C/C++Linux服务器开发

高级架构师课程

三年课程沉淀 | 五次精益升级 | 十年行业积累 | 百个实战项目 | 十万内容受众

讲师:darren/326873713

扫一扫 升职加薪

班主任:柚子/2690491738

讲师介绍--专业来自专注和实力

King老师

系统架构师，曾供职著名创业公司系统架构师，微软亚洲研究院、创维集团全球研发中心。国内第一代商业Paas平台开发者。著有多个软件专利，参与多个开源软件维护。在全球化，高可用的物联网云平台架构与智能硬件设计方面有丰富的研发与实战经验。

Darren老师

曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。

GO语言基础精讲和test方法

1. 初识Go语言
2. Go语言基础
3. Go函数
4. Go数组和切片
5. Go test方法

1.1 开发环境搭建

参考文档：《Windows Go语言环境搭建》
Linux MAC系统参考《Windows Go语言环境搭建》文档中的链接。

1.2.1 Go语言特性-垃圾回收

a. 内存自动回收，再也不需要开发人员管理内存
b. 开发人员专注业务实现，降低了心智负担
c. 只需要new分配内存，不需要释放

d. gc 垃圾回收　4ms

1.2.2 Go语言特性-天然并发

a. 从语言层面支持并发，非常简单
b. goroute，轻量级线程，创建成千上万个goroute成为可能
c. 基于CSP（Communicating Sequential Process）模型实现

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

代码：
1-1-go.go
1-1-goroute

进一步阅读：《GO Channel并发、死锁问题》https://www.cnblogs.com/show58/p/12699083.html
《Go的CSP并发模型实现：M, P, G》https://www.cnblogs.com/sunsky303/p/9115530.html

1.2.3 Go语言特性-channel

a. 管道，类似unix/linux中的pipe
b. 多个goroute之间通过channel进行通信
c. 支持任何类型

```
func main() {
  pipe := make(chan int,3)
pipe <- 1
pipe <- 2
}
```

1-2-channel.go

1.2.4 Go语言特性-多返回值

一个函数返回多个值

```
func calc(a int, b int)(int,int) {
    sum := a + b
    avg := (a+b)/2
    return sum, avg
}
```

代码：1-3-package

1.3 第一个golang程序

```
package main

import(
   “fmt”
)

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

代码：1-4-hello.go

1.4.1包的概念

1. 和python一样，把相同功能的代码放到一个目录，称之为包
2. 包可以被其他包引用
3. main包是用来生成可执行文件，每个程序只有一个main包
4. 包的主要用途是提高代码的可复用性

1.4.2 包的实战

1. 新建calc目录

2. 在calc目录下新建calc.go

```
package calc

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

代码：1-3-package

1. 4. 2 包的实战

1. 修改hello.go代码，如下

```
package main

import(
    “fmt”
    “calc”
)

func main() {
    sum := calc.Add(3,5)
    fmt.Println(“hello world,%d”,sum)
}
```

2 基本数据类型和操作符

1. 文件名&关键字&标识符

2. Go程序基本结构

3. 常量和变量

4. 数据类型和操作符

5. 字符串类型

2. 1文件名&关键字&标识符

1. 所有go源码以.go结尾
2. 标识符以字母或下划线开头，大小写敏感，比如：

a. boy　　b. Boy　　c. a+b　　d. 0boy

e. _boy　　f. =_boy　　g. _

3. _是特殊标识符，用来忽略结果

4. 保留关键字

2. 1文件名&关键字&标识符-关键字1

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

2. 1文件名&关键字&标识符-关键字2

- var和const： 变量和常量的声明
  var varName type 或者 varName : = value
- package and import: 包和导入
- func： 用于定义函数和方法
- return： 用于从函数返回
- defer someCode： 在函数退出之前执行
- go : 用于并行
- select 用于选择不同类型的通讯
- interface 用于定义接口
- struct 用于定义抽象数据类型

2. 1文件名&关键字&标识符-关键字3

- break、case、continue、for、fallthrough、else、if、switch、goto、default 流程控制
- fallthrough的用法注意总结 [推荐阅读 https://blog.csdn.net/ajdfhajdkfakr/article/details/79086125]
  - 1.加了fallthrough后，会直接运行【紧跟的后一个】case或default语句，不论条件是否满足都会执行
  - 2.加了fallthrough语句后，【紧跟的后一个】case条件不能定义常量和变量
  - 3.执行完fallthrough后直接跳到下一个条件语句，本条件执行语句后面的语句不执行

- chan用于channel通讯
- type用于声明自定义类型
- map用于声明map类型数据
- range用于读取slice、map、channel数据

2.2 Go程序的基本结构1

```
package main
import “fmt”
func main() {
    fmt.Println(“hello, world”)
}
```

1. 任何一个代码文件隶属于一个包

2. import 关键字，引用其他包：

```
import(“fmt”)
import(“os”)
```

通常习惯写成：

```
import (
    “fmt”
    “os”
)
```

2.2 Go程序的基本结构2

```
package main
import “fmt”
func main() {
    fmt.Println(“hello, world”)
}
```

3. golang可执行程序，package main，并且有且只有一个main入口函数

4. 包中函数调用：

   a. 同一个包中函数，直接调用

   b. 不同包中函数，通过包名+点+函数名进行调用

5. 包访问控制规则：

   a. 大写意味着这个函数/变量是可导出的
   b. 小写意味着这个函数/变量是私有的，包外部不能访问

2. 3函数声明和注释

1. 函数声明： func 函数名字 (参数列表) (返回值列表) {} **举例：**

```
func add() {

}
```

```
func add(a int , b int) int {

}
```

```
func add(a int , b int) (int, int) {

}
```

2. 注释，两种注释，单行注释: // 和多行注释 /* */ **举例：**

```
//add 计算两个整数的和，并返回结果
func add(a int, b int) int {

}
```

2.4 常量1

1. 常量使用const 修饰，代表永远是只读的，不能修改。
2. const 只能修饰boolean，number（int相关类型、浮点类型、complex）和string。
3. 语法：const identifier [type] = value，其中type可以省略。

举例：

```
const b string = “hello world”
const b = “hello world”
const Pi = 3.1414926
const a = 9/3
const c = getValue()
```

类型的自动推导

代码：2-4-const.go

2. 4 常量2

4. 比较优雅的写法：

尽量减少我们写代码

```
const (
a = 0
b = 1
c = 2
)
```

5. 更加专业的写法：

```
const (
a = iota
b    //1
c    //2
)
```

2.5 变量1

1. 语法： var identifier type

举例1：

```
var b string
var c  bool
var d int = 8
var e string = “hello world”
```

2.5 变量2

```
Var (
        a int           //默认为0
        b string        //默认为""
        c bool          //默认为false
        d int = 8
        e string = "hello world"
)
```

```
Var (
        a int           //默认为0
        b string        //默认为""
        c bool          //默认为false
        d = 8
        e = "hello world"
)
```

练习

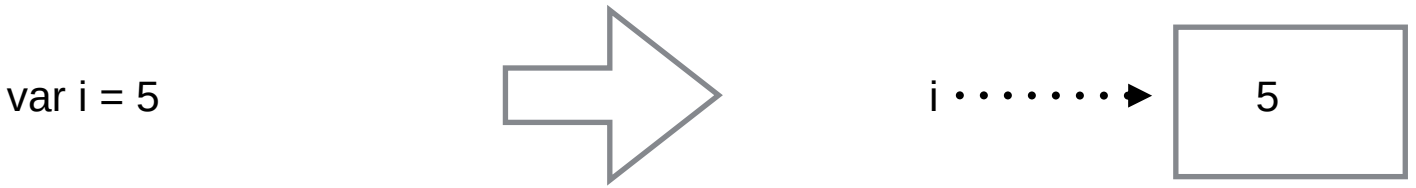
写一个程序获取当前运行的操作系统名称和PATH环境环境变量的值，并打印在终端。

```
package main
import (
    “fmt”
    “os”
)

func main() {
    var goos string = os.Getenv(“GOOS”)
    fmt.Printf(“The operating system is: %s\n”, goos)
    path := os.Getenv(“PATH”)
    fmt.Printf(“Path is %s\n”, path)
}
```

2.6 值类型和引用类型

1. 值类型：变量直接存储值，内存通常在栈中分配。



2. 引用类型：变量存储的是一个地址，这个地址存储最终的值。内存通常在堆上分配。通过GC回收。

2. 6值类型和引用类型

1. 值类型：基本数据类型int、float、bool、string以及数组和struct。

2. 引用类型：指针、slice、map、chan等都是引用类型。

练习：写一个程序，交换两个整数的值。比如： a=3; b=4; 交换之后： a=4;b=3

代码：2-6-swap.go

练习：写一个程序用来打印值类型和引用类型变量到终端，并观察输出结果。
代码： 2-6-value_quote.go

2.7 变量的作用域

1. 在函数内部声明的变量叫做局部变量，生命周期仅限于函数内部。

2.在函数外部声明的变量叫做全局变量，生命周期作用于整个包，如果是大写的，则作用于整个程序。

练习

源码： 2-7-var.go

请指出下面程序的输出是什么

```
package main

var a = “G”
func main() {
      n()   // G
      m() // O
      n()  // G
}
func n() {
     fmt.Println(a)
}
func m() {
     a := “O”  // 修改的是局部变量的
     fmt.Println(a)
}
```

```
package main

var a = “G”
func main() {
      n()  // G
      m()  // O
      n()  //O
}
func n() {
     fmt.Println(a)
}
func m() {
     a = “O”
     fmt.Println(a)
}
```

2.8 数据类型和操作符1

1. bool类型，只能存true和false

```
var a bool
var a bool = true
var a = true
```

2. 相关操作符，！、&&、||

```
var a bool = true
var b

请问!a 、!b、a && b、a || b的值分别是多少?
```

2.8 数据类型和操作符2

3. 数字类型，主要有int、int8、int16、int32、int64、uint8、uint16、uint32、uint64、float32、float64

4. 类型转换，type(variable），比如：var a int=8; var b int32=int32(a)

```
package main
func main() {
    var a int
    var b int32
    a = 15
    b = a + a // compiler error
    b = b + 5 // ok: 5 is a constant
}
```

2. 8 数据类型和操作符3

5. 字符类型：var a byte

   var a byte = ‘c’

6. 字符串类型： var str string

```
package main
import “fmt”
func main() {
    var str = “hello world”
        fmt.Println(“str=“, str)
}
```

练习

练习：请指出下面程序的输出是什么？

代码：2-8-type.go

```
package main

import "fmt"

func main() {
    var n int16 = 34
    var m int32

    m = n
    m = int32(n)
    fmt.Printf("32 bit int is: %d\n", m)
    fmt.Printf("16 bit int is:  %d\n", n)
}
```

练习：使用math/rand生成10个随机整数，10个小于100的随机整数以及10个随机浮点数 代码：

2. 9数据类型和操作符1

1. 逻辑操作符： == 、!=、<、<=、>和 >=

```
package main
import ”fmt”
func main() {
    var a int = 10
     if ( a > 10 ) {
         fmt.Println(a)
     }
}
```

2. 9数据类型和操作符2

2. 数学操作符： +、-、*、/等等

```
package main
Import ”fmt”
func main() {
    var a int = 10
    var b = a + 10
}
```

2.9 数据类型和操作符3

字符串表示两种方式： 1）双引号 2）`` （反引号）

```
package main
import “fmt”
func main() {
    var str = “hello world\n\n”
    var str2 = `hello \n \n \n
this is a test string
This is a test string too·`
        fmt.Println(“str=“, str)
        fmt.Println(“str2=“, str2)
}
```

源码： 2-9-string.go

3 流程控制

for range 语句

```
str := "hello world,中国"
for i, v := range str {
    fmt.Printf("index[%d] val[%c]\n", i, v)
}
```

用来遍历数组、slice、map、chan。

3 函数1

1. 声明语法：func 函数名 (参数列表) [(返回值列表)] {}

```
func add()
{

}
```

3 函数1

1. 声明语法：func 函数名 (参数列表) [(返回值列表)] {}

```
func add()
{

}
```

3 函数2

2. golang函数特点：

a. 不支持重载，一个包不能有两个名字一样的函数

b. 函数是一等公民，函数也是一种类型，一个函数可以赋值给变量

c. 匿名函数

d. 多返回值

3 函数2

2. golang函数特点：

```
package main

import "fmt"

func add(a, b int) int {
	return a + b
}

func main() {

	c := add
	fmt.Println(c)

	sum := c(10, 20)
	fmt.Println(sum)
	sf1 := reflect.ValueOf(c)
	sf2 := reflect.ValueOf(add)
	if sf1 == sf2  {
		fmt.Println("c equal add")
	}
}
```

```
package main

import "fmt"

type add_func func(int, int) int

func add(a, b int) int {
	return a + b
}

func operator(op add_func, a int, b int) int {
	return op(a, b)
}

func main() {
	c := add
	fmt.Println(c)
	sum := operator(c, 100, 200)
	fmt.Println(sum)
}
```

3 函数3

3. 函数参数传递方式：

1). 值传递

2). 引用传递

注意1：无论是值传递，还是引用传递，传递给函数的都是变量的副本，不过，值传递是值的拷贝。引用传递是地址的拷贝，一般来说，地址

拷贝更为高效。而值拷贝取决于拷贝的对象大小，对象越大，则性能越低。

3 函数3

3. 函数参数传递方式： 1). 值传递

2). 引用传递

注意2：map、slice、chan、指针、interface默认以引用的方式传递

```
package main

import "fmt"

func modify(a int) {
	a = 100
}

func main() {
	a := 8
	fmt.Println(a)
	modify(a)
	fmt.Println(a)
}
```

3 函数4

4. 命名返回值的名字

```
func add(a, b int) (c int) {
    c = a + b
    return
}

func calc(a, b int) (sum int, avg int) {
    sum = a + b
    avg = (a +b)/2
    return
}
```

3 函数5

5. _标识符，用来忽略返回值:

```
func calc(a, b int) (sum int, avg int) {
        sum = a + b
        avg = (a +b)/2
        return
}
func main() {
      sum, _ := calc(100, 200)
}
```

3 函数6

6. 可变参数:

```
func add(arg…int) int {
}
```

0个或多个参数

```
func add(a int, arg…int) int {
}
```

1个或多个参数

```
func add(a int, b int, arg…int) int {
}
```

2个或多个参数

注意：其中arg是一个slice，我们可以通过arg[index]依次访问所有参数

通过len(arg)来判断传递参数的个数

代码：3-6-var.go

3 函数7

7. defer用途：

1. 当函数返回时，执行defer语句。因此，可以用来做资源清理

2. 多个defer语句，按先进后出的方式执行

3. defer语句中的变量，在defer声明时就决定了。

3 函数7 defer用途

```
func a() {
        i := 0
        defer fmt.Println(i)
        i++
        return
}
```

打印是多少?

```
func f() {
        for i := 0; i < 5; i++ {
                defer fmt.Printf("%d ", i)
        }
}
```

3 函数7 defer用途

1 关闭文件句柄

```
func read() {
    file := open(filename)
    defer file.Close()

    //文件操作
}
```

3 函数7 defer用途

2. 锁资源释放

```
func read() {
    mc.Lock()
    defer mc.Unlock()
    //其他操作
}
```

3 函数7 defer用途

3. 数据库连接释放

```
func read() {
    conn := openDatabase()
    defer conn.Close()
    //其他操作
}
```

4 常用结构

1. 内置函数、闭包

2. 数组与切片

3. map数据结构

4. package介绍

4.1 内置函数

1. close：主要用来关闭channel

2. len：用来求长度，比如string、array、slice、map、channel

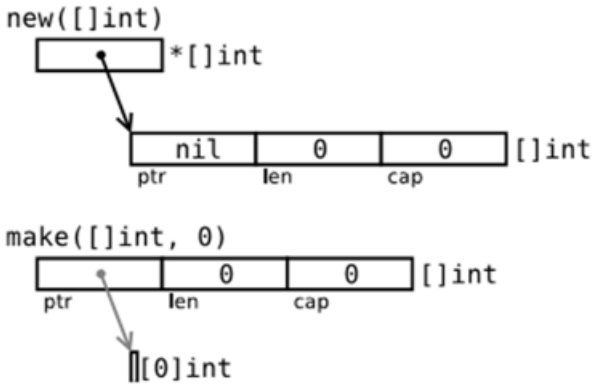
3. new：用来分配内存，主要用来分配值类型，比如int、struct。返回的是指针

4. make：用来分配内存，主要用来分配引用类型，比如chan、map、slice

5. append：用来追加元素到数组、slice中

6. panic和recover：用来做错误处理

7. new和make的区别



4. 2闭包

1. 闭包：一个函数和与其相关的引用环境组合而成的实体

```
package main

import “fmt”

func main() {
        var f = Adder()
        fmt.Print(f(1),” - “)
        fmt.Print(f(20),” - “)
        fmt.Print(f(300))
}

func Adder() func(int) int {
        var x int
        return func(delta int) int {
                x += delta
                return x
        }
}
```

4. 2闭包 例子

```
package main

import (
	"fmt"
	"strings"
)
func makeSuffixFunc(suffix string) func(string) string {
	return func(name string) string {
		if !strings.HasSuffix(name, suffix) {
			return name + suffix
		}
		return name
	}
}
func main() {
	func1 := makeSuffixFunc(".bmp")
	func2 := makeSuffixFunc(".jpg")
	fmt.Println(func1("test"))
	fmt.Println(func2("test"))
}
```

4. 3数组与切片

1. 数组：是同一种数据类型的固定长度的序列。

2. 数组定义：var a [len]int，比如：var a[5]int 一旦定义，长度不能变

3. 长度是数组类型的一部分，因此，var a[5] int和var a[10]int是不同的类型

4. 数组可以通过下标进行访问，下标是从0开始，最后一个元素下标是：len-1

```
for i := 0; i < len(a); i++ {
}
```

5. 访问越界，如果下标在数组合法范围之外，则触发访问越界，会panic

6. 数组是值类型，因此改变副本的值，不会改变本身的值

```
arr2 := arr1
arr2[2] = 100
```

4. 3数组与切片-案例

```go
package main

import (
	"fmt"
)

func modify(arr [5]int) {
	arr[0] = 100
	return
}

func main() {
	var a [5]int //数组大小是固定的

	modify(a)
	for i := 0; i < len(a); i++ {
		fmt.Println(a[i])
	}
}
```

```go
package main

import (
	"fmt"
)

func modify(arr *[5]int) {
	(*arr)[0] = 100
	return
}

func main() {
	var a [5]int

	modify(&a)
	for i := 0; i < len(a); i++ {
		fmt.Println(a[i])
	}
}
```

4-3-array.go

4. 3 数组与切片-数组

1. 数组初始化 对于数组 []里面肯定要有东西

a. var age0 [5]int = [5]int{1,2,3}

b. var age1 = [5]int{1,2,3,4,5}

c. var age2 = […]int{1,2,3,4,5,6}

d. var str = [5]string{3:”hello world”, 4:”tom”}

2. 多维数组

a. var age [5][3]int

b. var f [2][3]int = [...][3]int{{1, 2, 3}, {7, 8, 9}}

4-3-arrays.go

3. 多维数组遍历

```
package main

import (
        "fmt"
)

func main() {

        var f [2][3]int = [...][3]int{{1, 2, 3}, {7, 8, 9}}

        for k1, v1 := range f {
                for k2, v2 := range v1 {
                        fmt.Printf("(%d,%d)=%d ", k1, k2, v2)
                }
                fmt.Println()
        }
}
```

4.3 数组与切片-切片定义

1. 切片：切片是数组的一个引用，因此切片是引用类型

2. 切片的长度可以改变，因此，切片是一个可变的数组

3. 切片遍历方式和数组一样，可以用len()求长度

4. cap可以求出slice最大的容量，0 <= len(slice) <= (array)，其中array是slice引用的数组

5. 切片的定义：var 变量名 []类型，比如 var str []string var arr []int

4. 3 数组与切片-切片初始化

1. 切片初始化：var slice []int = arr[start:end]

   包含start到end之间的元素，但不包含end

2. Var slice []int = arr[0:end]可以简写为 var slice []int=arr[:end]

3. Var slice []int = arr[start:len(arr)] 可以简写为 var slice[]int = arr[start:]

4. Var slice []int = arr[0, len(arr)] 可以简写为 var slice[]int = arr[:]

5. 如果要切片最后一个元素去掉，可以这么写：

   Slice = slice[:len(slice)-1]

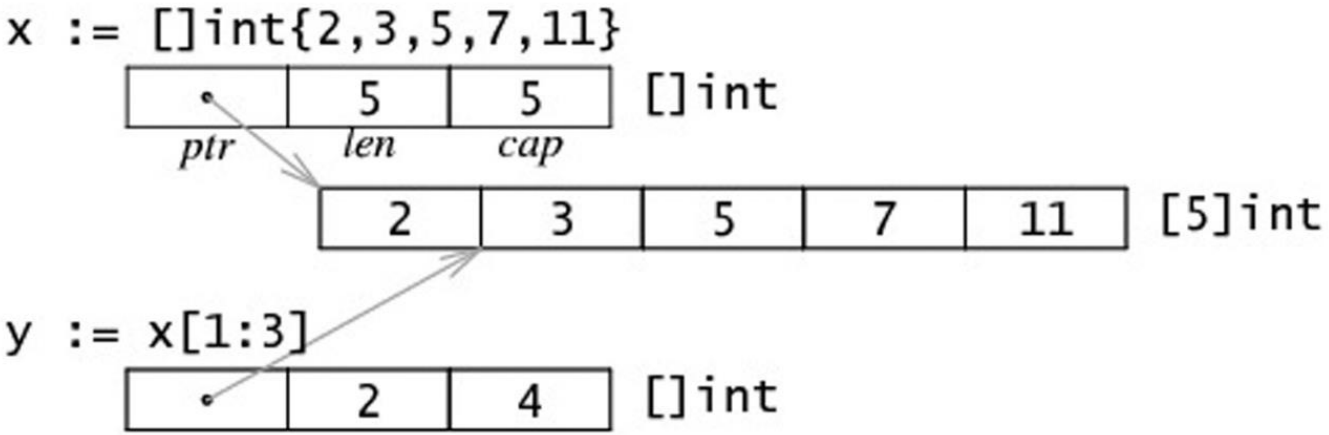
4.3 数组与切片-切片实战1

1. 练习：写一个程序，演示切片的各个用法

代码：4-3-slice1.go

4. 3 数组与切片-切片实战2

2. 切片的内存布局，类似C++ vector：



2. 练习，写一个程序，演示切片的内存布局

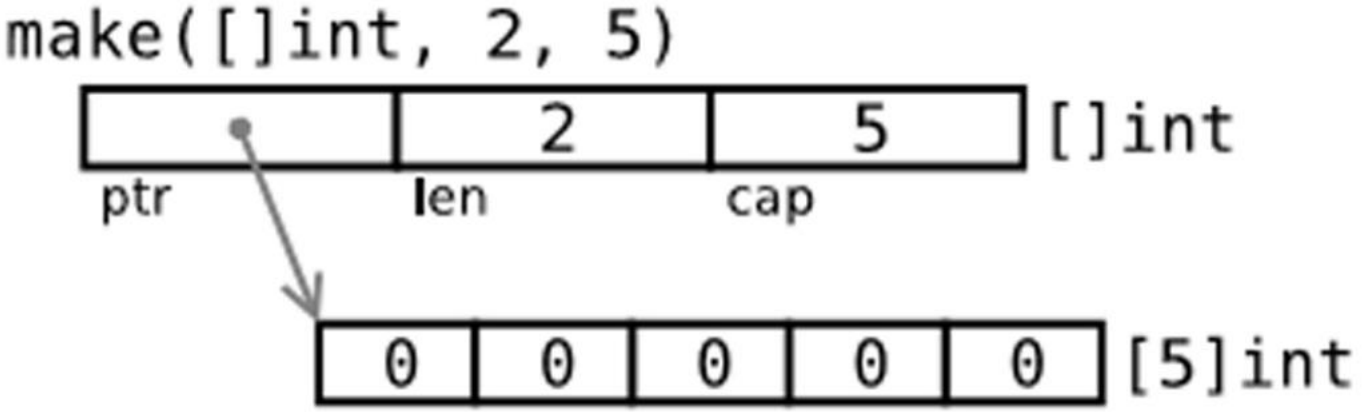
4. 3 数组与切片-切片实战3

3. 通过make来创建切片

```
var slice []type = make([]type, len)

slice  := make([]type, len)

slice  := make([]type, len, cap)
```



4. 3 数组与切片-切片实战4

4. 用append内置函数操作切片

```
slice = append(slice, 10)

var a = []int{1,2,3}
var b = []int{4,5,6}
a = append(a, b...)
```

5. For range 遍历切片

```
for index, val := range slice {
}
```

6. 切片resize

```
var a = []int {1,3,4,5}
b := a[1:2]
b = b[0:3]
```

7. 切片拷贝

```
s1 := []int{1,2,3,4,5}
s2 := make([]int, 10)
copy(s2, s1)
s3 := []int{1,2,3}
s3 = append(s3, s2...)
s3 = append(s3, 4,5,6)
```

4-3-slice-make.go

4. 3 数组与切片-切片实战5

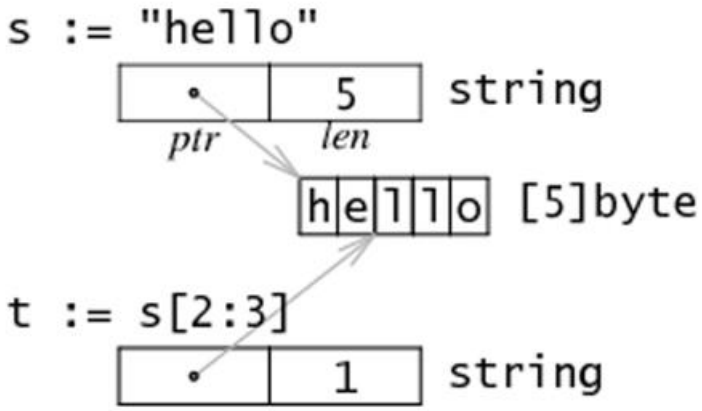
8. string与slice

string底层就是一个byte的数组，因此，也可以进行切片操作

```
str := “hello world”
s1 := str[0:5]
fmt.Println(s1)

s2 := str[5:]
fmt.Println(s2)
```

9. string的底层布局



4. 3 数组与切片-切片实战6

10. 如何改变string中的字符值?

string本身是不可变的，因此要改变string中字符，需要如下操作:

```
str := “hello world”
s := []byte(str)
s[0] = ‘o’
str = string(s)
```

4. 4 数组与切片的区别1

它们的定义:

- 数组: 类型 [n]T 表示拥有 n 个 T 类型的值的数组。
- 切片: 类型 []T 表示一个元素类型为 T 的切片。

数组的例子

```
var x[3]int = [3]int{1,2,3}
var y[3]int = x
fmt.Println(x,y)
y[0]=999
fmt.Println(x,y)
```

切片的例子

```
var x[]int = []int{1,2,3}
var y[]int = x
fmt.Println(x,y)
y[0]=999
fmt.Println(x,y)
```

4. 4 数组与切片的区别2

它们的定义:

- 数组：类型 [n]T 表示拥有 n 个 T 类型的值的数组。
- 切片：类型 []T 表示一个元素类型为 T 的切片。

数组是需要指定个数的，而切片则不需要。数组赋值也可是使用如下方式，忽略元素个数，使用“...”代替

```
x:= [...]int{1,2,3}
y := x
fmt.Println(x,y)
y[0]=999
fmt.Println(x,y)
```

4.5 new和make的区别

new

func new(Type) *Type

```
func main() {
 var i *int
 i=new(int)
 *i=10
 fmt.Println(*i)
}
```

make

func make(t Type, size ...IntegerType) Type

make也是用于内存分配的，但是和new不同，它只用于chan、map以及切片的内存创建，而且它返回的类型就是这三个类型本身，而不是他们的指针类型，因为这三种类型就是引用类型，所以就没有必要返回他们的指针了。

5 Go test

前置条件:
1、文件名须以"_test.go"结尾
2、方法名须以"Test"打头，并且形参为 (t *testing.T)

5 Go test 举例

举例：gotest.go

```
package mytest

import (
    "errors"
)

func Division(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("除数不能为0")
    }
    return a / b, nil
}
```

gotest_test.go

```
package mytest

import (
    "testing"
)

func Test_Division_1(t *testing.T) {
    if i, e := Division(6, 2); i != 3 || e != nil { //try a unit test on function
        t.Error("除法函数测试没通过") // 如果不是如预期的那么就报错
    } else {
        t.Log("第一个测试通过了") //记录一些你期望记录的信息
    }
}


func Test_Division_2(t *testing.T) {
    if _, e := Division(6, 0); e == nil { //try a unit test on function
        t.Error("Division did not work as expected.") // 如果不是如预期的那么就
报错
    } else {
        t.Log("one test passed.", e) //记录一些你期望记录的信息
    }
}
```

5 Go test 测试

1. 在目录下执行 **go test** 是测试目录所有以**XXX_test.go** 结尾的文件。

2.测试单个方法

```
go test -v  -run="Test_Division_1" -count 5
```

3.查看帮助 **go help test**

5 Go test 命令介绍1

通过go help test可以看到go test的使用说明:

格式形如:

go test [-c] [-i] [build flags] [packages] [flags for test binary]

参数解读:

-c : 编译go test成为可执行的二进制文件，但是不运行测试。

-i : 安装测试包依赖的package，但是不运行测试。

关于build flags，调用go help build，这些是编译运行过程中需要使用到的参数，一般设置为空

关于packages，调用go help packages，这些是关于包的管理，一般设置为空

关于flags for test binary，调用go help testflag，这些是go test过程中经常使用到的参数

-test.v : 是否输出全部的单元测试用例（不管成功或者失败），默认没有加上，所以只输出失败的单元测试用例。

-test.run pattern: 只跑哪些单元测试用例

-test.bench patten: 只跑那些性能测试用例

-test.benchmem : 是否在性能测试的时候输出内存情况

-test.benchtime t : 性能测试运行的时间，默认是1s

-test.cpuprofile cpu.out : 是否输出cpu性能分析文件

-test.memprofile mem.out : 是否输出内存性能分析文件

5 Go test 命令介绍2-续

-test.blockprofile block.out : 是否输出内部goroutine阻塞的性能分析文件

-test.memprofilerate n : 内存性能分析的时候有一个分配了多少的时候才打点记录的问题。这个参数就是设置打点的内存分配间隔，也就是profile中一个sample代表的内存大小。默认是设置为512 * 1024的。如果你将它设置为1，则每分配一个内存块就会在profile中有个打点，那么生成的profile的sample就会非常多。如果你设置为0，那就是不做打点了。

你可以通过设置memprofilerate=1和GOGC=off来关闭内存回收，并且对每个内存块的分配进行观察。

-test.blockprofilerate n: 基本同上，控制的是goroutine阻塞时候打点的纳秒数。默认不设置就相当于-test.blockprofilerate=1，每一纳秒都打点记录一下

-test.parallel n : 性能测试的程序并行cpu数，默认等于GOMAXPROCS。

-test.timeout t : 如果测试用例运行时间超过t，则抛出panic

-test.cpu 1,2,4 : 程序运行在哪些CPU上面，使用二进制的1所在位代表，和nginx的nginx_worker_cpu_affinity是一个道理

-test.short : 将那些运行时间较长的测试用例运行时间缩短