https://skoo.me/go/2013/08/28/go-rpcclient-inside

skoo's notes

努力记录一些自己觉得有趣的东西...

Go RPC Inside (client)

Go RPC Inside (client)

28 August 2013

by skoo

Go 语言标准库能够自带一个 rpc 框架还是非常给力的，这可以很大程度的降低写后端网络通信服务的门槛，特别是在大规模的分布式系统中，rpc 基本是跨机器通信的标配。rpc 能够最大程度屏蔽网络细节，让开发者专注在服务功能的开发上面。下面介绍 Go 语言 rpc 框架的客户端内部实现.

Go rpc 客户端的逻辑很简单，大体上，就是将一个个的调用请求序列化后原子的发送给服务器，然后有一个专门的 gorutine 等待服务器应答，这个 goroutine 会将收到的每个应答分发给对应的请求，这就完成了一次 rpc 调用。

调用入口

```
func NewClient(conn io.ReadWriteCloser) *Client



func (client *Client) Call(serviceMethod string, args interface{}, reply interface{}) error
```

使用 rpc 客户端，首先要 NewClient 得到一个客户端对象，然后通过这个客户端对象的 Call 方法去执行远程服务端方法。这里需要注意，NewClient 函数不光是创建了一个客户端对象，同时还创建了一个 input goroutine. input goroutine 是阻塞在连接上，等待服务端的响应。这里 Call 方法的主要工作是将一个 rpc 请求发送到服务端，同时放入一个等待队列，等候服务器的响应到来。

发送请求

经过客户端 Call 方法提交的 rpc 的请求，最终将调用如下的 send 方法发送给服务器，这里重点分析一下 send 的实现。

```
func (client *Client) send(call *Call) {
            //
            // 入口处就来了一把锁，这把锁显然是为了保证一个请求能够被原子的写入 socket 中．毕竟一个
            // rpc 连接是会被多个 goroutine 并发写入的，因此这里需要保证发送请求的原子性。
            //
            client.sending.Lock()
            defer client.sending.Unlock()


            //
            // 这里又来了一把锁，这把锁的目标是客户端的 pending 等待队列，也就是将每个 rpc 请求放入等待队列
```

```
		// 的时候，需要对这个 pending 队列做并发写保护。
		//
		// Register this call.
		client.mutex.Lock()
		if client.shutdown || client.closing {
			call.Error = ErrShutdown
			client.mutex.Unlock()
			call.done()
			return
		}
		//
		// pending 队列其实是使用 map 实现的，这里可以看到每个 rpc 请求都会生存一个唯一递增的 seq，这个
		// seq 就是用来标记请求的，这个很像 tcp 包的 seq。
		seq := client.seq
		client.seq++
		client.pending[seq] = call
		client.mutex.Unlock()

		//
		// 下面就是将一个 rpc 请求的所有数据（请求方法名、seq、请求参数等)进行序列化打包，然后发送
		// 出去．这里主要采用的是 Go 标准库自带的 gob 算法进行请求的序列化。
		//
		// Encode and send the request.
		client.request.Seq = seq
		client.request.ServiceMethod = call.ServiceMethod
		err := client.codec.WriteRequest(&client.request, call.Args)
		if err != nil {
			client.mutex.Lock()
			call = client.pending[seq]
			delete(client.pending, seq)
			client.mutex.Unlock()
			if call != nil {
				call.Error = err
				call.done()
			}
		}
}
```

通过对发送过程的分析，可以看出一个 rpc 请求发出去主要有三个过程，第一个过程是将请求放入等待队列，第二个过程就是序列化，最后一个就是写入 socket.

send 过程最重要的就是两把锁 + seq。复用连接的原子写、并发请求是实现 rpc 框架的基础。

读取应答

前面提到在创建一个 rpc 客户端对象的时候，同时会启动一个 input goroutine 来等待服务器的响应，下面的 input 方法就是这个 goroutine 做的所有工作。

```
func (client *Client) input() {
            var err error
            var response Response
            //
            // 这里的 for 循环就是永久负责这个连接的响应读取，只有在连接上发生错误后，才会退出。
            //
            for err == nil {
                        //
                        // 首先是读响应头，响应头一般有一个很重要的信息就是正文数据长度，有了这个长度信息，才知道
                        // 读多少正文才是一个应答完毕。
                        //
                        response = Response{}
                        err = client.codec.ReadResponseHeader(&response)
                        if err != nil {
                                    break
                        }
                        //
                        // 这里是一个很重要的步骤，从响应头里取到 seq，这个 seq 就是客户端生成的 seq，在上文的 send
                        // 过程中发送给了服务器，服务器应答的时候，必须将这个 seq 响应给客户端。只有这样客户端才
                        // 知道这个应答是对应 pending 队列中的那个请求的。
                        //
                        // 这里对 pending 队列枷锁保护，通过 seq 提取对应的请求 call 对象。
                        //
                        seq := response.Seq
                        client.mutex.Lock()
                        call := client.pending[seq]
                        delete(client.pending, seq)
                        client.mutex.Unlock()

                        //
                        // 这个 switch 里主要就是处理异常以及正常情况读取响应正文了。异常情况，英文解释很详细了。
                        //
```

```
                switch {
                case call == nil:
                        // We've got no pending call. That usually means that
                        // WriteRequest partially failed, and call was already
                        // removed; response is a server telling us about an
                        // error reading request body. We should still attempt
                        // to read error body, but there's no one to give it to.
                        err = client.codec.ReadResponseBody(nil)
                        if err != nil {
                                err = errors.New("reading error body: " + err.Error())
                        }
                case response.Error != "":
                        // We've got an error response. Give this to the request;
                        // any subsequent requests will get the ReadResponseBody
                        // error if there is one.
                        call.Error = ServerError(response.Error)
                        err = client.codec.ReadResponseBody(nil)
                        if err != nil {
                                err = errors.New("reading error body: " + err.Error())
                        }
                        call.done()
                default:
                        //
                        // 开始读取响应的正文，正文放到 call 中的 Reply 中去。
                        //
                        err = client.codec.ReadResponseBody(call.Reply)
                        if err != nil {
                                call.Error = errors.New("reading body " + err.Error())
                        }
                        call.done()
                }
        }

        //
        // 下面部分的代码都是在处理连接上出错，以及服务端关闭连接等情况的清理工作．这部分很重要，
        //  否则可能导致一些调用 rpc 的 goroutine 永久阻塞等待，不能恢复工作。
        //
        // Terminate pending calls.
```

```
            client.sending.Lock()
            client.mutex.Lock()
            client.shutdown = true
            closing := client.closing
            if err == io.EOF {
                        if closing {
                                    err = ErrShutdown
                        } else {
                                    err = io.ErrUnexpectedEOF
                        }
            }
            for _, call := range client.pending {
                        call.Error = err
                        call.done()
            }
            client.mutex.Unlock()
            client.sending.Unlock()
            if err != io.EOF && !closing {
                        log.Println("rpc: client protocol error:", err)
            }
}
```

不管有多少 goroutine 在并发写一个 rpc 连接，等待在连接上读取应答的 goroutine 只有一个，这个 goroutine 负责读取连接上的所有响应，然后将响应分发给对应的请求，也就完成了一次 rpc 请求了。

自定义编码解码器

```
type ClientCodec interface {
            WriteRequest(*Request, interface{}) error
            ReadResponseHeader(*Response) error
            ReadResponseBody(interface{}) error

            Close() error
}
```

ClientCodec 定义了编码解码器的接口，也就是说你如果要自定义一个编码解码器，只需要实现这个接口就可以了。

WriteRequest 做的事情就是将 rpc 请求的远程方法名、参数等信息进行序列化打包，然后写入 socket 中。你完全可以按照自己的想法去打包一请求进行发送。 两个 Read 接口就是从 socket 读取响应数据包，然后采用相应的算法进行反序列化解包即可。

Go 的 rpc 如果不能自定义编码解码器，那就只能用在客户端和服务器都是 Go 语言开发的环境中。这就大幅度的降低了其灵活性。能够自定义编码解码器后，理论上就可以用来实现其他 rpc 的协议，比如：thrift。github 的 go-thrift 项目确实是在 Go rpc 的基础上通过自定义编码解码器实现的 thrift 协议。

timeout

Call 调用并没有提供超时机制，如果服务器响应太慢，怎么办？死等下去吗？这很不合理。有人说可以自己实现 Call 方法，不使用 rpc 包默认提供的 Call。代码如下：

```
func (client *Client) Call(serviceMethod string, args interface{}, reply interface{}) error {
            select {
            case call := <-client.Go(serviceMethod, args, reply, make(chan *Call, 1)).Done:
                        return call.Error
            case <-time.After(5*time.Second):
            }
            return errors.New("timeout")
}
```

上面这段代码确实可以做到 rpc 调用 5s 没有得到响应，就超时放弃等待应答。

但是，这样做会有另外一个潜在的风险，就是内存泄漏. 因为，client.Go 这个方法会构造一个 call 对象，则个 call 对象其实就是一个请求，这个请求被发送出去的同时会放到一个 pending 队列中，上文已经分析过了。通过分析 input goroutine 的过程可以看到如果服务器不响应的话，这个 call 对象就一直待在 pending 队列中了，永远不会被删除掉。当然，这种情况说明我们的服务器设计有问题。

实现一个 rpc 客户端，最重要的事情就是连接复用，也就是说一个连接上需要并发的同时跑多个请求。Go rpc 在完成这件事情上，动用了两把锁，这个地方很可能会是影响性能的点。待服务器实现分析完后，我们来动手测试一下 rpc 的性能，看看究竟如何。

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