消息队列

使用场景

- 异步处理

发送者将消息发送给消息队列之后，不需要同步等待消息接收者处理完毕，而是立即返回进行其它操作。消息接收者从消息队列中订阅消息之后异步处理。

例如在注册流程中通常需要发送验证邮件来确保注册用户身份的合法性，可以使用消息队列使发送验证邮件的操作异步处理，用户在填写完注册信息之后就可以完成注册，而将发送验证邮件这一消息发送到消息队列中。

只有在业务流程允许异步处理的情况下才能这么做，例如上面的注册流程中，如果要求用户对验证邮件进行点击之后才能完成注册的话，就不能再使用消息队列。

- 流量削锋

在高并发的场景下，如果短时间有大量的请求到达会压垮服务器。

可以将请求发送到消息队列中，服务器按照其处理能力从消息队列中订阅消息进行处理。

- 应用解耦

如果模块之间不直接进行调用，模块之间耦合度就会很低，那么修改一个模块或者新增一个模块对其它模块的影响会很小，从而实现可扩展性。

通过使用消息队列，一个模块只需要向消息队列中发送消息，其它模块可以选择性地从消息队列中订阅消息从而完成调用。

缺点

- 系统可用性降低

  系统引入的外部依赖越多，越容易挂掉。本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整，MQ 一挂，整套系统崩溃的，你不就完了？如何保证消息队列的高可用，可以点击这里查看。

- 系统复杂度提高
  硬生生加个 MQ 进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？头大头大，问题一大堆，痛苦不已。

- 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。

RocketMQ的一些问题

消费模式

Rocketmq消费分为push和pull两种方式，push为被动消费类型，pull为主动消费类型，push方式最终还是会从broker中pull消息。不同于pull的是，push首先要注册消费监听器，当监听器处触发后才开始消费消息，所以被称为“被动”消费。

消息丢失

当你系统需要保证百分百消息不丢失，你可以使用生产者每发送一个消息，Broker 同步返回一个消息发送成功的反馈消息。即每发送一个消息，同步落盘后才返回生产者消息发送成功，这样只要生产者得到了消息发送生成的返回，事后除了硬盘损坏，都可以保证不会消息丢失。

消息堆积

根据不同的业务实现不同的丢弃任务。

顺序消息

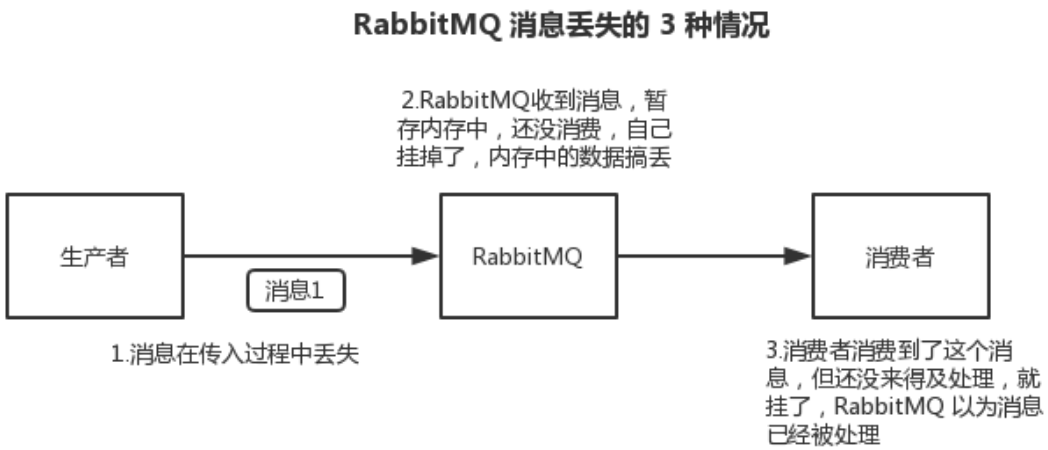
生产者生产消息时指定特定的 MessageQueue ，消费者消费消息时，消费特定的 MessageQueue，其实单机版的消息中心在一个 MessageQueue 就天然支持了顺序消息。注意：同一个 MessageQueue 保证里面的消息是顺序消费的前提是：消费者是串行的消费该 MessageQueue，因为就算 MessageQueue 是顺序的，但是当并行消费时，还是会有顺序问题，但是串行消费也同时引入了两个问题：

> 1. 引入锁来实现串行
> 2. 前一个消费阻塞时后面都会被阻塞

消息重复

- RocketMQ 会出现消息重复发送的问题，因为在网络延迟的情况下，这种问题不可避免的发生，如果非要实现消息不可重复发送，那基本太难，因为网络环境无法预知，还会使程序复杂度加大，因此默认允许消息重复发送
- RocketMQ 让使用者在消费者端去解决该问题，即需要消费者端在消费消息时支持幂等性的去消费消息
- 最简单的解决方案是每条消费记录有个消费状态字段，根据这个消费状态字段来是否消费或者使用一个集中式的表，来存储所有消息的消费状态，从而避免重复消费
- 具体实现可以查询关于消息幂等消费的解决方案

消息丢失



生产者弄丢了数据

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`。

```
// 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。**

所以一般来说，如果你要确保说写 RabbitMQ 的消息别丢，可以开启 `confirm` 模式，在生产者那里设置开启 `confirm` 模式之后，你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中，RabbitMQ 会给你回传一个 `ack` 消息，告诉你说这个消息 ok 了。如果 RabbitMQ 没能处理这个消息，会回调你的一个 `nack` 接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和 `confirm` 机制最大的不同在于，**事务机制是同步的**，你提交一个事务之后会**阻塞**在那儿，但是 `confirm` 机制是**异步**的，你发送个消息之后就可以发送下一个消息，然后那个消息 RabbitMQ 接收了之后会异步回调你的一个接口通知你这个消息接收到了。

所以一般在生产者这块**避免数据丢失**，都是用 `confirm` 机制的。

RabbitMQ 弄丢了数据

就是 RabbitMQ 自己弄丢了数据，这个你必须**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。除非极其罕见的是，RabbitMQ 还没持久化，自己就挂了，**可能导致少量数据丢失**，但是这个概率较小。

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。

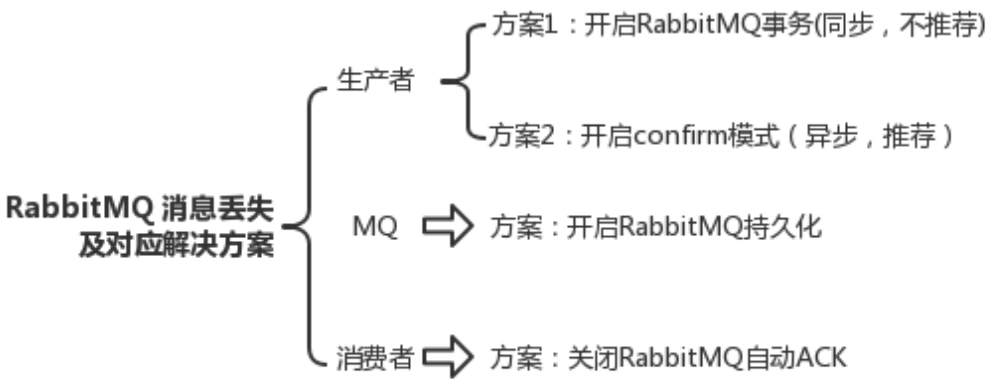
注意，哪怕是你给 RabbitMQ 开启了持久化机制，也有一种可能，就是这个消息写到了 RabbitMQ 中，但是还没来得及持久化到磁盘上，结果不巧，此时 RabbitMQ 挂了，就会导致内存里的一点点数据丢失。

所以，持久化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。

消费端弄丢了数据

RabbitMQ 如果丢失了数据，主要是因为你消费的时候，**刚消费到，还没处理，结果进程挂了**，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。

这个时候得用 RabbitMQ 提供的 `ack` 机制，简单来说，就是你必须关闭 RabbitMQ 的自动 `ack`，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，再在程序里 `ack` 一把。这样的话，如果你还没处理完，不就没有 `ack` 了？那 RabbitMQ 就认为你还没处理完，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。



分布式消息的实现

1. 需要前置知识：2PC
2. RocketMQ4.3 起支持，原理为2PC，即两阶段提交，prepared->commit/rollback
3. 生产者发送事务消息，假设该事务消息 Topic 为 Topic1-Trans，Broker 得到后首先更改该消息的 Topic 为 Topic1-Prepared，该 Topic1-Prepared 对消费者不可见。然后定时回调生产者的本地事务A执行状态，根据本地事务A执行状态，来是否将该消息修改为 Topic1-Commit 或 Topic1-Rollback，消费者就可以正常找到该事务消息或者不执行等

> 注意，就算是事务消息最后回滚了也不会物理删除，只会逻辑删除该消息

RocketMQ 不使用 ZooKeeper 作为注册中心的原因，以及自制的 NameServer 优缺点

- ZooKeeper 作为支持顺序一致性的中间件，在某些情况下，它为了满足一致性，会丢失一定时间内的可用性，RocketMQ 需要注册中心只是为了发现组件地址，在某些情况下，RocketMQ 的注册中心可以出现数据不一致性，这同时也是 NameServer 的缺点，因为 NameServer 集群间互不通信，它们之间的注册信息可能会不一致
- 另外，当有新的服务器加入时，NameServer 并不会立马通知到 Produer，而是由 Produer 定时去请求 NameServer 获取最新的 Broker/Consumer 信息（这种情况是通过 Producer 发送消息时，负载均衡解决）

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？

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 consumer 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 中的消息过期失效了

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。

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。

如何保证RocketMQ高可用?

推荐的几种 Broker 集群部署方式，这里的Slave 不可写，但可读，类似于 Mysql 主备方式。

- 单个 Master

这种方式风险较大，一旦Broker 重启或者宕机时，会导致整个服务不可用，不建议线上环境使用。

- 多 Master 模式

一个集群无 Slave，全是 Master，例如 2 个 Master 或者 3 个 Master
优点：配置简单，单个Master 宕机或重启维护对应用无影响，在磁盘配置为 RAID10 时，即使机器宕机不可恢复情况下，由与 RAID10 磁盘非常可靠，消息也不会丢（异步刷盘丢失少量消息，同步刷盘一条不丢）。性能最高。

缺点：单台机器宕机期间，这台机器上未被消费的消息在机器恢复之前不可订 阅，消息实时性会受到受到影响。 ###先启动 NameServer

###在机器 A，启动第一个 Master
###在机器 B，启动第二个 Master

- 多 Master 多 Slave 模式，异步复制

每个 Master 配置一个 Slave，有多对Master-Slave，HA 采用异步复制方式，主备有短暂消息延迟，毫秒级。

优点：即使磁盘损坏，消息丢失的非常少，且消息实时性不会受影响，因为 Master 宕机后，消费者仍然可以从 Slave 消费，此过程对应用透明。不需要人工干预。性能同多 Master 模式几乎一样。

缺点：Master 宕机，磁盘损坏情况，会丢失少量消息。

###先启动 NameServer
###在机器 A，启动第一个 Master
###在机器 B，启动第二个 Master
###在机器 C，启动第一个 Slave
###在机器 D，启动第二个 Slave

- 多 Master 多 Slave 模式，同步双写

每个 Master 配置一个 Slave，有多对Master-Slave，HA 采用同步双写方式，主备都写成功，向应用返回成功。 优点：数据与服务都无单点，Master宕机情况下，消息无延迟，服务可用性与 数据可用性都非常高

缺点：性能比异步复制模式略低，大约低 10%左右，发送单个消息的 RT 会略高。目前主宕机后，备机不能自动切换为主机，后续会支持自动切换功能 。

###先启动 NameServer
###在机器 A，启动第一个 Master
###在机器 B，启动第二个 Master
###在机器 C，启动第一个 Slave
###在机器 D，启动第二个 Slave

以上 Broker 与 Slave 配对是通过指定相同的brokerName 参数来配对，Master 的 BrokerId 必须是 0，Slave 的BrokerId 必须是大与 0 的数。另外一个 Master 下面可以挂载多个 Slave，同一 Master 下的多个 Slave通过指定不同的 BrokerId 来区分。

如何设计消息队列

- 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。