目　录

Gateway

HTTP 方式调用

双向通讯

介绍

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。

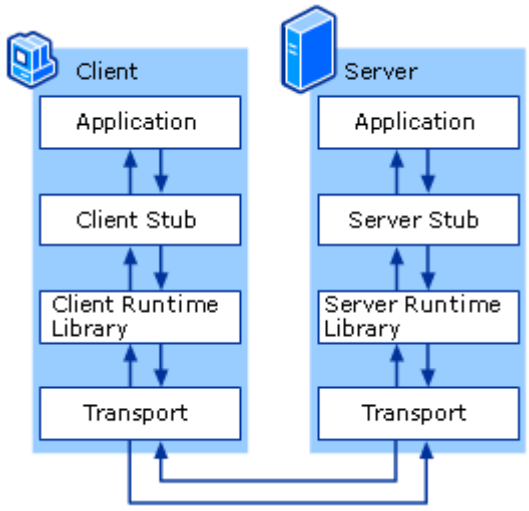
RPC介绍

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

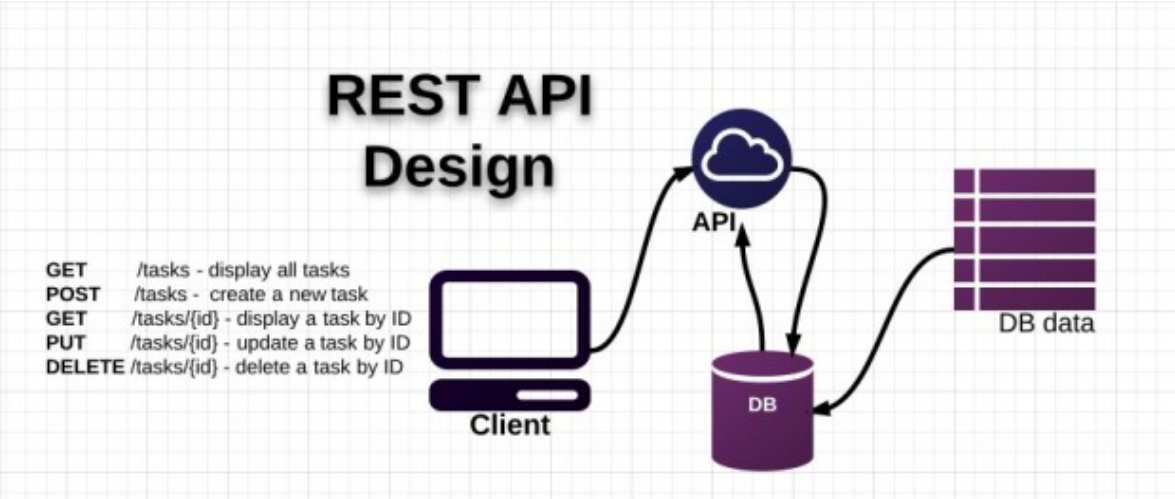
RPC 的消息传输可以通过 TCP、UDP 或者 HTTP等，所以有时候我们称之为 RPC over TCP、 RPC over HTTP。RPC 通过 HTTP 传输消息的时候和 RESTful的架构是类似的，但是也有不同。

首先我们比较 RPC over HTTP 和 RESTful。

首先 RPC 的客户端和服务器端是紧耦合的，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

开发起步

快速起步

快速起步

安装

首先，你需要安装 rpcx:

```
go get -u -v github.com/smallnest/rpcx/...
```

这一步只会安装 rpcx 的基础功能。如果你想要使用 etcd 作为注册中心，你需要加上 `etcd` 这个标签。 (see build tags)

```
go get -u -v -tags "etcd" github.com/smallnest/rpcx/...
```

如果你想要使用 quic ，你也需要加上 `quic` 这个标签。

```
go get -u -v -tags "quic etcd" github.com/smallnest/rpcx/...
```

方便起见，我推荐你安装所有的tags，即使你现在并不需要他们：

```
go get -u -v -tags "reuseport quic kcp zookeeper etcd consul ping" github.com/smallnest/rpcx/...
```

tags 对应:

- **quic**: 支持 quic 协议
- **kcp**: 支持 kcp 协议
- **zookeeper**: 支持 zookeeper 注册中心
- **etcd**: 支持 etcd 注册中心
- **consul**: 支持 consul 注册中心
- **ping**: 支持 网络质量负载均衡
- **reuseport**: 支持 reuseport

实现Service

实现一个 Sevice 就像写一个单纯的 Go 结构体:

```
import "context"

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 *Reply) error {
    reply.C = args.A * args.B
    return nil
}
```

`Arith` 是一个 Go 类型，并且它有一个方法 `Mul` 。
方法 `Mul` 的 第 1 个参数是 `context.Context` 。
方法 `Mul` 的 第 2 个参数是 `args` ， `args` 包含了请求的数据 `A` 和 `B` 。
方法 `Mul` 的 第 3 个参数是 `reply` ， `reply` 是一个指向了 `Reply` 结构体的指针。
方法 `Mul` 的 返回类型是 error (可以为 nil)。
方法 `Mul` 把 `A * B` 的结果 赋值到 `Reply.C`

现在你已经定义了一个叫做 `Arith` 的 service， 并且为它实现了 `Mul` 方法。 下一步骤中， 我们将会继续介绍如何把这个服务注册给服务器，并且如何用 client 调用它。

实现 Server

三行代码就可以注册一个服务：

```
    s := server.NewServer()
    s.RegisterName("Arith", new(Arith), "")
    s.Serve("tcp", ":8972")
```

这里你把你的服务命名 `Arith` 。

你可以按照如下的代码注册服务。

```
s.Register(new(example.Arith), "")
```

这里简单使用了服务的 类型名称 作为 服务名。

实现 Client

```
    // #1
    d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")
    // #2
    xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()

    // #3
    args := &example.Args{
        A: 10,
        B: 20,
    }

    // #4
    reply := &example.Reply{}

    // #5
    err := xclient.Call(context.Background(), "Mul", args, reply)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }
```

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

`#1` 定义了使用什么方式来实现服务发现。 在这里我们使用最简单的 `Peer2PeerDiscovery` （点对点）。客户端直连服务器来获取服务地址。

`#2` 创建了 `XClient` ， 并且传进去了 `FailMode` 、 `SelectMode` 和默认选项。
FailMode 告诉客户端如何处理调用失败：重试、快速返回，或者 尝试另一台服务器。
SelectMode 告诉客户端如何在有多台服务器提供了同一服务的情况下选择服务器。

`#3` 定义了请求：这里我们想获得 `10 * 20` 的结果。 当然我们可以自己算出结果是 `200` ，但是我们仍然想确认这与服务器的返回结果是否一致。

`#4` 定义了响应对象， 默认值是0值， 事实上 rpcx 会通过它来知晓返回结果的类型，然后把结果反序列化到这个对象。

`#5` 调用了远程服务并且同步获取结果。

异步调用 Service

以下的代码可以异步调用服务：

```
    d := client.NewPeer2PeerDiscovery("tcp@"+*addr2, "")
    xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()

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

    reply := &example.Reply{}
    call, err := xclient.Go(context.Background(), "Mul", args, reply, nil)
    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)
    }
```

你必须使用 `xclient.Go` 来替换 `xclient.Call` ， 然后把结果返回到一个channel里。你可以从chnanel里监听调用结果。

服务端开发示例

Server

Example: 102basic

你可以在服务端实现Service。

Service的类型并不重要。你可以使用自定义类型来保持状态，或者直接使用 `struct{}` 、 `int` 。

你需要启动一个TCP或UDP服务器来暴露Service。

你也可以添加一些plugin来为服务器增加新特性。

Service

作为服务提供者，首先你需要定义服务。
当前rpcx仅支持 可导出的 `methods` （方法） 作为服务的函数。 （see 可导出）
并且这个可导出的方法必须满足以下的要求：

- 必须是可导出类型的方法
- 接受3个参数，第一个是 `context.Context` 类型，其他2个都是可导出（或内置）的类型。
- 第3个参数是一个指针
- 有一个 error 类型的返回值

你可以使用 `RegisterName` 来注册 `rcvr` 的方法，这里这个服务的名字叫做 `name` 。
如果你使用 `Register` ， 生成的服务的名字就是 `rcvr` 的类型名。
你可以在注册中心添加一些元数据供客户端或者服务管理者使用。例如
`weight` 、 `geolocation` 、 `metrics` 。

```
func (s *Server) Register(rcvr interface{}, metadata string) error
func (s *Server) RegisterName(name string, rcvr interface{}, metadata string) error
```

这里是一个实现了 `Mul` 方法的例子：

```
import "context"

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 *Reply) error {
    reply.C = args.A * args.B
    return nil
}
```

在这个例子中，你可以定义 `Arith` 为 `struct{}` 类型， 它不会影响到这个服务。
你也可以定义 `args` 为 `Args` ， 也不会产生影响。

Server

在你定义完服务后，你会想将它暴露出去来使用。你应该通过启动一个TCP或UDP服务器来监听请求。

服务器支持以如下这些方式启动，监听请求和关闭：

```
    func NewServer(options ...OptionFn) *Server
    func (s *Server) Close() error
    func (s *Server) RegisterOnShutdown(f func())
    func (s *Server) Serve(network, address string) (err error)
    func (s *Server) ServeHTTP(w http.ResponseWriter, req *http.Request)
```

首先你应使用 `NewServer` 来创建一个服务器实例。其次你可以调用 `Serve` 或者 `ServeHTTP` 来监听请求。

服务器包含一些字段（有一些是不可导出的）：

```
type Server struct {
    Plugins PluginContainer
    // AuthFunc 可以用来鉴权
    AuthFunc func(ctx context.Context, req *protocol.Message, token string) error
    // 包含过滤后或者不可导出的字段
}
```

`Plugins` 包含了服务器上所有的插件。我们会在之后的章节介绍它。

`AuthFunc` 是一个可以检查客户端是否被授权了的鉴权函数。我们也会在之后的章节介绍它。

rpcx 提供了 3个 `OptionFn` 来设置启动选项：

```

    func WithReadTimeout(readTimeout time.Duration) OptionFn
    func WithTLSConfig(cfg *tls.Config) OptionFn
    func WithWriteTimeout(writeTimeout time.Duration) OptionFn
```

可以设置 读超时、写超时和tls证书。

`ServeHTTP` 将服务通过HTTP暴露出去。

`Serve` 通过TCP或UDP协议与客户端通信。

rpcx 支持如下的网络类型：

- tcp: 推荐使用
- http: 通过劫持http连接实现
- unix: unix domain sockets
- reuseport: 要求 `SO_REUSEPORT` socket 选项, 仅支持 Linux kernel 3.9+
- quic: support quic protocol
- kcp: sopport kcp protocol

下面是一个服务器的示例代码：

```
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.NewServer()
    //s.RegisterName("Arith", new(example.Arith), "")
    s.Register(new(example.Arith), "")
    s.Serve("tcp", *addr)
}
```

客户端开发示例

Client

Example: 101basic

客户端使用和服务同样的通信协议来发送请求和获取响应。

```
type Client struct {
    Conn net.Conn

    Plugins PluginContainer
    // 包含过滤后的或者不可导出的字段
}
```

`Conn` 代表客户端与服务器之前的连接。 `Plugins` 包含了客户端启用的插件。

他有这些方法：

```
    func (client *Client) Call(ctx context.Context, servicePath, serviceMethod string, args interface{}, repl
y interface{}) error
    func (client *Client) Close() error
    func (c *Client) Connect(network, address string) error
    func (client *Client) Go(ctx context.Context, servicePath, serviceMethod string, args interface{}, reply
interface{}, done chan *Call) *Call
    func (client *Client) IsClosing() bool
    func (client *Client) IsShutdown() bool
```

`Call` 代表对服务同步调用。客户端在收到响应或错误前一直是阻塞的。 然而 `Go` 是异步调用。它返回一个指向 Call 的指针， 你可以检查 `*Call` 的值来获取返回的结果或错误。

`Close` 会关闭所有与服务的连接。他会立刻关闭连接，不会等待未完成的请求结束。

`IsClosing` 表示客户端是关闭着的并且不会接受新的调用。
`IsShutdown` 表示客户端不会接受服务返回的响应。

`Client` uses the default CircuitBreaker (circuit.NewRateBreaker(0.95, 100)) to handle errors. This is a poplular rpc error handling style. When the error rate hits the threshold, this service is marked unavailable in 10 second window. You can implement your customzied CircuitBreaker.
`Client` 使用默认的 CircuitBreaker (circuit.NewRateBreaker(0.95, 100)) 来处理错误。这是rpc处理错误的普遍做法。当出错率达到阈值， 这个服务就会在接下来的10秒内被标记为不可用。你也可以实现你自己的 CircuitBreaker。

下面是客户端的例子：

```
    client := &Client{
        option: DefaultOption,
    }

    err := client.Connect("tcp", addr)
    if err != nil {
        t.Fatalf("failed to connect: %v", err)
    }
    defer client.Close()
```

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

    reply := &Reply{}
    err = client.Call(context.Background(), "Arith", "Mul", args, reply)
    if err != nil {
        t.Fatalf("failed to call: %v", err)
    }

    if reply.C != 200 {
        t.Fatalf("expect 200 but got %d", reply.C)
    }
```

XClient

`XClient` 是对客户端的封装，增加了一些服务发现和服务治理的特性。

```
type XClient interface {
    SetPlugins(plugins PluginContainer)
    ConfigGeoSelector(latitude, longitude float64)
    Auth(auth string)

    Go(ctx context.Context, serviceMethod string, args interface{}, reply interface{}, done chan *Call) (*Cal
l, error)
    Call(ctx context.Context, serviceMethod string, args interface{}, reply interface{}) error
    Broadcast(ctx context.Context, serviceMethod string, args interface{}, reply interface{}) error
    Fork(ctx context.Context, serviceMethod string, args interface{}, reply interface{}) error
    Close() error
}
```

`SetPlugins` 方法可以用来设置 Plugin 容器， `Auth` 可以用来设置鉴权token。

`ConfigGeoSelector` 是一个可以通过地址位置选择器来设置客户端的经纬度的特别方法。

一个XCLinet只对一个服务负责，它可以通过 `serviceMethod` 参数来调用这个服务的所有方法。如果你想调用多个服务，你必须为每个服务创建一个XClient。

一个应用中，一个服务只需要一个共享的XClient。它可以被通过goroutine共享，并且是协程安全的。

`Go` 代表异步调用， `Call` 代表同步调用。

XClient对于一个服务节点使用单一的连接，并且它会缓存这个连接直到失效或异常。

服务发现

rpcx 支持许多服务发现机制，你也可以实现自己的服务发现。

-
  - Peer to Peer: 客户端直连每个服务节点。 the client connects the single service directly. It acts like the `client` type.
  - Peer to Multiple: 客户端可以连接多个服务。服务可以被编程式配置。
  - Zookeeper: 通过 zookeeper 寻找服务。
  - Etcd: 通过 etcd 寻找服务。

- Consul: 通过 consul 寻找服务。
- mDNS: 通过 mDNS 寻找服务（支持本地服务发现）。
- In process: 在同一进程寻找服务。客户端通过进程调用服务，不走TCP或UDP，方便调试使用。

下面是一个同步的 rpcx 例子：

```
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", 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(), "Mul", args, reply)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

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

}
```

服务治理（失败模式与负载均衡）

在一个大规模的rpc系统中，有许多服务节点提供同一个服务。客户端如何选择最合适的节点来调用呢？如果调用失败，客户端应该选择另一个节点或者立即返回错误？这里就有了故障模式和负载均衡的问题。

rpcx 支持 故障模式：

- Failfast：如果调用失败，立即返回错误
- Failover：选择其他节点，直到达到最大重试次数
- Failtry：选择相同节点并重试，直到达到最大重试次数

对于负载均衡，rpcx 提供了许多选择器：

- Random： 随机选择节点

- Roundrobin： 使用 roundrobin 算法选择节点
- Consistent hashing: 如果服务路径、方法和参数一致，就选择同一个节点。使用了非常快的jump consistent hash算法。
- Weighted: 根据元数据里配置好的权重(`weight=xxx`)来选择节点。类似于nginx里的实现(smooth weighted algorithm)
- Network quality: 根据 `ping` 的结果来选择节点。网络质量越好，该节点被选择的几率越大。
- Geography: 如果有多个数据中心，客户端趋向于连接同一个数据机房的节点。
- Customized Selector: 如果以上的选择器都不适合你，你可以自己定制选择器。例如一个rpcx用户写过它自己的选择器，他有2个数据中心，但是这些数据中心彼此有限制，不能使用 `Network quality` 来检测连接质量。

下面是一个异步的 rpcx 例子：

```
package main

import (
    "context"
    "flag"
    "log"

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

var (
    addr2 = flag.String("addr", "localhost:8972", "server address")
)

func main() {
    flag.Parse()

    d := client.NewPeer2PeerDiscovery("tcp@"+*addr2, "")
    xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()

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

    reply := &example.Reply{}
    call, err := xclient.Go(context.Background(), "Mul", args, reply, nil)
    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)
    }
}
```

客户端使用了 `Failtry` 模式并且随机选择节点。

广播与群发

特殊情况下，你可以使用 XClient 的 `Broadcast` 和 `Fork` 方法。

```
    Broadcast(ctx context.Context, serviceMethod string, args interface{}, reply interface{}) error
    Fork(ctx context.Context, serviceMethod string, args interface{}, reply interface{}) error
```

`Broadcast` 表示向所有服务器发送请求，只有所有服务器正确返回时才会成功。此时FailMode 和 SelectMode的设置是无效的。请设置超时来避免阻塞。

`Fork` 表示向所有服务器发送请求，只要任意一台服务器正确返回就成功。此时FailMode 和 SelectMode的设置是无效的。

你可以使用 `NewXClient` 来获取一个 XClient 实例。

```
func NewXClient(servicePath string, failMode FailMode, selectMode SelectMode, discovery ServiceDiscovery, opt
ion Option) XClient
```

`NewXClient` 必须使用服务名称作为第一个参数， 然后是 failmode、 selector、 discovery等其他选项。

传输

Transport

rpcx 可以通过 TCP、HTTP、UnixDomain、QUIC和KCP通信。你也可以使用http客户端通过网关或者http调用来访问rpcx服务。

TCP

这是最常用的通信方式。高性能易上手。你可以使用TLS加密TCP流量。

Example: 101basic

服务端使用 `tcp` 做为网络名并且在注册中心注册了名为 `serviceName/tcp@ipaddress:port` 的服务。

```
s.Serve("tcp", *addr)
```

客户端可以这样访问服务：

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

HTTP Connect

你可以发送 `HTTP CONNECT` 方法给 rpcx 服务器。 Rpcx 服务器会劫持这个连接然后将它作为TCP连接来使用。需要注意，客户端和服务端并不使用http请求/响应模型来通信，他们仍然使用二进制协议。

网络名称是 `http` ， 它注册的格式是 `serviceName/http@ipaddress:port` 。

HTTP Connect并不被推荐。 TCP是第一选择。

如果你想使用http 请求/响应 模型来访问服务，你应该使用网关或者http_invoke。

Unixdomain

网络名称是 `unix` 。

Example: unix

QUIC

维基百科上QUIC的介绍 wikipedia

> QUIC (Quick UDP Internet Connections, pronounced quick) is an experimental transport layer network protocol designed by Jim Roskind at Google, initially implemented in 2012, and announced publicly in 2013 as experimentation broadened. QUIC supports a set of multiplexed connections between two endpoints over User Datagram Protocol (UDP), and was designed to provide security protection equivalent to TLS/SSL, along with reduced connection and transport latency, and bandwidth estimation

> in each direction to avoid congestion. QUIC's main goal is to improve perceived performance of connection-oriented web applications that are currently using TCP. It also moves control of the congestion avoidance algorithms into the application space at both endpoints, rather than the kernel space, which it is claimed will allow these algorithms to improve more rapidly.

> In June 2015, an Internet Draft of a specification for QUIC was submitted to the IETF for standardization. A QUIC working group was established in 2016. The QUIC working group foresees multipath support and optional forward error correction (FEC) as the next step. The working group also focuses on network management issues that QUIC may introduce, aiming to produce an applicability and manageability statement in parallel to the actual protocol work. Internet statistics in 2017 suggest that QUIC now accounts for more than 5% of Internet traffic.

网络名称是 `quic` 。

Example: quic

KCP

KCP 是一个快速并且可靠的ARQ协议。

网络名称是 `kcp` 。

当你使用 `kcp` 的时候，你必须设置 `Timeout` ，利用timeout保持连接的检测。因为kcp-go本身不提供keepalive/heartbeat的功能，当服务器宕机重启的时候，原有的连接没有任何异常，只会hang住，我们只能依靠 `Timeout` 避免hang住。

Example: kcp

reuseport

网络名称是 `reuseport` 。

Example: reuseport

它使用 `tcp` 协议并且在linux/uxix服务器上开启 SO_REUSEPORT socket 选项。

TLS

Example: TLS

你可以在服务端配置 TLS：

```
func main() {
    flag.Parse()

    cert, err := tls.LoadX509KeyPair("server.pem", "server.key")
    if err != nil {
        log.Print(err)
        return
    }

    config := &tls.Config{Certificates: []tls.Certificate{cert}}

    s := server.NewServer(server.WithTLSConfig(config))
    s.RegisterName("Arith", new(example.Arith), "")
    s.Serve("tcp", *addr)
}
```

你可以在客户端设置 TLS：

```
func main() {
    flag.Parse()

    d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")

    option := client.DefaultOption

    conf := &tls.Config{
        InsecureSkipVerify: true,
    }

    option.TLSConfig = conf

    xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d, option)
    defer xclient.Close()

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

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

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

}
```

函数为服务

Client

Example: function

通常我们将方法注册为服务的方法，这些方法必须满足以下的要求：

- 必须是可导出类型的方法
- 接受3个参数，第一个是 `context.Context` 类型，其他2个都是可导出（或内置）的类型。
- 第3个参数是一个指针
- 有一个 error 类型的返回值

Rpcx 也支持将纯函数注册为服务，函数必须满足以下的要求：

- 函数可以是可导出的或者不可导出的
- 接受3个参数，第一个是 `context.Context` 类型，其他2个都是可导出（或内置）的类型。
- 第3个参数是一个指针
- 有一个 error 类型的返回值

下面有一个例子。

服务端必须使用 `RegisterFunction` 来注册一个函数并且提供一个服务名。

```
// server.go
type Args struct {
    A int
    B int
}

type Reply struct {
    C int
}

func mul(ctx context.Context, args *Args, reply *Reply) error {
    reply.C = args.A * args.B
    return nil
}

func main() {
    flag.Parse()

    s := server.NewServer()
    s.RegisterFunction("a.fake.service", mul, "")
    s.Serve("tcp", *addr)
}
```

客户端可以通过服务名和函数名来调用服务：

```
// client.go
d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")
xclient := client.NewXClient("a.fake.service", client.Failtry, client.RandomSelect, d, client.DefaultOption)
```

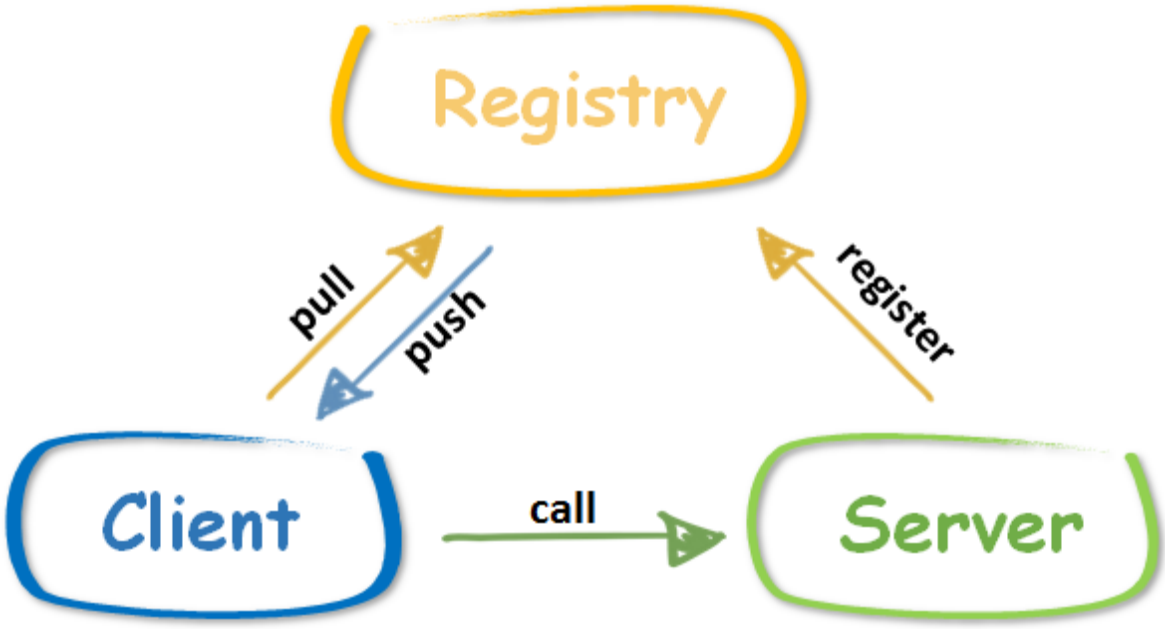
```
defer xclient.Close()

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

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

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

注册中心

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

Example: 102basic

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，连接服务。在系统扩展时，你可以进行一些更改，服务器不需要进行更多的配置
客户端使用 `Peer2PeerDiscovery` 来设置该服务的网络和地址。

由于只有有一个节点，因此选择器是不可用的。

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

注意:rpcx使用 `network @ Host: port` **格式表示一项服务。在** `network` **可以** `tcp` **，** `http` **，** `unix` **，** `quic` **或** `kcp` **。该** `Host` **可以所主机名或IP地址。**

`NewXClient` 必须使用服务名称作为第一个参数，然后使用failmode，selector，discovery和其他选项。

MultipleServers

Example: multiple

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。

如果你有多个服务但没有注册中心.你可以用编码的方式在客户端中配置服务的地址。
服务器不需要进行更多的配置。

客户端使用 `MultipleServersDiscovery` 并仅设置该服务的网络和地址。

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

你必须在MultipleServersDiscovery 中设置服务信息和元数据。如果添加或删除了某些服务，你可以调用 `MultipleServersDiscovery.Update` 来动态更新服务。

```
func (d *MultipleServersDiscovery) Update(pairs []*KVPair)
```

ZooKeeper

Example: 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
func main() {
    flag.Parse()

    s := server.NewServer()
    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
    d := client.NewZookeeperDiscovery(*basePath, "Arith",[]string{*zkAddr}, nil)
    xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()
```

Etcd

Example: 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
func main() {
    flag.Parse()

    s := server.NewServer()
    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
    d := client.NewEtcdDiscovery(*basePath, "Arith",[]string{*etcdAddr}, nil)
    xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d, client.DefaultOption)
    defer xclient.Close()
```

Consul

Example: 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consul，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
func main() {
    flag.Parse()

    s := server.NewServer()
    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。

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

mDNS

Example: mDNS

mdns 即多播dns（Multicast DNS），mDNS主要实现了在没有传统DNS服务器的情况下使局域网内的主机实现相互发现和通信，使用的端口为5353，遵从dns协议，使用现有的DNS信息结构、名语法和资源记录类型。并且没有指定新的操作代码或响应代码。

在局域网中，设备和设备之前相互通信需要知道对方的ip地址的，大多数情况，设备的ip不是静态ip地址，而是通过dhcp 协议动态分配的ip 地址，如何设备发现呢，就是要mdns大显身手，例如：现在物联网设备和app之间的通信，要么app通过广播，要么通过组播，发一些特定信息，感兴趣设备应答，实现局域网设备的发现，当然服务也一样。

mDns协议规定了消息的基本格式和消息的收发的基本顺序，DNS-SD 协议在这基础上，首先对实例名，服务名称，域名长度/顺序等作出了具体的定义，然后规定了如何方便地进行服务发现和描述。

服务实例名称 = <服务实例>.<服务类型>.<域名>

服务实例一般由一个或多个标签组成，标签之间用 . 隔开。

服务类型表明该服务是使用什么协议实现的，由 _ 下划线和服务使用的协议名称组成，如大部分使用的 *tcp 协议，另外，可以同时使用多个协议标签，如:* “http._tcp” 就表明该服务类型使用了基于tcp的http协议。

域名一般都固定为 “local”

DNS-SD 协议使用了PTR、SRV、TXT 3种类型的资源记录来完整地描述了一个服务。当主机通过查询得到了一个PTR响应记录后，就获得了一个它所关心服务的实例名称，它可以同通过继续获取 SRV 和 TXT 记录来拿到进一步的信息。其中的 SRV 记录中有该服务对应的主机名和端口号。TXT 记录中有该服务的其他附加信息。

服务器

```
func main() {
    flag.Parse()

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

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

func addRegistryPlugin(s *server.Server) {

    r := serverplugin.NewMDNSRegisterPlugin("tcp@"+*addr, 8972, metrics.NewRegistry(), time.Minute, "")
    err := r.Start()
    if err != nil {
        log.Fatal(err)
    }
    s.Plugins.Add(r)
}
```

客户端

```
func main() {
    flag.Parse()

    d := client.NewMDNSDiscovery("Arith", 10*time.Second, 10*time.Second, "")
    xclient := client.NewXClient("Arith", 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(), "Mul", args, reply)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

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

}
```

Inprocess

Example: inprocess

这个Registry用于进程内的测试。 在开发过程中，可能不能直接连接线上的服务器直接测试，而是写一些mock程序作为服务，这个时候就可以使用这个registry, 测试通过在部署的时候再换成相应的其它registry.

在这种情况下， client和server并不会走TCP或者UDP协议，而是直接进程内方法调用,所以服务器代码是和client代码在一起的。

```
func main() {
    flag.Parse()

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

    s.RegisterName("Arith", new(example.Arith), "")

    go func() {
        s.Serve("tcp", *addr)
```

```
    }()

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

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

    for i := 0; i < 100; i++ {

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

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

    }
}

func addRegistryPlugin(s *server.Server) {

    r := client.InprocessClient
    s.Plugins.Add(r)
}
```

特性

编解码

编解码

Example: iterator-go

当前rpcx提供了四种内置的编解码器，你也可以定义你自己的编解码器， 如Avro等:

```
// SerializeType defines serialization type of payload.
type SerializeType byte

const (
    // SerializeNone uses raw []byte and don't serialize/deserialize
    SerializeNone SerializeType = iota
    // JSON for payload.
    JSON
    // ProtoBuffer for payload.
    ProtoBuffer
    // MsgPack for payload
    MsgPack
)
```

服务会使用和客户端一样的编解码器，客户端使用JSON, 服务也返回 JSON 格式的数据。 rpcx 默认使用 msgpack 编解码器。

```
var DefaultOption = Option{
    Retries:        3,
    RPCPath:        share.DefaultRPCPath,
    ConnectTimeout: 10 * time.Second,
    Breaker:        CircuitBreaker,
    SerializeType:  protocol.MsgPack,
    CompressType:   protocol.None,
}
```

你可以设置你的option, 选择你自己的编解码器:

```
func NewXClient(servicePath string, failMode FailMode, selectMode SelectMode, discovery ServiceDiscovery, opt
ion Option)
```

SerializeNone

这种编解码器不会对数据进行编解码，并且要求数据是 `[]byte` 类型的数据。

JSON

JSON是一个通用的数据交换的格式，可以应用在很多语言中。

对性能要求不是非常高的场景，可以使用这种编解码。

Protobuf

Example: protobuf

Protobuf 是一个高性能的编解码器， 由google出品， 应用在很多项目中。

MsgPack

默认的编解码器

messagepack 是另外一种高性能的编解码器， 也是跨语言的编解码器。

定制编解码器

这个例子 gob 使用 gob作为编解码器。

失败模式

失败模式

在分布式架构中， 如SOA或者微服务架构，你不能担保服务调用如你所预想的一样好。有时候服务会宕机、网络被挖断、网络变慢等，所以你需要容忍这些状况。

rpcx支持四种调用失败模式，用来处理服务调用失败后的处理逻辑， 你可以在创建 `XClient` 的时候设置它。

`FailMode` 的设置仅仅对同步调用有效(`XClient.Call`), 异步调用用，这个参数是无意义的。

Failfast

示例: failfast

在这种模式下， 一旦调用一个节点失败， rpcx立即会返回错误。 注意这个错误不是业务上的 `Error` , 业务上服务端返回的 `Error` 应该正常返回给客户端，这里的错误可能是网络错误或者服务异常。

Failover

示例: failover

在这种模式下, rpcx如果遇到错误，它会尝试调用另外一个节点， 直到服务节点能正常返回信息，或者达到最大的重试次数。重试测试 `Retries` 在参数 `Option` 中设置， 缺省设置为3。

Failtry

示例: failtry

在这种模式下， rpcx如果调用一个节点的服务出现错误， 它也会尝试，但是还是选择这个节点进行重试， 直到节点正常返回数据或者达到最大重试次数。

Failbackup

示例: failbackup

在这种模式下， 如果服务节点在一定的时间内不返回结果， rpcx客户端会发送相同的请求到另外一个节点， 只要这两个节点有一个返回， rpcx就算调用成功。

这个设定的时间配置在 `Option.BackupLatency` 参数中。

这种通过资源换取延迟的方式可以参看 Jeff Dean的文章 Achieving Rapid Response Times in Large Online Services

> From http://highscalability.com/blog/2012/6/18/google-on-latency-tolerant-systems-making-a-predictable-whol.html
> Backup requests are the idea of sending requests out to multiple replicas, but in a particular way. Here's the example for a read operation for a distributed file system client:
>
> send request to first replica
> wait 2 ms, and send to second replica
> servers cancel request on other replica when starting read

> A request could wait in a queue stuck behind an expensive query or a packet could be dropped, so if a reply is not returned quickly other replicas are tried. Responses come back faster if requests sit in multiple queues.

Fork模式

Fork

Example: fork

`Fork` is a method of `XClient` and you can use it to send a request to all servers that contains this service.

If any of servers returns response whithout an error, `Fork` will return for this XClient. If all servers return errors, `Fork` returns an error of those errors.

It is like thee `Failbackup` mode. `Failbackup` uses at most two requests but `Fork` uses more requests (same to count of servers).

```
func main() {
    ......

    xclient := client.NewXClient("Arith", client.Failover, client.RoundRobin, d, client.DefaultOption)
    defer xclient.Close()

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

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

        log.Printf("%d * %d = %d", args.A, args.B, reply.C)
        time.Sleep(1e9)
    }

}
```

广播模式

广播模式

示例: broadcast

`Broadcast` 是 `XClient` 的一个方法， 你可以将一个请求发送到这个服务的所有节点。 如果所有的节点都正常返回，没有错误的话， `Broadcast` 将返回其中的一个节点的返回结果。 如果有节点返回错误的话， `Broadcast` 将返回这些错误信息中的一个。

```
func main() {
    ......

    xclient := client.NewXClient("Arith", client.Failover, client.RoundRobin, d, client.DefaultOption)
    defer xclient.Close()

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

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

        log.Printf("%d * %d = %d", args.A, args.B, reply.C)
        time.Sleep(1e9)
    }

}
```

路由

路由

在大型的微服务系统中，我们会为同一个服务部署多个节点， 以便服务可以支持大并发的访问。它们可能部署在同一个数据中心的多个节点，或者多个数据中心中。

那么， 客户端该如何选择一个节点呢？ rpcx通过 `Selector` 来实现路由选择， 它就像一个负载均衡器，帮助你选择出一个合适的节点。

rpcx提供了多个路由策略算法，你可以在创建 `XClient` 来指定。

注意，这里的路由是针对 `ServicePath` 和 `ServiceMethod` 的路由。

随机

示例: random

从配置的节点中随机选择一个节点。

最简单，但是有时候单个节点的负载比较重。这是因为随机数只能保证在大量的请求下路由的比较均匀，并不能保证在很短的时间内负载是均匀的。

轮询

示例: roundrobin

使用轮询的方式，依次调用节点，能保证每个节点都均匀的被访问。在节点的服务能力都差不多的时候适用。

WeightedRoundRobin

示例: weighted

使用Nginx 平滑的基于权重的轮询算法。

比如如果三个节点 `a` 、 `b` 、 `c` 的权重是 `{ 5, 1, 1 }` , 这个算法的调用顺序是 `{ a, a, b, a, c, a, a }` ,
相比较 `{ c, b, a, a, a, a, a }` , 虽然权重都一样，但是前者更好，不至于在一段时间内将请求都发送给 `a` 。

网络质量优先

示例: ping

首先客户端会基于 `ping(ICMP)` 探测各个节点的网络质量，越短的ping时间，这个节点的权重也就越高。但是，我们也会保证网络较差的节点也有被调用的机会。

假定 `t` 是ping的返回时间， 如果超过1秒基本就没有调用机会了:

- weight=191 if t <= 10
- weight=201 -t if 10 < t <=200
- weight=1 if 200 < t < 1000

- weight=0 if t >= 1000

一致性哈希

示例: hash

使用 JumpConsistentHash 选择节点， 相同的servicePath, serviceMethod 和 参数会路由到同一个节点上。
JumpConsistentHash 是一个快速计算一致性哈希的算法，但是有一个缺陷是它不能删除节点，如果删除节点，路由就不准确了，所以在节点有变动的时候它会重新计算一致性哈希。

地理位置优先

示例: geo

如果我们希望的是客户端会优先选择离它最新的节点， 比如在同一个机房。
如果客户端在北京， 服务在上海和美国硅谷，那么我们优先选择上海的机房。

它要求服务在注册的时候要设置它所在的地理经纬度。

如果两个服务的节点的经纬度是一样的， rpcx会随机选择一个。

比必须使用下面的方法配置客户端的经纬度信息：

```
func (c *xClient) ConfigGeoSelector(latitude, longitude float64)
```

定制路由规则

示例: customized

如果上面内置的路由规则不满足你的需求，你可以参考上面的路由器自定义你自己的路由规则。

曾经有一个网友提到， 如果调用参数的某个字段的值是特殊的值的话，他们会把请求路由到一个指定的机房。这样的需求就要求你自己定义一个路由器，只需实现实现下面的接口：

```
type Selector interface {
    Select(ctx context.Context, servicePath, serviceMethod string, args interface{}) string
    UpdateServer(servers map[string]string)
}
```

- `Select` : defines the select algorithm.
- `UpdateServer` : clients init the nodes and update if nodes change.

超时

超时

示例: timeout

超时机制可以保护服务调用陷入无限的等待之中。超时定义了服务的最长等待时间，如果在给定的时间没有相应，服务调用就进入下一个状态，或者重试、或者立即返回错误。

Server

你可以使用 `OptionFn` 设置服务器的 `readTimeout` 和 `writeTimeout` 。

```
type Server struct {
    ......
    readTimeout  time.Duration
    writeTimeout time.Duration
    ......
}
```

设置超时的 `OptionFn` 是：

```
func WithReadTimeout(readTimeout time.Duration) OptionFn
func WithWriteTimeout(writeTimeout time.Duration) OptionFn
```

Client

客户端有两种方式可是超时。

一种是设置连接的 read/write deadline, 一种是使用 `context.Context` .

read/write deadline

Client的 `Option` 可以设置连接的超时值:

```
type Option struct {
    ......
    //ConnectTimeout sets timeout for dialing
    ConnectTimeout time.Duration
    // ReadTimeout sets readdeadline for underlying net.Conns
    ReadTimeout time.Duration
    // WriteTimeout sets writedeadline for underlying net.Conns
    WriteTimeout time.Duration
    ......
}
```

`DefaultOption` 设置连接超时值为 10 秒，但是没有设置 ReadTimeout 和 WriteTimeout。 如果没有设置，则不会有超时限制。

由于多个服务可能共用同一个节点，有可能出现多个服务调用互相影响的状况。

context.Context

`context.Context` 也可以用来控制超时。

你可以使用 `context.WithTimeout` 来设置超时时间，这是推荐的设置超时的方式。

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

元数据

元数据

示例: metadata

客户端和服务器端可以互相传递元数据。

元数据不是服务请求和服务响应的业务数据，而是一些辅助性的数据。

元数据是一个键值队的列表，键和值都是字符串， 类似 `http.Header` 。

Client

如果你想在客户端传给服务器元数据， 你 **必须** 在上下文中设置 `share.ReqMetaDataKey` 。

如果你想在客户端读取客户端的数据， 你 **必须** 在上下文中设置 `share.ResMetaDataKey` 。

```
// client.go
reply := &example.Reply{}
ctx := context.WithValue(context.Background(), share.ReqMetaDataKey, map[string]string{"aaa": "from client"})
ctx = context.WithValue(ctx, share.ResMetaDataKey, make(map[string]string))
err := xclient.Call(ctx, "Mul", args, reply)
```

Server

服务器可以从上下文读取 `share.ReqMetaDataKey` 和 `share.ResMetaDataKey` :

```
// server.go
reqMeta := ctx.Value(share.ReqMetaDataKey).(map[string]string)
resMeta := ctx.Value(share.ResMetaDataKey).(map[string]string)
```

心跳

心跳

示例: heartbeat

你可以设置自动的心跳来保持连接不断掉。
rpcx会自动处理心跳(事实上它直接就丢弃了心跳)。

客户端需要启用心跳选项，并且设置心跳间隔：

```
option := client.DefaultOption
option.Heartbeat = true
option.HeartbeatInterval = time.Second
```

分组

分组

分组: group

当你在服务器端注册服务的时候，你可能注意到第三个参数我们一般设置它为空的字符串，事实上你可以为服务增加一些元数据。

你可以通过UI管理器查看服务的元数据 rpcx-ui)，或者增删一些元数据。

`group` 就是一个元数据。如果你为服务设置了设置 `group` ， 只有在这个 `group` 的客户端才能访问这些服务(这个限制是在路由的时候限制的， 当然你在客户端绕过这个限制)。

```
// server.go

func main() {
    flag.Parse()

    go createServer1(*addr1, "")
    go createServer2(*addr2, "group=test")

    select {}
}

func createServer1(addr, meta string) {
    s := server.NewServer()
    s.RegisterName("Arith", new(example.Arith), meta)
    s.Serve("tcp", addr)
}

func createServer2(addr, meta string) {
    s := server.NewServer()
    s.RegisterName("Arith", new(Arith), meta)
    s.Serve("tcp", addr)
}
```

客户端通过 `option.Group` 设置组。

如果在客户端你没有设置 `option.Group` ，客户端可以访问这些服务， 无论服务是否设置了组还是没设置。

```
// client.go
option := client.DefaultOption
option.Group = "test"
xclient := client.NewXClient("Arith", client.Failover, client.RoundRobin, d, option)
defer xclient.Close()

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

for {
    reply := &example.Reply{}
    err := xclient.Call(context.Background(), "Mul", args, reply)
```

```
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

    log.Printf("%d * %d = %d", args.A, args.B, reply.C)
    time.Sleep(1e9)
}
```

服务状态

服务状态

示例: state

`state` 是另外一个元数据。 如果你在元数据中设置了 `state=inactive` ，客户端将不能访问这些服务，即使这些服务是"活"着的。

你可以使用临时禁用一些服务，而不是杀掉它们， 这样就实现了服务的降级。 server.

你可以通过 rpcx-ui)来时实现禁用和启用的功能。

```
// server.go
func main() {
    flag.Parse()

    go createServer1(*addr1, "")
    go createServer2(*addr2, "state=inactive")

    select {}
}

func createServer1(addr, meta string) {
    s := server.NewServer()
    s.RegisterName("Arith", new(example.Arith), meta)
    s.Serve("tcp", addr)
}

func createServer2(addr, meta string) {
    s := server.NewServer()
    s.RegisterName("Arith", new(Arith), meta)
    s.Serve("tcp", addr)
}
```

```
// client.go
xclient := client.NewXClient("Arith", client.Failover, client.RoundRobin, d, client.DefaultOption)
defer xclient.Close()

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

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

    log.Printf("%d * %d = %d", args.A, args.B, reply.C)
    time.Sleep(1e9)
}
```

断路器

断路器模式

在一个节点失败的情况下，断路器可以避免这个错误影响其他服务，以免出现雪崩的情况。查看断路器的详细介绍: Pattern: Circuit Breaker.

客户端通过断路器调用服务， 一旦连续的错误达到一个阈值，断路器就会断开进行保护，这个时候如果还调用这个节点的话，直接就返回错误。等一定的时间，断路器会处于半开的状态，允许一定数量的请求尝试发送这个节点，如果正常访问，断路器就处于全开的状态，否则又进入短路的状态。

Rpcx 定义了 `Breaker` 接口， 你可以自己实现复杂情况的断路器。

```
type Breaker interface {
    Call(func() error, time.Duration) error
}
```

Rpcx 提供了一个简单的断路器 `ConsecCircuitBreaker` ， 它在连续失败一定次数后就会断开，再经过一段时间后打开。你可以将你的断路器设置到 `Option.Breaker` 中。

插件

Metrics

Metrics 插件

示例: metrics

Metrics 插件使用流行的go-metrics 来计算服务的指标。

它包含多个统计指标:

1. serviceCounter
2. clientMeter
3. “service_”+servicePath+”.”+serviceMethod+”*Read*Qps”
4. “service_”+servicePath+”.”+serviceMethod+”*Write*Qps”
5. “service_”+servicePath+”.”+serviceMethod+”_CallTime”

你可以将metrics输出到graphite中，通过grafana来监控。

限流

限流

实例: rate-limiting

限流是一种保护错误，避免服务被突发的或者大量的请求所拖垮。

这个插件使用 juju/ratelimit来限流。

使用 `func NewRateLimitingPlugin(fillInterval time.Duration, capacity int64) *RateLimitingPlugin` t来创建这个插件。

别名

别名

示例: alias

这个插件可以为一个服务方法设置一个别名。

下面的代码使用 `Arith` 的别名 `a.b.c.d` ， 为 `Mul` 设置别名 `Times` 。

```
func main() {
    flag.Parse()

    a := serverplugin.NewAliasPlugin()
    a.Alias("a.b.c.d", "Times", "Arith", "Mul")
    s := server.NewServer()
    s.Plugins.Add(a)
    s.RegisterName("Arith", new(example.Arith), "")
    err := s.Serve("reuseport", *addr)
    if err != nil {
        panic(err)
    }
}
```

客户端可以使用别名来调用服务：

```
func main() {
    flag.Parse()

    d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")

    option := client.DefaultOption
    option.ReadTimeout = 10 * time.Second

    xclient := client.NewXClient("a.b.c.d", client.Failtry, client.RandomSelect, d, option)
    defer xclient.Close()

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

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

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

}
```

身份认证

身份认证

示例: auth

出于安全的考虑， 很多场景下只有授权的客户端才可以调用服务。

客户端必须设置一个 `token` , 这个 `token` 可以从其他的 OAuth/OAuth2 服务器获得，或者由服务提供者分配。

服务接收到请求后，需要验证这个 `token` 。如果是 `token` 是从 OAuth2服务器中申请到，则服务需要到OAuth2服务中去验证， 如果是自己分配的，则需要和自己的记录进行对别。

因为rpcx提供的是一个身份验证的框架，所以具体的身份验证需要自己集成和验证。

```
func main() {
    flag.Parse()

    s := server.NewServer()
    s.RegisterName("Arith", new(example.Arith), "")
    s.AuthFunc = auth
    s.Serve("reuseport", *addr)
}

func auth(ctx context.Context, req *protocol.Message, token string) error {

    if token == "bearer tGzv3JOkF0XG5Qx2TlKWIA" {
        return nil
    }

    return errors.New("invalid token")
}
```

服务器必须定义 `AuthFunc` 来验证token。在上面的例子中， 只有token为 `bearer tGzv3JOkF0XG5Qx2TlKWIA` 才是合法的客户端。

客户端必须设置这个 `toekn` :

```
func main() {
    flag.Parse()

    d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")

    option := client.DefaultOption
    option.ReadTimeout = 10 * time.Second

    xclient := client.NewXClient("Arith", client.Failtry, client.RandomSelect, d, option)
    defer xclient.Close()

    //xclient.Auth("bearer tGzv3JOkF0XG5Qx2TlKWIA")
    xclient.Auth("bearer abcdefg1234567")

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

```
        B: 20,
    }

    reply := &example.Reply{}
    ctx := context.WithValue(context.Background(), share.ReqMetaDataKey, make(map[string]string))
    err := xclient.Call(ctx, "Mul", args, reply)
    if err != nil {
        log.Fatalf("failed to call: %v", err)
    }

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

}
```

注意: **你必须设置** `map[string]string` **为** `share.ReqMetaDataKey` **的值，否则调用会出错**

插件开发

插件开发

rpcx为服务器和客户端定义了几个插件接口，在一些处理点上可以调用插件。

Server Plugin

示例: trace

go doc

```
type PostConnAcceptPlugin

type PostConnAcceptPlugin interface {
    HandleConnAccept(net.Conn) (net.Conn, bool)
}


type PostReadRequestPlugin

type PostReadRequestPlugin interface {
    PostReadRequest(ctx context.Context, r *protocol.Message, e error) error
}

PostReadRequestPlugin represents .
type PostWriteResponsePlugin

type PostWriteResponsePlugin interface {
    PostWriteResponse(context.Context, *protocol.Message, *protocol.Message, error) error
}

PostWriteResponsePlugin represents .
type PreReadRequestPlugin

type PreReadRequestPlugin interface {
    PreReadRequest(ctx context.Context) error
}

PreReadRequestPlugin represents .
type PreWriteResponsePlugin

type PreWriteResponsePlugin interface {
    PreWriteResponse(context.Context, *protocol.Message) error
}

PreWriteResponsePlugin represents .
```

Client Plugin

go doc

```
type PluginContainer

type PluginContainer interface {
    Add(plugin Plugin)
    Remove(plugin Plugin)
    All() []Plugin

    DoPreCall(ctx context.Context, servicePath, serviceMethod string, args interface{}) error
    DoPostCall(ctx context.Context, servicePath, serviceMethod string, args interface{}, reply interface{}, e
rr error) error
}

type PostCallPlugin

type PostCallPlugin interface {
    DoPostCall(ctx context.Context, servicePath, serviceMethod string, args interface{}, reply interface{}, e
rr error) error
}

PostCallPlugin is invoked after the client calls a server.
type PreCallPlugin

type PreCallPlugin interface {
    DoPreCall(ctx context.Context, servicePath, serviceMethod string, args interface{}) error
}

PreCallPlugin is invoked before the client calls a server.
```

其它

Benchmark

Benchmark

Benchmark 的测试代码可以在这里找到：rpcx-ecosystem/rpcx-benchmark。

测试使用相同的测试环境，相同的测试数据，相同的测试参数，分别测试了 grpc, rpcx, dubbo, motan, thrift 和 go-micro rpc框架。

基于以前的测试结果, dubbo, motan 和 go-micro 的测试结果不乐观，所以它们最新的测试并没有列在这里，你可以运行测试代码来测试它们。

> 注意: 这个测试是基于io敏感的场景进行测试的， 也就是会让服务 `sleep` n秒钟， 对于cpu敏感的测试场景并没有实现。
> 测试假定网络条件很好， 如果是跨数据中心的服务调用，尤其是中美之间这种跨洋调用，服务的主要耗时并不在服务实现上，而是耗在了网络传输上,
> 这不是本测试要测试的场景,并且这个场景已经不太好区分出rpc框架的性能了。

测试逻辑

使用 `protobuf` 作为编解码器。proto文件位于 benchmark.proto:

```
syntax = "proto2";

package main;

option optimize_for = SPEED;


message BenchmarkMessage {
  required string field1 = 1;
  optional string field9 = 9;
  optional string field18 = 18;
  optional bool field80 = 80 [default=false];
  optional bool field81 = 81 [default=true];
  required int32 field2 = 2;
  required int32 field3 = 3;

  ......
}
```

Client 会创建一个 request ,并且对相应的字段进行赋值，最终的request序列化后的大小为 **518** 字节。

Server 接收这个请求，并且将第一个字段的值设置为 `OK` ,第二个字段设置为 `100` ,然后把这个对象返回给Client。

测试工具中下面两个参数用来控制并发数和总的请求数。

```
var concurrency = flag.Int("c", 1, "concurrency")
var total = flag.Int("n", 1, "total requests for all clients")
```

测试环境

- CPU: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz, 32 cores
- Memory: 32G
- Go: 1.9.2
- OS: CentOS 7 / 3.10.0-229.el7.x86_64

Client 和 Server 安装在同一台机器上 (忽略网络传输的影响)

测试结果

TPS

5000 并发client的情况下, rpcx 可以达到 176,894 transations/second 吞吐率, 但是 grpc-go 只能达到 105,219 transations/second 的吞吐率

concurrency	RPCX	GRPC-GO
5000	176894	105219
2000	161660	108245
1000	148227	111351
100	145479	93447

延迟: 平均时间

concurrency	RPCX	GRPC-GO
5000	27	47
2000	12	18
1000	6	8
100	0	1

延迟: 中位数时间

concurrency	RPCX	GRPC-GO
5000	3	42
2000	7	15
1000	5	7

concurrency	RPCX	GRPC-GO
100	0	0

UI管理工具

UI管理工具

rpcx提供了一个简单的UI管理程序，可以查看和搜索当前注册的服务以及服务的状态， 同时你也可以临时禁用服务，分组或者更改服务的元数据。

rpcx-ui

协议详解

协议

rpcx 的 请求(request)和响应(response)使用相同的数据结构。

一个消息由下面的项组成:

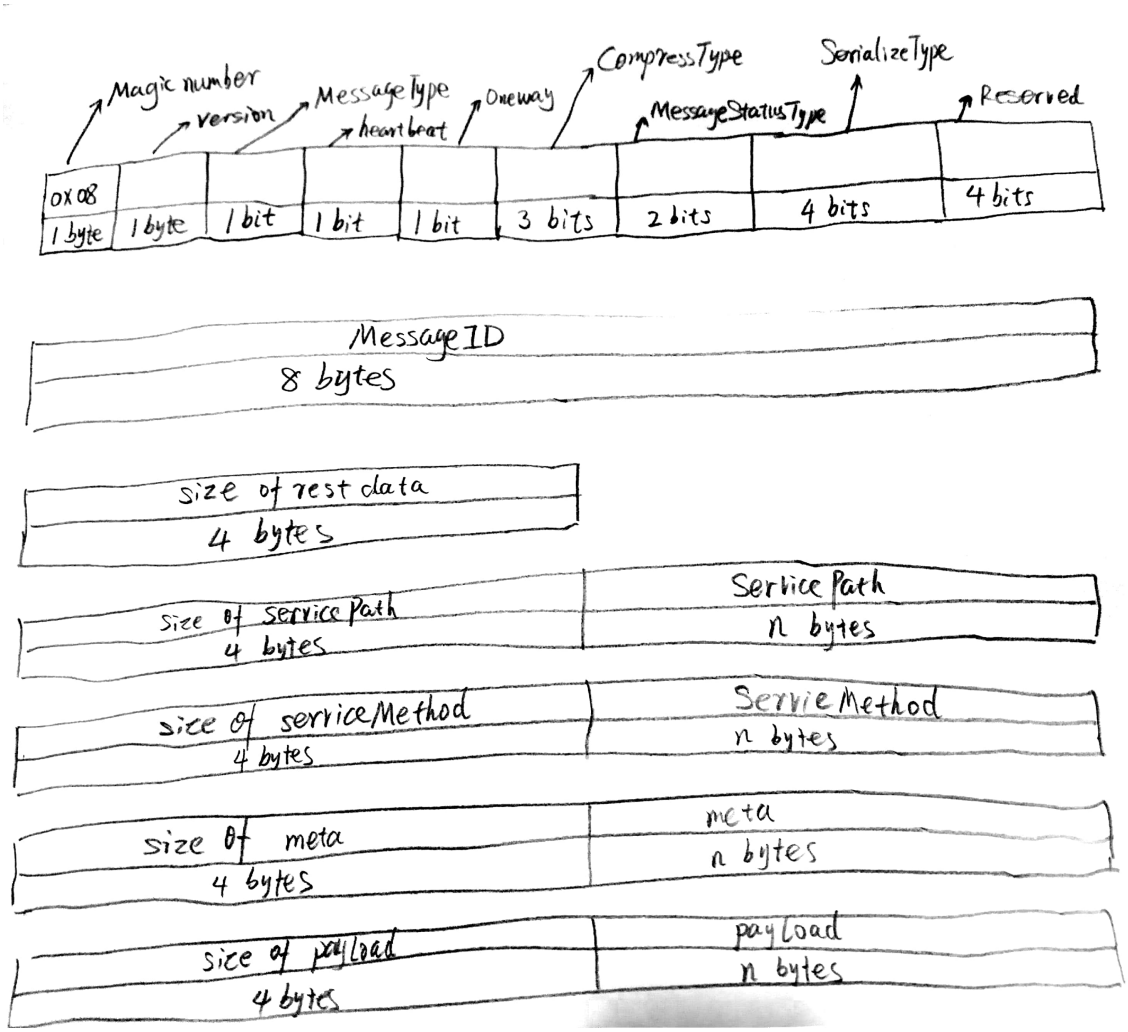
1. Header: 4 字节
2. Message ID: 8 字节
3. total size: 4 字节, 不包含header和它本身, uint32类型
4. servicePath值的长度: 4 字节, uint32类型
5. servicePath的值: UTF-8 字符串
6. serviceMethod值的长度: 4 字节, uint32类型
7. serviceMethod的值: UTF-8 字符串
8. metadata的大小: 4 字节, uint32类型
9. metadata: 格式: `size` `key1 string` `size` `value1 string`, 可以包含多个
10. playload的大小: 4 字节, uint32类型
11. playload的值: slice of byte

`#4` + `#6` + `#8` + `#10 + (4 + 4 + 4 + 4)` 的字节数加起来等于 `#3` 的值。

`servicePath` 、 `serviceMethod` 、 和 meta `中的key` 、 `value` 都是UTF-8 字符串。

rpcx 使用 `size of an element` + `element` 的格式定义可变长的元素， 就像 TLV， 但是 rpcx 不需要 `Type` 字段， 这是因为 元素的 `Type` 要么是 UTF-8 字符串，要么就是明确的slice。

对整数使用大端 `BigEndian` 编码 (integer type, int64, uint32 等等)



1、T第一个字节是 `0x08` , 它是一个魔数 (magic number)

2、第二个字节是 `version` . 当前的版本是 0.

3、MessageType 可以是:

- 0: Request
- 1: Response

4、Heartbeat: bool. 指示这个消息是否是heartbeat消息

5、Oneway: bool. true的话意味着服务不需要返回response

6、CompressType: 压缩类型

- 0: don’t compress
- 1: Gzip

7、MessageStatusType: 指示 response 是一个错误还是正常的返回值

- 0: Normal
- 1: Error

8、SerializeType: 编解码格式

- 0: 使用原始的byte slice
- 1: JSON
- 2: Protobuf
- 3: MessagePack

如果服务处理请求失败，它会返回error response， 它会设置 response 的 MessageStatusType为 `1` , 并且在meta中设置错误信息， meta中的 key值是 **rpcx_error**， 值是错误信息。

网关

Gateway

网关

Gateway 为 rpcx services 提供了 http 网关服务.

你可以使用你熟悉的编程语言， 比如Java、Python、C#、Node.js、Php、C\C++、Rust等等来调用 rpcx 服务。查看一些编程语言实现的例子 。

这意味着，你可以不用实现 rpcx的协议，而是使用熟悉的 http 访问方式调用 rpcx 服务， 设置用 `curl` 、 `wget` 等命令行工具。

部署模型

使用网关程序有两种部署模型: **Gateway** 和 **Agent**。

Gateway

你可以部署为网关模式。网关程序运行在独立的机器上，所有的client都将http请求发送给gateway, gateway负责将请求转换成rpcx的请求，并调用相应的rpcx 服务， 它将rpcx 的返回结果转换成http的response, 返回给client。

你可以部署多台gateway程序， 并可以利用nginx等进行负载均衡。

Agent

你可以将网关程序和1你的client一起部署， agent作为一个后台服务部署在 client机器上。 如果你的机器有多个client, 你只需部署一个agent。

Client发送 http 请求到本地的agent, 本地的agent将请求转为 rpcx请求，然后转发到相应的 rpcx服务上， 然后将 rpcx的response转换为 http response返回给 client。

它类似 mesh service的 Sidecar模式， 但是比较好的是， 同一台机器上的client只需一个agent。

http协议

你可以使用任意的编程语言来发送http请求，但是需要额外设置一些header (但是不一定设置全部的header, 按需设置):

- X-RPCX-Version: rpcx 版本
- X-RPCX-MesssageType: 设置为0,代表请求
- X-RPCX-Heartbeat: 是否是heartbeat请求, 缺省false
- X-RPCX-Oneway: 是否是单向请求, 缺省false.
- X-RPCX-SerializeType: 0 as raw bytes, 1 as JSON, 2 as protobuf, 3 as msgpack
- X-RPCX-MessageID: 消息id, uint64 类型
- X-RPCX-ServicePath: service path
- X-RPCX-ServiceMethod: service method
- X-RPCX-Meta: 额外的元数据

对于 http response, 可能会包含一些header:

- X-RPCX-Version: rpcx 版本
- X-RPCX-MesssageType: 1 ,代表response
- X-RPCX-Heartbeat: 是否是heartbeat请求
- X-RPCX-MessageStatusType: Error 还是正常返回结果
- X-RPCX-SerializeType: 0 as raw bytes, 1 as JSON, 2 as protobuf, 3 as msgpack
- X-RPCX-MessageID: 消息id, uint64 类型
- X-RPCX-ServicePath: service path
- X-RPCX-ServiceMethod: service method
- X-RPCX-Meta: extra metadata
- X-RPCX-ErrorMessage: 错误信息, 如果 X-RPCX-MessageStatusType 是 Error 的话

例子

下面是Go的一个例子:

```
package main

import (
    "bytes"
    "io/ioutil"
    "log"
    "net/http"

    "github.com/rpcx-ecosystem/rpcx-gateway"

    "github.com/smallnest/rpcx/codec"
)

type Args struct {
    A int
    B int
}

type Reply struct {
    C int
}

func main() {
    cc := &codec.MsgpackCodec{}

    // request
    args := &Args{
        A: 10,
        B: 20,
    }
    data, _ := cc.Encode(args)

    req, err := http.NewRequest("POST", "http://127.0.0.1:9981/", bytes.NewReader(data))
    if err != nil {
        log.Fatal("failed to create request: ", err)
        return
    }

    // set extra headers
    h := req.Header
    h.Set(gateway.XMessageID, "10000")
```

```
    h.Set(gateway.XMessageType, "0")
    h.Set(gateway.XSerializeType, "3")
    h.Set(gateway.XServicePath, "Arith")
    h.Set(gateway.XServiceMethod, "Mul")

    // send to gateway
    res, err := http.DefaultClient.Do(req)
    if err != nil {
        log.Fatal("failed to call: ", err)
    }
    defer res.Body.Close()

    // handle http response
    replyData, err := ioutil.ReadAll(res.Body)
    if err != nil {
        log.Fatal("failed to read response: ", err)
    }

    // parse reply
    reply := &Reply{}
    err = cc.Decode(replyData, reply)
    if err != nil {
        log.Fatal("failed to decode reply: ", err)
    }

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

不能进行双向通讯， 也就是服务端不能主动发送请求给客户端。

HTTP 方式调用

HTTP 调用

大部分场景下， rpcx服务是通过 TCP 进行通讯的， 但是你也可以直接通过 http 进行访问， http请求需要设置一些 header, 这和 gateway 中的 header 是一样的。

很明显，通过http调用不可能取得和 TCP 一样的性能， 因为 http 调用是 一问一答 方式进行通讯的， 并不能并发的请求(除非你使用很多client)， 但是调用方式简单， 也可以应用在一些场景中。

可以http调用的服务必须使用 TCP 方式部署，而不能使用 UDP或者其他方式， 它和TCP共用同一个接口。 一个连接只能使用http调用或者TCP调用。

你可以使用你熟悉的语言进行调用，设置你使用的编解码器，最常用的是JSON编解码，注意一定要添加相应的header,否则rpcx服务不知道该如果处理这个htp请求。

下面是一个http调用的例子:

```
func main() {
    cc := &codec.MsgpackCodec{}

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

    data, _ := cc.Encode(args)

    req, err := http.NewRequest("POST", "http://127.0.0.1:8972/", bytes.NewReader(data))
    if err != nil {
        log.Fatal("failed to create request: ", err)
        return
    }

    h := req.Header
    h.Set(gateway.XMessageID, "10000")
    h.Set(gateway.XMessageType, "0")
    h.Set(gateway.XSerializeType, "3")
    h.Set(gateway.XServicePath, "Arith")
    h.Set(gateway.XServiceMethod, "Mul")

    res, err := http.DefaultClient.Do(req)
    if err != nil {
        log.Fatal("failed to call: ", err)
    }
    defer res.Body.Close()

    // handle http response
    replyData, err := ioutil.ReadAll(res.Body)
    if err != nil {
        log.Fatal("failed to read response: ", err)
    }

    reply := &Reply{}
    err = cc.Decode(replyData, reply)
    if err != nil {
```

```
        log.Fatal("failed to decode reply: ", err)
    }

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

不能进行双向通讯， 也就是服务端不能主动发送请求给客户端。

双向通讯

双向通讯

示例: bidirectional

在正常情况下， 客户端发送请求，服务器返回结果，这样一问一答的方式就是 `request-response` rpc 模型。

但是对于一些用户， 比如 `IoT` 的开发者， 可能需要在某些时候发送通知给客户端。 如果客户端和服务端都配两套代码就显得多余和臃肿了。

rpcx实现了一个简单的通知机制。

首先你需要缓存客户端的连接，可能还需要将用户的ID和连接进行关联， 以便服务器知道将通知发送给哪个客户端。

Server

服务器使用 `SendMessage` 方法发送通知， 数据是 `[]byte` 类型。 你可以设置 `servicePath` 和 `serviceMethod` 以便提供给客户端更多的信息，用来区分不同的通知。

`net.Conn` 对象可以在客户端调用服务的时候从 `ctx.Value(server.RemoteConnContextKey)` 中获取。

```
func (s *Server) SendMessage(conn net.Conn, servicePath, serviceMethod string, metadata map[string]string, data []byte) error
```

```
// server.go
func main() {
    flag.Parse()

    ln, _ := net.Listen("tcp", ":9981")
    go http.Serve(ln, nil)

    s := server.NewServer()
    //s.RegisterName("Arith", new(example.Arith), "")
    s.Register(new(Arith), "")
    go s.Serve("tcp", *addr)

    for !connected {
        time.Sleep(time.Second)
    }

    fmt.Printf("start to send messages to %s\n", clientConn.RemoteAddr().String())
    for {
        if clientConn != nil {
            err := s.SendMessage(clientConn, "test_service_path", "test_service_method", nil, []byte("abcde"))
            if err != nil {
                fmt.Printf("failed to send messsage to %s: %v\n", clientConn.RemoteAddr().String(), err)
                clientConn = nil
            }
        }
        time.Sleep(time.Second)
    }
}
```

Client

你必须使用 `NewBidirectionalXClient` 创建 XClient 客户端， 你需要传如一个channel， 这样你就可以从channel中读取通知了。

```
// client.go
func main() {
    flag.Parse()

    ch := make(chan *protocol.Message)

    d := client.NewPeer2PeerDiscovery("tcp@"+*addr, "")
    xclient := client.NewBidirectionalXClient("Arith", client.Failtry, client.RandomSelect, d, client.Default
Option, ch)
    defer xclient.Close()

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

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

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

    for msg := range ch {
        fmt.Printf("receive msg from server: %s\n", msg.Payload)
    }
}
```