**MQ消息可达性&&幂等性&&延时性架构设计**

1. **MQ的作用**
2. ***作用***

消息总线(Message Queue)，简称MQ，是一种跨进程的通信机制，用于上下游传递消息。



在互联网架构中，MQ是一种非常常见的上下游“逻辑解耦+物理解耦”的消息通信服务。使用了MQ之后，消息发送上游只需要依赖MQ，逻辑上物理上都不用依赖其他服务。

1. ***MQ不适用***

调用和被调用的关系，无法被MQ取代。



他的不足在于：

* 系统更复杂，多了一个MQ组件
* 消息传递路径更长，延时会增长
* 消息的可靠性和重复性无法同时保证，互为矛盾
* 上游无法知道下游的执行结果

比如：用户登录场景，登录页面调用passport服务，passport服务的执行结果直接影响登录结果，此处的“登录页面”与“passport服务”就必须使用调用关系，而不能使用MQ通信。

调用方实时依赖业务执行结果的场景，只能用调用而不是MQ。

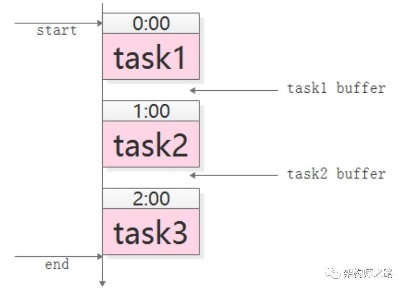
1. ***何时用MQ***

* 场景一：数据驱动的业务依赖

业务依赖就是业务之间有一定的依赖关系，比如：

* task3要以task2的输出作为输入
* task2要以task1的输出作为输入

这样，三个任务之间就有了依赖关系，必须task1先执行，然后是task2，task3



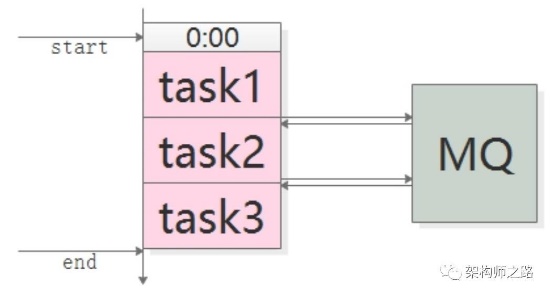
对于这类需求，常见的实现方式是使用cron人工排执行时间表

* task1，0:00执行，经验执行时间50min
* task2，1:00执行，为task预留10min buffer，经验执行时间50min
* task3，2:00执行，为task2预留10min buffer

这样做也有缺点

* 如果一个任务执行时间超过了预留buffer的时间，将会得到错误的结果，因为后置任务不清楚前置任务是否执行成功，此时要手动重跑任务，还有可能要调整排班表。
* 总任务的执行时间很长，总是要预留很多buffer，如果前置任务提前完成，后置任务不会提前开始
* 如果一个任务被多个任务依赖，这个任务将会称为关键路径，排班表很难体现依赖关系，容易出错
* 如果有一个任务的执行时间要调整，将会有多个任务的执行时间要调整

优化方案，采用MQ解耦：



* task1准时开始，结束后发一个“task1 done”的消息
* task2订阅“task1 done”的消息，收到消息后第一时间启动执行，结束后发一个“task2 done”的消息
* task3订阅“task2 done”的消息，收到消息后第一时间启动执行，结束后发一个“task3 done”的消息

这样优化的优点在于

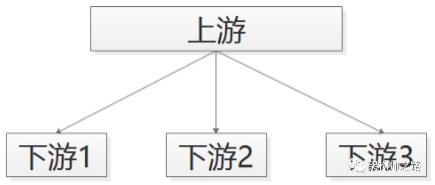
* 不需要预留buffer，上游任务执行完，下游任务总会在第一时间被执行
* 依赖多个任务，被多个任务依赖都很好处理，只需要订阅相关消息即可
* 有任务执行时间变化，下游任务都不需要调整执行时间

！注意：MQ只用来传递上游任务执行完成的消息，并不用于传递真正的输入输出数据

* 场景二：上游不关心执行结果

举个栗子，58同城的很多下游需要关注“用户发布帖子”这个事件，比如招聘用户发布帖子后，招聘业务要奖励58豆，房产用户发布帖子后，房产业务要送2个置顶，二手用户发布帖子后，二手业务要修改用户统计数据。

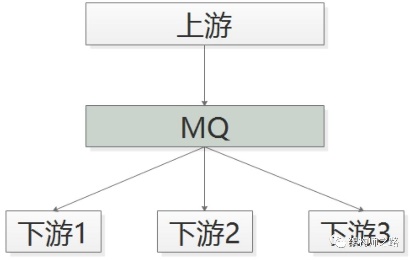
对于这类需求，常见的实现方式是使用调用关系：帖子发布服务执行完成之后，调用下游招聘业务、房产业务、二手业务，来完成消息的通知，但事实上，这个通知是否正常正确的执行，帖子发布服务根本不关注。



这种方法的坏处是

* 帖子发布流程的执行时间增加了
* 下游服务宕机，可能导致帖子发布服务受影响，上下游逻辑+物理依赖严重
* 每当增加一个需要知道“帖子发布成功”信息的下游，修改代码的是帖子发布服务，这一点是最恶心的，属于架构设计中典型的依赖倒转

优化方案，MQ解耦



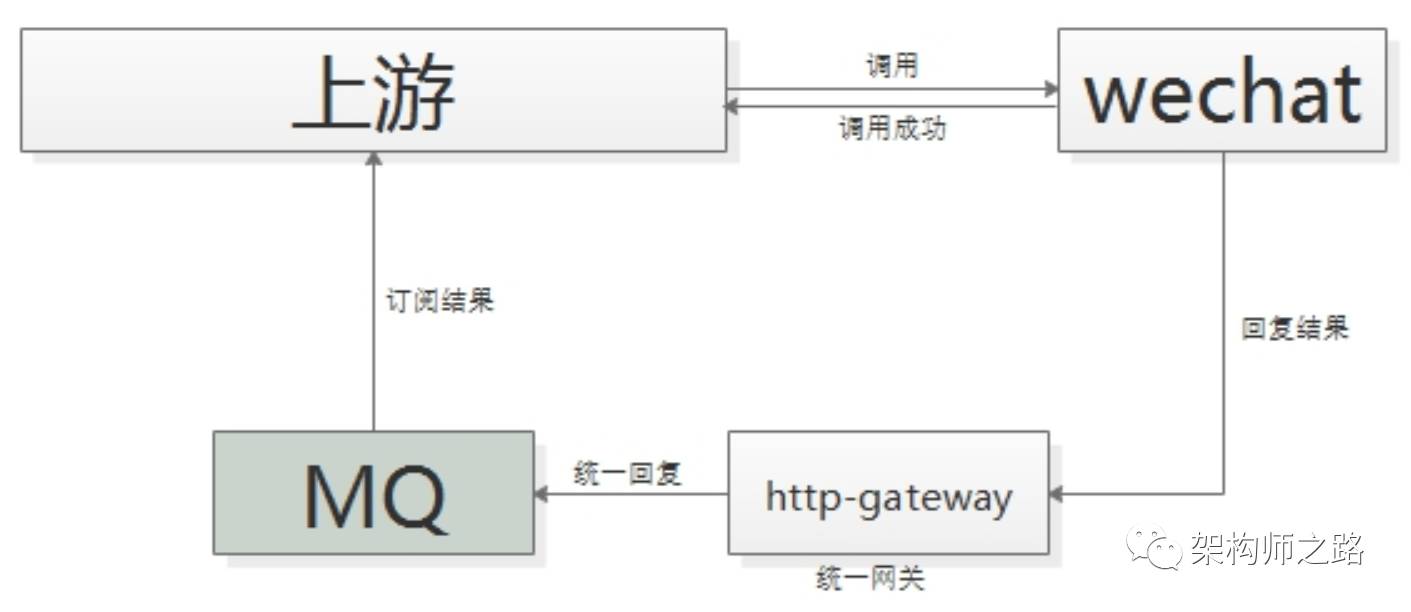
* 帖子发布成功后，向MQ发布一个“发布成功”的消息
* 哪个下游需要关注“发布成功”的消息，就主动去MQ订阅就好了

优点在于

* 上游执行时间短
* 上下游逻辑+物理解耦，除了与MQ有物理连接，模块之间都不相互依赖
* 新增一个下游消息关注方，上游不需要修改任何代码
* 上游关注执行结果，且执行时间长

有时候上游需要关注执行结果，但执行结果时间很长（典型的是调用离线处理，或者跨公网调用），也经常使用回调网关+MQ来解耦。

比如，微信支付，跨公网调用微信的接口，执行时间会比较长，但调用方又非常关注执行结果，此时一般采用“回调网关+MQ”方案来解耦



* 调用方直接跨公网调用微信接口
* 微信返回调用成功，此时并不代表返回成功
* 微信执行完成后，回调统一网关
* 网关将返回结果通知MQ
* 请求方收到结果通知

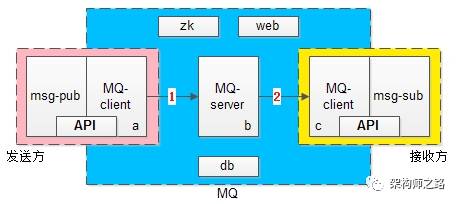
需要注意的是，不应该由回调网关来调用上游来通知结果，如果是这样的话，每次新增调用方，回调网关都需要修改代码，仍然会反向依赖，使用回调网关+MQ的方案，新增任何对微信支付的调用，都不需要修改代码啦。

1. **MQ，如何做到消息必达**
2. ***架构方向***

MQ要想做到消息必达，架构上必须有两个核心设计

* 消息落地
* 消息超时，重传，确认

1. ***核心架构***



MQ的核心架构图，基本可以分为三大块：

* 发送方 -> 左侧粉色部分
* MQ核心集群 -> 中间蓝色部分
* 接收方 -> 右侧黄色部分

粉色发送方又由两部分构成：业务调用方与MQ-client-sender，其中后者向前者提供了两个核心API：SendMsg(bytes[] msg)和SendCallback()

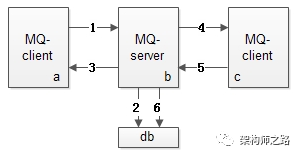
蓝色MQ核心集群又分为四个部分：MQ-server，zk，db，管理后台web

黄色接收方也由两部分构成：业务接收方与MQ-client-receiver，其中后者向前者提供了两个核心API：RecvCallback(bytes[] msg)和SendAck()

MQ是一个系统间解耦的利器，它能够很好的解除发布订阅者之间的耦合，它将上下游的消息投递解耦成两个部分，如上述架构图中的1箭头和2箭头：

* 发送方将消息投递给MQ，上半场
* MQ将消息投递给接收方，下半场

1. ***MQ消息可靠投递的核心流程***



MQ既然将消息投递拆成了上下半场，为了保证消息的可靠投递，上下半场都必须尽量保证消息必达。

MQ消息投递上半场，MQ-client-sender到MQ-server流程见上图1-3：

* MQ-client将消息发送给MQ-server（此时业务方调用的是API：SendMsg）
* MQ-server将消息落地，落地后即为发送成功
* MQ-server将应答发送给MQ-client（此时回调业务方是API：SendCallback）

MQ消息投递下半场，MQ-server到MQ-client-receiver流程见上图4-6：

* MQ-server将消息发送给MQ-client（此时回调业务方是API：RecvCallback）
* MQ-client回复应答给MQ-server（此时业务方主动调用API：SendAck）
* MQ-server收到ack，将之前已经落地的消息删除，完成消息的可靠投递

MQ消息投递的上下半场，都可以出现消息丢失，为了降低消息丢失的概率，MQ需要进行超时和重传。

* 上半场的消息超时与重传

MQ上半场的1或者2或者3如果丢失或者超时，MQ-client-sender内的timer会重发消息，直到期望收到3，如果重传N次后还未收到，则SendCallback回调发送失败，需要注意的是，这个过程中MQ-server可能会收到同一条消息的多次重发。

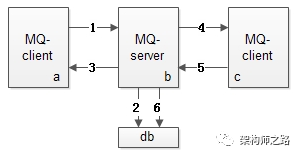
* 下半场的小喜超市与重传

MQ下半场的4或者5或者6如果丢失或者超时，MQ-server内的timer会重发消息，直到收到5并且成功执行6，这个过程可能会重发很多次消息，一般采用指数退避的策略，先隔x秒重发，2x秒重发，4x秒重发，以此类推，需要注意的是，这个过程中MQ-client-receiver也可能会收到同一条消息的多次重发。

MQ-client与MQ-server的消息又如何去重？

1. **MQ，如何做到幂等性**
2. ***问题来源***

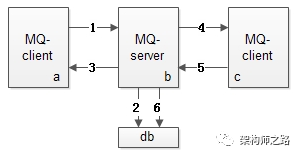
再次回顾消息总线核心架构，它由发送端、服务端、固化存储、接收端四大部分组成。



为保证消息的可达性，超时、重传、确认机制可能导致消息总线、或者业务方收到重复的消息，从而对业务产生影响。比如：购买会员卡，上游支付系统负责给用户扣款，下游系统负责给用户发卡，通过MQ异步通知。不管是上半场的ACK丢失，导致MQ收到重复的消息，还是下半场ACK丢失，导致购卡系统收到重复的购卡通知，都可能出现，上游扣了一次钱，下游发了多张卡。

为了解决这个问题，引入幂等性的方案。

1. ***上半场幂等性设计***



* 1，发送端MQ-client将消息发给服务端MQ-server
* 2，服务端MQ-server将消息落地
* 3，服务端MQ-server回ACK给发送端MQ-client

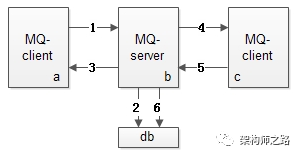
如果3丢失，发送端MQ-client超时后会重发消息，可能导致服务端MQ-server收到重复消息。

此时重发是MQ-client发起的，消息的处理是MQ-server，为了避免步骤2落地重复的消息，对每条消息，MQ系统内部必须生成一个inner-msg-id，作为去重和幂等的依据，这个内部消息ID的特性是：

* 全局唯一
* MQ生成，具备业务无关性，对消息发送方和消息接收方屏蔽

有了这个inner-msg-id，就能保证上半场重发，也只有1条消息落到MQ-server的DB中，实现上半场幂等。

1. ***下半场幂等性设计***



* 4，服务端MQ-server将消息发给接收端MQ-client
* 5，接收端MQ-client回ACK给服务端
* 6，服务端MQ-server将落地消息删除

需要强调的是，接收端MQ-client回ACK给服务端MQ-server，是消息消费业务方的主动调用行为，不能由MQ-client自动发起，因为MQ系统不知道消费方什么时候真正消费成功。

如果5丢失，服务端MQ-server超时后会重发消息，可能导致MQ-client收到重复的消息。

此时重发是MQ-server发起的，消息的处理是消息消费业务方，消息重发势必导致业务方重复消费（上例中的一次付款，重复发卡），为了保证业务幂等性，业务消息体中，必须有一个biz-id，作为去重和幂等的依据，这个业务ID的特性是：

* 对于同一个业务场景，全局唯一
* 由业务消息发送方生成，业务相关，对MQ透明
* 由业务消息消费方负责判重，以保证幂等

最常见的业务ID有：支付ID，订单ID，帖子ID等。

具体到支付购卡场景，发送方必须将支付ID放到消息体中，消费方必须对同一个支付ID进行判重，保证购卡的幂等。

有了这个业务ID，才能够保证下半场消息消费业务方即使收到重复消息，也只有1条消息被消费，保证了幂等。

1. **MQ，如何做到消息延时**
2. ***问题来源和初步解决方案***

很多时候，业务有“在一段时间之后，完成一个工作任务”的需求。例如：滴滴打车订单完成后，如果用户一直不评价，48小时后会将自动评价为5星。一般来说怎么实现这类“48小时后自动评价为5星”需求呢？

常见方案：启动一个cron定时任务，每小时跑一次，将完成时间超过48小时的订单取出，置为5星，并把评价状态置为已评价。

假设订单表的结构为：t\_order(oid, finish\_time, stars, status, …)，更具体的，定时任务每隔一个小时会这么做一次：select oid from t\_order where finish\_time > 48hours and status=0;update t\_order set stars=5 and status=1 where oid in[…];

如果数据量很大，需要分页查询，分页update，这将会是一个for循环。方案的不足

* 轮询效率比较低
* 每次扫库，已经被执行过记录，仍然会被扫描（只是不会出现在结果集中），有重复计算的嫌疑
* 时效性不够好，如果每小时轮询一次，最差的情况下，时间误差会达到1小时
* 如果通过增加cron轮询频率来减少第三条中的时间误差，第一条中的轮询低效和第二条中的重复计算的问题会进一步凸显

解决方案的不足，需要利用“延时消息”，对于每个任务只触发一次，保证效率的同时保证实时性。

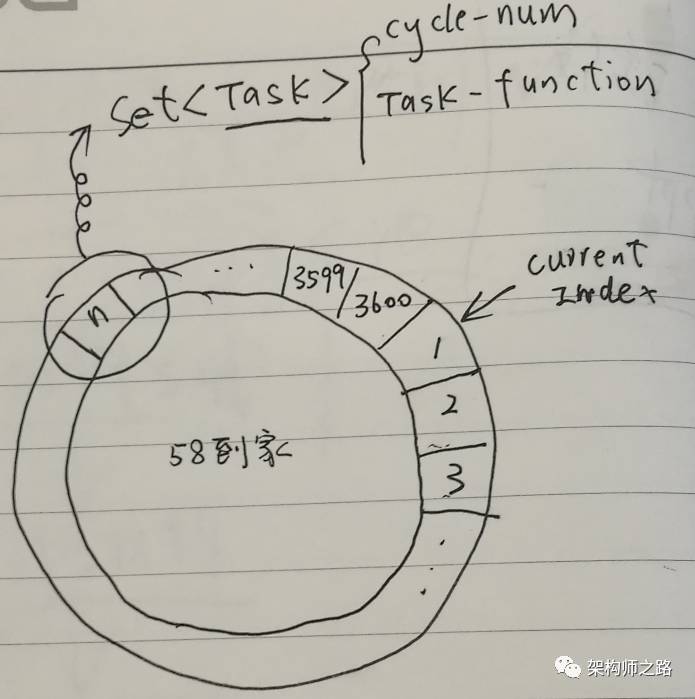
1. ***高效延时消息设计与实现***

高效延时消息，包含两个重要的数据结构：

* 环形队列，例如可以创建一个包含3600个slot的环形队列（本质是个数组）
* 任务集合，环上每一个slot是一个Set<Task>

同时，启动一个timer，这个timer每隔1s，在上述环形队列中移动一格，有一个Current Index指针来标识正在检测的slot。

task必须包含两个重要的属性：



* Cycle-Num：当Current Index第几圈扫描到这个Slot时，执行任务
* Task-Function：需要执行的任务指针

假设当前Current Index指向第一格，当有延时消息到达之后，例如希望3610秒之后，触发一个延时消息任务，只需

* 计算这个Task应该放在哪一个slot，现在指向1，3610秒之后，应该是第11格，所以这个Task应该放在第11个slot的Set<Task>中
* 计算这个Task的Cycle-Num，由于环形队列是3600格（每秒移动一格，正好1小时），这个任务是3610秒后执行，所以应该绕3610/3600=1圈之后再执行，于是Cycle-Num=1

Current Index不停的移动，每秒移动到一个新slot，这个slot中对应的Set<Task>，每个Task看Cycle-Num是不是0：

* 如果不是0，说明还需要多移动几圈，将Cycle-Num减1
* 如果是0，说明马上要执行这个Task了，取出Task-Funciton执行（可以用单独的线程来执行Task），并把这个Task从Set<Task>中删除

使用了“延时消息”方案之后，“订单48小时后关闭评价”的需求，只需将在订单关闭时，触发一个48小时之后的延时消息即可：

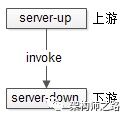
* 无需再轮询全部订单，效率高
* 一个订单，任务只执行一次
* 时效性好，精确到秒（控制timer移动频率可以控制精度）

1. **MQ，如何做到削峰填谷**
2. ***问题来源***

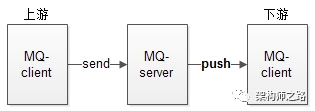
MQ除了上面提到的典型应用场景外，还有一个典型应用场景是缓冲流量，削峰填谷，而MQ做到这个需要什么实现细节引出了这个问题。

1. ***服务、站点之间通讯方式***

* 直接调用，通过RPC框架，上游直接调用下游



* MQ推送，上游将消息发给MQ，MQ将消息推送给下游



1. ***流量冲击的来源***

在2的两种通讯方式中，都有一个缺点，下游消息接收方无法控制到达自己的流量，如果调用方不限速，很有可能把下游压垮。

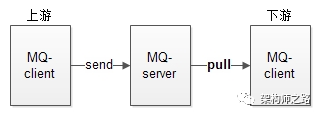
比如秒杀业务，上游发起下单操作，下游完成秒杀业务逻辑（库存检查，库存冻结，余额检查，余额冻结，订单生成，余额扣减，库存扣减，生成流水，余额解冻，库存解冻）。上游下单业务简单，每秒发起了10000个请求，下游秒杀业务复杂，每秒只能处理2000个请求，很有可能上游不限速的下单，导致下游系统被压垮，引发雪崩。

为了避免崩掉，常见优化方案如下

* 业务上游队列缓冲，限速发送
* 业务下游队列缓冲，限速执行

1. ***调整MQ使之能够缓冲流量***

由MQ-server推模式，升级为MQ-client拉模式。MQ-client根据自己的处理能力，每隔一定时间，或者每次拉取若干条消息，实施流控，达到保护自身的效果。并且这是MQ提供的通用功能，无需上下游修改代码。



为了避免消息在MQ中堆积，下游MQ-client拉取消息，需要下游消息接收方进行优化，方能够提升整体吞吐量，例如：批量写。