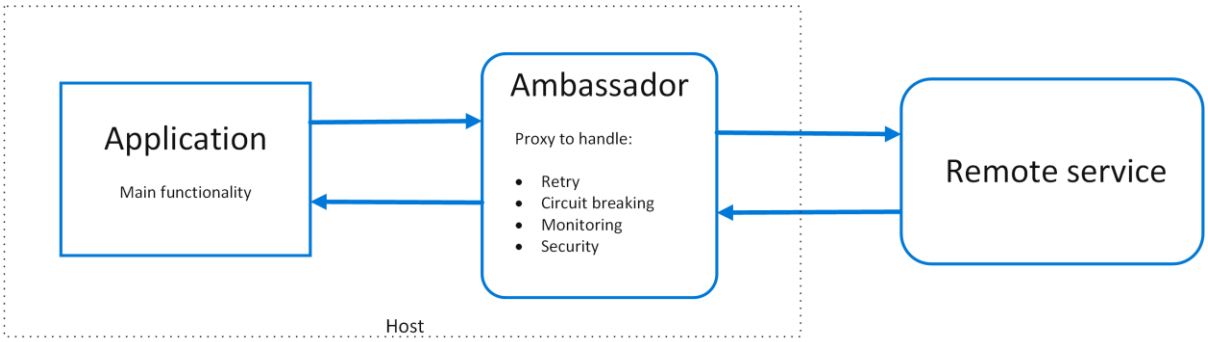
朱晔的互联网架构实践心得 S1E7：三十种架构设计模式（上）

【下载本文 PDF 进行阅读】

设计模式是前人通过大量的实践总结出来的一些经验总结和最佳实践。在经过多年的软件开发实践之后，回过头来去看 23 种设计模式你会发现很多平时写代码的套路和 OO 的套路和设计模式里总结的类似，这也说明了你悟到的东西和别人悟到的一样，经过大量实践总能趋向性得出一些最佳实践的结论。架构设计也是一样，这里结合自己的理解分析一下微软给出的云架构的一些模式。话说微软干这方面的事情真的很厉害，之前翻译过的《微软应用架构指南》写的也很不错。有了模式的好处是，技术人员和技术人员之间的对话可以毫不费力的通过几个模式关键词进行交流，就像现在大家沟通提到职责链模式，如果双方都理解这个模式的意义那么这五个字替代的可能就是半小时的解释。废话不多说，接下去来看一下这些其实已经很熟悉亲切的模式。

管理和监控

1、大使模式：创建代表消费者服务或应用程序发送网络请求的帮助服务



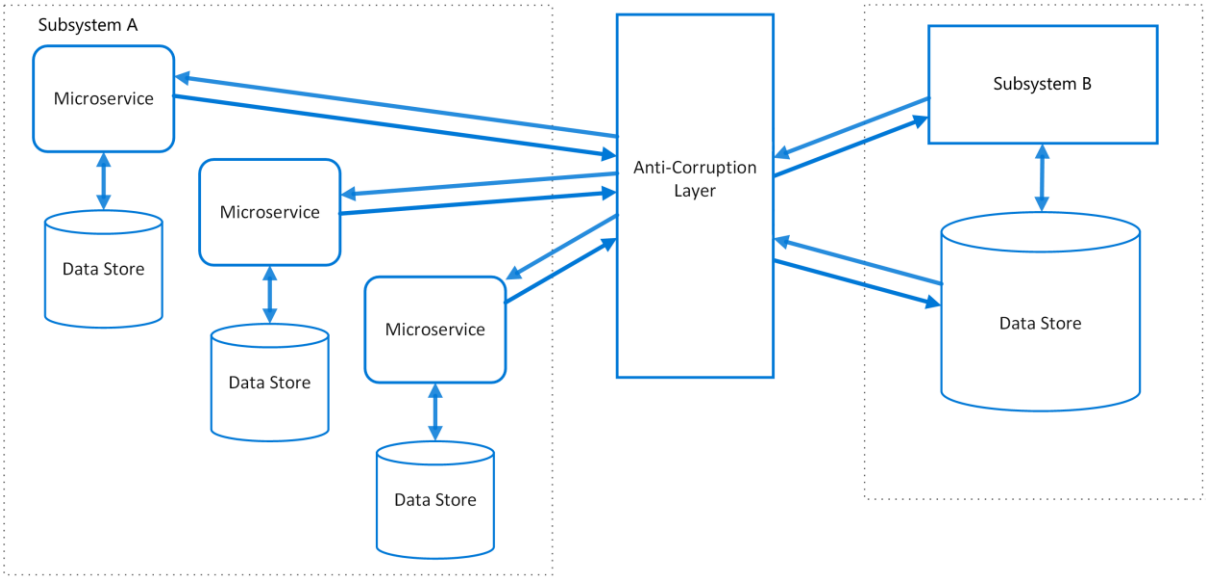
进程外的代理服务（之前介绍中间件的时候也提到了，很多框架层面的事情可以以软件框架的形式寄宿在进程内，也可以以独立的代理形式做一个网络中间件）。这里的大使模式意思就是这么一个网络代理进程，用于和远端的服务进行通讯，完成下面的工作：

- 服务路由
- 服务熔断
- 服务跟踪
- 服务监控

- 服务授权
- 数据加密
- 日志记录

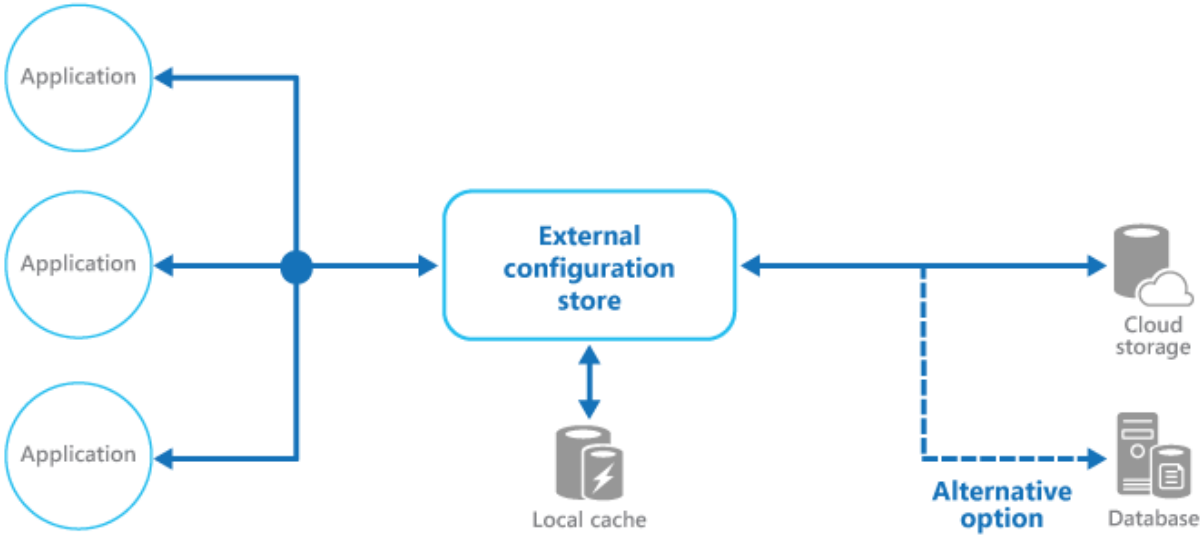
由于是独立进程的网络服务，所以这个模式适合于我们有多语言多框架都需要干同样的事情，那么我们的框架中客户端部分的很多工作可以移出来放到大使服务中去。当然了，多一层网络调用多一层开销，大使服务的部署也要考虑到性能不一定可以集中部署，这些都是要考虑的问题。

2、反腐模式：在现代应用程序和遗留系统之间实现装饰或适配器层



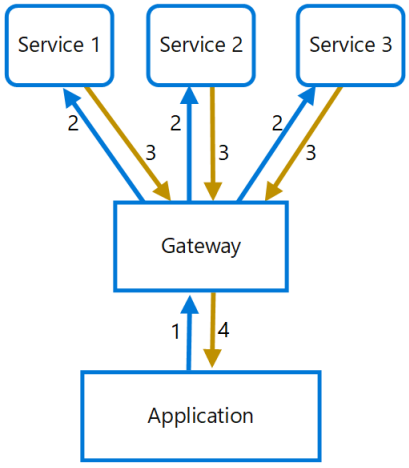
使用一层防腐层来作为新老系统通讯的中间人。这样新系统可以完全使用新的通讯方式和架构方式，老的系统又不用进行特别改造可以暂时保留，等老系统不用之后可以废弃这个反腐层。这种模式适合新老系统迁移的过渡方案，不属于永久使用的架构设计模式。

3、外部配置存储：将应用程序部署包中的配置信息移动到中心化的位置



这个模式说的就是可以有一个外部的配置服务来保存配置信息，在之前第五篇文章介绍中间件的时候我详细说明过配置服务的功能。不管是处于管理运维的角度还是方便安全的角度，具有配置共享配置外存特点的独立配置服务对于大型的网站来说必不可少。实现的话有很多开源项目提供了配置服务，见之前我的文章。

4、网关聚合模式：使用网关将多个单独的请求聚合到一个请求中



应用程序如果需要和多个服务交互的话，在中间构建起一个聚合网关层，网关并发发出多个请求给后面的服务，然后汇总数据给到应用程序。这种模式有几个好处：

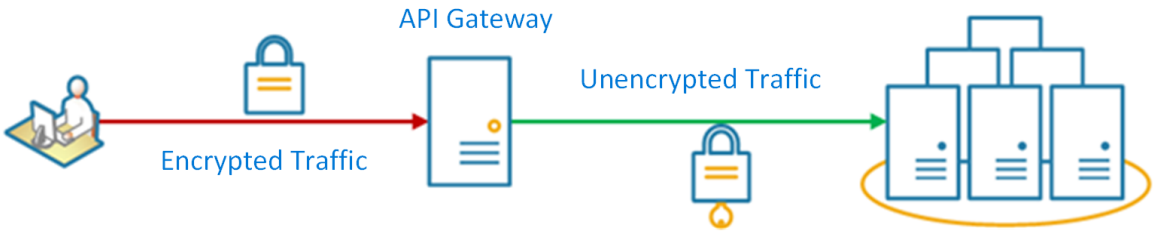
- 允许并发调用多个服务提高性能，允许只返回部分数据
- 网关里可以做一些弹性设计方案（熔断、重试、限流）
- 网关里可以做一些缓存方案
- 对于外网通讯的时候，可以让网关作为一个网络中间层

当然，使用这种模式需要考虑到网关的负载、高可用、高性能（异步 IO）等等。

其实这种模式不仅仅用于纯后端服务之间的通讯，很多面向前端的 API 请求都会做一个聚合层，这样前端可以只发一个请求的情况下任意向后端一次性索取多个 API 的返回，减少网络请求次数提高性能。

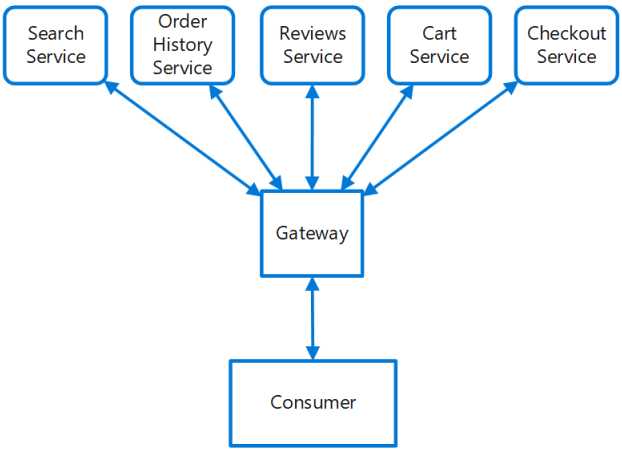
实现上最简单的方式可以使用 OpenResty 或 Nginx 实现。

5、网关卸压模式：把共享或特定的服务功能放到网关代理



名字有点难以理解，其实这种模式我们可能一直在用。就是用一个代理网关层做一些和业务无关的又麻烦的点，比如 SSL，实现上用 Nginx 实现就很简单。我们经常会对外启用 HTTPS 服务，然后对内服务实际提供的是 HTTP 接口，通过网关做一下协议转换。

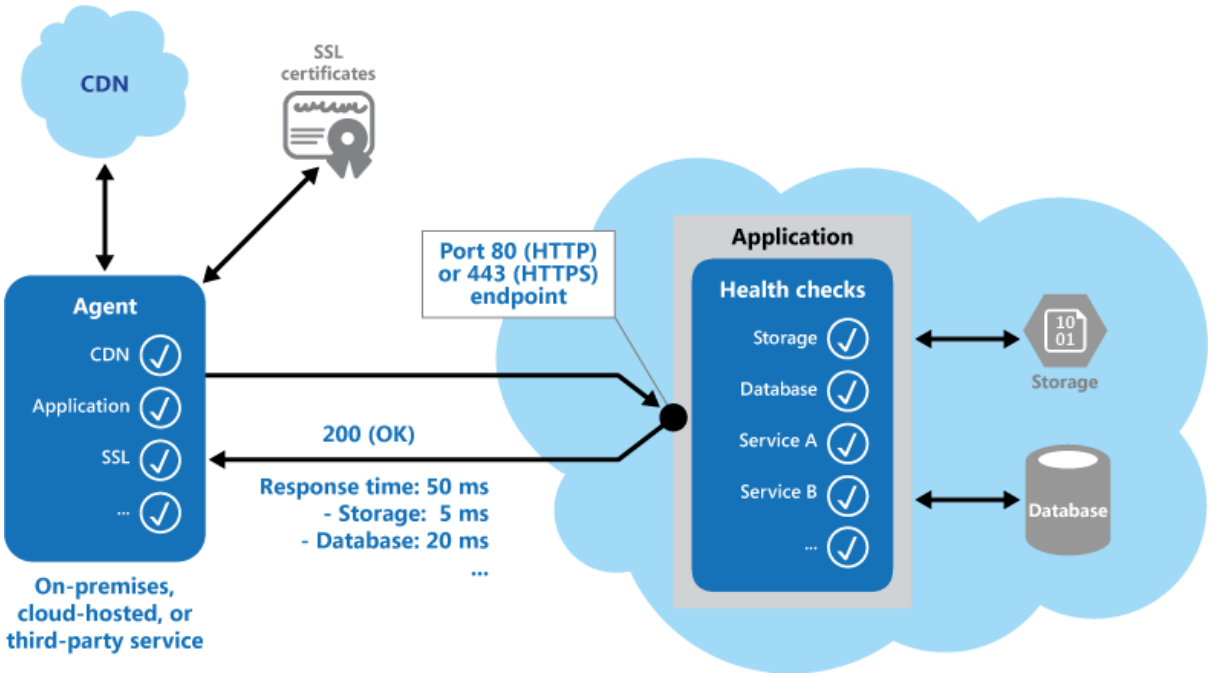
6、网关路由模式：使用单个端点将请求路由到多个服务



这也是很常见的作法，我们对外的接口可能是/cart、/order、/search 这样的 API，在其背后其实是不同的服务，通过网关层进行转发，不仅仅可以做后端服务的负载均衡和故障转移，在后端服务变更切换对外 API 路径（比如版本升级）的时候我们也可以进行灵活的路由，确保了

对外接口的一致性。可以使用 Nginx 来实现，相信大部分公司都是由 Nginx 这样的网关来对外的，不会把域名直接解析到底层服务上对外。

7、健康端点监控模式：在应用程序中执行功能检查，外部工具可以定期通过暴露的端点访问

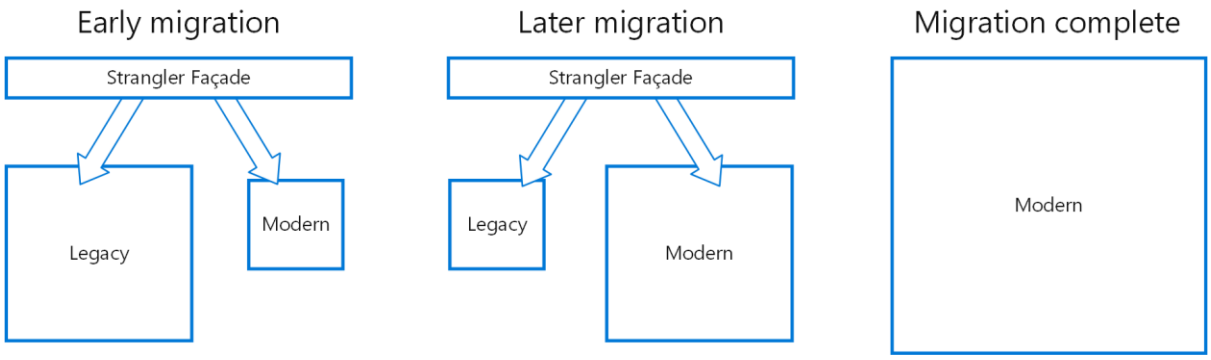


这个模式其实是挺重要的一点，有几个点需要注意：

- 需要暴露哪些信息？不仅仅是服务本身或框架本身是否启动成功，尽可能暴露出服务依赖的外部存储或系统是否可用，原因是网络通讯是复杂的，从外部看到某个服务可用不代表我们的网站就可以成功连接，如果底层的数据库都无法连接，即使这个网站本身启动成功，那么我们应该认为这个服务是不健康的。外部存储即使对于 A 节点是可以连通对于 B 节点不能连通也是有可能的，可能是因为网络问题或权限问题，还可能因为负载问题，有的时候对于长连接的请求 A 节点因为始终连着存储不会有问题，新的 B 节点要求连接的时候因为超出最大连接限制无法连接。如果有可能的话还暴露一些服务内部各种线程池、连接池和队列的信息吧（对象数，队列长度等），这些指标很关键，但是因为在程序内部所以外围很难感知到，有了一些关键指标的外露对于排查性能问题会方便很多。
- 不只是网站，服务也应该暴露出健康信息，一来我们可以在外部收集这些信息进行监控汇总，二来我们的负载均衡器或发布系统需要有一个方式来判断服务是否可用，不可用的时候进行重启或故障转移。
- 对外的服务注意 health 端口的授权，这里可能会有一些敏感信息，不宜让匿名用户看到。

实现上，我们应当把 health 端口作为插件形式集成到系统，配置一下即可启用，用不着每一个系统都自己开发一套。如果使用 SpringBoot 的话可以直接使用 Actuator 模块实现。

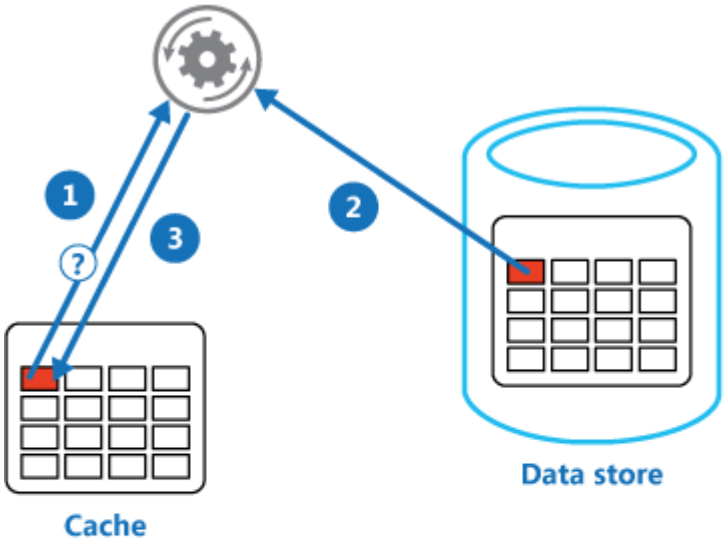
8、绞杀者模式：通过使用新的应用程序和服务逐渐替换特定功能部件来逐步迁移旧系统



名字挺吓人，这个模式说的是如何做迁移。通过建立一个门面来作为后端新老服务的路由，慢慢把服务替换为新服务，最后当所有的服务都是新服务后删除这个门面即可。这样对于消费者感知不到这个迁移的过程。在上一篇文章中我们提到的换引擎的方式其实说的是保留原有的门面，也是通过这个门面做底层引擎的替换。其实我觉得对于减少外围影响这种模式是完全可以理所当然想到的，真正难的过程还是之前说的数据迁移和底层服务实现的过程。

性能和可扩展性

9、缓存辅助模式：按需将数据从数据存储加载到缓存中



1: Determine whether the item is currently held in the cache.
2: If the item is not currently in the cache, read the item from the data store.
3: Store a copy of the item in the cache.

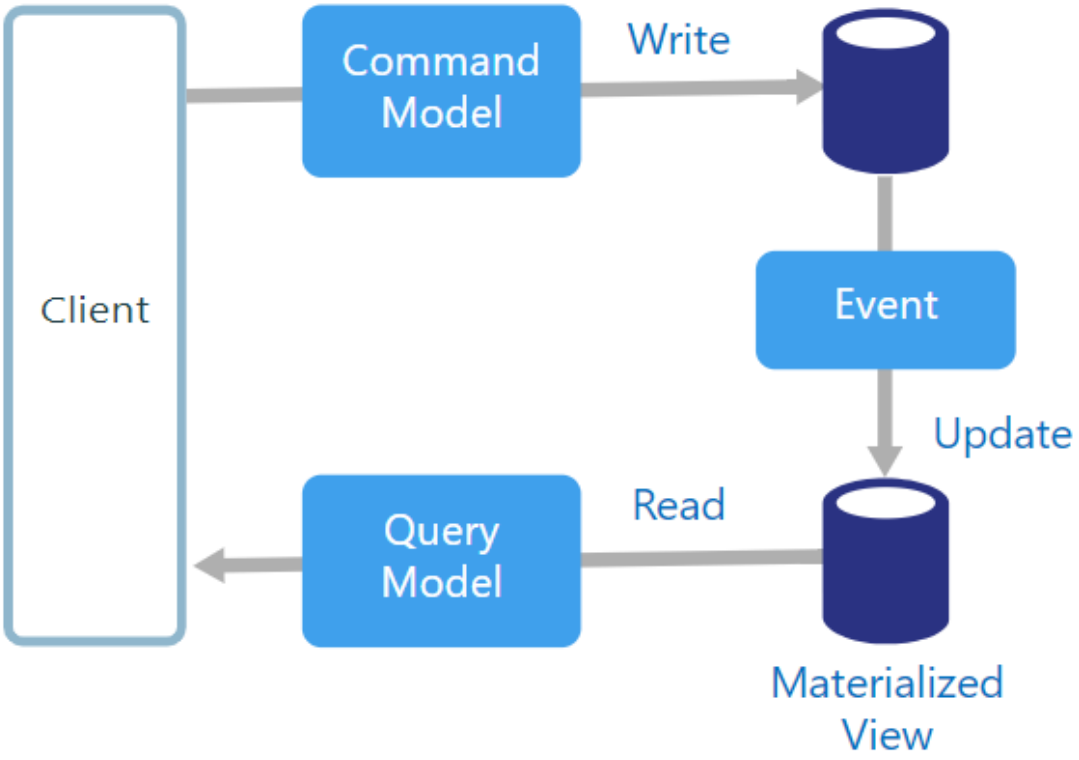
这个模式说的不是广义上的缓存使用，而是其中的一种使用方式。我们对于缓存的使用一般有这么几种方式：

- 查缓存，不存在查库，然后更新缓存
- 直接维护一大块“全量”数据，尽量和数据库同步

这个模式说的是后一种方式，对于数据变动不大，这种模式是性能最好的，几乎实现了 100% 的命中率，而且如果数据量不大可以容纳进进程的话不需要跨进程通讯。往细致一点去想，这里还有一层性能优化的点，因为我们在内存中维护了一套复杂的全量数据的数据结构，内存中对象的引用只是指针引用，内存中的数据搜索可以很快，对于数据量不大但是关系复杂的数据，这个搜索效率可以是数据库的几百倍。实现上一般会在应用程序启动的时候把数据完全加入内存，在后续通过一些策略进行数据更新：

- 定时更新同步数据，不同数据可以有不同的更新频率由后台线程来更新
- 数据具有不同的过期时间，过期后由请求触发主动更新或回调方式被动更新
- 数据修改后同步修改缓存和数据库中的数据

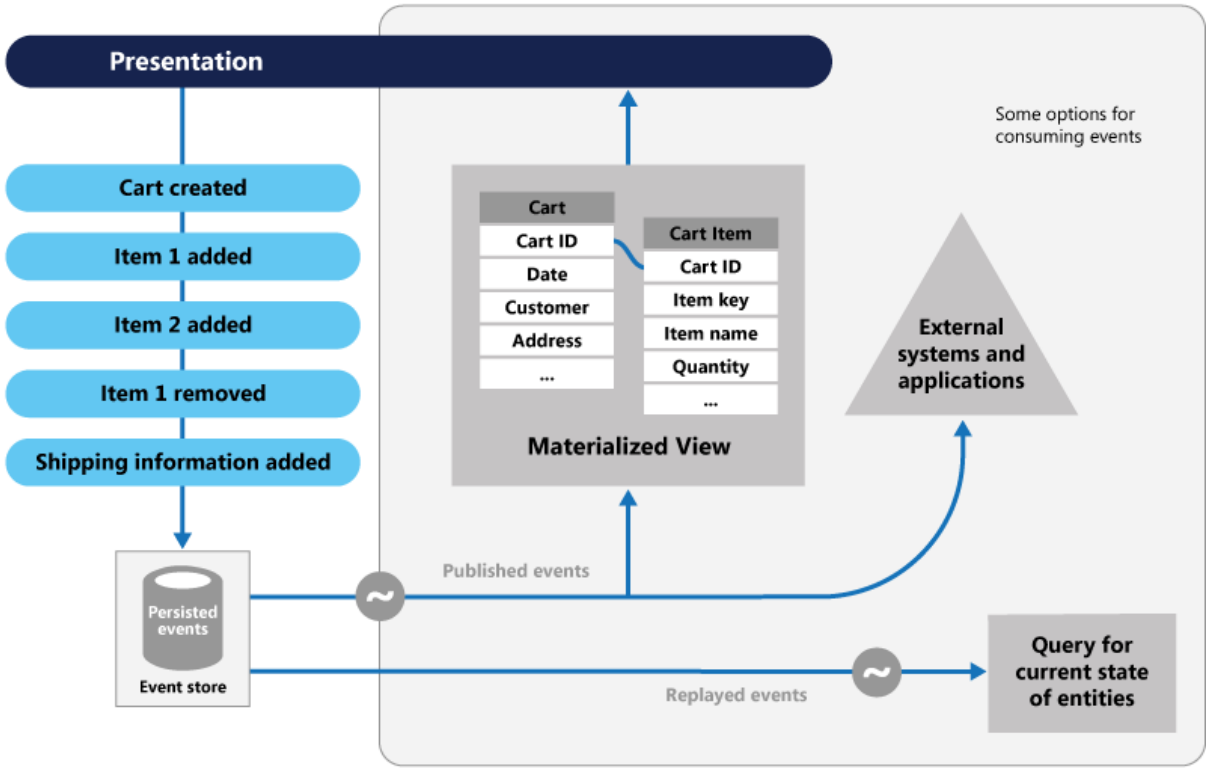
10、命令和查询责任分离模式：通过使用单独的接口来分离读取数据和更新数据的操作



英文缩写是 CQRS，看到这个关键字你可能会觉得有点熟悉了。CQRS 原来说的是我们可以有两套数据模型分别用于读和写。好处是，我们可以让读和写具有完全不同的数据结构，减少相互的干扰，减少权限控制的复杂度。这里说的不一定是指架构层面我们可以这么做，也指在程序内部，我们可以有两套命令模型来处理读写这两个事情，分别进行优化和定制。

现在一般的做法是类似于上图的做法，为读写配置两套独立的数据源，并且和事件溯源的方式结合起来做（见后面一节）。我们来说说读写两套模型在存储上分离这个事情，在《相辅相成的存储五件套》一文中我们的架构图其实就有这方面的意思。对于读写这两个事情，我们完全可以不用一套数据源，为读建立专门的物化视图，可以针对读进行优化，避免在读的时候做很多 Join 的工作，可以把性能做到极致（后面会有物化视图模式的介绍）。事件溯源+CQRS+物化视图三者一般会结合起来使用。

11、事件溯源模式：使用仅追加存储去记录描述对域中的数据采取的操作的完整系列事件



事件溯源（ES）是一种有趣的模式，说的是我们记录的不是数据的当前状态而是叠加的数据变化序列（是不是想到了区块链的数据记录方式）。传统的 CRUD 方式因为有更新这个操作，所以会产生性能并发方面的局限性，而且我们还需要配备额外的日志来做审计，否则就产生了信息丢失。而事件溯源模式记录的是事件而不是当前状态，所以有下面的特点：

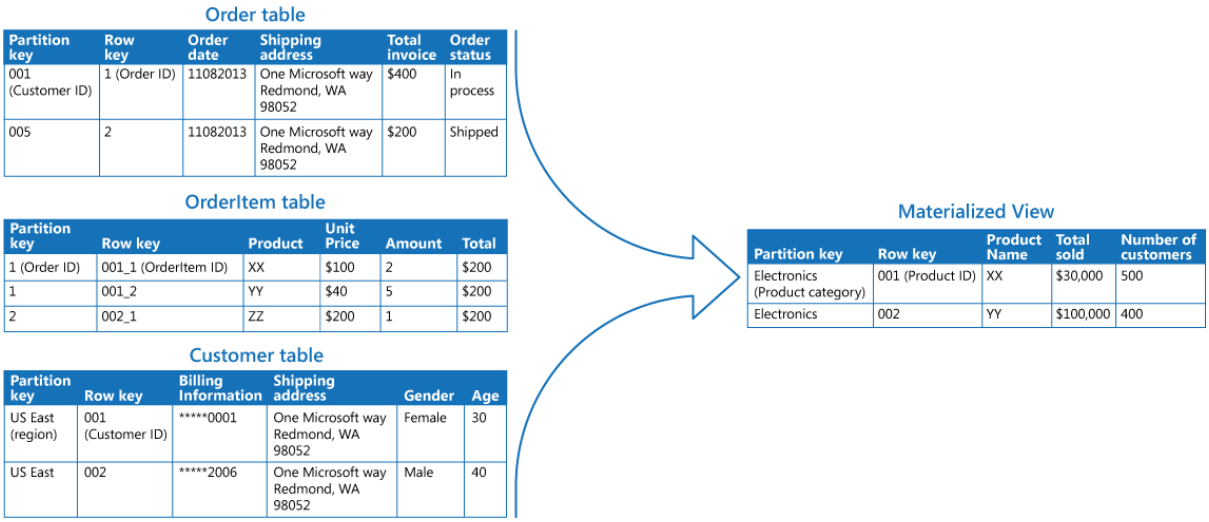
- 事件不可变，只是追加新的事件，没有冲突，性能高
- 以事件驱动做外部处理，耦合低
- 保留第一手原始信息，信息没有损耗

其实有一些业务场景下这种模式会比 CRUD 存储更适合：

- 业务更看重数据的意图和目的而不是当前的状态，注重审计、回滚、历史方面的功能
- 希望避免数据更新的冲突，希望数据的产生能有较高性能，又能接受数据状态的最终一致性
- 整个系统中本身就是以事件在驱动的（我们可以想一下在真实的世界中，物体和物体之间相互影响，通过事件来影响，每个物体观察到其它物体发出的事件来做出自己的反映，这是最自然的，而不是观察到别的物体属性的变化来调整自己的属性）

反过来说，业务逻辑很简单的系统，需要强一致性的系统，数据很少更新的系统不适合这种模式。不知你所了解到的采用 ES 模式的业务场景有哪些？大家一起交流一下。

12、物化视图模式：针对所需的查询操作，当数据没有理想地格式化时，在一个或多个数据存储中的数据上生成预填充视图



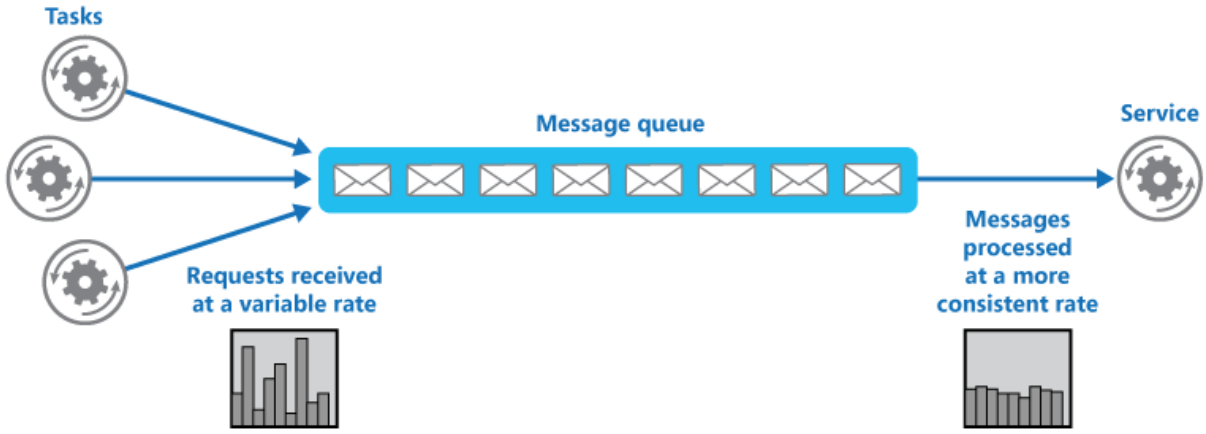
我们在使用数据存储的时候往往会更多考虑存储而不是读取。我们使用各种数据库范式来设计数据库，在读取数据的时候我们需要做大量的关联查询以输出符合需要的查询结果。这个时候性能往往会成为瓶颈，物化视图是一种空间换时间的做法。与其在查询的时候做关联，倒不如提前保存一份面向于查询和输出的数据格式。因此，物化视图适合下面的场景：

- 经过复杂的计算才能查询出数据
- 背后存储可能会有不稳定的情况
- 需要连接多个不同类型的存储才能查询到结果

但是因为需要考虑到物化视图计算保存的开销，所以也不太适合于数据变化太频繁的情况，因为数据加工需要时间，所以不适合需要数据强一致性的场景。

实现上一般是基于消息监听做额外维护一套物化视图的数据源和主流程解耦。惠普的 Vertica 是一款高性能的列式分析数据库，它的一个特性就是物化视图，通过事先提供 SQL 语句直接缓存面向于统计的查询结果，极大程度提高了性能，也是空间换时间的思想。

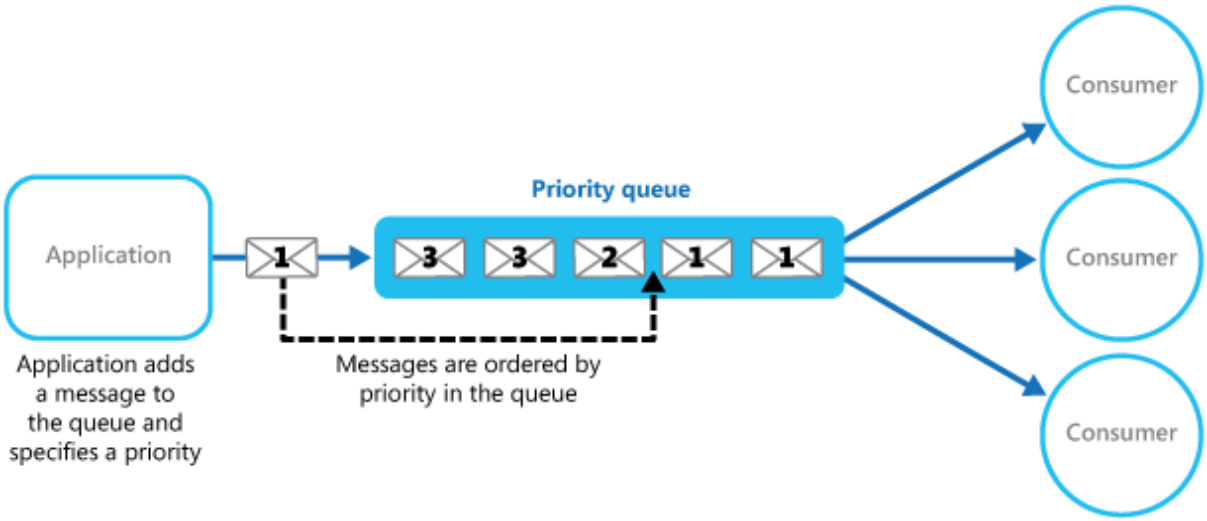
13、基于队列的负载均衡模式：使用一个队列作为任务和服务之间的缓冲区，平滑间歇性重负载

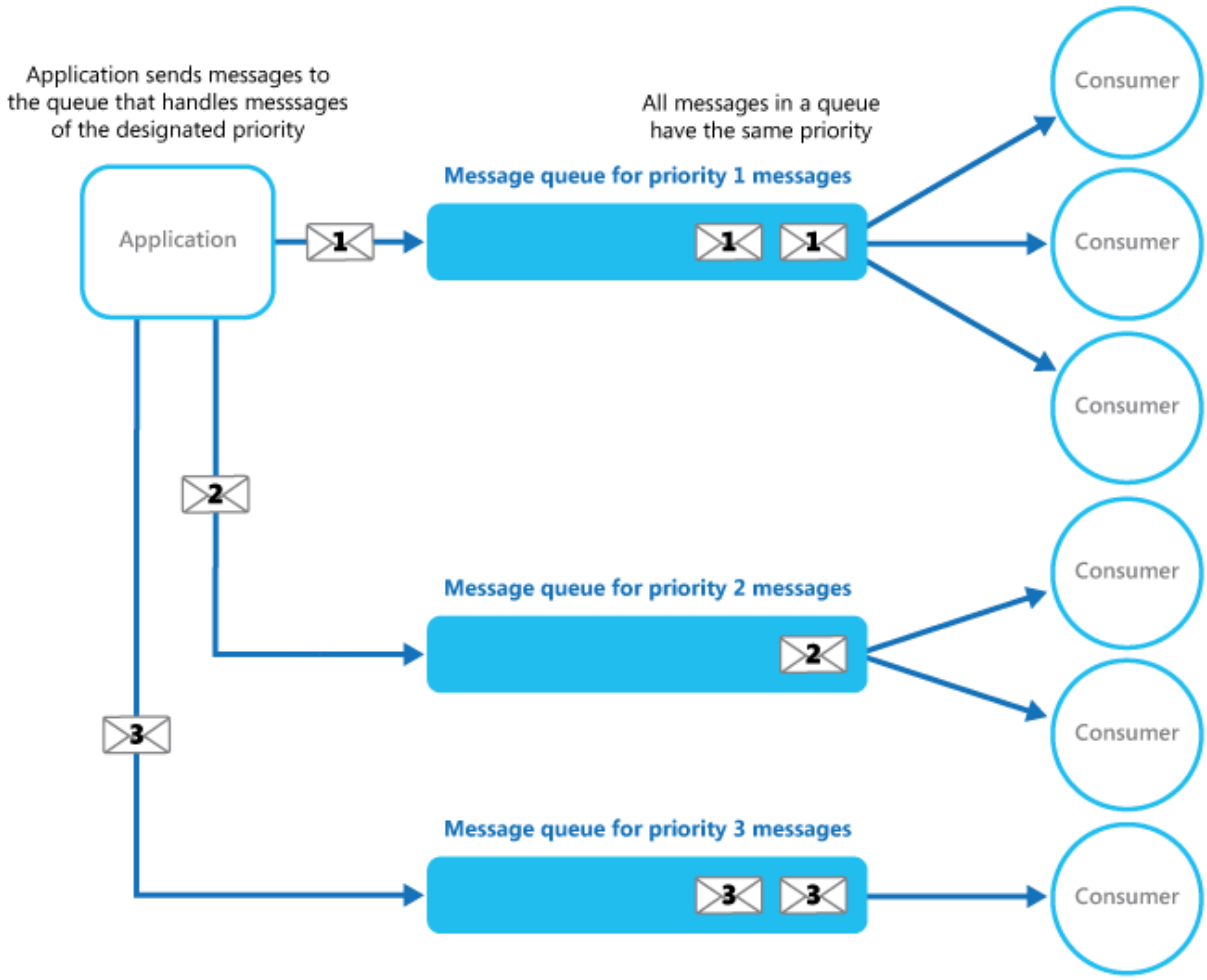


消息队列我们太熟悉了，之前我们也反复提高过好多次，甚至我说这是架构三马车之一。这个模式在这里强调的是削峰的优势。这里我还想提几点：

- 引入消息队列不会提高处理能力，而是会降低性能，只是我们把耦合解开了允许每一个部件单独有自己的弹性，对于不能负荷的部分在队列中进行缓冲，缓冲不是存储不意味无限制
- 队列看的是处理速度和入队速度的比例，一般而言，我们需要预先做评估确保处理 TPS 超过 2 倍的最高峰的入队 TPS，确保留出一半的富裕，这样在业务逻辑有修改的时候处理 TPS 哪怕下降了 30%，还能抗住压力

14、优先级队列模式：确定发送到服务的请求的优先级，使得具有较高优先级的请求更快地被接收和处理





区别于 FIFO 结构的队列，优先级队列允许消息标识处理优先级。这里实现上如上面两个图有两种方式：

- 消息优先级方式。在队列中进行实时位置重排，永远优先处理级别较高的消息。
- 不同的处理池方式。我们可以针对不同的处理级别配备专门的处理池来处理这些消息，高级别的消息具有更多的处理资源，更好的硬件来处理，这样势必会有较高的处理能力。

在方案选择和实现上要考虑消息优先级是否需要绝对按照优先级来处理，还是说相对优先处理即可，如果需要绝对优先那么除了消息位置重排还需要有抢占处理。还有，如果我们采用第二种多池的方式来处理的话可能会发生低级别的消息处理时间比高级别的消息更快的可能性（如果两者处理业务逻辑是完全不同的话）。

实现上的话 RabbitMQ 3.5 以上版本支持了消息优先级，实现的是第一种方式，在消息有缓冲的堆积的时候进行消息重排，消费端可以先看到先处理优先级高的消息，这种方式在消费速度大于产出速度的场景下是无法实现高级别消息优先处理的。

补充一点，对于队列中的消息，还有一种需要特别考虑的就是一直停留在队列的消息应当视为低优先级或是死信消息来处理，最好是有单独的消费者来处理，避免此类消息影响了整个队列的处理，见过很多个事故是由于队列中被废弃消息卡死导致彻底丧失处理能力的。

15、限流模式：控制应用程序，个人租户或整个服务的实例消耗的资源

在做压力测试的时候我们会发现，随着压力的上升系统的吞吐慢慢变大而且这个时候响应时间可以基本保持可控（1 秒内），当压力突破一个边界后，响应时间一下子会不可控，随之系统的吞吐就会下降，最后会彻底崩溃。任何系统对于压力的负荷是有边界的，超过这个边界之后系统的 SLA 肯定无法满足标准，导致大家都无法好好用这个服务。因为系统的扩展往往不是秒级可以做到的，所以这个时候最快的手段就是限流，只有限流了才能保护现在的系统不至于突破这个边界彻底崩溃。对于业务量超大的系统搞活动，对关键服务甚至入口层面做限流是必然的，别无它法，淘宝双 11 凌晨 0 点那一刻也能看到一定比例的下单被限流了。

常见的限流算法有这么几种：

- 计数器算法。最简单的算法，资源使用加一，释放减一，达到一定的计数拒绝服务。
- 令牌桶算法。按照固定速率往桶里加令牌，桶里最多存放 n 个令牌，填满丢弃。处理的时候需要获取令牌，获取不到则拒绝请求。
- 漏桶算法。一个固定容量的漏洞，按照一定的速度流出水滴（任务）。可以以任意速度流入水滴（任务），满了则溢出丢弃。

令牌桶算法限制的是平均流入速度，允许一定程度的突发请求，漏桶算法限制的是常量的流出速率用于平滑流入的速度。实现上，常用的一些开源类库都会有相关的实现，比如 google 的 Guava 提供的 RateLimiter 就是令牌桶算法。

限流模式有下面的一些注意事项：

- 限流需要快速执行，任何一个超出流量控制的请求不允许放行，否则没有意义。
- 限流需要提前执行，最好在系统能力达到 80%的时候进行限流，越晚限流风险越大。
- 可以返回特定的限流控制错误代码给客户端，让用户知道这不是错误是限流，可以稍后再试。
- 因为我们的系统很多地方都会做限流，在监控图上我们最好对这类限流的曲线有敏感，限流后的曲线是一下子失去了增长的梯度变为了平稳的状态，如果监控图看的时间范围过小的话会误判这是一个正常的请求量。
- 限流可以在边缘节点做。我们来考虑秒杀的场景，如果一秒有 100 万个请求，这 100 万个请求全部打到我们的应用服务器没有意义，我们可以在边缘节点（CDN）甚至上做简单的

边缘计算，让这 100 万个请求采用命中注定的方式直接随机放弃其中的 99.9%留下 1000 个请求，最终可以进入我们的业务服务，这样 TPS 在 1000 一般是没有问题的。所以很多时候我们参与秒杀系统会在极端的时间内毫无思考告知你活动已结束，说明你已经是被选中的命中注定的无法进入后端系统来参与秒杀的那些人。

在下篇中我们将会继续介绍数据、安全、消息、弹性方面的一些架构模式。