Golang基础介绍

fengge

一．课程目标

二．Golang语言发展背景

三．基本语法

1. 数据类型
2. 变量定义
3. 控制结构
4. 函数function
5. 结构体struct
6. 面向对象
7. 接口interface
8. 恐慌panic和恢复recover
9. 并发goroutine和channel
10. Import和package
11. main,init,test
12. 指针和内存分配(new和make)

四．总结下Golang的优缺点

五．Web小实例

课程目标

1：熟悉go语言的基本语法和一些特性

2：能够用go语言编写简单的web程序

Golang语言发展背景和特性概览

- **Ken Thompson**：1983年图灵奖（Turing Award）和1998年美国国家技术奖（National Medal of Technology）得主。他与Dennis Ritchie是Unix的原创者。
- **Rob Pike**：曾是贝尔实验室（Bell Labs）的Unix团队，和Plan 9操作系统计划的成员。他与Thompson共事多年，并共创出广泛使用的UTF-8 字元编码。
- **Robert Griesemer**：曾协助制作Java的HotSpot编译器，和Chrome浏览器的JavaScript引擎V8。

Go被设计为21世纪的C语言，同时它**吸收了很多现在编程语言的优点**。
基于BSD完全开源，所以能**免费**的被任何人用于适合商业目的。

它不是传统意义上的**面向对象**语言（没有类的概念），但它有接口（interface），由此实现多态特性。
函数（Function）是它的基本构成单元（也可以叫**面向函数**的程序设计语言）。

语言层面对**并发**的支持
　　在语言层面加入对并发的支持，而不是以库的形式提供
　　更高层次的并发抽象，而不是直接暴露OS的并发机制。

类型安全和内存安全：**有指针类型，但不允许对指针进行操作**
具有内存**垃圾回收**机制
支持网络通信、并发控制、并行计算和分布式计算。
在语言层面实现对多处理器（或多核）进行编程

数据类型

基本类型：

- 布尔：bool
- 字符串：string
- 整数：
  - int 在 32 位操作系统上，它们均使用 32 位（4 个字节），在 64 位操作系统上，它们均使用 64 位（8 个字节）
  - int8（-128 -> 127）
  - int16（-32768 -> 32767）
  - int32（-2,147,483,648 -> 2,147,483,647）
  - int64（-9,223,372,036,854,775,808 -> 9,223,372,036,854,775,807）
- 无符号整数：
  - uint
  - uint8（0 -> 255）
  - uint16（0 -> 65,535）
  - uint32（0 -> 4,294,967,295）
  - uint64（0 -> 18,446,744,073,709,551,615）
- 浮点型（IEEE-754 标准）：
  - 没有float
  - float32（+- 1e-45 -> +- 3.4 * 1e38）
  - float64（+- 5 1e-324 -> 107 1e308）
- 复数
  - complex64 (32 位实数和虚数)
  - complex128 (64 位实数和虚数)

(rune是int32的别称，byte是uint8的别称)

变量定义

使用var关键字是Go最基本的定义变量方式，与c++语言不同的是Go把变量类型放在变量名后面：

```
//定义一个名称为“variableName”，类型为"type"的变量
var variableName type
```

定义多个变量

```
//定义三个类型都是“type”的变量
var vname1, vname2, vname3 type
```

定义变量并初始化值

```
//初始化“variableName”的变量为“value”值，类型是“type”
var variableName type = value
```

同时初始化多个变量

```
/*
    定义三个类型都是"type"的变量,并且分别初始化为相应的值
    vname1为v1，vname2为v2，vname3为v3
*/
var vname1, vname2, vname3 type= v1, v2, v3
```

变量定义

Array

```
a := [3]int{1, 2, 3} // 声明了一个长度为3的int数组
b := [10]int{1, 2, 3} // 声明了一个长度为10的int数组，其中前三个元素初始化为1、2、3，其它默认为0
c := [...]int{4, 5, 6} // 可以省略长度而采用`...`的方式，Go会自动根据元素个数来计算长度
```

Slice

```
slice := []byte {'a', 'b', 'c', ‘d’}

// 声明一个数组
var array = [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
// 声明两个slice
var aSlice
// 一些简便操作
aSlice = array[:3] // 等价于aSlice = array[0:3] aSlice包含元素: a,b,c
aSlice = array[5:] // 等价于aSlice = array[5:10] aSlice包含元素: f,g,h,i,j
aSlice = array[:]  // 等价于aSlice = array[0:10] 这样aSlice包含了全部的元素
```

Map

```
// 声明一个key是字符串，值为int的字典,这种方式的声明需要在使用之前使用make初始化
var numbers map[string]int
// 另一种map的声明方式
numbers := make(map[string]int)
numbers["one"] = 1  //赋值
numbers["ten"] = 10 //赋值
numbers["three"] = 3
```

控制结构

if

```
if integer == 3 {
    fmt.Println("The integer is equal to 3")
} else if integer < 3 {
    fmt.Println("The integer is less than 3")
} else {
    fmt.Println("The integer is greater than 3")
}
```

```
// 计算获取值x,然后根据x返回的大小，判断是否大于10
if x := computedValue(); x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}

//这个地方如果这样调用就编译出错了，因为x是条件里面的变量
fmt.Println(x)
```

goto

```
func myFunc() {
    i := 0
Here:   //这行的第一个词，以冒号结束作为标签
    println(i)
    i++
    goto Here   //跳转到Here去
}
```

for

```
 sum := 0;
for index:=0; index < 10 ; index++ {
    sum += index
}
fmt.Println("sum is equal to ", sum)
```

```
sum := 1
for sum < 10 {
    sum += sum
}
```

```
for k,v:=range map {
    fmt.Println("map's key:",k)
    fmt.Println("map's val:",v)
}
```

控制结构

switch

```
i := 10
switch i {
case 1:
    fmt.Println("i is equal to 1")
case 2, 3, 4:
    fmt.Println("i is equal to 2, 3 or 4")
case 10:
    fmt.Println("i is equal to 10")
default:
    fmt.Println("All I know is that i is an integer")
}
```

switch默认相当于**每个case最后带有break**，
匹配成功后跳出整个switch;
可以使用**fallthrough**执行后面的case

```
integer := 6
switch integer {
case 4:
    fmt.Println("The integer was <= 4")
    fallthrough
case 5:
    fmt.Println("The integer was <= 5")
    fallthrough
case 6:
    fmt.Println("The integer was <= 6")
    fallthrough
case 7:
    fmt.Println("The integer was <= 7")
    fallthrough
case 8:
    fmt.Println("The integer was <= 8")
    fallthrough
default:
    fmt.Println("default case")
}
```

输出

```
The integer was <= 6
The integer was <= 7
The integer was <= 8
default case
```

函数function

func

1.可以有零至多个参数和零至多个返回值

```
func funcName(input1 type1, input2 type2)
(output1 type1, output2 type2) {
    //这里是处理逻辑代码
    //返回多个值
    return value1, value2
}
```

2.支持可变参

```
func myfunc(arg ...int) {}
```

3.忽略不需要的返回值

```
value1 _ : = funcName(input1 type1, input2 type2)
```

4.函数的另一种形态，带有接收者的函数，称为method

```
func (stu type0)funcName(input1 type1, input2 type2)
 (output1 type1, output2 type2) {
    //这里是处理逻辑代码
    //返回多个值
    return value1, value2
}
```

5.函数也可以当做参数传递

```
func function(a, b int, sum func(int, int) int) {
    fmt.Println(sum(a, b))
}

func sum(a, b int) int {
    return a + b
}
```

6.defer延迟函数,释放资源,恐慌恢复,在函数return后执行,可以修改return的值,但不建议

如果定义多个defer,采用后进先出模式

```
func ReadWrite() bool {
    file.Open("file")
    defer file.Close()
    if failureX {
        return false
    }
    if failureY {
        return false
    }
    return true
}
```

结构体struct

struct

1.定义

```
type person struct {
    name string
    age int
}

var P person  // P现在就是person类型的变量了
P.name = "Astaxie"  // 赋值"Astaxie"给P的name属性.
P.age = 25  // 赋值"25"给变量P的age属性
```

3.如果存在重复字段

Student.重名字段
Student.Human.重名字段

2.匿名字段

```
type Human struct {
    name string
    age int
    weight int
}

type Student struct {
    Human  // 匿名字段，那么默认Student就包含了Human的所有字段
    speciality string
}

func main() {
    // 我们初始化一个学生
    mark := Student{Human{"Mark", 25, 120}, "Computer Science"}

    // 我们访问相应的字段
    fmt.Println("His name is ", mark.name)
    fmt.Println("His age is ", mark.age)
    fmt.Println("His weight is ", mark.weight)
    fmt.Println("His speciality is ", mark.speciality)
```

面向对象

函数的另一种形态，带有接收者的函数，称为method

1.可以用**接受者.函数名**调用

2.method可以和strut中的字段一样被**继承**

3.method也可以被**重写**

4. **不支持重载**

```
type Human struct {
    name string
    age int
}

type Employee struct {
    Human //匿名字段
    company string
}

//Human定义method
func (h Human) SayHi() {
}

//Employee的method重写Human的method
func (e Employee) SayHi() {
}
```

```
type Rectangle struct {
    width, height float64
}
type Circle struct {
    radius float64
}

func (r Rectangle) area() float64 {
    return r.width*r.height
}
func (c Circle) area() float64 {
    return c.radius * c.radius * math.Pi
}

func main() {
    r1 := Rectangle{12, 2}
    r2 := Rectangle{9, 4}
    c1 := Circle{10}
    c2 := Circle{25}

    fmt.Println("Area of r1 is: ", r1.area())
    fmt.Println("Area of r2 is: ", r2.area())
    fmt.Println("Area of c1 is: ", c1.area())
    fmt.Println("Area of c2 is: ", c2.area())
}
```

接口interface

interface

简单的说，interface是一组method的组合，我们通过interface来定义对象的一组行为

```
type Human struct {
    name string
    age int
}
type Employee struct {
    Human //匿名字段
    company string
}
type Student struct {
    Human
    speciality string
}
//Human实现SayHi方法
func (h Human) SayHi() {
}

//Human实现Sing方法
func (h Human) Sing(lyrics string) {
}

//Employee重载Human的SayHi方法
func (e Employee) SayHi() {
}
….
```

```
// Interface Men被Human和Employee实现
// 因为这两个类型都实现了这两个方法
type Men interface {
    SayHi()
    Sing(lyrics string)
}

func main() {
    mike := Student{Human{"Mike", 25, "222-222-XXX"}, "MIT", 0.00}
    tom := Employee{Human{"Tom", 37, "222-444-XXX"}, "Things Ltd.", 5000}

    //定义Men类型的变量i
    var i Men

    //i能存储Student
    i = mike
    i.SayHi()
    i.Sing("November rain")

    //i也能存储Employee
    i = tom
    i.SayHi()
    i.Sing("Born to be wild")
}
```

interface的变量里面可以存储任意类型的数值(该类型实现了interface)

Interface可以像struct一样进行嵌套

空interface(interface{})不包含任何的method，所有的类型都实现了空interface,空interface可以存储任意类型的数值

恐慌panic和恢复recover

Panic

可以中断原有的控制流程
恐慌可以直接调用panic产生。也可以由运行时错误产生，例如访问越界的数组。

Recover

可以让进入令人恐慌的流程中的goroutine恢复过来
recover仅在延迟函数中有效。在正常的执行过程中，调用recover会返回nil，并且没有其它任何效果。
如果当前的goroutine陷入恐慌，调用recover可以捕获到panic的输入值，并且恢复正常的执行。

② c

③ 55

④ d

① a

```
func main() {
    //必须要先声明defer，否则不能捕获到panic异常
    defer func() {
        fmt.Println("c")
        if err := recover(); err != nil {
            //这里的err其实就是panic传入的内容，55
            fmt.Println(err)
        }
        fmt.Println("d")
    }()
    f()
}

func f() {
    fmt.Println("a")
    panic(55)
    fmt.Println("b")
}
```

并发goroutine和channel

goroutine

goroutine说到底其实就是线程，但是它比线程更小，几百个goroutine可能就是公用底层的五六个线程。执行goroutine只需极少的栈内存(大概是4~5KB)，当然会根据相应的数据伸缩。也正因为如此，可同时运行成千上万个并发任务。goroutine比thread更易用、更高效、更轻便。

```
func say(s string) {
    for i := 0; i < 5; i++ {
        //表示让CPU把时间片让给别人,下次某个时候继续恢复执行该goroutine
        runtime.Gosched()
        fmt.Println(s)
    }
}

func main() {
    go say("world") //开一个新的Goroutines执行
    say("hello") //当前Goroutines执行
```

我们的程序中显式调用 runtime.GOMAXPROCS(n) 可以告诉调度器同时使用几个线程。

并发goroutine和channel

channel

goroutine运行在相同的地址空间，因此访问共享内存必须做好同步，Go提供了一个很好的通信机制channel

1.必须使用make 创建channel：

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface{})
```

2.channel通过操作符<-来接收和发送数据

```
ch <- v    // 发送v到channel ch.
v := <-ch  // 从ch中接收数据，并赋值给v
```

3. 接收和发送数据都是阻塞的，而不需要显式的lock

4. Buffered Channels 缓冲流通道

ch := make(chan type, value)
value == 0 ! 无缓冲（阻塞）
value > 0 ! 缓冲（非阻塞，直到value 个元素）

```
var complete chan int = make(chan int)

func loop() {
    for i := 0; i < 10; i++ {
        fmt.Printf("%d ", i)
    }

    complete <- 0 // 执行完毕了，发个消息
}

func main() {
    go loop()
    <- complete // 直到线程跑完，取到消息. main在此阻塞住
}
```

5. 同样，存在死锁，所有goroutine里的非缓冲信道一定要一个线里存数据，一个线里取数据，要成对才行

http://blog.csdn.net/skh2015java/article/details/60330785?yyue=a21bo.50862.201879

Import和package

import

```
import(
    "fmt"
)

func main() {
    fmt.Println("hello world")
}
```

先去GOROOT目录下加载标准库
再去GOPATH/src中加载第三方库

1.相对路径

```
import  “./model”  //当前文件同一目录的model目录，但是不建议这种方式来import
```

2.绝对路径

```
import  “shorturl/model”  //加载gopath/src/shorturl/model模块
```

1.点操作

```
import(
    . "fmt"
)
```

可以省略前缀的包名，fmt.Println("hello world")可以省略的写成Println("hello world")

2.别名操作

```
import(
    f "fmt"
)
```

别名操作的话调用包函数时前缀变成了我们的前缀，即f.Println("hello world")

3._操作

```
import (
    "database/sql"
    _ "github.com/ziutek/mymysql/godrv"
)
```

_操作其实是引入该包，而不直接使用包里面的函数，而是调用了该包里面的init函数

Import和package

package

package是golang最基本的分发单位和工程管理中依赖关系的体现。

√ 每个golang源代码文件开头都拥有一个package声明，表示该golang代码所属的package。

√ 要生成golang可执行程序，必须建立一个名为**main**的package，并且在该package中必须包含一个名为**main()**的函数。

√ 在golang工程中，同一个路径下只能存在一个package，一个package可以拆成多个源文件组成。

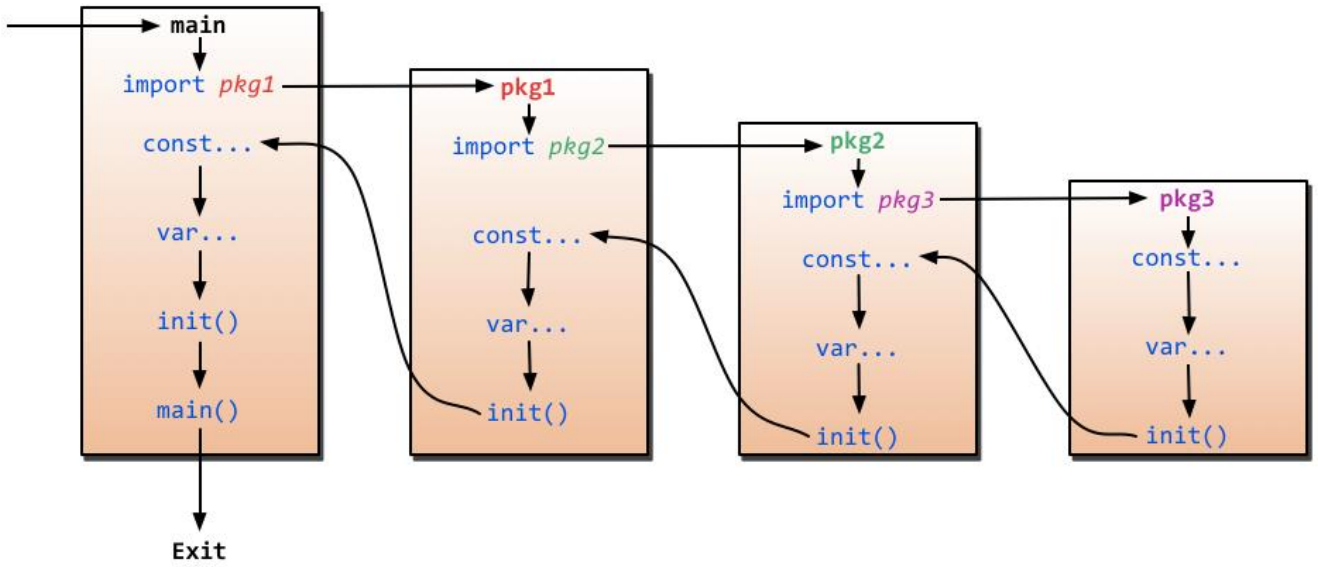
√ import关键字导入的是package路径，而在源文件中使用package时，才需要package名。经常可见的import的目录名和源文件中使用的package名一致容易造成import关键字后即是package名的错觉，真正使用时，这两者可以不同。

main,init,test

Go里面有两个保留的函数：init函数（能够应用于所有的package）和main函数（只能应用于package main）。

Go程序会自动调用init()和main()，所以你不需要在任何地方调用这两个函数。每个package中的init函数都是可选的，但package main就必须包含一个main函数。

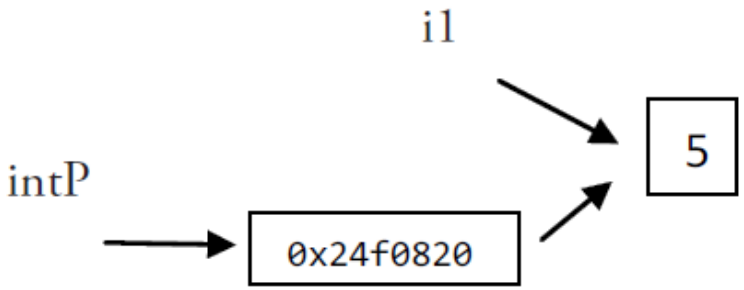
Go程序需要测试文件一律用”_test”结尾，测试的函数都用Test开头，使用命令$ go test 包名测试



指针和内存分配

不像 Java 和 .NET，Go 语言为程序员提供了控制数据结构的指针的能力；但是，不能进行指针运算。

```
var i1 = 5
var intP *int
 intP = &i1
```



常用：

1. struct的指针作为参数
2. struct的指针作为返回值
3. struct的指针作为接收者
4. function的指针作为参数
5. Interface的指针作为参数

在下列情况可以考虑使用指针：

1. 需要改变参数的值；
2. 避免复制操作；
3. 节省内存

指针和内存分配

make、new操作

new用于各种类型的内存分配。make用于内建类型（map、slice 和channel）的内存分配。

内建函数new本质上说跟其它语言中的同名函数功能一样：new(T)分配了零值填充的T类型的内存空间，并且返回其地址，即一个*T类型的值。用Go的术语说，它返回了一个指针，指向新分配的类型T的零值。有一点非常重要：

new返回指针。

内建函数make(T, args)与new(T)有着不同的功能，make只能创建slice、map和channel，并且返回一个有初始值(非零)的T类型，而不是*T。本质来讲，导致这三个类型有所不同的原因是指向数据结构的引用在使用前必须被初始化。例如，一个slice，是一个包含指向数据（内部array）的指针、长度和容量的三项描述符；在这些项目被初始化之前，slice为nil。对于slice、map和channel来说，make初始化了内部的数据结构，填充适当的值。

make返回初始化后的（非零）值。

总结下golang的优缺点

优点有哪些？

缺点有哪些？

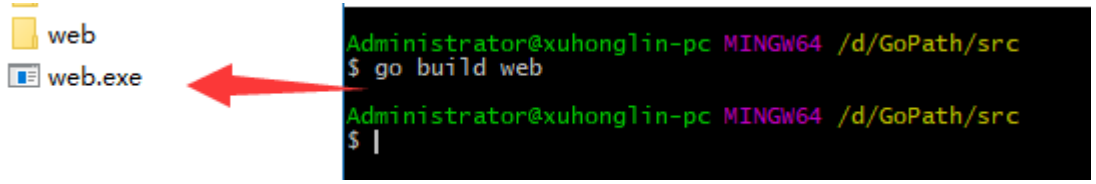
Web小实例

nginx、apache服务器不需要吗？

Go就是不需要这些，因为他直接就监听tcp端口了，然后sayhelloName这个其实就是我们写的逻辑函数了，跟控制层（controller）函数类似。

直接在IDE中运行

或者，编译成可执行文件



在浏览器输入http://localhost:8080
可以看到浏览器页面输出了Hello astaxie!

可以换一个地址试试：
http://localhost:8080/?url_long=111&url_long=222

```go
package main

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

func sayhelloName(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()  //解析参数，默认是不会解析的
    fmt.Println(r.Form)  //这些信息是输出到服务器端的打印信息
        for k, v := range r.Form {
        fmt.Println("key:", k)
        fmt.Println("val:", strings.Join(v, ""))
    }
    fmt.Fprintf(w, "Hello astaxie!") //这个写入到w的是输出到客户端的
}

func main() {
    http.HandleFunc("/", sayhelloName) //设置访问的路由
    err := http.ListenAndServe(":8080", nil) //设置监听的端口
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```