**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](https://golang.org/pkg/go/build/#hdr-Build_Constraints))

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。