1.主流mq

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比 RocketMQ、Kafka 低一个数量级	同 ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用	同 ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据		经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用

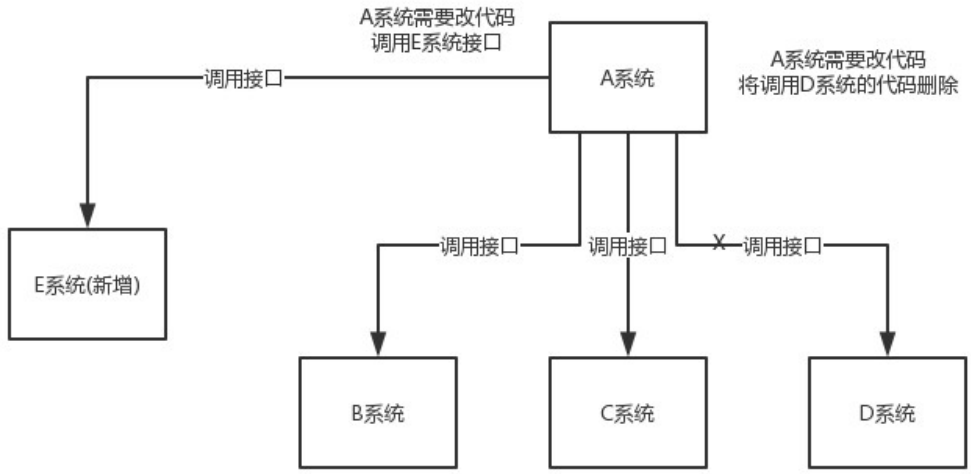
1.1 适用场景

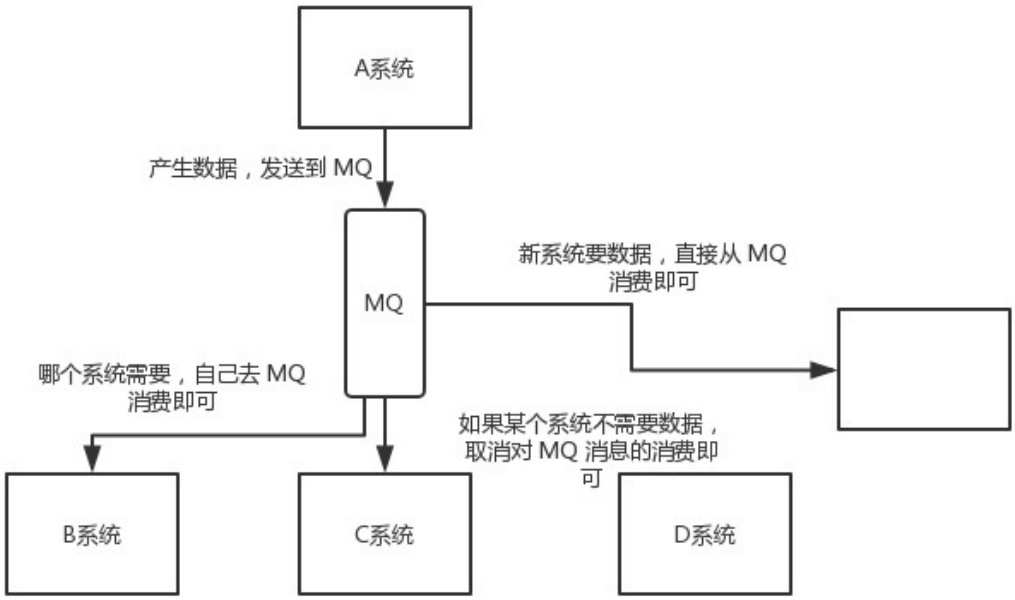
- 一般的业务系统要引入 MQ，最早大家都用 ActiveMQ，但是现在确实大家用的不多了，没经过大规模吞吐量场景的验证，社区也不是很活跃，所以大家还是算了吧，我个人不推荐用这个了
- 后来大家开始用 **RabbitMQ**，但是确实 erlang 语言阻止了大量的 Java 工程师去深入研究和掌控它，对公司而言，几乎处于不可控的状态，但是确实人家是开源的，比较稳定的支持，活跃度也高；
- 不过现在确实越来越多的公司，会去用 RocketMQ，确实很不错（阿里出品），但社区可能有突然黄掉的风险，对自己公司技术实力有绝对自信的，推荐用 RocketMQ，否则回去老老实实用 RabbitMQ 吧，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。

2.消息队列解决问题

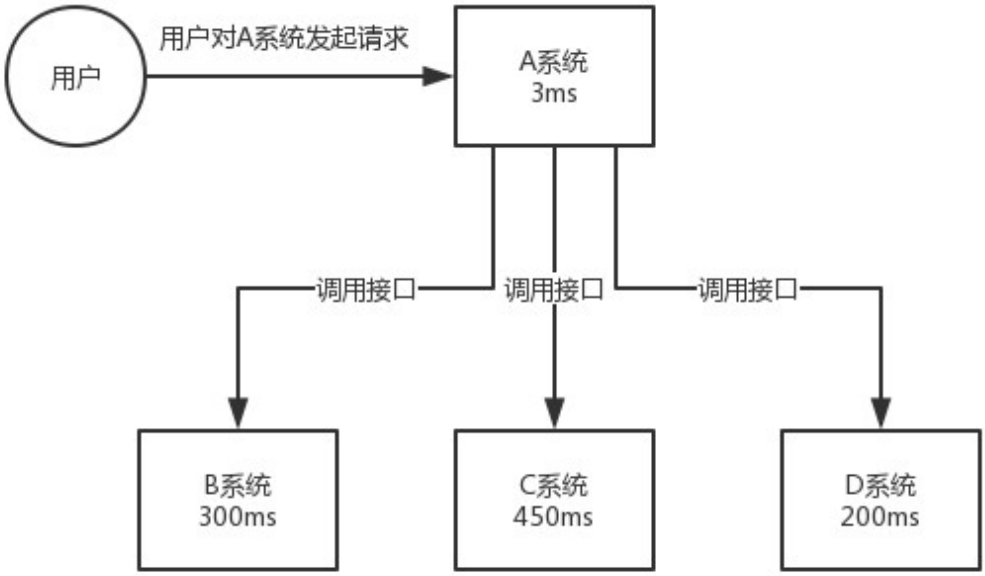
2.1解耦

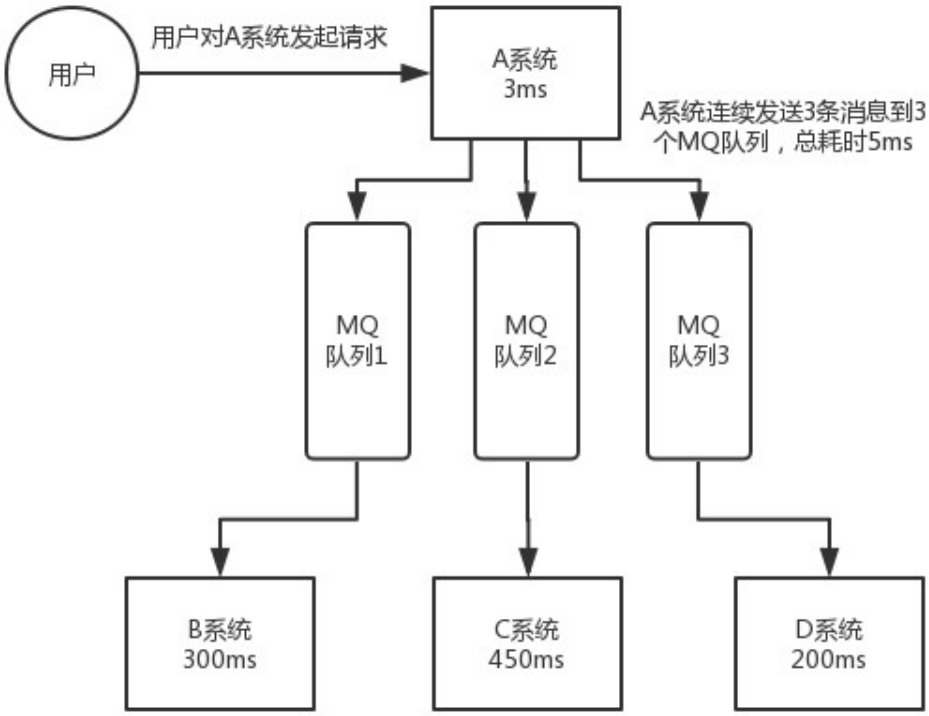




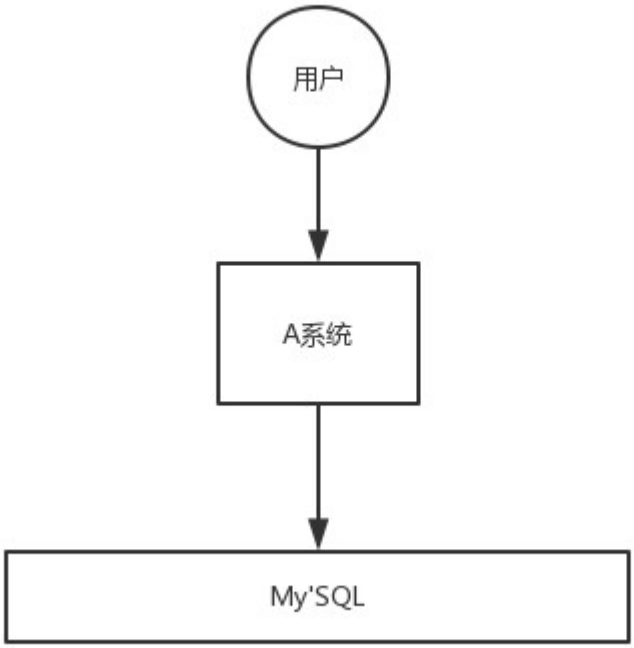
降低模块与模块之间调用的耦合度

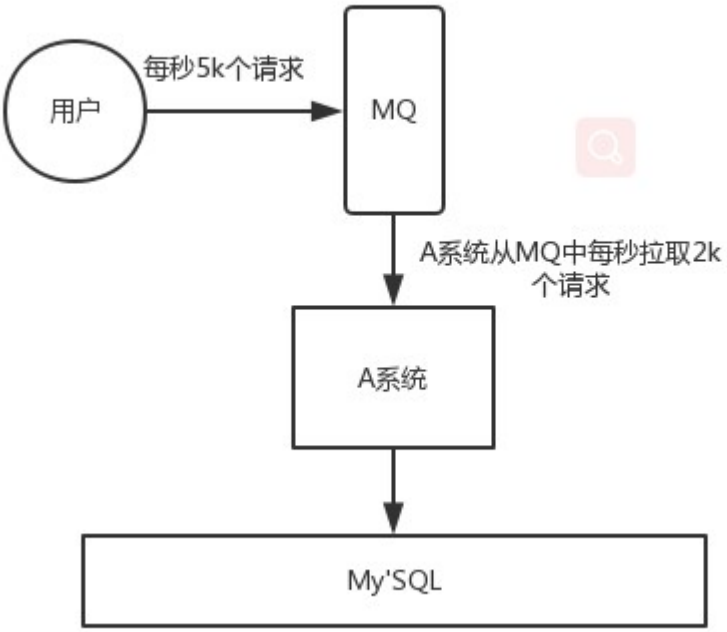
2.2异步


用户
用户对A系统发起请求
A系统
3ms
调用接口
调用接口
调用接口
B系统
300ms
C系统
450ms
D系统
200ms


用户
用户对A系统发起请求
A系统
3ms
A系统连续发送3条消息到3个MQ队列，总耗时5ms
MQ
队列1
MQ
队列2
MQ
队列3
B系统
300ms
C系统
450ms
D系统
200ms

2.3削峰





3.弊端

3.1系统可用性低

万一mq直接宕机怎么办? 保持消息队列的**高可用**

3.2系统复杂度提升

如何保证消息没有重复消费,如何处理消息没有**重复消费**,怎么**处理消息丢失**,怎能保证**消息传递顺序性,**

3.3如何保证数据的一致性

A系统处理完直接返回数据,但是如果bc系统失败了,此时已经相应用户成功了,这时候怎么办?

4.保持RabbitMQ高可用

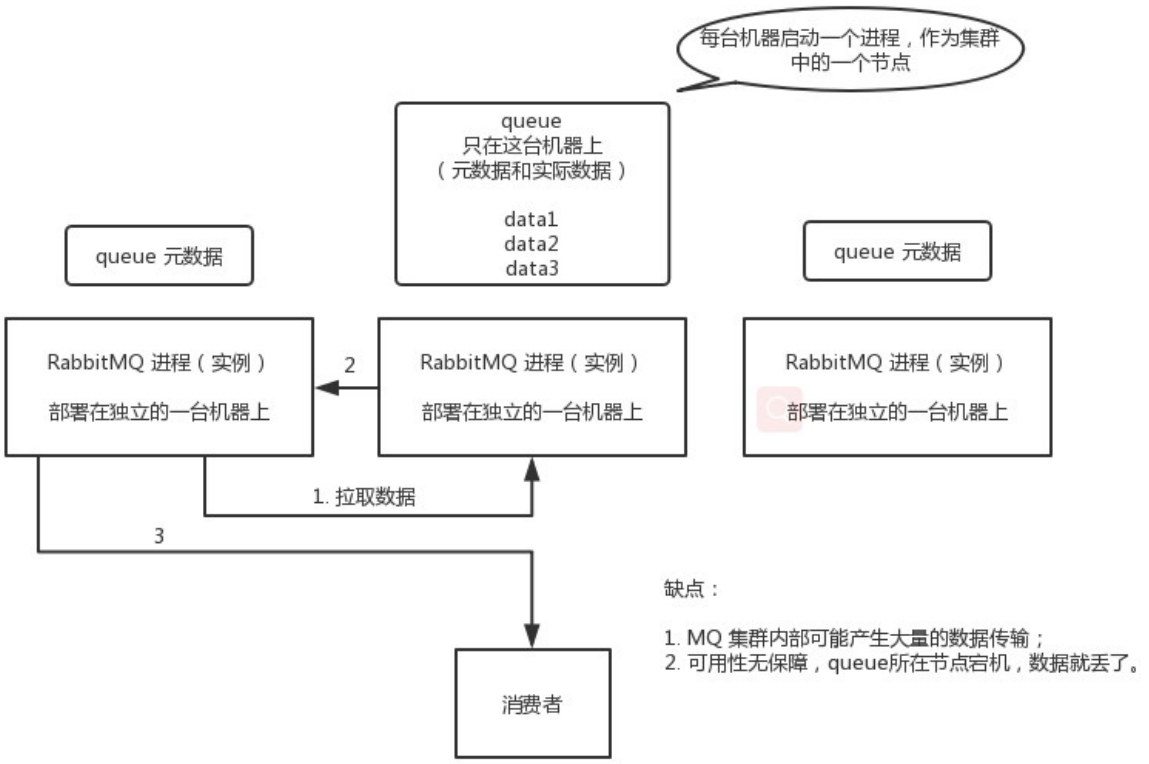
rabbitMQ是基于主从(非分布式)做高可用性的

4.1单机模式

企业没人玩

4.2普通集群模式(无高可用)

- 多个机器开启多个rabbitMQ实例
- 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）
- 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。



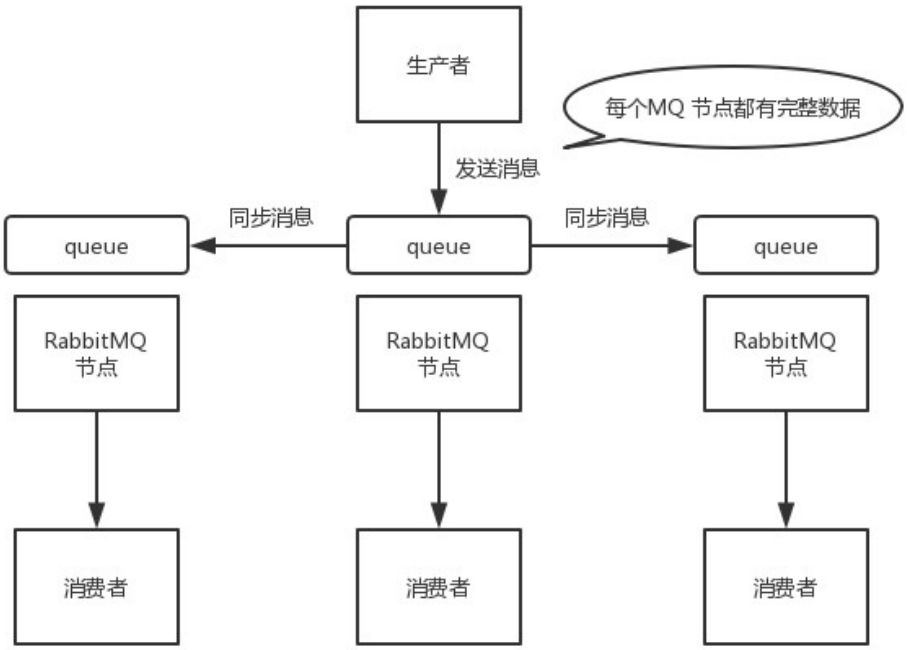
缺点

- 这种方式确实很麻烦，也不怎么好，**没做到所谓的分布式**，就是个普通集群。因为这导致你要么消费者每次随机连接一个实例然后拉取数据，要么固定连接那个 queue 所在实例消费数据，前者有**数据拉取的开销**，后者导致**单实例性能瓶颈**。
- 而且如果那个放 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，如果你**开启了消息持久化**，让 RabbitMQ 落地存储消息的话，**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

- 所以这个事儿就比较尴尬了，这就**没有什么所谓的高可用性，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

4.3镜像集群模式(高可用)

- 在此模式下,创建queue,无论元数据还是queue里的消息,都会**存在多个实例上**
- ,每个rabbitMQ都有一个queue完整镜像,每次写消息到queue时候
- ,会**自动把消息同步**到实例的queue上



开启镜像集群模式

- RabbitMQ有一个控制台,在后台新增一个策略,这个策略是镜像模式策略
- 指定的时候可以要求数据同步到所有节点
- 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,会自动同步数据到其他节点上去

4.4弊端

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！第二，这么玩儿，不是分布式的，就**没有扩展性可言**了，如果某个 queue 负载很重，你加机器，新增的机器也包含了这个 queue 的所有数据，并**没有办法线性扩展**你的 queue。你想，如果这个 queue 的数据量很大，大到这个机器上的容量无法容纳了，此时该怎么办呢

rabbitMQ不支持分布式

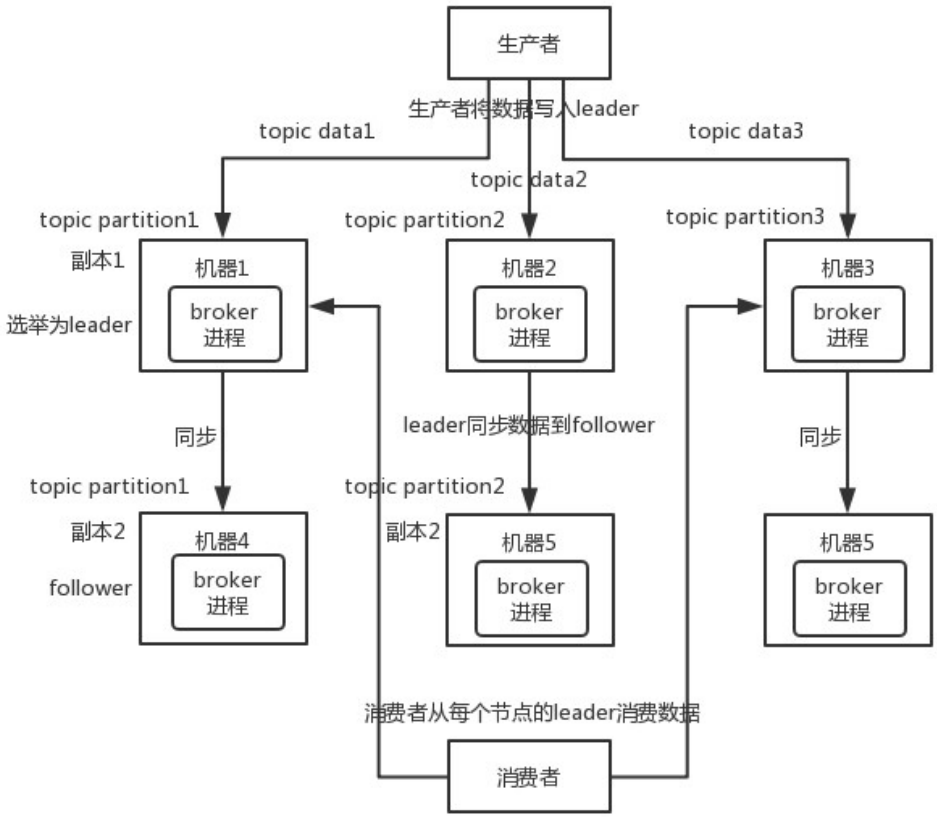
5.Kafka高可用

- kafka有多个broker构成,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可以存放在不同的broker(),**每个partition(隔离物)从存放一部分数据**
- 这就决定了kafka天然分布式消息列队,就是说一个topic,**分散在多台机器上,每台机器存放一部分数据**

(实际上 RabbmitMQ 之类的，并不是分布式消息队列，它就是传统的消息队列，只不过提供了一些集群、HA(High Availability, 高可用性) 的机制而已，因为无论怎么玩儿，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)

- kafka0.8 之后提供**HA机制**就是replica (复制品)每个partition的数据都会同步到其他机器上,形成自己多个replica副本,所有replica会推选出来一个leader,生产和消费都和leader打交道,然后其他的replica 就是follower
- 写leader会负责把数据同步到所有的follower上去,读的时候直接读,leader即可,
- **要是你可以随意读写每个 follower，那么就要 care 数据一致性的问题**，系统复杂度太高，很容易出问题
- Kafka 会均匀地将一个 partition 的所有 replica 分布在不同的机器上，这样才可以提高容错性

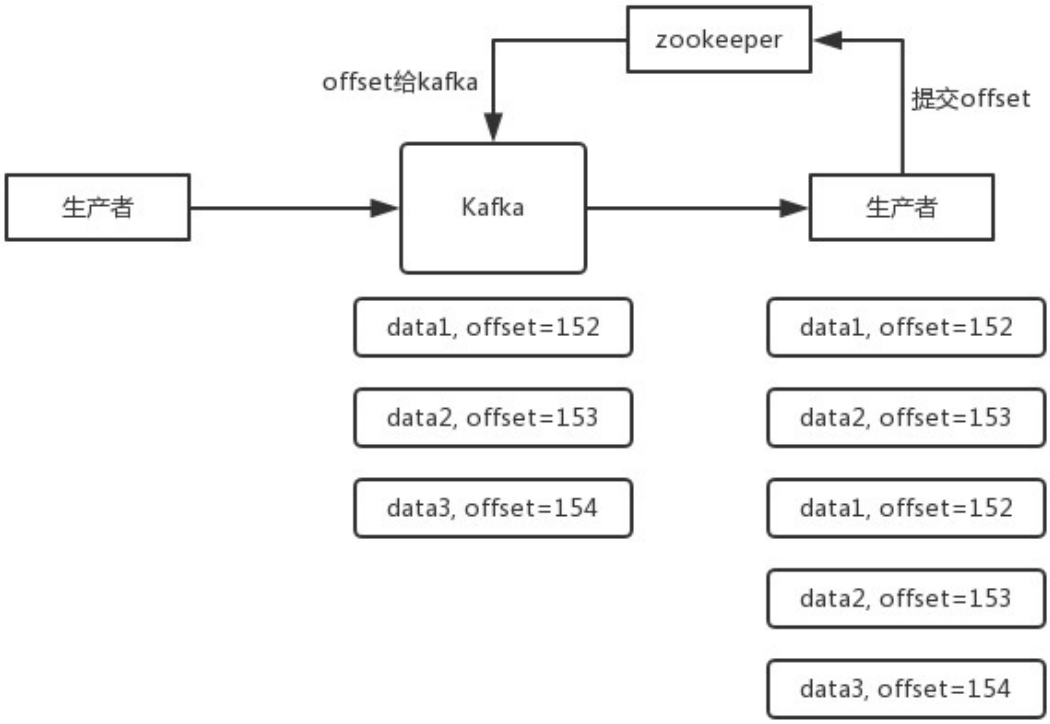


- 如果其中一个broker宕机,并且此broker上有partition的leader,此时就会从follower中选举出一个新的leader出来,继续供应读写操作
- 写数据的时候,生产者写给leader,然后其他的follower就会自动的进行pull操作,一旦同步完所有的数据,就会发送ack给leader,leader收到ack之后,返回消息给生产者

6.MQ重复消费问题

拿kafka举例

- kafka有一个offset的概念,每次消息写进入都有一个offset,代表消息的序号
- 在consumer消费之后每隔一段时间,会自己把消费 过的offset提交以下,下次如果重启,继续上次提交的offset地方进行消费
- 但是如果用kill -9强制终结程序,此时消费之后,但是没有提交offset,此时少数信息将会被再次消费一次



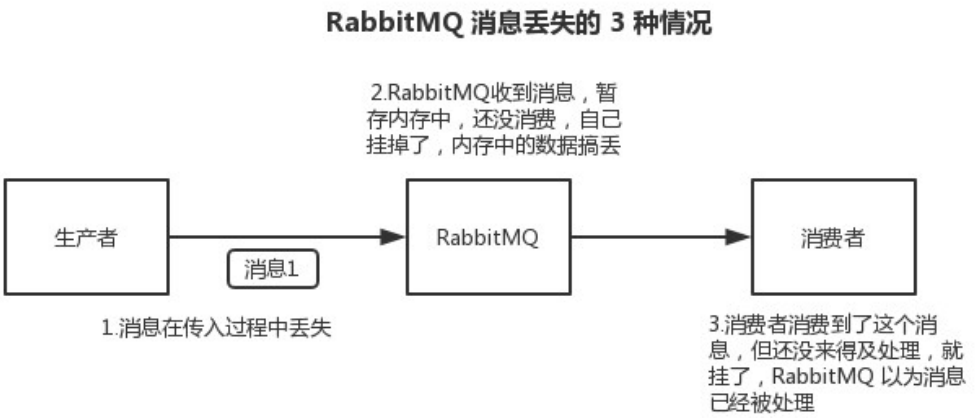
6.1如何保证幂等值

根据业务进行解决

- 比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，update 一下好吧。
- 比如你是写 Redis，那没问题了，反正每次都是 set，天然幂等性。
- 比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的 id，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。
- 比如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。

7.MQ保证消息的可靠性传输(消息丢失问题)

RabbitMQ



7.1生产者丢失数据

1.**RabbitMQ** 提供的事务

```
// 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
```

吞吐量会下降,太消耗性能

2.RabbitMQ提供confirm模式

- 在生产者这端开启confirm模式,之后每写一条消息,都会分配一个唯一的id,然后写入mq
- ,如果写入RabbitMQ会返回一个ack消息,告诉你这个消息ok
- 如果RabbitMQ没有写入成功会毁掉一个nack接口,告诉写入失败,可以重试,

3.事务机制和confirm机制最大不同在于,事务机制是同步的,提交事务之后会阻塞到哪里,但是confirm机制是异步的,你发送一个消息还可以发送下一个消息,

生产者这块大多数使用confirm模式

7.2RabbitMQ弄丢数据

开启RabbitMQ的持久化,恢复后自动读取之前存储的数据,一般数据不会丢失,但是如果RabbitMQ没有进行持久化,但是机器挂了,呢只会造成一点数据丢失

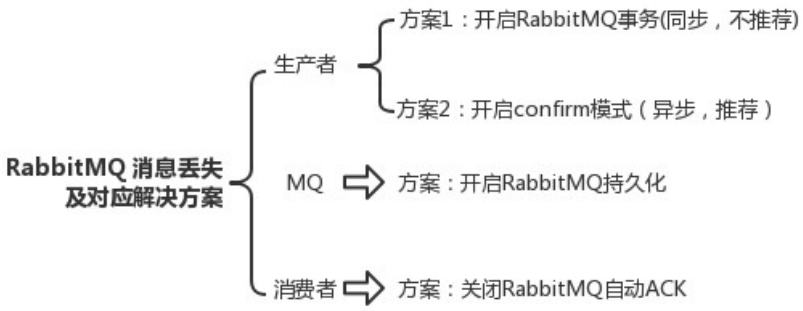
步骤

- 创建queue的时候持久化,这样rabbitMQ就会持久化queue里面的元数据,但是不会持久化queue里面的消息数据
- 在发送信息的时候,将deliveryMode设置为2,就是设置消息的持久化,此时RabbitMQ就会把消息持久化到磁盘上
- 持久化和生产者这边的confirm结合起来,只有消息被持久化到磁盘上之后,才会通知生产者ack,哪怕是持久化到磁盘之前挂掉,生产者没有收到ack,生产者可以重发

7.3消费端弄丢了数据

消息消费端刚刚消费到 ,还没处理,进程挂了,这个时候rabbitmq会觉得消费段已经消费了 数据,这个时候把数据标识为已消费,

- 关闭rabbitMQ的自动ack机制,
- 通过rabbitMQ提供的api,在处理完自己的代码之后,再在程序中ack一把



Kafka

7.4消费端弄丢了消息

- 这个和rabbitmq一样的解决办法,关闭kafka自动提交offset,手动调用api控制offset
- 如果自己刚刚消费完,自己该挂掉(消费端),但是没有手动提交offset,这个时候,重启会再次消费一次,这个时候自己需要保证幂等值即可

7.5Kafka弄丢了数据

Kafka 的某个borker宕机,重新推选leader,但是此时其他的follower还有些数据没有同步,结果此时leader挂掉了,重新推举的leader 会少不少数据,此时需要设置Kafka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。

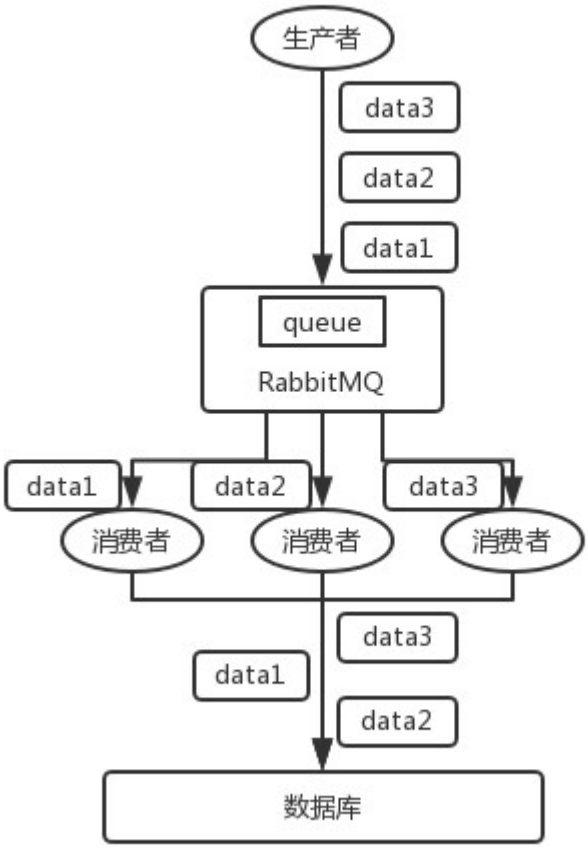
7.6生产者会不会弄丢数据？

如果按照上述的思路设置了 `acks=all`，一定不会丢，要求是，你的 leader 接收到消息，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，生产者会自动不断的重试，重试无限次

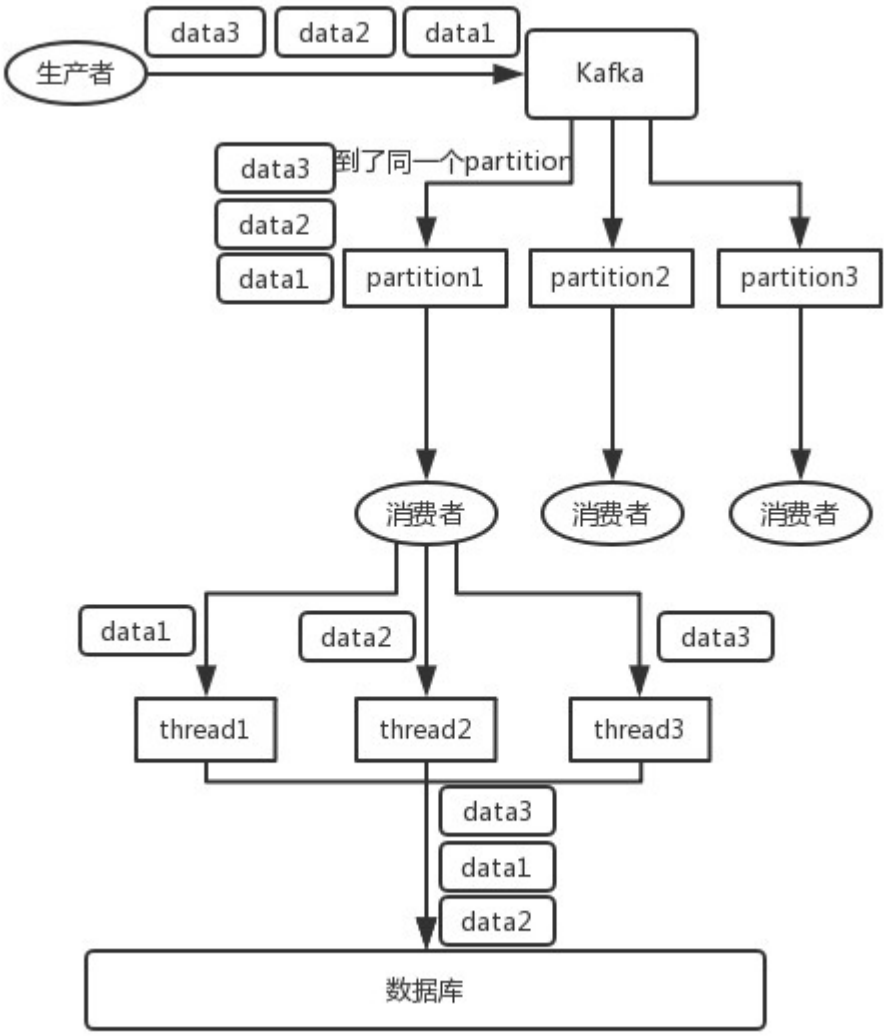
8.MQ保证消息的顺序性

先看看顺序会错乱的俩场景：

- **RabbitMQ**：一个 queue，多个 consumer。比如，生产者向 RabbitMQ 里发送了三条数据，顺序依次是 data1/data2/data3，压入的是 RabbitMQ 的一个内存队列。有三个消费者分别从 MQ 中消费这三条数据中的一条，结果消费者2先执行完操作，把 data2 存入数据库，然后是 data1/data3。这不明显乱了。



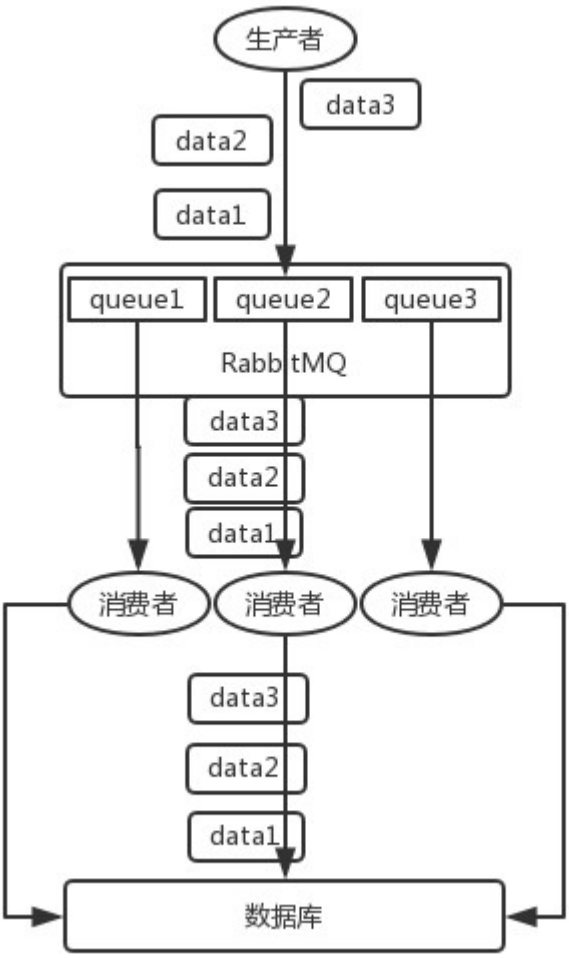
- **Kafka**：比如说我们建了一个 topic，有三个 partition。生产者在写的时候，其实可以指定一个 key，比如说我们指定了某个订单 id 作为 key，那么这个订单相关的数据，一定会被分发到同一个 partition 中去，而且这个 partition 中的数据一定是有顺序的。 消费者从 partition 中取出来数据的时候，也一定是有顺序的。到这里，顺序还是 ok 的，没有错乱。接着，我们在消费者里可能会搞**多个线程来并发处理消息**。因为如果消费者是单线程消费处理，而处理比较耗时的话，比如处理一条消息耗时几十 ms，那么 1 秒钟只能处理几十条消息，这吞吐量太低了。而多个线程并发跑的话，顺序可能就乱掉了。



解决方案

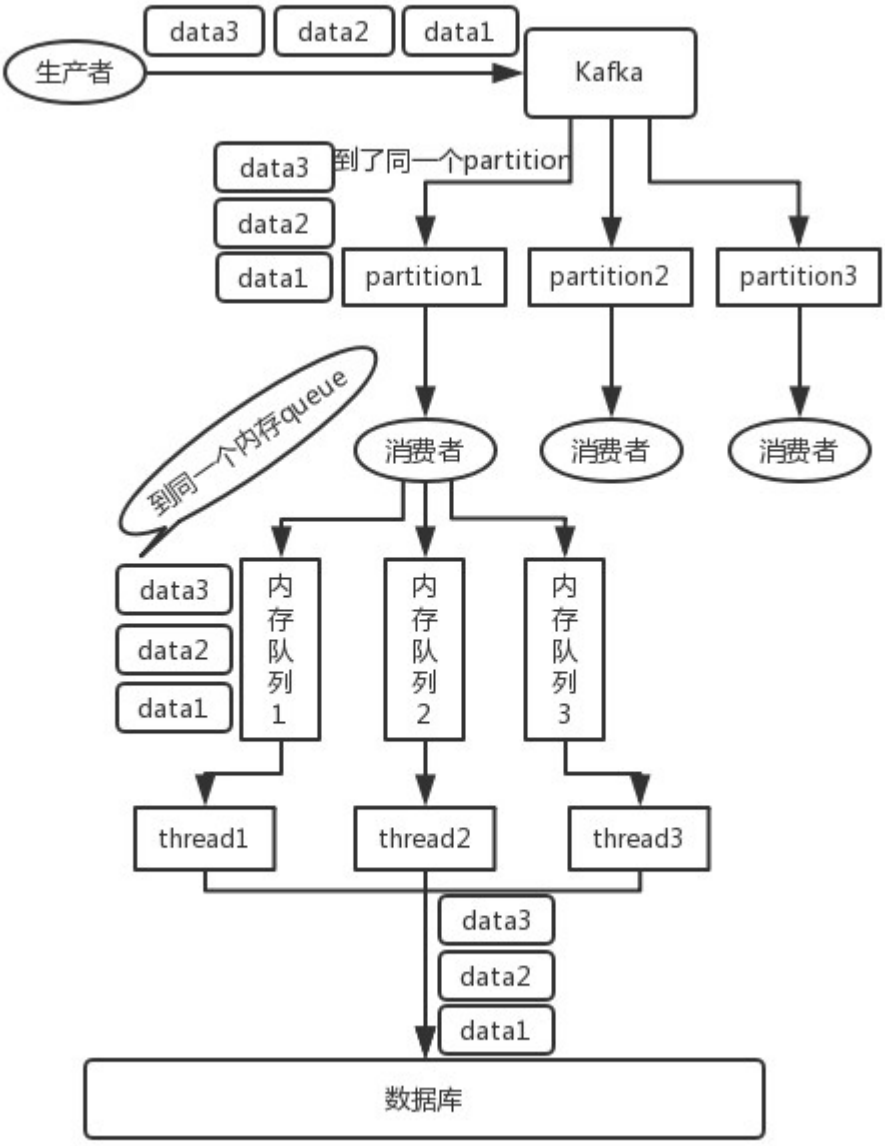
RabbitMQ

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。



Kafka

- 一个 topic，一个 partition，一个 consumer，内部单线程消费，单线程吞吐量太低，一般不会用这个。
- 写 N 个内存 queue，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。



9.设计一个MQ

其实回答这类问题，说白了，不求你看过那技术的源码，起码你要大概知道那个技术的基本原理、核心组成部分、基本架构构成，然后参照一些开源的技术把一个系统设计出来的思路说一下就好。

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。

mq 肯定是很复杂的，面试官问你这个问题，其实是个开放题，他就是看看你有没有从架构角度整体构思和设计的思维以及能力。确实这个问题可以刷掉一大批人，因为大部分人平时不思考这些东西。

10.如何解决消息队列的延时以及过期失效问题？

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**。

这个情况下，就不是说要增加 consumer 消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。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。

11.大量消息在 mq 里积压了几个小时了还没解决

几千万条数据在 MQ 里积压了七八个小时，从下午 4 点多，积压到了晚上 11 点多。这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复 consumer 的问题，让它恢复消费速度，然后傻傻的等待几个小时消费完毕。这个肯定不能在面试的时候说吧。

一个消费者一秒是 1000 条，一秒 3 个消费者是 3000 条，一分钟就是 18 万条。所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概 1 小时的时间才能恢复过来。

一般这个时候，只能临时紧急扩容了，具体操作步骤和思路如下：

- 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。

12.mq 都快写满了

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。