Go语言网络编程

一、课前准备

> 说明：提前需要让学生做的课前准备，比如环境安装部署、工具安装、插件安装等，需要学生提前做的都放这，并且给出下载链接或信息源；

1. Golang开发环境环境安装就绪；
2. Golang-IDE开发环境安装就绪；
3. 练习Go语言基础语法代码；
4. 熟练掌握Go语言并发与同步；
5. 了解一些网络知识；

二、课堂主题

> 说明：本堂课的总体概述，明确课堂主题和主体；

本节主要介绍Go语言当中的网络编程，内容包含文件IO处理，TCP编程，HTTP编程。

三、课堂目标

> 说明：主要是让学生了解，学了本堂课后，能达到的一个期望值，要量化；

1. 掌握文件IO处理；
2. 掌握目录IO处理；
3. 掌握TCP通信编程；
4. 掌握HTTP通信编程；

四、知识点

Go语言属于封装类语言，我们在进行网络编程时甚至不必知道网络通信的细节，也可以编写出非常不错的网络服务器代码。

1. 文件IO处理

1.1 文件描述符

不管是网络编程，还是文件IO操作，其实都是IO，只不过面向的设备不同。文件IO面向的是本地文件，网络编程面向的则是网络设备文件。当然学到这里，不得不提的一个概念是文件描述符。

文件描述符是一个非常抽象的概念，顾名思义它就是一个描述文件的符号，我们简单的认为它就是一个编号，一个句柄。

通过文件描述符，可以将数据写入缓冲区，进程的刷新机制会将内容通过设备驱动保存到磁盘，在读取文件时，原理是相同的。需要知道的是，在我们启动一个进程时，系统会默认打开3个文件描述符，分别是标准输入、标准输出以及标准错误。

1.2 文件IO相关的API

对一个文件操作，很显然要知道相关的API，首先我们借助 `os` 包可以打开或新创建一个文件。

```
func OpenFile(name string, flag int, perm FileMode) (*File, error)
```

- name 代表要打开的文件名字
- flag代表打开的权限
    - O_RDONLY 只读
    - O_WRONLY 只写
    - O_RDWR 读写
    - O_CREATE 创建文件
    - O_TRUNC 截断文件
- perm 8进制文件权限位，与umask共同作用，决定最终文件权限位
- 返回值

    - *File 文件描述符
    - error 错误信息，nil代表没有错误

在得到文件描述符后就方便多了，剩下的IO操作其实都是File对应的方法。

```
func (f *File) Read(b []byte) (n int, err error)
func (f *File) Write(b []byte) (n int, err error)
func (f *File) WriteString(s string) (n int, err error)
func (f *File) Seek(offset int64, whence int) (ret int64, err error)
```

上述函数看名称基本看名称就可以知道含义，注意`Read`和`Write`的返回值都是读或写的字节数。`Seek`需要特别介绍一下，Seek代表调整文件读写位置，它通过2个参数进行调整：

- offset 偏移量
- whence 位置
    - os.SEEK_SET 文件头
    - os.SEEK_CUR 当前位置
    - os.SEEK_END 结束位置
- 返回值
    - ret 移动的距离，错误信息

在了解了相关的API后，我们可以来做一个小案例：创建并打开一个文件，将`who am i`写入文件，并读取文件内容，将读到的信息打印在屏幕。

1.3 目录操作

在linux平台，我们认为一切皆文件，即使目录也是一种特殊的文件，虽然Go语言是跨平台的，但是基于这样的思想，我们的目录操作和文件操作是紧密相关的。对于目录的打开，我们使用`Open`函数，它实际内部调用的是`Openfile`。

```
func Open(name string) (*File, error) {
  return OpenFile(name, O_RDONLY, 0)
}
```

当我们`name`传入为一个目录时，我们就打开了一个目录，如果传入为一个文件，那么打开的就是一个文件。

```
func (f *File) Readdir(n int) ([]FileInfo, error)
```

再通过`Readdir`函数，我们可以清晰的读到一个目录有哪些内容，其中n代表要读取的数量，如果小于0则代表全部读取；[]FileInfo是读到的文件信息，对应的结构如下：

```
type FileInfo interface {
  Name() string       // base name of the file
  Size() int64        // length in bytes for regular files; system-dependent
for others
  Mode() FileMode     // file mode bits
  ModTime() time.Time // modification time
  IsDir() bool        // abbreviation for Mode().IsDir()
  Sys() interface{}   // underlying data source (can return nil)
}
```

基于以上我们API，我们再来实现一个小案例：读取当前目录下所有内容，并且打印到屏幕。

2. 错误处理

2.1 错误的分类

我们在前面处理文件时，看到了error这个错误返回类型，接下来我们介绍一下Go语言中的错误处理方式。在开发过程中，我们经常听到错误和异常这2个词语，这两个词语看上去差不多，实际上还是有一些区别的。

- 错误 发生非期望的已知行为，这类错误我们是可以预见的
- 异常 发生非期望的未知行为

异常是程序执行时发生的未预先定义的错误，这个错误被操作系统捕获，比如类似C语言的段错误。Go语言是类型安全的语言，它不会发生编译器或运行时（runtime）无法捕获的异常。所以，我们在Go语言处理的实际上都算是错误。

对于错误的处理原则无外乎2个，当前这也是提前就会定义好的：

- 可继续错误 当一些错误发生时，并不影响服务继续运行，进程该继续运行
- 无法继续错误 当某些错误发生时，服务已经无法正常运行，此时应该停止进程

2.2 panic和recover

至于错误的处理方式，开发者并不陌生。在大多数的编程开发中，在函数实现上我们多是通过返回值将函数调用情况返回给上层调用者，上层调用者再把自己的反馈传递给更上层调用者，这样逐层向上传递信息，由最高层来决定是退出还是继续。在Go语言当中，我们可以通过error这个返回值类型进行逐层传递，而error本身是一个interface接口。

```
type error interface  {
    Error() string
}
```

只要实现了Error() 函数，都可以当作错误做为输出。我们在处理时，判断有没有错误，通常都是先判断其值是否为“nil”，这个“nil”是Go语言当中控值的表示方法，如果error非空，则代表存在错误，需要处理。

不过在Go语言当中，给了我们另外一个选择，我们可以借助Go语言的`panic`和`recover`模式来处理错误。这两个都是Go语言的内置函数：

```
func panic(v interface{})
func recover(v interface{})
```

panic函数是抛出一个恐慌，不必被这个词吓到，一般代表碰到此类错误，事情比较严重了。panic的触发机制有两种，一种是运行时错误时自动触发的，还有一种是我们人为调用的，当发生panic时，程序会从panic函数的位置立即返回，逐层打印堆栈信息，如果我们不管不顾的话，进程就会退出。

panic恐慌发生时，我们如果不想退出，就可以使用recover来捕捉panic恐慌，不过recover使用却有一些小个性。例如我们这样使用，是捕获不到panic的。

```
package main

import (
  "fmt"
)

func panic_recover() {
  recover()
  panic("game over") //抛出错误

  fmt.Println("panic_recover run over")
}

func main() {
  recover()
  panic_recover()
  fmt.Println("heihei,game not over!")
}
```

执行上述代码，发现recover并没有像想象般的那样发挥出了作用。

recover能够顺利捕获panic的第一个要求是必须在panic执行前先要延迟执行recover，所以下面需要为大家介绍一下defer关键字，defer是Go语言设计中非常温馨的一个关键字，它解决了我们对于延迟执行的需求，defer后可放置要执行的函数语句，该语句会在其所在代码段运行结束后（正常结束、提前return、panic）触发该函数的执行。而recover的另外一个要求是必须在延迟的函数体内执行才可以。

2.3 defer关键字

defer关键字是Go语言设计中非常不错的发明，defer加上一个函数调用，代表当前代码段结束后才会执行defer。

```go
package main

import (
  "fmt"
)

func main() {
  fmt.Println("begin")
  defer func() {
    fmt.Println("I am defer")
  }()
  fmt.Println("end")
}
```

执行上述代码，得到如下结果：

```
localhost:demo teacher$ go run 02-defer.go
begin
end
I am defer
```

defer在文件打开，防止忘记关闭等方面有着非常天然的优势。不过接下来我们还是用defer解决panic与recover的问题。recover能够成功，必须要借助defer！

```go
package main

import (
  "fmt"
)

func panic_recover() {
  //recover()
  defer func() { //延迟定义捕获
    fmt.Println("defer recover panic") //会打印
    recover()
  }()
  panic("game over") //抛出错误

  fmt.Println("panic_recover run over") //不会打印
}

func main() {
  //recover()
  panic_recover()
  fmt.Println("heihei,game not over!") //捕获成功则会打印本句
}
```

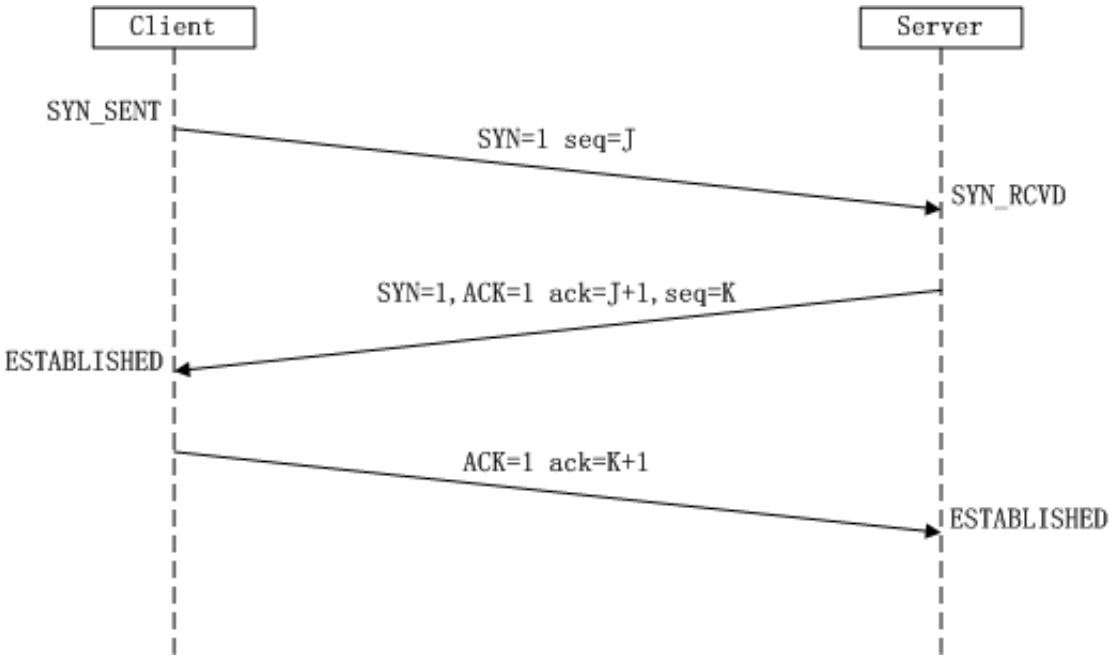
顺便再总结一下recover的要求：

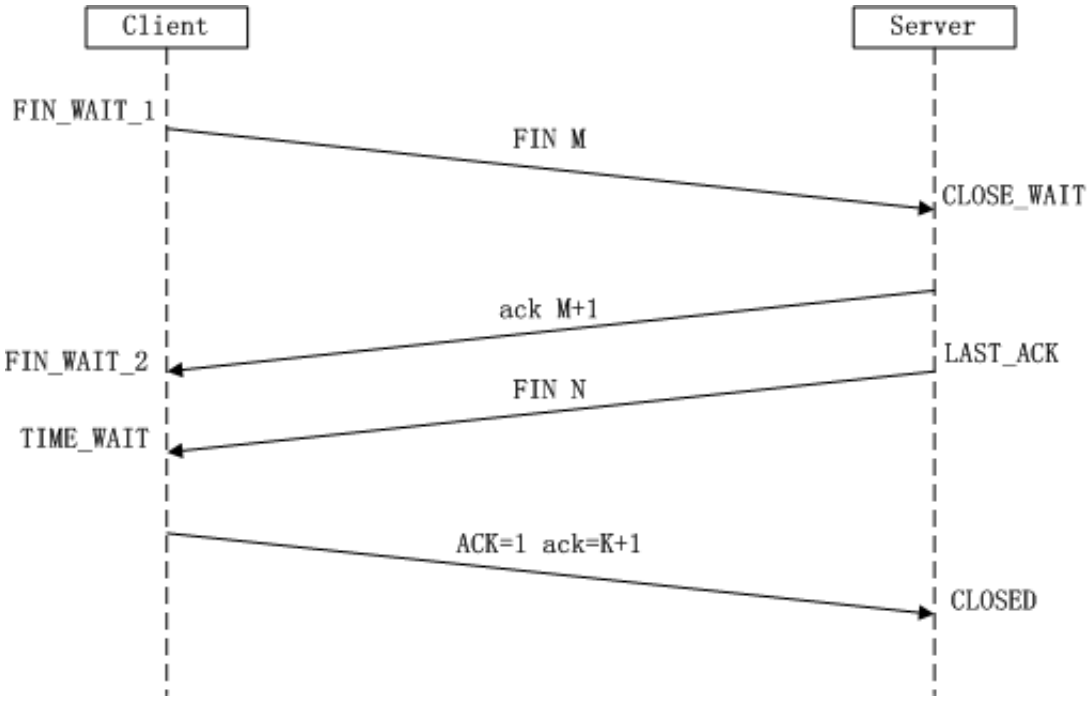
- 必须在panic执行前先要延迟执行recover
- 必须在延迟的函数体内执行才可以

3. TCP编程

3.1 TCP协议简介

在网络知识中，最著名的是IOS标准的七层模型，不过在使用上最为广泛的还是TCP/IP的四层模型，对应的四层分别是数据链路层、网络层、传输层以及应用层。网络编程解决的是两个进程间的通信问题，面向的场景是两个进程不在同一主机的情况。TCP/IP协议定义了两个进程要想建立连接需要通过三次握手，之后可以彼此传输消息，而要断开连接，需要四次握手。





TCP通信的核心思想就是数据包的封装与拆包，在发送消息端的进程，将目标发送消息通过网络分层时逐层封包，最终在底层网络传输给对方，而对方收到消息后，再由底层到上层逐层解包，拿到对方发送的实际消息，当然这个过程开发者不需要考虑，这是在协议层支持的。

TCP/IP协议层最关键的定义是IP和端口，IP可以在一个网络内唯一的标识一台主机，而端口则是在一个主机上可以唯一的标识一个进程，IP+端口的方式就可以确定唯一一个主机上的唯一一个进程。先来说说IP，目前网络编程多使用的是IPv4协议，ip地址是4组0-255的数字组合（这是人类可识别的表示方法），在网络编程中对应的其实就是0或1的数字。再来说说端口，端口是一个短整型的数字，它的范围是0-65535，需要需要注意的是有些端口已经被占用，并且形成了约定俗成的规则，比如80端口就是web服务器使用的端口，mysql用的端口一般是3306，MongoDB使用的端口默认是27017等。

3.2 Go语言TCP服务器

在Go语言当中，理论知道当然最好，但是开发的时候只是使用几个API就够了，它内部进行了很好的封装。建立一个TCP服务器的基本步骤是这样的：

1. 绑定IP和端口
2. 建立侦听
3. 等待侦听结果，得到新连接
4. 与得到的新连接进行通信

而我们在写这个服务的时候，借助net包可以很轻易的完成：

```
func Listen(network, address string) (Listener, error) //建立一个侦听者
```

这个函数在实现的时候，帮我们做了ip和端口的绑定，同时开始侦听。其中network是协议类型，我们可以传tcp，address是网络地址，它的格式是“ip:port”，如果针对本机任何ip都侦听，ip可以省略。

```
type Listener interface {
  // Accept waits for and returns the next connection to the listener.
  Accept() (Conn, error)

  // Close closes the listener.
  // Any blocked Accept operations will be unblocked and return errors.
  Close() error

  // Addr returns the listener's network address.
  Addr() Addr
}
```

得到的Listener是一个接口类型，它有一个Close方法和Accept方法，Accept方法就是用来获得新连接的，在得到新连接后，我们可以与它进行通信。得到的conn同样是一个接口类型：

```
type Conn interface {
  // Read reads data from the connection.
  // Read can be made to time out and return an Error with Timeout() == true
  // after a fixed time limit; see SetDeadline and SetReadDeadline.
  Read(b []byte) (n int, err error)

  // Write writes data to the connection.
  // Write can be made to time out and return an Error with Timeout() == true
  // after a fixed time limit; see SetDeadline and SetWriteDeadline.
  Write(b []byte) (n int, err error)

  // Close closes the connection.
  // Any blocked Read or Write operations will be unblocked and return errors.
  Close() error

  // LocalAddr returns the local network address.
  LocalAddr() Addr

  // RemoteAddr returns the remote network address.
  RemoteAddr() Addr

  ……
}
```

所以接下来的事情就是读写就够了！我们用上述的API来实现一个回射服务器，服务器做的事情就是客户端不管发来什么，都发回去什么！

```
package main

import (
  "fmt"
  "io"
  "log"
```

```
  "net"
)

func main() {
  //1. 指定为ipv4协议，绑定IP和端口，启动侦听
  listener, err := net.Listen("tcp", "localhost:8888")
  if err != nil {
    log.Panic("Failed to Listen", err) //输出错误信息，并且执行panic错误
  }
  defer listener.Close() //收尾工作
  for {
    //2. 循环等待新连接到来，Accept阻塞等待状态
    conn, err := listener.Accept()
    if err != nil {
      fmt.Println("Failed to Accept ", err)
      continue
    }
    fmt.Println("New conn->", conn.RemoteAddr().String())
    //3. 启动匿名函数来处理
    go func(conn net.Conn) {
      defer conn.Close() //收尾工作
      buf := make([]byte, 256)
      for {
        //从客户端读数据
        n, err := conn.Read(buf)
        if err != nil {
          if err == io.EOF { //这种错误代表客户端先关闭了，属于正常范围内的错误
            fmt.Println("Client ", conn.RemoteAddr().String(), " is Closed")
            break
          }
          fmt.Println("Failed to Read data ", err)
          break
        }
        //回射服务器，收到什么，写回什么
        n, err = conn.Write(buf[:n])
        if err != nil {
          fmt.Println("Failed to Write to client ",
conn.RemoteAddr().String(), err)
          break
        }
      }

    }(conn)
  }
}
```

把这个服务器启动后，就可以等待客户端来连接了，正常情况我们应该自己写一个客户端程序，不过在unix平台，可以借助一个nc（net connect）命令来测试服务器情况，新打开一个终端，如下启动即可：

```
localhost: teacher$ nc 127.0.0.1 8888
abcdef
abcdef
nihao
nihao
```

4. HTTP编程

4.1 HTTP协议

HTTP协议（超文本传输协议）是TCP/IP分层模型中应用层的协议，也是浏览器与web服务器之间的通信协议。它的特点在于侧重于表示，简单点说就是客户端发过来一个请求，通过协议的方式描述出具体的内容，比如客户端是什么样子（什么浏览器），请求的资源是什么等等。当然服务器端传给客户端的消息也要遵循HTTP协议，描述响应码，资源类型，内容长度，资源信息等。无论是请求还是发送，都是由一系列键值对组成。

如果我们想看一下HTTP协议到底有怎么样的效果，可以写一个tcp服务器，然后在浏览器请求一下，就可以见识到http的请求是什么样子了。

```go
package main

import (
  "fmt"
  "net"
)

func main() {
  listener, _ := net.Listen("tcp", ":8080")
  for {
    conn, _ := listener.Accept()
    go func(conn net.Conn) {
      buf := make([]byte, 2048)
      conn.Read(buf)
      fmt.Println(string(buf))
    }(conn)
  }
}
```

执行后，用浏览器请求就可以看到效果了：

```
localhost:demo teacher$ go run 03-tcp.go
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

```
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_4) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/78.0.3904.70 Safari/537.36
Sec-Fetch-User: ?1
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/
*;q=0.8,application/signed-exchange;v=b3
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9
```

请求消息中，第一行最为关键，它告诉我们浏览器请求的方法是什么，请求的资源是什么，以及所用的HTTP协议版本号是多少。至于请求的方法可以有：GET、POST、PUT、DELETE等方法，请求的资源“/”代表请求的是服务器的根目录，这里有个专业的名词叫URL（专业点说法叫统一资源定位符），表示资源所在的具体路径。User-Agent表达的是浏览器信息，在上述的例子里我们看到请求的浏览器是chrome。

4.2 HTTP协议服务器

我们在Go语言当中，想实现一个HTTP的服务器其实很容易。

```
func ListenAndServe(addr string, handler Handler) error //启动侦听并提供web服务
```

在此之前需要设置好handle路由器，对于不同的http请求，也就是请求的url不同的时候，给予不同的响应。

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))  //路由
函数
```

HandleFunc是用来设置当pattern请求到来的时候，使用后面的handler函数来处理。 这个是一个简单的正则规则，比如我们想让所有hello相关的请求都是类似这样的url：/hello/username，对于这样的请求，我们给予的响应是：hello, username， 我们可以这样设置路由：

```
http.HandleFunc("/hello/", HelloUserServer)
```

当然这个HelloUserServer是需要实现的，格式必须是handler的格式。

```
func HelloUserServer(w http.ResponseWriter, req *http.Request) {

  path := req.URL.Path //得到请求的url
  fmt.Println(path)
  users := strings.Split(path, "/")
  fmt.Println(len(users), users)
  if len(users) == 3 {
    io.WriteString(w, users[1]+","+users[2]) //组织一个响应消息
  }

}
```

整体可以看如下代码，代码中又增加了一个byebye的路由函数：

```
package main

import (
  "fmt"
  "io"
  "log"
  "net/http"
  "strings"
)

//router hello
func HelloUserServer(w http.ResponseWriter, req *http.Request) {

  path := req.URL.Path
  fmt.Println(path)
  users := strings.Split(path, "/")
  fmt.Println(len(users), users)
  if len(users) == 3 {
    io.WriteString(w, users[1]+","+users[2])
  }

}

//router byebye
func ByeUserServer(w http.ResponseWriter, req *http.Request) {

  path := req.URL.Path
  users := strings.Split(path, "/")
  if len(users) == 3 {
    io.WriteString(w, users[1]+","+users[2])
  }

}
```

```go
func main() {
  //设置hello的路由
  http.HandleFunc("/hello/", HelloUserServer)
  //设置byebye的路由
  http.HandleFunc("/bye/", ByeUserServer)
  //侦听并提供web服务，所有的事情都在前面设置的路由函数实现了
  err := http.ListenAndServe(":12345", nil)
  if err != nil {
    log.Fatal("ListenAndServe: ", err)
  }
}
```

将服务器启动后，在浏览器针对不同的输入可以看到不同的结果：

浏览器输入：http://localhost:12345/hello/teacher

浏览器输入：http://localhost:12345/bye/teacher

五、拓展点

> 说明：
>
> 1. 典型面试题、笔试题；
> 2. 新技术 or 经验分享；
> 3. 未来计划、行业趋势分享；

六、总结

> 说明：
>
> 1. 回顾本堂课所有知识点；
> 2. 提示`注意点`和`重点`；
> 3. 提示学习方法；
> 4. 提示哪些需要记、哪些需要背、哪些代码需要敲；

- 一切皆文件
- 关注TCP和HTTP协议
- 重点记忆panic和recover的使用方法
- 关注defer关键字的使用

七、大作业

> 说明：
>
> 1. 给出明确的作业要求，形成文字、图片或测试题等形式；

2. 给出明确的解答方式；
3. 频次低，可设计为：1次/周 或 1次/2周；

- 实现基于TCP协议的并发聊天室

八、集中答疑

> 说明：完成本堂课所有知识点的讲解后，由学员集中提问，讲师逐一解答；

九、检测题

> 说明：
>
> 1. 针对本堂课，设计5-10个题，由学员课下完成（课下刷题）；
> 2. 出题范围可从检查点里挑选，可以是面试题或笔试题；
> 3. 题型要求是 单选、多选、判断、填空中的一种或多种；
> 4. 频次高，原则上每次课都要有检测题；

十、下节课预告

> 说明：
>
> 1. Go语言数据库编程
> 2. Go语言博客项目开发；