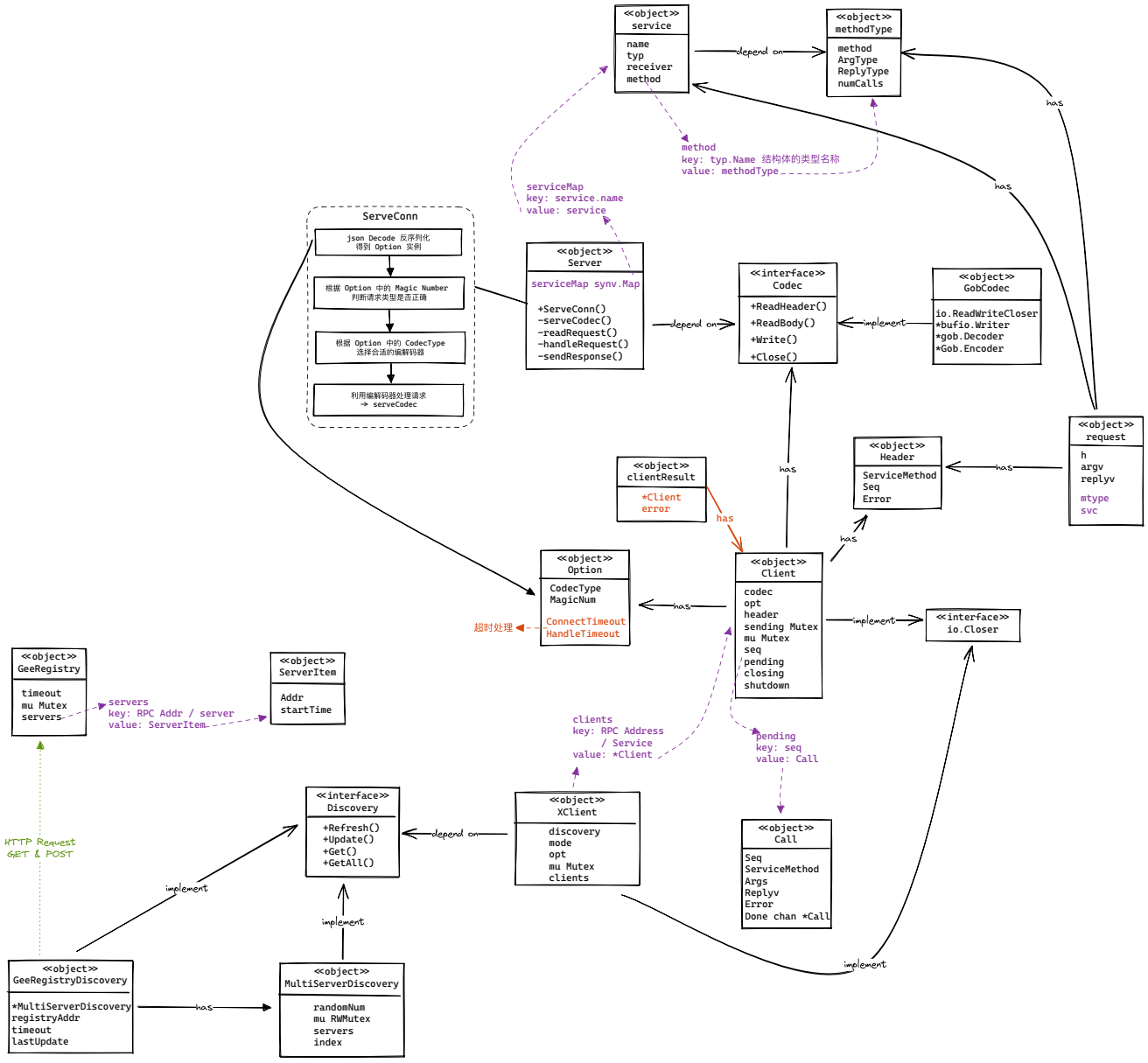
通信 - RPC

#golang #学习 #RPC

消息的编码与解码



> 编码：序列化；
> 解码：反序列化；

RPC 调用

```
err := client.Call("ServiceName.FuncName", args, &reply)
```

1. 客户端发送的请求：
   - 服务名 `ServiceName` → Header
   - 调用的方法名 `MethodName` → Header
   - 方法的参数 `args` → **Body**

- 补：请求的序号 `seq` → Header

2. 服务端的响应：
   - 错误 `err` → Header
   - 返回值 `reply` → **Body**

序列化与反序列化

- HTTP 报文 = Header + Body
  - Body 的格式与长度通过 Header 中的 `Content-Type` 和 `Content-Length` 指定；
  - 服务器 → 解析 Header → 读取 Body

```
// codec/codec.go
type Header struct {
        ServiceMethod string // "Service.Method"
        Seq           uint64 // 请求的序号，用来区分不同的请求；由客户端选定；
        Error         string // 错误信息，客户端处为 nil，服务端将错误放在这里；
}

type Codec interface {
        io.Closer                         // 关闭连接
        ReadHeader(*Header) error         // 读 Header
        ReadBody(interface{}) error       // 读 Body
        Write(*Header, interface{}) error // 写回复，第一个参数为 header，第二个参数为
body
}
// codec/gob.go
type GobCodec struct {
        conn io.ReadWriteCloser
        buf  *bufio.Writer
        dec  *gob.Decoder  // 用来实现 ReaderHeader 和 ReadBody
        enc  *gob.Encoder  // 用来实现 Write
}

// 确保 GobCodec 实现了 Codec 接口
var _ Codec = (*GobCodec)(nil)
```

通信过程

- 协商消息的编解码方式，设计一个名为 Option 的结构体来承载；

```
// server.go
type Option struct {
        MagicNumber int        // 确保这是一个 geerpc 请求
        CodecType   codec.Type // 用户可以选择的不同的编码解码器
}
```

- `Option{MagicNumber: xxx, CodecType: xxx}`
  - 固定 JSON 编码；
- `Header{ServiceMethod …}`
  - 编码方式由 `Option.CodeType` 决定；
- `Body interface{}`

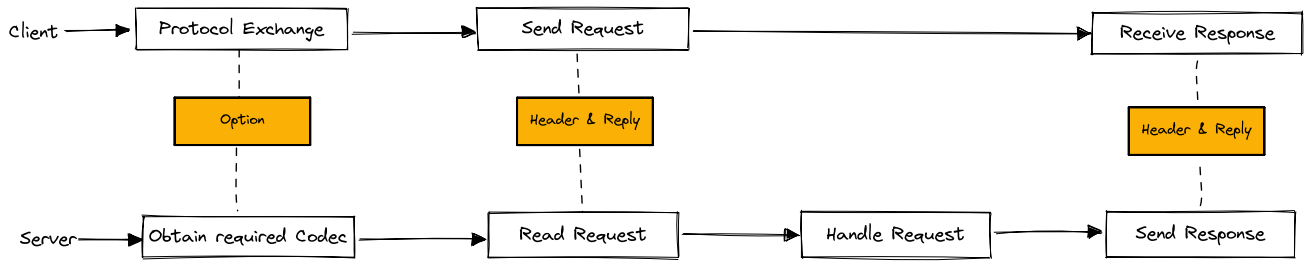
- 编码方式由 `Option.CodeType` 决定；

在一次连接中，Option 固定在报文的最开始，Header 和 Body 可以有多个，即：

```
| Option | Header1 | Body1 | Header2 | Body2 | ...
```

服务端的实现

```go
// Listen
// network: 通信所用的网络，如 tcp, unix ...
// address: 用于监听连接的主机服务器地址/端口号，如 :8888，若为 :0 则会自动选择空的端口号
Listen(network string, address string) (Listener, error)

// ServeConn → 解析 Option → 判断请求是否合法 → 选择 Codec → serveCodec
func (server *Server) ServeConn(conn io.ReadWriteCloser)

// serveCodec → readRequest → handleRequest → sendResponse
func (server *Server) serveCodec(cc codec.Codec)

// 存储 RPC 调用时的所有信息
type request struct {
        h            *codec.Header // Header
        argv, replyv reflect.Value // 请求方法的参数和返回值
}
```



高性能客户端

- 支持**并发和异步**；
- 一个能被 Remote 调用的函数需要满足的 **五个条件**：
  1. 方法的类型：`exported`；
  2. 方法：`exported`；
  3. 方法有两个参数，且参数的类型都是 `expoeted or builtin`；
  4. 方法的第二个参数是一个指针 `*pointer`；
  5. 方法返回值类型为 `error`；

```go
func (t *T) MethodName(argType T1, replayType *T2) error
```

Call

- 对 RPC 请求的封装；

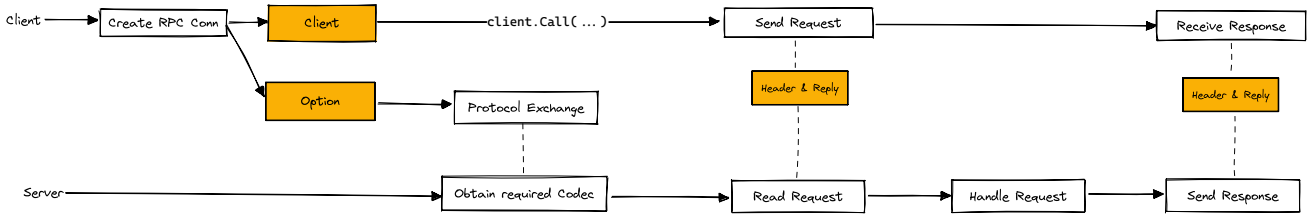
```go
// Call 代表一个活跃的 RPC 请求
type Call struct {
        Seq           uint64      // 请求的序号，用来区分不同的请求；由客户端选定；
        ServiceMethod string      // "Service.Method";
        Args          interface{} // 调用方法的参数;
        Reply         interface{} // 调用方法的返回值;
        Error         error       // 错误信息;
        Done          chan *Call  // 调用完成时的信号;
}

// 调用结束时，通知调用方;
func (call *Call) done() {
        call.Done ← call
}
```

Client

```go
type Client struct {
        cc       codec.Codec      // 编码解码器;
        opt      *Option          // Option 部分，用于确定请求的类型和编解码器的类型;
        sending  sync.Mutex       // 保证请求的有序发送，防止多个请求报文混淆;
        header   codec.Header     // Header 部分;
        mu       sync.Mutex       // 保证写 pending 时的互斥锁;
        seq      uint64           // 请求的序号，每个请求的序号是唯一的;
        pending  map[uint64]*Call // seq ⟷ Call;
        closing  bool             // 客户端结束，用户主动关闭；用于判断客户端是否可用
        shutdown bool             // 服务端结束，用户被动关闭；用于判断客户端是否可用
}

// 需要实现 io.Closer 接口
var _ io.Closer = (*Client)(nil)

// registerCall → mu 互斥锁 → 加入 pending 映射表
func (client *Client) registerCall(call *Call) (uint64, error)

// removeCall → mu 互斥锁 → 从 pending 映射表取出对应的 Call
func (client *Client) removeCall(seq uint64) *Call

// terminateCalls → sending 互斥锁 → mu 互斥锁 → client.shutdown
// → 对 pending 中剩余的 Call 设置错误信息 → Call.done
func (client *Client) terminateCalls(err error)

// receive 通常是作为 goroutine 被调用
// 解析 Header → 从 pending 映射表根据 Header.seq 删除对应的 Call
// 1. → call 不存在，可能是请求没有被完整发送，或是被取消了，但是服务端仍然处理了
// 2. → call 存在，但是服务端处理错误 → Header.error → call.done()
// 3. → call 存在，服务端正确处理 → 从 Body 中读取 Reply 的值 → call.done()
// 如果发生了错误 → terminateCalls(err)
func (client *Client) receive()

// NewClient → 向服务端发送 Option 协商好编解码方式 → newClientCodec
func NewClient(conn net.Conn, opt *Option) (*Client, error)

// newClientCodec → 创建 Client 实例 → 调用 receive 协程
func newClientCodec(cc codec.Codec, opt *Option) *Client
```

```
// send → sending 互斥锁 → registerCall → 设置 Header → 编码并发送请求 cc.Write()
func (client *Client) send(call *Call)

// Go 异步接口 → 发送并返回 Call 实例
func (client *Client) Go(serviceMethod string, args, reply interface{}, done chan
*Call) *Call

// Call 同步接口 → 调用 Go → 被 call.Done channel 阻塞 → 等待响应结束
func (client *Client) Call(serviceMethod string, args, reply interface{}) error
```

```
// parseOptions
func parseOptions(opts ...*Option) (*Option, error)

// Dial → parseOptions 解析 opts 获得 Option 实例
// → net.Dial 连接 RPC 服务端 → NewClient
func Dial(network, address string, opts ...*Option) (client *Client, err error)
```



服务注册

- 通过反射实现服务注册功能；
- 在服务端实现服务调用，**服务 → 结构体**；

客户端发送的请求，包括 ServiceMethod 和 argv

```
{
        "ServiceMethod": "ServiceName.MethodName", // 确定要调用的是类型 ServiceName
的 MethodName 方法
        "Argv": "0100101010 ... " // 序列化后的字节流
}
```

通过反射获得结构体的方法

```
package main

import (
        "log"
        "reflect"
        "strings"
        "sync"
)

func main() {
        var wg sync.Mutex
        typ := reflect.TypeOf(&wg)

        // (reflect.Type).NumMethod(): 返回类型拥有的方法的数目
```

```
        // (reflect.Type).Method(int): 返回第 i 个方法
        for i := 0; i < typ.NumMethod(); i++ {
                method := typ.Method(i)
                // make(type, length, capacity)
                argv := make([]string, 0, method.Type.NumIn())
                returns := make([]string, 0, method.Type.NumOut())

                // 第 0 个入参为 wg 自身
                for j := 1; j < method.Type.NumIn(); j++ {
                        argv = append(argv, method.Type.In(j).Name())
                }

                for j := 0; j < method.Type.NumOut(); j++ {
                        returns = append(returns, method.Type.Out(j).Name())
                }

                log.Printf("func (w %s) %s(%s) %s",
                        typ.Elem().Name(),
                        method.Name,
                        strings.Join(argv, ","),
                        strings.Join(returns, ","),
                )
        }
}
```

通过反射实现服务

```
// service.go

type methodType struct {
        method    reflect.Method // 方法本身
        ArgType   reflect.Type   // 参数类型
        ReplyType reflect.Type   // 返回值类型
        numCalls  uint64         // 被调用了多少次
}

type service struct {
        name     string                 // 映射的结构体的名称
        typ      reflect.Type           // 映射的结构体的类型
        receiver reflect.Value          // 结构体的实例
        method   map[string]*methodType // 结构方法映射表
}

func (s *service) registerMethods() {
        s.method = make(map[string]*methodType)
        for i := 0; i < s.typ.NumMethod(); i++ {
                method := s.typ.Method(i)
                mtype := method.Type

                // 三个入参(包括实例本身), 一个出参(error)
                if mtype.NumIn() != 3 || mtype.NumOut() != 1 {
                        continue
                }
                // 出参的类型必须为 error
                if mtype.Out(0) != reflect.TypeOf((*error)(nil)).Elem() {
                        continue
```

```
                }

                // 必须为 exported 或 builtin 类型
                argType, replyType := mtype.In(1), mtype.In(2)
                if !isExportedOrBuiltinType(argType) ||
!isExportedOrBuiltinType(replyType) {
                        continue
                }

                s.method[method.Name] = &methodType{
                        method:    method,
                        ArgType:   argType,
                        ReplyType: replyType,
                }
                log.Printf("RPC Server: register %s.%s\n", s.name, method.Name)
        }
}
// call 通过反射值调用方法
func (s *service) call(m *methodType, argv, replyv reflect.Value) error

// newService 构造新的服务实例
// 输入为需要映射为服务的结构体 rcvr → registerMethod
func newService(rcvr interface{}) *service
```

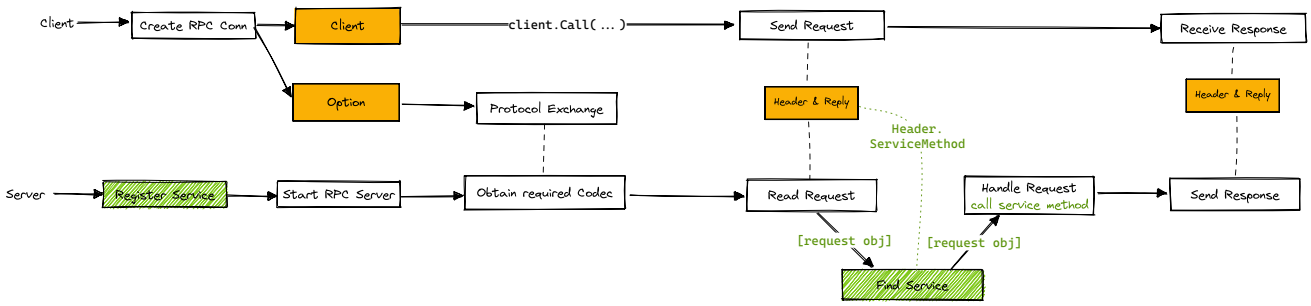
集成到服务端

1. 根据入参类型，将请求 body 解码/反序列化；
2. 调用 `service.call`，完成方法调用；
3. 将 reply 编码成字节流，构造响应报文返回；

```
type Server struct {
        serviceMap sync.Map // key: service.name, value: service
}

// Register → newService → 载入服务映射表 serviceMap
func (server *Server) Register(receiver interface{}) error {
        service := newService(receiver)
        if _, dup := server.serviceMap.LoadOrStore(service.name, service); dup {
                return errors.New("RPC: service already defined:" + service.name)
        }
        return nil
}

func Register(receiver interface{}) error {
        return DefaultServer.Register(receiver)
}

// findService → "ServiceName.MethodName" → "ServiceName", "MethodName"
// → 从 serviceMap 中找到对应 "ServiceName" 的 service 实例
// → 从 service 的实例的 method 中找到对应的 methodType
func (server *Server) findService(serviceMethod string) (svc *service, mtype
*methodType, err error)

// readRequest → 从 Header 中的 ServiceMethod 字段中解析要访问的服务和方法
// → 创建并返回 request 实例
```

```go
func (server *Server) readRequest(cc codec.Codec) (*request, error)

// handleRequest → 通过 request.svc.call 反射机制完成方法调用
func (server *Server) handleRequest(cc codec.Codec, req *request, sending
*sync.Mutex, wg *sync.WaitGroup) {
        defer wg.Done()
        err := req.svc.call(req.mtype, req.argv, req.replyv)
        if err != nil {
                req.h.Error = err.Error()
                server.sendResponse(cc, req.h, invalidRequest, sending)
        }
        server.sendResponse(cc, req.h, req.replyv.Interface(), sending)
}
```

```go
type request struct {
        h            *codec.Header // Header
        argv, replyv reflect.Value // 请求方法的参数和返回值
        mtype        *methodType   // 请求的方法
        svc          *service      // 请求的服务
}
```



超时处理

- 作用：如果缺少超时处理机制，无论是服务端还是客户端都容易因为网络或其他错误导致挂死，资源耗尽，降低服务的可用性；
- 添加地点：
  1. 客户端创建连接时；
  2. 客户端 `Client.Call()` 整个过程导致的超时（包含发送报文，等待处理，接收报文所有阶段）；
  3. 服务端处理报文，即 `Server.handleRequest` 超时；

客户端连接超时

```go
type Option struct {
        MagicNumber    int           // 确保这是一个 geerpc 请求
        CodecType      codec.Type    // 用户可以选择的不同的 Codec
        ConnectTimeout time.Duration // 最长的处理时间，0 代表无限制
        HandleTimeout  time.Duration
}

var DefaultOption = &Option{
        MagicNumber:    MagicNumber,
        CodecType:      codec.GobType,
```

```
        ConnectTimeout: time.Second * 10, // 10s
}
```

```
type clientResult struct {
        client *Client
        err    error
}

type newClientFunc func(conn net.Conn, opt *Option) (client *Client, err error)

// dialTimeout
// **连接建立超时**
// 解析 Option 实例 → 调用 net.DialTimeout 建立 RPC 连接
// 1. → 若连接创建超时 → 返回错误
// 2. → 连接建立未超时 → 使用 go routine 创建 Client 实例 → client channel
//
// **创建客户端实例超时**
// 1. timeAfter() 先于 client channel 收到消息 → 创建实例超时;
// 2. client channel 先收到消息 → 创建实例未超时;
//
// 返回客户端实例 / 错误信息
func dialTimeout(f newClientFunc, network string, address string, opts ...*Option)
(client *Client, err error) {
        opt, err := parseOptions(opts...)
        if err ≠ nil {
                return nil, err
        }
        conn, err := net.DialTimeout(network, address, opt.ConnectTimeout)
        if err ≠ nil {
                return nil, err
        }

        defer func() {
                if err ≠ nil {
                        _ = conn.Close()
                }
        }()

        ch := make(chan clientResult)
        go func() {
                client, err := f(conn, opt)
                ch ←clientResult{client: client, err: err}
        }()

        if opt.ConnectTimeout == 0 {
                result := ←ch
                return result.client, result.err
        }

        select {
        case ←time.After(opt.ConnectTimeout):
                return nil, fmt.Errorf("RPC Client: connect timeout: expect within
%s", opt.ConnectTimeout)
        case result := ←ch:
                return result.client, result.err
        }
```

```
}

func Dial(network string, address string, opts ...*Option) (client *Client, err
error) {
        return dialTimeout(NewClient, network, address, opts...)
}
```

客户端调用超时

```
// **客户端调用超时**, 使用 context 包来实现
// 使用 context.WithTimeout 创建具备超时检测能力的 context 对象来控制
func (client *Client) Call(ctx context.Context, serviceMethod string, args
interface{}, reply interface{}) error {
        call := client.Go(serviceMethod, args, reply, make(chan *Call, 1))
        select {
        case <-ctx.Done():
                client.removeCall(call.Seq)
                return errors.New("RPC Client: call failed: " + ctx.Err().Error())
        case call := <-call.Done:
                return call.Error
        }
}
```

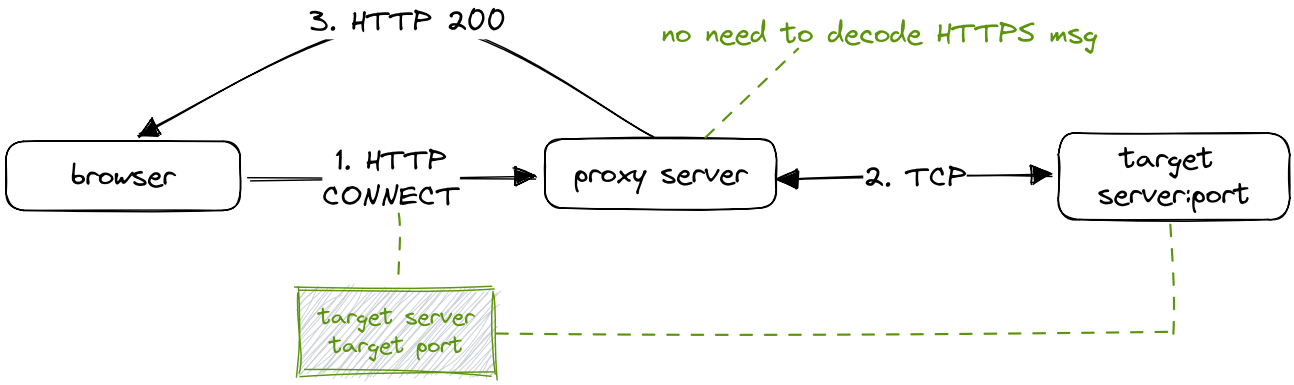
服务端处理超时

```
// **服务端处理超时**
// handleRequest
// 使用 select, called, goroutine 来判断处理时间是否超时
func (server *Server) handleRequest(cc codec.Codec, req *request, sending
*sync.Mutex, wg *sync.WaitGroup, timeout time.Duration) {
        defer wg.Done()
        called := make(chan struct{}) // 用于接收消息，代表处理没有超时，继续执行
sendResponse
        sent := make(chan struct{}) // 保证 sedResponse 只被设置了一次

        go func() {
                err := req.svc.call(req.mtype, req.argv, req.replyv)
                called <-struct{}{}
                if err != nil {
                        req.h.Error = err.Error()
                        server.sendResponse(cc, req.h, invalidRequest, sending)
                        sent <-struct{}{}
                        return
                }
                server.sendResponse(cc, req.h, req.replyv.Interface(), sending)
                sent <struct{}{}
        }()

        if timeout == 0 {
                <-called
                <-sent
                return
        }
        select {
        // 如果 time.After() 先于 called 收到消息，说明处理已经超时
```

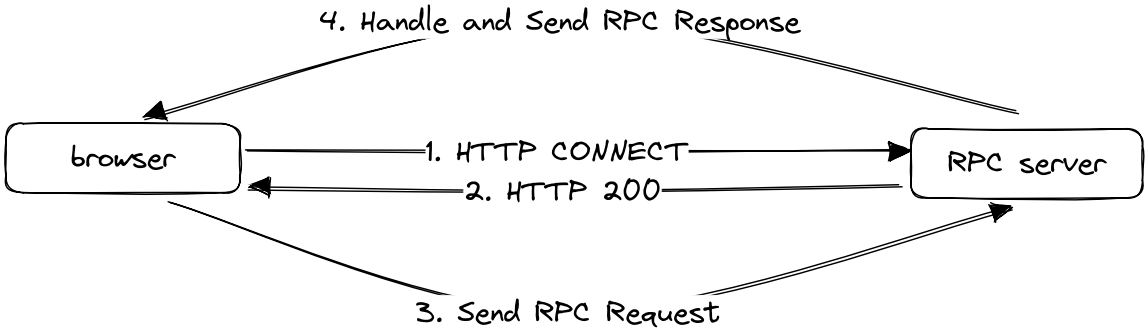
```
        // called 和 sent 都将被阻塞
        case <-time.After(timeout):
                req.h.Error = fmt.Sprintf("RPC Server: request handle timeout:
expect within %s", timeout)
                server.sendResponse(cc, req.h, invalidRequest, sending)
        case <-called:
                <-sent
        }
}
```

支持 HTTP 协议

支持 HTTP 协议的例子



RPC 服务端支持 HTTP



- RPC 的消息格式与 HTTP 协议并不兼容，需要协议转换 —— `CONNECT`；
  - `CONNECT` 一般用于代理服务；
- 具体过程：
  - 实际：将 HTTP 协议转换成 HTTPS 协议；
- **服务端：将 HTTP 协议转换成 RPC 协议，并返回 HTTP 200；**
  - `HTTP/1.0 200 Connected to GeeRPC`
- **客户端：新增通过 HTTP CONNECT 请求创建连接的逻辑；**
  - `CONNECT 10.0.0.1:9999/_geerpc_ HTTP/1.0`

- 通过创建好的连接发送 RPC 报文，先发送 Option，再发送 N 个请求报文；

服务端

```
const (
        connected        = "200 Connected to Gee RPC"
        defaultRPCPath   = "/_geerpc_"
        defaultDebugPath = "/debug/geerpc"
)

// ServeHTTP 实现 http.Handler 并回应 RPC 请求
// 根据 req.Method 判断是否为 CONNECT 方法 → 获得 TCP 套接字/连接 → 处理请求
func (server *Server) ServeHTTP(w http.ResponseWriter, req *http.Request) {
        // 判断请求方法是否为 CONNECT
        if req.Method != "CONNECT" {
                w.Header().Set("Content-Type", "text/plain; charset=utf-8")
                w.WriteHeader(http.StatusMethodNotAllowed)
                _, _ = io.WriteString(w, "405 must CONNECT\n")
                return
        }
        // 获取 tcp 套接字，http协议也是基于tcp协议的。
        conn, _, err := w.(http.Hijacker).Hijack()
        if err != nil {
                log.Print("RPC Hijacking ", req.RemoteAddr, ": ", err.Error())
                return
        }
    // 建立连接后统一服务
        _, _ = io.WriteString(conn, "HTTP/1.0 "+connected+"\n\n")
        server.ServeConn(conn)
}

// 监听默认的 defaultRPCPath 的请求并处理
func (server *Server) HandleHTTP() {
        http.Handle(defaultRPCPath, server)
}

func HandleHTTP() {
        DefaultServer.HandleHTTP()
}
```

客户端

- 发起 CONNECT 请求，检查返回状态码，若为 200 即可成功建立连接；

```
// NewHTTPClient 使用 HTTP 协议创建一个客户端
func NewHTTPClient(conn net.Conn, opt *Option) (*Client, error) {
        // 设置 CONNECT 请求
        _, _ = io.WriteString(conn, fmt.Sprintf("CONNECT %s HTTP/1.0\n\n",
defaultRPCPath))

        // 解析响应
        resp, err := http.ReadResponse(bufio.NewReader(conn),
&http.Request{Method: "CONNECT"})
        if err == nil && resp.Status == connected {
                // 转换为 RPC 协议
```

```
                return NewClient(conn, opt)
        }
        if err == nil {
                err = errors.New("Unexpected HTTP Response: " + resp.Status)
        }
        return nil, err
}

func DialHTTP(network string, address string, opts ...*Option) (*Client, error) {
        return dialTimeout(NewHTTPClient, network, address, opts...)
}

// rpcAddr is a general format (protocol@addr) to represent a rpc server
// eg, http@10.0.0.1:7001, tcp@10.0.0.1:9999, unix@/tmp/geerpc.sock
func XDial(rpcAddr string, opts ...*Option) (*Client, error) {
        parts := strings.Split(rpcAddr, "@")
        if len(parts) != 2 {
                return nil, fmt.Errorf("rpc client err: wrong format '%s', expect
protocol@addr", rpcAddr)
        }
        protocol, addr := parts[0], parts[1]
        switch protocol {
        case "http":
                return DialHTTP("tcp", addr, opts...)
        default:
                // tcp, unix or other transport protocol
                return Dial(protocol, addr, opts...)
        }
}
```

负载均衡

服务器有多个实例，每个实例提供相同的功能；为了提高整个系统的吞吐量，每个实例部署在不同的机器上。客户端选择任意的实例进行调用；

- 调用策略：随机选择 (Random)、轮询 (Round robin)、加权轮询、哈希/一致性哈希
- 负载均衡的前提：多个服务实例；

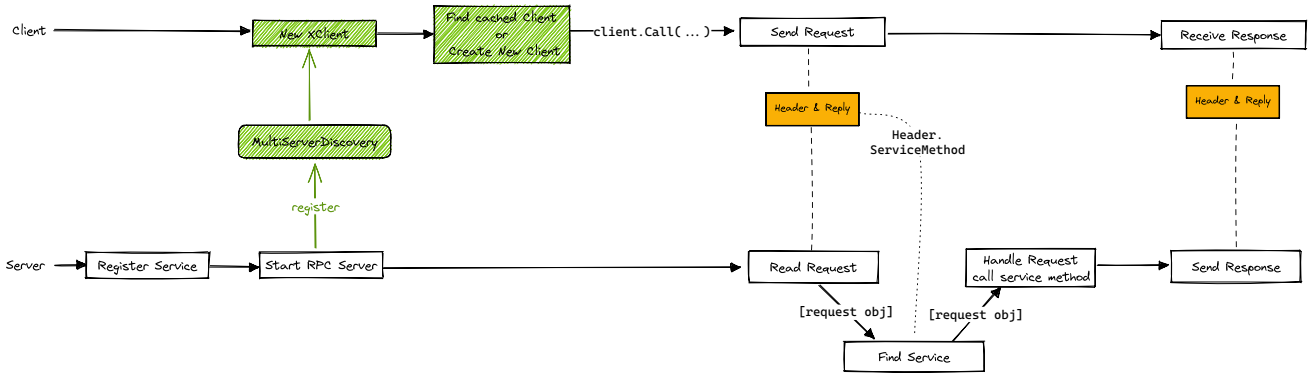
服务发现模块

```
type SelectMode int

const (
        RandomSelect     SelectMode = iota // 随机选择
        RoundRobinSelect                   //轮询
)

type Discovery interface {
        Refresh() error                      // 从注册中心更新服务列表
        Update(servers []string) error       // 手动更新服务列表
        Get(mode SelectMode) (string, error) // 根据复杂均衡策略，选择一个服务实例
        GetAll() ([]string, error)           // 获得所有的服务实例
}

// 需要实现 Discovery 接口
```

```go
type MultiServersDiscovery struct {
        r       *rand.Rand   // 随机数生成器
        mu      sync.RWMutex // 保证读写服务的互斥
        servers []string     // 所有可用的服务
        index   int          // 记录轮询算法选择的服务的索引
}

func NewMultiServerDiscovery(servers []string) *MultiServersDiscovery {
        d := &MultiServersDiscovery{
                r:       rand.New(rand.NewSource(time.Now().UnixNano())),
                servers: servers,
        }
        // 为了避免每次从 0 开始，初始化时随机设定一个值
        d.index = d.r.Intn(math.MaxInt32 - 1)
}
```

支持复杂均衡的客户端

```go
type XClient struct {
        d       Discovery  // 服务发现实例
        mode    SelectMode // 负载均衡的模式
        opt     *Option    // 协议选项
        mu      sync.Mutex
        clients map[string]*Client // RPC Address ⟷ Client
}

var _ io.Closer = (*XClient)(nil)

// dial → 检查 xc.clients 中是否有当前 rpcAddr 缓存的 Client
// 1. 有且 Client 可用 → 返回可用的 Client
// 2. 有但 Client 不可用 → 从 xc.clients 中删除 Client → 创建新的 Client 并加入到
xc.client 中 → 返回 Client
func (xc *XClient) dial(rpcAddr string) (*Client, error)

// call → dial 获得 Client 实例 → client.Call
func (xc *XClient) call(rpcAddr string, ctx context.Context, serviceMethod string,
args, reply interface{}) error

// Call → 根据策略选择合适的服务端 rpcAddr → xc.call
func (xc *XClient) Call(ctx context.Context, serviceMethod string, args, reply
interface{}) error

// Broadcast → 广播到所有符合的服务端实例
// 如果任意一个实例发生错误，则返回其中一个错误；如果调用成功，则返回其中一个的结果
func (xc *XClient) Broadcast(ctx context.Context, serviceMethod string, args,
reply interface{}) error
```



服务发现与注册中心

- 实现一个简单的注册中心，支持服务注册、接收心跳等功能；
- 客户端实现基于注册中心的服务发现机制；

注册中心

- 客户端和服务端都只需要感知注册中心的存在，而无需感知对方的存在：
  - 服务端启动后 → 向注册中心发送消息 → 注册中心知道服务已启动并可用；
    - 服务端还需要定期向注册中心发送心跳，证明自己还活着；
  - 客户端向注册中心询问那个服务是可用的 → 注册中心将可用的服务列表返回给客户端；
  - 客户端根据的到的服务列表，选择其中一个发起调用；
- 如果没有注册中心，需要硬编码服务端的地址，而且不能保证服务端是否处于可用状态；
- 常用的注册中心： etcd、zookeeper、consul

Gee Registry

> `/registry/`

- 默认超时时间设置为 5 min，任何注册的服务超过 5 min，即视为不可用状态;
- 采用 HTTP 协议提供服务，且所有的有用信息都承载在 HTTP Header 中；

```
type GeeRegistry struct {
        timeout time.Duration           // 时限
        mu      sync.Mutex              // 保证互斥 servers
        servers map[string]*ServerItem  // Server Addr ←→ ServerItem
}

type ServerItem struct {
        Addr  string
        start time.Time
}

// putServer 添加服务实例 → mu → 检查 servers 映射表
// 1. → 服务存在 → 更新服务的 startTime
// 2. → 服务不存在 → 创建新的 ServerItem
func (r *GeeRegistry) putServer(addr string)

// aliveServers 获得可用的服务列表 → mu
// → 遍历 servers
// 1. → 服务超时 → 删除服务
// 2. → 服务未超时 → 添加到结果列表中
```

```go
func (r *GeeRegistry) aliveServers() []string

// ServeHTTP 监听 /_geerpc_/registry
// 支持 GET 和 POST
// GET: 获得所有可用的服务
// POST: 更新/添加服务
func (r *GeeRegistry) ServeHTTP(w http.ResponseWriter, req *http.Request)

// HandleHTTP 使用 HTTP 实现注册中心
func (r *GeeRegistry) HandleHTTP(registryPath string)
```

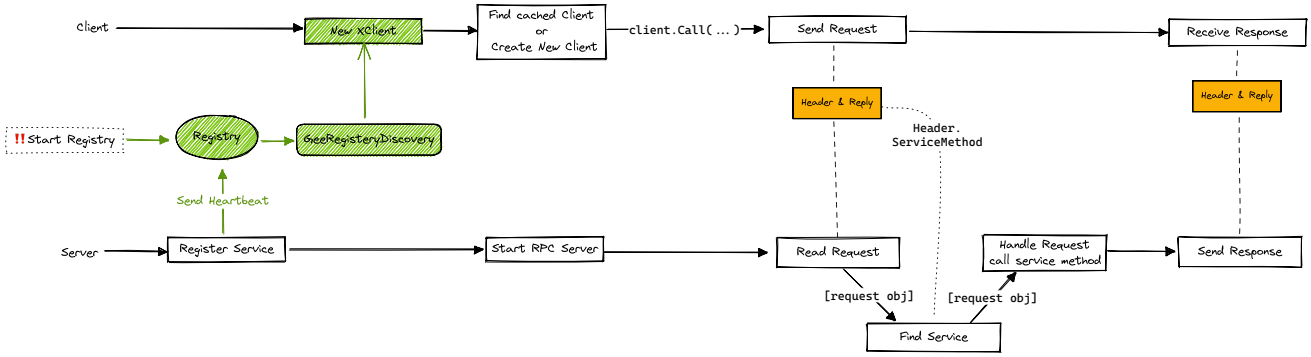
心跳

- 保证服务的可用性

```go
// sendHeartbeat
// 向注册中心发送 POST 请求 → 更新服务的 startTime → 保证服务可用
func sendHeartbeat(registry, addr string) error

// Heartbeat
// goroutine 定期执行 sendHeartbeat
func Heartbeat(registry, addr string, duration time.Duration)
```

服务中心发现模块

```go
type GeeRegistryDiscovery struct {
        *MultiServersDiscovery   // 属性复用
        registry   string        // 注册中心的地址
        timeout    time.Duration // 服务列表的过期时间
        lastUpdate time.Time     // 最后从注册中心更新服务列表的时间
}

// Update: 更新可用的服务列表 servers 及 lastUpdate
func (d *GeeRegistryDiscovery) Update(servers []string) error

// Refresh: 确保服务列表没有过期
// 向注册中心发送 GET 请求 → 获得所有可用的服务 → 更新 servers 及 lastUpdate
func (d *GeeRegistryDiscovery) Refresh() error

// GET 及 GetAll 需要先 Refresh
func (d *GeeRegistryDiscovery) Get(mode SelectMode) (string, error)
func (d *GeeRegistryDiscovery) GetAll() ([]string, error)
```



总结

参照 golang 标准库 net/rpc：

1. 服务端以及支持并发的客户端，支持选择不同的序列化与反序列化方式
2. 添加了超时处理机制；
3. 支持 TCP、Unix、HTTP 等多种传输协议；
4. 支持多种负载均衡模式；
5. 服务注册和发现中心；