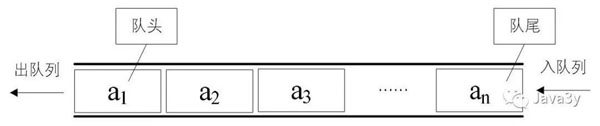
**1.什么是消息队列**

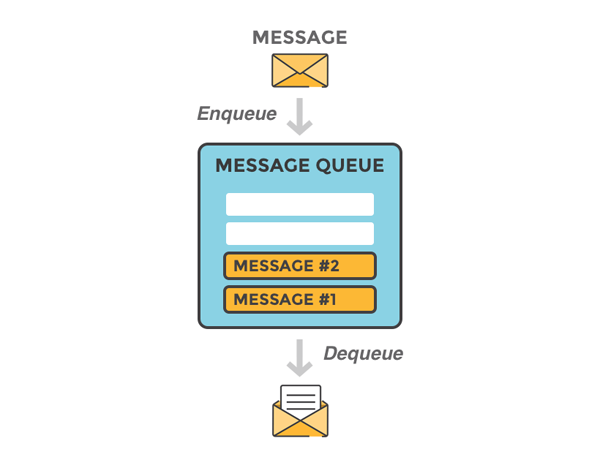
直白就是MQ(Message Queue)

队列是一种先进先出的数据结构



在java里,已经实现了不少的队列

消息队列可以简单理解为:把要传输的数据放在队列中。



科普:

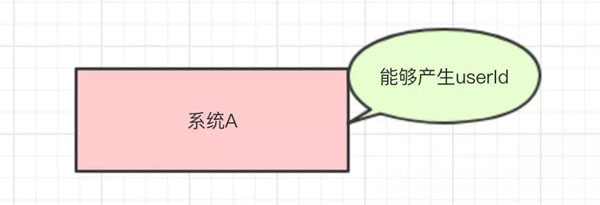
把数据放到消息队列叫作生产者

从消息队列里取数据叫作消费者

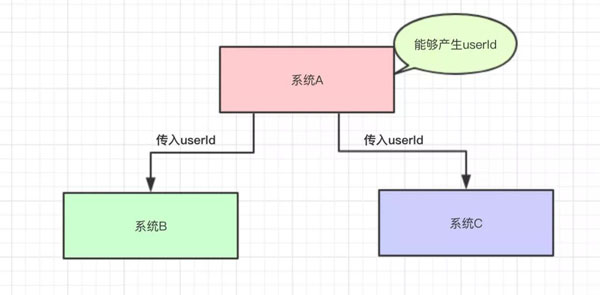
**2.为什么要用消息队列**

## 2.1解耦

我有一个系统A,系统A可以产生一个userId



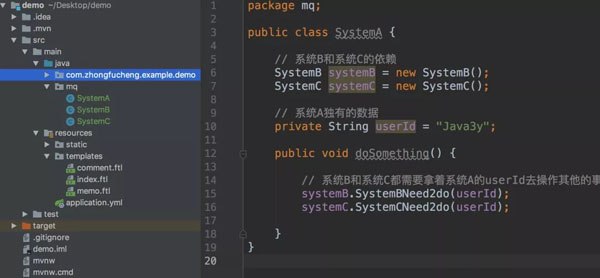
然后，现在有系统B和系统C都需要这个userId去做相关的操作



写成伪代码可能是这样的:

1. public class SystemA {
2. // 系统B和系统C的依赖
3. SystemB systemB = new SystemB();
4. SystemC systemC = new SystemC();
5. // 系统A独有的数据userId
6. private String userId = "Java3y";
7. public void doSomething() {
8. // 系统B和系统C都需要拿着系统A的userId去操作其他的事
9. systemB.SystemBNeed2do(userId);
10. systemC.SystemCNeed2do(userId);
11. }
12. }

结构如下



某一天，系统B的负责人告诉系统A的负责人，现在系统B的SystemBNeed2do(String userId)这个接口不再使用了，让系统A别去调它了。

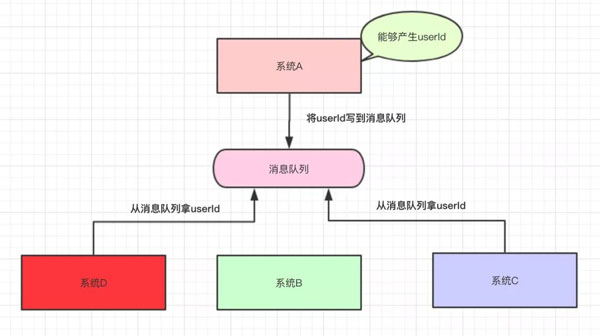
于是，系统A的负责人说"好的，那我就不调用你了。"，于是就把调用系统B接口的代码给删掉了：

又过了几天，系统D的负责人接了个需求，也需要用到系统A的userId，于是就跑去跟系统A的负责人说："老哥，我要用到你的userId，你调一下我的接口吧"

于是系统A说："没问题的，这就搞"

公司招来一个大佬，大佬经过几天熟悉，上来就说：将系统A的userId写到消息队列中，这样系统A就不用经常改动了。

为什么?



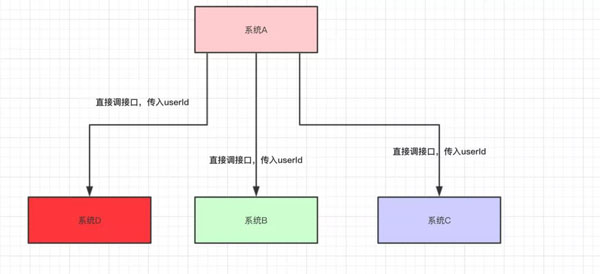
系统A将userId写到消息队列中，系统C和系统D从消息队列中拿数据。这样有什么好处？

* 系统A只负责把数据写到队列中，谁想要或不想要这个数据(消息)，系统A一点都不关心。
* 即便现在系统D不想要userId这个数据了，系统B又突然想要userId这个数据了，都跟系统A无关，系统A一点代码都不用改。
* 系统D拿userId不再经过系统A，而是从消息队列里边拿。系统D即便挂了或者请求超时，都跟系统A无关，只跟消息队列有关。

这样一来，系统A与系统B、C、D都解耦了。

## 2.2异步

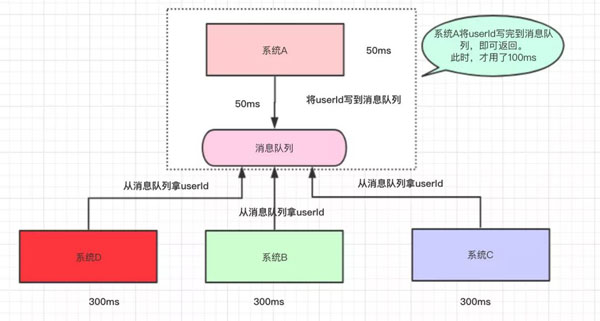
我们再来看看下面这种情况：系统A还是直接调用系统B、C、D



假设系统A运算出userId具体的值需要50ms，调用系统B的接口需要300ms，调用系统C的接口需要300ms，调用系统D的接口需要300ms。那么这次请求就需要50+300+300+300=950ms

并且我们得知，系统A做的是主要的业务，而系统B、C、D是非主要的业务。比如系统A处理的是订单下单，而系统B是订单下单成功了，那发送一条短信告诉具体的用户此订单已成功，而系统C和系统D也是处理一些小事而已。

那么此时，为了提高用户体验和吞吐量，其实可以异步地调用系统B、C、D的接口。所以，我们可以弄成是这样的：



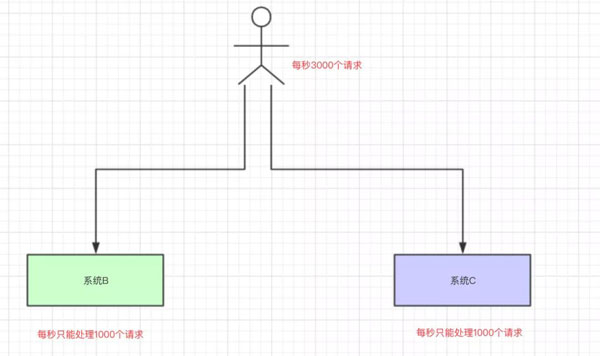
系统A执行完了以后，将userId写到消息队列中，然后就直接返回了(至于其他的操作，则异步处理)。

* 本来整个请求需要用950ms(同步)
* 现在将调用其他系统接口异步化，从请求到返回只需要100ms(异步)

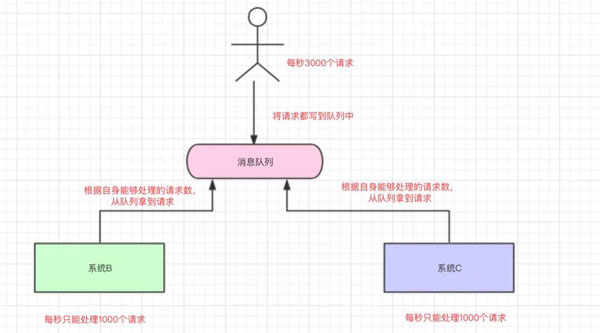
(例子可能举得不太好，但我觉得说明到点子上就行了，见谅。)

## 2.3削峰/限流

我们再来一个场景，现在我们每个月要搞一次大促，大促期间的并发可能会很高的，比如每秒3000个请求。假设我们现在有两台机器处理请求，并且每台机器只能每次处理1000个请求。



那多出来的1000个请求，可能就把我们整个系统给搞崩了…所以，有一种办法，我们可以写到消息队列中：



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。

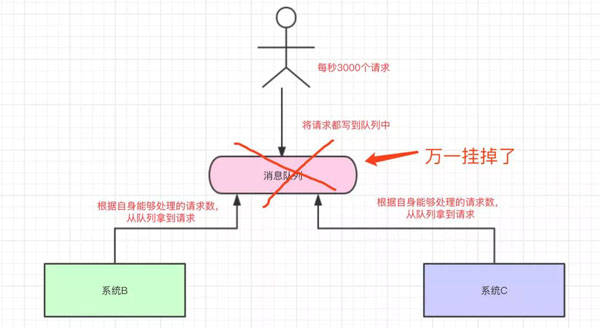
**3.使用消息队列有什么问题？**

经过我们上面的场景，我们已经可以发现，消息队列能做的事其实还是蛮多的。

说到这里，我们先回到文章的开头，"明明JDK已经有不少的队列实现了，我们还需要消息队列中间件呢？"其实很简单，JDK实现的队列种类虽然有很多种，但是都是简单的内存队列。为什么我说JDK是简单的内存队列呢？下面我们来看看要实现消息队列(中间件)可能要考虑什么问题。

## 3.1高可用

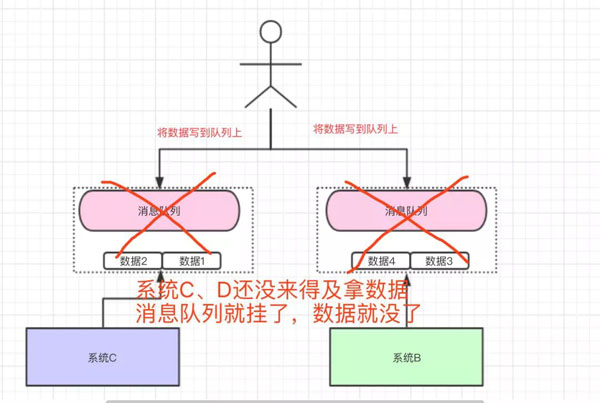
无论是我们使用消息队列来做解耦、异步还是削峰，消息队列肯定不能是单机的。试着想一下，如果是单机的消息队列，万一这台机器挂了，那我们整个系统几乎就是不可用了。



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。

## 3.2 数据丢失问题

我们将数据写到消息队列上，系统B和C还没来得及取消息队列的数据，就挂掉了。如果没有做任何的措施，我们的数据就丢了。



学过Redis的都知道，Redis可以将数据持久化磁盘上，万一Redis挂了，还能从磁盘从将数据恢复过来。同样地，消息队列中的数据也需要存在别的地方，这样才尽可能减少数据的丢失。

那存在哪呢？

* 磁盘？
* 数据库？
* Redis？
* 分布式文件系统？

同步存储还是异步存储？

## 3.3消费者怎么得到消息队列的数据？

消费者怎么从消息队列里边得到数据？有两种办法：

* 生产者将数据放到消息队列中，消息队列有数据了，主动叫消费者去拿(俗称push)
* 消费者不断去轮训消息队列，看看有没有新的数据，如果有就消费(俗称pull)

## 3.4其他

除了这些，我们在使用的时候还得考虑各种的问题：

* 消息重复消费了怎么办啊？
* 我想保证消息是绝对有顺序的怎么做？
* ……..

虽然消息队列给我们带来了那么多的好处，但同时我们发现引入消息队列也会提高系统的复杂性。市面上现在已经有不少消息队列轮子了，每种消息队列都有自己的特点，选取哪种MQ还得好好斟酌。

**4.消息队列技术选型**

对消息队列有几个要求。

第一：消息要能够按生产顺序消费；

第二：性能(QPS)要好；

第三：消息要能够持久化；

最后要尽可能保证数据的幂等性。

可以从社区查看他的更新频率

ActiveMq几个月才发一次版本，据说研究重心在他们的下一代产品Apollo。

RabbitMQ版本发布比ActiveMq频繁很多。至于RocketMQ和kafka就不带大家看了，总之也比ActiveMQ活跃的多。详情，可自行查阅。

再来一个性能对比表

| **特性** | **ActiveMQ** | **RabbitMQ** | **RocketMQ** | **kafka** |
| --- | --- | --- | --- | --- |
| 开发语言 | java | erlang | java | scala |
| 单机吞吐量 | 万级 | 万级 | 10万级 | 10万级 |
| 时效性 | ms级 | us级 | ms级 | ms级以内 |
| 可用性 | 高(主从架构) | 高(主从架构) | 非常高(分布式架构) | 非常高(分布式架构) |
| 功能特性 | 成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好 | 基于erlang开发，所以并发能力很强，性能极其好，延时很低;管理界面较丰富 | MQ功能比较完备，扩展性佳 | 只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广。 |
|  |  |  |  |  |

综合上面的材料得出以下两点:  
(1)中小型软件公司，建议选RabbitMQ.一方面，erlang语言天生具备高并发的特性，而且他的管理界面用起来十分方便。正所谓，成也萧何，败也萧何！他的弊端也在这里，虽然RabbitMQ是开源的，然而国内有几个能定制化开发erlang的程序员呢？所幸，RabbitMQ的社区十分活跃，可以解决开发过程中遇到的bug，这点对于中小型公司来说十分重要。不考虑rocketmq和kafka的原因是，一方面中小型软件公司不如互联网公司，数据量没那么大，选消息中间件，应首选功能比较完备的，所以kafka排除。不考虑rocketmq的原因是，rocketmq是阿里出品，如果阿里放弃维护rocketmq，中小型公司一般抽不出人来进行rocketmq的定制化开发，因此不推荐。  
(2)大型软件公司，根据具体使用在rocketMq和kafka之间二选一。一方面，大型软件公司，具备足够的资金搭建分布式环境，也具备足够大的数据量。针对rocketMQ,大型软件公司也可以抽出人手对rocketMQ进行定制化开发，毕竟国内有能力改JAVA源码的人，还是相当多的。至于kafka，根据业务场景选择，如果有日志采集功能，肯定是首选kafka了。具体该选哪个，看使用场景。

那所以第二个问题来了，怎么保证消息队列消费的幂等性？

其实还是得结合业务来思考，我这里给几个思路：

（1）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

（2）比如你是写redis，那没问题了，反正每次都是set，天然幂等性

（3）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。

还有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，我们之前线上系统就有这个问题，就是拿到数据的时候，每次重启可能会有重复，因为kafka消费者还没来得及提交offset，重复数据拿到了以后我们插入的时候，因为有唯一键约束了，所以重复数据只会插入报错，不会导致数据库中出现脏数据

如何保证MQ的消费是幂等性的，需要结合具体的业务来看

1.先讲严重的：服务挂掉。

这得从ActiveMQ的储存机制说起。在通常的情况下，非持久化消息是存储在内存中的，持久化消息是存储在文件中的，它们的最大限制在配置文件的<systemUsage>节点中配置。但是，在非持久化消息堆积到一定程度，内存告急的时候，ActiveMQ会将内存中的非持久化消息写入临时文件中，以腾出内存。虽然都保存到了文件里，但它和持久化消息的区别是，重启后持久化消息会从文件中恢复，非持久化的临时文件会直接删除。

那如果文件增大到达了配置中的最大限制的时候会发生什么？我做了以下实验：

设置2G左右的持久化文件限制，大量生产持久化消息直到文件达到最大限制，此时生产者阻塞，但消费者可正常连接并消费消息，等消息消费掉一部分，文件删除又腾出空间之后，生产者又可继续发送消息，服务自动恢复正常。

设置2G左右的临时文件限制，大量生产非持久化消息并写入临时文件，在达到最大限制时，生产者阻塞，消费者可正常连接但不能消费消息，或者原本慢速消费的消费者，消费突然停止。整个系统可连接，但是无法提供服务，就这样挂了。

具体原因不详，解决方案：尽量不要用非持久化消息，非要用的话，将临时文件限制尽可能的调大。

详细配置信息见文档：http://activemq.apache.org/producer-flow-control.html

2.丢消息。

这得从java的java.net.SocketException异常说起。简单点说就是当网络发送方发送一堆数据，然后调用close关闭连接之后。这些发送的数据都在接收者的缓存里，接收者如果调用read方法仍旧能从缓存中读取这些数据，尽管对方已经关闭了连接。但是当接收者尝试发送数据时，由于此时连接已关闭，所以会发生异常，这个很好理解。不过需要注意的是，当发生SocketException后，原本缓存区中数据也作废了，此时接收者再次调用read方法去读取缓存中的数据，就会报Software caused connection abort: recv failed错误。

通过抓包得知，ActiveMQ会每隔10秒发送一个心跳包，这个心跳包是服务器发送给客户端的，用来判断客户端死没死。如果你看过上面第一条，就会知道非持久化消息堆积到一定程度会写到文件里，这个写的过程会阻塞所有动作，而且会持续20到30秒，并且随着内存的增大而增大。当客户端发完消息调用connection.close()时，会期待服务器对于关闭连接的回答，如果超过15秒没回答就直接调用socket层的close关闭tcp连接了。这时客户端发出的消息其实还在服务器的缓存里等待处理，不过由于服务器心跳包的设置，导致发生了java.net.SocketException异常，把缓存里的数据作废了，没处理的消息全部丢失。

解决方案：用持久化消息，或者非持久化消息及时处理不要堆积，或者启动事务，启动事务后，commit()方法会负责任的等待服务器的返回，也就不会关闭连接导致消息丢失了。

关于java.net.SocketException请看我的详细研究：http://blog.163.com/\_kid/blog/static/3040547620160231534692/

3.持久化消息非常慢。

默认的情况下，非持久化的消息是异步发送的，持久化的消息是同步发送的，遇到慢一点的硬盘，发送消息的速度是无法忍受的。但是在开启事务的情况下，消息都是异步发送的，效率会有2个数量级的提升。所以在发送持久化消息时，请务必开启事务模式。其实发送非持久化消息时也建议开启事务，因为根本不会影响性能。

4.消息的不均匀消费。

有时在发送一些消息之后，开启2个消费者去处理消息。会发现一个消费者处理了所有的消息，另一个消费者根本没收到消息。原因在于ActiveMQ的prefetch机制。当消费者去获取消息时，不会一条一条去获取，而是一次性获取一批，默认是1000条。这些预获取的消息，在还没确认消费之前，在管理控制台还是可以看见这些消息的，但是不会再分配给其他消费者，此时这些消息的状态应该算作“已分配未消费”，如果消息最后被消费，则会在服务器端被删除，如果消费者崩溃，则这些消息会被重新分配给新的消费者。但是如果消费者既不消费确认，又不崩溃，那这些消息就永远躺在消费者的缓存区里无法处理。更通常的情况是，消费这些消息非常耗时，你开了10个消费者去处理，结果发现只有一台机器吭哧吭哧处理，另外9台啥事不干。

解决方案：将prefetch设为1，每次处理1条消息，处理完再去取，这样也慢不了多少。

详细文档：http://activemq.apache.org/what-is-the-prefetch-limit-for.html

5.死信队列。

如果你想在消息处理失败后，不被服务器删除，还能被其他消费者处理或重试，可以关闭AUTO\_ACKNOWLEDGE，将ack交由程序自己处理。那如果使用了AUTO\_ACKNOWLEDGE，消息是什么时候被确认的，还有没有阻止消息确认的方法？有！

消费消息有2种方法，一种是调用consumer.receive()方法，该方法将阻塞直到获得并返回一条消息。这种情况下，消息返回给方法调用者之后就自动被确认了。另一种方法是采用listener回调函数，在有消息到达时，会调用listener接口的onMessage方法。在这种情况下，在onMessage方法执行完毕后，消息才会被确认，此时只要在方法中抛出异常，该消息就不会被确认。那么问题来了，如果一条消息不能被处理，会被退回服务器重新分配，如果只有一个消费者，该消息又会重新被获取，重新抛异常。就算有多个消费者，往往在一个服务器上不能处理的消息，在另外的服务器上依然不能被处理。难道就这么退回--获取--报错死循环了吗？

在重试6次后，ActiveMQ认为这条消息是“有毒”的，将会把消息丢到死信队列里。如果你的消息不见了，去ActiveMQ.DLQ里找找，说不定就躺在那里。