https://skoo.me/go/2013/09/01/go-rpcserver-inside

Go RPC Inside (server)

Go RPC Inside (server)

01 September 2013

by skoo

说到 rpc 让我想起了刚毕业面试的时候，被问到是否了解 rpc？我记得当时我的回答是“课本上学过 rpc，只知道是远程过程调用，但没有用过，具体也不知道是什么”。的确，大学中间件这门课程里有讲到 rpc，里面还引入了一个非常难理解的概念——“桩”，英文应该叫”stub”。现在的 rpc 实现里，stub 这个概念好像都没见到了，应该都是叫”method”。

实现一个 rpc 服务器很难吗？rpc 服务器也就是在 tcp 服务器的基础上加上自定义的 rpc 协议而已。一个 rpc 协议里，主要有个 3 个非常重要的信息。

- 调用的远程 method 名字，一般就是一个函数名
- call 参数，也就是发送给服务器的数据
- 客户端生成的调用请求 seq

除了最后一点，其他两点显然就是组成一个普通的函数调用而已，这也就是远程过程调用了。最后一点的 seq 只是 rpc 内部实现的一个关键点而已，也许还有其他的实现方式，而不是依赖 seq 来保证单连接的并发请求。

####如何用 C 语言实现 rpc 服务器？ 用 C 语言来实现 rpc 服务器，先假设我们从 socket 里读取到了一个来自于客户端的 call 请求，这个 call 请求里面封装了上面提到的 3 点信息。

执行这个 call，最重要的就是——**“从 call 里取出 method，也就是一个函数名(字符串)，然后要通过这个函数名去执行对应的函数”。**

C 语言由于没有反射机制，于是不能通过函数的名字去调对应的函数；因此，可以使用一张 hash 表保存所有远程函数的名字到函数的对应关系。这样就可以通过 method 查找一次 hash 表得到真正的函数，接下来就可以执行函数了，函数的执行结果当然就是作为响应返回给客户端。

当然，这些需要被调用执行的函数都是在服务器初始化的时候事先注册到这张 hash 表中的。到这里，感觉一切都结束了。其实，还有**参数**(请求数据)和**返回值**(应答数据)，这部分主要是涉及到序列化算法，留到下次的*“反射和序列化”*再聊了。

####使用 Go RPC 框架写一个简单服务器

定义服务

```
type EchoServer bool
```

```
func (s *EchoServer) Echo(req *string, res *string) error {
            res = req
            return nil
}
```

注册服务

```
echo := new(EchoServer)
if err := rpc.Register(echo); err != nil {
            return err
}

// 下面基本是固定代码，创建一个 tcp 服务器
var listener *net.TCPListener
if tcpAddr, err := net.ResolveTCPAddr("tcp", addr); err != nil {
            return err
} else {
            if listener, err = net.ListenTCP("tcp", tcpAddr); err != nil {
                        return err
            }
}

for {
            conn, err := listener.Accept()
            if err != nil {
                        sc.LOG.Error(err)
                        continue
            }
            // 服务器走 rpc 框架的入口，也就是在一个 tcp 连接上采用 rpc 协议来处理请求和应答。
            go rpc.ServeConn(conn)
}
return nil
```

这样，就实现了一个简单的 echo 服务器，功能逻辑都在自定义的 EchoServer 里面实现。接下来只需要将 EchoServer 的一个实例对象注册到 rpc 框架中，客户端就可以直接调用 EchoServer 对象中的方法了。

Go rpc 服务器的内部实现在思路上和前文提到的 C 语言实现基本是一样的。细心的人可能注意到了，这里注册的是 EchoServer 对象，而不是 EchoSever 对象的方法，然后前文的 C 语言实现中注册的直接是函数，那么 rpc 服务器如何能够根据 `method` 去调用对应的方法呢？Go 语言在这里其实采用反射的手段，虽然表面上是注册的 EchoServer 对象，实际却是通过反射取得了 EchoServer 的所有方法，然后采用了 map 表保存了 method 到方法的映射，这就回到了前文 C 语言的实现思路中去了。如果没有反射的支持，就只能一个一个的方法全部注册一遍了，并且代码组织上也不够优雅。

####编码解码器 rpc 客户端有一个编码解码器定义了如何发送请求和读取应答，那么服务器端必然有一个编码解码器定义了如何读取请求和发送应答，刚好是一个相反的过程，这也就是序列化和反序列化的一个用处。

```
type ServerCodec interface {
            ReadRequestHeader(*Request) error
            ReadRequestBody(interface{}) error
            WriteResponse(*Response, interface{}) error

            Close() error
} 和客户端一样，你只需要实现 ServerCodec 这个接口，就可以自定义服务端的编码解码器，实现自定义的 rpc 协议了。 Go rpc 服务器端和客户端都是默认使用的 Gob 序列化协议数据。Go 里面为什么采用自实现的 Gob，而不是 Google 的 protobuf，这个也留到下次的*"反射和序列化"*聊吧。
```

注意，ServerCodec 接口中的 `ReadRequestBody(interface{}) error` 方法主要是用来读取客户端 call 请求的参数数据，也就是将 socket 中读出来的数据包解码到 `interface{}`所指向的具体对象中去，这里的 `interface{}`可以理解为 C 语言中的 `void *`。你一定注意到了，不同的 rpc 服务定义的参数类型完全不同，并且在 rpc 框架内部都采用了 `interface{}`来适配，那么框架内部如何知道读取的 socket 数据要解码到什么具体类型中去呢？这里又是涉及到了反射，因为有了反射就可以从 `interface{}`得到具体的类型。

接下来真得好好的说说*"反射和序列化/反序列化"*了。

####几个内部细节 * 服务器在发送应答的时候，同样采用了一把锁来保证原子性写入 socket * 请求/应答结构(ServerCodec 接口中出现的 `Request/Response`)采用链表实现一个 pool，需要一个 Requet/Response 的时候都从 free list 中获取，不再频繁的分配。这一点只是一个优化。 * Go 的 rpc 除了直接跑在 tcp 服务器上，还可以跑在更高层一点的 http 服务器上。 * Go 的 rpc 调用也提供了 json 序列化。

准备改变一下以前写技术文章老是大段大段代码的风格，试试小代码能不能将自己的想法交代清楚，哈哈。这一篇就先到此为止了。

下一篇就看一下 Go RPC 框架的性能，做一个 benchmark 比较下。

- Go 10
- Go 17
- language 2
- communication 2
- rpc 3