

流程梳理

服务端

概述

服务端就是处理客户端发送过来的请求。针对发送过来的请求，先读取协商信息，再读取请求头，请求体。然后进行处理，处理完毕以后返回响应结果给客户端。

格式规定

那先从发送请求入手，客户端怎么发送请求，什么格式，怎么处理。

客户端和服务端通信需要协商一些信息。比如传输的报文是什么样的格式。我要用什么方式来进行消息的编码解码。下面规定：

```
/*
  | Option(Json) | Requese(Codec) | --> | Option(Json) | Header(Codec)
  | Body(Codec) |
*/
```

基本格式如上，先发送协商信息，看看用什么格式进行编码解码，甚至还需要协商超时处理的时间。协商过后，就能发送RPC请求。

因为一个RPC调用是遵循如下格式的：

```
err = client.Call("service.method", args, &reply)
```

我们可以把RPC请求分成两部分，**header** 和 **body**。我们可以把**参数**和**返回值**放在**body**中，而其他的信息，比如**服务名.方法名**、**请求序号**、**错误信息**放置在**header**中。

为什么要这样安排？

- 对于服务端来说，可以先读取**header**。看看**服务名.方法名**是否存在，是否可用，**请求序号**是否正确，不正确没必要进行下面的步骤。
- 对于客户端来说，可以先读取**header**。看看是否有错误，如果有的话，也不需要再读取**body**之类的。

因此，结构体定义如下：

```
const MagicNumber = 0x79779200

// Option 协商信息
type Option struct {
    MagicNumber int // 标记这是MyRPC的请求
    CodecType   codec.Type // 客户端选择什么方式进行编码
    ConnectTimeout time.Duration // 连接超时 默认10s
```

```

    HandleTimeout time.Duration // 处理超时 默认不设限 0s
}

// Header 请求和响应中的参数(args)和返回值(reply)放在body[这里用request结构
// 体包括body了] 其余信息放在header
type Header struct {
    ServiceMethod string // 服务名.方法名
    Seq           uint64 // 请求的序号, 用来区分不同的请求
    Error         string // 错误信息, 客户端置为空, 服务端如果发送错误, 将信息存在
    Error中
}

// request 一个完整的请求, 请求头, 请求参数, 响应
// 有服务注册以后, 就得带上, 哪个服务什么方法
type request struct {
    h          *codec.Header
    argv, replyv reflect.Value
    mtype      *methodType
    svc        *service
}

```

编解码器接口

协商信息主要是协商用什么样的编码解码（序列化和反序列化）格式。因为编码解码格式有多种，这里我们可以用一个接口来屏蔽不同编码解码器，使得编码格式变的统一。

```

// Codec 抽象出对消息体进行编码解码的接口 可屏蔽下面具体的编码方式 编解码器：主
// 要是读写关闭
type Codec interface {
    io.Closer //io关闭的接口
    ReadHeader(header *Header) error
    ReadBody(interface{}) error
    Write(*Header, interface{}) error
}

```

然后这里我们定义一些基础的函数。比如初始化函数。

```

// 定义编码解码的格式

// 这样写更简洁一些
type NewCodecFunc func(io.ReadWriteCloser) Codec

type Type string

const (
    GobType Type = "application/gob"
    JsonType Type = "application/json"
)

var NewCodecFuncMap map[Type]NewCodecFunc

func init() {

```

```
// 每种编码方式返回唯一的构造函数，这里放回的不是实例
NewCodecFuncMap = make(map[Type]NewCodecFunc)
NewCodecFuncMap[GobType] = NewGobCodec
}
```

Gob格式编解码

在这我们实现一个满足编解码器接口结构体。即需要实现接口中的那些方法。还实现了构造函数，主要是存入编解码器字典中，方便使用。

为什么不直接使用实例而是使用构造函数？

如果使用实例的话，系统可能需要维护很多实例，如果很多实例从头到尾都没有被使用到，浪费资源。

而且，服务器的字典，应该是提供一个方法，能找到对应的编解码器。如果是实例的话，直接用这个实例来操作？那要是多个客户端都是这种编解码方式，同时使用一个实例？

```
/*
每个编解码器都需要实现的方法有：
1. 构造函数
2. Codec接口规定的方法
   - ReadHeader
   - ReadBody
   - Write
   - Close
结构体都需要有：
1. 链接实例
2. 缓冲区
3. 解码器
4. 编码器
*/

// GobCodec 定义Gob的结构体
type GobCodec struct {
    conn io.ReadWriteCloser // 由构造函数传入，通常是通过TCP或者Unix建立
socket时得到的链接实例
    buf *bufio.Writer      // 为了防止阻塞而创建的带缓冲的writer
    dec *gob.Decoder       // gob对应的解码器
    enc *gob.Encoder       // gob对应的编码器
}

// NewGobCodec Gob编码的构造函数
func NewGobCodec(conn io.ReadWriteCloser) Codec {
    buf := bufio.NewWriter(conn)
    return &GobCodec{
        conn: conn,
        buf:  buf,
        dec:  gob.NewDecoder(conn),
        enc:  gob.NewEncoder(buf),
    }
}
```

```

func (c *GobCodec) ReadHeader(h *Header) error {
    return c.dec.Decode(h)
}

func (c *GobCodec) ReadBody(body interface{}) error {
    return c.dec.Decode(body)
}

func (c *GobCodec) Write(h *Header, body interface{}) (err error) {
    defer func() {
        _ = c.buf.Flush() // 最后记得清空缓冲区
        if err != nil {
            _ = c.Close() // 出错要关闭连接
        }
    }()
    if err := c.enc.Encode(h); err != nil {
        log.Println("rpc codec: gob error encoding header: ", err)
        return err
    }
    if err := c.enc.Encode(body); err != nil {
        log.Println("rpc codec: gob error encoding body: ", err)
        return err
    }
    return nil
}

func (c *GobCodec) Close() error {
    return c.conn.Close()
}

```

服务端的具体实现

接下来，就是按照概述说的流程走。服务端打开，监听对应端口。然后接受请求并进行处理，最后发送响应。这里还需要定义一个服务端的结构体，其中有一个并发安全的 Map，是为了在后面服务注册时使用：

```

type Server struct {
    serviceMap sync.Map
}

func NewServer() *Server {
    return &Server{}
}

var DefaultServer = NewServer()

```

打开，监听对应端口

```
// Accept 监听输入请求并提供服务，传入连接
func (server *Server) Accept(lis net.Listener) {
    for { // 循环等待socket连接建立 并开启子线程处理 处理过程交给ServerConn
        conn, err := lis.Accept()
        if err != nil {
            log.Println("rpc server: accept error :", err)
            return
        }
        go server.ServerConn(conn)
    }
}

func Accept(lis net.Listener) {
    DefaultServer.Accept(lis)
}
```

接受请求并进行处理，最后发送响应

识别协商信息

```
// ServerConn 在本函数中主要是识别编解码的协商信息，然后调用进行具体的处理的函数
func (server *Server) ServerConn(conn io.ReadWriteCloser) {
    defer func() {
        _ = conn.Close()
    }()
    // 协议协商
    var opt Option
    if err := json.NewDecoder(conn).Decode(&opt); err != nil {
        log.Println("rpc server: options error: ", err)
        return
    }
    // 判断是不是发给本RPC的
    if opt.MagicNumber != MagicNumber {
        log.Printf("rpc server : invalid magic number %x", opt.MagicNumber)
        return
    }
    // 获取对应的编解码格式 返回的是构造函数
    f := codec.NewCodecFuncMap[opt.CodecType]
    if f == nil {
        log.Printf("rpc server: invalid codec type %s", opt.CodecType)
        return
    }
    server.serverCodec(f(conn), &opt)
}
```

接收请求

识别完协商信息，正确，可进入下一个阶段。在这里我们进行请求的接收，请求接收主要分成三个阶段：

- 读取请求 `readRequest`
- 处理请求 `handleRequest`
- 回复请求 `sendResponse`

而且这里我们使用无限制的循环来进行接收请求。因为一次连接中允许接受多个请求，尽力而为，只有在header解析失败（可能所有请求结束了），才终止循环。

```
// invalidRequest 是发生错误时 argv 的占位符
var invalidRequest = struct {}{}

// serverCodec 三个阶段 明确了编解码的格式 开始具体的处理
// 1. 读取请求 readRequest 2. 处理请求 handleRequest 3. 回复请求
// sendResponse
func (server *Server) serverCodec(cc codec.Codec, opt *Option) {
    sending := new(sync.Mutex) // 处理请求是并发的，但是发送的时候得按顺序，不然可能会混淆数据
    wg := new(sync.WaitGroup)
    // 为什么这里是无限制循环 因为一次连接中允许接受多个请求，尽力而为，只有在
    // header解析失败（可能所有请求结束了），才终止循环
    for {
        req, err := server.readRequest(cc)
        if err != nil {
            if req == nil {
                break
            }
            req.h.Error = err.Error()
            server.sendResponse(cc, req.h, invalidRequest, sending) // 出错向客户端返回错误信息
            continue
        }
        wg.Add(1)
        // 把请求信息传入，处理请求 这里的这个timeout要注意，这里我们写死了，以后来改
        go server.handleRequest(cc, req, sending, wg, opt.HandleTimeout)
    }
    wg.Wait()
    _ = cc.Close()
}
```

读取请求

接收请求的第一步就收读取请求。根据上面的格式我们知道，请求分成两部分：请求头和请求体。我们分别来读取。

```
// readRequestHeader 读取请求头
func (server *Server) readRequestHeader(cc codec.Codec)
(*codec.Header, error) {
    var h codec.Header
    if err := cc.ReadHeader(&h); err != nil {
```

```

    if err != io.EOF && err != io.ErrUnexpectedEOF {
        log.Println("rpc server: read header error: ", err)
    }
    return nil, err
}
return &h, nil
}

// readRequest 读取请求，先读取请求头，再读取请求体
func (server *Server) readRequest(cc codec.Codec) (*request, error) {
    h, err := server.readRequestHeader(cc)
    if err != nil {
        return nil, err
    }
    req := &request{h: h}
    req.svc, req.mtype, err = server.findService(h.ServiceMethod)
    if err != nil {
        return req, err
    }
    // reflect.TypeOf 获取对应的Type
    // reflect.New 返回一个值，该值表示指向指定类型的新零值的指针，这里其实是设置成，指向string类型的指针

    req.argv = req.mtype.newArgv()
    req.replyv = req.mtype.newReplyv()

    argvi := req.argv.Interface()
    if req.argv.Type().Kind() != reflect.Ptr {
        // 返回一个指针
        argvi = req.argv.Addr().Interface()
    }
    if err = cc.ReadBody(argvi); err != nil {
        log.Println("rpc server: read argv err: ", err)
        return req, err
    }

    return req, nil
}

```

请求处理

其中包含了超时处理的逻辑。这里可以先忽略。请求处理就是要调用对应的方法，成功的话返回响应。

```

// handleRequest 处理请求，带有超时处理
func (server *Server) handleRequest(cc codec.Codec, req *request,
    sending *sync.Mutex, wg *sync.WaitGroup, timeout time.Duration) {
    defer wg.Done()
    // 利用called和sent两个chan来进行阻塞
    called := make(chan struct{})
    sent := make(chan struct{})

    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 { // 一直等待
    <-called
    <-sent
    return
}
select {
case <-time.After(timeout): //超出时间限制
    req.h.Error = fmt.Sprintf("rpc server: request handle timeout: expect
within %s", timeout)
    server.sendResponse(cc, req.h, invalidRequest, sending)
    // 注意, 这里存在内存泄露风险, 超时以后, 协程可能没有办法退出
case <-called:
    // 这里也存在问题, 如果called不超时, sent超时了
    <-sent
}
}

```

返回响应

返回响应的时候需要注意, 因为信息会写入到缓冲区中, 而这个缓冲区应该是互斥进行操作的。处理请求是并发的, 但是回复请求的报文必须是逐个发送的, 并发容易导致多个回复报文交织在一起, 客户端无法解析。在这里使用锁(sending)保证。

```

// sendResponse 回复
func (server *Server) sendResponse(cc codec.Codec, h *codec.Header,
body interface{}, sending *sync.Mutex) {
    // 因为开启了子线程去处理, 所以需要锁机制确保对缓冲区的互斥写
    sending.Lock()
    defer sending.Unlock()
    // 回复信息, Write方法调用了gob包中的encode方法,
    // encode方法用到了一个我们在gob结构体中定义的bufio.Writer缓冲区, 所以需要
    自己上锁
    if err := cc.Write(h, body); err != nil {
        log.Println("rpc server: write response error: ", err)
    }
}

```

服务注册

概述

RPC是远程过程调用，客户端发送给服务端的虽然是一串数据序列，但是最终还是要提取出相应的数据，找到对应的过程调用。那就是说，我们通过客户端给的“名字”，找到对应的方法，运行。

很容易想到用 `switch` 语句来实现。但是这样有个很严重的问题，他是硬编码的方式，每次服务的增加，都需要修改代码。编码起来更麻烦，工作量也很大。能不能用动态的实现方法，编写一份代码即可。

反射。

在Go中，RPC中对服务端提供的方法 `func (t *T) MethodName(argType T1, replyType *T2) error` 是有以下五点约束的：

- 方法的类型必须是外部可见的
- 方法必须是外部可见的
- 方法参数只能有两个，而且必须是外部可见的类型或者是基本类型
- 方法的第二个参数类型必须是指针
- 方法的返回值必须是error类型

我们可以把服务注册到服务器中，服务器解析对应的服务，看看有哪些方法，保存下来。后续客户端发送请求到服务端，寻找对应的服务及其方法，在处理请求的时候进行调用。

在读取请求的时候，请求头中就有**服务名.方法名**，这时候我们就需要先判断到底有没有这个服务和方法。有的话找出来，没有的话返回错误。

```
req.svc, req.mtype, err = server.findService(h.ServiceMethod)
if err != nil {
    return req, err
}
```

然后在处理请求的时候，需要调用这个服务和方法，就需要用到 `call` 函数。

```
err := req.svc.call(req.mtype, req.argv, req.replyv)
```

| 结构体定义

客户端发送的请求是**服务名.方法名、参数、响应**。因此，在这里，最主要的就是定义一个**服务**结构体和一个**方法**结构体。其中，一个服务可能有多个方法。

```

type service struct {
    name    string           // 映射的结构体的名称
    typ     reflect.Type     // 结构体的类型
    rcvr    reflect.Value    // 结构体的实例本身，反射的时候第0个参数
    method  map[string]*methodType // 存储映射的结构体的所有符合条件的方法
}

type methodType struct {
    method    reflect.Method // 方法本身
    ArgType   reflect.Type  // 第一个参数的类型
    ReplyType reflect.Type  // 第二个参数的类型
    numCalls  uint64        // 统计方法调用次数
}

```

基础方法

当把结构体定义出来以后，想想有哪些需要实现的。

- `numCalls` 字段，那需要实现一个方法，返回这个方法到底被调用多少次。在后面提供debug的demo的时候会用到。
- `methodType` 中记录了第一个参数的类型，即形参的类型。找到对应服务对应方法以后，便知道类型了。但是我们还需要一个实例，这个类型的实例，去接受客户端发来的参数，最后在方法调用 `call` 的时候使用。
- `methodType` 中记录了第一个参数的类型，即响应类型。同理，我们也需要这个类型的实例。
- 这里的重点就是服务注册，服务注册及其需要的判断逻辑（5个条件）也不能忘。
- 最后是需要进行方法的调用的，自然也需要实现相应的逻辑
- 同时，在服务端，需要调用这些方法，对这些方法进行封装。比如封装服务注册。
- 服务注册后应该把对应的**服务名.方法名**保存在一个字典中，后面需要的话就去这里面查

统计调用次数

可能涉及到并发调用，所以需要原子操作。

```

func (m *methodType) NumCalls() uint64 {
    // func LoadUint64(addr *uint64) (val uint64) 传入一个指向uint64值的指针。然后返回加载到*addr的值
    return atomic.LoadUint64(&m.numCalls)
}

```

把两个参数类型的实例构造出来

注意指针类型和值类型的操作是不一样的。然后响应的类型一定是指针类型。

```

// newArgv 返回指向参数类型的零值指针，即创建对应类型的实例
func (m *methodType) newArgv() reflect.Value {
    var argv reflect.Value
    // Type是类型 Kind是类别 Kind范围更大
    if m.ArgType.Kind() == reflect.Ptr {

```

```

    // m.ArgType.Elem() 取指针类型的元素类型，即ArgType代表的那个参数是什么类型的
    // reflect.New 返回指向指定类型的零值的指针
    // 因为m.ArgType是指针，不是需要先获取具体的类型，然后返回执行具体类型零值的指针给argv
    argv = reflect.New(m.ArgType.Elem())
} else {
    // 因为m.ArgType不是指针类型 是值类型 想通过New获取对应零值的实例指针，然后通过指针指定类型的零值
    argv = reflect.New(m.ArgType).Elem()
}
return argv
}

// newReplyv 创建对应类型的实例
func (m *methodType) newReplyv() reflect.Value {
    // Reply 一定是一个指针类型
    replyv := reflect.New(m.ReplyType.Elem())
    switch m.ReplyType.Elem().Kind() {
    case reflect.Map:
        // Set 把一个x(形参)中的数据赋值到v(调用者)
        replyv.Elem().Set(reflect.MakeMap(m.ReplyType.Elem()))
    case reflect.Slice:
        replyv.Elem().Set(reflect.MakeSlice(m.ReplyType.Elem(), 0, 0))
    }
    return replyv
}

```

服务注册及逻辑判断

注册服务

需要判断服务的类型是否是外部可见的。

```

func newService(rcvr interface{}) *service {
    s := new(service)
    // 获得值的反射值对象, 包含有rcvr的值信息
    s.rcvr = reflect.ValueOf(rcvr)
    // Indirect返回v指向的值，如果v是个nil指针，Indirect返回0值，如果v不是指针，Indirect返回v本身
    s.name = reflect.Indirect(s.rcvr).Type().Name()
    s.typ = reflect.TypeOf(rcvr)
    // 通过检查抽象语法树，看对应名称的结构体是否是导出的（方法的类型是外部可见的）
    if !ast.IsExported(s.name) {
        log.Fatalf("rpc server: %s is not a valid service name", s.name)
    }
    s.registerMethods()
    return s
}

```

注册方法

注册方法需要满足的是

- 方法是外部可见的
- 方法有两个参数，外部可见或者基本类型
- 方法的第二个参数必须是指针
- 方法的返回值必须是error类型

满足这些条件的方法，才可以进行注册。这里有一点需要注意，就是方法的参数有两个，但是反射的时候使用 `NumIn()` 方法得到的是3个。这是因为第 0 个是自身，类似于 python 的 `self`，java 中的 `this`。其实也说的过去，毕竟方法的类型那里，很像 `this`。

```
// 注册方法，实现结构体和服务的映射
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
        // 符合条件的方法需要满足
        // 两个导出或内置类型的入参（反射时为 3 个，第 0 个是自身，类似于 python 的
        self, java 中的 this）
        // 返回值有且只有 1 个，类型为 error
        if mType.NumIn() != 3 || mType.NumOut() != 1 {
            continue
        }
        if mType.Out(0) != reflect.TypeOf((*error)(nil)).Elem() {
            continue
        }
        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", s.name, method.Name)
    }
}

// isExportedOrBuiltinType 判断是否导出或者内置类型
func isExportedOrBuiltinType(t reflect.Type) bool {
    // PkgPath返回包名，代表这个包的唯一标识符，所以可能是单一的包名 包名为空 内
    置类型
    return ast.IsExported(t.Name()) || t.PkgPath() == ""
}
```

方法调用

```
// call 实现通过反射值调用方法
func (s *Service) call(m *MethodType, argv, replyv reflect.Value) error {
    atomic.AddUint64(&m.numCalls, 1)
    f := m.method.Func
    // 传入参数，第一个是本身 类似Java的this，第二个是形参，第三个是响应值 最后返回函数运行结果error
    returnValues := f.Call([]reflect.Value{s.rcvr, argv, replyv})
    if errInter := returnValues[0].Interface(); errInter != nil {
        return errInter.(error)
    }
    return nil
}
```

服务端的封装

```
func (server *Server) Register(rcvr interface{}) error {
    s := newService(rcvr)
    // dup是true表示loaded
    if _, dup := server.serviceMap.LoadOrStore(s.name, s); dup {
        return errors.New("rpc: service already defined: " + s.name)
    }
    return nil
}

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

// findService ServiceMethod 的构成是 "Service.Method"
// 先在serviceMap 中找到对应的 service 实例，再从 service 实例的 method 中，找到对应的 methodType。
func (server *Server) findService(serviceMethod string) (svc *Service, mtype *MethodType, err error) {
    dot := strings.LastIndex(serviceMethod, ".")
    if dot < 0 {
        err = errors.New("rpc server: server/method request ill-formed: " + serviceMethod)
        return
    }
    serviceName, methodName := serviceMethod[:dot], serviceMethod[dot+1:]
    svci, ok := server.serviceMap.Load(serviceName)
    if !ok {
        err = errors.New("rpc server: can't find service " + serviceName)
        return
    }
    svc = svci.(*Service)
    mtype = svc.method[methodName]
    if mtype == nil {
        err = errors.New("rpc server: can't find method " + methodName)
    }
}
```

```
return
}
```

客户端

到这，服务端最最基础的部分就完成了。这个时候与之相对应的，就需要有一个客户端。

概述

客户端主要做的就是，发送请求和接收请求。前面提到请求的格式如下：

```
/*
    | Option(Json) | Requesse(Codec) | --> | Option(Json) |
Header(Codec) | Body(Codec) |
*/
```

只需要发送一次协商信息即可，后面该客户端的所有请求都按照这个来。因为一次协商，即一个客户端只需要一次协商，可以发起多次RPC调用。

结构体定义

从概述可以发现，我们需要把客户端和RPC请求分开来定义。客户端只有一个，RPC请求可能有多次。

```
type Client struct {
    cc      codec.Codec    // 编码解码器，用来序列化将要发送出去的请求，以及反
序列化接收到的响应
    opt     *Option      // 与服务端的协商信息
    header  codec.Header // 请求的消息头，只有在请求发送的时候才需要，而请
求发送是互斥的，因此每个客户端只需要一个，可复用
    pending map[uint64]*Call // 存储未处理完的请求，键是编号，值是Call实例
    sending sync.Mutex   // 保证请求的有序发送，防止出现多个请求报文混淆
    mu      sync.Mutex   // 客户端的互斥锁
    seq     uint64      // 给发送的请求编号，每个请求拥有唯一编号
    closing bool       // 用户主动关闭
    shutdown bool      // 一般是有错误发送
}
```

```
// Call 一次RPC调用需要的信息
type Call struct {
    Seq      uint64
    ServiceMethod string // 需要调用的函数，格式是service.method
    Args     interface{} // 形参
    Reply    interface{} // 响应
    Error    error      // 错误信息
    Done     chan *Call // 同步接口使用，结束标志
}
```

```
// DefaultOption 默认采用Gob编码方式
var DefaultOption = &Option{
    MagicNumber:    MagicNumber,
    CodecType:      codec.GobType,
    ConnectTimeout: time.Second * 10,
}
```

客户端基础操作

首先要设计客户端的基础操作：

- 创建客户端
- 客户端协商信息的确定
- 客户端是否在工作
- 客户端关闭
- 存储未处理完的请求
- 移出已处理完的请求
- 错误发生时关闭客户端

创建客户端

```
// NewClient 创建Client实例，首先需要完成协议交换，然后再创建子线程调用
// receive()接收响应
func NewClient(conn net.Conn, opt *Option) (*Client, error) {
    f := codec.NewCodecFuncMap[opt.CodecType]
    if f == nil {
        err := fmt.Errorf("invalid codec type %s", opt.CodecType)
        log.Println("rpc client: codec error: ", err)
        return nil, err
    }
    // 发送协议给服务端
    if err := json.NewEncoder(conn).Encode(opt); err != nil {
        log.Println("rpc client: options error: ", err)
        _ = conn.Close()
        return nil, err
    }
    return newClientCodec(f(conn), opt), nil
}

// newClientCodec 创建客户端，开始处理
func newClientCodec(cc codec.Codec, opt *Option) *Client {
    client := &Client{
        cc:    cc,
        opt:   opt,
        pending: make(map[uint64]*Call),
        seq:    1, // 从1开始，0表示无效
    }
    go client.receive()
    return client
}
```

协商信息

设置为可选的，不选就用默认的。

```
// parseOptions 用户确定协商信息，这里实现为可选参数，以便用户不设置可以默认
func parseOptions(opts ...*Option) (*Option, error) {
    if len(opts) == 0 || opts[0] == nil {
        return DefaultOption, nil
    }
    if len(opts) != 1 {
        return nil, errors.New("number of options is more than 1")
    }
    opt := opts[0]
    opt.MagicNumber = DefaultOption.MagicNumber
    if opt.CodecType == "" {
        opt.CodecType = DefaultOption.CodecType
    }
    return opt, nil
}
```

客户端是否在工作

```
// IsAvailable 看客户端是否还在工作
func (client *Client) IsAvailable() bool {
    client.mu.Lock()
    defer client.mu.Unlock()
    return !client.shutdown && !client.closing
}
```

客户端连接关闭

```
// Close 关闭连接
func (client *Client) Close() error {
    client.mu.Lock()
    defer client.mu.Unlock()
    if client.closing {
        return ErrShutdown
    }
    client.closing = true
    return client.cc.Close()
}
```

维护未处理完的请求


```
// registerCall 注册请求，将参数Call添加到client.pending中，并更新client.seq
func (client *Client) registerCall(call *Call) (uint64, error) {
    client.mu.Lock()
    defer client.mu.Unlock()
    if client.closing || client.shutdown {
        return 0, ErrShutdown
    }
    call.Seq = client.seq
    // 注册请求，按照编号来
    client.pending[call.Seq] = call
    client.seq++
    return call.Seq, nil
}
```

移出已处理完的请求

```
// removeCall 根据seq从client.pending中移除对应的Call并返回
func (client *Client) removeCall(seq uint64) *Call {
    client.mu.Lock()
    defer client.mu.Unlock()
    call := client.pending[seq]
    delete(client.pending, seq)
    return call
}
```

错误发生终止客户端

```
// terminateCalls 服务端或客户端发生错误时调用，将shutdown设置为true，且将错误信息通知所有pending状态的Call
func (client *Client) terminateCalls(err error) {
    client.sending.Lock()
    defer client.sending.Unlock()
    client.mu.Lock()
    defer client.mu.Unlock()
    client.shutdown = true
    for _, call := range client.pending {
        call.Error = err
        call.done()
    }
}
```

| RPC请求操作

在这无非就是三个操作：

- 调用（用户指定需要调用的远程方法）
- 发送（准备好数据，发送给服务端）
- 接收（接收服务端返回的响应）

调用

这里有两个方法，`Go` 和 `Call`。

- `Go` 和 `Call` 是客户端暴露给用户的两个 RPC 服务调用接口，`Go` 是一个异步接口，返回 `call` 实例。
- `Call` 是对 `Go` 的封装，阻塞 `call.Done`，等待响应返回，是一个同步接口。

好像Go中很多同步接口都是异步接口的封装，只是改一下channel中的缓冲区即可。

先看异步的 `Go` 方法：

```
// Go 返回调用的Call结构，没有阻塞，使其能够异步调用
func (client *Client) Go(serviceMethod string, args, reply interface{},
done chan *Call) *Call {
    if done == nil {
        done = make(chan *Call, 10)
    } else if cap(done) == 0 {
        log.Panic("rpc client : done channel is unbuffered")
    }
    call := &Call{
        ServiceMethod: serviceMethod,
        Args:          args,
        Reply:         reply,
        Done:         done,
    }
    client.send(call)
    return call
}
```

同步的 `Call` 只是对其进行封装。

```
// Call 带有超时处理，使用context包实现，控制权交给用户，控制更为灵活
// Call 调用对应的函数，等待完成，返回错误信息，阻塞call.Done，等待响应返回，是一个同步接口
// context主要就是用来在多个goroutine中设置截至日期，同步信号，传递请求相关值
// 他和WaitGroup的作用类似，但是更强大
https://www.cnblogs.com/failymao/p/15565326.html
func (client *Client) Call(ctx context.Context, serviceMethod string, args,
reply interface{}) error {
    call := client.Go(serviceMethod, args, reply, make(chan *Call, 1))
    select {
        // 返回一个 channel，用于判断 context 是否结束，多次调用同一个 context
        // done 方法会返回相同的 channel
        case <-ctx.Done():
            client.removeCall(call.Seq)
            return errors.New("rpc client: call failed: " + ctx.Err().Error())
        case call := <-call.Done:
            return call.Error
    }
}
```

发送

```
// send 发送请求
func (client *Client) send(call *Call) {
    client.sending.Lock()
    defer client.sending.Unlock()

    // 注册请求
    seq, err := client.registerCall(call)
    if err != nil {
        call.Error = err
        call.done()
    }

    // 准备请求头 因为互斥发送 客户端可以复用
    client.header.ServiceMethod = call.ServiceMethod
    client.header.Seq = seq
    client.header.Error = ""

    // 编码和发送请求--请求头和请求体
    // 不是发送请求体吗？为什么只发送了参数      响应类型服务端自己能解析出来
    if err := client.cc.Write(&client.header, call.Args); err != nil {
        call := client.removeCall(seq)
        if call != nil {
            call.Error = err
            call.done()
        }
    }
}
```

接受

接受分好几种情况，需要考虑出错的可能。

```
/*
    接收可能出现的情况：
    1. Call不存在，可能是请求没有发送完整，或者因为其他原因取消了，但是服务端仍旧
    处理了（客户端出问题）
    2. Call存在，服务端处理出错（服务端出问题）
    3. 正常
*/

// receive 接收响应
func (client *Client) receive() {
    var err error
    for err == nil {
        var h codec.Header
        if err = client.cc.ReadHeader(&h); err != nil {
            break
        }
        call := client.removeCall(h.Seq)
        switch {
```

```

    case call == nil: // 客户端的Call列表中没有这个请求。可能是请求没有发送完整，或者因为其他原因被取消，但是服务端仍旧处理了
        err = client.cc.ReadBody(nil)
    case h.Error != "": // call存在，但服务端处理出错
        call.Error = fmt.Errorf(h.Error)
        err = client.cc.ReadBody(nil)
        call.done()
    default: // 正常情况
        err = client.cc.ReadBody(call.Reply)
        if err != nil {
            call.Error = errors.New("reading body" + err.Error())
        }
        call.done()
    }
}
client.terminateCalls(err)
}

```

到这，一个简单可用的RPC框架便完成了。下面还可以添加一些高级的特性。

超时处理

概述

超时处理分两部分。客户端超时和服务端超时。

客户端超时有：

- 创建连接超时
 - 连接的时候超时
 - 协商信息交换的时候超时
- call调用超时
 - 发送报文超时
 - 等待处理超时
 - 接收报文超时

服务端超时有：

- 请求处理超时
 - call调用超时
 - 发送响应超时

结构体定义

之前在前面的结构体定义中，把超时处理的部分定义好了，这里不再赘述。这里需要定义的是，具备超时处理能力的客户端结构体。因为超时处理的话，就意味着你需要开协程进行处理原来的逻辑，在主协程中超时处理。不然的话，超时处理逻辑在具体执行逻辑后面的话，超时了也处理不了，因为运行不到那个地方。

一般超时处理会利用select、context、channel和协程进行协同处理。

```

type clientResult struct {
    client *Client
    err    error
}

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

```

| 客户端超时处理

创建连接超时

```

// dialTimeout 能处理超时的连接请求：这里处理了两个超时问题，第一个是连接的时候
// 超时，第二个是协议交换时候的超时
func dialTimeout(f newClientFunc, network, address string, opts
...*Option) (client *Client, err error) {
    // 生成协商信息
    opt, err := parseOptions(opts...)
    if err != nil {
        return nil, err
    }
    // 连接超时处理 利用net包中自带的API
    conn, err := net.DialTimeout(network, address, opt.ConnectTimeout)
    if err != nil {
        return nil, err
    }
    // 出错，最后记得关闭连接
    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是对信道的操作，匹配的case随机选择一个执行，不匹配会阻塞，所以要注意
    // select的超时处理
    // 协议交换超时处理
    select {
    case <-time.After(opt.ConnectTimeout): // 超时处理 开一个定时器
        return nil, fmt.Errorf("rpc client: connect timeout: expect within %s",
opt.ConnectTimeout)
    case result := <-ch:
        return result.client, result.err
    }
}

```

```
// Dial 带有超时处理的连接请求 封装，向上屏蔽具体的连接过程
func Dial(network, address string, opts ...*Option) (*Client, error) {
    return dialTimeout(NewClient, network, address, opts...)
}
```

call调用超时

这里对 `Call` 方法做了一些修改。不把channel的缓冲区写死。然后客户端call调用的超时，包括了发送报文、等待处理、接收报文的全过程，时间的设置取决于客户端自己的设置。通过context来表示。

```
// Call 带有超时处理，使用context包实现，控制权交给用户，控制更为灵活
// Call 调用对应的函数，等待完成，返回错误信息，阻塞call.Done，等待响应返回，是一个同步接口
// context主要就是用来在多个goroutine中设置截止日期，同步信号，传递请求相关值
// 他和WaitGroup的作用类似，但是更强大
https://www.cnblogs.com/failymao/p/15565326.html
func (client *Client) Call(ctx context.Context, serviceMethod string, args,
reply interface{}, buffSize int) error {
    call := client.Go(serviceMethod, args, reply, make(chan *Call,
buffSize)) // 同步不应该没有缓冲区吗
    select {
        // 返回一个 channel，用于判断 context 是否结束，多次调用同一个 context
        // done 方法会返回相同的 channel
        case <-ctx.Done():
            client.removeCall(call.Seq)
            return errors.New("rpc client: call failed: " + ctx.Err().Error())
        case call := <-call.Done:
            return call.Error
    }
}
```

服务端超时处理

请求处理超时

这里也包括两部分，call调用超时和发送响应信息超时。处理方法类似。

```
// handleRequest 处理请求，带有超时处理
func (server *Server) handleRequest(cc codec.Codec, req *request,
sending *sync.Mutex, wg *sync.WaitGroup, timeout time.Duration) {
    defer wg.Done()
    // 利用called和sent两个chan来进行阻塞
    called := make(chan struct{})
    sent := make(chan struct{})

    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 { // 一直等待
    <-called
    <-sent
    return
}
select {
case <-time.After(timeout): //超出时间限制
    req.h.Error = fmt.Sprintf("rpc server: request handle timeout: expect within %s", timeout)
    server.sendResponse(cc, req.h, invalidRequest, sending)
    // 注意, 这里存在内存泄露风险, 超时以后, 协程可能没有办法退出
case <-called:
    // 这里也存在问题, 如果called不超时, sent超时了
    <-sent
}
}

```

RPC over HTTP

概述

RPC可以使用HTTP协议也可以使用TCP协议。具体的可以看问题解释。这里说一下好处：可以使用浏览器访问，支持跨语言，不同路径可以提供不同的服务。

同样，也需要客户端和服务端的支持。两者都需要做出修改以支持HTTP协议。

这里还需要说一下。标准库net/rpc提供了**RPC over HTTP** 和 **RPC over TCP** 两种方式。唯一的区别就是建立连接时的区别，实际的**RPC over HTTP**也并没有使用http协议，只是用http server建立连接而已。

路径定义

首先要先定义提供服务的路径。

```

const (
    connected      = "200 Connected to MyRPC"
    defaultRPCPath = "/_myrpc_"
    defaultDebugPath = "/debug/myrpc"
)

```

服务端具体实现

使用HTTP，其实不可避免的要使用go自带的HTTP包。需要使用HTTP包中的Handle函数进行注册。

```
func Handle(pattern string, handler Handler) {
    DefaultServeMux.Handle(pattern, handler) }
```

他需要两个参数，一个是提供服务的路径名，一个是Handler类型接口的实现。

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

具体的处理逻辑，就在ServeHTTP方法中。因此，在这我们只需要给服务端实现这个方法即可。其实只有建立连接的时候用到了HTTP协议，后面的具体处理，还是使用前面定义的 **ServerConn** 方法。

```
// ServeHTTP 实现一个响应 RPC 请求的 http.Handler    ServeHTTP 应该将回复
// 头和数据写入 ResponseWriter 然后返回。
func (server *Server) ServeHTTP(w http.ResponseWriter, req
    *http.Request) {
    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
    }
    // Hijack()可以将HTTP对应的TCP连接取出，连接在Hijack()之后，HTTP的相关操
    // 作就会受到影响，调用方需要负责去关闭连接。
    conn, _, err := w.(http.Hijacker).Hijack()
    if err != nil {
        log.Print("rpc hijacking", req.RemoteAddr, ": ", err.Error())
        return
    }
    _, _ = io.WriteString(conn, "HTTP/1.0 "+connected+"\n\n")
    server.ServerConn(conn)
}
```

服务端还需要进行HTTP注册。

```
// HandleHTTP 为rpcPath上的RPC消息注册一个HTTP处理程序
// 实际上HandleHTTP就是使用http包的功能，将server自身注册到http的url映射上了
func (server *Server) HandleHTTP() {
    // 第一个参数是访问路径 第二个参数是Handler类型 一个接口 需要实现
    // ServerHTTP
    http.Handle(defaultRPCPath, server)
    http.Handle(defaultDebugPath, debugHTTP{server})
    log.Println("rpc server debug path:", defaultDebugPath)
}
```

在打开服务器的时候，建立连接以后再调用它即可。

客户端具体实现

客户端这边需要改动的就比较大，我们需要重新写一个客户端创建的方法，以及封装创建HTTP连接的 `Dial` 方法。但实际主函数中调用的时候改动却不大，只需要把原来的 `Dial` 改成 `DialHTTP` 即可。

```
//
// 客户端支持HTTP协议
//
// 支持 HTTP 协议的好处在于，RPC 服务仅仅使用了监听端口的 /_geerpc 路径，在其他路径上我们可以提供诸如日志、统计等更为丰富的功能。
//

// NewHTTPClient 创建通过HTTP连接的客户端
func NewHTTPClient(conn net.Conn, opt *Option) (*Client, error) {
    _, _ = io.WriteString(conn, fmt.Sprintf("CONNECT %s HTTP/1.0\n\n",
        defaultRPCPath))

    // 在转换成RPC协议之前 需要获得HTTP正确的响应
    // ReadResponse 发送Request 从 bufio.NewReader(conn) 读取并返回一个
    HTTP 响应
    resp, err := http.ReadResponse(bufio.NewReader(conn), &http.Request{
        Method: "CONNECT",
    })
    if err == nil && resp.Status == connected {
        return NewClient(conn, opt)
    }
    if err == nil {
        err = errors.New("unexpected HTTP response: " + resp.Status)
    }
    return nil, err
}
```

定义创建HTTP连接的封装。

```
// DialHTTP 创建HTTP连接
func DialHTTP(network, address string, opts ...*Option) (*Client, error) {
    return dialTimeout(NewHTTPClient, network, address, opts...)
}
```

这里提供一个统一的连接调用。

```
// XDial 简化调用 提供一个统一入口XDial。rpcAddr是一个通用格式
(protocol@addr)
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:
    return Dial(protocol, addr, opts...)
}
```

负载均衡

概述

要想实现负载均衡，就需要先实现服务发现，客户端要用到。因为我们需要获得服务列表，才能有的选，进一步才可以实现负载均衡。服务发现分为**客户端模式服务发现**和**服务端模式服务发现**。具体可以看问题解释。我们这里采用的是**客户端模式服务发现**。

服务发现模块起码包含以下四个功能：

- 刷新服务列表
- 更新服务列表
- 根据负载均衡策略选择服务实例
- 返回所有的访问实例

实现完服务发现以后，再根据负载均衡策略，实现负载均衡。因为我们使用的是**客户端模式服务发现**，因此只需要修改客户端的实现逻辑，编写一个支持负载均衡的客户端即可。

服务发现

结构体和接口定义

```
type SelectMode int // 代表不同负载均衡策略

const (
    RandomSelect    SelectMode = iota // 随机选择策略
    RoundRobinSelect // 轮询算法
)

// Discovery 包含服务发现所需要的最基本的接口
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) // 根据负载均衡策略，选择一个服务实例
    GetAll() ([]string, error) // 返回所有的服务实例
}

// MultiServersDiscovery 实现一个不需要注册中心，服务列表由手工维护的服务发现的结构体
type MultiServersDiscovery struct {
    r      *rand.Rand // 生成随机数
    mu     sync.RWMutex // 互斥访问控制
    servers []string // 服务列表
    index  int // 记录轮询算法已经选择的索引
}
```

手工维护的服务发现

新建服务发现实例

```
func NewMultiServerDiscovery(servers []string) *MultiServersDiscovery {
    d := &MultiServersDiscovery{
        // r 是一个产生随机数的实例，初始化时使用时间戳设定随机数种子，避免每次产生
        // 相同的随机数序列。
        r:      rand.New(rand.NewSource(time.Now().UnixNano())),
        servers: servers,
    }
    // index 记录 Round Robin 算法已经轮询到的位置，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)
    return d
}
```

接口方法的实现

```
// Refresh 刷新对 MultiServersDiscovery 没有意义，所以忽略它(因为他是手动维护
// 的)
func (d *MultiServersDiscovery) Refresh() error {
    return nil
}

// Update 更新服务列表
func (d *MultiServersDiscovery) Update(servers []string) error {
    d.mu.Lock()
    defer d.mu.Unlock()
    d.servers = servers
    return nil
}

func (d *MultiServersDiscovery) Get(mode SelectMode) (string, error) {
    d.mu.Lock()
    defer d.mu.Unlock()
    n := len(d.servers)
    if n == 0 {
        return "", errors.New("rpc discovery: no available servers")
    }
    switch mode {
    case RandomSelect:
        return d.servers[d.r.Intn(n)], nil
    case RoundRobinSelect:
        s := d.servers[d.index%n]
        d.index = (d.index + 1) % n
        return s, nil
    default:
        return "", errors.New("rpc discovery: not supported select mode")
    }
}

func (d *MultiServersDiscovery) GetAll() ([]string, error) {
    d.mu.RLock()
```

```

defer d.mu.RUnlock()
servers := make([]string, len(d.servers), len(d.servers))
copy(servers, d.servers)
return servers, nil
}

```

自动维护的服务发现

因为自动维护的服务发现需要用到注册中心，这个等下再说。但是大致操作和手动维护是差不多的。主要是需要引入服务的维护，有时间限制。客户端需要获取服务地址，然后每次会调用服务发现的相关方法去获取，获取之前相关逻辑会先看是否需要刷新服务列表，而这个操作就和时间有关。这里和注册中心通信，用的是HTTP协议。

结构体的定义

```

// 带有注册中心的服务发现

type MyRegistryDiscovery struct {
    *MultiServersDiscovery
    registry string // 注册中心地址
    timeout  time.Duration // 服务列表的过期时间
    lastUpdate time.Time // 代表最后从注册中心更新服务列表的时间，默认 10s
    // 过期，即 10s 之后，需要从注册中心更新新的列表
}

const defaultUpdateTimeout = time.Second * 10

```

新建服务发现实例

```

func NewMyRegistryDiscovery(registerAddr string, timeout
time.Duration) *MyRegistryDiscovery {
    if timeout == 0 {
        timeout = defaultUpdateTimeout
    }
    d := &MyRegistryDiscovery{
        MultiServersDiscovery: NewMultiServerDiscovery(make([]string, 0)),
        registry:              registerAddr,
        timeout:               timeout,
    }
    return d
}

```

接口方法实现

```

// Update 更新服务中心的服务列表
func (d *MyRegistryDiscovery) Update(servers []string) error {
    d.mu.Lock()
    defer d.mu.Unlock()
    d.servers = servers
    d.lastUpdate = time.Now()
    return nil
}

```

```

// Refresh 刷新本地的服务列表
func (d *MyRegistryDiscovery) Refresh() error {
    d.mu.Lock()
    defer d.mu.Unlock()
    // 没超时
    if d.lastUpdate.Add(d.timeout).After(time.Now()) {
        return nil
    }
    log.Println("rpc registry: refresh servers from registry", d.registry)
    resp, err := http.Get(d.registry)
    if err != nil {
        log.Println("rpc registry refresh err:", err)
        return err
    }
    servers := strings.Split(resp.Header.Get("X-Myrpc-Servers"), ",")
    d.servers = make([]string, 0, len(servers))
    for _, server := range servers {
        if strings.TrimSpace(server) != "" {
            d.servers = append(d.servers, strings.TrimSpace(server))
        }
    }
    d.lastUpdate = time.Now()
    return nil
}

func (d *MyRegistryDiscovery) Get(mode SelectMode) (string, error) {
    // 先确保服务列表没有过期
    if err := d.Refresh(); err != nil {
        return "", err
    }
    return d.MultiServersDiscovery.Get(mode)
}

func (d *MyRegistryDiscovery) GetAll() ([]string, error) {
    if err := d.Refresh(); err != nil {
        return nil, err
    }
    return d.MultiServersDiscovery.GetAll()
}

```

负载均衡策略

负载均衡有很多的策略，比如

- 随机选择策略 - 从服务列表中随机选择一个
- 轮询算法(Round Robin) - 依次调度不同的服务器，每次调度执行 $i = (i + 1) \text{ mode } n$
- 加权轮询(Weight Round Robin) - 在轮询算法的基础上，为每个服务实例设置一个权重，高性能的机器赋予更高的权重，也可以根据服务实例的当前的负载情况做动态的调整，例如考虑最近5分钟部署服务器的 CPU、内存消耗情况
- 哈希/一致性哈希策略 - 依据请求的某些特征，计算一个 hash 值，根据 hash 值将请求发送到对应的机器。一致性 hash 还可以解决服务实例动态添加情况下，调度

抖动的问题。一致性哈希的一个典型应用场景是分布式缓存服务。

这些策略是发生在选择服务的时候，因此在服务发现的时候就会用到，在 `Get` 方法中使用。

随机选择策略

```
case RandomSelect:
    return d.servers[d.r.Intn(n)], nil
```

轮询算法

```
case RoundRobinSelect:
    s := d.servers[d.index%n]
    d.index = (d.index + 1) % n
    return s, nil
```

| 负载均衡客户端

因为采用的是客户端模式的服务发现，实际流程就是，客户端向注册中心请求服务列表，然后自己调用服务发现逻辑选择一个服务进行连接。因此我们需要修改客户端的实现。

结构体定义

```
type XClient struct {
    d      Discovery
    mode   SelectMode
    opt    *MyRPC.Option
    mu     sync.Mutex
    clients map[string]*MyRPC.Client // 键是服务器的IP 值是与该IP服务器连接的客户端
}
```

方法实现

```
func NewXClient(d Discovery, mode SelectMode, opt *MyRPC.Option)
*XClient {
    return &XClient{
        d:      d,
        mode:   mode,
        opt:    opt,
        mu:     sync.Mutex{},
        clients: make(map[string]*MyRPC.Client),
    }
}
```

```

func (xc *XClient) Close() error {
    xc.mu.Lock()
    defer xc.mu.Unlock()
    for key, client := range xc.clients {
        _ = client.Close()
        delete(xc.clients, key)
    }
    return nil
}

```

```

func (xc *XClient) dial(rpcAddr string) (*MyRPC.Client, error) {
    xc.mu.Lock()
    defer xc.mu.Unlock()
    client, ok := xc.clients[rpcAddr]
    // 已经由存在的连接 不可用 关闭
    if ok && !client.IsAvailable() {
        _ = client.Close()
        delete(xc.clients, rpcAddr)
        client = nil
    }
    // 没有缓存的客户端
    if client == nil {
        var err error
        client, err = MyRPC.XDial(rpcAddr, xc.opt)
        if err != nil {
            return nil, err
        }
        xc.clients[rpcAddr] = client
    }
    // 返回缓存客户端
    return client, nil
}

```

```

func (xc *XClient) call(rpcAddr string, ctx context.Context, serviceMethod
string, args, reply interface{}) error {
    client, err := xc.dial(rpcAddr)
    if err != nil {
        return err
    }
    return client.Call(ctx, serviceMethod, args, reply, 1)
}

func (xc *XClient) Call(ctx context.Context, serviceMethod string, args,
reply interface{}) error {
    rpcAddr, err := xc.d.Get(xc.mode)
    if err != nil {
        return err
    }
    return xc.call(rpcAddr, ctx, serviceMethod, args, reply)
}

```

还提供了一个广播功能，Broadcast 将请求广播到所有的服务实例，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。

```
// Broadcast 将请求广播到所有的服务实例
func (xc *XClient) Broadcast(ctx context.Context, serviceMethod string,
args, reply interface{}) error {
    servers, err := xc.d.GetAll()
    if err != nil {
        return err
    }
    var wg sync.WaitGroup
    var mu sync.Mutex
    var e error
    replyDone := reply == nil // 如果reply是nil的话，不需要设置值
    ctx, cancel := context.WithCancel(ctx)
    for _, rpcAddr := range servers {
        wg.Add(1)
        go func(rpcAddr string) {
            defer wg.Done()
            var clonedReply interface{}
            if reply != nil {
                clonedReply =
reflect.New(reflect.ValueOf(reply).Elem().Type()).Interface()
            }
            err := xc.call(rpcAddr, ctx, serviceMethod, args, clonedReply)
            mu.Lock()
            if err != nil && e == nil {
                e = err
                cancel() // 实例发生错误，则返回其错误
            }
            if err == nil && !replyDone {
                reflect.ValueOf(reply).Elem().Set(reflect.ValueOf(clonedReply).Elem())
                // 某个实例调用成功，返回，其他的实例不需要返回
                replyDone = true
            }
            mu.Unlock()
        }(rpcAddr)
    }
    wg.Wait()
    return e
}
```

这里有个点需要解释一下。一开始我有点疑惑，客户端要保存一个IP地址 --> 客户端的字典。负载均衡选的是客户端？？？

主要是之前注释打错导致的（脑残了把客户端打成服务端）。实现了负载均衡的系统，其实就像一个分布式系统。不管我向哪台机器请求访问，最终得到的结果都应该是一致的。而当我并发执行的时候，可能会向不同的服务器访问，那就需要和一个新的服务器建立连接，就需要重新开始协商那些过程。所以这里复用原来的Client。同时也要记录下来，那个服务器对应哪个客户端。

如果不这样做的话，这里能实现的并发，也只不过是一个连接多次请求间的并发，一个客户端固定连着那个服务端。可能某个服务端压力很大了还是连着他，效率就没法提升。所以用这样一个IP地址 --> 客户端的字典，能实现更大力度的并发。

注册中心

概述

上面提了自动维护的话，是需要注册中心的。注册中心需要实现的功能有：

- 添加服务实例，如果服务已经存在，则更新start
- 给客户端返回可用的服务列表，如果存在超时的服务，则删除
- 让服务器定时给注册中心发送心跳信息
- 注册HTTP。注册中心和客户端、服务器之间采用的是HTTP协议进行通信的

结构体定义

```
type MyRegistry struct {
    timeout time.Duration //默认5分钟，任何注册的服务超过5分钟，都视为不可用
    mu      sync.Mutex
    servers map[string]*ServerItem
}

type ServerItem struct {
    Addr string
    start time.Time
}

const (
    defaultPath    = "/_geerpc_/registry"
    defaultTimeout = time.Minute * 5
)
```

基础方法实现

构造函数实现

```
func New(timeout time.Duration) *MyRegistry {
    return &MyRegistry{
        timeout: timeout,
        servers: make(map[string]*ServerItem),
    }
}

var DefaultMyRegister = New(defaultTimeout)
```

添加服务实例，如果服务已经存在，则更新start

```
// putServer 添加服务实例，如果服务已经存在，则更新start
func (r *MyRegistry) putServer(addr string) {
    r.mu.Lock()
    defer r.mu.Unlock()
    s := r.servers[addr]
    if s == nil {
        r.servers[addr] = &ServerItem{
            Addr:  addr,
            start: time.Now(),
        }
    } else {
        s.start = time.Now() // 更新时间，心跳信息
    }
}
```

给客户端返回可用的服务列表，如果存在超时的服务，则删除

```
// 给客户端返回可用的服务列表，如果存在超时的服务，则删除
func (r *MyRegistry) aliveServers() []string {
    r.mu.Lock()
    defer r.mu.Unlock()
    var alive []string
    for addr, s := range r.servers {
        if r.timeout == 0 || s.start.Add(r.timeout).After(time.Now()) {
            alive = append(alive, addr)
        } else {
            delete(r.servers, addr)
        }
    }
    sort.Strings(alive)
    return alive
}
```

注册HTTP

```
// MyRegistry 采用HTTP协议
func (r *MyRegistry) ServeHTTP(w http.ResponseWriter, req
*http.Request) {
    switch req.Method {
    case "GET": // 返回所有可用的服务列表
        w.Header().Set("X-Myrpc-Servers", strings.Join(r.aliveServers(), ","))
    case "POST": // 添加服务实例或发送心跳
        addr := req.Header.Get("X-Myrpc-Server")
        if addr == "" {
            w.WriteHeader(http.StatusInternalServerError)
            return
        }
        r.putServer(addr)
    default:
        w.WriteHeader(http.StatusMethodNotAllowed)
    }
}
```

```

    }
}

func (r *MyRegistry) HandleHTTP(registryPath string) {
    http.Handle(registryPath, r)
    log.Println("rpc registry path:", registryPath)
}

func HandleHTTP() {
    DefaultMyRegister.HandleHTTP(defaultPath)
}

```

服务端心跳信息发送

需要创建定时器。定时器会自动创建一个channel来阻塞，到时间了就会发送信息给channel。

```

//
// 服务端向注册中心发送心跳信息
//

// Heartbeat 方法，便于服务启动时定时向注册中心发送心跳，默认周期比注册中心设置的过期时间少 1 min。
func (server *Server) Heartbeat(registry, addr string, duration time.Duration) {
    if duration == 0 {
        duration = defaultTimeout - time.Duration(1)*time.Minute
    }
    var err error
    err = sendHeartbeat(registry, addr)
    go func() {
        // time.NewTicker 创建周期性定时器
        t := time.NewTicker(duration)
        for err == nil {
            // 从定时器中获取数据
            <-t.C
            err = sendHeartbeat(registry, addr)
        }
    }()
}

// sendHeartbeat 发送心跳信息
func sendHeartbeat(registry, addr string) error {
    log.Println(addr, "send heart beat to registry", registry)
    httpClient := &http.Client{}
    req, _ := http.NewRequest("POST", registry, nil)
    req.Header.Set("X-Myrpc-Server", addr)
    // httpClient.Do 发送HTTP请求用的
    if _, err := httpClient.Do(req); err != nil {
        log.Println("rpc server: heart beat err:", err)
        return err
    }
    return nil
}

```

```
}
```

主函数验证

```
package main

import (
    "MyRPC"
    "MyRPC/registry"
    "MyRPC/xclient"
    "context"
    "log"
    "net"
    "net/http"
    "sync"
    "time"
)

type Foo int

type Args struct{ Num1, Num2 int }

func (f Foo) Sum(args Args, reply *int) error {
    *reply = args.Num1 + args.Num2
    return nil
}

func (f Foo) Sleep(args Args, reply *int) error {
    time.Sleep(time.Second * time.Duration(args.Num1))
    *reply = args.Num1 + args.Num2
    return nil
}

// foo 封装一个方法 foo, 便于在 Call 或 Broadcast 之后统一打印成功或失败的日志。
func foo(xc *xclient.XClient, ctx context.Context, typ, serviceMethod string, args *Args) {
    var reply int
    var err error
    switch typ {
    case "call":
        err = xc.Call(ctx, serviceMethod, args, &reply)
    case "broadcast":
        err = xc.Broadcast(ctx, serviceMethod, args, &reply)
    }
    if err != nil {
        log.Printf("%s %s error: %v", typ, serviceMethod, err)
    } else {
        log.Printf("%s %s success: %d + %d = %d", typ, serviceMethod, args.Num1, args.Num2, reply)
    }
}
```

```

func startRegistry(wg *sync.WaitGroup) {
    l, _ := net.Listen("tcp", ":9999")
    registry.HandleHTTP()
    wg.Done()
    _ = http.Serve(l, nil)
}

func startServer(registryAddr string, wg *sync.WaitGroup) {
    var foo Foo
    l, _ := net.Listen("tcp", ":0")
    server := MyRPC.NewServer()
    _ = server.Register(&foo)
    server.Heartbeat(registryAddr, "tcp@"+l.Addr().String(), 0)
    wg.Done()
    server.Accept(l)
}

func call(registry string) {
    d := xclient.NewMyRegistryDiscovery(registry, 0)
    xc := xclient.NewXClient(d, xclient.RandomSelect, nil)
    defer func() { _ = xc.Close() }()
    // send request & receive response
    var wg sync.WaitGroup
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()
            foo(xc, context.Background(), "call", "Foo.Sum", &Args{Num1: i,
Num2: i * i})
        }(i)
    }
    wg.Wait()
}

func broadcast(registry string) {
    d := xclient.NewMyRegistryDiscovery(registry, 0)
    xc := xclient.NewXClient(d, xclient.RandomSelect, nil)
    defer func() { _ = xc.Close() }()
    var wg sync.WaitGroup
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()
            foo(xc, context.Background(), "broadcast", "Foo.Sum",
&Args{Num1: i, Num2: i * i})
            // expect 2 - 5 timeout
            ctx, _ := context.WithTimeout(context.Background(),
time.Second*2)
            foo(xc, ctx, "broadcast", "Foo.Sleep", &Args{Num1: i, Num2: i * i})
        }(i)
    }
    wg.Wait()
}

```

```
func main() {  
    log.SetFlags(0)  
    registryAddr := "http://localhost:9999/_geerpc_/registry"  
    var wg sync.WaitGroup  
    wg.Add(1)  
    go startRegistry(&wg)  
    wg.Wait()  
  
    time.Sleep(time.Second)  
    wg.Add(2)  
    go startServer(registryAddr, &wg)  
    go startServer(registryAddr, &wg)  
    wg.Wait()  
  
    time.Sleep(time.Second)  
    call(registryAddr)  
    broadcast(registryAddr)  
}
```

到此就大功告成了。