# Go语言学习笔记 1 特色 和 用途

2017-07-29 [BugQ](https://mp.weixin.qq.com/s?__biz=MzIyNTY5ODI2MQ==&tempkey=OTIwX2NheS92T05adTRvVm1ZU2lmcnJzbjFWRWZQTFd6RXRCczY2czZGajVYRHUyTjdxQXZzV1RoMjNOWklrbm5lakpaRUlVcjF0TndhSml1UGF5RnBnYUlZUFRxS01ObUJmd1h3SkxMSVkyRHg4UVU2UnNLYWhPRkFLdDRhZUVSNUI1djlnLWVDYmdRTE1FeDV0ajd4N0RuUnhmMDRVVC1xeXlweEYxN2d%2Bfg%3D%3D&chksm=687a8b765f0d0260a2b07eb61d18cd6a9b6920641c127a550a5345f5183c183b21fdfa2d86fd##)

前言：

学习如逆水行舟，不进则退。

选择一种新的东西开始学习吧，点点滴滴，汇成江河

## Go 语言特色

* 简洁、快速、安全
* 并行、有趣、开源
* 内存管理、v数组安全、编译迅速

## Go 语言用途

Go 语言被设计成一门应用于搭载 Web 服务器，存储集群或类似用途的巨型中央服务器的系统编程语言。

对于高性能分布式系统领域而言，Go 语言无疑比大多数其它语言有着更高的开发效率。它提供了海量并行的支持，这对于游戏服务端的开发而言是再好不过了。

# Go语言学习笔记 2 下载和HelloWorld

2017-07-30 [BugQ](https://mp.weixin.qq.com/s?__biz=MzIyNTY5ODI2MQ==&tempkey=OTIwXzdNd05OaGFRek5mSFJWL2hmcnJzbjFWRWZQTFd6RXRCczY2czZGajVYRHUyTjdxQXZzV1RoMjNOWklsYkxjak5tYjJKSGNsMlgyRGpoeXJtT3U2WDdVMm54em80SGIxR3ZhZ3JaQlNCb2dIV21oSFVINWRXcFFKam1uck02VFd4eEtOdC1FcS1PQURVclhsNEZmM0s1UGtyR3NJbjFNMW1JaW56VkF%2Bfg%3D%3D&chksm=687a8b6a5f0d027cedc1651dcdf47af5aa39075aa47849d98e228dbd2467e701be253b4ef76b##)

1.下载

从 https://www.golangtc.com/download 下载，选择 go1.9rc1.windows-amd64.zip

2.解压

解压到 D:\ws\ 目录里面

3.程序

文件名：helloworld.go

所在目录：D:\ws\gosrc

文件内容：

package main

import "fmt"

func main() {

   fmt.Println("Hello, World!")

}

4.执行

D:\ws\go\bin>go run ..\..\gosrc\helloworld.go

5.结果

Hello, World!

# Go语言学习笔记 3 数据类型和多变量赋值

2017-08-14 [BugQ](https://mp.weixin.qq.com/s?__biz=MzIyNTY5ODI2MQ==&tempkey=OTIwX2hvODFaSkZ5TFB2NWgvQmpmcnJzbjFWRWZQTFd6RXRCczY2czZGajVYRHUyTjdxQXZzV1RoMjNOWkluNWtZODVRcElWVjVzMTB1RWJRRHlEMk1laktIQXlIVXJOeDJMSmQ4QnB5ZUQxWFQ5eVRkamo5LUJnejRVRnlzZENfRnBlUExUNlVNTXN6S295UTQwTndIQjNrXzJZSzBGOWVwd2ZIY0JKMXd%2Bfg%3D%3D&chksm=687a8b5b5f0d024d39f6ae0b30f433661bf091dd1aa56c74f7c0396524087b6ec81102d1f770##)

**数据类型**

数据类型的出现是为了把数据分成所需内存大小不同的数据，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。

当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）；标识符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 protected ）。

声明变量的一般形式是使用 var 关键字：

**多变量声明**

var identifier type

//类型相同多个变量, 非全局变量

var vname1, vname2, vname3 type

vname1, vname2, vname3 = v1, v2, v3

var vname1, vname2, vname3 = v1, v2, v3 //和python很像,不需要显示声明类型，自动推断

vname1, vname2, vname3 := v1, v2, v3 //出现在:=左侧的变量不应该是已经被声明过的，否则会导致编译错误

// 这种因式分解关键字的写法一般用于声明全局变量

var(

    vname1 v\_type1

    vname2 v\_type2

)

# Go语言学习笔记 4 多变量赋值例子

2017-08-14 [BugQ](https://mp.weixin.qq.com/s?__biz=MzIyNTY5ODI2MQ==&tempkey=OTIwXzYxamN3NGlhNnI2WEJiTVdmcnJzbjFWRWZQTFd6RXRCczY2czZGajVYRHUyTjdxQXZzV1RoMjNOWkluc09FZ3FnTEs2V1hwa0dUZ0YxVW0zaXVCWUJxLVhZZWVvTldVUEZiNzhRdFlROFdXajdMSUVNTWoyOTBhcFFYMGF1UU5NVnBhb1FRZ2ZNUlpTcFZNYmFqQ0liaDl5LW9VajJQSFNGaHkxNXd%2Bfg%3D%3D&chksm=687a8b495f0d025f5ed441c944199af05d55a952b0840daa1ebd897c895538ed2b6c1e3f351e##)

程序1

package main

import "fmt"

func main() {

var a, b ,c int

a, b, c = 1, 2 , 3

fmt.Println (a, b, c)

}

结果：

D:\ws\go\bin>set path=d:\ws\go\bin;%path%

D:\ws\gosrc>go run multivar.go

1 2 3

注意：

1. 设定path变量

2.   go 后面的 run 别忘记了

程序2：

package main

import "fmt"

func main() {

var a, b ,c  = 1,"2", 'c'

fmt.Println (a, b, c)

}

结果：

D:\ws\gosrc>go run multivar-2.go

1 2 99

注意： 99是‘c'的ASCII码值

程序3：

package main

import "fmt"

func main() {

 a, b ,c  := 1,"2", 'd'

fmt.Println (a, b, c)

}

结果：

D:\ws\gosrc>go run multivar-3.go

# command-line-arguments

.\multivar-3.go:7:2: undefined: a

.\multivar-3.go:7:5: undefined: b

.\multivar-3.go:7:8: undefined: c

.\multivar-3.go:8:14: undefined: a

.\multivar-3.go:8:17: undefined: b

.\multivar-3.go:8:20: undefined: c

D:\ws\gosrc>go run multivar-3.go

1 2 100

注意：错误原因：忘记去掉var的定义了

程序4：

package main

import "fmt"

func main() {

var (

a int

b int

c string

)

 a, b ,c  = 1,'e', "hello world"

fmt.Println (a, b, c)

}

结果：

D:\ws\gosrc>go run multivar-4.go

# command-line-arguments

.\multivar-4.go:7:5: syntax error: unexpected {, expecting name

.\multivar-4.go:8:4: syntax error: unexpected int at end of statement

.\multivar-4.go:9:4: syntax error: unexpected char at end of statement

.\multivar-4.go:10:4: syntax error: unexpected string at end of statement

.\multivar-4.go:12:2: syntax error: non-declaration statement outside function body

D:\ws\gosrc>go run multivar-4.go

# command-line-arguments

.\multivar-4.go:9:4: undefined: char

D:\ws\gosrc>go run multivar-4.go

1 101 hello world

注意：

1. 把 （写成了 {
2. 使用了char的类型，实际上没有char的类型

# Go语言学习笔记 5 简短形式，使用 := 赋值操作符

2017-08-15 [BugQ](https://mp.weixin.qq.com/s?__biz=MzIyNTY5ODI2MQ==&tempkey=OTIwX1FPMitadWl3aVJFYmRpZnNmcnJzbjFWRWZQTFd6RXRCczY2czZGajVYRHUyTjdxQXZzV1RoMjNOWkluNTlLTklNZGhHODRON1dMNzN1RlRRZ0k3a0hnem5pNFNma3FpYWt5MDJrTF9PbklQVFVDUTdJc2dSSktOU25VaWZRNVlaV0ZrdllfdWRISzM2Zi00dk15NHBiZ0hpUGVWNGllVGZCWDNOOWd%2Bfg%3D%3D&chksm=687a8cbb5f0d05ad9c78284de5a1538b02a7758e74e41eecc5aefe773e0e019b79c83b6e1450##)

## 简短形式，使用 := 赋值操作符

我们知道可以在变量的初始化时省略变量的类型而由系统自动推断，声明语句写上 var 关键字其实是显得有些多余了，因此我们可以将它们简写为 a :=50 或 b := false。

a 和 b 的类型（int 和 bool）将由编译器自动推断。

这是使用变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。使用操作符 :=可以高效地创建一个新的变量，称之为初始化声明。

# Go语言学习笔记 6 变量声明的注意事项

2017-08-19 [BugQ](https://mp.weixin.qq.com/s?__biz=MzIyNTY5ODI2MQ==&tempkey=OTIwX1ZwMVo1blIvU0JNbXJjbFVmcnJzbjFWRWZQTFd6RXRCczY2czZGajVYRHUyTjdxQXZzV1RoMjNOWkltYkJzbWh6TF9ERWYwSW05SkF0MUFXZHJIZ2Z5TTBIeWRwV1IxS1k0NzRiZDV5aUJlcWZlSnJkM3RMeXpsOG9GLVduTTFVRDcySWZpY2NhMVNlcXZQU01LRWhhTS1kN042UUJNS2FxNWJsR1F%2Bfg%3D%3D&chksm=687a8caa5f0d05bc5bfd471412db44e12b8964af4d567bdf767460de002e73cd11e0203216ed##)

声明了一个局部变量却没有在相同的代码块中使用它，同样会得到编译错误

全局变量是允许声明但不使用

空白标识符 \_ 也被用于抛弃值，如值 5 在：\_, b = 5, 7 中被抛弃。

\_ 实际上是一个只写变量，你不能得到它的值。这样做是因为 Go 语言中你必须使用所有被声明的变量，但有时你并不需要使用从一个函数得到的所有返回值。

并行赋值也被用于当一个函数返回多个返回值时，比如这里的 val 和错误 err 是通过调用 Func1 函数同时得到：val, err = Func1(var1)

# Go语言学习笔记7 变量声明的注意事项示例

代码

package main

func main() {

var var1 int

var1 = 1

/\*\*var1 = var1 + 1\*/

}

执行结果

D:\ws\gosrc>go run var1.go

# command-line-arguments

.\var1.go:5:5: var1 declared and not used

注意：

1.光有赋值语句（var1=1）也是不够的

2.把注释的代码放开就可以正常执行了

代码：

package main

import "fmt"

var

(

a int

b int

)

func main() {

fmt.Println ("this is a test")

}

执行结果

D:\ws\gosrc>go run var2.go

this is a test

注意：犯了把变量定义在main函数中的问题，如果放到main函数中的话，和上面的例子一样会出现not used的错误

代码：

package main

import "fmt"

var

(

a int

b int

)

func main() {

\_ , b = 2 , 3

fmt.Println (a, b)

}

执行结果：

D:\ws\gosrc>go run var3.go

0 3

注意：a变量的输出值为0

代码：

package main

import "fmt"

var

(

a int

b int

)

func main() {

a , b = main2()

fmt.Println (a, b)

}

func main2()(int,int) {

return 100,200

}

执行结果：

D:\ws\gosrc>go run var4.go

100 200

注意：go函数的返回值写法竟然是这样的，事前还不知道，上网查了一下才写出这个函数的例子，可以看到两个返回值，a，b变量都有值了

# Go语言学习笔记8 常量

常量是一个简单值的标识符，在程序运行时，不会被修改的量。

常量的定义格式：

const identifier [type] = value

多常量的写法

const c\_name1, c\_name2 = value1, value2

枚举的写法

const (

Unknown = 0

Female = 1

Male = 2

)

特殊常量

iota，特殊常量，可以认为是一个可以被编译器修改的常量。

在每一个const关键字出现时，被重置为0，然后再下一个const出现之前，每出现一次iota，其所代表的数字会自动增加1。

iota 可以被用作枚举值：

第一个 iota 等于 0，每当 iota 在新的一行被使用时，它的值都会自动加 1

# Go语言学习笔记9 常量示例

代码：

package main

import "fmt"

func main() {

const a, b ,c = 1,'e', "hello world"

fmt.Println (a, b, c)

}

运行结果

D:\ws\gosrc>go run const1.go

1 101 hello world

注意

代码

package main

import "fmt"

func main() {

const(

a, b ,c = 1,'e', "hello world"

)

fmt.Println (a, b, c)

}

运行结果

D:\ws\gosrc>go run const2.go

1 101 hello world

注意

从const1.go改到const2.go的时候又忘记去掉括号里面的var的key world了

代码

package main

import "fmt"

func main() {

const (

a = 100

b = 200

c = 300

)

fmt.Println (a, b, c)

}

运行结果

D:\ws\gosrc>go run const3.go

100 200 300

注意

如果不是c=300，而是c的话会输出什么了，可以看下一段代码的注意事项

代码

package main

import "fmt"

func main() {

const (

a = iota

b

c

d = 100

e

f = iota

)

fmt.Println (a, b, c,d,e,f)

const (

a2 = 100

b2

c2 = iota

)

fmt.Println (a2, b2, c2)

const (

i=1<<iota

j=3<<iota

k

l

)

fmt.Println("i=",i)

fmt.Println("j=",j)

fmt.Println("k=",k)

fmt.Println("l=",l)

}

运行结果

D:\ws\gosrc>go run const4.go

0 1 2 100 100 5

100 100 2

i= 1

j= 6

k= 12

l= 24

注意

1. **常量e虽然没有赋值，但是输出值是**100,而不是0，也不是iota对应的值，而是**挨着它的上一个被赋值的常量的value**
2. 感觉就是有一个**隐藏的index为0开始的变量，由系统提供的**，要使用必须需要显示使用iota才好用，如c2输出的值。
3. 不依赖于第一次iota出现的位置，**如c2第一次使用iota，但是输出是2而不是0**
4. 不要写成itoa，是iota，不好记啊，开头记成 index of ？？？，顺序就不会乱？？？
5. 最后一个例子的输出好难理解 ，如下理解吧

1）首先不要弄混 << 操作符的方向

2）然后 因为 k没有缺省值，就和它前一个值一样吧，前一个是3， 那么k也是3，前一个是 3 << iota, 那么k也是 3 << iota, 然后因为 iota存在 +1 的性质，前一个iota是1，那么这个就是 k=3 << 2, j的输出24也同理

3）这个地方 太晦涩，难道上一个定义为 2 + （7\*9）-1 +iota，下一个没赋值的也是一样的吗？？

# Go语言学习笔记10 运算符

只对不熟悉的运算符进行学习了

## 1. 位运算符

位运算符对整数在内存中的二进制位进行操作。

Go 语言支持的位运算符如下表所示。假定 A 为60，B 为13：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| & | 按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。 | (A & B) 结果为 12, 二进制为 0000 1100 |
| | | 按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或 | (A | B) 结果为 61, 二进制为 0011 1101 |
| ^ | 按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1。 | (A ^ B) 结果为 49, 二进制为 0011 0001 |
| << | 左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。 | A << 2 结果为 240 ，二进制为 1111 0000 |
| >> | 右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。 | A >> 2 结果为 15 ，二进制为 0000 1111 |

## 2其他运算符

下表列出了Go语言的其他运算符。

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| & | 返回变量存储地址 | &a; 将给出变量的实际地址。 |
| \* | 指针变量。 | \*a; 是一个指针变量 |

如何生成一个set path的bat文件 2017/8/26

起因：每回执行go程序的时候都需要执行一个set path=d:\ws\go\bin;%path%，很麻烦。想自动生成一个包含这句话的批处理文件，

方法一，命令

echo set path=d:\ws\go\bin;%path% > setpath.bat

结果

set path=d:\ws\go\bin;C:\apache-maven-2.2.1\bin;D:\app\k\product\11.2.0\dbhome\_1\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Static

不是想要的结果，想要的结果为 set path=d:\ws\go\bin;%path%

方法二，命令

D:\ws\gosrc>set a=%

D:\ws\gosrc>echo set path=d:\ws\go\bin;%a%path%a% > setpath.bat

结果：

set path=d:\ws\go\bin;%path%

是想要的结果，但是执行了两个单独的命令

方法三，命令

D:\ws\gosrc>set a=% & echo set path=d:\ws\go\bin;%a%path%a% > setpath.bat

结果：

set path=d:\ws\go\bin;%aC:\apache-maven-2.2.1\bin;D:\app\k\product\11.2.0\dbhome\_1\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Statica%

不是正确的结果

在执行一遍

D:\ws\gosrc>set a=% & echo set path=d:\ws\go\bin;%a%path%a% > setpath.bat

结果

set path=d:\ws\go\bin;% path%

好像正确，其实在第一个%后面多了一个空格

调查原因，是set a=% & 的% 和 &之间有一个空格

方法四，解决空格问题，定义放在后面

执行D:\ws\gosrc>echo set path=d:\ws\go\bin;%a%path%a% > setpath.bat & set a=%

结果：

set path=d:\ws\go\bin;%aC:\apache-maven-2.2.1\bin;D:\app\k\product\11.2.0\dbhome\_1\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Statica%

不正确

执行第二遍

执行D:\ws\gosrc>echo set path=d:\ws\go\bin;%a%path%a% > setpath.bat & set a=%

结果：

set path=d:\ws\go\bin;%path%

正确

但是执行2遍

本想放弃，执行两遍就两遍吧，但是突然想到是否可以使用for来放在一行，就再试试

方法五

执行

for /l %i in (1,1,2) do @echo set path=d:\ws\go\bin;%a%path%a% > setpath.bat & set a=%

相当于再方法四的基础上执行两次，是否OK

结果

set path=d:\ws\go\bin;%aC:\apache-maven-2.2.1\bin;D:\app\k\product\11.2.0\dbhome\_1\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Statica%

不正确，

再次执行

for /l %i in (1,1,2) do @echo set path=d:\ws\go\bin;%a%path%a% > setpath.bat & set a=%

结果正确

set path=d:\ws\go\bin;%path%

和方法四一样，最后虽然多个空格，但是不影响bat文件的执行

也是执行2次，看来是两次循环属于程序内部的变量赋值，没有赋值到系统变量上面，所以不行

方法六

想到for是否可以循环字符串，简单查了帮助，

命令：for /F %i in ("%") do @echo set path=d:\ws\go\bin;%ipath%i > setpath.bat

结果：set path=d:\ws\go\bin;%path%

**OK。**

其他注意点：

1.每次实验前执行 set a=

2.set a=hello 和 set a = hello是不一样的结果，用set a可以看到两个，一个是a 一个是a后面有个空格

3.网上查的for的语法，为%%两个百分号，应该是在bat文件中的写法。直接在cmd中用一个%就可以了

4.保存在文件中的最终形式for /F %%i in ("%%") do @echo set path=d:\ws\go\bin;%%ipath%%i > setpath.bat，("%%")也是两个%%啊

# Go语言学习笔记11 运算符示例

程序：

package main

import "fmt"

func main() {

var aa uint = 10011001

var bb uint = 01100110

var cc uint

cc = aa & bb

fmt.Println("c = %d", cc)

}

结果：

D:\ws\gosrc>go run bit.go

c = %d 32840

注意：

奇怪，不对啊，原来是fmt.Println不对，应该为Printf才对

程序2：

package main

import "fmt"

func main() {

var aa int = 10011001

var bb int = 01100110

var cc int

cc = aa & bb

fmt.Printf("c = %d", cc)

}

结果：

D:\ws\gosrc>go run bit2.go

c = 32840

程序3：

package main

import "fmt"

func main() {

var aa int = 10011001

var bb int = 01100110

var cc int

cc = aa & bb

fmt.Printf("c 等于 %d", cc)

}

结果：

D:\ws\gosrc>go run bit3.go

# command-line-arguments

.\bit3.go:11:19: invalid UTF-8 encoding

注意：

什么情况，输出有中文的情况下，文件编码为ANSI不可以啊，改成UTF-8就没问题了

程序4：

package main

import "fmt"

func main() {

var aa int = 10011001

var bb int = 01100110

var cc int

cc = aa & bb

fmt.Printf("c 等于 %d\n", cc)

cc = aa | bb

fmt.Printf("c 等于 %d\n", cc)

cc = aa ^ bb

fmt.Printf("c 等于 %d\n", cc)

cc = aa <<1

fmt.Printf("c 等于 %d\n", cc)

cc = aa >>1

fmt.Printf("c 等于 %d\n", cc)

}

结果：

D:\ws\gosrc>go run bit4.go

c 等于 32840

c 等于 10273145

c 等于 10240305

c 等于 20022002

c 等于 5005500

注意：

以为 aa | bb 和 aa ^ bb 应该为相同值，怎么不一样了

因为aa和bb是int形式啊，而不是二进制形式啊，虽然看起来很像

程序5：

package main

import "fmt"

func main() {

a , b :=100,200

var c \* int

c = &a

fmt.Printf("a , b ,c %T, %T, %T\n", a , &b ,c)

fmt.Printf("a , b ,c %d, %d, %d\n", &a , &b ,c)

fmt.Printf("a , b ,c %d, %d, %d\n", a , b ,\*c)

}

结果：

D:\ws\gosrc>go run oper.go

a , b ,c int, \*int, \*int

a , b ,c 825741074600, 825741074624, 825741074600

a , b ,c 100, 200, 100

注意：

%T输出类型，&b输出的是\*int

\*的符号的定义写法 var 然后是变量名，然后是\*和类型

:=的赋值方法不能前面有var

&a和c的地址是一样的，value也是一样的

# Go语言学习笔记12 条件语句

If语句 – if关键字后面的条件竟然没有括号

当然了 if 嵌套里面 的if 条件也没有括号了

Case语句

Go 编程语言中 switch 语句的语法如下：

switch var {

case val1:

...

case val2:

...

default:

...

}

符号为{}和：

不需要break

Var 为任何类型

Var1 和var2 必须类型相同

可逗号分割

Switch后面可以不指定var

## Type Switch

switch 语句还可以被用于 type-switch 来判断某个 interface 变量中实际存储的变量类型。

Type Switch 语法格式如下：

switch x.(type){

case type:

statement(s);

case type:

statement(s);

/\* 你可以定义任意个数的case \*/

default: /\* 可选 \*/

statement(s);

}

注意：Go自己的Type switch

# Go语言学习笔记13 条件语句示例

代码：

package main

import "fmt"

func main() {

a,b:=101,200

if a==100 {

fmt.Println("is 100")

} else {

fmt.Println("is not 100")

if b == 200 {

fmt.Println("is 200")

} else {

fmt.Println("is not 200")

}

}

}

结果：

D:\ws\gosrc>go run if.go

is not 100

is 200

注意：

If后面的 {必须要有，如果是{在下一行开始，则出错

如：

package main

import "fmt"

func main() {

a,b:=101,200

if a==100

{

fmt.Println("is 100")

} else {

fmt.Println("is not 100")

if b == 200 {

fmt.Println("is 200")

} else {

fmt.Println("is not 200")

}

}

}

结果：

D:\ws\gosrc>go run if.go

# command-line-arguments

.\if.go:8:13: syntax error: unexpected newline, expecting { after if clause

Else前面的 }必须要有，如果}在上一行结束，也出错

如：

package main

import "fmt"

func main() {

a,b:=101,200

if a==100 {

fmt.Println("is 100")

} else {

fmt.Println("is not 100")

if b == 200 {

fmt.Println("is 200")

}

else {

fmt.Println("is not 200")

}

}

}

结果：

D:\ws\gosrc>go run if.go

# command-line-arguments

.\if.go:17:3: syntax error: unexpected else, expecting }

.\if.go:20:2: syntax error: non-declaration statement outside function body

**上面这两个，是不是有点变态？**

代码：

package main

import "fmt"

func main() {

//break确认，输出101

a,b:=101,200

switch a {

case 100:

fmt.Println("is 100")

case 101:

fmt.Println("is 101")

default:

fmt.Println("default")

}

// default确认，输出default

switch b {

case 100:

fmt.Println("is 100")

case 101:

fmt.Println("is 101")

default:

fmt.Println("default")

}

// 表达式确认， 输出101， 如果是 case 100 和 case a==101的话，会有错误，一个是int，一个是boolean，case的判断值不一致

switch {

case a==100:

fmt.Println("is 100")

case a==101:

fmt.Println("is 101")

default:

fmt.Println("default")

}

//确认计算表达式，输出200

switch {

case a==100:

fmt.Println("is 100")

case a==100 || b == 200:

fmt.Println("is 200")

default:

fmt.Println("default")

}

// 确认多值的逗号，输出200

switch {

case a==100:

fmt.Println("is 100")

case a==100 ,b == 200:

fmt.Println("is 200")

default:

fmt.Println("default")

}

}

结果：

D:\ws\gosrc>go run switch1.go

is 101

default

is 101

is 200

is 200

注意，如代码中红字部分

代码：

package main

import "fmt"

func main() {

var x interface{}

test(x)

a:=100

x = a

test(x)

b:="test"

x=b

test(x)

}

func test (b interface{}) ( ) {

switch b.(type) {

case nil:

fmt.Println("is nil")

case int:

fmt.Println("is int")

case func(int):

fmt.Println("is func int")

case bool,string:

fmt.Println("is bool or string")

default:

fmt.Println("default")

}

}

结果：

D:\ws\gosrc>go run switch2.go

is nil

is int

is bool or string

注意：

1.函数的写法，两个括号func test (b interface{}) ( ) {

2. b.(type)只能用在switch中

3. interface{}的理解，一个空接口，可以转换成任何类型，类似与cast，但是语法不需要cast

4. case func(int): 还不太理解，如何和inteface{}关联，也许以后接触到interface{}就会知道了

# Go语言学习笔记14 Select语句

select是Go中的一个控制结构，类似于用于通信的switch语句。每个case必须是一个通信操作，要么是发送要么是接收。

select随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。一个默认的子句应该总是可运行的。

### 语法

Go 编程语言中 select 语句的语法如下：

select {

case communication clause :

statement(s);

case communication clause :

statement(s);

/\* 你可以定义任意数量的 case \*/

default : /\* 可选 \*/

statement(s);

}

以下描述了 select 语句的语法：

* 每个case都必须是一个通信
* 所有channel表达式都会被求值
* 所有被发送的表达式都会被求值
* 如果任意某个通信可以进行，它就执行；其他被忽略。
* 如果有多个case都可以运行，Select会随机公平地选出一个执行。其他不会执行。   
  否则：
  1. 如果有default子句，则执行该语句。
  2. 如果没有default字句，select将阻塞，直到某个通信可以运行；Go不会重新对channel或值进行求值。

### 实例

package main

import "fmt"

func main() {

var c1, c2, c3 chan int

var i1, i2 int

select {

case i1 = <-c1:

fmt.Printf("received ", i1, " from c1\n")

case c2 <- i2:

fmt.Printf("sent ", i2, " to c2\n")

case i3, ok := (<-c3): // same as: i3, ok := <-c3

if ok {

fmt.Printf("received ", i3, " from c3\n")

} else {

fmt.Printf("c3 is closed\n")

}

default:

fmt.Printf("no communication\n")

}

}

以上代码执行结果为：

D:\ws\gosrc>go run select.go

no communication

理解注意点：

1.语法和switch一样，比较好记

2.还没有涉及到chan的概念和对应的usage，但是从代码中看为 chan，和 <-，而且<-的左右两边的类型决定了 receive 还是 send的意思

3.个人感觉，应该开始接触到了Go语言的自己的语言优势的地方了，关于上面的语法要仔细理解

# Go语言学习笔记15 循环语句

For的循环条件不需要括号

三种形式

和 C 语言的 for 一样：

for init; condition; post { }

和 C 的 while 一样：

for condition { }

和 C 的 for(;;) 一样：

for { }

for 循环的 range 格式可以对 slice、map、数组、字符串等进行迭代循环。格式如下：

for key, value := range oldMap {

newMap[key] = value

}

这个是新特征，不同于java啊

代码：

package main

import "fmt"

func main() {

index:=1

for index<10 {

fmt.Print("hello ")

index+=1

}

var a [10]int

a=[10]int{1,2,3,4,5}

fmt.Print("\n")

for i,x:= range a {

fmt.Print(i, x,"\n")

}

}

结果：

D:\ws\gosrc>go run for.go

hello hello hello hello hello hello hello hello hello

0 1

1 2

2 3

3 4

4 5

5 0

6 0

7 0

8 0

9 0

注意：

一段小程序，写的时候错误point有4

1. for index<10 { 最开始用的是 for 1<10，直接导致无限循环
2. range的定义，虽然还没接触到。var a [10]int 和 a=[10]int{1,2,3,4,5}， 定义时候不能为var a[] int, 必须指定长度，和赋值时候的[10]一致
3. for i,x:= range a {的写法应该是固定的，必须有range，
4. 既然有i,x还必须都得使用，不使用还不可以
5. 这里同样是，如果在for 后面同行上面没有 { 符号的话，要报错的

# Go语言学习笔记16 break，continue，和goto

Break，continue和java基本一致，

Goto也是可以使用，但是不推荐

示例：

package main

import "fmt"

func main() {

index:=1

for index<10 {

fmt.Print("hello ")

index+=1

break

}

var a [10]int

a=[10]int{1,2,3,4,5}

fmt.Print("\n")

for i,x:= range a {

fmt.Print(i, x,"\n")

continue

fmt.Print(i, x,"\n")

}

gotolabel:

for i,x:= range a {

fmt.Print(i, x,"\n")

goto gotolabel

fmt.Print(i, x,"\n")

}

fmt.Println("world")

}

结果：

D:\ws\gosrc>go run goto.go

hello

0 1

1 2

2 3

3 4

4 5

5 0

6 0

7 0

8 0

9 0

0 1

0 1

0 1

注意：

不要把gotolabel: 写成了:gotolabel

# Go语言学习笔记17 函数

终于到函数了，重要的一环

## 函数定义

Go 语言函数定义格式如下：

func function\_name( [parameter list] ) [return\_types] {

函数体

}

第一个惊讶点，返回类型在后面，并且是不需要括号的

## 函数返回多个值

Go 函数可以返回多个值，例如：

func swap(x, y string) (string, string) {

return y, x

}

第二点，返回多个值时候，需要括号

## 传递方式，

缺省值传递，和java一样，不改变传入参数值

还有引用传递，和c一样，定义为 \*类型，传递为 &变量，赋值为 \*变量，那么传入的参数值也随之改变了

示例：

package main

import "fmt"

func main() {

a,b:=100,200

order(&a,&b)

fmt.Println(a, " ", b)

c,d:=400,300

order(&c,&d)

fmt.Println(c, " ", d)

}

func order(a ,b \*int) {

if \*a<\*b {

// do nothing

} else {

t:=\*a

\*b=\*a

\*a=t

}

}

结果：

D:\ws\gosrc>go run fun1.go

100 200

400 400

什么情况，没有交换过来，冷静！！！分析！！！

估计是func定义的地方，在试试，

func order(a ,b \*int) { 改成func order(a \*int,b \*int) { 结果没变，还是不对，

只好加入 fmt的print了，发现问题了

原来是 t:=\*a 应该为 t:=\*b才对，修改后再执行

结果：OK

D:\ws\gosrc>go run fun1.go

100 200

300 400

# Go语言学习笔记18 函数用法

估计都是需要好好掌握的地方

| **函数用法** | **描述** |
| --- | --- |
| [函数作为值](http://www.runoob.com/go/go-function-as-values.html) | 函数定义后可作为值来使用 |
| [闭包](http://www.runoob.com/go/go-function-closures.html) | 闭包是匿名函数，可在动态编程中使用 |
| [方法](http://www.runoob.com/go/go-method.html) | 方法就是一个包含了接受者的函数 |

# Go语言学习笔记19 函数用法 值示例

作为值示例，

用途，初始化变量，相当于一小段代码，如同一个命名函数，和闭包里面的返回值的匿名函数形成对比

示例：

package main

import "fmt"

func main() {

//初始化

a:= func()(int,int) {

return 200,100

}

fmt.Println(a, " ")

// 计算逻辑

c,d := 100,200

e:= func(x,y int)(int) {

return x+y

}

fmt.Println(c,d,e)

// 交换

f,g := 100,200

i:= func(x,y int)(int,int) {

return y,x

}

fmt.Println(f,g,i)

fmt.Println(a())

fmt.Println(e(c,d))

fmt.Println(i(f,g))

}

结果：

D:\ws\gosrc>go run fun2.go

0x4966f0

100 200 0x496710

100 200 0x496730

200 100

300

200 100

注意：

第一次没理解，打印出了0x496710是什么鬼，后来理解了是函数的地址，和c里面类似啊

a,e,i都不是一般意义上的变量，而是代表func的一个变量，后面加（）和参数的

# Go语言学习笔记20 函数用法闭包示例

代码：

package main

import "fmt"

func main() {

//定义

increment:= func() ( func () (int) ) {

var i int =0

return func () int {

i = i +1

return i

}

}

inc := increment()

var count int = 0

count = inc()

count = inc()

fmt.Println("count = ", count);

}

结果：

D:\ws\gosrc>go run fun3.go

count = 2

注意：

increment:= func() ( func () (int) ) { 的 ( func () (int) ) { 的黄色背景有没有括号都可以

不要忘记了return 语句 return func () int {

里面是一个函数，所以需要 func() int {}的

其他理解：

和值的作用一样，返回值里面的函数相当于匿名函数，但是inc := increment()指定的是否，随便可以给它个名字啊。

另外一个作用，匿名函数里面可以定义变量，相当于mixin的方式（包含结构和行为）

对应java，increment()相当于返回一个class的实例？但是这个实例只能处理一件事

# Go语言学习笔记21 函数用法方法

Go 语言中同时有函数和方法。一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。所有给定类型的方法属于该类型的方法集。语法格式如下：

func (variable\_name variable\_data\_type) function\_name() [return\_type]{

/\* 函数体\*/

}

这段话好难理解，用示例来看

package main

import "fmt"

//定义

type person struct {

age int

}

func main() {

var xiaohong person

xiaohong.age = 99

age\_add := xiaohong.age\_add()

fmt.Println("age\_add = ", age\_add);

var count int = 0

count = age\_add()

count = age\_add()

fmt.Println("count = ", count);

for i:=0;i<3;i++ {

count = age\_add()

}

fmt.Println("count = ", count);

}

func (oneperson person) age\_add() (func () (int)) {

var i int =0

return func () int {

i ++

return oneperson.age + i

}

}

结果：

D:\ws\gosrc>go run fun4.go

age\_add = 0x496520

count = 101

count = 104

注意：

一般来说都是对go语言不熟练的场合，或者说第一次写go语言的代码是否会出现的问题。都是一些基本问题，写熟练了或者是多写几遍，或者说有了IDE以后应该就不会有类似的问题了

1. 这个形式的func不能指定一个函数名 = func 。。。的形式了。所以现在只知道需要单独起一个定义在main外面，不知道是不是必须的？

2. type的定义也只能定义在func的外面，如同global一样，因为main 和func都用它了

3.支持闭包的特性

4.验证了接受者可以是命名类型或者结构体类型的一个值或者是一个指针。这句话，如果指定一个int类型，会有提示错误的

5.写了一个for循环，第一次用到了分号（；）啊

# Go语言学习笔记22 语言作用变量域

作用域为已声明标识符所表示的常量、类型、变量、函数或包在源代码中的作用范围。

Go 语言中变量可以在三个地方声明：

* 函数内定义的变量称为局部变量
* 函数外定义的变量称为全局变量
* 函数定义中的变量称为形式参数

形式参数会作为函数的局部变量来使用

Go 语言程序中全局变量与局部变量名称可以相同，但是函数内的局部变量会被优先考虑

没有什么特殊的地方

# Go语言学习笔记23 数组

Index:

index从0开始

声明：

使用[size]声明 variant [size] type

初始化

可以使用大括号 variant [2] int {100,200}

也可以用 variant […] int {100,200}

赋值：

Variant[index] = value

访问

Type variang = Variant[index]

这个地方有点奇怪？？ Type怎么在前面？

示例：

package main

import "fmt"

func main() {

name := [...] string {"Han"," " ,"John"}

fmt.Println("name[0]=",name[0])

fmt.Println("name[2]=",name[2])

name[1] = " Peng "

fmt.Println("name[1]=",name[1])

temp := name[1]

fmt.Println("name[1]=",temp)

var temp2 string

temp2 = name[1]

fmt.Println("name[1]=",temp2)

}

结果：

D:\ws\gosrc>go run array1.go

name[0]= Han

name[2]= John

name[1]= Peng

name[1]= Peng

name[1]= Peng

注意：

Type variang = Variant[index]

这样的写的话，在示例里面compile会出错的，应该用go语言的赋值语句的语法才可以啊。网上教程也是存在错误的

# Go语言学习笔记24 多维数组和数组作为函数参数

多维数组没啥好说的 [][]的形式

作为函数参数，如下：

func( aArray [] int) 或者 func(aArray[size] int)

示例：

package main

import "fmt"

func main() {

count := [...] int {100,200}

sum:=sum(count)

fmt.Println("sum=",sum)

count2 := [] int {100,200}

sum2:=sum2(count2)

fmt.Println("sum2=",sum2)

}

func sum(aArray[2] int) int {

return aArray[0] + aArray[1]

}

func sum2(aArray[] int) int {

return aArray[0] + aArray[1]

}

结果：

D:\ws\gosrc>go run array2.go

sum= 300

sum2= 300

说明：

count := [...] int {100,200} 为固定长度，所以对应函数参数数组定义也必须为固定长度func sum(aArray[2] int) int {

count2 := [] int {100,200}为未指定长度，虽然实际长度为2，但是对应的函数参数数组定义为[],func sum2(aArray[] int) int {

以上定义如果不匹配，compile则会出错

**总结：纸上得来终觉浅，须知此事要躬行**

**追加，也许在IDE里面，编写代码的时候会自动出现提示错误吧**

# Go语言学习笔记25 指针1 基础

和c的指针一样，定义 var variant \*variant-type

定义和打印示例

package main

import "fmt"

func main() {

var a \*int

var b \*float32

fmt.Printf("%x\n", a)

fmt.Printf("%x\n", b)

c:=1

a=&c

var d float32 = 10.0

b = &d

fmt.Printf("%x\n", a)

fmt.Printf("%x\n", b)

}

结果：

D:\ws\gosrc>go run point1.go

0

0

c04204a0a0

c04204a0a8

注意

x输出 指针地址，x代表未知数，未知数代表指针，不被打印出来就不确定的意思，这样好记吗

指针未指定时候，输出为0，指定后输出为地址，但是指定的方式，是需要定义其他的变量，然后把变量的地址赋给指针的变量

Float32的类型的赋值，定义的写法带赋值现在会其他写法不会

空指针

**nil**

指针数组

定义 var name[size] \*type

示例：

package main

import "fmt"

func main() {

var list [3] \*int

color, big,small :=0,1,2

for i:=0;i<=2;i++ {

switch i {

case 0: list[i] = &color

case 1:list[i] = &big

case 2:list[i] = &small

}

}

fmt.Printf("%d\n", \*list[0])

fmt.Printf("%d\n", \*list[1])

fmt.Printf("%d\n", \*list[2])

}

结果：

D:\ws\gosrc>go run point2.go

0

1

2

注意：

这个也是一样，需要把地址给指针数组里面的每个指针元素变量啊

Float示例：

package main

import "fmt"

func main() {

var test float32

test = float32(1)

fmt.Printf("%d\n", test)

fmt.Println(test)

}

结果：

D:\ws\gosrc>go run float.go

%!d(float32=1)

1

注意：

原来可以强制转型test = float32(1)

还有输出的%!d(float32=1)是什么鬼

# Go语言学习笔记26 指针2 指针的指针

定义：var ptr \*\*int;

没啥说的，学过c的都知道咋回事

来个示例把，指针的指针的数组

package main

import "fmt"

func main() {

var names [2] \*\*int

var value = []int{100,200}

for i:=0;i < 2;i ++ {

var temp \*int

temp = &value[i]

names[i] = &temp

}

fmt.Printf("%d\n", \*\*names[0])

fmt.Printf("%d\n", \*\*names[1])

}

结果：

D:\ws\gosrc>go run point3.go

100

200

注意：

数组定义

输出需要两个\*\*

赋值两次&&

赋值的另外一种写法，结果不变

package main

import "fmt"

func main() {

var names [2] \*\*int

var value = []int{100,200}

for i:=0;i < 2;i ++ {

temp:=&value[i]

names[i] = &temp

}

fmt.Printf("%d\n", \*\*names[0])

fmt.Printf("%d\n", \*\*names[1])

}

# Go语言学习笔记27 指针3 指针和函数的组合

指针和函数结合在一起，就复杂了，和c类似，增加复杂度啊，学习一下吧，java里面没有

示例

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

var a int = 100

var b int= 200

var c int= 400

fmt.Printf("交换前 a 的值 : %d\n", a )

fmt.Printf("交换前 b 的值 : %d\n", b )

/\* 调用函数用于交换值

\* &a 指向 a 变量的地址

\* b 指向 b 变量的地址

\* c 指向 c 变量的地址地址

\*/

addressofc := &c

fmt.Printf("交换前 b 的地址 : %x\n", &b )

fmt.Printf("交换前 c 的值 : %d\n",c )

swap(&a, b, &addressofc);

fmt.Printf("交换后 a 的值 : %d\n", a )

fmt.Printf("交换后 b 的值 : %d\n", b )

fmt.Printf("交换后 c 的值 : %d\n", c )

fmt.Printf("交换后 b 的地址 : %x\n", &b )

}

func swap(x \*int, y int, z \*\*int) {

y = 300

\*x = y

fmt.Printf("交换中 b 的地址 : %x\n", &y )

\*\*z = \*x

}

结果：

D:\ws\gosrc>go run point5.go

交换前 a 的值 : 100

交换前 b 的值 : 200

交换前 b 的地址 : c04200e0a8

交换前 c 的值 : 400

交换中 b 的地址 : c04200e0e8

交换后 a 的值 : 300

交换后 b 的值 : 200

交换后 c 的值 : 300

交换后 b 的地址 : c04200e0a8

说明：

增加了一个c为 \*\*,b改成了值，可以看到b的地址只在func中改变，return 后不变

也可以看到 c的\*\*和\*和value的赋值的传递方法

联想，如果函数参数上面再有指针数组，二维数组，二维指针数组，指向func的指针，结构的指针，各种组合得多么复杂啊

# Go语言学习笔记28 IDE的学习

需要用IDE了，用了one month的Editplus，应该升级了吧？？

下载：

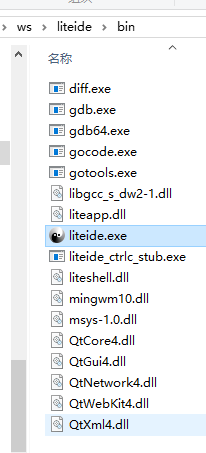
<https://sourceforge.net/projects/liteide/files/>

下载版本：

[**Download liteidex32.2.windows-qt4.zip (28.9 MB)**](https://sourceforge.net/projects/liteide/files/latest/download?source=files)

**文件：liteidex32.2.windows-qt4.zip**

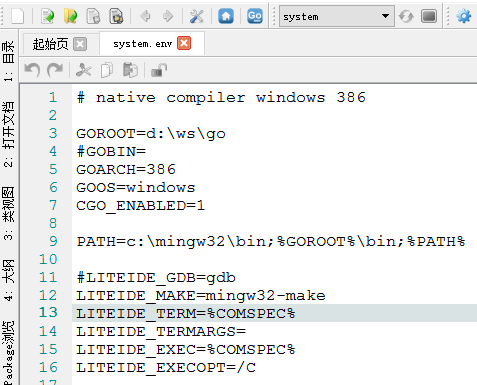
**解压，执行liteide.exe 启动**



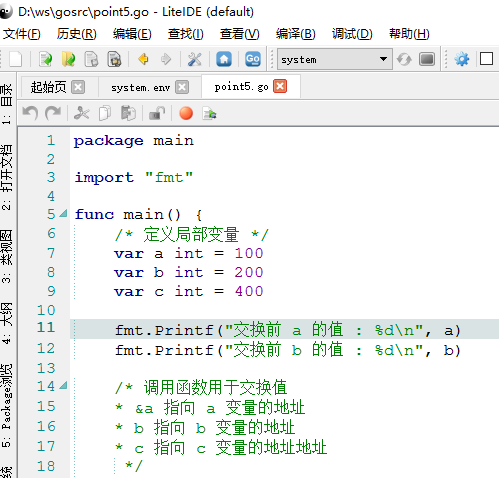
**如图**



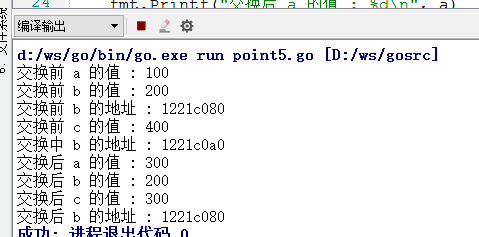
**环境设定，否则不能够运行，点击任务栏上的黑色小图标，如下修改**



**打开一个文件，点击FR图标，或者按Alt+F6，ALT+SHIFT+R**



**运行结果**

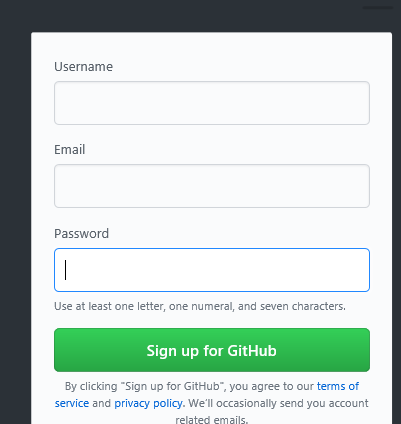


现在只是简单的能够编辑文件了，有语法颜色表示，和能够执行，以后再有更多的使用方法的了解，继续追加。

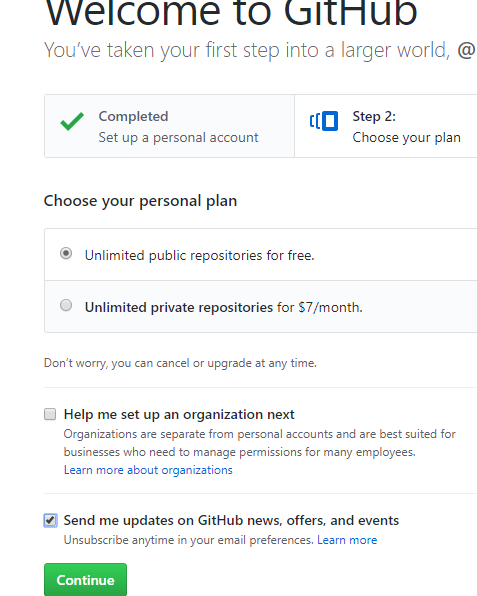
# Go语言学习笔记29 将Go语言学习的代码转移到git的方法

跟踪当前的潮流，决定将代码保存到git，需要研究一下

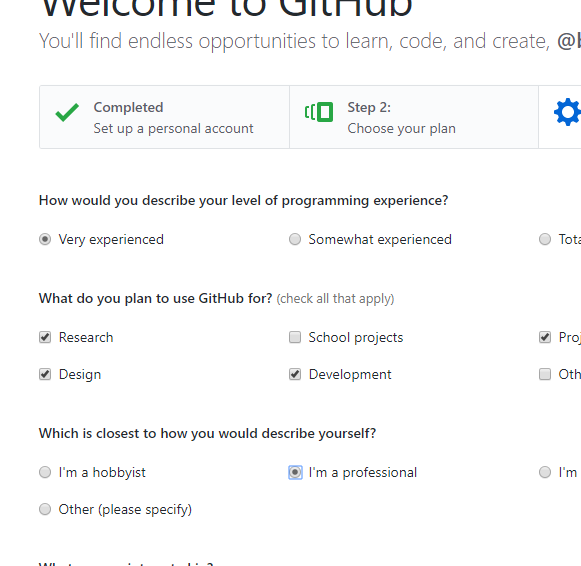
登录github，<https://github.com/>，输入内容



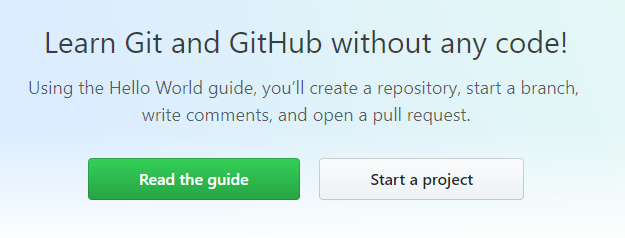
下一步，选着unlimited



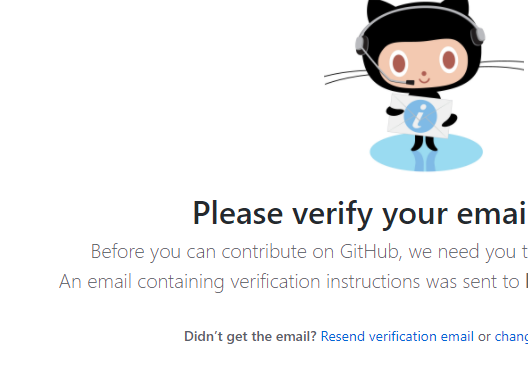
下一步continue

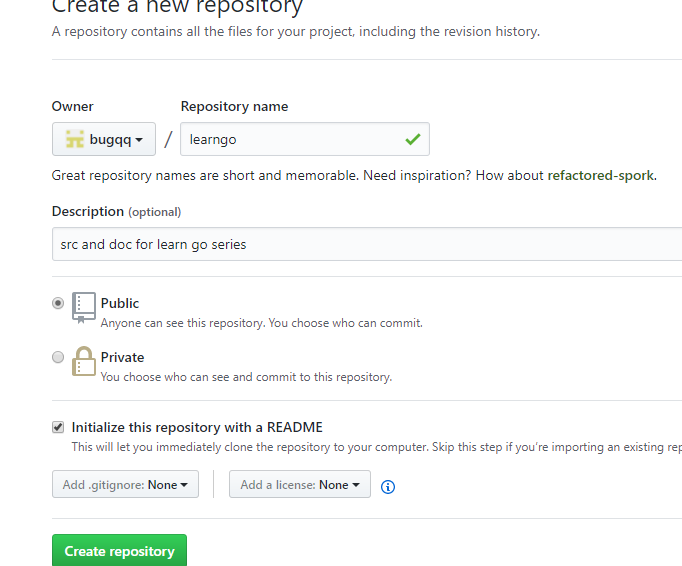


选择自己合适的选项，然后下一步

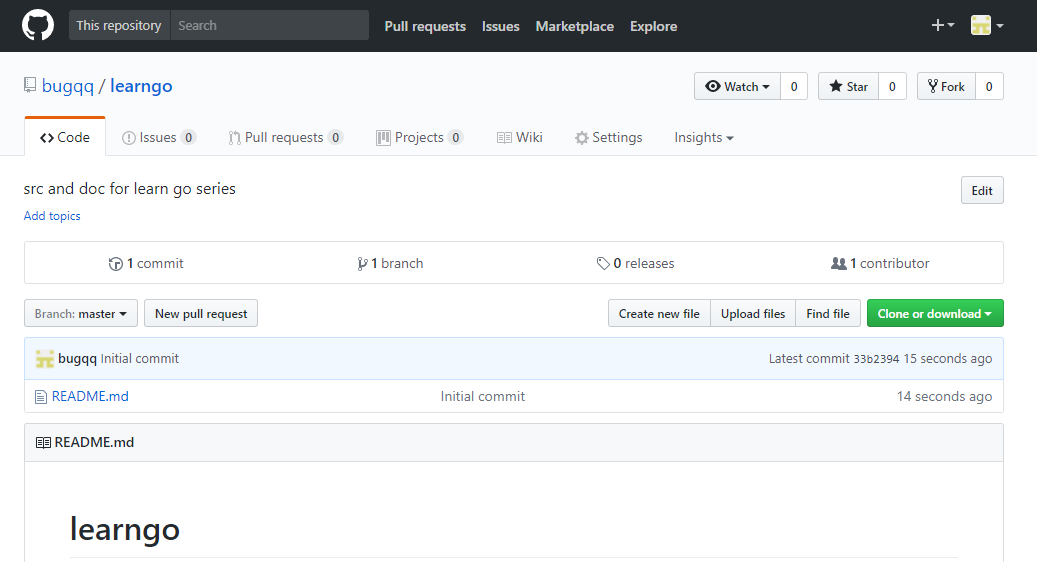


点击，start a project，



需要验证mail，到mail box里面找到mail，点击里面的link，进行验证， 

然后创建repository name，这里四learngo，选择public，只能是public，因为是free的



到这里可以看到learngo已经在上面了，之前选择了需要一个README.md文件，所以会有一个README.md文件在这里。

# Go语言学习笔记30 将Go语言学习的代码转移到git的方法之二 git的client端

Github已经设定完成了，需要传code上去啊，虽然也可以从http的网页github传，但是一般都要client端，选择熟悉的sourcetree吧

下载：

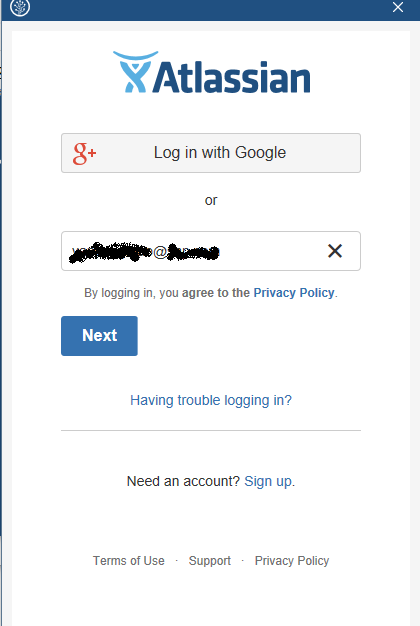
<http://sourcetreeapp.com/>

文件：SourceTreeSetup-2.1.11.0.exe

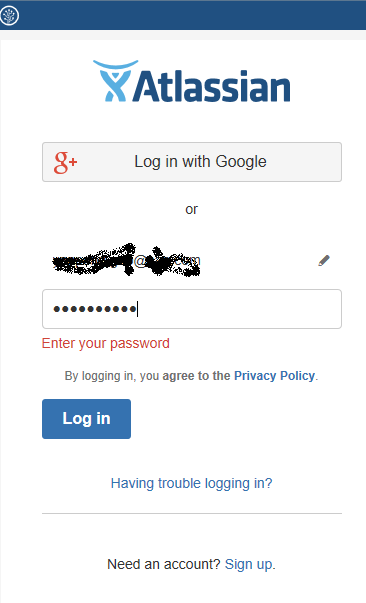
安装开始



同意协议，下一步



弹出登录框，输入mail



输入密码，点击Log in

这个地方应该也会很慢，需要等待啊

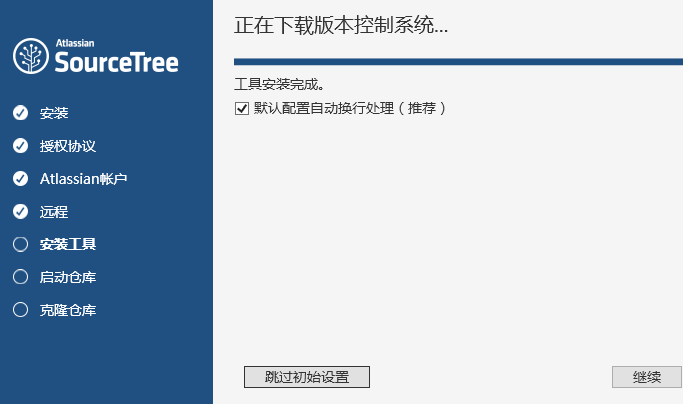


缺省为github啊，就继续

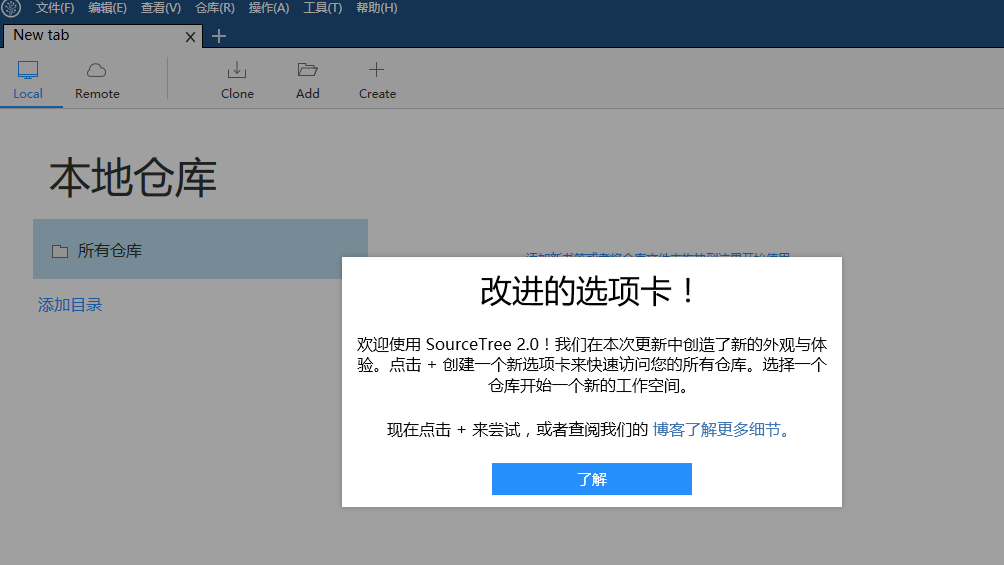
这中间打开浏览器，让输入github的username和password，输入后，可以和github绑定在一起，在最后的图可以看到



等待，需要一段时间，然后继续



安装完成，然后继续



可以看到，启动了



点击remote，可以看到和github的bugqq已经连接在一起了

# Go语言学习笔记31 将Go语言学习的代码转移到git的方法之三 source上传的简单说明

又开始用git了，真好

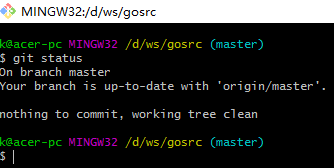
从sourcetree的页面选择remote，因为已经和github关联了，点击右边的learngo的clone链接



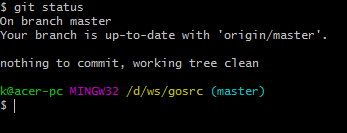
就会从github的remote repository上clone出来learngo到本地，然后打开terminal



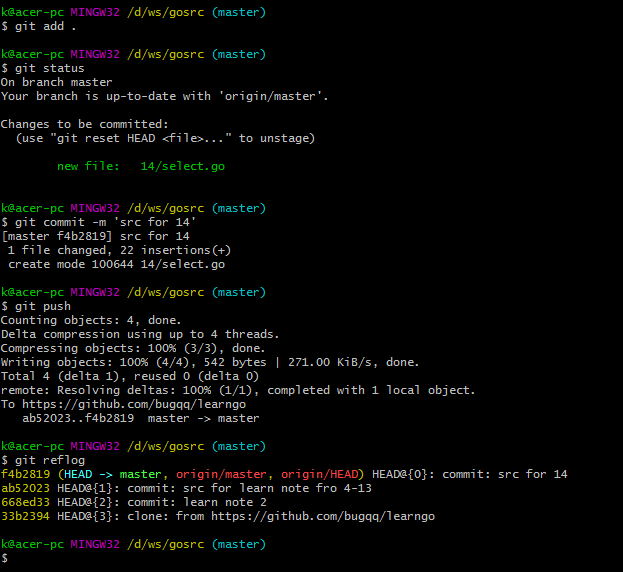
如图



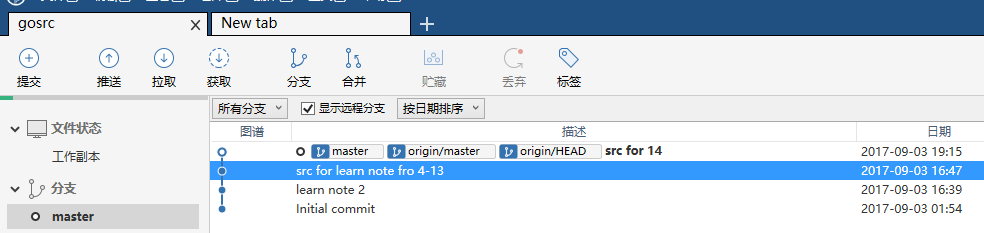
可以用git status看



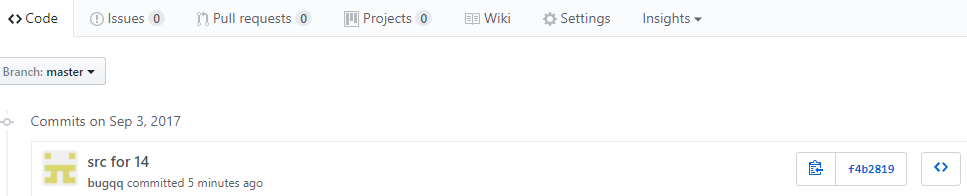
传一个文件，如14目录下面的文件



使用了git add， git commit ，git push， git reflog



点击拉取，sourcetree的显示会刷新



然后看看github的web页面，看commit里面，也能看到了

# Go语言学习笔记32 将Go语言学习的代码转移到git的方法之四 设定.gitignore文件

场景，之前的两个文件不需要被repository管理，所以需要被忽略

generate-setpath.bat

setpath.bat

参照<https://help.github.com/articles/ignoring-files/>帮助



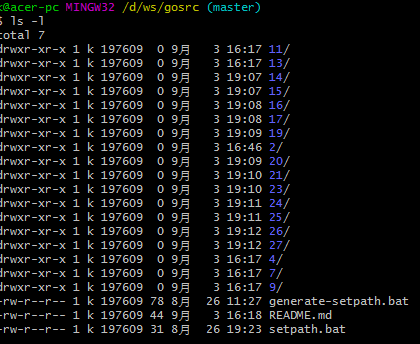
修改文件，用vi

$ cat .gitignore

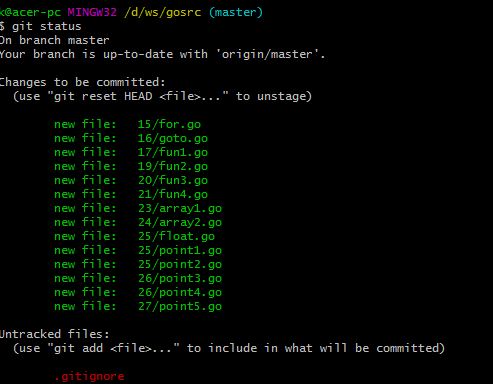
generate-setpath.bat

setpath.bat

文件如下：



Git status如下



这样虽然那两个bat文件没有了，

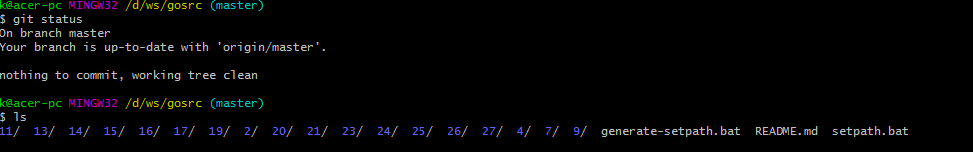
Add的时候 还会有.gitig nore自身，把它自身在加进去，进行add，commit，push即可了

$ cat .gitignore

generate-setpath.bat

setpath.bat

.gitignore



# Go语言学习笔记33 结构体

2017-09-05 [BugQ](https://mp.weixin.qq.com/s?__biz=MzIyNTY5ODI2MQ==&tempkey=OTIwX3lVSGJNaHVPVlpXYVNsaEFmcnJzbjFWRWZQTFd6RXRCczY2czZGajVYRHUyTjdxQXZzV1RoMjNOWklrMkhTRG9OWWtFelRCVWhrb2o0MWJHdGgwTnJ2QWd5Qkpvb0EybGlpOWJyeV85UlRLZXNYVl94azU0MHQyU3hLLTJza185WjJLYkRSbW5NMVpnQmpTVlhQYm1HQS1zYmdfTi1nRVViSG1QLXd%2Bfg%3D%3D&chksm=687a8d675f0d04710962afa848731715dcd582b4cff26b642f889498e34ad1b543174f5c5d60##)

结构体

和c的结构体没有太大区别

定义

type struct\_variable\_type struct {

  member definition;

  member definition;

  ...

  member definition;

}

使用

var varname struct\_variable\_type

varname := struct\_variable\_type  {xxx,xxx}

但是是不是结构体指针用 .来访问和c不一样，记得c是用->的啊

示例：

package main

import "fmt"

func main() {

type  person struct {

              firstnamestring;

              lastnamestring;

              ageint

              }

var gao person

hao:=person{"xiao","mei",30}

fmt.Println(hao.firstname, " " ,hao.lastname, " ", hao.age)

gao = hao

fmt.Println(gao.firstname, " " ,gao.lastname, " ", gao.age)

var qiao \*person

qiao = &gao

fmt.Println(qiao.firstname, " ",qiao.lastname, " ", qiao.age)

}

结果：

C:\Personal\learngo>go run struct.go

xiao  mei   30

xiao  mei   30

xiao  mei   30

注意：

type   person struct 的person 和struct 别定义反了

示例2

package main

import "fmt"

type  person struct {

              firstnamestring;

              lastnamestring;

              ageint

              }

func main() {

hao:=person{"xiao","mei",30}

fmt.Println(hao.firstname, " " ,hao.lastname, " ", hao.age)

var qiao \*person

qiao = &hao

qiao.age = 40

fmt.Println(qiao.firstname, " ",qiao.lastname, " ", qiao.age)

test\_struct\_point(qiao)

}

func test\_struct\_point(pp \*person) {

              fmt.Println(pp.firstname , " " , pp.lastname , " " , pp.age)

}

结果：

C:\Personal\learngo>go run struct2.go

xiao  mei   30

xiao  mei   40

xiao  mei   40

注意：

又一次忘记了把type的定义放到func外面，以后只要有type定义就放倒外面？这样好嘛

# Go语言学习笔记34 切片slice

这是一个新的概念， 和java nio里面的slice类似

Go 语言切片是对数组的抽象

适用场景，因为数组是固定长度，当需要动态数组怎么办，引入slice的概念

声明 var varname []type

或者make函数 varname:= make([]int , 20, 10), 这个地方的10是否可以小于20 （capacity < len) ?

理解下面这个写法就可以基本理解slice的用法

s := arr[startIndex:endIndex]

len() 和 cap() 函数

切片是可索引的，并且可以由 len() 方法获取长度。

切片提供了计算容量的方法 cap() 可以测量切片最长可以达到多少。

append()函数 和copy()函数

示例

package main

import "fmt"

func main() {

range1 := []int{1, 2, 3}

fmt.Printf("%v\n", range1)

range2 := make([]int, 10, 15)

fmt.Printf("%v\n", range2)

var range3 []int

var range4 [3]int = [3]int{1, 2, 3}

range3 = range4[1:3]

fmt.Printf("%v\n", range3)

len := len(range3)

cap := cap(range3)

fmt.Printf("len=%d,cap=%d\n", len, cap)

// will occur error, because the index out of rage for length 10

//range2[10] = 100

//fmt.Println(range2[10])

range2 = range3[:1]

fmt.Printf("%v\n", range2)

}

结果

C:\Personal\learngo\34>go run slice.go

[1 2 3]

[0 0 0 0 0 0 0 0 0 0]

[2 3]

len=2,cap=2

[2]

注意

Printf输出的格式为 %v

make的时候里面的slice定义 的顺序注意，java习惯的话容易写成int[],应该是[]int

数组的定义必须指定size或者是…的形式

Capacity小于size的时候，会报错range2 := make([]int, 10, 5)

给超过size但是小于capacity的slice元素赋值一样会出错

代码2

package main

import "fmt"

// test for append and copy

func main() {

range2 := make([]int, 10, 15)

// append(range2, 100) // this will be error, evaluated but not used

range2 = append(range2, 100)

fmt.Printf("%v, %d, %d\n", range2, len(range2), cap(range2))

range2 = append(range2, 100, 200, 300, 400)

fmt.Printf("%v, %d, %d\n", range2, len(range2), cap(range2))

range2 = append(range2, 100, 100)

fmt.Printf("%v, %d, %d\n", range2, len(range2), cap(range2))

subrange2 := []int{}

copy(subrange2, range2)

fmt.Printf("%v, %d, %d\n", subrange2, len(subrange2), cap(subrange2))

subrange3 := make([]int, 5, 10)

copy(subrange3, range2)

fmt.Printf("%v, %d, %d\n", subrange3, len(subrange3), cap(subrange3))

}

结果：

C:\Personal\learngo\34>go run slice2.go

[0 0 0 0 0 0 0 0 0 0 100], 11, 15

[0 0 0 0 0 0 0 0 0 0 100 100 200 300 400], 15, 15

[0 0 0 0 0 0 0 0 0 0 100 100 200 300 400 100 100], 17, 30

[], 0, 0

[0 0 0 0 0], 5, 10

注意

从第三行的输出结果来看append的追加不受制于capacity，而且会自动增加

从第四行的输出结果来看copy的结果不会影响被copy的slice的size和capacity

append需要由return值才行，而copy不需要，是不是很奇怪？

# Go语言学习笔记35 Range

range 关键字用于for循环中迭代数组(array)、切片(slice)、通道(channel)或集合(map)的元素。在数组和切片中它返回元素的索引值，在集合中返回 key-value 对的 key 值。

相当于iteration的处理， 得到 index，value，也可以单独得到其中一个，或者得到key-value

对于channel的还没有学习到

代码：

**package** main

**import** "fmt"

**func** main() {

    // test range

**for** key, value := **range** "go" {

        fmt.Println(key, "=", value)

    }

**for** key, value := **range** []int{1, 2, 3} {

        fmt.Println(key, "=", value)

    }

**for** \_, value := **range** []string{"hello", "world", "!"} {

        fmt.Println(value)

    }

**for** key, value := **range** **map**[string]string{"name": "shame on me", "sex": "undefined"} {

        fmt.Println(key, "-", value)

    }

}

结果：

**C:/Personal/go/bin/go.exe run range.go [C:/Personal/learngo/35]**

0 = 103

1 = 111

0 = 1

1 = 2

2 = 3

hello

world

!

name - shame on me

sex - undefined

**成功: プロセスがコード 0 で終了しました**

注意：

1．range是一个keyword，而不是一个函数或者方法，所以后面接着的对象，无括号，也无引号

2．因为前面写的都是for ，好几次忘掉写range

3.fmt.Println(key, "=", value)的时候value为字符串中的每个字符，因为不带格式，所以输出为ascii

4.map还没学到，对**map**[string]string的猜想应该是[]里面为key？后面的string为value？

# Go语言学习笔记36 Map

特点： 无序，可用range迭代，取得size用len，

现在这个例子中的map的key和value都必须指定类型，如何和java一样指定object类型，需要之前学习时候接触到的 interface{}吗？

基本的map的用户看代码

代码：

**package** main

**import** "fmt"

**func** main() {

    // def

**var** carinfo **map**[string]string

    // init

    carinfo = make(**map**[string]string)

    // set value

    carinfo["name"] = "eagle"

    carinfo["color"] = "black"

    carinfo["weight"] = "100"

    // get value

    fmt.Println(carinfo)

    fmt.Println(carinfo["name"])

    // loop

**for** k, v := **range** carinfo {

        fmt.Println(k, " ", v)

    }

    // contain

    k, v := carinfo["belong"]

    fmt.Println(k, " ", v)

    // delete

    delete(carinfo, "name")

    fmt.Println("after delete = ", carinfo["name"])

    fmt.Println("end")

    // size

    fmt.Println(len(carinfo))

}

结果：

**d:/ws/go/bin/go.exe run map.go [D:/ws/gosrc/36]**

map[name:eagle color:black weight:100]

eagle

weight 100

name eagle

color black

false

after delete =

end

2

**成功: 进程退出代码 0.**

注意：

倒是没有什么值得注意的，就是像delete和len这种需要外部函数来实现，还是有些不一样的感觉

# Go语言学习笔记37 递归

任何语言学习过程中都少不了递归，这也来一个例子

迭代注意事项

1. 退出条件
2. 递归变量的变化

代码：

**package** main

**import** "fmt"

**func** main() {

    x := recursion(5)

    fmt.Println(x)

}

**func** recursion(x int) int {

**if** x <= 1 {

        x = 1

    } **else** {

        x = x \* recursion(x-1)

    }

**return** x

}

结果：

**d:/ws/go/bin/go.exe run recursion.go [D:/ws/gosrc/37]**

120

**成功: 进程退出代码 0.**

注意：

比较简单，没有什么特殊注意的地方，就是recursion函数的定义的地方，还不习惯，忘记写int的返回值了

# Go语言学习笔记38 语言类型转换

语法：type\_name(expression)

示例：

**package** main

**import** "fmt"

**func** main() {

    x := recursion(5)

    fmt.Println(float32(x))

    fmt.Printf("%f\n", float32(x))

**var** f float32

    f = float32(x)

    fmt.Printf("%f\n", f)

}

**func** recursion(x int) int {

**if** x <= 1 {

        x = 1

    } **else** {

        x = x \* recursion(x-1)

    }

**return** x

}

结果：

**d:/ws/go/bin/go.exe run typeconvert.go [D:/ws/gosrc/38]**

120

120.000000

120.000000

**成功: 进程退出代码 0.**

注意

如果用Println用打印，结果还是120而没有小数位，要用Printf加上%f才行

# Go语言学习笔记39 语言接口

这是一个新概念，要注意

**1.就是原先的interface的概念，可以把interface转换成任何一个对象，现在已经知道的就是一个struct类型，用new（structtype）的形式。然后就可以调用interface里面method了**

代码：

**package** main

**import** "fmt"

**type** learn **interface** {

    learn()

}

**type** language **struct** {

    name string

}

**type** physics **struct** {

    name string

}

**func** (language) learn() {

    fmt.Println("learn go")

}

**func** (physics) learn() {

    fmt.Println("learn physics")

}

**func** main() {

**var** learn\_instance learn

    learn\_instance = new(language)

    learn\_instance.learn()

    learn\_instance = new(physics)

    learn\_instance.learn()

}

结果：

**d:/ws/go/bin/go.exe run interface.go [D:/ws/gosrc/39]**

learn go

learn physics

**成功: 进程退出代码 0.**

注意：

1.写代码的时候好多语法错误，如，**func** (language) learn()的（struct type）的写法

2. **func** (language) learn() {的如果有返回值的function定义还不知道如何写

3. 需要使用new 关键字还是函数（这个new），后面是带括号的type啊。learn\_instance = new(language)

**2.用struct的实例也可以访问interface的方法，这是上一种分别使用在那种场景呢？区别一个是使用interface的实例访问，一个是使用struct的实例访问，仅此而已 ？**

代码：

**package** main

**import** "fmt"

**type** learn **interface** {

    learn()

}

**type** language **struct** {

    name string

}

**type** physics **struct** {

    name string

}

**func** (lang language) learn() {

    fmt.Println("learn go")

    fmt.Println(lang.name)

}

**func** (phy physics) learn() {

    fmt.Println("learn physics")

    fmt.Println(phy.name)

}

**func** main() {

**var** java language

    java.name = "java"

    java.learn()

    earch := physics{"\n, a cource about earch"}

    earch.learn()

}

结果：

**d:/ws/go/bin/go.exe run interface2.go [D:/ws/gosrc/39]**

learn go

java

learn physics

, a cource about earch

**成功: 进程退出代码 0.**

注意：

1. func的实现定义可以有参数
2. 重新复习了struct的初始化写法
3. **增加了带返回值的例子**

代码：

**package** main

**import** "fmt"

**type** learn **interface** {

    learn() string

}

**type** language **struct** {

    name string

}

**type** physics **struct** {

    name string

}

**func** (lang language) learn() string {

    fmt.Println("learn go")

    fmt.Println(lang.name)

**return** "skill"

}

**func** (phy physics) learn() string {

    fmt.Println("learn physics")

    fmt.Println(phy.name)

**return** "ppphhh"

}

**func** main() {

**var** java language

    java.name = "java"

    result := java.learn()

    earch := physics{"\n, a cource about earch"}

    result2 := earch.learn()

    fmt.Println(result, " ", result2)

}

结果：

**d:/ws/go/bin/go.exe run interface3.go [D:/ws/gosrc/39]**

learn go

java

learn physics

, a cource about earch

skill ppphhh

**成功: 进程退出代码 0.**

说明：

就是增加了return value的定义部分，和调用的返回部分

# Go语言学习笔记40 错误处理

Error 是 内置类型，一般用于函数的最后一个返回值上面

error类型是一个接口类型，这是它的定义：

type error interface {

Error() string

}

使用errors.New 可返回一个错误信息。

方法为大写字母开头的Error。Type却是小写

代码：

**package** main

**import** (

    "errors"

    "fmt"

)

**type** CalAge **struct** {

    now float64

    birth float64

}

**func** (age CalAge) Error() string {

    message := "now = %f, birth = %f, now < birth is error\n"

**return** fmt.Sprintf(message, age.now, age.birth)

}

**func** checkage(age CalAge) (bool, string) {

**if** age.now < age.birth {

**return** false, age.Error()

    } **else** {

**return** true, ""

    }

}

**func** main() {

    age1 := CalAge{100.0, 200.0}

    fmt.Println(age1.Error())

    v, e := checkage(age1)

    fmt.Println(v, e)

**if** e == "" {

        fmt.Println("error is nil")

    }

    age1 = CalAge{200.0, 100.0}

    v, e = checkage(age1)

    fmt.Println(v, e)

**if** e == "" {

        fmt.Println("error is nil")

    }

    // new error

    err := errors.New("work later and back to home until 10pm, 2017/09/11")

    fmt.Println(err)

}

结果：

**d:/ws/go/bin/go.exe run error.go [D:/ws/gosrc/40]**

now = 100.000000, birth = 200.000000, now < birth is error

false now = 100.000000, birth = 200.000000, now < birth is error

true

error is nil

work later and back to home until 10pm, 2017/09/11

**成功: 进程退出代码 0.**

注意：

使用 errors.New 需要导入"errors"

Error接口定义的函数实现，不要忘记了括号**func** (age CalAge) Error() string {

函数的return的值的个数和定义的必须要一致，否则会出错 **return** false, age.Error()

代码2：

**package** main

**import** (

    "errors"

    "fmt"

)

**type** CalAge **struct** {

    now float64

    birth float64

}

**func** (age \*CalAge) Error() string {

    message := "now = %f, birth = %f, now < birth is error\n"

**return** fmt.Sprintf(message, age.now, age.birth)

}

**func** checkage(age CalAge) (bool, string) {

**if** age.now < age.birth {

**return** false, age.Error()

    } **else** {

**return** true, ""

    }

}

**func** main() {

    age1 := CalAge{100.0, 200.0}

    fmt.Println(age1.Error())

    v, e := checkage(age1)

    fmt.Println(v, e)

**if** e == "" {

        fmt.Println("error is nil")

    }

    age1 = CalAge{200.0, 100.0}

    v, e = checkage(age1)

    fmt.Println(v, e)

**if** e == "" {

        fmt.Println("error is nil")

    }

    // new error

    err := errors.New("work later and back to home until 10pm, 2017/09/11")

    fmt.Println(err)

}

结果：

**d:/ws/go/bin/go.exe run error2.go [D:/ws/gosrc/40]**

now = 100.000000, birth = 200.000000, now < birth is error

false now = 100.000000, birth = 200.000000, now < birth is error

true

error is nil

work later and back to home until 10pm, 2017/09/11

**成功: 进程退出代码 0.**

注意：

代码1和代码2只有一个地方不一样，在实现error的接口方法时候。但是输出结果却是一样的。

对两个地方进行了截图，只不过一个是struct 指针类型，一个是struct类型，但是不影响

Struct变量的元素访问。它们的区别是什么了

图1

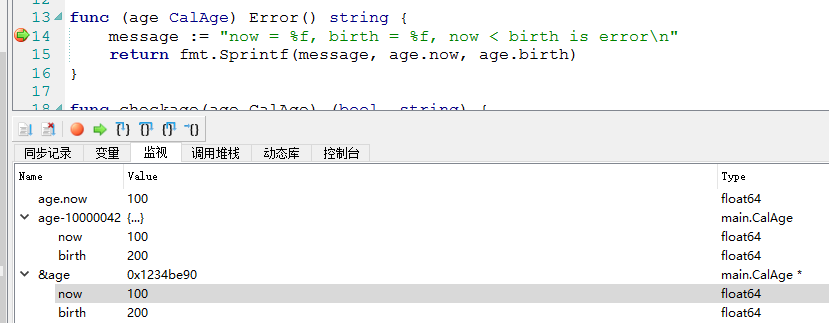
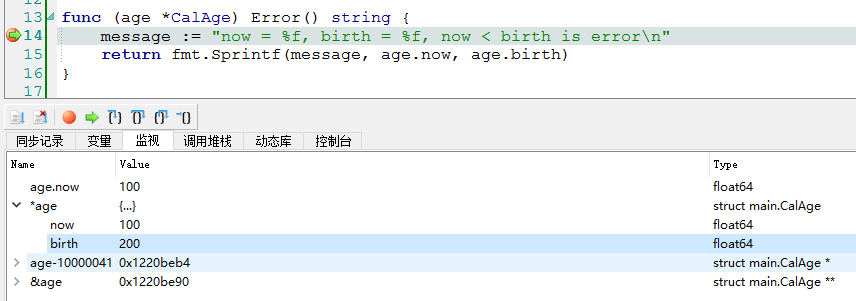


图2



# Go语言学习笔记41 XXX