gRPC (/tags/#gRPC) 负载均衡 (/tags/#负载均衡)

基于 gRPC 的服务发现与负载均衡（基础篇）

gRPC 负载均衡架构分析

Posted by pandaychen on July 11, 2019

0x00 前言

在后台服务开发中，高可用性是构建中核心且重要的一环。服务发现（Service discovery）和负载均衡（Load Balance）一直都是我关注的话题。今天来谈一下我在实际中是如何理解及落地的。

0x01 负载均衡 && 服务发现

基础

负载均衡，顾名思义，是通过某种手段将流量 / 请求分配到不通的服务器上去，保证后台的每个服务收到的请求都尽可能保持平衡

服务发现，就是指客户端按照某种约定的方式主动去（注册中心）寻找服务，然后再连接相应的服务

关于负载均衡的构建与实现，可以看下这几篇文章：

- gRPC 服务发现 & 负载均衡 (https://segmentfault.com/a/1190000008672912)
- gRPC Load Balancing (https://gRPC.io/blog/loadbalancing/)
- Load Balancing in gRPC (https://github.com/grpc/grpc/blob/master/doc/load-balancing.md)

服务发现概念

我们说的服务发现，一般理解为客户端如何发现 (并连接到) 服务，这里一般包含三个组件：

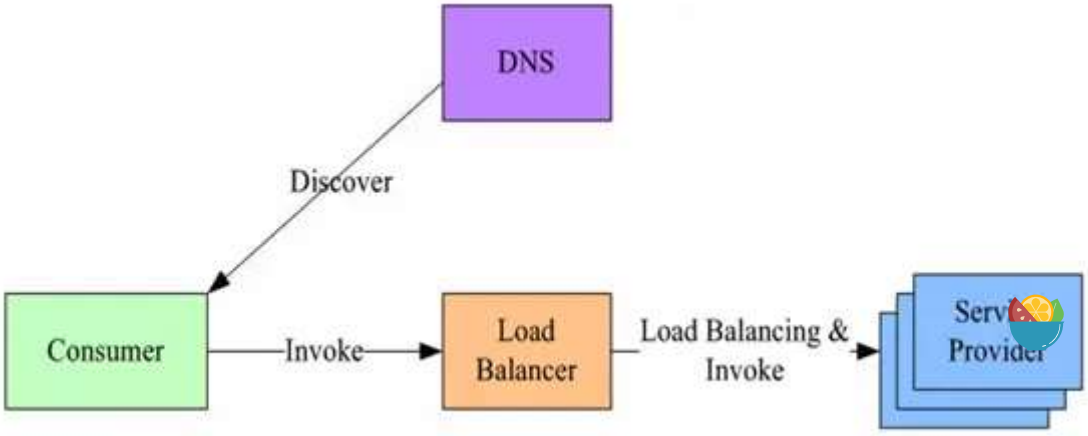
1. 服务消费者：一般指客户端（可以是简单的 TCP-Client 或者是 RPC-Client ）
2. 服务提供者：一般指服务提供方，如传统服务，微服务等
3. 服务注册中心：用来存储（Key-Value）服务提供者的服务，一般以 DNS/HTTP/RPC 等方式对外暴露接口

负载均衡概念

我们把 LB 看作一个组件，根据组件位置的不同，大致上分为三种：

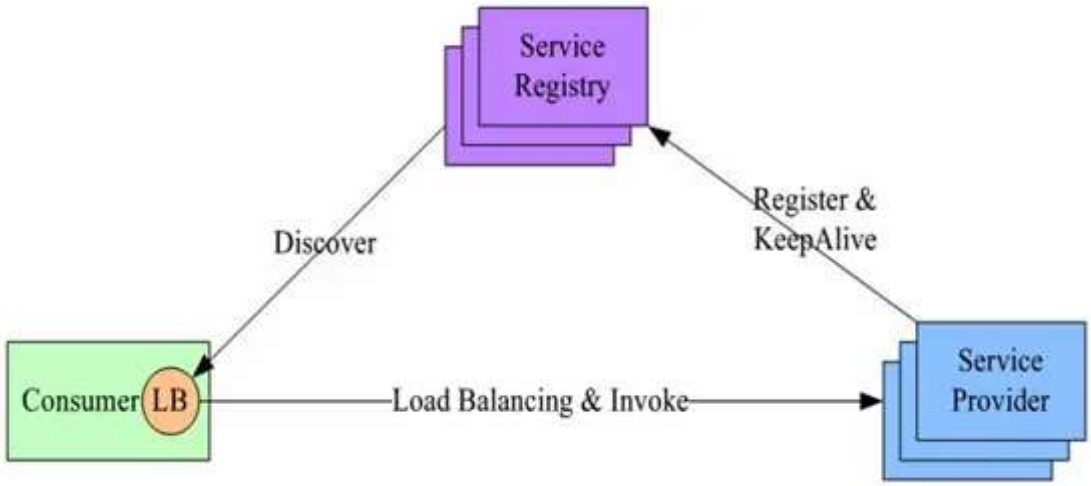
集中式 LB（Proxy Model）

独立的 LB, 可以是硬件实现，如 F5，或者是 nginx 这种内置 Proxy-pass 或者 upstream 功能的网关，亦或是 LVS/HAPROXY，之前也使用 DPDK (http://core.dpdk.org/doc/quick-start/) 开发过类似的专用网关。



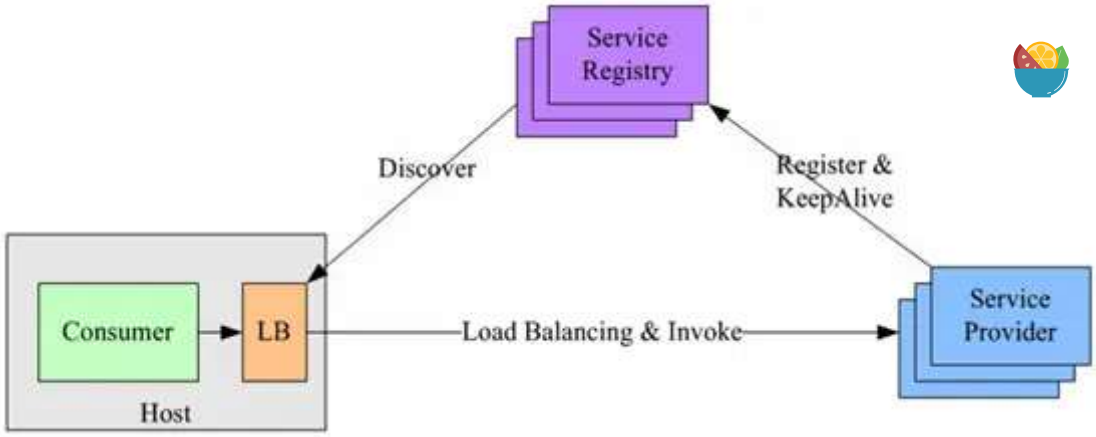
进程内 LB（Balancing-aware Client）

进程内 LB（集成到客户端），此方案将 LB 的功能集成到服务消费方进程里，也被称为软负载或者客户端负载方案。服务提供方启动时，首先将服务地址注册到服务注册表，同时定期报心跳到服务注册表以表明服务的存活状态，相当于健康检查，服务消费方要访问某个服务时，它通过内置的 LB 组件向服务注册表查询，同时缓存并定期刷新目标服务地址列表，然后以某种负载均衡策略选择一个目标服务地址，最后向目标服务发起请求。LB 和服务发现能力被分散到每一个服务消费者的进程内部，同时服务消费方和服务提供方之间是直接调用，没有额外开销，性能比较好。



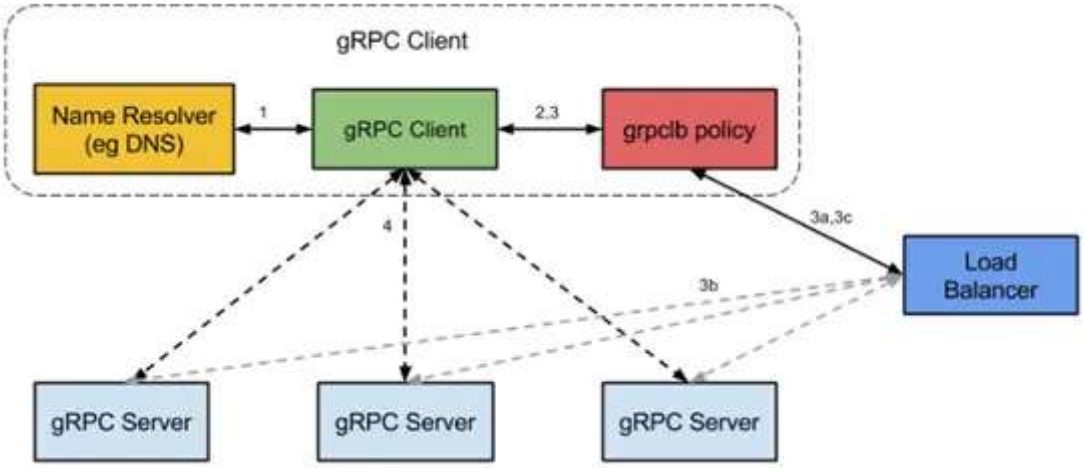
独立 LB 进程（External Load Balancing Service）

该方案是针对上一种方案的不足而提出的一种折中方案，原理和第二种方案基本类似。不同之处是将 LB 和服务发现功能从进程内移出来，变成主机上的一个独立进程。主机上的一个或者多个服务要访问目标服务时，他们都通过同一主机上的独立 LB 进程做服务发现和负载均衡。该方案也是一种分布式方案没有单点问题，一个 LB 进程挂了只影响该主机上的服务调用方，服务调用方和 LB 之间是进程内调用性能好，同时该方案还简化了服务调用方，不需要为不同语言开发客户库，LB 的升级不需要服务调用方改代码。 公司的 L5 是这种方式，每台机器上都安装了 L5 的 agent，供其他服务调用。该方案主要问题：部署较复杂，环节多，出错调试排查问题不方便。



gRPC 内置的方案

gRPC 的内置方案如下图所示：

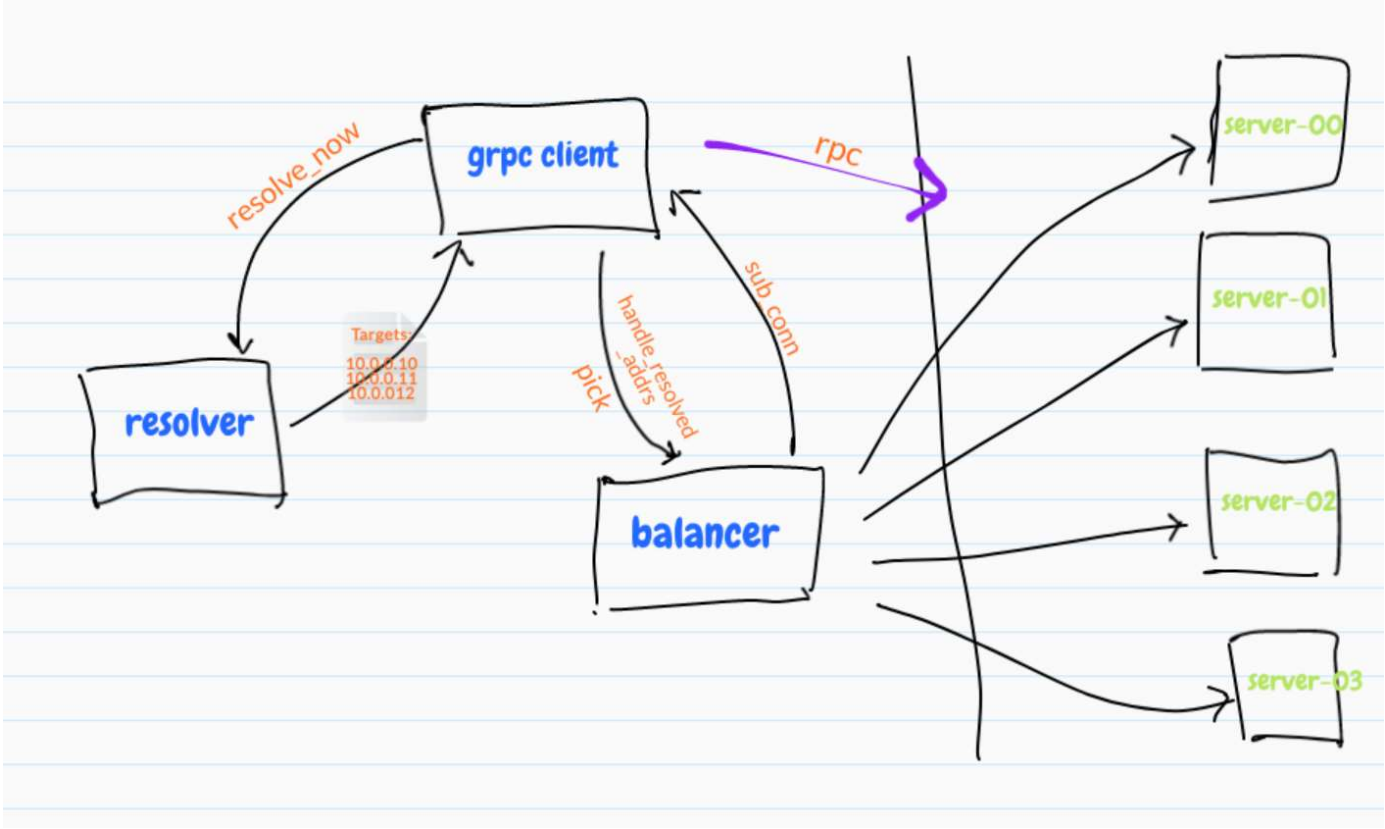


gRPC 在官网文档中提供了实现 LB 的思路，并在不同语言的 gRPC 代码 API 中已提供了命名解析和负载均衡接口供扩展。默认提供了 DNS-resolver 的实现 (https://github.com/gRPC/gRPC-go/blob/v1.8.0/resolver/resolver.go)，接口相当规范，实现起来也不复杂，只需要实现服务注册（Registry）和服务监听 + 解析（Watcher+Resolver）的逻辑就行了，这里简单介绍其基本实现过程：

1. 构建注册中心，这里注册中心一般要求具备分布式一致性（满足 CAP 定理的 AP 或 CP）的高可用的组件集群，如 Zookeeper、Consul、Etcd 等
2. 构建 gRPC 服务端的注册逻辑，服务启动后定时向注册中心注册自身的关键信息（一般开启新的 groutine 来完成），至少包含 IP 和端口，其他可选信息，如自身的负载信息（CPU 和 Memory）、当前实时连接数等，这些辅助信息有助于帮助系统更好的执行 LB 算法
3. gRPC 客户端向注册中心发出服务解析请求，注册中心将请求中关联的所有服务的信息返回给 gRPC 客户端，客户端与所有在线的服务建立起 HTTP2 长连接
4. gRPC 客户端发起 RPC 调用，根据 LB 均衡器中实现的负载均衡策略（gRPC 中默认提供的算法是 RoundRobin），选择其中一 HTTP2 长连接进行通信，即 LB 策略决定哪个子通道 - 即哪个 gRPC 服务器将接收请求

0x02 gRPC 负载均衡的运行机制

gRPC 提供了负载均衡实现的用户侧接口，我们可以非常方便的定制化业务的负载均衡策略，为了理解 gRPC 的负载均衡的实现机制，后续博客中我会分析下 gRPC 实现负载均衡的代码。



1. Resolver
    - 解析器，用于从注册中心实时获取当前服务端的列表，同步发送给 Balancer
2. Balancer
    - 平衡器，一是接收从 Resolver 发送的服务端列表，建立并维护（长）连接状态；二是每次当 Client 发起 Rpc 调用时，按照一定算法从连接池中选择一个连接进行 Rpc 调用
3. Register
    - 注册，用于服务端初始化和在线时，将自己信息上报到注册中心，主要信息有 Ip，端口等

0x03 负载均衡的算法及实现

在实践中，如何选取负载均衡策略是一个很有趣的话题，例如 Nginx 的 `upstream` 机制中就有很多经典的 LB 策略，如带权重的轮询 Weight-RoundRobin (https://github.com/nginx/nginx/blob/master/src/http/ngx_http_upstream_round_robin.c)，一般常用的负载均衡方法有如下几种：

1. RoundRobin（轮询）
2. Weight-RoundRobin（加权轮询）
    - 不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，而配置低、负载高的机器，给其分

配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。

3. Random（随机）
4. Weight-Random（加权随机）
    - 通过系统的随机算法，根据后端服务器的列表随机选取其中的一台服务器进行访问
5. 源地址哈希法
    - 源地址哈希的思想是根据获取客户端的 IP 地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客服端要访问服务器的序号。采用源地址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时，它每次都会映射到同一台后端服务器进行访问
6. 最小连接数法
    - 最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它是根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前的请求，尽可能地提高后端服务的利用效率，将负责合理地分流到每一台服务器
7. 一致性哈希算法
    - 常见的是 `Ketama` 算法，该算法是用来解决 `cache` 失效导致的缓存穿透的问题的，当然也可以适用于 gRPC 长连接的场景
8. 自适应算法（P2C：多选二，二选一）：即从可用节点列表中随机选择两个节点，计算它们的负载率，选择负载率较低的进行请求
    - 基于最小负载策略：该策略是 linkerd 的默认负载均衡器。当确定发送请求的位置时，linkerd 随机从负载均衡器池中选择两个副本，并选择两者中最少负载的副本。负载由每个副本的未完成请求数决定。该算法为单个副本提供了可管理的负载上限，与具有相似性能的其他算法相比，开销较少
    - 峰值 EWMA（预测）策略：该算法是上述的变体，同样在发送请求时仍然在两个副本之间进行选择。不一样的是，该算法需要保持观察到的延迟的动态平均值，并且使用它来对每个副本的未完成请求的数量进行加权。这种方法对延迟波动更敏感，并通过向较慢的后端发送更少的请求来允许他们恢复时间（可以通过调参来改变对请求延时的敏感度）

0x04 gRPC 服务治理的优势

在现网环境中，后端服务就是采用了 gRPC 与 Etcd 的服务治理方案，总结下有这么几个优点；

- 采用了 gRPC 实现负载均衡策略，模块之间通信采用长连接方式，避免每次 RPC 调用时新建连接的开销，充分发挥 HTTP2 的优势
- 扩容和缩容都及其方便，例如扩容，只要部署上服务，运行后，服务成功注册到 Etcd 便大功告成
- 灵活的自定义的 LB 算法，使得后端压力更为均衡
- 客户端加入重试逻辑，使得网络抖动情况下，可以通过重试连接上另外一台服务

0x05 总结

总之，利用 gRPC 接口实现服务端的负载均衡及高可用，这套方案现网实战的效果还是很不错的，后面再写一篇如何使用 Etcd 和 gRPC 实现自定义负载均衡算法的文章。

0x06 参考

- linkerd 官方文档 - 负载均衡 (https://doczhcn.gitbook.io/linkerd/index/te-xing/load-balancing#fu-zai-jun-heng-qi-xuan-xiang)

PREVIOUS
给 JEKYLL 增加 LATEX 公式渲染
(/2019/06/18/ADD-LATEX-SUPPORT-WITH-JEKYLL/)

NEXT
证书（CERTIFICATE）的那些事
(/2019/07/24/AUTH/)

FEATURED TAGS (/tags/)

Latex (/tags/#Latex) gRPC (/tags/#gRPC) 负载均衡 (/tags/#负载均衡) OpenSSH (/tags/#OpenSSH) Authentication (/tags/#Authentication) Consul (/tags/#Consul) Etcd (/tags/#Etcd) Kubernetes (/tags/#Kubernetes) 性能优化 (/tags/#性能优化) Python (/tags/#Python) 分布式锁 (/tags/#分布式锁)

WebConsole (/tags/#WebConsole) 后台开发 (/tags/#后台开发) Golang (/tags/#Golang)
OpenSource (/tags/#OpenSource) Nginx (/tags/#Nginx) Vault (/tags/#Vault) 网络安全 (/tags/#网络安全)
Perl (/tags/#Perl) 分布式理论 (/tags/#分布式理论) Raft (/tags/#Raft) 正则表达式 (/tags/#正则表达式)
Redis (/tags/#Redis) 分布式 (/tags/#分布式) 限流 (/tags/#限流) 微服务 (/tags/#微服务)
反向代理 (/tags/#反向代理) ReverseProxy (/tags/#ReverseProxy) Cache (/tags/#Cache) 缓存 (/tags/#缓存)
连接池 (/tags/#连接池) OpenTracing (/tags/#OpenTracing) GOMAXPROCS (/tags/#GOMAXPROCS)
GoMicro (/tags/#GoMicro) 微服务框架 (/tags/#微服务框架) 日志 (/tags/#日志) Pool (/tags/#Pool)
Kratos (/tags/#Kratos) Hystrix (/tags/#Hystrix) 熔断 (/tags/#熔断) 并发 (/tags/#并发)
Pipeline (/tags/#Pipeline) 证书 (/tags/#证书) Prometheus (/tags/#Prometheus) Metrics (/tags/#Metrics)
Breaker (/tags/#Breaker) 定时器 (/tags/#定时器) Timer (/tags/#Timer) Timeout (/tags/#Timeout)
Kafka (/tags/#Kafka) Xorm (/tags/#Xorm) MySQL (/tags/#MySQL) Fasthttp (/tags/#Fasthttp)
bytebufferpool (/tags/#bytebufferpool) 任务队列 (/tags/#任务队列) 队列 (/tags/#队列) 异步队列 (/tags/#异步队列)
GOIM (/tags/#GOIM) Pprof (/tags/#Pprof) errgroup (/tags/#errgroup) consistent-hash (/tags/#consistent-hash)
Zinx (/tags/#Zinx) 网络框架 (/tags/#网络框架) 设计模式 (/tags/#设计模式) HTTP (/tags/#HTTP)
Gateway (/tags/#Gateway) Queue (/tags/#Queue) Docker (/tags/#Docker) 网关 (/tags/#网关)
Statefulset (/tags/#Statefulset) NFS (/tags/#NFS) Machinery (/tags/#Machinery) Teleport (/tags/#Teleport)
Zero Trust (/tags/#Zero Trust) Oxy (/tags/#Oxy) 存储 (/tags/#存储) Confd (/tags/#Confd)
热更新 (/tags/#热更新) OAuth (/tags/#OAuth) SAML (/tags/#SAML) OpenID (/tags/#OpenID)
Openssl (/tags/#Openssl) AES (/tags/#AES) 微服务网关 (/tags/#微服务网关) IM (/tags/#IM)
KMS (/tags/#KMS) 安全 (/tags/#安全) 数据结构 (/tags/#数据结构) hashtable (/tags/#hashtable)
Sort (/tags/#Sort) Asynq (/tags/#Asynq) 基数树 (/tags/#基数树) Radix (/tags/#Radix)
Crontab (/tags/#Crontab) 热重启 (/tags/#热重启) 系统编程 (/tags/#系统编程) sarama (/tags/#sarama)
Go-Zero (/tags/#Go-Zero) RDP (/tags/#RDP) VNC (/tags/#VNC) 协程池 (/tags/#协程池) UDP (/tags/#UDP)
hashmap (/tags/#hashmap) 网络编程 (/tags/#网络编程) 自适应技术 (/tags/#自适应技术)
环形队列 (/tags/#环形队列) Ring Buffer (/tags/#Ring Buffer) Circular Buffer (/tags/#Circular Buffer)
InnoDB (/tags/#InnoDB) timewheel (/tags/#timewheel) GroupCache (/tags/#GroupCache)
Jaeger (/tags/#Jaeger) GOSSIP (/tags/#GOSSIP) CAP (/tags/#CAP) Bash (/tags/#Bash)
websocket (/tags/#websocket) Mysql (/tags/#Mysql) GC (/tags/#GC) singleflight (/tags/#singleflight)
闭包 (/tags/#闭包) Helm (/tags/#Helm) kubernetes (/tags/#kubernetes) network (/tags/#network)
iptables (/tags/#iptables) MITM (/tags/#MITM) HTTPS (/tags/#HTTPS) tap (/tags/#tap) tun (/tags/#tun)
路由 (/tags/#路由) wireguard (/tags/#wireguard) Tun (/tags/#Tun) gvisor (/tags/#gvisor) Git (/tags/#Git)
NAT (/tags/#NAT) DNS (/tags/#DNS)

FRIENDS

Apple Developer (https://developer.apple.com/)

(https://www.zhihu.com/people/pandaychen)

(https://github.com/pandaychen)

Theme on GitHub (https://github.com/pandaychen/pandaychen.github.io.git) | Star 12 本站总访问量 253508次