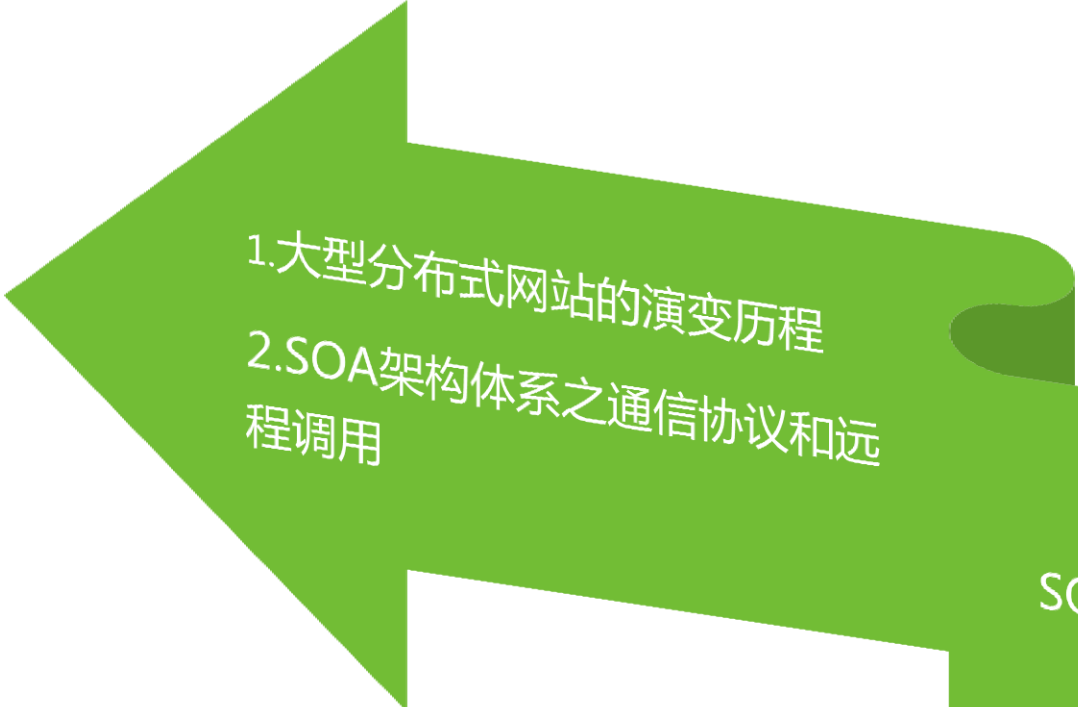




大型电商分布式系统实践 第2周

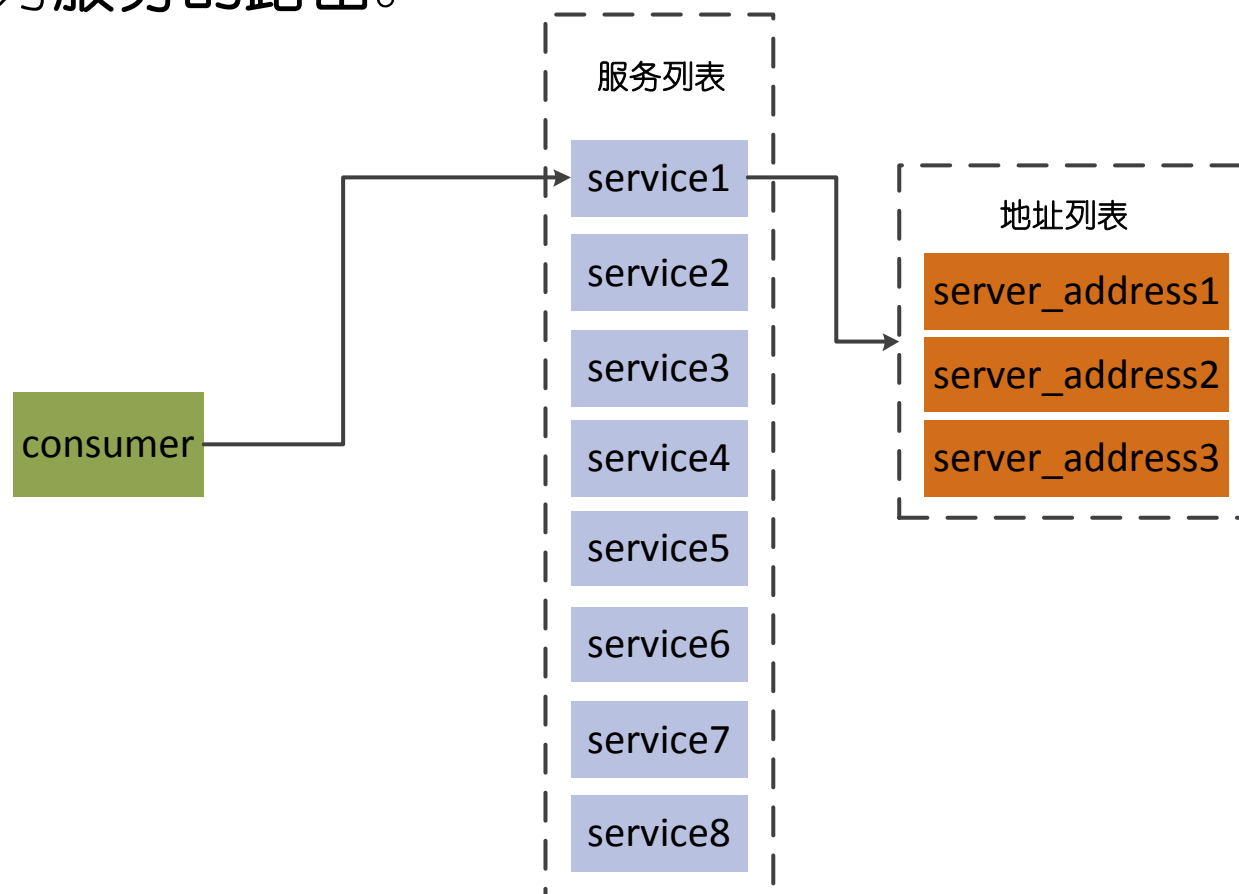
DATAGURU专业数据分析社区

- 
- 1.大型分布式网站的演变历程
 - 2.SOA架构体系之通信协议和远程调用

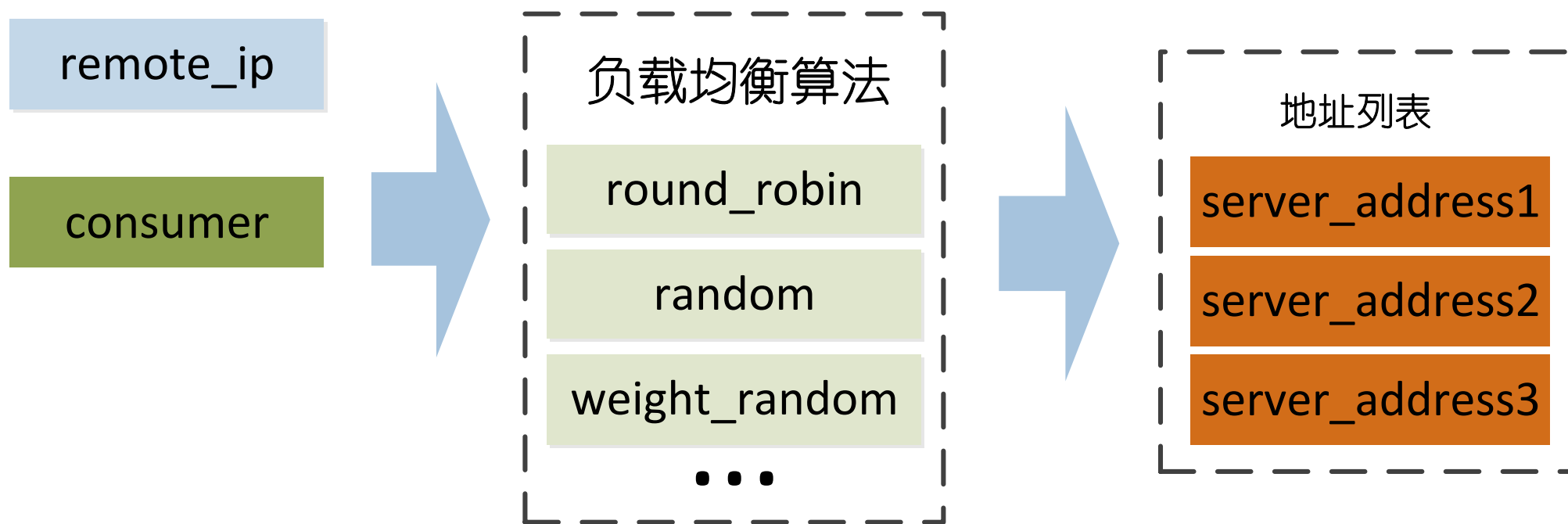


SOA架构体系之服务路由和服务治理

SOA架构中，服务消费者通过服务名称，在众多服务中找到要调用的服务的地址列表，称作为**服务的路由**。

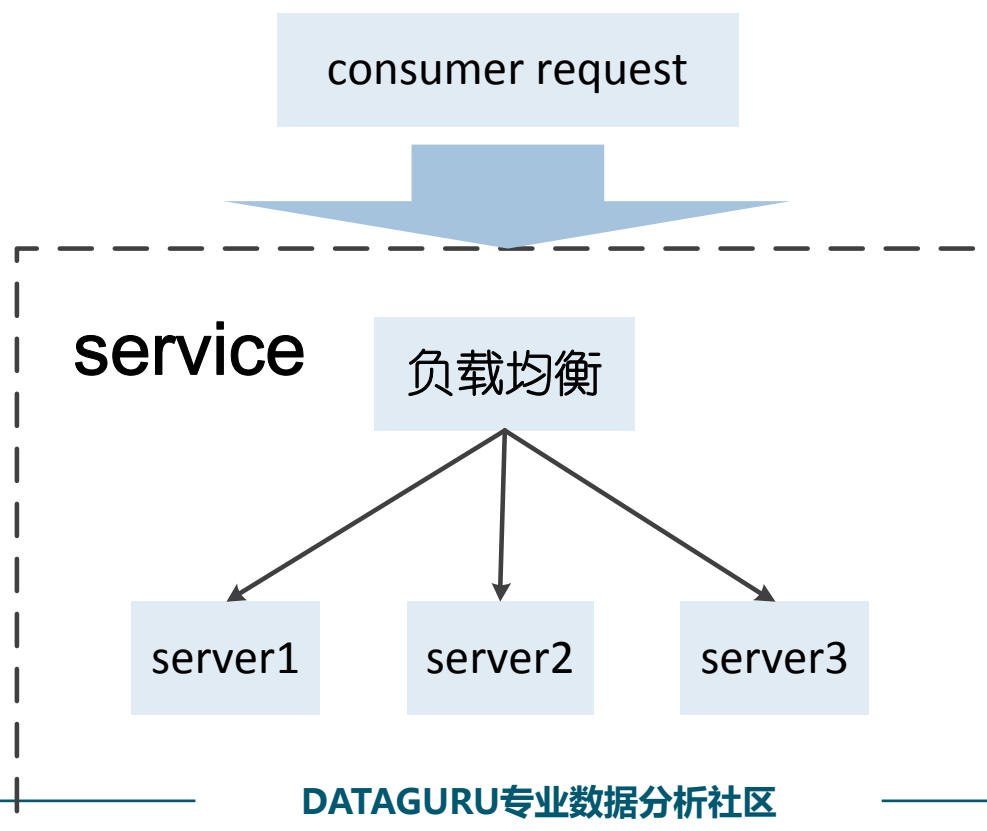


对于负载较高的服务来说，往往对应着由多台服务器组成的集群。在请求到来时，为了将请求均衡地分配到后端服务器，负载均衡程序将从服务对应的地址列表中，通过相应的负载均衡算法和规则，选取一台服务器进行访问，这个过程称为**服务的负载均衡**



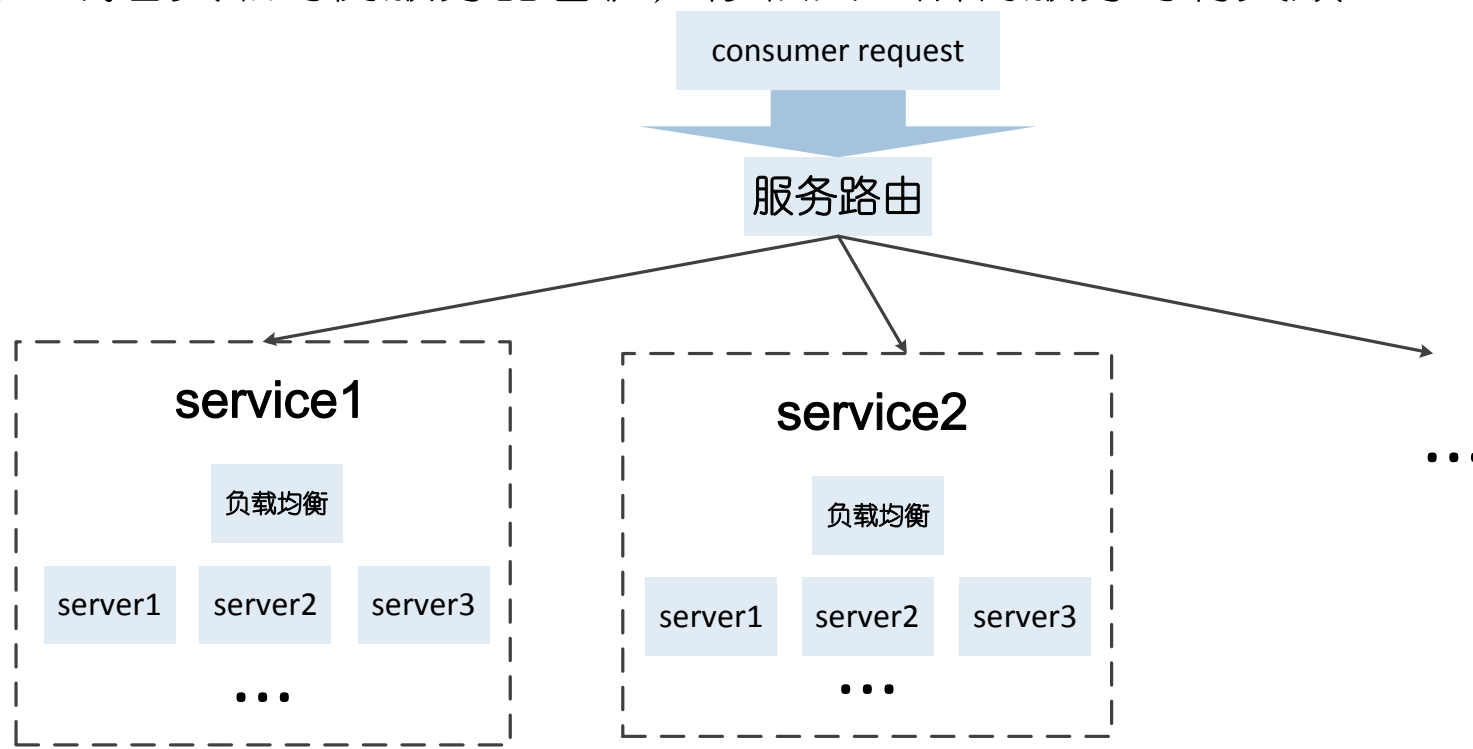
单个服务的负载均衡

当服务的规模较小时，可以采用硬编码的方式将服务地址和配置写在代码中，通过编码的方式来解决服务的路由和负载均衡的问题，也可以通过传统的硬件负载均衡设备如F5等，或者是采用LVS或nginx等软件解决方案，通过相关配置，来解决服务的路由和负载均衡问题。由于服务的机器数量在可控范围内，因此维护成本能够接受。



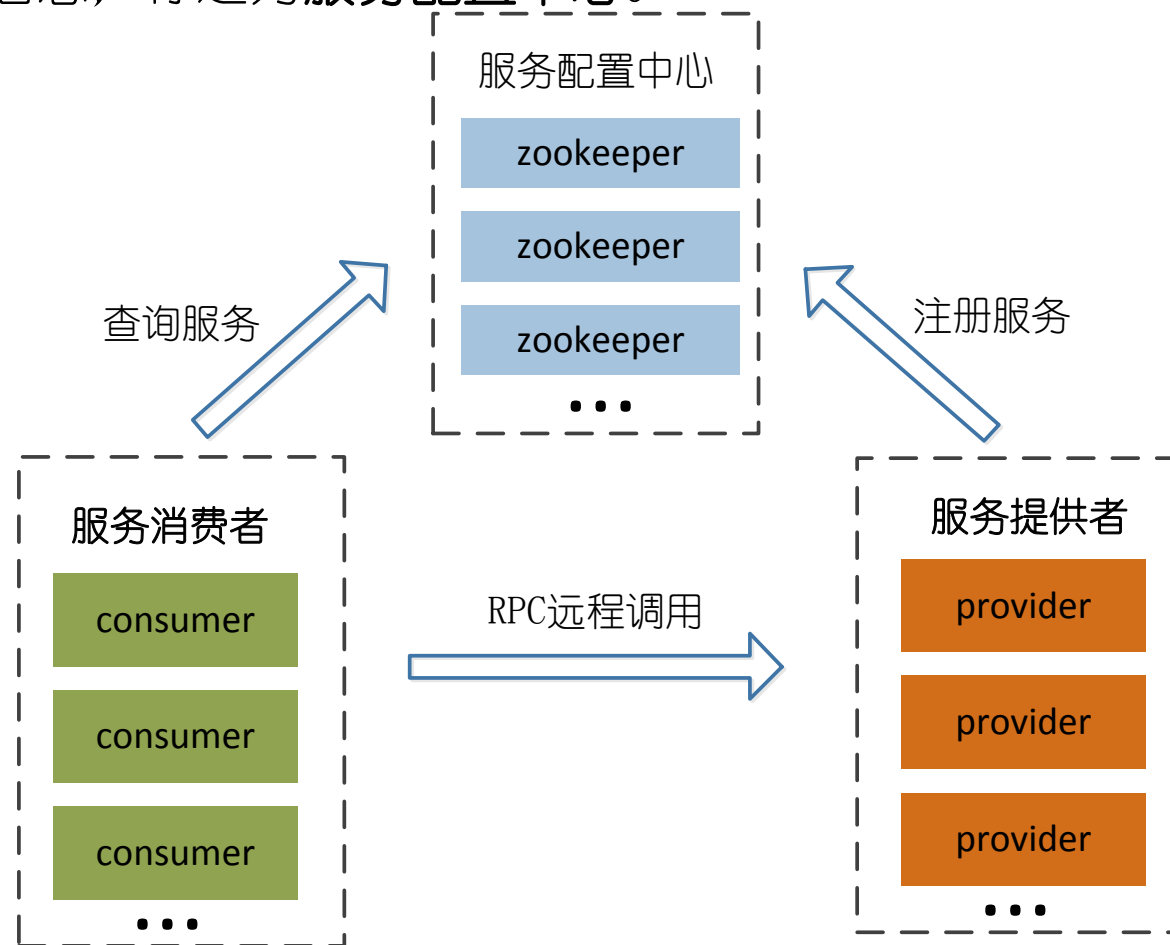
多服务路由及负载均衡

当服务越来越多，规模越来越大，对应的机器数量也越来越庞大，单靠人工来管理和维护服务及地址的配置信息，已经越来越困难。并且，依赖单一的硬件负载均衡设备或者使用LVS、nginx等软件方案进行路由和负载均衡调度，单点故障的问题也开始凸显，一旦服务路由或者负载均衡服务器宕机，依赖其的所有服务均将失效。



服务配置中心

此时，需要一个能够动态注册和获取服务信息的地方，来统一管理服务名称和其对应的服务器列表信息，称之为**服务配置中心**。



服务提供者在启动的时候，将其提供的服务名称，服务器地址注册到服务配置中心，服务消费者通过服务配置中心来获得需要调用的服务的机器列表，通过相应的负载均衡算法，选取其中一台服务器进行调用，当服务器宕机或者下线的时候，相应的机器需要能够动态的从服务配置中心里面移除，并通知相应的服务消费者，否则服务消费者就有可能调用到已经失效的服务而发生错误。

在这个过程中，服务消费者只有在第一次调用服务的时候需要查询服务配置中心，然后将查询到的信息缓存到本地，后面的调用直接使用本地缓存的服务地址列表信息，而不需要重新发起请求到服务配置中心去获取相应的服务地址列表，直到服务的地址列表有变更(机器上线或者下线)，这种无中心化的结构，解决了之前负载均衡设备所导致的单点故障的问题，并且大大减轻了服务配置中心的压力。

zookeeper是Hadoop下的一个子项目，它是一个针对大型分布式系统的可靠的协调系统，提供的功能包括：配置维护、名字服务、分布式同步、组服务等。zookeeper是可以集群复制的，集群间通过Zab(Zookeeper Atomic Broadcast)协议来保持数据的一致性。

基于zookeeper的持久和非持久节点，我们能够近乎实时的感知到后端服务器的状态(上线、下线、宕机)。通过集群间zab协议，使得服务配置信息能够保持一致。而zookeeper本身容错特性以及leader选举机制，能保障我们方便的进行扩容。通过zookeeper来实现的服务动态注册、机器上线与下线的动态感知，扩容方便，容错性好，且无中心化结构能够解决之前使用负载均衡设备所带来的单点故障问题，只有当配置信息更新时才会去zookeeper上取最新的服务地址列表，其他时候使用本地缓存即可。

轮询(Round Robin)法，轮询很容易理解，将请求按顺序轮流分配到后端服务器上，它均衡的对待后端每一台服务器，而不关心服务器实际的连接数和当前的系统负载。

随机(Random)法，通过系统随机函数，根据后端服务器列表的大小值，来随机选取其中一台进行访问，由概率统计的理论可以得知，随着调用量的增大，其实际效果越来越接近于平均分配流量到每一台后端服务器，也就是轮询的效果。

源地址哈希(Hash)法，源地址哈希的思想是获取客户端访问的ip地址值，通过哈希函数计算得到一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是要访问的服务器的序号。采用哈希法进行负载均衡，同一ip地址的客户端，当后端服务器列表不变的时候，它每次都会被映射到同一台后端服务器进行访问。

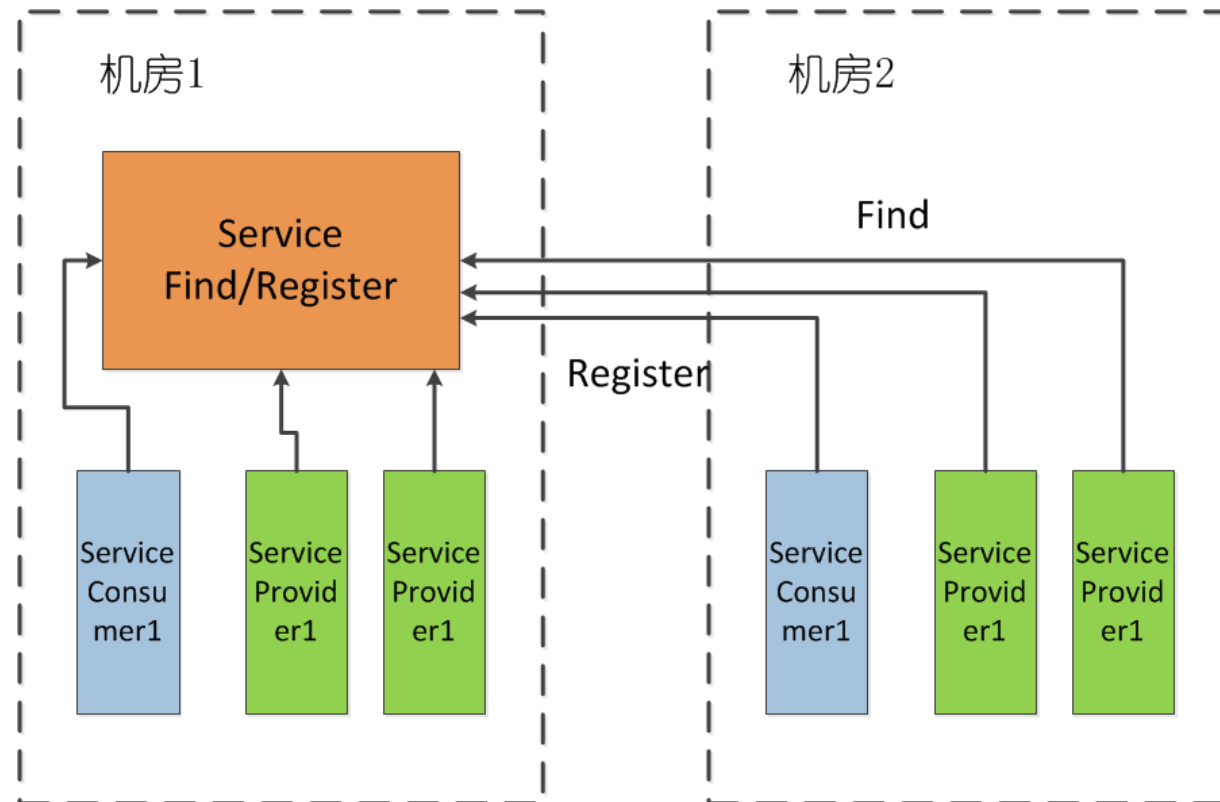
加权轮询(Weight Round Robin)法，不同的后端服务器，可能机器的配置和系统当前的负载并不相同，因此他们抗压能力也不尽相同，给配置高负载低的机器配置更高的权重，让其处理更多的请求，而低配置负载高的机器，则给其分配较低的权重，降低其系统负载，加权轮询能很好的处理这一问题，并将请求顺序且按照权重分配到后端。

加权随机 (Weight Random) 法，与加权轮询法类似，加权随机法也根据后端服务器不同的配置和负载情况，配置不同的权重，不同的是，其实按照权重来随机选取服务器，而非顺序。

最小连接数 (Least Connections) 法，最小连接数算法比较灵活和智能，由于后端服务器配置不尽相同，对于请求的处理有快有慢，它正是根据后端服务器当前的连接情况，动态的选取其中一台当前积压连接数最少的服务器，来处理当前请求，尽可能的提高后端服务器的利用效率，将负载合理的分流到每一台机器。

服务服务负载均衡—多机房场景

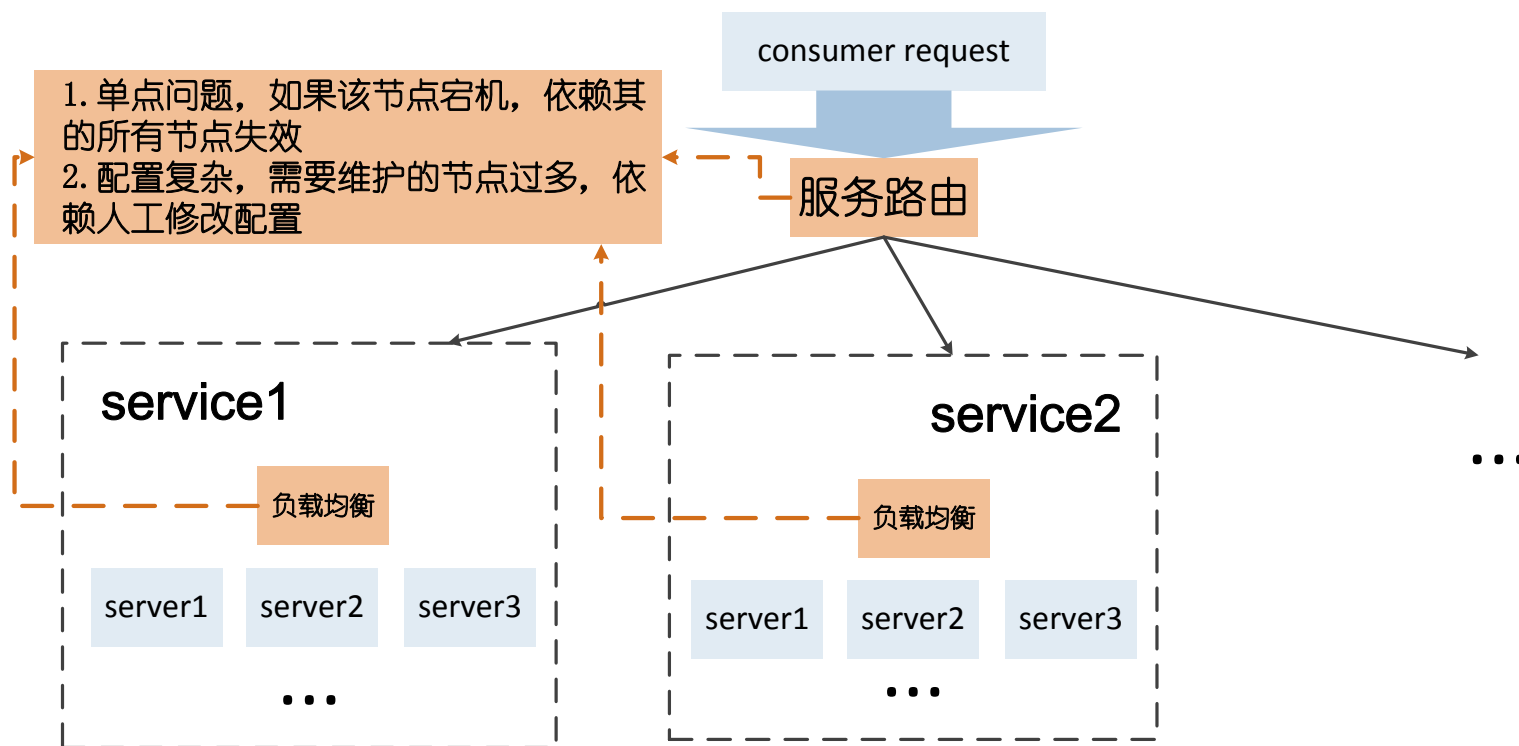
每个机房都有自己的容量上限，如果网站的规模非常大，就需要多个机房来支撑了，机房之间可能是跨地域的，机房之间的距离决定了我们采用什么样的架构策略。



先不考虑服务配置中心的跨机房问题，因为服务的信息是可以本地缓存的，来重点看一下服务消费者和服务提供者在多机房下的情况。

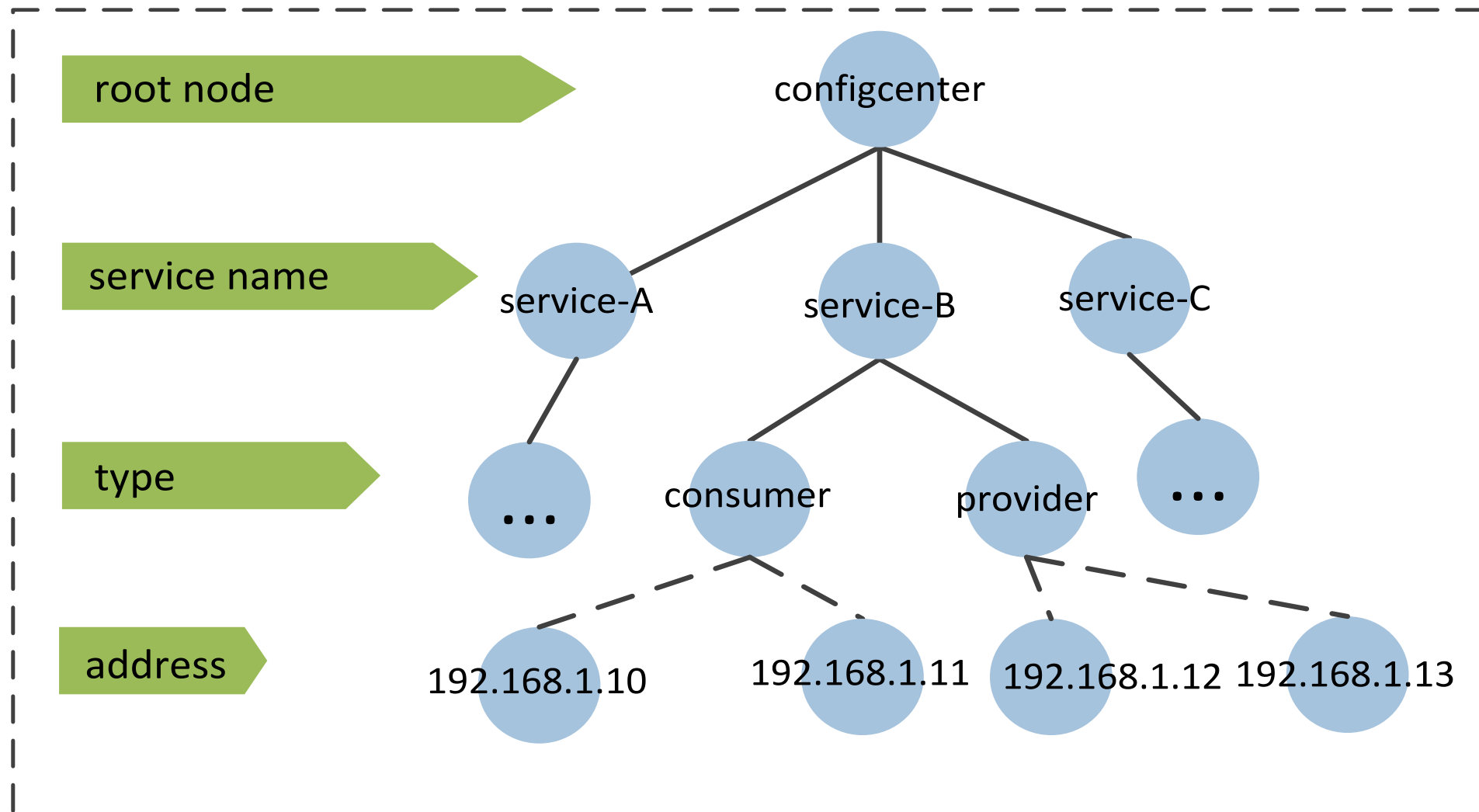
1. 对于只在接口中增加方法，这种情况比较简单，直接增加方法便可。需要使用新方法的调用者直接使用新方法，原来的调用者继续使用原来的方法。
2. 对于需要对接口中某些方法修改调用参数列表的情况，这种情况相对复杂，我们有几种方式来进行应对：
 - A. 对使用原来方法的代码都进行修改，然后和服务端一起发布。这仅仅从理论上来说可行，但是实际操作起来非常困难，因为依赖这个服务提供者的系统可能非常之多，要求这些系统同时发布，很难实现。
 - B. 通过版本号来解决，这是比较常用的方式，使用老方法的系统继续调用原来版本的服务，而需要使用新方法的系统则使用新版服务。
 - C. 在设计方法上考虑参数的扩展性，这是一个可行的方式，但是不太好，因为参数列表可扩展一般意味着采用类似map的方式来传递参数，这样使得参数校验会比较麻烦。

借助zookeeper实现路由与负载均衡



当服务越来越多, 规模越来越大, 对应的机器数量也越来越庞大, 单靠人工来管理和维护服务及地址的配置信息, 已经越来越困难。并且, 依赖单一的硬件负载均衡设备或者使用LVS、nginx等软件方案进行路由和负载均衡调度, 单点故障的问题也开始凸显, 一旦服务路由或者负载均衡服务器宕机, 依赖其的所有服务均将失效。如果采用双机高可用的部署方案, 使用一台服务器stand by, 能部分解决问题, 但是鉴于负载均衡设备的昂贵成本, 也难以全面推广。

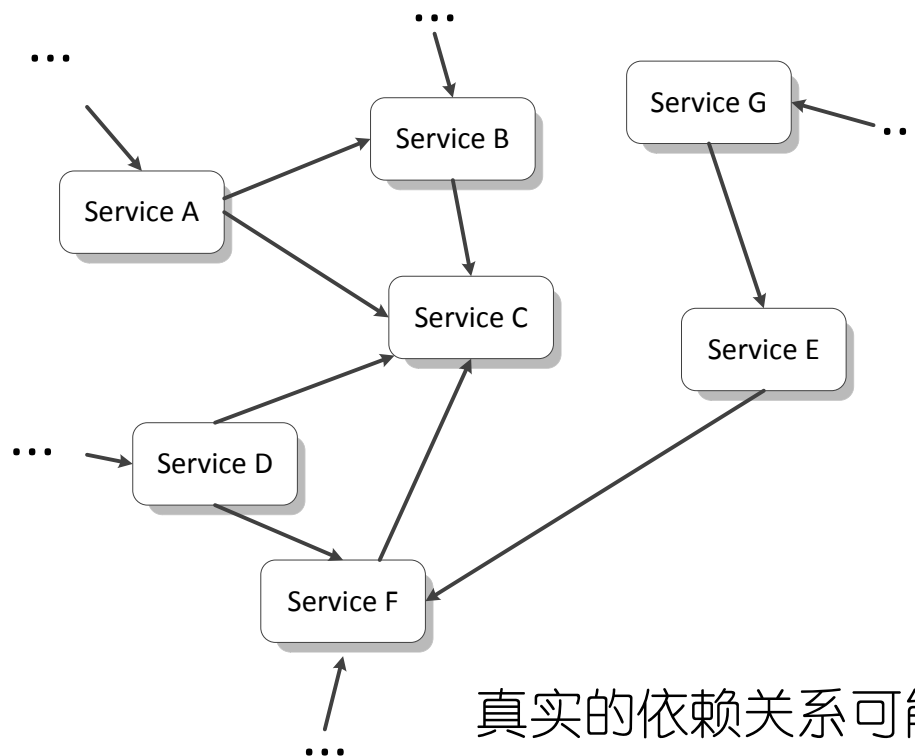
借助zookeeper实现路由与负载均衡



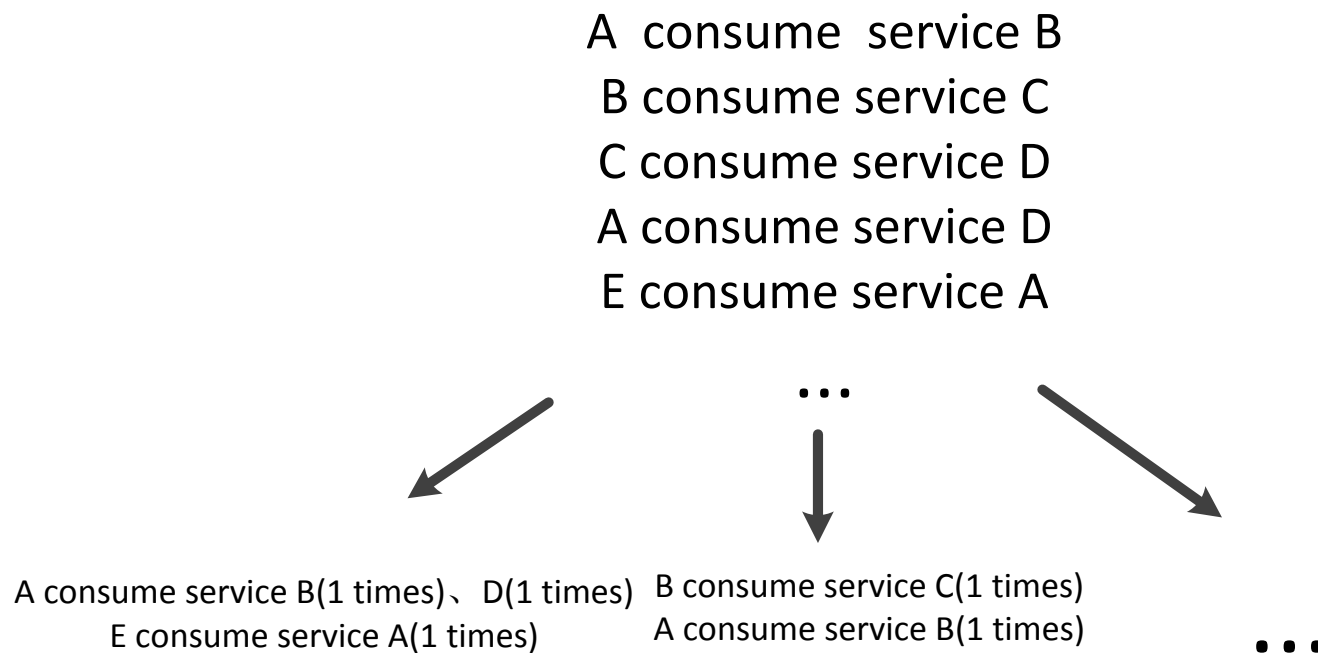
分布式SOA环境下系统的依赖错综复杂，同一个应用即可能是服务者也可能是服务消费者，作为服务提供者所提供的服务，可能被多个服务消费者调用，外部调用的时间、频次难以控制，而作为服务消费者，当前系统也可能会依赖其他第三方服务，但是第三方服务稳定性并不受服务调用方所控制，因此，如何及时发现和避免由于第三方服务不稳定而影响到当前系统，抑或是当前服务被某一第三方系统异常调用，导致整个系统瘫痪，成为亟待解决的问题，这也成为服务治理的初衷。

服务治理主要分为两个方面的内容，一个是服务的管理(服务上线下线、服务路由、服务限流和降级、服务归组、机房规则、服务授权)，另一方面即是服务相关信息的收集和展现(服务基本信息、服务质量的好坏、服务的容量和水位、服务依赖和被依赖、服务的机房分布、服务调用统计、服务提供的方法和参数、服务负责人、服务的统计报表、服务运行状态监视)，后面将主要介绍服务的稳定性管理。

依赖管理最重要的意义在于弄清楚谁调用了谁，谁被谁调用了，调用频次如何。分布式SOA架构体系的特点便是，系统高度解耦，不同的应用对外提供了大量的服务，而通过第三方的服务调用，大大提高了开发的工作效率，降低重复造轮子的几率，与此同时，作为服务提供方，又可能要依赖许多其他第三方所提供的服务，因而，最终将形成一个网状的依赖关系。



真实的依赖关系可能远比这个图复杂



通过服务消费日志，可以分析出每个服务依赖的服务，以及调用的频次，也可以分析出依赖于当前服务的应用，以及服务被调用的次数。

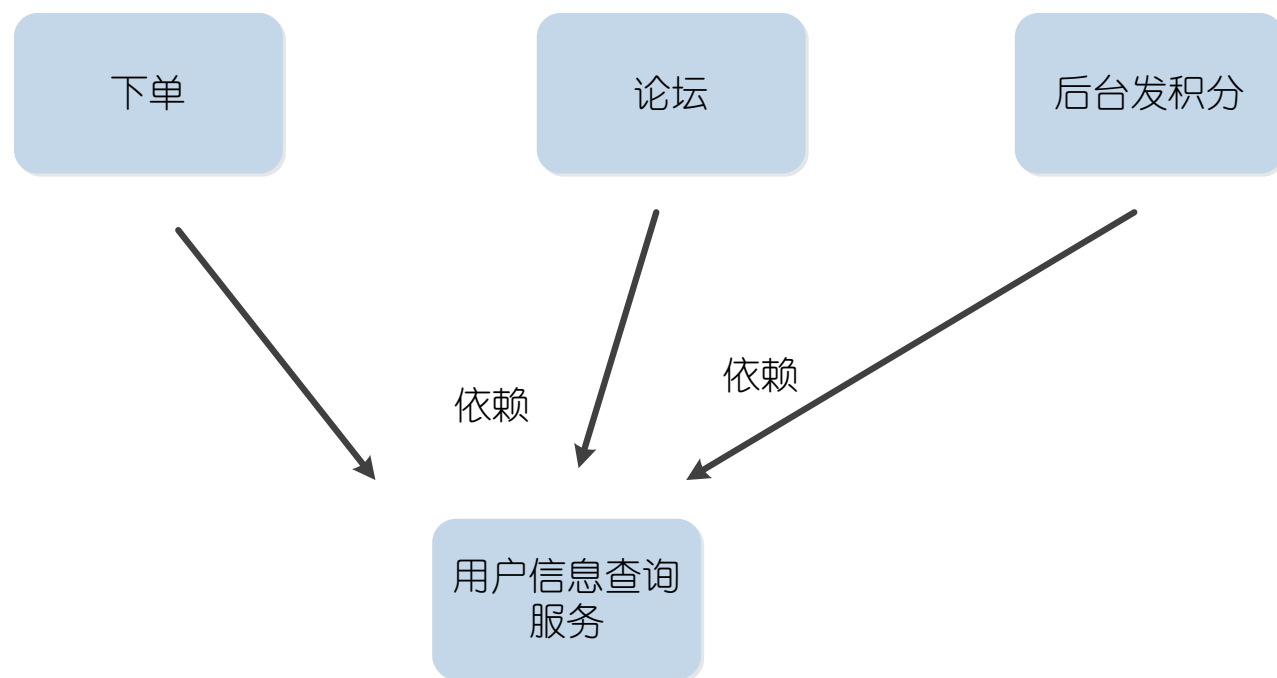
通过依赖的管理，我们能够知道，当前系统调用了哪些服务，被哪些服务调用。接下来，我们便可以根据当前系统所依赖的服务，以及系统的流程，判断依赖的服务是否会影响应用的主流程，以此来决定当前应用依赖的优先级。

对于服务提供者来说，需要清楚了解当前的服务到底被多少人调用，并建立应用白名单机制，服务调用需要事先申请，以便将调用方增加到白名单当中进行管理和容量规划。为保障系统稳定，对于未知的调用者，最好的方式便是直接拒绝，以免给系统带来不确定风险。如果没有事先的容量规划，当未知的调用者流量突增，很可能将系统拖垮。

服务提供者也需要对服务消费者的优先级进行区分，哪些调用将影响核心链路，哪些调用是非核心链路。当系统压力过大，无法承载的时候，必须优先确保重要等级高的应用，核心的调用链路优先确保畅通，而对于重要性不那么高的应用，则可以暂时先丢车保帅。

稳定性--服务分级--举例

举个例子来说，某电商网站的交易下单系统、论坛系统、后台奖金发放系统均依赖于用户信息查询服务，原因其实也很容易理解，下单需要用户的收货地址，论坛需要查询用户的等级，发放奖金需要查询用户资质。

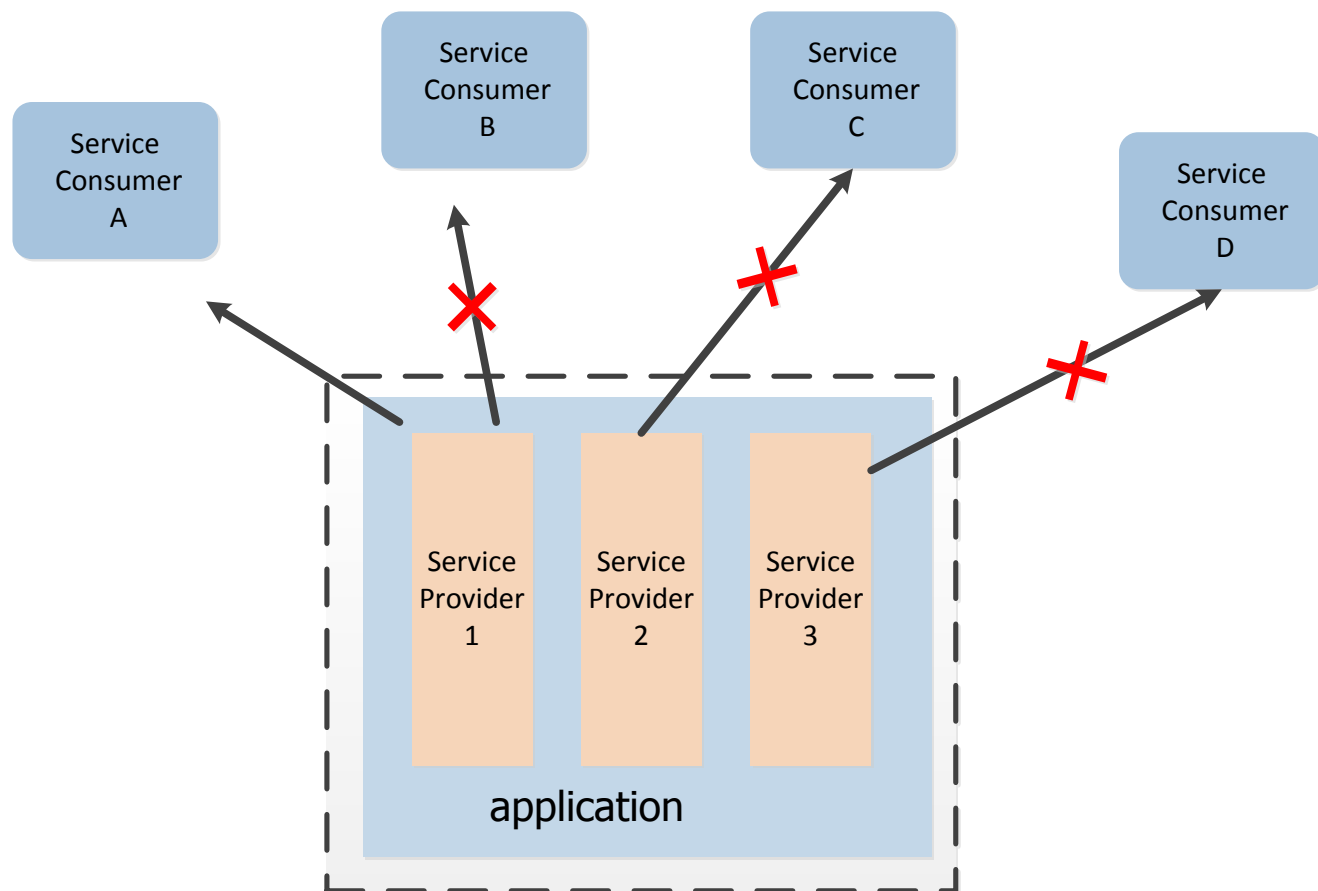


当依赖的服务出现不稳定，响应缓慢或者调用超时，或者依赖系统宕机，当前的系统需要能够及时感知到并进行相应处理，否则，大量超时的调用，有可能将当前系统的线程和可用连接数用完，导致新的请求进不来，服务僵死，这便是故障传递。如果处理不及时，故障的传递可将一个非核心链路的问题扩大，引起核心节点故障，最终形成多米诺骨牌效应，使得整个集群都不能对外提供服务。

这样，服务调用优雅降级的重要性便体现出来了。对于调用超时的非核心服务，可以设定一个阈值，如果调用超时的次数超过这个阈值，便自动将该服务降级。此时，服务调用者跳过对该服务的调用，并指定一个休眠的时间点，当时间点过了以后，再次对该服务进行重试，如果服务恢复，则取消降级，否则，继续保持该服务的降级状态，直到所依赖的服务故障恢复。这样，便可以一定程度上避免故障传递的现象发生。

当系统负载较高，即将突破警戒水位的时候，如何通过实时地屏蔽一些非核心链路的调用，降低系统的负载呢？这个时候，需要系统预先定义一些开关，来控制程序的服务提供策略。开关通过修改一些预先定义好的全局变量，来控制系统的关键路径和逻辑，比如，可以定义一个是否允许某一个级别的应用调用当前服务的开关，当系统处于流量高峰期的时候，将非核心链路的调用屏蔽，等高峰期过去之后，再将相应的开关打开。

当然，同一个应用，可能也会对外提供多个服务，如果服务耗费系统资源较多，且又不影响系统核心链路，这时，也可以将一些非核心的服务关闭掉，以减轻系统的负担，有效的提高系统对核心应用的服务能力。



服务消费者A和B依赖于服务提供者1，服务消费者C依赖于服务提供者2，服务消费者D则依赖于服务提供者3，服务提供者1、2、3则是部署在同一个应用当中，服务消费者A关联到核心链路，优先级最高，则当系统负载超过警戒水位时，为了保障对服务消费者A的服务稳定，可以暂且先通过开关屏蔽服务消费者B对服务提供者1的调用，并且视情况再通过开关，关闭服务提供者2和服务提供者3对外提供的服务，以降低应用的负载。

服务运行运行期间，需要对服务器相应指标进行监控，如系统load、磁盘利用率、内存占用率、网络流量、qps/tps等等

对于服务的调用，需要有统计的报表，按照小时/日/周/月 展示，并且能够设置异常情况监控，如流量突增突降，系统要能够及时报警。

紧急情况(如双十一、双十二、热点事件等)并不是时时刻刻都发生,大部分人在第一次面对突发事件时,难免会显得手足无措。因此,要想在系统出现故障的情况下,能够处变不惊,沉着应对,将损失降到最低,首先得准备一份应急预案,并且,得进行经常性的故障演练,以熟悉各种情况下对应的应急预案的操作流程和规范,避免紧急情况下错误的决策致使损失扩大,并且在实际操作中也能够积累经验。

应急预案中需要明确的规定服务的级别,梳理清楚核心应用的调用链路,对于每一种故障,都做出合理的假设,并且有针对性的处理方法,对于级别低的调用和功能,事先准备好屏蔽的开关和接口。

SOA架构将公共的业务拆分出来，形成可共用的服务，最大程度的保障了代码和逻辑的复用，避免了系统的重复建设，并且让应用程序的部署找到了一种持续可扩展的方案，给应用抗负载能力带来了质的飞跃。

SOA架构所面临的一大问题就是如何解决集成服务应用普遍存在的一致性问题，举例来说，同时调用多个服务，当其中一个服务调用失败时，其他服务已经处理执行的结果该如何进行回滚，这在单机本地调用的情况下使用事务比较好处理，而分布式环境下的事务将问题复杂化，并且性能开销难以承受，因此，只有在极端情况下才会考虑强一致性，一般情况下更多的关注最终一致性。另外一个就是安全问题，面向企业的平台级的SOA架构，需要对参数传递、响应内容以及各种用户私有信息的交互，有着更严格的且特殊的安全需求，如何构建一个安全的SOA架构体系，也给技术人员带来了很大的挑战。

Thanks

FAQ时间