gRPC (/tags/#gRPC)

gRPC 源码分析之 Resolver 篇

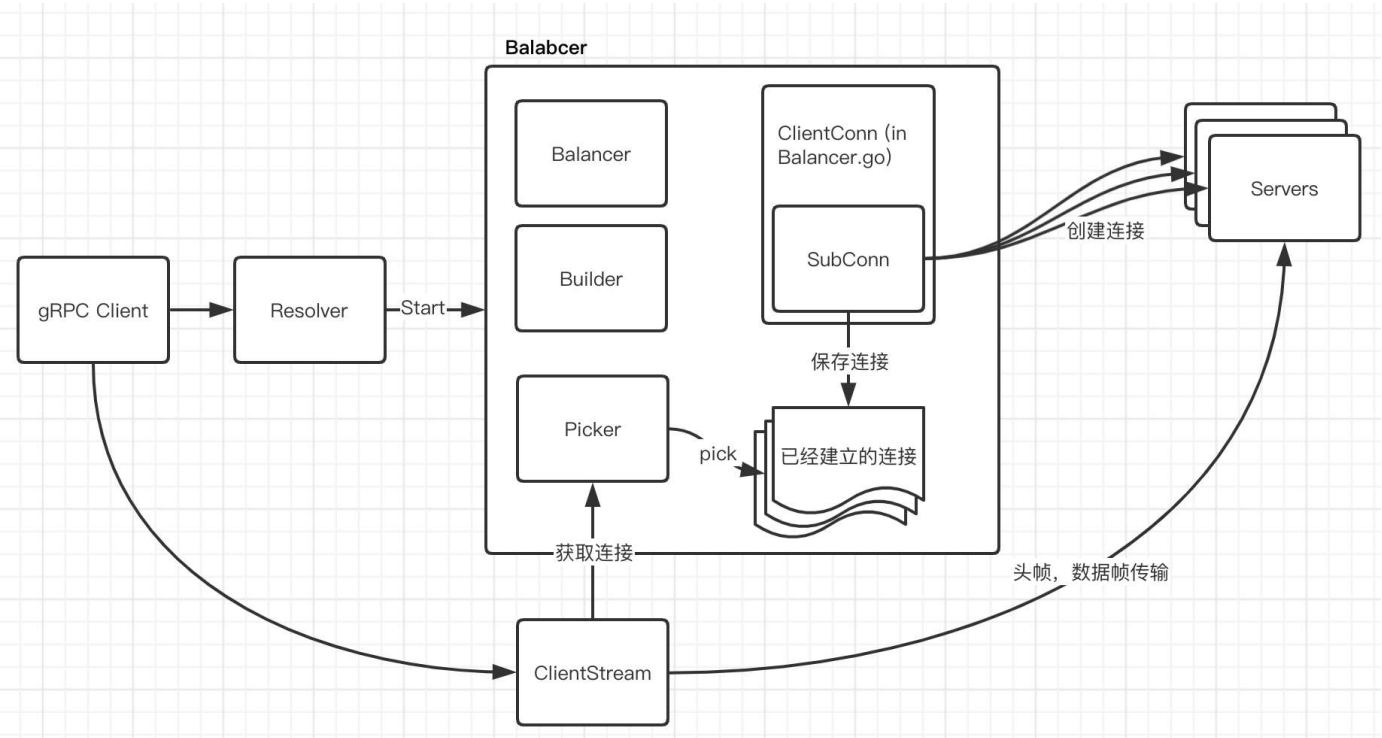
gRPC 客户端解析器实现分析

Posted by pandaychen on November 11, 2019

0x00 前言

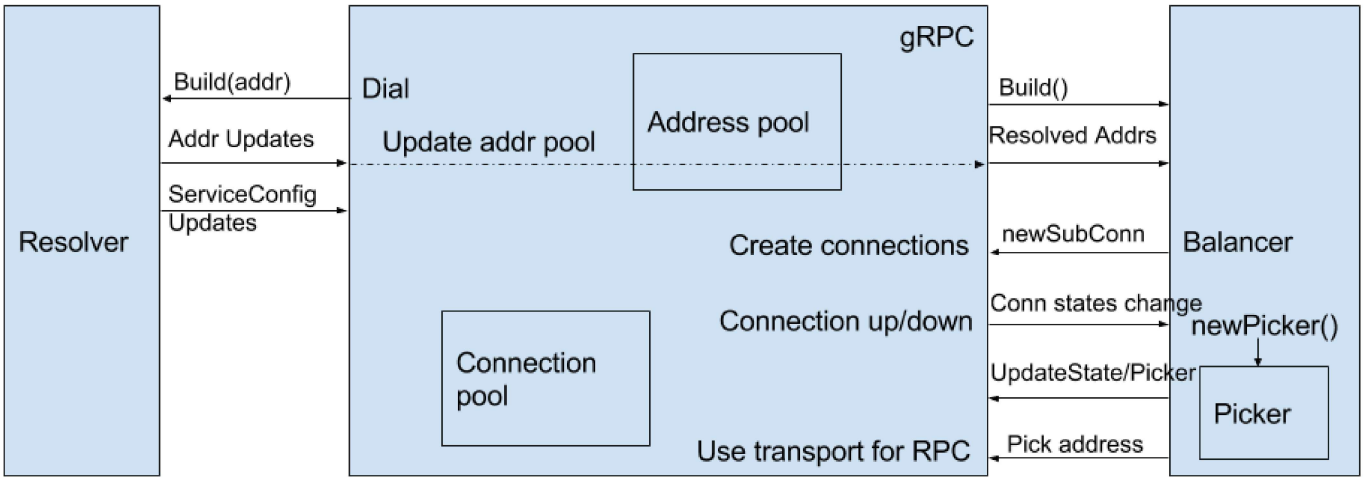
gRPC 负载均衡是针对每次请求，而不是连接，这样可以保证服务端负载的均衡性，所有 gRPC 负载均衡算法实现都在客户端。本系列文章对 gRPC 的负载均衡框架做深入的分析。

gRPC客户端的全景视图：



0x01 gRPC 的解析器 Resolver

Resolver，直观上就联想到域名解析配置 `/etc/resolv.conf`，配置域名解析规则，和 DNS 服务器交互，获取解析结果。
本篇文章详细分析下 gRPC-Resolver (https://godoc.org/google.golang.org/grpc/resolver) 的实现，上一篇文章 基于 gRPC 的服务发现与负载均衡（基础篇）(https://pandaychen.github.io/2019/07/11/GRPC-SERVICE-DISCOVERY/) 介绍了 gRPC 负载均衡的基础概念及基本组件。
首先，来看一张关于 gRPC 客户端负载均衡实现的官方架构图（截止目前最新的架构）：



从图中，可以看到 Resolver 解析器位于架构的最左方，它主要完成下面这几个功能：

- 服务发现的实现
- 和注册中心（`Etcd`、`CoreDNS`、`Consul` 等）通信，实时获取服务器的列表（或者处理变更信息）
- 将上步获取的数据（更新的结果），及时发送给 Balancer，用以更新 Connection Pool（内置 gRPC 长连接池）

0x02 Resolver 的应用

两个非常典型的实现，`DNSResolver`（PULL 方式）和 `EtcdResolver`（PUSH 方式）：

- DNSResolver (https://pandaychen.github.io/2019/07/11/GRPC-BALANCER-DNSRESOVLER-ANALYSIS/)，gRPC 官方提供的实现，以定时轮询方式访问域名服务器来获取服务器更新

- EtcdResolver (https://etcd.io/docs/v3.3.12/dev-guide/grpc_naming/)，Etcd 文档提供的示例，以 List-Watcher 方式实现的 Resolver

0x03 resolver.go 分析

本小节，来分析下 resolver.go (https://godoc.org/google.golang.org/grpc/resolver) 的主要结构。最早 gRPC 提供了 `Naming` (https://godoc.org/google.golang.org/grpc/naming) 包，用来完成解析的功能，不过其功能很有限，现在已经 deprecated 了，现在一般用 resolver 包来完成。

resolver.Address

Address 结构中的 `Addr` 字段一般包含 ip 和端口信息， `Metadata` 一般放入服务器的额外信息，比如权重、总连接数等等信息，用于负载均衡算法的判定：

```
// Address represents a server the client connects to.
// This is the EXPERIMENTAL API and may be changed or extended in the future.
type Address struct {
        // Addr is the server address on which a connection will be established.
        Addr string

        // ServerName is the name of this address.
        // If non-empty, the ServerName is used as the transport certification authority for
        // the address, instead of the hostname from the Dial target string. In most cases,
        // this should not be set.
        //
        // If Type is GRPCLB, ServerName should be the name of the remote load
        // balancer, not the name of the backend.
        //
        // WARNING: ServerName must only be populated with trusted values. It
        // is insecure to populate it with data from untrusted inputs since untrusted
        // values could be used to bypass the authority checks performed by TLS.
        ServerName string

        // Attributes contains arbitrary data about this address intended for
        // consumption by the load balancing policy.
        Attributes *attributes.Attributes

        // Type is the type of this address.
        //
        // Deprecated: use Attributes instead.
        Type AddressType

        // Metadata is the information associated with Addr, which may be used
        // to make load balancing decision.
        //
        // Deprecated: use Attributes instead.
        Metadata interface{}
}
```

resolver.Builder

官方文档的这句： `Builder creates a resolver that will be used to watch name resolution updates` ，大致意思是：当向 gRPC 注册（解析器）服务发现时，实际上注册的是 `Builder` ，一般在 `Build` 中会开启单独的 groutine，进行 List-watcher 逻辑。

Build() 参数中的 `cc ClientConn` ，提供了 `Builder` 和 `ClientConn` 交互的纽带，可以调用 `cc.UpdateState(resolver.State{Addresses: addrList})` 来向 `ClientConn` 即时发送服务器列表的更新。

这里先预埋一个问题，我们实现的 `resolver.Builder()` 在哪个 gRPC 阶段被调用?

```
type Builder interface {
        // Build creates a new resolver for the given target.
        //
        // gRPC dial calls Build synchronously, and fails if the returned error is
    // not nil.
    // 创建 Resolver，当 resolver 发现服务列表更新，需要通过 ClientConn 接口通知上层
        Build(target Target, cc ClientConn, opts BuildOptions) (Resolver, error)
        // Scheme returns the scheme supported by this resolver.
        // Scheme is defined at https://github.com/grpc/grpc/blob/master/doc/naming.md.
        Scheme() string
}
```

resolver.Resolver

```
// Resolver watches for the updates on the specified target.
// Updates include address updates and service config updates.
type Resolver interface {
        // ResolveNow will be called by gRPC to try to resolve the target name
        // again. It's just a hint, resolver can ignore this if it's not necessary.
        //
    // It could be called multiple times concurrently.
    // 当有连接被出现异常时，会触发该方法，因为这时候可能是有服务实例挂了，需要立即实现一次服务发
        ResolveNow(ResolveNowOptions)
        // Close closes the resolver.
        Close()
}
```

resolver.ClientConn

resolver 中的 `ClientConn` 结构提供了 resolver 通知 `ClientConn` 更新服务端列表的回调方法。日常封装自己的 resolver 时，建议使用一个成员变量来接收 `resolver.ClientConn`，这里注册中心用的是 `Etcd`：

```
pe EtcdResolver struct {
 ...
    cc            resolver.ClientConn       // 用来接收在 resolver.Build() 中的第二个参数，由外
    EtcdCli       *clientv3.Client
 ...
```

再看 `ClientConn` 的结构：

```
// ClientConn contains the callbacks for resolver to notify any updates
// to the gRPC ClientConn.
//
// This interface is to be implemented by gRPC. Users should not need a
// brand new implementation of this interface. For the situations like
// testing, the new implementation should embed this interface. This allows
// gRPC to add new methods to this interface.
type ClientConn interface {
    // UpdateState updates the state of the ClientConn appropriately.
    // 服务列表和服务配置更新回调接口，目前都使用这个通知 gRPC 上层，包含了下面两个已废弃接口的功
        UpdateState(State)
        // ReportError notifies the ClientConn that the Resolver encountered an
        // error. The ClientConn will notify the load balancer and begin calling
        // ResolveNow on the Resolver with exponential backoff.
        ReportError(error)
        // NewAddress is called by resolver to notify ClientConn a new list
        // of resolved addresses.
        // The address list should be the complete list of resolved addresses.
        //
        // Deprecated: Use UpdateState instead. -- 已废弃（服务列表更新通知接口）
        NewAddress(addresses []Address)
        // NewServiceConfig is called by resolver to notify ClientConn a new
        // service config. The service config should be provided as a json string.
        //
        // Deprecated: Use UpdateState instead. -- 已废弃（服务配置更新通知接口）
        NewServiceConfig(serviceConfig string)
        // ParseServiceConfig parses the provided service config and returns an
        // object that provides the parsed config.
        ParseServiceConfig(serviceConfigJSON string) *serviceconfig.ParseResult
}
```

resolver.Scheme

通过 `Scheme` 来标识 gRPC 使用的解析器名字，比如， `Etcd` 的 scheme 就可以写成 `etcdv3:///"` ， `CoreDNS` 的 scheme 可写成 `dns:///8.8.8.8:53`

```
// string. e.g. target string "unknown_scheme://authority/endpoint" will be parsed into
// &Target{Scheme: resolver.GetDefaultScheme(), Endpoint: "unknown_scheme://authority/endpoint"
type Target struct {
        Scheme    string
        Authority string
        Endpoint  string
}
```

resolver小结

现在我们对 resolver 做下小结：

其中 `Builder` 接口用来创建 `Resolver`，我们可以提供自己的服务发现实现逻辑，然后将其注册到 gRPC 中，其中通过 `scheme` 来标识，而 `Resolver` 接口则是提供服务发现功能。当 `Resolver` 发现服务列表发生变更时，会通过 `ClientConn` 回调接口通知上层。 下一小节，我们来看下 Resolver 在 gRPC 客户端流程中实例的调用链。

0x04 Resolver 的调用链

本小节，来分析下 Resolver 的调用链是什么。

在实际项目中，一般 gRPC + Etcd 实现的客户端的负载均衡调用的代码实现如下（Etcd 也有个官方的简单实现：Using etcd discovery with go-grpc (https://etcd.io/docs/v3.3.12/dev-guide/grpc_naming/)）：

1、通用的实现方式

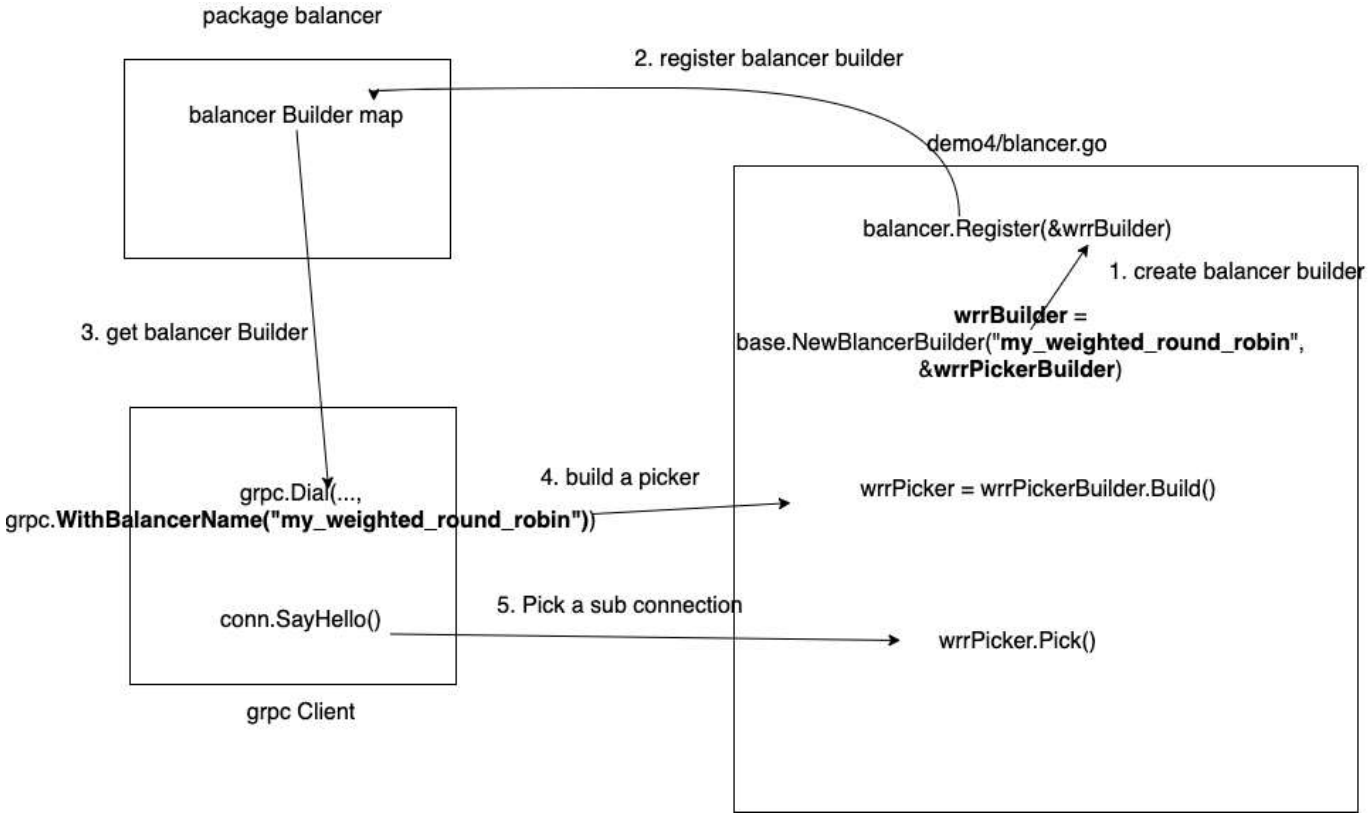
```
resolver := NewXXXResolver(service_name,registy_endpoints)      // 传入要 watcher 的 key 和 etc
balancer := grpc.RoundRobin(resolver)                           // 调用负载均衡器初始化 resolve
ctx, cancel := context.WithTimeout(context.Background(), GRPC_CONNECT_TIMEOUT*time.Second)
//Dial/DialContext 开启 balancer+resolver 的功能
conn, err = grpc.DialContext(ctx, registy_endpoints, grpc.WithTransportCredentials(creds), grpc
```

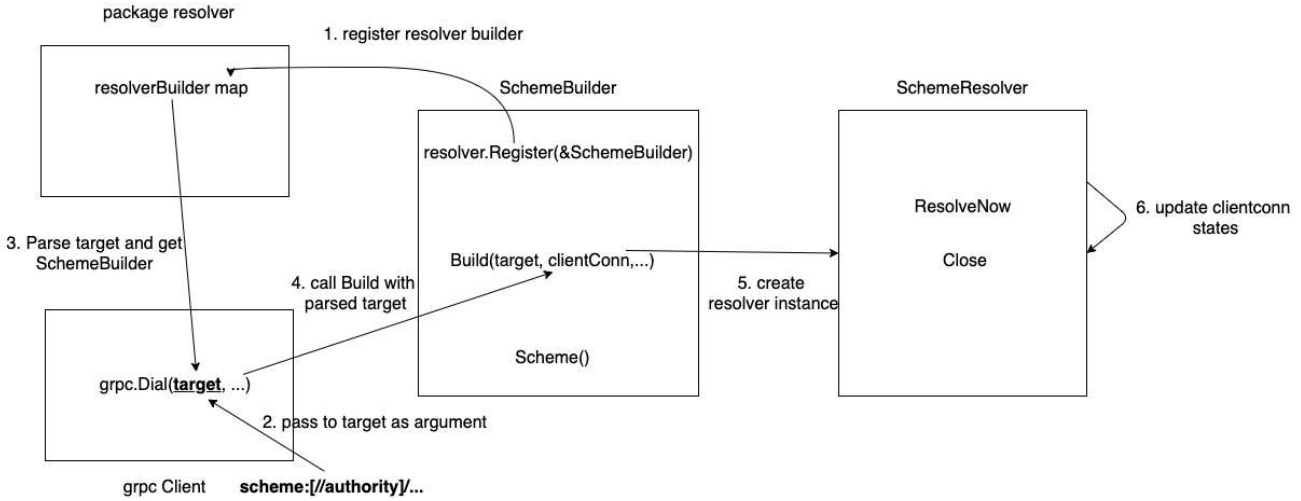
2、Etcd 文档的实现，大同小异

```
import (
        "go.etcd.io/etcd/clientv3"
        etcdnaming "go.etcd.io/etcd/clientv3/naming"
        "google.golang.org/grpc"
)
cli, cerr := clientv3.NewFromURL("http://localhost:2379")
r := &etcdnaming.GRPCResolver{Client: cli}
b := grpc.RoundRobin(r)
conn, gerr := grpc.Dial("my-service", grpc.WithBalancer(b), grpc.WithBlock(), ...)
```

调用链视图

`grpc.RoundRobin` –> `Dial/DialContext` –> `newCCResolverWrapper` –> 调用 `resolver.Build()` —> 调用 `Build()` 实现的 `Watcher()` —> 完成并返回状态 整个过程如下图所示（balancer与resovler）：





接下来，简单看下调用链上的这几个方法：

grpc.RoundRobin方法

grpc.RoundRobin (https://github.com/grpc/grpc-go/blob/master/balancer.go#L124) 是一个 `grpc.Balancer` 它的原型如下，可见传入的参数是 `naming.Resolver`，返回值是 `grpc.Balancer`，ps：从描述上看，此方法也即将 `Deprecated`，更新后的方法 balancer/roundrobin (https://godoc.org/google.golang.org/grpc/balancer/roundrobin)，这是一个很规范的实现 gRPC `Picker` 的模板，后面文章会详细分析。

```
// Deprecated: please use package balancer/roundrobin. May be removed in a future 1.x release.
func RoundRobin(r naming.Resolver) Balancer {
        return &roundRobin{r: r}
}
```

DialContext 方法

这里我们就从 `DialContext` 入手，看下 gRPC 对 Resolver 的处理过程。当我们使用 Dial 或者 DialContext 接口创建 gRPC 的客户端连接时，首先会解析参数 target（Etcd 集群地址），然后创建对应的 resolver：

```
func DialContext(ctx context.Context, target string, opts ...DialOption) (conn *ClientConn, err
        cc := &ClientConn{
                target:            target,
                csMgr:             &connectivityStateManager{},
                conns:             make(map[*addrConn]struct{}),        // 初始化 ClientConn 的
                dopts:             defaultDialOptions(),
                blockingpicker:    newPickerWrapper(),
                czData:            new(channelzData),
                firstResolveEvent: grpcsync.NewEvent(),
        }
    ......
    for _, opt := range opts {
        // 解析参数
                opt.apply(&cc.dopts)
        }
    ......

        // resolverBuilder，用于解析 target 为目标服务列表
        // 如果没有指定 resolverBuilder
        if cc.dopts.resolverBuilder == nil {
                // 解析 target，根据 target 的 scheme 获取对应的 resolver
                cc.parsedTarget = parseTarget(cc.target)
                cc.dopts.resolverBuilder = resolver.Get(cc.parsedTarget.Scheme)         // 重要
                // 如果 scheme 没有注册对应的 resolver
                if cc.dopts.resolverBuilder == nil {
            // 使用默认的 resolver
                        cc.parsedTarget = resolver.Target{
                                Endpoint: target, // 这时候参数 target 就是 endpoint，passthrou
                        }
            // 获取默认的 resolver，也就是 passthrough
                        cc.dopts.resolverBuilder = resolver.Get(cc.parsedTarget.Scheme)
                }
        } else {
        // 如果 Dial 的 option 中手动指定了需要使用的 resolver，那么 endpoint 也是 target
        // 实例中指定了我们自己实现的 NewXXXResolver，target 为 Etcd 集群的地址
        /* // &Target{Scheme: resolver.GetDefaultScheme(), Endpoint: "unknown_scheme://authorit
                cc.parsedTarget = resolver.Target{Endpoint: target}
        }

        ......

        // newCCResolverWrapper 方法内调用 builder 的 Build 接口创建 resolver
        rWrapper, err := newCCResolverWrapper(cc)
        if err != nil {
                return nil, fmt.Errorf("failed to build resolver: %v", err)
        }

        cc.mu.Lock()
        cc.resolverWrapper = rWrapper
        cc.mu.Unlock()

        ......
```

```
        return cc, nil
}
```

parseTarget

回想下 resolver 包中 `Build()` (https://godoc.org/google.golang.org/grpc/resolver#Get) 中的 `Scheme()` 方法，一般实现中自己设定一个解析器标识:

```
type Builder interface {
    // Build creates a new resolver for the given target.
    //
    // gRPC dial calls Build synchronously, and fails if the returned error is not nil.
    Build(target Target, cc ClientConn, opts BuildOptions) (Resolver, error)
    // Scheme returns the scheme supported by this resolver.
    // Scheme is defined at https://github.com/grpc/grpc/blob/master/doc/naming.md.
    Scheme() string
}
```

在项目中，需要设定一个解析器标识 scheme，例如，解析器名称设置为 `etcdv3resolver`，那么在 `Dial` 传入 `etcdv3resolver:///`，就在 `parseTarget` 这里做解析:

```
func (r EtcdNewResolver) Scheme() string {
        return "etcdv3resolver"
}
```

如果已经实现自定义的 resolver，那么传入 `Dial` 的 scheme 会在这里做解析:

```
// 有效的 target: scheme://authority/endpoint
func parseTarget(target string) (ret resolver.Target) {
        var ok bool
        ret.Scheme, ret.Endpoint, ok = split2(target, "://")    // 传入 etcdv3resolver:///, Sch
        if !ok {
        // 如果没有 scheme，则整个 target 作为 endpoint
                return resolver.Target{Endpoint: target}
        }
    // 如果指定了 sheme，那么必须有 `/`，分割 authorigy 和 endpoint
    // 当不需要指定 authority，比如使用 dnsResolver 时:`dns:///www.demo.com`
        ret.Authority, ret.Endpoint, ok = split2(ret.Endpoint, "/")
        if !ok {
                return resolver.Target{Endpoint: target}
        }
        return ret
}
```

resolver.Get(cc.parsedTarget.Scheme)方法

这里会根据解析得到的解析器的名称，去 resolver 包中 `m map[string]Builder` 这个全局 map 中查询对应的 `Builder`，然后返回给 `cc.dopts.resolverBuilder`，这样，我们自定义的 `resolver.Builder` 就成功的和 `ClientConn` 关联上了。

```
//resolver 中的全局 MAP
m = make(map[string]Builder)
// Get returns the resolver builder registered with the given scheme.
// If no builder is register with the scheme, nil will be returned.
func Get(scheme string) Builder {
        if b, ok := m[scheme]; ok {
                return b
        }
        return nil
}
```

newCCResolverWrapper方法

这里回答前面预埋的一个问题，我们自己构建 resolver 中的 `Build()` 方法，最终是在哪里被调用的？答案就是 `newCCResolverWrapper`。代码如下：

```
func newCCResolverWrapper(cc *ClientConn) (*ccResolverWrapper, error) {
        // 在 DialContext 方法中，已经初始化了 resolverBuilder
    rb := cc.dopts.resolverBuilder
        if rb == nil {
                return nil, fmt.Errorf("could not get resolver for scheme: %q", cc.parsedTarget
        }

        // ccResolverWrapper 实现 resolver.ClientConn 接口，用于提供服务列表变更之后的通知回调接
        ccr := &ccResolverWrapper{
                cc:     cc,     // 本客户端的 ClientConn
                addrCh: make(chan []resolver.Address, 1),   // 用于通知 channel
                scCh:   make(chan string, 1),                // 用于通知 channel
        }

        var err error
    // 创建 resolver，resolver 创建之后，需要立即执行服务发现逻辑，然后将发现的服务列表通过 reso

        // 非常重要：这里的 Build 也就是我们在自己的 resolver.go 中实现的 Build() 方法，传入的三
         // 在 gRPC 的 DNSresolver 实现里，调用 dnsBuilder.Build 函数创建 dnsResolver
        ccr.resolver, err = rb.Build(cc.parsedTarget, ccr, resolver.BuildOption{DisableServiceC
        if err != nil {
                return nil, err
        }
        return ccr, nil
}
```

0x05 Resovler 与 Balancer 的交互

0x06 总结

至此，对 gRPC-Resolver 的流程分析就基本完成了。

0x07 参考

- package resolver文档 (https://godoc.org/google.golang.org/grpc/resolver)
- package grpc文档 (https://godoc.org/google.golang.org/grpc)
- package roundrobin文档 (https://godoc.org/google.golang.org/grpc/balancer/roundrobin)
- golang grpc 客户端负载均衡、重试、健康检查 (https://yangxikun.com/golang/2019/10/19/golang-grpc-client-side-lb.html)

PREVIOUS
后台开发积累（2019 年） **(/2019/11/04/NICE-SUMUP/)**

NEXT
GRPC 源码分析之 DNSRESOLVER 篇 **(/2019/11/12/GRPC-BALANCER-DNSRESOVLER-ANALYSIS/)**

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 253503次