https://github.com/astaxie/build-web-application-with-golang/blob/master/zh/08.4.md

8.4 RPC

前面几个小节我们介绍了如何基于 Socket 和 HTTP 来编写网络应用，通过学习我们了解了 Socket 和 HTTP 采用的是类似"信息交换"模式，即客户端发送一条信息到服务端，然后(一般来说)服务器端都会返回一定的信息以表示响应。客户端和服务端之间约定了交互信息的格式，以便双方都能够解析交互所产生的信息。但是很多独立的应用并没有采用这种模式，而是采用类似常规的函数调用的方式来完成想要的功能。

RPC 就是想实现函数调用模式的网络化。客户端就像调用本地函数一样，然后客户端把这些参数打包之后通过网络传递到服务端，服务端解包到处理过程中执行，然后执行的结果反馈给客户端。

RPC（Remote Procedure Call Protocol）——远程过程调用协议，是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。它假定某些传输协议的存在，如 TCP 或 UDP，以便为通信程序之间携带信息数据。通过它可以使函数调用模式网络化。在 OSI 网络通信模型中，RPC 跨越了传输层和应用层。RPC 使得开发包括网络分布式多程序在内的应用程序更加容易。

RPC 工作原理

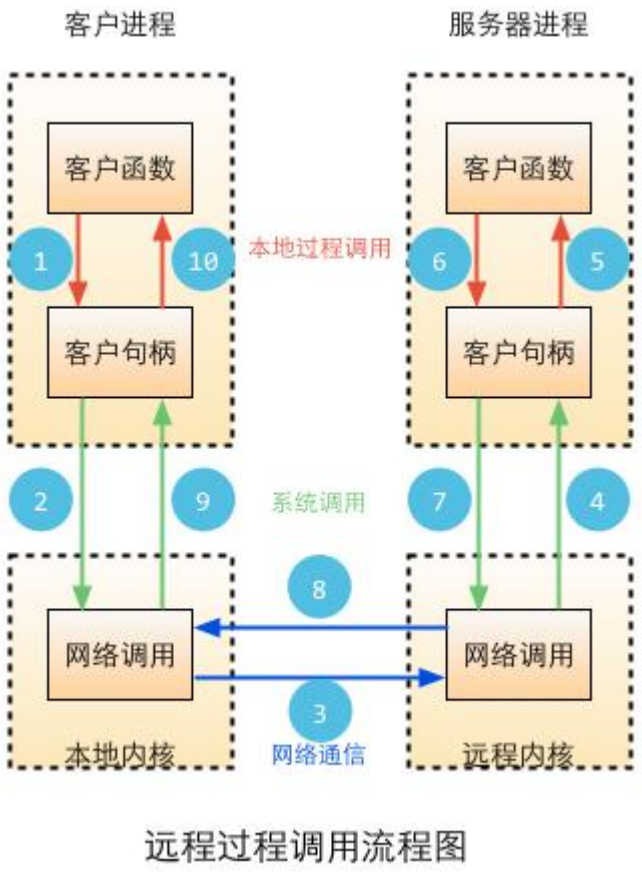

图 8.8 RPC 工作流程图

运行时,一次客户机对服务器的 RPC 调用,其内部操作大致有如下十步：

- 1.调用客户端句柄；执行传送参数
- 2.调用本地系统内核发送网络消息
- 3.消息传送到远程主机
- 4.服务器句柄得到消息并取得参数
- 5.执行远程过程
- 6.执行的过程将结果返回服务器句柄
- 7.服务器句柄返回结果，调用远程系统内核
- 8.消息传回本地主机
- 9.客户句柄由内核接收消息
- 10.客户接收句柄返回的数据

Go RPC

Go 标准包中已经提供了对 RPC 的支持，而且支持三个级别的 RPC：TCP、HTTP、JSONRPC。但 Go 的 RPC 包是独一无二的 RPC，它和传统的 RPC 系统不同，它只支持 Go 开发的服务器与客户端之间的交互，因为在内部，它们采用了 Gob 来编码。

Go RPC 的函数只有符合下面的条件才能被远程访问，不然会被忽略，详细的要求如下：

- 函数必须是导出的(首字母大写)
- 必须有两个导出类型的参数，
- 第一个参数是接收的参数，第二个参数是返回给客户端的参数，第二个参数必须是指针类型的
- 函数还要有一个返回值 error

举个例子，正确的 RPC 函数格式如下：

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

T、T1 和 T2 类型必须能被 `encoding/gob` 包编解码。

任何的 RPC 都需要通过网络来传递数据，Go RPC 可以利用 HTTP 和 TCP 来传递数据，利用 HTTP 的好处是可以直接复用 `net/http` 里面的一些函数。详细的例子请看下面的实现

HTTP RPC

http 的服务端代码实现如下：

```
package main

import (

            "errors"

            "fmt"
```

```
            "net/http"
            "net/rpc"
)
type Args struct {
            A, B int
}
type Quotient struct {
            Quo, Rem int
}
type Arith int
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
}
func main() {

            arith := new(Arith)
            rpc.Register(arith)
            rpc.HandleHTTP()

            err := http.ListenAndServe(":1234", nil)
            if err != nil {
```

```
                        fmt.Println(err.Error())

            }

}
```

通过上面的例子可以看到，我们注册了一个 Arith 的 RPC 服务，然后通过 `rpc.HandleHTTP` 函数把该服务注册到了 HTTP 协议上，然后我们就可以利用 http 的方式来传递数据了。

请看下面的客户端代码：

```
package main

import (
            "fmt"
            "log"
            "net/rpc"
            "os"
)
type Args struct {
            A, B int
}
type Quotient struct {
            Quo, Rem int
}
func main() {
            if len(os.Args) != 2 {
                        fmt.Println("Usage: ", os.Args[0], "server")
                        os.Exit(1)
            }
            serverAddress := os.Args[1]

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

```
            // Synchronous call
            args := Args{17, 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\n", args.A, args.B, reply)

            var quot Quotient
            err = client.Call("Arith.Divide", args, &quot)
            if err != nil {
                        log.Fatal("arith error:", err)
            }
            fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quo, quot.Rem)

}
```

我们把上面的服务端和客户端的代码分别编译，然后先把服务端开启，然后开启客户端，输入代码，就会输出如下信息：

```
$ ./http_c localhost
Arith: 17*8=136
Arith: 17/8=2 remainder 1
```

通过上面的调用可以看到参数和返回值是我们定义的 struct 类型，在服务端我们把它们当做调用函数的参数的类型，在客户端作为 `client.Call` 的第 2，3 两个参数的类型。客户端最重要的就是这个 Call 函数，它有 3 个参数，第 1 个要调用的函数的名字，第 2 个是要传递的参数，第 3 个要返回的参数(注意是指针类型)，通过上面的代码例子我们可以发现，使用 Go 的 RPC 实现相当的简单，方便。

TCP RPC

上面我们实现了基于 HTTP 协议的 RPC，接下来我们要实现基于 TCP 协议的 RPC，服务端的实现代码如下所示：

```
package main
import (
            "errors"
```

```
            "fmt"
            "net"
            "net/rpc"
            "os"
)
type Args struct {
            A, B int
}
type Quotient struct {
            Quo, Rem int
}
type Arith int
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
}
func main() {

            arith := new(Arith)
            rpc.Register(arith)

            tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
```

```
		checkError(err)



		listener, err := net.ListenTCP("tcp", tcpAddr)

		checkError(err)



		for {

			conn, err := listener.Accept()

			if err != nil {

				continue

			}

			rpc.ServeConn(conn)

		}



}

func checkError(err error) {

		if err != nil {

			fmt.Println("Fatal error ", err.Error())

			os.Exit(1)

		}

}
```

上面这个代码和 http 的服务器相比，不同在于:在此处我们采用了 TCP 协议，然后需要自己控制连接，当有客户端连接上来后，我们需要把这个连接交给 rpc 来处理。

如果你留心了，你会发现这它是一个阻塞型的单用户的程序，如果想要实现多并发，那么可以使用 goroutine 来实现，我们前面在 socket 小节的时候已经介绍过如何处理 goroutine。 下面展现了 TCP 实现的 RPC 客户端：

```
package main

import (

		"fmt"

		"log"

		"net/rpc"

		"os"
```

```
)

type Args struct {
	A, B int
}
type Quotient struct {
	Quo, Rem int
}
func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "server:port")
		os.Exit(1)
	}
	service := os.Args[1]

	client, err := rpc.Dial("tcp", service)
	if err != nil {
		log.Fatal("dialing:", err)
	}
	// Synchronous call
	args := Args{17, 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\n", args.A, args.B, reply)

	var quot Quotient
	err = client.Call("Arith.Divide", args, &quot)
	if err != nil {
```

```
                        log.Fatal("arith error:", err)

            }

            fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quo, quot.Rem)



}
```

这个客户端代码和 http 的客户端代码对比，唯一的区别一个是 DialHTTP，一个是 Dial(tcp)，其他处理一模一样。

JSON RPC

JSON RPC 是数据编码采用了 JSON，而不是 gob 编码，其他和上面介绍的 RPC 概念一模一样，下面我们来演示一下，如何使用 Go 提供的 json-rpc 标准包，请看服务端代码的实现：

```
package main

import (

            "errors"

            "fmt"

            "net"

            "net/rpc"

            "net/rpc/jsonrpc"

            "os"

)

type Args struct {

            A, B int

}

type Quotient struct {

            Quo, Rem int

}

type Arith int

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
}
func main() {

            arith := new(Arith)
            rpc.Register(arith)

            tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
            checkError(err)

            listener, err := net.ListenTCP("tcp", tcpAddr)
            checkError(err)

            for {
                        conn, err := listener.Accept()
                        if err != nil {
                                    continue
                        }
                        jsonrpc.ServeConn(conn)
            }

}
func checkError(err error) {
            if err != nil {
```

```
                        fmt.Println("Fatal error ", err.Error())

                        os.Exit(1)

                }

}
```

通过示例我们可以看出 json-rpc 是基于 TCP 协议实现的，目前它还不支持 HTTP 方式。

请看客户端的实现代码：

```
package main

import (

                "fmt"

                "log"

                "net/rpc/jsonrpc"

                "os"

)

type Args struct {

                A, B int

}

type Quotient struct {

                Quo, Rem int

}

func main() {

                if len(os.Args) != 2 {

                        fmt.Println("Usage: ", os.Args[0], "server:port")

                        log.Fatal(1)

                }

                service := os.Args[1]



                client, err := jsonrpc.Dial("tcp", service)

                if err != nil {

                        log.Fatal("dialing:", err)

                }
```

```
        // Synchronous call
        args := Args{17, 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\n", args.A, args.B, reply)

        var quot Quotient
        err = client.Call("Arith.Divide", args, &quot)
        if err != nil {
                log.Fatal("arith error:", err)
        }
        fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quo, quot.Rem)

}
```

总结

Go 已经提供了对 RPC 的良好支持，通过上面 HTTP、TCP、JSON RPC 的实现,我们就可以很方便的开发很多分布式的 Web 应用，我想作为读者的你已经领会到这一点。但遗憾的是目前 Go 尚未提供对 SOAP RPC 的支持，欣慰的是现在已经有第三方的开源实现了。

links

- 目录
- 上一节: REST
- 下一节: 小结