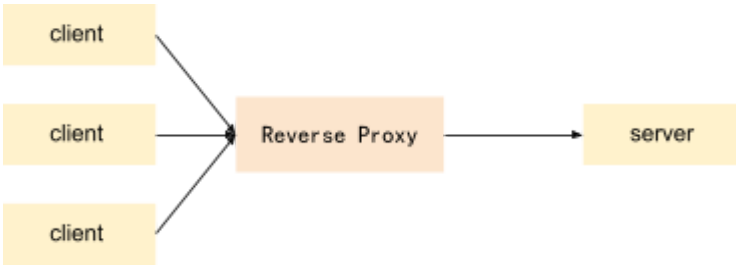
API 网关选型与能力分析

引言

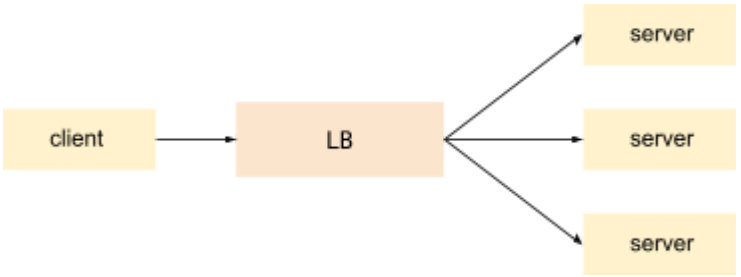
无论是南北向的流量还是东西向的流量，大多数情况下都是通过 API 去进行访问的，那么随着业务的发展以及整个移动互联网的这种发展，API 的规模是越来越大的，那么在每一个高速发展的一些企业当中它所提供的这些服务也会有很多很多的 API，尤其是包括现在的微服务架构的流行，导致这种 API 规模的指数上涨，那么 API Gateway 在这个时候它就产生了。

反向代理



反向代理路由到后端真正的 upstream 服务器上面，所有的 client 是不知道真正提供服务的那个 server 到底是哪个和 server 的 ip 是哪个，只知道它当前连接的反向代理的地址
隐藏它们的 server 更多的是做选型，后端放 was 防火墙接入 ddos 防护，过滤流量的影响和攻击可以做请求的转发和请求的改写，但 upstream 所实现的 api 都不一定实现的都是这样的路径，有可能后端实现的是 user 有可能是 v1。

Load Balancer



client 请求了一个 api，然后请求的这个 api 先到达的不是真正的 upstram 而是某一个代理服务器，代理服务器将请求路由到真正的 upstream 上面，LB 主要是分摊压力，最常见是这种 IP Hash，NGINX 可以做反向代理也可以去做 LB，是因为集成到了一个软件当中。

反向代理、负载均衡与开源网关

在架构层面上来讲，反向代理和 Load Balancer 的架构整体而言是一样的。但是它们之间是有不同的区别的。因为对反向代理 和 Load Balancer 我们的预期是不同的。

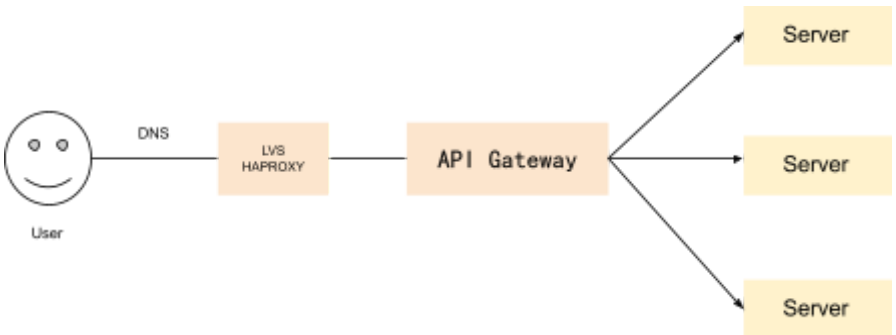
从技术的实现角度来看，envoy 它是 C++ 写的，NGINX 它的底层是 C，像这个自研的话，一种是 Golang，一种是 Java，整体而言，像 Java 的无论说它的这个内存消耗还是说它的性能上其实相对来说会弱势一些，如果是完全自研的话，然后没有去基于这种比较成熟的这些项目的话，那它的这个稳定性其实相对来说会有一定的缺失，但也并不是所有自研都不靠谱。

NGINX 使用的比例会相对来说比较高，所以这个东西相对来说是久经考验的，像 enovy 它是在这种云原生的这些项目当中是比较出风头的，你想如果是去做这个 Service Mesh，比如说基于像 istio 的这种它有多少 Pod 它就得有多少 Sidecar 的这个 Proxy，那它就得有多少 enovy 实例在上面。

那相对来说这些组件当中最靠谱的、最久经考验的其实是 NGINX，这个方案会更稳定一些。

Apache APISIX 就是基于 NGINX + Lua；Kong 也走一个这样的路，它们有一个比较大的区别。 Kong 它使用的是 PostgreSQL 来做数据存储，它是一个关系型的数据库，但是 Apache APISIX 使用的是 etcd 作为数据存储。

多层网关架构



对于去做这种网关的这种选型，无论是去做这种API gateway的这种选型；还是说去做一个纯粹的这种网关的选型；还是去做这种单纯的代理的这种选型；通常的这种架构设计方式，比如用户去请求的一个域名，然后先在的域名解析上面先去做一些一定的优化，比如说去做这种DNS解析的这种线路优化，不同地区的用户可以去解析到不同的节点上去，然后通过这种方式去做一定的负载均衡。

再有一种就是说在DNS这一层上面去做一些可以做的一些优化，那就在四层上面会选择一个性能特别好的东西来去扛一些负载，比较常见的就是比如说用 LVS 去做这种四层的代理。

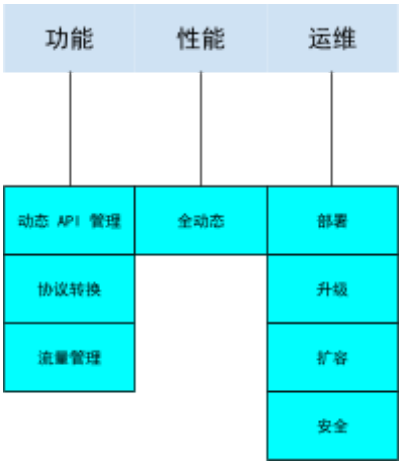
你可能也会去选择一些 HAProxy，之后在后面就是我们到了我们的这种七层上面，到我们七层上面的话，去选择一个网关或者选择一个API Gateway 来去做这种七层的这种流量的负载。

或者去做这种整个路由的网关的管理，这都是一种比较常规的、比较常见的一种设计，至于说在这一层的这个API Gateway当中怎么去选这是另外一个问题了，但是整体来说去做这么一套方案，大致的情况基本上都是这样子。

唯一的区别就是在不同的位置能做的事情和能做的优化有哪些，比如只是去做这种DNS的配置，但不去做这种线路的优化，不去考虑说不同的这种来源，你要走什么样的线路，然后你可能不去考虑这个东西的话和你考虑了这个东西之后的这个效率就完全不一样。

然后再有就是说你在这个四层上面，如果你直接用LVS、HAProxy其实也可以，现在可能会加一些其他的技术，比如说用 DPDK、eBPF 一些内核层的一些优化再有就是在七层上面可能还得配合着你去做一些内核参数的一些调整之类的事情。

API 网关的能力要求与设计考量



动态 API 管理

第一点就是API的管理，API的这个管理，通过服务的主动的注册，或者说人工的去注册，手动的去注册，这取决于具体的一个实现方式，但是需要具备API管理的能力，另外一种方式就是建议的是说如果去做一个API Gateway，建议去把这个API管理的这项能力做成一个动态的，就是允许动态的去添加或者删除API做成动态的。

协议转换

其次就是关于它的这种协议的转换，因为你是一个API Gateway，你相当于是流量的入口，那你既然是流量的入口，那免不了会跟各种服务去进行交互，它可能会是一种异构的服务，那你就是需要能够比如说通过协议的转换，比如说就 gRPC、Dubbo、HTTP 之类的这种协议之间的互相转换，这样子能够去做，这种协议转换才能够更好的跟这种整个这种异构的这种架构去更好的去结合去使用。

流量管理

其实希望说它能够把一些统一的一些能力都集中到API Gateway当中，然后API Gateway当中的话，比如说像整个的这种微服务架构下面的话，常提到的限流熔断，这都是一种是为了能够保护不受损耗，不受一些攻击。及时的止损，给用户有一个比较好的一种体验。

还有就是这种需要下沉的能力，包括我们这种认证授权之类的这种能力也是需要同样的把它给下沉到这种API Gateway当中的。

可观测性

再有就是像这种监控报警，其实也算是这种基础能力，因为毕竟所有的流量都是从这个口走的，而且是比较容易知道说这种上游的这些 upstream 服务器，它到底返回的是 200 还是 500，还是其他的一些这是关于功能上面所需要考虑的点。

全动态能力

关于性能的话，API Gateway会存在这种热点的问题，就是流量热点的问题，就是它一定会有这种API，它就一定会有这种流量热点的问题，所以它的性能一定要足够的好，而且要能够全动态，因为它如果不能全动态的话它如果遇到需要reload或者说需要重启，那其实就对业务流量是比较伤害比较大的。

可运维性

如果去做了一个API Gateway，想要去真正把这个东西放到生产上面的话，那其实一定要考虑到它的这个可运维的能力，可运维的能力包括说它是否易于部署，它的这个扩容是不是比较方便的，可以去做这种扩容，动态的扩容，然后以及说你说升级的时候，是不是可以做到这种零宕机就是对业务完全无损害的这种升级，还要就是怎么维护，升级就是维护的一种了，还有安全，就是说这个东西它的这个攻击面是不是很广，因为它毕竟是整个集群或者说整个的所有流量的一个入口点，如果你的这个入口点直接被人给攻破了，那后面的服务它怎么能够保证它的安全，所以它的安全性一定是需要考虑到的。