第三周 总结

这一周的工作主要是调通登录和建岛接口，本周的总结就记录一下这个过程中遇到的坑。

1. io.reader包中

io包中定义了一个接口：reader

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

接口非常简单，实现了Read方法的任何类型都是Reader。Read()方法读取一个对象并返回对象的字节数和一个错误。

实现io.reader接口的结构体有：

- os.File是一个Reader类型
- strings.NewReader()返回一个Reader类型
- http.requests和http.response
- bytes.buffer和bytes.NewReader

这里的问题是对Reader的读取操作之后读取器会停留在数据的结尾，想要再次读取就必须对读取器进行重置操作。在实际应用中，对于http响应的body的读取操作就会造成这个问题，http包中的client进行do操作，同样会使得读取器的指针停留在request的末尾，导致第二次读取到的数据为空：

```
s, _ := ioutil.ReadAll(response.Body)//读取器留在Body末尾
rsp, err := cilent.Do(request)//读取器停留在request末尾
```

这个时候需要调用reader的reset()方法重置读取器

2. 错误处理

由于测试环境经常出问题，这里特地总结了一下解决问题的基本思路，出问题的时候可以从这些方面去寻找原因：

- 查看niginx连接日志

> niginx的连接日志用于判断客户端的请求是否真的到达服务端
>
> 判断客户端的请求包是否为空
>
> 判断服务器响应的状态码，响应主体、响应包的大小

- 查看进程的状态

> 测试环境没有运行成功经常是某个进程挂掉或者没有启动的原因，比如nginx没有启动，etcd挂掉，或者是core服务等的问题

ps命令

> ps a 显示现行终端机下的所有程序，包括其他用户的程序。
>
> ps -A 显示所有进程。
>
> ps c 列出程序时，显示每个程序真正的指令名称，而不包含路径，参数或常驻服务的标示。
>
> ps -e 此参数的效果和指定"A"参数相同。
>
> ps e 列出程序时，显示每个程序所使用的环境变量。
>
> ps f 用ASCII字符显示树状结构，表达程序间的相互关系。
>
> ps -H 显示树状结构，表示程序间的相互关系。
>
> ps -N 显示所有的程序，除了执行ps指令终端机下的程序之外。

> ps s 采用程序信号的格式显示程序状况。
>
> ps S 列出程序时，包括已中断的子程序资料。
>
> ps -t<终端机编号 指定终端机编号，并列出属于该终端机的程序的状况。
>
> ps u 以用户为主的格式来显示程序状况。
>
> ps x 显示所有程序，不以终端机来区分。

grep命令

> [options]主要参数：
>
> —c：只输出匹配行的计数。
>
> —I：不区分大 小写(只适用于单字符)。
>
> —h：查询多文件时不显示文件名。
>
> —l：查询多文件时只输出包含匹配字符的文件名。
>
> —n：显示匹配行及 行号。
>
> —s：不显示不存在或无匹配文本的错误信息。
>
> —v：显示不包含匹配文本的所有行。

pattern正则表达式主要参数：

> \： 忽略正则表达式中特殊字符的原有含义。
>
> ^：匹配正则表达式的开始行。
>
> $: 匹配正则表达式的结束行。
>
> \<：从匹配正则表达 式的行开始。
>
> >：到匹配正则表达式的行结束。
>
> [ ]：单个字符，如[A]即A符合要求 。
>
> [-]：范围，如[A-Z]，即A、B、C一直到Z都符合要求 。
>
> .：所有的单个字符。
>
> * ：有字符，长度可以为0。

- 查看core，game的日志

> 通过查看这些服务的日志可以知道，请求包和响应包的具体字段和内容
>
> 响应包中还包含了请求包的响应状态码setCode,通过状态码判断错误类型
>
> 这些日志都被写到log文件夹中
>
> 实时查看日志：tail -f [file]

3. 实现并发连接

压力测试需要模拟实际运营中大量用户同时登录、建岛时的高并发场景，这需要在测试程序中构建多个tcp连接。

通过http包中默认空实现的http client，并且构造多个并发的goroutine，然后在这些goroutine中模拟高并发的http，现在的问题是这样实现的并发，是不是真实的并发连接。

先看http包中的client结构：

```
type Client struct {
```

```
    Transport RoundTripper//建立TCP连接

    CheckRedirect func(req *Request, via []*Request) error

    Jar CookieJar

    Timeout time.Duration
}
```

建立tcp连接，发送http请求都是通过Transport完成的。再来看看Transport的结构

```
type Transport struct {
idleMu     sync.Mutex
wantIdle   bool // user has requested to close all idle conns


idleConn   map[connectMethodKey][]*persistConn//不同请求的长连接映射


idleConnCh map[connectMethodKey]chan *persistConn//并发请求时在不同的goroutine里相互传输长连接

reqMu       sync.Mutex
reqCanceler map[*Request]func()

altMu    sync.RWMutex
altProto map[string]RoundTripper


//Dial获取一个tcp 连接
Dial func(network, addr string) (net.Conn, error)
}
```

client的发送http请求建立连接只偶调用的都是Transport的RoundTrip方法：

```
func (t *Transport) RoundTrip(req *Request) (resp *Response, err error) {
...

pconn, err := t.getConn(req, cm)//getConn获取一个TCP长连接
if err != nil {
    t.setReqCanceler(req, nil)
    req.closeBody()
    return nil, err
}

...

return pconn.roundTrip(treq)//调用持久连接的roundTrip方法
}
```

关于getConn如何实现的：

```
func (t *Transport) getConn(req *Request, cm connectMethod) (*persistConn, error) {

...

dialc := make(chan dialRes)
//定义了一个发送 persistConn的channel

...

cancelc := make(chan struct{})
t.setReqCanceler(req, func() { close(cancelc) })

// 启动了一个goroutine, 这个goroutine 获取里面调用dialConn搞到
// persistConn, 然后发送到上面建立的channel  dialc里面,
go func() {
    pc, err := t.dialConn(cm)
    dialc <- dialRes{pc, err}
}()

idleConnCh := t.getIdleConnCh(cm)
select {
case v := <-dialc:
    // dialc 我们的 dial 方法先搞到通过 dialc通道发过来了
    return v.pc, v.err
case pc := <-idleConnCh:
    // 这里代表其他的http请求用完了归还的persistConn通过idleConnCh这个
    // channel发送来的
```

```
    handlePendingDial()
    return pc, nil
case <-req.Cancel:
    handlePendingDial()
    return nil, errors.New("net/http: request canceled while waiting for connection")
case <-cancelc:
    handlePendingDial()
    return nil, errors.New("net/http: request canceled while waiting for connection")
}
```

}

首先定义了一个发送 persistConn的channel dialc， 然后启动了一个goroutine, 这个goroutine 获取里面调用dial方法建立的TCP长连接, 然后发送到dialc里面，主协程goroutine在select里面监听多个channel,看看哪个通道里面先发过来 persistConn，就用哪个，然后return。

idleConnCh 这个通道里面发送来的是其他的http请求用完了归还的persistConn， 如果从这个通道里面搞到了，就通过handlePendingDial这个方法把dialc通道里面的persistConn也发到idleConnCh，等待后续gorountine的http请求使用

结论是通过client实现的并发连接是通过连接池的方式实现的，连接池会重复使用goroutine建立的TCP连接，通过http包的请求的并发数量实际上少于预期，需要手动建立tcp连接才能实现。