go语言笔记

1.变量初始化：

1. var [name] [type]

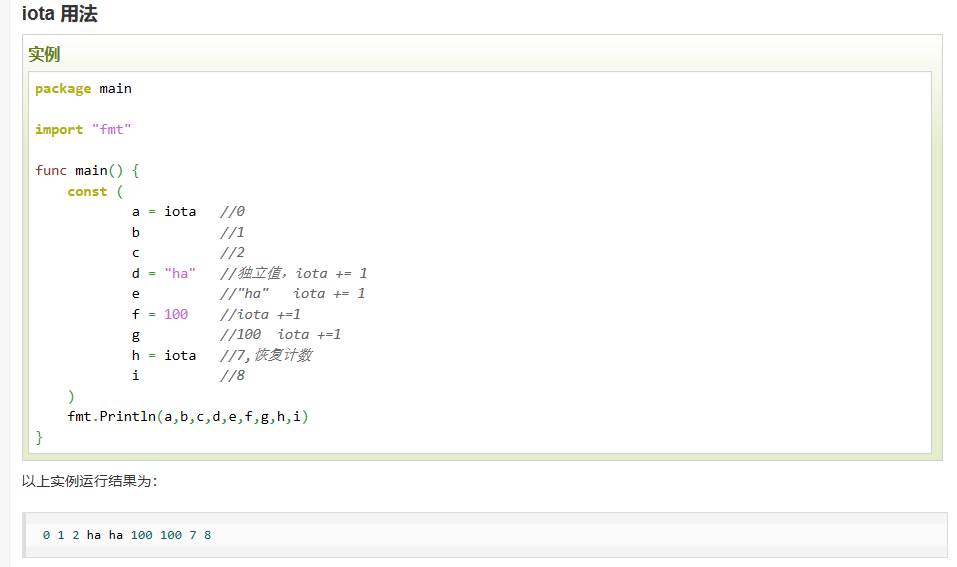
2. var [name] = [数值] 隐式判断变量类型

3. [name] := [数值]

2.iota

iota，特殊常量，可以认为是一个可以被编译器修改的常量。

iota 在 const关键字出现时将被重置为 0(const 内部的第一行之前)，const 中每新增一行常量声明将使 iota 计数一次(iota 可理解为 const 语句块中的行索引)。



iota 可以被用作枚举值：

3.在定义常量组时，如果不提供初始值，则表示将使用上行的表达式。

4.for循环：



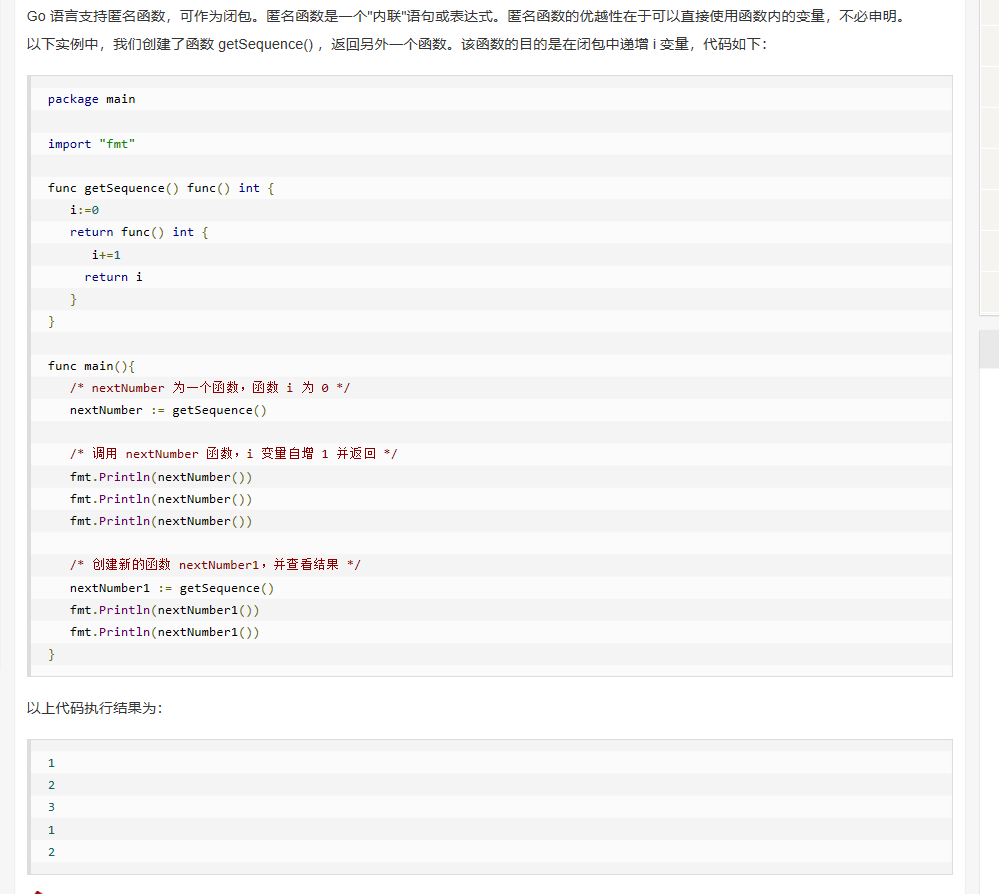
5.go语言函数返回多个值



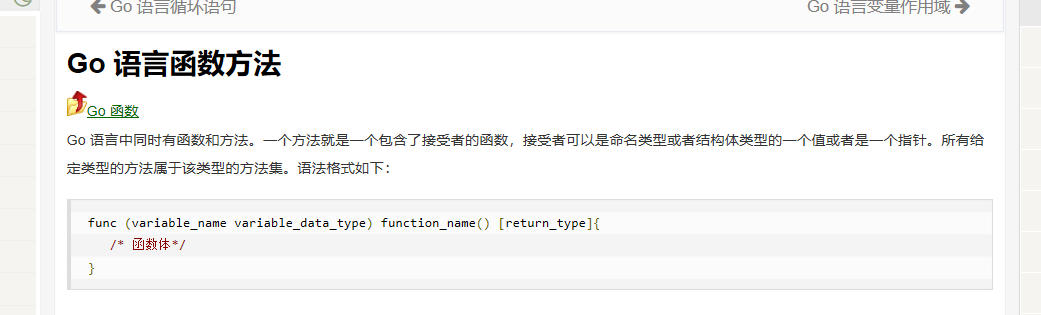
6.go语言函数作为实参



7.函数闭包



8.针对特定类型才能使用的函数（方法）（variable\_name 叫做方法的接收器(receiver)，这里的建议是可以使用其类型的第一个字母，比如这里使用了Point的首字母p。）

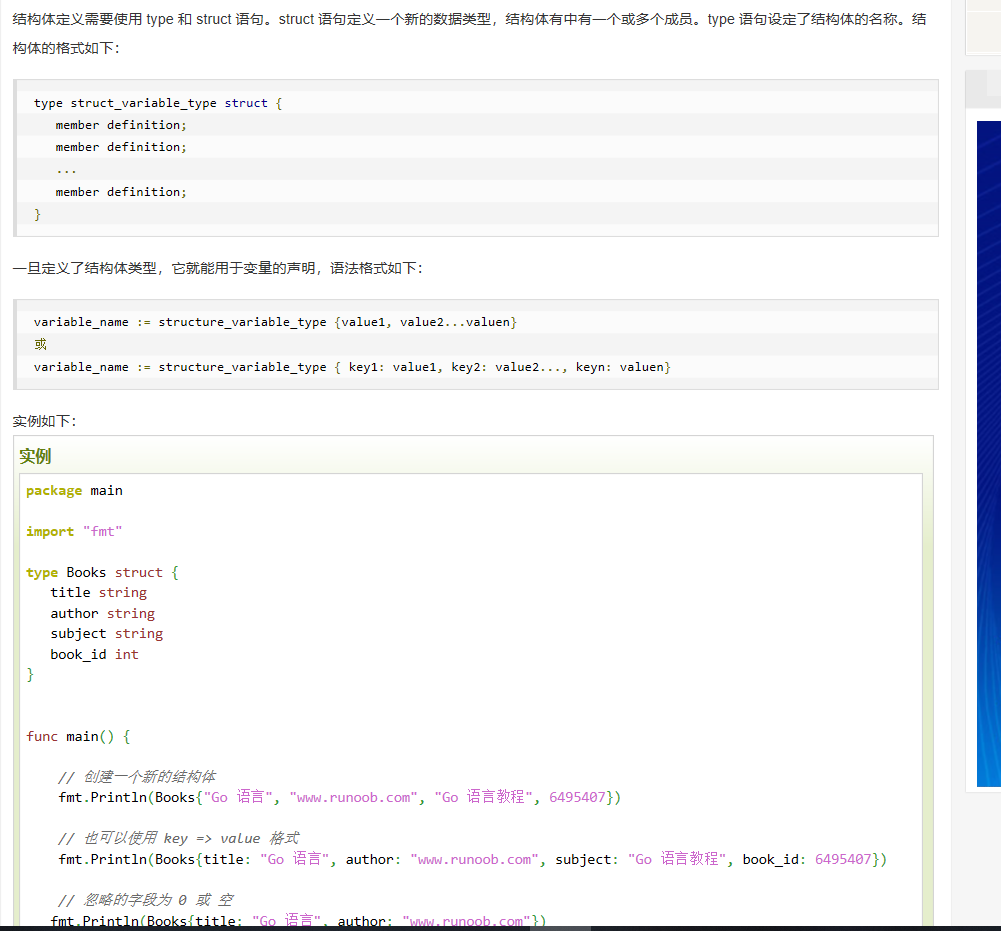


9.向函数传递数组时可以设定数组大小也可以不设定



10.空指针：nil

11.结构体

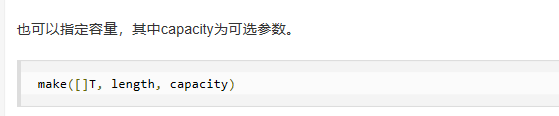


12.结构体指针访问成员

使用.操作符

13.切片

13.1使用make（）创建切片（好处是可以设定capacity大小，也可以不设定）



使用len（）可以获得其长度

使用cap（）可以获得其容量

切片不是复制，被切内容改变，原数组也会改变

修改切片的副本时，原切片内容也会变，但是原切片的len，和cap属性不会发生改变

13.2同时向切片添加多个元素append



13.3拷贝切片copy



14.键值循环（for range）





15.map

15.1 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。

Map 是一种集合，所以我们可以像迭代数组和切片那样迭代它。不过，Map 是无序的，我们无法决定它的返回顺序，这是因为 Map 是使用 hash 表来实现的。

如果不初始化 map，那么就会创建一个 nil map。nil map 不能用来存放键值对

### 15.2定义 Map

可以使用内建函数 make 也可以使用 map 关键字来定义 Map:



15.3对map用key值取结果返回两个值：第一个value第二个是是否存在



15.4 Delete删除map中元素



16.go语言类型强制转换：type\_name(expression)， type\_name 为类型，expression 为表达式。

17.goroutine轻量级线程

Go 允许使用 go 语句开启一个新的运行期线程， 即 goroutine，以一个不同的、新创建的 goroutine 来执行一个函数。 同一个程序中的所有 goroutine 共享同一个地址空间。

使用方法：go 函数名( 参数列表 )

18通道channel

18.1通道（channel）是用来传递数据的一个数据结构。

通道可用于两个 goroutine 之间通过传递一个指定类型的值来同步运行和通讯。操作符 <- 用于指定通道的方向，发送或接收。如果未指定方向，则为双向通道。数据接收后就会消失。

ch <- v // 把 v 发送到通道 ch

v := <-ch // 从 ch 接收数据

// 并把值赋给 v

v, ok := <-ch 获取ch值可以获取是否存在

18.2声明一个通道很简单，我们使用chan关键字即可，通道在使用前必须先创建。

ch := make(chan int)

默认情况下，通道是不带缓冲区的。发送端发送数据，同时必须接收端相应的又接收数据。

通道可以设置缓冲区，通过 make 的第二个参数指定缓冲区大小：

ch := make(chan int, 100)

18.3 通道遍历：Go 通过 range 关键字来实现遍历读取到的数据，类似于与数组或切片。

18.4 for range遍历channel：

for range 可以遍历通道（channel），但是通道在遍历时，只输出一个值，即管道内的类型对应的数据。且遍历后channel内数据就会消失。

18.5 channel装完数据后就用close(ch)关闭，以便range遍历的时候不出错。

19.go语言随机数

使用："math/rand","time"包，和rand.Seed(time.Now().Unix())函数，每个程序中只可使用一次seed函数。使用rand.Float64或者rand.Intn(n int)去产生随机数。

20.csv读写

一、写csv文件

函数：

func NewWriter(w io.Writer) \*Writer

func (w \*Writer) Flush() //用以保证缓冲区的数据都写入文件

func (w \*Writer) Write(record []string) (err os.Error)

func (w \*Writer) WriteAll(records [][]string) (err os.Error)

二、读csv文件

函数：

func NewReader(r io.Reader) \*Reader

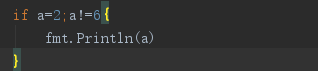
func (r \*Reader) Read() (record []string, err os.Error)

func (r \*Reader) ReadAll() (records [][]string, err os.Error)

读csv文件时如果不是每行列数相等，使用Read()就会返回err但是依旧可以读出,使用ReadAll()就会什么也读不到

21.defer

22. “;”分号的作用：可以写在if函数后面这样就会预先执行“;”前面的内容，然后“;”后的内容作为if判断的条件，如下：



23.获得变量类型：

reflect.TypeOf（）

24. go模块中要导出的函数，必须首字母大写。

25. go语言中 … 的用法：传递可变数目参数

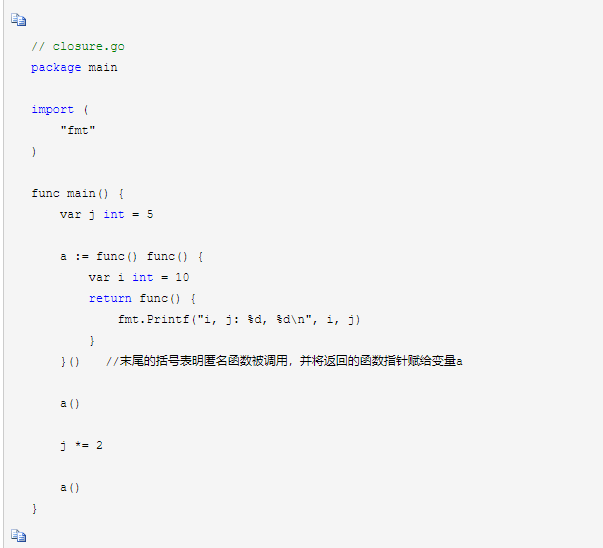


26. go语言结构体组合+匿名字段（类似于继承）：父结构体类型作为子结构体的属性

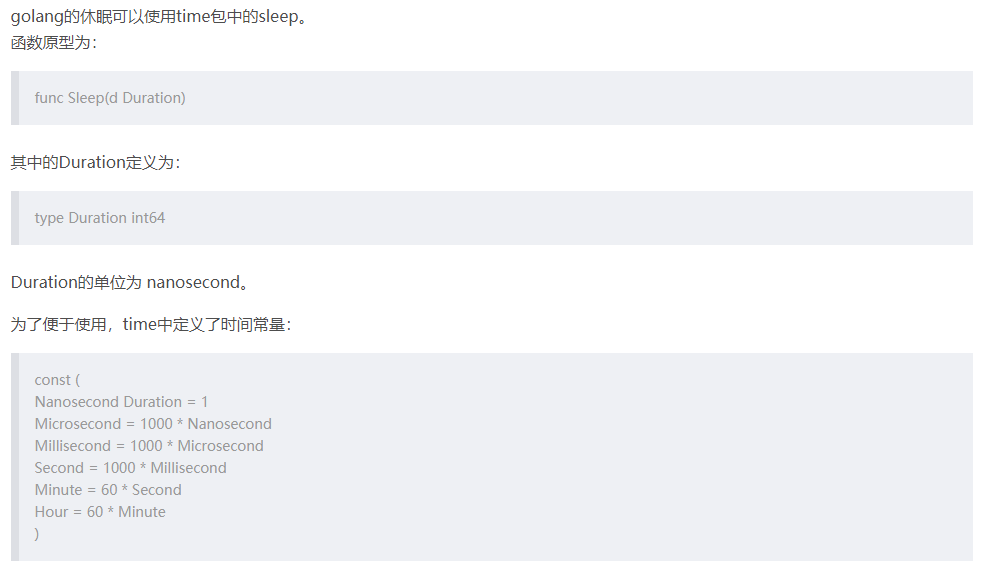




27.调用匿名函数



28.go sleep与time：





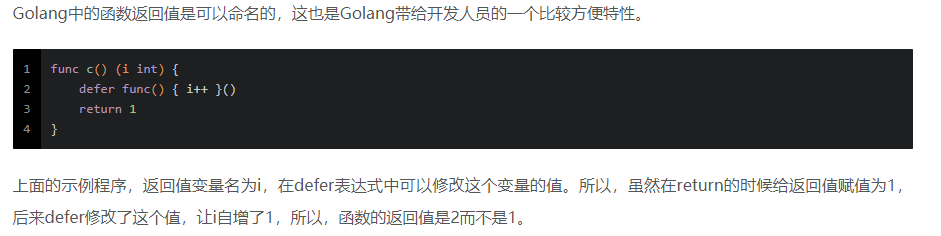
29.匿名函数的作用域是外层函数的作用域

30. defer：defer 语句中的函数调用是当前函数最后执行的东西，即便函数抛出了异常，也会被执行的。 这样就不会因程序出现了错误，而导致资源不会释放了。

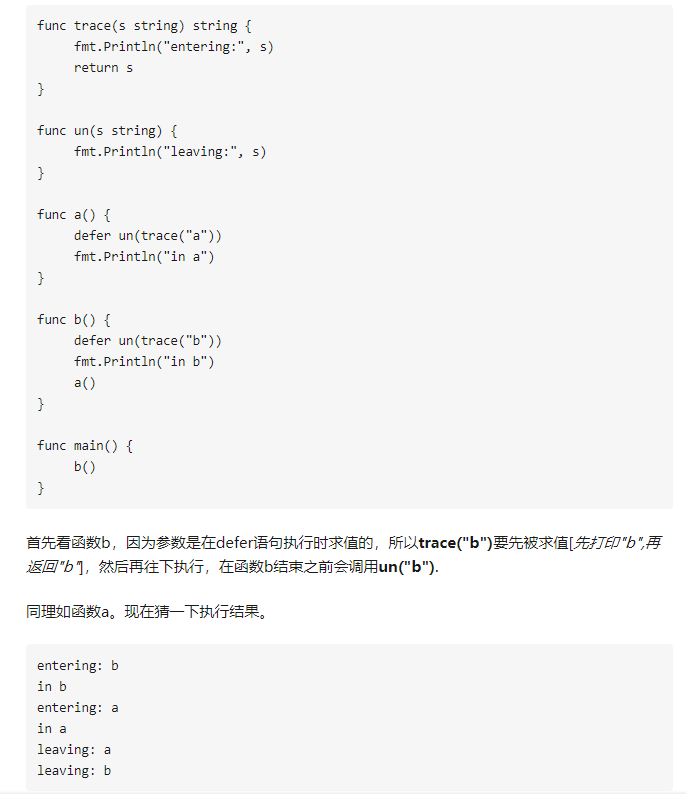
defer表达式中变量的值在defer表达式被定义时就已经明确

defer表达式的调用顺序是按照先进后出的方式

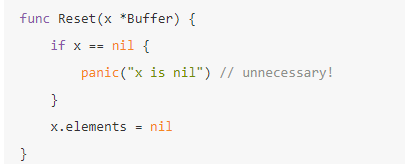
defer表达式中可以修改函数中的命名返回值



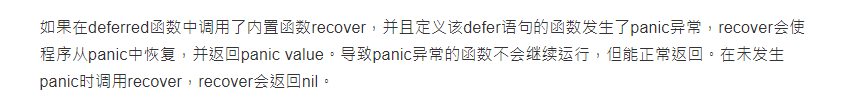
参数在defer语句执行求值，而不是在函数调用执行时求值



31.go panic



32.go recover：

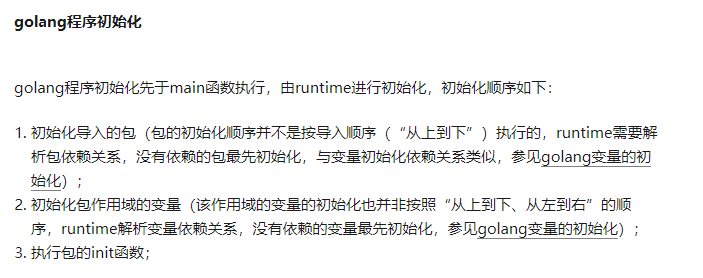


33.类型断言 .(int): <https://www.cnblogs.com/zrtqsk/p/4157350.html>

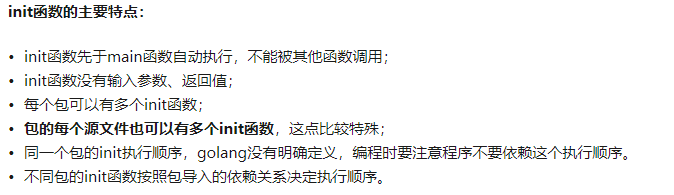
34. 入参中，如果连续的参数类型一致，则可以省略连续多个参数的类型，只保留最后一个类型声明。如 func IsEqual(a, b float64) bool 这个方法就只保留了一个类型声明,此时入参a和b均是float64数据类型。

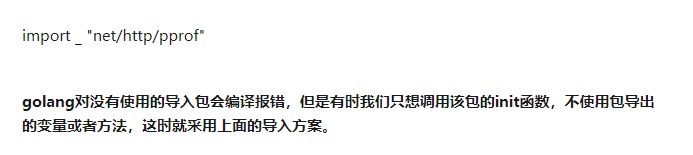
35.go中函数及其变量以大写开头则可以在别的包中引用，否则则不能

36.go语言初始化：

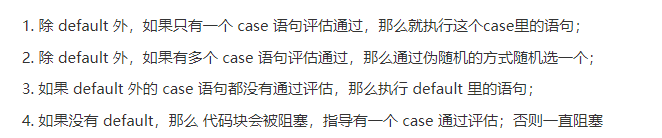


37.init函数特点：

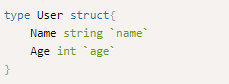


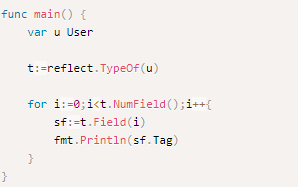


38.go select：



39.go获得结构体tag信息：

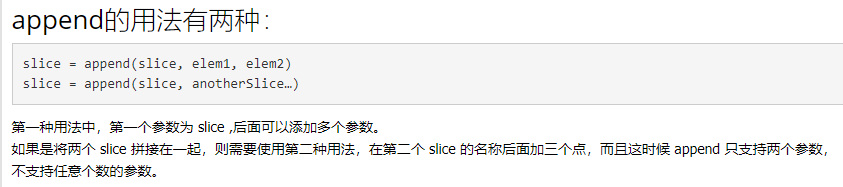




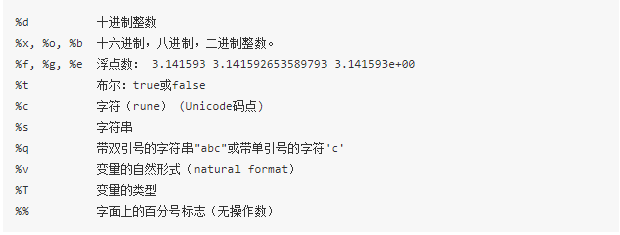
40.指针声明后复制出错的问题



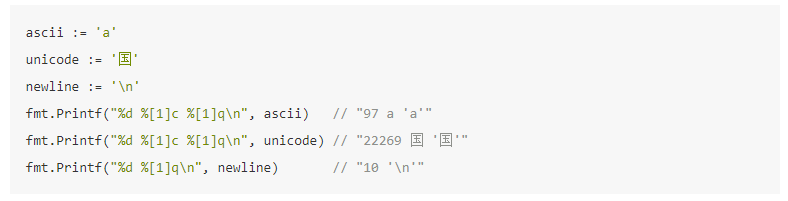
41.append



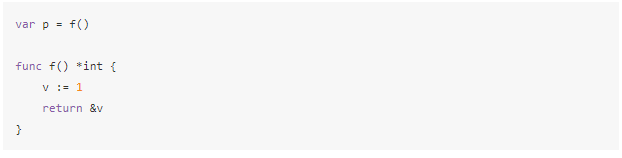
1. printf函数的转换字符：



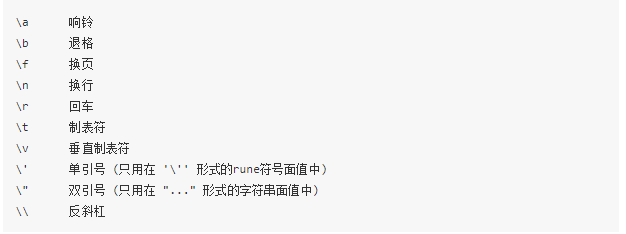
可以通过[]重复使用参数：



1. 在Go语言中，返回函数中局部变量的地址也是安全的



1. 编译器会自动选择在栈上还是在堆上分配局部变量的存储空间，但可能令人惊讶的是，这个选择并不是由用var还是new声明变量的方式决定的。
2. 类型转换：对于每一个类型T，都有一个对应的类型转换操作T(x)，用于将x转为T类型（译注：如果T是指针类型，可能会需要用小括弧包装T，比如(\*int)(0)）。只有当两个类型的底层基础类型相同时，才允许这种转型操作，或者是两者都是指向相同底层结构的指针类型，这些转换只改变类型而不会影响值本身。
3. Go语言将数据类型分为四类：基础类型（数字、字符串和布尔型）、复合类型（数组、结构体）、引用类型（指针、字典、切片、通道、函数）和接口类型。
4. %取模运算符的符号和被取模数的符号总是一致的，除法运算符/的行为则依赖于操作数是否为全为整数，比如5.0/4.0的结果是1.25，但是5/4的结果是1，因为整数除法会向着0方向截断余数。
5. 一个float32类型的浮点数可以提供大约6个十进制数的精度，而float64则可以提供约15个十进制数的精度；通常应该优先使用float64类型，因为float32类型的累计计算误差很容易扩散
6. Inf：正无穷，-Inf：负无穷，NaN：非数
7. 字符串的值是不可变的：一个字符串包含的字节序列永远不会被改变。因为Go语言源文件总是用UTF8编码，并且Go语言的文本字符串也以UTF8编码的方式处理，因此我们可以将Unicode码点也写到字符串面值中。
8. go语言\转义序列：



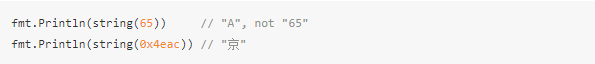
1. 原生字符串面值形式是`...`（由反斜杠包裹起来的字符串）



1. UTF-8：



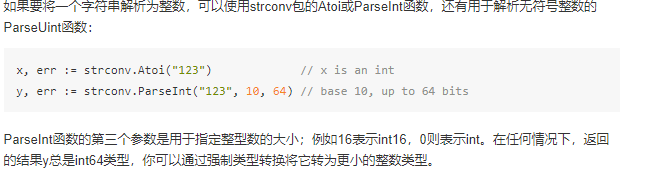
1. 将一个整数转型为字符串意思是生成以只包含对应Unicode码点字符的UTF8字符串：



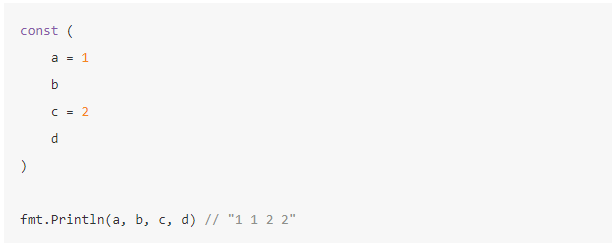
1. 整数转化为字符串（a代表ASCII）：



字符串转化为整数：



1. 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，对应的常量类型也一样的。例如：



1. 在数组字面值中，如果在数组的长度位置出现的是“...”省略号，则表示数组的长度是根据初始化值的个数来计算。如：

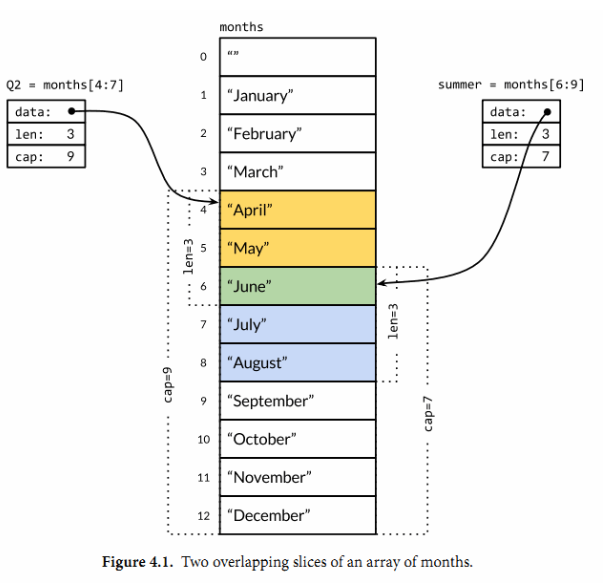
q := [...]int{1, 2, 3}

1. 数组可以以指定一个索引和对应值列表的方式初始化，就像下面这样：（这种形式的数组字面值形式中，初始化索引的顺序是无关紧要的，而且没用到的索引可以省略，和前面提到的规则一样，未指定初始值的元素将用零值初始化。）



1. 如果一个数组的元素类型是可以相互比较的，那么数组类型也是可以相互比较的，这时候我们可以直接通过==比较运算符来比较两个数组，只有当两个数组的所有元素都是相等的时候数组才是相等的。不相等比较运算符!=遵循同样的规则。
2. slice（切片）

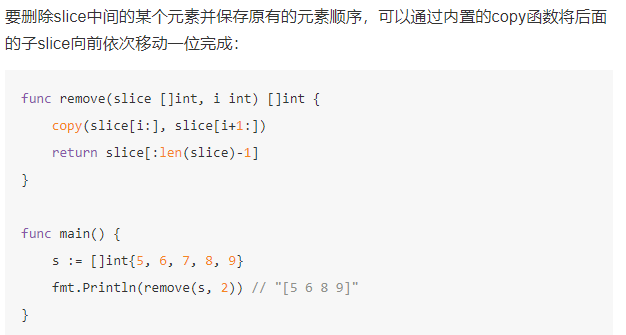
一个slice由三个部分构成：指针、长度和容量。指针指向第一个slice元素对应的底层数组元素的地址，要注意的是slice的第一个元素并不一定就是数组的第一个元素。多个slice之间可以共享底层的数据，并且引用的数组部分区间可能重叠。



和数组不同的是，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。不过标准库提供了高度优化的bytes.Equal函数来判断两个字节型slice是否相等（[]byte），但是对于其他类型的slice，我们必须自己展开每个元素进行比较。

\*除了文档已经明确说明的地方，所有的Go语言函数应该以相同的方式对待nil值的slice和0长度的slice。除了和nil相等比较外，一个nil值的slice的行为和其它任意0长度的slice一样；例如reverse(nil)也是安全的。

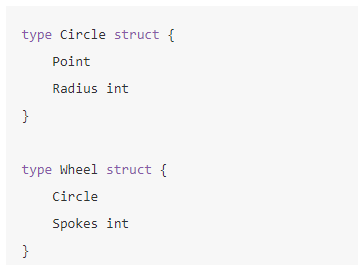
删除slice中一个元素：

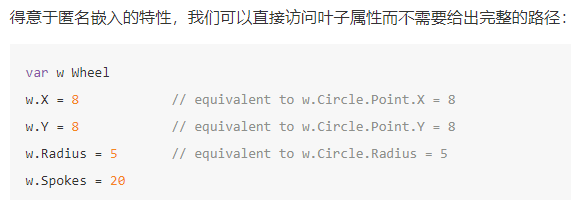


1. map的key必须是可比较的类型，x += y和x++等简短赋值语法也可以用在map上。



1. 如果结构体的全部成员都是可以比较的，那么结构体也是可以比较的，可比较的结构体类型和其他可比较的类型一样，可以用于map的key类型。
2. 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**。匿名成员的数据类型必须是命名的类型或指向一个命名的类型的指针。下面的代码中，Circle和Wheel各自都有一个匿名成员。我们可以说Point类型被嵌入到了Circle结构体，同时Circle类型被嵌入到了Wheel结构体。





不幸的是，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：





因为匿名成员也有一个隐式的名字，因此不能同时包含两个类型相同的匿名成员，这会导致名字冲突。同时，因为成员的名字是由其类型隐式地决定的，所有匿名成员也有可见性的规则约束。

外层的结构体不仅仅是获得了匿名成员类型的所有成员，而且也获得了该类型导出的全部的方法。这个机制可以用于将一个有简单行为的对象组合成有复杂行为的对象。组合是Go语言中面向对象编程的核心

1. 文本和HTML模板：https://books.studygolang.com/gopl-zh/ch4/ch4-06.html
2. fmt.Errorf函数使用fmt.Sprintf格式化错误信息并返回。



1. **几种错误处理策略**：

1.使用fmt.Errorf函数添加调用者的信息后将错误返回，当错误最终由main函数处理时，错误信息应提供清晰的从原因到后果的因果链，就像美国宇航局事故调查时做的那样：



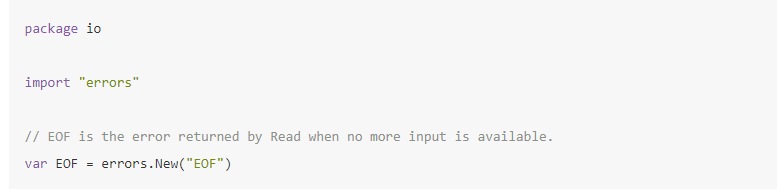
2. 如果错误的发生是偶然性的，或由不可预知的问题导致的。一个明智的选择是重新尝试失败的操作。在重试时，我们需要限制重试的时间间隔或重试的次数，防止无限制的重试。

3.如果错误发生后，程序无法继续运行，我们就可以采用第三种策略：输出错误信息并结束程序。需要注意的是，这种策略只应在main中执行。对库函数而言，应仅向上传播错误，除非该错误意味着程序内部包含不一致性，即遇到了bug，才能在库函数中结束程序。

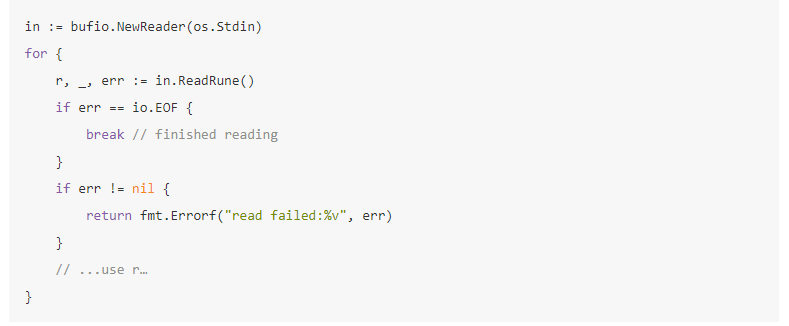
4. 有时，我们只需要输出错误信息就足够了，不需要中断程序的运行。我们可以通过log包提供函数

5. 我们可以直接忽略掉错误。

1. io包保证任何由文件结束引起的读取失败都返回同一个错误——io.EOF，该错误在io包中定义：



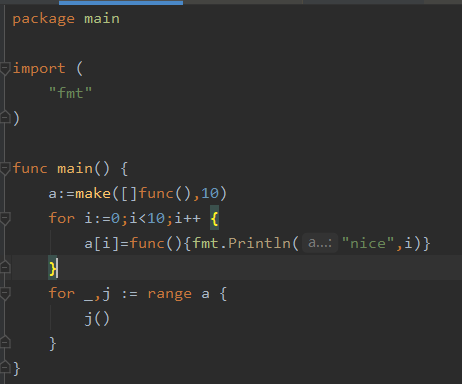
调用者只需通过简单的比较，就可以检测出这个错误。下面的例子展示了如何从标准输入中读取字符，以及判断文件结束。

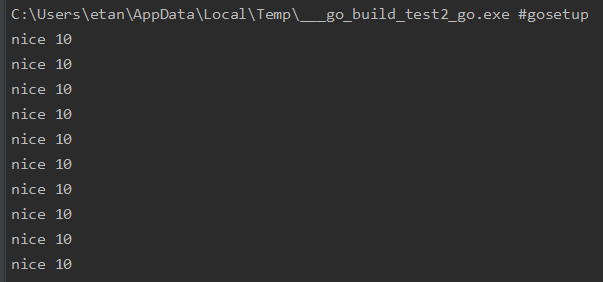


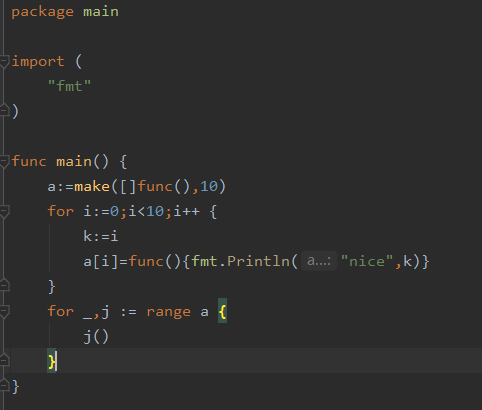
1. go语言中函数像其他值一样，拥有类型，可以被赋值给其他变量，传递给函数，从函数返回。对函数值（function value）的调用类似函数调用。
2. **使用匿名函数时可能遇到的问题（坑）**：

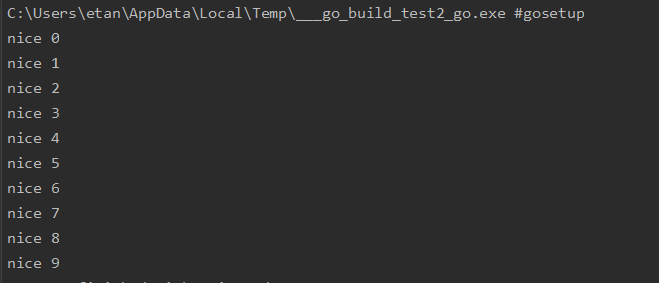


举例：

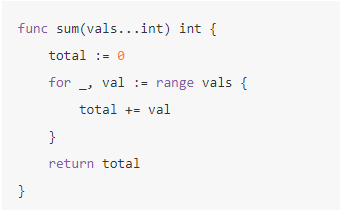




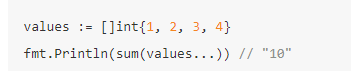




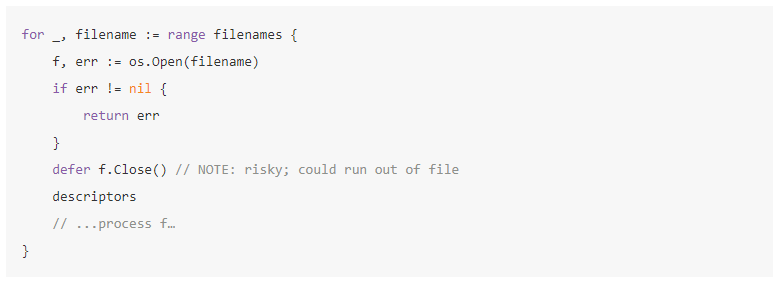
1. 在声明可变参数函数时，需要在参数列表的最后一个参数类型之前加上省略符号“...”，这表示该函数会接收任意数量的该类型参数。sum函数返回任意个int型参数的和。在函数体中,vals被看作是类型为**[] int的切片**。sum可以接收任意数量的int型参数。（可变参数只能放在参数列表的末尾）



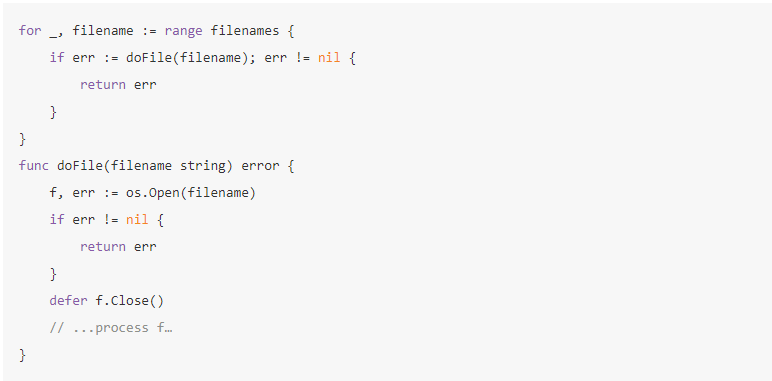
在上面的代码中，调用者隐式的创建一个数组，并将原始参数复制到数组中，再把数组的一个切片作为参数传给被调函数。如果原始参数已经是切片类型，我们该如何传递给sum？只需在最后一个参数后加上省略符。下面的代码功能与上个例子中最后一条语句相同。



1. defer的使用：当defer语句被执行时，跟在defer后面的函数会被延迟执行。直到包含该defer语句的函数执行完毕时，defer后的函数才会被执行，不论包含defer语句的函数是通过return正常结束，还是由于panic导致的异常结束。你可以在一个函数中执行多条defer语句，它们的执行顺序与声明顺序相反。
2. 在循环体中的defer语句需要特别注意，因为只有在函数执行完毕后，这些被延迟的函数才会执行。下面的代码会导致系统的文件描述符耗尽，因为在所有文件都被处理之前，没有文件会被关闭。



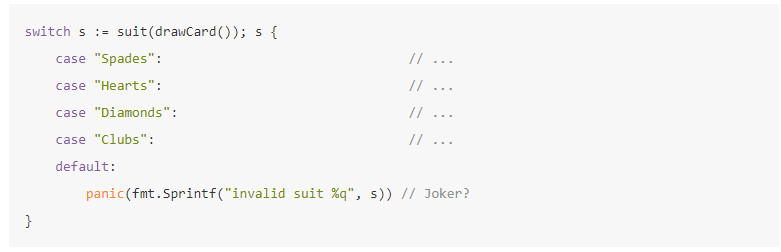
一种解决方法是将循环体中的defer语句移至另外一个函数。在每次循环时，调用这个函数。



defer使用需要注意的一点：

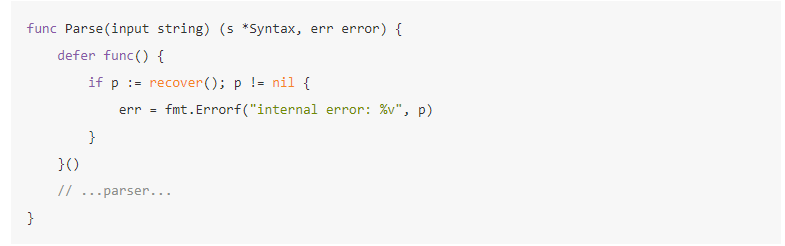


1. 当panic异常发生时，程序会中断运行，并立即执行在该goroutine中被延迟的函数（defer 机制）（延迟函数的调用在释放堆栈信息之前）。随后，程序崩溃并输出日志信息。直接调用内置的panic函数也会引发panic异常；panic函数接受任何值作为参数。当某些不应该发生的场景发生时，我们就应该调用panic。比如，当程序到达了某条逻辑上不可能到达的路径：



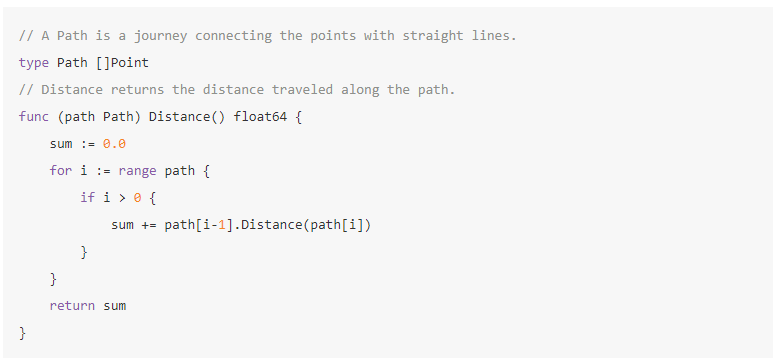
1. 通常来说，不应该对panic异常做任何处理，但有时，也许我们可以从异常中恢复，至少我们可以在程序崩溃前，做一些操作。举个例子，当web服务器遇到不可预料的严重问题时，在崩溃前应该将所有的连接关闭；如果不做任何处理，会使得客户端一直处于等待状态。如果web服务器还在开发阶段，服务器甚至可以将异常信息反馈到客户端，帮助调试。

如果在deferred函数中调用了内置函数recover，并且定义该defer语句的函数发生了panic异常，recover会使程序从panic中恢复，并返回panic value。导致panic异常的函数不会继续运行，但能正常返回。在未发生panic时调用recover，recover会返回nil。



不应该试图去恢复其他包引起的panic。公有的API应该将函数的运行失败作为error返回，而不是panic。同样的，你也不应该恢复一个由他人开发的函数引起的panic，比如说调用者传入的回调函数，因为你无法确保这样做是安全的。但有时我们很难完全遵循规范，基于以上原因，安全的做法是有选择性的recover。换句话说，只恢复应该被恢复的panic异常，此外，这些异常所占的比例应该尽可能的低。

1. 我们可以给同一个包内的任意命名类型定义方法，只要这个**命名类型**(这个例子里，底层类型是指[]Point这个slice，Path就是命名类型,用type去命名)的底层类型不是指针或者interface。



1. 为了避免歧义，在声明方法时，如果一个类型名本身是一个指针的话，是不允许其出现在接收器中的，比如下面这个例子：



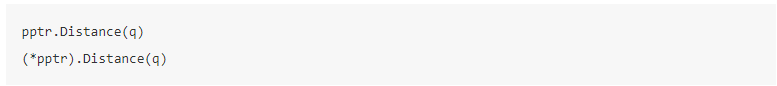
1. 如果接收器p是一个Point类型的变量，并且其方法需要一个Point指针作为接收器，我们可以用下面这种简短的写法：



编译器会隐式地帮我们用&p去调用ScaleBy这个方法。这种简写方法只适用于“变量”，包括struct里的字段比如p.X，以及array和slice内的元素比如perim[0]。我们不能通过一个无法取到地址的接收器来调用指针方法，比如临时变量的内存地址就无法获取得到：



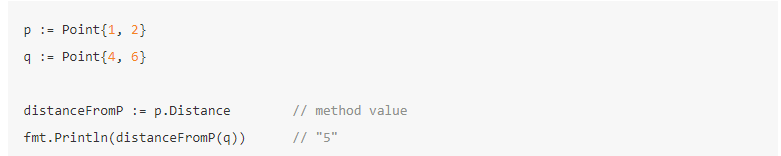
我们可以用一个\*Point这样的接收器来调用Point的方法，因为我们可以通过地址来找到这个变量，只要用解引用符号\*来取到该变量即可。编译器在这里也会给我们隐式地插入\*这个操作符，所以下面这两种写法等价的：



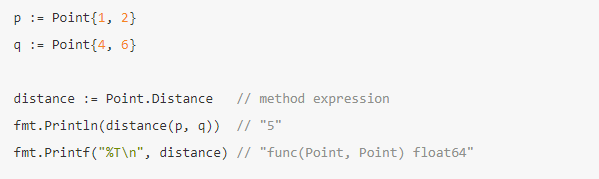
1. 不管你的method的receiver是指针类型还是非指针类型，都是可以通过指针/非指针类型进行调用的，编译器会帮你做类型转换。
2. 在声明一个method的receiver该是指针还是非指针类型时，你需要考虑两方面的内部，第一方面是这个对象本身是不是特别大，如果声明为非指针变量时，调用会产生一次拷贝（**这次拷贝是浅复制。注意，对于浅复制，如果一个结构体中包含一个例如slice的引用变量成员，在方法中修改slice内容时，原来变量中的slice成员也会被改变。但如果原来slice变量是nil的时候则不会，因为传递的浅复制也是个nil，不会影响原来变量**）；第二方面是如果你用指针类型作为receiver，那么你一定要注意，这种指针类型指向的始终是一块内存地址，就算你对其进行了拷贝。
3. \*如果一个结构体中包含一个匿名对象，那么可以直接将这个结构体类型的变量作为接收器去调用其包含的匿名对象的方法，调用的方法会一层层的向下搜索，如果同一层级有两个同名方法则会冲突。

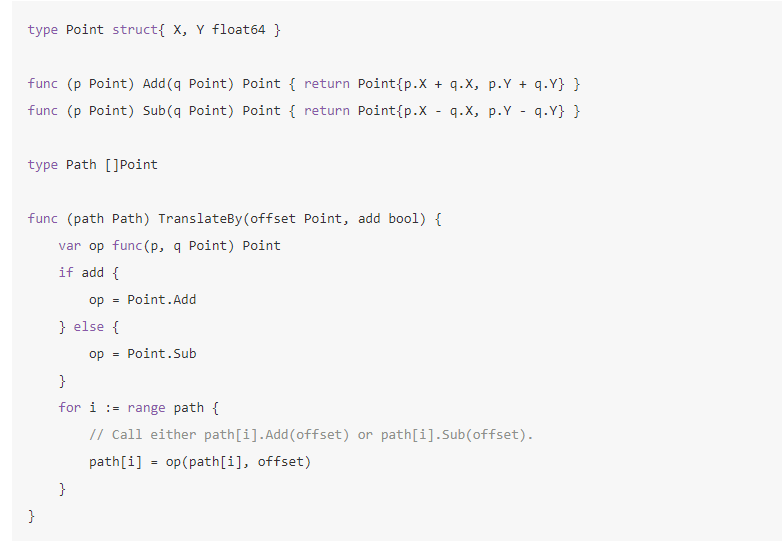




实际上可以将p.Distance()分成两步来执行。p.Distance叫作“**选择器**”，选择器会返回一个方法"值"——一个将方法(Point.Distance)绑定到特定接收器变量的函数。这个函数可以不通过指定其接收器即可被调用；即调用时不需要指定接收器，只要传入函数的参数即可：

1. **方法表达式**：当调用一个方法时，与调用一个普通的函数相比，我们必须要用选择器(p.Distance)语法来指定方法的接收器。当T是一个类型时，方法表达式可能会写作T.f或者(\*T).f，会返回一个函数"值"，这种函数会**将其第一个参数用作接收器**，所以可以用通常(不写选择器)的方式来对其进行调用：





1. “^”符号作为一元运算符是按位求反，作为二元运算符是异或
2. **匿名结构体**：

struct{

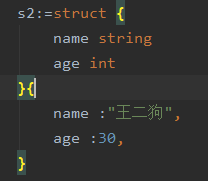
字段定义

…

}{

赋值

}

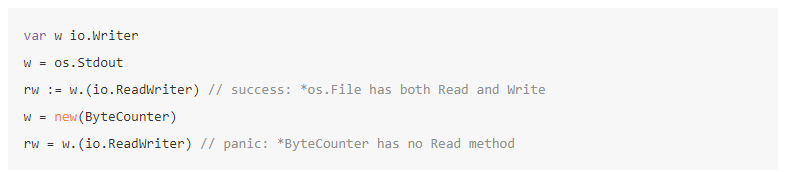


1. 判断系统是32还是64位的智能表达式：32 << (^uint(0) >> 63)
2. 在命名一个getter方法时，我们通常会省略掉前面的Get前缀。这种简洁上的偏好也可以推广到各种类型的前缀比如Fetch，Find或者Lookup。
3. 在现实的程序里，一般会约定如果Point这个类有一个指针作为接收器的方法，那么所有Point的方法都必须有一个指针接收器，即使是那些并不需要这个指针接收器的函数。
4. 接口类型封装和隐藏具体类型和它的值。即使具体类型有其它的方法也只有接口类型暴露出来的方法会被调用到
5. 空接口类型：interface{}，因为空接口类型对实现它的类型没有要求，所以我们可以将任意一个值赋给空接口类型。对于创建的一个interface{}值持有一个boolean，float，string，map，pointer，或者任意其它的类型；我们当然不能直接对它持有的值做操作，因为interface{}没有任何方法。我们会在7.10章中学到一种用类型断言来获取interface{}中值的方法。
6. 接口值，由两个部分组成，一个具体的类型和那个类型的值。它们被称为接口的动态类型和动态值。
7. **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。因为接口值是可比较的，所以它们可以用在map的键或者作为switch语句的操作数。
8. 接口类型断言：x.(T) ，这里有两种可能。

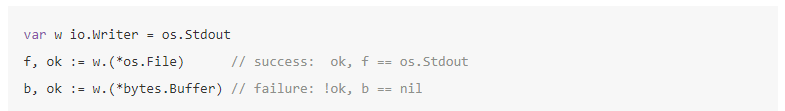
第一种，如果断言的类型T是一个具体类型，然后类型断言检查x的动态类型是否和T相同。如果这个检查成功了，类型断言的结果是x的动态值，当然它的类型是T。换句话说，具体类型的类型断言从它的操作对象中获得具体的值。如果检查失败，接下来这个操作会抛出panic。

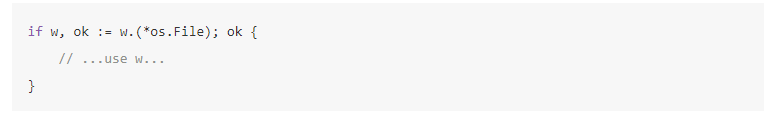


第二种：断言的类型T是一个接口类型



1. 如果断言操作的对象是一个nil接口值，那么不论被断言的类型是什么这个类型断言都会失败。
2. 这个操作不会在失败的时候发生panic但是代替地返回一个额外的第二个结果，这个结果是一个标识成功的布尔值：





1. 特殊的断言switch（类型开关）：这里我们已经将新的变量也命名为x；和类型断言一样，重用变量名是很常见的。和一个switch语句相似地，一个类型开关隐式的创建了一个语言块，因此新变量x的定义不会和外面块中的x变量冲突。每一个case也会隐式的创建一个单独的语言块。



在这个版本的函数中，在每个单一类型的case内部，变量x和这个case的类型相同。例如，变量x在bool的case中是bool类型和string的case中是string类型。在所有其它的情况中，变量x是switch运算对象的类型（接口）；在这个例子中运算对象是一个interface{}。当多个case需要相同的操作时，比如int和uint的情况，类型开关可以很容易的合并这些情况。

1. **主函数返回时，所有的goroutine都会被直接打断，程序退出**。除了从主函数退出或者直接终止程序之外，没有其它的编程方法能够让一个goroutine来打断另一个的执行，但是之后可以看到一种方式来实现这个目的，通过goroutine之间的通信来让一个goroutine请求其它的goroutine，并让被请求的goroutine自行结束执行。
2. 两个相同类型的channel可以使用==运算符比较。如果两个channel引用的是相同的对象，那么比较的结果为真。一个channel也可以和nil进行比较。
3. 不带缓存的channel：

使用make(chan int)这种方式是生成的不带缓存的channel，这种channel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，反之亦然。

1. 单方向的Channel：

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。**因为关闭操作只用于断言不再向channel发送新的数据，所以只有在发送者所在的goroutine才会调用close函数，因此对一个只接收的channel调用close将是一个编译错误。**

1. 带缓存的Channel:

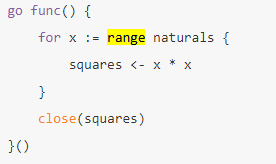
带缓存的Channel内部持有一个元素队列。队列的最大容量是在调用make函数创建channel时通过第二个参数指定的

ch = make(chan string, 3).

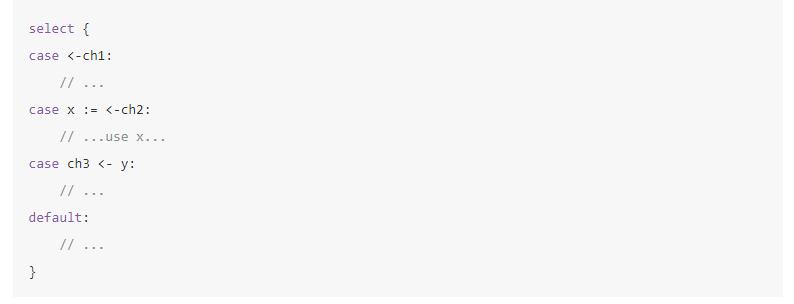
在某些特殊情况下，程序可能需要知道channel内部缓存的容量，可以用内置的cap函数获取：

cap(ch)

1. 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。在x, ok := <-naturals 这种形式中，ok为true表示成功从channels接收到值，false表示channels已经被关闭并且里面没有值可接收。
2. channel没有人接收而被永远卡住的情况叫做goroutines泄漏，和垃圾变量不同，泄漏的goroutines并不会被自动回收，因此确保每个不再需要的goroutine能正常退出是重要的。
3. channel中的range循环：它依次从channel接收数据，当channel被关闭并且没有值可接收时跳出循环。



1. select语句：select会等待case中有能够执行的case时去执行。当条件满足时，select才会去通信并执行case之后的语句；这时候其它通信是不会执行的。一个没有任何case的select语句写作select{}，会永远地等待下去。如果多个case同时就绪时，select会随机地选择一个执行，这样来保证每一个channel都有平等的被select的机会。



nil的channel有时候也是有一些用处的。因为对一个nil的channel发送和接收操作会永远阻塞，在select语句中操作nil的channel永远都不会被select到。这使得我们可以用nil来激活或者禁用case，来达成处理其它输入或输出事件时超时和取消的逻辑。

1. 数据竞争：在两个以上的goroutine并发访问相同的变量且至**少其中一个为写**操作时发生。

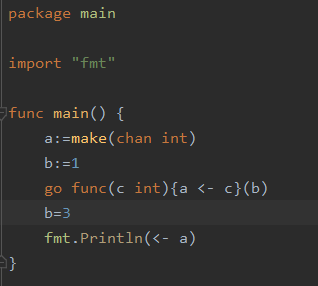
避免数据竞争的三种方法：

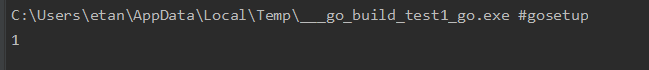
1. 不要去写变量

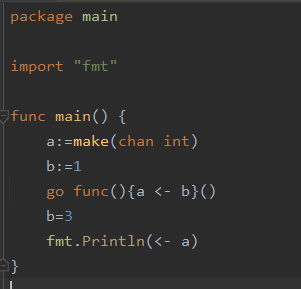
2. 避免从多个goroutine访问变量 （不要使用共享数据来通信；使用通信来共享数据）

3.允许很多goroutine去访问变量，用“互斥”的办法

1. 通道和匿名函数使用的问题（坑）：









1. mutex互斥锁

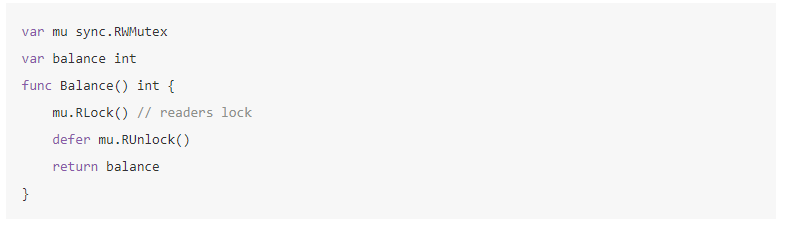
惯例来说，被mutex所保护的变量是在mutex变量声明之后立刻声明的。



1. go没有重入锁的解决方案



1. RWMutex读写锁：



使用mu.Lock和mu.Unlock方法来获取和释放一个写或互斥锁，使用mu.Rlock和mu.Runlock方法来获取和释放一个读互斥锁。

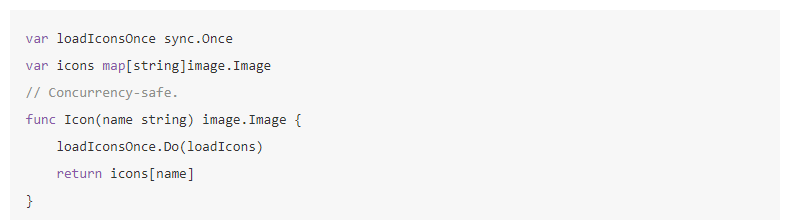
一般来说，我们不应该假设逻辑上的只读函数/方法也不会去更新某一些变量。比如一个方法功能是访问一个变量，但它也有可能会同时去给一个内部的计数器+1。如果不清楚就用mutex。

RWMutex需要更复杂的内部记录，所以会让它比一般的无竞争锁的mutex慢一些。

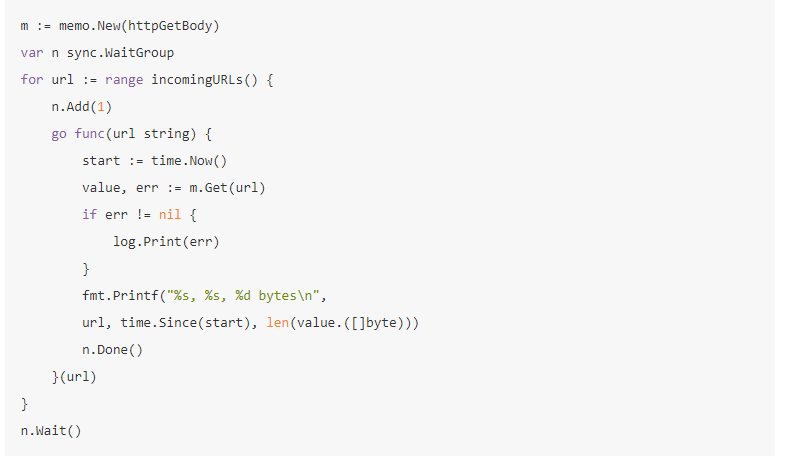
1. 可能的话，将变量限定在goroutine内部；如果是多个goroutine都需要访问的变量，使用互斥条件来访问
2. sync.Once:一次性初始化的解决方案。如果初始化成本比较大的话，那么将初始化延迟到需要的时候再去做就是一个比较好的选择。如果在程序启动的时候就去做这类的初始化的话会增加程序的启动时间并且因为执行的时候可能也并不需要这些变量所以实际上有一些浪费。可以使用懒初始化。

概念上来讲，一次性的初始化需要一个互斥量mutex和一个boolean变量来记录初始化是不是已经完成了；互斥量用来保护boolean变量和客户端数据结构。Do这个唯一的方法需要接收初始化函数作为其参数。

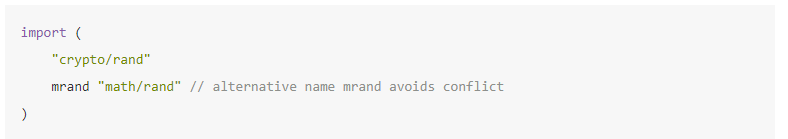




1. 竞争条件检测：只要在go build，go run或者go test命令后面加上-race的flag，就会使编译器创建一个你的应用的“修改”版或者一个附带了能够记录所有运行期对共享变量访问工具的test（详见 https://books.studygolang.com/gopl-zh/ch9/ch9-06.html）
2. sync.WaitGroup 计数goroutine，并等待：n.Add,n.Done,n.Wait



1. 可以在一个Go语言源文件包声明语句之后，其它非导入声明语句之前，包含零到多个导入包声明语句。每个导入声明可以单独指定一个导入路径，也可以通过圆括号同时导入多个导入路径。
2. 如果我们想同时导入两个有着名字相同的包，例如math/rand包和crypto/rand包，那么导入声明必须至少为一个同名包指定一个新的包名以避免冲突。这叫做导入包的重命名。导入包的重命名只影响当前的源文件。



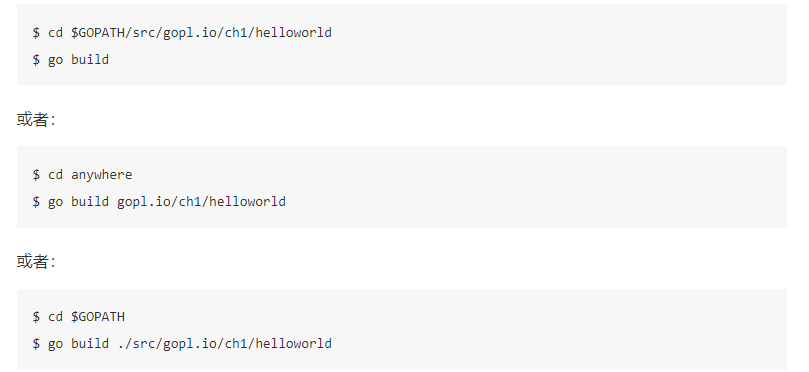
1. 在go的GOPATH模式导入非标准库包使用的路径是$GOPATH/src下的相对路径（无论对于第三方包还是自己编写的包都是如此），比如$GOPATH/src/project1/add包就用语句import “project1/add”导入。
2. 包的匿名导入：如果只是导入一个包而并不使用导入的包将会导致一个编译错误。但是有时候我们只是想利用导入包而产生的副作用：它会计算包级变量的初始化表达式和执行导入包的init初始化函数。这时候我们需要抑制“unused import”编译错误，我们可以用下划线“\_”来重命名导入的包。

import \_ “math”

1. 第一个环境变量GOPATH对应的工作区目录有三个子目录。其中src子目录用于存储源代码。其中pkg子目录用于保存编译后的包的目标文件，bin子目录用于保存编译后的可执行程序，例如helloworld可执行程序。

第二个环境变量GOROOT用来指定Go的安装目录，还有它自带的标准库包的位置。

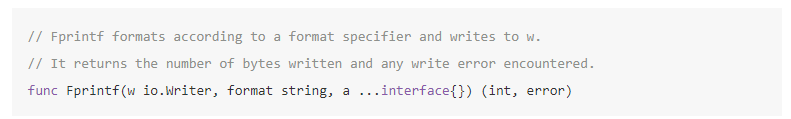
1. **go build**命令编译命令行参数指定的每个包。如果包是一个库，则忽略输出结果；这可以用于检测包是可以正确编译的。如果包的名字是main，go build将调用链接器在当前目录创建一个可执行程序；以导入路径的最后一段作为可执行程序的名字。每个包可以由它们的导入路径指定，或者用一个相对目录的路径名指定，相对路径必须以.或..开头。如果没有指定参数，那么默认指定为当前目录对应的包。



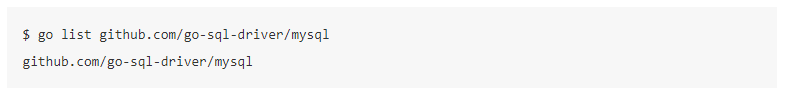
个人总结：go build的那个目录有main函数则在当前目录生成可执行文件，否则则为一个包，可以检测包可正确编译。带有main函数的文件夹中文件的包名必须为main，而此文件夹名字则可任意取。其他文件夹必须和其包名相同。

1. **go run**命令实际上是结合了构建和运行的两个步骤
2. **go install**命令和go build命令很相似，但是它会保存每个包的编译成果，而不是将它们都丢弃。被编译的包会被保存到$GOPATH/pkg目录下，目录路径和 src目录路径对应，可执行程序被保存到$GOPATH/bin目录。还有，go install命令和go build命令都不会重新编译没有发生变化的包，这可以使后续构建更快捷。
3. Go语言的编码风格鼓励为每个包提供良好的文档。包中每个导出的成员和包声明前都应该包含目的和用法说明的注释。

Go语言中的文档注释一般是完整的句子，第一行通常是摘要说明，以被注释者的名字开头。注释中函数的参数或其它的标识符并不需要额外的引号或其它标记注明。例如，下面是fmt.Fprintf的文档注释。

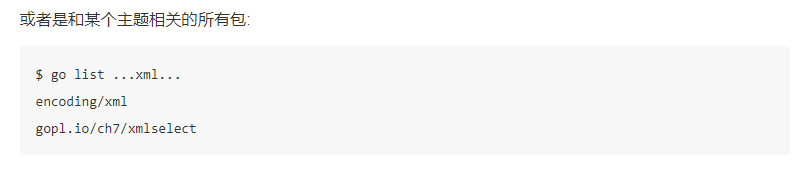


1. 如果注释后仅跟着包声明语句，那注释对应整个包的文档。包文档对应的注释只能有一个（译注：其实可以有多个，它们会组合成一个包文档注释），包注释可以出现在任何一个源文件中。如果包的注释内容比较长，一般会放到一个独立的源文件中
2. 内部包：Go语言的构建工具对包含internal名字的路径段的包导入路径做了特殊处理。这种包叫internal包，一个internal包只能被和internal目录有同一个父目录的包所导入。例如，net/http/internal/chunked内部包只能被net/http/httputil或net/http包导入，但是不能被net/url包导入。不过net/url包却可以导入net/http/httputil包。
3. **go list**命令可以查询可用包的信息。其最简单的形式，可以测试包是否在工作区并打印它的导入路径：



go list命令的参数还可以用"..."表示匹配任意的包的导入路径。我们可以用它来列出工作区中的所有包：





go list命令还可以获取每个包完整的元信息，而不仅仅只是导入路径，这些元信息可以以不同格式提供给用户。其中-json命令行参数表示用JSON格式打印每个包的元信息。

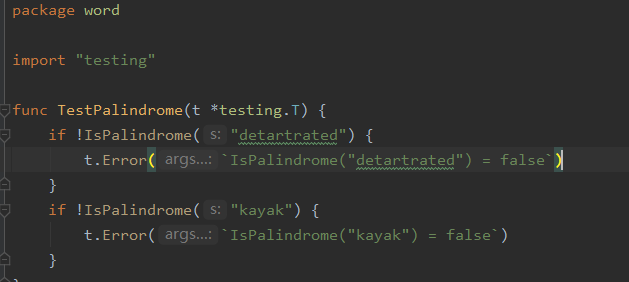


1. 在包目录内，所有以\_test.go为后缀名的源文件在执行go build时不会被构建成包的一部分，它们是go test测试的一部分。

在\*\_test.go文件中，有三种类型的函数：**测试函数、基准测试(benchmark)函数、示例函数**。一个测试函数是以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。基准测试函数是以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函数以计算一个平均的执行时间。示例函数是以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。

go test命令会遍历所有的\*\_test.go文件中符合上述命名规则的函数，生成一个临时的main包用于调用相应的测试函数，接着构建并运行、报告测试结果，最后清理测试中生成的临时文件。

1. go语言测试，test：需要引入testing包，测试函数以Test开头，需要一个\*testing.T类型的参数t。



1. 表格驱动测试：



1. 随机测试：对于一个随机的输入，我们如何能知道希望的输出结果呢？这里有两种处理策略。第一个是编写另一个对照函数，使用简单和清晰的算法，虽然效率较低但是行为和要测试的函数是一致的，然后针对相同的随机输入检查两者的输出结果。第二种是生成的随机输入的数据遵循特定的模式，这样我们就可以知道期望的输出的模式。
2. 显示测试覆盖率：

go test -cover -v -coverprofile=c.out [文件夹名]

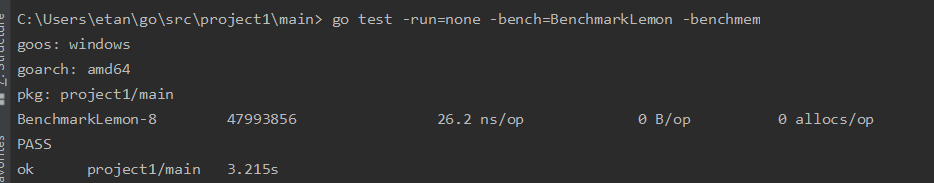
-covermode=count 是统计代码块运行次数

以html显示c.out文件：

go tool cover -html=c.out

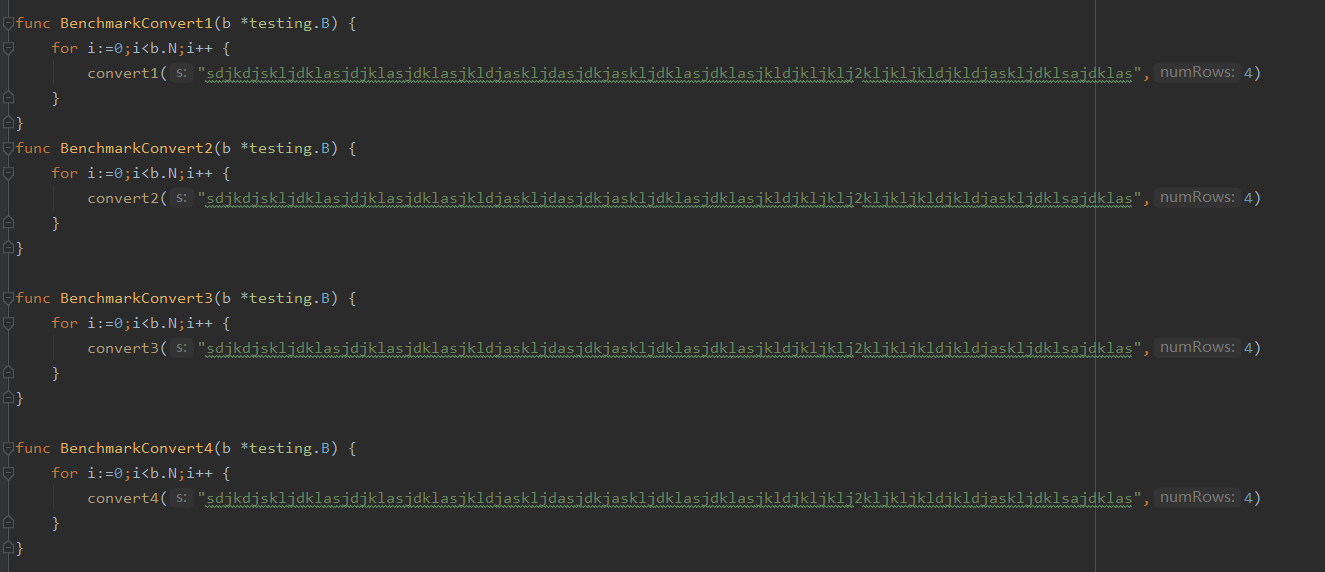
1. 基准测试：

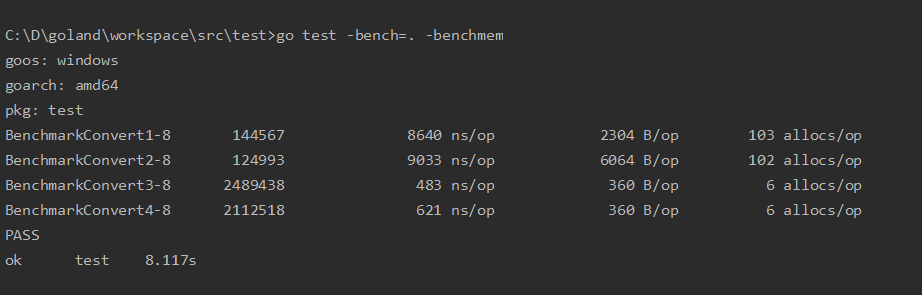
go test -run=none(保证bench之外的测试不会运行) -bench=[函数名（“.”会匹配任意函数）] -benchmem（显示内存使用状况）



-benchmem 选项。这个选项可以提供每次操作分配内存的次数，以及总共分配内存的字节数。其单位为 allocs/op，其代表每次操作从堆上分配内存的次数。单位为 B/op 的值表示每次操作分配的字节数。



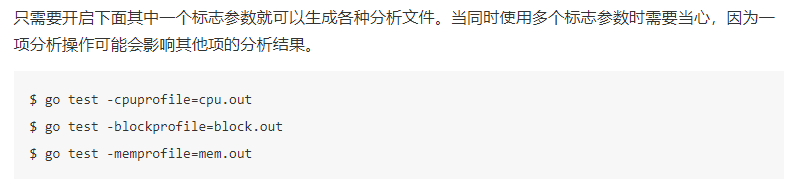




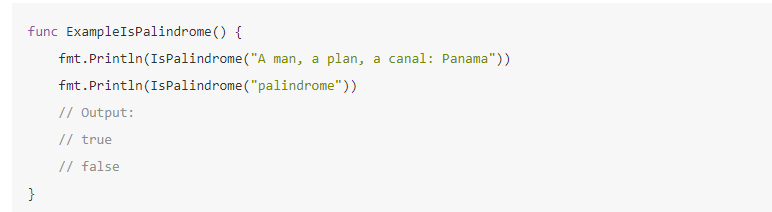
1. **CPU剖析**数据标识了最耗CPU时间的函数。在每个CPU上运行的线程在每隔几毫秒都会遇到操作系统的中断事件，每次中断时都会记录一个剖析数据然后恢复正常的运行。

**阻塞剖析**则记录阻塞goroutine最久的操作，例如系统调用、管道发送和接收，还有获取锁等。每当goroutine被这些操作阻塞时，剖析库都会记录相应的事件。

**堆剖析**则标识了最耗内存的语句。剖析库会记录调用内部内存分配的操作，平均每512KB的内存申请会触发一个剖析数据。



1. 示例函数Example开头：



示例函数有两个用处。

最主要的一个是作为文档：一个包的例子可以更简洁直观的方式来演示函数的用法

示例函数的第二个用处是，在go test执行测试的时候也会运行示例函数测试。如果示例函数内含有类似上面例子中的// Output:格式的注释，那么测试工具会执行这个示例函数，然后检查示例函数的标准输出与注释是否匹配。

1. go语言遍历字符串可以以下标（字节数组）和range两种方式遍历，其中前者每个遍历出的数据是byte类型（暂一个字节），后者遍历出的数据为rune类型（占4字节），所以后者遍历中文字符串时可以遍历出每个字。
2. Go 语言的字符串是不可修改的，可以认为它们是只读的字节切片。这就是说，每次创建字符串都要分配新的内存，可能还会给垃圾回收器造成更多工作。而[]byte是可以改变的，在构建一个不断增长的字符串时，推荐使用[]byte而非string。可以使用strings包下的strings.Builder来作为扩展字符串。strings.Builder如果不使用grow()函数提前分配内存，运行效率和[]byte切片其实相差不大。

