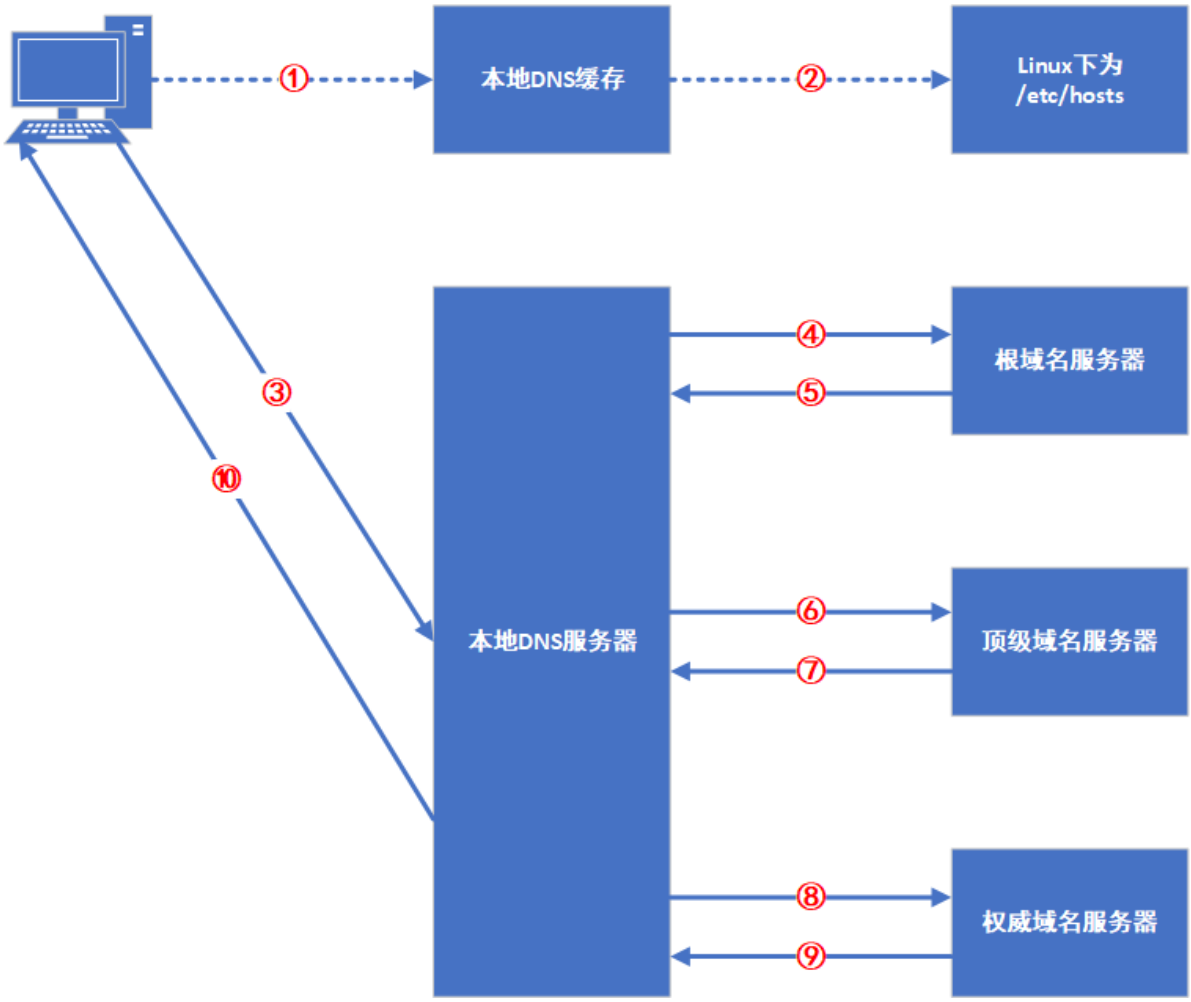
0x00 前言

本文是项目中的 DNS 相关开发工作记录

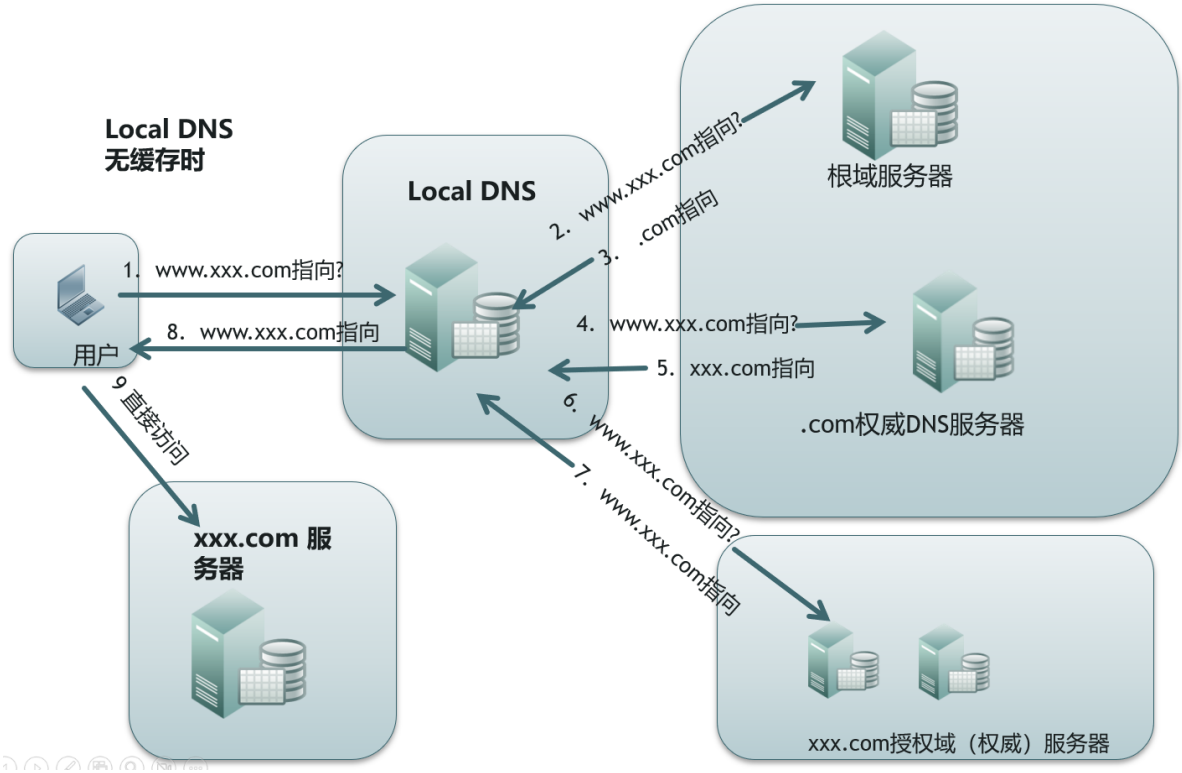
1. 如何合适的实现 DNS 查询缓存?
2. DOH、DOT 实现
3. 如何优雅的实现一个 DNS 代理
4. 透明网关中的 DNS 机制、DNS 分流机制等

0x01 基础

先回顾一下 DNS 解析的基础流程，如下图：



DNS 解析流程



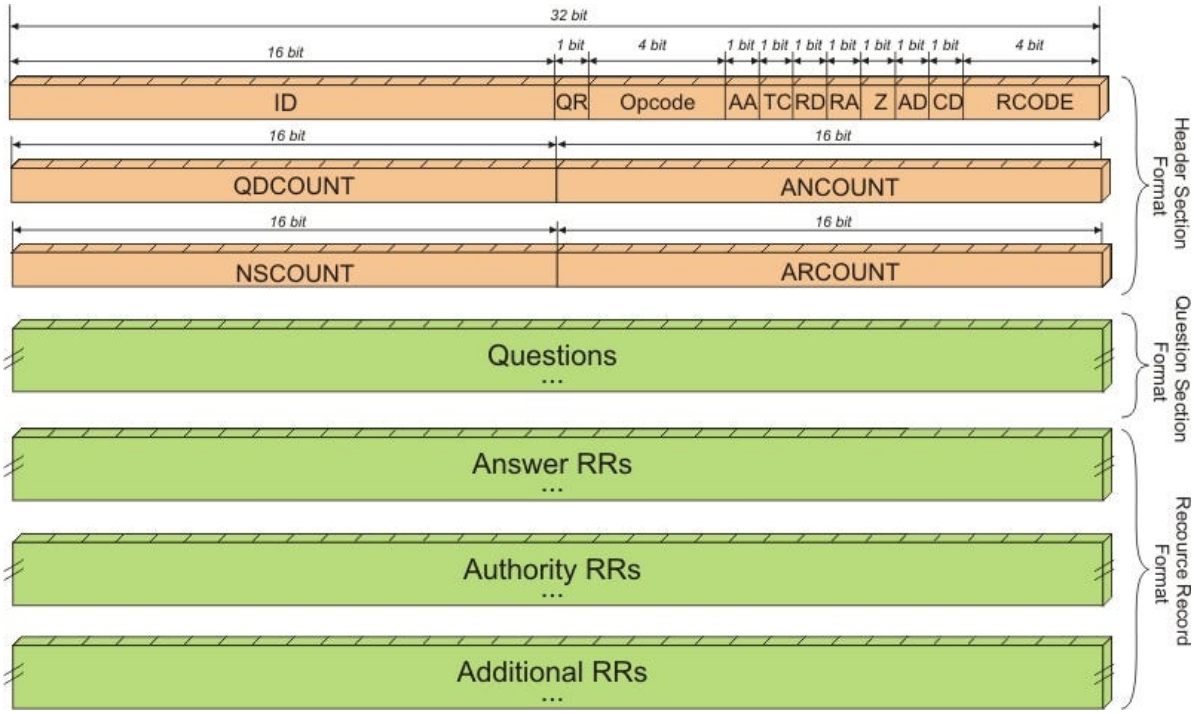
常用的 public DNS（UDP）

- `8.8.8.8`：Google
- `1.1.1.1`：Cloudflare
- `119.29.29.29`：阿里云
- `114.114.114.114`：电信
- `183.60.83.19`/`183.60.82.98`：腾讯云

DNS-over-HTTPS/TLS

- DOH：参考 DNS over HTTPs(DoH)
- DOT：参考 DNS over HTTPs(DoT)

DNS 报文格式



0x02 Kubernetes 的 DNS

温馨提示：特别注意域名结尾是否有一个点号 `.`

1、当 `ndots` 小于 `options ndots` 时

`options ndots` 的值默认是 `1`，在 K8S 中为 `5`，参考下面的例子，在一个 namespace `demo-ns` 中有两个 `SVC`，分别为 `demo-svc1` 和 `demo-svc2`，其 `/etc/resolv.conf` 内容大致如下：

```
nameserver 10.32.0.10
search demo-ns.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

当在 `demo-svc1` 中直接请求域名 `demo-svc2`，此时 `ndots` 为 `1`（小于配置中的 `5`），因此会触发配置中的 `search` 规则，即第一个解析的域名是 `demo-svc2.demo-ns.svc.cluster.local`，当解析不出来的时候继续下面的 `demo-svc2.svc.cluster.local`、`demo-svc2.cluster.local`，最后才是直接去解析 `demo-svc2.`；注意此规则适用于任何一个域名，如在 pod 中去访问一个外部域名如 `github.io` 时也会依次进行上述查询。

2、当 `ndots` 大于等于 `options ndots` 时

在 `demo-svc1` 中直接请求域名 `demo-svc2.demo-ns.svc.cluster.local`，此时的 `ndots` 为 `4`，依然会触发上面的 `search` 规则。而请求域名 `demo-svc2.demo-ns.svc.cluster.local.` 时，`ndots` 为 `5`，因此不会触发 `search` 规则，直接去解析 `demo-svc2.demo-ns.svc.cluster.local.` 这个域名并返回结果；又如请求更长的 pod 域名 `pod-1.demo-svc2.demo-ns.svc.cluster.local.`，此时的 `ndots` 为 `6`，亦不会触发 `search` 规则，会直接查询域名并返回解析

小结

- 同命名空间（namespace）内的服务直接通过 `service_name` 进行互相访问而不需要使用全域名（FQDN），此时 DNS 解析速度最快
- 跨命名空间（namespace）的服务，可以通过 `service_name.namespace_name` 进行互相访问，此时 DNS 解析第一次查询失败，第二次才会匹配到正确的域名
- 所有的服务之间通过全域名（FQDN）`service_name.namespace_name.svc.cluster_name.` 访问的时候 DNS 解析的速度最快
- 在 K8S 集群内访问大部分的常见外网域名（ndots 小于 `5`）都会触发 `search` 规则，因此在访问外部域名的时候可以使用 FQDN，即在域名的结尾配置一个点号 `.`

0x03 golang 的 DNSCACHE

通过 `net.Dial`/`net.DialContext` 创建连接时，或者使用 `Resolver` 解析 DNS 时，都是无缓存的；所以客户端并发场景下每创建一个连接，`Resolver` 都会去解析一次 DNS，如果遇到网络不好或 DNS 服务器问题，大概率会遇到类似错误 `dial tcp: lookup xxxx.com`，已有多个不错的 DNScache 库实现：

- A DNS Cache for Go
- Go package for caching DNS lookup results in memory.

这里分析下 dnscache 的实现，核心是建立了一个本地 map，缓存 dns 查询结果，同时异步开启逻辑轮询查询缓存域名的 DNS 解析结果；缺点是没有过期机制，导致内存会被放大：

```go
type Resolver struct {
  lock sync.RWMutex
  cache map[string][]net.IP // 域名对应多个地址
}
```

```
// 调用 net.LookupIP 查询，成功情况下缓存结果
func (r *Resolver) Lookup(address string) ([]net.IP, error) {
  ips, err := net.LookupIP(address)     //CALL net.LookupIP
  if err != nil {return nil, err}

  r.lock.Lock()
  r.cache[address] = ips
  r.lock.Unlock()
  return ips, nil
}


func New(refreshRate time.Duration) *Resolver {
  resolver := &Resolver {
    cache: make(map[string][]net.IP, 64),
  }
  if refreshRate > 0 {
    go resolver.autoRefresh(refreshRate)
  }
  return resolver
}

// 异步查询域名，并将正确的结果缓存
func (r *Resolver) Refresh() {
  i := 0
  r.lock.RLock()
  addresses := make([]string, len(r.cache))
  for key, _ := range r.cache {
    addresses[i] = key
    i++
  }
  r.lock.RUnlock()

  for _, address := range addresses {
    r.Lookup(address)
    time.Sleep(time.Second * 2)
  }
}
```

如果使用 `http.Transport`，可以在 `Dial` 中调用此包：

```
transport := &http.Transport {
  MaxIdleConnsPerHost: 64,
  Dial: func(network string, address string) (net.Conn, error) {
    separator := strings.LastIndex(address, ":")
    ip, _ := dnscache.FetchString(address[:separator])
    return net.Dial("tcp", ip + address[separator:])
  },
}
```

0x04 常用的 DNS 库

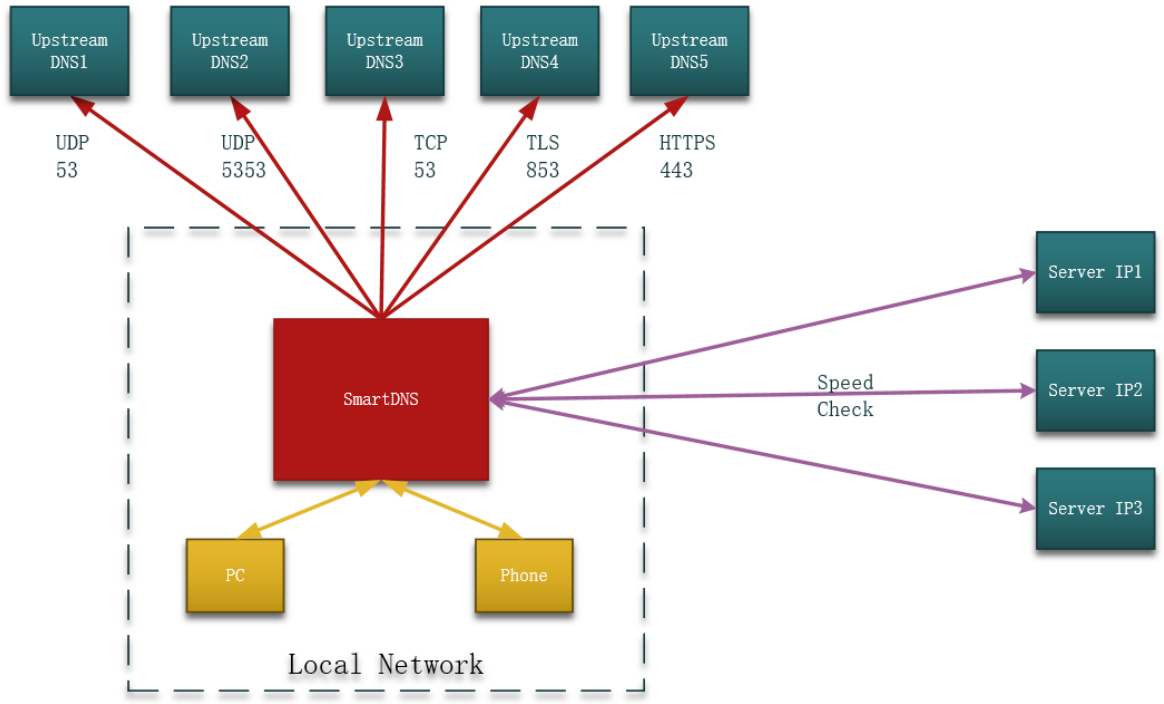
DNS 常用库 1：miekg/dns

用例可以参考：Go DNS example programs

DNS 常用库 2：coredns

0x05 DNS 代理实现：smartDNS

smartDNS 是本地的 DNS 代理服务器，接受本地客户端的 DNS 查询请求，然后从多个上游 DNS 服务器获取 DNS 查询结果，并将访问速度最快的结果返回给客户端，以此提高网络访问速度。与 DNSmasq 的 all-servers 机制不同，SmartDNS 返回的是访问速度最快的解析结果



主要工作流程如下：

1. SmartDNS 接收本地网络设备的 DNS 查询请求
2. 然后将查询请求发送到多个上游 DNS 服务器，可支持 UDP 标准端口或非标准端口查询，以及 TCP 查询
3. 上游 DNS 服务器返回域名对应的服务器 IP 地址列表，SmartDNS 则会检测从本地网络访问速度最快的服务器 IP
4. 最后将访问速度最快的服务器 IP 返回给本地客户端

0x06 透明代理中的 DNS

笔者在网关项目中也实现了类似的 DNSproxy，大致功能如下：

- 拦截所有经由网关的 DNS 请求（查询），由网关进行代理查询（或者丢弃）
- 支持 TUN-fake-DNS 查询及伪造 A 记录返回；不开启 fake 模式，则作为 DNS 代理，直接转发 DNS 请求到上游查询
- 支持 DNS 分流，指定 DNS 选用指定的 `nameserver`
- 支持 DNS 报文经过 `socks5` 代理服务端进行域名查询及返回
- 支持 `/etc/hosts` 本地解析（或是自定义 domain=>ip 的解析）

在实现中，DNSProxy 转发解析请求的时候会优化为并发查询：即会同时向所有配置的 DNS 上游服务器进行 DNS 查询，并选取最快的返回结果，以提高性能（参考 `dnsmasq` 实现）

通过在物理网卡上启动混杂模式，同时设置 `iptables -t nat -A PREROUTING -p udp --dport 53 -j REDIRECT --to-ports 53`，将所有 UDP 目的端口为 `53` 的流量重定向到本地端口 `53`（本地 `53` 启动的正式 DNSproxy）实现了 DNS 透明代理功能

DNS-FAKE-IP的实现思路

0x07 关于 DNS 的一些细节

1、Optimistic DNS（DNS 乐观解析）

由于现代网络的复杂性，大多数网站会将 DNS 的记录有效期（TTL）配置为很短的时间，如 `30s`。这样可以使得网络管理员修改 DNS 记录后迅速生效，不必再等待所有节点 TTL 超时，利于故障排除和维护。但这带来了一个问题，客户端会严格按照 TTL 去进行查询，那么每隔很短的时间就会进行再次查询，一次 DNS 查询的时间开销短至几毫秒，然而最长可以要数秒。频繁重复查询会造成不必要的延迟。Optimistic DNS 的优化方案是在建立新连接时，如果本地 DNS 缓存已经过期，那么也先继续使用旧的结果，同时进行 DNS 查询，如果连接建立失败，则用新的结果重试。绝大多数情况下，DNS 记录是不变的，这样的方案根本不会影响正常使用，当极小概率遇到 DNS 记录切换时，也只会耽误一两个请求，可以极大的提升效率

2、DNS leaking

DNS 泄漏（DNS leaking）是指使用 VPN 连接时，计算机设备仍然使用 ISP 提供的 DNS 服务器，而不是 VPN 提供的 DNS 服务器。这意味着此网络活动可能会暴露给 ISP 或其他第三方，因为他们可以访问 DNS 查询历史记录。这可能会导致隐私受到威胁

3、DNSSEC && TC

由于 DNSSEC 会导致返回的数据包变大，一旦超出 UDP 所允许的大小，那么数据包会被截断，DNS 服务器返回一个 `TC=1` 的标志位，用户接到 `TC` 的标识后，会改为用 TCP 方式传输

0x08 DNSproxy的实现思路

指标与监控设计

TODO

0x09 DNS 性能压测

列举下笔者在开发 DNS 代理中使用到的 DNS 客户端性能压测库，推荐使用 dnstrace：

- dnspyre：CLI tool for a high QPS DNS benchmark
- dnstrace：Command-line DNS benchmark

0x0A 参考

- Examples made with Go DNS
- CoreDNS 篇 10 - 分流与重定向
- CoreDNS 篇 9-kubernetes 插件
- adguard-dns：概览
- 抓包就明白 CoreDNS 域名解析
- Public DNS resolver that protects you from ad trackers
- 回顾与展望 AdGuard DNS
- DNS-over-QUIC
- 盘点国内外优秀公共 DNS
- Golang DNS 解析
- kubernetes 集群中夺命的 5 秒 DNS 延迟