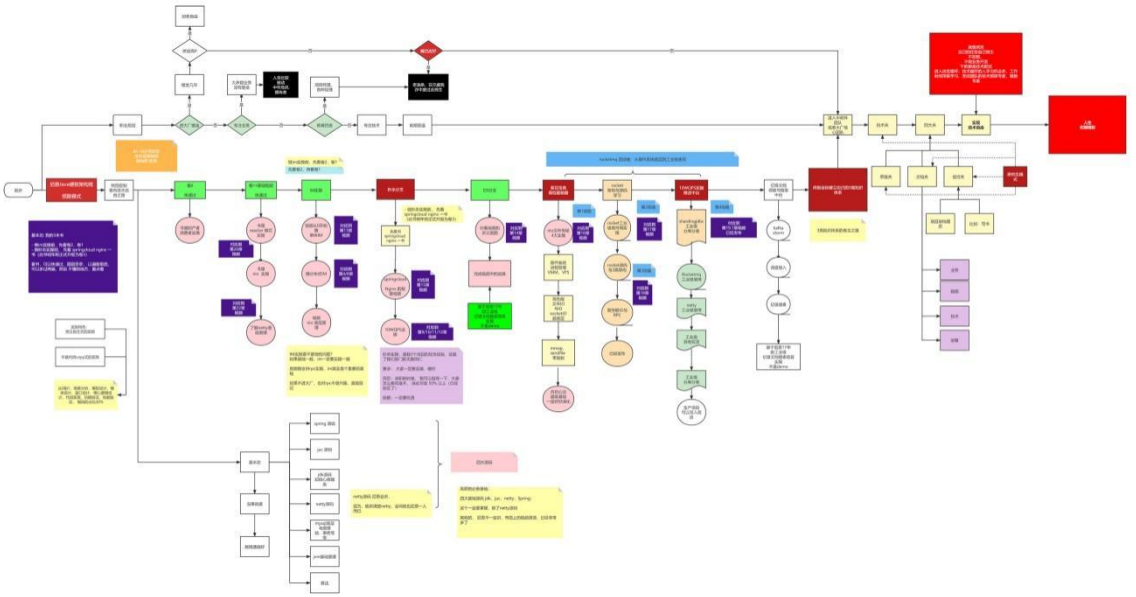
牛逼的职业发展之路

40 岁老架构尼恩用一张图揭秘：Java 工程师的高端职业发展路径，走向食物链顶端的之路

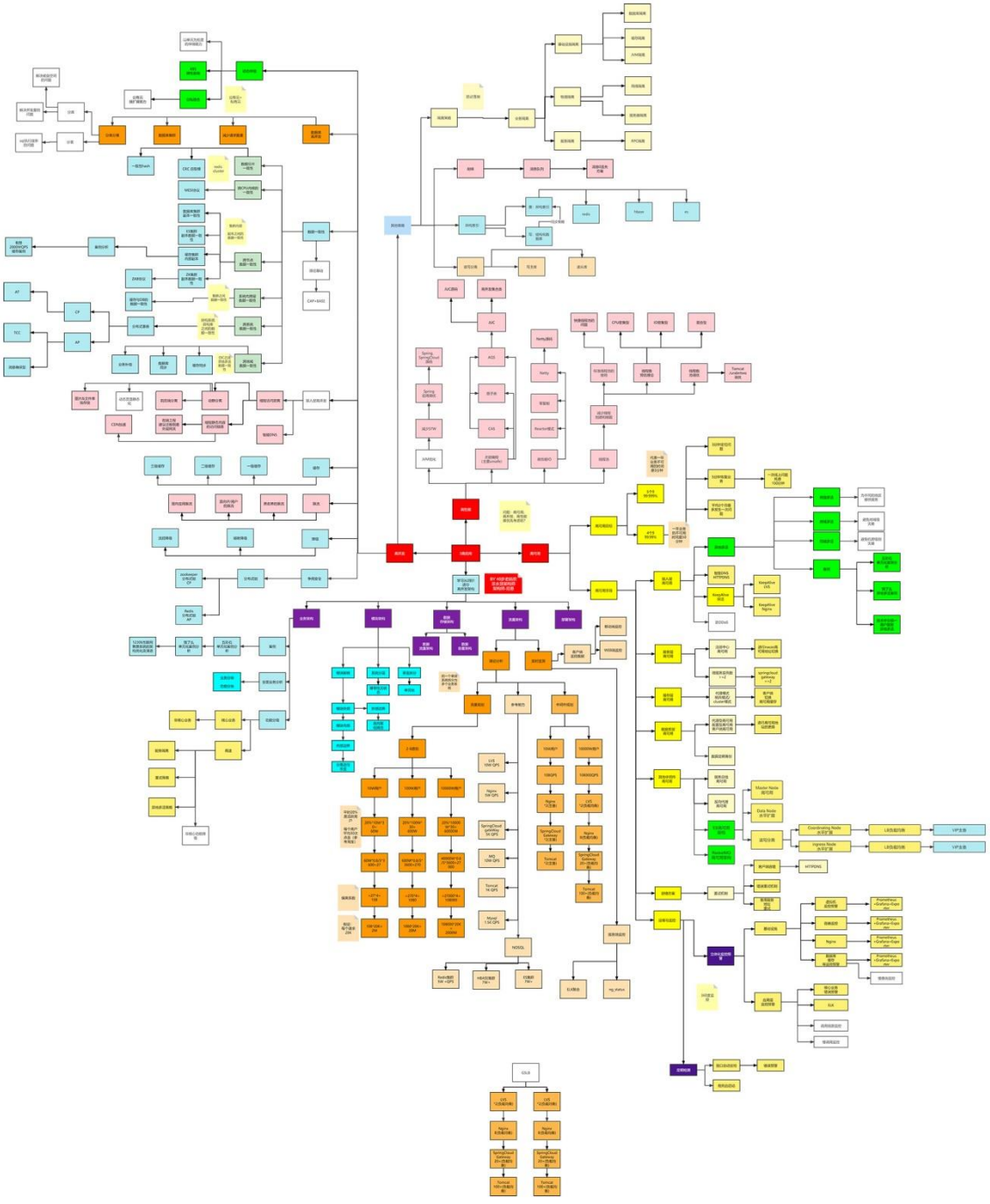
链接：https://www.processon.com/view/link/618a2b62e0b34d73f7eb3cd7



史上最全：价值10W的架构师知识图谱

此图梳理于尼恩的多个 3 高生产项目：多个亿级人民币的大型 SAAS 平台和智慧城市项目

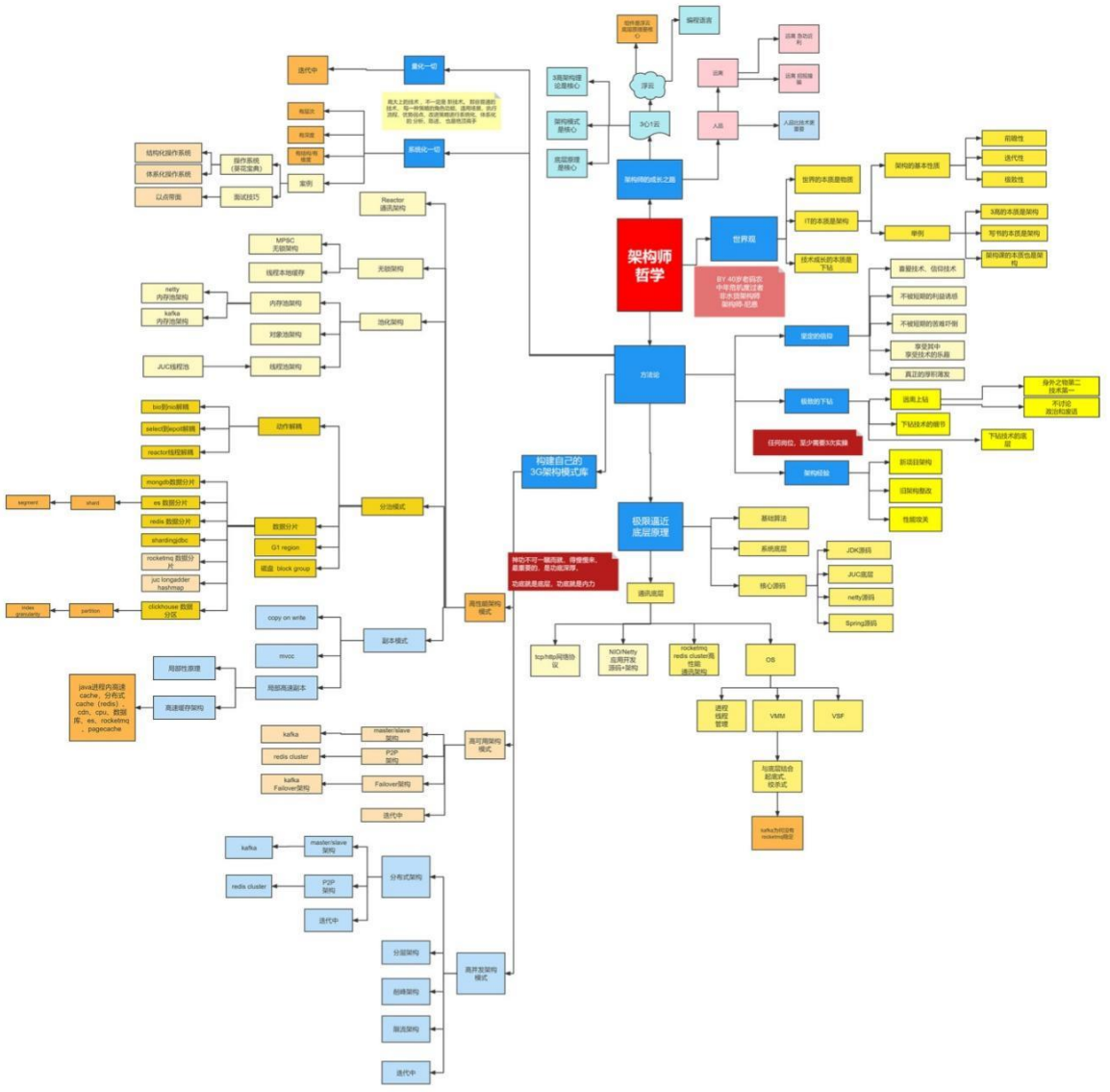
链接：https://www.processon.com/view/link/60fb94216376897l9d246739



牛逼的架构师哲学

40 岁老架构师尼恩对自己的 20 年的开发、架构经验总结

链接：https://www.processon.com/view/link/616f801963768961e9d9aec8



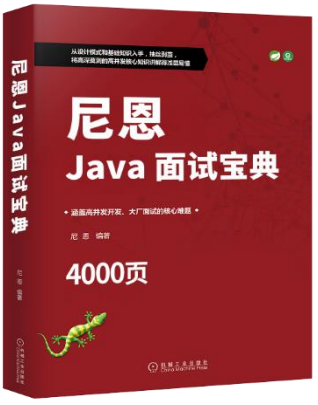
牛逼的3高架构知识宇宙

尼恩 3 高架构知识宇宙，帮助大家穿透 3 高架构，走向技术自由，远离中年危机

链接：https://www.processon.com/view/link/635097d2e0b34d40be778ab4

尼恩Java面试宝典

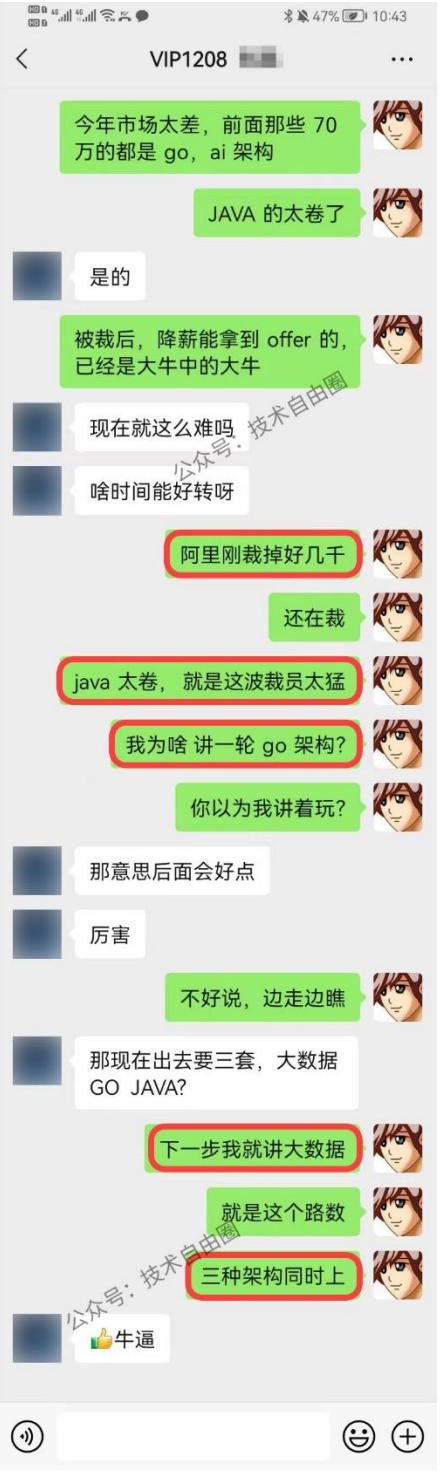
40 个专题（卷王专供+ 史上最全 + 2023 面试必备）
详情：https://www.cnblogs.com/crazymakercircle/p/13917138.html



名称

专题01：JVM面试题（卷王专供 + 史上最全 + 2022面试必备）-V81-from-尼恩Java面试宝典.pdf
专题02：Java算法面试题（卷王专供 + 史上最全 + 2022面试必备）-V80-from-Java面试红宝书.pdf
专题03：Java基础面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书 -release.pdf
专题04：架构设计面试题（卷王专供+ 史上最全 + 2023面试必备）-V86-from-尼恩Java面试宝典.pdf
专题05：Spring面试题_专题06：SpringMVC_专题07：Tomcat面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩面试宝典-release.pdf
专题08：SpringBoot面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题09：网络协议面试题（卷王专供+ 史上最全 + 2023面试必备）-V46-from-尼恩Java面试宝典-release.pdf
专题10：TCPIP协议（卷王专供+ 史上最全 + 2022面试必备）-V57-from-Java面试红宝书.pdf
专题11：JUC并发包与容器类（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题12：设计模式面试题（卷王专供+ 史上最全 + 2022面试必备）-V84-from-Java面试红宝书.pdf
专题13：死锁面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题14：Redis 面试题 （卷王专供+ 史上最全 + 2022面试必备）-V65-from-Java面试红宝书.pdf
专题15：分布式锁 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题16：Zookeeper 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题17：分布式事务面试题（卷王专供 + 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题18：一致性协议 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题19：Zab协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题20：Paxos 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题21：raft 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题22：Linux面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题23：Mysql 面试题（卷王专供+ 史上最全 + 2023面试必备）-V82-from-尼恩Java面试宝典.pdf
专题24：SpringCloud 面试题（卷王专供+ 史上最全 + 2023面试必备）-V12-from-Java面试红宝书-release.pdf
专题25：Netty 面试题 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题26：消息队列面试题：RabbitMQ、Kafka、RocketMQ（卷王专供+ 史上最全 + 2023面试必备）-V10-from-Java面试红宝书-release.pdf
专题27：内存泄漏 内存溢出（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题28：JVM 内存溢出 实战（卷王专供+ 史上最全 + 2023面试必备）-V17-from-Java面试红宝书-release.pdf
专题29：多线程面试题（卷王专供+ 史上最全 + 2023面试必备）-V66-from-Java面试红宝书.pdf
专题30：HR面试题：过五关斩六将后，小心阴沟翻船！（史上最全、避坑宝典）-V2-from-Java面试红宝书-release.pdf
专题31：Hash连环炮面试题（卷王专供+ 史上最全 + 2022面试必备）-V68-from-Java面试红宝书.pdf
专题32：大厂面试的基本流程和面试准备（阿里、腾讯、网易、京东、头条.....）-V2-from-Java面试红宝书-release.pdf
专题33：BST、AVL、RBT红黑树、三大核心数据结构（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题34：Elasticsearch面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-Java面试红宝书-release.pdf
专题35：Mybatis面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩Java面试宝典-release.pdf
专题36：Dubbo面试题（卷王专供+ 史上最全 + 2023面试必备）-V21-from-尼恩Java面试宝典-release.pdf
专题37：Docker面试题（卷王专供+ 史上最全 + 2023面试必备）-V47-from-尼恩Java面试宝典.pdf
专题38：K8S面试题（卷王专供+ 史上最全 + 2023面试必备）-V59-from-尼恩Java面试宝典.pdf
专题39：Nginx面试题（卷王专供+ 史上最全 + 2023面试必备）-V27-from-尼恩Java面试宝典-release.pdf
专题40：操作系统面试题（卷王专供+ 史上最全 + 2023面试必备）-V28-from-尼恩Java面试宝典-release.pdf
专题41：大厂面试真题（卷王专供+ 史上最全 + 2023面试必备）-V84-from-尼恩Java面试宝典.pdf

未来职业，如何突围：三栖架构师



专题26：消息队列面试题：RabbitMQ、Kafka、RocketMQ（史上最全、定期更新）

本文版本说明：V100

> 此文的格式，由markdown 通过程序转成而来，由于很多表格，没有来的及调整，出现一个格式问题，尼恩在此给大家道歉啦。
>
> 由于社群很多小伙伴，在面试，不断的交流最新的面试难题，所以，《尼恩Java面试宝典》， 后面会不断升级，迭代。
>
> 本专题，作为 《尼恩Java面试宝典》专题之一，《尼恩Java面试宝典》一共**41个面试专题，后续还会增加**

面试问题交流说明：

如果遇到面试难题，或者职业发展问题， 或者中年危机问题，都可以来 疯狂创客圈社群交流，

加入交流群，加尼恩微信即可，

尼恩的微信二维码在哪里呢？ 具体参见文末

升级说明：

V100升级说明（2023-09-12）：

网易一面：单节点2000Wtps，Kafka怎么做的?

###

V91升级说明（2023-07-27）：

痛失网易30K之二：看你牛逼轰轰，请写一个阻塞队列

V89升级说明（2023-07-24）：

网易一面，痛失30K：为啥用阻塞队列，list不行吗?

V10升级说明（2022-11-08）

- 场景题：说说消息队列的高可用、不重复消费、可靠传输、顺序消费、消息堆积?

MQ基础题目

问：为什么使用MQ？MQ的优点

简答

- 异步处理 - 相比于传统的串行、并行方式，提高了系统吞吐量。
- 应用解耦 - 系统间通过消息通信，不用关心其他系统的处理。
- 流量削锋 - 可以通过消息队列长度控制请求量；可以缓解短时间内的高并发请求。
- 日志处理 - 解决大量日志传输。
- 消息通讯 - 消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。

详答

主要是：解耦、异步、削峰。

解耦：A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃...A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。如果使用 MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。

就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用 MQ 给它异步化解耦。

异步：A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 3 + 300 + 450 + 200 = 953ms，接近 1s，用户感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求。如果使用 MQ，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 3 + 5 = 8ms。

削峰：减少高峰时期对服务器压力。

问：消息队列有什么优缺点？RabbitMQ有什么优缺点？

优点上面已经说了，就是**在特殊场景下有其对应的好处，解耦、异步、削峰。**

缺点有以下几个：

系统可用性降低

本来系统运行好好的，现在你非要加入个消息队列进去，那消息队列挂了，你的系统不是呵呵了。因此，系统可用性会降低；

系统复杂度提高

加入了消息队列，要多考虑很多方面的问题，比如：一致性问题、如何保证消息不被重复消费、如何保证消息可靠性传输等。因此，需要考虑的东西更多，复杂性增大。

一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉，做好之后，你会发现，妈呀，系统复杂度提升了一个数量级，也许是复杂了 10 倍。但是关键时刻，用，还是得用的。

问：你们公司生产环境用的是什么消息中间件？

这个首先你可以说下你们公司选用的是什么消息中间件，比如用的是RabbitMQ，然后可以初步给一些你对不同MQ中间件技术的选型分析。

举个例子：比如说ActiveMQ是老牌的消息中间件，国内很多公司过去运用的还是非常广泛的，功能很强大。

但是问题在于没法确认ActiveMQ可以支撑互联网公司的高并发、高负载以及高吞吐的复杂场景，在国内互联网公司落地较少。而且使用较多的是一些传统企业，用ActiveMQ做异步调用和系统解耦。

然后你可以说说RabbitMQ，他的好处在于可以支撑高并发、高吞吐、性能很高，同时有非常完善便捷的后台管理界面可以使用。

另外，他还支持集群化、高可用部署架构、消息高可靠支持，功能较为完善。

而且经过调研，国内各大互联网公司落地大规模RabbitMQ集群支撑自身业务的case较多，国内各种中小型互联网公司使用RabbitMQ的实践也比较多。

除此之外，RabbitMQ的开源社区很活跃，较高频率的迭代版本，来修复发现的bug以及进行各种优化，因此综合考虑过后，公司采取了RabbitMQ。

但是RabbitMQ也有一点缺陷，就是他自身是基于erlang语言开发的，所以导致较为难以分析里面的源码，也较难进行深层次的源码定制和改造，毕竟需要较为扎实的erlang语言功底才可以。

然后可以聊聊RocketMQ，是阿里开源的，经过阿里的生产环境的超高并发、高吞吐的考验，性能卓越，同时还支持分布式事务等特殊场景。

而且RocketMQ是基于Java语言开发的，适合深入阅读源码，有需要可以站在源码层面解决线上生产问题，包括源码的二次开发和改造。

另外就是Kafka。Kafka提供的消息中间件的功能明显较少一些，相对上述几款MQ中间件要少很多。

但是Kafka的优势在于专为超高吞吐量的实时日志采集、实时数据同步、实时数据计算等场景来设计。

因此Kafka在大数据领域中配合实时计算技术（比如Spark Streaming、Storm、Flink）使用的较多。但是在传统的MQ中间件使用场景中较少采用。

问：Kafka、ActiveMQ、RabbitMQ、RocketMQ 有什么优缺点？

	ActiveMQ	RabbitMQ	RocketMQ	Kafka	ZeroMQ
单机吞吐量	比RabbitMQ低	2.6w/s（消息做持久化）	11.6w/s	17.3w/s	29w/s
开发语言	Java	Erlang	Java	Scala/Java	C
主要维护者	Apache	Mozilla/Spring	Alibaba	Apache	iMatix，创始人已去世
成熟度	成熟	成熟	开源版本不够成熟	比较成熟	只有C、PHP等版本成熟
订阅形式	点对点(p2p)、广播（发布-订阅）	提供了4种：direct, topic ,Headers和fanout。fanout就是广播模式	基于topic/messageTag以及按照消息类型、属性进行正则匹配的发布订阅模式	基于topic以及按照topic进行正则匹配的发布订阅模式	点对点(p2p)
持久化	支持少量堆积	支持少量堆积	支持大量堆积	支持大量堆积	不支持
顺序消息	不支持	不支持	支持	支持	不支持

	ActiveMQ	RabbitMQ	RocketMQ	Kafka	ZeroMQ
性能稳定性	好	好	一般	较差	很好
集群方式	支持简单集群模式，比如'主-备'，对高级集群模式支持不好。	支持简单集群，'复制'模式，对高级集群模式支持不好。	常用 多对'Master-Slave' 模式，开源版本需手动切换Slave变成Master	天然的'Leader-Slave'无状态集群，每台服务器既是Master也是Slave	不支持
管理界面	一般	较好	一般	无	无

综上，各种对比之后，有如下建议：

一般的业务系统要引入 MQ，最早大家都用 ActiveMQ，但是现在确实大家用的不多了，没经过大规模吞吐量场景的验证，社区也不是很活跃，所以大家还是算了吧，我个人不推荐用这个了；

后来大家开始用 RabbitMQ，但是确实 erlang 语言阻止了大量的 Java 工程师去深入研究和掌控它，对公司而言，几乎处于不可控的状态，但是确实人家是开源的，比较稳定的支持，活跃度也高；

不过现在确实越来越多的公司会去用 RocketMQ，确实很不错，毕竟是阿里出品，但社区可能有突然黄掉的风险（目前 RocketMQ 已捐给 Apache，但 GitHub 上的活跃度其实不算高）对自己公司技术实力有绝对自信的，推荐用 RocketMQ，否则回去老老实实用 RabbitMQ 吧，人家有活跃的开源社区，绝对不会黄。

所以**中小型公司**，技术实力较为一般，技术挑战不是特别高，用 RabbitMQ 是不错的选择；**大型公司**，基础架构研发实力较强，用 RocketMQ 是很好的选择。

如果是**大数据领域**的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。

问：MQ 有哪些常见问题？如何解决这些问题？

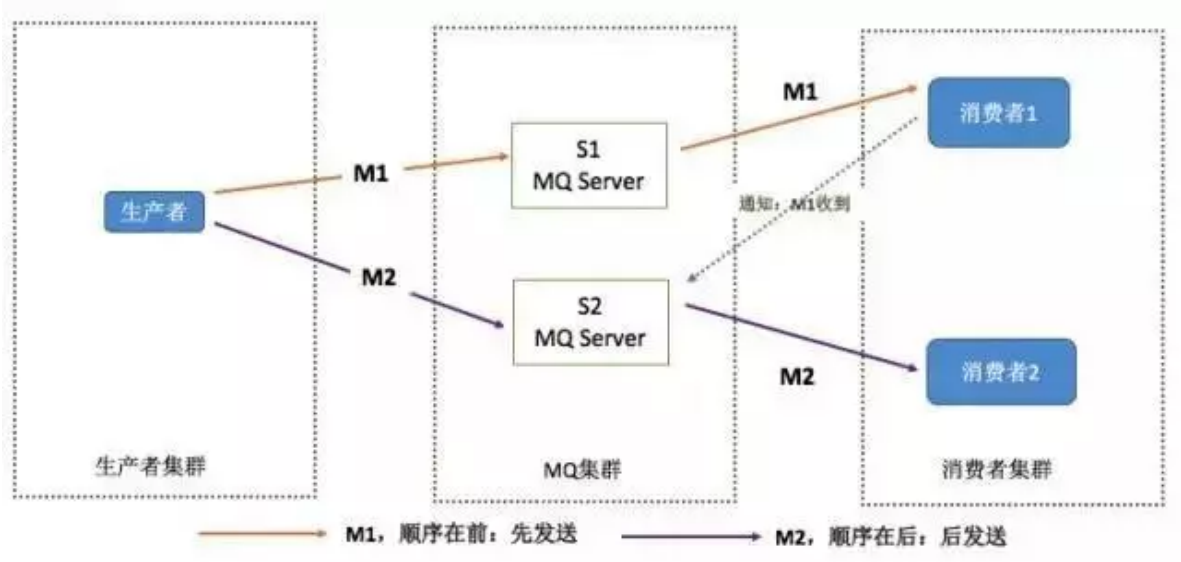
MQ 的常见问题有：

1. 消息的顺序问题
2. 消息的重复问题

消息的顺序问题

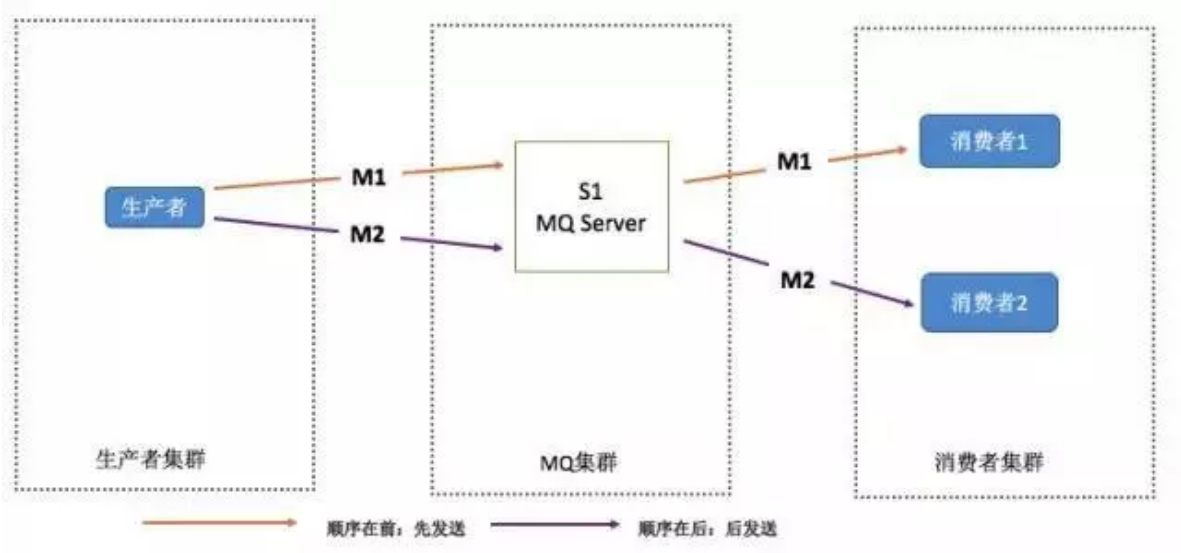
消息有序指的是可以按照消息的发送顺序来消费。

假如生产者产生了 2 条消息：M1、M2，假定 M1 发送到 S1，M2 发送到 S2，如果要保证 M1 先于 M2 被消费，怎么做？



解决方案：

（1）保证生产者 - MQServer - 消费者是一对一对一的关系



缺陷：

- 并行度就会成为消息系统的瓶颈（吞吐量不够）
- 更多的异常处理，比如：只要消费端出现问题，就会导致整个处理流程阻塞，我们不得不花费更多的精力来解决阻塞的问题。 （2）通过合理的设计或者将问题分解来规避。
- 不关注乱序的应用实际大量存在
- 队列无序并不意味着消息无序 所以从业务层面来保证消息的顺序而不仅仅是依赖于消息系统，是一种更合理的方式。

消息的重复问题

造成消息重复的根本原因是：网络不可达。

所以解决这个问题的办法就是绕过这个问题。那么问题就变成了：如果消费端收到两条一样的消息，应该怎样处理？

消费端处理消息的业务逻辑保持幂等性。只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样。保证每条消息都有唯一编号且保证消息处理成功与去重表的日志同时出现。利用一张日志表来记录已经处理成功的消息的 ID，如果新到的消息 ID 已经在日志表中，那么就不再处理这条消息。

问：说说设计MQ思路?

比如说这个消息队列系统，我们从以下几个角度来考虑一下：

首先这个 mq 得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下 kafka 的设计理念，broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。如果现在资源不够了，简单啊，给 topic 增加 partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

其次你得考虑一下这个 mq 的数据要不要落地磁盘吧？那肯定要了，落磁盘才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是 kafka 的思路。

其次你考虑一下你的 mq 的可用性啊？这个事儿，具体参考之前可用性那个环节讲解的 kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。

能不能支持数据 0 丢失啊？可以的，参考我们之前说的那个 kafka 数据零丢失方案。

RabbitMQ

问：什么是RabbitMQ?

RabbitMQ是一款开源的，Erlang编写的，基于AMQP协议的消息中间件

问：rabbitmq 的使用场景

（1）服务间异步通信

（2）顺序消费

（3）定时任务

（4）请求削峰

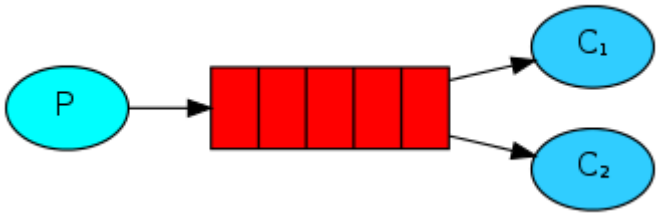
问：RabbitMQ基本概念

- Broker： 简单来说就是消息队列服务器实体
- Exchange： 消息交换机，它指定消息按什么规则，路由到哪个队列
- Queue： 消息队列载体，每个消息都会被投入到一个或多个队列
- Binding： 绑定，它的作用就是把exchange和queue按照路由规则绑定起来
- Routing Key： 路由关键字，exchange根据这个关键字进行消息投递
- VHost： vhost 可以理解为虚拟 broker ，即 mini-RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。
- Producer： 消息生产者，就是投递消息的程序
- Consumer： 消息消费者，就是接受消息的程序
- Channel： 消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务

由Exchange、Queue、RoutingKey三个才能决定一个从Exchange到Queue的唯一的线路。

问：RabbitMQ的工作模式

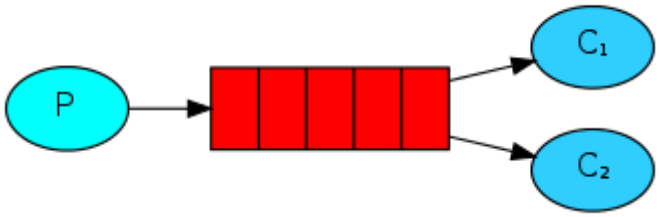
一.simple模式（即最简单的收发模式）



1.消息产生消息，将消息放入队列

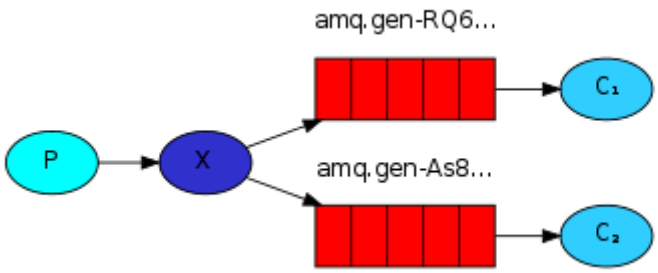
2.消息的消费者(consumer) 监听 消息队列,如果队列中有消息,就消费掉,消息被拿走后,自动从队列中删除(隐患 消息可能没有被消费者正确处理,已经从队列中消失了,造成消息的丢失，这里可以设置成手动的ack,但如果设置成手动ack，处理完后要及时发送ack消息给队列，否则会造成内存溢出)。

二.work工作模式(资源的竞争)



1.消息产生者将消息放入队列消费者可以有多个,消费者1,消费者2同时监听同一个队列,消息被消费。C1 C2共同争抢当前的消息队列内容,谁先拿到谁负责消费消息(隐患：高并发情况下,默认会产生某一个消息被多个消费者共同使用,可以设置一个开关(syncronize) 保证一条消息只能被一个消费者使用)。

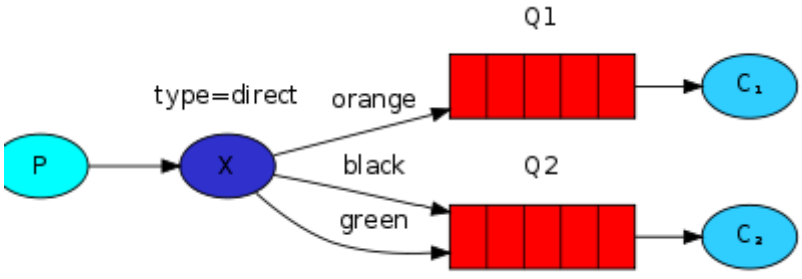
三.publish/subscribe发布订阅(共享资源)



1、每个消费者监听自己的队列；

2、生产者将消息发给broker，由交换机将消息转发到绑定此交换机的每个队列，每个绑定交换机的队列都将接收到消息。

四.routing路由模式



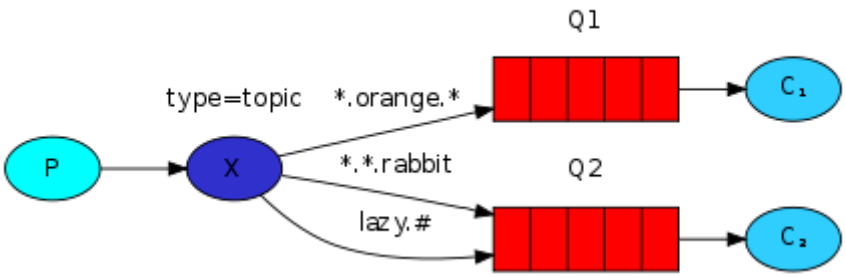
1.消息生产者将消息发送给交换机按照路由判断,路由是字符串(info) 当前产生的消息携带路由字符(对象的方法),交换机根据路由的key,只能匹配上路由key对应的消息队列,对应的消费者才能消费消息;

2.根据业务功能定义路由字符串

3.从系统的代码逻辑中获取对应的功能字符串,将消息任务扔到对应的队列中。

4.业务场景:error 通知;EXCEPTION;错误通知的功能;传统意义的错误通知;客户通知;利用key路由,可以将程序中的错误封装成消息传入到消息队列中,开发者可以自定义消费者,实时接收错误;

五.topic 主题模式(路由模式的一种)



1.星号井号代表通配符

2.星号代表多个单词,井号代表一个单词

3.路由功能添加模糊匹配

4.消息产生者产生消息,把消息交给交换机

5.交换机根据key的规则模糊匹配到对应的队列,由队列的监听消费者接收消息消费

（在我的理解看来就是routing查询的一种模糊匹配，就类似sql的模糊查询方式）

问：如何保证RabbitMQ消息的顺序性?

拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

问：消息如何分发?

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。通过路由可实现多消费的功能

问：消息怎么路由?

消息提供方->路由->一至多个队列消息发布到交换器时，消息将拥有一个路由键（routing key），在消息创建时设定。通过队列路由键，可以把队列绑定到交换器上。消息到达交换器后，RabbitMQ 会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）；

常用的交换器主要分为一下三种：

fanout：如果交换器收到消息，将会广播到所有绑定的队列上

direct：如果路由键完全匹配，消息就被投递到相应的队列

topic：可以使来自不同源头的消息能够到达同一个队列。 使用 topic 交换器时，可以使用通配符

问：消息基于什么传输?

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制。

问：如何保证消息不被重复消费？或者说，如何保证消息消费时的幂等性?

先说为什么会重复消费：正常情况下，消费者在消费消息的时候，消费完毕后，会发送一个确认消息给消息队列，消息队列就知道该消息被消费了，就会将该消息从消息队列中删除；

但是因为网络传输等等故障，确认信息没有传送到消息队列，导致消息队列不知道自己已经消费过该消息了，再次将消息分发给其他的消费者。

针对以上问题，一个解决思路是：保证消息的唯一性，就算是多次传输，不要让消息的多次消费带来影响；保证消息等幂性；

比如：在写入消息队列的数据做唯一标示，消费消息时，根据唯一标识判断是否消费过；

假设你有个系统，消费一条消息就往数据库里插入一条数据，要是你一个消息重复两次，你不就插入了两条，这数据不就错了？但是你要是消费到第二次的时候，自己判断一下是否已经消费过了，若是就直接扔了，这样不就保留了一条数据，从而保证了数据的正确性。

问：如何确保消息正确地发送至 RabbitMQ？ 如何确保消息接收方消费了消息?

发送方确认模式

将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。

一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。

如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（notacknowledged，未确认）消息。

发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

接收方确认机制

消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。

这里并没有用到超时机制，RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。保证数据的最终一致性；

下面罗列几种特殊情况

- 如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要去重）
- 如果消费者接收到消息却没有确认消息，连接也未断开，则 RabbitMQ 认为该消费者繁忙，将不会给该消费者分发更多的消息。

问：如何保证RabbitMQ消息的可靠传输?

消息不可靠的情况可能是消息丢失，劫持等原因；

丢失又分为：生产者丢失消息、消息列表丢失消息、消费者丢失消息；

生产者丢失消息：从生产者弄丢数据这个角度来看，RabbitMQ提供transaction和confirm模式来确保生产者不丢消息；

transaction机制就是说：发送消息前，开启事务（channel.txSelect()）,然后发送消息，如果发送过程中出现什么异常，事务就会回滚（channel.txRollback()）,如果发送成功则提交事务（channel.txCommit()）。然而，这种方式有个缺点：吞吐量下降；

confirm模式用的居多：一旦channel进入confirm模式，所有在该信道上发布的消息都将会被指派一个唯一的ID（从1开始），一旦消息被投递到所有匹配的队列之后；

rabbitMQ就会发送一个ACK给生产者（包含消息的唯一ID），这就使得生产者知道消息已经正确到达目的队列了；

如果rabbitMQ没能处理该消息，则会发送一个Nack消息给你，你可以进行重试操作。

消息队列丢数据：消息持久化。

处理消息队列丢数据的情况，一般是开启持久化磁盘的配置。

这个持久化配置可以和confirm机制配合使用，你可以在消息持久化磁盘后，再给生产者发送一个Ack信号。

这样，如果消息持久化磁盘之前，rabbitMQ阵亡了，那么生产者收不到Ack信号，生产者会自动重发。

那么如何持久化呢?

这里顺便说一下吧，其实也很容易，就下面两步

1. 将queue的持久化标识durable设置为true,则代表是一个持久的队列
2. 发送消息的时候将deliveryMode=2

这样设置以后，即使rabbitMQ挂了，重启后也能恢复数据

消费者丢失消息：消费者丢数据一般是因为采用了自动确认消息模式，改为手动确认消息即可!

消费者在收到消息之后，处理消息之前，会自动回复RabbitMQ已收到消息；

如果这时处理消息失败，就会丢失该消息；

解决方案：处理消息成功后，手动回复确认消息。

问：为什么不应该对所有的 message 都使用持久化机制?

首先，必然导致性能的下降，因为写磁盘比写 RAM 慢的多，message 的吞吐量可能有 10 倍的差距。

其次，message 的持久化机制用在 RabbitMQ 的内置 cluster 方案时会出现"坑爹"问题。矛盾点在于，若 message 设置了 persistent 属性，但 queue 未设置 durable 属性，那么当该 queue 的 owner node 出现异常后，在未重建该 queue 前，发往该 queue 的 message 将被 blackholed ； 若 message 设置了 persistent 属性，同时 queue 也设置了 durable 属性，那么当 queue 的 owner node 异常且无法重启的情况下，则该 queue 无法在其他 node 上重建，只能等待其 owner node 重启后，才能恢复该 queue 的使用，而在这段时间内发送给该 queue 的 message 将被 blackholed 。

所以，是否要对 message 进行持久化，需要综合考虑性能需要，以及可能遇到的问题。若想达到 100,000 条/秒以上的消息吞吐量（单 RabbitMQ 服务器），则要么使用其他的方式来确保 message 的可靠 delivery ，要么使用非常快速的存储系统以支持全持久化（例如使用 SSD）。另外一种处理原则是：仅对关键消息作持久化处理（根据业务重要程度），且应该保证关键消息的量不会导致性能瓶颈。

问：如何保证高可用的？RabbitMQ 的集群

RabbitMQ 是比较有代表性的，因为是基于主从（非分布式）做高可用性的，我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。

单机模式，就是 Demo 级别的，一般就是你本地启动了玩玩儿的?，没人生产用单机模式

普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读写操作。

镜像集群模式：这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！
RabbitMQ 一个 queue 的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个 queue 的完整数据。

问：如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

消息积压处理办法：临时紧急扩容：

先修复 consumer 的问题，确保其恢复消费速度，然后将现有 cnosumer 都停掉。
新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。
然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。
接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。
等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息。
MQ中消息失效：假设你用的是 RabbitMQ，RabbtiMQ 是可以设置过期时间的，也就是 TTL。如果消息在 queue 中积压超过一定的时间就会被 RabbitMQ 给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是大量的数据会直接搞丢。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。假设 1 万个订单积压在 mq 里面，没有处理，其中 1000 个订单都丢了，你只能手动写程序把那 1000 个订单给查出来，手动发到 mq 里去再补一次。

mq消息队列块满了：如果消息积压在 mq 里，你很长时间都没有处理掉，此时导致 mq 都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

Kafka

基础题

问：1、Apache Kafka 是什么?

Apach Kafka 是一款分布式流处理框架，用于实时构建流处理应用。它有一个核心 的功能广为人知，即作为企业级的消息引擎被广泛使用。

你一定要先明确它的流处理框架地位，这样能给面试官留 下一个很专业的印象。

问：2、什么是消费者组?

消费者组是 Kafka 独有的概念，如果面试官问这 个，就说明他对此是有一定了解的。我先给出标准答案：
1、定义：即消费者组是 Kafka 提供的可扩展且具有容错性的消费者机制。
2、原理：在 Kafka 中，消费者组是一个由多个消费者实例 构成的组。多个实例共同订阅若干个主题，实现共同消费。同一个组下的每个实例都配置有 相同的组 ID，被分配不同的订阅分区。当某个实例挂掉的时候，其他实例会自动地承担起 它负责消费的分区。

此时，又有一个小技巧给到你:消费者组的题目，能够帮你在某种程度上掌控下面的面试方向。

- 如果你擅长位移值原理，就不妨再提一下**消费者组的位移提交机制**;
- 如果你擅长 Kafka Broker，可以提一下**消费者组与 Broker 之间的交互**;
- 如果你擅长与消费者组完全不相关的 Producer，那么就可以这么说:“**消费者组要消 费的数据完全来自于 Producer 端生产的消息，我对 Producer 还是比较熟悉的。**”

问：3、在 Kafka 中，ZooKeeper 的作用是什么?

这是一道能够帮助你脱颖而出的题目。碰到这个题目，请在心中暗笑三声。

目前，Kafka 使用 ZooKeeper 存放集群元数据、成员管理、Controller 选举，以及其他一些管理类任务。之后，等 KIP-500 提案完成后，Kafka 将完全不再依赖 于 ZooKeeper。

记住，**一定要突出“目前”**，以彰显你非常了解社区的演进计划。“存放元数据”是指主题 分区的所有数据都保存在 ZooKeeper 中，且以它保存的数据为权威，其他“人”都要与它 保持对齐。“成员管理”是指 Broker 节点的注册、注销以及属性变更，等 等。“Controller 选举”是指选举集群 Controller，而其他管理类任务包括但不限于主题 删除、参数配置等。

不过，抛出 KIP-500 也可能是个双刃剑。碰到非常资深的面试官，他可能会进一步追问你 KIP-500 是做的。一言以蔽之:**KIP-500 思想，是使用社区自研的基于 Raft 的共识算法， 替代 ZooKeeper，实现 Controller 自选举。**

问：4、解释下 Kafka 中位移(offset)的作用

在 Kafka 中，每个 主题分区下的每条消息都被赋予了一个唯一的 ID 数值，用于标识它在分区中的位置。这个 ID 数值，就被称为位移，或者叫偏移量。一旦消息被写入到分区日志，它的位移值将不能 被修改。

答完这些之后，你还可以把整个面试方向转移到你希望的地方。常见方法有以下 3 种:

1. 如果你深谙 Broker 底层日志写入的逻辑，可以强调下消息在日志中的存放格式;
2. 如果你明白位移值一旦被确定不能修改，可以强调下“Log Cleaner 组件都不能影响位 移值”这件事情;
3. 如果你对消费者的概念还算熟悉，可以再详细说说位移值和消费者位移值之间的区别。

问：5、阐述下 Kafka 中的领导者副本(Leader Replica)和追随者副本 (Follower Replica)的区别

这道题表面上是考核你对 Leader 和 Follower 区别的理解，但很容易引申到 Kafka 的同步 机制上。因此，我建议你主动出击，一次性地把隐含的考点也答出来，也许能够暂时把面试 官“唬住”，并体现你的专业性。

你可以这么回答:**Kafka 副本当前分为领导者副本和追随者副本。只有 Leader 副本才能 对外提供读写服务，响应 Clients 端的请求。Follower 副本只是采用拉(PULL)的方 式，被动地同步 Leader 副本中的数据，并且在 Leader 副本所在的 Broker 宕机后，随时 准备应聘 Leader 副本。**

通常来说，回答到这个程度，其实才只说了 60%，因此，我建议你再回答两个额外的加分 项。

- **强调 Follower 副本也能对外提供读服务。**自 Kafka 2.4 版本开始，社区通过引入新的 Broker 端参数，允许 Follower 副本有限度地提供读服务。
- **强调 Leader 和 Follower 的消息序列在实际场景中不一致。**很多原因都可能造成 Leader 和 Follower 保存的消息序列不一致，比如程序 Bug、网络问题等。这是很严重 的错误，必须要完全规避。你可以补充下，之前确保一致性的主要手段是高水位机制， 但高水位值无法保证 Leader 连续变更场景下的数据一致性，因此，社区引入了 Leader Epoch 机制，来修复高水位值的弊端。关于“Leader Epoch 机制”，国内的资料不是 很多，它的普及度远不如高水位，不妨大胆地把这个概念秀出来，力求惊艳一把。

问：6、如何设置 Kafka 能接收的最大消息的大小?

这道题除了要回答消费者端的参数设置之外，一定要加上 Broker 端的设置，这样才算完整。毕竟，如果 Producer 都不能向 Broker 端发送数据很大的消息，又何来消费一说呢? 因此，你需要同时设置 Broker 端参数和 Consumer 端参数。

- Broker 端参数:message.max.bytes、max.message.bytes(主题级别)和 replica.fetch.max.bytes。
- Consumer 端参数:fetch.message.max.bytes。

Broker 端的最后一个参数比较容易遗漏。我们必须调整 Follower 副本能够接收的最大消 息的大小，否则，副本同步就会失败。因此，把这个答出来的话，就是一个加分项。

问：7、监控 Kafka 的框架都有哪些?

面试官其实是在 考察你对监控框架的了解广度，或者说，你是否知道很多能监控 Kafka 的框架或方法。下 面这些就是 Kafka 发展历程上比较有名气的监控系统。

1. **Kafka Manager**:应该算是最有名的专属 Kafka 监控框架了，是独立的监控系统。
2. **Kafka Monitor**:LinkedIn 开源的免费框架，支持对集群进行系统测试，并实时监控测试结果。
3. **CruiseControl**:也是 LinkedIn 公司开源的监控框架，用于实时监测资源使用率，以及 提供常用运维操作等。无 UI 界面，只提供 REST API。
4. **JMX 监控**:由于 Kafka 提供的监控指标都是基于 JMX 的，因此，市面上任何能够集成 JMX 的框架都可以使用，比如 Zabbix 和 Prometheus。
5. **已有大数据平台自己的监控体系**:像 Cloudera 提供的 CDH 这类大数据平台，天然就提 供 Kafka 监控方案。
6. **JMXTool**:社区提供的命令行工具，能够实时监控 JMX 指标。答上这一条，属于绝对 的加分项，因为知道的人很少，而且会给人一种你对 Kafka 工具非常熟悉的感觉。如果 你暂时不了解它的用法，可以在命令行以无参数方式执行一下kafka-run-class.sh kafka.tools.JmxTool，学习下它的用法。

问：8、Broker 的 Heap Size 如何设置?

如何设置 Heap Size 的问题，其实和 Kafka 关系不大，它是一类非常通用的面试题目。一 旦你应对不当，面试方向很有可能被引到 JVM 和 GC 上去，那样的话，你被问住的几率就 会增大。因此，我建议你简单地介绍一下 Heap Size 的设置方法，并把重点放在 Kafka Broker 堆大小设置的最佳实践上。

比如，你可以这样回复:**任何 Java 进程 JVM 堆大小的设置都需要仔细地进行考量和测 试。一个常见的做法是，以默认的初始 JVM 堆大小运行程序，当系统达到稳定状态后，手动触发一次 Full GC，然后通过 JVM 工具查看 GC 后的存活对象大小。之后，将堆大小设 置成存活对象总大小的 1.5~2 倍。对于 Kafka 而言，这个方法也是适用的。不过，业界有 个最佳实践，那就是将 Broker 的 Heap Size 固定为 6GB。经过很多公司的验证，这个大 小是足够且良好的。**

问：9、如何估算 Kafka 集群的机器数量?

这道题目考查的是**机器数量和所用资源之间的关联关系。**所谓资源，也就是 CPU、内存、磁盘和带宽。

通常来说，CPU 和内存资源的充足是比较容易保证的，因此，你需要从磁盘空间和带宽占用两个维度去评估机器数量。

在预估磁盘的占用时，你一定不要忘记计算副本同步的开销。如果一条消息占用 1KB 的磁 盘空间，那么，在有 3 个副本的主题中，你就需要 3KB 的总空间来保存这条消息。显式地 将这些考虑因素答出来，能够彰显你考虑问题的全面性，是一个难得的加分项。

对于评估带宽来说，常见的带宽有 1Gbps 和 10Gbps，但你要切记，**这两个数字仅仅是最大值。**因此，你最好和面试官确认一下给定的带宽是多少。然后，明确阐述出当带宽占用接 近总带宽的 90% 时，丢包情形就会发生。这样能显示出你的网络基本功。

问：10、Leader 总是 -1，怎么破?

在生产环境中，你一定碰到过“某个主题分区不能工作了”的情形。使用命令行查看状态的 话，会发现 Leader 是 -1，于是，你使用各种命令都无济于事，最后只能用“重启大 法”。

但是，有没有什么办法，可以不重启集群，就能解决此事呢?这就是此题的由来。

参考答案:**删除 ZooKeeper 节点 /controller，触发 Controller 重选举。 Controller 重选举能够为所有主题分区重刷分区状态，可以有效解决因不一致导致的 Leader 不可用问题。**我几乎可以断定，当面试官问出此题时，要么就是他真的不知道怎么 解决在向你寻求答案，要么他就是在等你说出这个答案。所以，千万别一上来就说“来个重 启”之类的话。

提高题

问：1.Kafka 的设计时什么样的呢?

Kafka 将消息以 topic 为单位进行归纳

将向 Kafka topic 发布消息的程序成为 producers.

将预订 topics 并消费消息的程序成为 consumer.

Kafka 以集群的方式运行，可以由一个或多个服务组成，每个服务叫做一个 broker.

producers 通过网络将消息发送到 Kafka 集群，集群向消费者提供消息

问：2.数据传输的事务定义有哪三种?

数据传输的事务定义通常有以下三种级别：

（1）最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输

（2）最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输.

（3）精确的一次（Exactly once）: 不会漏传输也不会重复传输,每个消息都传输被一次而

且仅仅被传输一次，这是大家所期望的

问：kafka事务。

分享一篇大佬讲kafka事务的博客，这一篇讲的更深入：http://matt33.com/2018/11/04/kafka-transaction/

同时分享一下这两篇博文，感觉这篇博文讲的更容易理解一些，面试我感觉看这两篇就够了：https://www.jianshu.com/p/64c93065473e，https://www.cnblogs.com/middleware/p/9477133.html

> Kafka从0.11版本开始引入了事务支持。事务可以保证Kafka在Exactly Once语义的基础上，生产和消费可以跨分区和会话，要么全部成功，要么全部失败。
>
> 1）Producer事务
>
> 为了实现跨分区跨会话的事务，需要引入一个全局唯一的Transaction ID，并将Producer获得的PID和Transaction ID绑定。这样当Producer重启后就可以通过正在进行的Transaction ID获得原来的PID。
>
> 为了管理Transaction，Kafka引入了一个新的组件Transaction Coordinator。Producer就是通过和Transaction Coordinator交互获得Transaction ID对应的任务状态。Transaction Coordinator还负责将事务所有写入Kafka的一个内部Topic，这样即使整个服务重启，由于事务状态得到保存，进行中的事务状态可以得到恢复，从而继续进行。
>
> 2）Consumer事务
>
> 上述事务机制主要是从Producer方面考虑，对于Consumer而言，事务的保证就会相对较弱，尤其时无法保证Commit的信息被精确消费。这是由于Consumer可以通过offset访问任意信息，而且不同的Segment File生命周期不同，同一事务的消息可能会出现重启后被删除的情况。

问：3.Kafka 判断一个节点是否还活着有那两个条件?

（1）节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每个节点的连接

（2）如果节点是个 follower,他必须能及时的同步 leader 的写操作，延时不能太久

问：4.producer 是否直接将数据发送到 broker 的 leader(主节点)?

producer 直接将数据发送到 broker 的 leader(主节点)，不需要在多个节点进行分发，为了帮助 producer 做到这点，所有的 Kafka 节点都可以及时的告知:哪些节点是活动的，目标 topic 目标分区的 leader 在哪。这样 producer 就可以直接将消息发送到目的地了

问：5、Kafa consumer 是否可以消费指定分区消息?

Kafa consumer 消费消息时，向 broker 发出"fetch"请求去消费特定分区的消息，consumer 指定消息在日志中的偏移量（offset），就可以消费从这个位置开始的消息，customer 拥有了 offset 的控制权，可以向后回滚去重新消费之前的消息，这是很有意义的

问：6、Kafka 消息是采用 Pull 模式，还是 Push 模式?

Kafka 最初考虑的问题是，customer 应该从 brokes 拉取消息还是 brokers 将消息推送到 consumer，也就是 pull 还 push。在这方面，Kafka 遵循了一种大部分消息系统共同的传统的设计：producer 将消息推送到 broker，consumer 从 broker 拉取消息

一些消息系统比如 Scribe 和 Apache Flume 采用了 push 模式，将消息推送到下游的 consumer。

这样做有好处也有坏处：由 broker 决定消息推送的速率，对于不同消费速率的

consumer 就不太好处理了。消息系统都致力于让 consumer 以最大的速率最快速的消费消息，但不幸的是，push 模式下，当 broker 推送的速率远大于 consumer 消费的速率时，consumer 恐怕就要崩溃了。

最终， Kafka 还是选取了传统的 pull 模式。

Pull 模式的另外一个好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。Push 模式必须在不知道下游 consumer 消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免 consumer 崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull 模式下，consumer 就可以根据自己的消费能力去决定这些策略

Pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询，直到新消息到 t 达。为了避免这点，Kafka 有个参数可以让 consumer 阻塞知道新消息到达 (当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量Pull)

问：7.Kafka 存储在硬盘上的消息格式是什么?

消息由一个固定长度的头部和可变长度的字节数组组成。头部包含了一个版本号和 CRC32 校验码。

- 消息长度: 4 bytes (value: 1+4+n)
- 版本号: 1 byte
- CRC 校验码: 4 bytes
- 具体的消息: n bytes

问：8.Kafka 高效文件存储设计特点：

(1).Kafka 把 topic 中一个 parition 大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。

(2).通过索引信息可以快速定位 message 和确定 response 的最大大小。

(3).通过 index 元数据全部映射到 memory，可以避免 segment file 的 IO 磁盘操作。

(4).通过索引文件稀疏存储，可以大幅降低 index 文件元数据占用空间大小。

问：9.Kafka 与传统消息系统之间有三个关键区别

(1).Kafka 持久化日志，这些日志可以被重复读取和无限期保留

(2).Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性

(3).Kafka 支持实时的流式处理

10.Kafka 创建 Topic 时如何将分区放置到不同的 Broker 中

- 副本因子不能大于 Broker 的个数；
- 第一个分区（编号为 0）的第一个副本放置位置是随机从 brokerList 选择的；

□其他分区的第一个副本放置位置相对于第 0 个分区依次往后移。也就是如果我们有 5 个 Broker，5 个分区，假设第一个分区放在第四个 Broker 上，那么第二个分区将会放在第五个 Broker 上；第三个分区将会放在第一个 Broker 上；第四个分区将会放在第二个 Broker 上，依次类推；

□剩余的副本相对于第一个副本放置位置其实是由 nextReplicaShift 决定的，而这个数也是随机产生的

问：11.Kafka 新建的分区会在哪个目录下创建

在启动 Kafka 集群之前，我们需要配置好 log.dirs 参数，其值是 Kafka 数据的存放目录，这个参数可以配置多个目录，目录之间使用逗号分隔，通常这些目录是分布在不同的磁盘上用于提高读写性能。

当然我们也可以配置 log.dir 参数，含义一样。只需要设置其中一个即可。

如果 log.dirs 参数只配置了一个目录，那么分配到各个 Broker 上的分区肯定只能在这个目录下创建文件夹用于存放数据。

但是如果 log.dirs 参数配置了多个目录，那么 Kafka 会在哪个文件夹中创建分区目录呢？

答案是：Kafka 会在含有分区目录最少的文件夹中创建新的分区目录，分区目录名为 Topic 名+分区 ID。注意，是分区文件夹总数最少的目录，而不是磁盘使用量最少的目录！也就是说，如果你给 log.dirs 参数新增了一个新的磁盘，新的分区目录肯定是先在这个新的磁盘上创建直到这个新的磁盘目录拥有的分区目录不是最少为止。

问：12.partition 的数据如何保存到硬盘

topic 中的多个 partition 以文件夹的形式保存到 broker，每个分区序号从 0 递增，

且消息有序

Partition 文件下有多个 segment（xxx.index，xxx.log）

segment 文件里的 大小和配置文件大小一致可以根据要求修改 默认为 1g

如果大小大于 1g 时，会滚动一个新的 segment 并且以上一个 segment 最后一条消息的偏移量命名

问：13.kafka 的 ack 机制

request.required.acks 有三个值 0 1 -1

0:生产者不会等待 broker 的 ack，这个延迟最低但是存储的保证最弱当 server 挂掉的时候就会丢数据

1：服务端会等待 ack 值 leader 副本确认接收到消息后发送 ack 但是如果 leader 挂掉后他不确保是否复制完成新 leader 也会导致数据丢失

-1：同样在 1 的基础上 服务端会等所有的 follower 的副本受到数据后才会受到 leader 发出的 ack，这样数据不会丢失

问：14.Kafka 的消费者如何消费数据

消费者每次消费数据的时候，消费者都会记录消费的物理偏移量（offset）的位置

等到下次消费时，他会接着上次位置继续消费

问：15.消费者负载均衡策略

一个消费者组中的一个分片对应一个消费者成员，他能保证每个消费者成员都能访问，如

果组中成员太多会有空闲的成员

问：16.数据有序

一个消费者组里它的内部是有序的

消费者组与消费者组之间是无序的

问：17.kafaka 生产数据时数据的分组策略

生产者决定数据产生到集群的哪个 partition 中

每一条消息都是以（key，value）格式

Key 是由生产者发送数据传入

所以生产者（key）决定了数据产生到集群的哪个 partition

深度思考题

问：11、LEO、LSO、AR、ISR、HW 都表示什么含义?

- **LEO**:Log End Offset。日志末端位移值或末端偏移量，表示日志下一条待插入消息的 位移值。举个例子，如果日志有 10 条消息，位移值从 0 开始，那么，第 10 条消息的位 移值就是 9。此时，LEO = 10。
- **LSO**:Log Stable Offset。这是 Kafka 事务的概念。如果你没有使用到事务，那么这个 值不存在(其实也不是不存在，只是设置成一个无意义的值)。该值控制了事务型消费 者能够看到的消息范围。它经常与 Log Start Offset，即日志起始位移值相混淆，因为 有些人将后者缩写成 LSO，这是不对的。在 Kafka 中，LSO 就是指代 Log Stable Offset。
- **AR**:Assigned Replicas。AR 是主题被创建后，分区创建时被分配的副本集合，副本个 数由副本因子决定。
- **ISR**:In-Sync Replicas。Kafka 中特别重要的概念，指代的是 AR 中那些与 Leader 保 持同步的副本集合。在 AR 中的副本可能不在 ISR 中，但 Leader 副本天然就包含在 ISR 中。关于 ISR，**还有一个常见的面试题目是如何判断副本是否应该属于 ISR**。目前的判断 依据是:**Follower 副本的 LEO 落后 Leader LEO 的时间，是否超过了 Broker 端参数 replica.lag.time.max.ms 值**。如果超过了，副本就会被从 ISR 中移除。
- **HW**:高水位值(High watermark)。这是控制消费者可读取消息范围的重要字段。一 个普通消费者只能“看到”Leader 副本上介于 Log Start Offset 和 HW(不含)之间的 所有消息。水位以上的消息是对消费者不可见的。关于 HW，问法有很多，我能想到的 最高级的问法，就是让你完整地梳理下 Follower 副本拉取 Leader 副本、执行同步机制 的详细步骤。这就是我们的第 20 道题的题目，一会儿我会给出答案和解析。

问：12、Kafka 能手动删除消息吗?

其实，Kafka 不需要用户手动删除消息。它本身提供了留存策略，能够自动删除过期消息。 当然，它是支持手动删除消息的。因此，你最好从这两个维度去回答。

- 对于设置了 Key 且参数 cleanup.policy=compact 的主题而言，我们可以构造一条 <Key，null> 的消息发送给 Broker，依靠 Log Cleaner 组件提供的功能删除掉该 Key 的消息。
- 对于普通主题而言，我们可以使用 kafka-delete-records 命令，或编写程序调用 Admin.deleteRecords 方法来删除消息。这两种方法殊途同归，底层都是调用 Admin 的 deleteRecords 方法，通过将分区 Log Start Offset 值抬高的方式间接删除消息。

问：13、__consumer_offsets 是做什么用的?

这是一个内部主题，公开的官网资料很少涉及到。因此，我认为，此题属于面试官炫技一类 的题目。你要小心这里的考点:该主题有 3 个重要的知识点，你一定要全部答出来，才会显得对这块知识非常熟悉。

它是一个内部主题，无需手动干预，由 Kafka 自行管理。当然，我们可以创建该主题。

它的主要作用是负责注册消费者以及保存位移值。可能你对保存位移值的功能很熟悉， 但其实**该主题也是保存消费者元数据的地方。千万记得把这一点也回答上。**另外，这里 的消费者泛指消费者组和独立消费者，而不仅仅是消费者组。

Kafka 的 GroupCoordinator 组件提供对该主题完整的管理功能，包括该主题的创建、 写入、读取和 Leader 维护等。

问：14、分区 Leader 选举策略有几种?

分区的 Leader 副本选举对用户是完全透明的，它是由 Controller 独立完成的。你需要回答的是，在哪些场景下，需要执行分区 Leader 选举。每一种场景对应于一种选举策略。当前，Kafka 有 4 种分区 Leader 选举策略。

- **OfflinePartition Leader 选举**:每当有分区上线时，就需要执行 Leader 选举。所谓的分区上线，可能是创建了新分区，也可能是之前的下线分区重新上线。这是最常见的分区 Leader 选举场景。
- **ReassignPartition Leader 选举**:当你手动运行 kafka-reassign-partitions 命令，或者是调用 Admin 的 alterPartitionReassignments 方法执行分区副本重分配时，可能触发此类选举。假设原来的 AR 是[1，2，3]，Leader 是 1，当执行副本重分配后，副本集 合 AR 被设置成[4，5，6]，显然，Leader 必须要变更，此时会发生 Reassign Partition Leader 选举。
- **PreferredReplicaPartition Leader 选举**:当你手动运行 kafka-preferred-replica- election 命令，或自动触发了 Preferred Leader 选举时，该类策略被激活。所谓的 Preferred Leader，指的是 AR 中的第一个副本。比如 AR 是[3，2，1]，那么， Preferred Leader 就是 3。
- **ControlledShutdownPartition Leader 选举**:当 Broker 正常关闭时，该 Broker 上 的所有 Leader 副本都会下线，因此，需要为受影响的分区执行相应的 Leader 选举。

这 4 类选举策略的大致思想是类似的，即从 AR 中挑选首个在 ISR 中的副本，作为新 Leader。当然，个别策略有些微小差异。不过，回答到这种程度，应该足以应付面试官 了。毕竟，微小差别对选举 Leader 这件事的影响很小。

问：Kafka中有那些地方需要选举？这些地方的选举策略又有哪些?

参考：https://blog.csdn.net/u013256816/article/details/89369160

> 控制器的选举
>
> - Kafka Controller的选举是依赖Zookeeper来实现的，在Kafka集群中哪个broker能够成功创建/controller这个临时（EPHEMERAL）节点他就可以成为Kafka Controller。
>
> 分区leader的选举
>
> - https://www.jianshu.com/p/1f02328a4f2e
>
> 消费者相关的选举

> - 组协调器GroupCoordinator需要为消费组内的消费者选举出一个消费组的leader，这个选举的算法也很简单，分两种情况分析。如果消费组内还没有leader，那么第一个加入消费组的消费者即为消费组的leader。如果某一时刻leader消费者由于某些原因退出了消费组，那么会重新选举一个新的leader。

问：15、Kafka 的哪些场景中使用了零拷贝(Zero Copy)?

Zero Copy 是特别容易被问到的高阶题目。在 Kafka 中，体现 Zero Copy 使用场景的地方有两处:**基于 mmap 的索引和日志文件读写所用的 TransportLayer。**

先说第一个。索引都是基于 MappedByteBuffer 的，也就是让用户态和内核态共享内核态 的数据缓冲区，此时，数据不需要复制到用户态空间。不过，mmap 虽然避免了不必要的 拷贝，但不一定就能保证很高的性能。在不同的操作系统下，mmap 的创建和销毁成本可 能是不一样的。很高的创建和销毁开销会抵消 Zero Copy 带来的性能优势。由于这种不确 定性，在 Kafka 中，只有索引应用了 mmap，最核心的日志并未使用 mmap 机制。

再说第二个。TransportLayer 是 Kafka 传输层的接口。它的某个实现类使用了 FileChannel 的 transferTo 方法。该方法底层使用 sendfile 实现了 Zero Copy。对 Kafka 而言，如果 I/O 通道使用普通的 PLAINTEXT，那么，Kafka 就可以利用 Zero Copy 特 性，直接将页缓存中的数据发送到网卡的 Buffer 中，避免中间的多次拷贝。相反，如果 I/O 通道启用了 SSL，那么，Kafka 便无法利用 Zero Copy 特性了。

问：16、Kafka 为什么不支持读写分离?

这道题目考察的是你对 Leader/Follower 模型的思考。

Leader/Follower 模型并没有规定 Follower 副本不可以对外提供读服务。很多框架都是允 许这么做的，只是 Kafka 最初为了避免不一致性的问题，而采用了让 Leader 统一提供服 务的方式。

不过，在开始回答这道题时，你可以率先亮出观点:**自 Kafka 2.4 之后，Kafka 提供了有限度的读写分离，也就是说，Follower 副本能够对外提供读服务。**

说完这些之后，你可以再给出之前的版本不支持读写分离的理由。

- **场景不适用。**读写分离适用于那种读负载很大，而写操作相对不频繁的场景，可 Kafka 不属于这样的场景。
- **同步机制。**Kafka 采用 PULL 方式实现 Follower 的同步，因此，Follower 与 Leader 存 在不一致性窗口。如果允许读 Follower 副本，就势必要处理消息滞后(Lagging)的问题。

问：17、如何调优 Kafka?

回答任何调优问题的第一步，就是**确定优化目标，并且定量给出目标!**这点特别重要。对于 Kafka 而言，常见的优化目标是吞吐量、延时、持久性和可用性。每一个方向的优化思路都 是不同的，甚至是相反的。

确定了目标之后，还要明确优化的维度。有些调优属于通用的优化思路，比如对操作系统、 JVM 等的优化;有些则是有针对性的，比如要优化 Kafka 的 TPS。我们需要从 3 个方向去考虑

- **Producer 端**:增加 batch.size、linger.ms，启用压缩，关闭重试等。
- **Broker 端**:增加 num.replica.fetchers，提升 Follower 同步 TPS，避免 Broker Full GC 等。
- **Consumer**:增加 fetch.min.bytes 等

问：18、Controller 发生网络分区(Network Partitioning)时，Kafka 会怎 么样?

这道题目能够诱发我们对分布式系统设计、CAP 理论、一致性等多方面的思考。不过，针 对故障定位和分析的这类问题，我建议你首先言明“实用至上”的观点，即不论怎么进行理论分析，永远都要以实际结果为准。一旦发生 Controller 网络分区，那么，第一要务就是 查看集群是否出现“脑裂”，即同时出现两个甚至是多个 Controller 组件。这可以根据 Broker 端监控指标 ActiveControllerCount 来判断。

现在，我们分析下，一旦出现这种情况，Kafka 会怎么样。

由于 Controller 会给 Broker 发送 3 类请求，即LeaderAndIsrRequest、 StopReplicaRequest 和 UpdateMetadataRequest，因此，一旦出现网络分区，这些请求将不能顺利到达 Broker 端。这将影响主题的创建、修改、删除操作的信息同步，表现为 集群仿佛僵住了一样，无法感知到后面的所有操作。因此，网络分区通常都是非常严重的问 题，要赶快修复。

问：19、Java Consumer 为什么采用单线程来获取消息?

在回答之前，如果先把这句话说出来，一定会加分:**Java Consumer 是双线程的设计。一 个线程是用户主线程，负责获取消息;另一个线程是心跳线程，负责向 Kafka 汇报消费者 存活情况。将心跳单独放入专属的线程，能够有效地规避因消息处理速度慢而被视为下线 的“假死”情况。**

单线程获取消息的设计能够避免阻塞式的消息获取方式。单线程轮询方式容易实现异步非阻塞式，这样便于将消费者扩展成支持实时流处理的操作算子。因为很多实时流处理操作算子都不能是阻塞式的。另外一个可能的好处是，可以简化代码的开发。多线程交互的代码是非常容易出错的。

问：20、简述 Follower 副本消息同步的完整流程

首先，Follower 发送 FETCH 请求给 Leader。接着，Leader 会读取底层日志文件中的消 息数据，再更新它内存中的 Follower 副本的 LEO 值，更新为 FETCH 请求中的 fetchOffset 值。最后，尝试更新分区高水位值。Follower 接收到 FETCH 响应之后，会把 消息写入到底层日志，接着更新 LEO 和 HW 值。

Leader 和 Follower 的 HW 值更新时机是不同的，Follower 的 HW 更新永远落后于 Leader 的 HW。这种时间上的错配是造成各种不一致的原因。

重点：kafka如何实现高吞吐?

问：kafka如何实现高吞吐?

Kafka是分布式消息系统，需要处理海量的消息，Kafka的设计是把所有的消息都写入速度低容量大的硬盘，以此来换取更强的存储能力，但实际上，使用硬盘并没有带来过多的性能损失。kafka主要使用了以下几个方式实现了超高的吞吐率：

- 顺序读写；
- 零拷贝
- 文件分段
- 批量发送
- 数据压缩。

具体来说：

读写文件依赖OS文件系统的页缓存，而不是在JVM内部缓存数据，利用OS来缓存，内存利用率高

sendfile技术（零拷贝），避免了传统网络IO四步流程

支持End-to-End的压缩

顺序IO以及常量时间get、put消息

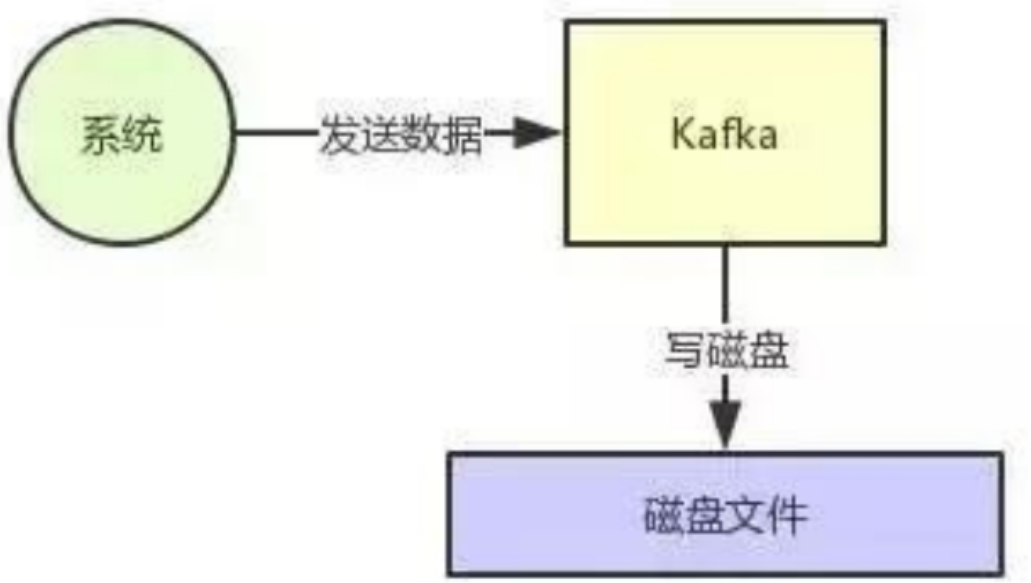
Partition 可以很好的横向扩展和提供高并发处理

问：Kafka如何实现每秒上百万的超高并发写入？掌握好面试给你打满分

Kafka 是高吞吐低延迟的高并发、高性能的消息中间件，在大数据领域有极为广泛的运用。配置良好的 Kafka 集群甚至可以做到每秒几十万、上百万的超高并发写入。

页缓存技术 + 磁盘顺序写

首先 Kafka 每次接收到数据都会往磁盘上去写，如下图所示：



那么在这里我们不禁有一个疑问了，如果把数据基于磁盘来存储，频繁的往磁盘文件里写数据，这个性能会不会很差?大家肯定都觉得磁盘写性能是极差的。

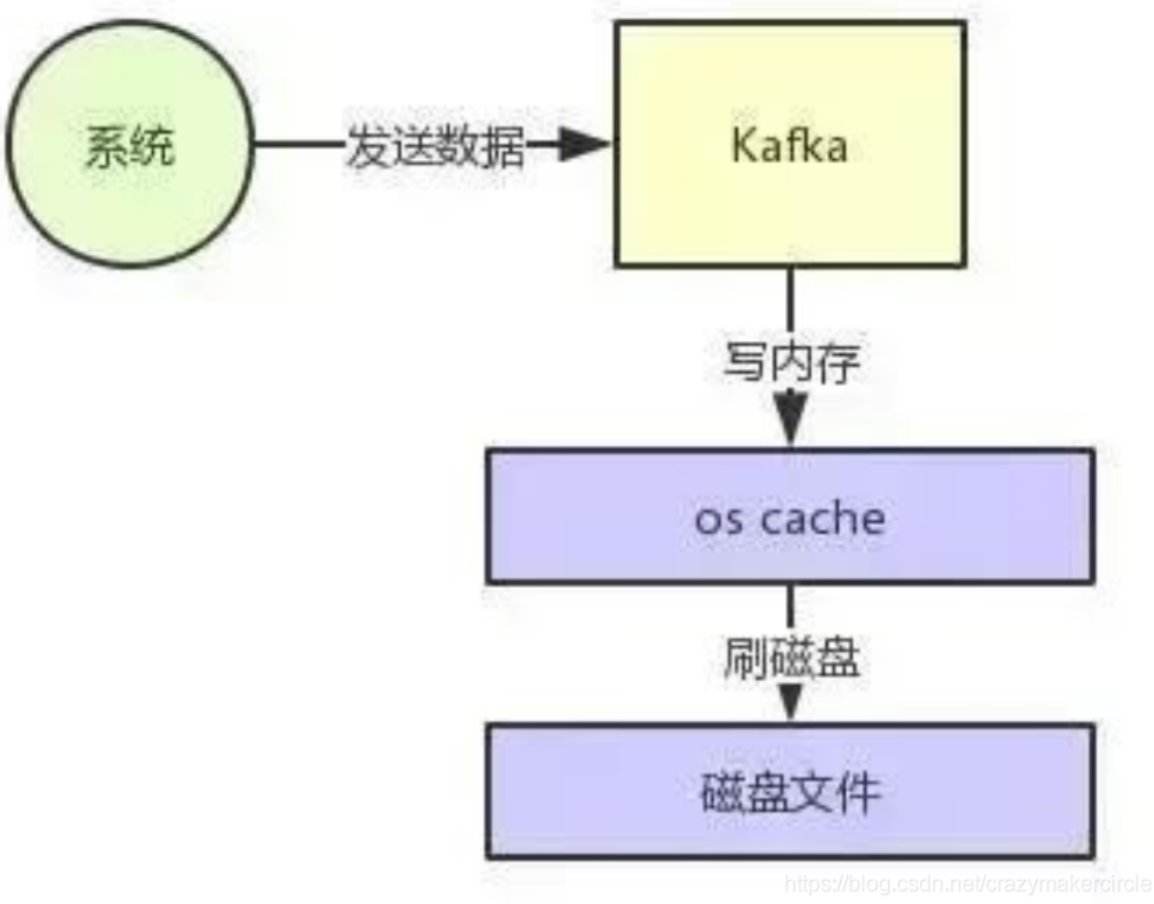
没错，要是真的跟上面那个图那么简单的话，那确实这个性能是比较差的。

但是实际上 Kafka 在这里有极为优秀和出色的设计，就是为了保证数据写入性能，首先 Kafka 是基于操作系统的页缓存来实现文件写入的。

操作系统本身有一层缓存，叫做 Page Cache，是在内存里的缓存，我们也可以称之为 OS Cache，意思就是操作系统自己管理的缓存。

你在写入磁盘文件的时候，可以直接写入这个 OS Cache 里，也就是仅仅写入内存中，接下来由操作系统自己决定什么时候把 OS Cache 里的数据真的刷入磁盘文件中。

仅仅这一个步骤，就可以将磁盘文件写性能提升很多了，因为其实这里相当于是在写内存，不是在写磁盘，大家看下图：



接着另外一个就是 kafka 写数据的时候，非常关键的一点，它是以磁盘顺序写的方式来写的。

也就是说，仅仅将数据追加到文件的末尾，不是在文件的随机位置来修改数据。

普通的机械磁盘如果你要是随机写的话，确实性能极差，也就是随便找到文件的某个位置来写数据。

但是如果你是追加文件末尾按照顺序的方式来写数据的话，那么这种磁盘顺序写的性能基本上可以跟写内存的性能本身也是差不多的。

所以大家就知道了，上面那个图里，Kafka 在写数据的时候，一方面基于 OS 层面的 Page Cache 来写数据，所以性能很高，本质就是在写内存罢了。

另外一个，它是采用磁盘顺序写的方式，所以即使数据刷入磁盘的时候，性能也是极高的，也跟写内存是差不多的。

基于上面两点，Kafka 就实现了写入数据的超高性能。那么大家想想，假如说 Kafka 写入一条数据要耗费 1 毫秒的时间，那么是不是每秒就是可以写入 1000 条数据?

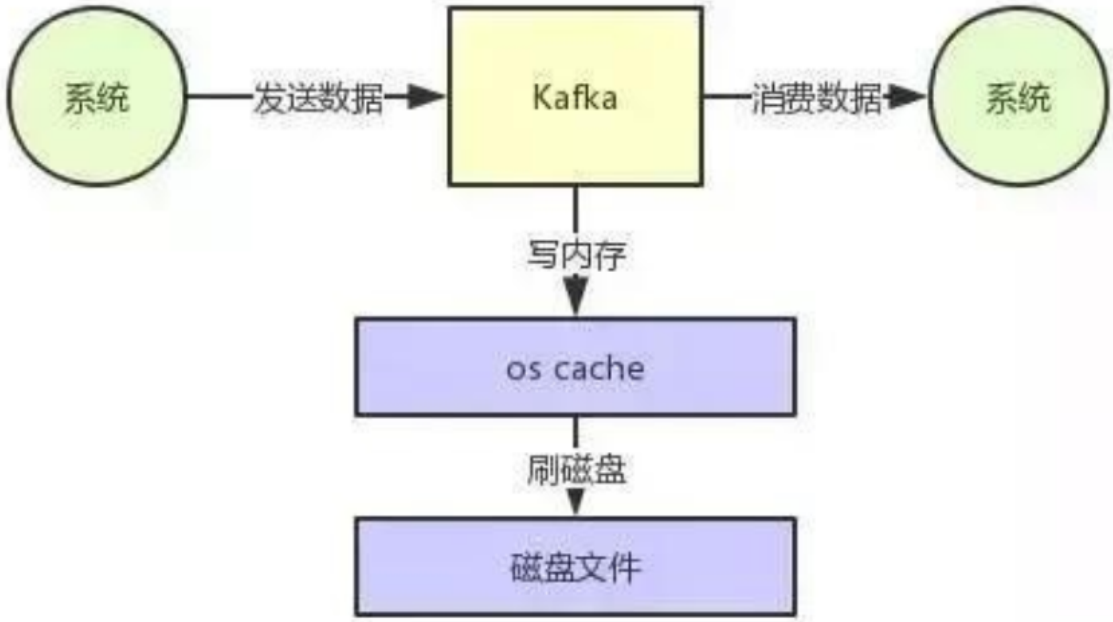
但是假如 Kafka 的性能极高，写入一条数据仅仅耗费 0.01 毫秒呢?那么每秒是不是就可以写入 10 万条数据?

所以要保证每秒写入几万甚至几十万条数据的核心点，就是尽最大可能提升每条数据写入的性能，这样就可以在单位时间内写入更多的数据量，提升吞吐量。

零拷贝技术

说完了写入这块，再来谈谈消费这块。

大家应该都知道，从 Kafka 里我们经常要消费数据，那么消费的时候实际上就是要从 Kafka 的磁盘文件里读取某条数据然后发送给下游的消费者，如下图所示：



那么这里如果频繁的从磁盘读数据然后发给消费者，性能瓶颈在哪里呢?

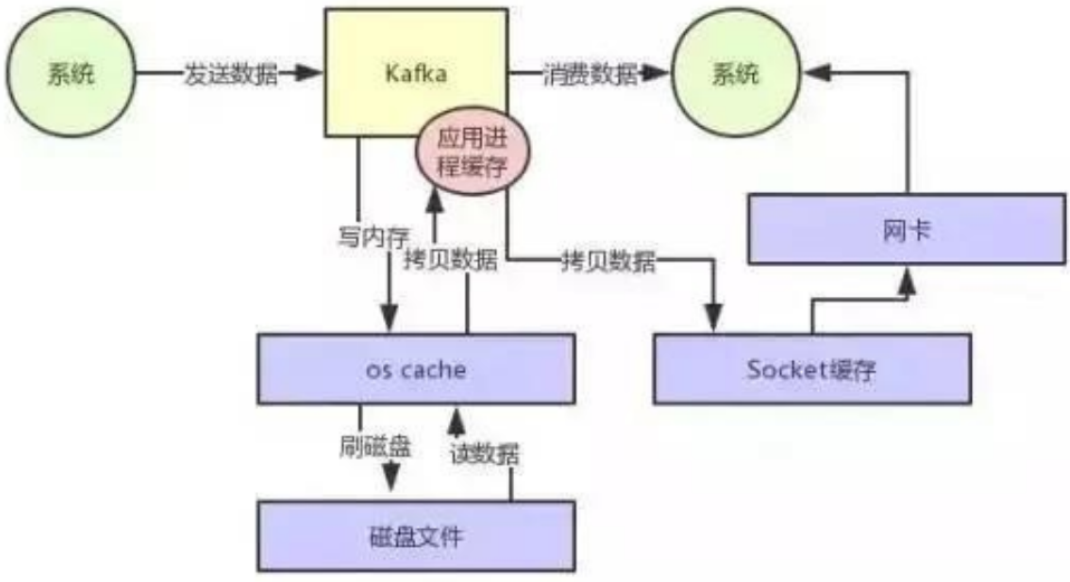
假设要是 Kafka 什么优化都不做，就是很简单的从磁盘读数据发送给下游的消费者，那么大概过程如下所示：

先看看要读的数据在不在 OS Cache 里，如果不在的话就从磁盘文件里读取数据后放入 OS Cache。

接着从操作系统的 OS Cache 里拷贝数据到应用程序进程的缓存里，再从应用程序进程的缓存里拷贝数据到操作系统层面的 Socket 缓存里。

最后从 Socket 缓存里提取数据后发送到网卡，最后发送出去给下游消费。

整个过程，如下图所示：



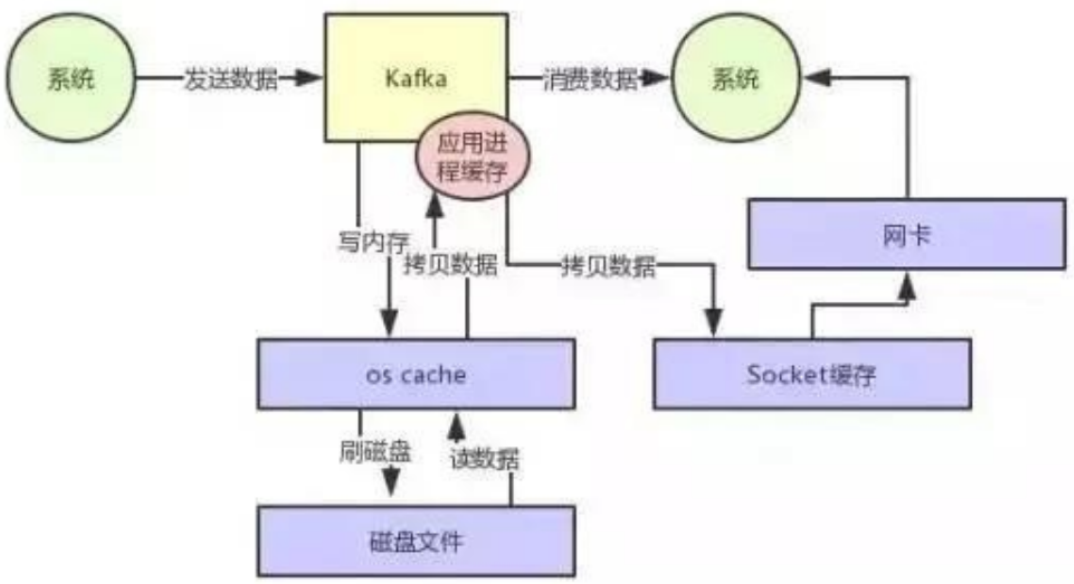
大家看上图，很明显可以看到有两次没必要的拷贝吧!一次是从操作系统的 Cache 里拷贝到应用进程的缓存里，接着又从应用程序缓存里拷贝回操作系统的 Socket 缓存里。

而且为了进行这两次拷贝，中间还发生了好几次上下文切换，一会儿是应用程序在执行，一会儿上下文切换到操作系统来执行。

所以这种方式来读取数据是比较消耗性能的。Kafka 为了解决这个问题，在读数据的时候是引入零拷贝技术。

也就是说，直接让操作系统的 Cache 中的数据发送到网卡后传输给下游的消费者，中间跳过了两次拷贝数据的步骤，Socket 缓存中仅仅会拷贝一个描述符过去，不会拷贝数据到 Socket 缓存。

大家看下图，体会一下这个精妙的过程：



通过零拷贝技术，就不需要把 OS Cache 里的数据拷贝到应用缓存，再从应用缓存拷贝到 Socket 缓存了，两次拷贝都省略了，所以叫做零拷贝。

对 Socket 缓存仅仅就是拷贝数据的描述符过去，然后数据就直接从 OS Cache 中发送到网卡上去了，这个过程大大的提升了数据消费时读取文件数据的性能。

而且大家会注意到，在从磁盘读数据的时候，会先看看 OS Cache 内存中是否有，如果有的话，其实读数据都是直接读内存的。

如果 Kafka 集群经过良好的调优，大家会发现大量的数据都是直接写入 OS Cache 中，然后读数据的时候也是从 OS Cache 中读。

相当于是 Kafka 完全基于内存提供数据的写和读了，所以这个整体性能会极其的高。

RocketMQ

问：多个MQ如何选型?

RabbitMQ
erlang开发，对消息堆积的支持并不好，当大量消息积压的时候，会导致 RabbitMQ 的性能急剧下降。每秒钟可以处理几万到十几万条消息。

RocketMQ
java开发，面向`互联网集群化`，功能丰富，对在线业务的响应时延做了很多的优化，大多数情况下可以做到毫秒级的响应，每秒钟大概能处理几十万条消息。

Kafka
Scala开发，面向`日志`，功能丰富，性能最高。当你的业务场景中，每秒钟消息数量没有那么多的时候，Kafka 的时延反而会比较高。所以，Kafka 不太适合在线业务场景。

ActiveMQ
java开发，简单，稳定，性能不如前面三个。不推荐。

问：RocketMQ组成部分（角色）有哪些?

生产者（Producer）：负责产生消息，生产者向消息服务器发送由业务应用程序系统生成的消息。

消费者（Consumer）：负责消费消息，消费者从消息服务器拉取信息并将其输入用户应用程序。

消息服务器（Broker）：是消息存储中心，主要作用是接收来自 Producer 的消息并存储， Consumer 从这里取得消息。

名称服务器（NameServer）：用来保存 Broker 相关 Topic 等元信息并给 Producer ，提供 Consumer 查找 Broker 信息。

问：RocketMQ消费模式有几种?

集群消费

- 一条消息只会被同Group中的一个Consumer消费
- 多个Group同时消费一个Topic时，每个Group都会有一个Consumer消费到数据

广播消费

- 消息将对一个Consumer Group 下的各个 Consumer 实例都消费一遍。即即使这些 Consumer 属于同一个Consumer Group ，消息也会被 Consumer Group 中的每个 Consumer 都消费一次。

问：消息重复消费如何解决?

出现原因
正常情况下在consumer真正消费完消息后应该发送ack，通知broker该消息已正常消费，从queue中剔除
当ack因为网络原因无法发送到broker，broker会认为词条消息没有被消费，此后会开启消息重投机制把消息再次投递到consumer。

消费模式：在`CLUSTERING`模式下，消息在broker中会保证相同group的consumer消费一次，但是针对不同group的consumer会推送多次

解决方案

- 数据库表：处理消息前，使用消息主键在表中带有约束的字段中insert
- Map：单机时可以使用map做限制，消费时查询当前消息id是不是已经存在
- Redis：使用分布式锁。

问：RocketMQ如何保证消息的顺序消费?

首先多个queue只能保证单个queue里的顺序，queue是典型的FIFO，天然顺序。多个queue同时消费是无法绝对保证消息的有序性的。
可以使用同一`topic`，同一个QUEUE，发消息的时候一个线程去发送消息，消费的时候 一个线程去消费一个queue里的消息。

问：RocketMQ如何保证消息不丢失?

Producer端

采取`send()`同步发消息，发送结果是同步感知的。
发送失败后可以重试，设置重试次数。默认3次。

Broker端
修改刷盘策略为同步刷盘。默认情况下是异步刷盘的。
集群部署

Consumer端
完全消费正常后在进行手动ack确认

问：RocketMQ执行流程

1、启动 Namesrv，Namesrv起 来后监听端口，等待 Broker、Producer、Consumer 连上来，相当于一个路由控制中心。

2、Broker 启动，跟所有的 Namesrv 保持长连接，定时发送心跳包。

3、收发消息前，先创建 Topic 。创建 Topic 时，需要指定该 Topic 要存储在 哪些 Broker上。也可以在发送消息时自动创建Topic。

4、Producer 发送消息。

5、Consumer 消费消息。

问：请说说你对 Producer 的了解?

1、获得 Topic-Broker 的映射关系。

Producer 启动时，也需要指定 Namesrv 的地址，从 Namesrv 集群中选一台建立长连接。

生产者每 30 秒从 Namesrv 获取 Topic 跟 Broker 的映射关系，更新到本地内存中。然后再跟 Topic 涉及的所有 Broker 建立长连接，每隔 30 秒发一次心跳。

2、生产者端的负载均衡。

生产者发送时，会自动轮询当前所有可发送的broker，一条消息发送成功，下次换另外一个broker发送，以达到消息平均落到所有的broker上。

问：说说你对 Consumer 的了解?

1、获得 Topic-Broker 的映射关系。

Consumer 启动时需要指定 Namesrv 地址，与其中一个 Namesrv 建立长连接。消费者每隔 30 秒从 Namesrv 获取所有Topic 的最新队列情况，

Consumer 跟 Broker 是长连接，会每隔 30 秒发心跳信息到Broker .

2、消费者端的负载均衡。根据消费者的消费模式不同，负载均衡方式也不同。

问：消费者消费模式有几种?

消费者消费模式有两种：集群消费和广播消费。

1. 集群消费

消费者的一种消费模式。一个 Consumer Group 中的各个 Consumer 实例分摊去消费消息，即一条消息只会投递到一个 Consumer Group 下面的一个实例。

2. 广播消费

消费者的一种消费模式。消息将对一 个Consumer Group 下的各个 Consumer 实例都投递一遍。即即使这些 Consumer 属于同一个Consumer Group ，消息也会被 Consumer Group 中的每个 Consumer 都消费一次。

问：消费者获取消息有几种模式?

消费者获取消息有两种模式：推送模式和拉取模式。

1. PushConsumer

推送模式（虽然 RocketMQ 使用的是长轮询）的消费者。消息的能及时被消费。使用非常简单，内部已处理如线程池消费、流控、负载均衡、异常处理等等的各种场景。

2. PullConsumer

拉取模式的消费者。应用主动控制拉取的时机，怎么拉取，怎么消费等。主动权更高。但要自己处理各种场景。

问：什么是定时消息？如何实现?

定时消息，是指消息发到 Broker 后，不能立刻被 Consumer 消费，要到特定的时间点或者等待特定的时间后才能被消费。

问：RocketMQ如何实现分布式事务?

1、生产者向MQ服务器发送half消息。
2、half消息发送成功后，MQ服务器返回确认消息给生产者。
3、生产者开始执行本地事务。
4、根据本地事务执行的结果（`UNKNOW`、`commit`、`rollback`）向MQ Server发送提交或回滚消息。
5、如果错过了（可能因为网络异常、生产者突然宕机等导致的异常情况）提交/回滚消息，则MQ服务器将向同一组中的每个生产者发送回查消息以获取事务状态。
6、回查生产者本地事物状态。
7、生产者根据本地事务状态发送提交/回滚消息。

8、MQ服务器将丢弃回滚的消息，但已提交（进行过二次确认的half消息）的消息将投递给消费者进行消费。

`Half Message`：预处理消息，当broker收到此类消息后，会存储到`RMQ_SYS_TRANS_HALF_TOPIC`的消息消费队列中

`检查事务状态`：Broker会开启一个定时任务，消费`RMQ_SYS_TRANS_HALF_TOPIC`队列中的消息，每次执行任务会向消息发送者确认事务执行状态（提交、回滚、未知），如果是未知，Broker会定时去回调在重新检查。

超时：如果超过回查次数，默认回滚消息。
也就是他并未真正进入Topic的queue，而是用了临时queue来放所谓的`half message`，等提交事务后才会真正的将half message转移到topic下的queue。

问：RocketMQ的消息堆积如何处理?

1、如果可以添加消费者解决，就添加消费者的数据量
2、如果出现了queue，但是消费者多的情况。可以使用准备一个临时的topic，同时创建一些queue，在临时创建一个消费者来把这些消息转移到topic中，让消费者消费。

场景题：说说消息队列的高可用、不重复消费、可靠传输、顺序消费、消息堆积?

如何保证消息队列的高可用?

RabbitMQ的高可用

RabbitMQ的高可用是基于主从（非分布式）做高可用性。

RabbitMQ 有三种模式：单机模式（Demo级别）、普通集群模式（无高可用性）、镜像集群模式（高可用性）。

- 普通集群模式

多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。

一个 queue只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。

消费的时候，如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。

普通集群模式下，存在不高可用的问题：**如果那个放 queue 的实例宕机了，会导致接下来其他实例就无法从那个实例拉取**

当然，可以进行 RabbitMQ 存储消息持久化，消息不一定会丢，得等这个实例恢复了，然后才可以继续从这个 queue 拉取数据。

但是，如果磁盘挂了， 就彻底没戏了。

- 镜像集群模式

这种模式，才是所谓的 RabbitMQ 的高可用模式。

在镜像集群模式下，创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，

就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的元数据、消息数据。

写的时候，为了保证高可用，都会自动把消息同步到多个实例的 queue 上。

那么如何开启这个镜像集群模式呢?

RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，

再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。

Kafka的高可用

Kafka 0.8 以前，是没有 HA 机制的，就是任何一个 broker 宕机了，那个 broker 上的 partition 就废了，没法写也没法读，没有什么高可用性可言。

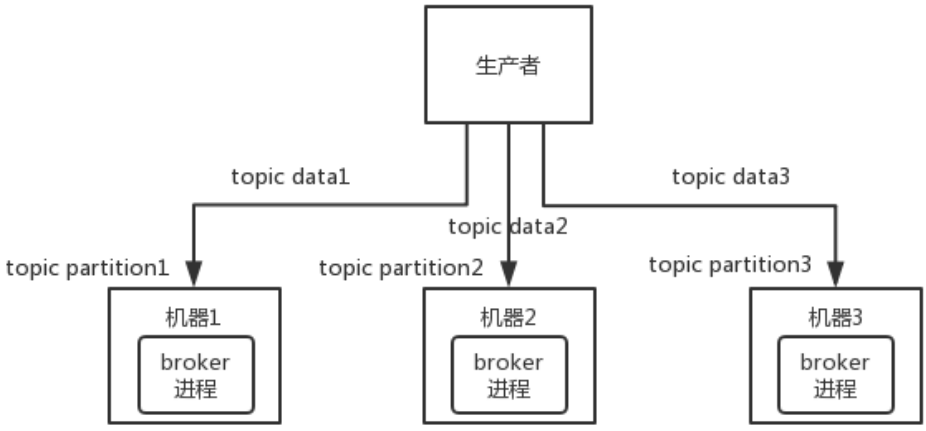
比如说，我们假设创建了一个 topic，指定其 partition 数量是 3 个，分别在三台机器上。

说明：Kafka 有个partition 的自动平衡能力：

由多个 broker 组成，每个 broker 是一个节点；你创建一个 topic，这个 topic 可以划分为多个 partition，每个 partition 可以存在于不同的 broker 上，每个 partition 就放一部分数据。

但是，自平衡能力并不能替代高可用。

如果第二台机器宕机了，会导致这个 topic 的 1/3 的数据就丢了，因此这个是做不到高可用的。



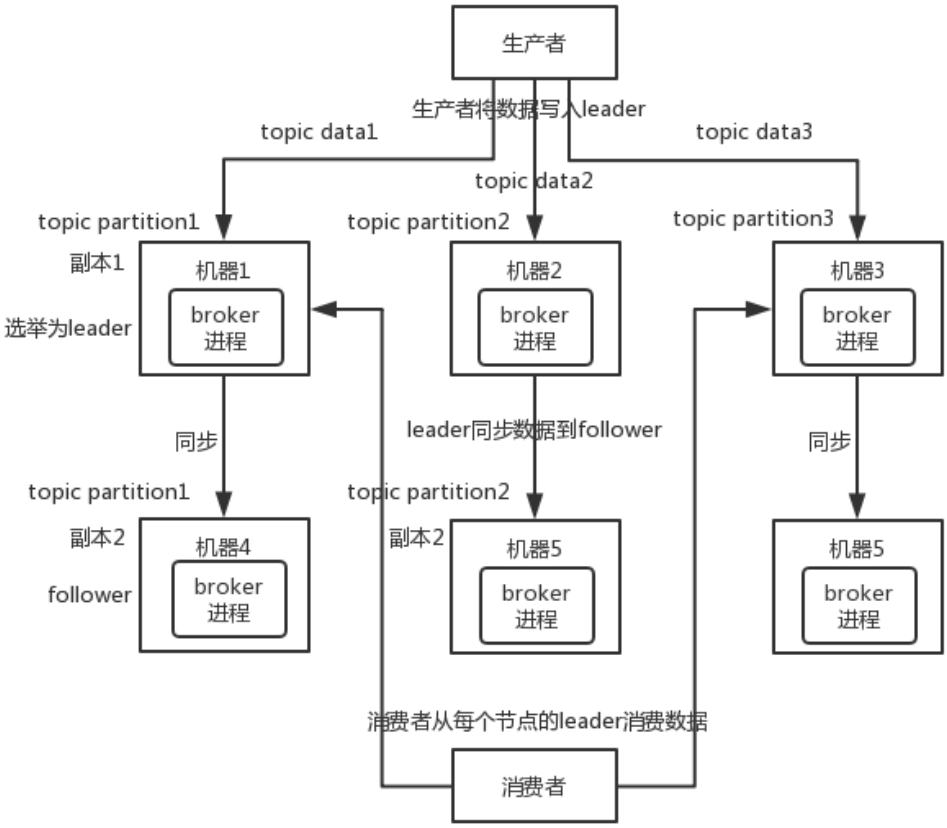
Kafka 0.8 以后，提供了 **HA 机制**，就是 **replica（复制品）** 副本机制。

每个 partition 的数据都会同步到其它机器上，形成自己的多个 replica 副本。

所有 replica 会选举一个 leader 出来，那么生产和消费都跟这个 leader 打交道，然后其他 replica 就是 follower。

写的时候，leader 会负责把数据同步到所有 follower 上去，读的时候就直接读 leader 上的数据即可。

另外，Kafka 会有很好的自平衡能力：均匀地将一个 partition 的所有 replica 分布在不同的机器上，这样才可以提高容错性。



写数据的时候，生产者就写 leader，然后 leader 将数据落地写本地磁盘，接着其他 follower 自己主动从 leader 来 pull 数据。一旦所有 follower 同步好数据了，就会发送 ack 给 leader，leader 收到所有 follower 的 ack 之后，就会返回写成功的消息给生产者。（当然，这只是其中一种模式，还可以适当调整这个行为）

消费的时候，只会从 leader 去读，但是只有当一个消息已经被所有 follower 都同步成功返回 ack 的时候，这个消息才会被消费者读到。

如何保证消息不重复消费（幂等性）？

所有的消息队列都会有这样重复消费的问题，

因为**这是不MQ来保证**，而是通过业务逻辑保证的，

下面，使用Kakfa来讨论是如何实现幂等性。

Kakfa有个offset的概念，就是每个消息写进去都会有一个offset值，代表消费的序号，然后consumer消费了数据之后，默认每隔一段时间会把自己消费过的消息的offset值提交，表示我已经消费过了，

消费者对offset进行持久化（老版本通过zk）， 下次重启之后，从之前的offset处来继续消费。

但是，如果offset值，被不小心调整了，或者 丢失了，就会导致重复消费

其实重复消费不可怕，那么重复消费之后，怎么保证幂等性？答案是通过业务逻辑保证的。

在业务逻辑中，对消息进行业务数据的重复判断，比如 消息中有id，可以根据id进行业务数据的判断处理，看看是否已经处理过，

如果是，表示已经收到了消息，并且处理完成，直接跳过此次处理，实现幂等性。

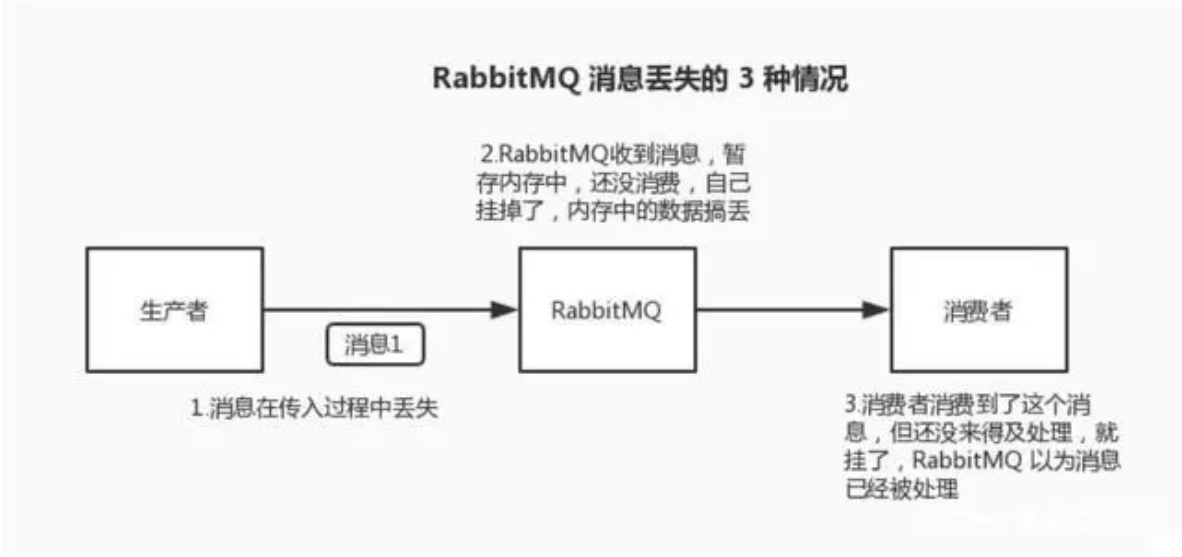
那么，业务中，大致有哪些思路实现幂等性呢：

- 写库 场景，可以先根据主键查一下，如果这数据都有了，你就别插入了，update 一下好吧。
- 写 Redis场景，那没问题了，反正每次都是 set，天然幂等性。
- 基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。
- ....很多其他的业务处理策略，保证只处理一次就ok

如何保证消息的可靠性传输?

数据的丢失问题，可能出现在生产者、MQ、消费者中，咱们从 RabbitMQ 和 Kafka 分别来分析一下吧。

RabbitMQ



生产者环节的高可靠

生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络问题啥的，都有可能。

此时可以选择用 RabbitMQ 提供的事务功能，就是生产者发送数据之前开启 RabbitMQ 事务 channel.txSelect，然后发送消息，

如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务 channel.txRollback，然后重试发送消息；如果收到了消息，那么可以提交事务channel.txCommit。

示例代码：

```
// 开启事务
channel.txSelect
try {
  // 这里发送消息
  } catch (Exception e) {
  channel.txRollback
  // 这里再次重发这条消息
}

// 提交事务
channel.txCommit
```

但是，高可靠，意味着低性能。

但是问题是，RabbitMQ 事务机制（同步）一搞，基本上吞吐量会下来，因为太耗性能。

RabbitMQ 环节的高可靠

必须开启 RabbitMQ 的持久化，就是消息写入之后会持久化到磁盘，

哪怕是 RabbitMQ 自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。

当然，这样也不是100%，可能RabbitMQ 还没持久化，自己就挂了，可能导致少量数据丢失。

设置持久化有两个步骤：

创建 queue 的时候将其设置为持久化；这样就可以保证 RabbitMQ 持久化 queue 的元数据，但是它是不会持久化 queue 里的数据的。

第二个是发送消息的时候将消息的 deliveryMode 设置为 2就是将消息设置为持久化的，此时 RabbitMQ 就会将消息持久化到磁盘上去。

必须要同时设置这两个持久化才行，RabbitMQ 哪怕是挂了，再次重启，也会从磁盘上重启恢复 queue，恢复这个 queue 里的数据。

如何能保证100%呢?

所以，RabbitMQ 环节的高可靠，可以跟生产者那边的 confirm 机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者 ack 了，所以哪怕是在持久化到磁盘之前，RabbitMQ 挂了，数据丢了，生产者收不到 ack，你也是可以自己重发的。

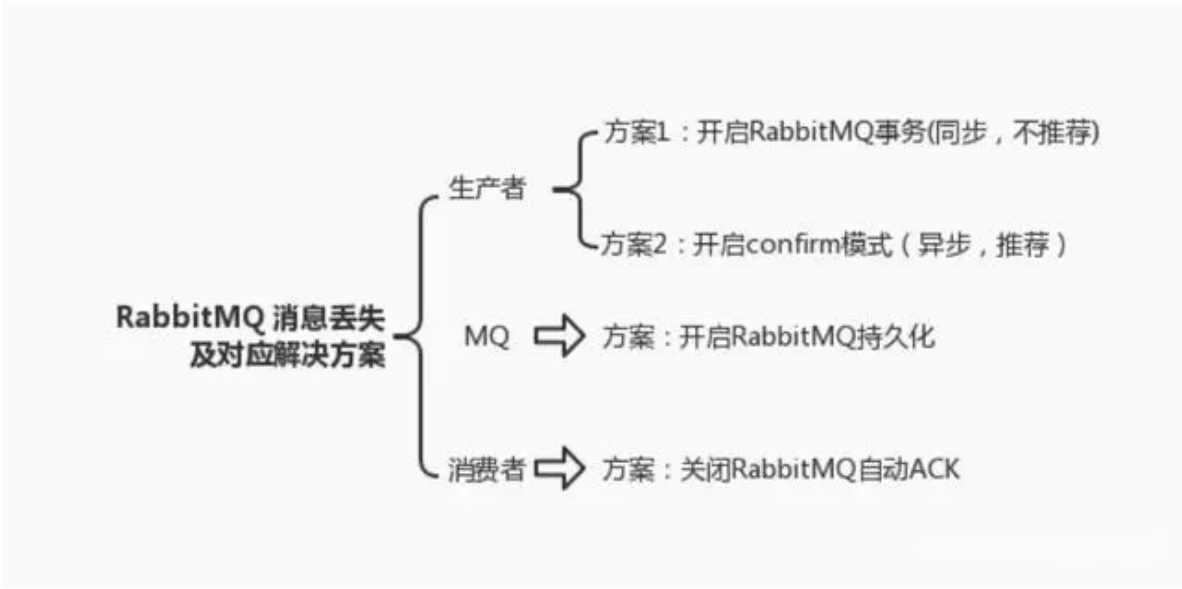
消费端环节的高可靠

必须关闭 RabbitMQ 的自动 ack机制，消费端得用 RabbitMQ 提供的 ack api，进行手动 ack

消费端可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再在消费者程序里 ack 。

没有处理完，就不做手动 ack

没有ack报文，那 RabbitMQ 就认为消息没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。



Kafka如何保证消息的可靠

消费端环节的高可靠

跟 RabbitMQ 差不多吗， Kafka的消费端默认 会自动提交 offset，

那么只要消费端关闭自动提交 offset，

在消费端环节处理完，之后自己手动提交 offset，就可以保证数据不会丢。

但是此时确实还是可能会有重复消费，此时肯定会重复消费一次，自己保证幂等性就好了。

Kafka环节的高可靠

首先必须是高可用的kafka集群

其次，如果 Kafka 某个 broker 宕机，然后重新选举 partition 的 leader。

此时其他的 follower 刚好还有些数据没有同步，结果此时 leader 挂了，然后选举某个follower 成 leader 之后，将 follower 切换为 leader 。不就少了一些数据?

关键，要设置 acks=all 属性，只有完成多个副本的写入，producer 端 才算写入完成。

为了保证数据不丢失，一般是要求起码设置如下 4 个参数：

- 给 topic 设置 replication.factor 参数：这个值必须大于 1，要求每个 partition 必须有至少 2 个副本。
- 在 Kafka 服务端设置 min.insync.replicas 参数：这个值必须大于 1，这个是要求一个 leader至少感知到有至少一个 follower 还跟自己保持联系，没掉队，这样才能确保 leader 挂了还有一个 follower 吧。
- 在 producer 端设置 acks=all：这个是要求每条数据，必须是写入所有 replica 之后，才能认为是写成功了。
- 在 producer 端设置 retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了。

- [ ] 我们生产环境就是按照上述要求配置的，这样配置之后，至少在 Kafka broker 端就可以保证在 leader 所在 broker 发生故障，进行 leader 切换时，数据不会丢失。

生产者环节的高可靠

如果按照上述的思路设置了 acks=all，一定不会丢，

此时，leader 接收到消息，要等待所有的 follower都同步到了消息之后，才认为本次写成功了。

如果没满足这个条件，生产者会自动不断地重试，重试无限次。

如何保证消息的顺序性?

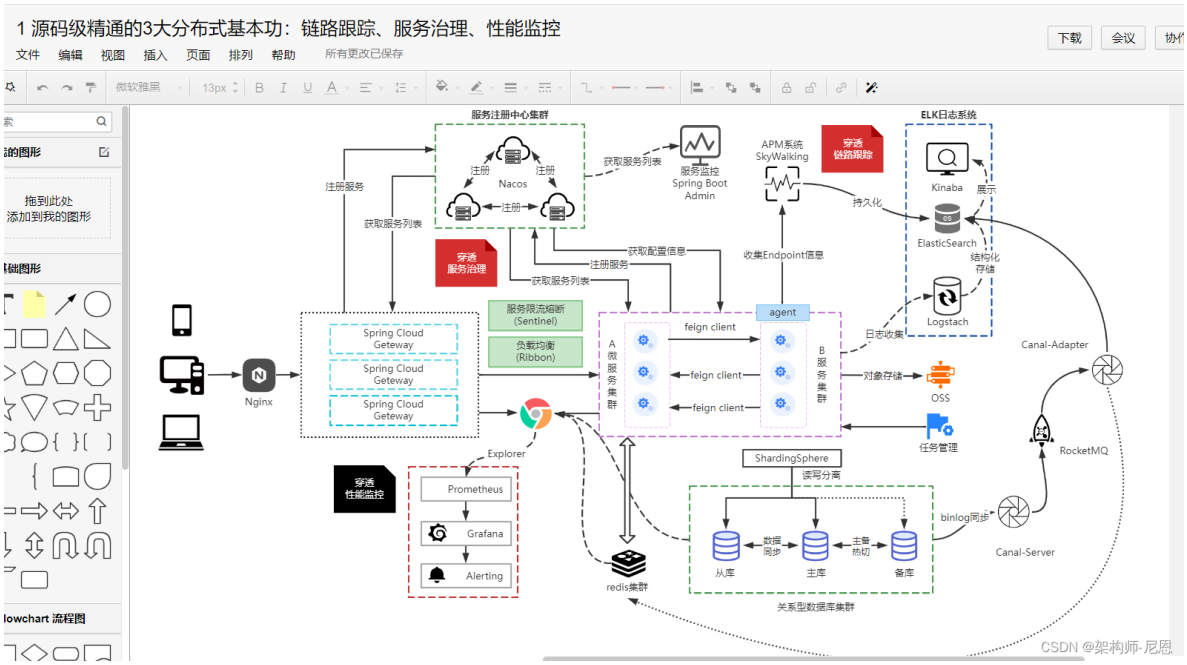
这个使用场景，太普遍，

比如，一般通过mysql binlog 同步 cache，es 的数据

在 mysql 里增删改一条数据，对应出来了增删改 3 条 binlog 日志，接着这三条 binlog 发送到 MQ 里面，

在消费的时候，得保证消息的顺序行，得依次执行，

如果 binlog 日志的日志为：增加、修改、删除；如果是换了顺序给执行成删除、修改、增加，不全错了么。



RabbitMQ解决方案

方案1：

拆分多个queue，每个queue一个consumer，发送的时候， 根据主键路由，保证同一个主键的消息，路由到同一个queue

不好的地方，多出一些queue

方案2：

一个queue但是对应一个consumer，然后这个consumer做个内存队列，一个队列由一个线程负责消费。

保证同一个主键的消息，路由到同一个内存队列

Kafka解决方案

消费端做个内存队列，，具有相同 key 的数据都到同一个内存 queue；

然后对于 N 个线程，每个线程分别消费一个内存 queue 即可，这样就能保证顺序性。

如何处理消息推积？

为什么产生消息堆积？

大多是因为 Consumer 出问题了，没有及时发现，或者故障恢复需要较长的时间，导致大量消息积压在 MQ 中。

消息堆积会有什么后果呢？

消息被丢弃

例如 RabbitMQ有一个消息过期时间 TTL，过期的消息会被扔掉，这样消息就彻底没有了。

磁盘满了

如果堆积量太大，可能导致磁盘空间不足，那么新消息就进不来了。

海量消息待处理

如果消息没过期，并且磁盘空间也够用，那么就是产生海量消息等待被消费，Consumer 的噩梦。

如何应对呢？

消息被丢弃的情况

首先，要实现防止消息过期问题，不应该设置过期时间。

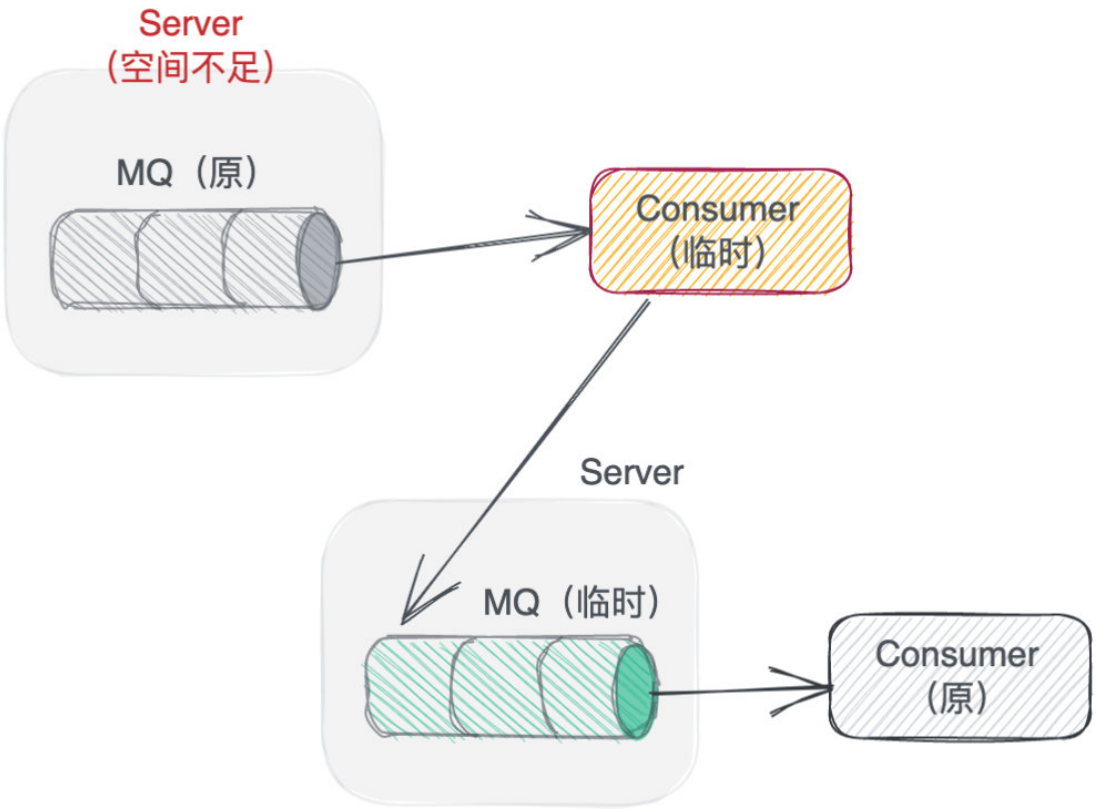
如果就是设置了过期时间，导致了消息丢失，怎么补救呢？

那就只能在夜深人静，趁着访问量最低的时候，写一个临时程序来补消息了。

例如有订单消息丢了，那就需要找出哪些订单消息丢了，然后重新发到队列。

磁盘不足的情况

系统通常都是有监控的，达到空间阈值时就会发警报，这时就要马上处理了。



例如，在其他机器上创建临时的消息队列，再写一个临时的 Consumer，作为消息的中转，把消息积压队列中的消息取出来，放到临时队列里面去。

快速疏散积压的消息，让磁盘空间恢复正常水平。

快速处理海量积压消息

Consumer 恢复正常之后，面对堆积如山的消息，怎么处理呢？

如何按照之前正常情况处理的话，猴年马月才能消费完，此过程中还有新消息在不断进来。

例如，积压了 100 万条，有 3 个 Consumer，每一个每秒能处理 200 条，3 个 Consumer 每秒一共能处理 600 条。

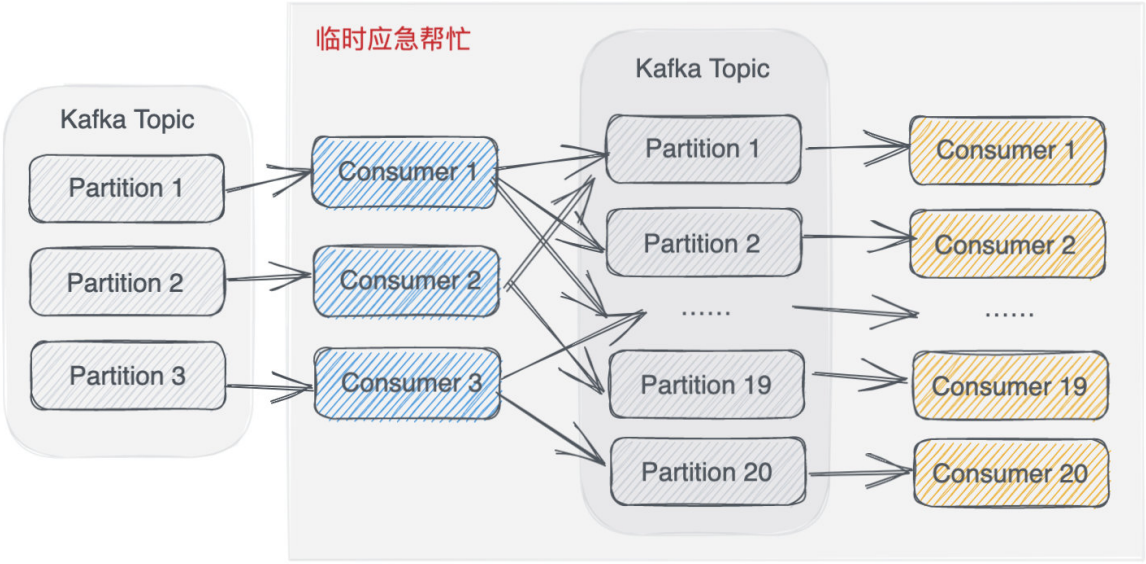
大概需要一个多小时才能处理完。

这一个多小时又会积压多少新的消息呢？

所以正常处理肯定不行，需要提速。

例如 Kafka，这个消息积压的 Topic 有 3 个 Partition，那最多就能用 3 个 Consumer，所以增加 Consumer 没有用。

还是可以使用**临时队列**的方式。



新建一个 Topic，设置为 20 个 Partition

Consumer 不再处理业务逻辑了，只负责搬运，把消息放到临时 Topic 中

这 20 个 Partition 可以有 20个 Consumer 了，它们来处理原来的业务逻辑。

这 20 个 Consumer 每秒一共能处理 4000 条了，这样几分钟就可以处理完积压的 100 万条。

这几分钟新来的消息也不会太多，所以很快就可以恢复正常水平，之后，再把整体结构恢复为原来的形式。

小结一下，消息积压还是比较麻烦的，最好是提前防范，做好硬件和消息系统的健康监控。如果出现消息丢失，就要人工查找丢失的消息，然后补上。在消费不过来的时候，可以考虑使用临时队列作为中转，提升处理能力。

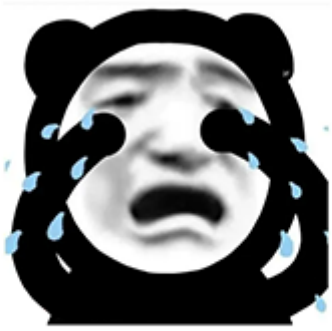
网易一面，痛失30K：为啥用阻塞队列，list不行吗？

说在前面

在40岁老架构师 尼恩的**读者交流群**(50+)中，最近有小伙伴拿到了一线互联网企业如网易、极兔、有赞、希音、百度、美团的面试资格，遇到一几个很重要的面试题：

> 为啥要用阻塞队列，用list不行吗？

阻塞队列，是面试的绝对重点和难点。小伙伴没有答上来，痛失30K月薪。



痛失30K优质offer

这里尼恩给大家做一下系统化、体系化的线程池梳理，使得大家可以充分展示一下大家雄厚的 “技术肌肉”，**让面试官爱到 “不能自已、口水直流”。**

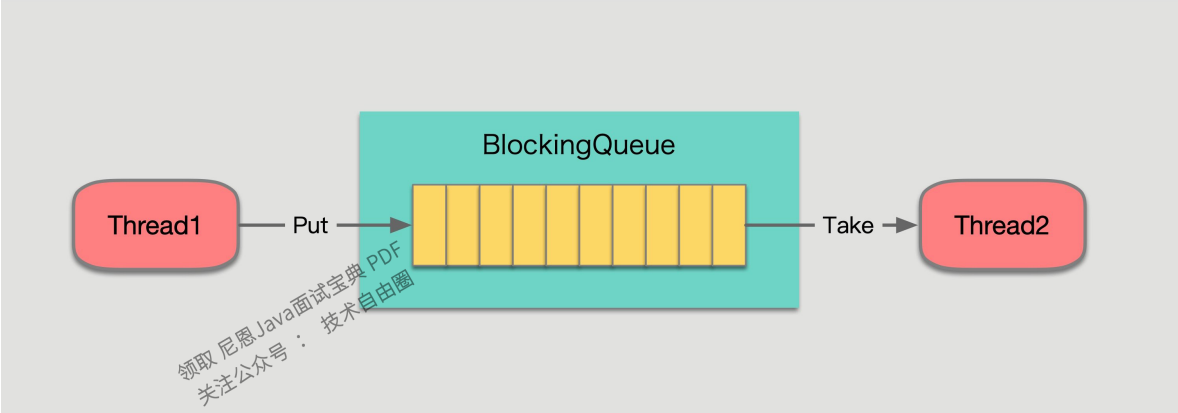
也一并把这个题目以及参考答案，收入咱们的 《尼恩Java面试宝典》V89版本，供后面的小伙伴参考，提升大家的 3高 架构、设计、开发水平。

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，请关注公众号【技术自由圈】领取，暗号：领电子书

1、什么是阻塞队列?

阻塞队列是一种队列，阻塞队列是一种特殊的队列。

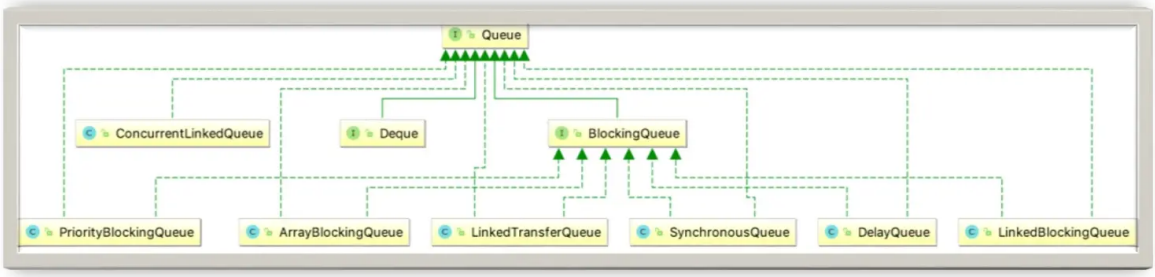
阻塞队列是一种可以在多线程环境下使用，并且支持阻塞等待的队列。



线程 1 往阻塞队列中添加元素，当阻塞队列是满的，线程 1就会阻塞，直到队列不满

线程 2 从阻塞队列中移除元素，当阻塞队列是空的，线程 2 会阻塞，直到队列不空；

2、主要并发队列关系图



上图展示了 Queue 最主要的实现类，可以看出 Java 提供的线程安全的队列（也称为并发队列）分为**阻塞队列和非阻塞队列**两大类。

BlockingQueue 下面有 6 种最主要的阻塞队列实现，分别是

- ArrayBlockingQueue、
- LinkedBlockingQueue、
- SynchronousQueue、
- DelayQueue、
- PriorityBlockingQueue
- LinkedTransferQueue。

非阻塞并发队列的典型例子是 ConcurrentLinkedQueue，这个类不会让线程阻塞，利用 CAS 保证了线程安全。

我们可以根据需要自由选取阻塞队列或者非阻塞队列来满足业务需求。

还有一个和 Queue 关系紧密的 Deque 接口，它继承了 Queue，如代码所示：

```
public interface Deque<E> extends Queue<E> {//...}
```

Deque 的意思是双端队列，音标是 [dek]，是 double-ended-queue 的缩写，它从头和尾都能添加和删除元素；而普通的 Queue 只能从一端进入，另一端出去。这是 Deque 和 Queue 的不同之处，Deque 其他方面的性质都和 Queue 类似。

3、阻塞队列和 List、Set 的区别是什么?

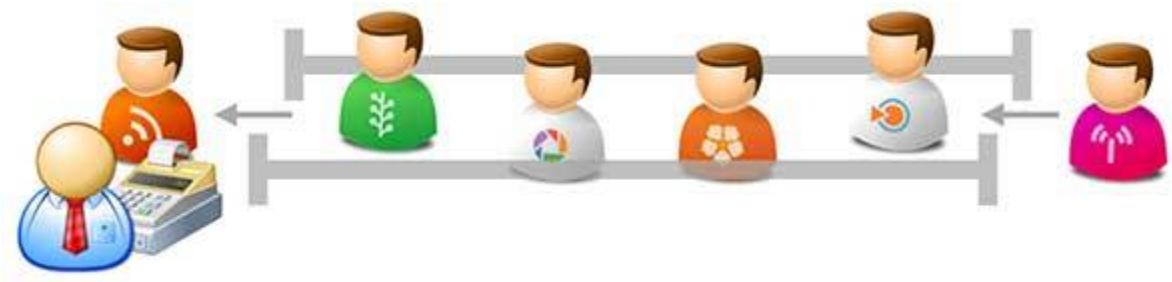
阻塞队列和 List、Set 一样都继承自 Collection。

阻塞队列它和 List 的区别在于，List 可以在任意位置添加和删除元素。

而阻塞队列属于 Queue 队列的一种，Queue 只有两个操作：

- 把元素添加到队列末尾；
- 从队列头部取出元素。

超市的收银台就是一个队列：



我们常用的 LinkedList 就可以当队列使用，实现了 Dequeue 接口，还有 ConcurrentLinkedQueue，他们都属于非阻塞队列。

4、阻塞队列和普通Queue 队列的区别是什么?

阻塞队列和一般的队列的区别就在于：

- 多线程环境支持，多个线程可以安全的访问队列
- 支持生产和消费等待，多个线程之间互相配合，在某些情况下会挂起线程，一旦条件满足，被挂起的线程又会自动被唤醒。
- 当阻塞队列是空的，消费线程会阻塞，从队列中获取元素的操作将会被阻塞，直到队列不空；
- 当阻塞队列是满的，生产线程就会阻塞，往队列里添加元素的操作将会被阻塞，直到队列不满

5、阻塞队列的作用

阻塞队列，也就是 BlockingQueue，它是一个接口，如代码所示：

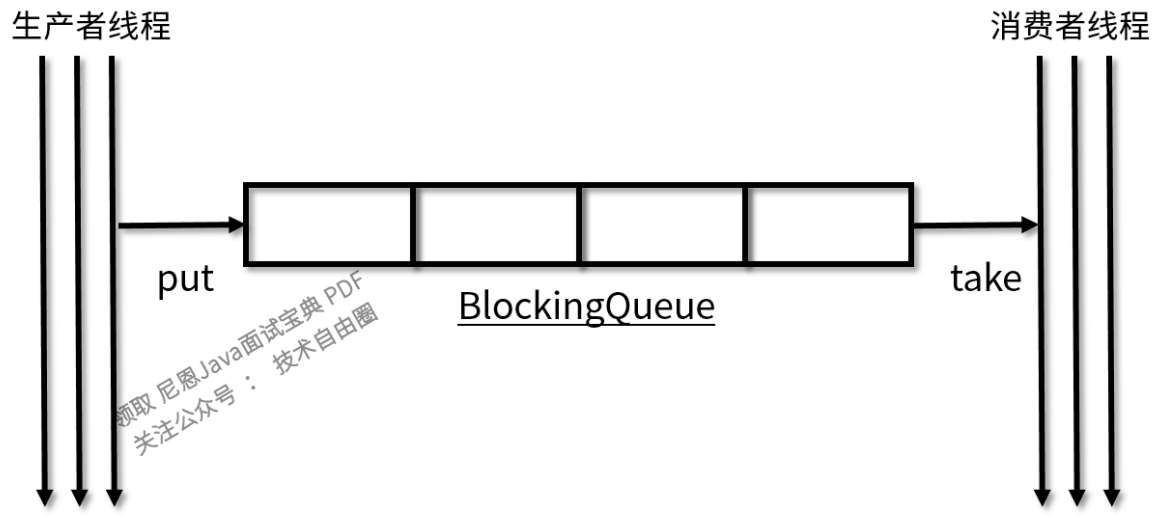
```
public interface BlockingQueue<E> extends Queue<E>{...}
```

BlockingQueue 继承了 Queue 接口，是队列的一种。

Queue 和 BlockingQueue 都是在 Java 5 中加入的。

BlockingQueue 是线程安全的，在很多场景下都可以利用线程安全的队列来优雅地**解决业务自身的线程安全问题。**

比如说，使用生产者/消费者模式的时候，生产者只需要往队列里添加元素，而消费者只需要从队列里取出它们就可以了，如图所示：



在图中，左侧有三个生产者线程，它会把生产出来的结果放到中间的阻塞队列中，而右侧的三个消费者也会从阻塞队列中取出它所需要的内容并进行处理。因为阻塞队列是线程安全的，所以生产者和消费者都可以是多线程的，不会发生线程安全问题。

既然队列本身是线程安全的，队列可以安全地从一个线程向另外一个线程传递数据，所以生产者/消费者直接使用线程安全的队列就可以，而不需要自己去考虑更多的线程安全问题。这也就意味着，考虑锁等线程安全问题的重任从“你”转移到了“队列”上，降低了开发的难度和工作量。

同时，**队列还能起到一个隔离的作用。**

比如说开发一个银行转账的程序，那么生产者线程不需要关心具体的转账逻辑，只需要把转账任务，如账户和金额等信息放到队列中就可以，而不需要去关心银行这个类如何实现具体的转账业务。而作为银行这个类来讲，它会去从队列里取出来将要执行的具体的任务，再去通过自己的各种方法来完成本次转账。

这样就实现了具体任务与执行任务类之间的解耦，任务被放在了阻塞队列中，而负责放任务的线程是无法直接访问到银行具体实现转账操作的对象的，实现了隔离，提高了安全性。

6、阻塞队列的功能

阻塞队列区别于其他类型的队列的最主要的特点就是“阻塞”这两个字，

所以下面重点介绍阻塞功能：**阻塞功能使得生产者和消费者两端的能力得以平衡，当有任何一端速度过快时，阻塞队列便会把过快的速度给降下来。**

7、阻塞队列的核心方法

方法类型	抛出异常	特殊值	阻塞	超时
插入	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除	remove()	poll()	take()	poll(time,unit)
检查	element	peek	不可用	不可用

1. 抛异常的方法 就是在插入满了之后，会报一个异常，remove一样，element是检查队头的元素或者是否为空。
2. 特殊值的方法是在插入满之后返回值变成了false而不是一个异常，取出失败的时候返回null。
3. 阻塞方法是在插入满之后把这个方法阻塞，一直等待队列空出来一个之后再进行加入，会出现一直等待，也可能出现饥饿现象。
4. 超时方法的话，当阻塞队列满时，队列会阻塞生产者线程一定时间，超过限时后生产者线程会退出。

实现阻塞最重要的两个方法是 take 方法和 put 方法。

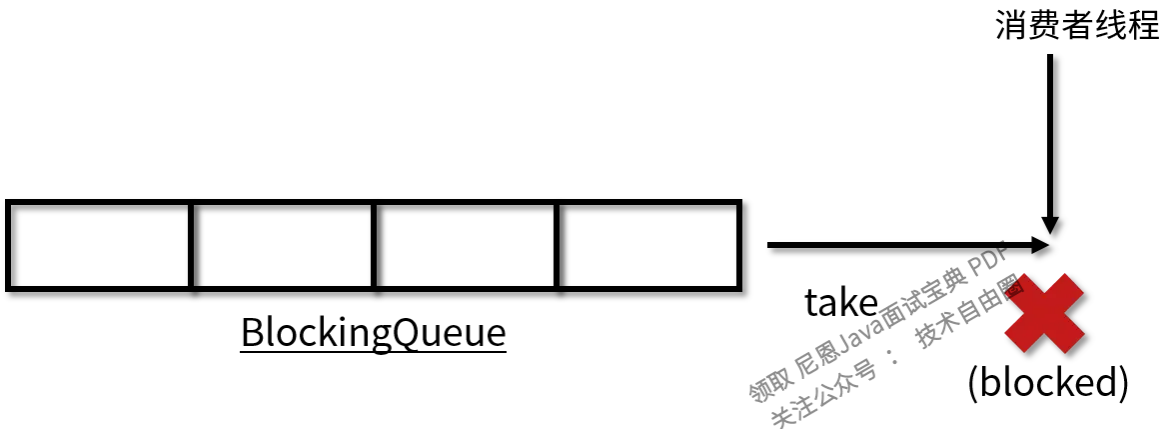
7.1 take 方法

take 方法的功能是获取并移除队列的头结点，通常在队列里有数据的时候是可以正常移除的。

可是一旦执行 take 方法的时候，队列里无数据，则阻塞，直到队列里有数据。

一旦队列里有数据了，就会立刻解除阻塞状态，并且取到数据。

过程如图所示：

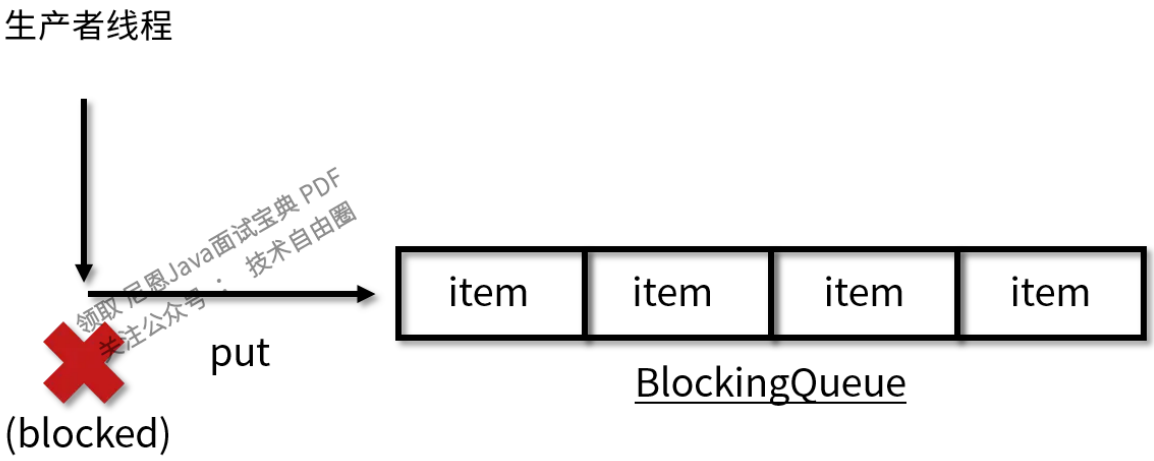


7.2 put 方法

put 方法插入元素时，如果队列没有满，那就和普通的插入一样是正常的插入，但是如果队列已满，那么就无法继续插入，则阻塞，直到队列里有了空闲空间。

如果后续队列有了空闲空间，比如消费者消费了一个元素，那么此时队列就会解除阻塞状态，并把需要添加的数据添加到队列中。

put 过程如图所示：



以上过程中的阻塞和解除阻塞，都是 BlockingQueue 完成的，不需要我们自己处理。

7.3 是否有界（容量有多大）

此外，阻塞队列还有一个非常重要的属性，那就是容量的大小，分为有界和无界两种。

- 有的阻塞队列是无界的，无界队列意味着里面可以容纳非常多的元素，例如 LinkedBlockingQueue 的上限是 Integer.MAX_VALUE，**约为 2 的 31 次方，是非常大的一个数**，可以近似认为是无限容量，因为几乎无法把这个容量装满。
- 但是有的阻塞队列是有界的，例如 ArrayBlockingQueue 如果容量满了，也**不会扩容**，所以一旦满了就无法再往里放数据了。

尼恩提示

更多的 JUC 高并发知识，请参见《Java 高并发核心编程 卷2 加强版》

说在后面

问题回答到这里，已经20分钟过去了，

既然 **对阻塞队列 表达得那么娴熟**，一定是遇到过很多的高并发场景，解决很多高并发问题，那么一定是技术大佬、技术高手，面试官已经爱到 “不能自已、口水直流” 啦。

offer，当然也就来了。

痛失网易30K之二：看你牛逼轰轰，请写一个阻塞队列

说在前面

在40岁老架构师 尼恩的**读者交流群**(50+)中，最近有小伙伴拿到了一线互联网企业如网易、极兔、有赞、希音、百度、美团的面试资格，遇到2个很重要的面试题：

> 第一弹：为啥要用阻塞队列，用list不行吗？
>
> 第二弹：手写一个阻塞队列

阻塞队列，是面试的绝对重点和难点。

小伙伴 第一弹没有回答好，面试官又来了第二弹要求手写一个阻塞队列，**又没有写出来**.....

小伙伴和尼恩说，阻塞队列虽然天天用，但是怎么实现， 还真没想过，还是要求手写.....，当时就懵逼了

于是30K的优质offer，白白就溜走了。

为了让后面的小伙伴不在同一个地方躺坑。

这里尼恩给大家做一下系统化、体系化的线程池梳理，使得大家可以充分展示一下大家雄厚的 “技术肌肉”，**让面试官爱到 “不能自已、口水直流”。**

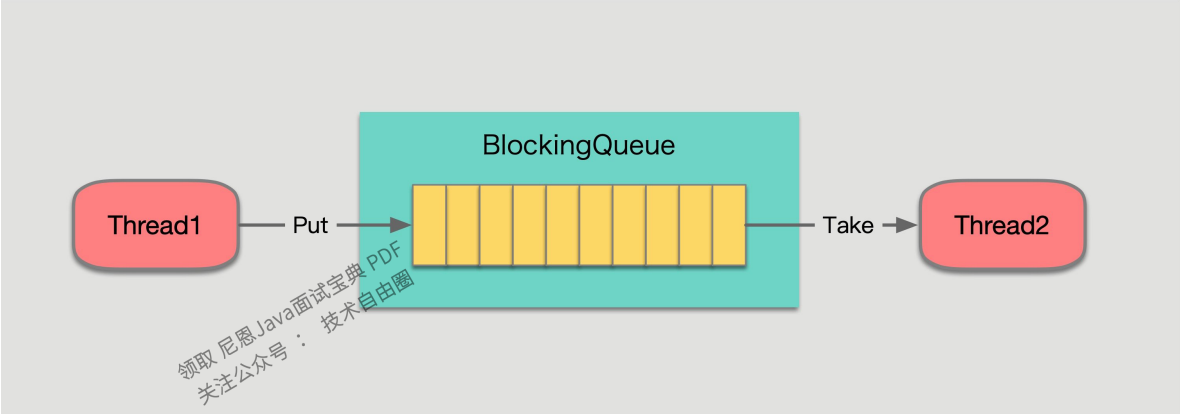
也一并把这个题目以及参考答案，收入咱们的 《尼恩Java面试宝典》V91版本，供后面的小伙伴参考，提升大家的 3高 架构、设计、开发水平。

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，请关注公众号【技术自由圈】领取，暗号：领电子书

1、什么是阻塞队列?

阻塞队列是一种队列，阻塞队列是一种特殊的队列。

阻塞队列是一种可以在多线程环境下使用，并且支持阻塞等待的队列。



线程 1 往阻塞队列中添加元素，当阻塞队列是满的，线程 1就会阻塞，直到队列不满

线程 2 从阻塞队列中移除元素，当阻塞队列是空的，线程 2 会阻塞，直到队列不空；

2、阻塞队列的作用

阻塞队列，也就是 BlockingQueue，它是一个接口，如代码所示：

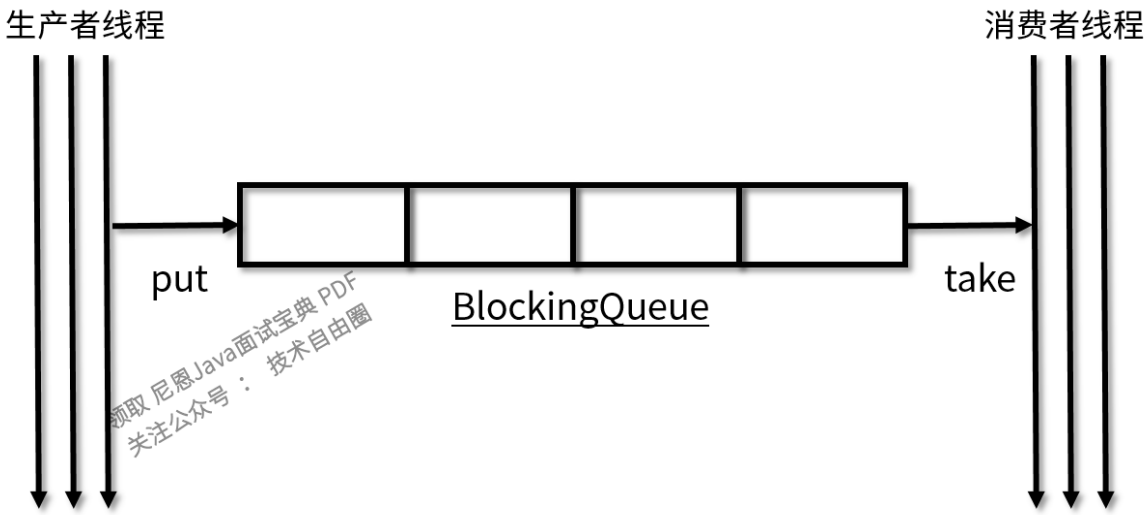
```
public interface BlockingQueue<E> extends Queue<E>{...}
```

BlockingQueue 继承了 Queue 接口，是队列的一种。

Queue 和 BlockingQueue 都是在 Java 5 中加入的。

BlockingQueue 是线程安全的，在很多场景下都可以利用线程安全的队列来优雅地**解决业务自身的线程安全问题。**

比如说，使用生产者/消费者模式的时候，生产者只需要往队列里添加元素，而消费者只需要从队列里取出它们就可以了，如图所示：



阻塞队列区别于其他类型的队列的最主要的特点就是“阻塞”这两个字，

阻塞功能使得生产者和消费者两端的能力得以平衡，当有任何一端速度过快时，阻塞队列便会把过快的速度给降下来。

3、阻塞队列的核心方法

方法类型	抛出异常	特殊值	阻塞	超时
插入	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除	remove()	poll()	take()	poll(time,unit)
检查	element	peek	不可用	不可用

1、抛异常的方法 就是在插入满了之后，会报一个异常，remove一样，element是检查队头的元素或者是否为空。
2、特殊值的方法是在插入满之后返回值变成了false而不是一个异常，取出失败的时候返回null。
3、阻塞方法是在插入满之后把这个方法阻塞，一直等待队列空出来一个之后再进行加入，会出现一直等待，也可能出现饥饿现象。
4、超时方法的话，当阻塞队列满时，队列会阻塞生产者线程一定时间，超过限时后生产者线程会退出。

实现阻塞最重要的两个方法是 take 方法和 put 方法。

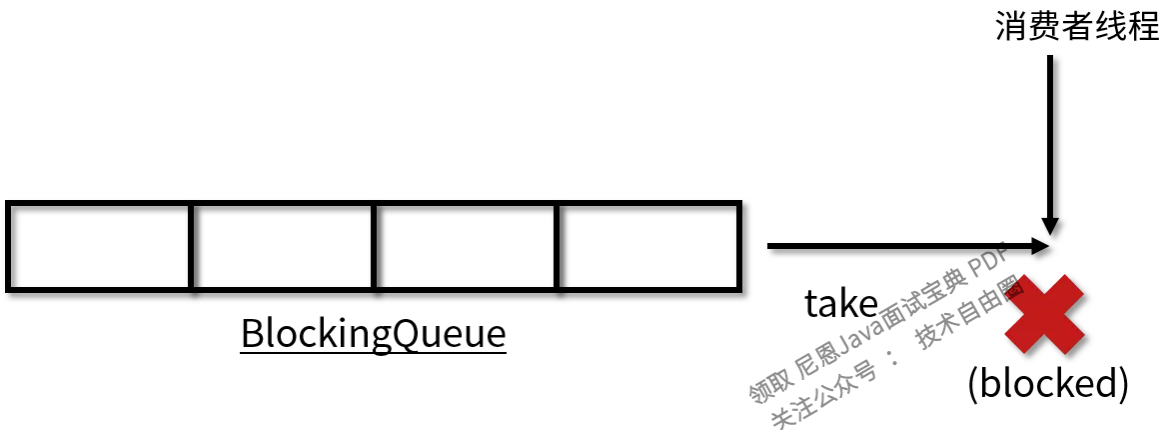
3.1 take 方法

take 方法的功能是获取并移除队列的头结点，通常在队列里有数据的时候是可以正常移除的。

可是一旦执行 take 方法的时候，队列里无数据，则阻塞，直到队列里有数据。

一旦队列里有数据了，就会立刻解除阻塞状态，并且取到数据。

过程如图所示：

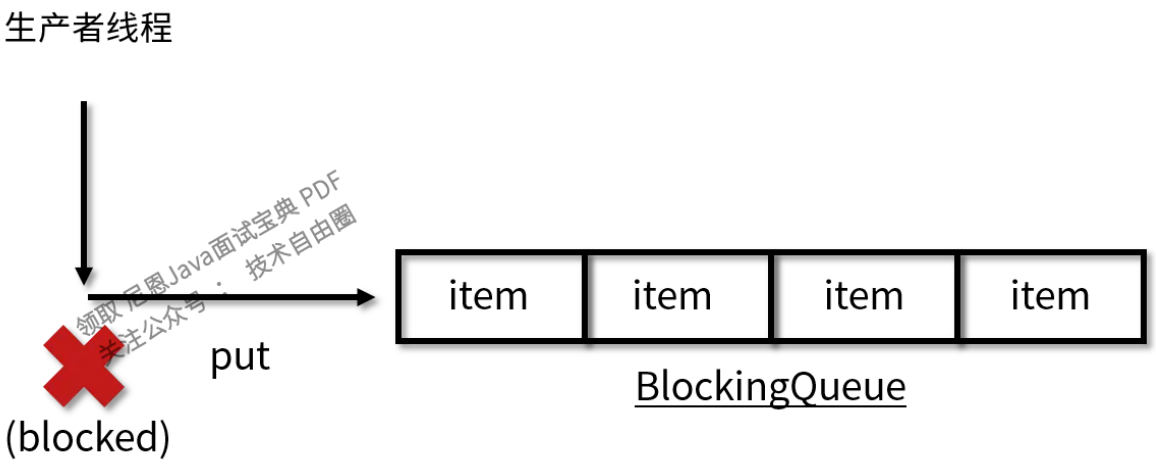


3.2 put 方法

put 方法插入元素时，如果队列没有满，那就和普通的插入一样是正常的插入，但是如果队列已满，那么就无法继续插入，则阻塞，直到队列里有了空闲空间。

如果后续队列有了空闲空间，比如消费者消费了一个元素，那么此时队列就会解除阻塞状态，并把需要添加的数据添加到队列中。

put 过程如图所示：



以上过程中的阻塞和解除阻塞，都是 BlockingQueue 完成的，不需要我们自己处理。

4、手写模拟实现一个阻塞队列

手写模拟实现一个阻塞队列，可以基于数组实现的阻塞队列，如何手写呢?

我们先从功能设计开始：

1. 首先它是一个队列，队列需要具备入队、出队的能力， 所以，设计两个方法 put、take
2. put操作的时候，需要在队列已满时，对入队的请求进行阻塞，当队列有剩余空间时，释放入队请求;
3. take操作的时候，在队列为空时，需要对出队的请求进行阻塞，当队列中有元素时，释放出队请求;
4. 由于ArrayBlockingQueue是一个在多线程情况下使用的数据结构，需要保证它的操作的线程安全性，所以，这里需要用到锁

4.1 用数组实现队列

如何用数组实现数据的入队出队操作呢?

如何写入呢?

这个简单，可以通过一个index字段存储当前数组下一个写入的位置。

如何处理出队呢?

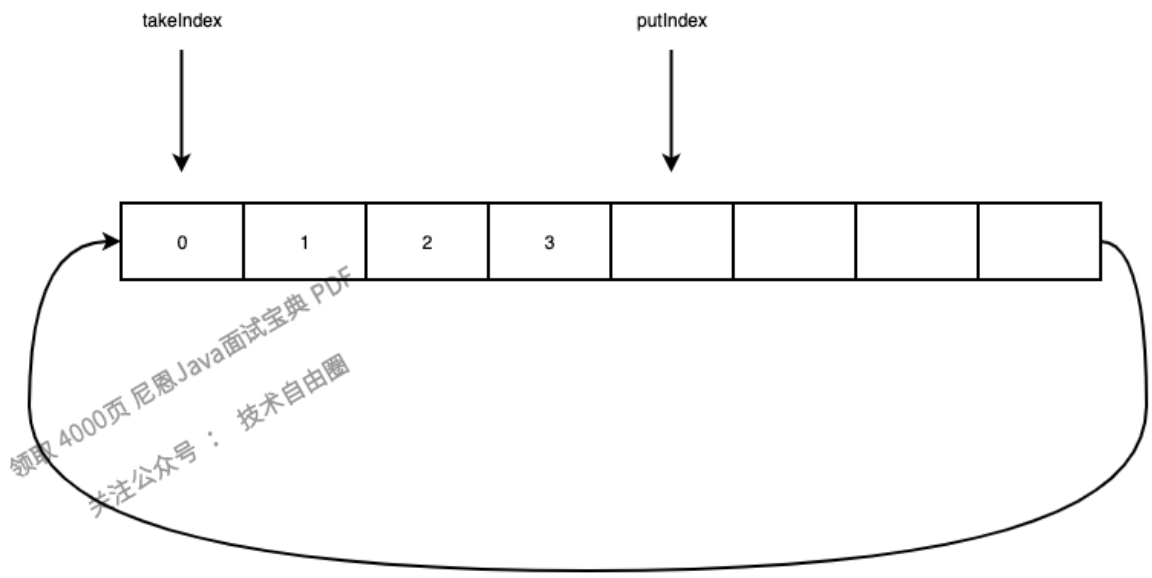
一种简单的方法： 简单的返回数组第一个元素，并且把后面所有的元素向前移动一位。

如果这么操作，出队时会移动大量的元素，它的时间复杂度是O(n)。

那有没有更高效的方案呢?

还有另一个循环数组的方案，我们通过两个int字段，分别记录下一个要入队和下一个要出队的元素的位置，当入队到数组末尾时，从0开始，同样当出队到末尾时，也从0开始。

另外当队列为空和队列已满的时候，takeIndex和putIndex都指向相同的位置，所以为了进行区分，我们可以用一个count字段存储队列元素数量，这样当count=0的时候说明队列为0，count=数组容量的时候说明队列已满



4.2 使用 synchronized 实现

由于 synchronized 是同一把锁，所以使用 notify() 可能会唤醒非目标线程，notifyAll() 唤醒全部线程则会带来大量的 CPU 上下文交换和锁竞争

```
package com.crazymakercircle.queue;

public class ArrayBlockingQueue{
    private Object[] array;     //数组
    private int takeIndex;          //头
    private int putIndex;           //尾
    private volatile int count; //元素个数

    public ArrayBlockingQueue(int capacity){
        this.array = new Object[capacity];
    }

    //写入元素
    public synchronized void put(Object o) throws InterruptedException{
        //当队列满时，阻塞
        while(count == array.length){
            this.wait();
        }
        array[putIndex++] = o;
        if(putIndex ==array.length){
            putIndex = 0;
        }
        count++;
        //唤醒线程
        this.notifyAll();
    }

    //取出元素
```

```
    public synchronized Object take() throws InterruptedException{
        //当队列为空，阻塞
        while(count == 0){
            this.wait();
        }
        Object o = array[takeIndex++];
        if(takeIndex == array.length){
            takeIndex = 0;
        }
        count--;
        //唤醒线程
        this.notifyAll();
        return o;
    }
}
```

4.3 使用 ReentrantLock

可以使用 Condition 指定要唤醒的线程，所以效率高

```
package com.crazymakercircle.queue;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class ArrayBlockingQueueReentrantLock{
    private Object[] array;     //数组
    private int takeIndex;          //头
    private int putIndex;           //尾
    private volatile int count; //元素个数
    private ReentrantLock lock = new ReentrantLock();   //锁
    private Condition notEmpty = lock.newCondition();   //非空
    private Condition notFull = lock.newCondition();    //非满

    public ArrayBlockingQueueReentrantLock(int capacity){
        this.array = new Object[capacity];
    }

    //写入元素
    public void put(Object o) throws InterruptedException{
        try{
            lock.lock();
            //当队列满时，阻塞
            while(count == array.length){
                notFull.wait();
            }
            array[putIndex++] = o;
            if(putIndex == array.length){
                putIndex = 0;
            }
            count++;
            //唤醒线程
            notEmpty.notifyAll();
        }finally{
            lock.unlock();
        }
    }
```

```
    //取出元素
    public Object take() throws InterruptedException{
         lock.lock();
         try{
            //当队列为空，阻塞
            while(count == 0){
                notEmpty.wait();
            }
            Object o = array[takeIndex++];
            if(takeIndex == array.length){
                takeIndex = 0;
            }
            count--;
            //唤醒线程
            notFull.notifyAll();
            return o;
         }finally{
            lock.unlock();
         }
    }
}
```

最终，咱们要回到源码

接下来，拆解JUC源码中，ArrayBlockingQueue的实现步骤

5、拆解ArrayBlockingQueue实现步骤

我们先拆解一下问题，把拆解ArrayBlockingQueue实现步骤分成两个步骤

1. 用数组实现队列
2. 给队列加上阻塞能力和保证线程安全

5.1 用数组实现队列

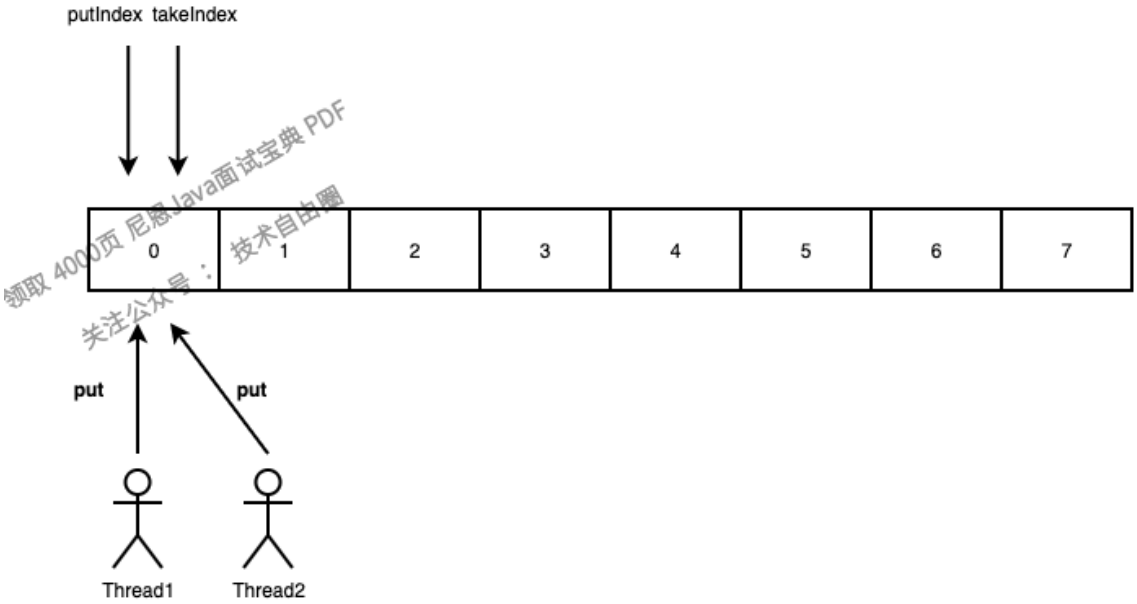
使用 takeIndex、putIndex 避免数组复制

下面代码展示了用数组实现队列的具体实现。

```
class ArrayBlockingQueue<E> {
    final Object[] items;
    int takeIndex;
    int putIndex;
    int count;
    public ArrayBlockingQueue(int capacity) {
        if (capacity <= 0)
            throw new IllegalArgumentException();
        this.items = new Object[capacity];
    }
    private void enqueue(E e) {
        Object[] items = this.items;
        items[putIndex] = e;
        if (++putIndex == items.length) putIndex = 0;
        count++;
    }
```

```
    private E dequeue() {
        Object[] items = this.items;
        E e = (E) items[takeIndex];
        items[takeIndex] = null;
        if (++takeIndex == items.length) takeIndex = 0;
        count--;
        return e;
    }
}
```

5.2 实现条件阻塞和线程安全



「在队列已满时，对入队的请求进行阻塞，当队列有剩余空间时，释放入队请求」这个需求本质上是一个条件等待的特例，写入的条件是队列不满，不满足条件的时候需要等待，直到满足条件为止。

在Java中，实现条件等待有synchronized+Object.wait和Lock+Condition.await两种方式，这里不用synchronized方案，是因为

1. synchronized不支持interrupt
2. synchronized无法支持多个条件

通过Lock和Condition的方案，还能够保证线程安全，因为上面的环形数组实现中，线程间共享的变量有items数组、takeIndex、putIndex、count，线程安全涉及到原子性可见性重排序几个方面，通过Lock类加锁可以对共享变量的读写操作进行保护。

定义阻塞的Lock对象和Condition，条件分为不满和不空两个条件。

```
class ArrayBlockingQueue<E> {
    final Object[] items;
    int takeIndex;
    int putIndex;
    int count;
    ReentrantLock lock;
    private final Condition notEmpty;
    private final Condition notFull;
    public ArrayBlockingQueue(int capacity) {
```

```
        if (capacity <= 0)
            throw new IllegalArgumentException();
        this.items = new Object[capacity];
        // 创建lock对象
        lock = new ReentrantLock();
        // 创建非空的Condition
        notEmpty = lock.newCondition();
        // 创建不满的Condition
        notFull =  lock.newCondition();
    }
}
```

以入队操作添加实现为例，能够入队的条件是队列不满，也就是count < items.length，不能入队的条件反过来就是count == items.length。

当满足条件后，我们就可以入队了，入队之后，还需要唤醒等待出队的线程。

5.3 put方法的流程为

1. 先加锁
2. 在锁中while循环判断条件是否满足，不满足调用notFull.await()，await()方法会释放锁，被其他线程signal唤醒后会重新抢锁，再次获得锁后会继续走到while循环判断条件的地方。
3. 如果条件已经满足，则执行入队操作
4. 入队完之后调用notEmpty.signal()唤醒一个等待notFull条件的线程
5. finally中释放锁

```
public void put(E e) throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
            notFull.await();
        enqueue(e);
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

方法中还有一些小细节

1. put方法中，为什么要先用一个声明一个lock局部变量呢?

```
ReentrantLock lock = this.lock;
```

这是因为如果不使用局部变量，后面所有使用实例变量的调用，在字节码指令层面需要变成先调用aload 0获取到this，再调用getField指令获取字段值，再进行其他操作。

而先把lock存到局部变量中，后面所有的获取lock就可以变成一个aload xxx指令，从而节省了指令数量，也就会加快方法的执行速度。

2. 为什么while循环需要放在锁内呢?

如果不放在锁内，则可能会出现多个线程同时看到满足条件，进而去加锁入队。

虽然入队还是在临界区，但是会出现队列已满，仍然在执行入队操作的情况。

这个问题和单例的double check locking中少些一个check的问题类似。

5.4 take方法的流程为

take方法是和put相对应的出队方法，和put流程基本一致

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();
        E element = dequeue();
        notFull.signal()
        return element;
    } finally {
        lock.unlock();
    }
}
```

尼恩提示

更多的 JUC 高并发知识，请参见《Java 高并发核心编程 卷2 加强版》

说在后面

如果手写到后面一个版本，并且能把 **实例变量的调用的性能优化**，**while循环为何要放在锁内**等这些高超的技术点写出来，那么太牛了。

那么面试官一定将你归为**技术大佬、技术高手**，面试官已经爱到 “不能自已、口水直流” 啦。

offer，当然也就来了。

此真题收入5000页《尼恩Java面试宝典》V91版，最新的PDF找尼恩获取。

网易一面：单节点2000Wtps，Kafka高性能原理是什么？

说在前面

在40岁老架构师 尼恩的**读者交流群**(50+)中，最近有小伙伴拿到了一线互联网企业如网易、有赞、希音、百度、网易、滴滴的面试资格，遇到一几个很重要的面试题：

> 问题1：单节点2000Wtps，Kafka高性能原理是什么？
>
> 问题2：做过Kafka 进行性能压测吗？单个节点的极限处理能力是多少？是怎么做到的？

注意，单个节点的极限处理能力接近每秒 2000万 条消息，吞吐量达到每秒 600MB

那 Kafka 为什么这么快？如何做到这个高的性能？

尼恩提示，Kafka相关的问题，是开发的核心知识，又是线上的重点难题。

所以，这里尼恩给大家做一下系统化、体系化的线程池梳理，使得大家可以充分展示一下大家雄厚的“技术肌肉”，**让面试官爱到 “不能自已、口水直流”。**

也一并把这个题目以及参考答案，收入咱们的 《尼恩Java面试宝典》V100版本，供后面的小伙伴参考，提升大家的 3高 架构、设计、开发水平。

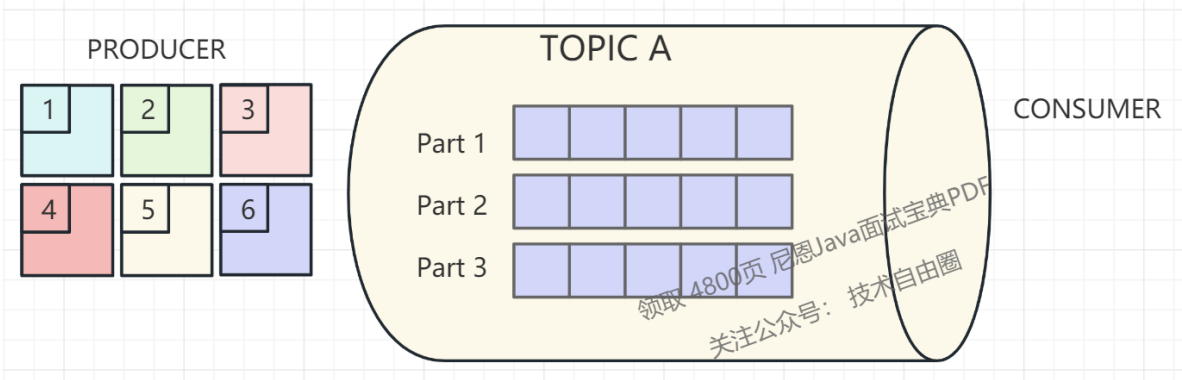
> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，请从公众号 【技术自由圈】获取。

这里，主要从这 3 个角度来分析：

- **生产端**
- **服务端 Broker**
- **消费端**

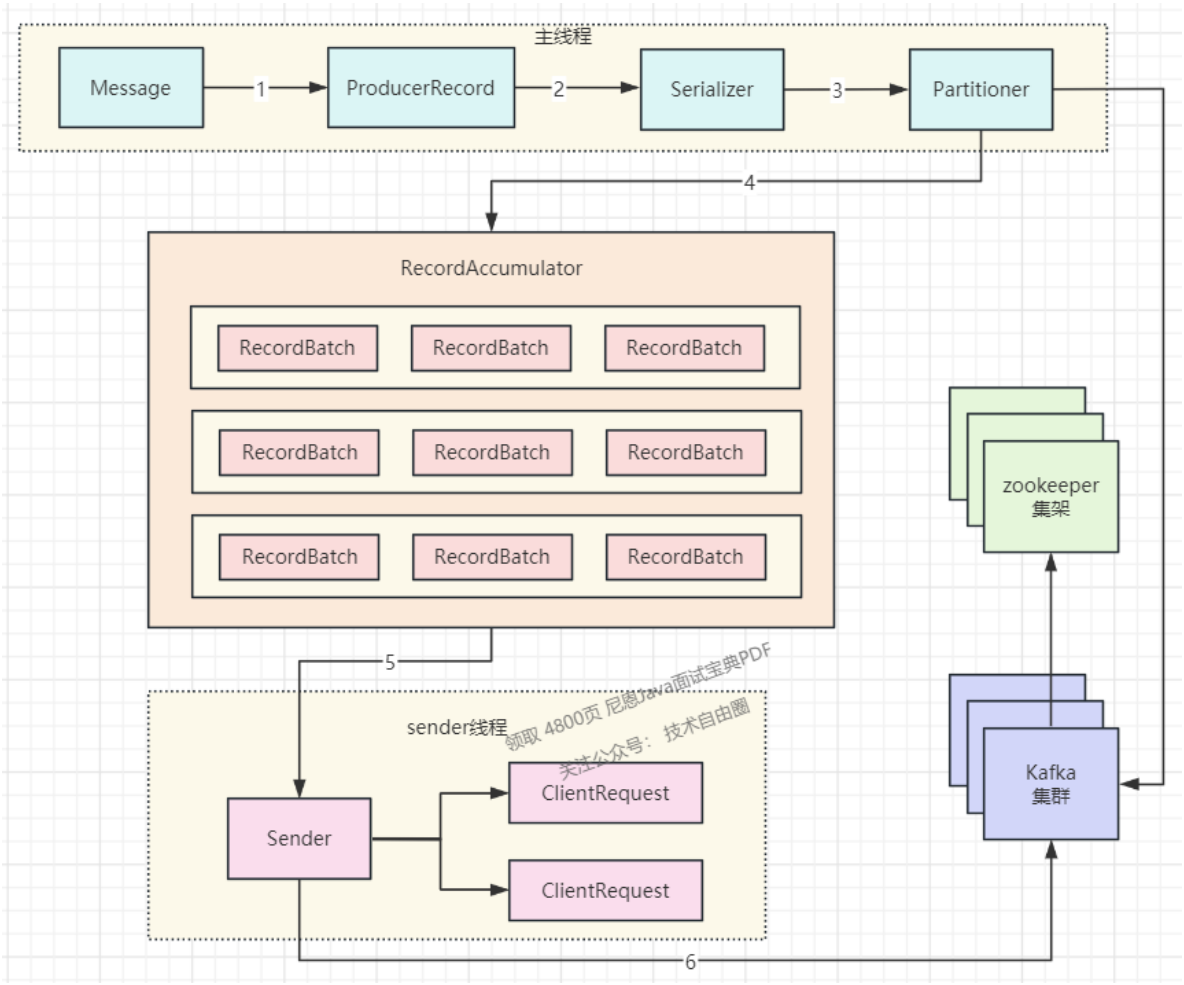
先来看下生产端发送消息，Kafka 做了哪些优化？

一、生产端 Producer



先来回顾下 Producer 生产者发送消息的流程：

Kafka的源码最核心的是由client模块和core模块构成，用一幅图大致介绍一下生产者发送消息的流程。



生产者发送消息流程

1. 将消息封装成ProducerRecord对象
2. Serializer对消息的key和value做序列化
3. 根据Partitioner将消息分发到不同的分区，需要先获取集群的元数据
4. RecordAccumulator封装很多分区的消息队列，每个队列代表一个分区，每个分区里面有很多的批次，每个批次里面由多条消息组成
5. Sender会从RecordAccumulator拉取消息，封装成批次，发送请求
6. 通过网络将请求发送到kafka集群

前置知识：队列缓存+批量写入架构

队列缓存+批量写入架构，是尼恩反复强调的一个高并发写入架构，

kafka、netty都用到这个架构

kafka的生产者，也用了这个架构。 设计了一个核心组件RecordAccumulator

1. RecordAccumulator：每一个是生产上都会维护一个固定大小的内存空间，主要用于合并单条消息，进行批量发送，提高吞吐量，减少带宽消耗。
2. RecordAccumulator的大小是可配置的，可以配置buffer.memory来修改缓冲区大小，默认值为：33554432（32M）
3. RecordAccumulator内存结构分为两部分

- 第一部分为已经使用的内存，这一部分主要存放了很多的队列。每一个主题的每一个分区都会创建一个队列，来存放当前分区下待发送的消息集合。
- 第二部分为未使用的内存，这一部分分为已经池化后的内存和未池化的整个剩余内存（nonPooledAvailableMemory）。池化的内存的会根据batch.size（默认值为16K）的配置进行池化多个ByteBuffer，放入一个队列中。所有的剩余空间会形成一个未池化的剩余空间。

生产者发送消息流程源码

1. 将消息封装成ProducerRecord对象：

生产者生成某个消息后，ProducerRecord首先会经过一个或多个组成的拦截器链。

2. Serializer对消息的key和value做序列化：

当消息通过所有的拦截器之后，会进行序列化，会根据key和value的序列化配置进行序列化消息内容，生产者和消费者必须使用相同的key-value序列化方式。

```
// 消息key序列化
properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
// 消息value序列化
properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
```

3. 经过序列化后，会根据自定义的分区器或者Kafka默认的分区器进行获取消息的所属的分区。

自定义分区器可以参考下面。

Kafka默认的分区器规则：

- 1）当消息的key存在时，首先获取当前topic下的所有分区数，然后对key进行求hash值，根据hash值和分区总数进行取余，获取所属的的分区。
- 2）如果key不存在时，会根据topic获取一个递增的数值，然后通过和分区数进行取余，获取所属的分区。

Kafka默认分区器源码：

```
public class DefaultPartitioner implements Partitioner {

    private final ConcurrentMap<String, AtomicInteger> topicCounterMap = new
ConcurrentHashMap<>();

    public void configure(Map<String, ?> configs) {}

    public int partition(String topic, Object key, byte[] keyBytes, Object
value, byte[] valueBytes, Cluster cluster) {
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
        if (keyBytes == null) {
            int nextValue = nextValue(topic);
            List<PartitionInfo> availablePartitions =
cluster.availablePartitionsForTopic(topic);
            if (availablePartitions.size() > 0) {
                int part = Utils.toPositive(nextValue) %
availablePartitions.size();
                return availablePartitions.get(part).partition();
            } else {
                // no partitions are available, give a non-available partition
```

```
                return Utils.toPositive(nextValue) % numPartitions;
            }
        } else {
            // hash the keyBytes to choose a partition
            return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
        }
    }

    private int nextValue(String topic) {
        AtomicInteger counter = topicCounterMap.get(topic);
        if (null == counter) {
            counter = new AtomicInteger(ThreadLocalRandom.current().nextInt());
            AtomicInteger currentCounter = topicCounterMap.putIfAbsent(topic,
counter);
            if (currentCounter != null) {
                counter = currentCounter;
            }
        }
        return counter.getAndIncrement();
    }

    public void close() {}

}
```

自定义分区器：

```
public class CustomerPartitions implements Partitioner{
    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object
value, byte[] valueBytes, Cluster cluster) {
        int partition = 0;
        if(key == null) {
        } else {
            String keyStr = key.toString();
            if(keyStr.contains("Test")) {
                partition = 1;
            } else {
                partition = 2;
            }
        }
        return partition;
    }

    @Override
    public void close() {

    }

    @Override
    public void configure(Map<String, ?> configs) {

    }
}
```

4. 获取到消息所属的分区后，消息会被存放到消息缓冲区中（RecordAccumulator）中，

根据topic和分区可以确定一个双端队列(Deque)中，这个队列中每个节点为多个消息的合集（ProducerBatch）,新的消息会被放到队列的最后一个节点上，存放会存在多种情况。

场景一：消息大小不足16K。

首先会根据topic和分区获取所属队列的最后一个ProducerBatch，

- 如果最后一个ProducerBatch+当前消息 <= 16K时，会把当前消息存入这个ProducerBatch中，等待发送。
- 如果最后一个ProducerBatch+当前消息 > 16K时，此时消息不会放入这个ProducerBatch中，而是会向池化的队列中获取一个ByteBuffer，把这个ByteBuffer放到队列的尾部，然后把消息放入这个新增的ProducerBatch中。
- 如果最后一个ProducerBatch+当前消息 > 16K时，并且池化的队列中没有可用的ByteBuffer时，池化队列会向剩余的未使用的内存空间（nonPooledAvailableMemory）申请一个大小为16K的内存空间，添加到池化队列尾部。然后把这个新增的ByteBuffer添加到分区下的队列尾部，存储新的消息。

场景二：消息大小超过16K

- 当消息超过16K时，任何一个ProducerBatch都无法存储这个消息。此时会直接向剩余的空间（nonPooledAvailableMemory）的进行分配和当前的消息大小一样的内存空间，加到队列的尾部，然后存储消息，等待发送。
- 当剩余的空间（nonPooledAvailableMemory） < 消息大小时，nonPooledAvailableMemory会向池化队列获取空间，每次获取一个ByteBuffer(16K)，直到nonPooledAvailableMemory的空间大于或等于消息大小时。获取的ByteBuffer会经过jvm的GC垃圾回收。过程比较慢。当nonPooledAvailableMemory空间大于获取等于消息大小时，会把分配消息大小的空间放入分区队列的尾部，把消息存入这个ProducerBatch内。

5. 生产者会有一个send线程，用于不断的获取消息和发送消息。

sender线程会不断的扫描RecordAccumulator中所有的ProducerBatch，如果ProducerBatch达到batch.size（默认16K）大小或者最早的一个消息已经等待超过linger.ms（默认为0）时，这个ProducerBatch会被sender线程收集到。由于不同的topic和分区会被分到不同的Broker节点上，sender线程会把发送到相同Broker姐节点的ProducerBatch合并在一个Request请求中，一个Request请求不会超过max.request.size（默认1048576B = 1M）

6. 每个请求都会缓存在一个inFlightRequest缓冲区内，里面为每一个Broker分配了一个队列。

新的请求会放在队列尾部，每个队列最多能够容纳max.in.flight.requests.per.connection（默认值为5）个Request，队列满了不会产生新的Request。

7. selector获取到Request会发往相对应的Broker节点。Broker节点收到Request后会进行ACK确认这个Request

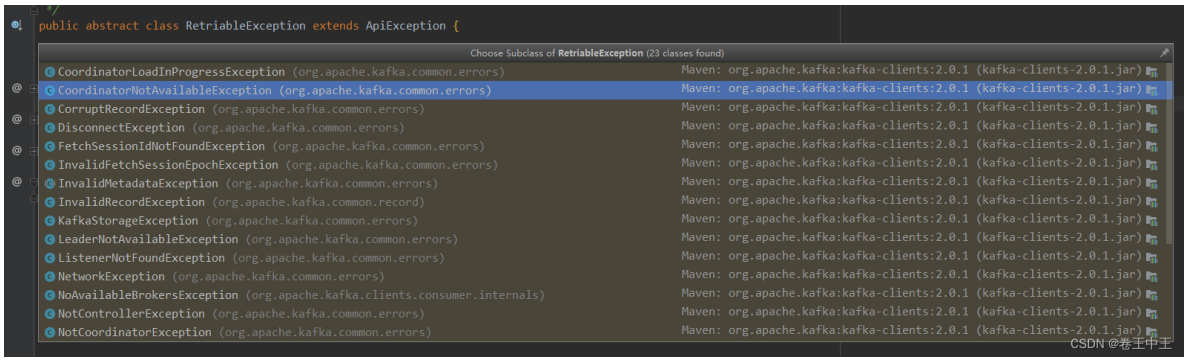
> acks 有三个配置值:[-1 , 0 , 1]
> acks = -1 表示不需要收到leader节点的ACK回复就会发送下一个Request。高吞吐，低一致性
> acks = 0 表示只需要接收到leader节点的ACK后就可以发送下一个Request。
> acks = 1 表示 需要接收到leaer节点和ISR节点的ACK后才会发送下一个Request。一致性较高

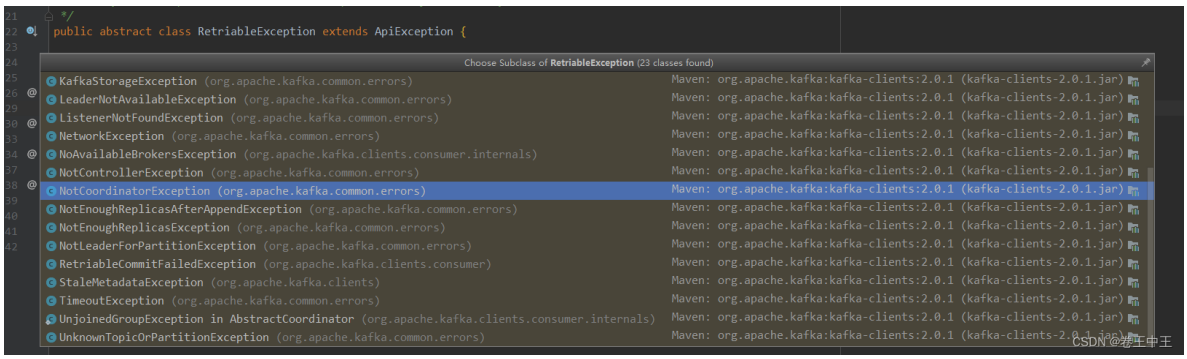
8. 当收到Broker对某个Request的ACK后，会删除inFlightRequest队列中这个Request。然后调用clear方法清除对应的ProducerBatch。

RecordAccumulator Clear清理场景：针对2.4.1.1，2.4.1.2，2.4.1.3，ProducerBatch都会标记为删除，然后放入池化队列中，不会进行GC。2.4.1.3中从nonPooledAvailableMemory获取的内存也不会归还給nonPooledAvailableMemory，任然存放在池化队列中。
针对2.4.2.1，2.4.2.2，超过16K的消息内存空间会被GC进行回收，然后作为nonPooledAvailableMemory的一部分

9. 如果发送过程中产生了异常，消息发送会存在重试机制。条件为重试次数小于指定值&&异常为RetriableException

```
private boolean canRetry(ProducerBatch batch, ProduceResponse.PartitionResponse
response) {
    return batch.attempts() < this.retries &&
        ((response.error.exception() instanceof RetriableException) ||
         (transactionManager != null && transactionManager.canRetry(response,
batch)));
}
```





生产端的高并发核心架构设计

在消息发送时候，可发现这两个亮点：**批量消息**和**自定义协议格式。**

- **批量发送**：减少了与服务端 Broker 处理请求的次数，从而提升总体的处理能力。

调用 send() 方法时，不会立刻把消息发送出去，而是缓存起来，选择恰当时机把缓存里的消息划分成一批数据，按批次发送给服务端 Broker。

- **自定义协议格式**：序列化方式和压缩格式都能减少数据体积，从而节省网络资源消耗。

各种压缩算法对比：

- 吞吐量方面：LZ4 > Snappy > zstd 和 GZIP
- 压缩比方面：zstd > LZ4 > GZIP > Snappy

Compressor name	Ratio	Compression	Decompress.
zstd 1.3.4-1	2.877	470 MB/s	1380 MB/s
zlib 1.2.11-1	2.743	110 MB/s	400 MB/s
brotli 1.0.2-0	2.701	410 MB/s	430 MB/s
quicklz 1.5.0-1	2.238	550 MB/s	710 MB/s
lzo1x2.09-1	2.108	650 MB/s	830 MB/s
lz4 1.8.1	2.101	750 MB/s	3700 MB/s
snappy 1.1.4	2.091	530 MB/s	1800 MB/s
lzf 3.6-1	2.077	400 MB/s	860 MB/s

二、服务端 Broker

Broker 的高性能主要从这 3 个方面体现：

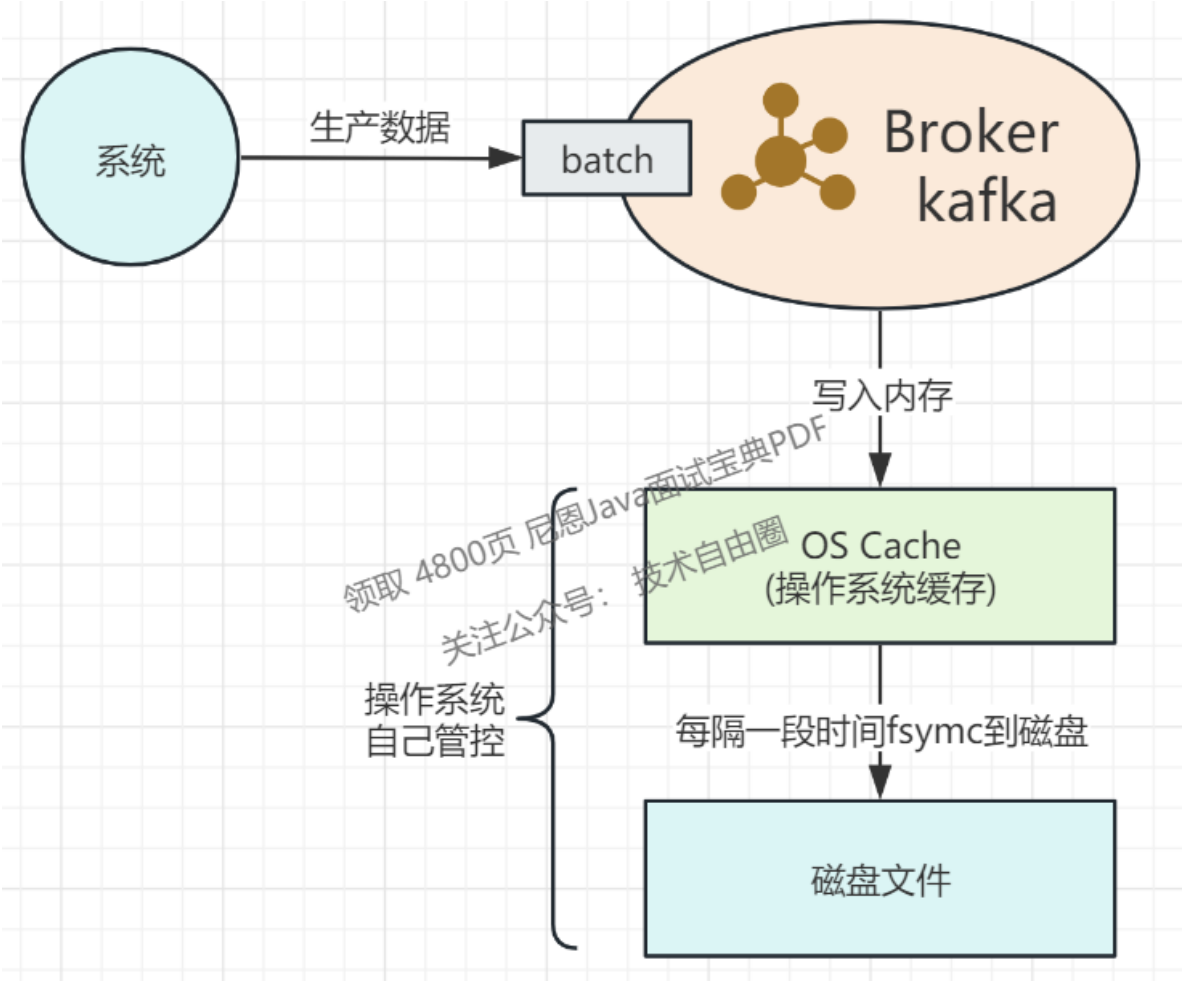
- PageCache 缓存
- Kafka 的文件布局 以及 磁盘文件顺序写入
- 零拷贝 sendfile：加速消费流程

下面展开讲讲。

1、PageCache 加速消息读写

使用 PageCache 主要能带来如下好处：

- 写入文件的时候：操作系统会先把数据写入到内存中的 PageCache，然后再一批一批地写到磁盘上，从而减少磁盘 IO 开销。

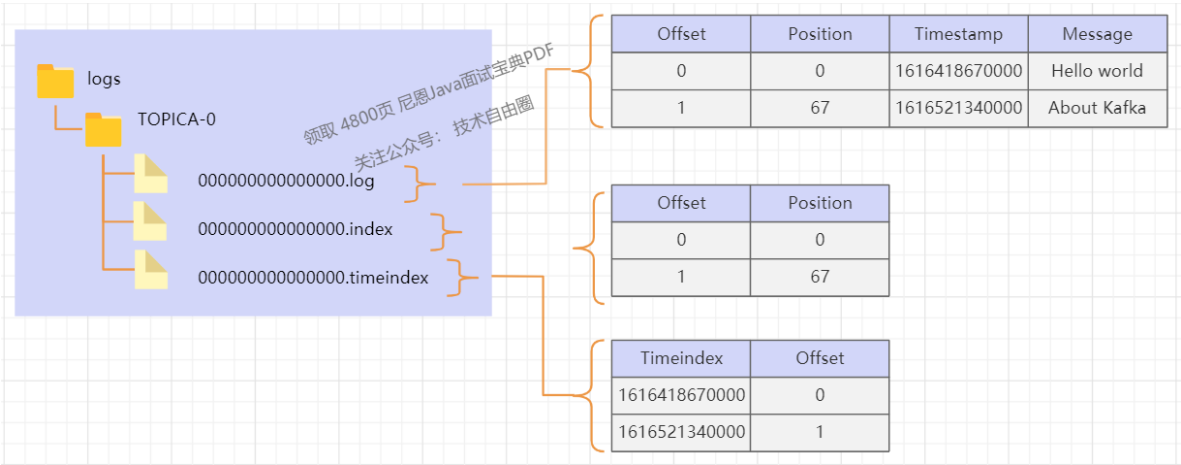


- 读取文件的时候：也是从 PageCache 中来读取数据。

如果消息刚刚写入到服务端就会被消费，按照 LRU 的“优先清除最近最少使用的页”这种策略，读取的时候，对于这种刚刚写入的 PageCache，命中的几率会非常高。

2、Kafka 的文件布局 以及 磁盘文件顺序写入

文件布局如下图所示：



主要特征是：文件的组织方式是“topic + 分区”，每一个 topic 可以创建多个分区，每一个分区包含单独的文件夹。

Kafka 在分区级别实现文件顺序写：即多个文件同时写入，更能发挥磁盘 IO 的性能。

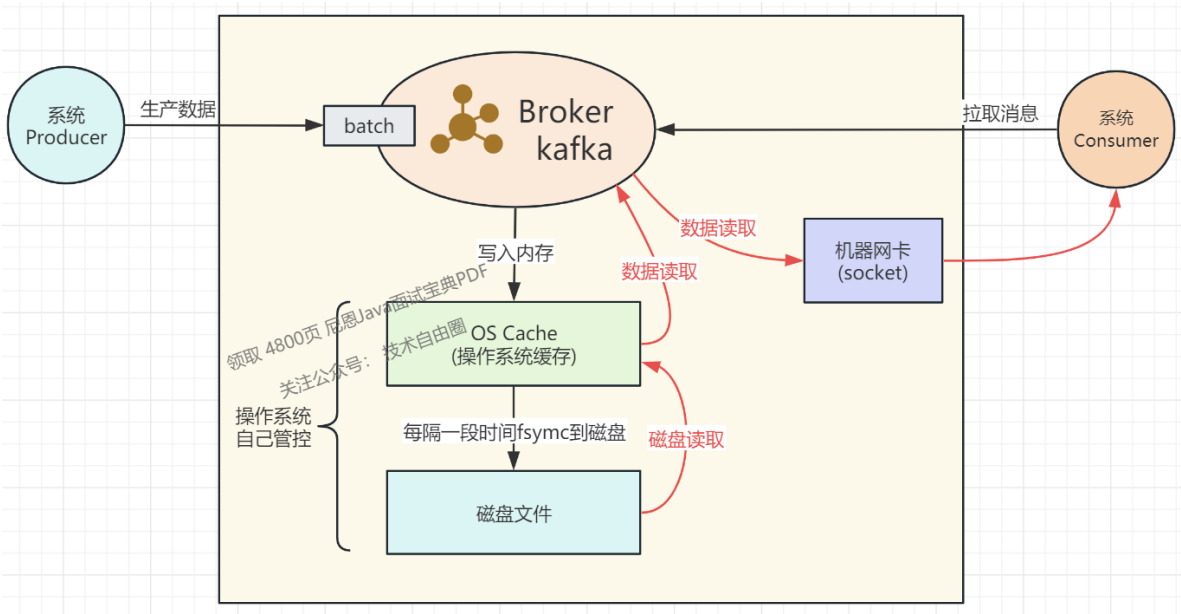
- **相对比 RocketMQ**：RocketMQ 在消息写入时追求极致的顺序写，所有的消息不分主题一律顺序写入 commitlog 文件， topic 和 分区数量的增加不会影响写入顺序。

- **弊端**： Kafka 在消息写入时的 IO 性能，会随着 topic 、分区数量的增长先上升，后下降。

所以使用 Kafka 时，要警惕 Topic 和 分区数量。

3、零拷贝 sendfile：加速消费流程

当不使用零拷贝技术读取数据时：



流程如下：

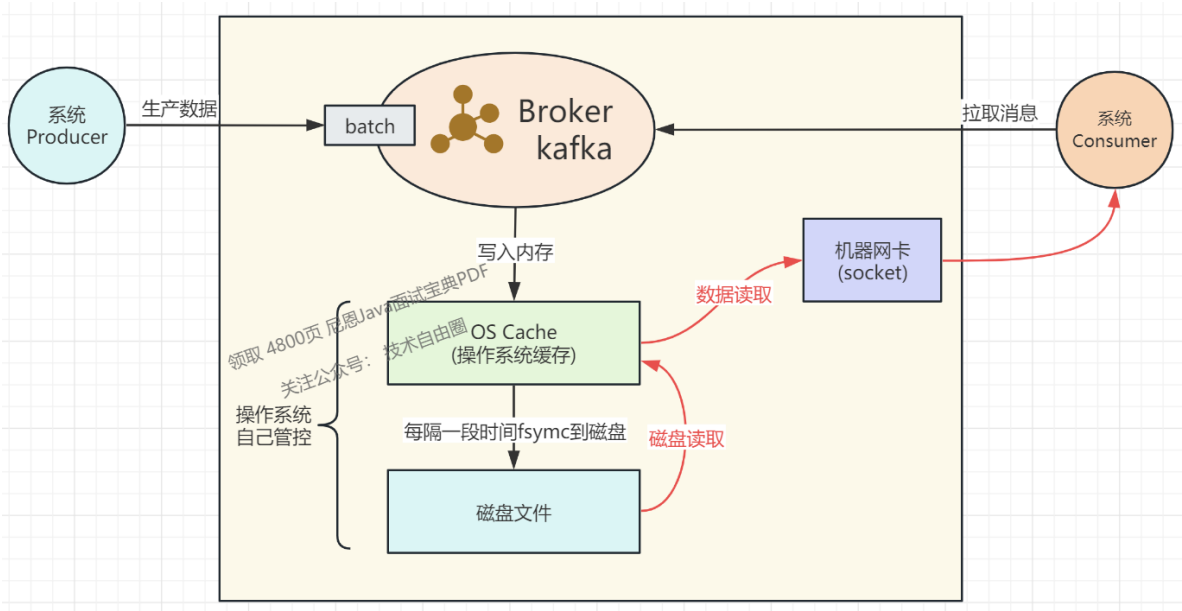
1）消费端 Consumer：向 Kafka Broker 请求拉取消息

2）Kafka Broker 从 OS Cache 读取消息到 应用程序的内存空间：

- 若 OS Cache 中有消息，则直接读取；
- 若 OS Cache 中无消息，则从磁盘里读取。

3）再通过网卡，socket 将数据发送给 消费端Consumer

当使用零拷贝技术读取数据：



Kafka 使用零拷贝技术可以把这个复制次数减少一次，直接从 PageCache 中把数据复制到 Socket 缓冲区中。

- 这样不用将数据复制到用户内存空间。
- DMA 控制器直接完成数据复制，不需要 CPU 参与，速度更快。

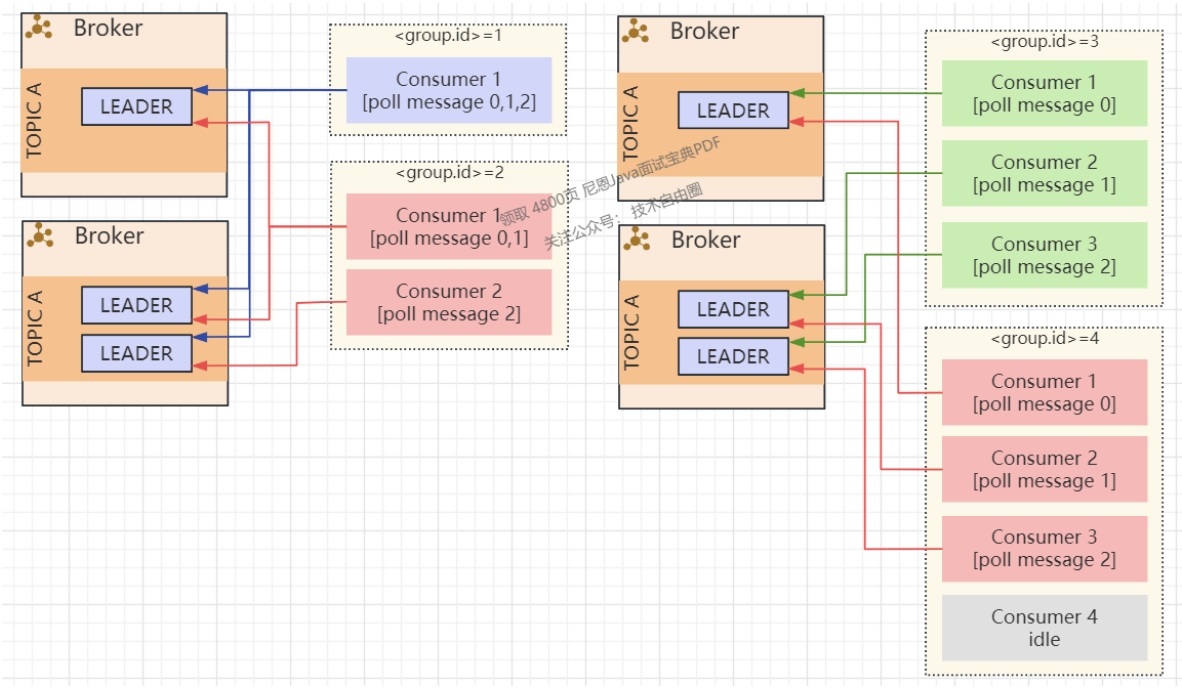
三、消费端 Consumer

消费者只从 Leader分区批量拉取消息。

为了提高消费速度，多个消费者并行消费比不可少。Kafka 允许创建消费组(唯一标识 group.id)，在同一个消费组的消费者共同消费数据。

举个例子：

- 有两个 Kafka Broker，即有 2个机子
- 有一个主题：TOPICA，有 3 个分区(0, 1, 2)



如上图，举例 4 中情况：

- group.id = 1，有一个消费者：这个消费者要处理所有数据，即 3 个分区的数据。
- group.id = 2，有两个消费者：consumer 1消费者需处理 2个分区的数据，consumer2 消费者需处理 1个分区的数据。

- group.id = 3，有三个消费者：**消费者数量与分区数量相等**，刚好每个消费者处理一个分区。
- group.id = 4，有四个消费者：**消费者数量 > 分区数量**，第四个消费者则会处于空闲状态。

说在最后

kafka相关面试题，是非常常见的面试题。

以上的内容，如果大家能对答如流，如数家珍，基本上 面试官会被你 震惊到、吸引到。

最终，**让面试官爱到 “不能自已、口水直流”**。offer， 也就来了。

学习过程中，如果有啥问题，大家可以来 找 40岁老架构师尼恩交流。

参考文献

https://juejin.cn/post/7134463012563320868

https://blog.csdn.net/duysh/article/details/116914395

推荐阅读

《百亿级访问量，如何做缓存架构设计》

《多级缓存 架构设计》

《消息推送 架构设计》

《阿里2面：你们部署多少节点？1000W并发，当如何部署？》

《美团2面：5个9高可用99.999%，如何实现？》

《网易一面：单节点2000Wtps，Kafka怎么做的？》

《字节一面：事务补偿和事务重试，关系是什么？》

《网易一面：25Wqps高吞吐写Mysql，100W数据4秒写完，如何实现？》

《亿级短视频，如何架构？》

《炸裂，靠“吹牛”过京东一面，月薪40K》

《太猛了，靠“吹牛”过顺丰一面，月薪30K》

《炸裂了...京东一面索命40问，过了就50W+》

《问麻了...阿里一面索命27问，过了就60W+》

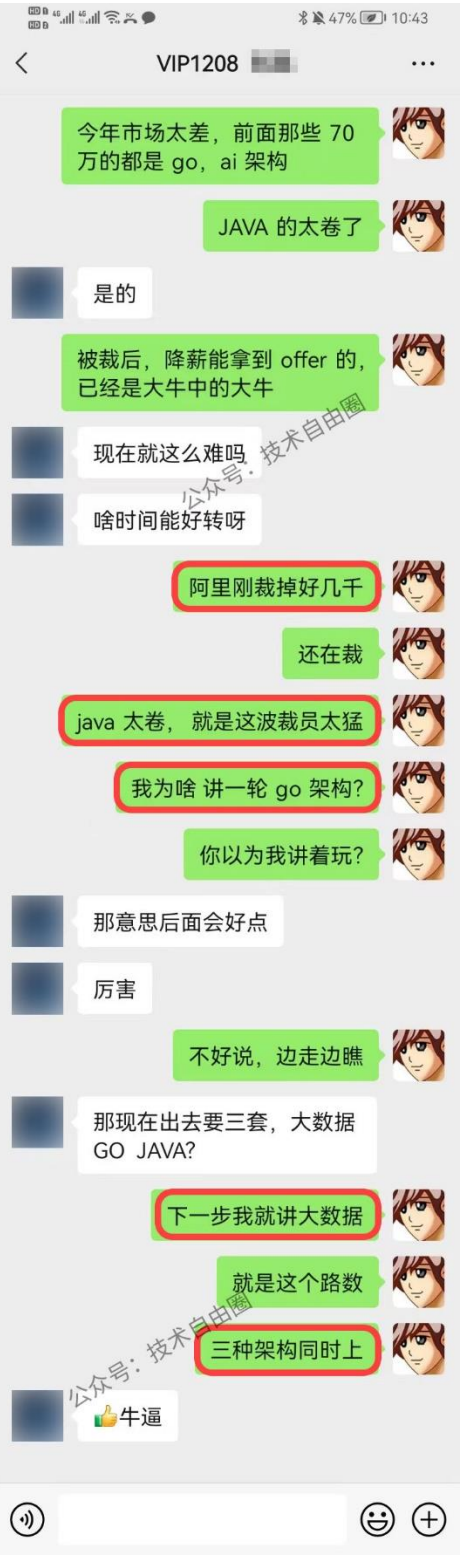
《百度狂问3小时，大厂offer到手，小伙真狠！》

《饿了么太狠：面个高级Java，抖这多硬活、狠活》

《字节狂问一小时，小伙offer到手，太狠了！》

《收个滴滴Offer：从小伙三面经历，看看需要学点啥？》

未来职业，如何突围：三栖架构师



成功案例：2年翻3倍，35岁卷王成功转型为架构师

详情：http://topcoder.cloud/forum.php?mod=forumdisplay&fid=43&page=1

最新　最后发表　热门　精华

- 成功案例：[1057号卷王] 3年小伙拿到外企offer，薪酬涨了200%
  卷王1号　超级版主　前天 17:41
- 成功案例：[645号卷王] 4年经验卷王逆袭，被毕业后，反涨24W
  卷王1号　超级版主　2022-9-21
- 成功案例：[878号卷王] 小伙8年经验，年薪60W
  卷王1号　超级版主　2022-8-13
- 年薪70W案例：通过尼恩的指导，小伙伴年薪从40W涨到70W
  卷王1号　超级版主　2022-2-11
- 成功案例：[493号卷王] 5年小伙拿满意offer，就业寒冬季逆涨30%
  卷王1号　超级版主　前天 17:43
- 成功案例：[250号卷王] 就业极寒时代，收offer 涨25%
  卷王1号　超级版主　前天 17:38
- 成功案例：[612号卷王] 就业极寒时代，从外包到自研
  卷王1号　超级版主　前天 17:15
- 成功案例：[913号卷王] 热烈祝贺6年经验卷王，年薪40W
  卷王1号　超级版主　2022-9-21
- 成功案例：[959号卷王] 4年经验卷王，喜获百度、Boss直聘等N个优质offer，最高涨100%
  卷王1号　超级版主　2022-9-21
- 成功案例：[529号卷王] 5年经验卷王喜收2大offer，最高涨5K
  卷王1号　超级版主　2022-9-21
- 成功案例：[811号卷王] 热烈祝贺7年经验卷王，薪酬涨30%
  卷王1号　超级版主　2022-9-21
- 成功案例：[287号卷王] 不惧大寒潮，卷王逆市收4 offer，涨30%，可喜可贺
  卷王1号　超级版主　2022-5-30
- 成功案例：[1002号卷王] 5月份"被毕业"，改简历后，斩获顶级央企Offer，涨薪7000+
  卷王1号　超级版主　2022-7-5

成功案例：[7号卷王] 热烈祝贺小伙伴涨薪120%

卷王1号 超级版主 2022-8-13

成功案例：[134号卷王] 大三小伙卷1年，斩获顶级央企Offer，成功逆袭

卷王1号 超级版主 2022-7-6

成功案例：[1008号卷王] 5年经验卷王收42W offer，月涨8000，可喜可贺

卷王1号 超级版主 2022-5-30

成功案例：[453号卷王] 非全日制 6年卷王喜提3 offer，年薪30W，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[924号卷王] 6年卷王喜提4 offer，最高涨薪9000，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[15号卷王] 4年卷王入职 微软，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-12

成功案例：[527号卷王] 4年卷王喜提2 offer，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-13

成功案例：[788号卷王] 3年卷王喜提优质Offer，涨薪60%

卷王1号 超级版主 2022-5-11

成功案例：热烈祝贺：非全日制卷王，喜提2个心仪offer，面3家过2家

卷王1号 超级版主 2022-4-21

成功案例：[693号卷王] 二线城市6年卷王喜提4大优质Offer，含央企offer，最高薪酬35W

卷王1号 超级版主 2022-4-16

成功案例：[85号卷王] 双非2本小伙，春招大捷，喜提9个offer，最高薪酬近30万

卷王1号 超级版主 2022-4-14

成功案例：[741号卷王] 卷王逆袭！6年小伙从很少面试机会到搞定35K*14薪Offer

卷王1号 超级版主 2022-4-12

成功案例：[642号卷王] 热烈祝贺，6年卷王喜提优质国企offer

卷王1号 超级版主 2022-4-7

成功案例：[796号卷王] 热烈祝贺，36岁卷王喜提52万优质offer

卷王1号 超级版主 2022-3-25

成功案例：[15号卷王] 小伙卷1年，涨薪9K+，喜收ebay等多个优质offer

卷王1号 超级版主 2022-3-24

成功案例：[821号卷王] 小伙狠卷3个月，喜提10多个offer

卷王1号 超级版主 2022-3-21

成功案例：[736号卷王] 3年半经验收22k offer，但是小伙志存高远，冲击25k+

卷王1号 超级版主 2022-3-20

成功案例：热烈祝贺1群小卷王offer拿到手软，甚至拒了阿里offer

卷王1号 超级版主 2022-3-16

简历案例：简历一改，腾讯的邀请就来了！热烈祝贺，小伙收到一大堆面试邀请

卷王1号 超级版主 2022-3-10

成功案例：祝贺我圈两大超级卷王，一个过了阿里HR面，一个过了阿里2面

卷王1号 超级版主 2022-3-10

成功案例：小伙伴php转Java，卷1.5年Java，涨薪50%，喜收多个优质offer

卷王1号 超级版主 2022-3-10

成功案例：4年小伙狠卷半年，拿到 移动、京东 两大顶级offer

尼恩 超级版主 2022-3-5

成功案例：[267号卷王] 助力3年经验卷王，拿到蜂巢的17k x 14薪的offer

卷王1号 超级版主 2022-2-27

成功案例：[143号卷王] 二本院校00后卷神，毕业没到一年跳到字节，年薪45W

卷王1号 超级版主 2022-2-27

成功案例：[494号卷王] 尼恩分布式事务助力卷王拿到 中信银行offer

卷王1号 超级版主 2022-2-27

成功案例：[76号卷王] 2线城市卷王，狠卷1.5年，喜收22K offer

卷王1号 超级版主 2022-2-27

成功案例：[429号卷王] 小伙伴在社群卷5个月，涨8k+

卷王1号 超级版主 2022-2-27

成功案例：[154号卷王] 双非学校毕业卷王，连拿 京东到家&滴滴 两个大厂Offer

卷王1号 超级版主 2022-2-27

成功案例：[232号卷王] 涨薪10K，继续卷向食物链顶端

卷王1号 超级版主 2022-2-27

成功案例：狠卷1年技术，喜收 腾讯、阿里、微软三大Offer，最高年薪56W

卷王1号 超级版主 2022-2-27

成功案例：[449号卷王] 应届毕业卷王喜收 滴滴offer, 年薪33W

卷王1号 超级版主 2022-2-27

成功案例：[551号卷王] 小伙伴学完后，成功进入大厂，并且推荐自己的朋友加VIP学习

卷王1号 超级版主 2022-2-10

成功案例：[214号卷王] 助力2年经验卷王，成功拿到17K月薪

卷王1号 超级版主 2022-2-10

成功案例：[92号卷王] 课程实操助力社群小伙伴喜收 喜马拉雅Offer

卷王1号 超级版主 2022-2-10

成功案例：社群卷王小伙伴成功过了滴滴三面 获滴滴Offer

卷王1号 超级版主 2022-2-10

[612号卷王]滴滴小伙伴，蹲点考察半年，觉得靠谱后加入 疯狂创客圈

卷王1号 超级版主 2022-2-10

成功案例：[732号卷王] 尼恩助力3年经验卷王收获 京东offer, 年薪35W

卷王1号 超级版主 2022-2-27

成功案例：[558号卷王] 2年经验卷王，喜收 网易和阿里子公司两个优质offer

卷王1号 超级版主 2022-2-27

成功案例：[569号卷王] 双非应届生卷王, 喜收字节跳动实习offer

卷王1号 超级版主 2022-2-25

成功案例：[420号卷王] 狠卷1年，卷王涨薪80%，涨薪12000元!

卷王1号 超级版主 2022-2-25

成功案例：[76号卷王] 通过尼恩1年半的指导，专科学历小伙伴从0.8K涨到22K

卷王1号 超级版主 2022-2-10

硬核推荐：尼恩Java硬核架构班

详情：https://www.cnblogs.com/crazymakercircle/p/9904544.html

架构班（社群 VIP）的起源：

最初的视频，主要是给读者加餐。很多的读者，需要一些高质量的实操、理论视频，所以，我就围绕书，和底层，做了几个实操、理论视频，然后效果还不错，后面就做成迭代模式了。

架构班（社群 VIP）的功能：

提供高质量实操项目整刀真枪的架构指导、快速提升大家的:

- 开发水平
- 设计水平
- 架构水平

弥补业务中 CRUD 开发短板，帮助大家尽早脱离具备 3 高能力，掌握：

- 高性能
- 高并发
- 高可用

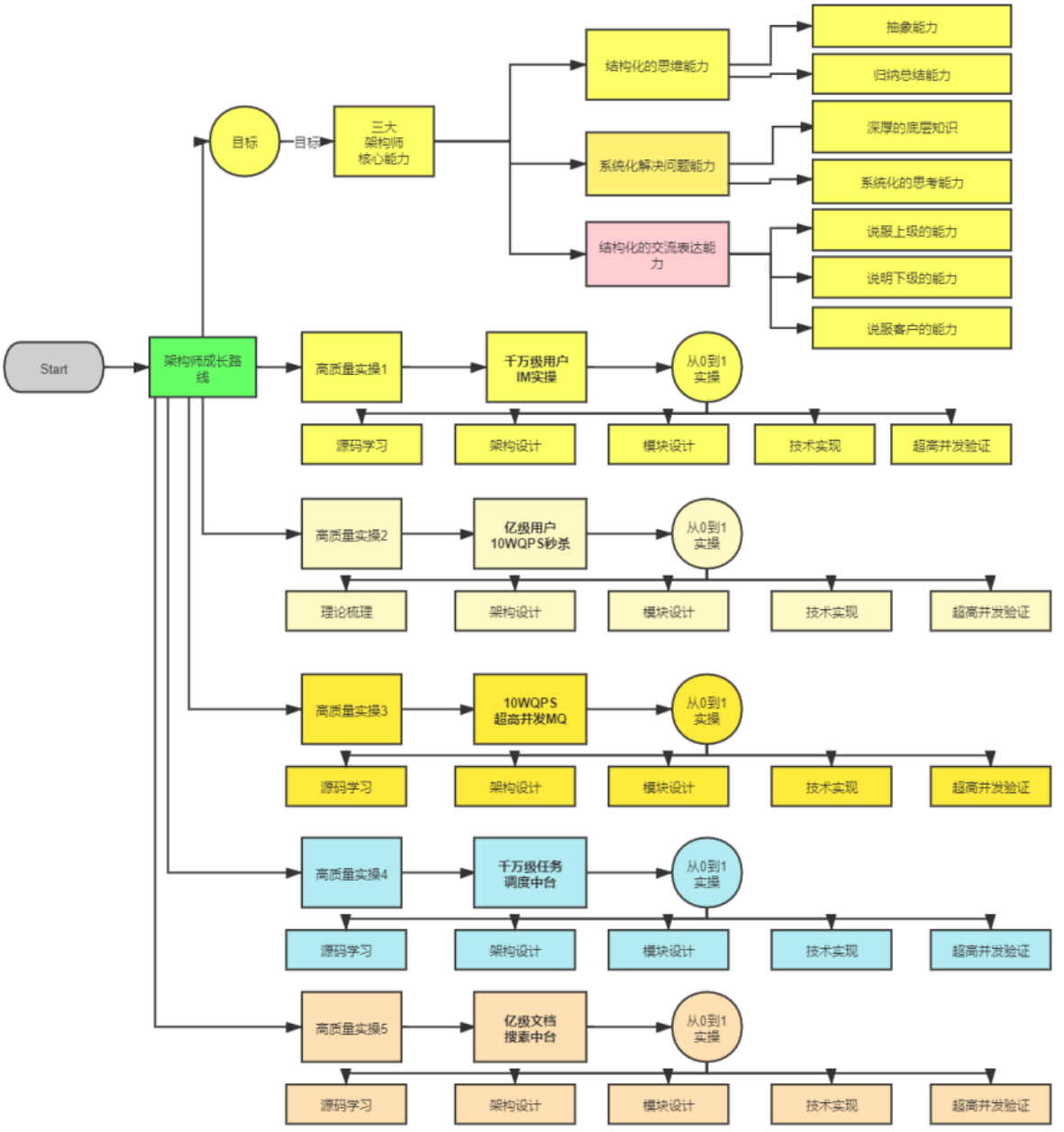
作为一个高质量的架构师成长、人脉社群，把所有的卷王聚焦起来，一起卷：

- 卷高并发实操
- 卷底层原理
- 卷架构理论、架构哲学
- 最终成为顶级架构师，实现人生理想，走向人生巅峰

架构班（社群 VIP）的目的：

- 高质量的实操，大大提升简历的含金量，吸引力，增强面试的召唤率
- 为大家提供九阳真经、葵花宝典，快速提升水平
- 进大厂、拿高薪
- 一路陪伴，提供助学视频和指导，辅导大家成为架构师
- 自学为主，和其他卷王一起，卷高并发实操，卷底层原理、卷大厂面试题，争取狠卷 3 月成高手，狠卷 3 年成为顶级架构师

N 个超高并发实操项目：简历压轴、个顶个精彩



【样章】第 17 章：横扫全网 Rocketmq 视频第 2 部曲：工业级 rocketmq 高可用（HA）底层原理和实操

工业级 rocketmq 高可用底层原理，包含：消息消费、同步消息、异步消息、单向消息等不同消息的底层原理和源码实现；消息队列非常底层的主从复制、高可用、同步刷盘、异步刷盘等底层原理。

工业级 rocketmq 高可用底层原理和搭建实操，包含：高可用集群的搭建。

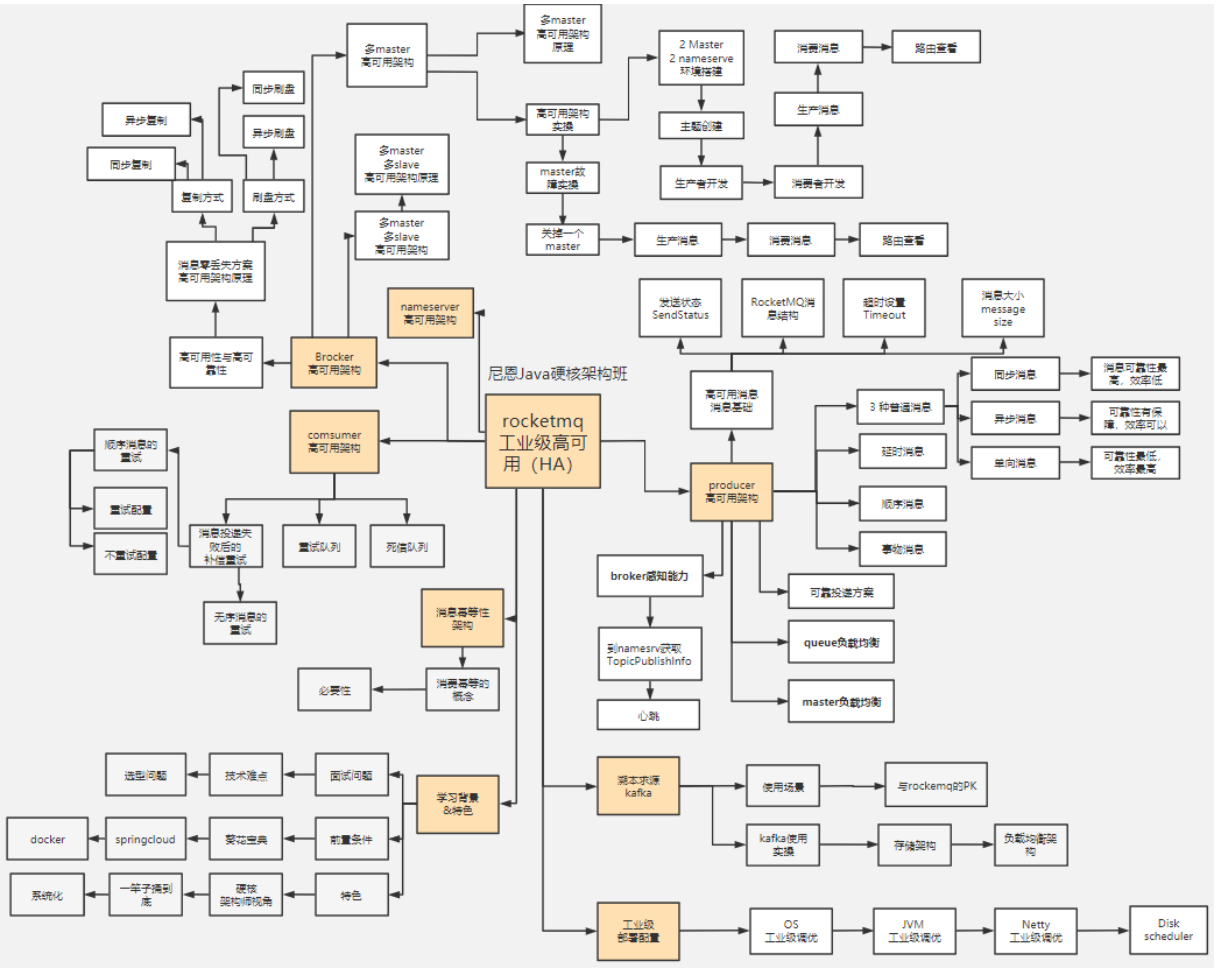
解决以下难题：

1、技术难题：RocketMQ 如何最大限度的保证消息不丢失的呢？RocketMQ 消息如何做到高可靠投递？

2、技术难题：基于消息的分布式事务，核心原理不理解

3、选型难题： kafka or rocketmq ，该娶谁？

下图链接：https://www.processon.com/view/6178e8ae0e3e7416bde9da19



简历优化后的成功涨薪案例（VIP含免费简历优化）


6年小伙收60W年薪
一月速提3大offer
4.24号改简历
5.27号报喜
VIP1239 6年
4月24日 晚上19:10
预期的岗位：60W
预期的岗位：go 和 java 的后端开
4月24日 晚上19:56
秘诀：
简历指导+ 狠狠卷
4月24日 晚上20:50
6年经验 1239
年薪 60W.
21.7 KB
微信电脑版
前几天改 ai，今天改 go
谢谢，我根据这个模式，再整理一下我简历，到时候老师再帮我把把关
ok
4月27日 下午14:37
老师，再帮我看看
准备开始投简历了
简历-6 年后端开发.pdf
210.1 KB
PDF
VIP1239 6年
嗯嗯，我先对着简历准备些东西，然后再开始投吧，反正现在刚刚好五一放假
上午8:01
来还愿啥，3周斩获3个offer，准备入职了
上午8:05
您太牛啦
简历优化后，面试机会太多了，拿完3个offer后，还有许多公司在流程中的都拒绝了
这几天大厂动作不断，联想，阿里都在裁员，您太牛啦
还是老师你强，offer中也达到了预期60w
非常不错
现在都上岸有offer就行，薪酬还能达到预期，已经超级牛掰啦
很多人，连一个面试电话都没有，崩溃的一塌糊涂
抖音上到处是这种
主要是简历优化后，感觉如有神助，每天基本3个面试，除了字节一面没过，其它都通过了
恭喜您
谢谢老师
后续有啥问题，可以找我支撑哈
好嘞，学习圈一直在，要持续提高自己
好的，撸起袖子加油卷，搞技术前途无限好
被裁后转架构，逆涨50%
8年小伙喜提年薪75W
4.16号改简历
5.6号报喜
VIP1236
最近面试了几个一轮游
捞了太多人上岸了
都是你这号
捞我
助力我 一个月时间
绞杀 下钻 打破瓶颈
咱们开始不？
好
4月16日 下午15:04
预期岗位：高级开发、架构
4月16日 下午15:12
预期薪酬：60W
-8 年高级服务端-0404 - 副本(2).
30.2 KB
微信电脑版
4月16日 下午16:19
秘诀：
简历指导+ 狠狠卷
4月16日 下午19:21
通话时长 03:02:25
辛苦老师
VIP1236
上午11:08
尼架，我决定要去上海了。
拿了几个 offer 了？
两个
上海这个是架构师对吧？还有一个呢？
还有个广州高级 Java，待遇 40w 左右，老板比较喜欢我，开了很多绿灯，薪资可以再加，但我还是想闯闯，昨天拒绝了。
年包多少呢？
就之前说的
那都快 80 个 W 了
我 argue 了下，他们控制内部薪酬平衡有点难办到 80，但已经是标出来的上限了。
都快是 高级 java 的 两倍
你撤回了一条消息
75 w 也非常多了，在现在的环境下
除开税，差不了多少，主要是这个方向的潜在价值
关键对你来说，这个是一个 成长机会
是的，寒意还是有的
您是我的贵人
是呢 有个外接大脑就是爽
好好卷，先祝贺您，拿到年薪 75W +
再预祝您，2 年之后，年薪 200W+
谢谢


9年 小伙伴拿到
年薪90W offer
9月11日改简历
11月29日晒offer
秘诀:
简历指导+ 狠狠卷
这个是给你微调的
省略号的地方，需要你再补充一点
上面你留着这个就行
Java 开发 9年-修改.docx
34.1 KB
薪资高 稳
主要的工作是啥?
提升了自己的实力，就不用怕
易所 关于数字货 的 易
po 也有 打盹的时候，该裁员，照样一个不少
年包比 po 多 19w
这么多
po 估计有 70 万
那你不有 90 个 W?
po 给我 68
就是吗


小伙8年经验
年薪60w
7月12日改简历
8月10日晒offer
秘诀:
改简历 + 狠狠卷
明天晚上哈
好哒
恩哥，今晚还改简历吗?
今晚还在外边应酬，估计回去比较晚
要不，咱们延迟到明天，如何
明天白天也行
好的，白天吧，答应别人明天给他们简历了。
OK
那就上午 11 点左右哈
好的
涨幅多少呀?
或者，额度多少呀
之前 36*15，现在这个 39*15
今年行情不太好，还有一些 offer 基本都是平薪，或降薪的。
OK
这个马上来
刚在指导简历
哈哈
稍等哈
辛苦啊恩哥
这次找工作，您的 起到效果了。
我这次复习基本看的都是咱们课程的 。


6年小伙伴
年薪40w
9月6日改简历
9月21日晒offer
秘诀:
简历指导+ 狠狠卷
Java - 6年.docx
24.7 KB
恩哥，简历我改好了，您再帮我看一下
Java - 6年(1).docx
24.5 KB
恩哥，看第二个吧
给你前面调整了一下
涨这多
今年这行情，也算可以了
总包多少呀，让我也陶醉一下
大概 40w 吧
谢谢恩哥的指导和鼓励
是在深圳
深圳的行情尤其难
能有面试电话就不错啦
是的
太牛啦
哈哈哈哈，恩哥的鼓励指导也很重要


5年小伙喜提3个offer
年薪 35个W
5月22日改简历
11月29日晒offer
我简历还没写出来 不然可以给老师您看看
算法方面，可以找点 刷刷，当作复习
项目的话，再好好改造一下
好的 谢谢老师
java 开发-本科统招-2022-5 年经验.docx
36.0 KB
OK
我排好号，回头找你哈
秘诀:
简历指导+ 狠狠卷
尼恩老师晚上好，汇报下最近的 offer 情况。最近面试收到了三个 offer，两个是平薪，一个是跨境电商公司的 offer，涨幅暂时不到 20%。通过这次面试也让我知道自己距离高级开发还有一点点距离，还要再多看才能突破。
最后结合自己的情况，先选择去跨境电商的公司再提升下
之前是 20k*13.5，跨境电商这个是说 23k*（14-15）
恭喜恭喜
就业寒冬，能提到 3 个 offer，已经很 ，已经是高手了
很多小伙伴，面试电话一个都接不到，简历海投 7000 份，只收到 3 次面试机会，没有一个机会拿到最终 offer
您这个年薪，算下来也有 35W 了吧
年终部分还要确认下，大部分人听说只有两个月
这个时间点，拿到这个水平，挺不错的啦
持续加油卷哈
嗯嗯！看看明年自己有没有能力冲击高开
把你我视频都吃透
辛苦老师再帮忙指导下哈


1.5年小伙搞定15K offer
就业寒冬涨100%
5月7日改简历
11月21日晒offer
秘诀：
简历指导+狠狠卷
多谢啦
你好，好多公司都觉得我的简历太简单了，我感觉是不是没法优化了
简历.pdf
397.4 KB
好的
晚上指导你改
推送中台需要加不
我还没做啊，准备八股文暂时没时间搞这些，入职了会开始搞
OK
还有其他的项目吗
一个项目，太少啦
那种练习项目也行
他那个不止两倍吧，我才是两倍，原先7k，现在15k
那也很牛
都翻倍了，都是牛人
正常就30%
慢慢来
下一步可以瞄准25k
其实我工作一年
那你就更牛逼拉
一年经验，搞定15k offer，舍你其谁


卷王逆袭成功案例
6年小伙从很少面试机会到
搞定35K*14薪
3月5日改简历
4月11日拿offer
面试邀请少
小伙很苦恼
理想很丰满
目标35K
一个月拿到了
理想的offer
面试法宝
rocketmq四部曲

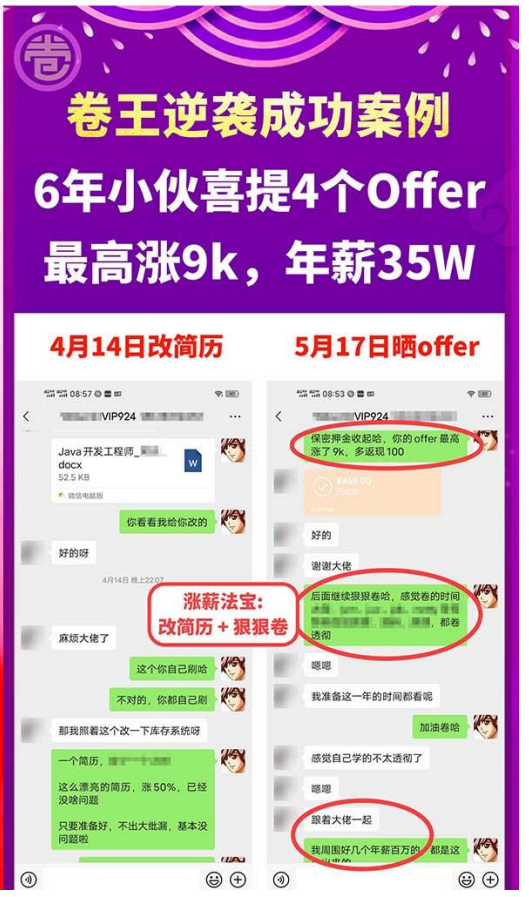

6年 经验小伙伴
喜收25K offer
3月12日改简历
12月1日晒offer
秘诀：
简历指导+狠狠卷
老师
咱们以这个项目为模板改哈
嗯好
项目有多少人，你带领多少人？
你工作几年了？
6年了
项目有多少人，你带领多少人？
语音交流一下
好
你方便吗
OK
不知道往哪个方向发展
改简历挪窝，20 涨到 25 k，高级开发，P7 带的后端团队
任务重，道路远，并不是没有方向


7年经验卷王
薪酬涨30%
7月11日改简历
9月1日晒offer
秘诀：
改简历 + 狠狠卷
咱们开始修改哈
要语音还是打字也行，还没到家
解决过最复杂的难题是啥？说 3 个哈
看看，有没有问题？
没问题
看看，有没有问题？
只能挪窝
下次你喊足一点哦
只要本事好，拿 offer 比较容易的
喊了，但是我流水太差了。。。我这家是补贴，银行不认这玩意，只能在上家的基础上加到 30% 封顶（特别优秀除外，我只是不错。。。）
入职了，最终并不能大幅涨薪，还是旧的太低，只涨了 30%
恭喜一下
现在不比往年
能涨 30% 是大牛
拿到 offer 都算大牛
现在很多人投几百发，连个电话都没有
你已经很


4年经验卷王逆袭
被毕业后，反涨24W
7月改简历
8月30日晒offer
秘诀:
改简历 + 狠狠卷
这就是你的简历
差得太多啦
ok
总共写了四个项目，最近的一年的还没补充上
你是在职，还是离职呢?
离职
原因大概是啥?
这个地方要这样写
项目被终止
方便语音沟通不
ok
是的 经你这么指导，非常重要
老哥 我八月十号开始找工作，今天已经入职了
现金基本持平，股票 +24W
总计涨了多少呢
能涨24W
股票这个吧只能到手了才算
也不错啦
很多小伙伴，面试机会都没有
感谢老哥的指导，继续跟你卷技术
继续狠狠卷哈，马上就技术自由拉

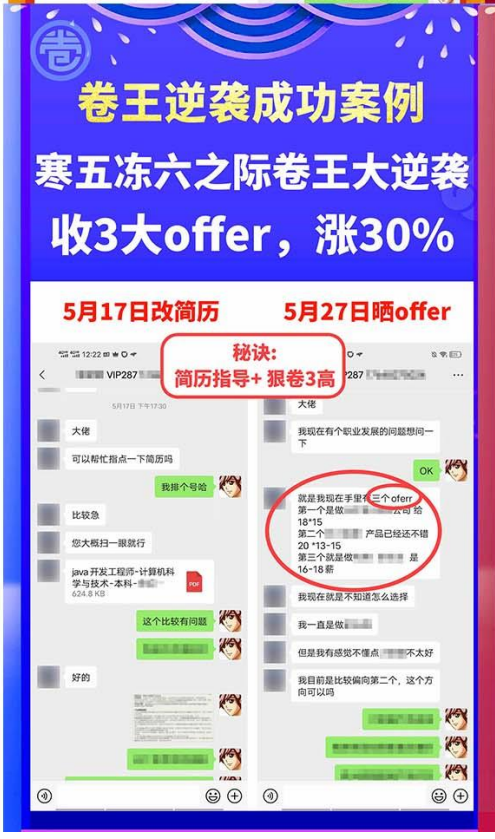

小伙5月份"被毕业"，改简历后
斩获顶级央企Offer
涨薪7000+
5月29日改简历
7月5日晒offer
秘诀:
简历指导+ 狠卷3高
快速看书，就要不求甚解，把目录和 场景大概一下，然后重点的地方，用到的地方，再去回炉
尼恩 我被“ 毕业 ”了
这周末或下周我你改一下简历
毕业了没有关系
ok，发我吧
it行业，跳来跳去，太频繁啦
嗯，其实有点心理准备
简历
不太会写简历
尼恩 我拿到的offer了
涨20%，不还价的
看起来不差钱呀
超过了8000没
平均算下来
7000多
好的
有啥面试的心得吗
可以分享给其他小伙伴的


卷王逆袭成功案例
武汉6年喜收4个优质offer
最高的年薪35W
2月9日改简历
4月15日晒offer
尼恩老师，新年好!
能帮忙修改下简历吗?
金三银四准备跳了
可以的
java 开发-6年-简历-.pdf
拜托了，恩师。希望能拿 25k 回来给你报喜
好，我加一下
还有吗?
面试法宝:
改简历 + 实操
恩哥，决赛圈 offer 了
截图是以我目前认知能写出来的评分了，麻烦帮我参考下
选择大于努力，恩师助我上岸
这么多 offer，我看看哈
都是恩哥指点有方，本来还有个新能源汽车的，35w给拒了，主要太远了
跟着恩哥卷的时间太短了，目前实力也只能到这儿了
这里边有个大数据的，感觉也不错

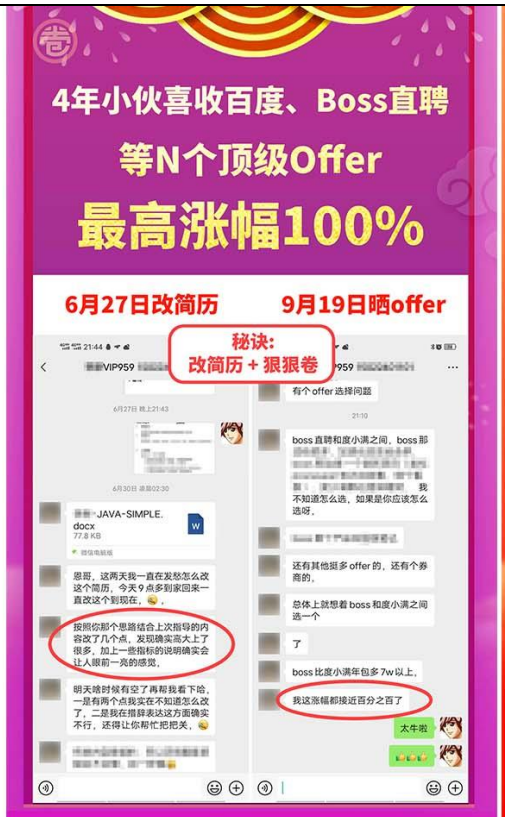

卷王逆袭成功案例
6年小伙喜提4个Offer
最高涨9k，年薪35W
4月14日改简历
5月17日晒offer
Java 开发工程师_.docx
你看看我给你改的
好的呀
麻烦大佬了
这个你自己刷哈
不对的，你都自己刷
那我照着这个改一下库存系统呀
一个简历，
这么漂亮的简历，涨 50%，已经没啥问题
只要准备好，不出大纰漏，基本没问题啦
涨薪法宝:
改简历 + 狠狠卷
保密押金收起哈，你的 offer 最高涨了 9k，多返现 100
好的
谢谢大佬
后面继续狠狠卷哈，感觉卷的时间
透彻
嗯嗯
我准备这一年的时间都看呢
加油卷哈
感觉自己学的不太透彻了
嗯嗯
跟着大佬一起
我周围好几个年薪百万的，都是这


卷王逆袭成功案例
5年经验小伙收2个offer
最高涨薪8k，年薪42W
5月9日改简历
5月30日晒offer
秘诀:
简历指导+ 狠卷3高
以此为样
大家狠狠卷
打造最卷IT社群


卷王逆袭成功案例
非全日制 6年经验卷王
喜提3个Offer，年包30W
5月9日改简历
5月18日晒offer
面试法宝:
改简历 + 狠狠卷


卷王逆袭成功案例
寒五冻六之际卷王大逆袭
收3大offer，涨30%
5月17日改简历
5月27日晒offer
秘诀:
简历指导+ 狠卷3高


卷王逆袭成功案例
4年卷王入职微软，涨50%
3月7日改简历
5月12日晒offer
涨薪法宝:
改简历 + 狠狠卷


4年小伙喜收百度、Boss直聘
等N个顶级Offer
最高涨幅100%
6月27日改简历
9月19日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
4年卷王入收2个offer，涨50%
3月23日改简历
5月12日晒offer
涨薪法宝:
改简历＋狠狠卷


小伙大三暑期很焦虑
跟着尼恩卷一年
校招斩获顶级央企Offer
去年5月19日加入VIP群
今年7月5日晒offer
秘诀:
狠狠卷 书+视频


小伙高中学历
薪酬涨120%
5月6日改简历
7月22日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
非全日制卷王 面试3家
收2个offer 涨薪30%
4月13日改简历
4月21日晒offer
面试法宝：
改简历 + 面试题


5年卷王喜收2大Offer
最高涨5K
5月19日改简历
9月13日晒offer
秘诀：
改简历 + 狠狠卷


卷王逆袭成功案例
3年经验卷王，涨60%
4月16日改简历
5月11日晒offer
涨薪法宝：
改简历 + 狠狠卷


卷王逆袭成功案例
双非二本小伙春招大翻身
喜提9大offer
2月22日改简历
4月13日晒offer
面试法宝：
改简历 + IM实操
9大offer
最高年薪30万

修改简历找尼恩（资深简历优化专家）

- 如果面试表达不好，尼恩会提供 简历优化指导
- 如果项目没有亮点，尼恩会提供 项目亮点指导
- 如果面试表达不好，尼恩会提供 面试表达指导

作为 40 岁老架构师，尼恩长期承担技术面试官的角色：

- **从业以来，“阅历”无数，对简历有着点石成金、改头换面、脱胎换骨**的指导能力。
- **尼恩指导过刚刚就业的小白，也指导过 P8 级的老专家，都指导他们上岸。**

如何联系尼恩。尼恩微信，请参考下面的地址：

语雀：https://www.yuque.com/crazymakercircle/gkkw8s/khigna
码云：https://gitee.com/crazymaker/SimpleCrayIM/blob/master/疯狂创客圈总目录.md