Rpc核心中间件开发

联系QQ：2816010068，加入会员群

二叉树出品

目录

- go module介绍
- Rpc核心流程
- 服务发现中间件开发
- 负载均衡中间件开发
- 短链接中间件开发
- 代码优化
- 测试和演示

go module

- Go module
  - go 1.11版本推出的依赖管理工具
- 环境准备
  - 当前项目在GoPath下面
    - 默认不开启go module的支持
    - 需要在环境变量添加 GO111MODULE=on
      - Windows：我的电脑-》属性-》高级系统设置里面添加
      - Linux： 在~/.bash_profile文件添加：export GO111MODULE=on
  - 当前项目不在GoPath下面
    - 默认支持
  - 依赖包下载加速：
    - 添加环境变量：GOPROXY=https://goproxy.io

go module

- 初始化
  - 已有项目
    - go mod init
  - 新项目
    - go mod init 模块名
- 包管理元信息
  - go.mod，存储当前模块依赖的包名以及版本
  - go.sum，存储当前模块依赖的包名以及版本、哈希信息
  - 提交代码的时候，需要把这两个文件提交上去
- 包下载
  - go build、go run等编译命令，会自动下载包
  - go mod tidy，自动下载当前项目的依赖

go module

- 版本升级
  - go get github.com/objcoding/testmod@v1.0.1
- 列出当前依赖
  - go list -m all
- 列出可以升级的依赖
  - go list -m -u all
- 升级所有依赖
  - go get -u

rpc核心流程

- 核心思想
  - 执行一次rpc调用，就是执行一系列中间件的过程
  - 有些中间件，是在rpc真正调用之前执行
  - 有些中间件，是在rpc真正调用之后执行
- 核心中间件
  - 服务发现中间件
  - 负载均衡中间件
  - 短链接中间件
  - grpc调用（也被封装成中间件函数）
  - …

rpc核心流程

- 核心流程
  - 通过服务发现中间件，获取当前存活的节点列表A
  - 通过负载均衡中间件，从列表A中间选择一台可用的节点B（或机器）
  - 通过短链接中间件，建立当前客户端到服务B之间的连接
  - 使用步骤3建立的连接，执行真正的grpc调用
  - 完成rpc调用
- 中间件数据传递
  - 通过context进行传递
  - 封装了RpcMeta数据结构

rpc核心流程

- RpcMeta数据结构

```go
type RpcMeta struct {
    //调用方名字
    Caller string
    //服务提供方
    ServiceName string
    //调用的方法
    Method string
    //调用方集群
    CallerCluster string
    //服务提供方集群
    ServiceCluster string
    //TraceID
    TraceID string
    //环境
    Env string
    //调用方IDC
    CallerIDC string
    //服务提供方IDC
    ServiceIDC string
    //当前节点
    CurNode *registry.Node
    //历史选择节点
    HistoryNodes []*registry.Node
    //服务提供方的节点列表
    AllNodes []*registry.Node
    //当前请求使用的连接
    Conn *grpc.ClientConn
}
```

rpc核心流程

- 方法
  - 获取数据
  - 初始化RpcMeta

```
39  type rpcMetaContextKey struct{}
40
41  func GetRpcMeta(ctx context.Context) *RpcMeta {
42      meta, ok := ctx.Value(rpcMetaContextKey{}).(*RpcMeta)
43      if !ok {
44          meta = &RpcMeta{}
45      }
46
47      return meta
48  }
49
50  func InitRpcMeta(ctx context.Context, service, method, caller string) context.Context {
51      meta := &RpcMeta{
52          Method:      method,
53          ServiceName: service,
54          Caller:      caller,
55      }
56      return context.WithValue(ctx, rpcMetaContextKey{}, meta)
57  }
```

服务发现中间件开发

- 核心代码

```go
func NewDiscoveryMiddleware(discovery registry.Registry) Middleware {
    return func(next MiddlewareFunc) MiddlewareFunc {
        return func(ctx context.Context, req interface{}) (resp interface{}, err error) {
            //从ctx获取rpc的metadata
            rpcMeta := meta.GetRpcMeta(ctx)
            if len(rpcMeta.AllNodes) > 0 {
                return next(ctx, req)
            }

            service, err := discovery.GetService(ctx, rpcMeta.ServiceName)
            if err != nil {
                logs.Error(ctx, "discovery service:%s failed, err:%v", rpcMeta.ServiceName, err)
                return
            }

            rpcMeta.AllNodes = service.Nodes
            resp, err = next(ctx, req)
            return
        }
    }
}
```

服务发现中间件开发

- NewDiscoveryMiddleware生成一个中间件
  - 传入registry.Registry实例，因为闭包的特性，所以这个registry实例，可以在以后中间件处理函数一直使用
  - 通过registry实例进行服务发现，如果当前已经拿到节点列表，则调用下一个中间件
  - 传入的registry.Registry实例，需要在client实例化的时候，进行初始化

负载均衡中间件开发

- 核心代码

```go
func NewLoadBalanceMiddleware(balancer loadbalance.LoadBalance) Middleware {
    return func(next MiddlewareFunc) MiddlewareFunc {
        return func(ctx context.Context, req interface{}) (resp interface{}, err error) {
            //从ctx获取rpc的metadata
            rpcMeta := meta.GetRpcMeta(ctx)
            if len(rpcMeta.AllNodes) == 0 {
                err = errno.NotHaveInstance
                logs.Error(ctx, "not have instance")
                return
            }
            //生成loadbalance的上下文,用来过滤已经选择的节点
            ctx = loadbalance.WithBalanceContext(ctx)
            for {
                rpcMeta.CurNode, err = balancer.Select(ctx, rpcMeta.AllNodes)
                if err != nil {
                    return
                }

                logs.Debug(ctx, "select node:%#v", rpcMeta.CurNode)
                rpcMeta.HistoryNodes = append(rpcMeta.HistoryNodes, rpcMeta.CurNode)
                resp, err = next(ctx, req)
                if err != nil {
                    //连接错误的话，进行重试
                    if errno.IsConnectError(err) {
                        continue
                    }
                    return
                }
```

负载均衡中间件开发

- 核心思路
  - 通过服务发现，已经拿到要调用的服务的机器列表
  - 通过meta.GetRpcMeta(ctx)，拿到RpcMeta，从而拿到机器列表
  - 使用之前写的负载均衡库，进行机器选择
- 新特性
  - 剔除已经选择的节点

剔除已经选择的节点

- 核心思路
  - 负载均衡之前，初始化selectedNodes结构体，保存在context中

```go
type selectedNodes struct {
    selectedNodeMap map[string]bool
}

type loadbalanceFilterNodes struct{}

func WithBalanceContext(ctx context.Context) context.Context {

    sel := &selectedNodes{
        selectedNodeMap: make(map[string]bool),
    }
    return context.WithValue(ctx, loadbalanceFilterNodes{}, sel)
}
```

剔除已经选择的节点

- 核心思路
  - 每次选择节点后，更新到selectedNodes的map中

```
func setSelected(ctx context.Context, node *registry.Node) {

    sel := GetSelectedNodes(ctx)
    if sel == nil {
        return
    }

    addr := fmt.Sprintf("%s:%d", node.IP, node.Port)
    logs.Debug(ctx, "filter node:%s", addr)
    sel.selectedNodeMap[addr] = true
}
```

剔除已经选择的节点

- 核心思路
  - 每次选择节点时，把当前selectedNodes中的节点过滤调

```go
func filterNodes(ctx context.Context, nodes []*registry.Node) []*registry.Node {

    var newNodes []*registry.Node
    sel := GetSelectedNodes(ctx)
    if sel == nil {
        return newNodes
    }

    for _, node := range nodes {
        addr := fmt.Sprintf("%s:%d", node.IP, node.Port)
        _, ok := sel.selectedNodeMap[addr]
        if ok {
            logs.Debug(ctx, "addr:%s ok", addr)
            continue
        }
        newNodes = append(newNodes, node)
    }

    return newNodes
}
```

短链接中间件开发

- 核心思路
  - 使用负载均衡拿到的节点，进行连接建立
  - 通过meta.GetRpcMeta(ctx)，拿到RpcMeta，从而拿到负载均衡选择的机器
  - 使用负载均衡选择的机器，进行建立连接

短链接中间件开发

- 核心代码

```
func ShortConnectMiddleware(next MiddlewareFunc) MiddlewareFunc {
    return func(ctx context.Context, req interface{}) (resp interface{}, err error) {
        //从ctx获取rpc的metadata
        rpcMeta := meta.GetRpcMeta(ctx)
        if rpcMeta.CurNode == nil{
            err = errno.InvalidNode
            logs.Error(ctx, "invalid instance")
            return
        }

        address := fmt.Sprintf("%s:%d", rpcMeta.CurNode.IP, rpcMeta.CurNode.Port)
        conn, err := grpc.Dial(address, grpc.WithInsecure())
        if err != nil {
            logs.Error(ctx, "connect %s failed, err:%v", address, err)
            return nil, errno.ConnFailed
        }

        rpcMeta.Conn = conn
        defer conn.Close()
        resp, err = next(ctx, req)
        return
    }
}
```

代码优化

- 自动生成的代码原则
  - 入口原则
    - 仅仅作为入口，逻辑尽可能的简化
    - 复杂逻辑，在包里进行实现

代码优化

- Rpc client代码生成调整
  - 独立封装KoalaClient类
  - 自动代码生成的XXXClient封装KoalaClient，所有调用逻辑在KoalaClient中进行实现

KoalaClient封装

- 实现rpc调用的核心逻辑
  - 初始化注册中心
  - 初始化负载均衡器
  - 初始化各种选项配置
  - 实现中间件构造
  - 实现rpc请求调用

KoalaClient封装

- 各种选项配置

```
type RpcOptions struct {
    ConnTimeout  time.Duration
    WriteTimeout time.Duration
    ReadTimeout  time.Duration
    ServiceName  string
    //注册中心名字
    RegisterName string
    //注册中心地址
    RegisterAddr string
    //注册中心路径
    RegisterPath string
}
```

KoalaClient封装

- 初始化注册中心

```
initRegistryOnce.Do(func() {
    ctx := context.TODO()
    var err error
    globalRegister, err = registry.InitRegistry(ctx,
        client.opts.RegisterName,
        registry.WithAddrs([]string{client.opts.RegisterAddr}),
        registry.WithTimeout(time.Second),
        registry.WithRegistryPath(client.opts.RegisterPath),
        registry.WithHeartBeat(10),
    )
    if err != nil {
        logs.Error(ctx, "init registry failed, err:%v", err)
        return
    }
})

client.register = globalRegister
```

KoalaClient封装

- 实现方法调用

```
func (k *KoalaClient) Call(ctx context.Context, method string, r interface{}, handl

    //构建中间件
    caller := k.getCaller(ctx)
    ctx = meta.InitRpcMeta(ctx, k.opts.ServiceName, method, caller)
    middlewareFunc := k.buildMiddleware(handle)
    resp, err = middlewareFunc(ctx, r)
    if err != nil {
        return nil, err
    }

    return resp, err
}
```

KoalaClient封装

- 实现中间件构造

```
func (k *KoalaClient) buildMiddleware(handle middleware.MiddlewareFunc) middleware.Middle

    var mids []middleware.Middleware
    mids = append(mids, middleware.NewDiscoveryMiddleware(k.register))
    mids = append(mids, middleware.NewLoadBalanceMiddleware(k.balance))
    mids = append(mids, middleware.ShortConnectMiddleware)
    m := middleware.Chain(mids[0], mids...)
    return m(handle)

}
```

代码生成代码更改

- XXXClient封装了KoalaClient

```
type {{Capitalize .Package.Name}}Client struct {
    serviceName string
    client *rpc.KoalaClient
}

func New{{Capitalize .Package.Name}}Client(serviceName string, opts...rpc.RpcOptionFunc) *{{Capitalize
    c :=  &{{Capitalize .Package.Name}}Client{
        serviceName: serviceName,
    }
    c.client = rpc.NewKoalaClient(serviceName, opts...)
    return c
}
```

代码生成代码更改

- Rpc调用函数修改

```
{{range .Rpc}}
func (s *{{Capitalize $.Package.Name}}Client) {{.Name}}(ctx context.Context, r*{{$.Package.Name}}.{
    /*
    middlewareFunc := rpc.BuildClientMiddleware(mwClient{{.Name}})
    mkResp, err := middlewareFunc(ctx, r)
    if err != nil {
        return nil, err
    }
*/
    mkResp, err := s.client.Call(ctx, "{{.Name}}", r, mwClient{{.Name}})
    if err != nil {
        return nil, err
    }
    resp, ok := mkResp.(*{{$.Package.Name}}.{{.ReturnsType}})
    if !ok {
        err = fmt.Errorf("invalid resp, not *{{$.Package.Name}}.{{.ReturnsType}}")
        return nil, err
    }

    return resp, err
}
```

测试和演示