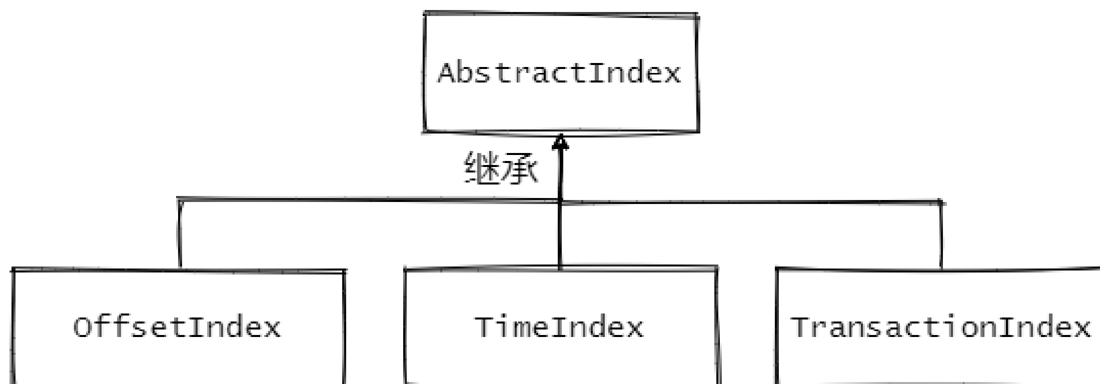
与之相关的源码如下：

1、AbstractIndex.scala：抽象类，**封装了所有索引的公共操作**

2、OffsetIndex.scala：位移索引，**保存了位移值和对应磁盘物理位置的关系**

3、TimeIndex.scala：时间戳索引，**保存了时间戳和对应位移值的关系**

4、TransactionIndex.scala：事务索引，启用Kafka事务之后才会出现这个索引（本文暂不涉及事务相关内容）



先来看看AbstractIndex的定义

```
abstract class AbstractIndex(@volatile private var _file: File, //索引对应的文件
                             val baseOffset: Long,    //索引文件的起始位移值
                             val maxIndexSize: Int = -1, //索引文件最大长度，对应segment.index.bytes，默认10MB
                             val writable: Boolean) //索引文件打开方式，false表示只读打开
                                          extends Closeable {
    protected def entrySize: Int
    protected def _warmEntries: Int = 8192 / entrySize
```

AbstractIndex的定义在代码里已经注释了，成员变量里面还有个`entrySize`。这个变量其实是每个索引项的大小，每个索引项的大小是固定的。

entrySize

在`OffsetIndex`中是`override def entrySize = 8`，8个字节。
在`TimeIndex`中是`override def entrySize = 12`,12个字节。

为何是8 和12?

在`OffsetIndex`中，每个索引项存储了位移值和对应的磁盘物理位置，因此4+4=8，但是不对啊，磁盘物理位置是整型没问题，但是`AbstractIndex`的定义baseOffset来看，位移值是长整型，不是因为8个字节么?

因此**存储的位移值实际上是相对位移值**，即**真实位移值-baseOffset的值**。

相对位移用整型存储够么？够，因为一个日志段文件大小的参数`log.segment.bytes`是整型，因此同一个日志段对应的index文件上的位移值-baseOffset的值的差值肯定在整型的范围内。

为什么要这么麻烦，还要存个差值?

1、为了节省空间，一个索引项节省了4字节，想想那些日消息处理数万亿的公司。

2、因为内存资源是很宝贵的，索引项越短，内存中能存储的索引项就越多，索引项多了直接命中的概率就高了。这其实和MySQL InnoDB 为何建议主键不宜过长一样。每个辅助索引都会存储主键的值，主键越长，每条索引项占用的内存就越大，缓存页一次从磁盘获取的索引数就越少，一次查询需要访问磁盘次数就可能变多。而磁盘访问我们都知道，很慢。

互相转化的源码如下，就这么个简单的操作:

```
private def toRelative(offset: Long): Option[Int] = {
  val relativeOffset = offset - baseOffset //真实位移值转相对位移值
  if (relativeOffset < 0 || relativeOffset > Int.MaxValue)
    None
  else
    Some(relativeOffset.toInt)
}
// 查找指定的索引项
override protected def parseEntry(buffer: ByteBuffer, n: Int /*第几个索引项*/): OffsetPosition = {
  OffsetPosition(baseOffset + relativeOffset(buffer, n)/*转成了真实的offset*/, physical(buffer, n))
}
                                                                       //相对位移 * 索引项大小
 private def relativeOffset(buffer: ByteBuffer, n: Int): Int = buffer.getInt(n * entrySize)

 private def physical(buffer: ByteBuffer, n: Int): Int = buffer.getInt(n * entrySize + 4)
```

上述解释了位移值是4字节，因此`TimeIndex`中时间戳8个字节 + 位移值4字节 = 12字节。

_warmEntries

这个是干什么用的?

首先思考下我们能通过索引项快速找到日志段中的消息，但是我们如何快速找到我们想要的索引项呢?一个索引文件默认10MB，一个索引项8Byte，因此一个文件可能包含100多W条索引项。

不论是消息还是索引，其实都是单调递增，并且都是追加写入的，因此数据都是有序的。在有序的集合中快速查询，脑海中突现的就是二分查找了！

那就来个二分！

```
def binarySearch(begin: Int, end: Int) : (Int, Int) = {
  // binary search for the entry
  var lo = begin
  var hi = end
  while(lo < hi) {
    val mid = (lo + hi + 1) >>> 1
    val found = parseEntry(idx, mid)
    val compareResult = compareIndexEntry(found, target, searchEntity)
    if(compareResult > 0)
      hi = mid - 1
    else if(compareResult < 0)
      lo = mid
    else
      return (mid, mid)
  }
  (lo, if (lo == _entries - 1) -1 else lo + 1)
}
```

这和 `_warmEntries` 有什么关系？首先想想二分有什么问题？

就Kafka而言，索引是在文件末尾追加的写入的，并且一般写入的数据立马就会被读取。所以数据的热点集中在尾部。并且操作系统基本上都是**用页为单位缓存和管理内存的，内存又是有限的**，因此会通过类LRU机制淘汰内存。

看起来LRU非常适合Kafka的场景，但是使用标准的二分查找会有缺页中断的情况，毕竟二分是跳着访问的。

这里要说一下kafka的注释写的是真的清晰，咱们来看看注释怎么说的

> when looking up index, the standard binary search algorithm is not cache friendly, and can cause unnecessary
> page faults (the thread is blocked to wait for reading some index entries from hard disk, as those entries are not
> cached in the page cache)

翻译下：当我们查找索引的时候，标准的二分查找对缓存不友好，可能会造成不必要的缺页中断(线程被阻塞等待从磁盘加载没有被缓存到page cache 的数据)

注释还友好的给出了例子

```
  For example, in an index with 13 pages, to lookup an entry in the last page (page #12), the standard binary search
   algorithm will read index entries in page #0, 6, 9, 11, and 12.
   page number: |0|1|2|3|4|5|6|7|8|9|10|11|12 |
   steps:       |1| | | | | |3| | |4|  |5 |2/6|
   In each page, there are hundreds log entries, corresponding to hundreds to thousands of kafka messages. When the
   index gradually growing from the 1st entry in page #12 to the last entry in page #12, all the write (append)
   operations are in page #12, and all the in-sync follower / consumer lookups read page #0,6,9,11,12. As these pages
   are always used in each in-sync lookup, we can assume these pages are fairly recently used, and are very likely to be
   in the page cache. When the index grows to page #13, the pages needed in a in-sync lookup change to #0, 7, 10, 12,
   and 13:
   page number: |0|1|2|3|4|5|6|7|8|9|10|11|12|13 |
   steps:       |1| | | | | | |3| | | 4|5 | 6|2/7|
   Page #7 and page #10 have not been used for a very long time. They are much less likely to be in the page cache, than
   the other pages. The 1st lookup, after the 1st index entry in page #13 is appended, is likely to have to read page #7
   and page #10 from disk (page fault), which can take up to more than a second. In our test, this can cause the
   at-least-once produce latency to jump to about 1 second from a few ms.
```

简单的来讲，假设某索引占page cache 13页，此时数据已经写到了12页。按照kafka访问的特性，此时访问的数据都在第12页，因此二分查找的特性，此时缓存页的访问顺序依次是0，6，9，11，12。因为频繁被访问，所以这几页一定存在page cache中。

当第12页不断被填充，满了之后会申请新页第13页保存索引项，而按照二分查找的特性，此时缓存页的访问顺序依次是：0，7，10，12。这7和10很久没被访问到了，很可能已经不再缓存中了，然后需要从磁盘上读取数据。注释说：**在他们的测试中，这会导致至少会产生从几毫秒跳到1秒的延迟。**

基于以上问题，Kafka使用了改进版的二分查找，改的不是二分查找的内部，而且把所有**索引项分为热区和冷区**

这个改进可以让**查询热数据部分时，遍历的Page永远是固定的**，这样能避免缺页中断。

看到这里其实我想到了**一致性hash，一致性hash相对于普通的hash不就是在node新增的时候缓存的访问固定，或者只需要迁移少部分数据**。

好了，让我们先看看源码是如何做的

```
  private def indexSlotRangeFor(idx: ByteBuffer, target: Long, searchEntity: IndexSearchEntity): (Int, Int) = {
    // check if the index is empty
    if(_entries == 0)
      return (-1, -1)

    def binarySearch(begin: Int, end: Int) : (Int, Int) = {
      // binary search for the entry
      // 就是刚才标准的二分查找代码，在此省略
      ....
    }
    //_warmEntries = 8192 / entrySize (在OffsetIndex中为8，TimeIndex为12)
    val firstHotEntry = Math.max(0, _entries - 1 - _warmEntries) //找到热区所在的第一个索引项
    // check if the target offset is in the warm section of the index
    //判断查找的数据是否在热区
    if(compareIndexEntry(parseEntry(idx, firstHotEntry), target, searchEntity) < 0) {
      return binarySearch(firstHotEntry, _entries - 1)
    }

    // check if the target offset is smaller than the least offset
    //确保要查找的位移不能小于当前最小位移
    if(compareIndexEntry(parseEntry(idx, 0), target, searchEntity) > 0)
      return (-1, 0)
    //冷区查找
    binarySearch(0, firstHotEntry)
  }
```

实现并不难，但是为何是把尾部的8192作为热区？

这里就要再提一下源码了，讲的很详细。

> 1. This number is small enough to guarantee all the pages of the "warm" section is touched in every warm-section lookup. So that, the entire warm section is really "warm".
>   When doing warm-section lookup, following 3 entries are always touched: indexEntry(end), indexEntry(end-N), and indexEntry((end*2 -N)/2). If page size >= 4096, all the warm-section pages (3 or fewer) are touched, when we
>   touch those 3 entries. As of 2018, 4096 is the smallest page size for all the processors (x86-32, x86-64, MIPS, SPARC, Power, ARM etc.).

> 大致内容就是现在处理器一般缓存页大小是4096，那么8192可以保证页数小于等3，用于二分查找的页面都能命中

> 2. This number is large enough to guarantee most of the in-sync lookups are in the warm-section. With default Kafka settings, 8KB index corresponds to about 4MB (offset index) or 2.7MB (time index) log messages.

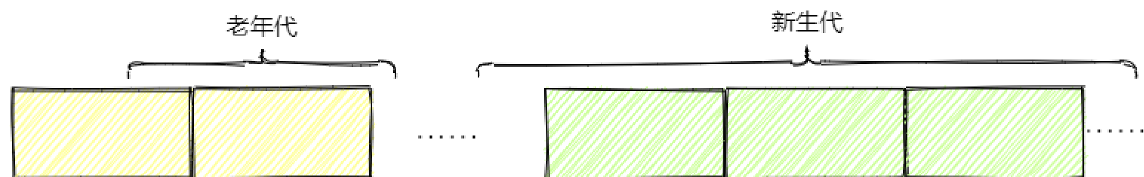
> 8KB的索引可以覆盖 4MB (offset index) or 2.7MB (time index)的消息数据，足够让大部分在in-sync内的节点在热区查询

以上就解释了什么是`_warmEntries`，并且为什么需要`_warmEntries`。

可以看到**朴素的算法在真正工程上的应用还是需要看具体的业务场景的，不可生搬硬套。并且彻底的理解算法也是很重要的，例如死记硬背二分，怕是看不出来以上的问题。还有底层知识的重要性。不然也是看不出来对缓存不友好的。**

从Kafka的索引冷热分区到MySQL InnoDB的缓冲池管理

从上面这波冷热分区我又想到了MySQL的buffer pool管理。MySQL的将缓冲池分为了新生代和老年代。默认是37分，即老年代占3，新生代占7。即看作一个链表的尾部30%为老年代，前面的70%为新生代。**替换了标准的LRU淘汰机制。**



MySQL的缓冲池分区是为了解决**预读失效**和**缓存污染**问题。

1、预读失效：因为会预读页，假设预读的页不会用到，那么就白白预读了，因此让预读的页插入的是老年代头部，淘汰也是从老年代尾部淘汰。不会影响新生代数据。

2、缓存污染：在类似like全表扫描的时候，会读取很多冷数据。并且有些查询频率其实很少，因此让这些数据仅仅存在老年代，然后快速淘汰才是正确的选择，MySQL为了解决这种问题，仅仅分代是不够的，还设置了一个时间窗口，默认是1s，即在老年代被再次访问并且存在超过1s，才会晋升到新生代，这样就不会污染新生代的热数据。

小结

文章先从索引入手，这就是时间和空间的互换。然后引出Kafka中索引存储使用了相对位移值，节省了空间，并且讲述了索引项的访问是由二分查找实现的，并结合Kafka的使用场景解释了Kafka中使用的冷热分区实现改进版的二分查找，并顺带提到了下一致性Hash，再由冷热分区联想到了MySQL缓冲池变形的LRU管理。

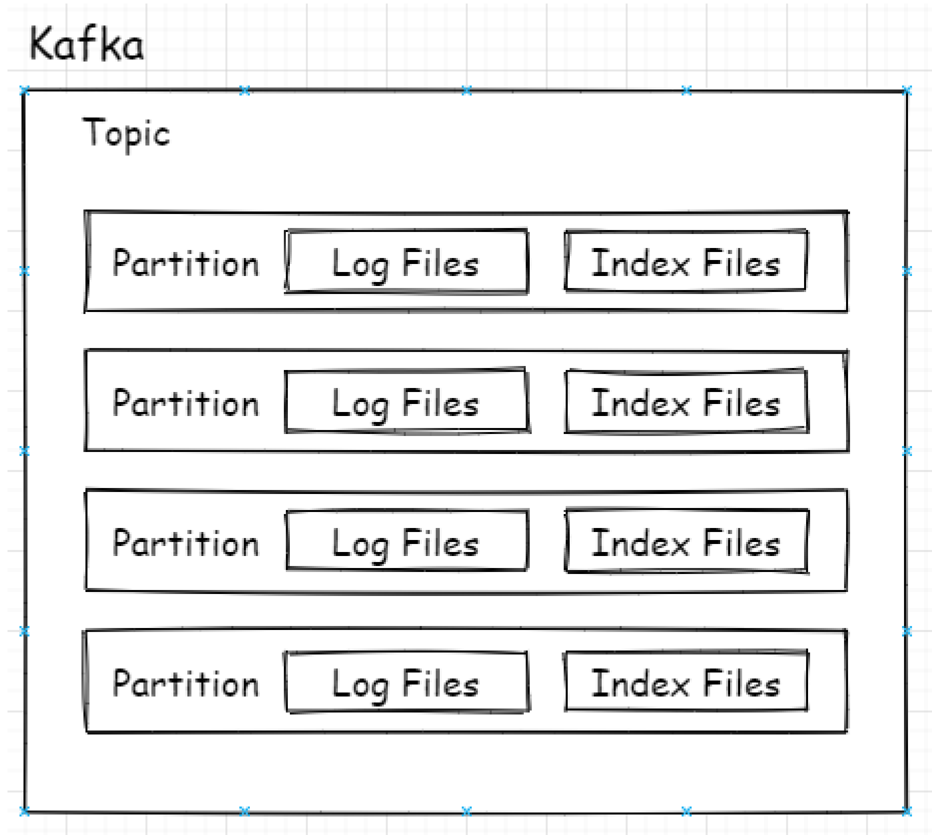
这一步步实际上都体现算法在工程中的灵活运用和变形实现。有些同学认为算法没用，刷算法题只是为了面试，实际上各种中间件和一些底层实现都体现了算法的重要性。

不说了，头有点冷。

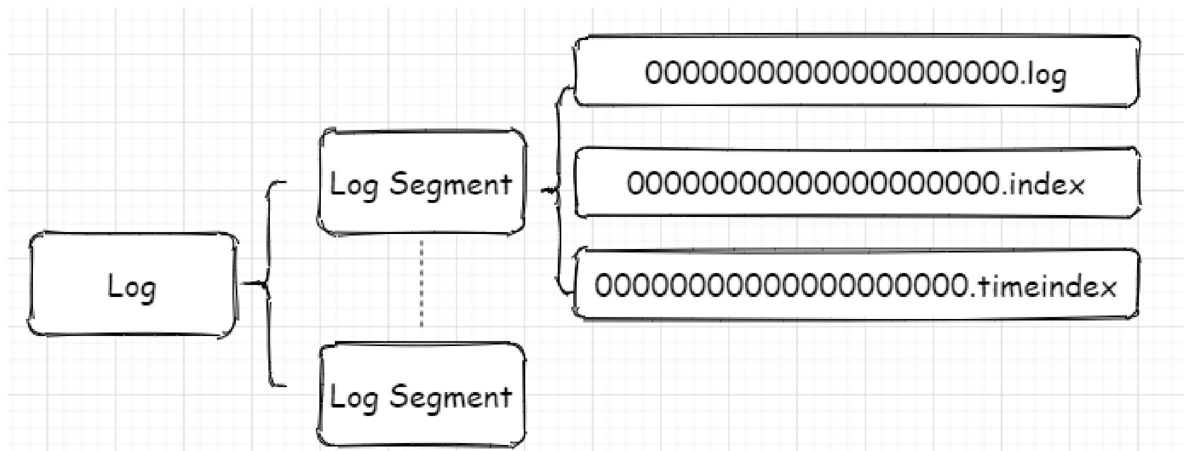
Kafka日志段如何读写解析?

Kafka的存储结构

众所周知，Kafka的Topic可以有多个分区，分区其实就是最小的读取和存储结构，即Consumer看似订阅的是Topic，实则是从Topic下的某个分区获得消息，Producer也是发送消息也是如此。



上图是总体逻辑上的关系，映射到实际代码中在磁盘上的关系则是如下图所示：



每个分区对应一个Log对象，在磁盘中就是一个子目录，子目录下面会有多组日志段即多Log Segment，每组日志段包含：消息日志文件(以log结尾)、位移索引文件(以index结尾)、时间戳索引文件(以timeindex结尾)。其实还有其它后缀的文件，例如.txnindex、.deleted等等。篇幅有限，暂不提起。

以下为日志的定义

```
@threadsafe
class Log(@volatile private var _dir: File,  //分区所在目录
          @volatile var config: LogConfig,   //配置
          @volatile var logStartOffset: Long,//暴露给客户端的起始offset
          @volatile var recoveryPoint: Long, //恢复点
          scheduler: Scheduler,              //用于后台操作的线程池调度
          brokerTopicStats: BrokerTopicStats,
          val time: Time,
          val maxProducerIdExpirationMs: Int,
          val producerIdExpirationCheckIntervalMs: Int,
          val topicPartition: TopicPartition,
          val producerStateManager: ProducerStateManager,
          logDirFailureChannel: LogDirFailureChannel) extends Logging with KafkaMetricsGroup {
    ...
     /* the actual segments of the log ,日志段们，*/
private val segments: ConcurrentNavigableMap[java.lang.Long, LogSegment] = new
ConcurrentSkipListMap[java.lang.Long, LogSegment]
    ...
                                                                                        }
```

以下为日志段的定义

```
@nonthreadsafe
class LogSegment private[log] (val log: FileRecords, //实际保存消息的对象
                               val lazyOffsetIndex: LazyIndex[OffsetIndex], //位移索引
                               val lazyTimeIndex: LazyIndex[TimeIndex], //时间戳索引
                               val txnIndex: TransactionIndex, //已中止事务索引
                               val baseOffset: Long, // 起始位移
                               val indexIntervalBytes: Int, //多少字节插一个索引
                               val rollJitterMs: Long, // 日志段新增扰动值
                               val time: Time) extends Logging {
```

`indexIntervalBytes`可以理解为插了多少消息之后再建一个索引，由此可以看出**Kafka的索引其实是稀疏索引**，这样可以避免索引文件占用过多的内存，从而可以**在内存中保存更多的索引**。对应的就是 Broker 端参数`log.index.interval.bytes` 值，默认4KB。

实际的通过索引查找消息过程是先通过offset找到索引所在的文件，然后通过二分法找到离目标最近的索引，再顺序遍历消息文件找到目标文件。这波操作时间复杂度为`O(log2n)+O(m)`,n是索引文件里索引的个数，m为稀疏程度。

这就是空间和时间的互换，又经过数据结构与算法的平衡，妙啊！

再说下`rollJitterMs`,这其实是个扰动值，对应的参数是`log.roll.jitter.ms`,这其实就要说到日志段的切分了，`log.segment.bytes`,这个参数控制着日志段文件的大小，默认是1G，即当文件存储超过1G之后就新起一个文件写入。这是以大小为维度的，还有一个参数是`log.segment.ms`,以时间为维度切分。

那配置了这个参数之后如果有很多很多分区，然后因为这个参数是全局的，因此同一时刻需要做很多文件的切分，这磁盘IO就顶不住了啊，因此需要设置个`rollJitterMs`，来岔开它们。

怎么样有没有联想到redis缓存的过期时间？过期时间加个随机数，防止同一时刻大量缓存过期导致缓存击穿数据库。 看看知识都是通的啊！

日志段的写入

```
  @nonthreadsafe
  def append(largestOffset: Long, //这批消息里面最大的位移值
             largestTimestamp: Long, //这批消息里面最大的时间戳
             shallowOffsetOfMaxTimestamp: Long, //最大时间戳对应的位移值
             records: MemoryRecords): Unit = { //消息们
    if (records.sizeInBytes > 0) {
      trace(s"Inserting ${records.sizeInBytes} bytes at end offset $largestOffset at position
${log.sizeInBytes} " +
            s"with largest timestamp $largestTimestamp at shallow offset $shallowOffsetOfMaxTimestamp")
      val physicalPosition = log.sizeInBytes() //获取当前日志的位移
      if (physicalPosition == 0)               //说明当前日志为空，则记录时间戳作为切分的依据
        rollingBasedTimestamp = Some(largestTimestamp)

      ensureOffsetInRange(largestOffset)       //确保位移值合法

      // append the messages
      val appendedBytes = log.append(records)
      trace(s"Appended $appendedBytes to ${log.file} at end offset $largestOffset")
      // Update the in memory max timestamp and corresponding offset.
      if (largestTimestamp > maxTimestampSoFar) {  //更新最大时间戳和其对应的位移值
        maxTimestampSoFar = largestTimestamp
        offsetOfMaxTimestampSoFar = shallowOffsetOfMaxTimestamp
      }
      // append an entry to the index (if needed)
      if (bytesSinceLastIndexEntry > indexIntervalBytes) {
        offsetIndex.append(largestOffset, physicalPosition)
        timeIndex.maybeAppend(maxTimestampSoFar, offsetOfMaxTimestampSoFar)
        bytesSinceLastIndexEntry = 0
      }
      bytesSinceLastIndexEntry += records.sizeInBytes
    }
  }
```

1、判断下当前日志段是否为空，空的话记录下时间，来作为之后日志段的切分依据

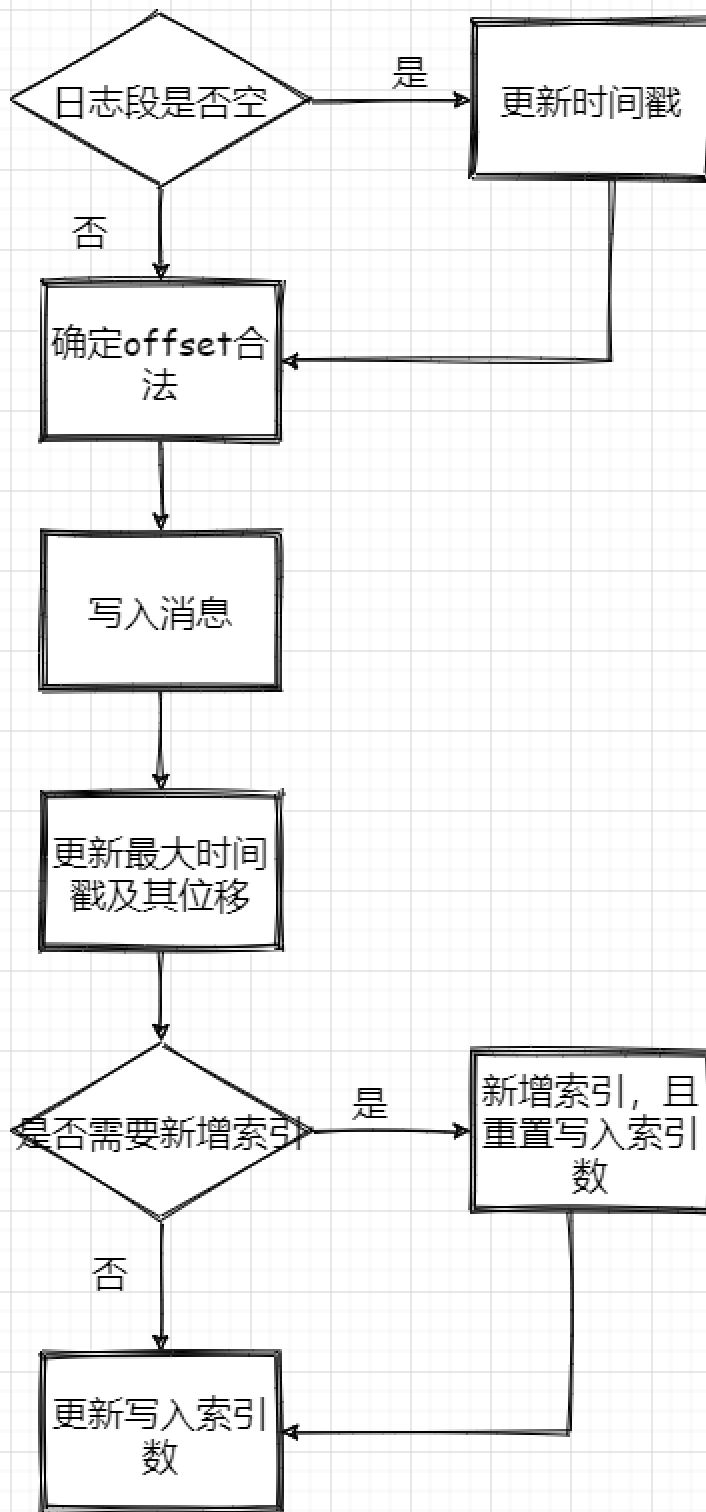
2、确保位移值合法，最终调用的是`AbstractIndex.toRelative(..)`方法，即使判断offset是否小于0，是否大于int最大值。

3、append消息，实际上就是通过`FileChannel`将消息写入，当然只是写入内存中及页缓存，是否刷盘看配置。

4、更新日志段最大时间戳和最大时间戳对应的位移值。这个时间戳其实用来作为定期删除日志的依据

5、更新索引项，如果需要的话`(bytesSinceLastIndexEntry > indexIntervalBytes)`

最后再来个流程图



日志段的读取

```
@threadsafe
def read(startOffset: Long,  //读取的第一条消息的位移
         maxSize: Int,       //能读取最大字节数
         maxPosition: Long = size,  //最大能读到的文件位置
         minOneMessage: Boolean = false): FetchDataInfo = { //是否至少返回一条
  if (maxSize < 0)
    throw new IllegalArgumentException(s"Invalid max size $maxSize for log read from segment $log")

  val startOffsetAndSize = translateOffset(startOffset)  //根据位移找到消息物理位置和大小

  // if the start position is already off the end of the log, return null
  if (startOffsetAndSize == null)
    return null

  val startPosition = startOffsetAndSize.position
  val offsetMetadata = LogOffsetMetadata(startOffset, this.baseOffset, startPosition)

  val adjustedMaxSize = //如果minOneMessage为true则调整下maxSize为一条消息的size
    if (minOneMessage) math.max(maxSize, startOffsetAndSize.size)
    else maxSize

  // return a log segment but with zero size in the case below
  if (adjustedMaxSize == 0)
    return FetchDataInfo(offsetMetadata, MemoryRecords.EMPTY)

  // calculate the length of the message set to read based on whether or not they gave us a maxOffset
再算下最小能获取的size
  val fetchSize: Int = min((maxPosition - startPosition).toInt, adjustedMaxSize)

    //从指定位置获取指定大小的集合
  FetchDataInfo(offsetMetadata, log.slice(startPosition, fetchSize),
    firstEntryIncomplete = adjustedMaxSize < startOffsetAndSize.size)
}
```

1、根据第一条消息的offset，通过`OffsetIndex`找到对应的消息所在的物理位置和大小。

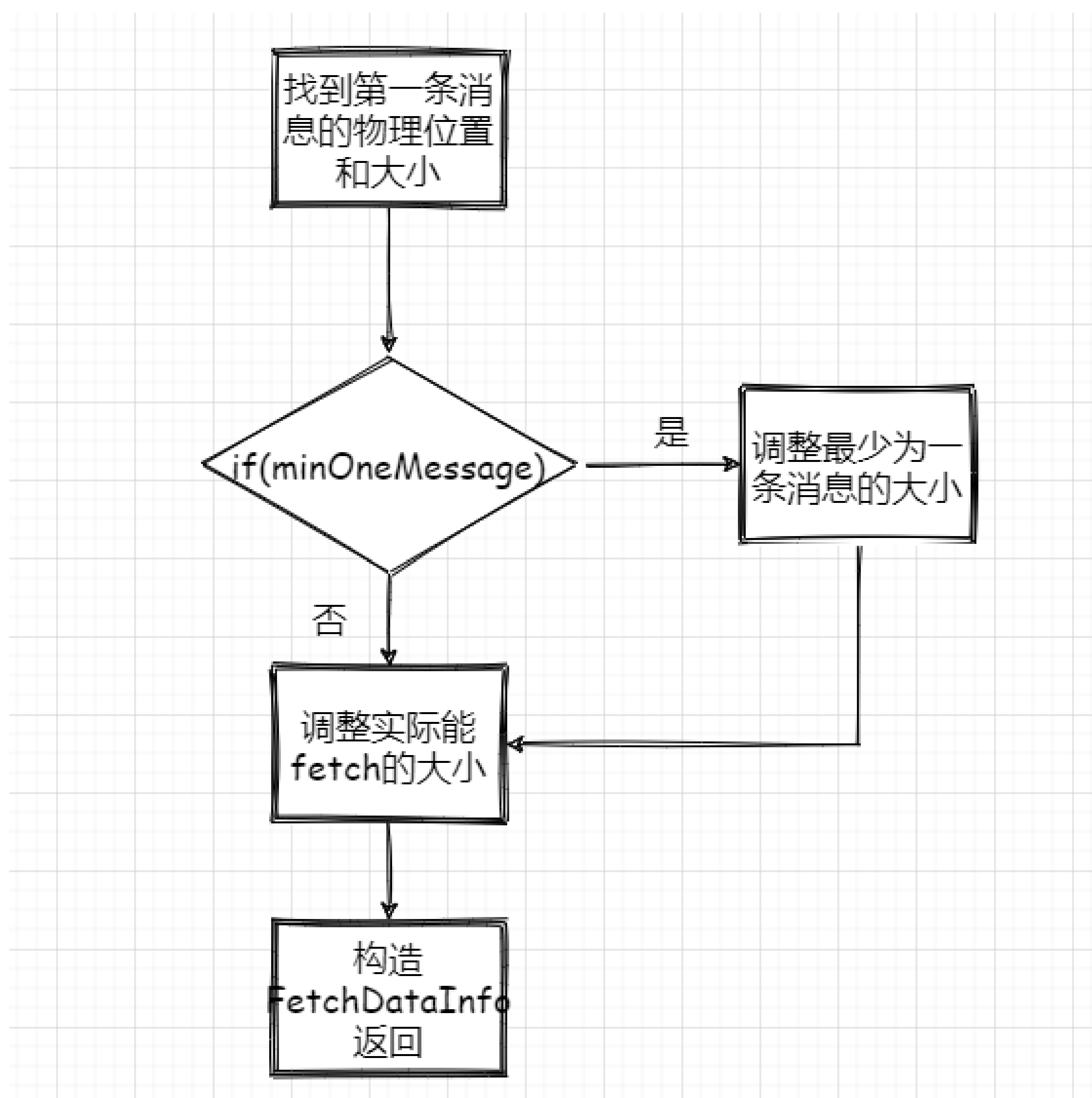
2、获取`LogOffsetMetadata`,元数据包含消息的offset、消息所在segment的起始offset和物理位置

3、判断`minOneMessage`是否为`true`,若是则调整为必定返回一条消息大小，其实就是在单条消息大于`maxSize`的情况下得以返回，防止消费者饿死

4、再计算最大的`fetchSize`,即（最大物理位移-此消息起始物理位移）和`adjustedMaxSize`的最小值(这波我不是很懂，因为以上一波操作`adjustedMaxSize`已经最小为一条消息的大小了)

5、调用 `FileRecords` 的 `slice` 方法从指定位置读取指定大小的消息集合，并且构造`FetchDataInfo`返回

再来个流程图：



情景剧

老白正目不转睛盯着监控大屏，“为什么？为什么Kafka Broker物理磁盘 I/O 负载突然这么高？”。寥寥无几的秀发矗立在老白的头上，显得如此的无助。

“是不是设置了 `log.segment.ms` 参数？ 试试 `log.roll.jitter.ms` 吧”，老白抬头间我已走出了办公室，留下了一个伟岸的背影和一颗锃亮的光头！

“我变秃了，也变强了”

Kafka控制器事件处理全流程解析

前言

这篇文章我分为两部分，第一部分就是直接图文来说清整个 `Kafka` 控制器事件处理全流程，然后再通过 `Controller选举流程` 进行一波源码分析，再来走一遍处理全流程。

正文

在深入源码之前我们得先搞明白 `Controller` 是什么？它有什么用？这样在看源码的时候才能**有的放矢。**

`Controller` 是**核心组件**，它的作用是**管理和协调**整个 `Kafka集群`。

具体管理和协调什么呢?

- 主题的管理，创建和删除主题;
- 分区管理，增加或重分配分区;
- 分区`Leader`选举;
- 监听`Broker`相关变化，即`Broker`新增、关闭等;
- 元数据管理，向其他`Broker`提供元数据服务;

为什么需要`Controller`?

我个人理解：凡是管理或者协调某样东西，都需要有个`Leader`，由他来把控全局，管理内部，对接外部，咱们就跟着`Leader`干就完事了。这其实对外也是好的，外部不需要和我们整体沟通，他只要和一个决策者交流，效率更高。

再来看看朱大是怎么说的，以下内容来自《深入理解Kafka：核心设计与实践原理》。

> 在Kafka的早期版本中，并没有采用 Kafka Controller 这样一概念来对分区和副本的状态进行管理，而是依赖于 ZooKeeper，每个 broker都会在 ZooKeeper 上为分区和副本注册大量的监听器（Watcher)。
> 当分区或副本状态变化时，会唤醒很多不必要的监听器，这种严重依赖 ZooKeeper 的设计会有脑裂、羊群效应，以及造成 ZooKeeper 过载的隐患。在目前的新版本的设计中，只有 Kafka Controller 在 ZooKeeper 上注册相应的监听器，其他的 broker 极少需要再监听 ZooKeeper 中的数据变化，这样省去了很多不必要的麻烦。

简单说下ZooKeeper

了解了`Controller`的作用之后我们还需要在简单的了解下`ZooKeeper`，因为`Controller`是极度依赖`ZooKeeper`的。（不过社区准备移除`ZooKeeper`，文末再提一下）

`ZooKeeper`是一个开源的分布式协调服务框架，最常用来作为注册中心等。`ZooKeeper`的数据模型就像文件系统一样，以根目录 "/" 开始，结构上的每个节点称为`znode`，可以存储一些信息。节点分为持久节点和临时节点，临时节点会随着会话结束而自动被删除。

并且有`Watcher`功能，节点自身数据变更、节点新增、节点删除、子节点数量变更都可以通过变更监听器通知客户端。

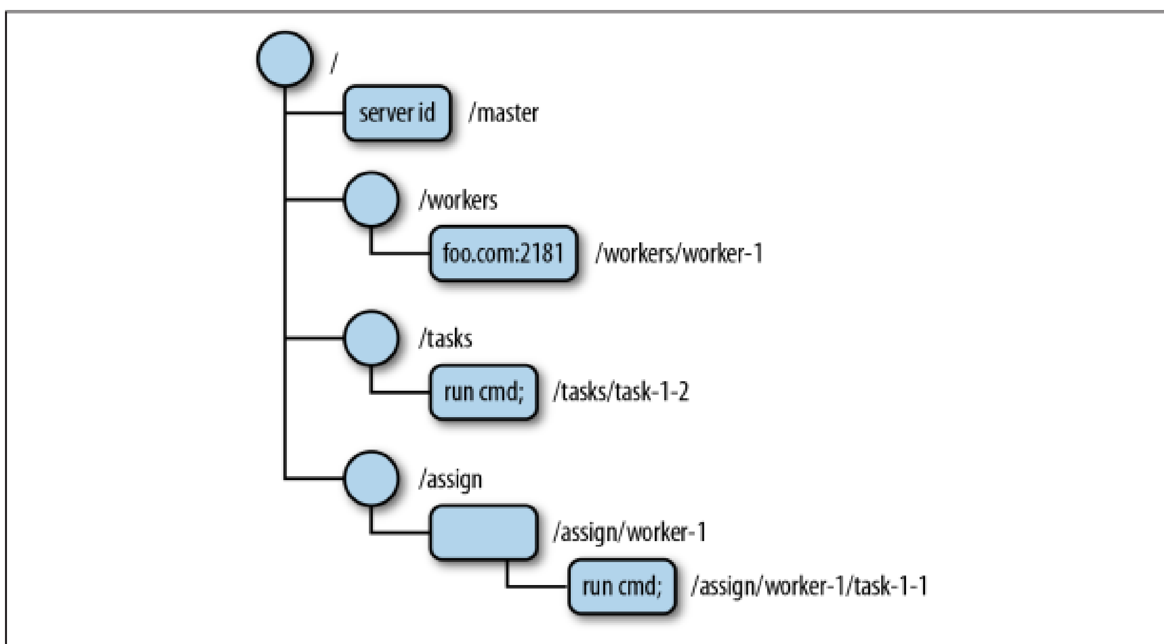


Figure 2-1. ZooKeeper data tree example

Controller是如何依赖ZooKeeper的

每个`Broker`在启动时会尝试向`ZooKeeper`注册`/controller`节点来竞选控制器，第一个创建`/controller`节点的`Broker`会被指定为控制器。**这就是是控制器的选举**。

`/controller`节点是个临时节点，其他`Broker`会监听着此节点，当`/controller`节点所在的`Broker`宕机之后，会话就结束了，此节点就被移除。其他`Broker`伺机而动，都来争当控制器，还是第一个创建`/controller`节点的`Broker`被指定为控制器。**这就是控制器故障转移，即`Failover`**。

当然还包括各种节点的监听，例如主题的增减等，都通过`watcher`功能，来实现相关的监听，进行对应的处理。

`Controller`在初始化的时候会从`ZooKeeper`拉取集群元数据信息，保存在自己的缓存中，然后通过向集群其他`Broker`发送请求的方式将数据同步给对方。

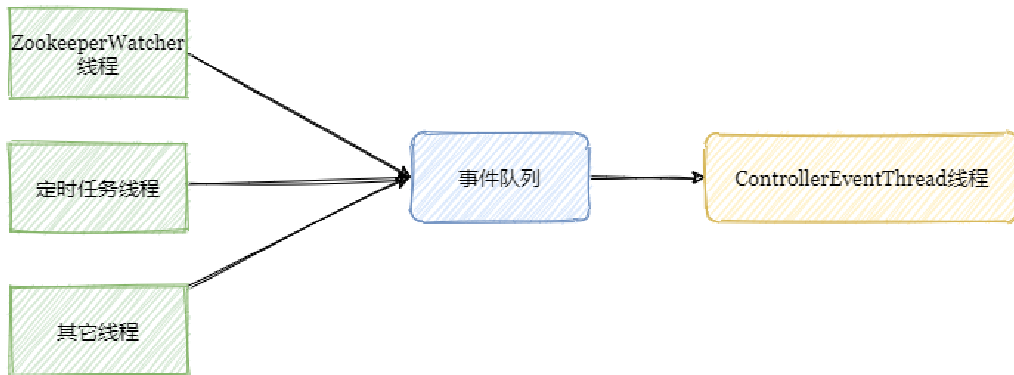
Controller 底层事件模型

不管是监听`Watcher`的`ZooKeeperWatcher线程`，还是定时任务线程亦或是其他线程都需要访问或更新`Controller`从集群拉取的元数据。`多线程 + 数据竞争 = 线程不安全`。因此需要加锁来保证线程安全。

一开始`Kafka`就是用大量的锁来保证线程间的同步，各种加锁使得性能下降，并且多线程加锁的方式使得代码复杂度急剧上升，一不小心就会出各种问题，bug难修复。

因此在0.11版本之后将多线程并发访问改成了**单线程事件队列模式。将涉及到共享数据竞争相关方面的访问抽象成事件，将事件塞入阻塞队列中，然后单线程处理**。

也就是说其它线程还是在的，只是把涉及共享数据的操作封装成事件由专属线程处理。



先小结一下

到这我们已经清楚了`Controller`主要用来管理和协调集群，具体是通过`ZooKeeper`临时节点和`Watcher`机制来监控集群的变化(当然还有来自定时任务或其他线程的事件驱动)，更新集群的元数据，并且通知集群中的其他`Broker`进行相关的操作(这部分下文会讲)。

而由于集群元数据会有并发修改问题，因此将操作抽象成事件，由阻塞队列和单线程处理来替换之前的多线程处理，降低代码的复杂度，提升代码的可维护性和性能。

接下来我们再讲讲`Controller`通知集群中的其他`Broker`的相关操作。

Controller的请求发送

`Controller`从`ZooKeeper`那儿得到变更通知之后，需要告知集群中的`Broker`（包括它自身）做相应的处理。

`Controller`只会给集群的`Broker`发送三种请求：分别是 `LeaderAndIsrRequest`、`StopReplicaRequest`和 `UpdateMetadataRequest`

LeaderAndIsrRequest

告知`Broker`主题相关分区`Leader`和`ISR`副本都在哪些 `Broker`上。

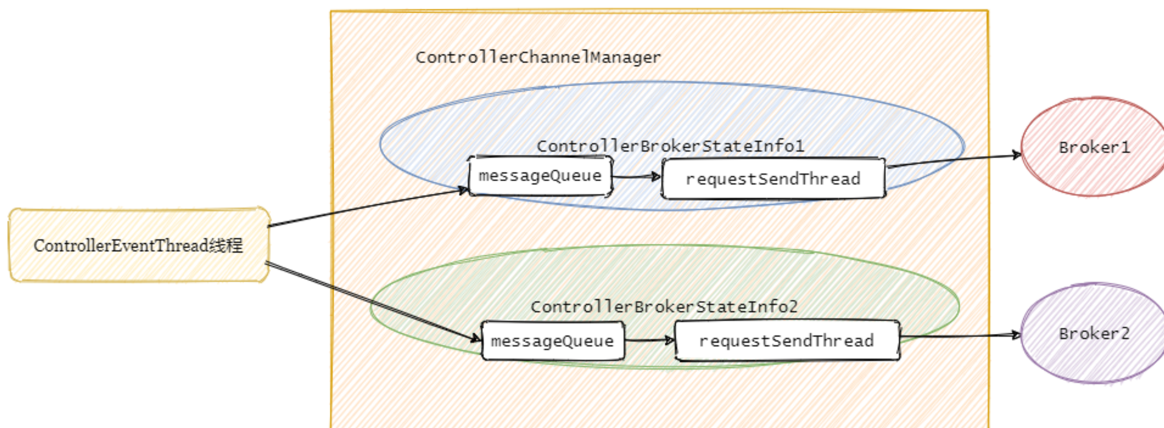
StopReplicaRequest

告知`Broker`停止相关副本操作，用于删除主题场景或分区副本迁移场景。

UpdateMetadataRequest

更新`Broker`上的元数据。

`Controller事件处理线程`会把事件封装成对应的请求，然后将请求写入对应的`Broker`的请求阻塞队列，然后`RequestSendThread`不断从阻塞队列中获取待发送的请求。



先解释下`controllerBrokerStateInfo`，它就是个 POJO类，可以理解为集群每个`broker`对应一个`controllerBrokerStateInfo`.

```
case class ControllerBrokerStateInfo(networkClient: NetworkClient, //封装的通信工具类
                                     brokerNode: Node, //封装了Broker 的连接信息如主机名端口
                                     messageQueue: BlockingQueue[QueueItem],//请求阻塞队列
                                     requestSendThread: RequestSendThread,//发送请求的线程
                                     queueSizeGauge: Gauge[Int],
                                     requestRateAndTimeMetrics: Timer,
                                     reconfigurableChannelBuilder: Option[Reconfigurable])
```

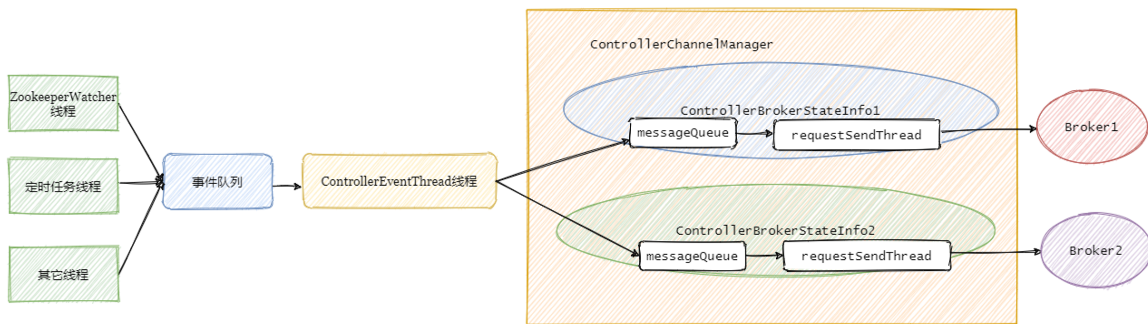
然后再看下`ControllerChannelManager`，从名字可以看出它管理`Controller`和集群`Broker`之间的连接，并为每个`Broker`创建一个`RequestSendThread`线程。

```
class ControllerChannelManager(controllerContext: ControllerContext, //元信息
                               config: KafkaConfig, // Broker配置
                               time: Time,
                               metrics: Metrics,
                               stateChangeLogger: StateChangeLogger,
                               threadNamePrefix: Option[String] = None) extends Logging with
KafkaMetricsGroup {
  //brokerId 和 brokerStateInfo 关联
  protected val brokerStateInfo = new HashMap[Int, ControllerBrokerStateInfo]
```

再小结一下

接着上个小结，事件处理线程将事件队列里面的事件处理之后再进行对应的请求封装，塞入需要通知的集群`Broker`对应的阻塞队列中，然后由每个`Broker`专属的`requestSendThread`发送请求至对应的`Broker`。

总的步骤如下图：



现在应该已经清楚`Controller`大概是如何运作的，整体看起来还是**生产者-消费者模型**。

接下来就进入源码环节。

Controller选举流程源码分析

事件处理的流程都是一样的，只是具体处理的事件逻辑不同，我们从`Controller选举`入手，来走一遍处理流程。

ControllerChangeHandler

选举会触发此`handler`，可以看到直接往`ControllerEventManager`的事件队列里塞。

```
class ControllerChangeHandler(eventManager: ControllerEventManager) extends ZNodeChangeHandler {
  override val path: String = ControllerZNode.path

  override def handleCreation(): Unit = eventManager.put(ControllerChange) //新增
  override def handleDeletion(): Unit = eventManager.put(Reelect) //删除，即重选
  override def handleDataChange(): Unit = eventManager.put(ControllerChange) //数据变化
}

//ControllerEventManager#put
  def put(event: ControllerEvent): QueuedEvent = inLock(putLock) {
    val queuedEvent = new QueuedEvent(event, time.milliseconds()) //封装一下
    queue.put(queuedEvent) //入队
    queuedEvent
  }
```

这个`QueueEvent`和`ControllerEventManager`，我们先来看看是啥。不过在此之前先了解下`ControllerEvent`和`ControllerEventProcessor`。

ControllerEvent：事件

```
sealed trait ControllerEvent {
  //状态，可以理解为处理事件的时候所处在的状态。例如 shutdownEvent，此时状态是controllerShutdown
  def state: ControllerState
}
```

ControllerEventProcessor： 事件处理接口

此接口的唯一实现类是 `KafkaController`。

```
trait ControllerEventProcessor {
  def process(event: ControllerEvent): Unit //接收一个事件并处理
  def preempt(event: ControllerEvent): Unit //接收一个事件，并优先处理
}
```

ControllerEventManager：事件处理器

此类主要用来管理事件处理线程和事件队列。

```
class ControllerEventManager(controllerId: Int, // broker id
                             processor: ControllerEventProcessor, //kafkacontroller
                             time: Time,
                             rateAndTimeMetrics: Map[ControllerState, KafkaTimer]) extends
KafkaMetricsGroup {
  import ControllerEventManager._

  @volatile private var _state: ControllerState = ControllerState.Idle /
  private val putLock = new ReentrantLock()
  private val queue = new LinkedBlockingQueue[QueuedEvent] //事件队列
  // Visible for test
  private[controller] val thread = new ControllerEventThread(ControllerEventThreadName) //事件处理线程
```

QueuedEvent：封装了ControllerEvent的类

主要是记录了下入队时间，并且提供了事件需要调用的方法。

```
class QueuedEvent(val event: ControllerEvent, //事件
                  val enqueueTimeMs: Long) { //入队时间
  val processingStarted = new CountDownLatch(1) //标识事件是否开始处理
  val spent = new AtomicBoolean(false) //标识事件是否被处理过

  def process(processor: ControllerEventProcessor): Unit = { //调用的是kafkacontroller的process方法
    if (spent.getAndSet(true))
      return
    processingStarted.countDown()
    processor.process(event)
  }

  def preempt(processor: ControllerEventProcessor): Unit = {//调用的是kafkacontroller的preempt方法
    if (spent.getAndSet(true))
      return
    processor.preempt(event)
  }

  def awaitProcessing(): Unit = {
    processingStarted.await()
  }

  override def toString: String = {
    s"QueuedEvent(event=$event, enqueueTimeMs=$enqueueTimeMs)"
  }
}
```

ControllerEventThread：事件处理线程

整体而言还是很简单的，从队列拿事件，然后处理。

```
class ControllerEventThread(name: String) extends ShutdownableThread(name = name, isInterruptible =
false) {

    override def doWork(): Unit = {
      val dequeued = queue.take() //从事件队列获取事件
      dequeued.event match {
        case ShutdownEventThread => //如果是关闭事件忽略
        case controllerEvent => //如果是controller事件
          _state = controllerEvent.state

          eventQueueTimeHist.update(time.milliseconds() - dequeued.enqueueTimeMs) //更新事件在队列中存在的时间

          try {
            def process(): Unit = dequeued.process(processor) //处理事件
            ....
          } catch {
            case e: Throwable => error(s"Uncaught error processing event $controllerEvent", e)
          }
          _state = ControllerState.Idle
      }
    }
  }
```

KafkaController#process

就是个switch，根据事件调用对应的`processXXXX`方法。

```
  override def process(event: ControllerEvent): Unit = {
    try {
      event match {
        case event: MockEvent =>
          // Used only in test cases
          event.process()
        case ShutdownEventThread =>
        case BrokerChange =>
          processBrokerChange()
        case BrokerModifications(brokerId) =>
          processBrokerModification(brokerId)
        case ControllerChange =>
          processControllerChange()
        case Reelect => //我们来看重选事件
          processReelect()
        ......
      }
    } catch {
     .....
  }
```

来关注下`controller`重选事件

```scala
private def processReelect(): Unit = {
  maybeResign() //如果之前是controller，需要做些卸任处理。主要就是一些数据初始化，移除zk监听等。
  elect() //
}

private def elect(): Unit = {
  activeControllerId = zkClient.getControllerId.getOrElse(-1) //获取当前controller 的brokerid
  if (activeControllerId != -1) { //不等于-1说明已经被人选了
    return
  }

  try {
    val (epoch, epochZkVersion) =
zkClient.registerControllerAndIncrementControllerEpoch(config.brokerId) // 创建 /controller节点。
    controllerContext.epoch = epoch
    controllerContext.epochZkVersion = epochZkVersion
    activeControllerId = config.brokerId
    ....
    onControllerFailover() //执行竞选成功后的动作，例如注册各种监听器，等等。
  } catch { //竞选失败
    case e: ControllerMovedException =>
      maybeResign()
      ....
    case t: Throwable =>
      triggerControllerMove()
  }
}
```

然后在`onControllerFailover`里面会调用`sendUpdateMetadataRequest`方法

```scala
private[controller] def sendUpdateMetadataRequest(brokers: Seq[Int], partitions: Set[TopicPartition]):
Unit = {
    try {
      brokerRequestBatch.newBatch()
      brokerRequestBatch.addUpdateMetadataRequestForBrokers(brokers, partitions)
      brokerRequestBatch.sendRequestsToBrokers(epoch) //发送请求
    } catch {
      case e: IllegalStateException =>
        handleIllegalState(e)
    }
  }
```

中间省略调用，内容太多了，不是重点，到后来调用`ControllerBrokerRequestBatch#sendRequest`

```scala
def sendRequest(brokerId: Int,
                request: AbstractControlRequest.Builder[_ <: AbstractControlRequest],
                callback: AbstractResponse => Unit = null): Unit = {
  controllerChannelManager.sendRequest(brokerId, request, callback)
}
```

最后还是调用了`controllerChannelManager#sendRequest`.

```scala
def sendRequest(brokerId: Int, request: AbstractControlRequest.Builder[_ <: AbstractControlRequest],
                callback: AbstractResponse => Unit = null): Unit = {
  brokerLock synchronized {
    val stateInfoOpt = brokerStateInfo.get(brokerId) //获取brokerId 对应的broker info
    stateInfoOpt match {
      case Some(stateInfo) => //将请求塞入请求阻塞队列
        stateInfo.messageQueue.put(QueueItem(request.apiKey, request, callback, time.milliseconds()))
      case None =>
        warn(s"Not sending request $request to broker $brokerId, since it is offline.")
    }
  }
}
```

然后`RequestSendThread#doWork`，不断从请求队列里拿请求，发送请求。

```
override def doWork(): Unit = {

  val QueueItem(apiKey, requestBuilder, callback, enqueueTimeMs) = queue.take() //从阻塞队列获取请求
  ...
  var clientResponse: ClientResponse = null
  try {
    var isSendSuccessful = false
    while (isRunning && !isSendSuccessful) {
      try {
        if (!brokerReady()) {
          ....
        }
        else {
          val clientRequest = networkClient.newClientRequest(brokerNode.idString, requestBuilder,
            time.milliseconds(), true) //封装成统一request
          clientResponse = NetworkClientUtils.sendAndReceive(networkClient, clientRequest, time) //发送
          isSendSuccessful = true
        }
      } catch {

      }
    }
    ........
}
```

一个环节完成了！我们来看下整体流程图

最后我们来看下元数据到底有啥和`KafkaController`的一些字段。

ControllerContext：元数据

主要有运行中的`Broker`、所有主题信息、主题分区副本信息等。

```
class ControllerContext {
  val stats = new ControllerStats //统计信息
  var offlinePartitionCount = 0 //离线分区数
  var preferredReplicaImbalanceCount = 0 //首选副本不平衡数
  val shuttingDownBrokerIds = mutable.Set.empty[Int] //关闭中的 BrokerID
  private val liveBrokers = mutable.Set.empty[Broker] //运行中的 Broker
  private val liveBrokerEpochs = mutable.Map.empty[Int, Long] //运行中的Broker 的 epoch
  var epoch: Int = KafkaController.InitialControllerEpoch //controller 的epoch
  var epochZkVersion: Int = KafkaController.InitialControllerEpochZkVersion//zk中controller节点 dataversion
  val allTopics = mutable.Set.empty[String] //所有主题
  val partitionAssignments = mutable.Map.empty[String, mutable.Map[Int, ReplicaAssignment]] //主题分区副本
   //主题分区LeaderIsr副本
  private val partitionLeadershipInfo = mutable.Map.empty[TopicPartition, LeaderIsrAndControllerEpoch]
  val partitionsBeingReassigned = mutable.Set.empty[TopicPartition] //正在重分配的主题分区
  val partitionStates = mutable.Map.empty[TopicPartition, PartitionState] //主题分区状态
  val replicaStates = mutable.Map.empty[PartitionAndReplica, ReplicaState] //主题分区副本状态
  val replicasOnOfflineDirs = mutable.Map.empty[Int, Set[TopicPartition]] //不在磁盘上的副本
  val topicsToBeDeleted = mutable.Set.empty[String] //待删除的主题
  val topicsWithDeletionStarted = mutable.Set.empty[String] //已开始删除的主题
  val topicsIneligibleForDeletion = mutable.Set.empty[String] //暂时无法删除的主题
```

KafkaController

基本上关键的字段都解释了，关于状态机那一块篇幅有限，之后再说。

```scala
class KafkaController(val config: KafkaConfig, //broker配置
                      zkClient: KafkaZkClient, //ZooKeeper客户端
                      time: Time,
                      metrics: Metrics, //监控相关
                      initialBrokerInfo: BrokerInfo, //broker信息
                      initialBrokerEpoch: Long, //broker epoch，拒绝已卸任的controller请求
                      tokenManager: DelegationTokenManager,
                      threadNamePrefix: Option[String] = None)
  extends ControllerEventProcessor with Logging with KafkaMetricsGroup {

  @volatile private var brokerInfo = initialBrokerInfo
  @volatile private var _brokerEpoch = initialBrokerEpoch

  //集群元数据
  val controllerContext = new ControllerContext
  // Controller通道管理器
  var controllerChannelManager = new ControllerChannelManager(controllerContext, config, time, metrics,
    stateChangeLogger, threadNamePrefix)
  //定时调度，定时leader重选
  private[controller] val kafkaScheduler = new KafkaScheduler(1)
  //事件管理器
  private[controller] val eventManager = new ControllerEventManager(config.brokerId, this, time,
    controllerContext.stats.rateAndTimeMetrics)
  //发送请求用的
  private val brokerRequestBatch = new ControllerBrokerRequestBatch(config, controllerChannelManager,
    eventManager, controllerContext, stateChangeLogger)
    //副本状态机
  val replicaStateMachine: ReplicaStateMachine = new ZkReplicaStateMachine(config, stateChangeLogger,
controllerContext, zkClient,
    new ControllerBrokerRequestBatch(config, controllerChannelManager, eventManager, controllerContext,
stateChangeLogger))
    //分区状态机
  val partitionStateMachine: PartitionStateMachine = new ZkPartitionStateMachine(config,
stateChangeLogger, controllerContext, zkClient,
    new ControllerBrokerRequestBatch(config, controllerChannelManager, eventManager, controllerContext,
stateChangeLogger))
    //主题删除管理器
  val topicDeletionManager = new TopicDeletionManager(config, controllerContext, replicaStateMachine,
    partitionStateMachine, new ControllerDeletionClient(this, zkClient))

  private val controllerChangeHandler = new ControllerChangeHandler(eventManager)
  private val brokerChangeHandler = new BrokerChangeHandler(eventManager)
  //....各种ZooKeeper监听器
```

最后

整体的流程就是将`Controller`相关操作都封装成一个个事件，然后将事件入队，由一个事件处理线程来处理，保证数据的安全（从这也可以看出，不是多线程就是好，有利有弊最终还是看场景）。

最后在通知集群中`Broker`的过程是每个`Broker`配备一个发送线程，因为发送是同步的，因此每个`Broker`线程隔离可以防止某个`Broker`阻塞而导致整体都阻塞的情况。

前面有说到`Kafka Controller` 强依赖 `ZooKeeper`。但是现在社区打算移除 `ZooKeeper`，因为`ZooKeeper`不适合频繁写，并且是CP的。而且用`Kafka`还需要维护`ZooKeeper`集群，提升了系统的复杂度和运维难度，降低了系统的稳定性。

像位移信息，已经通过内部主题的方式保存，绕开了`ZooKeeper`。

社区打算通过类 Raft 共识算法来选举`Controller`，并且把元数据存储在 Log 中的方式来做。

Kafka请求处理全流程解析

今天来讲讲 `Kafka Broker`端处理请求的全流程，剖析下底层的网络通信是如何实现的、Reactor在kafka上的应用。

再说说社区为何在2.3版本将请求类型划分成两大类，又是如何实现两类请求处理的优先级。

在源码分析之前我先总结性的说了说`Kafka`底层的通信模型。应对面试官询问`Kafka`请求全过程已经够了。

Reactor模式

在扯到`Kafka`之前我们先来说说`Reactor模式`，基本上只要是底层的高性能网络通信就离不开`Reactor模式`。像Netty、Redis都是使用`Reactor模式`。

像我们以前刚学网络编程的时候以下代码可是非常的熟悉，新来一个请求，要么在当前线程直接处理了，要么新起一个线程处理。

```
while (true) {
     socket = accept();
     handle(socket);
}

while (true) {
     socket = accept();
     new Thread(socket);
}
```

在早期这样的编程是没问题的，但是随着互联网的快速发展，单线程处理不过来，也不能充分的利用计算机资源。

而每个请求都新起一个线程去处理，资源的要求就太高了，并且创建线程也是一个重操作。

说到这有人想到了，那搞个线程池不就完事了嘛，还要啥`Reactor`。

```
while (true) {
   socket = accept();
   threadPool.execute(new Task(socket));
}
```

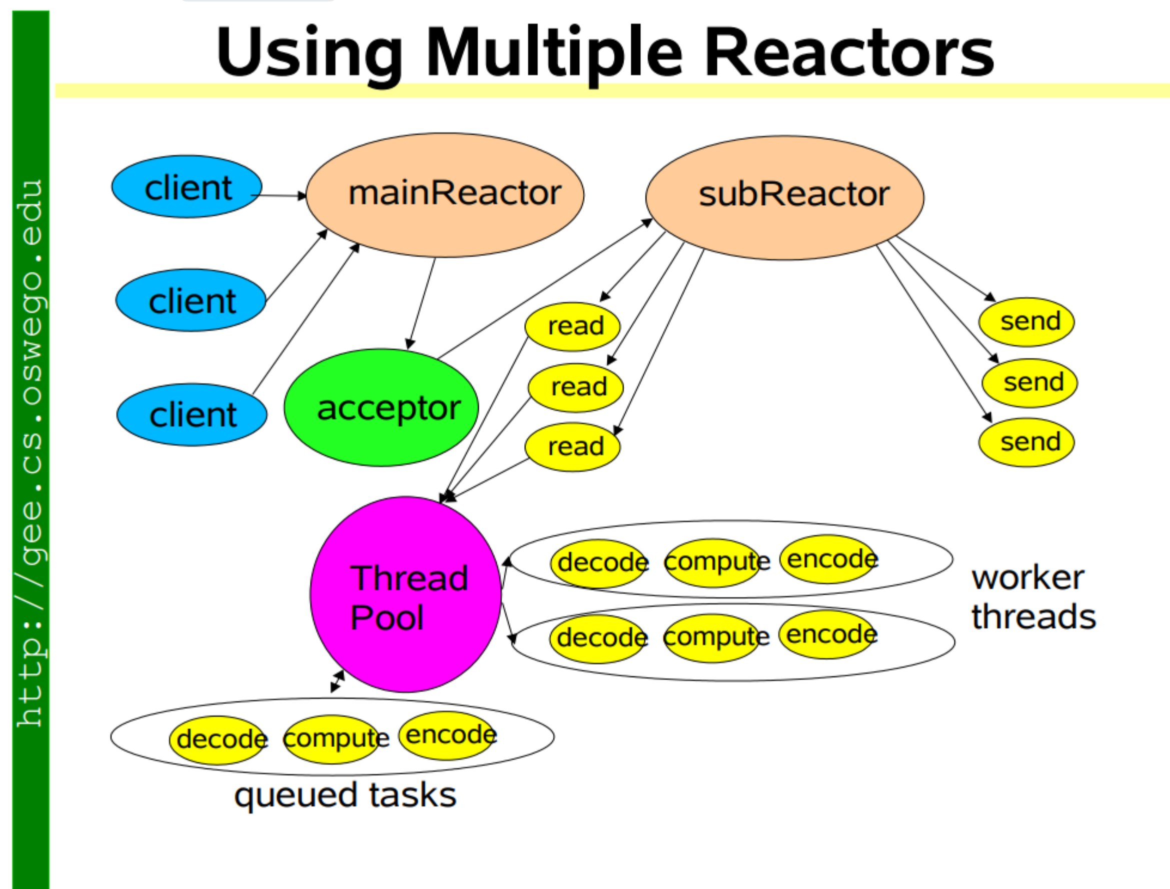
池化技术确实能缓解资源的问题，但是池子是有限的，池子里的一个线程不还是得候着某个连接，等待指示嘛。现在的互联网时代早已突破`C10K`了。

因此引入的`IO多路复用`，由**一个线程来监视一堆连接**，同步等待一个或多个IO事件的到来，然后将事件分发给对应的`Handler`处理，这就叫`Reactor模式`。

网络通信模型的发展如下

> 单线程 => 多线程 => 线程池 => Reactor模型

Kafka所采用的`Reactor模型`如下



Kafka Broker 网络通信模型

简单来说就是，Broker 中有个`Acceptor(mainReactor)`监听新连接的到来，与新连接建连之后轮询选择一个`Processor(subReactor)`管理这个连接。

而`Processor`会监听其管理的连接，当事件到达之后，读取封装成`Request`，并将`Request`放入共享请求队列中。

然后IO线程池不断的从该队列中取出请求，执行真正的处理。处理完之后将响应发送到对应的`Processor`的响应队列中，然后由`Processor`将`Response`返还给客户端。

每个`listener`只有一个`Acceptor线程`，因为它只是作为新连接建连再分发，没有过多的逻辑，很轻量，一个足矣。

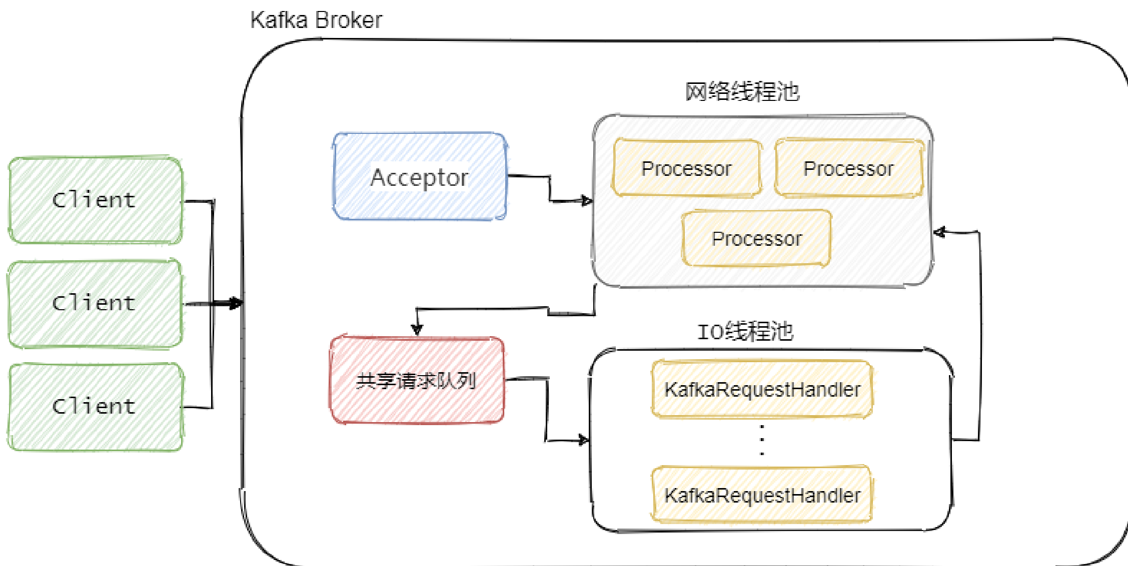
`Processor` 在Kafka中称之为网络线程，默认网络线程池有3个线程，对应的参数是`num.network.threads`。并且可以根据实际的业务动态增减。

还有个 IO 线程池，即`KafkaRequestHandlerPool`，执行真正的处理，对应的参数是`num.io.threads`，默认值是 8。IO线程处理完之后会将`Response`放入对应的`Processor`中，由`Processor`将响应返还给客户端。

可以看到网络线程和IO线程之间利用的经典的生产者 - 消费者模式，不论是用于处理Request的共享请求队列，还是IO处理完返回的Response。

这样的好处是什么？生产者和消费者之间解耦了，可以对生产者或者消费者做独立的变更和扩展。并且可以平衡两者的处理能力，例如消费不过来了，我多加些IO线程。

如果你看过其他中间件源码，你会发现生产者-消费者模式真的是太常见了，所以面试题经常会有手写一波生产者-消费者。



源码级别剖析网络通信模型

Kafka 网络通信组件主要由两大部分构成：**SocketServer 和 KafkaRequestHandlerPool。**

##SocketServer

```
class SocketServer(val config: KafkaConfig,
                   val metrics: Metrics,
                   val time: Time,
                   val credentialProvider: CredentialProvider)
  extends Logging with KafkaMetricsGroup with BrokerReconfigurable {

  //共享请求队列长度，由Broker端参数queued.max.requests值而定，默认值是500
  private val maxQueuedRequests = config.queuedMaxRequests

  // data-plane
  //处理数据类型请求的Processors 线程池
  private val dataPlaneProcessors = new ConcurrentHashMap[Int, Processor]()
  //处理数据类请求的Acceptor线程池，每套监听器对应一个Acceptor线程
   private[network] val dataPlaneAcceptors = new ConcurrentHashMap[EndPoint, Acceptor]()
  //处理数据类请求RequestChannel对象
  val dataPlaneRequestChannel = new RequestChannel(maxQueuedRequests, DataPlaneMetricPrefix, time)

  // control-plane
  //处理控制类型请求的Processors，就一个线程
  private var controlPlaneProcessorOpt : Option[Processor] = None
  //处理控制类型请求的Acceptor，就一个线程
  private[network] var controlPlaneAcceptorOpt : Option[Acceptor] = None
  //处理控制类型请求的RequestChannel对象
  val controlPlaneRequestChannelOpt: Option[RequestChannel] = config.controlPlaneListenerName.map(_ =>
    new RequestChannel(20, ControlPlaneMetricPrefix, time))
```

可以看出 `SocketServer` 旗下管理着，`Acceptor 线程`、`Processor 线程` 和 `RequestChannel` 等对象。

`data-plane` 和 `control-plane` 稍后再做分析，先看看 `RequestChannel` 是什么。

RequestChannel

```scala
class RequestChannel(val queueSize: Int, val metricNamePrefix : String, time: Time) extends
KafkaMetricsGroup {
  ...
  //共享请求阻塞队列，线程安全
  private val requestQueue = new ArrayBlockingQueue[BaseRequest](queueSize) //为SocketServer传入，默认500
  // Processor 线程池
  private val processors = new ConcurrentHashMap[Int, Processor]()
  .....

  def addProcessor(processor: Processor): Unit = { //添加Processor线程
    if (processors.putIfAbsent(processor.id, processor) != null)
      warn(s"Unexpected processor with processorId ${processor.id}")

    newGauge(responseQueueSizeMetricName, () => processor.responseQueueSize,
      Map(ProcessorMetricTag -> processor.id.toString))
  }

  def removeProcessor(processorId: Int): Unit = { //删除Processor线程
    processors.remove(processorId)
    removeMetric(responseQueueSizeMetricName, Map(ProcessorMetricTag -> processorId.toString))
  }

  def sendRequest(request: RequestChannel.Request): Unit = { //将请求塞入队列
    requestQueue.put(request)
  }

  def sendResponse(response: RequestChannel.Response): Unit = { //将响应塞入对应的Processor的响应队列中
    ...省略部分代码
    val processor = processors.get(response.processor)
    if (processor != null) {
      processor.enqueueResponse(response)
    }
  }

  def receiveRequest(timeout: Long): RequestChannel.BaseRequest = //待超时的阻塞获取请求
    requestQueue.poll(timeout, TimeUnit.MILLISECONDS)


  def receiveRequest(): RequestChannel.BaseRequest = //阻塞从队列获取请求
    requestQueue.take()
```

关键的属性和方法都已经在下面代码中注释了，可以看出这个对象主要就是**管理** `Processor` 和**作为传输** `Request` **和** `Response` **的中转站**。

##Acceptor
接下来我们再看看 `Acceptor`

```scala
private[kafka] class Acceptor(val endPoint: EndPoint,//定义的Broker连接信息，主机名端口等
                              //出站网络IO底层缓存区大小，默认100KB，对应socket.send.buffer.bytes
                              val sendBufferSize: Int,
                              //入站网络IO底层缓存区大小，默认100KB，对应socket.receive.buffer.bytes
                              val recvBufferSize: Int,
                              brokerId: Int,
                              connectionQuotas: ConnectionQuotas,
                              metricPrefix: String) extends AbstractServerThread(connectionQuotas) with
KafkaMetricsGroup {
  //就是JAVA 的Selector 只是取了个别名 import java.nio.channels.{Selector => NSelector}
  private val nioSelector = NSelector.open()
  //创建ServerSocketChannel
  val serverChannel = openServerSocket(endPoint.host, endPoint.port)
  //创建processors数组,保存Processors用于分发新建立的连接
  private val processors = new ArrayBuffer[Processor]()
  private val processorsStarted = new AtomicBoolean
```

可以看到它继承了 `AbstractServerThread`，接下来再看看它run些啥

```scala
def run(): Unit = {
  serverChannel.register(nioSelector, SelectionKey.OP_ACCEPT) //就 ACCPET 事件感兴趣
  startupComplete() // 等待Acceptor启动完成
  try {
    var currentProcessorIndex = 0
    while (isRunning) {
      try {
        val ready = nioSelector.select(500) //每500毫秒获取一次就绪事件
        if (ready > 0) {
          val keys = nioSelector.selectedKeys()
          val iter = keys.iterator()
          while (iter.hasNext && isRunning) { //遍历获取到的事件
            try {
              val key = iter.next
              iter.remove()

              if (key.isAcceptable) {
                accept(key).foreach { socketChannel => //调用accept获取到socketChannel
                  var retriesLeft = synchronized(processors.length)
                  var processor: Processor = null
                  do {
                    retriesLeft -= 1
                    processor = synchronized { //通过 currentProcessorIndex 轮询得到一个processor
                      currentProcessorIndex = currentProcessorIndex % processors.length
                      processors(currentProcessorIndex)
                    }
                    currentProcessorIndex += 1
                            //将socketChannel 放入processor 的新连接队列中
                  } while (!assignNewConnection(socketChannel, processor, retriesLeft == 0))
                }
              } else
                throw new IllegalStateException("Unrecognized key state for acceptor thread.")
            } catch {
              case e: Throwable => error("Error while accepting connection", e)
            }
      // 省略以下代码
}
```

再来看看`accept(key)`做了啥

```scala
  private def accept(key: SelectionKey): Option[SocketChannel] = {
    val serverSocketChannel = key.channel().asInstanceOf[ServerSocketChannel]
    val socketChannel = serverSocketChannel.accept()
    try {
      connectionQuotas.inc(endPoint.listenerName, socketChannel.socket.getInetAddress,
blockedPercentMeter)
      socketChannel.configureBlocking(false) //设置为非阻塞
      socketChannel.socket().setTcpNoDelay(true) //禁用纳格算法
      socketChannel.socket().setKeepAlive(true) //设置tcp保活
      if (sendBufferSize != Selectable.USE_DEFAULT_BUFFER_SIZE)
        socketChannel.socket().setSendBufferSize(sendBufferSize)
      Some(socketChannel)
    } catch {
      case e: TooManyConnectionsException =>
        info(s"Rejected connection from ${e.ip}, address already has the configured maximum of ${e.count}
connections.")
        close(endPoint.listenerName, socketChannel)
        None
    }
  }
```

很简单，标准`selector`的处理，获取准备就绪事件，调用`serverSocketChannel.accept()`得到`socketChannel`，将`socketChannel`交给通过轮询选择出来的`Processor`，之后由它来处理IO事件。

##Processor

接下来我们再看看`Processor`，相对而言比`Acceptor`复杂一些。

先来看看三个关键的成员

```scala
  // Accceptor得到的socketChannel 存入的就是此阻塞队列
  private val newConnections = new ArrayBlockingQueue[SocketChannel](connectionQueueSize)
  // 存放已经发送给client的response，存着是为了待发送成功之后做一些response的回调
  private val inflightResponses = mutable.Map[String, RequestChannel.Response]()
  //response 队列，IO线程处理完之后response就塞到这里
  private val responseQueue = new LinkedBlockingDeque[RequestChannel.Response]()
```

再来看看主要的处理逻辑。

```
override def run(): Unit = {
  startupComplete()
  try {
    while (isRunning) {
      try {
        configureNewConnections() //将newConnections里面的channel取出来注册到selector中
        processNewResponses() //发送IO线程存入的Response，并将其加入到inflightResponses中
        poll() //执行NIO的poll，获取对应SocketChannel上就绪的I/O事件
        processCompletedReceives() //将接收到的Request放入Request共享阻塞队列中
        processCompletedSends() //为inflightResponsesResponse 成功发送的response执行回调逻辑
        processDisconnected() //将断开连接的response从inflightResponsesResponse中移除
        closeExcessConnections() //关闭超过配额限制的连接
      } catch {
        case e: Throwable => processException("Processor got uncaught exception.", e)
      }
    }
  } finally {
    debug(s"Closing selector - processor $id")
    CoreUtils.swallow(closeAll(), this, Level.ERROR)
    shutdownComplete()
  }
}
```

可以看到`Processor`主要是将底层读事件IO数据封装成`Request`存入队列中，然后将IO线程塞入的`Response`，返还给客户端，并处理`Response`的回调逻辑。

#KafkaRequestHandlerPool

IO线程池，实际处理请求的线程。

```
class KafkaRequestHandlerPool(val brokerId: Int,
                              val requestChannel: RequestChannel, //刚ScoketServer里说的RequestChannel对象
                              val apis: KafkaApis, //请求处理逻辑封装类，可以说是源码查找逻辑的入口
                              time: Time,
                              numThreads: Int, // IO线程数
                              requestHandlerAvgIdleMetricName: String,
                              logAndThreadNamePrefix : String) extends Logging with KafkaMetricsGroup {

  private val threadPoolSize: AtomicInteger = new AtomicInteger(numThreads) //为之后动态新增线程数做准备

  val runnables = new mutable.ArrayBuffer[KafkaRequestHandler](numThreads) //线程数组
  for (i <- 0 until numThreads) {
    createHandler(i)
  }

  def createHandler(id: Int): Unit = synchronized { //创建IO线程，即KafkaRequestHandler
    runnables += new KafkaRequestHandler(id, brokerId, aggregateIdleMeter, threadPoolSize, requestChannel,
apis, time)
    KafkaThread.daemon(logAndThreadNamePrefix + "-kafka-request-handler-" + id, runnables(id)).start()
  }
```

再来看看IO线程都干了些啥

```scala
class KafkaRequestHandler(id: Int,
                          brokerId: Int,
                          val aggregateIdleMeter: Meter,
                          val totalHandlerThreads: AtomicInteger,
                          val requestChannel: RequestChannel,
                          apis: KafkaApis,
                          time: Time) extends Runnable with Logging {
  @volatile private var stopped = false

  def run(): Unit = {
    while (!stopped) {
      ..省略
      val req = requestChannel.receiveRequest(300) //从RequestChannel 获取请求
      req match {
        case RequestChannel.ShutdownRequest => //如果是shutdown 类型请求
          debug(s"Kafka request handler $id on broker $brokerId received shut down command")
          shutdownComplete.countDown()
          return

        case request: RequestChannel.Request => //如果是正常请求
          try {
            request.requestDequeueTimeNanos = endTime
            trace(s"Kafka request handler $id on broker $brokerId handling request $request")
            apis.handle(request) //核心就是这里
          } catch {
             ...省略
          } finally {
            request.releaseBuffer()
          }

        case null => // continue
      }
    }
    shutdownComplete.countDown()
  }
```

很简单，核心就是不断的从`requestChannel`拿请求，然后调用handle处理请求。

`handle`方法是位于`KafkaApis`类中，可以理解为通过`switch`，根据请求头里面不同的`apikey`调用不同的`handle`来处理请求。

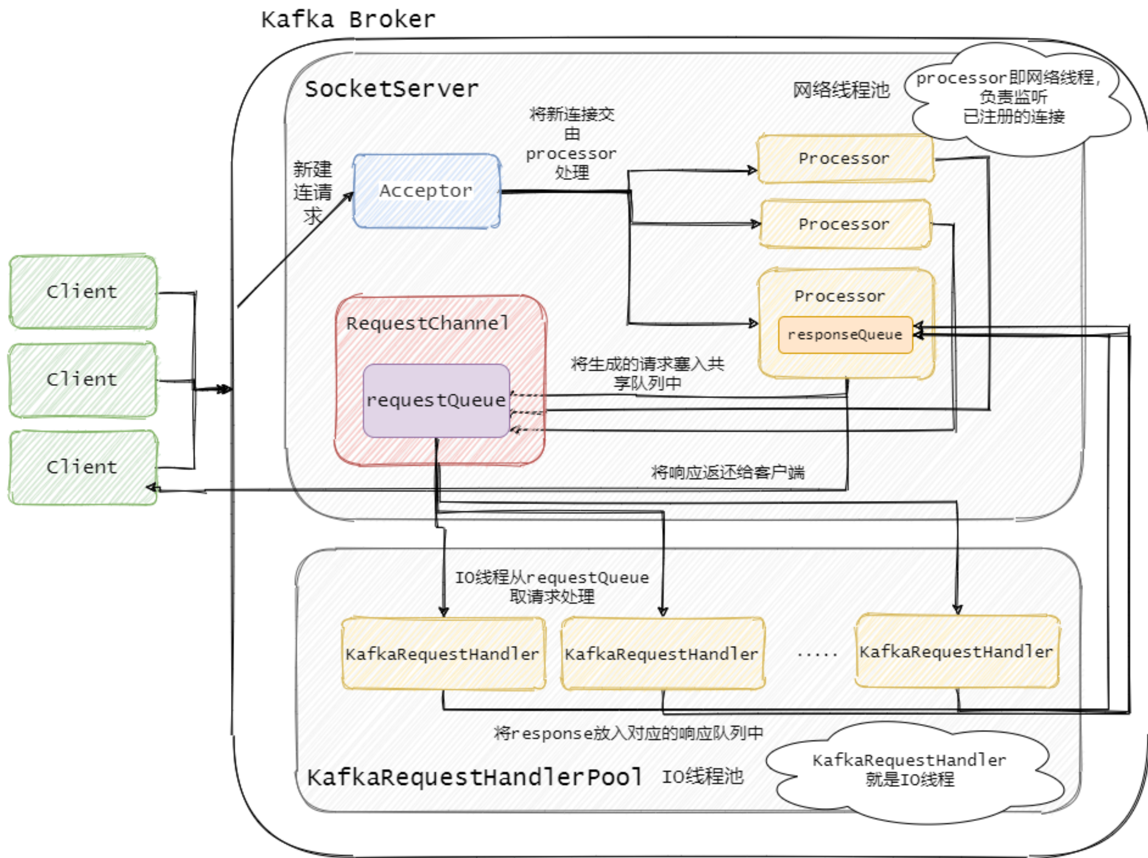
```scala
  def handle(request: RequestChannel.Request): Unit = {
   try {
     request.header.apiKey match {
       case ApiKeys.PRODUCE => handleProduceRequest(request)
       case ApiKeys.FETCH => handleFetchRequest(request)
       case ApiKeys.LIST_OFFSETS => handleListOffsetRequest(request)
       .....省略，有几十个
     }
   } catch {
     case e: FatalExitError => throw e
     case e: Throwable => handleError(request, e)
   } finally {
     // The local completion time may be set while processing the request. Only record it if it's unset.
     if (request.apiLocalCompleteTimeNanos < 0)
       request.apiLocalCompleteTimeNanos = time.nanoseconds
   }
 }
```

我们再举例看下较为简单的处理`LIST_OFFSETS`的过程，即`handleListOffsetRequest`，来完成一个请求的闭环。

我用红色箭头标示了调用链。表明处理完请求之后是塞给对应的 `Processor` 的。

```
  def handleListOffsetRequest(request: RequestChannel.Request): Unit = {
    //省略之前代码
    //可以看出构造了一个ListOffsetResponse 并且调用 sendResponseMaybeThrottle
    sendResponseMaybeThrottle(request, requestThrottleMs => new ListOffsetResponse(requestThrottleMs,
mergedResponseMap.asJava))
  }

  private def sendResponseMaybeThrottle(request: RequestChannel.Request,
                                        createResponse: Int => AbstractResponse,
                                        onComplete: Option[Send => Unit] = None): Unit = {
      //省略之前代码
      //调用了 sendResponse
    sendResponse(request, Some(createResponse(throttleTimeMs)), onComplete)
  }

  private def sendResponse(request: RequestChannel.Request,
                           responseOpt: Option[AbstractResponse],
                           onComplete: Option[Send => Unit]): Unit = {

    val response = responseOpt match {
      case Some(response) =>
        val responseSend = request.context.buildResponse(response)
        val responseString =
          if (RequestChannel.isRequestLoggingEnabled) Some(response.toString(request.context.apiVersion))
          else None
            //构造SendResponse
        new RequestChannel.SendResponse(request, responseSend, responseString, onComplete)
      case None =>
        new RequestChannel.NoOpResponse(request)
    }

    requestChannel.sendResponse(response) //最终往 requestChannel里面塞
  }

 def sendResponse(response: RequestChannel.Response): Unit = {
   //实际是往对应的processor 里面塞，这也是为什么之前方法参数都带着request的原因，因为在构造response的时候
   //把request.processor塞进去了。
   /*
     abstract class Response(val request: Request) {

     def processor: Int = request.processor
   */
    val processor = processors.get(response.processor)

    if (processor != null) {
      processor.enqueueResponse(response)
    }
  }
```

最后再来个更详细的总览图，把源码分析到的类基本上都对应的加上去了。



请求处理优先级

上面提到的`data-plane`和`control-plane`是时候揭开面纱了。这两个对应的就是数据类请求和控制类请求。

为什么需要分两类请求呢？直接在请求里面用key标明请求是要读写数据啊还是更新元数据不就行了吗？

简单点的说比如我们想删除某个topic，我们肯定是想这个topic马上被删除的，而此时producer还一直往这个topic写数据，那这个情况可能是我们的删除请求排在第N个...等前面的写入请求处理好了才轮到删除的请求。实际上前面哪些往这个topic写入的请求都是没用的，平白的消耗资源。

再或者说进行`Preferred Leader`选举时候，`producer`将`ack`设置为`all`时候，`老leader`还在等着`follower`写完数据向他报告呢，谁知`follower`已经成为了`新leader`，而通知它leader已经变更的请求由于被一堆数据类型请求堵着呢，`老leader`就傻傻的在等着，直到超时。

就是为了解决这种情况，社区将请求分为两类。

那如何让控制类的请求优先被处理？优先队列？

社区采取的是两套`Listener`，即数据类型一个`listener`，控制类一个`listener`。

对应的就是我们上面讲的**网络通信模型，在kafka中有两套！** kafka通过两套监听变相的实现了请求优先级，毕竟数据类型请求肯定很多，控制类肯定少，这样看来控制类肯定比大部分数据类型先被处理！

迂回战术啊。

控制类的和数据类区别就在于，就一个`Porcessor线程`，并且请求队列写死的长度为20。

最后

看源码主要就是得耐心，耐心跟下去。然后再跳出来看。你会发现不过如此，哈哈哈。

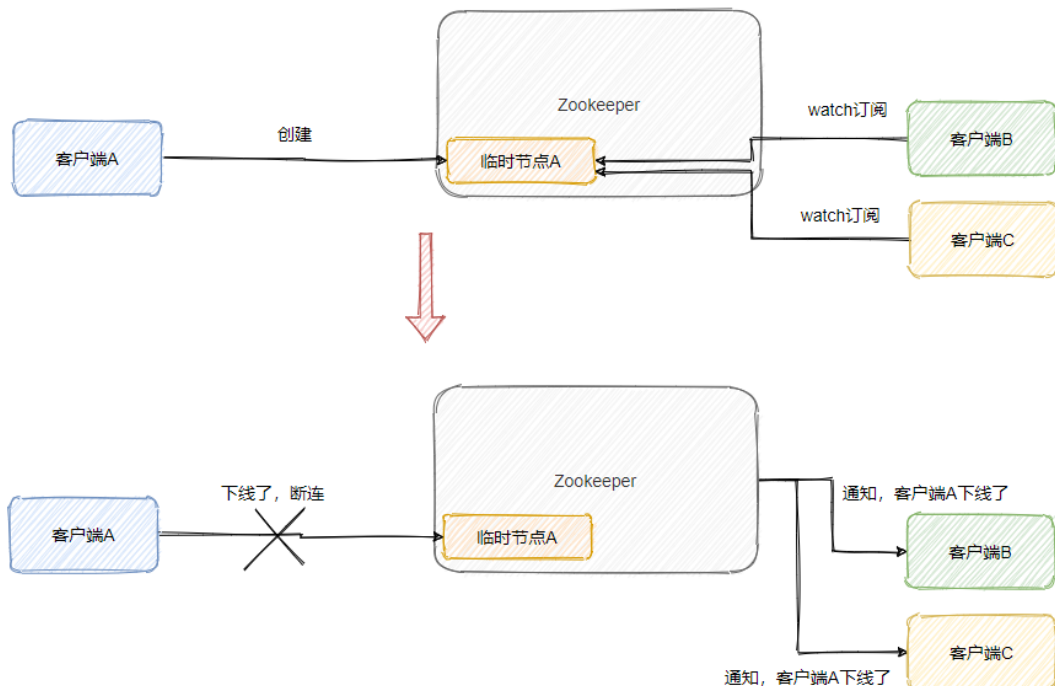
Kafka为什么要抛弃Zookeeper?

ZooKeeper 的作用

ZooKeeper 是一个开源的分布式协调服务框架，你也可以认为它是一个可以保证一致性的分布式(小量)存储系统。特别适合存储一些公共的配置信息、集群的一些元数据等等。

它有持久节点和临时节点，而临时节点这个玩意再配合 Watcher 机制就很有用。

当创建临时节点的客户端与 ZooKeeper 断连之后，这个临时节点就会消失，并且订阅了节点状态变更的客户端会收到这个节点状态变更的通知。

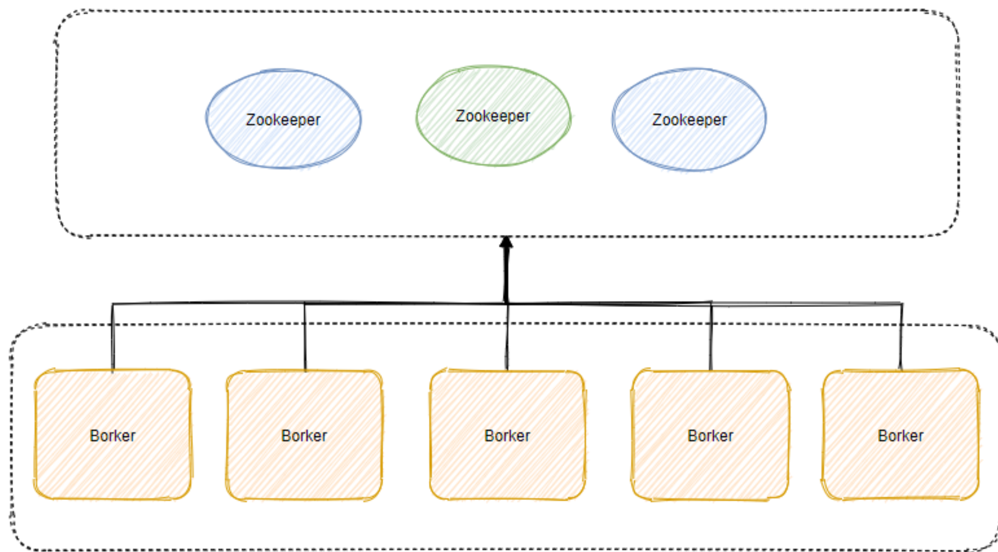


所以集群中某一服务上线或者下线，都可以被检测到。因此可以用来实现服务发现，也可以实现故障转移的监听机制。

Kafka 就是强依赖于 ZooKeeper，没有 ZooKeeper 的话 Kafka 都无法运行。

ZooKeeper 为 Kafka 提供了元数据的管理，例如一些 Broker 的信息、主题数据、分区数据等等。

在每个 Broker 启动的时候，都会和 ZooKeeper 进行交互，这样 ZooKeeper 就存储了集群中所有的主题、配置、副本等信息。



还有一些选举、扩容等机制也都依赖 ZooKeeper 。

例如控制器的选举：每个 Broker 启动都会尝试在 ZooKeeper 注册 `/controller` 临时节点来竞选控制器，第一个创建 `/controller` 节点的 Broker 会被指定为控制器。

竞争失败的节点也会依赖 watcher 机制，监听这个节点，如果控制器宕机了，那么其它 Broker 会继续来争抢，实现控制器的 failover。

从上面就可以得知 ZooKeeper 对 Kafka 来说，**很重要**。

那为什么要抛弃 ZooKeeper

软件架构都是演进的，之所以要变更那肯定是因为出现了瓶颈。

> 先来看看运维的层面的问题。

首先身为一个中间件，需要依赖另一个中间件，这就感觉有点奇怪。

你要说依赖 Netty 这种，那肯定是没问题的。

但是 Kafka 的运行需要提供 ZooKeeper 集群，这其实有点怪怪的。

就等于如果你公司要上 Kafka 就得跟着上 ZooKeeper ，被动了增加了运维的复杂度。

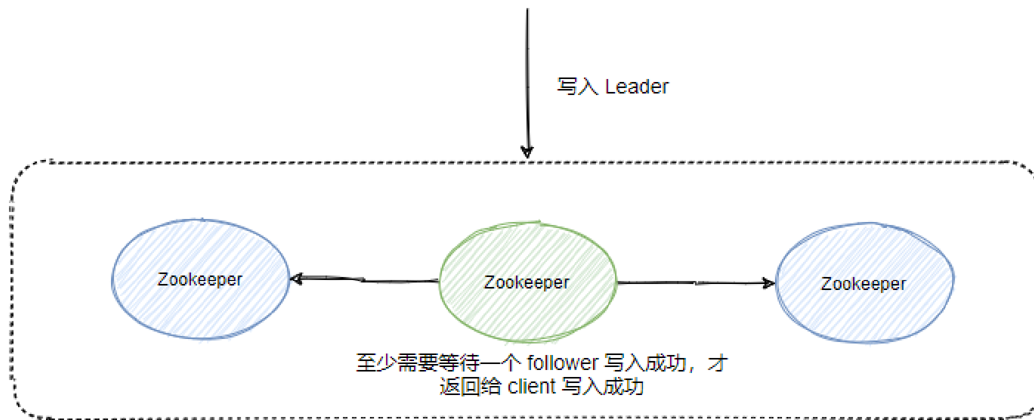
好比你去商场买衣服，要买个上衣，服务员说不单卖，要买就得买一套，这钱是不是多花了？

所以运维人员不仅得照顾 Kafka 集群，还得照顾 ZooKeeper 集群。

> 再看性能层面的问题。

ZooKeeper 有个特点，**强一致性**。

如果 ZooKeeper 集群的某个节点的数据发生变更，则会通知其它 ZooKeeper 节点同时执行更新，就得等着大家(超过半数)都写完了才行，这写入的性能就比较差了。



然后看到上面我说的**小量**存储系统了吧，一般而言，ZooKeeper 只适用于存储一些简单的配置或者是集群的元数据，不是真正意义上的存储系统。

如果写入的数据量过大，ZooKeeper 的性能和稳定性就会下降，可能导致 Watch 的延时或丢失。

所以在 Kafka 集群比较大，分区数很多的时候，ZooKeeper 存储的元数据就会很多，性能就差了。

还有，ZooKeeper 也是分布式的，也需要选举，它的选举也不快，而且发生选举的那段时候是不提供服务的！

> 基于 ZooKeeper 的性能问题 Kafka 之前就做了一些升级。

例如以前 Consumer 的位移数据是保存在 ZooKeeper 上的，所以当提交位移或者获取位移的时候都需要访问 ZooKeeper ，这量一大 ZooKeeper 就顶不住。

所以后面引入了位移主题(Topic是__consumer_offsets)，将位移的提交和获取当做消息一样来处理，存储在日志中，避免了频繁访问 ZooKeeper 性能差的问题。

还有像一些大公司，可能要支持百万分区级别，这目前的 Kafka 单集群架构下是无法支持稳定运行的，也就是目前单集群可以承载的分区数有限。

所以 Kafka 需要去 ZooKeeper 。

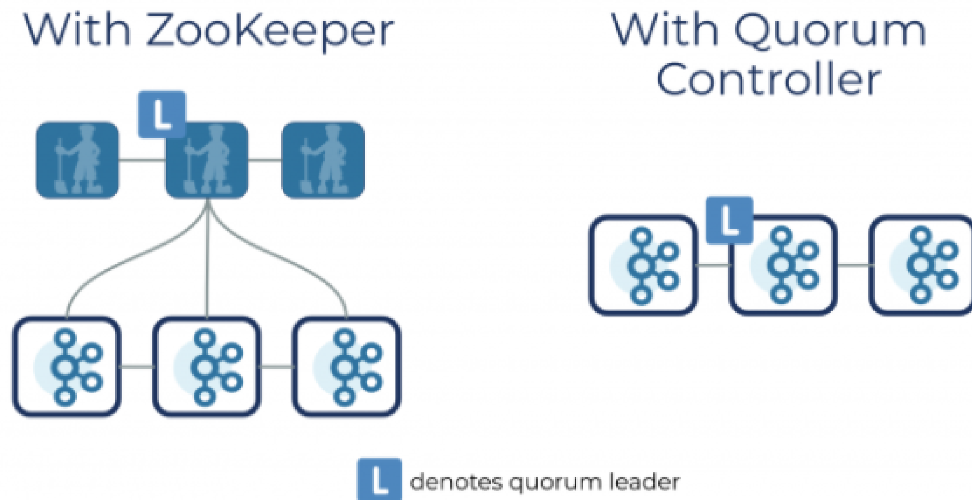
所以没了 Zookeeper 之后的 Kafka 的怎样的？

没了 Zookeeper 的 Kafka 把元数据就存储到自己内部了，利用之前的 Log 存储机制来保存元数据。

就和上面说到的位移主题一样，会有一个元数据主题，元数据会像普通消息一样保存在 Log 中。

所以元数据和之前的位移一样，利用现有的消息存储机制稍加改造来实现了功能，完美！

然后还搞了个 KRaft 来实现 Controller Quorum。



这个协议是基于 Raft 的，协议具体就不展开了，就理解为它能解决 Controller Leader 的选举，并且让所有节点达成共识。

在之前基于 Zookeeper 实现的**单个** Controller 在分区数太大的时候还有个问题，故障转移太慢了。

当 Controller 变更的时候，需要重新加载所有的元数据到新的 Controller 身上，并且需要把这些元数据同步给集群内的所有 Broker。

而 Controller Quorum 中的 Leader 选举切换则很快，因为元数据都已经在 quorum 中同步了，也就是 quorum 的 Broker 都已经有全部了元数据，所以不需要重新加载元数据！

并且其它 Broker 已经基于 Log 存储了一些元数据，所以只需要增量更新即可，不需要全量了。

这波改造下来就解决了之前元数据过多的问题，可以支持更多的分区！

最后

可能看到这里有人会说，那为何一开始不这么实现？

因为 ZooKeeper 是一个功能强大且经过验证的工具，在早期利用它来实现一些功能，多简单哟，都不需要自己实现。

要不是 ZooKeeper 的机制导致了这个瓶颈，也不可能会有这个改造的。

软件就是这样，没必要重复造轮子，合适就好。

参考资料：

- *https://www.confluent.io/blog/kafka-without-zookeeper-a-sneak-peek/*
- *https://time.geekbang.org/column/article/253202*
- *https://www.infoq.cn/article/PHF3gFjUTDhWmctg6kXe*

进阶必看的 RocketMQ，这次一网打尽

今天就和大家一起深入生产级别消息中间件 - RocketMQ 的内核实现，来看看真正落地能支撑万亿级消息容量、低延迟的消息队列到底是如何设计的。

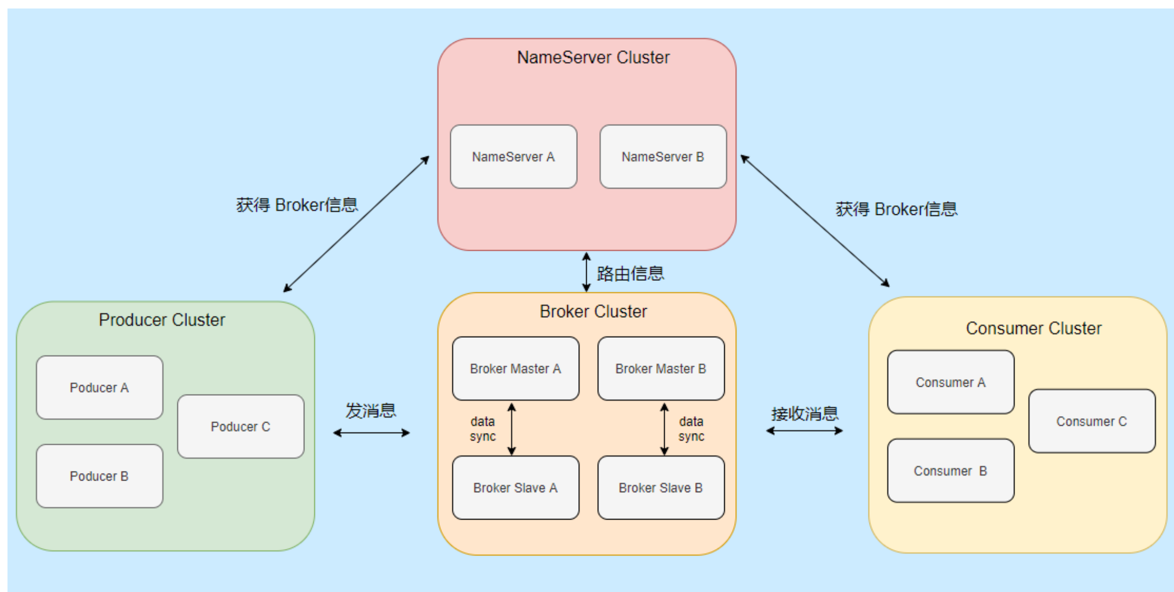
这篇文章我会先介绍整体的架构设计，然后再深入各核心模块的详细设计、核心流程的剖析。

还会提及使用的一些注意点和最佳实践。

话不多说，上车。

RocketMQ 整体架构设计

整体的架构设计主要分为四大部分，分别是：Producer、Consumer、Broker、NameServer。



为了更贴合实际，我画的都是集群部署，像 Broker 我还画了主从。

- Producer：就是消息生产者，可以集群部署。它会先和 NameServer 集群中的随机一台建立长连接，得知当前要发送的 Topic 存在哪台 Broker Master上，然后再与其建立长连接，支持多种负载平衡模式发送消息。
- Consumer：消息消费者，也可以集群部署。它也会先和 NameServer 集群中的随机一台建立长连接，得知当前要消息的 Topic 存在哪台 Broker Master、Slave上，然后它们建立长连接，支持集群消费和广播消费消息。
- Broker：主要负责消息的存储、查询消费，支持主从部署，一个 Master 可以对应多个 Slave，Master 支持读写，Slave 只支持读。**Broker 会向集群中的每一台 NameServer 注册自己的路由信息。**
- NameServer：是一个很简单的 Topic 路由注册中心，支持 Broker 的动态注册和发现，保存 Topic 和 Borker 之间的关系。通常也是集群部署，但是**各 NameServer 之间不会互相通信， 各 NameServer 都有完整的路由信息**，即无状态。

我再用一段话来概括它们之间的交互：

先启动 NameServer 集群，各 NameServer 之间无任何数据交互，Broker 启动之后会向所有 NameServer 定期（每 30s）发送心跳包，包括：IP、Port、TopicInfo，NameServer 会定期扫描 Broker 存活列表，如果超过 120s 没有心跳则移除此 Broker 相关信息，代表下线。

这样每个 NameServer 就知道集群所有 Broker 的相关信息，此时 Producer 上线从 NameServer 就可以得知它要发送的某 Topic 消息在哪个 Broker 上，和对应的 Broker （Master 角色的）建立长连接，发送消息。

Consumer 上线也可以从 NameServer 得知它所要接收的 Topic 是哪个 Broker，和对应的 Master、Slave 建立连接，接收消息。

简单的工作流程如上所述，相信大家对整体数据流转已经有点印象了，我们再来看看每个部分的详细情况。

NameServer

它的特点就是轻量级，无状态。角色类似于 Zookeeper 的情况，从上面描述知道其主要的两个功能就是：Broker 管理、路由信息管理。

总体而言比较简单，我再贴一些字段，让大家有更直观的印象知道它存储了些什么。

```
private final HashMap<String/* topic */, List<QueueData>> topicQueueTable; 主题和队列信息
private final HashMap<String/* brokerName */, BrokerData> brokerAddrTable; Broker 信息
private final HashMap<String/* clusterName */, Set<String/* brokerName */>> clusterAddrTable; 集群对应的 Broker
private final HashMap<String/* brokerAddr */, BrokerLiveInfo> brokerLiveTable; 存活的 Broker
```

Producer

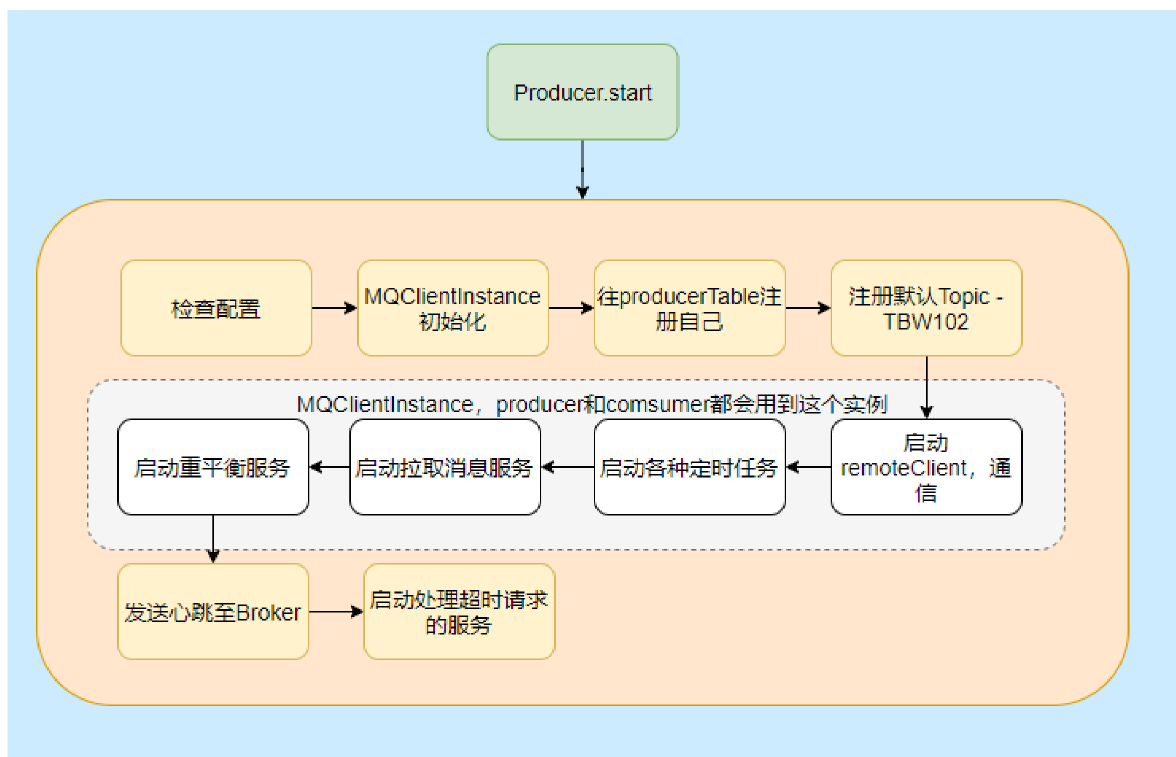
Producer 无非就是消息生产者，那首先它得知道消息要发往哪个 Broker，于是每 30s 会从某台 NameServer 获取 Topic 和 Broker 的映射关系存在本地内存中，如果发现新的 Broker 就会和其建立长连接，每 30s 会发送心跳至 Broker 维护连接。

并且会**轮询当前可以发送的 Broker 来发送消息**，达到负载均衡的目的，在**同步发送情况**下如果发送失败会默认重投两次（retryTimesWhenSendFailed = 2），并且不会选择上次失败的 broker，会向其他 broker 投递。

在**异步发送**失败的情况下也会重试，默认也是两次 （retryTimesWhenSendAsyncFailed = 2），但是仅在同一个 Broker 上重试。

Producer 启动流程

然后我们再来看看 Producer 的启动流程看看都干了些啥。

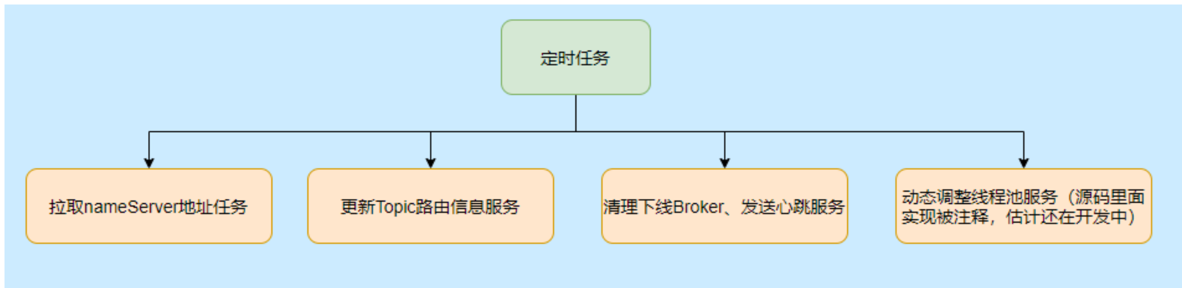


大致启动流程图中已经表明的很清晰的，但是有些细节可能还不清楚，比如重平衡啊，TBW102 啥玩意啊，有哪些定时任务啊，别急都会提到的。

有人可能会问这生产者为什么要启拉取服务、重平衡?

因为 Producer 和 Consumer 都需要用 MQClientInstance，而同一个 clientId 是共用一个 MQClientInstance 的， clientId 是通过本机 IP 和 instanceName（默认值 default）拼起来的，所以多个 Producer 、Consumer 实际用的是一个MQClientInstance。

至于有哪些定时任务，请看下图：



Producer 发消息流程

我们再来看看发消息的流程，大致也不是很复杂，无非就是找到要发送消息的 Topic 在哪个 Broker 上，然后发送消息。

发送消息（同步）
根据 Topic 在本地找路由信息
无
去 nameServer 拉
有无该路由信息
无
有无默认路由主题 - TBW102的 Broker
无
报错
有
有
有
拉倒本地
就选它了
发送消息

现在就知道 TBW102 是啥用的，就是接受自动创建主题的 Broker 启动会把这个默认主题登记到 NameServer，这样当 Producer 发送新 Topic 的消息时候就得知哪个 Broker 可以自动创建主题，然后发往那个 Broker。

而 Broker 接受到这个消息的时候发现没找到对应的主题，但是它接受创建新主题，这样就会创建对应的 Topic 路由信息。

自动创建主题的弊端

自动创建主题那么有可能该主题的消息都只会发往一台 Broker，起不到负载均衡的作用。

因为创建新 Topic 的请求到达 Broker 之后，Broker 创建对应的路由信息，但是心跳是每 30s 发送一次，所以说 NameServer 最长需要 30s 才能得知这个新 Topic 的路由信息。

假设此时发送方还在连续快速的发送消息，那 NameServer 上其实还没有关于这个 Topic 的路由信息，所以**有机会**让别的允许自动创建的 Broker 也创建对应的 Topic 路由信息，这样集群里的 Broker 就能接受这个 Topic 的信息，达到负载均衡的目的，但也有个别 Broker 可能，没收到。

如果发送方这一次发了之后 30s 内一个都不发，之前的那个 Broker 随着心跳把这个路由信息更新到 NameServer 了，那么之后发送该 Topic 消息的 Producer 从 NameServer 只能得知该 Topic 消息只能发往之前的那台 Broker ，这就不均衡了，如果这个新主题消息很多，那台 Broker 负载就很高了。

所以不建议线上开启允许自动创建主题，即 autoCreateTopicEnable 参数。

发送消息故障延迟机制

有一个参数是 sendLatencyFaultEnable，默认不开启。这个参数的作用是对于之前发送超时的 Broker 进行一段时间的退避。

发送消息会记录此时发送消息的时间，如果超过一定时间，那么此 Broker 就在一段时间内不允许发送。

```
private long[] latencyMax = {50L, 100L, 550L, 1000L, 2000L, 3000L, 15000L};
private long[] notAvailableDuration = {0L, 0L, 30000L, 60000L, 120000L, 180000L, 600000L};
```

比如发送时间超过 15000ms 则在 600000 ms 内无法向该 Broker 发送消息。

这个机制其实很关键，发送超时大概率表明此 Broker 负载高，所以先避让一会儿，让它缓一缓，这也是实现消息发送高可用的关键。

小结一下

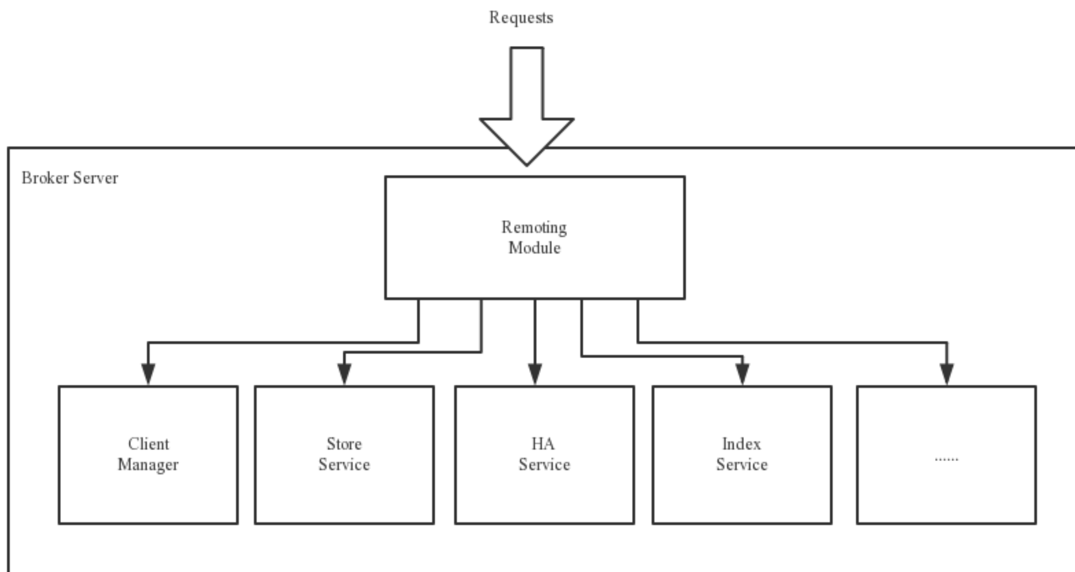
Producer 每 30s 会向 NameSrv 拉取路由信息更新本地路由表，有新的 Broker 就和其建立长连接，每隔 30s 发送心跳给 Broker 。

不要在生产环境开启 autoCreateTopicEnable。

Producer 会通过重试和延迟机制提升消息发送的高可用。

Broker

Broker 就比较复杂一些了，但是非常重要。大致分为以下五大模块，我们来看一下官网的图。



- Remoting 远程模块，处理客户请求。
- Client Manager 管理客户端，维护订阅的主题。
- Store Service 提供消息存储查询服务。
- HA Serivce，主从同步高可用。
- Index Serivce，通过指定key 建立索引，便于查询。

有几个模块没啥可说的就不分析了，先看看存储的。

Broker 的存储

RocketMQ 存储用的是本地文件存储系统，效率高也可靠。

主要涉及到三种类型的文件，分别是 CommitLog、ConsumeQueue、IndexFile。

CommitLog

RocketMQ 的所有主题的消息都存在 CommitLog 中，单个 CommitLog 默认 1G，并且文件名以起始偏移量命名，固定 20 位，不足则前面补 0，比如 00000000000000000000 代表了第一个文件，第二个文件名就是 00000000001073741824，表明起始偏移量为 1073741824，以这样的方式命名用偏移量就能找到对应的文件。

所有消息都是顺序写入的，超过文件大小则开启下一个文件。

ConsumeQueue

ConsumeQueue 消息消费队列，可以认为是 CommitLog 中消息的索引，因为 CommitLog 是糅合了所有主题的消息，所以通过索引才能更加高效的查找消息。

ConsumeQueue 存储的条目是固定大小，只会存储 8 字节的 commitlog 物理偏移量，4 字节的消息长度和 8 字节 Tag 的哈希值，固定 20 字节。

在实际存储中，ConsumeQueue 对应的是一个Topic 下的某个 Queue，每个文件约 5.72M，由 30w 条数据组成。

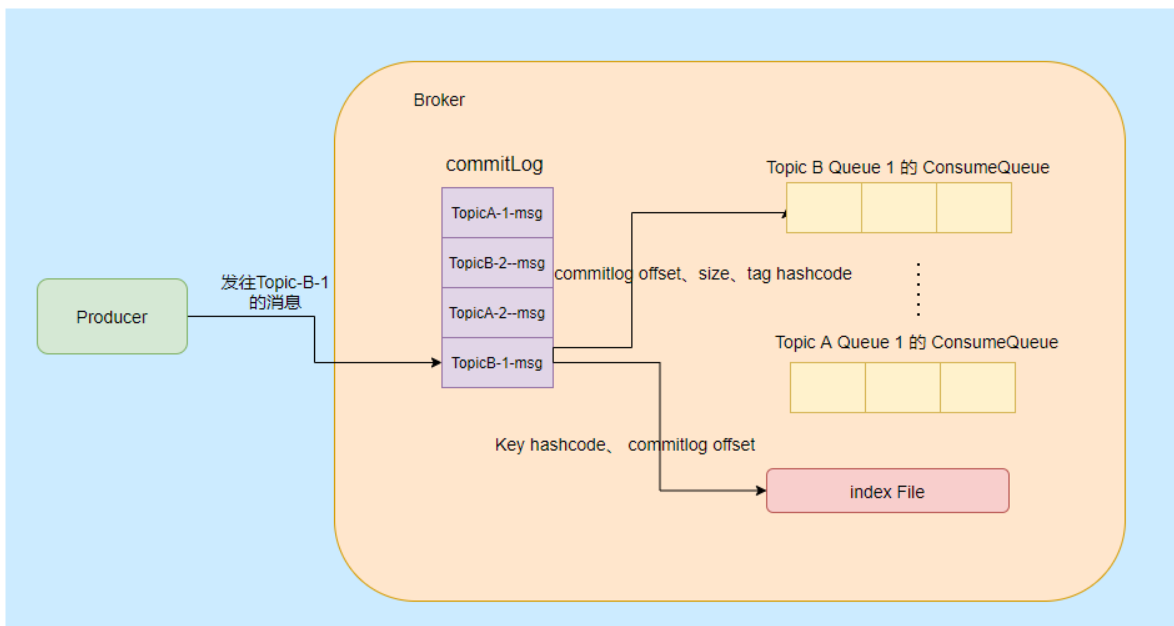
消费者是先从 ConsumeQueue 来得到消息真实的物理地址，然后再去 CommitLog 获取消息。

IndexFile

IndexFile 就是索引文件，是额外提供查找消息的手段，不影响主流程。

通过 Key 或者时间区间来查询对应的消息，文件名以创建时间戳命名，固定的单个 IndexFile 文件大小约为400M，一个 IndexFile 存储 2000W个索引。

我们再来看看以上三种文件的内容是如何生成的：



消息到了先存储到 Commitlog，然后会有一个 ReputMessageService 线程接近实时地将消息转发给消息消费队列文件与索引文件，也就是说是异步生成的。

消息刷盘机制

RocketMQ 提供消息同步刷盘和异步刷盘两个选择，关于刷盘我们都知道效率比较低，单纯存入内存中的话效率是最高的，但是可靠性不高，影响消息可靠性的情况大致有以下几种：

1. Broker 被暴力关闭，比如 kill -9
2. Broker 挂了
3. 操作系统挂了
4. 机器断电
5. 机器坏了，开不了机
6. 磁盘坏了

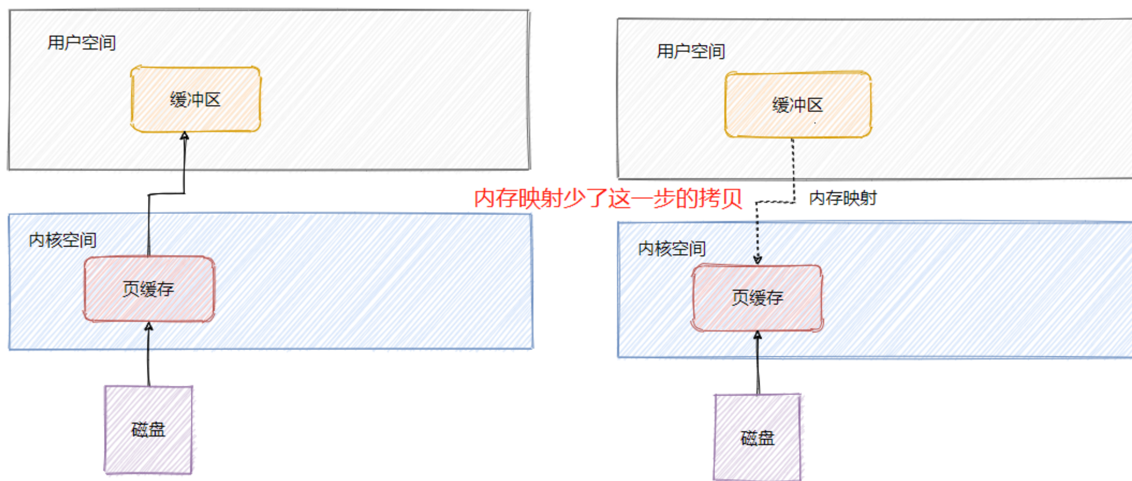
如果都是 1-4 的情况，同步刷盘肯定没问题，异步的话就有可能丢失部分消息，5 和 6就得依靠副本机制了，如果同步双写肯定是稳的，但是性能太差，如果异步则有可能丢失部分消息。

所以需要看场景来使用同步、异步刷盘和副本双写机制。

页缓存与内存映射

Commitlog 是混合存储的，所以所有消息的写入就是顺序写入，对文件的顺序写入和内存的写入速度基本上没什么差别。

并且 RocketMQ 的文件都利用了内存映射即 Mmap，将程序虚拟页面直接映射到页缓存上，无需有内核态再往用户态的拷贝，来看一下我之前文章画的图。



页缓存其实就是操作系统对文件的缓存，用来加速文件的读写，也就是说对文件的写入先写到页缓存中，操作系统会不定期刷盘（时间不可控），对文件的读会先加载到页缓存中，并且根据局部性原理还会预读临近块的内容。

其实也是因为使用内存映射机制，所以 RocketMQ 的文件存储都使用定长结构来存储，方便一次将整个文件映射至内存中。

文件预分配和文件预热

而内存映射也只是做了映射，只有当真正读取页面的时候产生缺页中断，才会将数据真正加载到内存中，所以 RocketMQ 做了一些优化，防止运行时的性能抖动。

文件预分配

CommitLog 的大小默认是1G，当超过大小限制的时候需要准备新的文件，而 RocketMQ 就起了一个后台线程 AllocateMappedFileService，不断的处理 AllocateRequest，AllocateRequest 其实就是预分配的请求，会提前准备好下一个文件的分配，防止在消息写入的过程中分配文件，产生抖动。

文件预热

有一个 warmMappedFile 方法，它会把当前映射的文件，每一页遍历多去，写入一个0字节，然后再调用mlock 和 madvise(MADV_WILLNEED)。

mlock：可以将进程使用的部分或者全部的地址空间锁定在物理内存中，防止其被交换到 swap 空间。

madvise：给操作系统建议，说这文件在不久的将来要访问的，因此，提前读几页可能是个好主意。

小结一下

CommitLog 采用混合型存储，也就是所有 Topic 都存在一起，顺序追加写入，文件名用起始偏移量命名。

消息先写入 CommitLog 再通过后台线程分发到 ConsumerQueue 和 IndexFile 中。

消费者先读取 ConsumerQueue 得到真正消息的物理地址，然后访问 CommitLog 得到真正的消息。

利用了 mmap 机制减少一次拷贝，利用文件预分配和文件预热提高性能。

提供同步和异步刷盘，根据场景选择合适的机制。

Broker 的 HA

从 Broker 会和主 Broker 建立长连接，然后获取主 Broker commitlog 最大偏移量，开始向主 Broker 拉取消息，主 Broker 会返回一定数量的消息，循环进行，达到主从数据同步。

消费者消费消息会先请求主 Broker ，如果主 Broker 觉得现在压力有点大，则会返回从 Broker 拉取消息的建议，然后消费者就去从服务器拉取消息。

Consumer

消费有两种模式，分别是广播模式和集群模式。

广播模式：一个分组下的每个消费者都会消费完整的Topic 消息。

集群模式：一个分组下的消费者瓜分消费Topic 消息。

一般我们用的都是集群模式。

而消费者消费消息又分为推和拉模式，详细看我这篇文章消息队列推拉模式，分别从源码级别分析了 RokcetMQ 和 Kafka 的消息推拉，以及推拉模式的优缺点。

Consumer 端的负载均衡机制

Consumer 会定期的获取 Topic 下的队列数，然后再去查找订阅了该 Topic 的同一消费组的所有消费者信息，默认的分配策略是类似分页排序分配。

将队列排好序，然后消费者排好序，比如队列有 9 个，消费者有 3 个，那消费者-1 消费队列 0、1、2 的消息，消费者-2 消费队列 3、4、5，以此类推。

所以如果负载太大，那么就加队列，加消费者，通过负载均衡机制就可以感知到重平衡，均匀负载。

Consumer 消息消费的重试

难免会遇到消息消费失败的情况，所以需要提供消费失败的重试，而一般的消费失败要么就是消息结构有误，要么就是一些暂时无法处理的状态，所以立即重试不太合适。

RocketMQ 会给**每个消费组**都设置一个重试队列，Topic 是 `%RETRY%+consumerGroup`，并且设定了很多重试级别来延迟重试的时间。

为了利用 RocketMQ 的延时队列功能，重试的消息会先保存在 Topic 名称为“SCHEDULE_TOPIC_XXXX”的延迟队列，在消息的扩展字段里面会存储原来所属的 Topic 信息。

delay 一段时间后再恢复到重试队列中，然后 Consumer 就会消费这个重试队列主题，得到之前的消息。

如果超过一定的重试次数都消费失败，则会移入到死信队列，即 Topic `%DLQ%" + ConsumerGroup` 中，存储死信队列即认为消费成功，因为实在没辙了，暂时放过。

然后我们可以通过人工来处理死信队列的这些消息。

消息的全局顺序和局部顺序

全局顺序就是消除一切并发，一个 Topic 一个队列，Producer 和 Consuemr 的并发都为一。

局部顺序其实就是指某个队列顺序，多队列之间还是能并行的。

可以通过 MessageQueueSelector 指定 Producer 某个业务只发这一个队列，然后 Comsuer 通过 MessageListenerOrderly 接受消息，其实就是加锁消费。

在 Broker 会有一个 mqLockTable ，顺序消息在创建拉取消息任务的时候需要在 Broker 锁定该消息队列，之后加锁成功的才能消费。

而严格的顺序消息其实很难，假设现在都好好的，如果有个 Broker 宕机了，然后发生了重平衡，队列对应的消费者实例就变了，就会有可能会出现乱序的情况，如果要保持严格顺序，那此时就只能让整个集群不可用了。

一些注意点

1、订阅消息是以 ConsumerGroup 为单位存储的，所以ConsumerGroup 中的每个 Consumer 需要有相同的订阅。

因为订阅消息是随着心跳上传的，如果一个 ConsumerGroup 中 Consumer 订阅信息不一样，那么就会出现互相覆盖的情况。

比如消费者 A 订阅 Topic a，消费者 B 订阅 Topic b，此时消费者 A 去 Broker 拿消息，然后 B 的心跳包发出了，Broker 更新了，然后接到 A 的请求，一脸懵逼，没这订阅关系啊。

2、RocketMQ 主从读写分离

从只能读，不能写，并且只有当前客户端读的 offset 和 当前 Broker 已接受的最大 offset 超过限制的物理内存大小时候才会去从读，所以**正常情况下从分担不了流量**

3、单单加机器提升不了消费速度，队列的数量也需要跟上。

4、之前提到的，不要允许自动创建主题

RocketMQ 的最佳实践

这些最佳实践部分参考自官网。

Tags的使用

建议一个应用一个 Topic，利用 tages 来标记不同业务，因为 tages 设置比较灵活，且一个应用一个 Topic 很清晰，能直观的辨别。

Keys的使用

如果有消息业务上的唯一标识，请填写到 keys 字段中，方便日后的定位查找。

提高 Consumer 的消费能力

1、提高消费并行度：增加队列数和消费者数量，提高单个消费者的并行消费线程，参数 consumeThreadMax。

2、批处理消费，设置 consumeMessageBatchMaxSize 参数，这样一次能拿到多条消息，然后比如一个 update语句之前要执行十次，现在一次就执行完。

3、跳过非核心的消息，当负载很重的时候，为了保住那些核心的消息，设置那些非核心的消息，例如此时消息堆积 1W 条了之后，就直接返回消费成功，跳过非核心消息。

NameServer 的寻址

请使用 HTTP 静态服务器寻址（默认），这样 NameServer 就能动态发现。

JVM选项

以下抄自官网：

如果不关心 RocketMQ Broker的启动时间，通过"预触摸" Java 堆以确保在 JVM 初始化期间每个页面都将被分配。

那些不关心启动时间的人可以启用它： -XX:+AlwaysPreTouch
禁用偏置锁定可能会减少JVM暂停， -XX:-UseBiasedLocking
至于垃圾回收，建议使用带JDK 1.8的G1收集器。

> -XX:+UseG1GC -XX:G1HeapRegionSize=16m
> -XX:G1ReservePercent=25
> -XX:InitiatingHeapOccupancyPercent=30

另外不要把-XX:MaxGCPauseMillis的值设置太小，否则JVM将使用一个小的年轻代来实现这个目标，这将导致非常频繁的minor GC，所以建议使用rolling GC日志文件:

> -XX:+UseGCLogFileRotation
> -XX:NumberOfGCLogFiles=5
> -XX:GCLogFileSize=30m

Linux内核参数

以下抄自官网:

- **vm.extra_free_kbytes**，告诉VM在后台回收（kswapd）启动的阈值与直接回收（通过分配进程）的阈值之间保留额外的可用内存。RocketMQ使用此参数来避免内存分配中的长延迟。（与具体内核版本相关）
- **vm.min_free_kbytes**，如果将其设置为低于1024KB，将会巧妙的将系统破坏，并且系统在高负载下容易出现死锁。
- **vm.max_map_count**，限制一个进程可能具有的最大内存映射区域数。RocketMQ将使用mmap加载CommitLog和ConsumeQueue，因此建议将为此参数设置较大的值。（agressiveness --> aggressiveness）
- **vm.swappiness**，定义内核交换内存页面的积极程度。较高的值会增加攻击性，较低的值会减少交换量。建议将值设置为10来避免交换延迟。
- **File descriptor limits**，RocketMQ需要为文件（CommitLog和ConsumeQueue）和网络连接打开文件描述符。我们建议设置文件描述符的值为655350。
- Disk scheduler，RocketMQ建议使用I/O截止时间调度器，它试图为请求提供有保证的延迟。

最后

其实还有很多没讲，比如流量控制、消息的过滤、定时消息的实现，包括底层通信 1+N+M1+M2 的 Reactor 多线程设计等等。

主要是内容太多了，而且也不太影响主流程，所以还是剥离出来之后写吧，大致的一些实现还是讲了的。

包括元信息的交互、消息的发送、存储、消费等等。

关于事务消息的那一块我之前文章也分析过了，所以这个就不再贴了。

可以看到要实现一个生产级别的消息队列还是有很多很多东西需要考虑的，不过大致的架构和涉及到的模块差不多就这些了。

至于具体的细节深入，还是得靠大家自行研究了，我就起个抛砖引玉的作用。

最后个人能力有限，如果哪里有纰漏请抓紧联系鞭挞我！

Kafka和RocketMQ底层存储揭秘，为什么能这么快？

我们都知道 RocketMQ 和 Kafka 消息都是存在磁盘中的，那为什么消息存磁盘读写还可以这么快？有没有做了什么优化？都是存磁盘它们两者的实现之间有什么区别么？各自有什么优缺点？

今天我们就来一探究竟。

存储介质-磁盘

一般而言消息中间件的消息都存储在本地文件中，因为从效率来看直接放本地文件是最快的，并且稳定性最高。毕竟要是放类似数据库等第三方存储中的话，就多一个依赖少一份安全，并且还有网络的开销。

那对于将消息存入磁盘文件来说一个流程的瓶颈就是磁盘的写入和读取。我们知道磁盘相对而言读写速度较慢，那通过磁盘作为存储介质如何实现高吞吐呢？

顺序读写

答案就是**顺序读写**。

首先了解一下**页缓存**，页缓存是操作系统用来作为磁盘的一种缓存，减少磁盘的I/O操作。

在写入磁盘的时候其实是写入页缓存中，使得对磁盘的写入变成对内存的写入。写入的页变成脏页，然后操作系统会在合适的时候将脏页写入磁盘中。

在读取的时候如果页缓存命中则直接返回，如果页缓存 miss 则产生缺页中断，从磁盘加载数据至页缓存中，然后返回数据。

并且在读的时候会**预读**，根据局部性原理当读取的时候会把相邻的磁盘块读入页缓存中。在写入的时候会**后写**，写入的也是页缓存，这样存着可以将一些小的写入操作合并成大的写入，然后再刷盘。

而且根据磁盘的构造，顺序 I/O 的时候，磁头几乎不用换道，或者换道的时间很短。

根据网上的一些测试结果，顺序写盘的速度比随机写内存还要快。

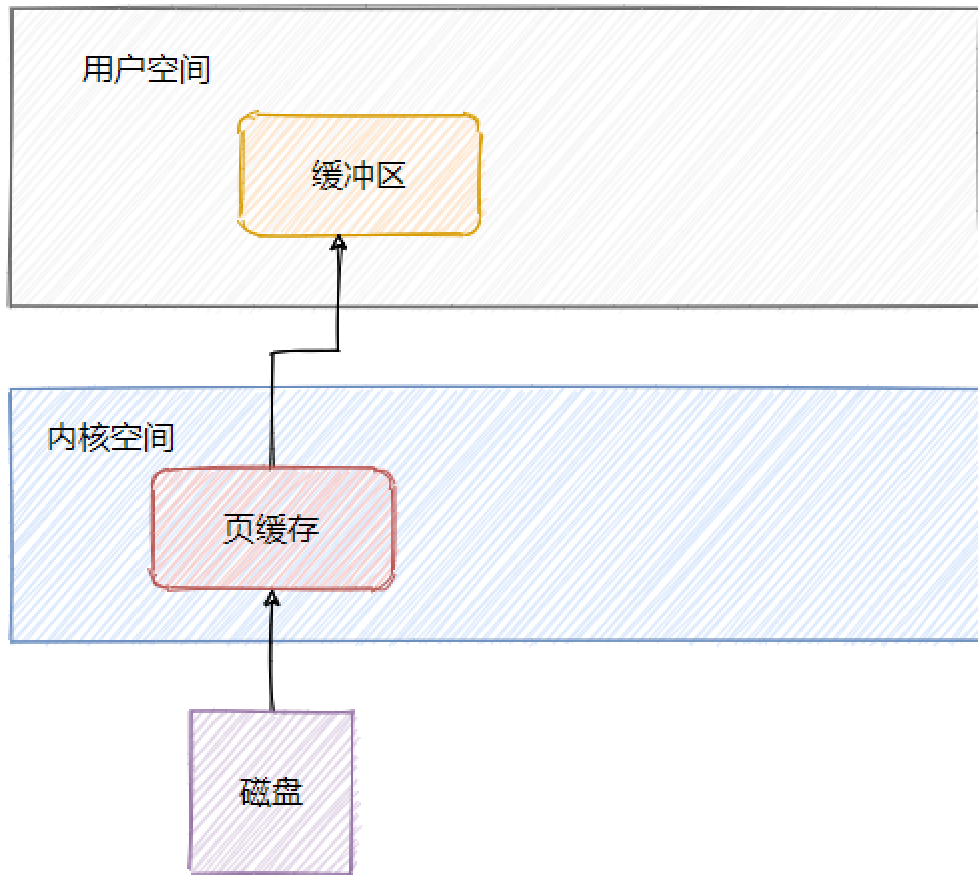
当然这样的写入存在数据丢失的风险，例如机器突然断电，那些还未刷盘的脏页就丢失了。不过可以调用 `fsync` 强制刷盘，但是这样对于性能的损耗较大。

因此**一般建议通过多副本机制来保证消息的可靠，而不是同步刷盘。**

可以看到顺序 I/O 适应磁盘的构造，并且还有预读和后写。 RocketMQ 和 Kafka 都是顺序写入和近似顺序读取。它们都采用文件追加的方式来写入消息，只能在日志文件尾部写入新的消息，老的消息无法更改。

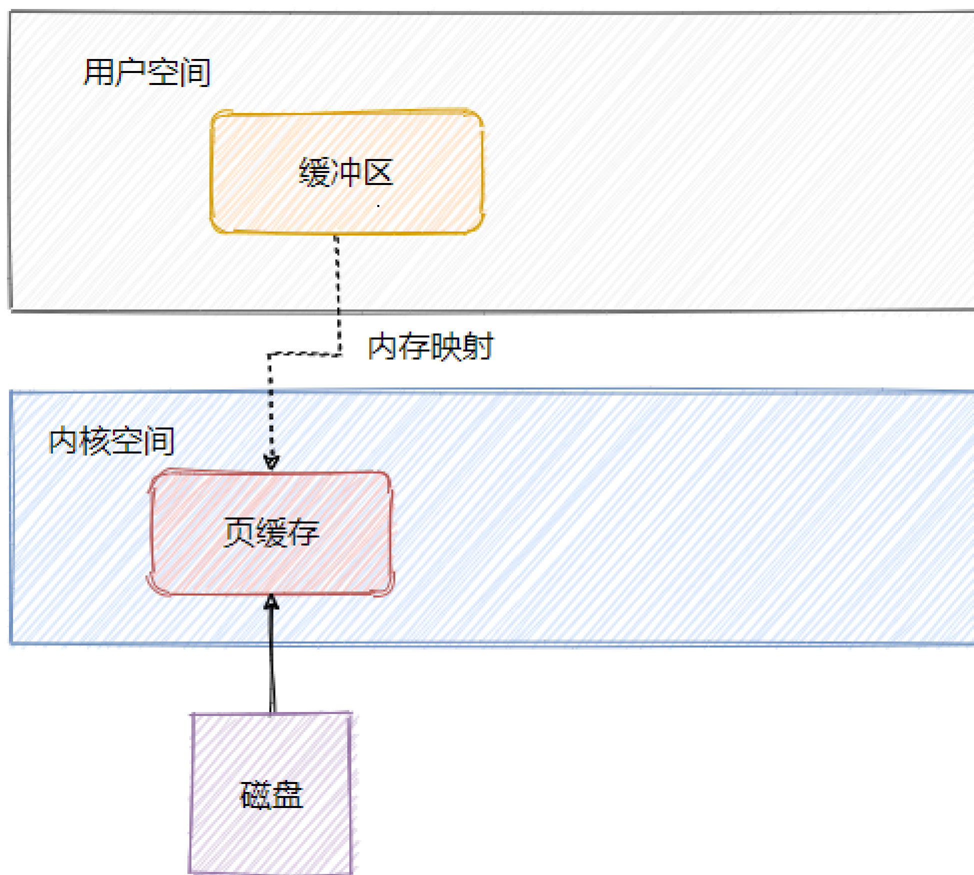
mmap-文件内存映射

从上面可知访问磁盘文件会将数据加载到页缓存中，但是页缓存属于内核空间，用户空间访问不了，因此数据还需要拷贝到用户空间缓冲区。



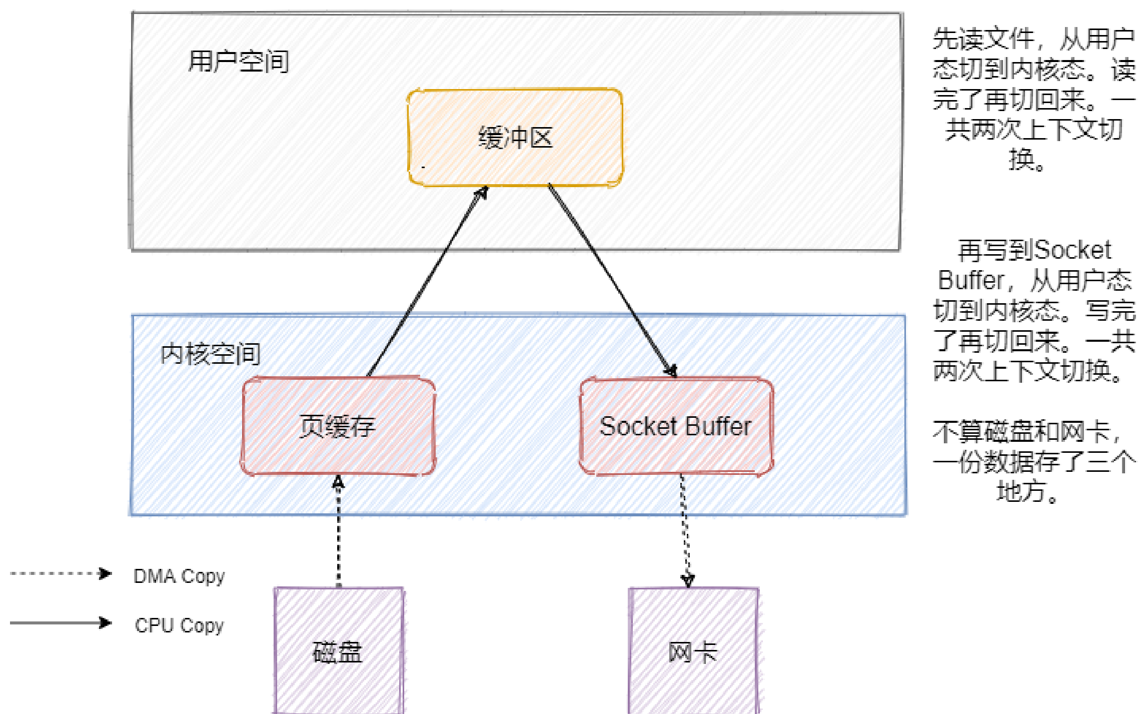
可以看到数据需要从页缓存再经过一次拷贝程序才能访问的到，因此还可以通过`mmap`来做一波优化，利用内存映射文件来避免拷贝。

简单的说**文件映射就是将程序虚拟页面直接映射到页缓存上，这样就无需有内核态再往用户态的拷贝，而且也避免了重复数据的产生**。并且也不必再通过调用`read`或`write`方法对文件进行读写，**可以通过映射地址加偏移量的方式直接操作。**



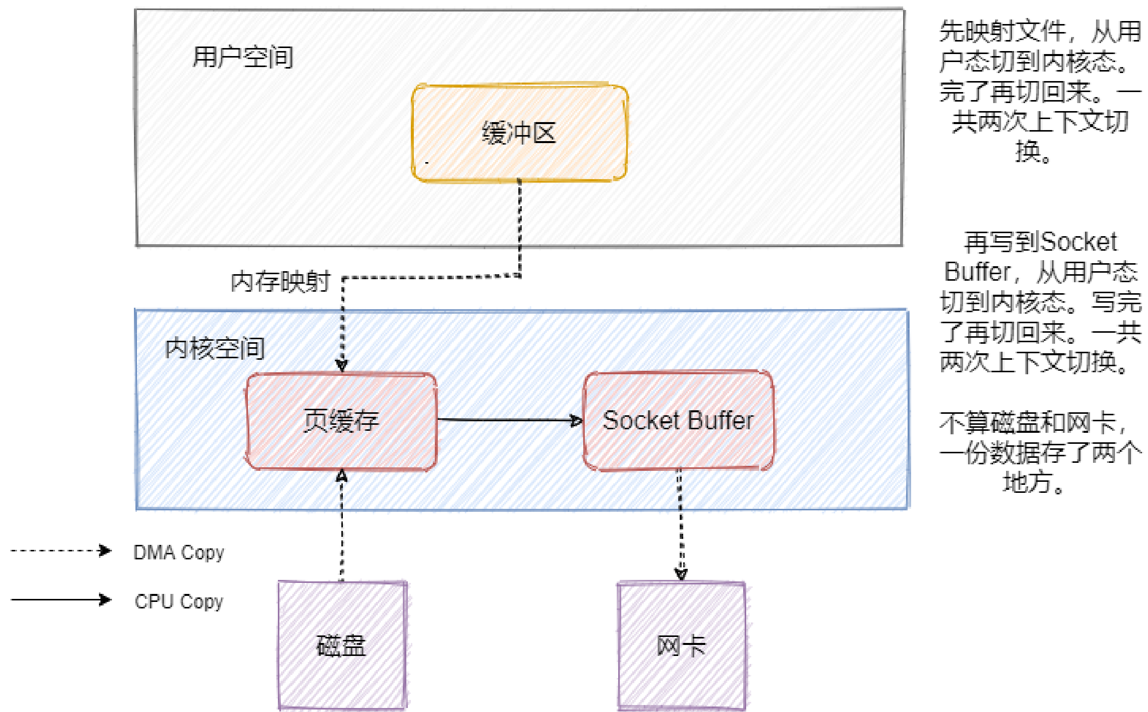
sendfile-零拷贝

既然消息是存在磁盘中的，那消费者来拉消息的时候就得从磁盘拿。我们先来看看一般发送文件的流程是如何的。



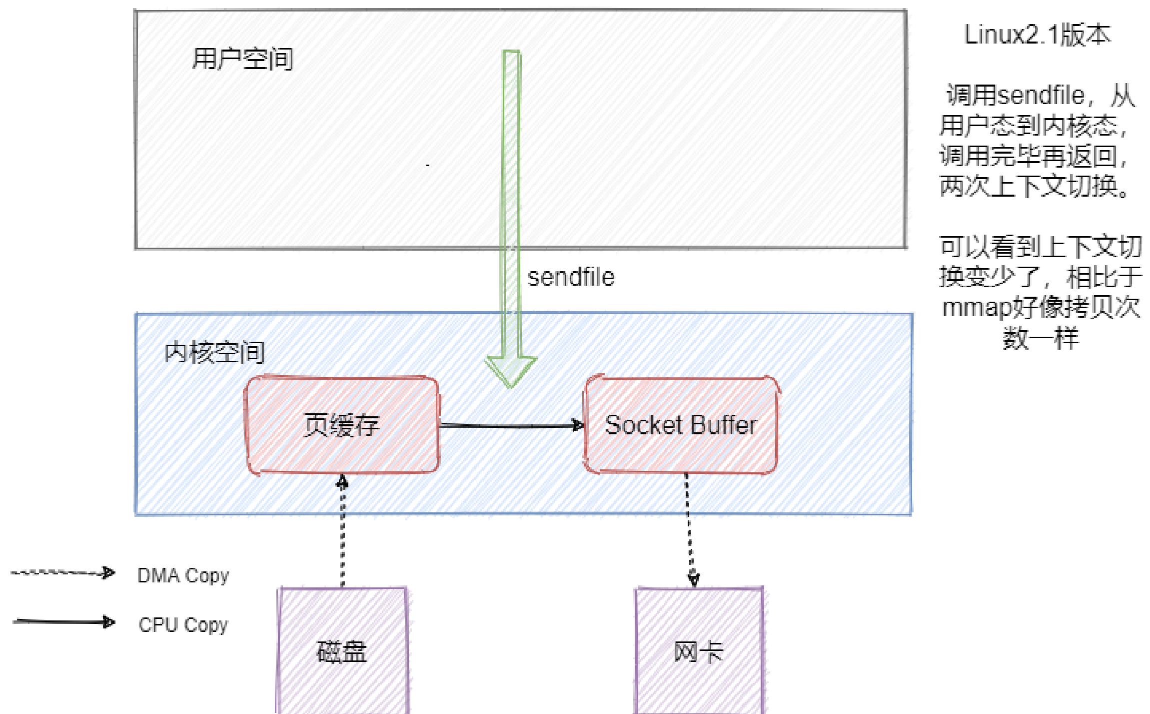
简单说下 `DMA` 是什么，全称 Direct Memory Access ，它可以**独立地直接读写系统内存**，不需要 CPU 介入，像显卡、网卡之类都会用 `DMA` 。

可以看到数据其实是冗余的，那我们来看看 `mmap` 之后的发送文件流程是怎样的。

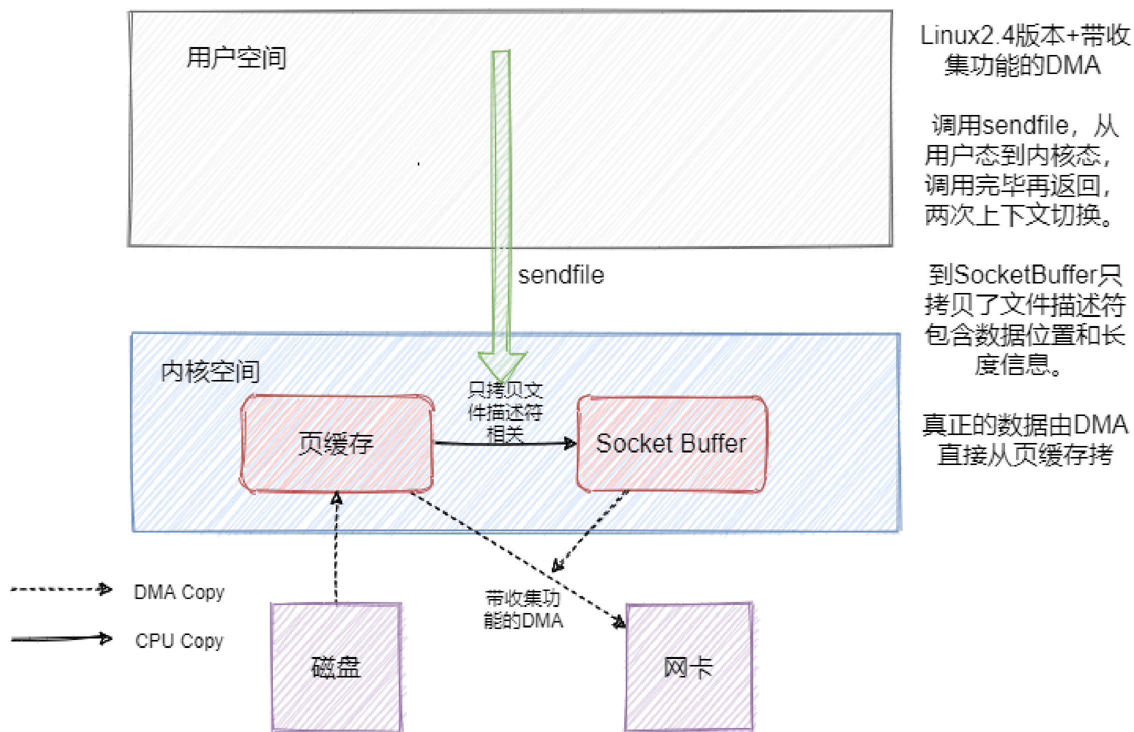


可以看到上下文切换的次数没有变化，但是数据少拷贝一份，这和我们上文提到的`mmap`能达到的效果是一样的。

但是数据还是冗余了一份，这不是可以直接把数据从页缓存拷贝到网卡不就好了嘛？`sendfile`就有这个功效。我们先来看看Linux2.1版本中的`sendfile`。



因为就一个系统调用就满足了发送的需求，相比`read + write`或者`mmap + write`上下文切换肯定是少了的，但是好像数据还是有冗余啊。是的，因此 Linux2.4 版本的 sendfile ＋带「分散-收集（Scatter-gather）」的DMA。实现了真正的无冗余。



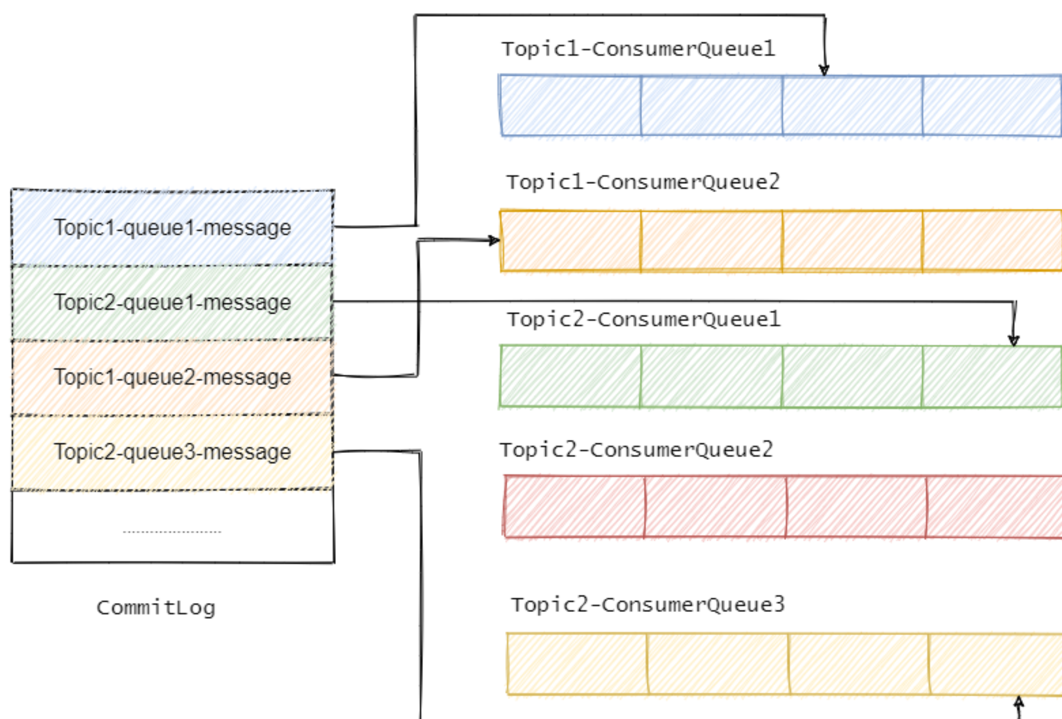
这就是我们常说的零拷贝，在 Java 中`FileChannal.transferTo()`底层用的就是`sendfile`。

接下来我们看看以上说的几点在 RocketMQ 和 Kafka中是如何应用的。

RocketMQ 和 Kafka 的应用

RocketMQ

采用`Topic混合追加方式`，即一个 CommitLog 文件中会包含分给此 Broker 的所有消息，不论消息属于哪个 Topic 的哪个 Queue 。



所以所有的消息过来都是**顺序追加写入到 CommitLog 中**，并且建立消息对应的 CosumerQueue ，然后消费者是通过 CosumerQueue 得到消息的真实物理地址再去 CommitLog 获取消息的。可以将 CosumerQueue 理解为消息的索引。

在 RocketMQ 中不论是 CommitLog 还是 CosumerQueue 都采用了 mmap。

```
// RocketMQ - MappedFile#init
this.fileChannel = new RandomAccessFile(this.file, "rw").getChannel();
this.mappedByteBuffer = this.fileChannel.map(MapMode.READ_WRITE, 0, fileSize);
```

在发消息的时候默认用的是**将数据拷贝到堆内存中，然后再发送**。我们来看下代码。

```
    private boolean transferMsgByHeap = true; //默认是true

    public boolean isTransferMsgByHeap() {
        return transferMsgByHeap;
    }

    public void setTransferMsgByHeap(final boolean transferMsgByHeap) { //也可以配置
        this.transferMsgByHeap = transferMsgByHeap;
    }
```

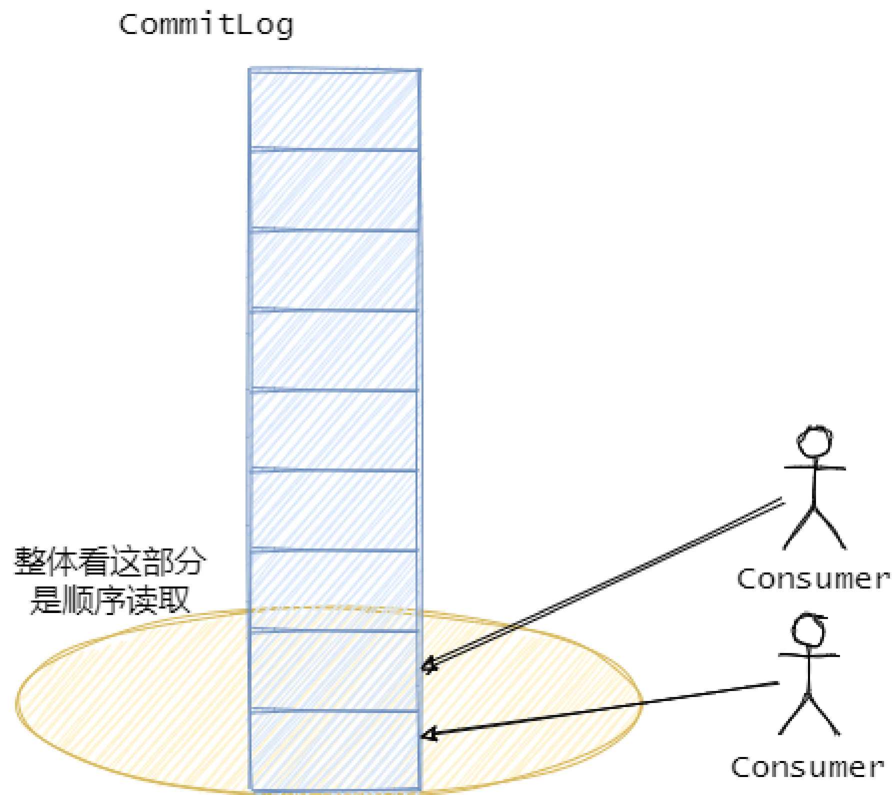
可以看到这个配置 `transferMsgByHeap` 默认是 true ，那我们再看消费者拉消息时候的代码。

```
if (this.brokerController.getBrokerConfig().isTransferMsgByHeap()) { //默认走这里
    final long beginTimeMills = this.brokerController.getMessageStore().now();
    final byte[] r = this.readGetMessageResult(getMessageResult, requestHeader.getConsumerGroup(),
requestHeader.getTopic(), requestHeader.getQueueId()); //将数据写入堆Buffer中，返回字节数组
    this.brokerController.getBrokerStatsManager().incGroupGetLatency(requestHeader.getConsumerGroup(),
        requestHeader.getTopic(), requestHeader.getQueueId(),
        (int) (this.brokerController.getMessageStore().now() - beginTimeMills));
    response.setBody(r); //塞入响应中
} else {
    try {
        //不经过堆，直接 mapedBuffer 发送
        FileRegion fileRegion =
            new ManyMessageTransfer(response.encodeHeader(getMessageResult.getBufferTotalSize()),
getMessageResult);
        channel.writeAndFlush(fileRegion).addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                getMessageResult.release();
                ..........
            }
        });
    } catch (Throwable e) {
       .....
    }

    response = null;
}
```

可以看到 RocketMQ 默认把消息拷贝到堆内 Buffer 中，再塞到响应体里面发送。但是可以通过参数配置不经过堆，不过也并没有用到真正的零拷贝，而是通过mapedBuffer 发送到 SocketBuffer 。

所以 RocketMQ 用了顺序写盘、mmap。并没有用到 sendfile ，还有一步页缓存到 SocketBuffer 的拷贝。

然后拉消息的时候严格的说对于 CommitLog 来说读取是随机的，因为 CommitLog 的消息是混合的存储的，**但是从整体上看，消息还是从 CommitLog 顺序读的，都是从旧数据到新数据有序的读取。**并且一般而言消息存进去马上就会被消费，因此消息这时候应该还在页缓存中，所以不需要读盘。



而且我们在上面提到，**页缓存会定时刷盘，这刷盘不可控，并且内存是有限的，会有swap等情况**，而且**mmap其实只是做了映射，当真正读取页面的时候产生缺页中断，才会将数据真正加载到内存中**，这对于消息队列来说可能会产生监控上的毛刺。

因此 RocketMQ 做了一些优化，有：**文件预分配和文件预热。**

文件预分配

CommitLog 的大小默认是1G，当超过大小限制的时候需要准备新的文件，而 RocketMQ 就起了一个后台线程 `AllocateMappedFileService`，不断的处理 `AllocateRequest`，AllocateRequest其实就是预分配的请求，会提前准备好下一个文件的分配，防止在消息写入的过程中分配文件，产生抖动。

文件预热

有一个`warmMappedFile`方法，它会把当前映射的文件，每一页遍历多去，写入一个0字节，然后再调用`mlock` 和 `madvise(MADV_WILLNEED)`。

```java
public void warmMappedFile(FlushDiskType type, int pages) {
    long beginTime = System.currentTimeMillis();
    ByteBuffer byteBuffer = this.mappedByteBuffer.slice();
    int flush = 0;
    long time = System.currentTimeMillis();

    for (int i = 0, j = 0; i < this.fileSize; i += MappedFile.OS_PAGE_SIZE, j++) {
        byteBuffer.put(i, (byte) 0);
        // force flush when flush disk type is sync
        if (type == FlushDiskType.SYNC_FLUSH) {
            if ((i / OS_PAGE_SIZE) - (flush / OS_PAGE_SIZE) >= pages) {
                flush = i;
                mappedByteBuffer.force();
            }
        }

        // prevent gc
        if (j % 1000 == 0) {
            log.info("j={}, costTime={}", j, System.currentTimeMillis() - time);
            time = System.currentTimeMillis();
            try {
                Thread.sleep(0); //让其他线程有机会执行，为什么能防止gc，我不太明白。
            } catch (InterruptedException e) {
                log.error("Interrupted", e);
            }
        }
    }

    // force flush when prepare load finished
    if (type == FlushDiskType.SYNC_FLUSH) {
        log.info("mapped file warm-up done, force to disk, mappedFile={}, costTime={}",
            this.getFileName(), System.currentTimeMillis() - beginTime);
        mappedByteBuffer.force();
    }
    log.info("mapped file warm-up done. mappedFile={}, costTime={}", this.getFileName(),
        System.currentTimeMillis() - beginTime);

    this.mlock();
}
```

我们再来看下`this.mlock`，内部其实就是调用了`mlock`和`madvise(MADV_WILLNEED)`。

```java
public void mlock() {
    final long beginTime = System.currentTimeMillis();
    final long address = ((DirectBuffer) (this.mappedByteBuffer)).address();
    Pointer pointer = new Pointer(address);
    {
        int ret = LibC.INSTANCE.mlock(pointer, new NativeLong(this.fileSize));

    }

    {
        int ret = LibC.INSTANCE.madvise(pointer, new NativeLong(this.fileSize), LibC.MADV_WILLNEED);

    }
}
```

mlock：可以将进程使用的部分或者全部的地址空间锁定在物理内存中，防止其被交换到swap空间。

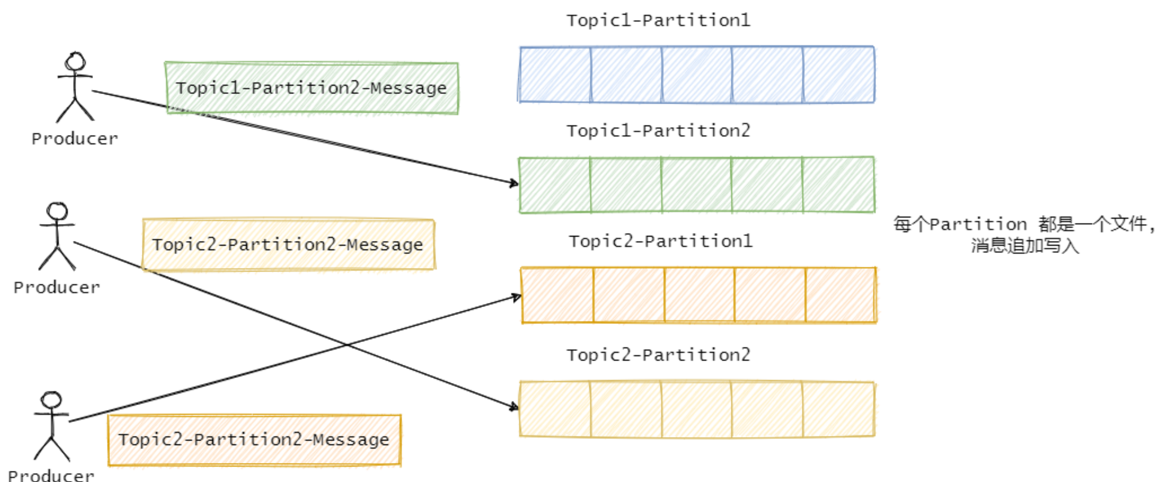
madvise：给操作系统建议，说这文件在不久的将来要访问的，因此，提前读几页可能是个好主意。

RocketMQ 小结

顺序写盘，整体来看是顺序读盘，并且使用了 mmap，不是真正的零拷贝。又因为页缓存的不确定性和 mmap 惰性加载(访问时缺页中断才会真正加载数据)，用了文件预先分配和文件预热即每页写入一个0字节，然后再调用`mlock`和`madvise(MADV_WILLNEED)`。

Kafka

Kafka 的日志存储和 RocketMQ 不一样，它是一个分区一个文件。



Kafka 的消息写入对于单分区来说也是顺序写，如果分区不多的话从整体上看也算顺序写，它的日志文件并没有用到 mmap，而索引文件用了 mmap。但发消息 Kafka 用到了零拷贝。

对于消息的写入来说 mmap 其实没什么用，因为消息是从网络中来。而对于发消息来说 sendfile 对比 mmap+write 我觉得效率更高，因为少了一次页缓存到 SocketBuffer 中的拷贝。

来看下Kafka发消息的源码，最终调用的是 `FileChannel.transferTo`，底层就是 sendfile。

```
@Override
public long writeTo(GatheringByteChannel destChannel, long offset, int length) throws IOException {
    .............
    final long bytesTransferred;
    if (destChannel instanceof TransportLayer) {
        TransportLayer tl = (TransportLayer) destChannel;
        bytesTransferred = tl.transferFrom(channel, position, count); //看下面
    } else {
        bytesTransferred = channel.transferTo(position, count, destChannel);
    }
    return bytesTransferred;
}

@Override
public long transferFrom(FileChannel fileChannel, long position, long count) throws IOException {
    return fileChannel.transferTo(position, count, socketChannel); //最终调用的是这个
}
```

从 Kafka 源码中我没看到有类似于 RocketMQ的 mlock 等操作，我觉得原因是首先日志也没用到 mmap，然后 swap 其实可以通过 Linux 系统参数 `vm.swappiness` 来调节，这里建议设置为1，而不是0。

假设内存真的不足，设置为 0 的话，在内存耗尽的情况下，又不能 swap，则会突然中止某些进程。设置个 1，起码还能拖一下，如果有良好的监控手段，还能给个机会发现一下，不至于突然中止。

RocketMQ & Kafka 对比

首先**都是顺序写入，不过 RocketMQ 是把消息都存一个文件中，而 Kafka 是一个分区一个文件。**

每个分区一个文件在**迁移或者数据复制层面上来说更加得灵活。**

但是**分区多了的话，写入需要频繁的在多个文件之间来回切换，对于每个文件来说是顺序写入的，但是从全局看其实算随机写入，并且读取的时候也是一样，算随机读。**而就一个文件的 RocketMQ 就没这个问题。

从发送消息来说 RocketMQ 用到了 mmap + write 的方式，并且通过预热来减少大文件 mmap 因为缺页中断产生的性能问题。而 Kafka 则用了 sendfile，相对而言我觉得 kafka 发送的效率更高，因为少了一次页缓存到 SocketBuffer 中的拷贝。

并且 swap 问题也可以通过系统参数来设置。

番外篇：定时任务的终极实现--时间轮，在Netty和Kafka中如何应用的？为什么不用Timer、延时线程池？

最近看 Kafka 看到了时间轮算法，记得以前看 Netty 也看到过这玩意，没太过关注。今天就来看看时间轮到底是什么东西。

为什么要用时间轮算法来实现延迟操作?

延时操作 Java 不是提供了 Timer 么?

还有 DelayQueue 配合线程池或者 ScheduledThreadPool 不香吗?

我们先来简单看看 Timer、DelayQueue 和 ScheduledThreadPool 的相关实现，看看它们是如何实现延时任务的，源码之下无秘密。再来剖析下为何 Netty 和 Kafka 特意实现了时间轮来处理延迟任务。

Timer

Timer 可以实现延时任务，也可以实现周期性任务。我们先来看看 Timer 核心属性和构造器。

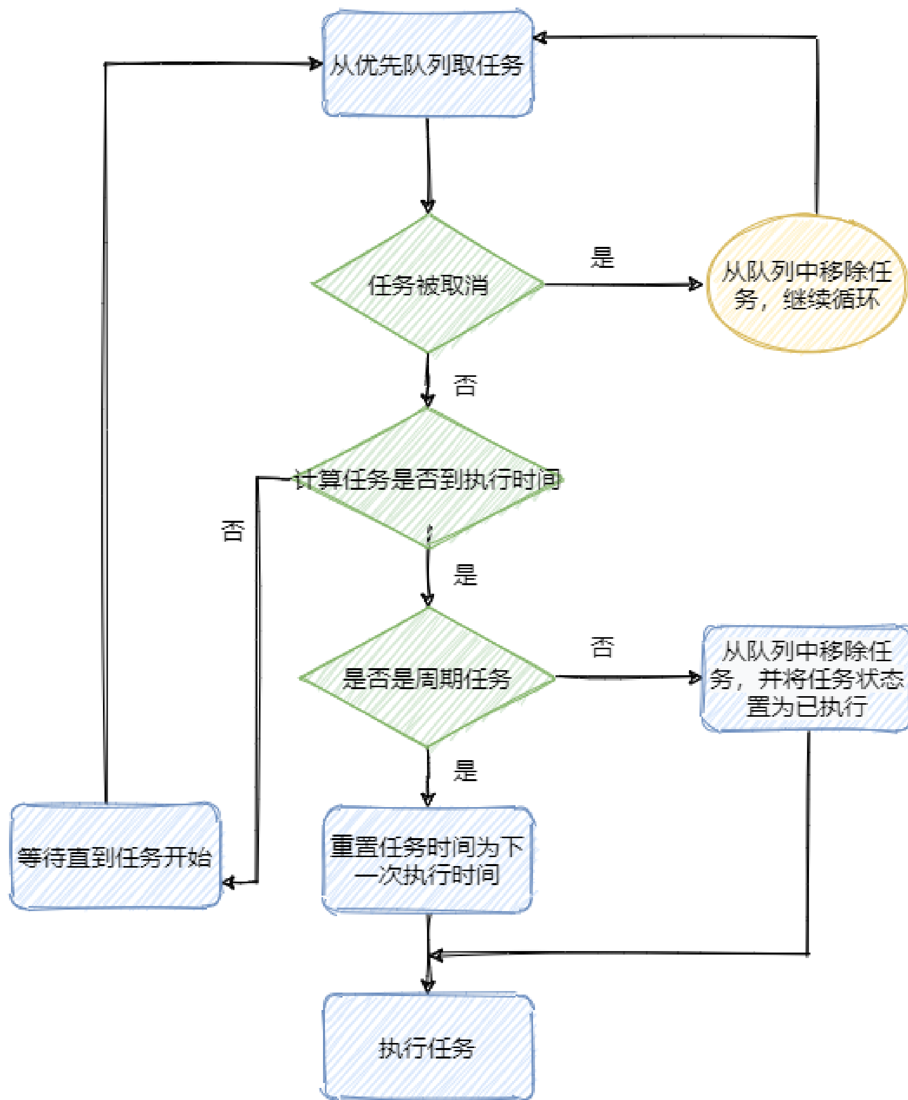
```
private final TaskQueue queue = new TaskQueue(); //基于数组实现的优先队列

private final TimerThread thread = new TimerThread(queue); //执行延时任务的线程

public Timer(String name) {
    thread.setName(name);
    thread.start(); //构造时候默认启动线程
}
```

核心就是一个优先队列和封装的执行任务的线程，从这我们也可以看到一个 Timer 只有一个线程执行任务。

再来看看如何实现延时和周期性任务的。我先**简单的概括一下**，首先维持一个小顶堆，即最快需要执行的任务排在优先队列的第一个，根据堆的特性我们知道插入和删除的时间复杂度都是 O(logn)。

然后 TimerThread 不断地拿排着的第一个任务的执行时间和当前时间做对比。如果时间到了先看看这个任务是不是周期性执行的任务，如果是则修改当前任务时间为下次执行的时间，如果不是周期性任务则将任务从优先队列中移除。最后执行任务。如果时间还未到则调用 `wait()` 等待。

再看下图，整理下流程。



流程知道了再对着看下代码，这块就差不多了。看代码不爽的可以跳过代码部分，影响不大。

先来看下 TaskQueue，就简单看下插入任务的过程，就是个普通的堆插入操作。

```
class TaskQueue {
  private TimerTask[] queue = new TimerTask[128]; //默认128的TimerTask数组

  void add(TimerTask task) {
    // Grow backing store if necessary
    if (size + 1 == queue.length) //扩容
        queue = Arrays.copyOf(queue, 2*queue.length);

    queue[++size] = task; //先加到数组最后一个位置
    fixUp(size); //然后调整堆
  }
   private void fixUp(int k) { //时间复杂度为 O(logn)
      while (k > 1) {
          int j = k >> 1;
          if (queue[j].nextExecutionTime <= queue[k].nextExecutionTime) //通过任务执行时间对比
              break;
          TimerTask tmp = queue[j];  queue[j] = queue[k]; queue[k] = tmp;
          k = j;
      }
  }
}
```

再来看看 TimerThread 的 `run` 操作。

```java
private void mainLoop() {
    while (true) {
        try {
            synchronized(queue) {
                while (queue.isEmpty() && newTasksMayBeScheduled)
                    queue.wait();
                if (queue.isEmpty())
                    break;
                long currentTime, executionTime;
                task = queue.getMin(); //获取任务
                synchronized(task.lock) {
                    if (task.state == TimerTask.CANCELLED) { //取消则移除并继续循环
                        queue.removeMin();
                        continue;
                    }
                    currentTime = System.currentTimeMillis();
                    executionTime = task.nextExecutionTime;
                    if (taskFired = (executionTime<=currentTime)) { //如何执行时间到了
                        if (task.period == 0) { //不是周期任务
                            queue.removeMin(); //移除任务
                            task.state = TimerTask.EXECUTED; //设置状态已执行
                        } else { // 如果是周期任务，更新时间为下次执行时间
                            queue.rescheduleMin(
                              task.period<0 ? currentTime   - task.period
                                            : executionTime + task.period);
                        }
                    }
                }
                if (!taskFired) //还未到达则等待
                    queue.wait(executionTime - currentTime);
            }
            if (taskFired) //执行任务
                task.run();
        } catch(InterruptedException e) {
        }
    }
}
```

小结一下

可以看出 Timer 实际就是根据任务的执行时间维护了一个优先队列，并且起了一个线程不断地拉取任务执行。

有什么弊端呢？

首先**优先队列的插入和删除的时间复杂度是O(logn)**，当数据量大的时候，频繁的入堆出堆性能有待考虑。

并且是**单线程执行**，那么如果一个任务执行的时间过久则会影响下一个任务的执行时间(当然你任务的run要是异步执行也行)。

并且从代码可以看到**对异常没有做什么处理**，那么一个任务出错的时候会导致之后的任务都无法执行。

ScheduledThreadPoolExecutor

在说 ScheduledThreadPoolExecutor 之前我们再看下 Timer 的注释，注释可都是干货千万不要错过。我做了点修改，突出了下重点。

> Java 5.0 introduced ScheduledThreadPoolExecutor, It is effectively a more versatile replacement for the Timer, it allows multiple service threads. Configuring with one thread makes it equivalent to Timer。

简单翻译下：1.5 引入了 ScheduledThreadPoolExecutor，它是一个具有更多功能的 Timer 的替代品，**允许多个服务线程**。如果设置一个服务线程和 Timer 没啥差别。

从注释看出相对于 Timer ，可能就是单线程跑任务和多线程跑任务的区别。我们来看下。

```java
public class ScheduledThreadPoolExecutor
        extends ThreadPoolExecutor
        implements ScheduledExecutorService {

    ......
    public ScheduledThreadPoolExecutor(int corePoolSize) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
              new DelayedWorkQueue());
    }
    private class ScheduledFutureTask<V>
        extends FutureTask<V> implements RunnableScheduledFuture<V> {
    ......
}
```

继承了 ThreadPoolExecutor，实现了 ScheduledExecutorService。可以定性操作就是正常线程池差不多了。区别就在于两点，一个是 ScheduledFutureTask ，一个是 DelayedWorkQueue。

其实 DelayedWorkQueue 就是优先队列，也是利用数组实现的小顶堆。而 ScheduledFutureTask 继承自 FutureTask 重写了 run 方法，实现了周期性任务的需求。

```java
/**
 * Overrides FutureTask version so as to reset/requeue if periodic.
 */
public void run() {
    boolean periodic = isPeriodic();
    if (!canRunInCurrentRunState(periodic))
        cancel(false);
    else if (!periodic) //如果不是周期性任务 run
        ScheduledFutureTask.super.run();
    else if (ScheduledFutureTask.super.runAndReset()) { //是周期性任务
        setNextRunTime(); //重设下一次执行任务时间
        reExecutePeriodic(outerTask); //任务重新入队
    }
}
```

小结一下

ScheduledThreadPoolExecutor 大致的流程和 Timer 差不多，也是**维护一个优先队列**，然后通过重写 task 的 run 方法来实现周期性任务，主要差别在于**能多线程运行任务，不会单线程阻塞**。

并且 Java 线程池的设定是 task 出错会把错误吃了，无声无息的。因此**一个任务出错也不会影响之后的任务**。

DelayQueue

Java 中还有个延迟队列 DelayQueue，加入延迟队列的元素都必须实现 Delayed 接口。延迟队列内部是利用 PriorityQueue 实现的，所以还是利用优先队列！Delayed 接口继承了Comparable 因此优先队列是通过 delay 来排序的。

```java
public interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
//元素必须实现 Delayed 接口
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
    implements BlockingQueue<E> { //同时也是个阻塞队列

    private final transient ReentrantLock lock = new ReentrantLock();
    private final PriorityQueue<E> q = new PriorityQueue<E>(); //优先队列，内部还是数组实现
```

然后我们再来看下延迟队列是如何获取元素的。

```java
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            E first = q.peek();
            if (first == null)
                available.await();
            else {
                long delay = first.getDelay(NANOSECONDS); //小于等于0说明到时间了
                if (delay <= 0)
                    return q.poll(); //出队
                first = null; // don't retain ref while waiting
                if (leader != null) //leader 是为了减少不必要的等待，没抢到当leader的线程都await
                    available.await();
                else {
                    Thread thisThread = Thread.currentThread();
                    leader = thisThread;
                    try {
                        available.awaitNanos(delay); //等待时间到达
                    } finally {
                        if (leader == thisThread) //执行完了重置leader
                            leader = null;
                    }
                }
            }
        }
    } finally {
        if (leader == null && q.peek() != null)
            available.signal();
        lock.unlock();
    }
}
```

小结一下

也是利用优先队列实现的，元素通过实现 Delayed 接口来返回延迟的时间。不过延迟队列就是个容器，需要其他线程来获取和执行任务。

这下是搞明白了 Timer 、ScheduledThreadPool 和 DelayQueue，总结的说下**它们都是通过优先队列来获取最早需要执行的任务**，因此插入和删除任务的时间复杂度都为O(logn)，并且 Timer 、ScheduledThreadPool 的周期性任务是通过重置任务的下一次执行时间来完成的。

问题就出在时间复杂度上，插入删除时间复杂度是O(logn)，那么假设频繁插入删除次数为 m，总的时间复杂度就是O(mlogn)，这种时间复杂度满足不了 Kafka 这类中间件对性能的要求，而时间轮算法的插入删除时间复杂度是O(1)。我们来看看时间轮算法是如何实现的。

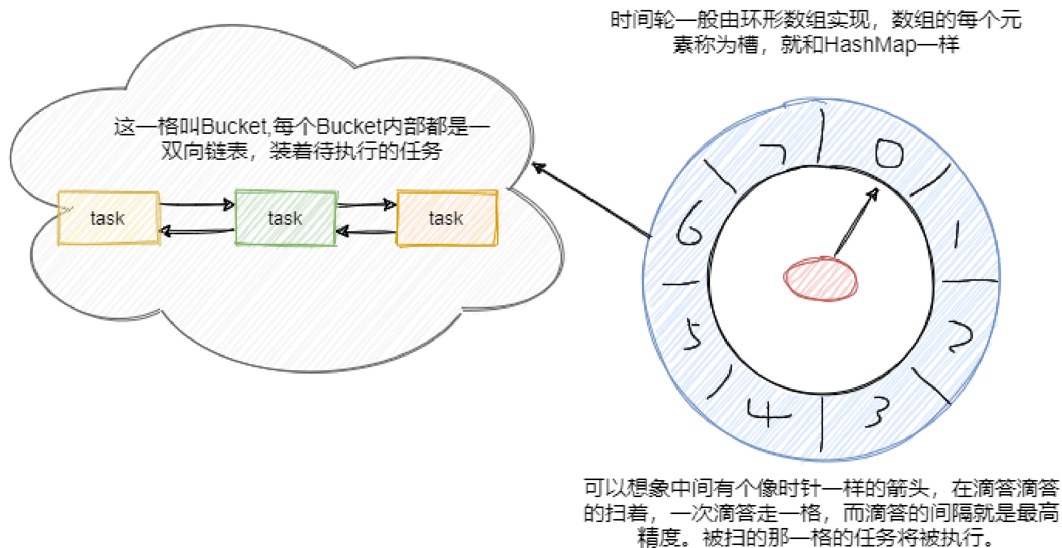
时间轮算法

俗话说艺术源于生活，技术也能从日常生活中找到灵感。咱们先来看块表，嗯金色的表。

都看清楚了吧，时间轮就是和手表时钟很相似的存在。时间轮用环形数组实现，数组的每个元素可以称为槽，和 HashMap一样称呼。

槽的内部用双向链表存着待执行的任务，添加和删除的链表操作时间复杂度都是 O(1)，槽位本身也指代时间精度，比如一秒扫一个槽，那么这个时间轮的最高精度就是 1 秒。

也就是说延迟 1.2 秒的任务和 1.5 秒的任务会被加入到同一个槽中，然后在 1 秒的时候遍历这个槽中的链表执行任务。



从图中可以看到此时指针指向的是第一个槽，一共有八个槽0~7，假设槽的时间单位为 1 秒，现在要加入一个延时 5 秒的任务，计算方式就是 5 % 8 + 1 = 6，即放在槽位为 6，下标为 5 的那个槽中。更具体的就是拼到槽的双向链表的尾部。

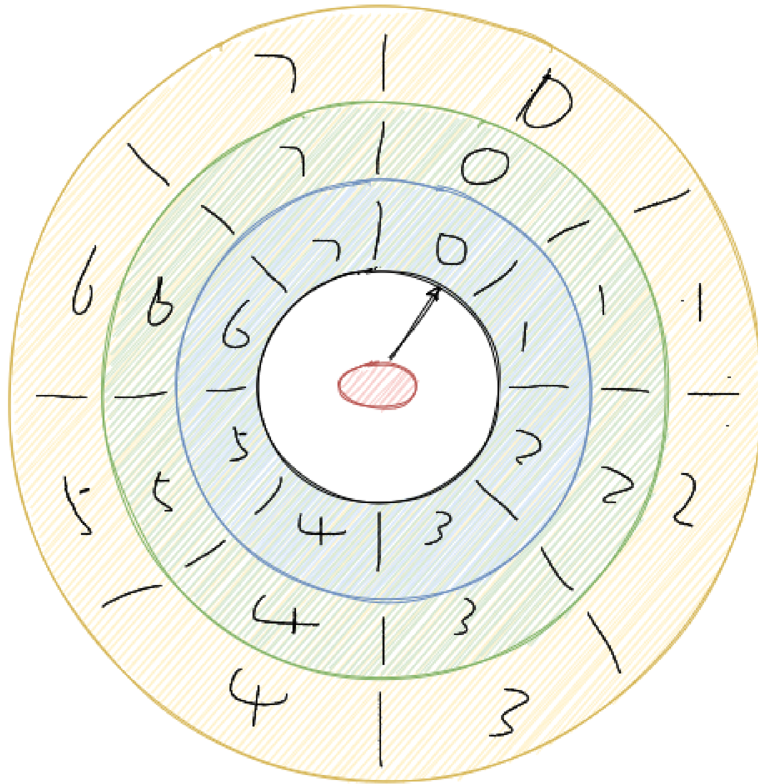
然后每秒指针顺时针移动一格，这样就扫到了下一格，遍历这格中的双向链表执行任务。然后再循环继续。

可以看到插入任务从计算槽位到插入链表，时间复杂度都是O(1)。那假设现在要加入一个50秒后执行的任务怎么办？这槽好像不够啊？难道要加槽嘛？和HashMap一样扩容？

不是的，常见有两种方式，**一种是通过增加轮次的概念**。50 % 8 + 1 = 3，即应该放在槽位是 3，下标是 2 的位置。然后 (50 - 1) / 8 = 6，即轮数记为 6。也就是说当循环 6 轮之后扫到下标的 2 的这个槽位会触发这个任务。Netty 中的 HashedWheelTimer 使用的就是这种方式。

还有一种是通过多层次的时间轮，这个和我们的手表就更像了，像我们秒针走一圈，分针走一格，分针走一圈，时针走一格。

多层次时间轮就是这样实现的。假设上图就是第一层，那么第一层走了一圈，第二层就走一格，可以得知第二层的一格就是8秒，假设第二层也是 8 个槽，那么第二层走一圈，第三层走一格，可以得知第三层一格就是 64 秒。那么一格三层，每层8个槽，一共 24 个槽时间轮就可以处理最多延迟 512 秒的任务。



而多层次时间轮还会有降级的操作，假设一个任务延迟 500 秒执行，那么刚开始加进来肯定是放在第三层的，当时间过了 436 秒后，此时还需要 64 秒就会触发任务的执行，而此时相对而言它就是个延迟 64 秒后的任务，因此它会被降低放在第二层中，第一层还放不下它。

再过个 56 秒，相对而言它就是个延迟 8 秒后执行的任务，因此它会再被降级放在第一层中，等待执行。

降级是为了保证时间精度一致性。Kafka内部用的就是多层次的时间轮算法。

Netty中的时间轮

在 Netty 中时间轮的实现类是 HashedWheelTimer，代码中的 wheel 就是上图画的循环数组，mask 的设计和HashMap一样，通过限制数组的大小为2的次方，利用位运算来替代取模运算，提高性能。tickDuration 就是每格的时间即精度。可以看到配备了一个工作线程来处理任务的执行。

```java
public HashedWheelTimer(
        ThreadFactory threadFactory,
        long tickDuration, TimeUnit unit, int ticksPerWheel, boolean leakDetection) {

    if (threadFactory == null) {
        throw new NullPointerException("threadFactory");
    }
    if (unit == null) {
        throw new NullPointerException("unit");
    }
    if (tickDuration <= 0) {
        throw new IllegalArgumentException("tickDuration must be greater than 0: " + tickDuration);
    }
    if (ticksPerWheel <= 0) {
        throw new IllegalArgumentException("ticksPerWheel must be greater than 0: " + ticksPerWheel);
    }

    // Normalize ticksPerWheel to power of two and initialize the wheel.
    wheel = createWheel(ticksPerWheel); //ticksPerWheel默认512,即槽的数量
//mask是为了位运算用的为了提高性能，限制wheel.length为2的次方，即 tick & (wheel.length - 1) = tick % wheel.length
    mask = wheel.length - 1;

    // Convert tickDuration to nanos.
    this.tickDuration = unit.toNanos(tickDuration); //就是每格的时间

    // Prevent overflow.
    if (this.tickDuration >= Long.MAX_VALUE / wheel.length) { //底层的代码就是这么的严谨。
        throw new IllegalArgumentException(String.format(
                "tickDuration: %d (expected: 0 < tickDuration in nanos < %d",
                tickDuration, Long.MAX_VALUE / wheel.length));
    }
    workerThread = threadFactory.newThread(worker); //一个工作线程

    leak = leakDetection || !workerThread.isDaemon() ? leakDetector.open(this) : null; //内存泄露探测
}
```

接下来我们再来看看任务是如何添加的。

```java
private final Queue<HashedWheelTimeout> timeouts = PlatformDependent.newMpscQueue();

public Timeout newTimeout(TimerTask task, long delay, TimeUnit unit) {
    if (task == null) {
        throw new NullPointerException("task");
    }
    if (unit == null) {
        throw new NullPointerException("unit");
    }
    start(); //可以看到第一次添加任务启动时间轮。这个方法会启动工作线程，然后工作线程其实会记录当前时间为startTime

    long deadline = System.nanoTime() + unit.toNanos(delay) - startTime; //算出延迟时间
    HashedWheelTimeout timeout = new HashedWheelTimeout(this, task, deadline); //创建任务。
    timeouts.add(timeout); //将任务添加到mpsc队列中
    return timeout;
}
```

可以看到任务并没有直接添加到时间轮中，而是先入了一个 mpsc 队列，我简单说下 mpsc 是 JCTools 中的并发队列，用在多个生产者可同时访问队列，但只有一个消费者会访问队列的情况。篇幅有限，有兴趣的朋友自行了解实现。

然后我们再来看看工作线程是如何运作的。

```java
do {
    final long deadline = waitForNextTick(); //等待执行任务的时间到来
    if (deadline > 0) { //如果来了
        int idx = (int) (tick & mask);
        processCancelledTasks(); //先处理那些被取消了的任务，即取消队列poll，然后移除任务
        HashedWheelBucket bucket =  //选择对应的槽
                wheel[idx];
        transferTimeoutsToBuckets(); //将之前加入mpsc队列的任务加入到时间轮的槽中
        bucket.expireTimeouts(deadline); //处理时间到了的任务
        tick++; //记一次移动
    }
} while (WORKER_STATE_UPDATER.get(HashedWheelTimer.this) == WORKER_STATE_STARTED);
```

很直观没什么花头，我们先来看看 waitForNextTick，是如何得到下一次执行时间的。

```java
private long waitForNextTick() {
        long deadline = tickDuration * (tick + 1); //计算下一次需要检查的时间

        for (;;) {
            final long currentTime = System.nanoTime() - startTime;
          //我认为这+999999是为了保证足够的sleep时间，比如deadline - currentTime计算5纳秒，
          //那5纳秒转毫秒不就0了，而实际上时间还没到呢。
            long sleepTimeMs = (deadline - currentTime + 999999) / 1000000;

            if (sleepTimeMs <= 0) { //说明不用睡了时间已经到了赶紧执行
                if (currentTime == Long.MIN_VALUE) { //可能是溢出了？
                    return -Long.MAX_VALUE;
                } else {
                    return currentTime;
                }
            }

            // 在window下有bug，sleep时间得是10整数倍，See https://github.com/netty/netty/issues/356
            if (PlatformDependent.isWindows()) {
                sleepTimeMs = sleepTimeMs / 10 * 10;
            }

            try {
                Thread.sleep(sleepTimeMs); //等待时间的到来
            } catch (InterruptedException ignored) {
                if (WORKER_STATE_UPDATER.get(HashedWheelTimer.this) == WORKER_STATE_SHUTDOWN) {
                    return Long.MIN_VALUE;
                }
            }
        }
    }
```

简单的说就是通过 tickDuration 和此时已经滴答的次数算出下一次需要检查的时间，时候未到就sleep等着。

再来看下任务如何入槽的。

```java
private void transferTimeoutsToBuckets() {
    // 一次最多搬运100000，得流出时间处理，怕延迟太多。
    for (int i = 0; i < 100000; i++) {
        HashedWheelTimeout timeout = timeouts.poll(); //从队列拿任务
        if (timeout == null) {
            break;
        }
        if (timeout.state() == HashedWheelTimeout.ST_CANCELLED) {
            continue;
        }

        long calculated = timeout.deadline / tickDuration;
        timeout.remainingRounds = (calculated - tick) / wheel.length; //计算排在第几轮
        // Ensure we don't schedule for past，让延迟的任务得以执行
        final long ticks = Math.max(calculated, tick);
        int stopIndex = (int) (ticks & mask); //计算放在那个槽中

        HashedWheelBucket bucket = wheel[stopIndex];
        bucket.addTimeout(timeout); //入槽，就是个链表入队操作
    }
}
```

注释的很清楚了，实现也和上述分析的一致。

最后再来看下如何执行的。

```java
public void expireTimeouts(long deadline) {
    HashedWheelTimeout timeout = head; //拿到槽的链表头部

    // process all timeouts
    while (timeout != null) {
        boolean remove = false;
        if (timeout.remainingRounds <= 0) { //如果到这轮了
            if (timeout.deadline <= deadline) { //并且时间到了
                timeout.expire(); //执行，实际就是调用任务的 run 方法
            } else { //应该不会走到这
                // The timeout was placed into a wrong slot. This should never happen.
                throw new IllegalStateException(String.format(
                        "timeout.deadline (%d) > deadline (%d)", timeout.deadline, deadline));
            }
            remove = true;
        } else if (timeout.isCancelled()) {
            remove = true;
        } else {
            timeout.remainingRounds --; //否则轮数减一
        }
        // store reference to next as we may null out timeout.next in the remove block.
        HashedWheelTimeout next = timeout.next; //保存下一个任务
        if (remove) {
            remove(timeout); //移除执行完的任务
        }
        timeout = next;//继续下一个
    }
}
```

就是通过轮数和时间双重判断，执行完了移除任务。

小结一下

总体上看 Netty 的实现就是上文说的时间轮通过轮数的实现，完全一致。可以看出时间精度由 TickDuration 把控，并且工作线程的除了处理执行到时的任务还做了其他操作，因此任务不一定会被精准的执行。

而且任务的执行如果不是新起一个线程，或者将任务扔到线程池执行，那么耗时的任务会阻塞下个任务的执行。

并且会有很多无用的 tick 推进，例如 TickDuration 为1秒，此时就一个延迟350秒的任务，那就是有349次无用的操作。

但是从另一面来看，如果任务都执行很快(当然你也可以异步执行)，并且任务数很多，通过分批执行，并且增删任务的时间复杂度都是O(1)来说。时间轮还是比通过优先队列实现的延时任务来的合适些。

Kafka 中的时间轮

上面我们说到 Kafka 中的时间轮是多层次时间轮实现，总的而言实现和上述说的思路一致。不过细节有些不同，并且做了点优化。

先看看添加任务的方法。在添加的时候就设置任务执行的绝对时间。

```
//TimerTaskEntry的就是包装了任务，并且记录的任务的执行时间，即延时+当前时间
new TimerTaskEntry(timerTask, timerTask.delayMs + Time.SYSTEM.hiResClockMs)

def add(timerTaskEntry: TimerTaskEntry): Boolean = {
   val expiration = timerTaskEntry.expirationMs

   if (timerTaskEntry.cancelled) {
     // Cancelled
     false
   } else if (expiration < currentTime + tickMs) { //如果已经到期，返回false
     // Already expired
     false
   } else if (expiration < currentTime + interval) { //如果在本层范围内
     // Put in its own bucket
     val virtualId = expiration / tickMs
     val bucket = buckets((virtualId % wheelSize.toLong).toInt) //计算槽位
     bucket.add(timerTaskEntry) //添加到槽内的双向链表中

     // Set the bucket expiration time
     if (bucket.setExpiration(virtualId * tickMs)) { //更新槽过期时间
       queue.offer(bucket) //将槽加入delayQueue，通过delayQueue来推进执行
     }
     true
   } else { //如果超过本层能表示的延迟时间则将任务添加到上层，这里可以看到上层是按需创建的
     // Out of the interval. Put it into the parent timer
     if (overflowWheel == null) addOverflowWheel()
     overflowWheel.add(timerTaskEntry)
   }
 }
```

那么时间轮是如何推动的呢？Netty 中是通过固定的时间间隔扫描，时候未到就等待来进行时间轮的推动。上面我们分析到这样会有空推进的情况。

而 Kafka 就利用了空间换时间的思想，通过 DelayQueue，来保存每个槽，通过每个槽的过期时间排序。这样拥有最早需要执行任务的槽会有优先获取。如果时候未到，那么 delayQueue.poll 就会阻塞着，这样就不会有空推进的情况发送。

我们来看下推进的方法。

```
def advanceClock(timeoutMs: Long): Boolean = {
  var bucket = delayQueue.poll(timeoutMs, TimeUnit.MILLISECONDS) //从延迟队列获取槽
  if (bucket != null) {
    writeLock.lock()
    try {
      while (bucket != null) {
        timingWheel.advanceClock(bucket.getExpiration()) //更新每层时间轮的currentTime
        bucket.flush(reinsert) //因为更新的currentTime，在进行一波任务重新插入，来实现任务的时间轮降级
        bucket = delayQueue.poll() //获取下一个槽
      }
    } finally {
      writeLock.unlock()
    }
    true
  } else {
    false
  }
}
// #timingWheel.advanceClock
def advanceClock(timeMs: Long): Unit = {
  if (timeMs >= currentTime + tickMs) {
    currentTime = timeMs - (timeMs % tickMs) //更新currentTime,做了微调，必须是tickMs的整数倍

    // Try to advance the clock of the overflow wheel if present
    if (overflowWheel != null) overflowWheel.advanceClock(currentTime) //推进上层时间轮也更新currentTime
  }
}
```

从上面的 add 方法我们知道每次对比都是根据 `expiration < currentTime + interval` 来进行对比的，而 `advanceClock` 就是用来推进更新 currentTime 的。

小结一下

Kafka 用了多层次时间轮来实现，并且是按需创建时间轮，采用任务的绝对时间来判断延期，并且对于每个槽(槽内存放的也是任务的双向链表)都会维护一个过期时间，利用 DelayQueue 来对每个槽的过期时间排序，来进行时间的推进，防止空推进的存在。

每次推进都会更新 currentTime 为当前时间戳，当然做了点微调使得 currentTime 是 tickMs 的整数倍。并且每次推进都会把能降级的任务重新插入降级。

可以看到这里的 DelayQueue 的元素是每个槽，而不是任务，因此数量就少很多了，这应该是权衡了对于槽操作的延时队列的时间复杂度与空推进的影响。

总结

首先介绍了 Timer、DelayQueue 和 ScheduledThreadPool，它们都是基于优先队列实现的，O(logn) 的时间复杂度在任务数多的情况下频繁的入队出队对性能来说有损耗。因此适合于任务数不多的情况。

Timer 是单线程的会有阻塞的风险，并且对异常没有做处理，一个任务出错 Timer 就挂了。而 ScheduledThreadPool 相比于 Timer 首先可以多线程来执行任务，并且线程池对异常做了处理，使得任务之间不会有影响。

并且 Timer 和 ScheduledThreadPool 可以周期性执行任务。 而 DelayQueue 就是个具有优先级的阻塞队列。

对比而言时间轮更适合任务数很大的延时场景，它的任务插入和删除时间复杂度都为O(1)。对于延迟超过时间轮所能表示的范围有两种处理方式，一是通过增加一个字段-轮数，Netty 就是这样实现的。二是多层次时间轮，Kakfa 是这样实现的。

相比而言 Netty 的实现会有空推进的问题，而 Kafka 采用 DelayQueue 以槽为单位，利用空间换时间的思想解决了空推进的问题。

可以看出延迟任务的实现都不是很精确的，并且或多或少都会有阻塞的情况，即使你异步执行，线程不够的情况下还是会阻塞。

巨人的肩膀

《深入理解Kafka:核心设计与实践原理》
https://www.cnblogs.com/luozhiyun/p/12075326.html

未完待续~

哈哈，0.1版本暂时就更新到这里啦~过段时间会再进行纠错和补充。

这个系列一共近 6W 字+ 151张图，全部是我个人手打原创的，希望对你有所帮助。

也欢迎关注我的个人公众号「**yes的练级攻略**」，我会持续输出更多文章哒！如有错误也可以通过公众号联系我~

按照开源 PDF 的惯例，文末都会放个赞赏码，我也来放个，如果可以的话给 yes 午饭加个鸡腿吧，金额不重要，主要是支持的心意，我先谢过啦，哈哈。

推荐使用支付宝

yes(**仕)

打开支付宝[扫一扫]

申请官方收钱码：拨打 95188-6