1、什么是消息队列

消息队列，一般我们会简称它为MQ(Message Queue)，嗯，就是很直白的简写。

队列是一种先进先出的数据结构。

Java中已经实现了很多队列了，那么为什么还需要MQ这种消息中间件呢？？

消息队列可以简单地理解为：将要传输的数据放在队列中。

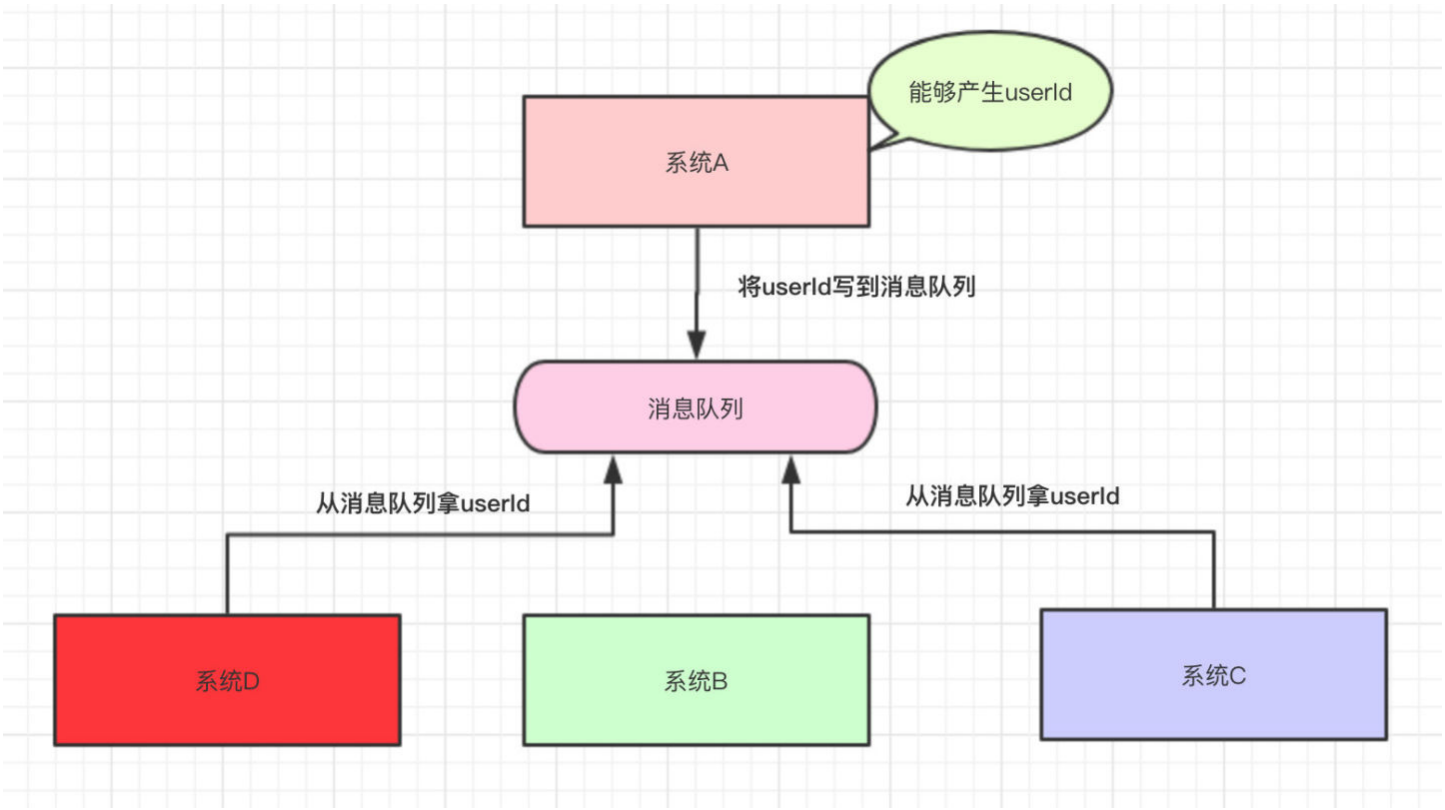
把数据放到消息队列叫做生产者

从消息队列里边取数据叫做消费者

1.1 为什么要使用消息队列？

1. 解耦

系统A不再关系谁需要userId，它只负责将userId写入到消息队列中，谁需要则自己来取，完成了系统A与系统B、C、D的解耦。

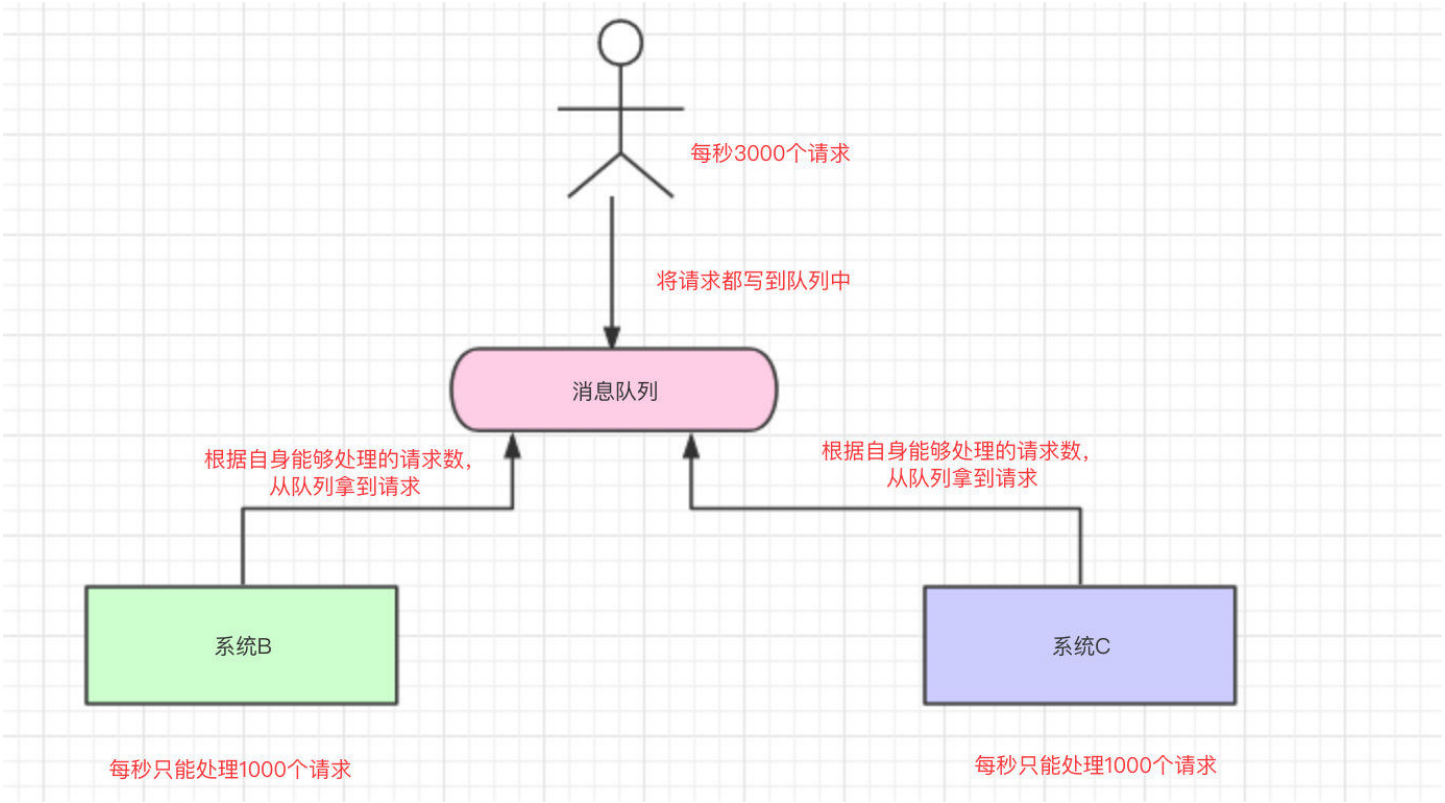


2. 异步

若系统A直接与系统B、C、D交互，A每次都需要等待BCD的任务完成。为了提高用户体验，可以异步地调用B、C、D的接口。系统A执行完了以后，将userId写到消息队列中，然后就直接返回了(至于其他的操作，则异步处理)。

3. 削峰/限流

系统B和系统C根据自己的能够处理的请求数去消息队列中拿数据，这样即便有每秒有8000个请求，那只是把请求放在消息队列中，去拿消息队列的消息由系统自己去控制，这样就不会把整个系统给搞崩。



1.2 使用消息队列会遇到的问题

1. **高可用**

消息队列服务器一挂，整个系统都挂。所以，当我们项目中使用消息队列，都是得集群/分布式的。要做集群/分布式就必然希望该消息队列能够提供现成的支持，而不是自己写代码手动去实现。

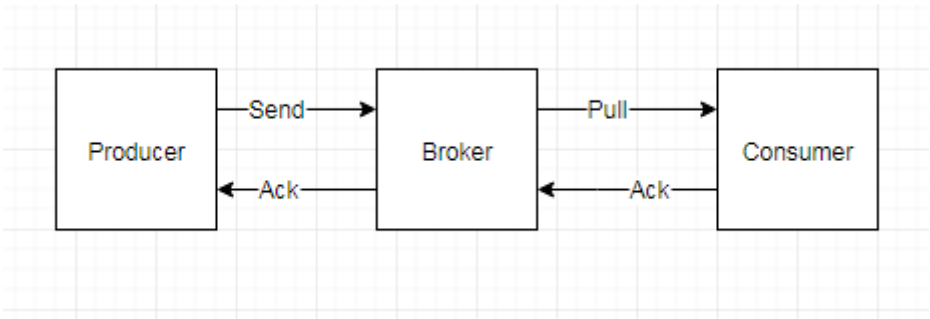
2. 消息丢失

消息从生产到消费可以经历三个阶段：生产阶段、存储阶段和消费阶段。

生产阶段：在这个阶段，从消息在Producer创建出来，经过网络传输发送到Broker端。

存储阶段： 消息在Broker端存储，如果是集群，消息会在这个阶段被复制到其他的副本上。

消费阶段：Consumer从Broker上拉取消息，经过网络 传输发送在Consumer上。



1.2.1. 生产阶段丢失数据

消息队列通常使用确认机制，来保证消息可靠传递：当你代码调用发送消息的方法，消息队列的客户端会把消息发送到Broker，Broker接受到消息会返回给客户端一个确认。只要Producer收到了Broker的确认响应，就可以保证消息在生产阶段不会丢失。有些消息队列在长时间没收到发送的确认响应后，会自动重试，如果重试再失败，就会一返回值或者异常方式返回给客户端。所以在编写发送消息的代码，需要正确处理消息发送返回值或者异常，保证这个阶段消息不丢失。

1.2.2 存储阶段（MQ中间件）丢失数据

如果对消息可靠性要求非常高，可以通过配置Broker参数来避免因为宕机丢消息。对于单个节点Broker，需要配置Broker参数，在收到消息后，将消息写入磁盘再给Producer返回确认响应。如果是Broker集群，需要将Broker集群配置成：至少两个以上节点收到消息，再给客户端发送确认响应。

开启rabbitmq的持久化。设置持久化有两个步骤，第一个是创建queue的时候将其设置为持久化的，这样就可以保证rabbitmq持久化queue的元数据，但是不会持久化queue里的数据；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

1.2.3 消费阶段丢失数据

消费阶段采用和生产阶段类似的确认机制来保证消息的可靠传递。Consumer收到消息后，需在执行消费逻辑后在发送确认消息。

3. 重复消费

3.1 什么情况下会发生重复消费的情况?

举个栗子：

有这么个场景。数据 1/2/3 依次进入 kafka，kafka 会给这三条数据每条分配一个 offset，代表这条数据的序号，分配的 offset 依次是 152/153/154。消费者从 kafka 去消费的时候，也是按照这个顺序去消费。假如当消费者消费了 offset=153 的这条数据，刚准备去提交 offset 到 zookeeper，此时消费者进程被重启了。那么此时消费过的数据 1/2 的 offset 并没有提交，kafka 也就不知道你已经消费了 offset=153 这条数据。那么重启之后，消费者会找 kafka 说，嘿，哥儿们，你给我接着把上次我消费到的那个地方后面的数据继续给我传递过来。数据 1/2 再次被消费。

3.2 什么叫幂等性？

幂等性，通俗点说，就一个数据，或者一个请求，给你重复来多次，你得确保对应的数据是不会改变的，不能出错。

3.3 如何保证幂等性？

这个问题需要结合业务来思考：

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。(这个出现较多)

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。

4. 消息的顺序性

4.1 rabbitmq保证数据的顺序性

如果存在多个消费者，那么就让每个消费者对应一个queue，然后把要发送 的数据全都放到一个queue，这样就能保证所有的数据只到达一个消费者从而保证每个数据到达数据库都是顺序的。
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

5. 分布式事务

MQ作为解决分布式事务的一种，是如何解决分布式事务问题的，如下图，支付服务完成支付步骤后，往MQ发送一条已支付消息，订单服务收到消息后更改订单状态。
有人就想说，这没什么区别么，甚至加了一层MQ，导致需要两次调用，更复杂风险更大？
MQ解决分布式事务是这样做的，A服务保证消息发送成功，MQ保证消息不会丢失，B服务保证消息消费成功。会出现以下情况：
A服务生产消息失败，回滚支付操作，业务回到最初状态，保证了一致性。
A服务生产消息成功，完成支付操作，MQ宕机，MQ重启后，消息还存在，订单服务消费消息，完成修改订单操作。保证了一致性。
A服务生产消息成功，MQ良好，订单服务消费消息失败。但是消息还存在，等订单服务重启后继续消费消息，保证了一致性。
以上就是分布式事务中的最终一致性：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。

6. 场景题

6.1 MQ积压几百万条数据怎么办？

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，让他恢复消费速度，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，一分钟是18万条，1000多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，只能操作临时紧急扩容了，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倍或者20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，得恢复原先部署架构，重新用原先的consumer机器来消费消息

6.2 设置TTL导致大量数据丢失

假设你用的是rabbitmq，rabbitmq是可以设置过期时间的，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。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

原文链接：https://blog.csdn.net/Sparkleii/article/details/107464188