目錄

Go RPC 开发指南

本书首先介绍了使用Go官方库开发RPC服务的方法，然后介绍流行gRPC库以及其它一些RPC框架如Thrift等，后面重点介绍高性能的分布式全功能的RPC框架rpcx。读者通过阅读本书，可以快速学习和了解Go生态圈的RPC开发技术，并且应用到产品的开发中。

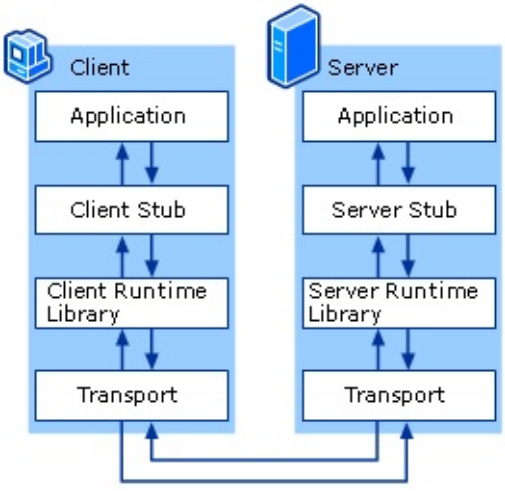
RPC介绍

远程过程调用（Remote Procedure Call，缩写为 RPC）是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互作用编程。 如果涉及的软件采用面向对象编程，那么远程过程调用亦可称作远程调用或远程方法调用，比如 Java RMI。

有关RPC的想法至少可以追溯到1976年以“信使报”（Courier）的名义使用。RPC首次在UNIX平台上普及的执行工具程序是SUN公司的RPC（现在叫ONC RPC）。它被用作SUN的NFC的主要部件。ONC RPC今天仍在服务器上被广泛使用。 另一个早期UNIX平台的工具是“阿波罗”计算机网络计算系统（NCS），它很快就用做OSF的分布计算环境（DCE）中的DCE/RPC的基础，并补充了DCOM。

远程过程调用是一个分布式计算的客户端-服务器（Client/Server）的例子，它简单而又广受欢迎。 远程过程调用总是由客户端对服务器发出一个执行若干过程请求，并用客户端提供的参数。执行结果将返回给客户端。 由于存在各式各样的变体和细节差异，对应地派生了各式远程过程调用协议，而且它们并不互相兼容。

为了允许不同的客户端均能访问服务器，许多标准化的 RPC 系统应运而生了。其中大部分采用接口描述语言（Interface Description Language，IDL），方便跨平台的远程过程调用。



从上图可以看出, RPC 本身是 client-server模型,也是一种 request-response 协议。

有些实现扩展了远程调用的模型，实现了双向的服务调用，但是不管怎样，调用过程还是由一个客户端发起，服务器端提供响应，基本模型没有变化。

服务的调用过程为：

1. client调用client stub，这是一次本地过程调用
2. client stub将参数打包成一个消息，然后发送这个消息。打包过程也叫做 marshalling
3. client所在的系统将消息发送给server
4. server的的系统将收到的包传给server stub
5. server stub解包得到参数。 解包也被称作 unmarshalling
6. 最后server stub调用服务过程. 返回结果按照相反的步骤传给client

国内外知名的**RPC**框架

RPC只是描绘了 Client 与 Server 之间的点对点调用流程，包括 stub、通信、RPC 消息解析等部分，在实际应用中，还需要考虑服务的高可用、负载均衡等问题，所以产品级的 RPC 框架除了点对点的 RPC 协议的具体实现外，还应包括服务的发现与注销、提供服务的多台 Server 的负载均衡、服务的高可用等更多的功能。 目前的 RPC 框架大致有两种不同的侧重方向，一种偏重于服务治理，另一种偏重于跨语言调用。

服务治理型的 RPC 框架有Alibab Dubbo、Motan 等，这类的 RPC 框架的特点是功能丰富，提供高性能的远程调用以及服务发现和治理功能，适用于大型服务的微服务化拆分以及管理，对于特定语言（Java）的项目可以十分友好的透明化接入。但缺点是语言耦合度较高，跨语言支持难度较大。

跨语言调用型的 RPC 框架有 Thrift、gRPC、Hessian、Finagle 等，这一类的 RPC 框架重点关注于服务的跨语言调用，能够支持大部分的语言进行语言无关的调用，非常适合于为不同语言提供通用远程服务的场景。但这类框架没有服务发现相关机制，实际使用时一般需要代理层进行请求转发和负载均衡策略控制。

Dubbo 是阿里巴巴公司开源的一个Java高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和 Spring框架无缝集成。 不过，遗憾的是，据说在淘宝内部，dubbo由于跟淘宝另一个类似的框架HSF（非开源）有竞争关系，导致dubbo团队已经解散（参见 http://www.oschina.net/news/55059/druid-1-0-9 中的评论）。 不过反倒是墙内开花墙外香，其它的一些知名电商如当当 (dubbox)、京东、国美维护了自己的分支或者

在dubbo的基础开发，但是官方的实现缺乏维护，其它电商虽然维护了自己的版本，但是还是不能做大的架构的改动和提升，相关的依赖类比如Spring，Netty还是很老的版本(Spring 3.2.16.RELEASE, netty 3.2.5.Final)，而且现在看来，Dubbo的代码结构也过于复杂了。

所以，尽管Dubbo在电商的开发圈比较流行的时候，国内一些的互联网公司也在开发自己的RPC框架，比如Motan。Motan是新浪微博开源的一个Java 框架。它诞生的比较晚，起于2013年，2016年5月开源。Motan 在微博平台中已经广泛应用，每天为数百个服务完成近千亿次的调用。Motan的架构相对简单，功能也能满足微博内部架构的要求, 虽然Motan的架构的目的主要不是跨语言，但是目前也在开发支持php client和C server特性。

gRPC是Google开发的高性能、通用的开源RPC框架，其由Google主要面向移动应用开发并基于HTTP/2协议标准而设计，基于ProtoBuf(Protocol Buffers)序列化协议开发，且支持众多开发语言。它的目标的跨语言开发，支持多种语言，服务治理方面需要自己去实现，所以要实现一个综合的产品级的分布式RPC平台还需要扩展开发。Google内部使用的也不是gRPC,而是Stubby。

thrift是Apache的一个跨语言的高性能的服务框架，也得到了广泛的应用。它的功能类似 gRPC, 支持跨语言，不支持服务治理。

rpcx 是一个分布式的Go语言的 RPC 框架，支持Zookepper、etcd、consul多种服务发现方式，多种服务路由方式，是目前性能最好的 RPC 框架之一。

RPC vs RESTful

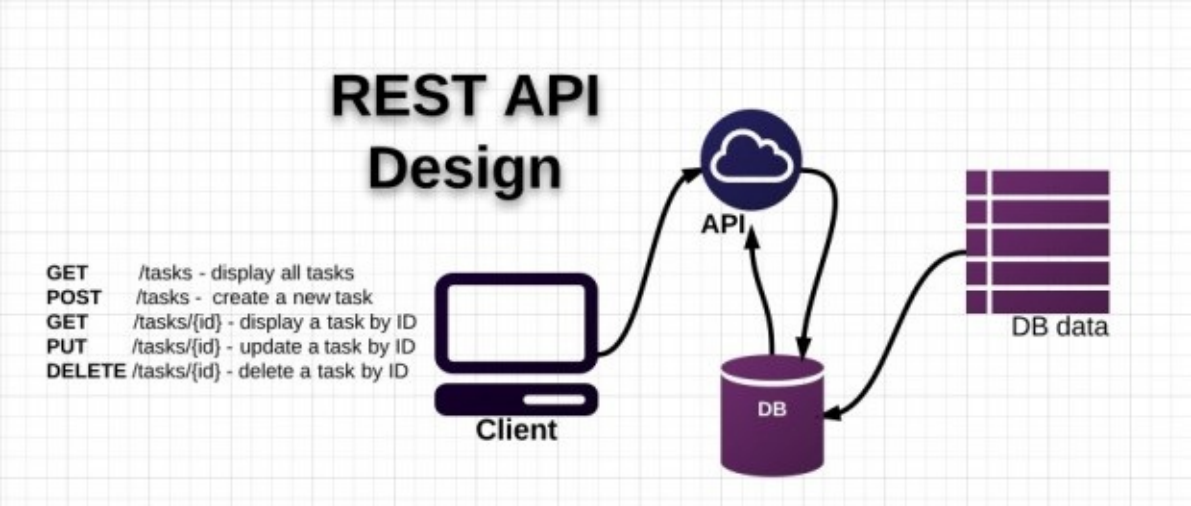
RPC 的消息传输可以通过 TCP、UDP 或者 HTTP等，所以有时候我们称之为 RPC over TCP、 RPC over HTTP。RPC 通过 HTTP 传输消息的时候和 RESTful的架构是类似的，但是也有不同。

首先我们比较 RPC over HTTP 和 RESTful。

首先 RPC 的客户端和服务器端师紧耦合的，客户端需要知道调用的过程的名字，过程的参数以及它们的类型、顺序等。一旦服务器更改了过程的实现，客户端的实现很容易出问题。RESTful基于 http的语义操作资源，参数的顺序一般没有关系，也很容易的通过代理转换链接和资源位置，从这一点上来说，RESTful 更灵活。

其次，它们操作的对象不一样。RPC 操作的是方法和过程，它要操作的是方法对象。RESTful 操作的是资源(resource)，而不是方法。

第三，RESTful执行的是对资源的操作，增加、查找、修改和删除等,主要是CURD，所以如果你要实现一个特定目的的操作，比如为名字姓张的学生的数学成绩都加上10这样的操作， RESTful的API设计起来就不是那么直观或者有意义。在这种情况下, RPC的实现更有意义，它可以实现一个 `Student.Increment(Name, Score)` 的方法供客户端调用。



我们再来比较一下 RPC over TCP 和 RESTful。 如果我们直接使用socket实现RPC，除了上面的不同外，我们可以获得性能上的优势。

RPC over TCP可以通过长连接减少连接的建立所产生的花费，在调用次数非常巨大的时候(这是目前互联网公司经常遇到的情况,大并发的情况下)，这个花费影响是非常巨大的。 当然 RESTful 也可以通过 keep-alive 实现长连接， 但是它最大的一个问题是它的request-response模型是阻塞的 (http1.0和 http1.1, http 2.0没这个问题)， 发送一个请求后只有等到response返回才能发送第二个请求 (有些http server实现了pipeling的功能，但不是标配)， RPC的实现没有这个限制。

在当今用户和资源都是大数据大并发的趋势下，一个大规模的公司不可能使用一个单体程序提供所有的功能，微服务的架构模式越来越多的被应用到产品的设计和开发中， 服务和服务之间的通讯也越发的重要， 所以 RPC 不失是一个解决服务之间通讯的好办法， 本书给大家介绍 Go 语言的 RPC的开发实践。

参考文档

1. https://en.wikipedia.org/wiki/Remote_procedure_call
2. https://technet.microsoft.com/en-us/library/cc738291(v=ws.10).aspx
3. https://tools.ietf.org/html/rfc1057
4. https://tools.ietf.org/html/rfc5531
5. http://apihandyman.io/do-you-really-know-why-you-prefer-rest-over-rpc/
6. https://www.quora.com/What-is-the-difference-between-REST-and-RPC

7. http://stackoverflow.com/questions/15056878/rest-vs-json-rpc
8. https://cascadingmedia.com/insites/2015/03/http-2.html

By *smallnest* *updated* 2017-10-30 01:59:26

官方RPC标准库

Go官方提供了一个RPC库：`net/rpc`。

包rpc提供了通过网络访问一个对象的输出方法的能力。
服务器需要注册对象，通过对象的类型名暴露这个服务。注册后这个对象的输出方法就可以远程调用，这个库封装了底层传输的细节，包括序列化(默认GOB序列化器)。
服务器可以注册多个不同类型的对象，但是注册相同类型的多个对象的时候会出错。

同时，如果对象的方法要能远程访问，它们必须满足一定的条件，否则这个对象的方法会被忽略。

这些条件是：

- 方法的类型是可输出的 (the method's type is exported)
- 方法本身也是可输出的（the method is exported）
- 方法必须由两个参数，必须是输出类型或者是内建类型 (the method has two arguments, both exported or builtin types)
- 方法的第二个参数必须是指针类型 (the method's second argument is a pointer)
- 方法返回类型为 error (the method has return type error)

所以一个输出方法的格式如下：

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

这里的`T`、`T1`、`T2`能够被`encoding/gob`序列化，即使使用其它的序列化框架，将来这个需求可能回被弱化。

这个方法的第一个参数代表调用者(client)提供的参数，
第二个参数代表要返回给调用者的计算结果，
方法的返回值如果不为空，那么它作为一个字符串返回给调用者。
如果返回error，则reply参数不会返回给调用者。

服务器通过调用`ServeConn`在一个连接上处理请求，更典型地，它可以创建一个network listener然后accept请求。
对于HTTP listener来说，可以调用`HandleHTTP`和`http.Serve`。细节会在下

面介绍。

客户端可以调用 `Dial` 和 `DialHTTP` 建立连接。 客户端有两个方法调用服务: `Call` 和 `Go` ，可以同步地或者异步地调用服务。
当然，调用的时候，需要把服务名、方法名和参数传递给服务器。异步方法调用 `Go` 通过 `Done` channel通知调用结果返回。

除非显示的设置 `codec` ,否则这个库默认使用包 `encoding/gob` 作为序列化框架。

简单例子

首选介绍一个简单的例子, 这个例子摘自官方标准库，是一个非常简单的服务。

这个例子中提供了对两个数相乘和相除的两个方法。

第一步你需要定义传入参数和返回参数的数据结构：

```
package server

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}
```

第二步定义一个服务对象，这个服务对象可以很简单， 比如类型是 `int` 或者是 `interface{}` ,重要的是它输出的方法。
这里我们定义一个算术类型 `Arith` ，其实它是一个int类型，但是这个int的值我们在后面方法的实现中也没用到，所以它基本上就起一个辅助的作用。

```
type Arith int
```

第三步实现这个类型的两个方法， 乘法和除法：

```
func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

目前为止，我们的准备工作已经完成，喝口茶继续下面的步骤。

第四步实现RPC服务器:

```
arith := new(Arith)
rpc.Register(arith)
rpc.HandleHTTP()
l, e := net.Listen("tcp", ":1234")
if e != nil {
    log.Fatal("listen error:", e)
}
go http.Serve(l, nil)
```

这里我们生成了一个Arith对象，并使用 `rpc.Register` 注册这个服务，然后通过HTTP暴露出来。

客户端可以看到服务 `Arith` 以及它的两个方法 `Arith.Multiply` 和 `Arith.Divide` 。

第五步创建一个客户端，建立客户端和服务器端的连接:

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
```

然后客户端就可以进行远程调用了。比如同步的方式调用：

```
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
    log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)
```

或者异步的方式：

```
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args, quotient, nil)
replyCall := <-divCall.Done    // will be equal to divCall
// check errors, print, etc.
```

服务器代码分析

首先，`net/rpc`定义了一个缺省的Server,所以Server的很多方法你可以直接调用，这对于一个简单的Server的实现更方便，但是你如果需要配置不同的Server，比如不同的监听地址或端口，就需要自己生成Server:

```
var DefaultServer = NewServer()
```

Server有多种Socket监听的方式:

```
    func (server *Server) Accept(lis net.Listener)
    func (server *Server) HandleHTTP(rpcPath, debugPath string)
    func (server *Server) ServeCodec(codec ServerCodec)
    func (server *Server) ServeConn(conn io.ReadWriteCloser)
    func (server *Server) ServeHTTP(w http.ResponseWriter, req *
http.Request)
    func (server *Server) ServeRequest(codec ServerCodec) error
```

其中，`ServeHTTP`实现了处理http请求的业务逻辑，它首先处理http的`CONNECT`请求，接收后就Hijacker这个连接conn，然后调用`ServeConn`在这个连接上 处理这个客户端的请求。

它其实是实现了 `http.Handler` 接口，我们一般不直接调用这个方法。
‘Server.HandleHTTP‘设置rpc的上下文路径，‘rpc.HandleHTTP‘使用默认的上下文路径‘DefaultRPCPath‘、 `DefaultDebugPath` 。
这样，当你启动一个http server的时候 ‘http.ListenAndServe‘，上面设置的上下文将用作RPC传输，这个上下文的请求会教给 `ServeHTTP` 来处理。

以上是RPC over http的实现，可以看出 `net/rpc` 只是利用 http CONNECT建立连接，这和普通的 RESTful api还是不一样的。

‘Accept‘用来处理一个监听器，一直在监听客户端的连接，一旦监听器接收了一个连接，则还是交给 `ServeConn` 在另外一个goroutine中去处理：

```
func (server *Server) Accept(lis net.Listener) {
    for {
        conn, err := lis.Accept()
        if err != nil {
            log.Print("rpc.Serve: accept:", err.Error())
            return
        }
        go server.ServeConn(conn)
    }
}
```

可以看出，很重要的一个方法就是 `ServeConn` ：

```
func (server *Server) ServeConn(conn io.ReadWriteCloser) {
    buf := bufio.NewWriter(conn)
    srv := &gobServerCodec{
        rwc:    conn,
        dec:    gob.NewDecoder(conn),
        enc:    gob.NewEncoder(buf),
        encBuf: buf,
    }
    server.ServeCodec(srv)
}
```

连接其实是交给一个 `ServerCodec` 去处理，这里默认使用 `gobServerCodec` 去处理，这是一个未输出默认的编解码器，你可以使用其它的编解码器，我们下面再介绍，
这里我们可以看看 `ServeCodec` 是怎么实现的：

```
func (server *Server) ServeCodec(codec ServerCodec) {
    sending := new(sync.Mutex)
    for {
        service, mtype, req, argv, replyv, keepReading, err := s
erver.readRequest(codec)
        if err != nil {
            if debugLog && err != io.EOF {
                log.Println("rpc:", err)
            }
            if !keepReading {
                break
            }
            // send a response if we actually managed to read a
header.
            if req != nil {
                server.sendResponse(sending, req, invalidRequest
, codec, err.Error())
                server.freeRequest(req)
            }
            continue
        }
        go service.call(server, sending, mtype, req, argv, reply
v, codec)
    }
    codec.Close()
}
```

它其实一直从连接中读取请求，然后调用 `go service.call` 在另外的goroutine中处理服务调用。

我们从中可以学到：

1. 对象重用。Request和Response都是可重用的，通过Lock处理竞争。这在大并发的情况下很有效。有兴趣的读者可以参考fasthttp的实现。
2. 使用了大量的goroutine。和Java中的线程不同，你可以创建非常多的goroutine, 并发处理非常好。如果使用一定数量的goutine作为worker池去处理这个case，可能还会有些性能的提升，但是更复杂了。使用goroutine已经获得了非常好的性能。
3. 业务处理是异步的, 服务的执行不会阻塞其它消息的读取。

注意一个codec实例必然和一个connnection相关，因为它需要从connection中读取request和发送response。

go的rpc官方库的消息(request和response)的定义很简单， 就是消息头(header)+内容体(body)。

请求的消息头的定义如下，包括服务的名称和序列号：

```
type Request struct {
        ServiceMethod string // format: "Service.Method"
        Seq           uint64 // sequence number chosen by client
        // contains filtered or unexported fields
}
```

消息体就是传入的参数。

返回的消息头的定义如下：

```
type Response struct {
        ServiceMethod string // echoes that of the Request
        Seq           uint64 // echoes that of the request
        Error         string // error, if any.
        // contains filtered or unexported fields
}
```

消息体是reply类型的序列化后的值。

Server还提供了两个注册服务的方法:

```
    func (server *Server) Register(rcvr interface{}) error
    func (server *Server) RegisterName(name string, rcvr interfa
ce{}) error
```

第二个方法为服务起一个别名，否则服务名已它的类型命名,它们俩底层调用 `register` 进行服务的注册。

```
func (server *Server) register(rcvr interface{}, name string, us
eName bool) error
```

受限于Go语言的特点， 我们不可能在接到客户端的请求的时候，根据反射动态的创建一个对象，就是Java那样，
因此在Go语言中，我们需要预先创建一个服务map这是在编译的时候完成的:

```
server.serviceMap = make(map[string]*service)
```

同时每个服务还有一个方法map: map[string]*methodType,通过suitableMethods建立:

```
func suitableMethods(typ reflect.Type, reportErr bool) map[string
]*methodType
```

这样rpc在读取请求header，通过查找这两个map，就可以得到要调用的服务及它的对应方法了。

方法的调用:

```
func (s *service) call(server *Server, sending *sync.Mutex, mtyp
e *methodType, req *Request, argv, replyv reflect.Value, codec S
erverCodec) {
    mtype.Lock()
    mtype.numCalls++
    mtype.Unlock()
    function := mtype.method.Func
    // Invoke the method, providing a new value for the reply.
    returnValues := function.Call([]reflect.Value{s.rcvr, argv,
replyv})
    // The return value for the method is an error.
    errInter := returnValues[0].Interface()
    errmsg := ""
    if errInter != nil {
        errmsg = errInter.(error).Error()
    }
    server.sendResponse(sending, req, replyv.Interface(), codec,
 errmsg)
    server.freeRequest(req)
}
```

客户端代码分析

客户端要建立和服务器的连接，可以有以下几种方式：

```
    func Dial(network, address string) (*Client, error)
    func DialHTTP(network, address string) (*Client, error)
    func DialHTTPPath(network, address, path string) (*Client, e
rror)
    func NewClient(conn io.ReadWriteCloser) *Client
    func NewClientWithCodec(codec ClientCodec) *Client
```

`DialHTTP` 和 `DialHTTPPath` 是通过HTTP的方式和服务器建立连接，他俩的区别之在于是否设置上下文路径:

```
func DialHTTPPath(network, address, path string) (*Client, error
) {
    var err error
    conn, err := net.Dial(network, address)
    if err != nil {
        return nil, err
    }
    io.WriteString(conn, "CONNECT "+path+" HTTP/1.0\n\n")

    // Require successful HTTP response
    // before switching to RPC protocol.
    resp, err := http.ReadResponse(bufio.NewReader(conn), &http.
Request{Method: "CONNECT"})
    if err == nil && resp.Status == connected {
        return NewClient(conn), nil
    }
    if err == nil {
        err = errors.New("unexpected HTTP response: " + resp.Sta
tus)
    }
    conn.Close()
    return nil, &net.OpError{
        Op:   "dial-http",
        Net:  network + " " + address,
        Addr: nil,
        Err:  err,
    }
}
```

首先发送 `CONNECT` 请求，如果连接成功则通过 `NewClient(conn)` 创建client。

而 `Dial` 则通过TCP直接连接服务器：

```
func Dial(network, address string) (*Client, error) {
    conn, err := net.Dial(network, address)
    if err != nil {
        return nil, err
    }
    return NewClient(conn), nil
}
```

根据服务是over HTTP还是over TCP选择合适的连接方式。

`NewClient`则创建一个缺省codec为glob序列化库的客户端:

```
func NewClient(conn io.ReadWriteCloser) *Client {
    encBuf := bufio.NewWriter(conn)
    client := &gobClientCodec{conn, gob.NewDecoder(conn), gob.Ne
wEncoder(encBuf), encBuf}
    return NewClientWithCodec(client)
}
```

如果你想用其它的序列化库，你可以调用`NewClientWithCodec`方法<:></:>

```
func NewClientWithCodec(codec ClientCodec) *Client {
    client := &Client{
        codec:   codec,
        pending: make(map[uint64]*Call),
    }
    go client.input()
    return client
}
```

重要的是`input`方法，它以一个死循环的方式不断地从连接中读取response,然后调用map中读取等待的Call.Done channel通知完成。

消息的结构和服务器一致，都是Header+Body的方式。

客户端的调用有两个方法:`Go`和`Call`。`Go`方法是异步的，它返回一个 Call 指针对象， 它的Done是一个channel，如果服务返回，
Done就可以得到返回的对象(实际是Call对象，包含Reply和error信息)。`Call`是同步的方式调用，它实际是调用`Go`实现的，
我们可以看看它是怎么实现的，可以了解一下异步变同步的方式：

```
func (client *Client) Call(serviceMethod string, args interface{
}, reply interface{}) error {
    call := <-client.Go(serviceMethod, args, reply, make(chan *C
all, 1)).Done
    return call.Error
}
```

从一个Channel中读取对象会被阻塞住，直到有对象可以读取，这种实现很简单，也很方便。

其实从服务器端的代码和客户端的代码实现我们还可以学到锁Lock的一种实用方式，也就是尽快的释放锁，而不是`defer mu.Unlock`直到函数执行到最后才释放，那样锁占用的时间太长了。

codec／序列化框架

前面我们介绍了rpc框架默认使用gob序列化库，很多情况下我们追求更好的效率的情况下，或者追求更通用的序列化格式，我们可能采用其它的序列化方式， 比如protobuf, json, xml等。

gob序列化库有个要求，就是对于接口类型的值，你需要注册具体的实现类型：

```
func Register(value interface{})
func RegisterName(name string, value interface{})
```

初次使用rpc的人容易犯这个错误，导致序列化不成功。

Go官方库实现了JSON-RPC 1.0。JSON-RPC是一个通过JSON格式进行消息传输的RPC规范，因此可以进行跨语言的调用。
Go的`net/rpc/jsonrpc`库可以将JSON-RPC的请求转换成自己内部的格式，比如request header的处理：

```
func (c *serverCodec) ReadRequestHeader(r *rpc.Request) error {
    c.req.reset()
    if err := c.dec.Decode(&c.req); err != nil {
        return err
    }
    r.ServiceMethod = c.req.Method
    c.mutex.Lock()
    c.seq++
    c.pending[c.seq] = c.req.Id
    c.req.Id = nil
    r.Seq = c.seq
    c.mutex.Unlock()

    return nil
}
```

JSON-RPC 2.0官方库布支持，但是有第三方开发者提供了实现，比如:

- https://github.com/powerman/rpc-codec
- https://github.com/dwlnetnl/generpc

一些其它的codec如 bsonrpc、messagepack、protobuf等。
如果你使用其它特定的序列化框架，你可以参照这些实现来写一个你自己的rpc codec。

关于Go序列化库的性能的比较你可以参考 gosercomp。

其它

有一个提案 deprecate net/rpc：

> The package has outstanding bugs that are hard to fix, and cannot support TLS without major work. So although it has a nice API and allows one to use native Go types without an IDL, it should probably be retired.
>
> The proposal is to freeze the package, retire the many bugs filed against it, and add documentation indicating that it is frozen and that suggests alternatives such as GRPC.

但我认为net/rpc的设计相当的优秀，性能超好，如果不继续开发就太可惜了。提案中提到的一些bug和TLS并不是不能修复，可能Go team缺乏相应的资源，或者开发者兴趣不在这里而已。我相信这个提案有很大的反对意见。

目前看来 `gRPC` 的性能远远逊于 `net/rpc`，不仅仅是吞吐率，还包括CPU的占有率。

参考文档

1. https://golang.org/pkg/net/rpc/
2. https://golang.org/pkg/encoding/gob/
3. https://golang.org/pkg/net/rpc/jsonrpc/
4. https://github.com/golang/go/issues/16844

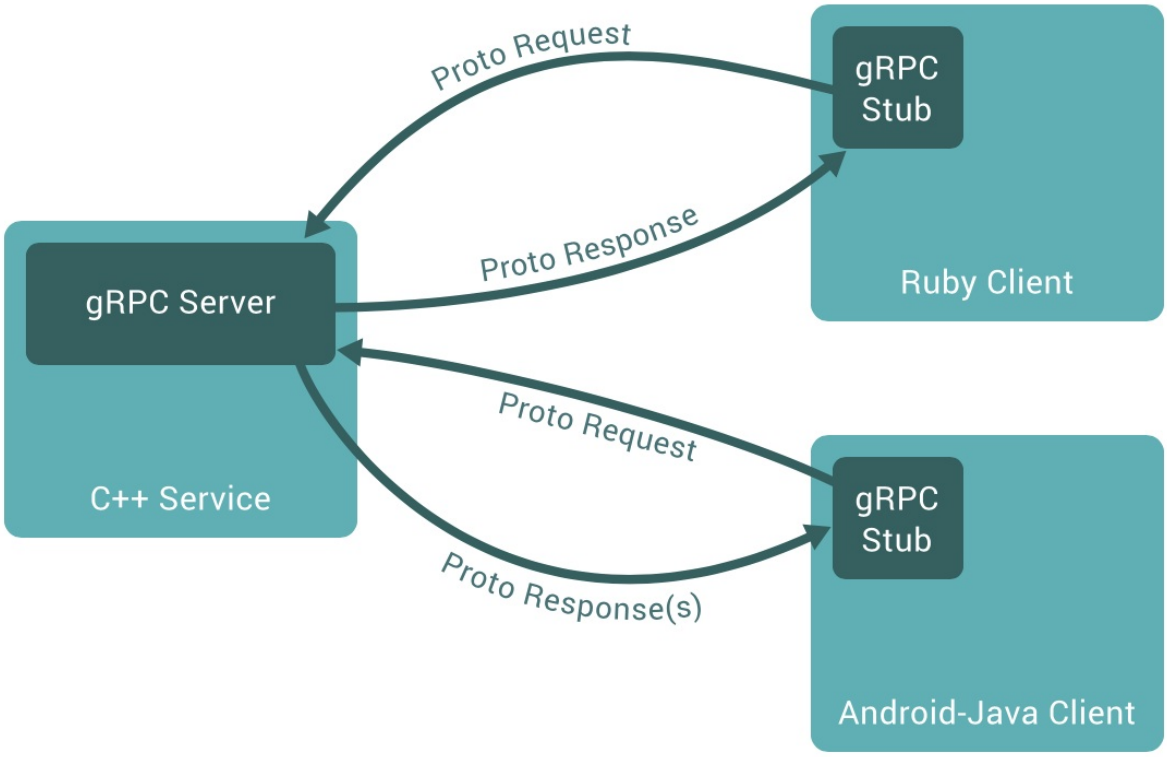
By *smallnest* *updated* 2017-10-30 01:59:26

gRPC库介绍

gRPC是一个高性能、通用的开源RPC框架，其由Google主要面向移动应用开发并基于HTTP/2协议标准而设计，基于ProtoBuf(Protocol Buffers)序列化协议开发，且支持众多开发语言。 gRPC提供了一种简单的方法来精确地定义服务和为iOS、Android和后台支持服务自动生成可靠性很强的客户端功能库。 客户端充分利用高级流和链接功能，从而有助于节省带宽、降低的TCP链接次数、节省CPU使用、和电池寿命。

gRPC具有以下重要特征：

1. 强大的IDL特性 RPC使用ProtoBuf来定义服务，ProtoBuf是由Google开发的一种数据序列化协议，性能出众，得到了广泛的应用。
2. 支持多种语言 支持C++、Java、Go、Python、Ruby、C#、Node.js、Android Java、Objective-C、PHP等编程语言。
3. 基于HTTP/2标准设计



gRPC已经应用在Google的云服务和对外提供的API中。

我们以 gRPC-go 为例介绍一下gRPC的开发。

首先下载相应的库:

```
go get google.golang.org/grpc
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
```

同时保证按照Protocol Buffers v3 编译器到你的开发环境中(protoc)。

定义你的protobuf文件 (helloworld.proto):

```
syntax = "proto3";

option java_package = "com.colobu.rpctest";

package greeter;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

这个文件定义了一个 `Greeter` 服务，它有一个 `SayHello` 方法，这个方法接收一个Request，返回一个Response。

然后我们可以编译这个文件，生成服务器和客户端的stub:

```
protoc -I protos protos/helloworld.proto --go_out=plugins=grpc:s
rc/greeter
```

因为上面我们安装了 `proto` 和 `protoc-gen-go` ，所以 `protoc` 可以生成响应的Go代码。

然后我们就可以利用这个生成的代码创建服务器代码和客户端代码了。

服务器端的代码如下：

```
package main

import (
    "log"
    "net"

    pb "greeter"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
)

const (
    port = ":50051"
)

type server struct{}

// SayHello implements helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *pb.HelloReque
st) (*pb.HelloReply, error) {
    return &pb.HelloReply{Message: "Hello " + in.Name}, nil
}

func main() {
    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s := grpc.NewServer()
    pb.RegisterGreeterServer(s, &server{})
    s.Serve(lis)
}
```

客户端的测试代码如下：

```
package main

import (
```

```
    "fmt"
    "log"
    "os"
    "strconv"
    "sync"
    "time"

    pb "greeter"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
)

const (
    address     = "localhost:50051"
    defaultName = "world"
)

func invoke(c pb.GreeterClient, name string) {
    r, err := c.SayHello(context.Background(), &pb.HelloRequest{
Name: name})
    if err != nil {
        log.Fatalf("could not greet: %v", err)
    }
    _ = r
}

func syncTest(c pb.GreeterClient, name string) {
    i := 10000
    t := time.Now().UnixNano()
    for ; i>0; i-- {
        invoke(c, name)
    }
    fmt.Println("took", (time.Now().UnixNano() - t) / 1000000, "
ms")
}


func asyncTest(c [20]pb.GreeterClient, name string) {
    var wg sync.WaitGroup
    wg.Add(10000)
```

```
    i := 10000
    t := time.Now().UnixNano()
    for ; i>0; i-- {
        go func() {invoke(c[i % 20], name);wg.Done()}()
    }
    wg.Wait()
    fmt.Println("took", (time.Now().UnixNano() - t) / 1000000, "
ms")
}


func main() {
    // Set up a connection to the server.
    conn, err := grpc.Dial(address)
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()
    var c [20]pb.GreeterClient


    // Contact the server and print out its response.
    name := defaultName
    sync := true
    if len(os.Args) > 1 {
        sync, err = strconv.ParseBool(os.Args[1])
    }

    //warm up
    i := 0
    for ; i < 20; i++ {
        c[i] = pb.NewGreeterClient(conn)
        invoke(c[i], name)
    }

    if sync {
        syncTest(c[0], name)
    } else {
        asyncTest(c, name)
    }
}
```

参考文档

1. http://www.grpc.io/docs/quickstart/go.html
2. http://www.infoq.com/cn/news/2015/03/grpc-google-http2-protobuf
3. https://github.com/smallnest/RPC-TEST

By *smallnest* *updated* 2017-10-30 01:59:26

其它 Go RPC 库介绍

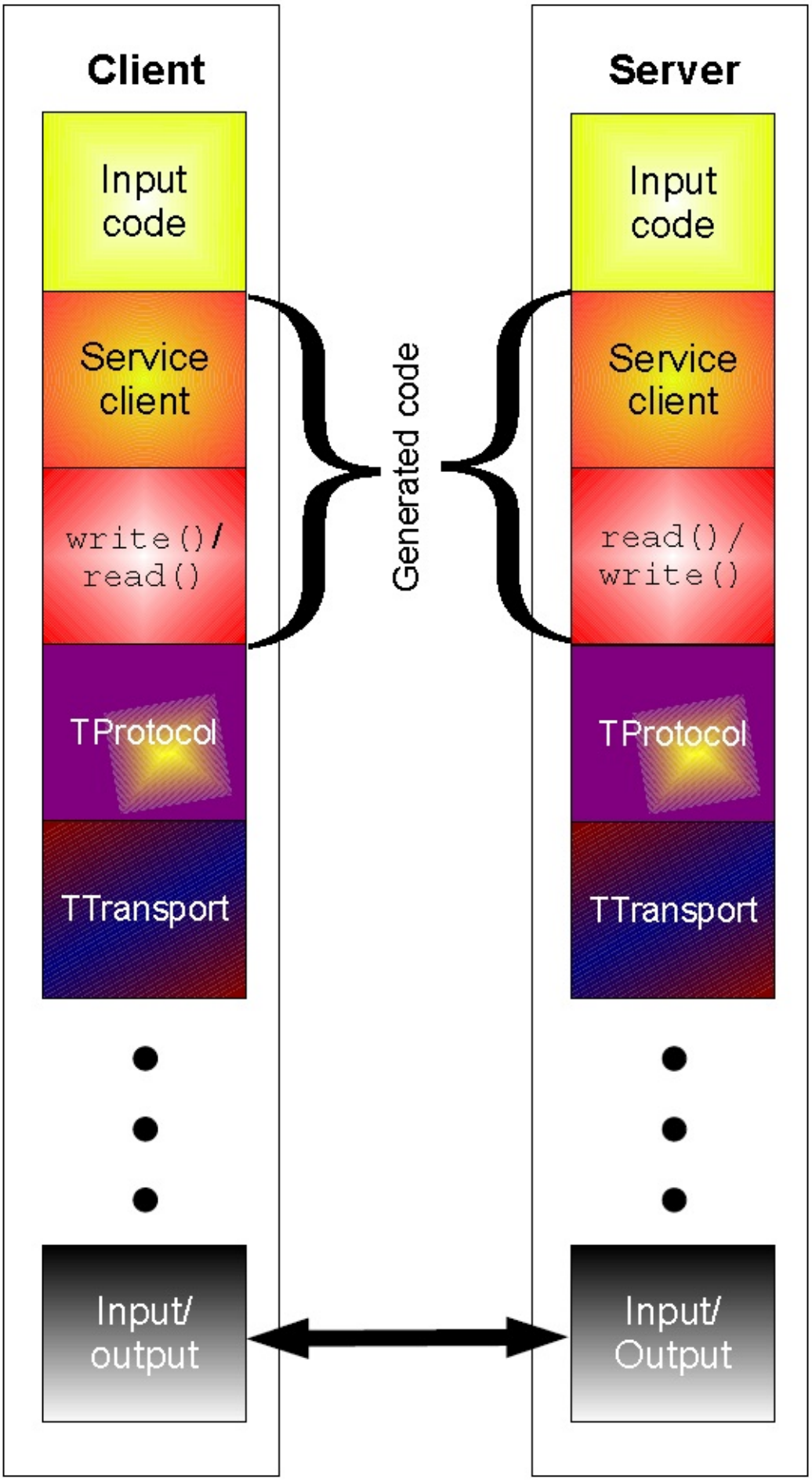
当然，其它的一些 RPC框架也有提供了Go的绑定，知名的比如Thrift。

Thrift

2007年开源，2008看5月进入Apache孵化器，2010年10月成为Apache的顶级项目。

Thrift是一种接口描述语言和二进制通讯协议，它被用来定义和创建跨语言的服务。它被当作一个远程过程调用（RPC）框架来使用，是由Facebook为“大规模跨语言服务开发”而开发的。 它通过一个代码生成引擎联合了一个软件栈，来创建不同程度的、无缝的跨平台高效服务，可以使用C#、C++（基于POSIX兼容系统）、Cappuccino、Cocoa、Delphi、Erlang、Go、Haskell、Java、Node.js、OCaml、Perl、PHP、Python、Ruby和Smalltalk编程语言开发。 2007由Facebook开源，2008年5月进入Apache孵化器， 2010年10月成为Apache的顶级项目。

Thrift包含一套完整的栈来创建客户端和服务端程序。[7]顶层部分是由Thrift定义生成的代码。而服务则由这个文件客户端和处理器代码生成。在生成的代码里会创建不同于内建类型的数据结构，并将其作为结果发送。协议和传输层是运行时库的一部分。有了Thrift，就可以定义一个服务或改变通讯和传输协议，而无需重新编译代码。除了客户端部分之外，Thrift还包括服务器基础设施来集成协议和传输，如阻塞、非阻塞及多线程服务器。栈中作为I/O基础的部分对于不同的语言则有不同的实现。



Thrift一些已经明确的优点包括：[

- 跟一些替代选择，比如SOAP相比，跨语言序列化的代价更低，因为它使用二进制格式。
- 它有一个又瘦又干净的库，没有编码框架，没有XML配置文件。
- 绑定感觉很自然。例如，Java使用java/util/ArrayList.html ArrayList；C++使用std::vector。
- 应用层通讯格式与序列化层通讯格式是完全分离的。它们都可以独立修改。
- 预定义的序列化格式包括：二进制、对HTTP友好的和压缩的二进制。
- 兼作跨语言文件序列化。
- 支持协议的[需要解释]。Thrift不要求一个集中的和明确的机制，象主版本号/次版本号。松耦合的团队可以自由地进化RPC调用。
- 没有构建依赖或非标软件。不混合不兼容的软件许可证。

首先你需要安装 thrift编译器: download。 然后安装thift-go库：

```
git.apache.org/thrift.git/lib/go/thrift
```

对于一个服务，你需要定义thrift文件 (helloworld.thrift)：

```
namespace go greeter

service Greeter {

    string sayHello(1:string name);

}
```

编译创建相应的stub,它会创建多个辅助文件:

```
thrift -r --gen go  -out src thrift/helloworld.thrift
```

服务端的代码:

```
package main

import (
    "fmt"
    "os"

    "git.apache.org/thrift.git/lib/go/thrift"
```

```
    "greeter"
)

const (
    NetworkAddr = "localhost:9090"
)


type GreeterHandler struct {

}

func NewGreeterHandler() *GreeterHandler {
    return &GreeterHandler{}
}

func (p *GreeterHandler) SayHello(name string)(r string, err err
or) {
    return "Hello " + name, nil
}

func main() {
    var protocolFactory thrift.TProtocolFactory = thrift.NewTBin
aryProtocolFactoryDefault()
    var transportFactory thrift.TTransportFactory = thrift.NewTB
ufferedTransportFactory(8192)
    transport, err := thrift.NewTServerSocket(NetworkAddr)
    if err != nil {
        fmt.Println("Error!", err)
        os.Exit(1)
    }

    handler := NewGreeterHandler()
    processor := greeter.NewGreeterProcessor(handler)
    server := thrift.NewTSimpleServer4(processor, transport, tra
nsportFactory, protocolFactory)
    fmt.Println("Starting the simple server... on ", NetworkAddr
)
    server.Serve()
}
```

客户端的测试代码：

```
package main

import (
    "os"
    "fmt"
    "time"
    "strconv"
    "sync"

    "greeter"

    "git.apache.org/thrift.git/lib/go/thrift"

)

const (
    address     = "localhost:9090"
    defaultName = "world"
)

func syncTest(client *greeter.GreeterClient, name string) {
    i := 10000
    t := time.Now().UnixNano()
    for ; i>0; i-- {
        client.SayHello(name)
    }
    fmt.Println("took", (time.Now().UnixNano() - t) / 1000000, "
ms")
}

func asyncTest(client [20]*greeter.GreeterClient, name string) {
    var locks [20]sync.Mutex
    var wg sync.WaitGroup
    wg.Add(10000)

    i := 10000
    t := time.Now().UnixNano()
    for ; i>0; i-- {
        go func(index int) {
        locks[index % 20].Lock()
```

```
        client[ index % 20].SayHello(name)
        wg.Done()
        locks[index % 20].Unlock()
        }(i)
    }
    wg.Wait()
    fmt.Println("took", (time.Now().UnixNano() - t) / 1000000, "
ms")
}


func main() {
    transportFactory := thrift.NewTBufferedTransportFactory(8192
)
    protocolFactory := thrift.NewTBinaryProtocolFactoryDefault()

    var client [20]*greeter.GreeterClient

    //warm up
    for i := 0; i < 20; i++ {
        transport, err := thrift.NewTSocket(address)
        if err != nil {
            fmt.Fprintln(os.Stderr, "error resolving address:",
err)
            os.Exit(1)
        }
        useTransport := transportFactory.GetTransport(transport)
        defer transport.Close()

        if err := transport.Open(); err != nil {
            fmt.Fprintln(os.Stderr, "Error opening socket to loc
alhost:9090", " ", err)
            os.Exit(1)
        }

        client[i] = greeter.NewGreeterClientFactory(useTransport
, protocolFactory)
        client[i].SayHello(defaultName)
    }

    sync := true
    if len(os.Args) > 1 {
```

```
        sync, _ = strconv.ParseBool(os.Args[1])
    }

    if sync {
        syncTest(client[0], defaultName)
    } else {
        asyncTest(client, defaultName)
    }
}
```

其它一些Go RPC框架如

1. http://www.gorillatoolkit.org/pkg/rpc
2. https://github.com/valyala/gorpc
3. https://github.com/micro/go-micro
4. https://github.com/go-kit/kit

参考文档

1. https://thrift.apache.org/tutorial/go
2. https://github.com/smallnest/RPC-TEST

By *smallnest* *updated* 2017-10-30 01:59:26

RPCX 起步

rpcx是一个分布式的服务框架，致力于提供一个产品级的、高性能、透明化的RPC远程服务调用框架。它参考了目前在国内非常流行的Java生态圈的RPC框架Dubbo、Motan等，为Go生态圈提供一个丰富功能的RPC平台。

随着互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构以及流动计算架构势在必行，亟需一个治理系统确保架构有条不紊的演进。

目前，随着网站的规模的扩大，一般会将单体程序逐渐演化为微服务的架构方式，这是目前流行的一种架构模式。

即使不是微服务，也会将业务拆分成不同的服务的方式，服务之间互相调用。

那么，如何实现服务(微服务)之间的调用的？一般来说最常用的是两种方式: RESTful API和RPC两种服务通讯方式。本书的第一章就介绍了这两种方式的特点和优缺点，那么本书重点介绍的是RPC的方式。

RPCX就是为Go生态圈提供的一个全功能的RPC框架,它参考了国内电商圈流行的RPC框架Dubbo的功能特性，实现了一个高性能的、可容错的，插件式的RPC框架。

RPCX的目标：

1. 简单： 易于学习、易于开发、易于集成和易于发布
2. 高性能：远远高于grpc-go, 更不用说dubbo和motan
3. 服务发现和服务治理：方便开发大规模的微服务集群
4. 跨平台： rpcx 3.0底层不再使用标准rpc库，而是采用跨平台的二进制协议，高效但是方便多语言开发

它的特点包括：

1、指出纯的go方法， 不需要额外的定义 2、可插拔的设计，可以方便扩展服务发现插件、tracing等 3、支持 TCP、HTTP、QUIC、KCP等协议 4、支持多种编码方式， 比如JSON、Protobuf、MessagePack 和 原始字节数据 5、服务发现支持 单机对单机、单机对多机、zookeeper、etcd、consul、mDNS等多种发现方式 6、容错支持 Failover、Failfast、Failtry等多种模式 7、负载均衡支持随机选取、顺序选

取、一致性哈希、基于权重的选取、基于网络质量的选取和就近选取等多种均衡方式 8、支持压缩 9、支持扩展信息传递（元数据） 10、支持身份验证 11、支持自动heartbeat和单向请求 12、支持metrics、log、timeout、别名、断路器、TLS等特性

本章就让我们举两个的例子，看看如何利用rpcx进行开发。

端对端的例子

首先让我们看一个简单的例子，一个服务器和一个客户端。 这并不是一个常用的应用场景，因为部署的规模太小，其实可以直接官方的库或者其它的RPC框架如gRPC、Thrift就可以实现，当然使用rpcx也很简单，我们就以这个例子，先熟悉一下rpcx的开发。

本书中常用的一个例子就是提供一个乘法的服务，客户端提供两个数，服务器计算这两个数的乘积返回。

服务器端开发

首先，我们需要实现自己的服务，这很简单，就是定义普通的方法即可:

```
type Args struct {
    A int
    B int
}

type Reply struct {
    C int
}

type Arith int

func (t *Arith) Mul(ctx context.Context, args *Args, reply *Repl
y) error {
    reply.C = args.A * args.B
    return nil
}
```

`Args` 作为传入的参数，它的两个字段 `A` 、 `B` 代表两个乘数。 `Reply` 的 `C` 代表返回的结果。 `Mul` 就是业务方法，对乘数进行相乘，然后返回结果。

然后注册这个服务启动就可以了：

```
func main() {
    s := server.Server{}
    s.RegisterName("Arith", new(example.Arith), "")
    s.Serve("tcp", *addr)
}
```

三行代码就可以将一个方法暴露成一个服务，它使用默认的编码和传输方式(TCP)。但是如果你想使用定制的服务器，你可以使用 `rpcx.NewServer` 生成一个新的服务器对象。

客户端同步调用

客户端代码也很简单，首先你需要定义一些访问服务器的一些策略以及服务器的一些信息,然后创建一个 `XClient` 对象，并且在程序不再使用这个对象的时候关闭它:

```
 d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")
 xclient := client.NewXClient("Arith", "Mul", client.Failtry, cl
ient.RandomSelect, d, client.DefaultOption)
 defer xclient.Close()
```

`NewXClient` 第一个参数是服务路径，在Go实现的服务中就是服务端注册的服务名，Java实现的服务器可以是类名全路径。第二个参数是要调用的方法名。第三个参数是容错模式，这个使用的是同一个服务器多次重试，默认重试三次。第四个参数是在多个服务器同时存在的情况下的负载均衡模式。因为我们这个例子就一个服务器，所以也无所谓随机了，总是选择我们的那一个服务器。第五个参数是服务器的信息，这里我们使用的是点对点的服务发现模式，如果是其它的服务模式，比如etcd，那么就要提供etcd的相关信息。第六个参数可以提供一些额外的信息，比如编码模式、TLS信息等。

客户端所使用的数据结构(只需请求类型和返回类型，不需要服务的定义)和服务端一样。如果客户端和服务器端在一个工程里，所以你可以访问，它们可以共享一套数据结构。但是有些情况下服务方只提供一个说明文档，数据结构还得客户端自己定义：

```
type Args struct { A int `msg:"a"` B int `msg:"b"`}

type Reply struct { C int `msg:"c"`}
```

同步调用的代码如下：

```
args := &example.Args{
        A: 10,
        B: 20,
    }

reply := &example.Reply{}
err := xclient.Call(context.Background(), args, reply)
if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

log.Printf("%d * %d = %d", args.A, args.B, reply.C)
```

`Call`调用是一个同步的调用。

输出结果为：

```
10 * 20 = 200
```

客户端异步调用

客户端异步调用其实是使用`Go`方法，从返回的对象Done字段中可以得到返回的结果信息：

```
    reply := &example.Reply{}
    call, err := xclient.Go(context.Background(), args, reply, n
il)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

    replyCall := <-call.Done
    if replyCall.Error != nil {
        log.Fatalf("failed to call: %v", replyCall.Error)
    } else {
        log.Printf("%d * %d = %d", args.A, args.B, reply.C)
}
```

开发起来是不是超简单？

多服务调用

我们再看一个复杂点的例子，这个例子中我们部署了两个相同的服务器，这两个服务注册的名字相同，都是计算两个数的乘积。为了区分客户端调用不同的服务，我们故意为一个服务器的乘积放大了十倍，便于我们观察调用的结果。

数据结构如下:

```
type Args struct { A int `msg:"a"` B int `msg:"b"`}

type Reply struct { C int `msg:"c"`}

type Arith1 int

func (t *Arith1) Mul(ctx context.Context, args *Args, reply *Rep
ly) error {
    reply.C = args.A * args.B
    return nil
}

type Arith2 int

func (t *Arith2) Mul(ctx context.Context, args *Args, reply *Rep
ly) error {
    reply.C = args.A * args.B * 10
    return nil
}
```

`Arith2` 的Mul方法中我们将计算结果放大了10倍，所以如果传入两个参数10和20,它返回的结果是2000,而 `Arith` 返回200。

服务器端的代码

在服务器端我们启动两个服务器，每个服务器都注册了相同名称的一个服务 `Arith` ,它们分别监听本地的8972和8973端口:

```
func main() {
    go func() {
        s := server.NewServer(nil)
        s.RegisterName("Arith", new(Arith1), "")
        s.Serve("tcp", "localhost:8972")
    }()

    go func() {
        s := server.NewServer(nil)
        s.RegisterName("Arith", new(Arith2), "")
        s.Serve("tcp", "localhost:8973")
    }()

    select{}
}
```

客户端代码

因为我们没有使用注册中心，所以这里客户端需要定义这两个服务器的信息，然后定义路由方式是随机选取(RandomSelect)，调用十次看看服务器返回的结果：

```
type Args struct { A int `msg:"a"` B int `msg:"b"`}

type Reply struct { C int `msg:"c"`}

func main() {
    d := client.NewMultipleServersDiscovery([]*client.KVPair{{Ke
y: "tcp@localhost:8972"}, {Key: "tcp@localhost:8973"}})
    xclient := client.NewXClient("Arith", "Mul", client.Failtry,
 client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()

    args := &example.Args{
        A: 10,
        B: 20,
    }

    for i :=0; i < 10; i++ {
        reply := &example.Reply{}
        err := xclient.Call(context.Background(), args, reply)
        if err != nil {
            log.Fatalf("failed to call: %v", err)
        }

        log.Printf("%d * %d = %d", args.A, args.B, reply.C)
        }

    }
```

输出结果,可以看到调用基本上随机的分布在两个服务器上。

是不是使用rpcx可以将你的本机方法很方便地转换成服务的调用？在后面的章节中，我们还会使用这个例子演示更多的功能。

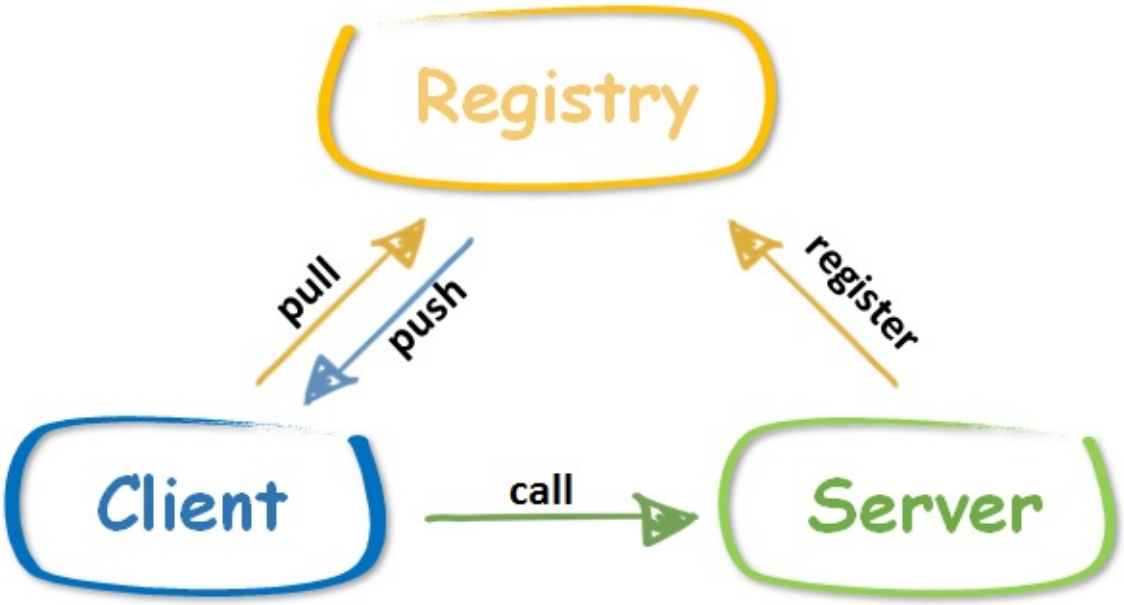
所以的例子都可以在 https://github.com/rpcx-ecosystem/rpcx-examples3 查看。

By *smallnest* *updated* 2017-10-30 01:59:26

服务注册中心

服务注册中心用来实现服务发现和服务的元数据存储。

当前rpcx支持多种注册中心， 并且支持进程内的注册中心，方便开发测试。



rpcx会自动将服务的信息比如服务名，监听地址，监听协议，权重等注册到注册中心，同时还会定时的将服务的吞吐率更新到注册中心。

如果服务意外中断或者宕机，注册中心能够监测到这个事件，它会通知客户端这个服务目前不可用，在服务调用的时候不要再选择这个服务器。

客户端初始化的时候会从注册中心得到服务器的列表，然后根据不同的路由选择选择合适的服务器进行服务调用。 同时注册中心还会通知客户端某个服务暂时不可用。

通常客户端会选择一个服务器进行调用。

下面看看不同的注册中心的使用情况。

Peer2Peer

点对点是最简单的一种注册中心的方式，事实上没有注册中心，客户端直接得到唯一的服务器的地址。

服务器

服务器并没有配置注册中心，而是直接启动。

```
package main

import (
    "flag"

    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/server"
)

var (
    addr = flag.String("addr", "localhost:8972", "server address"
)
)

func main() {
    flag.Parse()

    s := server.Server{}
    s.RegisterName("Arith", new(example.Arith), "")
    s.Serve("tcp", *addr)
}
```

客户端

客户端直接配置了服务器的地址，格式是 `network@ipaddress:port` 的格式，并没有通过第三方组件来查找。

```
package main

import (
    "context"
    "flag"
    "log"

    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/client"
)

var (
    addr = flag.String("addr", "localhost:8972", "server address"
)
)

func main() {
    flag.Parse()

    d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")
    xclient := client.NewXClient("Arith", "Mul", client.Failtry,
 client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()

    args := &example.Args{
        A: 10,
        B: 20,
    }

    reply := &example.Reply{}
    err := xclient.Call(context.Background(), args, reply)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

    log.Printf("%d * %d = %d", args.A, args.B, reply.C)

}
```

当然，如果服务器宕机，客户端访问就会报错。

MultipleServers

上面的方式只能访问一台服务器，假设我们有固定的几台服务器提供相同的服务，我们可以采用这种方式。

服务器

服务器还是和上面的代码一样，只需启动自己的服务，不需要做额外的配置。下面这个例子启动了两个服务。

```
package main

import (
    "flag"

    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/server"
)

var (
    addr1 = flag.String("addr1", "localhost:8972", "server1 addr
ess")
    addr2 = flag.String("addr2", "localhost:9981", "server2 addr
ess")
)

func main() {
    flag.Parse()

    go createServer(*addr1)
    go createServer(*addr2)

    select {}
}

func createServer(addr string) {
    s := server.NewServer(nil)
    s.RegisterName("Arith", new(example.Arith), "")
    s.Serve("tcp", addr)
}
```

客户端

客户度需要使用 `MultipleServersDiscovery` 来配置同一个服务的多个服务器地址，这样客户端就能基于规则从中选择一个进行调用。

可以看到，除了初始化 `XClient` 有所不同外，实际调用服务是一样的， 后面介绍的注册中心也是一样，只有初始化客户端有所不同，后续的调用都一样。

```
package main

import (
    "context"
    "flag"
    "log"

    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/client"
)

var (
    addr1 = flag.String("addr1", "tcp@localhost:8972", "server1
address")
    addr2 = flag.String("addr2", "tcp@localhost:9981", "server2
address")
)

func main() {
    flag.Parse()

    d := client.NewMultipleServersDiscovery([]*client.KVPair{{Ke
y: *addr1}, {Key: *addr2}})
    xclient := client.NewXClient("Arith", "Mul", client.Failtry,
 client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()

    args := &example.Args{
        A: 10,
        B: 20,
    }

    reply := &example.Reply{}
    err := xclient.Call(context.Background(), args, reply)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

    log.Printf("%d * %d = %d", args.A, args.B, reply.C)

}
```

ZooKeeper

Apache ZooKeeper是Apache软件基金会的一个软件项目，他为大型分布式计算提供开源的分布式配置服务、同步服务和命名注册。 ZooKeeper曾经是Hadoop的一个子项目，但现在是一个独立的顶级项目。

ZooKeeper的架构通过冗余服务实现高可用性。因此，如果第一次无应答，客户端就可以询问另一台ZooKeeper主机。ZooKeeper节点将它们的数据存储于一个分层的命名空间，非常类似于一个文件系统或一个前缀树结构。客户端可以在节点读写，从而以这种方式拥有一个共享的配置服务。更新是全序的。

使用ZooKeeper的公司包括Rackspace、雅虎和eBay，以及类似于象Solr这样的开源企业级搜索系统。

ZooKeeper Atomic Broadcast (ZAB)协议是一个类似Paxos的协议，但也有所不同。

Zookeeper一个应用场景就是服务发现，这在Java生态圈中得到了广泛的应用。Go也可以使用Zookeeper，尤其是在和Java项目混布的情况。

服务器

基于rpcx用户的反馈， rpcx 3.0进行了重构，目标之一就是对rpcx进行简化， 因为有些用户可能只需要zookeeper的特性，而不需要etcd、consul等特性。rpcx解决这个问题的方式就是使用 `tag` ，需要你在编译的时候指定所需的特性的 `tag` 。

比如下面这个例子， 需要加上 `-tags zookeeper` 这个参数， 如果需要多个特性，可以使用 `-tags "tag1 tag2 tag3"` 这样的参数。

服务端使用Zookeeper唯一的工作就是设置 `ZooKeeperRegisterPlugin` 这个插件。

它主要配置几个参数：

- ServiceAddress: 本机的监听地址， 这个对外暴露的监听地址， 格式为 `tcp@ipaddress:port`
- ZooKeeperServers: Zookeeper集群的地址
- BasePath: 服务前缀。 如果有多个项目同时使用zookeeper，避免命名冲突，可以设置这个参数，为当前的服务设置命名空间
- Metrics: 用来更新服务的TPS
- UpdateInterval: 服务的刷新间隔， 如果在一定间隔内(当前设为2 * UpdateInterval)没有刷新,服务就会从Zookeeper中删除

需要说明的是：插件必须在注册服务之前添加到**Server**中，否则插件没有办法获取注册的服务的信息。

```
// go run -tags zookeeper server.go
package main

import (
    "flag"
    "log"
    "time"

    metrics "github.com/rcrowley/go-metrics"
    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/server"
    "github.com/smallnest/rpcx/serverplugin"
)

var (
    addr     = flag.String("addr", "localhost:8972", "server add
ress")
    zkAddr   = flag.String("zkAddr", "localhost:2181", "zookeepe
r address")
    basePath = flag.String("base", "/rpcx_test", "prefix path")
)

func main() {
    flag.Parse()

    s := server.NewServer(nil)
    addRegistryPlugin(s)

    s.RegisterName("Arith", new(example.Arith), "")
    s.Serve("tcp", *addr)
}

func addRegistryPlugin(s *server.Server) {

    r := &serverplugin.ZooKeeperRegisterPlugin{
        ServiceAddress:   "tcp@" + *addr,
        ZooKeeperServers: []string{*zkAddr},
        BasePath:         *basePath,
        Metrics:          metrics.NewRegistry(),
```

```
        UpdateInterval:   time.Minute,
    }
    err := r.Start()
    if err != nil {
        log.Fatal(err)
    }
    s.Plugins.Add(r)
}
```

客户端

客户端需要设置 `ZookeeperDiscovery`，指定 `basePath` 和zookeeper集群的地址。

```
// go run -tags zookeeper client.go
package main

import (
    "context"
    "flag"
    "log"

    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/client"
)

var (
    zkAddr   = flag.String("zkAddr", "localhost:2181", "zookeepe
r address")
    basePath = flag.String("base", "/rpcx_test/Arith", "prefix p
ath")
)

func main() {
    flag.Parse()

    d := client.NewZookeeperDiscovery(*basePath, []string{*zkAdd
r}, nil)
    xclient := client.NewXClient("Arith", "Mul", client.Failtry,
 client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()
```

```
    args := &example.Args{
        A: 10,
        B: 20,
    }

    reply := &example.Reply{}
    err := xclient.Call(context.Background(), args, reply)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

    log.Printf("%d * %d = %d", args.A, args.B, reply.C)

}
```

Etcd

etcd 是 CoreOS 团队于 2013 年 6 月发起的开源项目，它的目标是构建一个高可用的分布式键值（key-value）数据库，基于 Go 语言实现。我们知道，在分布式系统中，各种服务的配置信息的管理分享，服务的发现是一个很基本同时也是很重要的问题。CoreOS 项目就希望基于 etcd 来解决这一问题。

因为是用Go开发的，在Go的生态圈中得到广泛的应用。当然，因为etcd提供了RESTful的接口，其它语言也可以使用。

etcd registry使用和zookeeper非常相像。

编译的时候需要加上 `etcd` tag。

服务器

服务器需要增加 `EtcdRegisterPlugin` 插件， 配置参数和Zookeeper的插件相同。

它主要配置几个参数：

- ServiceAddress: 本机的监听地址， 这个对外暴露的监听地址， 格式为 `tcp@ipaddress:port`
- EtcdServers: etcd集群的地址
- BasePath: 服务前缀。 如果有多个项目同时使用zookeeper，避免命名冲突，

可以设置这个参数，为当前的服务设置命名空间

- Metrics: 用来更新服务的TPS
- UpdateInterval: 服务的刷新间隔，如果在一定间隔内(当前设为2 * UpdateInterval)没有刷新,服务就会从etcd中删除

再说明一次：插件必须在注册服务之前添加到**Server**中，否则插件没有办法获取注册的服务的信息。以下的插件相同，就不赘述了

```
// go run -tags etcd server.go
package main

import (
    "flag"
    "log"
    "time"

    metrics "github.com/rcrowley/go-metrics"
    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/server"
    "github.com/smallnest/rpcx/serverplugin"
)

var (
    addr     = flag.String("addr", "localhost:8972", "server address")
    etcdAddr = flag.String("etcdAddr", "localhost:2379", "etcd address")
    basePath = flag.String("base", "/rpcx_test", "prefix path")
)

func main() {
    flag.Parse()

    s := server.NewServer(nil)
    addRegistryPlugin(s)

    s.RegisterName("Arith", new(example.Arith), "")
    s.Serve("tcp", *addr)
}

func addRegistryPlugin(s *server.Server) {
```

```
    r := &serverplugin.EtcdRegisterPlugin{
        ServiceAddress: "tcp@" + *addr,
        EtcdServers:    []string{*etcdAddr},
        BasePath:       *basePath,
        Metrics:        metrics.NewRegistry(),
        UpdateInterval: time.Minute,
    }
    err := r.Start()
    if err != nil {
        log.Fatal(err)
    }
    s.Plugins.Add(r)
}
```

客户端

客户端需要设置 `EtcdDiscovery` 插件，设置 `basepath` 和etcd集群的地址。

```
// go run -tags etcd client.go
package main

import (
    "context"
    "flag"
    "log"

    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/client"
)

var (
    etcdAddr = flag.String("etcdAddr", "localhost:2379", "etcd a
ddress")
    basePath = flag.String("base", "/rpcx_test/Arith", "prefix p
ath")
)

func main() {
    flag.Parse()

    d := client.NewEtcdDiscovery(*basePath, []string{*etcdAddr},
```

```
  nil)
     xclient := client.NewXClient("Arith", "Mul", client.Failtry,
  client.RandomSelect, d, client.DefaultOption)
     defer xclient.Close()

     args := &example.Args{
         A: 10,
         B: 20,
     }

     reply := &example.Reply{}
     err := xclient.Call(context.Background(), args, reply)
     if err != nil {
         log.Fatalf("failed to call: %v", err)
     }

     log.Printf("%d * %d = %d", args.A, args.B, reply.C)

 }
```

Consul

Consul是HashiCorp公司推出的开源工具，用于实现分布式系统的服务发现与配置。Consul是分布式的、高可用的、 可横向扩展的。它具备以下特性:

- 服务发现: Consul提供了通过DNS或者HTTP接口的方式来注册服务和发现服务。一些外部的服务通过Consul很容易的找到它所依赖的服务。
- 健康检测: Consul的Client提供了健康检查的机制，可以通过用来避免流量被转发到有故障的服务上。
- Key/Value存储: 应用程序可以根据自己的需要使用Consul提供的Key/Value存储。 Consul提供了简单易用的HTTP接口，结合其他工具可以实现动态配置、功能标记、领袖选举等等功能。
- 多数据中心: Consul支持开箱即用的多数据中心. 这意味着用户不需要担心需要建立额外的抽象层让业务扩展到多个区域。

Consul也是使用Go开发的，在Go生态圈也被广泛应用。

使用 `consul` 需要添加 `consul` tag。

服务器

服务器端的开发和zookeeper、consul类似。

需要配置 `ConsulRegisterPlugin` 插件。

它主要配置几个参数：

- ServiceAddress: 本机的监听地址， 这个对外暴露的监听地址， 格式为 `tcp@ipaddress:port`
- ConsulServers: consul集群的地址
- BasePath: 服务前缀。 如果有多个项目同时使用zookeeper，避免命名冲突，可以设置这个参数，为当前的服务设置命名空间
- Metrics: 用来更新服务的TPS
- UpdateInterval: 服务的刷新间隔， 如果在一定间隔内(当前设为2 * UpdateInterval)没有刷新,服务就会从consul中删除

```
// go run -tags consul server.go
package main

import (
    "flag"
    "log"
    "time"

    metrics "github.com/rcrowley/go-metrics"
    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/server"
    "github.com/smallnest/rpcx/serverplugin"
)

var (
    addr       = flag.String("addr", "localhost:8972", "server a
ddress")
    consulAddr = flag.String("consulAddr", "localhost:8500", "co
nsul address")
    basePath   = flag.String("base", "/rpcx_test", "prefix path"
)
)

func main() {
    flag.Parse()

    s := server.NewServer(nil)
```

```
    addRegistryPlugin(s)

    s.RegisterName("Arith", new(example.Arith), "")
    s.Serve("tcp", *addr)
}

func addRegistryPlugin(s *server.Server) {

    r := &serverplugin.ConsulRegisterPlugin{
        ServiceAddress: "tcp@" + *addr,
        ConsulServers:  []string{*consulAddr},
        BasePath:       *basePath,
        Metrics:        metrics.NewRegistry(),
        UpdateInterval: time.Minute,
    }
    err := r.Start()
    if err != nil {
        log.Fatal(err)
    }
    s.Plugins.Add(r)
}
```

客户端

配置 `ConsulDiscovery` ，使用 `basepath` 和consul的地址。

```
package main

import (
    "context"
    "flag"
    "log"

    example "github.com/rpcx-ecosystem/rpcx-examples3"
    "github.com/smallnest/rpcx/client"
)

var (
    consulAddr = flag.String("consulAddr", "localhost:8500", "co
nsul address")
    basePath   = flag.String("base", "/rpcx_test/Arith", "prefix
 path")
)

func main() {
    flag.Parse()

    d := client.NewConsulDiscovery(*basePath, []string{*consulAd
dr}, nil)
    xclient := client.NewXClient("Arith", "Mul", client.Failtry,
 client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()

    args := &example.Args{
        A: 10,
        B: 20,
    }

    reply := &example.Reply{}
    err := xclient.Call(context.Background(), args, reply)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

    log.Printf("%d * %d = %d", args.A, args.B, reply.C)

}
```

mDNS

服务器

客户端

Inprocess

服务器

客户端

By *smallnest* *updated* 2017-10-30 01:59:26

服务器端开发

前面两章已经看到了简单的服务器的开发，接下来的两章我们将了解的服务器和客户端更详细的设置和开发。

服务器提供了几种启动服务器的方法：

```
func (s *Server) Serve(network, address string)
func (s *Server) ServeByHTTP(ln net.Listener, rpcPath string)
func (s *Server) ServeHTTP(w http.ResponseWriter, req *http.Requ
est)
func (s *Server) ServeListener(ln net.Listener)
func (s *Server) ServeTLS(network, address string, config *tls.C
onfig)
func (s *Server) Start(network, address string)
func (s *Server) StartTLS(network, address string, config *tls.C
onfig)
```

`ServeXXX` 方法的方法会阻塞当前的goroutine，如果不想阻塞当前的goroutine，可以调用 `StartXXX` 方法。

一些例子：

```
ln, _ := net.Listen("tcp", "127.0.0.1:0")
go s.ServeByHTTP(ln, "foo")
```

```
server.Start("tcp", "127.0.0.1:0")
```

`RegisterName` 用来注册服务，可以指定服务名和它的元数据。

```
func (s *Server) RegisterName(name string, service interface{},
metadata ...string)
```

另外Server还提供其它几个方法。 `NewServer` 用来创建一个新的Server对象。

```
func NewServer() *Server
```

`Address` 返回Server监听的地址。如果你设置的时候设置端口为0,则Go会选择一个合适的端口号作为监听的端口号，通过这个方法可以返回实际的监听地址和端口。

```
func (s *Server) Address() string
```

`Close` 则关闭监听，停止服务。

```
func (s *Server) Close() error
```

`Auth` 提供一个身份验证的方法，它在你需要实现服务权限设置的时候很有用。客户端会将一个身份验证的token传给服务器，但是rpcx并不限制你采用何种验证方式，普通的用户名+密码的明文也可以，OAuth2也可以，只要客户端和服务器端协商好一致的验证方式即可。 rpcx会将解析的验证token,服务名称以及额外的信息传给下面的设置的方法 `AuthorizationFunc` ,你需要实现这个方法。比如通过OAuth2的方式，解析出access_token,然后检查这个access_toekn是否对这个服务有授权。

```
func (s *Server) Auth(fn AuthorizationFunc) error
```

我们可以看一个例子,服务器的代码如下：

```
func main() {
 server := rpcx.NewServer()

 fn := func(p *rpcx.AuthorizationAndServiceMethod) error {
  if p.Authorization != "0b79bab50daca910b000d4f1a2b675d604257e4
2" || p.Tag != "Bearer" {
   fmt.Printf("error: wrong Authorization: %s, %s\n", p.Authoriz
ation, p.Tag)
   return errors.New("Authorization failed ")
  }

  fmt.Printf("Authorization success: %+v\n", p)
  return nil
 }

 server.Auth(fn)

 server.RegisterName("Arith", new(Arith)) server.Serve("tcp", "1
27.0.0.1:8972")}
```

这个简单的例子演示了只有用户使用"Bear 0b79bab50daca910b000d4f1a2b675d604257e42" acces_token 访问时才提供服务，否则报错。

我们可以看一下客户端如何设置这个access_token:

```
func main() {
 s := &rpcx.DirectClientSelector{Network: "tcp", Address: "127.0
.0.1:8972", DialTimeout: 10 * time.Second}
 client := rpcx.NewClient(s)

 //add Authorization info
 err := client.Auth("0b79bab50daca910b000d4f1a2b675d604257e42_AB
C", "Bearer")
 if err != nil {
  fmt.Printf("can't add auth plugin: %#v\n", err)
 }

 args := &Args{7, 8}
 var reply Reply
 err = client.Call("Arith.Mul", args, &reply)
 if err != nil {
  fmt.Printf("error for Arith: %d*%d, %v \n", args.A, args.B, er
r)
 } else {
  fmt.Printf("Arith: %d*%d=%d \n", args.A, args.B, reply.C)
 }

 client.Close()
}
```

客户端很简单，调用 `Auth` 方法设置access_token和token_type(optional)即可。

rpcx创建了一个缺省的 `Server` ，并提供了一些辅助方法来暴露Server的方法，因此你也可以直接调用 `rpcx.XXX` 方法来调用缺省的Server的方法。

`Server` 是一个struct类型，它还包含一些有用的字段：

```
type Server struct {
 ServerCodecFunc ServerCodecFunc //PluginContainer must be confi
gured before starting and Register plugins must be configured be
fore invoking RegisterName method
 PluginContainer IServerPluginContainer
 //Metadata describes extra info about this service, for example
, weight, active status Metadata string
 Timeout time.Duration
 ReadTimeout time.Duration
 WriteTimeout time.Duration
 // contains filtered or unexported fields
 }
```

`ServerCodecFunc` 用来设置序列化方法。 `PluginContainer` 是插件容器，服务器端设置的插件都要加入到这个容器之中，比如注册中心、日志、监控、限流等等。 你还可以设置超时的值。超时的值很容易被忽视，但是在实际的开发应用中却非常的有用和必要。因为经常会遇到网络的一些意外的状况，如果不设置超时，很容易导致服务器性能的下降。

By *smallnest* *updated* 2017-10-30 01:59:26

客户端开发

不同的注册中心有不同的ClientSelector, rpcx利用ClientSelector配置到注册中心的连接以及客户端的连接，然后根据ClientSelector生成rpcx.Client。

注册中心那一章我们已经介绍了三种ClientSeletor,目前rpcx支持五种ClientSelector:

```
type ConsulClientSelector

func NewConsulClientSelector(consulAddress string, serviceName s
tring, sessionTimeout time.Duration, sm rpcx.SelectMode, dailTim
eout time.Duration) *ConsulClientSelector

func NewEtcdClientSelector(etcdServers []string, basePath string
, sessionTimeout time.Duration, sm rpcx.SelectMode, dailTimeout
time.Duration) *EtcdClientSelector

func NewMultiClientSelector(servers []*ServerPeer, sm rpcx.Selec
tMode, dailTimeout time.Duration) *MultiClientSelector

func NewZooKeeperClientSelector(zkServers []string, basePath str
ing, sessionTimeout time.Duration, sm rpcx.SelectMode, dailTimeo
ut time.Duration) *ZooKeeperClientSelector

type DirectClientSelector struct { Network, Address string DialT
imeout time.Duration Client *Client }
```

它们都实现了 `ClientSelector` 接口。

```
type ClientSelector interface {
 //Select returns a new client and it also update current client

 Select(clientCodecFunc ClientCodecFunc, options ...interface{})
 (*rpc.Client, error)
 //SetClient set current client
 SetClient(*Client)
 SetSelectMode(SelectMode)
 //AllClients returns all Clients
 AllClients(clientCodecFunc ClientCodecFunc) []*rpc.Client }
```

`Select` 从服务列表中根据路由算法选择一个服务来调用，它返回的是一个rpc.Client对象，这个对象建立了对实际选择的服务的连接。 `SetClient` 用来建立对当前选择的Client的引用，它用来关联一个rpcx.Client。 `SetSelectMode` 可以用来设置路由算法，路由算法根据一定的规则从服务列表中来选择服务。 `AllClients` 返回对所有的服务的rpc.Client slice。

所以你可以看到，底层rpcx还是利用官方库 `net/rpc` 进行通讯的。因此通过 `NewClient` 得到的rpcx.Client调用方法和官方库类似， `Call` 是同步调用， `Go` 是异步调用，调用完毕后可以 `Close` 释放连接。 `Auth` 方法可以设置授权验证的信息。

```
type Client

func NewClient(s ClientSelector) *Client

func (c *Client) Auth(authorization, tag string) error

func (c *Client) Call(serviceMethod string, args interface{}, re
ply interface{}) (err error)

func (c *Client) Close() error

func (c *Client) Go(serviceMethod string, args interface{}, repl
y interface{}, done chan *rpc.Call) *rpc.Call
```

rpcx.Client的定义如下：

```
type Client struct {
 ClientSelector ClientSelector
 ClientCodecFunc ClientCodecFunc
 PluginContainer IClientPluginContainer
 FailMode FailMode
 TLSConfig *tls.Config
 Retries int
 //Timeout sets deadline for underlying net.Conns
 Timeout time.Duration
 //Timeout sets readdeadline for underlying net.Conns
 ReadTimeout time.Duration
 //Timeout sets writedeadline for underlying net.Conns
 WriteTimeout time.Duration
 // contains filtered or unexported fields
 }
```

你可以设置rpcx.Client的序列化方式、TLS、失败模式、失败重试次数，超时等，还可以往它的插件容器中增加插件。

重试次数只在失败模式Failover或者Failtry下起作用。

By *smallnest* *updated* 2017-10-30 01:59:26