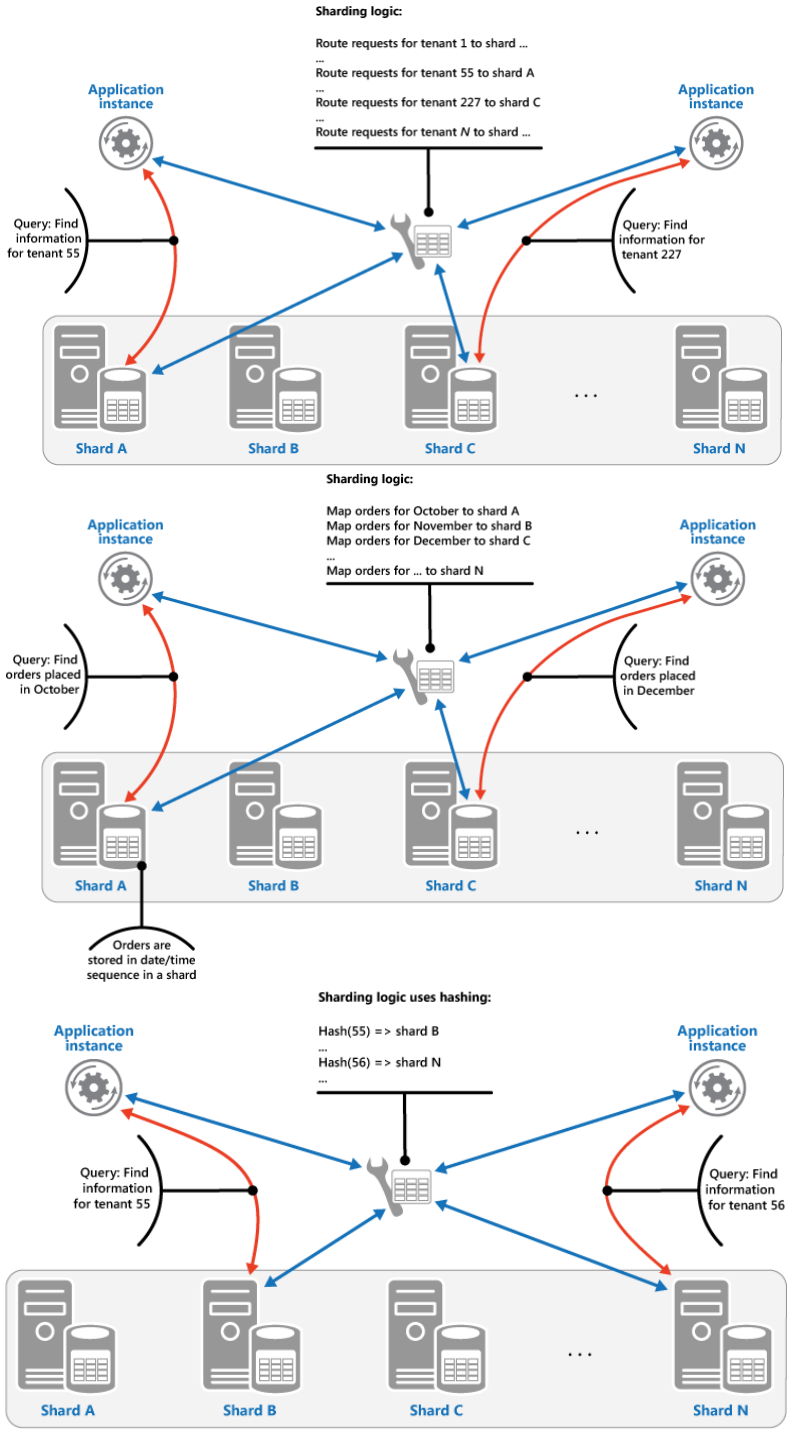
# 朱晔的互联网架构实践心得 S1E8：三十种架构设计模式（下）

【下载本文 PDF 进行阅读】

接上文，继续剩下的 15 个模式。

# 数据管理模式

## 16、分片模式：将数据存储区划分为一组水平分区或分片




一直有一个说法就是不到没路可走的时候不要考虑数据库分片。有的时候业务量大到单个业务表在经过缓存+队列削峰等措施之后的平均的 TPS 超过 1 万，单表实在是扛不住，还是只能考虑分片手段。

分片前：

- 需要根据数据分布、压力情况、业务逻辑确定分片的方式，按照条件还是范围还是哈希等等（三个图展示了三种策略）。
- 需要进行业务代码改造，改掉所有不允许的 SQL。
- 需要确定用 HardCode 方式还是框架方式还是中间件方式做数据路由。

分片后：

- 需要有运维工具可以对这么多套分片的数据进行统一的加索引等操作。
- 最好有数据仓库可以汇总所有数据，使得 adhoc 查询可以更方便。
- 最好有辅助工具可以用来帮助定位数据会在哪个分片中。

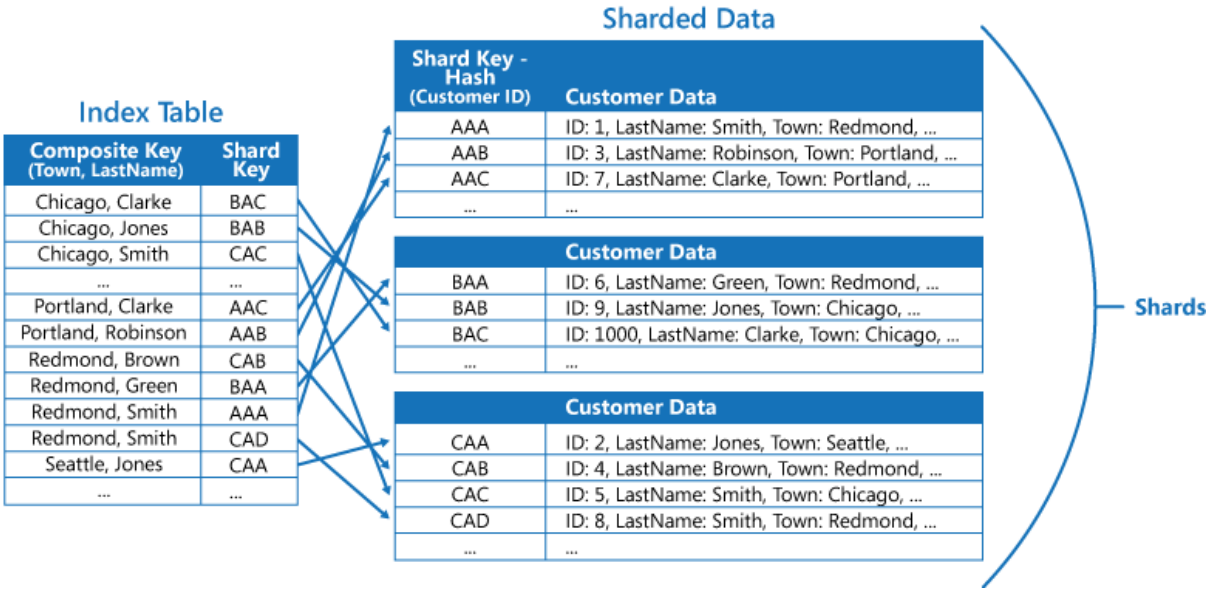
## 17、静态内容托管模式：将静态内容部署到基于云的存储服务，可以将它们直接传递给客户端

相信互联网公司 90%+肯定都使用了这个模式。把静态资源从动态网站中剥离由 Nginx 等高性能服务器来处理静态资源，然后使用三方 CDN 对静态资源进行加速，不但减轻了动态网站的负载而且数据在边缘节点加速让用户的访问跟快，使用单独的一个或多个子域名做静态资源还能提高下载资源的并行度提高网页加载的速度。

使用 CDN 来进行资源加速一般有主动数据传送到 CDN 存储和在 CDN 配置回源站拉取两种方式，文件类一般使用主动推送数据，静态资源类一般使用回源方式。在使用 CDN 的时候考虑下面的问题：

- CDN 以什么方式来认定同一个文件的，CDN 提供了什么工具来刷新边缘节点的缓存？根据不同的策略做相应的缓存刷新方案。
- 源站对于相同的文件需要有一致性（最好版本变化后文件名变化），不能今天是这个版本明天是另一个版本，这样很可能导致边缘节点缓存了不同版本的文件，导致各种怪问题。
- 使用了 CDN 后不同地区的用户访问的都是 CDN 节点上的数据，一旦出现问题排查比较困难，考虑引入前端的错误处理框架来记录前端出现脚本错误时的调用栈，方便定位问题。

### 18、索引表模式：为查询经常引用的数据存储区中的字段创建索引



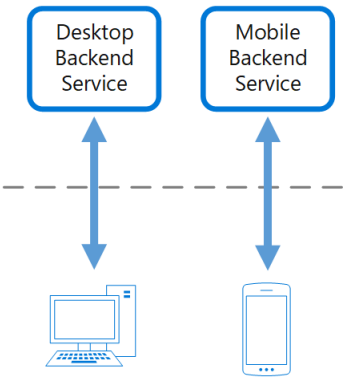

在第三第五两篇文章中我都提到了索引表的做法。出于下面的原因，我们会考虑索引表：

- 虽然我们的关系型数据库大多支持主键之外的非聚集索引，但是在某些情况下直接对大表做很多索引性能并不好。
- 做了 Sharding 后我们确实没有办法以分片键之外的维度来查询数据。
- 希望以空间换时间，直接把某个维度的复合查询作为主键单独保存一份数据。

不过需要考虑一点索引只有在数据区分度高的情况下才能发挥价值，如果 90%以上的数据都是相同的值，那么走索引进行查询性能会比全表扫还要差一点。

## 设计和实现模式

### 19、前端专用的后端模式：通过使用单独的接口来分离读取数据和更新数据的操作

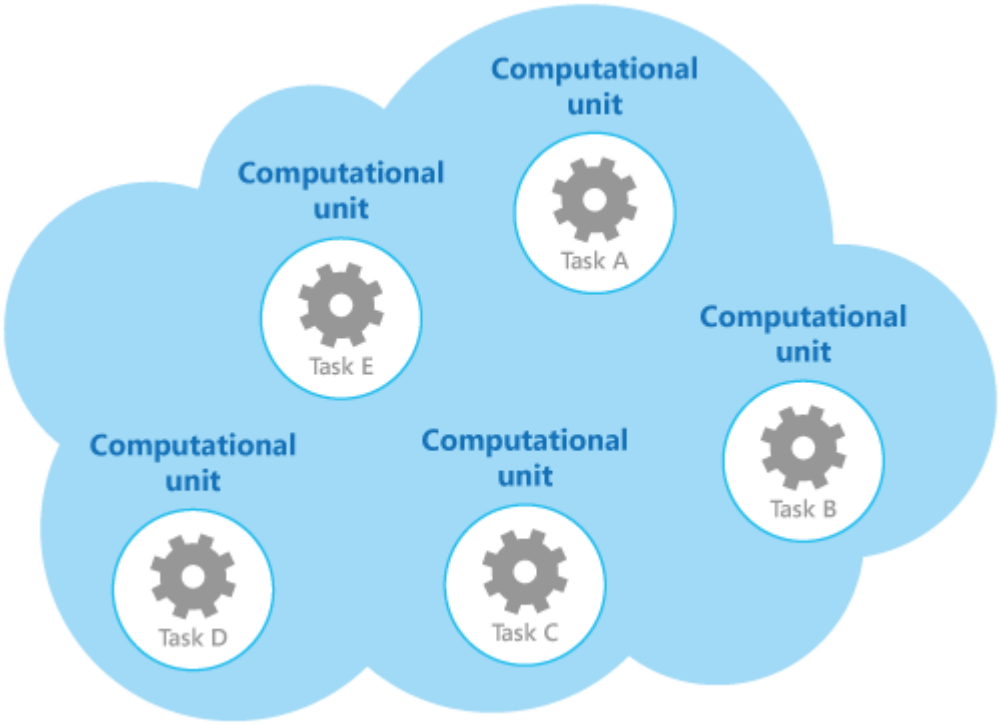


这里说的是不同的前端配以不同的专用后端。比如 PC 网站和 APP 的后端是两套程序。这种模式是否适合其实还是看两端的后端提供的数据差异有多大，我们总是希望可以尽量统一一套后端，业务逻辑不用重复写，但是我们要考虑到 PC 网站和 APP 的差异性：

- APP 系统的接口交互一般会签名验证，有的时候还会加密
- PC 系统的流程一般和 APP 系统不一样
- PC 一个页面能显示的内容会比 APP 一个界面显示的更多
- 安全性设计上 PC 和 APP 不一样，APP 很少有图形验证码

考虑到这些差异，我们是在一个工程内根据来源做适配，还是独立两套工程来做独立的后端取决于差异度有多大了。

## 20、计算资源整合模式：将多个任务或操作整合到单个计算单元中




这个模式从资源节省的角度来说我们的计算单元任务可以进行一些合并，减少因为资源限制导致不必要的开销。

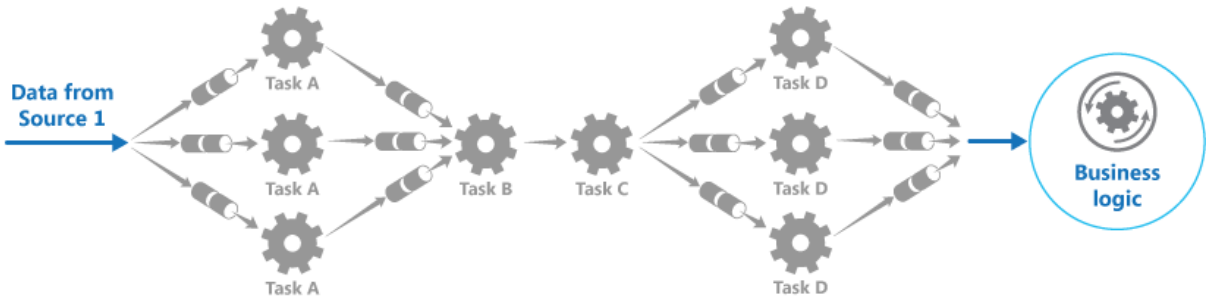
## 21、选举模式：通过选举一个实例作为负责管理其它实例的负责人，来协调分布式应用程序中的协作任务实例集合执行的操作

对于分布式服务，我们趋向于把服务设计为无状态可以任意扩展的，但是在某些业务场景下我们不得不在服务中选举出一个 Leader（Primary 节点，Master 节点）来做一些不适合重复做的协调管理工作。这个时候我们需要有算法来做选举。

最常见的实现方式是使用 Zookeeper 来实现，我们知道 ZK 的 znode 有 Sequence 和 NonSequence 两种，前者多个客户端只有一个可创建成功同名节点，后者创建后会自动加上序列号命名多个客户端可以创建多个同名节点，利用这个特性有两种常见实现方式：

- 非公平实现。多个客户端同时创建 EPHEMERAL+NONSEQUENCE 节点。只有一个可以创建成功，创建成功的就是 Leader，其它的 Follower 需要注册 watch，一旦 Leader 放弃节点（注意，EPHEMERAL 意味着 Leader 待机后 Session 结束节点被删除），再一次重复之前的过程注册节点抢占成为 Leader。这个模式实现简单，问题是在节点数量过多的时候一旦发生重新竞选，这个时候可能会有性能问题。
- 公平实现。多个客户端同时创建 EPHEMERAL+SEQUENCE 节点。客户端都可以创建成功节点，客户端如果判断自己是最小的节点则为 Leader 否则为 Follower。每一个 Follower 都去 watch 序号比自己小的节点（大家都看前一位）。一旦有 Leader 节点因为宕机被删除（还是 EPHEMERAL 特性），收到通知的节点会看自己是不是最小的序号，如果是的话成为 Leader。节点宕机后，原先 watch 宕机节点的客户端重新 watch 比自己序号小的有效节点。这个模式实现复杂，但是由于 watch 的都只是一个节点所以不会发生像非公平实现的性能问题，而且竞选根据节点序号来而不是抢占式所以显得 Leader 的选举公平有序。

## 22、管道和过滤器模式：将需要执行复杂处理的任务分解成可以重复使用的一系列单独的元素



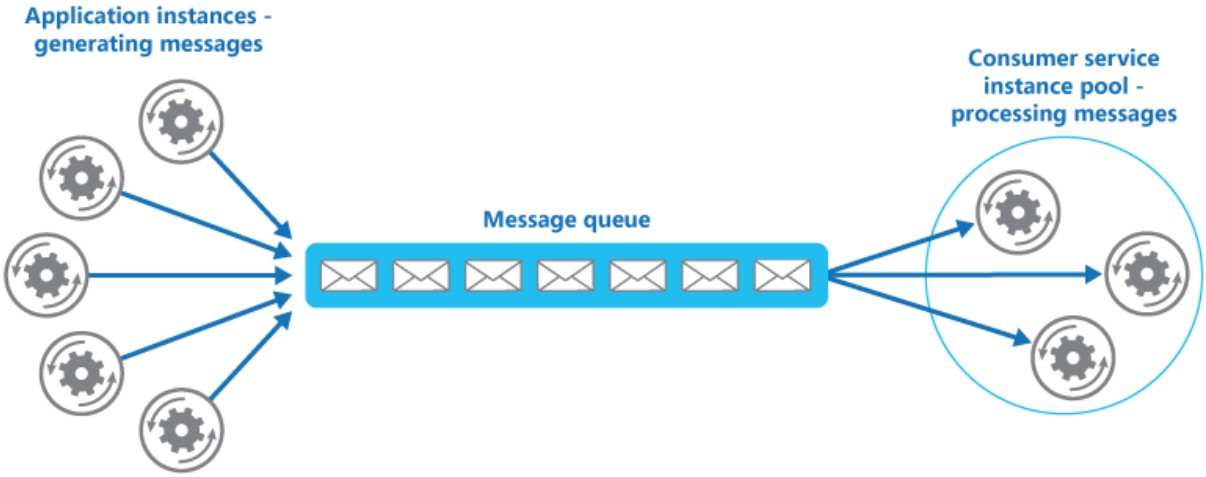

在软件设计模式中过滤器构成的管道这种模式很常见（图上的业务逻辑就是 Handler，之前的那些 Task 就是 Filter，模式上可以是 Filter+Handler 也可以是 Filter+Handler+Filter 也可以是

Handler+Filter），不管是 Spring MVC 框架也好，Netty 这种网络框架也好都提供了这样的设计。每一个过滤器单独完成一个功能，可以独立插拔随意组合配置成一套管道，不但数据处理的整个过程清晰可见还增加了灵活性。

对于架构上也可以有这样的模式，在数据源进入到业务逻辑处理之前（或之后，或前后），我们可以配置一系列的数据过滤器完成各种数据转化和处理的任务。Task 和 Task 之间可以是同步调用，也可以使用 MQ 做一定的可伸缩性设计。还可以把过滤器的配置信息保存在配置系统中甚至根据上下文动态构建出管道，实现更灵活的前置或后置流程处理。

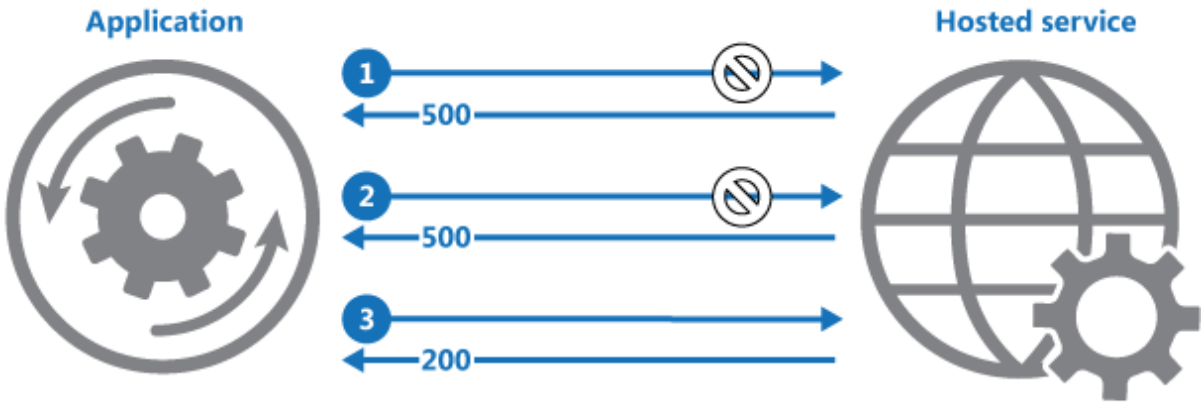
## 消息模式

### 23、竞争消费者模式：使用多个并发消费者来处理在同一消息通道上接收的消息




这里说的是消息队列的消息消费者是一组对等的消费者，通过竞争方式来拉取数据执行。之前提到过这是 MQ 的最常见的一种模式，一般而言我们会部署多个消费节点进行负载均衡，在负载较大的时候可以方便得增加消费者进行消费能力扩容。不过对于这种模式消费者应当是对等的无状态的，在某个消费者在消费失败的时候消息重新回到队列随后可能会被另一个消费者进行处理。

## 24、重试模式：在应用程序尝试连接到服务或网络资源遇到预期的临时故障时，让程序通过透明地重试以前失败的操作来处理




1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

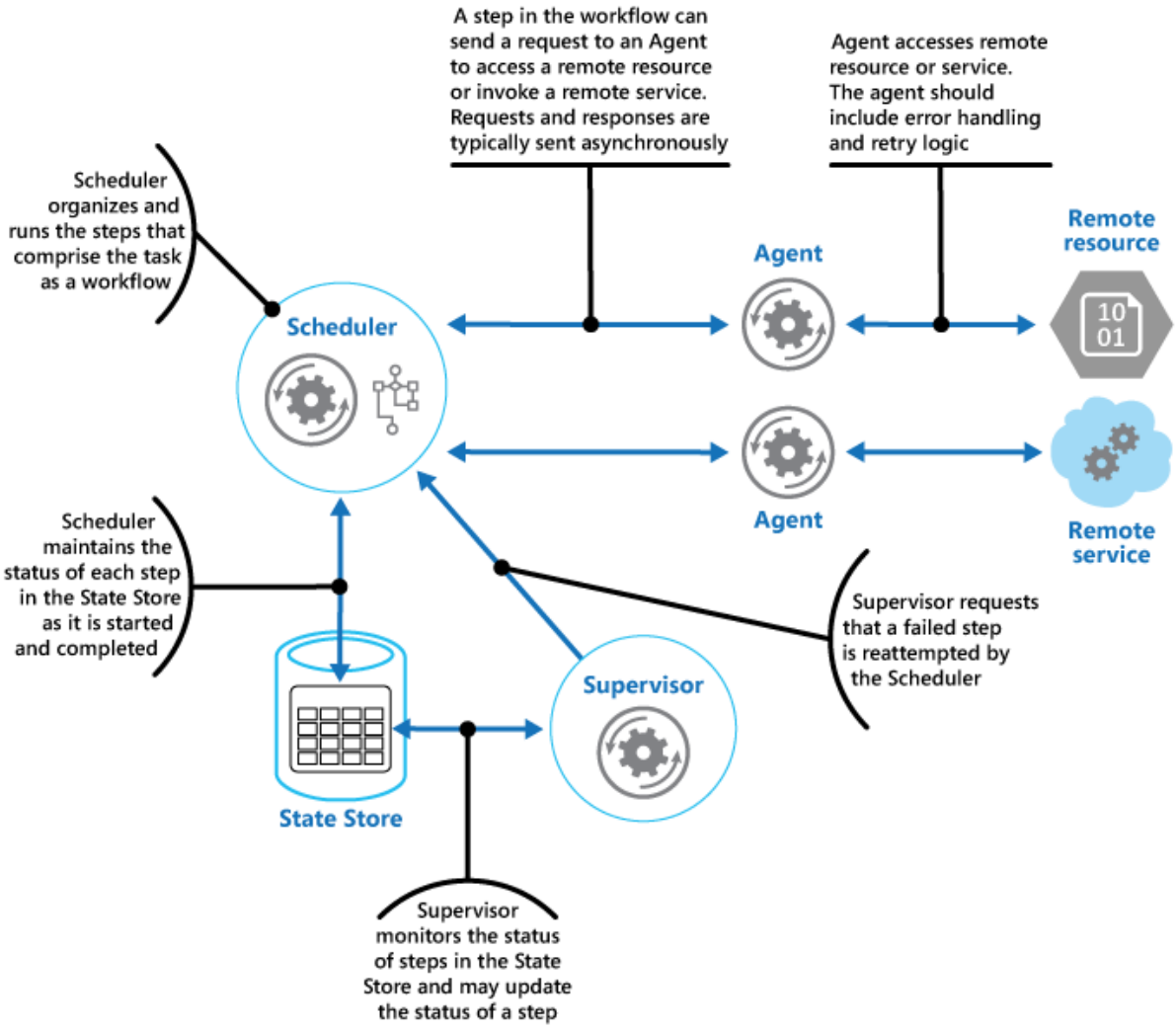
重试适用于瞬态故障，之后会提到断路器模式，两种模式可以结合使用。首先说说重试的几个发起人：

- 让用户自己发起，遇到错误的时候及时返回错误信息，让用户自己稍后重试整个业务功能。这种方式不容易产生瞬时的压力，但是体验较差。
- 在中间件自动发起，比如在 RPC 调用的时候遇到服务超时自动进行一定次数的重试，这样可以在外部没有感知的情况下有一定概率消除错误。这个方式要求服务是支持重试的。
- 由业务逻辑手动发起，不同的业务逻辑根据需求在代码中去写重试的逻辑（当然也可以通过类似 Spring-Retry 这种组件来做）。实现繁琐但是不容易出错。
- 由补偿逻辑发起进行同步转异步操作，非重要逻辑同步行则行，不行不在主流程重试，由单独的异步流程进行重试补偿。

重试也要考虑几种策略：

- 次数。最多重试几次。
- 异常。遇到什么样的异常（黑白名单）应该去重试。
- 等待。考虑每次重试是相同的间隔呢还是有一个延迟的递增，随着重试次数增加而增加延时时间。

## 25、调度、代理、主管模式：在一组分布式服务和其它远程资源之间协调一组操作



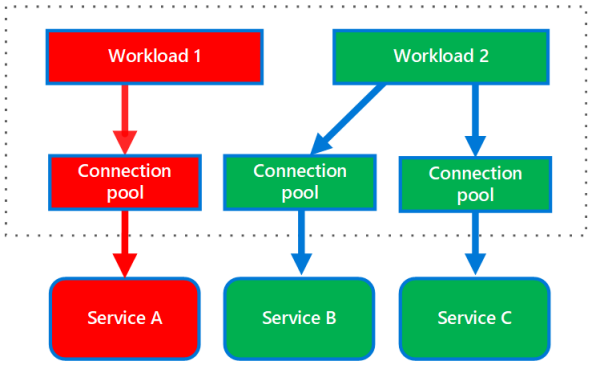

这个模式说的是三者的角色：

- 调度负责安排任务，在执行每个步骤的时候维护任务的状态，具体业务逻辑由代理负责。
- 代理负责和远程的服务和资源进行通讯，实现错误处理和重试。
- 管理者负责监视任务的执行状态，作为调度的补充，在合适的时候要求调度进行补偿。

三个角色相互配合完成复杂的，具有较多远程服务参与的任务，确保任务的最终有效执行。在之前架构三马车一文中说到定时任务的时候提到过一种任务驱动表的模式，说到了一些驱动表的实现细节，其实整体和这个模式是类似的思想。当我们的一个复杂逻辑有多个步骤构成，每一步都依赖外部服务，这个时候我们可以选择全程 MQ+补偿方式（乐观方式），也可以选择全程任务驱动的被动模式（悲观方式），具体选择取决于更看重可靠性还是及时性。

## 弹性模式

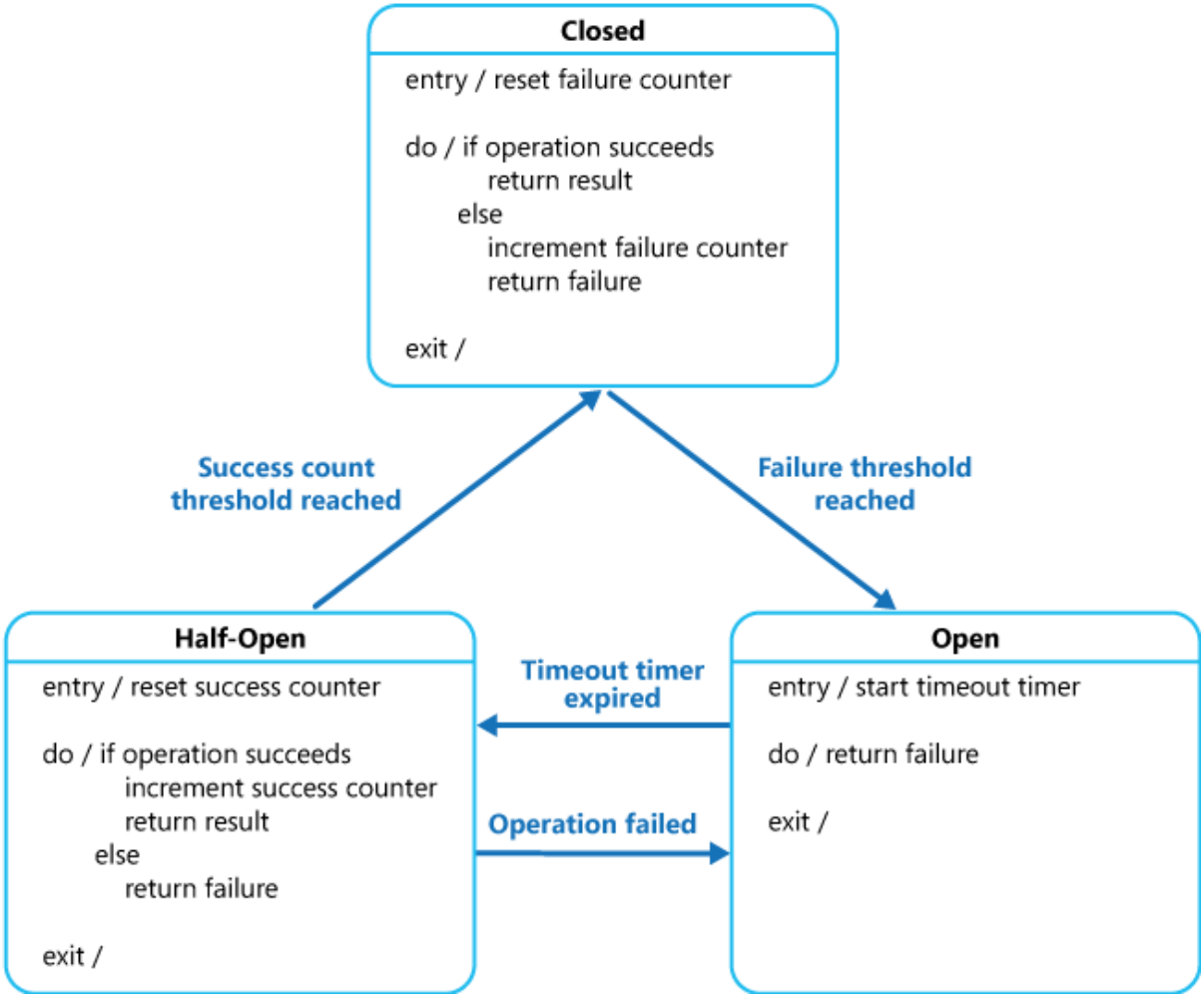
### 26、舱壁模式：将应用程序的元素隔离到池中，如果其中一个失败，其它的将继续运行




资源隔离有好几个层次，可以在进程内部做线程池或队列的隔离，在微服务的服务划分上考虑隔离出单独的物理服务，或是在服务器层面通过虚拟化技术或 Docker 技术进行资源隔离。隔离了就不会相互影响，但是会有成本、性能、管理便利性方面的开销。实现能够根据需求分析出可能的资源相互影响的点，提前规划隔离往往可以避免很多问题的发生。之前有遇到过几个事故是这样的：

- 程序内部大量使用了 Java8 的 ParallelStream 特性进行并行处理，由于默认共享了相同的线程池，某一个业务的执行占满了线程影响了其它业务的正常进行。
- 消息队列因为没有对执行过多次失败的死信消息和正常的新消息进行隔离，导致一些业务下线后无法处理的死消息占满了整个队列，正常消息无法消费。
- 某个服务提供了类似文件上传的重量级操作，也提供了数据查询的轻量级操作，在上传业务大的时候服务的线程都被 IO 所占满，导致其它查询操作无法进行。

## 27、断路器模式：连接到远程服务或资源时，处理可能需要花费时间来修复的故障




分布式应用环节多网络环境复杂，如果遇到依赖服务调用失败的情况我们或许可以进行重试期待服务马上可以恢复，但是在某些时候依赖的服务是彻底挂了而不是网络故障无法及时恢复，如果不考虑进行熔断的，可能服务调用方会被服务提供方拖死。这个时候可以引入断路器机制，如图所示断路器一般采用三态实现，瞬间恢复可能会让底层服务压力过大：

- 关闭：出现错误的时候增加计数器
- 打开：计数器到达阈值打开断路器，直接返回错误
- 半开：超时后允许一定的请求通过，成功率达到阈值关闭断路器，操作还是失败的话还是进入打开状态

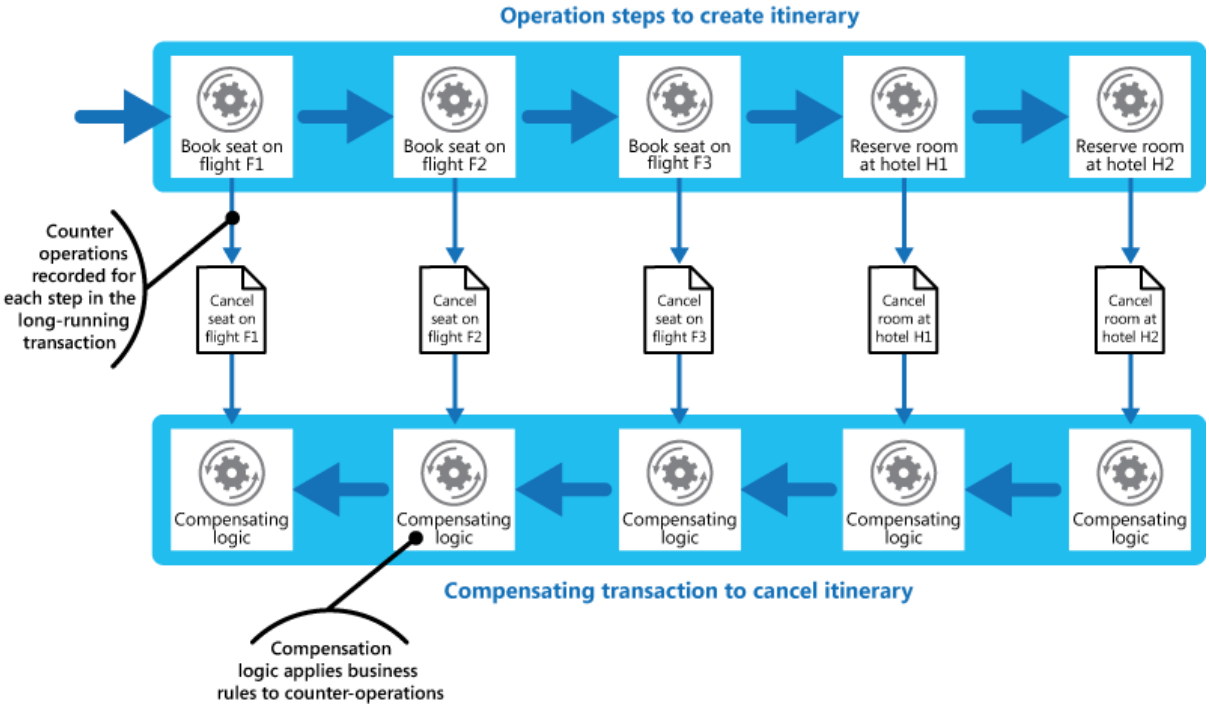
实现模式的时候考虑下面注意点：

- 考虑熔断后怎么来处理，熔断后我们肯定拿不到实际的处理结果，这个时候考虑是功能降级还是采用后备的数据提供方来提供数据
- 紧急的时候需要人工介入，最好在外部提供手动的方式可以干预断路器的三态

- 不同的业务考虑不同的断路器打开阈值，每一个错误还能有不同的权重，比如对于下游程序返回了太多请求的错误，每次错误可以+2 提高权重尽可能早断路
- 断路器应当记录熔断时的请求原始信息，在之后必要的时候可以进行重放或数据修复工作
- 注意设置好外部服务的超时，如果客户端超时比服务端短，很可能进行错误的熔断

实现上我们可以看一下 Netflix 的 Hystrix 进行进一步了解。

## 28、事务补偿模式：撤消通过一系列步骤执行的工作，它们一起定义最终一致的操作




这个模式说的是失败时必须进行撤销的操作，可以由一组补偿程序来做相应的补偿。在这里我想说的更广一点，在服务调用的时候，调用失败有几种可能：

- 请求客户端发出但是没到服务端，业务逻辑没有执行
- 请求客户端发出服务端收到也处理成功了，业务逻辑执行了客户端没收到结果
- 请求客户端发出服务端收到但处理失败了，客户端没收到结果

所以在出现服务调用失败或超时的时候，服务端执行究竟有没有成功客户端是不明确的（只有客户端收到了明确的服务端返回的业务错误才真正代表执行失败），这个时候需要有补偿逻辑来同步服务端的执行状态。如果这样的补偿不可避免而且需要补偿的服务特别多，这样的逻辑逐一来写是一件很烦的事情，我们可以把这个工作封装成一个补偿中间件来处理：

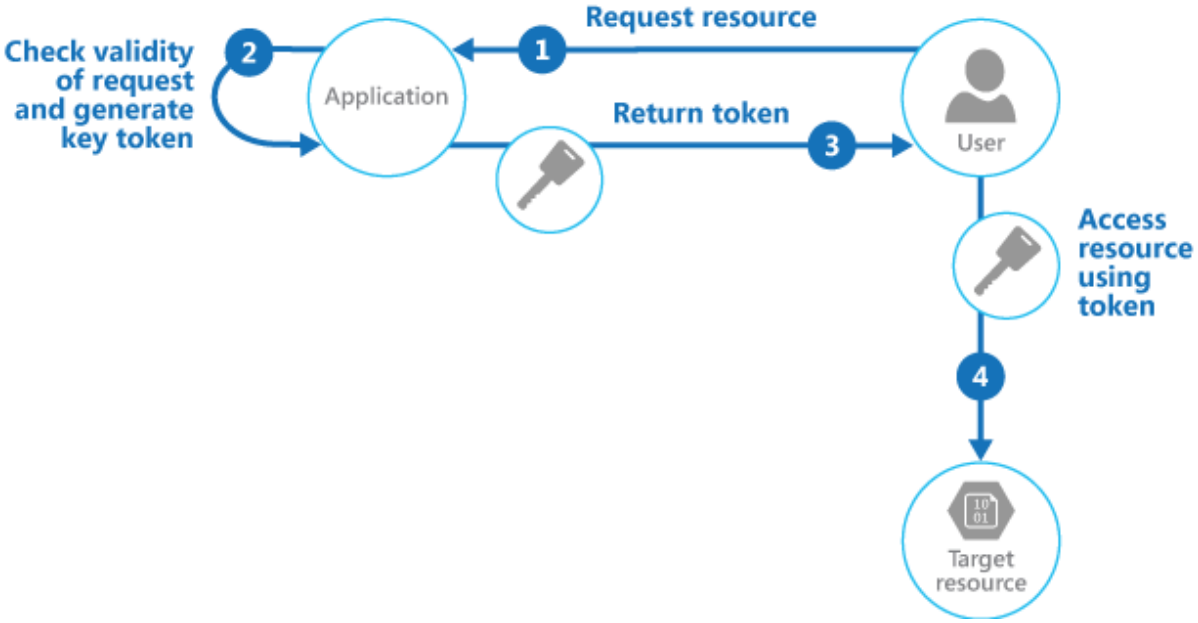
- 所有关键服务调用标记为需要自动补偿
- 补偿中间件在数据库记录服务的调用状态

- 关键服务的提供者提供统一状态查询接口，消费者提供统一的补偿回调接口（来处理成功和失败的情况）
- 补偿中间件根据数据库的记录调用服务提供方的状态查询和服务消费方的补偿回调接口进行补偿

这样，我们在服务调用的时候就不需要考虑补偿逻辑的实现，只要实现这个标准即可。

## 安全模式

### 29、代客密钥模式：使用向客户端提供对特定资源或服务的有限直接访问权限的令牌或密钥

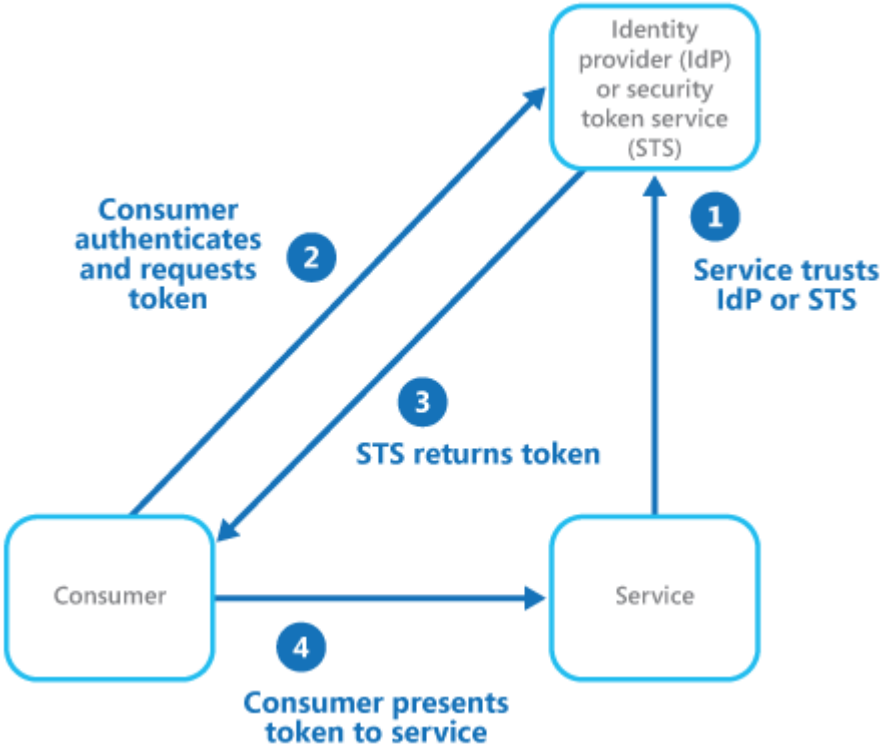



这个模式说的是，在访问敏感资源的时候，我们可以不必让应用程序在其中作为一个代理转一层做权限控制，而是生成一个密钥，让用户直接拿着密钥到资源池换数据。

一些 CDN 在提供资源上传下载服务的时候一般都会提供类似的安全策略，需要实现生成 Token 才能去使用下载和上传服务，避免了 CDN 数据被非法利用作为图床的可能。

实现上比较简单，应用程序和资源提供方约定好 Token 的生成算法，对资源+请求资源的时间+密钥联合在一起做签名，资源提供方如果校验到签名不正确或 Token 过期或资源不匹配都将拒绝服务。

## 30、联合身份模式：将认证委托给外部身份提供者




这个模式说的是将身份验证委托给专门的程序或模块来做。使用专门的模块来统一负责登录授权不仅仅可以提供单点登录的功能，而且服务实现上更简单不需要每次都考虑登录那套东西。实现上可以看一下 Spring Security 实现的 OAuth 2.0。

总结一下，对于其中的很多模式，我们可以发现其实在之前的一些介绍或多或少有一些涉及。这里提到的 30 种模式有些体现的是一些设计细节，有些体现的是一种设计理念，它们大多时候是组合使用的，适合的就是最好的，大家可以细细品味一下每种模式的适合场景，在合适的时候可以想到它或许会有一种豁然开朗的感觉。