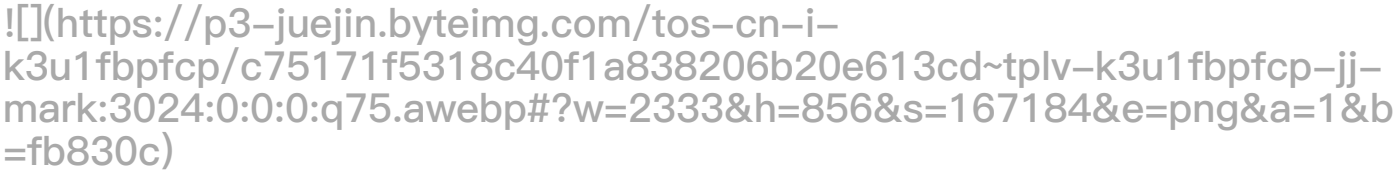
我们在顺序消息和事务消息方面的实践

导读

最近团队内部在RocketMQ的业务实践上有一些心得，想给大家分享一下，首先转转这边是有架构团队自研的ZZMQ的，所以我们自然而然的用的ZZMQ，考虑到受众人群，开篇会先讲开源版本的一些基础知识，然后从顺序消息和事务消息2个炽手可热的话题上逐渐转入到与ZZMQ的比较，希望可以帮助到大家绕过”坑”。

第一部分: 基本介绍

1. 领域模型概述



1.1 消息生产

生产者（Producer）：

Apache RocketMQ 中用于产生消息的运行实体，一般集成于业务调用链路的上游。生产者是轻量级匿名无身份的。

1.2 消息存储

* 主题（Topic）：
+ Apache RocketMQ 消息传输和存储的分组容器，主题内部由多个队列组成，消息的存储和水平扩展实际是通过主题内的队列实现的。
* 队列（MessageQueue）：
+ Apache RocketMQ 消息传输和存储的实际单元容器，类比于其他消息队列

中的分区。 Apache RocketMQ 通过流式特性的无限队列结构来存储消息，消息在队列内具备顺序性存储特征。
* 消息（Message）：
+ Apache RocketMQ 的最小传输单元。消息具备不可变性，在初始化发送和完成存储后即不可变。

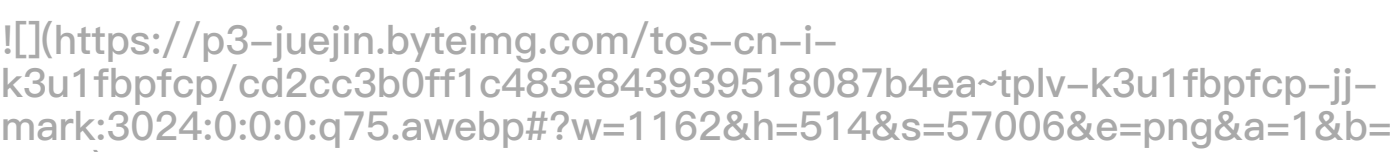
1.3 消息消费

* 消费者分组（ConsumerGroup）：
+ Apache RocketMQ 发布订阅模型中定义的独立的消费身份分组，用于统一管理底层运行的多个消费者（Consumer）。同一个消费组的多个消费者必须保持消费逻辑和配置一致，共同分担该消费组订阅的消息，实现消费能力的水平扩展。
* 消费者（Consumer）：
+ Apache RocketMQ 消费消息的运行实体，一般集成在业务调用链路的下游。消费者必须被指定到某一个消费组中。
* 订阅关系（Subscription）：
+ Apache RocketMQ 发布订阅模型中消息过滤、重试、消费进度的规则配置。订阅关系以消费组粒度进行管理，消费组通过定义订阅关系控制指定消费组下的消费者如何实现消息过滤、消费重试及消费进度恢复等。
+ Apache RocketMQ 的订阅关系除过滤表达式之外都是持久化的，即服务端重启或请求断开，订阅关系依然保留。

2. 消息传输模型介绍

主流的消息中间件的传输模型主要为点对点模型和发布订阅模型。

点对点模型

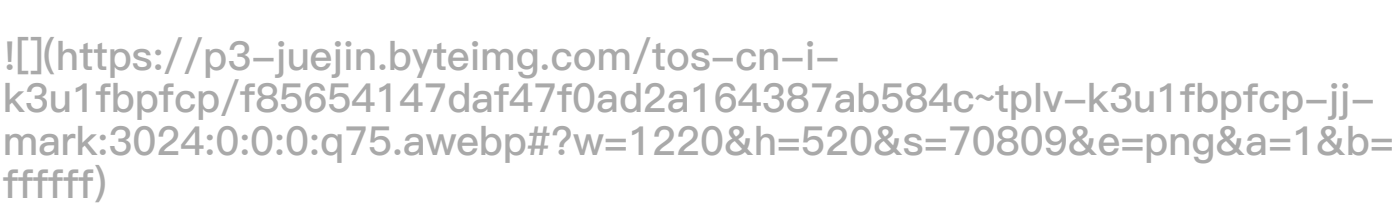


点对点模型也叫队列模型，具有如下特点：

* 消费匿名：消息上下游沟通的唯一的身份就是队列，下游消费者从队列获取消息无法申明独立身份。
* 一对一通信：基于消费匿名特点，下游消费者即使有多个，但都没有自己独立的身份，因此共享队列中的消息，每一条消息都只会被唯一一个消费者处理。

因此点对点模型只能实现一对一通信。

发布订阅模型



发布订阅模型具有如下特点：

* 消费独立：相比队列模型的匿名消费方式，发布订阅模型中消费方都会具备的身份，一般叫做订阅组（订阅关系），不同订阅组之间相互独立不会相互影响。
* 一对多通信：基于独立身份的设计，同一个主题内的消息可以被多个订阅组处理，每个订阅组都可以拿到全量消息。因此发布订阅模型可以实现一对多通信。

传输模型对比

点对点模型和发布订阅模型各有优势，点对点模型更为简单，而发布订阅模型的扩展性更高。 Apache RocketMQ 使用的传输模型为发布订阅模型，因此也具有发布订阅模型的特点。

注：以上信息来源于官网

3. 普通消息的可靠性

普通消息一般应用于微服务解耦、事件驱动、数据集成等场景，这些场景大多数要求数据传输通道具有可靠传输的能力，且对消息的处理时机、处理顺序没有特别要求。

3.1 发送端怎么保证可靠性

3.1.1 ACK机制

3.2 存储端怎么保证消息可靠性

RocketMQ存储端也即Broker端在存储消息的时候会面临以下的存储可靠性挑战：

1. Broker正常关闭
2. Broker异常Crash
3. OS Crash
4. 机器掉电，但是能立即恢复供电情况
5. 机器无法开机（可能是cpu、主板、内存等关键设备损坏）
6. 磁盘设备损坏

1正常关闭，Broker可以正常启动并恢复所有数据。2、3、4同步刷盘可以保证数据不丢失，异步刷盘可能导致少量数据丢失。5、6属于单点故障，且无法恢复。解决单点故障可以采用增加Slave节点，主从异步复制仍然可能有极少量数据丢失，同步复制可以完全避免单点问题。

这里一般来说就需要在性能和可靠性之间做出取舍，对于RocketMQ来说，Broker的可靠性主要由两个方面保障：

* 单机的刷盘机制
* 主从同步

3.2.1 单机的刷盘机制

页缓存：操作系统中用于存储文件系统缓存的内存区域。RocketMQ通过将消息首先写入页缓存，实现了消息在内存中的持久化。

CommitLog：是RocketMQ中消息的物理存储结构，包含了所有已发送的消息。CommitLog的持久化保证了即使在异常情况下，如Broker宕机，消息也能够

被恢复。
同步刷盘：是指将内存中的数据同步刷写到磁盘。RocketMQ确保消息在被发送后，首先在内存中得到持久化，然后再刷写到磁盘，从而防止数据的丢失。

异步刷盘：消息写入到页缓存中，就立刻给客户端返回写操作成功，当页缓存中的消息积累到一定的量时，触发一次写操作，或者定时等策略将页缓存中的消息写入到磁盘中。这种方式吞吐量大，性能高，但是页缓存中的数据可能丢失，不能保证数据绝对的安全。

实际应用中要结合业务场景，合理设置刷盘方式，尤其是同步刷盘的方式，由于频繁的触发磁盘写动作，会明显降低性能。

3.2.2 主从同步

主 Broker： 负责消息的读写和写入 CommitLog。
从 Broker： 用于备份主 Broker 的消息，确保在主 Broker 故障时可以顺利切换。
同步复制： 主节点将消息同步复制到所有从节点，确保从节点具有相同的消息副本。
切换： 在主节点发生故障时，从节点可以快速切换为新的主节点，确保消息服务的持续性。

3.3 消费端怎么保证可靠性

3.3.1 ACK

Rocket Mq是通过offset来标记一个消费者组在队列上的消费进度，消费成功之后都会返回一个ACK消息告诉broker去更新offset，但是RocketMQ并不是每消费一条消息就做一次ACK，而是消费完批量消息后只做一次ACK。

所以ACK机制是为了准确的告知Broker批量消费成功的信息并且更新消费进度。

那批量消费时具体是如何更新消费进度?

* 每一条消息消费成功后，会按照当前消息最小的offset来更新本地的消费进度
* 由5秒的定时任务将offset提交到Broker

k3u1fbpfcp/73e8812cbb19444aaaefaf18eee1a5c3~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1280&h=522&s=94201&e=png&b=fdfcfc)

优点：防止消息丢失。

缺点：会造成消息重复消费（使用方需要做幂等。

3.3.2 重试

消费者从RocketMQ拉取到消息之后，需要返回消费成功来表示业务方正常消费完成。因此只有返回CONSUME_SUCCESS才算消费完成，如果返回CONSUME_LATER 则会按照【重试次数】进行再次消费，【重试间隔为阶梯时间】。如果消费满16次之后还是未能消费成功，则不再重试，会将消息发送到死信队列，从而保证【消息消费】的可靠性。

3.3.3 死信消息

默认最多重试16次，总时长4小时46分钟。

未能成功消费的消息，消息队列并不会立刻将消息丢弃，而是将消息发送到死信队列，其名称是在【消费组】前加 %DLQ% 的【特殊 topic】，如果消息最终进入了死信队列，则可以通过RocketMQ提供的相关接口从死信队列获取到相应的消息，进行报警人工干预或其他手段，保证了消费组的至少一次消费。

小节
--

至此应该清晰的知道RocketMq为了保证可靠性做了哪些工作。接下来我们再把视角切换到今天的第一个核心问题**顺序消息**。

第二部分: 顺序消息
----

1. 我们遇到的线上问题

```
 客服同学: @XXX 我们判责了，为啥客户还收不到退款？售后单号xxxxxx。

 研发同学: 让我看看。
```

.....

30分钟后

研发同学: 修复了，再看一下。

测试同学: 提个online， 研发填一下问题原因，责任归属。

研发同学:问题原因: 历史代码，我们没有顺序消费消息，正常的流程是 先判责完成，再打款. 这一单 先消费了打款消息， 还没有消费判责完成消息，状态不对导致打款失败。

测试同学:那后续如何修改啊。

研发同学: 为了保证消息的有序性，我等会把消息修改为顺序消费。

.....

以上故事纯属于虚构

2. RocketMQ消息队列为什么会有顺序问题？

从上面的消息队列模型我们知道，1个topic有N个queue，将数据均匀分配到各个queue上，这样可以提升消费端总体的消费性能。比如一个topic发送10条消息，这10条消息会自动分散在topic下的所有queue中，所以消费的时候不一定是先消费哪个queue，后消费哪个queue，这就导致了无序消费。

3. 顺序消息的使用场景

* 金融交易、订单流程处理。 比如我们的暗拍场景下，相同出价，先出价商户优先成单，比如我们的订单的发生售后时，售后订单的处理流程。

* 实时同步数据的场景，如数据库增量同步，顺序消息也可以发挥其作用。通过使用顺序消息，可以确保数据按照正确的顺序进行同步，从而保持数据的一致性和准确性。

4. 实际开发过程中如何保证消息的顺序性?

4.1 生产顺序性

* 多生产者单线程
+ 消息生产的顺序性仅支持单一生产者，如果不同生产者分布在不同的系统，那么不同生产者之间产生的消息，我们无法知道消息之间实际的先后顺序。
* 单生产者多线程
+ 相同业务的消息按照先后顺序被存储在同一个队列。
+ 不同业务的消息可以混合在同一个队列中，且不保证连续。
+ 如果生产者使用多个线程进行并行发送，那么不同线程间产生的消息，我们无法知道消息之间实际的先后顺序。

4.2 消费者顺序性

消费消息时需要严格按照接收—处理—应答的顺序处理消息，避免使用异步回调或多线程处理，这样可以防止消息处理过程中的并发问题。对于每条消息，只有当它完全处理完毕并发送应答后，才继续处理下一条消息。

5. 使用顺序消息需要注意的点

* 消费消息时异常如何处理
+ 如果发生异常，需要消费方进行处理，顺序消费默认是 无限重试消费的 ，

无限重试会导致当前消息队列阻塞，影响后续消息消费。
+ 需要我们在重试一定次数后进行处理，即一条消息如果一直重试失败，超过最大重试次数后将不再重试，跳过这条消息消费 并 监控和告警，人工介入处理，不能一直阻塞后续消息处理。
+ 比如我们业务售后流程中，某个节点的售后单消息消费异常，我们目前解决方案是： 把重试3次仍然失败的消息存储到数据库中，同时把同一售后单后续消息也先存入数据库中，同时发出告警，后续人工介入处理后重新消费该异常售后单的消息。保证不影响同一队列下其它售后单消息消费。
* 消息组尽可能打散，避免集中导致热点
+ Apache RocketMQ 保证相同消息组的消息存储在同一个队列中，如果不同业务场景的消息都集中在少量或一个消息组中，则这些消息存储压力都会集中到服务端的少量队列或一个队列中。容易导致性能热点，为了提高系统的吞吐量和稳定性，避免因为某些消息组过于集中而导致资源瓶颈或性能下降。
+ 在设计消息键时，应尽量避免让消息集中到少数几个MessageQueue中。可以考虑将业务相关的多个字段组合成消息键，比如 订单ID、用户ID作为消息键，或者使用哈希算法来生成消息键，以增加消息分发的随机性和均匀性。可实现同一用户、同一订单的消息按照顺序处理，不同用户或者不同订单的消息无需保证顺序。

小节
--

顺序消息是一个老生常谈的问题，但是简单粗暴的硬搬网上的解决方案往往效果不尽如意，实际想要解决的彻底的确不是那么容易，希望可以带给各位看官一些思考和帮助。我们再把视角切换到**事务消息**身上去。

第三部分: 事务消息
----

1. 我们遇到的线上问题

客服同学: @XXX 用户在我们的app上一直获取不到报价，比较急，麻烦看一下怎么回事。

```
 产品同学: 好的，收到，我这边马上找研发同学看一下，@XXX 需要帮忙看一下。

 研发同学：好的，我看一下。
```

.....

30分钟后

研发同学: 修复了，再看一下。

产品同学: 把问题原因同步一下吧。

研发同学:询价过程中，风控命中了特殊报价池，我们这边把特殊报价池的数据存到了数据库，同时发送了MQ消息，结果MQ消息发送成功了，但是数据库存储失败，导致再次询价的时候，查不到数据导致的。

测试同学:那后续如何修改啊。

研发同学: 后续我们会把普通消息改成事务消息，这样就能保证消息发送和数据库存储的一致性了。

.....

以上故事纯属于虚构

2. 为什么使用了消息队列反而不可控呢?

MQ消息本身就具有解耦性，消息本身并不接收方的状态是否符合预期，只要消息成功发送并且被成功接收，在MQ本身看来就是成功，如果想要保证发送方和接受方的状态变更符合预期，就要保证本次事务操作和消息发送的一致性，这里我们就必须要提到事务消息。

所谓事务消息，其实是为了解决上下游写一致性，也即是完成当前操作的同时给下游发送指令，并且保证上下游要么同时成功或者同时失败。

3. 事务消息的使用场景

* 经典场景

+ 支付发起后，当笔订单处于中间状态，给支付网关发起指令，如果发起转账失败则不发送指令，发送成功后等待支付网关反馈更新支付状态。如果在同一个数据库中进行，事务可以保证这两步操作，要么同时成功，要么同时不成功。这样就保证了转账的数据一致性。但是在微服务架构中，因为各个服务都是独立的模块，都是远程调用，都没法在同一个事务中，都会遇到分布式事务问题。
* 我方场景
+ 在我们售后判责(用户与卖家发生了纠纷)的过程中，如果用户对于判责结果不满意，可以进行复检申诉，我方需要对复检申诉进行改判或者维持原判，如果需要改判，我方需要将改判结果进行保存，同时，对于我们下游的行星售后等系统进行发送消息，之所以发送消息而不是直接调用，是因为消息的接收方不止一个，如果全部是RPD调用的话，代码的侵入性太强，但是消息有很强的解耦性，并不能保证上下游状态的一致性，这个时候，事务消息就很符合这个场景，如果我们本地事务提交成功，就发送事务消息，下游同步修改如果我们本地事务失败，就不在发送消息，从而保持本地事务与消息的一致性。

4. 为什么需要引入分布式事务消息

* MQ本身就具备了实现了系统之间的解耦特性。
* 分布式事务保障本地事务和消息发送的原子性。
* 具备以上特性的同时还可以保证最终的数据一致性。

5. 开源版本事务消息

5.1 基本实现原理

基于MQ的事务消息方案主要依靠MQ的Half消息机制来实现投递消息和参与者自身本地事务的一致性保障。

Half消息：在原有队列消息执行后的逻辑，如果后面的本地逻辑出错，则不发送该消息，如果通过则告知MQ发送。Half消息机制实现原理其实借鉴的2PC的思路，是二阶段提交的广义拓展。

(https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/550111d7b2f94cea8092819fe68a1761~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1961&h=729&s=71832&e=jpg&b=fefefe)

* 事务发起方producer首先发送Half消息到broker
* MQ通知发送方消息发送成功
* 在发送Half消息成功后producer执行本地事务
* 本地事务完毕，根据事务的状态，Producer向Broker发送二次确认消息，确

认该Half Message的Commit或者Rollback状态。Broker收到二次确认消息后，对于Commit状态，则直接发送到Consumer端执行消费逻辑，而对于Rollback则直接标记为失败，一段时间后清除，并不会发给Consumer。正常情况下，到此分布式事务已经完成，剩下要处理的就是超时问题，即一段时间后Broker仍没有收到Producer的二次确认消息；
* 针对超时状态，Broker主动向Producer发起消息回查；
* Producer处理回查消息，返回对应的本地事务的执行结果；
* Broker针对回查消息的结果，执行Commit或Rollback操作，同4；

事务消息共有三种状态，提交状态、回滚状态、中间状态：

CommitTransaction: 提交事务，它允许消费者消费此消息。
RollbackTransaction: 回滚事务，它代表该消息将被删除，不允许被消费。
Unknown: 中间状态，它代表需要检查消息队列来确定状态。
事务消息的核心类为TransactionListenerImpl，里面提供了两个核心方法，具体的代码如下图：

executeLocalTransaction方法：用来执行本地事务，返回本地事务给到broker，同时，将事务状态进行记录：

checkLocalTransaction 方法：用来查询本地事务的执行结果提供给broker

5.2 事务消息发送逻辑—producer发送

事务消息是由两个消息来实现的，一个是RMQ_SYS_TRANS_HALF_TOPIC消息，作用是用来存储第一阶段的parpare消息，事务消息首先先进入到该主题消息，消息具体是提交还是回滚要根据第二阶段的消息来判断。另一个是RMQ_SYS_TRANS_OP_HALF_TOPIC消息，用来接收第二阶段的Commit或Rollback消息。

特别需要注意的一点，RMQ_SYS_TRANS_HALF_TOPIC消息是用来存储不能被消费者发现的消息，通过RMQ_SYS_TRANS_OP_HALF_TOPIC消息，来对RMQ_SYS_TRANS_HALF_TOPIC消息对应的事务状态来进行确认的，确认commit之后，需要将一阶段中设置的特殊Topic和Queue替换成真正的目标的Topic和Queue，后通过一次普通消息的写入操作来生成一条对用户可见的消息。所以RocketMQ事务消息二阶段其实是利用了一阶段存储的消息的内容，在二阶段时恢复出一条完整的普通消息。

5.3 事务消息发送逻辑--broker回查

如果在RocketMQ事务消息的二阶段过程中失败了，例如在做Commit操作时，出现网络问题导致Commit失败，那么需要通过一定的策略使这条消息最终被Commit。RocketMQ采用了一种补偿机制，称为“回查”。

Broker端对未确定状态的消息发起回查，将消息发送到对应的Producer端（同一个Group的Producer），由Producer根据消息来检查本地事务的状态，进而执行Commit或者Rollback。Broker端通过对比Half消息和Op消息进行事务消息的回查并且推进CheckPoint（记录那些事务消息的状态是确定的）。

需要注意的是，RocketMQ并不会无休止的的信息事务状态回查，默认回查15次，如果15次回查还是无法得知事务状态，RocketMQ默认回滚该消息。

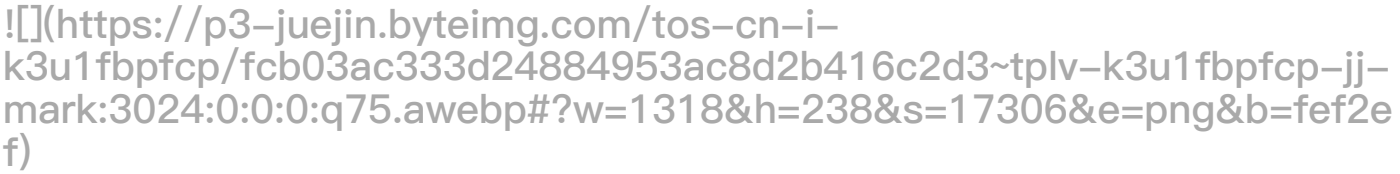
6. 转转版本事务消息

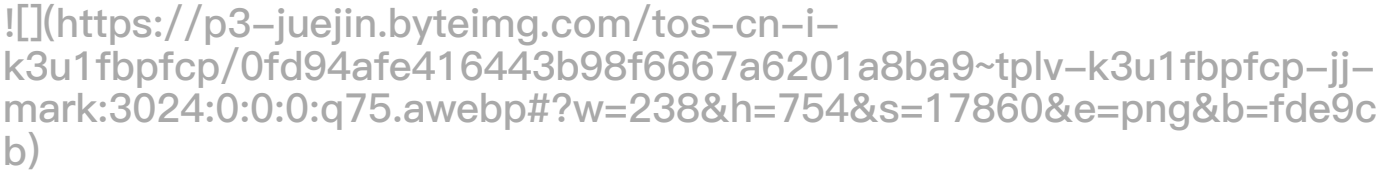
6.1 差异

* 设计方式的不同
+ 转转版本
– 基于公司业务场景发展，在**水平方向**做了拓展，对RocketMQ统一做了一层封装（此时开源版本并没有事务消息），方便使用，我们的设计思想可以迁移到**任何不支持事务消息**的MQ中，没有额外依赖，便于拓展，使得事务消息不在局限于RocketMQ本身。
+ 开源版本
– 基于MQ消息本身立场，在垂直方向做了拓展，在RocketMQ的服务里面，直接嵌入了事务消息，相当于把这种能力重新下沉到MQ中，便于使用，不用在做任何的额外工作。
* 实现方式的不同
+ 转转版本
– 由数据库的本地事务来保证事务的原子性，并由重试机制保证消息发送的可靠性。相当于把业务系统和消息队列的分布式事务重新“降级”为数据库中的本地事务。
+ 开源版本
– 开源版本是通过内部队列和状态回查实现了事务的最终一致性。

6.2 基本实现原理

* 事务消息的发送流程，在事务过程中，会将事务信息消息记录到数据库中。


* 获取到msg之后，执行校验逻辑，在发送失败或者未查询到数据后，会将这个msg丢入到另一个队列timeWheelQueue ， 由另一个定时任务去处理。具体的流程如下图。

* 因为涉及到jvm的内存存储，所以要考虑上线或者其他情况导致服务重启，未发送完的消息该如何处理。


小节

事务消息无论是开源版本还是转转版本，都是绕不过去的点，因为我们作为业务侧团队在如今遍地都是分布式系统的情况下太需要这样的能力来帮我们兜底了。而作为各位看官，为了能够正确得使用事务消息以及方便排查这里的问题，也是很有必要了解清楚这里的技术实现细节。

作者

黄培祖 转转采货侠后端工程师

朱洪旭 转转采货侠后端工程师

团队

我们是隶属于转转C2B2C循环经济体中的B2B技术团队，我们叫”采货侠-根”，致力于打造高标准化高专业度的B2B二手电商SaaS平台

`> 转转研发中心及业界小伙伴们的技术学习交流平台，定期分享一线的实战经验及业界前沿的技术话题。`

`> 公众号「转转技术」（综合性）、「大转转FE」（专注于FE）、「转转QA」（专注于QA），更多干货实践，欢迎交流分享~`

原文链接: https://juejin.cn/post/7361612219216232489