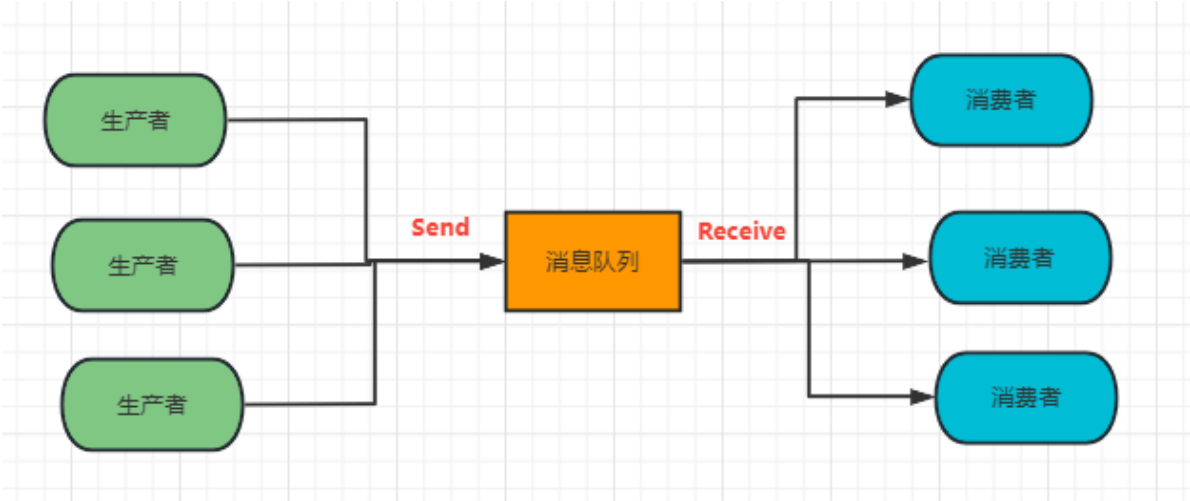
消息队列

1. 消息队列概述



消息队列（Message Queue）是一种进程间通信或同一进程的不同线程间的通信方式。进程或者线程之间通过消息进行通信，消息发送后可以立即返回，由消息系统来确保信息的可靠传递，消息发布者（生产者）只管把消息发布到消息队里中而不用管谁来消费，消息使用者（消费者）只管从消息队列中获取消息以进一步处理而不用管理谁发布的消息，这样发布者和使用者都不用知道对方的存在。

消息（Message）是指在应用之间传送的数据。消息可以非常简单，比如只包含文本字符串，也可以很复杂，如嵌入对象。

2. 消息队列的特点

2.1 采用异步处理模式

消息发送者可以发送一个消息而无须等待响应。消息发送者将消息发送到一条虚拟的通道（主题或队列）上，消息接收者则订阅或是监听该通道。一条信息可能最终转发给一个或多个消息接收者，这些接收者都无需对消息发送者做出同步回应。整个过程都是异步的。

2.2 应用系统之间解耦合

主要体现在如下两点：

- 发送者和接受者不必了解对方、只需要确认消息；
- 发送者和接受者不必同时在线。

比如在线交易系统为了保证数据的最终一致，在支付系统处理完成后会把支付结果放到消息中间件里，通知订单系统修改订单支付状态。两个系统是通过消息中间件解耦的。

2.3 分布式

通过对消费者的横向扩展，降低了消息队列阻塞的风险，以及单个消费者产生单点故障的可能性。

2.4 可靠性

消息队列一般会把接收的消息存储到本地硬盘上（当消息被处理完之后，存储信息根据不同的消息队列实现，有可能将其删除），这样即使应用挂掉或者消息队列本身挂掉，消息也能够重新加载。

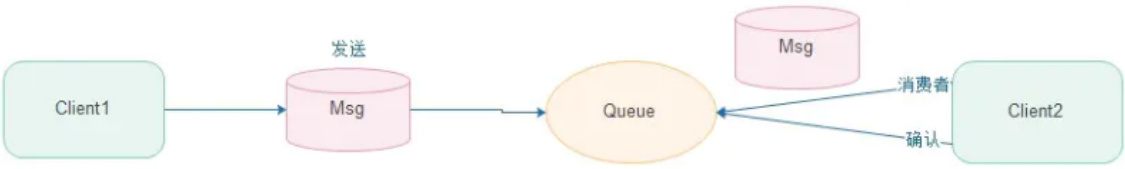
3. 消息模型

3.1 点对点模型

点对点模型用于**消息生产者**和**消息消费者**之间**点到点**的通信。消息生产者将消息发送到由某个名字标识的特定消费者。这个名字实际上对于消费服务中的一个**队列**（`Queue`），在消息传递给消费者之前它被**存储**在这个队列中。**队列消息**可以放在**内存**中也可以**持久化**，以保证在消息服务出现故障时仍然能够传递消息。

传统的点对点消息中间件通常由**消息队列服务**、**消息传递服务**、**消息队列**和**消息应用程序接口** `API` 组成，如下图所示：

![image-20230104145907112](images/image-20230104145907112.png



特点

- 每个消息只有一个消费者（Consumer），即一旦消息被消费，消息就不再在消息队列中。
- 生产者和消费者之间没有依赖性，生产者发送消息之后，不管有没有消费者在运行，都不会影响到生产者下次发送消息。
- 消费者在成功接收消息之后需向队列应答成功，以便消息队列删除当前接收的消息。

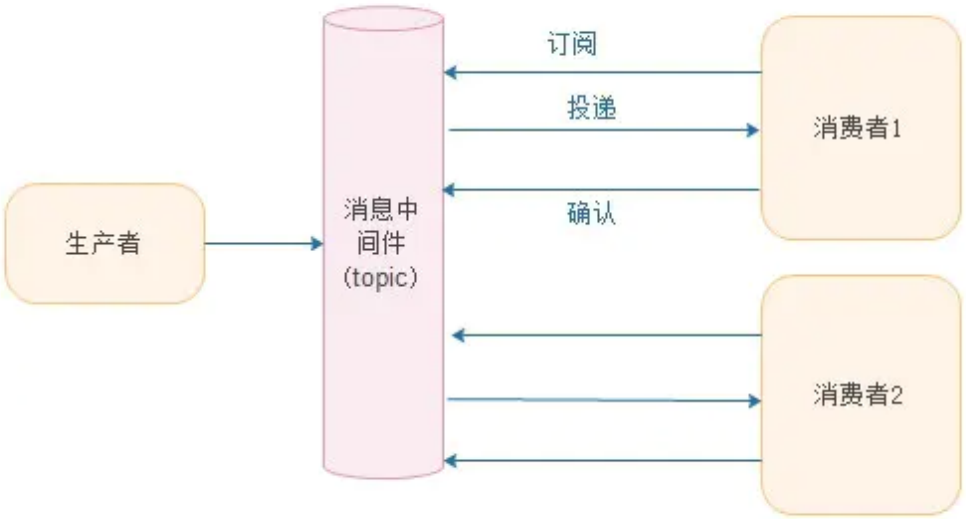
3.2 发布/订阅模型（Pub/Sub）

发布者/订阅者模型支持向一个特定的**消息主题**生产消息。`0`或**多个订阅者**可能对接收来自**特定消息主题**的消息感兴趣。

在这种模型下，发布者和订阅者彼此不知道对方，就好比是匿名公告板。

这种模式被概况为：多个消费者可以获得消息，在**发布者**和**订阅者**之间存在**时间依赖性。**发布者需要建立一个**订阅**（`subscription`），以便能够消费者订阅。**订阅者**必须保持**持续的活动状态**并**接收消息。**

在这种情况下，在订阅者**未连接时**，发布的消息将在订阅者**重新连接**时**重新发布**，如下图所示：



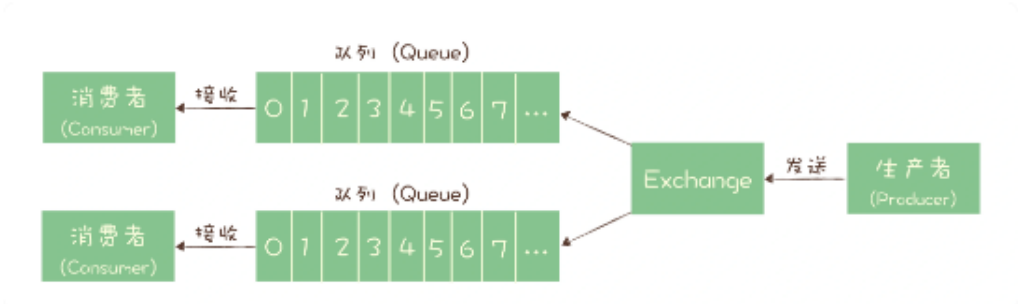
特点

- 每个消息可以有多个订阅者。
- 发布者和订阅者之间有时间上的依赖性，针对某个主题（Topic）的订阅者，它必须创建一个订阅之后，才能消费发布者的消息。
- 为了消费消息，订阅者需要提前订阅该角色主题，并保持在线运行。

3.3 中间件

3.3.1 RabbitMQ的消息模型

在 RabbitMQ 中，Exchange 位于生产者和队列之间，生产者并不关心将消息发送给哪个队列，而是将消息发送给 Exchange，由 Exchange 上配置的策略来决定将消息投递到哪些队列中。



同一份消息如果需要被多个消费者来消费，需要配置 Exchange 将消息发送到多个队列，每个队列中都存放一份完整的消息数据，可以为一个消费者提供消费服务。这也可以变相地实现新发布 - 订阅模型中，“一份消息数据可以被多个订阅者来多次消费”这样的功能。

3.3.2 RocketMQ的消息模型

- RocketMQ队列的作用?

可以确保消息的有序性。但是同时也带来新的问题，为了确保消息的有序性，在某一条消息被成功消费之前，下一条消息是不能被消费的，否则就会出现消息空洞，违背了有序性这个原则。也就是说，每个主题在任意时刻，至多只能有一个消费者实例在进行消费，那就没法通过水平扩展消费者的数量来提升消费端总体的消费性能。为了解决这个问题，RocketMQ 在主题下面增加了队列的概念。

- RocketMQ如何提升消费性能

  一个主题可以包含多个队列，可以通过多个队列来实现多实例并行生产和消费。

  （需要注意的是，RocketMQ 只在队列上保证消息的有序性，主题层面是无法保证消息的严格顺序的）

- RocketMQ消费组

  每个消费组通过订阅主题，都可以消费主题中一份完整的消息，不同消费组之间消费进度彼此不受影响，也就是说，一条消息被 Consumer Group1 消费过，也会再给 Consumer Group2 消费。

  消费组中包含多个消费者，同一个组内的消费者是竞争消费的关系，每个消费者负责消费组内的一部分消息。如果一条消息被消费者 Consumer1 消费了，那同组的其他消费者就不会再收到这条消息。

> 消费者组和队列数没有关系，消费组是为了重复消费消息，而队列是为了并行消费，提升消费速度。

3.3.3 Kafka消息的模型

Kafka 的消息模型和 RocketMQ 是完全一样的，RocketMQ 中对应的概念，和生产消费过程中的确认机制，都完全适用于 Kafka。

唯一的区别是，在 Kafka 中，队列这个概念的名称不一样，Kafka 中对应的名称是“分区（Partition）”，含义和功能是没有任何区别的。

4. 推拉模型

- **Push推消息模型**：消息生产者将消息发送给消息队列，消息队列又将消息推给消息消费者。
- **Pull拉消息模型**：消息生产者将消息发送给消息队列，消息消费者从消息队列中拉该消息。

两种类型区别：

模型	Push	Pull
服务端	消息存储 处理请求 保存推送轨迹 保存订阅关系 消费者负载均衡 集中式	消息存储 处理请求 分布式
客户端	处理响应和请求	处理响应和请求 保存pull状态，如拉取位置的偏移量 offset 异常情况下的消息暂存和recover
实时性	较好，收到数据后可立即发送给客户端	取决于pull的间隔时间
消费者故障	消费者故障情况下，服务端堆积消息，重复推送耗费资源。 保存推送轨迹压力很大	消费者故障，对服务端无影响
其他	对消息推送有更多控制，能实现多样化的推送机制。 当消费者数量增多的时候，推送压力大，性能天花板。 消费者处理能力差异，导致堆消息。	需要在客户端实现消息过滤，浪费资源。 需要在不同客户端之间协调，做负载均衡。

5. 应用场景

消息队列最常被使用的三种场景：异步处理、流量控制和服务解耦。当然，消息队列的适用范围不仅仅局限于这些场景，还有包括：

- 作为发布 / 订阅系统实现一个微服务级系统间的观察者模式；
- 连接流计算任务和数据；
- 用于将消息广播给大量接收者。

简单的说，我们在单体应用里面需要用队列解决的问题，在分布式系统中大多都可以用消息队列来解决。

同时我们也要认识到，消息队列也有它自身的一些问题和局限性，包括：

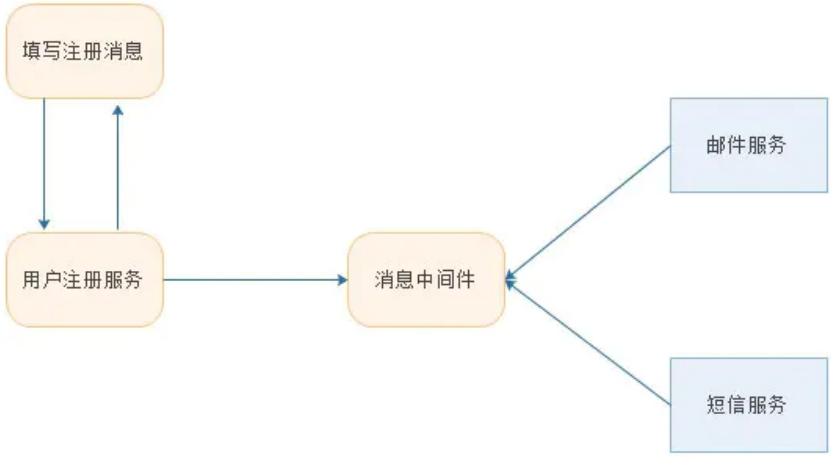
- 引入消息队列带来的延迟问题；
- 增加了系统的复杂度；
- 可能产生数据不一致的问题。

5.1 异步处理

`目的：减少请求响应时间 ，实现非核心流程异步化，提高系统响应性能。`

- 同步处理是指从请求的发起一直到最终的处理完成期间，请求的调用方一直在同步阻塞等待调用的处理完成。
- 异步处理处理是指在请求发起的处理过程中，客户端的代码已经返回了，它可以继续进行自己的后续操作，而不需要等待调用处理完成。

对一些比较耗时且不需要即时（同步）返回操作结果的操作，可以把处理过程通过消息队列进行异步处理。这样做可以推迟耗时操作的处理，使耗时操作异步化，而不必阻塞客户端程序，客户端的程序在得到处理结果之前可以继续执行，从而提高客户端程序的处理性能。

> ## 应用案例
>
> ①网站用户注册，注册成功后会过一会发送邮件确认或者短信。
>
> 
>
>
> ②如何设计一个秒杀系统?
>
> 秒杀系统需要解决的核心问题是，如何利用有限的服务器资源，尽可能多地处理短时间内的海量请求。我们知道，处理一个秒杀请求包含了很多步骤，例如：
>
> - 风险控制；
> - 库存锁定；
> - 生成订单；
> - 短信通知；
> - 更新统计数据。
>
> 如果没有任何优化，正常的处理流程是：App 将请求发送给网关，依次调用上述 5 个流程，然后将结果返回给 APP。
>
> 对于这 5 个步骤来说，能否决定秒杀成功，实际上只有风险控制和库存锁定这 2 个步骤。只要用户的秒杀请求通过风险控制，并在服务端完成库存锁定，就可以给用户返回秒杀结果了，对于后续的生成订单、短信通知和更新统计数据等步骤，并不一定要在秒杀请求中处理完成。

> 所以当服务端完成前面 2 个步骤，确定本次请求的秒杀结果后，就可以马上给用户返回响应，然后把请求的数据放入消息队列中，由消息队列异步地进行后续的操作。

> 处理一个秒杀请求，从 5 个步骤减少为 2 个步骤，这样不仅响应速度更快，并且在秒杀期间，我们可以把大量的服务器资源用来处理秒杀请求。秒杀结束后再把资源用于处理后面的步骤，充分利用有限的服务器资源处理更多的秒杀请求。

5.2 系统解耦

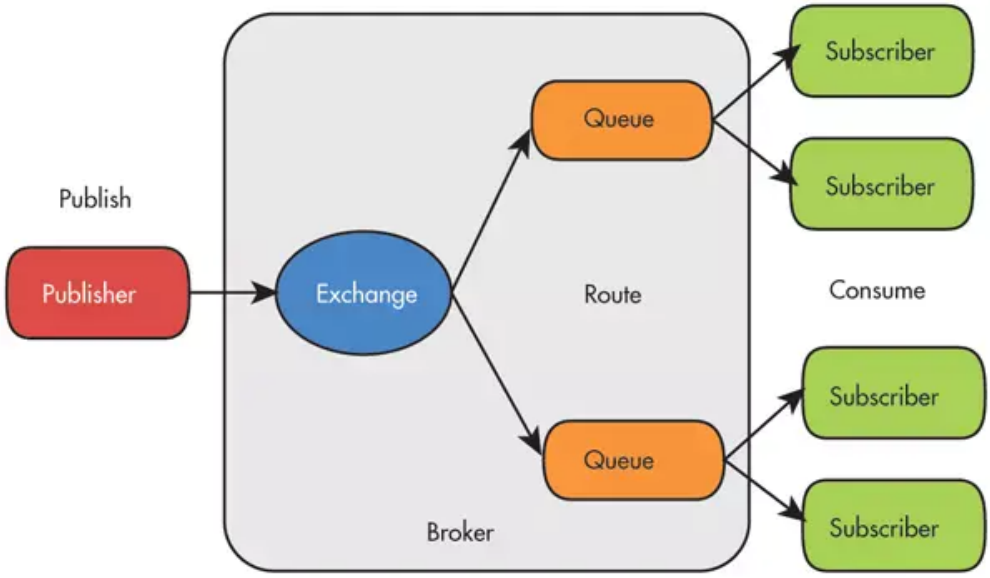
`目的：提升系统健壮性、可维护性`

消息队列的另外一个作用，就是实现系统应用之间的解耦。

> **应用案例**
>
> 我们知道订单是电商系统中比较核心的数据，当一个新订单创建时：
>
> - 支付系统需要发起支付流程；
> - 风控系统需要审核订单的合法性；
> - 客服系统需要给用户发短信告知用户；
> - 经营分析系统需要更新统计数据；
> - ……
>
> 这些订单下游的系统都需要实时获得订单数据。随着业务不断发展，这些订单下游系统不断的增加，不断变化，并且每个系统可能只需要订单数据的一个子集，负责订单服务的开发团队不得不花费很大的精力，应对不断增加变化的下游系统，不停地修改调试订单系统与这些下游系统的接口。任何一个下游系统接口变更，都需要订单模块重新进行一次上线，对于一个电商的核心服务来说，这几乎是不可接受的。
>
> 所有的电商都选择用消息队列来解决类似的系统耦合过于紧密的问题。引入消息队列后，订单服务在订单变化时发送一条消息到消息队列的一个主题 Order 中，所有下游系统都订阅主题 Order，这样每个下游系统都可以获得一份实时完整的订单数据。
>
> 无论增加、减少下游系统或是下游系统需求如何变化，订单服务都无需做任何更改，实现了订单服务与下游服务的解耦。

5.3 广播

生产者/消费者模式，只需要关心消息是否**送达队列**，至于谁希望订阅和需要消费，是**下游**的事情，无疑极大地减少了开发和联调的工作量。



5.4 流量削峰和流控

`目的：削峰填谷，平摊流量，避免服务崩溃`

当**上下游系统**处理能力存在差距的时候，利用**消息队列**做一个通用的**“漏斗”**，进行**限流控制**。在下游有能力处理的时候，再进行分发。

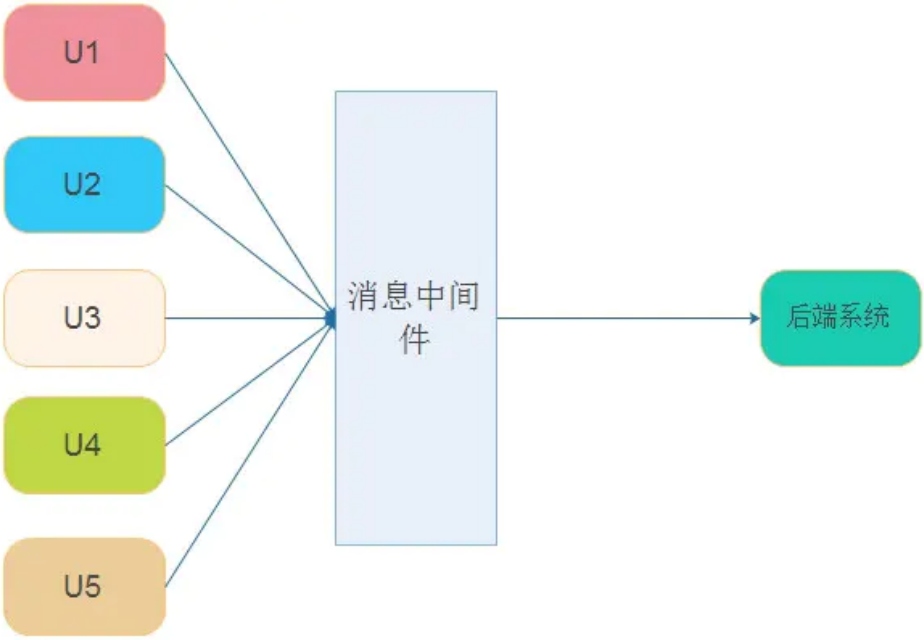
举个例子：

用户在支付系统成功结账后，订单系统会通过短信系统向用户推送扣费通知。**短信系统**可能由于**短板效应**，速度卡在**网关**上（每秒几百次请求），跟**前端的并发量**不是一个数量级。于是，就造成**支付系统**和**短信系统**的处理能力出现差异化。

然而用户晚上个半分钟左右收到短信，一般是不会有太大问题的。如果没有消息队列，两个系统之间通过**协商**、**滑动窗口**等复杂的方案也不是说不能实现。但**系统复杂性**指数级增长，势必在**上游**或者**下游**做**存储**，并且要处理**定时**、**拥塞**等一系列问题。而且每当有**处理能力有差距**的时候，都需要**单独**开发一套逻辑来维护这套逻辑。

所以，利用中间系统转储两个系统的通信内容，并在下游系统有能力处理这些消息的时候，再处理这些消息，是一套相对较通用的方式。

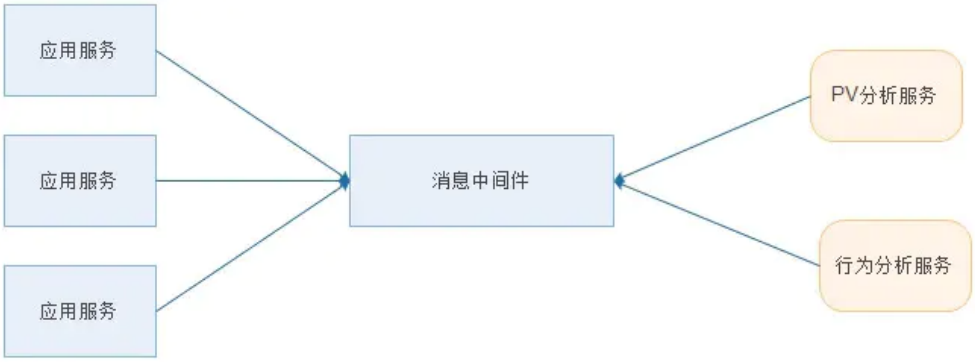
> ### 应用案例
>
> - 把消息队列当成可靠的**消息暂存地**，进行一定程度的**消息堆积**；
> - 定时进行消息投递，比如模拟**用户秒杀**访问，进行**系统性能压测**。



5.5 日志处理

将消息队列用在**日志处理**中，比如`Kafka`的应用，解决**海量日志**传输和缓冲的问题。

> **应用案例**
>
> 把日志进行集中收集，用于计算`PV`、**用户行为分析**等等。



5.7 消息通讯

消息队列一般都内置了**高效的通信机制**，因此也可以用于单纯的**消息通讯**，比如实现**点对点消息队列**或者**聊天室**等。

6. 分类

6.1 MQ的分类

- ActiveMQ

  优点：单机吞吐量万级，时效性 ms 级，可用性高，基于主从架构实现高可用性，消息可靠性较低的概率丢失数据。

  缺点:官方社区现在对 ActiveMQ 5.x 维护越来越少，高吞吐量场景较少使用。

- Kafka

  优点: 性能卓越，单机写入 TPS 约在百万条/秒，最大的优点，就是吞**吐量高。**时效性 ms 级可用性非常高，kafka 是分布式的，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,消费者采用 Pull 方式获取消息, 消息有序, 通过控制能够保证所有消息被消费且仅被消费一次;有优秀的第三方KafkaWeb 管理界面 Kafka-Manager；在日志领域比较成熟，被多家公司和多个开源项目使用；功能支持： 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

  缺点：Kafka 单机超过 64 个队列/分区，Load 会发生明显的飙高现象，队列越多，load 越高，发送消息响应时间变长，使用短轮询方式，实时性取决于轮询间隔时间，消费失败不支持重试；支持消息顺序，但是一台代理宕机后，就会产生消息乱序，**社区更新较慢**；

- RocketMQ

  RocketMQ 出自阿里巴巴的开源产品，用 Java 语言实现，在设计时参考了 Kafka，并做出了自己的一些改进。被阿里巴巴广泛应用在订单，交易，充值，流计算，消息推送，日志流式处理，binglog 分发等场景

  优点：单机吞吐量十万级,可用性非常高，分布式架构,消息可以做到 0 丢失,MQ 功能较为完善，还是分布式的，扩展性好,支持 10 亿级别的消息堆积，不会因为堆积导致性能下降,源码是 java 我们可以自己阅读源码，定制自己公司的 MQ
  缺点：支持的客户端语言不多，目前是 java 及 c++，其中 c++不成熟；社区活跃度一般,没有在MQ核心中去实现 JMS 等接口,有些系统要迁移需要修改大量代码

- RabbitMQ

  优点：由于 erlang 语言的高并发特性，性能较好；吞吐量到万级，MQ 功能比较完备,健壮、稳定、易用、跨平台、支持多种语言 如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持 AJAX 文档齐全；开源提供的管理界面非常棒，用起来很好用,社区活跃度高；更新频率相当高。

  缺点：erlang开发，很难去看懂源码，基本职能依赖于开源社区的快速维护和修复bug，不利于做二次开发和维护；RabbitMQ确实吞吐量会低一些，这是因为他做的实现机制比较重；需要学习比较复杂的接口和协议，学习和维护成本较高。

6.2 MQ的选择

- Kafka
  Kafka 主要特点是基于Pull 模式来处理消息消费，追求高吞吐量，一开始的目的就是用于日志收集和传输，适合产生大量数据的互联网服务的数据收集业务。大型公司建议可以选用，如果有日志采集功能，肯定是首选 kafka 了。

- RocketMQ
  天生为金融互联网领域而生，对于可靠性要求很高的场景，尤其是电商里面的订单扣款，以及业务削峰，在大量交易涌入时，后端可能无法及时处理的情况。RoketMQ在稳定性上可能更值得信赖，这些业务场景在阿里双 11已经经历了多次考验，如果你的业务有上述并发

场景，建议可以选择 RocketMQ。

- RabbitMQ

  结合erlang语言本身的并发优势，性能好时效性微秒级，社区活跃度也比较高，管理界面用起来十分方便，如果你的数据量没有那么大，中小型公司优先选择功能比较完备的RabbitMQ。

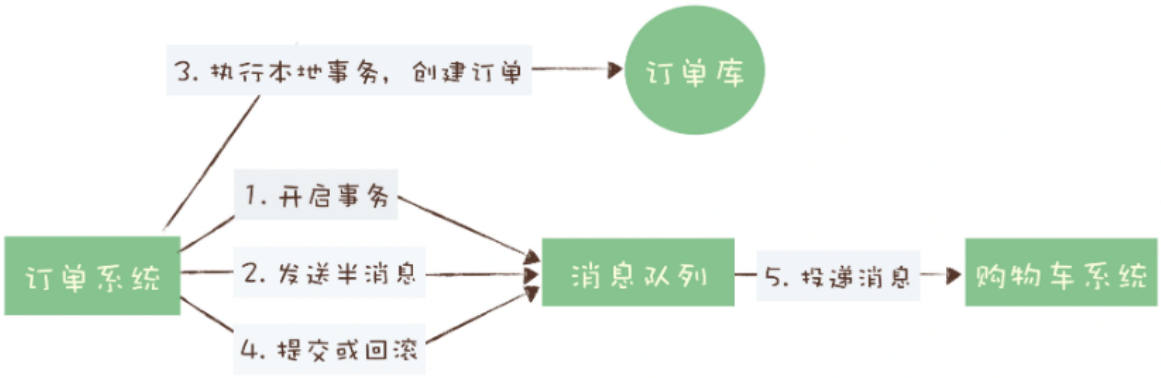
7. 其他

7.1 事务问题

事务消息需要消息队列提供相应的功能才能实现，Kafka 和 RocketMQ 都提供了事务相关功能。

例如：

订单和购物车这个例子，我们一起来看下如何用消息队列来实现分布式事务。



- 首先，订单系统在消息队列上开启一个事务。
- 然后订单系统给消息服务器发送一个“半消息”。

  这个**半消息**不是说消息内容不完整，它包含的内容就是完整的消息内容，半消息和普通消息的唯一区别是，在事务提交之前，对于消费者来说，这个消息是不可见的。半消息发送成功后，订单系统就可以执行本地事务了，在订单库中创建一条订单记录，并提交订单库的数据库事务。

- 然后根据本地事务的执行结果决定提交或者回滚事务消息。

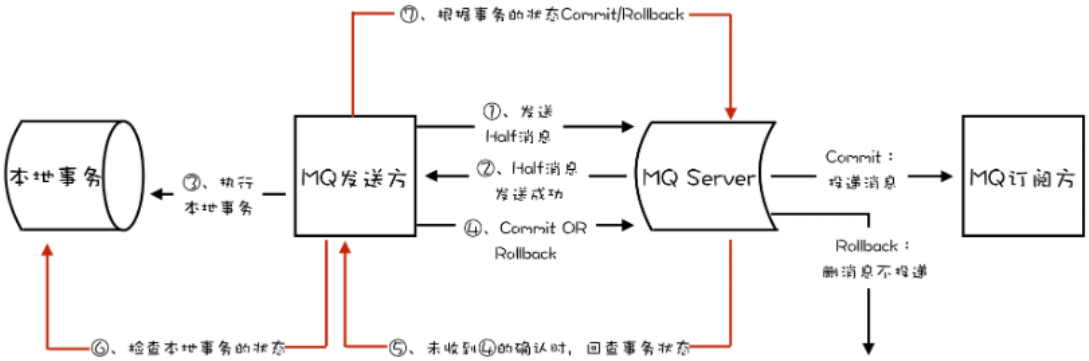
  如果订单创建成功，那就提交事务消息，购物车系统就可以消费到这条消息继续后续的流程。

  如果订单创建失败，那就回滚事务消息，购物车系统就不会收到这条消息。

这样就基本实现了“要么都成功，要么都失败”的一致性要求。

如果在第四步提交事务消息时失败了怎么办？对于这个问题，Kafka 和 RocketMQ 给出了 2 种不同的解决方案。

- Kafka 的解决方案比较简单粗暴，直接抛出异常，让用户自行处理。我们可以在业务代码中反复重试提交，直到提交成功，或者删除之前创建的订单进行补偿
- RocketMQ 则增加了事务反查的机制来解决事务消息提交失败的问题



7.2 确保消息可靠传递

- 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: 在这个阶段，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: 在这个阶段，Consumer 从 Broker 上拉取消息，经过网络传输发送到 Consumer 上。

不同阶段如何保证消息可靠传递

- 生产阶段：需要在处理完全部消费业务逻辑之后，再发送消费确认。
- 存储阶段：可以通过配置刷盘和复制相关的参数，让消息写入到多个副本的磁盘上，来确保消息不会因为某个 Broker 宕机或者磁盘损坏而丢失。
- 消费阶段: 需要在处理完全部消费业务逻辑之后，再发送消费确认。

7.3 重复消费问题

用幂等性解决重复消息问题。一般解决重复消息的办法是，在消费端，让我们消费消息的操作具备幂等性。

- 利用数据库的唯一约束实现幂等（比如：ID）
- 为更新的数据设置前置条件

  具体的实现方法：给数据变更设置一个前置条件，如果满足条件就更新数据，否则拒绝更新数据，在更新数据的时候，同时变更前置条件中需要判断的数据。这样，重复执行这个操作时，由于第一次更新数据的时候已经变更了前置条件中需要判断的数据，不满足前置条件，则不会重复执行更新数据操作。
- 记录并检查操作（适用范围最广，但是实现难度和复杂度也比较高，一般不推荐使用）

  实现的思路：在执行数据更新操作之前，先检查一下是否执行过这个更新操作。

  具体的实现方法：在发送消息时，给每条消息指定一个全局唯一的 ID，消费时，先根据这个 ID 检查这条消息是否有被消费过，如果没有消费过，才更新数据，然后将消费状态置为已消费。

  存在的问题：全局唯一ID的实现有一定的复杂度，需要确保检查消费状态、更新数据、以及更新消费状态三个操作原子性，解决方式涉及到分布式锁和分布式事务，并且对高性能、高并发也有一定的影响。

7.4 消息积压问题

消息积压的直接原因，一定是系统中的某个部分出现了性能问题，来不及处理上游发送的消息，才会导致消息积压。

最粗粒度的原因，只有两种：要么是发送变快了，要么是消费变慢了。

- 优化性能来避免消息积压

  ①发送端性能优化

  可以通过增加批量或者是增加并发方式来提升性能

  > 例如：
  >
  > 如果消息发送端是一个微服务，主要接受 RPC 请求处理在线业务。很自然的，微服务在处理每次请求的时候，就在当前线程直接发送消息就可以了，因为所有 RPC 框架都是多线程支持多并发的，自然也就实现了并行发送消息。并且在线业务比较在意的是请求响应时延，选择批量发送必然会影响 RPC 服务的时延。这种情况，比较明智的方式就是通过并发来提升发送性能。
  >
  > 如果你的系统是一个离线分析系统，离线系统在性能上的需求是什么呢？它不关心时延，更注重整个系统的吞吐量。发送端的数据都是来自于数据库，这种情况就更适合批量发送，你可以批量从数据库读取数据，然后批量来发送消息，同样用少量的并发就可以获得非常高的吞吐量。

  ② 消费端性能优化

  在设计系统的时候，一定要保证消费端的消费性能要高于生产端的发送性能，这样的系统才能健康的持续运行。

  消费端的性能优化除了优化消费业务逻辑以外，也可以通过水平扩容，增加消费端的并发数来提升总体的消费性能。

  注意：在扩容 Consumer 的实例数量的同时，必须同步扩容主题中的分区（也叫队列）数量，确保 Consumer 的实例数和分区数量是相等的。如果 Consumer 的实例数量超过分区数量，这样的扩容实际上是没有效果的。原因我们之前讲过，因为对于消费者来说，在每个分区上实际上只能支持单线程消费。

- 如何处理消息积压问题

  ① 临时扩容，增加消费端，用硬件提升消费速度。

  ② 服务降级，关闭一些非核心业务，减少消息生产。

  ③ 通过日志分析，监控等找到挤压原因，消息队列三部分，上游生产者是否异常生产大量数据，中游消息队列存储层是否出现问题，下游消费速度是否变慢，就能确定哪个环节出了问题

  ④ 根据排查解决异常部分。

  ⑤ 等待积压的消息被消费，恢复到正常状态，撤掉扩容服务器。

8. 扩展

如何使用异步设计提升系统性能?

异步思想就是，当我们要执行一项比较耗时的操作时，不去等待操作结束，而是给这个操作一个命令：“当操作完成后，接下来去执行什么。”

使用异步编程模型，虽然并不能加快程序本身的速度，但可以减少或者避免线程等待，只用很少的线程就可以达到超高的吞吐能力。

同时我们也需要注意到异步模型的问题：相比于同步实现，异步实现的复杂度要大很多，代码的可读性和可维护性都会显著的下降。虽然使用一些异步编程框架会在一定程度上简化异步开发，但是并不能解决异步模型高复杂度的问题。异步性能虽好，但一定不要滥用，只有类似在像消息队列这种业务逻辑简单并且需要超高吞吐量的场景下，或者必须长时间等待资源的地方，才考虑使用异步模型。

如果系统的业务逻辑比较复杂，在性能足够满足业务需求的情况下，采用符合人类自然的思路且易于开发和维护的同步模型是更加明智的选择。

Kafka如何实现高性能IO

- 使用批量处理的方式来提升系统吞吐能力。
- 基于磁盘文件高性能顺序读写的特性来设计的存储结构。
- 利用操作系统的 PageCache 来缓存数据，减少 IO 并提升读性能。
- 使用零拷贝技术加速消费流程。

如何使用缓存来减少磁盘IO

按照读写性质，可以分为读写缓存和只读缓存，读写缓存实现起来非常复杂，并且只在消息队列等少数情况下适用。只读缓存适用的范围更广，实现起来也更简单。

在实现只读缓存的时候，你需要考虑的第一个问题是如何来更新缓存。这里面有三种方法：

- 第一种是在更新数据的同时去更新缓存
- 第二种是定期来更新全部缓存
- 第三种是给缓存中的每个数据设置一个有效期，让它自然过期以达到更新的目的。

这三种方法在更新的及时性上和实现的复杂度这两方面，都是依次递减的，你可以按需选择。

对于缓存的置换策略，最优的策略一定是你根据业务来设计的定制化的置换策略，当然你也可以考虑 LRU 这样通用的缓存置换算法。

如何正确使用锁保护共享数据，协调异步线程?

- 避免滥用锁

  ① 如果能不用锁，就不用锁；如果你不确定是不是应该用锁，那也不要用锁。虽然说使用锁可以保护共享资源，但是代价还是不小的。

  - 加锁和解锁过程都是需要 CPU 时间的，这是一个性能的损失
  - 如果对锁使用不当，很容易造成死锁，导致整个程序“卡死

  ② 只有在并发环境中，共享资源不支持并发访问，或者说并发访问共享资源会导致系统错误的情况下，才需要使用锁。

  ③ 使用完锁，一定要释放它

- 如何避免死锁

  死锁：由于某种原因，锁一直没有释放，后续需要获取锁的线程都将处于等待锁的状态，从而导致程序卡死。

关于避免死锁，建议如下：

- 避免滥用锁，程序里用的锁少，写出死锁 Bug 的几率自然就低。
- 对于同一把锁，加锁和解锁必须要放在同一个方法中，这样一次加锁对应一次解锁，代码清晰简单，便于分析问题。
- 尽量避免在持有一把锁的情况下，去获取另外一把锁，就是要尽量避免同时持有多把锁。
- 如果需要持有多把锁，一定要注意加解锁的顺序，解锁的顺序要和加锁顺序相反。比如，获取三把锁的顺序是 A、B、C，释放锁的顺序必须是 C、B、A。
- 给你程序中所有的锁排一个顺序，在所有需要加锁的地方，按照同样的顺序加解锁。

- 使用读写锁要兼顾性能和安全性
  - 读访问可以并发执行；
  - 写的同时不能并发读，也不能并发写。

如何用硬件同步原语（CAS）替代锁?

硬件同步原语（Atomic Hardware Primitives）是由计算机硬件提供的一组原子操作，我们比较常用的原语主要是 CAS 和 FAA 这两种。

- CAS（Compare and Swap），它的字面意思是：先比较，再交换。
- FAA 原语的语义是，先获取变量 p 当前的值 value，然后给变量 p 增加 inc，最后返回变量 p 之前的值 value

CSS和FAA原语的特殊之处就是，它们都是由计算机硬件，具体说就是 CPU 提供的实现，可以保证操作的原子性。

对于类似：“先读取数据，做计算，然后再更新数据”这样的业务逻辑，可以使用 CAS 原语 + 反复重试的方式来保证数据安全，前提是，线程之间的碰撞不能太频繁，否则太多重试会消耗大量的 CPU 资源，反而得不偿失。

数据压缩：时间换空间

- 数据压缩

  数据压缩不仅能节省存储空间，还可以用于提升网络传输性能。这种使用压缩来提升系统性能的方法，不仅限于在消息队列中使用，我们日常开发的应用程序也可以使用。比如，我们的程序要传输大量的数据，或者要在磁盘、数据库中存储比较大的数据，这些情况下，都可以考虑使用数据压缩来提升性能，还能节省网络带宽和存储空间。

- 什么情况适合使用数据压缩

  压缩它的本质是资源的置换，是一个时间换空间，或者说是 CPU 资源换存储资源的游戏。

  - 压缩和解压的操作都是计算密集型的操作，非常耗费 CPU 资源。如果你的应用处理业务逻辑就需要耗费大量的 CPU 资源，就不太适合再进行压缩和解压
  - 如果你的系统的瓶颈是磁盘的 IO 性能，CPU 资源又很闲，这种情况就非常适合在把数据写入磁盘前先进行压缩。
  - 如果你的系统读写比严重不均衡，你还要考虑，每读一次数据就要解压一次是不是划算。

- 应该选择什么压缩算法

  目前常用的压缩算法包括：ZIP，GZIP，SNAPPY，LZ4 等等。

  选择压缩算法的时候，主要需要考虑数据的压缩率和压缩耗时。一般来说，压缩率越高的算法，压缩耗时也越高。

    - 如果是对性能要求高的系统，可以选择压缩速度快的算法，比如 LZ4；
    - 如果需要更高的压缩比，可以考虑 GZIP 或者压缩率更高的 XZ 等算法。
- Kafka 是如何处理消息压缩的

  Kafka 在生产者上，对每批消息进行压缩，批消息在服务端不解压，消费者在收到消息之后再进行解压。简单地说，Kafka 的压缩和解压都是在客户端完成的。