从类库到服务

- ## 引入微服务产生的问题
  - **服务发现：**对消费者来说，外部的服务由谁提供？具体在什么网络位置？
  - **服务的网关路由：**对生产者来说，内部哪些服务需要暴露？哪些应当隐藏？应当以何种形式暴露服务？以什么规则在集群中分配请求？
  - **服务的负载均衡：**对调用过程来说，如何保证每个远程服务都接收到相对平均的流量，获得尽可能高的服务质量与可靠性？
- ## 服务发现
  - **背景：**所有的远程服务调用都使用全限定名（代表了网络中某台主机的精确位置）、端口号（代表了主机上提供了 TCP/UDP 网络服务的程序）与服务标识（代表了该程序所提供的某个具体的方法入口）来确定远程服务的精确坐标。全限定名、端口号的含义对所有的远程服务来说都一致，而服务标识则与具体的应用层协议相关，如 REST 的标识是 URL 地址，RMI 的标识是 Stub 类中的方法。
  - **分类：**远程服务标识的多样性，决定了服务发现也可以有两种不同的理解。一种是以 UDDI 为代表的的服务发现、另一种是类似于 DNS的服务发现。
  - **历史发展**
    - 服务发现原本只依赖 DNS 将全限定名翻译为一至多个 IP 地址或其他类型的记录，位于 DNS 之后的负载均衡器也实质上承担了一部分服务发现的职责，完成了外部 IP 地址到各个服务内部实际 IP 的转换。
    - 随着微服务的逐渐流行，服务的非正常宕机、重启和正常的上线、下线变得越发频繁，仅靠着 DNS 服务器和负载均衡器等无法跟上服务变动的步伐了。
    - 尝试使用 ZooKeeper 这样的分布式 K/V 框架，通过软件自身来完成服务注册与发现，是微服务早期的主流选择，但毕竟 ZooKeeper 是很底层的分布式工具，用户自己还需要做相当多的工作才能满足服务发现的需求。
    - 到后面的Eureka、Nacos等，服务发现框架发展得相当成熟，不仅支持通过 DNS 或者 HTTP 请求进行符号与实际地址的转换，还支持多种服务健康检查方式，支持集中配置、K/V 存储、跨数据中心的数据交换等多种功能。
    - 云原生时代来临，基础设施的灵活性得到大幅度的增强，最初的使用基础设施来透明化地做服务发现的方式又重新被人们所重视。
  - **可用与可靠**
    - **服务发现操作**
      - **服务的注册Service Registration：**当服务启动的时通过某些形式将自己的坐标信息通知到服务注册中心，分为自注册模式（由应用程序本身来完成）和第三方注册模式（由容器编排框架或第三方注册工具来完成）。
      - **服务的维护Service Maintaining：**服务发现框架必须维护服务列表的正确性，以避免告知服务的坐标后，服务却不能使用的情况。现在的服务发现框架，往往都能支持多种协议（HTTP、TCP 等）、多种方式（长连接、心跳、探针、进程状态等）去监控服务是否健康存活，将不健康的服务自动从服务注册表中剔除。
      - **服务的发现Service Discovery：**特指消费者从服务发现框架中，把一个符号转换为服务实际坐标的过程，一般通过 HTTP API 请求或DNS Lookup 操作来完成。
    - **服务发现的重要性和处理**
      - 注册中心被所有其他服务共同依赖，是系统中最基础的服务，几乎不可能在业务层面进行容错。服务注册中心一旦崩溃，整个系统都不再可用。实际上服务注册中心都是以集群的方式进行部署的，通常使用三个到七个节点（更多了会导致日志复制的开销太高）来保证高可用。
    - **可用和可靠的矛盾**
      - 期望服务注册中心一直可用永远健康的同时，也能够在访问每一个节点中都能取到可靠一致的数据，而不是从注册中心拿到的服务地址可能已经下线，这两个需求就构成了 CAP 矛盾，不可能同时满足。
    - **Eureka选择优先保证高可用性AP**
      - **节点间采用异步复制来交换服务注册信息**，新服务注册时不等待信息同步，而是马上在该服务发现节点宣告服务可见，但不保证在其他节点上多长时间后才会可见。
      - **旧的服务发生下线或者断网时由超时机制来控制何时从服务注册表中移除**，变动信息不会实时同步给所有服务端与客户端。使得不论是服务端还是客户端，都能够持有自己的服务注册表缓存，并以 TTL机制来进行更新，哪怕服务注册中心完全崩溃，客户端在仍然可以维持最低限度的可用。
      - Eureka对节点关系相对固定，服务不会频繁上下线的系统很合适，以较小的同步代价换取了最高的可用性。且即使客户端拿到了已经发生变动的错误地址，也**能够通过 Ribbon 和 Hystrix 模块配合来兜底**，实现故障转移或者快速失败。
    - **Consul选择优先保证高可靠性CP**
      - 采用Raft 算法，**要求多数派节点写入成功后服务的注册或变动才算完成**，严格地保证了在集群外部读取到的服务发现结果必定是一致的。
      - 同时采用 Gossip 协议，**支持多数据中心之间更大规模的服务同步**。
      - Consul 优先保证高可靠性基于它没有着全家桶式的微服务组件，万一从服务发现中取到错误地址，就没有其他组件为它兜底了。
    - **CP和AP的选择**
      - **场景：**假设系统形成了 A、B 两个网络分区后，A/B区的服务只能从区域内的服务发现节点获取到 A/B 区的服务坐标，这对你的系统会有什么影响？
      - **AP：**假设 A、B 是不同的机房，由于网络交换机导致服务发现集群出现分区问题，但每个分区中的服务仍然能独立提供完整且正确服务，但网络分区在事实上避免了跨机房的服务请求，反而还带来了服务调用链路优化的效果。
      - **CP：**譬如系统中大量依赖了集中式缓存、消息总线、或者其他有状态的服务，一旦这些服务全部或者部分被分隔到某一个分区中，会对整个系统的操作的正确性产生直接影响，干脆直接停机避免生成错误数据。
  - **注册中心实现**

- **在分布式 K/V 存储框架上自己开发服务发现**，如ZooKeeper、Doozerd、Etcd。Etcd/Raft算法和Zookeeper/Multi Paxos派生算法决定了这类型服务发现都是CP的，基础的能力如服务如何注册、如何做健康检查，等等都必须自己去实现。
- **以基础设施主要指 DNS 服务器来实现服务发现**，如SkyDNS、CoreDNS。K8S 1.11 版，采用的CoreDNS，是 CP 还是 AP 取决于后端采用的存储，如基于 Etcd 实现CP，基于内存异步复制的方案实现AP。以基础设施来做服务发现，好处是对应用透明，任何语言、框架、工具都支持 HTTP、DNS ，不受技术选型约束；坏处是必须考虑如何去做客户端负载均衡、如何调用远程方法等这些问题，且必须遵循或者说受限于这些基础设施本身所采用的实现机制。
- **专门用于服务发现的框架和工具**，如Eureka、Consul 和 Nacos。它们可以被应用程序感知，所以还需要考虑所用的程序语言技术框架的集成问题

网关路由

- **存在原因：**微服务架构下，每个服务节点都可能由不同团队负责，有着独立互不相同的接口，如果服务集群缺少统一对外交互的代理人角色，那外部的服务消费者就必须知道所有微服务节点在集群中的精确坐标，导致消费者受到网络限制（不能确保每个节点都有外网连接）、安全限制（服务节点的安全，外部受到如浏览器同源策略的约束）、依赖限制（服务坐标这类信息随时可能变动，不应该依赖它）。
- **职责：**微服务中网关的首要职责是作为统一的出口对外提供服务，将外部访问网关地址的流量，根据适当的规则路由到内部集群中正确的服务节点之上，因此，微服务中的网关，也常被称为“服务网关”或者“API 网关”。在此基础上还可以根据需要作为流量过滤器来使用，提供某些额外的可选的功能。
- **关注点**
  - **网络协议层次**
    - **定义：**指负载均衡中介绍过的四层流量转发与七层流量代理
    - **从技术实现角度来看：**对于路由这项工作，负载均衡器与服务网关在实现上是没有什么差别的，很多服务网关本身就是基于老牌的负载均衡器来实现的。
    - **从目的角度看：**负载均衡器是为了根据均衡算法对流量进行平均地路由，服务网关是为了根据流量中的某种特征进行正确地路由。
    - **网关必须能够识别流量中的特征，即网关能够支持的网络通信协议的层次将会直接限制后端服务节点能够选择的服务通信方式**。如果服务集群只提供像 Etcd 这样直接基于 TCP 的访问的服务，那只部署四层网关便可，网关以 IP 报文中源地址、目标地址为特征进行路由；如果服务集群要提供 HTTP 服务的话，那就必须部署一个七层网关，网关根据 HTTP 报文中的 URL、Header 等信息为特征进行路由；如果服务集群还要提供更上层的 WebSocket、SOAP 等服务，那就必须要求网关同样能够支持这些上层协议，才能从中提取到特征。
  - **性能与可用性**
    - **影响因素：**网关的性能与工作模式和自身实现算法都有关系，但工作模式是最关键的因素，若采用 DSR 三角传输模式，原理上决定了性能一定会比代理模式来的强
    - **与网络I/O模型关系：**由于REST 和 JSON-RPC 等基于 HTTP 协议的服务接口在对外部提供的服务中占主流的地位，所以服务网关默认都必须支持七层路由，通常就默认无法直接进行流量转发，只能采用代理模式。在这个前提约束下，网关的性能主要取决于它们如何代理网络请求，也即它们的网络 I/O 模型。
  - **网络I/O模型**
    - **定义：**在套接字接口抽象下，网络 I/O 的出入口就是 Socket 的读和写，Socket 在操作系统接口中被抽象为数据流，网络 I/O 可以理解为对流的操作。当发生一次网络请求发生后，将会按顺序经历“等待数据从远程主机到达缓冲区”和“将数据从缓冲区拷贝到应用程序地址空间”两个阶段。
    - **异步I/O：**数据到达缓冲区后，不需要由调用进程主动进行从缓冲区复制数据的操作，而是复制完成后由操作系统向线程发送信号，所以它一定是非阻塞的。
    - **同步I/O**
      - **阻塞 I/O：**在数据没有到达应用程序地址空间时线程休眠，即线程被挂起。阻塞 I/O 是最直观的 I/O 模型，逻辑清晰，比较节省 CPU 资源，但缺点就是线程休眠所带来上下文切换，这是一种需要切换到内核态的重负载操作，不应当频繁进行。
      - **非阻塞 I/O：**非阻塞 I/O 能够避免线程休眠，线程每隔一段时间去检查数据是否到达应用程序地址空间，对于一些很快就能返回结果的请求，非阻塞 I/O 可以节省切换上下文切换的消耗，但是对于较长时间才能返回的请求，非阻塞 I/O 反而白白浪费了 CPU 资源，所以目前并不常用。
      - **多路复用 I/O：**多路复用 I/O 本质上是阻塞 I/O 的一种，但它可以在同一条阻塞线程上处理多个不同端口的监听。当某个端口的数据到达后，即马上通知它，然后继续监听其他端口，多路复用 I/O 是目前的高并发网络应用的主流。
      - **信号驱动 I/O：**信号驱动 I/O 与异步 I/O 的区别是“从缓冲区获取数据”这个步骤的处理，前者收到的通知是可以开始进行复制操作了，在复制完成之前线程处于阻塞状态，所以它仍属于同步 I/O 操作，而后者收到的通知是复制操作已经完成。
  - **网关可用性**
    - 由于网关的地址具有唯一性，就不像之前服务发现那些注册中心那样直接做个集群
    - 网关应尽可能轻量，尽管网关作为服务集群统一的出入口，可以很方便地做安全、认证、授权、限流、监控，等等的功能，但给网关附加这些能力时还是要仔细权衡，取得功能性与可用性之间的平衡，过度增加网关的职责是危险的。
    - 网关选型时，应该尽可能选择较成熟的产品实现，譬如 Nginx Ingress Controller、KONG、Zuul 这些经受过长期考验的产品，而不能一味只考虑性能选择最新的产品，性能与可用性之间的平衡也需要权衡。
    - 在需要高可用的生产环境中，应当考虑在网关之前部署负载均衡器或者等价路由器（ECMP），让那些更成熟健壮的设施（往往是硬件物理设备）去充当整个系统的入口地址，这样网关也可以进行扩展了。

客户端负载均衡

- **背景：**以前负载均衡器大多只部署在整个服务集群的前端，将用户的请求分流到各个服务进行处理，即集中式的负载均衡。随着微服务日渐流行，越来越多的访问请求是由集群内部的某个服务发起，由集群内部的另一个服务进行响应的，对于这类流量的负载均衡，针内部流量的特点，直接在服务集群内部消化掉，是更合理的。它与服务端负载均衡器的关键差别所在：客户端均衡器是和服务实例一一对应的，而且与服务实例并存于同一个进程之内。
- **客户端负载均衡优势**
  - 均衡器与服务之间信息交换是进程内的方法调用，不存在任何额外的网络开销。
  - 不依赖集群边缘的设施，所有内部流量都仅在服务集群的内部循环，避免了出现集群内部流量要“绕场一周”的尴尬局面。
  - 分散式的均衡器意味着天然避免了集中式的单点问题，它的带宽资源将不会像集中式均衡器那样敏感，这在以七层均衡器为主流、不能通过 IP 隧道和三角传输这样方式节省带宽的微服务环境中显得更具优势。
  - 客户端均衡器要更加灵活，能够针对每一个服务实例单独设置均衡策略等参数，访问某个服务，是不是需要具备亲和性，选择服务的策略是随

机、轮询、加权还是最小连接等等，都可以单独设置而不影响其它服务。

- **客户端负载均衡缺陷**
  - 它与服务运行于同一个进程之内，意味着它的选型受到服务所使用的编程语言的限制，要为每种语言都实现对应的能够支持复杂网络情况的均衡器是非常难的。
  - 从个体服务来看，由于是共用一个进程，均衡器的稳定性会直接影响整个服务进程的稳定性，消耗的资源也同样影响到服务的可用资源。从集群整体来看，在服务数量达成千乃至上万规模时，客户端均衡器消耗的资源总量是相当可观的。
  - 由于请求的来源可能是来自集群中任意一个服务节点，而不再是统一来自集中式均衡器，这就使得内部网络安全和信任关系变得复杂，当攻破任何一个服务时，更容易通过该服务突破集群中的其他部分。
  - 服务集群的拓扑关系是动态的，每一个客户端均衡器必须持续跟踪其他服务的健康状况，以实现服务队列维护。由于这些操作都需要通过访问服务注册中心来完成，数量庞大的客户端均衡器一直持续轮询服务注册中心，也会为它带来不小的负担。
- **代理负载均衡器**
  - **定义：**对此前的客户端负载均衡器的改进是将原本嵌入在服务进程中的均衡器提取出来，作为一个进程之外，同一 Pod 之内的特殊服务，放到边车代理中去实现。
  - 代理均衡器与服务实例不再是进程内通信，而是通过网络协议栈进行数据交换的，数据要经过操作系统的协议栈，要进行打包拆包、计算校验和、维护序列号等网络数据的收发步骤，流量比起之前的客户端均衡器确实多增加了一系列处理步骤。
  - **优势**
    - 代理均衡器不再受编程语言的限制。集中不同编程语言的使用者的力量，更容易打造出能面对复杂网络情况的、高效健壮的均衡器。且独立于服务进程的均衡器也不会由于自身的稳定性影响到服务进程的稳定。
    - 在服务拓扑感知方面代理均衡器也要更有优势。由于边车代理接受控制平面的统一管理，当服务节点拓扑关系发生变化时，控制平面就会主动向边车代理发送更新服务清单的控制指令，这避免了此前客户端均衡器必须长期主动轮询服务注册中心的浪费。
    - 在安全性、可观测性上，由于边车代理都是一致的实现，有利于在服务间建立双向 TLS 通信，也有利于对整个调用链路给出更详细的统计信息。