Golang 学习笔记

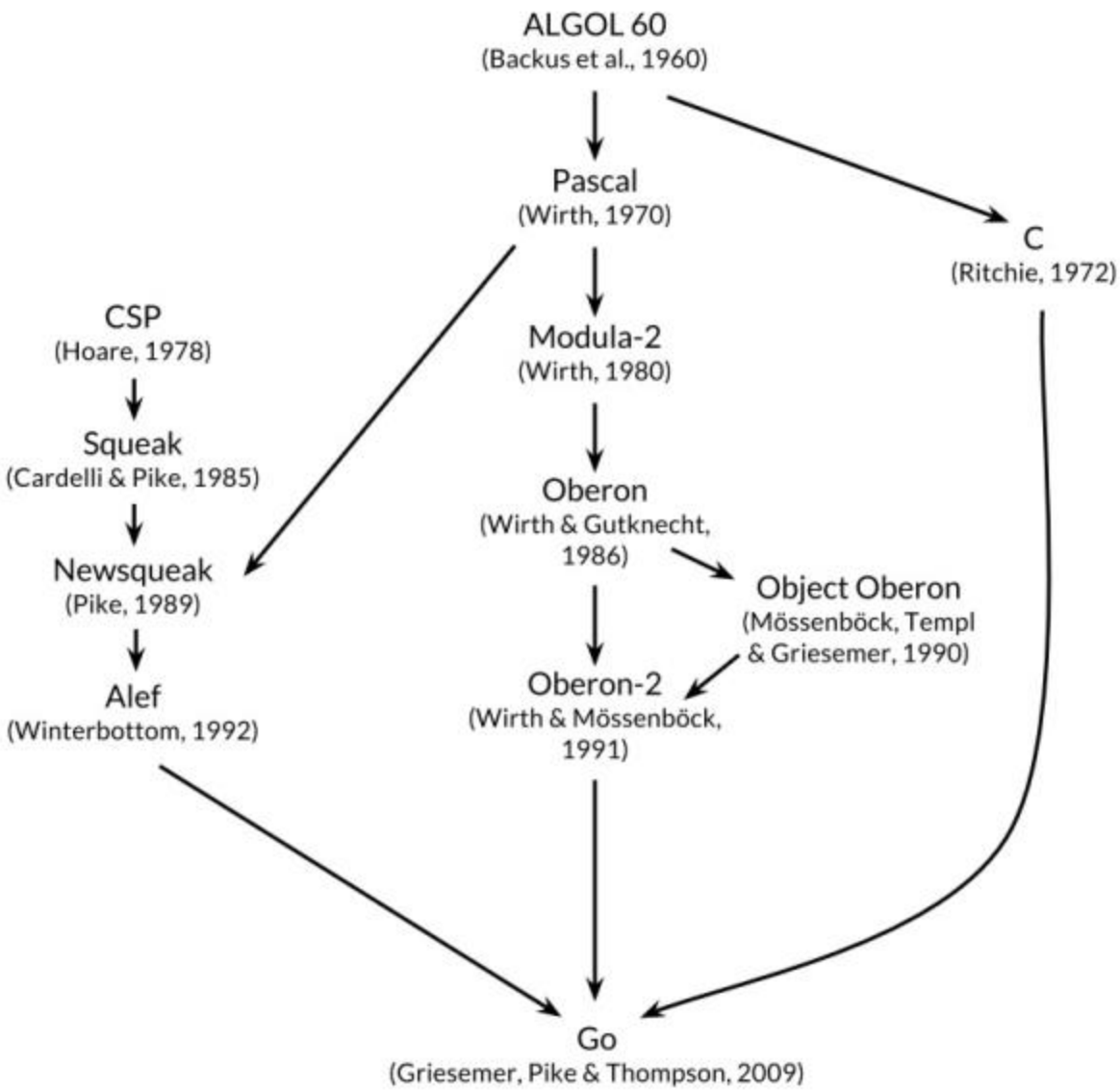
Go语言学习笔记 部分代码在https://github.com/YechengChu/golang-simple-example中

Golang 学习笔记01

2020/07/03

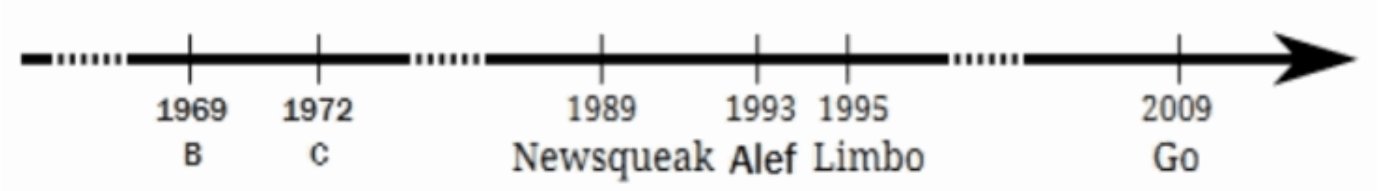
cyc/Golang 学习笔记/Golang 学习笔记01

Go语言的起源



Go语言的这些地方都做的还不错：拥有自动垃圾回收、一个包系统、函数作为一等公民、词法作用域、系统调用接口、只读的UTF8字符串等。但是Go语言本身只有很少的特性，也不太可能添加太多的特性。例如，它没有隐式的数值转换，没有构造函数和析构函数，没有运算符重载，**没有默认参数**，**也没有继承**，**没有泛型**，**没有异常**，没有宏，没有函数修饰，更没有线程局部存储。但是，语言本身是成熟和稳

定的，而且承诺保证向后兼容：用之前的Go语言编写程序可以用新版本的Go语言编译器和标准库直接构建而不需要修改代码。



入门——基础知识与命令行及循环使用体验

Hello World

hello.go

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello World!")
7 } // main
```

作为C语言中招牌的`printf`格式化函数移植了到了Go语言中，函数放在`fmt`包中(`fmt`是格式化单词`format`的缩写)。
使用如下代码运行

```
1 $ go run hello.go
```

运行结果如下

```
1 Hello World!
```

如果不只是一次性实验，你肯定希望能够编译这个程序，保存编译结果以备将来之用。可以用build子命令：

```
1 $ go build hello.go
```

这个命令生成一个名为hello的可执行的二进制文件，之后你可以随时运行它，不需任何处理。

```
$ ./hello
```

- Go语言的代码通过**包(package)**组织，包类似于其它语言里的**库(libraries)**或者**模块(modules)**。一个包由位于单个目录下的一个或多个.go源代码文件组成，目录定义包的作用。每个源文件都以一条`package`声明语句开始，这个例子里就是`package main`，表示该文件属于哪个包，紧跟着一系列**导入(import)**的包，之后是存储在这个文件里的程序语句。
- Go的标准库提供了100多个包，以支持常见功能。比如`fmt`包，就含有格式化输出、接收输入的函数。`Println`是其中一个基础函数。
- 必须告诉编译器源文件需要哪些包，这就是跟随在`package`声明后面的`import`声明扮演的角色。hello world例子只用到了一个包，大多数程序需要导入多个包。必须恰当导入需要的包，**缺少了必要的包或者导入了不需要的包，程序都无法编译通过。**
- `import`声明必须跟在文件的`package`声明之后。随后，则是组成程序的函数、变量、常量、类型的声明语句(分别由关键字`func`，`var`，`const`，`type`定义)。
- 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 private** ）。
- 编译器会主动把特定符号后的换行符转换为分号，因此换行符添加的位置会影响Go代码的正确解析(比如行末是标识符、整数、浮点数、虚数、字符或字符串文字、关键字`break`、`continue`、`fallthrough`或`return`中的一个、运算符和分隔符`++`、`--`、`)`、`]`或`}`中的一个)。举个例子，**函数的左括号`{`必须和`func`函数声明在同一行上，且位于末尾，不能独占一行**，而在表达式`x + y`中，**可在`+`后换行，不能在`+`前换行**(以`+`结尾的话不会被插入分号分隔符，但是以`x`结尾的话则会被分号分隔符，从而导致编译错误)。

Go语言的数据类型

序号	类型和描述
1	**布尔型** 布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。
2	**数字类型** 整型 int 和浮点型 float32、float64，Go 语言支持整型和浮点型数字，并且支持复数，其中位的运算采用补码。
3	**字符串类型** 字符串就是一串固定长度的字符连接起来的字符序列。Go 的字符串是由单个字节连接起来的。Go 语言的字符串的字节使用 UTF-8 编码标识 Unicode 文本。

4	**派生类型** 包括： • (a) 指针类型（Pointer） • (b) 数组类型 • (c) 结构化类型(struct) • (d) Channel 类型 • (e) 函数类型 • (f) 切片类型 • (g) 接口类型（interface） • (h) Map 类型

命令行参数

- `os`包以跨平台的方式，**提供了一些与操作系统交互的函数和变量**。程序的**命令行参数可从os包的Args变量获取**；os包外部使用os.Args访问该变量。
- os.Args的第一个元素，**os.Args[0], 是命令本身的名字**；**其它的元素**则是程序启动时传给它的**参数**。
- s[m:n]形式的切片表达式如果省略切片表达式的m或n，会默认传入0或len(s)，因此os.Args[1:len(os.Args)]可以简写成os.Args[1:]。

命令行参数和for循环的使用

hello.go

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    // var声明定义了两个string类型的变量info和seperation
    // 变量会在声明时直接初始化。如果变量没有显式初始化，则被隐式地赋予其类型的零值
    // 字符串类型是空字符串""，所以info和separation隐式地初始化成空字符串
    var info, seperation string
    for i := 1; i < len(os.Args); i++ {
        // 是一条赋值语句，将info的旧值跟seperation与os.Args[i]连接后赋值回info
        info += seperation + os.Args[i]
        seperation = " "
```

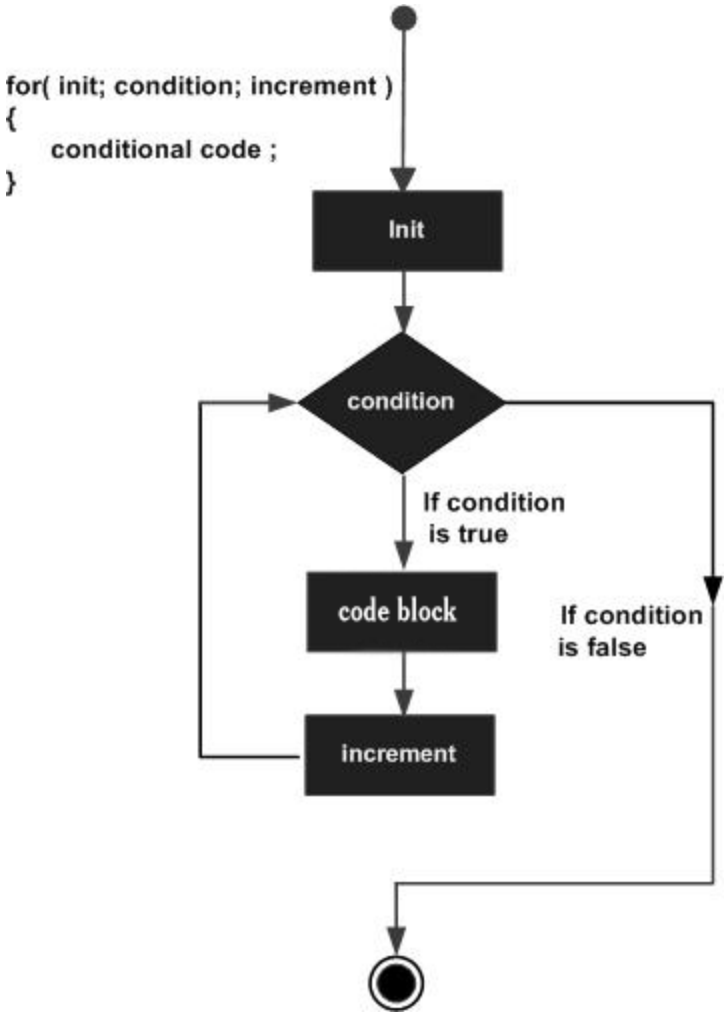
```
    } // for
    fmt.Println(info)
} // main
```

运行结果如下

```
$ go run hello.go Hello world form this test
Hello world form this test
```

- 如果没有初始化，则变量默认为零值，就是变量没有做初始化时系统默认设置的值。
  - 数值类型（包括complex64/128）为 0
  - 布尔类型为 false
  - 字符串为 ""（空字符串）
  - 以下几种类型为 nil：
    - var a *int
    - var a []int
    - var a map[string] int
    - var a chan int
    - var a func(string) int
    - var a error // error 是接口
- 自增语句`i++`给`i`加1，这和`i += 1`以及`i = i + 1`都是等价的。对应的还有`i--`给`i`减1。它们是**语句**，而不像C系的其它语言那样是表达式。所以`j = i++`非法，而且++和--都只能放在变量名后面，因此`--i`也非法！
- Go语言只有for循环这**一种循环语句**。for循环有多种形式，其中一种如下所示：

```
for initialization; condition; post {
    // zero or more statements
}
```



- for循环的这三个部分每个都可以省略，如果省略`initialization`和`post`，分号也可以省略：

```
// a traditional "while" loop
for condition {
    // ...
}
```

- 如果连`condition`也省略了，像下面这样，就变成一个无限循环，尽管如此，还可以用其他方式终止循环，如一条`break`或`return`语句。

```
// a traditional infinite loop
for {
    // ...
}
```

for循环的另一种形式

`for`循环的另一种形式，在某种数据类型的区间（range）上遍历，如字符串或切片。

hello.go

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // 使用一条短变量声明来声明并初始化info和seperation，也可以将这两个变量分开声明
    info, seperation := "", ""
    for _, arg := range os.Args[1:] {
        info += seperation + arg
        seperation = " "
    } // for
    fmt.Println(info)
} // main
```

运行结果如下

```
$ go run hello.go Hello world form this new for loop!
Hello world form this new for loop!
```

- 每次循环迭代，`range`产生一对值；索引以及在该索引处的元素值。(In each iteration of the loop, `range` produces a pair of values: the *index* and *the value of the element at that index*.) 这个例子不需要索引，但`range`的语法要求，要处理元素，必须处理索引。
- 一种思路是把索引赋值给一个临时变量，如`temp`，然后忽略它的值，但Go语言不允许使用无用的局部变量(local variables)，因为这会导致编译错误。
- Go中的解决方法是用`空标识符`(blank identifier)，即`_`(也就是下划线)。空标识符可用于任何语法**需要变量名但程序逻辑不需要**的时候，例如，在循环里，丢弃不需要的循环索引，保留元素值。
- 声明一个变量有好几种方式，下面这些都等价：

```
// 短变量声明，最简洁，但只能用在函数内部，而不能用于包变量
```

```
s := ""
// 依赖于字符串的默认初始化零值机制，被初始化为""
var s string
// 用得很少，除非同时声明多个变量
var s = ""
// 显式地标明变量的类型，当变量类型与初值类型相同时，类型冗余
// 但如果两者类型不同，变量类型就必须了
var s string = ""

// 一般使用前两种形式中的某个
// 初始值重要的话就显式地指定变量的类型
// 否则使用隐式初始化
```

使用strings包的Join函数

hello.go

```
package main

import (
    "fmt"
    "os"
    "strings"
)

func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
} // main
```

运行结果如下

```
$ go run hello.go Hello world form functions provided by strings p
ackage
Hello world form functions provided by strings package
```

检测重复行

命令行输入中检测重复行

hello.go

```go
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    // 创建一个key为string，value为int的map
    counts := make(map[string]int)
    // 创建一个scanner从standard input读取信息
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        // map中input作为key对应的value加1
        counts[input.Text()]++
    } // for

    // 遍历counts这个map
    for line, n := range counts {
        // if语句条件两边也不加括号，但是主体部分需要加
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        } // if
    } // for
} // main
```

运行结果如下

```
$ go run hello.go
hello
hahaha
hahaha
hahaha
helloyou
zizizi
```

```
hello
 (以^D终止)
2       hello
3       hahaha
```

- 每次迭代得到两个结果，键和其在`map`中对应的值。`map`的**迭代顺序并不确定**，从实践来看，该顺序随机，每次运行都会变化。
- `bufio`包，它使处理输入和输出方便又高效。`Scanner`类型是该包最有用的特性之一，它读取输入并将其拆成行或单词；通常是处理行形式的输入最简单的方法。
  - 程序使用短变量声明创建`bufio.Scanner`类型的变量`input`。

```
input := bufio.NewScanner(os.Stdin)
```

  - 该变量从程序的标准输入中读取内容。每次调用`input.Scan()`，即读入下一行，并移除行末的换行符；读取的内容可以调用`input.Text()`得到。`Scan`函数在读到一行时返回`true`，不再有输入时返回`false`。
- `fmt.Printf`函数对一些表达式产生格式化输出。该函数的首个参数是个格式字符串，指定后续参数被如何格式化。各个参数的格式取决于"转换字符"(conversion character)，形式为百分号后跟一个字母。

```
%d          十进制整数
%x, %o, %b  十六进制，八进制，二进制整数。
%f, %g, %e  浮点数:  3.141593 3.141592653589793 3.141593e+00
%t          布尔：true或false
%c          字符（rune） (Unicode码点)
%s          字符串
%q          带双引号的字符串"abc"或带单引号的字符'c'
%v          变量的自然形式（natural format）
%T          变量的类型
%%          字面上的百分号标志（无操作数）
```

- 转义字符

```
\n：换行符
\r：回车符
\t：tab 键
\u 或 \U：Unicode 字符
\\：反斜杠自身
```

命令行和文件中检测重复行

创建3个文件来测试

```
test.txt
hi
hhh
houhou
hhh
hhh
hhh
hi
--------------------
t2.txt
xixi
hhhhh
hhhhh
xxx
lololo
xixi
hhhhh
--------------------
t3.txt
beijing
wuhan
shanghai
suzhou
shanghai
wuxi
chongqing
dalian
suzhoukunshan
```

hello.go

```
package main

import (
```

```
    "bufio"
    "fmt"
    "os"
)

func main() {
    // 创建一个key为string，value为int的map
    counts := make(map[string]int)
    // 从命令行读取输入要查看的文件名
    files := os.Args[1:]
    // 如果没有输入文件名，则当是从命令行读取(和之前的程序一样)
    // 否则读取文件
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        // 遍历从命令行读入的要打开的文件名
        // _是index，arg是index对应的值
        for _, arg := range files {
            // f是返回的文件，error是错误信息
            // error为nil就说明没有错误
            f, error := os.Open(arg)
            // 如果一个文件打开有误，则打印出报错信息，并继续
            if error != nil {
                fmt.Fprintf(os.Stderr, "error message: %v\n", err
or)
                continue
            } // if
            countLines(f, counts)
            // 关闭文件
            f.Close()
        } // for
    } // else

    // 遍历counts这个map
    for line, n := range counts {
        // if语句条件两边也不加括号，但是主体部分需要加
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        } // if
    } // for
```

```
} // main

func countLines(f *os.File, counts map[string]int) {
    // 创建一个scanner从f读取信息
    input := bufio.NewScanner(f)
    for input.Scan() {
        // map中input作为key对应的value加1
        counts[input.Text()]++
    } // for
} // countLines
```

运行结果如下

```
$ go run hello.go test.txt t2.txt t3.txt
3       hhhhh
2       hi
2       xixi
2       shanghai
4       hhh
----------------------------------------------------
$ go run hello.go test.txt t2.txt t3.txt
2       shanghai
2       hi
2       xixi
4       hhh
3       hhhhh
----------------------------------------------------
$ go run hello.go test.txt t2.txt t3.txt
4       hhh
2       xixi
2       shanghai
3       hhhhh
2       hi
----------------------------------------------------
$ go run hello.go test.txt t2 t3.txt
error message: open t2: no such file or directory
2       hi
4       hhh
2       shanghai
```

- `os.Open`函数返回两个值。
  - 第一个值是被打开的文件(`*os.File`)，其后被`Scanner`读取。
  - 第二个值是内置`error`类型的值。
    - 如果`err`等于内置值`nil`，那么文件被成功打开。读取文件，直到文件结束，然后调用`Close`关闭该文件，并释放占用的所有资源。
    - 相反的话，如果`err`的值不是`nil`，说明打开文件时出错了。这种情况下，错误值描述了所遇到的问题。我们的错误处理非常简单，只是使用`Fprintf`与表示任意类型默认格式值的动词`%v`，向标准错误流打印一条信息，然后`dup`继续处理下一个文件；`continue`语句直接跳到`for`循环的下个迭代开始执行

只执行文件中的重复行检测

hello.go

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)

func main() {
    // 创建一个key为string，value为int的map
    counts := make(map[string]int)
    // 遍历从命令行读入的要打开的文件名
    // _是index，filename是index对应的值
    for _, filename := range os.Args[1:] {
        // ReadFile函数返回一个字节切片(byte slice)
        // error是错误信息error为nil就说明没有错误
        data, error := ioutil.ReadFile(filename)
        if error != nil {
            fmt.Fprintf(os.Stderr, "error message: %v\n", error)
            continue
        }
        // 字节切片必须把转换为string，才能用strings.Split分割
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
```

```
        } // for
    } // for
    // 遍历counts这个map
    for line, n := range counts {
        // 如果出现次数大于1
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        } // if
    } // for
} // main
```

运行结果如下

```
$ go run hello.go test.txt t2.txt t3.txt
4       hhh
2       xixi
2       shanghai
2       hi
3       hhhhh
-------------------------------------------------
$ go run hello.go test.txt t2.txt t3
dup3: open t3: no such file or directory
2       hi
4       hhh
2       xixi
3       hhhhh
```

参考资料

Go语言圣经（中文版）

https://books.studygolang.com/gopl-zh/index.html

Go语言高级编程(Advanced Go Programming)

https://chai2010.cn/advanced-go-programming-book/

菜鸟教程——Go 语言教程

https://www.runoob.com/go/go-tutorial.html

Go 入门教程

https://xueyuanjun.com/books/golang-tutorials

《Go入门指南》

https://github.com/unknwon/the-way-to-go_ZH_CN

Golang 学习笔记02

2020/07/07

cyc/Golang 学习笔记/Golang 学习笔记02

入门——并发体验与其他要点介绍

获取URL

hello.go

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
)

func main() {
    // _是用户输入url的index
    // url是用户输入的url
    for _, url := range os.Args[1:] {
        // http.Get函数是创建HTTP请求的函数
        // 在response这个结构体中得到访问的请求结果
        // error中返回错误信息，没错的话error就为nil
        response, error := http.Get(url)
        // 如果有错误发生
        if error != nil {
            fmt.Fprintf(os.Stderr, "Error message when fetching: %v\n", error)
            // 终止进程，并且返回一个status错误码1
            os.Exit(1)
        } // if
        // response的Body字段包括一个可读的服务器响应流
```

```
        // 从response body中读取到全部内容，将值放入message中
        message, err := ioutil.ReadAll(response.Body)
        // 关闭response的Body流，防止资源泄露
        response.Body.Close()
        // 如果有错误发生
        if err != nil {
            fmt.Fprintf(os.Stderr, "Error message when reading response body: %v\n", err)
            // 终止进程，并且返回一个status错误码1
            os.Exit(1)
        } // if
        // 印出response body的信息
        fmt.Printf("%s\n", message)
    } // for
} // main
```

运行结果如下(无vpn情况下)

```
$ go run hello.go https://www.baidu.com https://www.google.com.hk
<html>
<head>
        <script>
                location.replace(location.href.replace("https://","http://"));
        </script>
</head>
<body>
        <noscript><meta http-equiv="refresh" content="0;url=http://www.baidu.com/"></noscript>
</body>
</html>
Error message when fetching: Get "https://www.google.com.hk": dial tcp 69.171.227.37:443: i/o timeout
exit status 1
------------------------------------------------------
$ go run hello.go https://asf
Error message when fetching: Get "https://asf": dial tcp: lookup asf: no such host
exit status 1
```

```
18 ------------------------------------------------------
19 $ go run hello.go avbdd
20 Error message when fetching: Get "avbdd": unsupported protocol sc
   heme ""
21 exit status 1
```

Printf()和Fprintf()函数的区别用法是什么?

- 都是把格式好的字符串输出，只是输出的目标不一样:
- Printf(), 是把格式字符串输出到标准输出(一般是屏幕，可以重定向)。Printf()是和标准输出文件(stdout)关联的, Fprintf则没有这个限制。
- Fprintf(), 是把格式字符串输出到**指定文件设备**中，所以参数笔printf多一个文件指针FILE*。主要用于文件操作。Fprintf()是格式化输出到一个stream，通常是到文件。

并发获取多个URL

下面程序和之前的程序执行类似的工作，但这个程序使用并发——它会**同时去获取**所有的URL，所以这个程序的**总执行时间不会超过执行时间最长的那一个任务**，前面的程序执行时间则是所有任务执行时间之和。

fetchAll.go

```go
1  package main
2
3  import (
4      "fmt"
5      "io"
6      "io/ioutil"
7      "net/http"
8      "os"
9      "time"
10 )
11
12 func main() {
13     startTime := time.Now()
14     // make函数创建了一个传递string类型参数的channel
15     myChannel := make(chan string)
16     for _, url := range os.Args[1:] {
17         // 创建一个goroutine，并且让函数在这个goroutine异步执行http.Get
   方法
18         go fetch(url, myChannel) // start a goroutine
```

```
    } // for
    for range os.Args[1:] {
        fmt.Println(<-myChannel) // receive from channel myChannel
    } // for
    fmt.Printf("%.2fs elapsed\n", time.Since(startTime).Seconds())
} // main

func fetch(url string, myChannel chan<- string) {
    startTime := time.Now()
    response, errorMsg := http.Get(url)
    if errorMsg != nil {
        myChannel <- fmt.Sprint(errorMsg) // send to channel myChannel
    } // if
    // io.Copy会把响应的Body内容拷贝到ioutil.Discard输出流中
    nbytes, errMsg := io.Copy(ioutil.Discard, response.Body)
    response.Body.Close() // don't leak resources
    if errMsg != nil {
        myChannel <- fmt.Sprintf("Erro while loding %s: %v", url, errMsg)
        return
    } // if
    timeSpent := time.Since(startTime).Seconds()
    // 当请求返回内容时，fetch函数都会往myChannel这个channel里写入一个字符串
    myChannel <- fmt.Sprintf("%.2fs  %7d  %s", timeSpent, nbytes, url)
} // fetch
```

运行结果如下

```
$ go run fetchAll.go https://books.studygolang.com/gopl-zh/ch1/ch1-06.html https://www.baidu.com https://www.bilibili.com https://www.planespotters.net https://fanyi.baidu.com https://chai2010.cn/advanced-go-programming-book/ https://studentnet.cs.manchester.ac.uk/me/spot/
0.33s      227  https://www.baidu.com
```

```
0.40s    70075  https://books.studygolang.com/gopl-zh/ch1/ch1-06.html
0.45s     2206  https://www.bilibili.com
0.63s    39757  https://chai2010.cn/advanced-go-programming-book/
0.69s   223962  https://fanyi.baidu.com
1.48s      162  https://www.planespotters.net
3.12s    21046  https://studentnet.cs.manchester.ac.uk/me/spot/
3.12s elapsed
```

- goroutine是一种函数的并发执行方式，而channel是用来在goroutine之间进行参数传递。main函数本身也运行在一个goroutine中，而go function则表示**创建一个新的goroutine**，并**在这个新的goroutine中执行这个函数**。
- main函数中用make函数创建了一个**传递string类型参数的channel**，对每一个命令行参数，我们都用go这个关键字来创建一个goroutine，并且让函数在这个goroutine异步执行http.Get方法。
- 这个程序里的io.Copy会把响应的Body内容拷贝到ioutil.Discard输出流中(可以把这个变量看作一个垃圾桶，可以向里面写一些不需要的数据)，因为我们需要这个方法返回的字节数，但是又不想要其内容。
- 每当请求返回内容时，fetch函数都会往myChannel这个channel里写入一个字符串，由main函数里的第二个for循环来处理并打印channel里的这个字符串。
- 当一个goroutine尝试在一个channel上做**send或者receive**操作时，这个**goroutine会阻塞在调用处，直到另一个goroutine往这个channel里写入、或者接收值**，这样两个goroutine才会继续执行channel操作之后的逻辑。
- 在这个例子中，每一个fetch函数在**执行时都会往channel里发送一个值(myChannel <- expression)，主函数负责接收这些值(<-myChannel)**。这个程序中我们**用main函数来接收所有fetch函数传回的字符串，可以避免在goroutine异步执行还没有完成时main函数提前退出**。

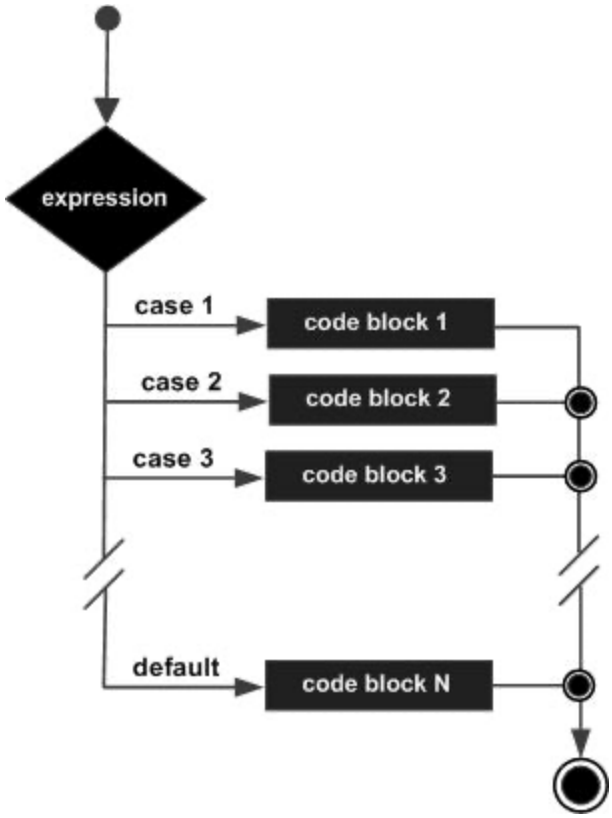
其他要点

控制流的switch语句

- Go语言并不需要显式地在每一个case后写break，语言默认执行完case后的逻辑语句会自动退出。
- 如果想要相邻的几个case都执行同一逻辑的话，需要显式地写上一个fallthrough语句来覆盖这种默认行为。
- Go语言里的switch还可以不带操作对象(switch不带操作对象时默认用true值代替，然后**将每个case的表达式和true值进行比较**)；可以直接罗列多种条件，**像其它语言里面的多个if else一样**。

```
// 普通的switch
switch(条件){
    case 情况1:
        // code
```

```
    case 情况2:
        // code
    case 情况3:
        // code
    default:
        // code
}

// 不带操作对象的switch
switch {
    case 条件1:
        // code
    case 条件2:
        // code
    case 条件3:
        // code
    default:
        // code
}
```



命名类型

类型声明使得我们可以很方便地给一个特殊类型一个名字。因为struct类型声明通常非常地长，所以我们总要给这种struct取一个名字。

```
type Point struct {
    X, Y int
}
var p Point
```

指针

指针是可见的内存地址，**&操作符**可以**返回一个变量的内存地址**，并且***操作符**可以**获取指针指向的变量内容**，但是在Go语言里没有指针运算。

方法和接口

方法是和命名类型关联的一类函数。接口是一种**抽象类型**，这种类型可以让我们以同样的方式来处理不同的固有类型，不用关心它们的具体实现，而只需要关注它们提供的方法。

程序结构

命名

一个名字必须以一个字母或下划线开头，后面可以跟任意数量的字母、数字或下划线。区分大小写！

关键字和保留字

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

声明

Go语言主要有四种类型的声明语句：var、const、type和func，分别对应变量、常量、类型和函数实体对象的声明。

circle.go

```go
package main

import (
    "fmt"
    "os"
    "strconv"
)

const PI = 3.14

func main(){
    // 从命令行读入半径数据
    var radiusString = os.Args[1]
    // 把半径转化为float
    var radius,err = strconv.ParseFloat(radiusString,32)
    // 如果有错误打印错误信息并退出
    if err != nil{
    fmt.Fprintf(os.Stderr, "Error message: %v\n",err)
        os.Exit(1)
    } // if
    // 计算出圆的面积
    // A = PI * r ^ 2
    var area = PI * radius * radius
    // 打印信息
    fmt.Printf("The circle with radius %.2f has an area of %.2f
\n", radius, area)
} // main
```

运行结果如下

```
$ go run circle.go 5
[ruby-2.6.3p62]
The circle with radius 5.00 has an area of 78.50
$ go run circle.go ads
[ruby-2.6.3p62]
Error message: strconv.ParseFloat: parsing "ads": invalid syntax
exit status 1
```

- 在包一级声明语句声明的名字可在整个包对应的每个源文件中访问，而不是仅仅在其声明语句所在的源文件中访问。相比之下，局部声明的名字就只能在函数内部很小的范围被访问。
- 一个函数的声明由一个函数名字、参数列表（由函数的调用者提供参数变量的具体值）、一个可选的返回值列表和包含函数定义的函数体组成。如果函数没有返回值，那么返回值列表是省略的。

circle.go

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

const PI = 3.14

func main(){
    var radiusString = os.Args[1]
    var radius,err = strconv.ParseFloat(radiusString,32)
    if err != nil{
    fmt.Fprintf(os.Stderr, "Error message: %v\n",err)
        os.Exit(1)
    } // if
    var area = calculateArea(radius)
    fmt.Printf("The circle with radius %.2f has an area of %.2f
\n", radius, area)
} // main

func calculateArea(r float64) float64{
    var area =  PI * r * r
    return area
} // calculateArea
```

运行结果和之前一致

常量

常量使用关键字`const`定义，用于存储不会改变的数据。

存储在常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

常量的格式为

```
const 常量名字 类型 = 表达式
```

可以省略类型，编译器会自动推断

```
// 显式定义
const str string = "abc"
// 隐式定义
const str = "abc"
```

- 常量的值必须是能够在编译时就能够确定的。可以在其赋值表达式中涉及计算过程，但是所有用于计算的值必须在编译期间就能获得。也就是说常量定义不可以涉及自定义函数，只能用内置函数。

变量

声明变量的一般形式是使用 `var` 关键字，格式为

```
var 变量名字 类型 = 表达式
```

类型和表达式可省略一个

- 省略类型：根据表达式自动推断

```
var i, j, k int                 // int, int, int
var b, f, s = true, 2.3, "four" // bool, float64, string
```

- 省略表达式：零值初始化该变量
  - 数值类型（包括complex64/128）为 0
  - 布尔类型为 false
  - 字符串为 ""（空字符串）
  - 以下几种类型为 nil：
    - var a *int
    - var a []int
    - var a map[string] int
    - var a chan int
    - var a func(string) int
    - var a error // error 是接口

```
var a int
```

```
var b bool
var str string

// 可以改写成如下形式
var (
    a int
    b bool
    str string
)

// 一般用于声明全局变量
```

可用于函数返回值的声明

简短变量声明

只能用于声明和初始化**局部变量**！简短声明语法 `:=`
变量的类型会根据表达式自动推导

```
名字 := 表达式
a := 1
b := true
```

简短变量声明被广泛用于大部分的局部变量的声明和初始化。var形式的声明语句往往是用于：

- 需要显式指定变量类型地方
- 因为变量稍后会被重新赋值而初始值无关紧要的地方。

```
i := 100                  // an int
var i float64 = 100       // a float64
```

⚠️：不要搞混`:=`和`=`，`:=`是用于变量声明，`=`用于赋值
简短变量声明亦可用于函数返回值的声明

值类型和引用类型

程序中用到的内存用箱子(“字”)来表示，所有字都具有32bits(4bytes)或64bits(8bytes)的相同长度，使用相关的内存地址来进行表示(十六进制)

- int, float, bool和string这些基本类型都属于值类型，使用这些类型的变量直接指向存在内存中的值

Fig 4.1: Value type

像数组和结构这些复合类型也是值类型。
当使用等号`=`将一个变量的值赋值给另一个变量时，如:`j = i`，实际上是在内存中将 i 的值进行了拷贝

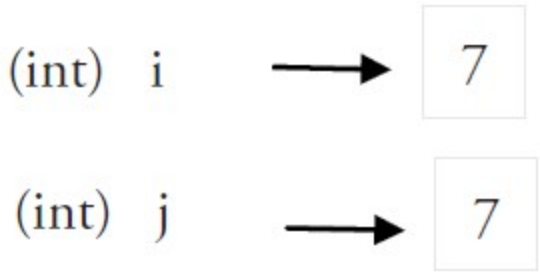


Fig 4.2: Assignment of value types

可以通过`&i`来获取变量`i`的内存地址，值类型的变量的值存储在栈中。

- 更复杂的数据通常会需要使用多个字，这些数据一般使用引用类型(reference type)保存。

一个引用类型的变量r1存储的是r1的**值所在的内存地址**，或内存地址中**第一个字**所在的位置，这个内存地址被称之为指针。

Fig 4.3: Reference types and assignment

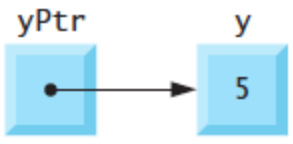
同一个引用类型的指针指向的多个字可以是在连续的内存地址中也可以将这些字分散存放在内存中，每个字都指示了下一个字所在的内存地址。
当使用赋值语句 `r2 = r1` 时，只有引用(也就是地址)被复制。如果r1的值被改变了，那么这个值的所有引用都会指向被修改后的内容，r2也会受到影响。指针属于引用类型，其它的引用类型还包括 slices，maps和 channel。

指针

Go语言允许控制特定集合的数据结构、分配的数量以及内存访问模式。一个指针变量可以**指向任何一个值的内存地址**。
`&`是获取变量的内存地址，将产生一个指向该变量的指针。Go语言的取地址符是`&`，放到一个变量前使用就会返回相应变量的内存地址。
`*`是找指针指向地址上所存储的值。 `*`是一个类型更改器。使用一个指针引用一个值被称为间接引用。

```
x := 1
// p存了x的地址
// 指针对应的数据类型是*int，指针被称之为“指向int类型的指针”
p := &x         // p, of type *int, points to x
// 打印p这个地址上的值
fmt.Println(*p) // "1"
// 把p这个地址上对应的值改为2
*p = 2          // equivalent to x = 2
// x的值被改动了
fmt.Println(x)  // "2"
```

- 任何类型的指针的零值都是nil。如果p指向某个有效变量，那么`p != nil`测试为真。
- 指针之间也是可以进行相等测试的，只有当它们指向同一个变量或全部是nil时才相等。
- 指针的格式化标识符为 `%p`
- 在书写表达式类似 `var p *type` 时，切记在*号和指针名称间留有一个空格
- 对于任何一个变量 var， 如下表达式都是正确的：`var == *(&var)`。



new函数

另一个创建变量的方法是调用用内建的new函数。表达式new(T)将创建一个T类型的匿名变量，初始化为T类型的零值，然后返回**变量地址**，返回的指针类型为`*T`。

```
p := new(int)   // p, *int 类型, 指向匿名的 int 变量
fmt.Println(*p) // "0"
*p = 2          // 设置 int 匿名变量的值为 2
fmt.Println(*p) // "2"
```

- 和上面的区别就是不需要声明临时变量名。
- 每次调用new函数都是返回一个**新的变量的地址**。

赋值

除了之前已经使用过的，还有结构体字段赋值

```
person.name = "Darren"
```

注意：`a++`是语句不是表达式不能写成`x=a++`这种

元组赋值

允许同时更新多个变量的值，比如可以互换2个变量的值，或者进行一些其它操作

```
x, y = y, x
a[i], a[j] = a[j], a[i]
x, y = y, x%y
x, y = y, x+y
i, j, k = 2, 3, 5
```

- 赋值语句右边的所有表达式将会先进行求值，然后再统一更新左边对应变量的值

处理多返回值的函数

```
v, ok = m[key]             // map lookup map查找
v, ok = x.(T)              // type assertion 类型断言
v, ok = <-ch               // channel receive 通道接收
```

- 这几个并不一定是产生两个结果，也可能只产生一个结果，如果出错返回零值

可以用下划线空白标识符`_`来丢弃不需要的值。

```
_, err = io.Copy(dst, src) // 丢弃字节数
_, ok = x.(T)              // 只检测类型，忽略具体值
```

类型

一个类型声明语句**创建了一个新的类型名称**，和现有类型具有相同的底层结构。新命名的类型提供了一个方法，用来**分隔不同概念的类型**，这样**即使它们底层类型相同也是不兼容**的。

```
type 类型名字 底层类型
```

使用 type 关键字可以定义你自己的类型，可以定义一个已经存在的类型的“别名”(并不是真正意义上的别名，因为使用这种方法定义之后的类型可以拥有更多的特性，且在类型转换时必须**显式转换**)

```
// int是变量a的底层类型
type IZ int
var a IZ = 5

// 还可以一次创建多个类型
type (
    IZ int
    FZ float64
    STR string
)
// 每个值都必须在经过编译后属于某个类型
```

底层数据类型决定了内部结构和表达方式，也决定是否可以像底层类型一样对内置运算符的支持。

类型之间显式转换

```
type Celsius float64    // 摄氏温度
type Fahrenheit float64 // 华氏温度
var c Celsius
var f Fahrenheit

fmt.Println(c == f)          // compile error: type mismatch
```

```
7 fmt.Println(c == Celsius(f)) // "true"!
```

Celsius(f)就是在使用显式转换，把Fahrenheit类型的f转换为Celsius类型

参考资料

Go语言圣经（中文版）
https://books.studygolang.com/gopl-zh/index.html
GO语言笔试题：Printf()、Sprintf()、Fprintf()函数的区别用法是什么?
https://www.dailybtc.cn/go语言笔试题：printf、sprintf、fprintf函数的区别用法是什么？/
Go 学习笔记：Println 与 Printf 的区别，以及 Printf 的详细用法
https://blog.csdn.net/zgh0711/article/details/78843361
菜鸟教程——Go 语言教程
https://www.runoob.com/go/go-tutorial.html
《Go入门指南》
https://github.com/unknwon/the-way-to-go_ZH_CN
golang string、int、int64 float 互相转换
https://blog.csdn.net/qwdafedv/article/details/80453511

Golang 学习笔记03

2020/07/08

cyc/Golang 学习笔记/Golang 学习笔记03

基础数据类型和运算

运算符

Go语言的一些运算符

```
二元比较运算符
==      等于
!=      不等于
<       小于
<=      小于等于
>       大于
>=      大于等于

一元的加法和减法运算符
+       一元加法 (无效果)
-       负数

位操作符
&       位运算 AND
|       位运算 OR
^       位运算 XOR
&^      位清空 (AND NOT)
<<      左移
>>      右移
```

运算符优先级递减顺序

```
*       /       %       <<      >>      &       &^
+       -       |       ^
```

```
3 ==      !=      <       <=        >       >=
4 &&
5 ||
```

- 同一个优先级，使用左优先结合规则，但是使用括号可以明确优先顺序
- 取模运算符`%`仅用于整数间的运算

整型

- `int32`表示32位有符号整数
- `uint64`表示64位**无符号**整数
- 共有8、16、32、64四种位数的整数
- 不同类型不能进行运算，如`int32的数字`+`int16的数字`会报错！(type mismatch) 可以都显示转化为常见类型比如int(`int32的数字`)+int(`int16的数字`)
- 算术运算的结果，不管是有符号或者是无符号的，如果需要更多的bit位才能正确表示的话，就说明计算结果是**溢出**了。**超出的高位的bit位部分将被丢弃**。如果原始的数值是**有符号类型，而且最左边的bit位是1的话，那么最终结果可能是负**的。

```go
var u uint8 = 255
fmt.Println(u, u+1, u*u) // "255 0 1"
var i int8 = 127
fmt.Println(i, i+1, i*i) // "127 -128 1"
```

- 当使用fmt包打印一个数值时，我们可以用`%d`、`%o`或`%x`参数控制输出的进制格式

```go
o := 0666
fmt.Printf("%d %[1]o %#[1]o\n", o) // "438 666 0666"
x := int64(0xdeadbeef)
fmt.Printf("%d %[1]x %#[1]x %#[1]X\n", x)
// Output:
// 3735928559 deadbeef 0xdeadbeef 0XDEADBEEF
```

  - 通常Printf格式化字符串包含多个`%`参数时将会包含对应相同数量的额外操作数，但是`%`之后的`[1]`副词告诉Printf函数再次使用第一个操作数。
  - `%`后的#副词告诉Printf在用`%o`、`%x`或`%X`输出时生成0、0x或0X前缀。

浮点数

- 两种精度的浮点数，`float32`和`float64`

- math.IsNaN用于测试一个数是否是非数NaN，math.NaN则返回非数对应的值

复数

- 两种精度的复数类型，`complex64`和`complex128`，分别对应float32和float64两种浮点数精度
- 创建与使用

```
// 创建一个complex number x，其为1+2i
var x complex64 = complex(1,2)
// 也可以简写成
x := 1 + 2i
// 2i就是0+2i，实数部分为0
// 得复数的实部
real(x) // 为1
// 得复数的虚部
imag(x) // 为2
```

- ==来比较只有当实部和虚部都相等时才会为true

布尔型

- 可以和&&（AND）和||（OR）操作符结合
- 有短路行为：如果运算符左边值已经可以确定整个布尔表达式的值，那么运算符右边的值将不再被求值

字符串

- len(string)可以得到字符串长度

常见的ASCII控制代码的转义方式

```
\a      响铃
\b      退格
\f      换页
\n      换行
\r      回车
\t      制表符
\v      垂直制表符
\'      单引号（只用在 '\'' 形式的rune符号面值中）
\"      双引号（只用在 "..." 形式的字符串面值中）
\\      反斜杠
```

前缀、后缀与包含

`HasPrefix` 判断字符串 `s` 是否以 `prefix` 开头

```
strings.HasPrefix(s, prefix string) bool
```

`HasSuffix` 判断字符串 `s` 是否以 `suffix` 结尾

```
strings.HasSuffix(s, suffix string) bool
```

`Contains` 判断字符串 `s` 是否包含 `substr`：

```
strings.Contains(s, substr string) bool
```

preSuf.go

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
    var testStr = "This is a test string"
    fmt.Printf("Does the string \"%s\" has prefix \"Th\"?\n", testStr)
    fmt.Printf("%t\n", strings.HasPrefix(testStr, "Th"))
    fmt.Printf("Does the string \"%s\" has prefix \"th\"?\n", testStr)
    fmt.Printf("%t\n", strings.HasPrefix(testStr, "th"))
    fmt.Printf("Does the string \"%s\" has suffix \"ng\"?\n", testStr)
    fmt.Printf("%t\n", strings.HasSuffix(testStr, "ng"))
    fmt.Printf("Does the string \"%s\" contains \"es\"?\n", testStr)
    fmt.Printf("%t\n", strings.Contains(testStr, "es"))
```

```
} // main
```

运行结果

```
Does the string "This is a test string" has prefix "Th"?
true
Does the string "This is a test string" has prefix "th"?
false
Does the string "This is a test string" has suffix "ng"?
true
Does the string "This is a test string" contains "es"?
true
```

与数字的转换

string到int

```
// string到int
int,err := strconv.Atoi(string)

// string到int64
int64, err := strconv.ParseInt(string, 10, 64)
// 第二个参数为基数（2~36），
// 第三个参数位大小表示期望转换的结果类型，其值可以为0, 8, 16, 32和64，
// 分别对应 int, int8, int16, int32和int64
```

- atoi：ascii to integer是把字符串转换成整型数的一个函数

int到string

```
// int到string
string := strconv.Itoa(int)
// 等价于
string := strconv.FormatInt(int64(int),10)

// int64到string
string := strconv.FormatInt(int64,10)
```

```
// 第二个参数为基数，可选2~36
// 对于无符号整形，可以使用FormatUint(i uint64, base int)
```

string到float

```
// string到float64
float,err := strconv.ParseFloat(string,64)

// string到float32
float,err := strconv.ParseFloat(string,32)
```

float到string

```
// float到string
string := strconv.FormatFloat(float32,'E',-1,32)
string := strconv.FormatFloat(float64,'E',-1,64)
```

- `strconv.FormatFloat(f float64, fmt byte, prec int, bitSize int) string` 将 64 位浮点型的数字转换为字符串，其中 `fmt` 表示格式(其值可以是 `'b'`、`'e'`、`'f'` 或 `'g'`)，`prec` 表示精度(如果prec 为-1，则代表使用最少数量的，但又必需的数字来表示f)，`bitSize` 则使用 32 表示 float32，用 64 表示 float64。

```
// 'b' (-ddddp±ddd，二进制指数)
// 'e' (-d.ddddе±dd，十进制指数)
// 'E' (-d.ddddE±dd，十进制指数)
// 'f' (-ddd.dddd，没有指数)
// 'g' ('e':大指数，'f':其它情况)
// 'G' ('E':大指数，'f':其它情况)
```

常量

基础的声明已在Golang 学习笔记02中列出，这里列举了更多有关常量的应用

批量声明

- 常量可和变量一样批量声明

```go
package main

import "fmt"

const (
    a = "string"
    b = 2
    c = true
)

func main() {
    fmt.Printf("a is %s\n", a)
    fmt.Printf("b is %d\n", b)
    fmt.Printf("c is %t\n", c)
} // main
```

```
$ go run test.go
[ruby-2.6.3p62]
a is string
b is 2
c is true
```

- 如果是批量声明的常量，除了第一个外其它的常量右边的初始化表达式都可以省略，如果省略初始化表达式则表示使用前面常量的初始化表达式写法。

```go
package main

import "fmt"

const (
    a = "test string"
    b
    c
)

func main() {
    fmt.Printf("a is %s\n", a)
    fmt.Printf("b is %d\n", b)
```

```
    fmt.Printf("c is %t\n", c)
} // main
```

```
$ go run test.go
[ruby-2.6.3p62]
a is test string
b is %!d(string=test string)
c is %!t(string=test string)
```

Go语言圣经中的例子

```
const (
    a = 1
    b
    c = 2
    d
)
fmt.Println(a, b, c, d) // "1 1 2 2"
```

iota 常量生成器

- 在一个const声明语句中，在第一个声明的常量所在的行，**iota将会被置为0**，然后在每一个有常量声明的行加一。
- `iota` 也可以用在表达式中，如：`iota + 50`。在每遇到一个新的常量块或单个常量声明时，`iota` 都会重置为 0（**简单地讲，每遇到一次const关键字，iota就重置为0**）。

```
type Weekday int

const (
    Sunday Weekday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)
```

```
12 // 其他几个都会被设置成Weekday类型，并且Monday的值为1， Tuesdays的值为2，
   以此类推。。。
```

参考资料

Go语言圣经（中文版）
https://books.studygolang.com/gopl-zh/index.html
《Go入门指南》
https://github.com/unknwon/the-way-to-go_ZH_CN
golang string、int、int64 float 互相转换
https://blog.csdn.net/qwdafedv/article/details/80453511
Go基础系列：数据类型转换(strconv包)
https://www.cnblogs.com/f-ck-need-u/p/9863915.html

Golang 学习笔记04

2020/07/09

cyc/Golang 学习笔记/Golang 学习笔记04

复合数据类型——数组、切片和map

数组

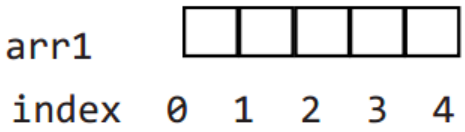
- 数组的**长度**是**固定**的
- 数组内数据有**相同且唯一的类型**

数组的声明和初始化

```
var 数组名字 [长度]数据类型

// 也可以在声明时初始化
var 数组名字 [长度]数据类型 = [长度]数据类型{数据1,数据2,数据3,...}
// 简短声明
数组名字 := [长度]数据类型{数据1,数据2,数据3,...}
```

如

```
var arr1 [5]int  // 和 arr1 := [5]int{} 效果一致

// 没有被初始化的元素会被赋予该数据类型的零值，对于整型来说就是0
var arr1 [5]int = [5]int{1, 2, 4}
// 比如上方数组arr1中的arr1[3]就为0
// 也可以简写成
var arr1 = [5]int{1, 2, 4}  // arr1的类型编译器会自动推断
// 简短声明
arr1 := [5]int{1, 2, 4}
```



for-range循环

- 打印数组元素(element)与索引(index)

```
// Print the indices and elements.
for i, v := range arr1 {
    fmt.Printf("%d %d\n", i, v)
}
```

长度自动推断

- 使用`...`根据初始化值的个数自动推断数组长度

```
q := [...]int{1, 2, 3}
fmt.Printf("%T\n", q) // "[3]int"
```

key: value 语法

- 使用key: value来指定数组中被赋值的内容

```
var arrKeyValue = [5]string{3: "Chris", 4: "Ron"}
```

```
Person at 0 is
Person at 1 is
Person at 2 is
Person at 3 is Chris
Person at 4 is Ron
```

演示

test.go

```go
package main

import "fmt"

func main() {
    testArr1 := [...]int{2, 4, 34, 32, 17}
    var testArr2 = [4]int{2, 4}   // 和 testArr2 := [4]int{2, 4}
    testArr3 := [5]int{1: 12, 4: 23}   // 和 var testArr3 = [5]int{1: 12, 4: 23} 一样
    var testArr4 [3]int
    for i, x := range testArr1 {
        fmt.Printf("The element at index %d in testArr1 is %d\n", i, x)
    } // for
    fmt.Println("------------------------------------------------")
    for i, x := range testArr2 {
        fmt.Printf("The element at index %d in testArr2 is %d\n", i, x)
    } // for
    fmt.Println("------------------------------------------------")
    for i, x := range testArr3 {
        fmt.Printf("The element at index %d in testArr3 is %d\n", i, x)
    } // for
    fmt.Println("------------------------------------------------")
    for i, x := range testArr4 {
        fmt.Printf("The element at index %d in testArr4 is %d\n", i, x)
    } // for
} // main
```

运行结果如下

```
$ go run test.go
[ruby-2.6.3p62]
The element at index 0 in testArr1 is 2
```

```
The element at index 1 in testArr1 is 4
The element at index 2 in testArr1 is 34
The element at index 3 in testArr1 is 32
The element at index 4 in testArr1 is 17
------------------------------------------------
The element at index 0 in testArr2 is 2
The element at index 1 in testArr2 is 4
The element at index 2 in testArr2 is 0
The element at index 3 in testArr2 is 0
------------------------------------------------
The element at index 0 in testArr3 is 0
The element at index 1 in testArr3 is 12
The element at index 2 in testArr3 is 0
The element at index 3 in testArr3 is 0
The element at index 4 in testArr3 is 23
------------------------------------------------
The element at index 0 in testArr4 is 0
The element at index 1 in testArr4 is 0
The element at index 2 in testArr4 is 0
```

注意

- Go 语言中的数组是一种**值类型**(不像C中是指向首元素的指针)，所以可以通过`new()`来创建：`var arr1 = new([5]int)`。
- 那么这种方式和 `var arr2 [5]int` 的区别是什么呢?
  - arr1 的类型是 `*[5]int`，而 arr2的类型是 `[5]int`。
  - 这样的结果就是当把一个数组赋值给另一个时，需要**再做一次数组内存的拷贝操作**。

test2.go

```go
package main

import "fmt"

func main() {
    // arr1是一个指向一个5个int的指针
    var arr1 = new([5]int)
    arr2 := *arr1
    arr3 := arr1
    arr2[0] = 18
```

```
    arr2[2] = 20
    fmt.Printf("arr1 is: %T\n", arr1)
    fmt.Printf("*arr1 is: %T\n", *arr1)
    fmt.Printf("arr2 is: %T\n", arr2)
    fmt.Println("-----------------------------------")
    fmt.Println(arr1)
    fmt.Println(*arr1)
    fmt.Println(arr2)
    fmt.Println(arr3)
    fmt.Println("-----------------------------------")
    fmt.Println("arr1")
    for _, x := range arr1 {
        fmt.Printf("%d\n", x)
    } // for
    fmt.Println("-----------------------------------")
    fmt.Println("arr2")
    for _, x := range arr2 {
        fmt.Printf("%d\n", x)
    } // for
    fmt.Println("-----------------------------------")
    // change the third element in arr1 to 8
    arr1[2] = 8
    (*arr1)[3] = 9999
    // and the last element in arr3 to 78
    arr3[4] = 78
    fmt.Println("arr1")
    for _, x := range arr1 {
        fmt.Printf("%d\n", x)
    } // for
    fmt.Println("-----------------------------------")
    fmt.Println("arr2")
    for _, x := range arr2 {
        fmt.Printf("%d\n", x)
    } // for
    fmt.Println("-----------------------------------")
    fmt.Println("arr3")
    for _, x := range arr3 {
        fmt.Printf("%d\n", x)
    } // for
} // main
```

注：第33行不能写成`*arr1[3]=9999`必须写成`(*arr1)[3]=9999`否则会报错！

- *arr1[3] = 9999
- 乍一看这句代码并无任何问题。因为arr1是数组指针，*arr1就是数组本身，而*arr1[3]自然就是数组的第一个元素，但代码的运行结果是一个错误！
- 这是因为在go语言中*寻址运算符和[]中括号运算符的优先级是不同的！
- [ ]中括号是初等运算符
- *寻址运算符是单目运算符
- 初等运算符的优先级是大于单目运算符的，因此先参与计算的是arr1[3]。arr1[3]其实就是数组的第四个元素，就是数字0。数字0必然是int类型，而不是一个地址，因此针对数字0使用*寻址运算符自然也就发生了错误。
- 解决问题的办法很简单，就是添加一个小括号就可以了。
  即：(*arr1)[3] = 9999
- 使用中`arr1[3] = 9999`这种写法更常见！

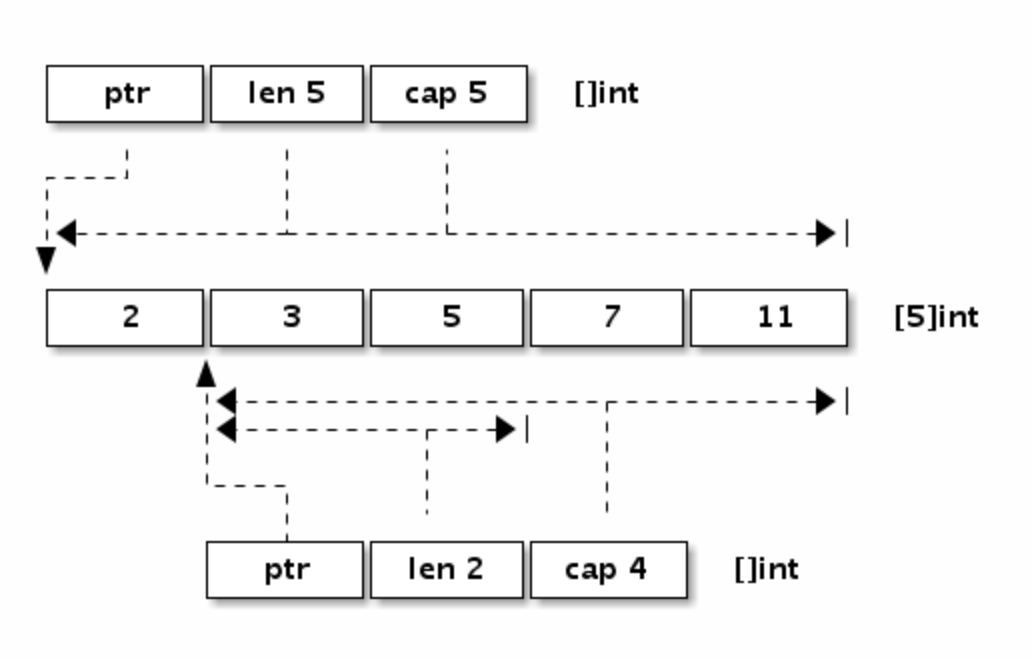
运行结果

```
 $ go run test2.go
[ruby-2.6.3p62]
arr1 is: *[5]int
*arr1 is: [5]int
arr2 is: [5]int
-----------------------------------
&[0 0 0 0 0]
[0 0 0 0 0]
[18 0 20 0 0]
&[0 0 0 0 0]
-----------------------------------
arr1
0
0
0
0
0
-----------------------------------
arr2
18
0
20
```

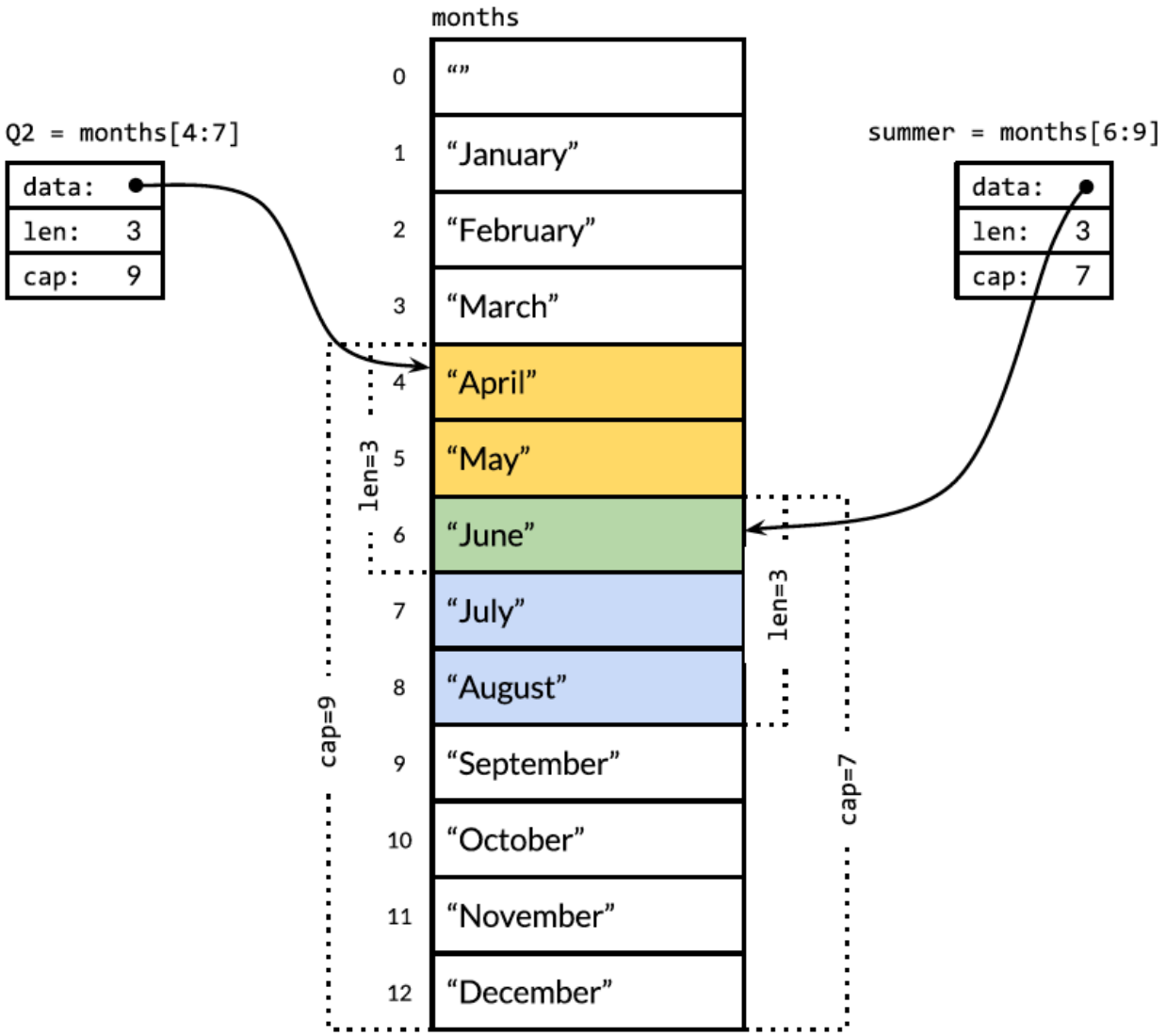
```
0
0
------------------------------------
arr1
0
0
8
9999
78
------------------------------------
arr2
18
0
20
0
0
------------------------------------
arr3
0
0
8
9999
78
```

切片(slice)

- 是对数组(称之为**相关数组**)一个连续片段的引用，所以切片是一个**引用类型**(更类似于C中的数组，或Python中的list）。这个片段可以是整个数组，或者是由起始和终止索引标识的一些项的子集。
- 切片没有固定长度，它的长度可以在运行时修改，最小为0最大为相关数组的长度。可以说切片是一个**长度可变的数组**。
- 因为切片是引用，所以它们不需要使用额外的内存并且比使用数组更有效率，所以在Go代码中切片比数组更常用。
- 切片是可索引的，并且可以由`len()`函数获取**长度**。
- 切片提供了计算容量的函数`cap()`可以测量切片最长可以达到多少，也就是切片指向的内存空间的最大**容量**。
- slice之间**不能比较**，因此我们不能使用==操作符来判断两个slice是否含有全部相等元素。不过标准库提供了bytes.Equal函数来判断两个**字节型**slice是否相等，对于其他类型的slice要自己写方法来比较。slice唯一合法的比较操作是**和nil比较**。
- `x := []int{2,3,5,7,11}`和`y := x[1:3]`两个切片对应的内存结构



    - 一个slice由三个部分构成：指针、长度和容量。
        - 指针指向第一个slice元素对应的底层数组元素的地址，要注意的是slice的第一个元素并不一定就是数组的第一个元素。
        - 长度对应slice中元素的数目；长度不能超过容量。
        - 容量一般是从slice的开始位置到底层数据的结尾位置。
- 多个slice之间可以共享底层的数据，并且引用的数组部分区间可能重叠。下图显示了表示一年中每个月份名字的字符串数组，还有重叠引用了该数组的两个slice。



切片的声明和定义

```
var 切片名字 []数据类型     // 切片不需要说明长度
// 一个切片在未初始化之前默认为 nil，长度为 0。
// 切片初始化格式
var 切片名字 []数据类型 = 某数组[start:end]
// 表示该切片是由某数组从start索引到end-1索引之间的元素构成的子集
```

切片的其他定义方式

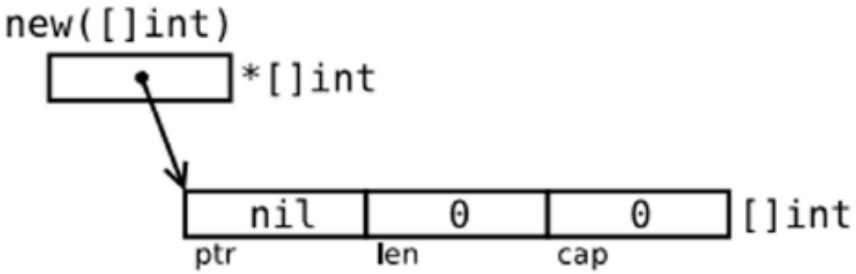
```
a []int                 // nil切片，和 nil 相等，一般用来表示一个不存在的切片
b = []int{}             // 空切片，和 nil 不相等，一般用来表示一个空的集合
```

```
c = []int{1, 2, 3}     // 有3个元素的切片, len和cap都为3
d = c[:2]              // 有2个元素的切片, len为2, cap为3
e = c[0:2:cap(c)]      // 有2个元素的切片, len为2, cap为3
f = c[:0]              // 有0个元素的切片, len为0, cap为3
g = make([]int, 3)     // 有3个元素的切片, len和cap都为3
h = make([]int, 2, 3) // 有2个元素的切片, len为2, cap为3
i = make([]int, 0, 3) // 有0个元素的切片, len为0, cap为3
```

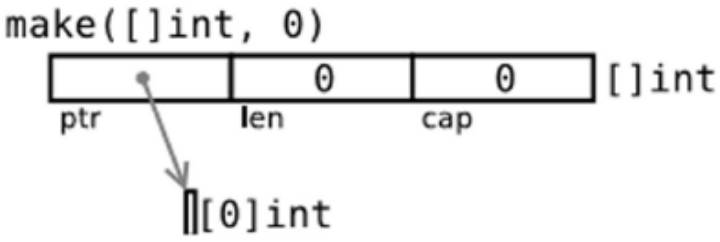
new()和make()的区别

它们都在堆上分配内存，但是它们的行为不同，适用于不同的类型。

- new(T)为每个新的类型T分配一片内存，初始化为0并且返回类型为*T的内存地址。这种方法返回一个指向类型为T，值为0的地址的**指针**，它*适用于值类型如数组和结构体*。
- make(T)返回一个类型为T的**初始值**，它*只适用于3种内建的引用类型：切片、map和channel*。




```
var p *[]int = new([]int) // *p == nil; with len and cap 0
// 或者是如下形式
p := new([]int)
```




```
// 切片已经被初始化，但是指向一个空的数组
p := make([]int, 0)
```

test.go

```
package main

import "fmt"

func main() {
    var p1 = new([]int)
    var p2 = make([]int, 0)
    fmt.Printf("p1 has type %T\n", p1)
    fmt.Println(p1)
    fmt.Printf("p2 has type %T\n", p2)
    fmt.Println(p2)
    *p1 = append(*p1, 2)
    p2 = append(p2, 3)
    fmt.Printf("The first element in p1 is %d.\nThat of p2 is %
d.\n", (*p1)[0], p2[0])
} // main
```

运行结果

```
$ go run test.go
[ruby-2.6.3p62]
p1 has type *[]int
&[]
p2 has type []int
[]
The first element in p1 is 2.
That of p2 is 3.
```

append()和copy()

append()用于向切片中增加元素。如果想增加切片的容量，可以创建一个新的更大的切片并把原分片的内容都拷贝过来，这时候会用到copy()。

append()

```
var numbers []int
```

```
// 可以往空切片中添加内容
numbers = append(numbers, 0)
// 同时添加多个内容
numbers = append(numbers, 2,3,4)
```

- 内置的append函数可能使用复杂的内存扩展策略。因此，通常我们不知道append调用是否导致了内存的重新分配，也不能确认新的slice和原始的slice是否引用的是相同的底层数组空间。同样，我们不能确认在原先的slice上的操作是否会影响到新的slice。因此，通常是**将append返回的结果直接赋值给输入的slice变量**
- 如果想去掉 slice1 的最后一个元素，只要 `slice1 = slice1[:len(slice1)-1]`。

copy()

```
// 第一个参数是要复制的目标slice，第二个参数是源slice
// 目标和源的位置顺序和dst = src赋值语句是一致的
copy(numbers1,numbers)    // 拷贝numbers的内容到numbers1
```

字典(map)

- Map是一种**无序**的键值对的集合。Map最重要的一点是通过key来快速检索数据，key类似于索引，指向数据的值。
- 在声明的时候不需要知道map的长度，map是可以**动态增长**的，不存在固定长度或者最大限制。

key

- 可以是任意可以用`==`或者`!=`操作符比较的类型，比如 string、int、float。
- 所以数组、切片和结构体不能作为key(含有数组切片的结构体不能作为key，只包含内建类型的struct是可以作为key的)
- 但是指针和接口类型可以
- 如果要用结构体作为 key 可以提供`Key()`和`Hash()`方法，这样可以通过结构体的域计算出唯一的数字或者字符串的 key。

value

- 可以是任意类型

声明和初始化

声明

```
var map名称 map[keytype]valuetype
```

- 未初始化的map的值是nil

初始化

创建空map

```
ages := map[string]int{}

// 或者使用make
ages := make(map[string]int)

// 可以标明map的初始容量(capacity)
map1 := make(map[string]float32, 100)
```

初始化并赋值

```
ages := map[string]int{
    "alice":   31,
    "charlie": 34,
}

// 或者使用make
ages := make(map[string]int)
ages["alice"] = 31
ages["charlie"] = 34
```

- map是**引用类型**的，内存用make方法来分配，make会返回map的引用。

map的使用

- 可以用`len(map)`方法可以获得map中的pair数目
- 删除元素

```
delete(ages, "alice") // remove element ages["alice"]
```

- 简短赋值语法可以用在map的value上

```
ages["alice"]++
```

```
ages["alice"] += 2
```

- 遍历map

在Golang 学习笔记01的检测重复行时用到过map的遍历

```
for name, age := range ages{
    fmt.Printf("Name: %s\tAge: %d\n",name, age)
}
```

⚠ Map的迭代**顺序是不确定的**，遍历的顺序是随机的，每一次遍历的顺序都不相同。

参考资料

Go语言圣经（中文版）
https://books.studygolang.com/gopl-zh/index.html
《Go入门指南》
https://github.com/unknwon/the-way-to-go_ZH_CN
Go语言高级编程(Advanced Go Programming)
https://chai2010.cn/advanced-go-programming-book/
菜鸟教程——Go 语言教程
https://www.runoob.com/go/go-tutorial.html
Go语言学习笔记07--数组指针、切片指针与结构体指针
https://blog.csdn.net/u013792921/article/details/84565336

Golang 学习笔记05

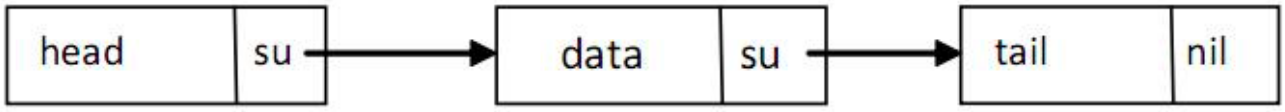
2020/07/11

cyc/Golang 学习笔记/Golang 学习笔记05

复合数据类型——结构体、JSON及文本和HTML模版

结构体(struct)

- 复合类型
- 由零个或多个任意类型的值聚合成的实体
- 组成结构体类型的那些数据称为**字段(fields)**。每个字段都有一个类型和一个名字；在一个结构体中，字段名字必须是唯一的。
- 如果结构体的全部成员(全部字段)都是可以比较的，那么结构体也是可以比较的，那样的话两个结构体将可以使用==或!=运算符进行比较。
- 一个命名为S的结构体类型将不能再包含S类型的成员：因为一个聚合的值**不能包含它自身**。但是S类型的结构体可以**包含*S指针类型**的成员，这可以让我们创建递归的数据结构




```
type Node struct {
    data    float64
    su      *Node
}
```

定义

```
type 结构体的名字 struct {
    字段1 字段1的类型
    字段2 字段2的类型
    ...
}
```

- 简单的结构体也可以这样定义

```
type T struct {a, b int}
```

- 结构体变量的成员可以通过**点操作符**访问。无论变量是一个结构体类型还是一个结构体类型指针，都使用同样的选择器符(selector-notation)来引用结构体的字段：

```
type myStruct struct { i int }
var v myStruct    // v是结构体类型变量
var p *myStruct   // p是指向一个结构体类型变量的指针
v.i
p.i
```

创建结构体

- 使用new()来创建
  - new(strucName)
  - 和&structName{}一样，其实&structName{}就是调用了new(strucName)的方法
- 返回一个指向结构体的**指针**
- 此时结构体字段的值是它们所属类型的零值

```
type T struct {a, b int}
// t是一个指向T的指针
var t *T = new(T)  // 简单声明 t := new(T)
// 用点操作符来访问
t.a = 3
t.b = 5
// 或者写成
var t = new(T)
*t = T{3, 5}
// 再或者写成
var t = &T{3, 5}
```

演示结构体的定义和初始化

struct.go

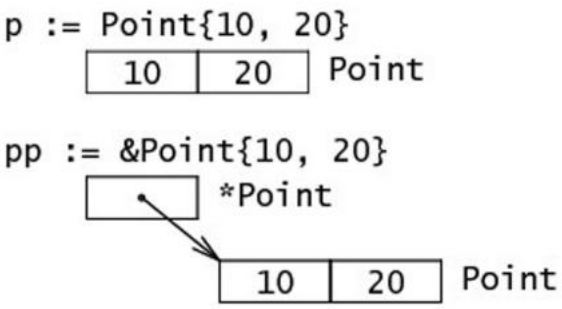
```go
package main

import "fmt"

type animal struct {
    name       string
    animalType string
    weight     int
    age        int
}

func addWeight1(a animal, w int) animal {
    a.weight += w
    return a
} // addWeight1

// 使用指针传入和返回结构体提高效率
func addWeight2(a *animal, w int) *animal {
    a.weight += w
    return a
} // addWeight2

func main() {
    // animal1的类型是animal
    var animal1 animal = animal{"lion", "feline", 230, 2}
    fmt.Println(animal1)
    fmt.Printf("The type of animal1 is: %T\n", animal1)
    fmt.Printf("animal1 has name: %s\ttype:%s\tweight:%d\tage:%d
\n", animal1.name, animal1.animalType, animal1.weight, animal1.ag
e)
    // animal2的类型是*animal，其实&Type{}相当于调用new(Type)
    var animal2 = &animal{age: 3, name: "tiger", weight: 12}
    fmt.Println(animal2)
    fmt.Printf("The type of animal2 is: %T\n", animal2)
    // *aniaml2的*可以省略
    fmt.Printf("animal2 has name: %s\ttype:%s\tweight:%d\tage:%d
\n", animal2.name, (*animal2).animalType, animal2.weight, animal2
.age)
    // animal3的类型是*animal
    animal3 := new(animal)
```

```
    animal3.animalType = "canine"
    animal3.age = 3
    // *可以省略，但如果要加*号，括号必须要加
    (*animal3).name = "wolf"
    fmt.Println(animal3)
    fmt.Printf("The type of animal3 is: %T\n", animal3)
    fmt.Printf("animal3 has name: %s\ttype:%s\tweight:%d\tage:%d\n", animal3.name, animal3.animalType, (*animal3).weight, animal3.age)
    fmt.Println("------------------------------------------------------------------------")
    animal1 = addWeight1(animal1, 12)
    animal1.age = 0
    fmt.Printf("animal1 has name: %s\ttype:%s\tweight:%d\tage:%d\n", animal1.name, animal1.animalType, animal1.weight, animal1.age)
    animal1New := addWeight2(&animal1, 10)
    animal1New.age = 1
    fmt.Printf("animal1 has name: %s\ttype:%s\tweight:%d\tage:%d\n", animal1New.name, animal1New.animalType, animal1New.weight, animal1New.age)
} // main
```

运行结果

```
$ go run struct.go
[ruby-2.6.3p62]
{lion feline 230 2}
The type of animal1 is: main.animal
animal1 has name: lion  type:feline weight:230  age:2
&{tiger  12 3}
The type of animal2 is: *main.animal
animal2 has name: tiger type:   weight:12   age:3
&{wolf canine 0 3}
The type of animal3 is: *main.animal
animal3 has name: wolf  type:canine weight:0    age:3
------------------------------------------------------------------------
animal1 has name: lion  type:feline weight:242  age:0
```

```
animal1 has name: lion  type:feline weight:252  age:1
```



结构体嵌入和匿名成员

为了解决结构体嵌套语句的繁琐可以使用**匿名成员**。Go语言有一个特性让我们**只声明一个成员对应的数据类型而不指名成员的名字**，这类成员就叫匿名成员。
《Go语言圣经》中给出例子如下

```go
type Point struct {
    X, Y int
}

type Circle struct {
    Point      // 匿名成员，只有Point类型，没有名字
    Radius int
}

type Wheel struct {
    Circle    // 匿名成员，只有Circle类型，没有名字
    Spokes int
}

var w Wheel
w.X = 8            // equivalent to w.Circle.Point.X = 8
w.Y = 8            // equivalent to w.Circle.Point.Y = 8
w.Radius = 5       // equivalent to w.Circle.Radius = 5
w.Spokes = 20

// 结构体字面值并没有简短表示匿名成员的语法， 因此下面的语句都不能编译通过
w = Wheel{8, 8, 5, 20}                       // compile error: un
```

```
known fields
w = Wheel{X: 8, Y: 8, Radius: 5, Spokes: 20} // compile error: un
known fields

// 结构体字面值必须遵循形状类型声明时的结构，所以我们只能用下面的两种语法，它们
彼此是等价的
w = Wheel{Circle{Point{8, 8}, 5}, 20}

w = Wheel{
    Circle: Circle{
        Point:  Point{X: 8, Y: 8},
        Radius: 5,
    },
    Spokes: 20, // NOTE: trailing comma necessary here (and at Ra
dius)
}
```

JSON(JavaScript Object Notation)

- 一个JSON数组是一个有序的值序列，写在一个**方括号**中并以**逗号**分隔
- 一个JSON数组可以用于编码Go语言的数组和slice。
- 一个JSON对象是一个字符串到值的映射，写成以系列的name:value对形式，用**花括号**包含并以逗号分隔
- JSON的对象类型可以用于编码Go语言的map类型（key类型是字符串）和结构体。

```
boolean         true
number          -273.15
string          "She said \"Hello, BF\""
array           ["gold", "silver", "bronze"]
object          {"year": 1980,
                 "event": "archery",
                 "medals": ["gold", "silver", "bronze"]}
```

编码与解码

- 将Go语言转化为JSON叫做编码(marshaling)
- 编组通过调用json.Marshal函数完成
- 编码的逆操作是解码，对应将JSON数据解码为Go语言的数据结构，Go语言中一般叫unmarshaling，通过json.Unmarshal函数完成。

文本和HTML模板

- 一个模板是一个字符串或一个文件，里面包含了一个或多个由双花括号包含的`{{action}}`对象。
- 大部分的字符串只是按字面值打印，但是对于actions部分将触发其它的行为。

《Go语言圣经》给出的例子

```
const templ = `{{.TotalCount}} issues:
{{range .Items}}----------------------------------------
Number: {{.Number}}
User:   {{.User.Login}}
Title:  {{.Title | printf "%.64s"}}
Age:    {{.CreatedAt | daysAgo}} days
{{end}}`
```

- 这个模板先打印匹配到的issue总数，然后打印每个issue的编号、创建用户、标题还有存在的时间。
- 对于每一个action，都有一个**当前值**的概念，对应点操作符，写作“.”。当前值“.”最初被初始化为调用模板时的参数。
- 模板中`{{.TotalCount}}`对应action将展开为结构体中TotalCount成员以默认的方式打印的值。
- 模板中`{{range .Items}}`和`{{end}}`对应一个**循环action**，因此它们直接的内容可能会被展开多次，循环**每次迭代的当前值对应当前的Items元素的值**。
- 在一个action中，`|`操作符表示**将前一个表达式的结果作为后一个函数的输入**，类似于UNIX中管道的概念。
- 在Title这一行的action中，第二个操作是一个printf函数，是一个基于fmt.Sprintf实现的内置函数，所有模板都可以直接使用。
- 对于Age部分，第二个动作是一个自己写的daysAgo的函数。

函数——声明、参数传递、返回值和函数值

函数声明

- 函数声明包括函数名、形式参数列表、返回值列表(可省略)以及函数体。

```
func name(parameter-list) (result-list) {
    // body
}
```

按值传递(value)和按引用传递(reference)

- Go默认使用按**值传递**来传递参数，也就是传递参数的副本。函数接收参数副本之后，在使用变量的过程中可能对副本的值进行更改，但**不会影响到原来的变量**。
- 如果希望函数可以直接修改参数的值，而不是对参数的副本进行操作，需要将**参数的地址**(变量名前面添加&符号，比如 &variable)传递给函数，这就是按引用传递。
- 在函数调用时，像切片(slice)、字典(map)、接口(interface)、通道(channel)这样的引用类型都是**默认使用引用传递**。

多返回值

- 函数可以返回多个值
- 只要在返回值列表中声明就可以了
- `return`时用`,`隔开即可
- 如果有返回值不需要被使用可以将其赋予空白符(blank identifier): `_`
- 如果一个函数将所有的返回值都显示的变量名，那么该函数的return语句可以省略操作数。这称之为bare return。

bareReturn.go

```go
package main

import "fmt"

func HelloWorld() (message string, num int) {
    // 这里的message和num已经在上一行声明过了
    // 如果再次声明，编译的时候会报错
    message = "Hello world!"
    num = 3
    return
} // HelloWorld

func main() {
    msg, no := HelloWorld()
    fmt.Printf("%s\n%d\n", msg, no)
} // main
```

运行结果

```
$ go run bareReturn.go
  [ruby-2.6.3p62]
Hello world!
```

```
3 3
```

《Go语言圣经》给出的例子

```
// CountWordsAndImages does an HTTP GET request for the HTML
// document url and returns the number of words and images in it.
func CountWordsAndImages(url string) (words, images int, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        err = fmt.Errorf("parsing HTML: %s", err)
    return
    }
    words, images = countWordsAndImages(doc)
    return
}
func countWordsAndImages(n *html.Node) (words, images int) { /* ... */ }

// 按照返回值列表的次序，返回所有的返回值，在上面的例子中，每一个return语句等价于
return words, images, err
```

错误

- 函数可能有额外返回值来返回error
- error可以是nil或non-nil
  - 如果是nil的话就是函数运行成功
  - 如果是non-nil说明有错误

错误处理策略

常用的五种处理方式

第一种 返回错误

- 直接将错误返回

```
resp, err := http.Get(url)
if err != nil{
    return nil, err
}
```

- 构造新的错误信息返回给调用者

```
doc, err := html.Parse(resp.Body)
resp.Body.Close()
if err != nil {
    return nil, fmt.Errorf("parsing %s as HTML: %v", url,err)
}
```

第二种 有限再次尝试

- 如果错误的发生是偶然性的，或由不可预知的问题导致的，应当重新尝试失败的操作。
- 在重试时，需要限制重试的时间间隔或重试的次数，防止无限制的重试。

第三种 输出报错并退出

- 如果错误发生后，程序无法继续运行，我们就可以输出错误信息并结束程序。
- 这种策略**只应在main中执行**。

```
// (In function main.)
if err := WaitForServer(url); err != nil {
    fmt.Fprintf(os.Stderr, "Site is down: %v\n", err)
    os.Exit(1)
}
```

第四种 输出报错并继续

- 有时候，只需要输出错误信息就足够了，不需要中断程序的运行。
- 可以通过log包提供函数，或者标准错误流输出错误信息。

```
if err := Ping(); err != nil {
    // log包中的所有函数会为没有换行符的字符串增加换行符
    log.Printf("ping failed: %v; networking disabled",err)
```

```
}

if err := Ping(); err != nil {
    fmt.Fprintf(os.Stderr, "ping failed: %v; networking disabled
\n", err)
}
```

第五种 忽略报错

```
dir, err := ioutil.TempDir("", "scratch")
if err != nil {
    return fmt.Errorf("failed to create temp dir: %v",err)
}
// ...use temp dir…
os.RemoveAll(dir) // ignore errors; $TMPDIR is cleaned periodicall
y
```

- 尽管os.RemoveAll会失败，但上面的例子并没有做错误处理。这是因为操作系统会定期的清理临时目录。
- 正因如此，虽然程序没有处理错误，但程序的逻辑不会因此受到影响。
- 应该在每次函数调用后，都考虑错误处理。
- 当决定忽略某个错误时，你应该记录下意图。

文件结尾错误(EOF)

- EOF说明读到文件结尾了
- 该错误定义在io包中

```
if err == io.EOF {
     break // finished reading
 }
```

函数值

- 在Go中，函数被看作**第一类值**(first-class values)：函数像其他值一样，拥有类型，可以被赋值给其他变量，传递给函数，从函数返回。
- 函数类型的零值是nil。调用值为nil的函数值会引起panic错误。

func.go

```go
package main

import "fmt"

type people struct {
    name    string
    age     int
    job     string
    address string
}

func greeting(mode int) (msg string) {
    switch mode {
    case 1:
        msg = "Good morning"
    case 2:
        msg = "Hello world!"
    case 3:
        msg = "Hi, nice to meet you!"
    default:
        msg = "Hi!"
    } // switch
    return
} // greeting

func createStruct() *people {
    var p *people = new(people)
    *p = people{"Darren", 20, "Student", "Suzhou"}
    return p
} // createStruct

func greeting2(g func(int) string) {
    fmt.Println("--------------------------------------")
    for i := 0; i <= 5; i++ {
        fmt.Println(g(i))
    } // for
    fmt.Println("--------------------------------------")
} // greeting 2

```

```go
func main() {
    var g func(int) string
    if g != nil {
        fmt.Println(g(2))
    } else {
        // 此时g的值是nil
        // 不进行赋值直接使用会引起panic错误
        g = greeting
        fmt.Println(g(3))
    }
    greeting2(g)
    c := createStruct
    fmt.Printf("%T\n", c)
    fmt.Println(c)
    var p1 *people = c()
    p1.name = "Tim"
    (*p1).age = 16
    fmt.Println(p1)
    fmt.Println(*c())
} // main
```

运行结果

```
$ go run func.go
[ruby-2.6.3p62]
Hi, nice to meet you!
-------------------------------------
Hi!
Good morning
Hello world!
Hi, nice to meet you!
Hi!
Hi!
-------------------------------------
func() *main.people
0x109ec60
&{Tim 16 Student Suzhou}
{Darren 20 Student Suzhou}
```

参考资料

Go语言圣经（中文版）
https://books.studygolang.com/gopl-zh/index.html
《Go入门指南》
https://github.com/unknwon/the-way-to-go_ZH_CN
菜鸟教程——Go 语言教程
https://www.runoob.com/go/go-tutorial.html
菜鸟教程——JSON 教程
https://www.runoob.com/json/json-tutorial.html

Golang 学习笔记06

2020/07/13 & 2020/07/15

cyc/Golang 学习笔记/Golang 学习笔记06

函数 Function —— 匿名、闭包、变参、defer和panic处理

闭包和匿名函数

- 有函数名的函数只能在包级语法块中被声明
- 当不希望给函数起名字的时候，可以使用**匿名函数**(anonymous function, also known under the names of a lambda function, a function literal函数字面量, or a closure闭包)
- 匿名函数不能独立存在，但能**被赋予某个变量**。

简单的匿名函数

closure.go

```
package main

func main() {
    for i := 0; i < 3; i++ {
        func() {
            println(i)
        }()
    } // for
} // main
```

- 参数列表的第一对括号必须紧挨着关键字 `func`，因为匿名函数没有名称
- 花括号`{}`涵盖着函数体
- 最后的一对括号表示对该匿名函数的调用
  - 这里不加最后的一对括号`()`会报错
  - ./closure.go:5:3: func literal evaluated but not used
  - 因为定义了这个匿名函数，但函数没有被调用

运行结果

```
$ go run closure.go
[ruby-2.6.3p62]
0
1
2
```

closure2.go

```
package main

func main() {
    var g int
    for i := 0; i < 3; i++ {
        func(s string) {
            g = i
            println(g)
            println(s)
        }("Hello World!")
    } // for
} // main
```

- 将"Hello World!"作为匿名函数的参数输入

运行结果

```
$ go run closure2.go
[ruby-2.6.3p62]
0
Hello World!
1
Hello World!
2
Hello World!
```

counter.go

```
package main

```

```go
import "fmt"

func getClosure() func() {
    count := 0
    fmt.Printf("The count in getClosure() is %d\n", count)
    return func() {
        count++
        fmt.Println(count)
    }
} // getClosure

func main() {
    counter := getClosure()
    counter()
    counter()
    counter()
    counter()
    counter()
    counter()
    counter()
    counter()
    counter()
    counter2 := getClosure()
    counter2()
    counter()
} // main
```

- 这里将getClosure()返回的匿名函数赋予了counter
- 在函数中定义的内部函数可以引用该函数的变量
- 在getClosure中定义的匿名内部函数**可以访问和更新getClosure中的局部变量**，这意味着匿名函数和getClosure中，存在**变量引用**

运行结果

```
$ go run counter.go
[ruby-2.6.3p62]
The count in getClosure() is 0
1
2
3
4
```

```
5
6
7
8
The count in getClosure() is 0
1
9
```

变参函数

函数的**最后一个参数**是采用 `...type` 的形式，那么这个函数就可以处理一个变长的参数，这个长度可以为 0，这样的函数称为变参函数。

```go
func myFunc(a, b, arg ...int) {}
```

在函数体中,arg被看作是类型为[] int的切片。myfunc可以接收任意数量的int型参数

简单的变参函数

greeting.go

```go
package main

import "fmt"

func greeting(names ...string) {
    for _, i := range names {
        fmt.Printf("Hello %s!\n", i)
    } // for
} // greeting

func main() {
    greeting("Jim", "Sean", "Barry", "Gareth", "Tobby")
} // main
```

运行结果

```
$ go run greeting.go
[ruby-2.6.3p62]
```

```
Hello Jim!
Hello Sean!
Hello Barry!
Hello Gareth!
Hello Tobby!
```

Deferred函数

- `defer`允许我们推迟到
  - 函数返回之前
  - 或任意位置执行`return`语句之后一刻

  才执行某个语句或函数，类似于Java中的`finally`
- 直到包含该`defer`的函数执行完毕时，`defer`后的函数才会被执行，不论包含`defer`的函数是通过return正常结束，还是由于panic导致的异常结束
- 当有多个`defer`行为被注册时，它们会以逆序执行(**后进先出**LIFO)

之前闭包函数的defer修改版

reference.go

```go
package main

import "fmt"

func main() {
    for i := 0; i < 3; i++ {
        // 因为defer了，所以当func执行时i的值为3
        defer func() {println(i)}()
    } // for
    // 因为其他都defer了，所以这行会先执行
    fmt.Println("---------------------------------")
    for i := 0; i < 5; i++ {
        // defer是后进先出，所以最后一个defer会先执行
        // 也就是The value of i is: 2
        defer fmt.Printf("The value of i is: %d\n", i)
    } // for
    fmt.Println("Looks like the end of main func!")
} // main
```

运行结果

```
$ go run reference.go
[ruby-2.6.3p62]
--------------------------------
Looks like the end of main func!
The value of i is: 4
The value of i is: 3
The value of i is: 2
The value of i is: 1
The value of i is: 0
3
3
3
```

Panic异常及Recover捕获异常

- Go没有像Java的`try/catch`异常机制: 不能执行抛异常操作。但是有一套 `defer-panic-and-recover`机制
- **运行时错误**或**直接调用`panic`函数**，会引起`painc`异常
- `panic`会引起程序的崩溃，因此`panic`一般用于**严重错误**
- 对于大部分漏洞，我们应该**使用Go提供的错误机制**，而不是`panic`，尽量避免程序的崩溃
- `recover`只有在`defer`**调用的函数中**有效
- 但不加区分的恢复所有的panic异常，不是可取的做法，应该分情况值恢复部分异常，如：

```
defer func() {
        switch p := recover(); p {
        case nil:       // no panic
        // 当panic value是bailout{}类型时，deferred函数生成一个error返回给调用者
        case bailout{}: // "expected" panic
            err = fmt.Errorf("multiple title elements")
        // 当panic value是其他non-nil值时，表示发生了未知的panic异常
        default:
            panic(p) // unexpected panic; carry on panicking
        }
    }()
```

recoverTest.go

```go
package main

import (
    "fmt"
    "log"
)

func protect(g func()) {
    defer func() {
        log.Println("done protect")
        // Println executes normally even if there is a panic
        if err := recover(); err != nil {
            log.Printf("run time panic: %v", err)
        }
    }()
    log.Println("start protect")
    g() // have runtime-error
    fmt.Println("I am doing something in protect function!")
} // prorect

func canNotProtect(g func()) {
    defer recover()
    log.Println("start can not protect function")
    g() // have runtime-error
    fmt.Println("I am doing something in protect function!")
} // prorect

func raisePanic() {
    panic("I am very panic!!! :-(")
    fmt.Println("I am doing something in raisePanic function!")
} // raisePanic

func main() {
    panicFunc := raisePanic
    protect(panicFunc)
    fmt.Println("I am doing something in main function!")
    canNotProtect(panicFunc)
} // main
```

运行结果

```
$ go run recoverTest.go
[ruby-2.6.3p62]
2020/07/15 20:25:58 start protect
2020/07/15 20:25:58 done protect
2020/07/15 20:25:58 run time panic: I am very panic!!! :-(
I am doing something in main function!
2020/07/15 20:25:58 start can not protect function
panic: I am very panic!!! :-(

goroutine 1 [running]:
main.raisePanic()
        /Users/cyc/Projects/exe/recoverTest.go:29 +0x39
main.canNotProtect(0x10d7e58)
        /Users/cyc/Projects/exe/recoverTest.go:24 +0x95
main.main()
        /Users/cyc/Projects/exe/recoverTest.go:37 +0x99
exit status 2
```

参考资料

Go语言圣经（中文版）
https://books.studygolang.com/gopl-zh/index.html
《Go入门指南》
https://github.com/unknwon/the-way-to-go_ZH_CN
ruilisi/golang-examples
https://github.com/ruilisi/golang-examples

Golang 学习笔记07

2020/07/18

cyc/Golang 学习笔记/Golang 学习笔记07

方法 Method

方法声明

- Go方法是作用在接收者(receiver)上的一个函数，接收者是某种类型的变量。因此方法是一种特殊类型的函数。接收者类型几乎可以是任何类型，但是不能是一个接口类型。接收者也不能是一个指针类型，但是它可以是任何其他允许类型的指针。
  - 接收者的值可以为`nil`
- 定义方法的一般格式如下

```
func (recv receiver_type) methodName(parameter_list) (return_value_list) {
  // body
}
```

  - 如果`recv`是receiver的实例，Method1是它的方法名，那么方法调用遵循传统的`object.name`选择器符号：**recv.Method1()**。
  - 如果`recv`是一个指针，Go 会自动解引用。
- 如果方法不需要使用`recv`的值，可以用`_`替换它，比如

```
func (_ receiver_type) methodName(parameter_list) (return_value_list) {
  // body
}
```

- 函数和方法的区别
  - 函数将变量作为参数：**Function(recv)**
  - 方法在变量上被调用：**recv.Method()**

演示

以Go语言圣经的例子为例

geometry/geometry.go

```go
package geometry

import "math"

type Point struct{ X, Y float64 }
// A Path is a journey connecting the points with straight lines.
type Path []Point

// traditional function
func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

// same thing, but as a method of the Point type
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

// Distance returns the distance traveled along the path.
func (path Path) Distance() float64 {
    sum := 0.0
    for i := range path {
        if i > 0 {
            sum += path[i-1].Distance(path[i])
        }
    }
    return sum
}
```

geometryTest.go

```go
package main

import (
    "fmt"
```

```
5
6       "./geometry"
7   )
8
9   func main() {
10      p := geometry.Point{1, 2}
11      q := geometry.Point{4, 6}
12      fmt.Println(geometry.Distance(p, q)) // "5", function call
13      fmt.Println(p.Distance(q))           // "5", method call
14      fmt.Println("----------------------------------------------")
15      perim := geometry.Path{{1, 1}, {5, 1}, {5, 4}, {1, 1}}
16      // 下行代码不能通过编译报错信息为
17      // not enough arguments in call to geometry.Distance
18      //   have (geometry.Path)
19      //   want (geometry.Point, geometry.Point)
20      // fmt.Println(geometry.Distance(perim)) // "12", standalone
     function
21      // 如果要让上面代码可以执行，可将geometry包中的Distance名字改为PathDis
    tance，再调用
22      // fmt.Println(geometry.PathDistance(perim))
23      fmt.Println(perim.Distance()) // "12", method of geometry.Pat
    h
24  }
```

运行结果

```
1 $ go run geometryTest.go
  [ruby-2.6.3p62]
2 5
3 5
4 ----------------------------------------------
5 12
```

基于指针对象的方法

- 当接收者值比较大的时候可以使用指针

scale.go

```go
package main

import "fmt"

type Point struct{ X, Y float64 }

// 这个方法的名字是(*Point).ScaleBy 这里的括号是必须的
func (p *Point) ScaleBy(factor float64) {
  p.X *= factor
  p.Y *= factor
} // ScaleBy

func main() {
  // 调用方法1
  // addP1是一个指向Point{1, 2}的指针，存着Point{1, 2}的地址
  addP1 := &Point{1, 2}
  addP1.ScaleBy(2)
  fmt.Println(*addP1) // "{2, 4}"
  // 调用方法2
  // p2是一个Point
  p2 := Point{3, 4}
  (&p2).ScaleBy(2)
  fmt.Println(p2) // "{6, 8}"
  // 调用方法3
  p3 := Point{5, 6}
  // 编译器会隐式调用 &p3
  p3.ScaleBy(10)
  fmt.Println(p3) // "{50, 60}"
} // main
```

运行结果

```
$ go run scale.go
[ruby-2.6.3p62]
{2 4}
{6 8}
{50 60}
```

通过嵌入结构体来扩展类型

聚合

- 包含一个所需功能类型的具名字段。

embed1.go

```go
package main

import (
  "fmt"
)

type Log struct {
  msg string
}

type Customer struct {
  Name string
  log  *Log
}

func main() {
  c := new(Customer)
  // c.Name = "Sean"
  // c.log = new(Log)
  // c.log.msg = "1 - Hello everyone!"
  // The above three lines have same effect as the line below
  c = &Customer{"Sean", &Log{"1 - Hello everyone!"}}
  fmt.Println(c)
  fmt.Println("---------------------------------")
  c.log.Add("2 - Welcome to computation class!")
  fmt.Println(c)
  fmt.Println("---------------------------------")
  d := Customer{"Barry", &Log{"1 - Hi!"}}
  d.log.Add("2 - Welcome to Mobile Systems class!")
  fmt.Println(d)
  fmt.Printf("The type of d is %T\n", d)
} // main
```

```
33
34 func (l *Log) Add(s string) {
35     l.msg += "\n" + s
36 } // Add
37
38 func (l *Log) String() string {
39     return l.msg
40 } // String
41
42 func (c *Customer) String() string {
43     return c.Name + "\nLog:\n" + (c.log).String()
44 } // String
```

运行结果

```
1 $ go run embed1.go
  [ruby-2.6.3p62]
2 Sean
3 Log:
4 1 - Hello everyone!
5 ----------------------------------
6 Sean
7 Log:
8 1 - Hello everyone!
9 2 - Welcome to computation class!
10 ----------------------------------
11 {Barry 0xc0001101f0}
12 The type of d is main.Customer
```

内嵌

- 内嵌(匿名地)所需功能类型

embed2.go

```
1 package main
2
3 import (
4   "fmt"
```

```
)

type Log struct {
  msg string
}

type Customer struct {
  Name string
  Log
}

func main() {
  c := &Customer{"Sean", Log{"1 - Hello everyone!"}}
  fmt.Println(c)
  fmt.Println("---------------------------------")
  c.Add("2 - Welcome to Computation class!")
  fmt.Println(c)
  fmt.Println("---------------------------------")
  d := Customer{"Barry", Log{"1 - Hi!"}}
  d.Add("2 - Welcome to Mobile Systems class!")
  fmt.Println(d)
  fmt.Printf("The type of d is %T\n", d)
} // main

func (l *Log) Add(s string) {
  l.msg += "\n" + s
} // Add

func (l *Log) String() string {
  return l.msg
} // String

func (c *Customer) String() string {
  return c.Name + "\nLog:\n" + (&(c.Log)).String()
} // String
```

运行结果

```
$ go run embed2.go
```

```
  [ruby-2.6.3p62]
2 Sean
3 Log:
4 1 - Hello everyone!
5 ---------------------------------
6 Sean
7 Log:
8 1 - Hello everyone!
9 2 - Welcome to Computation class!
10 ---------------------------------
11 {Barry {1 - Hi!
12 2 - Welcome to Mobile Systems class!}}
13 The type of d is main.Customer
```

封装

- 一个对象的变量或者方法如果对调用方是不可见的话，一般就被定义为封装(Encapsulation)。
- Go语言只有一种控制可见性的手段：
  - **大写首字母**的标识符会从定义它们的包中被**导出**
  - **小写首字母**的则**不会被导出**
  - 这种限制包内成员的方式同样适用于struct或者一个类型的方法
  - 如果我们想要封装一个对象，我们**必须将其定义为一个struct**

封装的优点

1. 因为调用方不能直接修改对象的变量值，其只需要关注少量的语句并且只要弄懂少量变量的可能的值即可。
2. 隐藏实现的细节，可以防止调用方依赖那些可能变化的具体实现
3. 阻止了外部调用方对对象内部的值任意地进行修改

参考资料

Go语言圣经（中文版）

https://books.studygolang.com/gopl-zh/index.html

《Go入门指南》

https://github.com/unknwon/the-way-to-go_ZH_CN

Golang import 包时可以使用相对路径吗

https://blog.csdn.net/k346k346/article/details/89390729

Golang 学习笔记08

2020/07/20

cyc/Golang 学习笔记/Golang 学习笔记08

接口 Interface

- Go语言里面**没有类和继承**的概念。
- Go语言里有**接口**，通过它可以实现面向对象的特性。
- 接口提供了一种方式来说明对象的行为。
- 接口类型是对其它类型行为的**抽象和概括**，它不会和特定的实现细节绑定在一起。

接口的定义

- 接口定义了一组方法，但是这些方法不包含实现代码(它们是抽象的)。
- 接口里**不能包含变量**。
- 接口的定义方法

```
type Name interface {
    Method1(param_list) return_type
    Method2(param_list) return_type
}
```

实现接口的条件

- 一个类型如果拥有**一个接口需要的所有方法**，那么这个类型就实现了这个接口。

```
var w io.Writer
w = os.Stdout           // OK: *os.File has Write method
w = new(bytes.Buffer)   // OK: *bytes.Buffer has Write method
w = time.Second         // compile error: time.Duration lacks Writ
e method

var rwc io.ReadWriteCloser
rwc = os.Stdout         // OK: *os.File has Read, Write, Close met
hods
rwc = new(bytes.Buffer) // compile error: *bytes.Buffer lacks Clos
```

```
e method
```

interfaceTest.go

```go
package main

import "fmt"

type Shaper interface {
    Area() float32
}

type Square struct {
    side float32
}

// *Square has Area() function so can be a Shaper
func (sq *Square) Area() float32 {
    return sq.side * sq.side
} // Area

type Rectangle struct {
    length, width float32
}

// Rectangle has Area() function so can be a Shaper
func (r Rectangle) Area() float32 {
    return r.length * r.width
} // Area

func main() {
    sq1 := new(Square)
    sq1.side = 3.5

    var areaIntf Shaper
    areaIntf = sq1
    // shorter,without separate declaration:
    // areaIntf := Shaper(sq1)
    // or even:
```

```
    // areaIntf := sq1
    fmt.Println("The square1 has area: ", areaIntf.Area())

    fmt.Println("--------------------------------------")

    r := Rectangle{5, 3} // Area() of Rectangle needs a value
    sq2 := &Square{5}    // Area() of Square needs a pointer
    // The above two lines the same effect as the following 6 lin
es
    // rAddr := new(Rectangle)
    // rAddr.length = 5
    // rAddr.width = 3
    // r := *rAddr
    // sq2 := new(Square)
    // sq2.side = 5

    // shapes := []Shaper{Shaper(r), Shaper(sq2)}
    // or shorter
    shapes := []Shaper{r, sq2}
    fmt.Println("Looping through shapes for area ...")
    for n, _ := range shapes {
        fmt.Println("Shape details: ", shapes[n])
        fmt.Println("Area of this shape is: ", shapes[n].Area())
    } // for
} // main
```

运行结果

```
$ go run interfaceTest.go
[ruby-2.6.3p62]
The square1 has area:  12.25
--------------------------------------
Looping through shapes for area ...
Shape details:  {5 3}
Area of this shape is:  15
Shape details:  &{5}
Area of this shape is:  25
```

空接口类型

interface{}被称为空接口类型。因为空接口类型对实现它的类型没有要求，所以我们可以将任意一个值赋给空接口类型。

```
var any interface{}
any = true
any = 12.34
any = "hello"
any = map[string]int{"one": 1}
any = new(bytes.Buffer)
```

Go语言和Java语言接口的区别

- Go语言中不需要特殊声明implement(实现类不需要明确声明自己实现了某个接口)
- 在Go语言中，一个类只需要实现了接口要求的所有函数，就说这个类实现了该接口。
- 例：

```
type File struct {
  // ...
}

func (f *File) Read(buf []byte) (n int, err error)
func (f *File) Write(buf []byte) (n int, err error)
func (f *File) Seek(off int64, whence int) (pos int64, err error)
func (f *File) Close() error
```

上文定义了一个File类，并实现有Read()、Write()、Seek()、Close()等方法。假设有如下接口

```
type IFile interface {
  Read(buf []byte) (n int, err error)
  Write(buf []byte) (n int, err error)
  Seek(off int64, whence int) (pos int64, err error)
  Close() error
}

type IReader interface {
  Read(buf []byte) (n int, err error)
```

```
}

type IWriter interface {
  Write(buf []byte) (n int, err error)
}

type ICloser interface {
  Close() error
}
```

尽管File类并没有从这些接口继承，甚至可以不知道这些接口的存在，但是File类实现了这些接口，可以进行赋值

```
var file1 IFile = new(File)
var file2 IReader = new(File)
var file3 IWriter = new(File)
var file4 ICloser = new(File)
```

接口值

- 接口值，由两个部分组成，一个具体的**类型(type)**和那个类型的**值(value)**。它们被称为接口的动态类型和动态值。
- 下面4个语句中，变量w得到了3个不同的值。（开始和最后的值是相同的）

```
var w io.Writer
w = os.Stdout
w = new(bytes.Buffer)
w = nil
```

    - 第一个语句定义了变量w

```
var w io.Writer
```

        - 对于一个接口的零值就是它的类型和值的部分都是nil

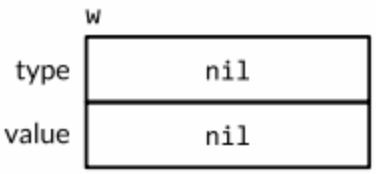


Figure 7.1. A nil interface value.

- 第二个语句将一个`*os.File`类型的值赋给变量w

```
w = os.Stdout
```

- 赋值过程调用了一个具体类型到接口类型的隐式转换
- 和显式的使用io.Writer(os.Stdout)是等价的
- 这个接口值的动态类型被设为`*os.Stdout`指针的类型描述符，它的动态值持有os.Stdout的拷贝
- 这是一个代表处理标准输出的os.File类型变量的指针

Figure 7.2. An interface value containing an `*os.File` pointer.

```
// 调用一个包含*os.File类型指针的接口值的Write方法，使得(*os.File).Write方法被调用
w.Write([]byte("hello")) // "hello"
// 编译器必须把代码生成在类型描述符的方法Write上，然后间接调用那个地址
// 调用的接收者是一个接口动态值的拷贝，os.Stdout
// 效果和下面这个直接调用一样
os.Stdout.Write([]byte("hello")) // "hello"
```

- 第三个语句给接口值赋了一个*bytes.Buffer类型的值

```
w = new(bytes.Buffer)
```

- 现在动态类型是*bytes.Buffer并且动态值是一个指向新分配的缓冲区的指针

Figure 7.3. An interface value containing a `*bytes.Buffer` pointer.

- 第四个语句将nil赋给了接口值

```
w = nil
```

  - 这个重置将它所有的部分都设为nil值，把变量w恢复到和它之前定义时相同的状态图。

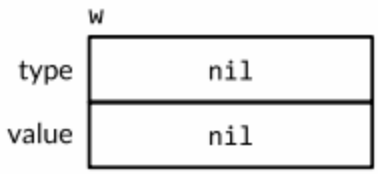


Figure 7.1. A nil interface value.

- 接口值可以使用`==`和`!=`来进行比较。两个接口值相等仅当它们都是nil值或者它们的动态类型相同并且动态值也根据这个动态类型的`==`操作相等。
- 然而，如果两个接口值的动态类型相同，但是这个动态类型是不可比较的(比如切片)，将它们进行比较就会失败并且panic

接口嵌套接口

- 一个接口可以包含一个或多个其他的接口，这相当于直接将这些内嵌接口的方法列举在外层接口中一样。
- 比如接口`File`包含了`ReadWrite`和`Lock`的所有方法，它还额外有一个`Close()`方法。

```
type ReadWrite interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
}

type Lock interface {
    Lock()
    Unlock()
}

```

```
type File interface {
    ReadWrite
    Lock
    Close()
}
```

error接口

- error是一个interface

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

类型断言与类型分支

类型断言

- 类型断言是一个使用在接口值上的操作。语法上它看起来像x.(T)被称为断言类型，这里x表示一个**接口的类型**和T表示一个**类型**。一个类型断言检查它操作对象的动态类型是否和断言的类型匹配。

```
// 如果断言类型T是一个具体类型，那么类型断言会检查x的动态类型是否就是T
// 检查成功，类型断言的结果就是x的动态值，类型是T
// 检查失败，程序崩溃
// w是一个io.Writer接口的变量(Writer接口要实现Write(p []byte) (n int, err error)方法)
var w io.Writer
w = os.Stdout
// 检查w接口变量是否包含*os.File类型
// 注：w必须是一个接口变量，否则编译器会报错
f := w.(*os.File)      // success: f == os.Stdout
c := w.(*bytes.Buffer) // panic: interface holds *os.File, not *bytes.Buffer
// ------------------------------------------------------------------
// 如果断言类型T是一个接口类型，那么类型断言检查x的动态类型是否满足T
var w io.Writer
w = os.Stdout
```

```
15 rw := w.(io.ReadWriter) // success: *os.File has both Read and Wr
   ite
16 w = new(ByteCounter)
17 rw = w.(io.ReadWriter) // panic: *ByteCounter has no Read method
```

- 更安全的方式是使用以下形式来进行类型断言

```
1 // 假设varI是一个接口变量
2 if v, ok := varI.(T); ok {  // checked type assertion
3     Process(v)
4     return
5 }
6 // varI is not of type T
```

  - 如果转换合法，`v`是`varI`转换到类型`T`的值，`ok`会是`true`。
  - 否则`v`是类型`T`的零值，`ok`是`false`，也没有运行时错误(panic)发生。

类型分支

- 在断言时可以使用类型分支来判断接口的动态类型或者接口的类型

```
1 switch x.(type) {
2     case nil:       // ...
3     case int, uint: // ...
4     case bool:      // ...
5     case string:    // ...
6     default:        // ...
7 }
```

  - 这里直接写关键字`type`而不是一个特定的类型

运用案例

interfaceType.

```
1 package main
2
3 import (
4     "fmt"
5     "math"
```

```
)

type Square struct {
    side float32
}

type Circle struct {
    radius float32
}

type Shaper interface {
    Area() float32
}

type MoreShaper interface {
    Shaper
    Perimeter() float32
}

type Animal interface {
    Bark()
}

type Dog struct{
    weight float32
}

func main() {
    var areaIntf Shaper = &Square{5}
    var aSquare = Square{4}
    var areaIntf2 Shaper = &Circle{2}
    var anAnimal Animal = Dog{21}
    fmt.Println("--------断言类型是一个具体类型--------")
    // Is Square the type of areaIntf ?
    if t, ok := areaIntf.(*Square); ok {
        fmt.Printf("The type of areaIntf is: %T\n", t)
        fmt.Printf("t is: %v\n", t)
    } // if
    if u, ok := areaIntf.(*Circle); ok {
        fmt.Printf("The type of areaIntf is: %T\n", u)
```

```
46      } else {
47          fmt.Println("areaIntf does not contain a variable of typ
    e Circle")
48          fmt.Printf("u is: %v\n", u)
49      } // if ... else
50      // Is Square the type of &aSquare ?
51      if t, ok := (Shaper)(&aSquare).(*Square); ok {
52          fmt.Printf("The type of &aSquare is: %T\n", t)
53          fmt.Printf("t is: %v\n", t)
54      } // if
55      fmt.Println("--------断言类型是一个接口--------")
56      // is areaIntf a Shaper ?
57      if z, ok := areaIntf.(Shaper); ok {
58          fmt.Println("The of areaIntf is a Shaper")
59          fmt.Printf("z is: %v\n", z)
60      } // if
61      // is areaIntf2 a MoreShaper ?
62      if u, ok := areaIntf2.(MoreShaper); ok {
63          fmt.Printf("The type of areaIntf2 is: %T\n", u)
64      } else {
65          fmt.Println("areaIntf2 is not an interface MoreShaper")
66          fmt.Printf("u is: %v\n", u)
67      } // if ... else
68      // is &aSquare a MoreShaper ?
69      if z, ok := (Shaper)(&aSquare).(MoreShaper); ok {
70          fmt.Println("The &aSquare is a MoreShaper")
71          fmt.Printf("z is: %v\n", z)
72      } // if
73      fmt.Println("--------switch--------")
74      // testing with switch:
75      switchType(areaIntf)
76      switchType(anAnimal)
77      switchInterface((Shaper)(&aSquare))
78      switchInterface(anAnimal)
79  } // main
80
81  func switchType(x interface{}) {
82      switch t := x.(type) {
83      case *Square:
84          fmt.Printf("Type Square %T with value %v\n", t, t)
```

```
85      case *Circle:
86          fmt.Printf("Type Circle %T with value %v\n", t, t)
87      case Dog:
88          fmt.Printf("Type Dog %T with value %v\n", t, t)
89      default:
90          fmt.Printf("Unexpected type %T", t)
91      } // switch
92  } // switchType
93
94  func switchInterface(x interface{}) {
95      switch u := x.(type) {
96      case MoreShaper:
97          fmt.Printf("A MoreShaper type with value %v\n", u)
98      case Shaper:
99          fmt.Printf("A shaper interface with value %v\n", u)
100     default:
101         fmt.Printf("Unexpected interface with value %v\n", u)
102     } // switch
103 } // switchInterface
104
105 func (sq *Square) Area() float32 {
106     return sq.side * sq.side
107 } // Area
108
109 func (d Dog) Bark() {
110     switch {
111   case d.weight < 10:
112         fmt.Println("woof-woof")
113     case 10 <= d.weight && d.weight < 20:
114         fmt.Println("ruff-ruff")
115     default:
116         fmt.Println("bow-wow")
117     } // switch
118 } // Area
119
120 func (sq *Square) Perimeter() float32 {
121     return sq.side * 4
122 } // Perimeter
123
124 func (ci *Circle) Area() float32 {
```

```
125     return ci.radius * ci.radius * math.Pi
126 } // Area
```

运行结果

```
1 $ go run interfaceType.go
  [ruby-2.6.3p62]
2 --------断言类型是一个具体类型--------
3 The type of areaIntf is: *main.Square
4 t is: &{5}
5 areaIntf does not contain a variable of type Circle
6 u is: <nil>
7 The type of &aSquare is: *main.Square
8 t is: &{4}
9 --------断言类型是一个接口--------
10 The of areaIntf is a Shaper
11 z is: &{5}
12 areaIntf2 is not an interface MoreShaper
13 u is: <nil>
14 The &aSquare is a MoreShaper
15 z is: &{4}
16 --------switch--------
17 Type Square *main.Square with value &{5}
18 Type Dog main.Dog with value {21}
19 A MoreShaper type with value &{4}
20 Unexpected interface with value {21}
```

参考资料

《Go语言编程》

第3章 面向对象编程 3.5 接口

Go语言圣经（中文版）

https://books.studygolang.com/gopl-zh/index.html

《Go入门指南》

https://github.com/unknwon/the-way-to-go_ZH_CN

Golang 学习笔记09

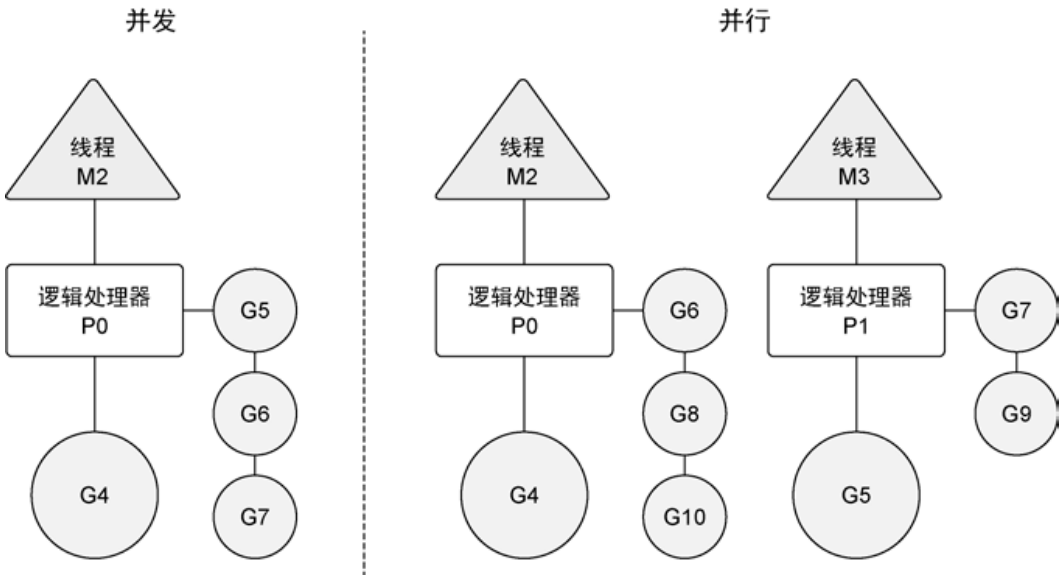
2020/07/22

cyc/Golang 学习笔记/Golang 学习笔记09

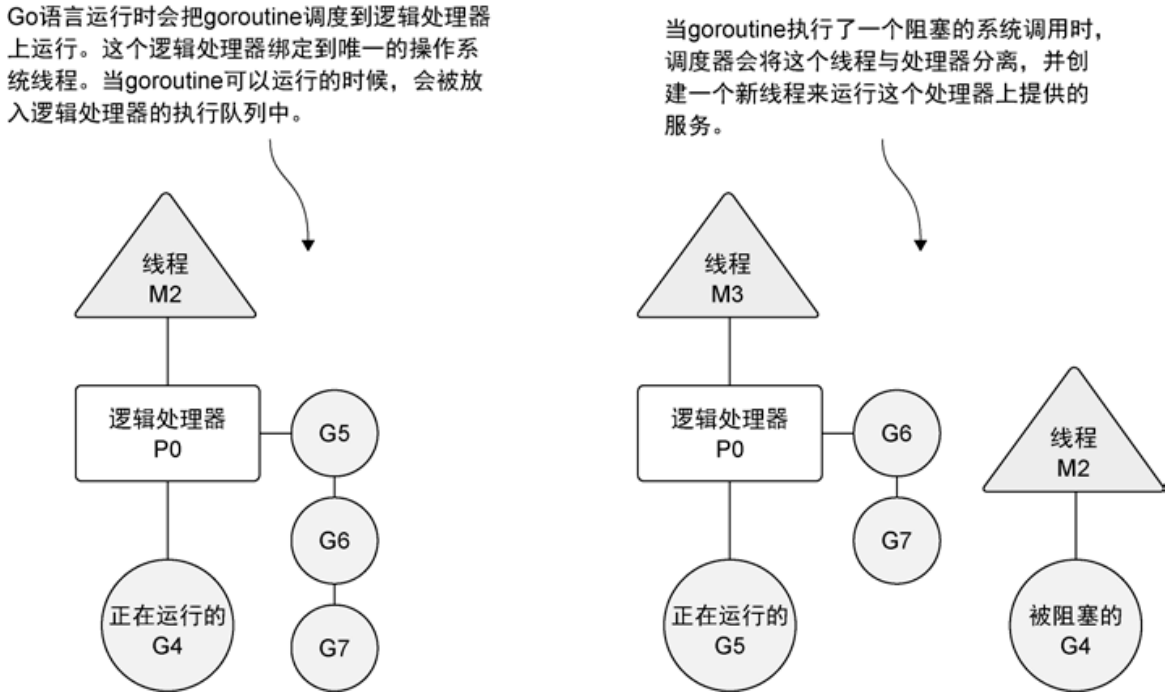
Gorotine和通道(Channel)

Goroutine

- 每一个并发的执行单元叫作一个goroutine，通过通道来通信
  - 并发和并行的区别：并行是让不同的代码片段同时在不同的物理处理器上执行。并行的关键是同时做很多事情，而并发是指同时管理很多事情



  - 默认情况下，Go会给每个可用的物理处理器都分配一个逻辑处理器，所以一般来说goroutine是并发+并行的，除非加上：*runtime.GOMAXPROCS(1) //只创建一个逻辑管理器*



- 在语法上，go语句是一个普通的函数或方法调用前加上关键字go。

```
1 f()    // call f(); wait for it to return
2 go f() // create a new goroutine that calls f(); don't wait
```

- 主函数返回时，所有的goroutine都会被直接打断，程序退出。主函数退出的时候，即使goroutine还没执行完也会被退出，goroutine执行的代码可能不会有结果。

Channels

- channel是Go语言在语言级别提供的goroutine间的通信方式。我们可以使用channel在两个或多个goroutine之间传递消息

声明和定义

- 可以使用make创建通道

```
1 // 一般channel的声明形式为:
2 var chanName chan ElementType
3 // 声明一个传递类型为int的channel
4 var ch chan int
5 // 我们声明一个map，元素是bool型的channel
6 var m map[string] chan bool
7
8 // 定义channel
```

```
// 无缓冲的整型通道
unbuffered := make(chan int)   // unbuffered has type 'chan int'
// 有缓冲的字符串通道
buffered := make(chan string, 10)
```

- make 的第一个参数需要是关键字chan，之后跟着允许通道交换的数据的类型。
- 如果创建的是一个有缓冲的通道，之后还需要在第二个参数指定这个通道的缓冲区的大小。

写入、读出和关闭

```
// 将一个数据写入（发送）至channel
ch <- x  // a send statement
// 从channel中读取数据
x = <-ch // a receive expression in an assignment statement
<-ch     // a receive statement; result is discarded
```

- 如果channel之前没有写入数据，那么从channel中读取数据也会导致程序阻塞，直到channel中被写入数据为止
- Channel还支持close操作，用于关闭channel，随后对基于该channel的任何**发送操作**都将导致panic异常。
- 对一个已经被close过的channel进行接收操作**依然可以接受到之前已经成功发送的数据**
- 如果channel中已经没有数据的话将产生一个零值的数据。

```
close(ch)
```

无缓冲通道/无缓存通道

- 一个基于无缓存Channels的发送操作将导致**发送者goroutine阻塞，直到另一个goroutine在相同的Channels上执行接收操作**，当发送的值通过Channels成功传输之后，两个goroutine可以继续执行后面的语句。
- 反之，如果接收操作先发生，那么**接收者goroutine也将阻塞，直到有另一个goroutine在相同的Channels上执行发送操作**。
- 无缓存Channels有时候也被称为同步Channels，因为接收者的接受操作在发送者被唤醒前发生。(必须要接收者接受了消息，发送者才能继续执行)
- 无缓存通道可能导致程序阻塞

blocking.go

```
package main
```

```go

import (
    "fmt"
)

func f1(in chan int) {
    fmt.Println(<-in)
}

func main() {
    out := make(chan int)
    out <- 2
    // 此时发送方main函数开始休眠了
    // 以下部分无法执行，形成死锁
    go f1(out)
}
```

运行结果：所有的协程都休眠了 - 死锁!

```
$ go run blocking.go
[ruby-2.6.5]
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        /home/vagrant/Projects/exe/blocking.go:13 +0x55
exit status 2
```

修改

```go
package main

import (
    "fmt"
)

func f1(in chan int) {
    fmt.Println(<-in)
```

```
}

func main() {
    out := make(chan int)
    go f1(out)
    out <- 2
}
```

运行结果

```
$ go run blocking.go
[ruby-2.6.5]
2
```

串联的Channels（管道 Pipeline）

- Channels可以用于将多个goroutine连接在一起，**一个Channel的输出作为下一个Channel的输入**。这种串联的Channels就是所谓的管道(pipeline)。
- 《Go语言圣经》中给出如下例子

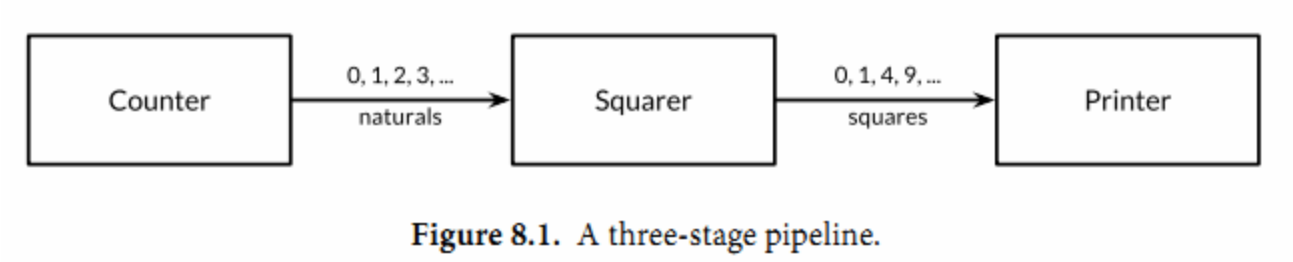


Figure 8.1. A three-stage pipeline.

pipeline1.go（对《Go语言圣经》给出代码有部分改动）

```
package main

import "fmt"

func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // Counter
    go func() {
```

```
        for x := 0; x < 20; x++ {
            // 将x的值传入通道naturals
            naturals <- x
        }
    }()

    // Squarer
    go func() {
        for {
            // 从naturals通道中拿出一个int，并将其赋予x
            x := <-naturals
            squares <- x * x
        }
    }()

    // Printer (in main goroutine)
    for x := 0; x < 10; x++ {
        // 从squares通道中拿出一个int打印出来
        fmt.Println(<-squares)
    } // for
} // main
```

运行结果

```
$ go run pipeline1.go
[ruby-2.6.3p62]
0
1
4
9
16
25
36
49
64
81
```

- 因为main函数提前退出，所以Counter中x大于等于10的值没有被执行

- 可以通过关闭channel来告诉接收者goroutine，所有的数据已经全部发送
- 可以改进以上代码

pipeline2.go

```go
package main

import "fmt"

func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // Counter
    go func() {
        for x := 0; x < 20; x++ {
            // 将x的值传入通道naturals
            naturals <- x
        }
        close(naturals)
    }()

    // Squarer
    go func() {
        // 当naturals还有的时候
        // 接收naturals的值到x
        for x := range naturals {
            squares <- x * x
        }
        close(squares)
    }()

    // Printer (in main goroutine)
    // 当squares还有的时候
    // 接收squares的值到x
    for x := range squares {
        fmt.Println(x)
    } // for
} // main
```

运行结果

```
$ go run pipeline2.go
[ruby-2.6.3p62]
0
1
4
9
16
25
36
49
64
81
100
121
144
169
196
225
256
289
324
361
```

ruilisi/golang-example中的例子
passBall.go

```go
package main

import (
    "fmt"
)

func main() {
    // 首先print "Hello"
    fmt.Println("Hello")

```

```
    // 声明myBall是个int值为1314
    myBall := 1314
    // 建4个通道
    ch := make(chan int)
    ch1 := make(chan int)
    ch2 := make(chan int)
    ch3 := make(chan int)

    go func() {
        // 刚开始运行时ch为空这个func会阻塞
        // 直到main函数将myball放入ch此func才会继续运行
        // 将main函数放入ch的值取出，赋予ball
        // 此函数是3个go func()中第一个恢复运行的
        ball := <-ch
        // 打印“world”
        fmt.Println("world")
        // 将ball放入ch1中
        ch1 <- ball
    }()

    go func() {
        // 同理，刚开始运行时ch2为空这个func会阻塞
        // 直到第43行的函数将ball放入ch2此func才会继续运行
        // 将放入ch2的值取出，赋予ball
        // 此函数是3个go func()中最后一个恢复运行的
        ball := <-ch2
        // 打印“you”
        fmt.Println("you")
        // 将ball放入ch3中
        ch3 <- ball
    }()

    go func() {
        // 同理，刚开始运行时ch1为空这个func会阻塞
        // 直到第19行的函数将ball放入ch1此func才会继续运行
        // 将放入ch1的值取出，赋予ball
        // 此函数是3个go func()中第二个恢复运行的
        ball := <-ch1
        // 打印“I love”
        fmt.Println("I love")
```

```
        // 将ball放入ch2中
        ch2 <- ball
    }()

    // 将myBall放入ch中，这样第19行的函数就能恢复运行了
    ch <- myBall
    // 一开始ch3值为空，main函数会在这里阻塞
    // 直到第31行的函数将ball放入ch3中main函数才会恢复执行
    // 将ch3中的值取出并打印出来
    fmt.Println(<-ch3)
} // main
```

运行结果

```
$ go run passBall.go
[ruby-2.6.3p62]
Hello
world
I love
you
1314
```

单向通道类型

- 单向channel只能用于发送或者接收数据。channel本身必然是同时支持读写的，所谓的单向channel概念，其实只是对channel的一种使用限制，以避免误用。
- 类型`chan<- int`表示一个只发送int的channel，只能发送不能接收。相反，类型`<-chan int`表示一个只接收int的channel，只能接收不能发送。（箭头`<-`和关键字chan的相对位置表明了channel的方向）

pipeline3.go

```
package main

import "fmt"

// 在counter中out只能用于发送int
func counter(out chan<- int) {
    for x := 0; x < 20; x++ {
```

```
        out <- x
    } // for
    close(out)
} // counter

// 在squarer中in只能用于接收int，out只能用于发送int
func squarer(out chan<- int, in <-chan int) {
    for v := range in {
        out <- v * v
    } // for
    close(out)
} // squarer

// 在printer中in只能用于接收int
func printer(in <-chan int) {
    for v := range in {
        fmt.Println(v)
    } // for
} // printer

func main() {
    naturals := make(chan int)
    squares := make(chan int)
    go counter(naturals)
    go squarer(squares, naturals)
    printer(squares)
} // main
```

运行结果和上面代码一致

缓冲通道/有缓冲通道

- 有缓冲的通道(buffered channel)是一种在被接收前能存储一个或者多个值的通道。
- 只有在通道中没有要接收的值时，接收动作才会阻塞。
- 只有在通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。
- 无缓冲的通道保证进行发送和接收的goroutine会在同一时间进行数据交换；有缓冲的通道没有这种保证。

```
ch = make(chan string, 3)
// 此时通道缓存为空，发送方不会阻塞
```

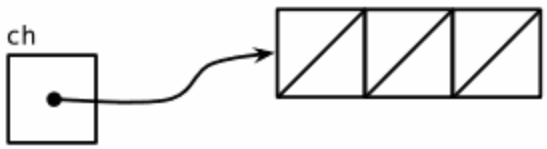


Figure 8.2. An empty buffered channel.

```
ch <- "A"
ch <- "B"
ch <- "C"
// 此时通道缓存满了，发送方会阻塞
```

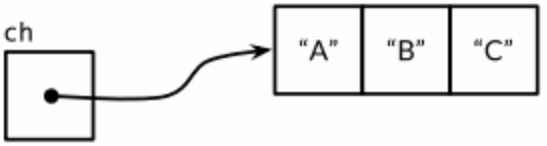


Figure 8.3. A full buffered channel.

```
fmt.Println(<-ch) // "A"
// 缓存没满，发送方不会阻塞
```

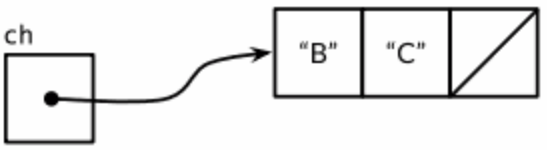


Figure 8.4. A partially full buffered channel.

```
// 获取通道缓存的容量
fmt.Println(cap(ch)) // "3"
// 获取通道缓存中有有效元素的个数
fmt.Println(len(ch)) // "2"
```

无缓存通道和缓存通道图示

- 无缓存通道

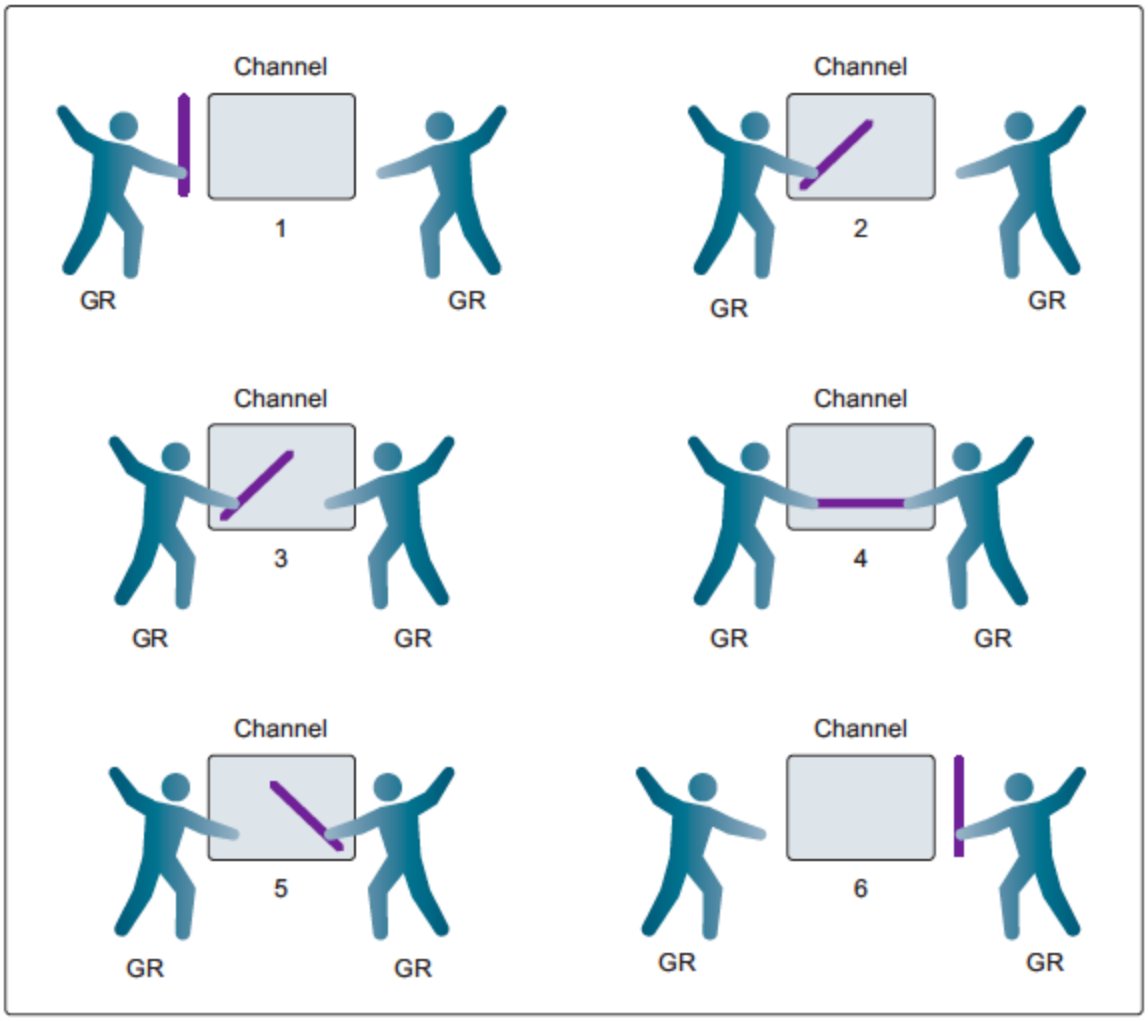


Figure 6.6 Synchronization between goroutines using an unbuffered channel

- 缓存通道

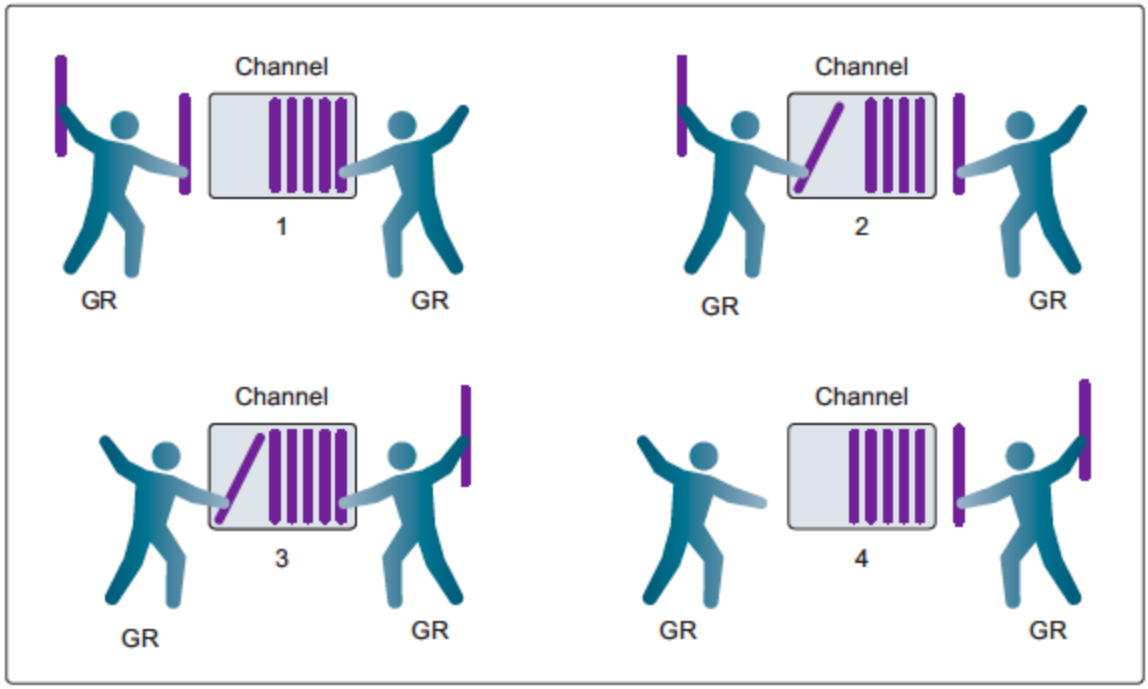


Figure 6.7 Synchronization between goroutines using a buffered channel

select语句

- 从不同的并发执行的协程中获取值可以通过关键字select来完成，它和switch控制语句非常相似也被称作通信开关

```
select {
  case u:= <- ch1:
    // ...
  case v:= <- ch2:
    // ...
  default: // no value ready to be received
    // ...
}
```

  - 如果都阻塞了，会等待直到其中一个可以处理
  - 如果多个可以处理，随机选择一个
  - 如果没有通道操作可以处理并且写了default语句，所以使用default可以防止程序阻塞
- select不像switch，后面并不带判断条件，而是直接去查看case语句
- 每个case语句都必须是一个**面向channel的操作**

```
select {
```

```
  case <-chan1:
  // 如果chan1成功读到数据，则进行该case处理语句
  case chan2 <- 1:
  // 如果成功向chan2写入数据，则进行该case处理语句
  default:
  // 如果上面都没有成功，则进入default处理流程
}
```

运用Goroutine和Channel

目标

建立一个tcp服务器，telnet该服务器，返回一个真随机算子（利用gorouine中的知识）

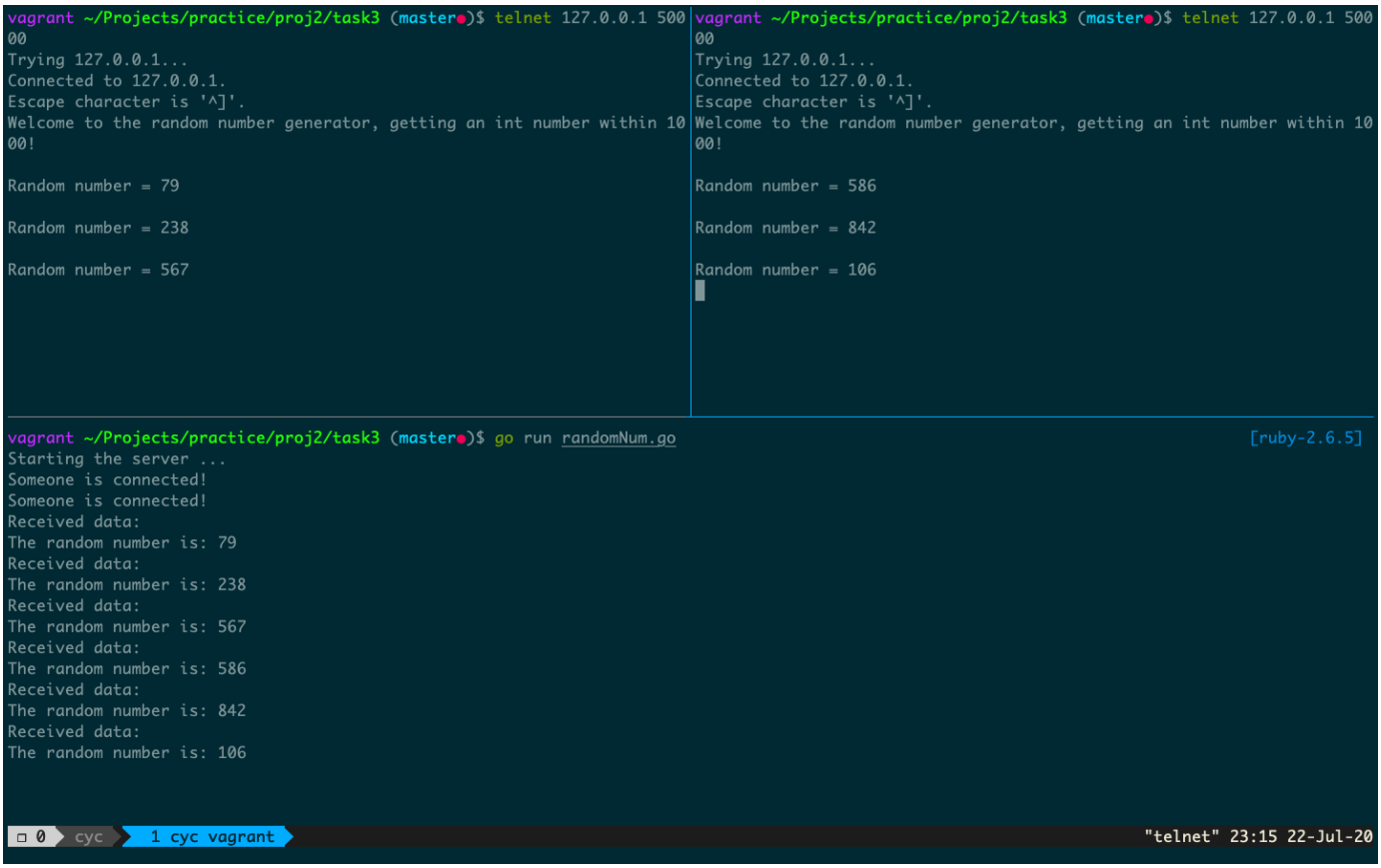
练习

randomNum.go

```
// code adapted from https://www.cnblogs.com/lfri/p/11769254.html
package main

import (
    "fmt"
    "math/rand"
    "net"
    "strings"
    "time"
)

func main() {
    fmt.Println("Starting the server ...")
    // 创建 listener
    listener, err := net.Listen("tcp", "localhost:50000")
    if err != nil {
        fmt.Println("Error listening", err.Error())
        return //终止程序
    } // if
    // 监听并接受来自客户端的连接
    for {
        conn, err := listener.Accept()
```

```
23          if err != nil {
24              fmt.Println("Error accepting", err.Error())
25              return // 终止程序
26          } else {
27              fmt.Println("Someone is connected!")
28          }
29          go doServerStuff(conn)
30      } // for
31  } // main
32
33  func randomNo() int {
34      s1 := rand.NewSource(time.Now().UnixNano())
35      // fmt.Println(time.Now().UnixNano())
36      // fmt.Printf("Time is: %v\n", time.Now().UnixNano())
37      r1 := rand.New(s1)
38      randomNumber := r1.Intn(1000)
39
40      // fmt.Printf("The random number has type: %T\n", r1.Intn(100
    0))
41      fmt.Printf("The random number is: %v\n", randomNumber)
42      return randomNumber
43  } // randomNo
44
45  func doServerStuff(conn net.Conn) {
46      fmt.Fprintf(conn, "Welcome to the random number generator, ge
    tting an int number within 1000!\n")
47      for {
48          buf := make([]byte, 512)
49          len, err := conn.Read(buf)
50          if err != nil {
51              fmt.Println("Error reading", err.Error())
52              return //终止程序
53          } // if
54          inputSting := strings.Trim(string(buf[:len]), "\r\n")
55          fmt.Printf("Received data: %v\n", inputSting)
56          fmt.Fprintf(conn, "Random number = %v\n", randomNo())
57      } // for
58  } // doServerStuff
```

运行



改进版本一

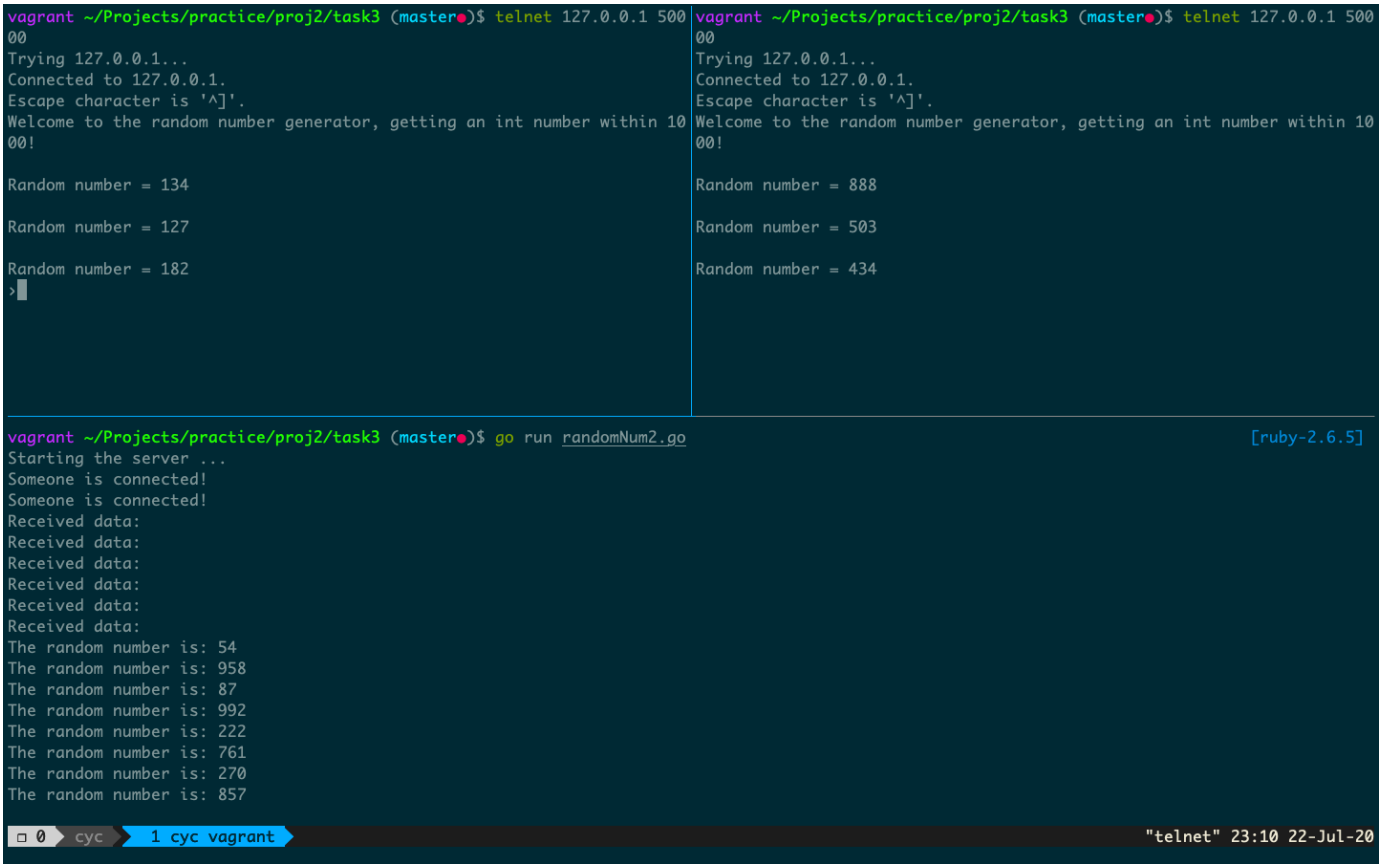
- 改进1: 上一个版本使用的math/random并不能得到一个真正的随机数，虽然使用时间作为seed能生成一个比较随机的数字，这次使用crypto/rand来生成随机数
- 改进2: 上一个版本在运用随机数时，没有使用到goroutine这一版使用了goroutine和channel，具体使用如下
  - 在doServerStuff中增加int类型的randomChannel用于接收随机数
  - 使用go randomNo(randomChannel)，并发执行randomNo来获取随机数
  - randomNo使用crypto/rand不断生成随机数并将其放入randomChannel中
  - 使用go randHandler(randomCh)来提取出生成的随机数，否则channel会被阻塞，后续随机数不会被生成
  - 用time.Sleep(1 * 1e9)暂停1秒再进行提取，这样在doServerStuff的fmt.Fprintf(conn, "Random number = %v\n", <-randomChannel)中就可以将randomChannel中的值提取出来
  - 暂停1s也可以减慢随机数生成速度(因为接收者不接收，发送方就会一直阻塞)，避免程序卡顿
- 生成随机数机理
  - 利用crypto/rand生成随机数
  - 在后台不断生成随机数，利用用户输入的随机性得到随机值

randomNum2.go

```go
// code adapted from https://www.cnblogs.com/lfri/p/11769254.html
package main

import (
    "crypto/rand"
    "fmt"
    "math/big"
    "net"
    "strings"
    "time"
)


func main() {
    fmt.Println("Starting the server ...")
    // 创建 listener
    listener, err := net.Listen("tcp", "localhost:50000")
    if err != nil {
        fmt.Println("Error listening", err.Error())
        return //终止程序
    } // if
    // 监听并接受来自客户端的连接
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Error accepting", err.Error())
            return // 终止程序
        } else {
            fmt.Println("Someone is connected!")
        }
        go doServerStuff(conn)
    } // for
} // main

func randomNo(randomCh chan int) {
    go randHandler(randomCh)
    for {
        // use crypto/rand to generate a true random number
        n, _ := rand.Int(rand.Reader, big.NewInt(1000))
        randomNumber := int(n.Int64())
```

```
41          // fmt.Printf("The random number is: %v\n", randomNumber)
42          randomCh <- randomNumber
43      } // for
44  } // randomNo
45
46  func randHandler(randomCh chan int) {
47      for{
48          // wait for 1 second before receive so that the doServerS
    tuff can get the
49          // random number generated
50          time.Sleep(1 * 1e9) // sleep for 1 second
51          x := <-randomCh
52          fmt.Printf("The random number is: %v\n", x)
53      } // for
54  } // randHandler
55
56  func doServerStuff(conn net.Conn) {
57      randomChannel := make(chan int)
58      go randomNo(randomChannel)
59      fmt.Fprintf(conn, "Welcome to the random number generator, ge
    tting an int number within 1000!\n")
60      for {
61          buf := make([]byte, 512)
62          len, err := conn.Read(buf)
63          if err != nil {
64              fmt.Println("Error reading", err.Error())
65              return //终止程序
66          } // if
67          inputSting := strings.Trim(string(buf[:len]), "\r\n")
68          fmt.Printf("Received data: %v\n", inputSting)
69          fmt.Fprintf(conn, "Random number = %v\n", <-randomChannel
    )
70      } // for
71  } // doServerStuff
```

运行



改进版本二

- 改进：使用5个goroutine向管道中竞争传值实现随机
  - 比上一版本更加随机
  - 接收者的接收时间可以适当放缓，因为竞争机制已经能够保证随机，这里使用了5秒
- 生成随机数机理
  - 利用crypto/rand生成随机数
  - 在后台不断生成随机数，利用用户输入的随机性得到随机值
  - 使用5个goroutine相互竞争往管道中传值的随机性

randomNum3.go

```go
// code adapted from https://www.cnblogs.com/lfri/p/11769254.html
package main

import (
    "crypto/rand"
    "fmt"
    "math/big"
    "net"
    "strings"
```

```
    "time"
)


func main() {
    fmt.Println("Starting the server ...")
    // 创建 listener
    listener, err := net.Listen("tcp", "localhost:50000")
    if err != nil {
        fmt.Println("Error listening", err.Error())
        return //终止程序
    } // if
    // 监听并接受来自客户端的连接
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Error accepting", err.Error())
            return // 终止程序
        } else {
            fmt.Println("Someone is connected!")
        }
        go doServerStuff(conn)
    } // for
} // main

func randomNo(randomCh chan int) {
    for {
        // use crypto/rand to generate a true random number
        n, _ := rand.Int(rand.Reader, big.NewInt(1000))
        randomNumber := int(n.Int64())
        // fmt.Printf("The random number is: %v\n", randomNumber)
        randomCh <- randomNumber
    } // for
} // randomNo

func randHandler(randomCh chan int) {
    for{
        // wait for 5 seconds before receive so that the doServer
Stuff can get the
        // random number generated
```

```
        time.Sleep(5 * 1e9) // sleep for 5 seconds
        x := <-randomCh
        fmt.Printf("The random number is: %v\n", x)
    } // for
} // randHandler

func doServerStuff(conn net.Conn) {
    randomChannel := make(chan int)
    // create 5 goroutines
    for i := 1; i < 6; i++{
        go randomNo(randomChannel)
    } // for
    go randHandler(randomChannel)
    fmt.Fprintf(conn, "Welcome to the random number generator, ge
tting an int number within 1000!\n")
    for {
        buf := make([]byte, 512)
        len, err := conn.Read(buf)
        if err != nil {
            fmt.Println("Error reading", err.Error())
            return //终止程序
        } // if
        inputSting := strings.Trim(string(buf[:len]), "\r\n")
        fmt.Printf("Received data: %v\n", inputSting)
        fmt.Fprintf(conn, "Random number = %v\n", <-randomChannel
)
    } // for
} // doServerStuff
```

运行结果和上一个版本类似

代码在GitHub的practice-go-project repository中的practice-go-project目录下可见，此练习为task3
https://github.com/YechengChu/practice-go-project/tree/master/proj2

参考资料

《Go语言实战》《Go in Action》

第6章 并发

《Go语言编程》

第4章 并发编程

Go语言圣经（中文版）

https://books.studygolang.com/gopl-zh/index.html

《Go入门指南》

https://github.com/unknwon/the-way-to-go_ZH_CN

ruilisi/golang-examples/channel/pass-ball.go

https://github.com/ruilisi/golang-examples/blob/master/channel/pass-ball.go

[Go]goroutine、go的并发与并行、channels

https://blog.csdn.net/weixin_43446767/article/details/98868704

Rogn

https://www.cnblogs.com/lfri/p/11769254.html

8.2. 示例: 并发的Clock服务

https://books.studygolang.com/gopl-zh/ch8/ch8-02.html

Go by Example: Random Numbers

https://gobyexample.com/random-numbers

Golang随机数生成

https://www.cnblogs.com/jukaiit/p/10785433.html

golang 生成随机数或者字符

https://zhuanlan.zhihu.com/p/94684495

Convert between int, int64 and string

https://yourbasic.org/golang/convert-int-to-string/

Golang 学习笔记10

2020/08/03

cyc/Golang 学习笔记/Golang学习笔记10

测试 Testing

go test工具和测试变量及结构体

- go test命令是一个按照一定的约定和组织来测试代码的程序。在包目录内，所有以`_test.go`为后缀名的源文件在执行go build时不会被构建成包的一部分，它们是go test测试的一部分。
- 在`*_test.go`文件中，有三种类型的函数比较特殊
  - 功能测试函数(test function)：以Test为函数名前缀的函数，用于测试程序的一些逻辑行为是否正确。go test命令会调用这些测试函数并报告测试结果是PASS或FAIL。
  - 基准测试函数(benchmark function)：以Benchmark为函数名前缀的函数，它们用于衡量一些函数的性能。go test命令会多次运行基准测试函数以计算一个平均的执行时间。
  - 示例函数(example function)：以Example为函数名前缀的函数，提供一个由编译器保证正确性的示例文档。

```
TestXxxx(t *testing.T)     // 基本测试用例
BenchmarkXxxx(b *testing.B) // 压力测试的测试用例
Example_Xxx()  // 测试控制台输出的例子
TestMain(m *testing.M) // 测试 Main 函数
```

testing的变量

gotest的变量有这些：

- test.short：一个快速测试的标记，在测试用例中可以使用 testing.Short() 来绕开一些测试
- test.outputdir：输出目录
- test.coverprofile：测试覆盖率参数，指定输出文件
- test.run：指定正则来运行某个 / 某些测试用例
- test.memprofile：内存分析参数，指定输出文件
- test.memprofilerate：内存分析参数，内存分析的抽样率
- test.cpuprofile：cpu 分析输出参数，为空则不做 cpu 分析
- test.blockprofile：阻塞事件的分析参数，指定输出文件
- test.blockprofilerate：阻塞事件的分析参数，指定抽样频率

- test.timeout：超时时间
- test.cpu：指定 cpu 数量
- test.parallel：指定运行测试用例的并行数

testing的结构体

- B：压力测试
- BenchmarkResult：压力测试结果
- Cover：代码覆盖率相关结构体
- CoverBlock：代码覆盖率相关结构体
- InternalBenchmark：内部使用的结构体
- InternalExample：内部使用的结构体
- InternalTest：内部使用的结构体
- M：main 测试使用的结构体
- PB：Parallel benchmarks 并行测试使用的结构体
- T：普通测试用例
- TB：测试用例的接口

单元测试 Unit testing

单元测试是用来测试包或者程序的一部分代码或者一组代码的函数。测试的目的是确认目标代码在给定的场景下，有没有按照期望工作。

- `testing`为Go语言package提供自动化测试的支持。通过`go test`命令，能够自动执行如下形式的任何函数

```
1 func TestXxx(*testing.T)
```

  - 注意⚠️：Xxx 可以是任何字母数字字符串，但是第一个字母不能是小写字母。如，TestFmtInterface，TestPayEmployees，TestCos，TestLog等。
  - 不但函数名字要以Test开头，函数的签名还必须接收一个指向testing.T 类型的指针，并且不返回任何值
  - 指向testing.T类型的指针很重要。这个指针提供的机制可以报告每个测试的输出和状态。
- 在这些函数中，使用`Error`、`Fail`或相关方法来发出失败信号，这里仅作简单介绍，下文将详细介绍。
  - `func (t *T) Fail()`：标记测试函数为失败，然后继续执行剩下的测试
  - `func (t *T) FailNow()`：标记测试函数为失败并中止执行；文件中别的测试也被略过，继续执行下一个文件
  - `func (t *T) Log(args ...interface{})`：args被用默认的格式格式化并打印到错误日志中
  - `func (t *T) Fatal(args ...interface{})`：效果就和先执行`Log`，然后执行`FailNow`一样

- 运行测试代码时可以调用`go test -v`(-v 表示提供冗余输出，每个执行的测试函数以及测试状态会被打印)。不加的话，除非测试失败，否则我们是看不到任何测试输出的。
- 写测试代码时可以使用`t.Errorf`和`t.Logf`来格式化打印失败信息和日志

表组测试 Table-Driven Test

- 如果测试可以接受一组不同的输入并产生不同的输出的代码，那么应该使用表组测试的方法进行测试。
- 表组测试除了会有一组不同的输入值和期望结果之外，其余部分都很像基础单元测试。
- 测试会依次迭代不同的值，来运行要测试的代码。
- 每次迭代的时候，都会检测返回的结果。
- 这便于在一个函数里测试不同的输入值和条件。

fib.go

```go
package main

func Fib(n int) int {
    if n < 2 {
        return n
    } // if
    return Fib(n-1) + Fib(n-2)
} // Fib
```

fib_test.go

```go
package main

import "testing"

// 对号 (√)
const checkMark = "\u2713"
// 叉号 (×)
const ballotX = "\u2717"

func TestFib(t *testing.T) {
    var fibTests = []struct {
        in       int // input value
        expected int // expected result
    }{
        // pairs of input value and expected result
        {1, 1},
```

```
        {2, 1},
        {3, 2},
        {4, 3},
        {5, 5},
        {6, 8},
        // add a failing test
        // {7,10},
        {7, 13},
    }

    // for elements in fibTests
    for _, tt := range fibTests {
        // call Fib function use in value
        actual := Fib(tt.in)
        if actual != tt.expected {
            // if actual result not equal to expected result
            // print error
            t.Errorf("%v Fib(%d) = %d; expected %d", ballotX, tt.
in, actual, tt.expected)
        } else {
            // else, print log
            t.Logf("%v Fib(%d) = %d", checkMark, tt.in, actual)
        }
    } // for
} // TestFib
```

运行结果

```
$ go test
PASS
ok      _/Users/cyc/Projects/exe/test/table-driven  0.400s


$ go test -v
=== RUN   TestFib
    TestFib: fib_test.go:36: ✓ Fib(1) = 1
    TestFib: fib_test.go:36: ✓ Fib(2) = 1
    TestFib: fib_test.go:36: ✓ Fib(3) = 2
    TestFib: fib_test.go:36: ✓ Fib(4) = 3
```

```
    TestFib: fib_test.go:36: ✓ Fib(5) = 5
    TestFib: fib_test.go:36: ✓ Fib(6) = 8
    TestFib: fib_test.go:36: ✓ Fib(7) = 13
--- PASS: TestFib (0.00s)
PASS
ok      _/Users/cyc/Projects/exe/test/table-driven  0.210s
```

如果把那行错误的值{7,10}uncomment，运行结果如下。因为测试失败了，所以即使不加-v也会显示测试输出。

```
$ go test
--- FAIL: TestFib (0.00s)
    fib_test.go:36: ✓ Fib(1) = 1
    fib_test.go:36: ✓ Fib(2) = 1
    fib_test.go:36: ✓ Fib(3) = 2
    fib_test.go:36: ✓ Fib(4) = 3
    fib_test.go:36: ✓ Fib(5) = 5
    fib_test.go:36: ✓ Fib(6) = 8
    fib_test.go:33: ✗ Fib(7) = 13; expected 10
    fib_test.go:36: ✓ Fib(7) = 13
FAIL
exit status 1
FAIL    _/Users/cyc/Projects/exe/test/table-driven  1.385s
```

报告方法

- 当我们遇到一个断言错误的时候，标识这个测试失败

```
Fail : 测试失败，测试继续，也就是之后的代码依然会执行
FailNow : 测试失败，测试中断
```

- 当我们遇到一个断言错误，只希望跳过这个错误，但是不希望标识测试失败

```
SkipNow : 跳过测试，测试中断
```

- 当我们只希望打印信息

```
Log : 输出信息
Logf : 输出格式化的信息
```

- 当我们希望跳过这个测试，并且打印出信息

```
Skip : 相当于 Log + SkipNow
Skipf : 相当于 Logf + SkipNow
```

- 当我们希望断言失败的时候，标识测试失败，并打印出必要的信息，但是测试继续

```
Error : 相当于 Log + Fail
Errorf : 相当于 Logf + Fail
```

- 当我们希望断言失败的时候，标识测试失败，打印出必要的信息，但中断测试

```
Fatal : 相当于 Log + FailNow
Fatalf : 相当于 Logf + FailNow
```

基准测试 Benchmarking

基准测试是一种测试代码性能的方法。想要测试解决同一问题的不同方案的性能，以及查看哪种解决方案的性能更好时，基准测试就会很有用。基准测试能测量一个程序在固定工作负载下的性能。

- 在 _test.go结尾的测试文件中，如下形式的函数：

```
func BenchmarkXxx(*testing.B)
```

- 基准测试函数必须以Benchmark开头，接受一个指向testing.B类型的指针作为唯一参数。
- `*testing.B`参数除了提供和`*testing.T`类似的方法，还有额外一些和性能测量相关的方法。它还提供了一个整数N，用于指定操作执行的循环次数。
- 被认为是基准测试，通过`go test`命令，加上`-bench`标志来执行。多个基准测试按照顺序运行。基准测试函数的形式如下

```
func BenchmarkHello(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fmt.Sprintf("hello")
    } // for
```

```
} // BenchmarkHello
```

- 基准函数会运行目标代码 b.N 次
- 在基准执行期间，程序会自动调整 b.N 直到基准测试函数持续足够长的时间

```
BenchmarkHello    10000000    282 ns/op
```

- 意味着循环执行了 10000000 次，每次循环花费 282 纳秒 (ns)

基准测试示例

延续单元测试的test文件，加入一些基准测试的代码

fib_test.go

```go
package main

import "testing"

// 对号 (√)
const checkMark = "\u2713"
// 叉号 (×)
const ballotX = "\u2717"

func TestFib(t *testing.T) {
    var fibTests = []struct {
        in       int // input value
        expected int // expected result
    }{
        // pairs of input value and expected result
        {1, 1},
        {2, 1},
        {3, 2},
        {4, 3},
        {5, 5},
        {6, 8},
        // add a failing test
        // {7,10},
        {7, 13},
    }
```

```
26
27      // for elements in fibTests
28      for _, tt := range fibTests {
29          // call Fib function use in value
30          actual := Fib(tt.in)
31          if actual != tt.expected {
32              // if actual result not equal to expected result
33              // print error
34              t.Errorf("%v Fib(%d) = %d; expected %d", ballotX, tt.
    in, actual, tt.expected)
35          } else {
36              // else, print log
37              t.Logf("%v Fib(%d) = %d", checkMark, tt.in, actual)
38          }
39      } // for
40  } // TestFib
41
42  func BenchmarkFib1(b *testing.B)  { benchmarkFib(1, b) }
43  func BenchmarkFib2(b *testing.B)  { benchmarkFib(2, b) }
44  func BenchmarkFib3(b *testing.B)  { benchmarkFib(3, b) }
45  func BenchmarkFib10(b *testing.B) { benchmarkFib(10, b) }
46  func BenchmarkFib20(b *testing.B) { benchmarkFib(20, b) }
47  func BenchmarkFib40(b *testing.B) { benchmarkFib(40, b) }
48
49  func benchmarkFib(i int, b *testing.B) {
50      for n := 0; n < b.N; n++ {
51          Fib(i)
52      } // for
53  } // benchmarkFib
```

运行结果

```
1  $ go test -bench=.
2  goos: darwin
3  goarch: amd64
4  BenchmarkFib1-4     448914968            2.82 ns/op
5  BenchmarkFib2-4     164449497            7.24 ns/op
6  BenchmarkFib3-4     91887103            12.6 ns/op
7  BenchmarkFib10-4     2624431           450 ns/op
```

```
BenchmarkFib20-4       20455         56821 ns/op
BenchmarkFib40-4           2     870592587 ns/op
PASS
ok      _/Users/cyc/Projects/exe/test/table-driven  14.134s
```

- 和普通测试不同的是，默认情况下不运行任何基准测试。
- 我们需要通过`-bench`命令行标志参数手工指定要运行的基准测试函数。
- 该参数是一个正则表达式，用于匹配要执行的基准测试函数的名字，默认值是空的。其中“.”模式将可以匹配所有基准测试函数。

参考资料

《Go语言实战》《Go in Action》
第9章 测试和性能
Go语言圣经（中文版）
https://books.studygolang.com/gopl-zh/index.html
《Go入门指南》
https://github.com/unknwon/the-way-to-go_ZH_CN
ruilisi/golang-examples/test
https://github.com/ruilisi/golang-examples/tree/master/test
《Go语言标准库》第九章 测试
https://books.studygolang.com/The-Golang-Standard-Library-by-Example/chapter09/09.0.html
Golang 测试
https://sanyuesha.com/2019/08/21/go-test/