消息队列与中间件

讲师：pansir

消息队列

消息队列用作企业级应用之间 内部通信

特点：

1. 低耦合
2. 可靠投递
3. 广播
4. 流量控制
5. 最终一致性

常见队列

RabbitMQ，RocketMQ，ActiveMQ，Kafka，ZeroMQ，MetaMQ，Redis，MySQL(也可以)

概念

消息队列 是指利用 **高效可靠** 的 **消息传递机制** 进行与平台无关的 **数据交流**，并基于 **数据通信** 来进行分布式系统的集成

说人话：A服务和B服务进行数据传递的一种手段

作用

通过提供 **消息传递** 和 **消息排队** 模型，它可以在 **分布式环境** 下提供如下功能：

1. 应用解耦：拆分功能 -> 拆分服务
2. 弹性伸缩
3. 冗余存储
4. 流量削峰：
5. 异步通信：
6. 数据同步：不同环境，比如国内和国外环境，机房隶属于不同地区，为了保证数据一致性，需要进行数据同步，通常通过消息中间件进行同步

消息队列是 分布式系统架构 的重要组件

应用解耦说明

解耦前：

1. 调用方和被调用者紧密耦合
2. 代码放到一起有很大的风险，一旦改动出错会导致整个功能不可用
3. 代码维护困难，并且不能进行并行开发，开发效率低

利用消息队列进行解耦：

1. 生产者和消费者不关心对方是干嘛的，只需要确认消息
2. 生产者和消费者不必同时在线

解耦后：

1. 上游服务和下游服务的代码不相关，可以并行开发
2. 上/下游定完协议(json/二进制/proto)，各自开发，各自测试
3. 完成后联调通过即可上线

例如支付系统：

1. 支付系统确认支付后，把支付结果生产到消息中间件
2. 订单系统消费MQ里的消息，去修改订单支付状态

两个系统通过消息中间件解耦

异步处理

生产消费的整个过程都是异步的

1. 消息生产者发送一个消息后无需等待响应
2. 生产者把消息发送到虚拟通道：queue/topic
3. 消费者订阅或者监听该通道
    - pub/sub 发布-订阅
    - 监听
4. 可以有多个消费者
5. 消费者不需要同步响应

传输模式

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，**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

特点：

1. 每个消息只用一个消费者；
2. 发送者和接受者没有时间依赖；
3. 接受者确认消息接受和处理成功。

发布订阅模型

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。发布者和订阅者彼此不知道对方，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**。

特点：

1. 每个消息可以有多个订阅者；
2. 客户端只有订阅后才能接收到消息；
3. 持久订阅和非持久订阅。

> 注意：
>
> 1. 发布者和订阅者有时间依赖：接受者和发布者只有建立订阅关系才能收到消息；
> 2. 持久订阅：订阅关系建立后，消息就不会消失，不管订阅者是否都在线；
> 3. 非持久订阅：订阅者为了接受消息，必须一直在线。 当只有一个订阅者时约等于点对点模式

应用场景

应用场景包括： **应用程序松耦合**、**异步处理模式**、**发布与订阅**、**最终一致性**、**错峰流控** 和 **日志缓冲**

如果需要 **强一致性**，关注业务逻辑的处理结果，则使用 `RPC` 显得更为合适

异步处理

非核心 流程 **异步化**，减少系统 **响应时间**，提高 **吞吐量**。例如：**短信通知**、**终端状态推送**、`App` **推送**、**用户注册** 等

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。

应用案例

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。

系统解耦

- 系统之间不是 **强耦合的**，**消息接受者** 可以随意增加，而不需要修改 **消息发送者的代码**。**消息发送者** 的成功不依赖 **消息接受者**（比如：有些银行接口不稳定，但调用方并不需要依赖这些接口）。
- **不强依赖** 于非本系统的核心流程，对于 **非核心流程**，可以放到消息队列中让 **消息消费者** 去按需消费，而 **不影响核心主流程**。

最终一致性

最终一致性 不是 **消息队列** 的必备特性，但确实可以依靠 **消息队列** 来做 **最终一致性** 的事情。

- **先写消息再操作**，确保操作完成后再修改消息状态。**定时任务补偿机制** 实现消息 **可靠发送接收**、业务操作的可靠执行，要注意 **消息重复** 与 **幂等设计**。
- 所有不保证 `100%` **不丢消息** 的消息队列，理论上无法实现 **最终一致性**。

> 像 `Kafka` 一类的设计，在设计层面上就有 **丢消息** 的可能（比如 **定时刷盘**，如果掉电就会丢消息）。哪怕只丢千分之一的消息，业务也必须用其他的手段来保证结果正确。

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。

流量削峰和流控

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

1. 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**；
2. 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**。

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。

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。

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

消费者 请求 **消息队列** 接受消息，**消息生产者** 从 **消息队列** 中拉该消息