Go语言的特点和优势

1．简单易学

Go语言语法简单，包含了类似C语言的语法。如果读者已经掌握了两到三门编程语言，那么学习Go语言只需要几天的熟悉过程。即使一名刚入门的开发者，花几个星期也能写出来性能较高的程序。

2．自由高效

Go语言的编译速度明显优于 Java 和 C++，此外，Go语言还拥有接近C语言的运行效率和接近PHP的开发效率。换言之，Go语言等同于将开发效率和运行效率进行了完美的融合。Go语言支持当前所有的编程范式，其中包括过程式编程、面向对象编程、面向接口编程、函数式编程。开发者们可根据需求自由的组合。

3．强大的标准库

Go中的标准库非常稳定，丰富的标准库覆盖网络、系统、加密、编码、图形等各个方面，尤其是网络和系统的库非常实用。使得开发者在开发大型程序中，几乎无需依赖第三方库。

4．部署方便

Go语言不使用虚拟机，Go语言的代码可以直接输出为目标平台的二进制可执行文件。Go语言有自己的链接器，不依赖任何系统提供的编译器和链接器。因此编译出的二进制可执行文件几乎可以运行在任何的系统环境中。

5．原生支持并发

Go 是一种非常高效的语言，从语言层原生支持并发，使用起来非常的简单。Go的并发是基于Goroutine。Goroutine 类似于线程，但并非线程，是 Go 面向线程的轻量级方法。创建 Goroutine 的成本很低，只需几千个字节的额外内存。通常一台普通的桌面主机运行上百个线程就会负载过大，同样的主机却可以运行上千甚至上万个 Goroutine。Goroutine 之间可以通过Channel实现通信。Goroutine 以及基于Channel的并发性方法可最大限度地使用 CPU 资源。

6．稳定性强

Go拥有强大的编译检查、严格的编码规范，具有很强的稳定性，此外Go还提供了软件生命周期（开发、测试、部署、维护等等）的各个环节的工具，如go tool、go fmt、go test。

7．垃圾回收

Go语言的使用者只需要关注内存的申请而不必关心内存的释放，Go语言内置Runtime来自动进行管理。虽然目前来说GC(内存垃圾回收机制)不算完美，但是足以应付开发者所能遇到的大多数情况，使开发者将更多精力集中在业务上，同时Go语言也允许开发者对此项工作进行优化。

安装和配置Golang

<https://blog.csdn.net/qq_57287010/article/details/128690648>

搭建集成开发环境GoLand

<https://www.jetbrains.com.cn/go/>

编写第一个程序HelloWorld

新建文件a.go

package main  
  
import "fmt"  
  
func main() {  
 fmt.Println("Hello World!")  
}

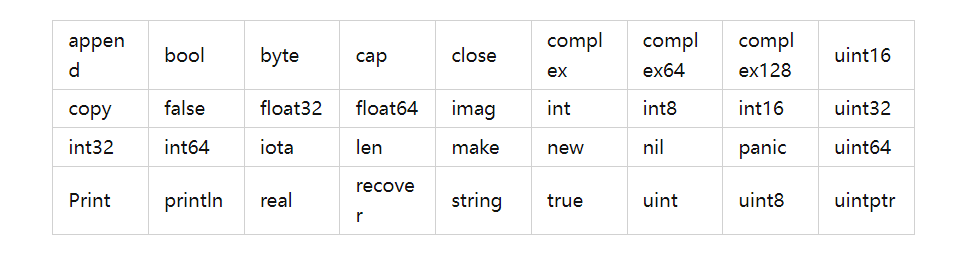
进入文件所在cmd，执行go run a.go

编译go build a.go，会生成a.exe可执行文件，直接点击运行

上述代码解释：  
第1行 package main 定义了包名。必须在源文件中非注释的第1行指明这个文件属于哪个包，如package main。package main表示一个可独立执行的程序，每个 Go 应用程序都需要包含一个名为 main 的包。并且在该包包含一个叫main()的函数（该函数是Go可执行程序的执行起点，既不能带参数，也不能定义返回值）。  
第2行 import "fmt" ，import语句用于导入该程序所依赖的包。由于本示例程序用到了Println()函数，所以需要导入该函数所属的fmt包。fmt 包实现了格式化 IO（输入/输出）。  
第3行 func main() 是程序入口。所有Go函数以关键字func开头，每一个可执行程序所必须包含main 函数，通常是程序启动后第一个执行的函数，如果有 init() 函数则会先执行init()函数。  
第4行 /\*...\*/ 是注释，代表着在程序执行时，这部分代码将被忽略。  
第5行 fmt.Println(...) ，将字符串输出到控制台。  
除此之外，还有以下几点值得注意。  
（1） 只有 package 名称为 main 的包可以包含 main 函数。  
（2） 一个可执行程序有且仅有一个 main 包。  
（3） 通过 import 关键字来导入其他非 main 包。  
（4） 可以通过 import 关键字单个导入，也可以同时导入多个

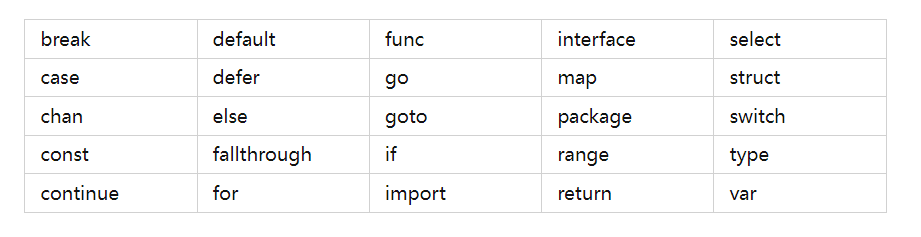
预定义标识符

预定义标识符是Go语言系统预先定义的标识符，具有见名知义的特点，如函数“输出”（[printf](https://link.zhihu.com/?target=https%3A//baike.baidu.com/item/printf" \t "_blank)）、“新建”（[new](https://link.zhihu.com/?target=https%3A//baike.baidu.com/item/printf" \t "_blank)）、“复制”（copy）等。预定义标识符可以作为用户标识符使用，只是这样会失去系统规定的原意，使用不当还会使程序出错。下面列举了36个预定义标识符，如图所示。



关键字

Go语言的关键字是系统自带的，是具有特殊含义的标识符。Go语言内置了25个关键字用于开发。下面列举了Go代码中会使用到的25个关键字或保留字，如图所示。



自定义标识符

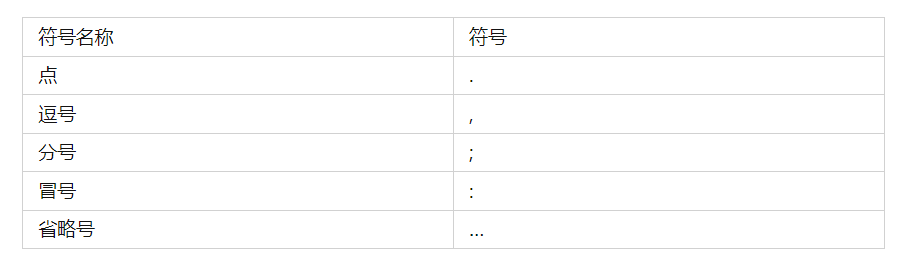
用户根据需要自定义的标识符，一般用来给变量、类型、函数等程序实体起名字。  
自定义标识符实际上是一个或是多个字母(A~Z和a~z)、数字(0~9)、下画线（\_）组成的序列，但是第一个字符必须是字母或下画线而不能是数字。  
Go不允许在自定义标识符中使用@、$和%等标点符号，或预定义标识符和关键字。Go是一种区分大小写的编程语言。因此，Manpower和manpower是两个不同的标识符。  
无效的标识符如图所示。



Go语言的编码规范

分隔符

程序中可能会使用到的分隔符：括号()、中括号[]和大括号{}。  
程序中可能会使用到的标点符号，如图所示。



可见性规则

Go语言中，使用大小写来决定标识符（常量、变量、类型、接口、结构或函数）是否可以被外部包所调用。  
如果标识符以一个大写字母开头，那么其对象就可以被外部包的代码所使用（使用时程序需要先导入这个包），如同面向对象语言中的 public。  
如果标识符的对象以小写字母开头，则对包外是不可见的，但是它们在整个包的内部是可见并且可用的，如同面向对象语言中的 private 。

变量声明

初始化变量的标准格式：



未初始化的批量格式，不用每行都用var申明，具体语法格式如下所示：



1. 整形和浮点型变量默认值：0
2. 字符串默认值为空字符串
3. 布尔型默认值为false
4. 函数、指针变量、切片默认值为nil

初始化变量的编译器自动推断类型格式：



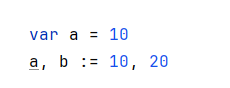
初始化变量的简短声明格式（短变量声明格式）：



使用 := 赋值操作符，:= 可以高效地创建一个新的变量，称之为初始化声明。声明语句省略了var 关键字，变量类型将由编译器自动推断。这是声明变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。该变量名必须是没有定义过的变量，若定义过，将发生编译错误。



多个短变量声明和赋值中，至少有一个新声明的变量出现在左侧中，那么即便有其他变量名可能是重复声明的，编译器也不会报错。情况如下所示。

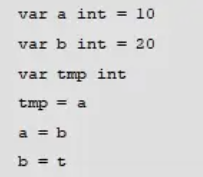


虽然这种方法不会报错，但是在使用过程中应尽量避免。

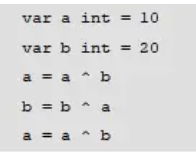
变量多重赋值

变量多重赋值是指多个变量同时赋值。Go语法中，变量初始化和变量赋值是两个不同的概念。Go语言的变量赋值与其他语言一样，但是Go提供了其他程序员期待已久的多重赋值功能，可以实现变量交换。多重赋值让Go语言比其他语言减少了代码量。

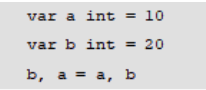
以简单的算法交换变量为例，传统写法如下所示。



新定义的变量是需要内存的，于是有人设计了新的算法来取代中间变量，其中一种写法如下如下所示。



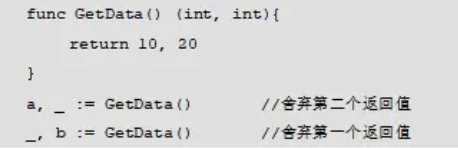
以Go语言的多重赋值功能为例，写法如下所示。



匿名变量

Go语言的函数可以返回多个值，而事实上并不是所有的返回值都用得上。那么就可以使用匿名变量，用“\_”下画线替换即可。

例如，定义一个函数，功能为返回两个int型变量，第一个返回10，第二个返回20，第一次调用舍弃第二个返回值，第二次调用舍弃第一个返回值，具体语法格式如下所示。



匿名变量既不占用命名空间，也不会分配内存。

常量

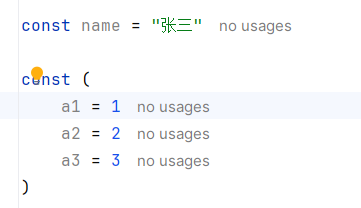
使用const关键字声明常量，常量的类型可以不写

常量在声明时必须赋值

常量不能修改

常量只能修饰bool、数值类型、string类型

通过首字母的大小写来控制常量的访问范围



数据类型

基本数据类型（原生数据类型）：整型、浮点型、布尔型、字符串、字符（byte、rune）。

复合数据类型（派生数据类型）：指针（pointer）、数组（array）、切片（slice）、映射（map）、函数（function）、结构体（struct）、通道（channel）。

整型

整型分两大类

有符号整形：int8、int16、int32、int64、int

无符号整型：uint8、uint16、uint32、uint64、uint

其中uint8就是byte型，int16对应C语言的short型，int64对应C语言的long型。

对整型的详细描述，如图所示。



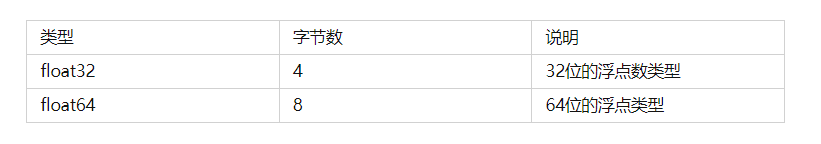
声明方式如下所示。

var a int8 //声明有符号8位整型

var b uint8 //声明无符号8位整型

浮点型

浮点型表示存储的数据是实数，如3.145。关于浮点类型的说明，如表所示。



声明方式如下所示。

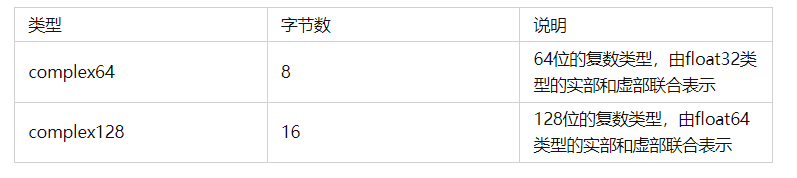
var x float32 //声明32位浮点数类型

float32的最大范围是3.4e38，用常量定义是：math.MaxFloat32，在代码中直接使用可以代表最大范围的值。

float64的最大范围是1.8e308，用常量定义是：math.MaxFloat64，在代码中直接使用可以代表最大范围的值。

复数型

复数型用于表示数学中的复数，如1+2j、1-2j、-1-2j等。关于复数类型的说明，如图所示。



布尔型

布尔类型用系统标识符bool表示。在其他编程语言中，对于布尔类型的值定义，非0表示真，0表示假。而在Go语言中，布尔值只可以是常量true或者false。

声明方式如下所示。

var flag bool

布尔型无法参与数值运算，也无法与其他类型进行转换。

字符串

字符串在Go语言中是以基本数据类型出现的，使用字符串就像使用其他原生基本数据类型int、float32、float64、bool一样。

字符串在C++语言中，是以类的方式进行封装，不属于基本数据类型。



有些字符都没有现成的文字代号。所以只能用转义字符来表示 。常用的转义字符如图所示。



定义多行字符串的方法如下。

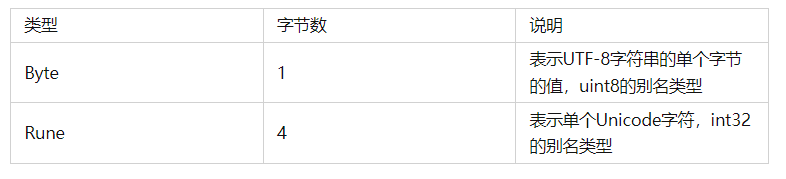
双引号书写字符串被称为字符串字面量（string literal），这种字面量不能跨行；

多行字符串需要使用反引号“`”，多用于内嵌源码和内嵌数据;

在反引号中的所有代码不会被编译器识别，而只是作为字符串的一部分。

字符

字符串中的每一个元素叫做“字符”，定义字符时使用单引号。Go语言的字符有两种，如图所示。



声明示例如下所示。



数据类型转换

Go语言采用数据类型前置加括号的方式进行类型转换。格式如：T(表达式)，T表示要转换的类型；表达式包括变量、数值、函数返回值等。

类型转换时，需要考虑两种类型之间的关系和范围，是否会发生数值截断。就像将1000毫升的水倒入容积为500毫升的瓶子里，余出来500毫升的水便会流失。值得注意的是，布尔型无法与其他类型进行转换。

浮点型与整型之间转换

float和int的类型精度不同，使用时需要注意float转int时精度的损失。

整型转字符串类型

这种类型的转换，其实相当于是byte或rune转string。该int数值是ASCII码的编号或Unicode字符集的编号。转成string就是将根据字符集，将对应编号的字符查找出来。当该数值超出Unicode编号范围，则转成的字符串显示为乱码。例如19968转string，就是“一”。

备注：

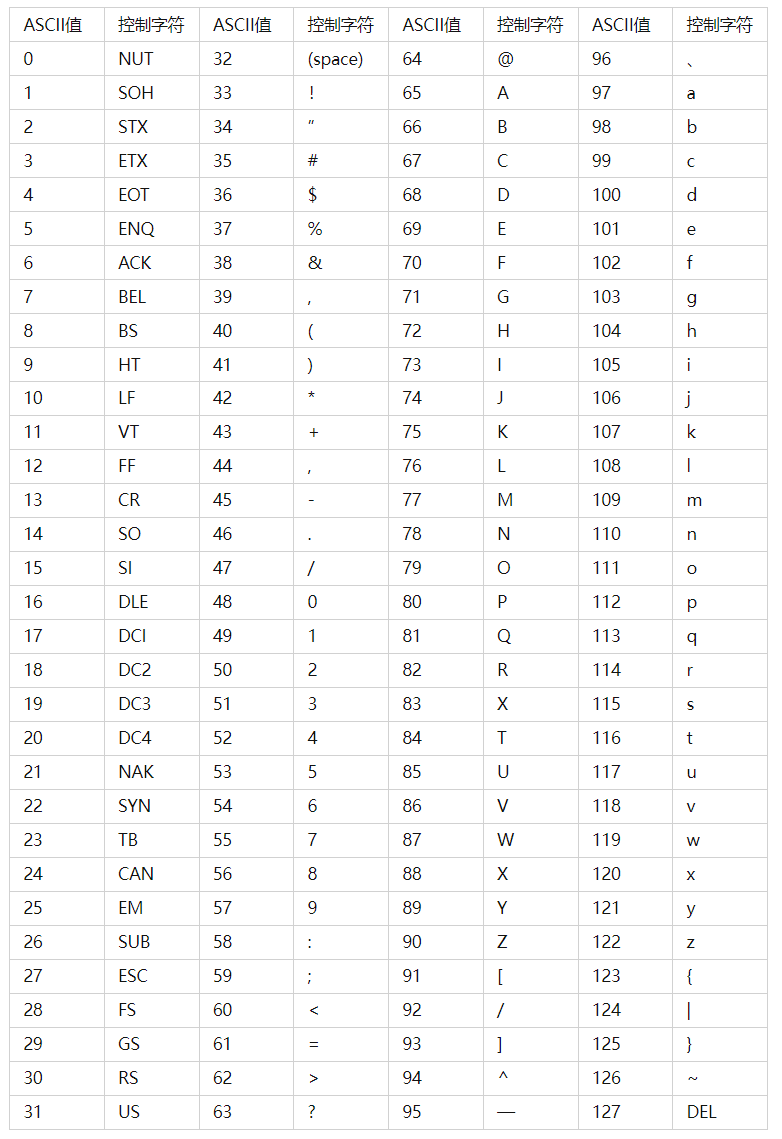
l ASCII字符集中数字的10进制范围是48~57

l ASCII字符集中大写字母的10进制范围是65~90

l ASCII字符集中小写字母的10进制范围是97~122

l unicode字符集中汉字的范围是4e00-9fa5，十进制范围是19968 - 40869

详情如图所示。

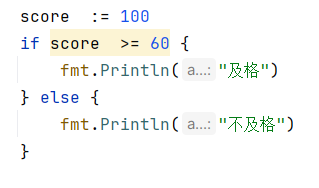


注意：在Go语言中，不允许字符串转int，会产生如下错误。

cannot convert str (type string) to type int

流程控制概述

条件判断语句



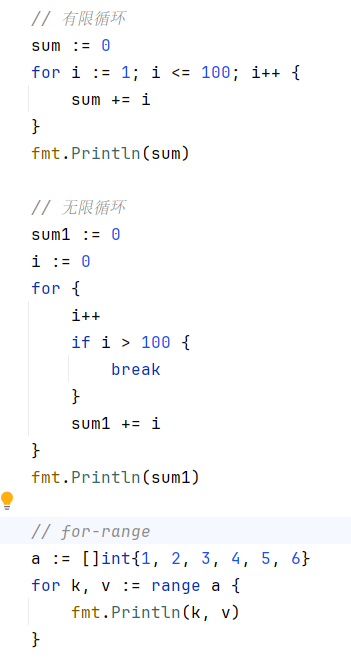
条件分支语句





循环语句

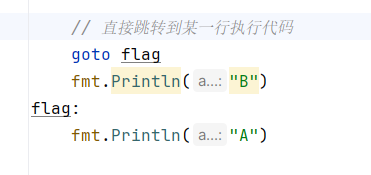
与其它编程语言不同的是，Go 语言中的循环语句只支持 for 关键字，而不支持 while 和 do-while 结构。关键字 for 的基本使用方法与其他语言类似，只是循环条件不含括号



循环控制语句

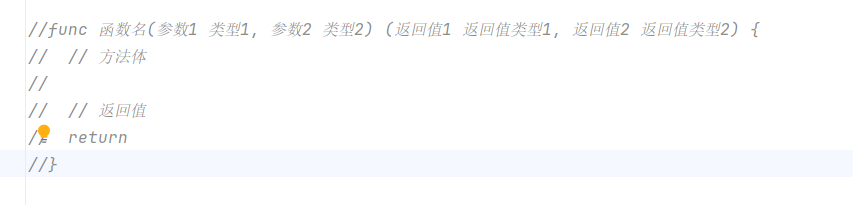


goto 可以打破原有代码执行顺序，直接跳转到某一行执行代码。



函数

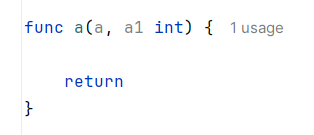
函数定义



1. 函数由func声明
2. 函数名和参数列表一起构成了函数签名。函数名由字母、数字和下画线组成。函数名的第一个字母不能为数字。在同一个包内，函数不能重名。
3. 参数列表指定的是参数类型、顺序及参数个数。参数是可选的，也就是说函数可以不包含参数。
4. Go语言的函数可以返回多个值，参数也是可选的。

参数类型简写

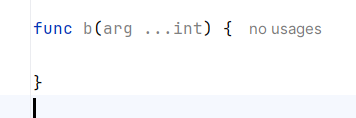
如果有多个参数，并且参数的类型一致可以简写成：



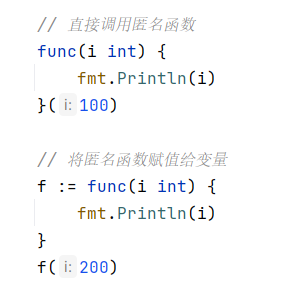
如果没有返回值可以不写后面的括号

Go语言的函数支持可变参数。接受变参的函数是有着不定数量的参数的。语法格式如下所示。

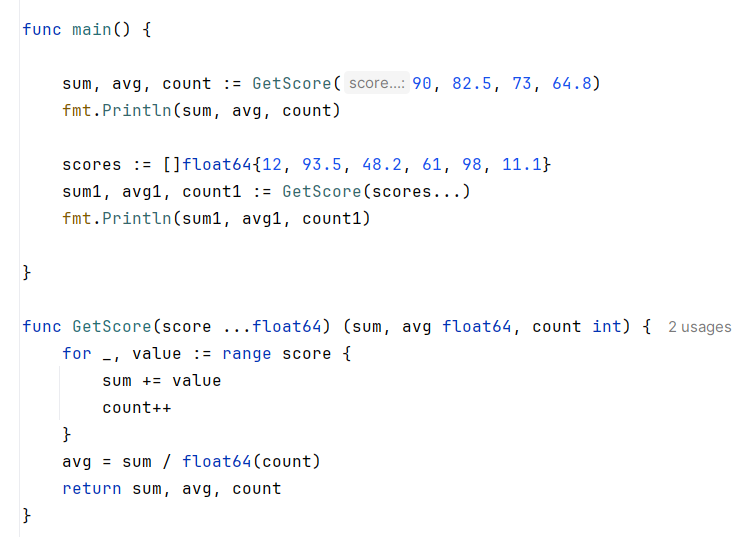
Func



匿名函数



可变参数



闭包

闭包的概念

闭包并不是什么新奇的概念，它早在高级语言开始发展的年代就产生了。闭包（Closure）是词法闭包（Lexical Closure）的简称。

闭包是由函数和与其相关的引用环境组合而成的实体。在实现深约束时，需要创建一个能显式表示引用环境的东西，并将它与相关的子程序捆绑在一起，这样捆绑起来的整体被称为闭包。函数+引用环境=闭包。

闭包只是在形式和表现上像函数，但实际上不是函数。函数是一些可执行的代码，这些代码在函数被定义后就确定了，不会在执行时发生变化，所以一个函数只有一个实例。

闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。闭包在某些编程语言中被称为Lambda表达式。

函数本身不存储任何信息，只有与引用环境结合后形成的闭包才具有“记忆性”。函数是编译器静态的概念，而闭包是运行期动态的概念。

对象是附有行为的数据，而闭包是附有数据的行为。

闭包的优点

（1）加强模块化。闭包有益于模块化编程，它能以简单的方式开发较小的模块，从而提高开发速度和程序的可复用性。和没有使用闭包的程序相比，使用闭包可将模块划分得更小。

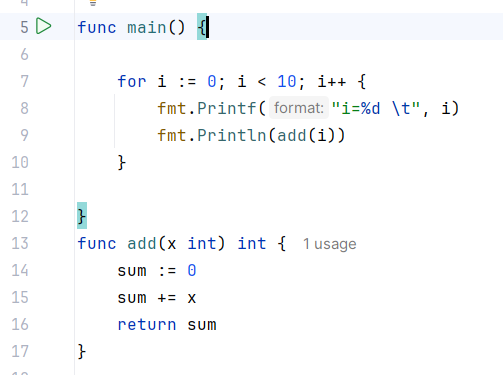
比如要计算一个数组中所有数字的和，这只需要循环遍历数组，把遍历到的数字加起来就行了。如果现在要计算所有元素的积，又或者要打印所有的元素呢？解决这些问题都要对数组进行遍历，如果是在不支持闭包的语言中，程序员不得不一次又一次重复地写循环语句。而这在支持闭包的语言中是不必要的。这种处理方法多少有点像回调函数，不过要比回调函数写法更简单，功能更强大。

（2）抽象。闭包是数据和行为的组合，这使得闭包具有较好抽象能力。

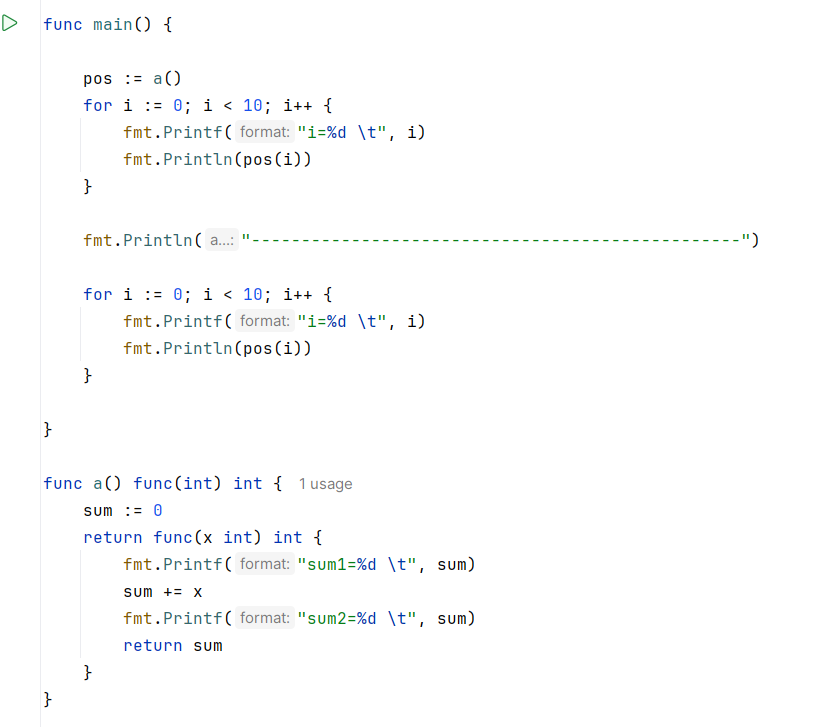
（3）简化代码。一个编程语言需要以下特性来支持闭包。

* 函数是一阶值（First-class value，一等公民），即函数可以作为另一个函数的返回值或参数，还可以作为一个变量的值。
* 函数可以嵌套定义，即在一个函数内部可以定义另一个函数。
* 允许定义匿名函数。
* 可以捕获引用环境，并把引用环境和函数代码组成一个可调用的实体；

没有使用闭包进行计数的代码:



使用闭包函数进行计数的代码：



由于闭包函数“捕获”了和它在同一作用域的其他常量和变量。所以当闭包在任何地方被调用，闭包都可以使用这些常量或者变量。它不关心这些变量是否已经超出作用域，只要闭包还在使用它，这些变量就依然存在。

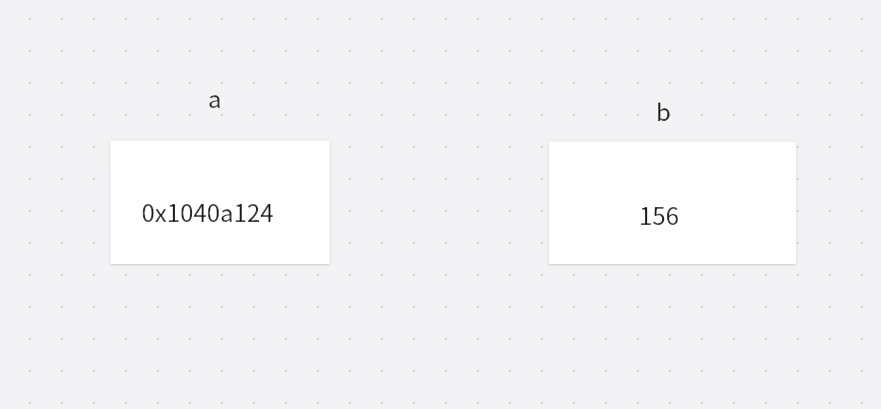
指针

指针是存储另一个变量的内存地址的变量。

变量是一种使用方便的占位符，变量都指向计算机的内存地址。

一个指针变量可以指向任何一个值的内存地址。

例如：变量b的值为156，存储在内存地址0x1040a124。变量a持有b的地址，则a被认为指向b。如图所示。



## 指针及地址的使用方法

Go语言中使用&字符放在变量前面对变量进行“取地址”操作。Go语言中的值类型（int、float、bool、string、array、struct）都有对应的指针类型，如：\*int、\*int64、\*string等

当使用 & 操作符对普通变量进行取地址操作时，可以得到变量的指针。此时可以对指针使用 \* 操作符，可以得到变量值（此操作也叫指针取值），如以下代码：



说明：取地址操作符 & 和取值操作符 \* 是一对互补操作符， & 取出地址， \* 根据地址取出地址指向的值。

变量、指针地址、指针变量、取地址、取值的相互关系和特性如下：

* 对变量进行取地址操作使用 & 操作符，可以获得这个变量的指针变量。
* 指针变量的值是指针地址。
* 对指针变量进行取值操作使用 \* 操作符，可以获得指针变量指向的原变量的值。

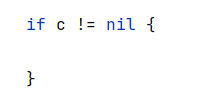
\* 操作符的根本意义就是操作指针指向的变量。当操作在右值时，就是取指向变量的值，当操作在左值时，就是将值设置给指向的变量。如下代码进行说明：



在对普通变量使用&操作符取地址后会获得这个变量的指针，然后可以对指针使用\*操作，也就是指针取值

空指针

在Go语言中，当一个指针被定义后没有分配到任何变量时，它的值为 nil。 nil 指针也称为空指针。nil在概念上和其它语言的null、None、NULL一样，都指代零值或空值。



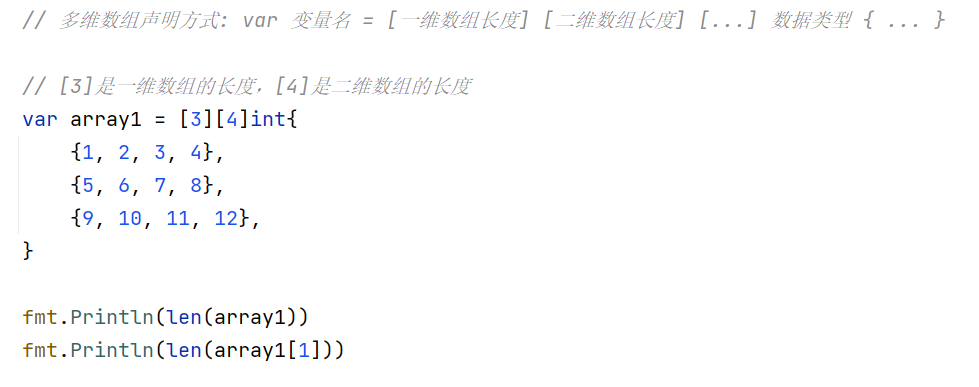
数组

数组是相同类型的一组数据构成的长度固定的序列，其中数据类型包含了基本数据类型、复合数据类型和自定义类型。数组中的每一项被称为数组的元素。数组名是数组的唯一标识符， 数组的每一个元素都是没有名字的，只能通过索引下标（位置）进行访问。因为数组的内存是一段连续的存储区域，所以数组的检索速度是非常快的，但是数组也有一定的缺陷，就是定义后长度不能更改。

一维数组



多维数组



切片

Go 语言中数组的长度不可改变，但在很多应用场景中，在初始定义数组时，数组的长度并不可预知，这样的序列集合无法满足要求，Go中提供了另外一种内置类型“切片（slice）”，弥补了数组的缺陷。切片是可变长度的序列，序列中每个元素都是相同的类型。切片的语法和数组很像。

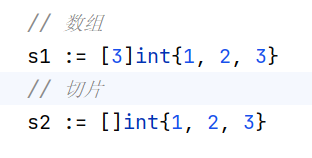
从底层来看，切片引用了数组的对象。切片可以追加元素，在追加时可能使切片的容量增大。与数组相比，切片不需要设定长度，在[]中不用设定值，相对来说比较自由。

切片的数据结构可理解为一个结构体，这个结构体包含了三个元素：

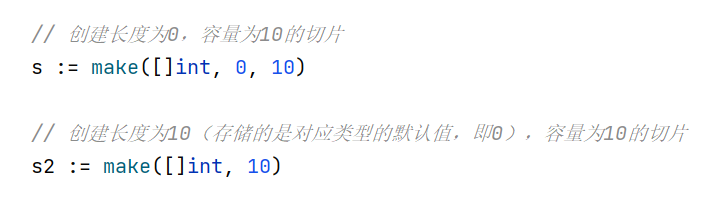
* 指针，指向数组中切片指定的开始位置。
* 长度，即切片的长度。
* 容量，也就是切片开始位置到数组的最后位置的长度。

创建和初始化

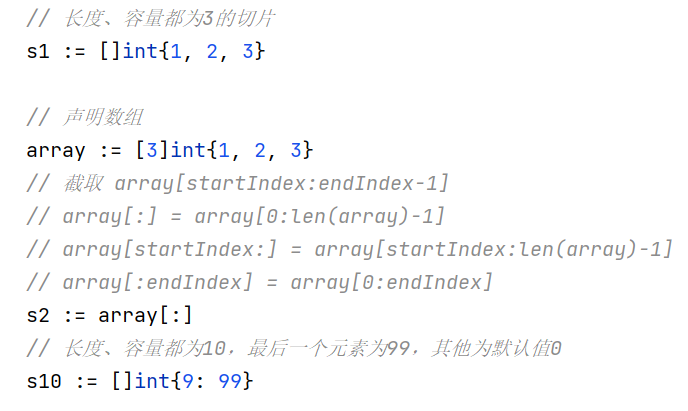
数组和切片的创建区别在于[]中是否指定了长度



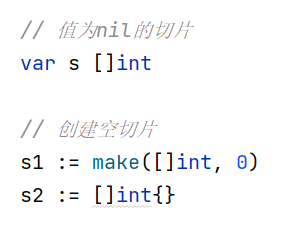
make函数创建



初始化创建

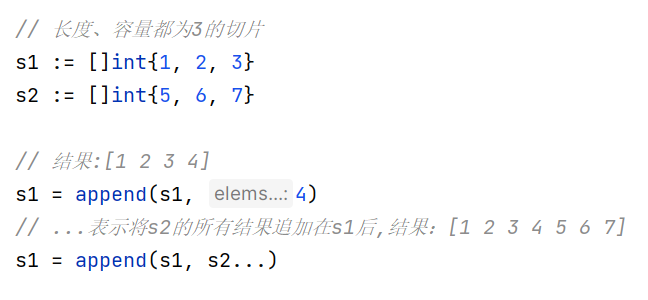


空切片与nil



操作

Go语言的内建函数append（）可以为切片动态添加元素，每个切片会指向一片内存空间，这片空间能容纳一定数量的元素。当空间不能容纳足够多的元素时，切片就会进行扩容（2倍），扩容操作往往发生在append（）函数调用时。



使用Go语言内建的[copy函数](https://so.csdn.net/so/search?q=copy%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020" \t "_blank)，可以迅速地将一个切片的数据复制到另一个切片空间中，如果加入的两个数组切片不一样大，就会按照其中较小的那个数组切片的元素个数进行复制。



Go语言并没有对删除切片元素提供专用的语法或者接口，需要使用切片本身的特性来删除元素，根据要删除元素的位置有三种情况，分别是从开头位置删除、从中间位置删除和从尾部删除，其中删除切片尾部的元素速度最快。



扩容

Slice依托数组实现，底层数组容量不足时可自动重新分配。追加数据通过append（返回值一定要再赋值给原slice）；容量不足时会自动扩容：

* <1024时，扩大为原来2倍；
* >=1024时，扩大为原来的1.25倍；

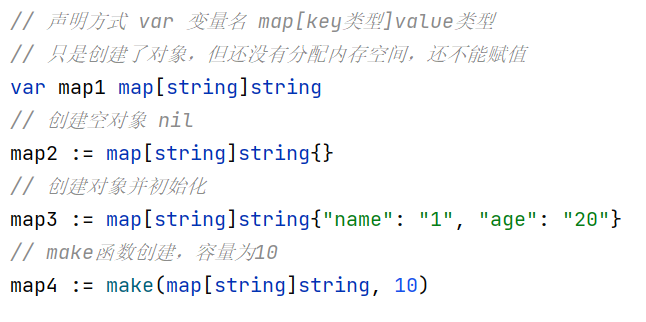
Map

以键值对为元素的数据集合，特点：

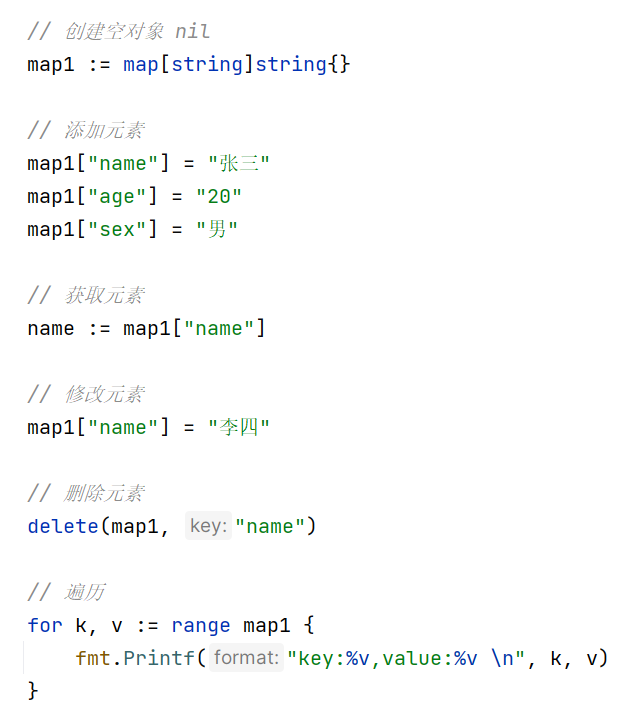
1. 键不能重复
2. 键必须可哈希（int/bool/float/string/array）
3. 无序

快：可以直接根据键值找到数据存储的位置，而不用从前到后一一比对

map声明和初始化



map常见操作



常用包

字符串

### 检索字符串



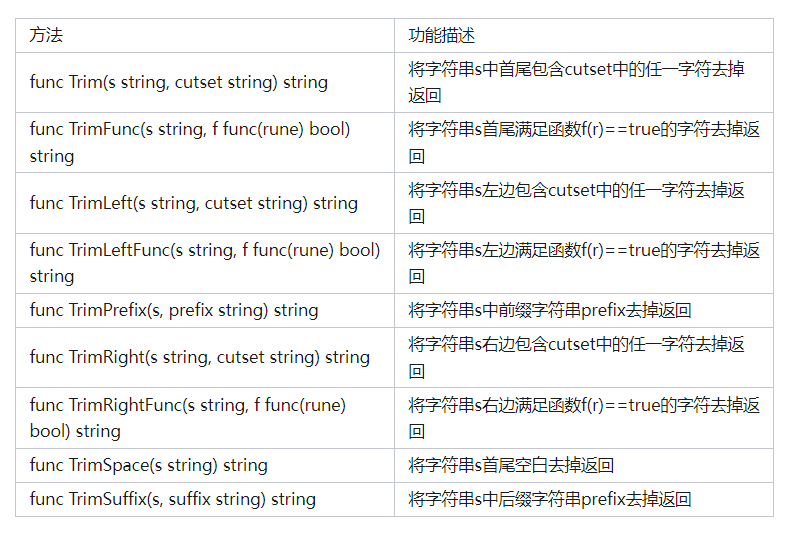
### 分割字符串



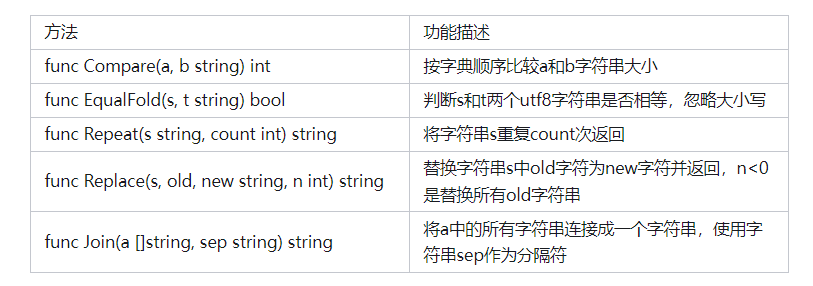
### 大小写转换



### 修剪字符串

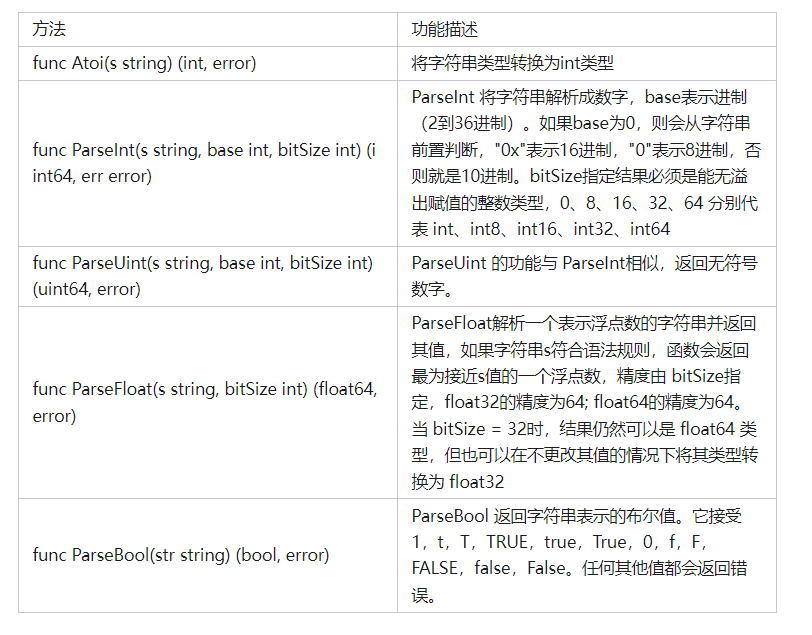


### 比较字符串



### Parse类函数

Parse类函数主要的功能是将字符串转其他类型



### Fotmat类函数

Format类函数主要的功能是将其他类型格式化成字符串



### regexp正则表达式包

正则表达式(regular expression)就是由元字符组成的一种字符串匹配的模式，使用这种模式可以实现对文本内容解析、校验、替换。

正则表达式经常应用在验证用户ID、密码、身份证、手机号等是否符合格式要求。如果填写的内容与正则表达式不匹配，就可以断定内容是不符合要求或者是虚假的信息。

正则表达式还有其他重要的用途，就是模糊查询与批量替换。可以在文档中使用一个正则表达式来查找匹配的特定文字，然后或者将其删除，或者替换为别的文字。

#### 正则表达式中主要元字符

#### regexp包中核心函数及方法介绍

## time包



## math包

math包提供了基本的数学常数和数学函数，使用时需要import "math"

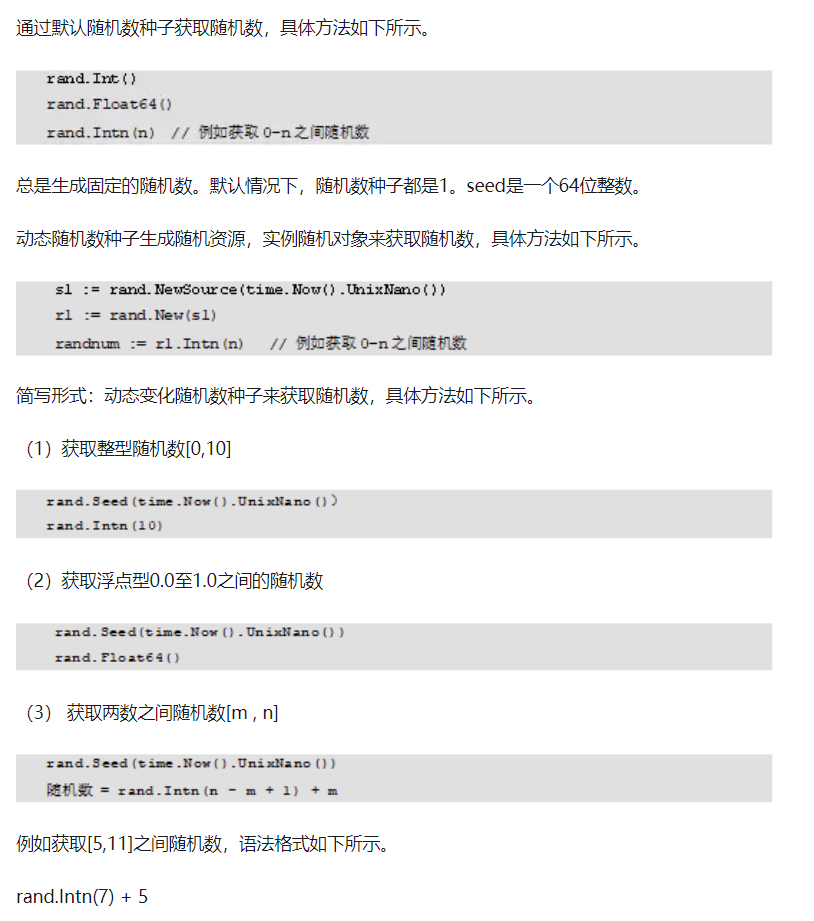


## rand包（随机数）

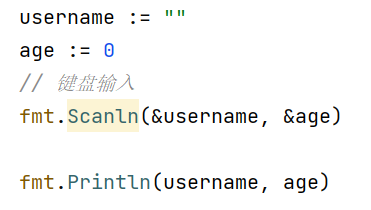
使用随机数时需要import "math/rand"，rand包实现了伪随机数生成器。随机数从资源生成。包水平的函数都使用的默认的公共资源。该资源会在程序每次运行时都产生确定的序列。如果需要每次运行产生不同的序列，应使用Seed函数进行初始化。默认资源可以安全的用于多协程并发。



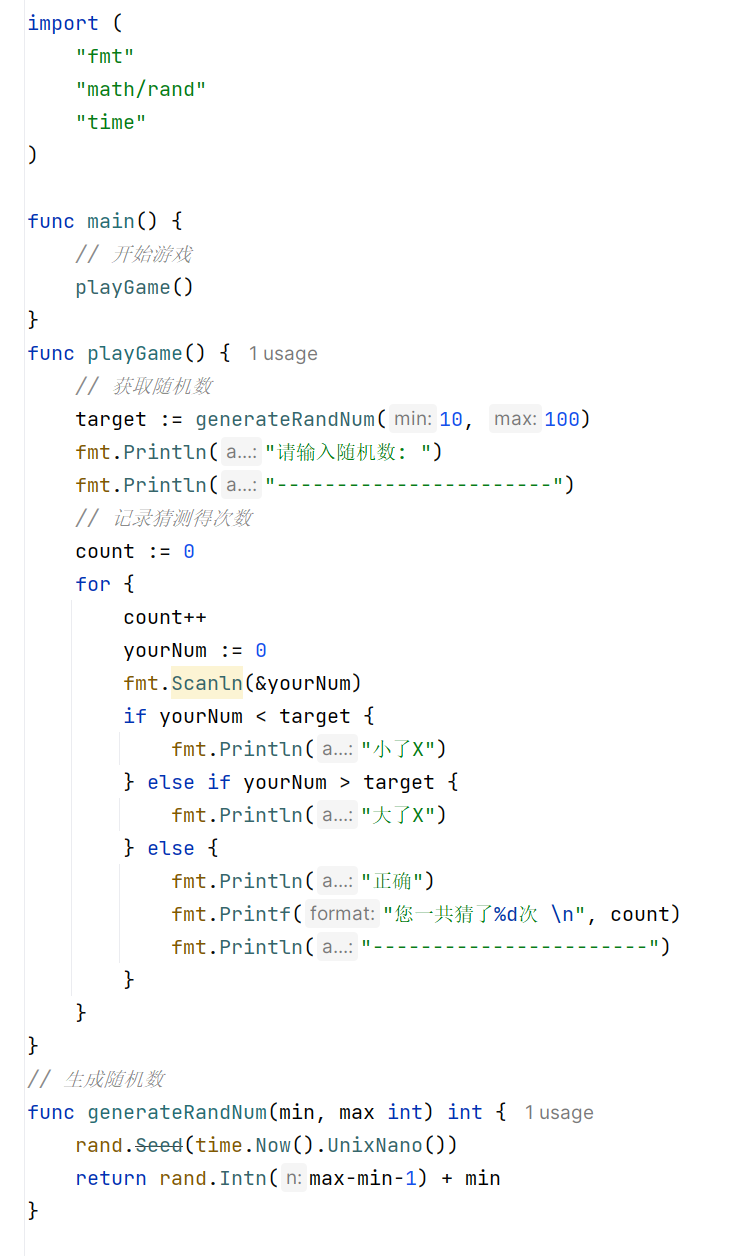
### 获取随机数的几种方式



## Scanln()函数（键盘输入）



## 随机数+键盘输入案例—猜数字游戏



# 面向对象编程

## Go语言面向对象

在其他编程语言中大多使用关键字“类”来定义封装对象，表示该类的具体特征，然而Go并不是一个纯面向对象的编程语言。在Go语言中的面向对象，采用更灵活的结构体替代了“类”。

Go语言并没有提供类（class），但是它提供了结构体（struct），方法（method），可以在结构体上添加。提供了捆绑数据和方法的行为，这些数据和方法与类类似。

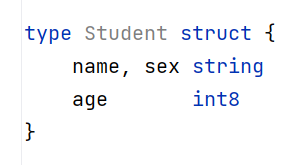
Go语言设计的非常简洁优雅，Go没有沿袭传统面向对象编程中的诸多概念，比如继承、虚方法、构造方法和析构方法等。

虽然Go语言没有继承和多态。但是Go语言可以通过匿名字段实现继承，通过接口实现多态。在Go语言中学习面向对象，主要学习结构体（struct）、方法（method）、接口（interface）。

## 定义结构体与实例化



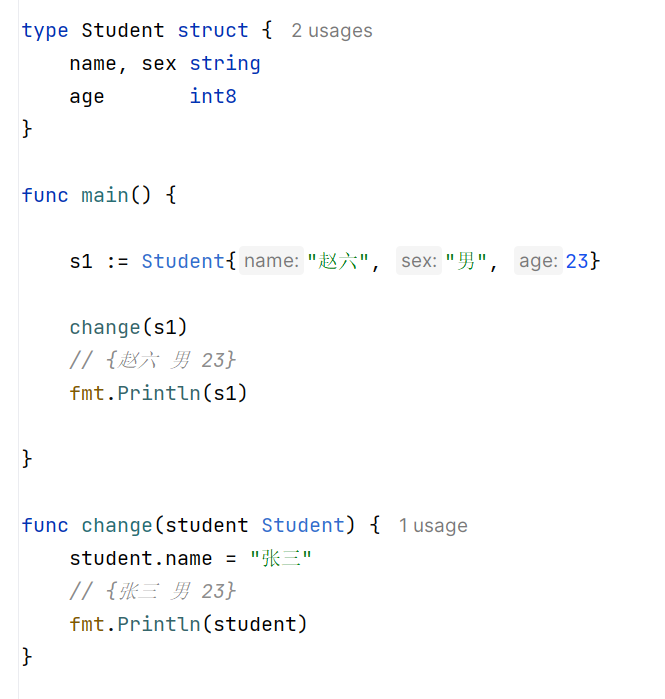
1. 类型名：标识结构体的名称，在同一个包内不能重复。
2. 结构体中属性，也叫字段，必须唯一。
3. 同类型的成员属性可以写在一行。





## 结构体是值类型

结构体作为函数参数，实际将参数复制一份传递到函数中，在函数中对参数进行修改，不会影响到实际参数。证明结构体是值类型



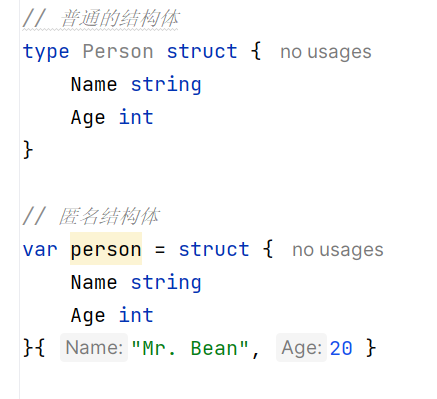
## 结构体的深拷贝和浅拷贝

值类型是深拷贝，深拷贝就是为新的对象分配了内存。引用类型是浅拷贝，浅拷贝只是复制了对象的指针。



## 匿名结构体和匿名字段

匿名结构体是golang语言特有的概念。它是指没有名字的结构体，通常被用来表示简单的数据结构或者临时的数据类型。匿名结构体的定义方式与普通的结构体类似，但是没有名字。在定义时，在struct之后省略了结构体的名称，直接定义结构体的成员。



同时，我们也可以在匿名结构体中再定义匿名结构体来进行嵌套的操作。此时，我们可以直接调用内层匿名结构体的属性或方法。



优点

* 提供了一种简洁的方式来定义结构体
* 提高了数据封装的灵活性

使用场景

用于函数参数和返回值



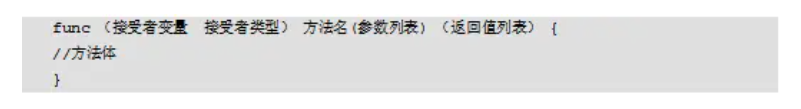
## 方法

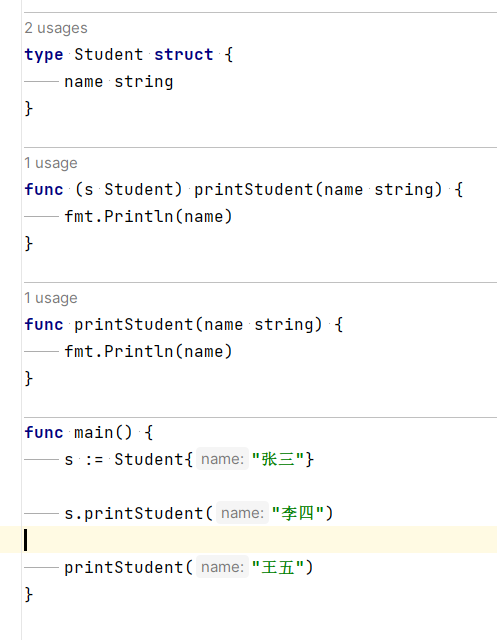
### 概念

函数（function）是一段具有独立功能的代码，可以被反复多次调用，从而实现代码复用。而方法（method）是一个类的行为功能，只有该类的对象才能调用。

* Go语言的方法（method）是一种作用于特定类型变量的函数。这种特定类型变量叫做Receiver（接受者、接收者、接收器）。接受者的概念类似于传统面向对象语言中的this或self关键字。
* Go语言的接受者强调了方法具有作用对象，而函数没有作用对象。一个方法就是一个包含了接受者的函数。
* Go语言中，接受者的类型可以是任何类型，不仅仅是结构体，也可以是struct类型外的其他任何类型。
* 只要接受者不同，方法名就可以相同。

### 语法





接受者在func关键字和方法名之间编写，接受者可以是struct类型或非struct类型，可以是指针类型和非指针类型。 接受者中的变量在命名时，官方建议使用接受者类型的第一个小写字母。

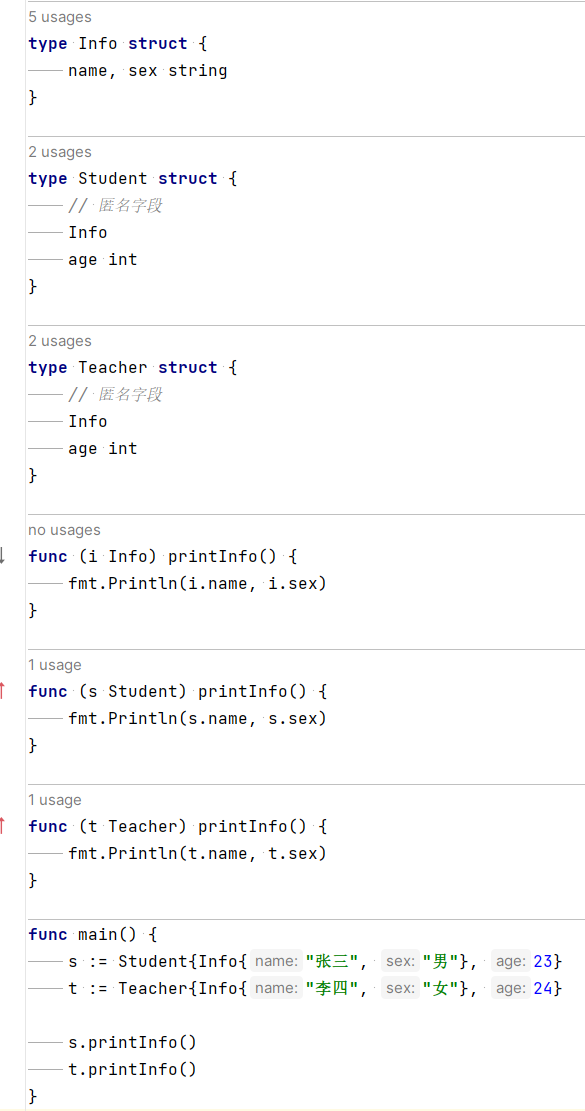
### 方法和函数

既然可以用函数来写相同的程序，却还要使用方法，主要有以下两个原因。

* Go不是一种纯粹面向对象的编程语言，它不支持类。因此其方法是一种实现类似于类的行为的方法。
* 相同名称的方法可以在不同的类型上定义，而具有相同名称的函数是不允许的。假设有一个正方形和圆形的结构。可以在正方形和圆形上定义一个名为Area的求取面积的方法。

### 方法继承和重写

方法是可以继承的，如果匿名字段实现了一个方法，那么包含这个匿名字段的struct也能调用该匿名结构体中的方法。



## 接口

面向对象语言中，接口用于定义对象的行为。接口只指定对象应该做什么，实现这种行为的方法(实现细节)是由对象来决定。

在Go语言中，接口是一组方法签名。接口只指定了类型应该具有的方法，类型决定了如何实现这些方法。当某个类型为接口中的所有方法提供了具体的实现细节时，这个类型就被称为实现了该接口。接口定义了一组方法，如果某个对象实现了该接口的所有方法，则此对象就实现了该接口。

Go语言的类型都是隐式实现接口的。任何定义了接口中所有方法的类型都被称为隐式地实现了该接口。





## duck typing

go没有 implements或extends 关键字，其实这种编程语言叫做duck typing编程语言。

Duck typing是描述事物的外部行为而非内部结构。"当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。"扩展后，可以理解为：“看起来像鸭子，那么它就是鸭子”。在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的。

使用 duck typing 的编程语言往往被归类到“动态类型语言”或者“解释型语言”里，比如 Python, Javascript, Ruby 等等，而非duck typing语言往往被归到“静态类型语言“中，比如 C/C++/Java。

以 Java为例， 一个类必须显式地声明：“类实现了某个接口”， 然后才能用在这个接口可以使用的地方。 如果有一个第三方的 Java 库，这个库中的某个类没有声明它实现了某个接口，那么即使这个类中真的有那些接口中的方法，也不能把这个类的对象用在那些要求用接口的地方。但如果在duck typing的语言中就可以这样做，因为它不要求一个类显式地声明它实现了某个接口。

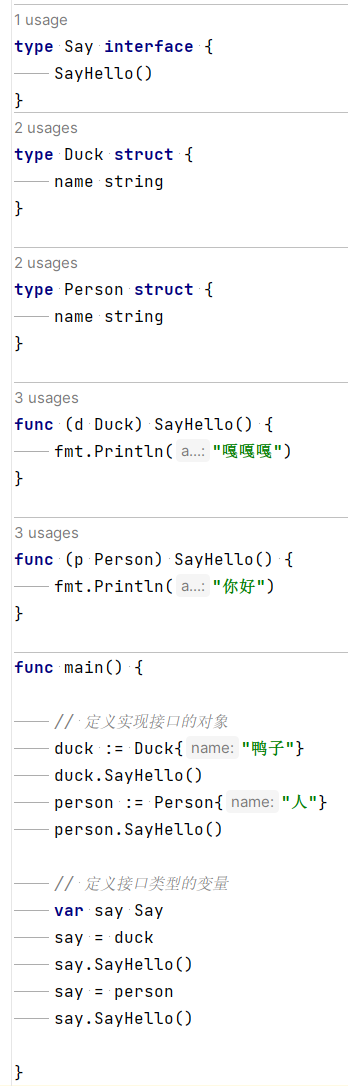
### 动态类型的优缺点

动态类型的好处很多，Python代码写起来很快。但是缺陷也是显而易见的：错误往往要在运行时才能被发现。相反，静态类型语言往往在编译时就发现这类错误：如果某个变量的类型没有显式的声明实现了某个接口，那么，这个变量就不能用在要求一个实现了这个接口的地方。

Go 类型系统采取了折中的办法，静态类型语言。原因如下。

第一，结构体类型T不需要显式地声明它实现了接口 I。只要类型 T 实现了接口 I 规定的所有方法，它就自动地实现了接口 I。 这样就像动态语言一样省了很多代码，少了许多限制。

第二，将结构体类型的变量显式或者隐式地转换为接口 I 类型的变量 i。这样就可以和其他静态类型语言一样，在编译时检查参数的合法性。



一个函数如果接收接口类型作为参数，那么实际上可以传入该接口的任意实现类的对象作为参数。定义一个接口变量，那么实际上可以赋值任意实现了该接口的对象。如果定义了一个接口类型的容器（数组或切片），实际上该容器中可以存储任意的实现类对象。

## 多态

如果有几个相似而不完全相同的对象，有时人们要求在向它们发出同一个消息时，它们的反应各不相同，分别执行不同的操作。这种情况就是多态现象。例如，甲乙丙三个班都是初中一年级，他们有基本相同的属性和行为，在同时听到上课铃声的时候，他们会分别走向三个不同的教室，而不会走向同一个教室。

多态就是事物的多种形态，Go语言中的多态性是在接口的帮助下实现的。定义接口类型，创建实现该接口的结构体对象。

定义接口类型的对象，可以保存实现该接口的任何类型的值。Go语言接口变量的这个特性实现了Go语言中的多态性。接口类型的对象，不能访问其实现类中的属性字段。

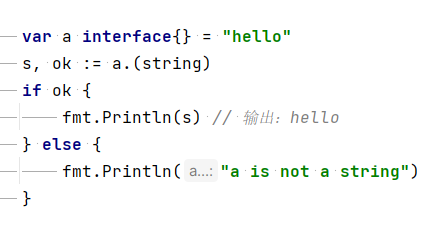
### 空接口

空接口中没有任何的方法。任意类型都可以实现该接口。空interface这样定义：interface{}，也就是包含0个method的interface。用空接口表示任意数据类型。类似于java中的object。

空接口常用于以下情形：

1. println的参数就是空接口
2. 定义一个map：key是string，value是任意数据类型
3. 定义一个切片，其中存储任意类型的数据

### 接口对象转型



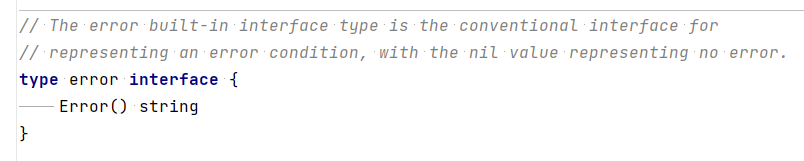
在上面的代码中，我们将一个类型为 interface{}的值转换为 string类型，并使用带检查的类型断言。由于该值实际上是一个字符串，因此转换成功，布尔值 ok 为 true，将转换后的字符串赋值给变量 s。如果转换失败，布尔值 ok 为 false，并执行 else语句块中的代码。

异常处理

## error

### error接口

Go 语言通过内置的错误类型提供了非常简单的错误处理机制。即error接口



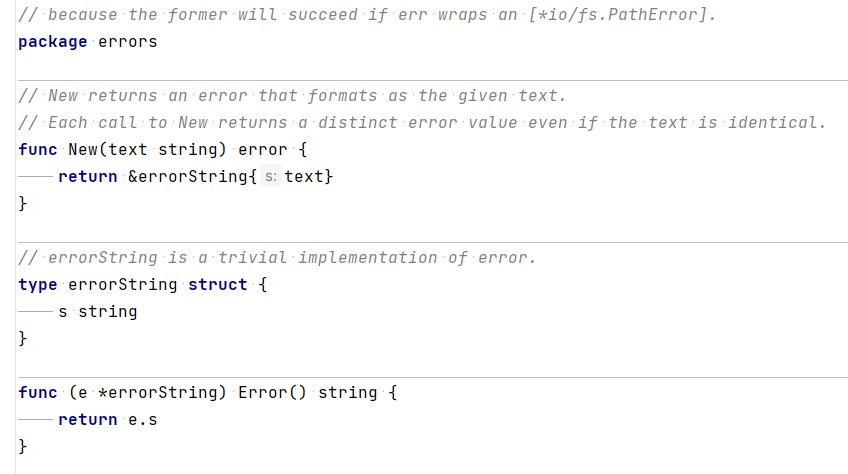
error本质上是一个接口类型，其中包含一个Error()方法。

在Go语言中处理错误的方式通常是将返回的错误与nil进行比较。nil值表示没有发生错误，而非nil值表示出现错误。

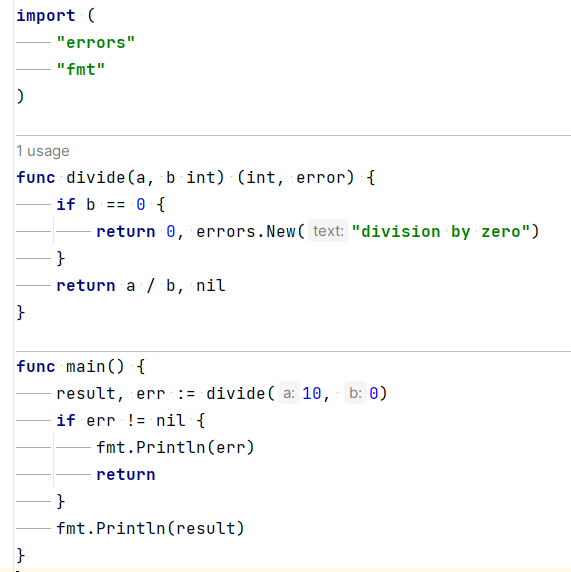
### 创建error对象

结构体只要实现了Error() string这种格式的，就代表实现了该错误接口，返回值为错误的具体描述。

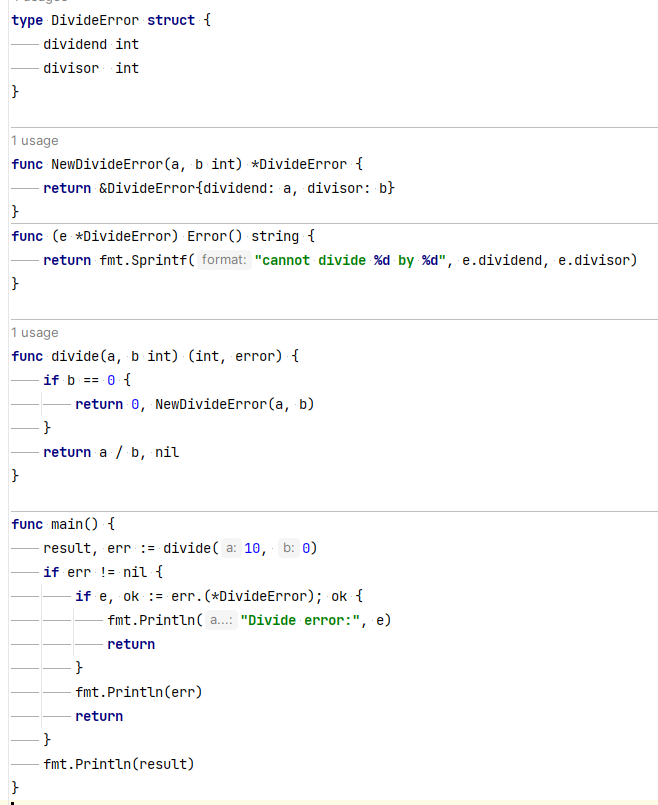
Go语言errors包对外提供了可供用户自定义的方法，errors包下的New()函数返回error对象，[errors.New](https://link.zhihu.com/?target=http%3A//errors.new/" \t "_blank)()创建新的错误。



fmt包下的Errorf()函数返回error对象，fmt包下的Errorf()[函数本质上还是调errors.New](https://link.zhihu.com/?target=http%3A//xn--errors-9o7ir3yll1cvtbq8dr47dhd3demd54m.new/)()。使用格式如下。

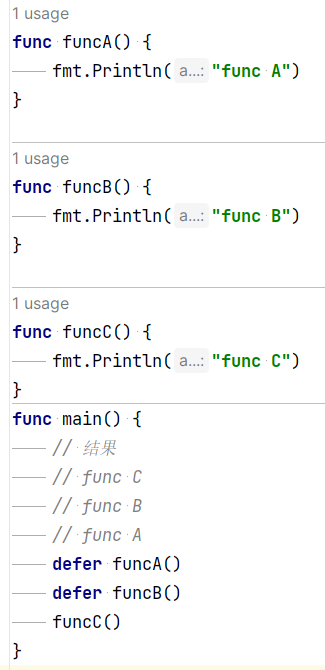


### 自定义错误类型

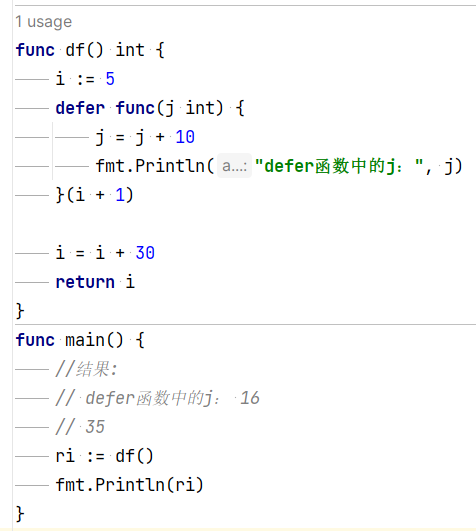


## defer

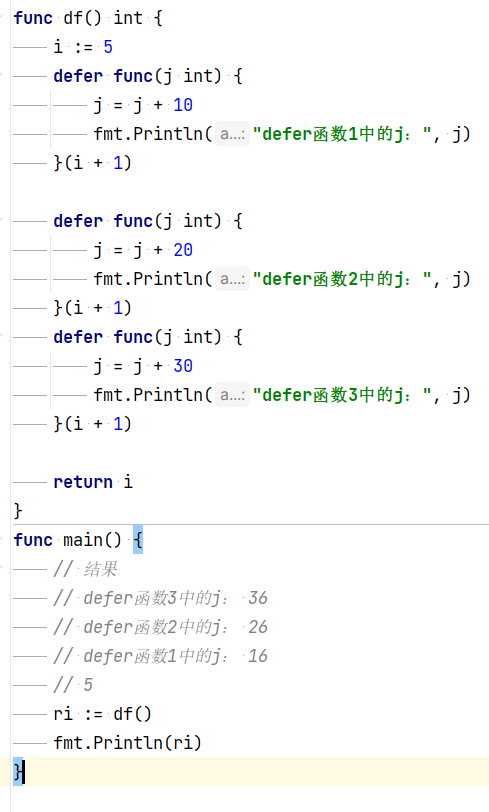
关键字defer用于延迟一个函数或者方法（或者当前所创建的匿名函数）的执行。defer语句只能出现在函数或方法的内部。



示例1:



示例2:



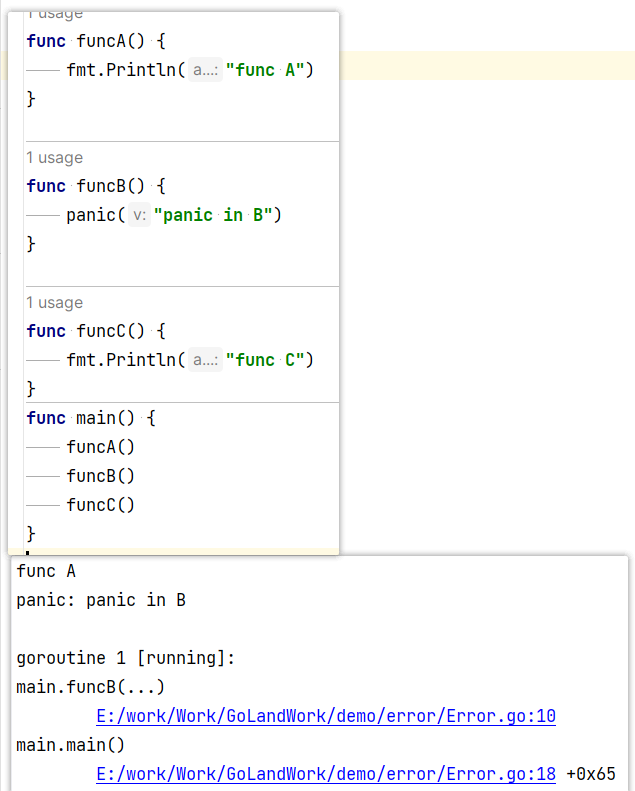
1. 延迟执行：defer关键字后的函数都是在整个函数执行结束return之后才执行的。正如上述例1代码中最终df函数的返回值是5而不是15，且先执行打印“df中的i”再执行打印“defer函数中的i”所示。上述代码是先执行完df函数中除了defer后面的函数之外的语句。return i（i的值依旧是5）之后，再执行defer后面的函数，执行i = i +10，且打印i
2. 参数预计算：defer函数的形参会在定义时就完成了该参数的拷贝。正如上述例2代码中传入到defer后面函数的形参j的实参i+1不是35，而是6。
3. FILO：先进后出，若多个defer函数在同一函数内，执行顺序遵循先进后出原理。即第一个defer函数最后一个被执行。正如上述例3代码中第三个defer函数先执行，然后再是第二个defer函数执行，最后再是第一个defer函数执行

## panic和recove

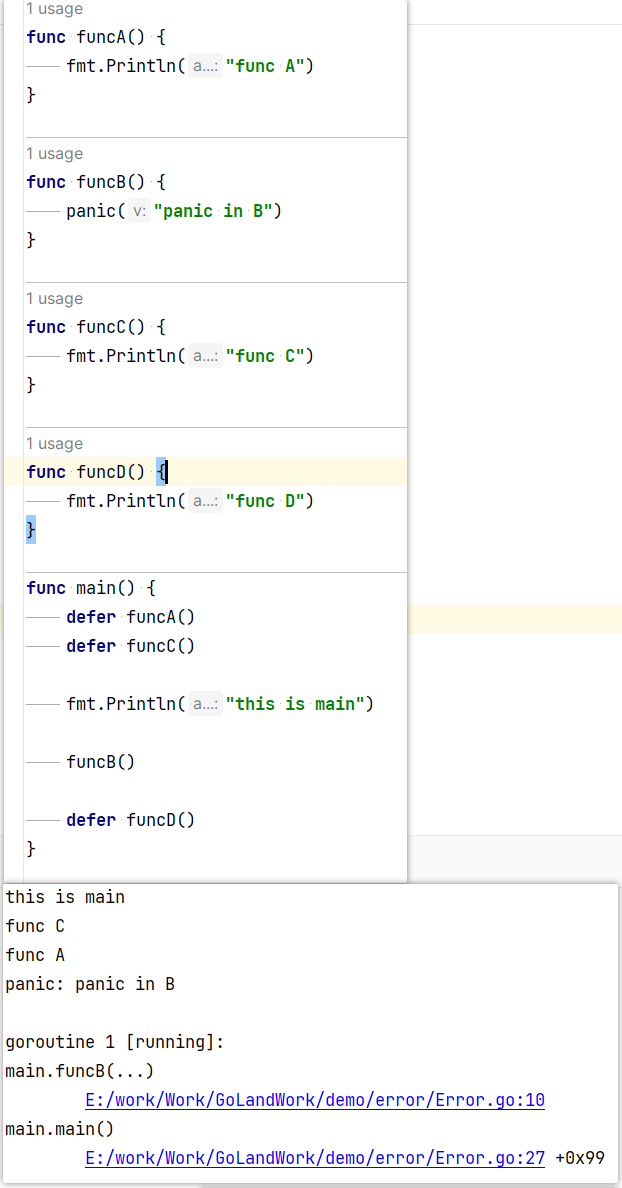
panic（宕机） 和 recover（恢复）是[Go语言](https://so.csdn.net/so/search?q=Go%E8%AF%AD%E8%A8%80&spm=1001.2101.3001.7020" \t "_blank)的两个内置函数，这两个内置函数用来处理Go的运行时错误(runtime errors)。panic 用来主动抛出异常，recover 用来捕获panic 抛出的异常。

### Panic

引发panic有两种情况：一种是程序主动调用panic()函数，另一种是程序产生运行时错误，由运行时检测并抛出。



当panic()触发的宕机发生时，panic后面的代码将不会被执行，但是在panic()函数前面的已经执行过的defer语句依然会在宕机发生时执行defer中的延迟函数。

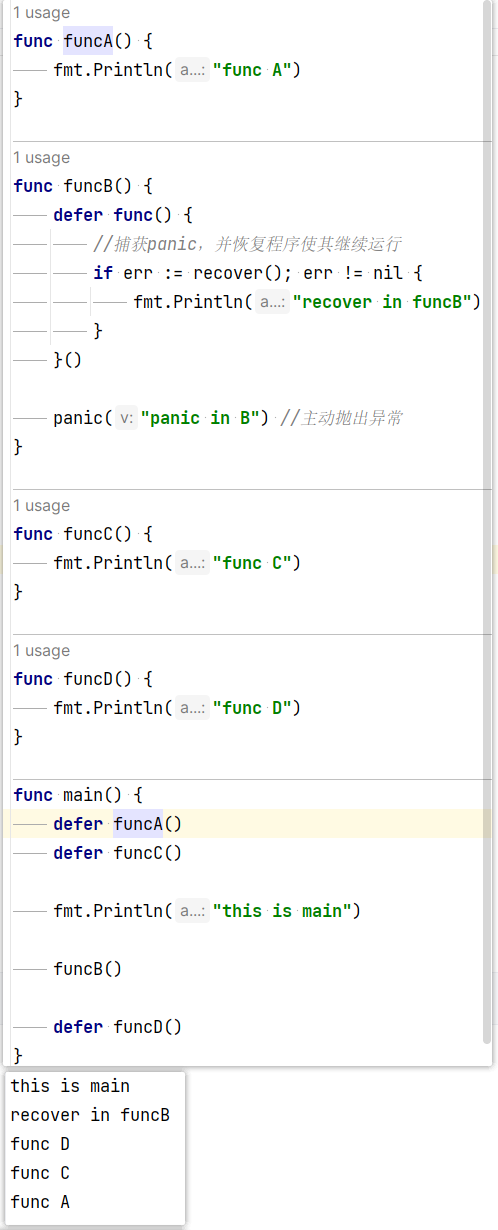


### recover

recover()函数用来捕获或者说是拦截panic的，阻止panic继续向上层传递。无论是主动调用panic()函数触发的宕机还是程序在运行过程中由Runtime层抛出的异常，都可以配合defer 和 recover 实现异常捕获和恢复，让代码在发生panic后能够继续执行。

在其他编程语言中，如Java，宕机往往以异常的形式存在。底层抛出异常，上层逻辑通过try...catch...机制捕获异常并处理，没有被捕获到的严重异常会导致程序崩溃，捕获的异常可以被处理，让代码可以继续执行。Go语言没有异常系统，其使用panic触发宕机类似于其他语言的抛出异常，而recover的宕机恢复机制就对应try...catch机制。

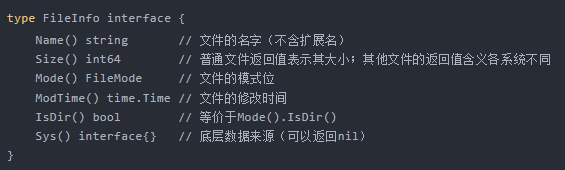
recover()函数被调用后，会返回一个 interface{} 接口类型的返回值，如果返回值等于nil，说明没有触发panic；反之，说明程序发生了panic，就应该采取相应的措施。



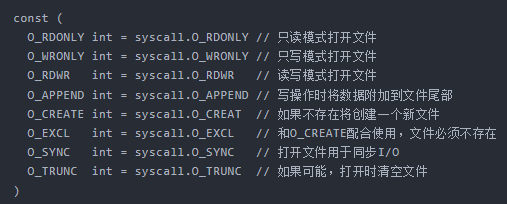
* recover 必须搭配defer 语句使用，并且recover()函数必须在延迟函数内被调用执行才能正常工作。
* defer一定要在可能引发panic的语句之前定义。

# 文件操作

## FileInfo接口



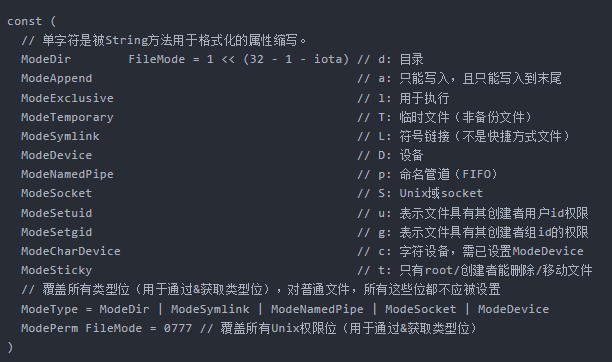
### 模式位



### 文件权限

文件有三种基本权限：r 读权限（read）、w写权限（write）、x执行权限（execute）。文件权限说明如图所示。



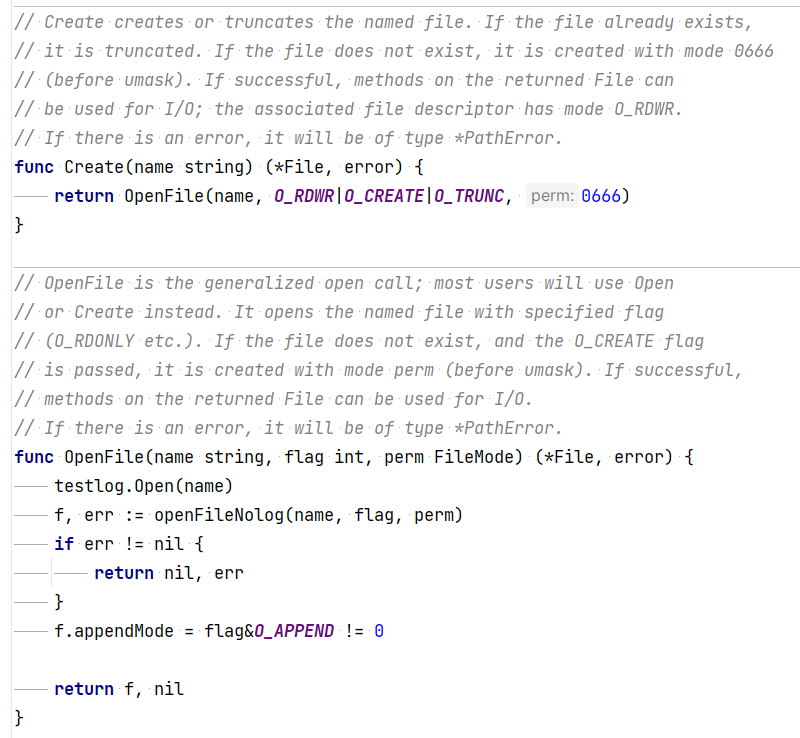


## 创建目录



## 创建文件

该函数本质上是在调用os.OpenFile()函数。





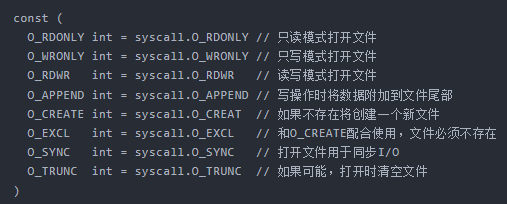
## 打开和关闭文件

打开文件，让当前的程序和指定的文件建立了一个链接。os.Open()函数本质上是在调用os.OpenFile()函数



第一个参数: 打开的路径

第二个参数: 打开的模式

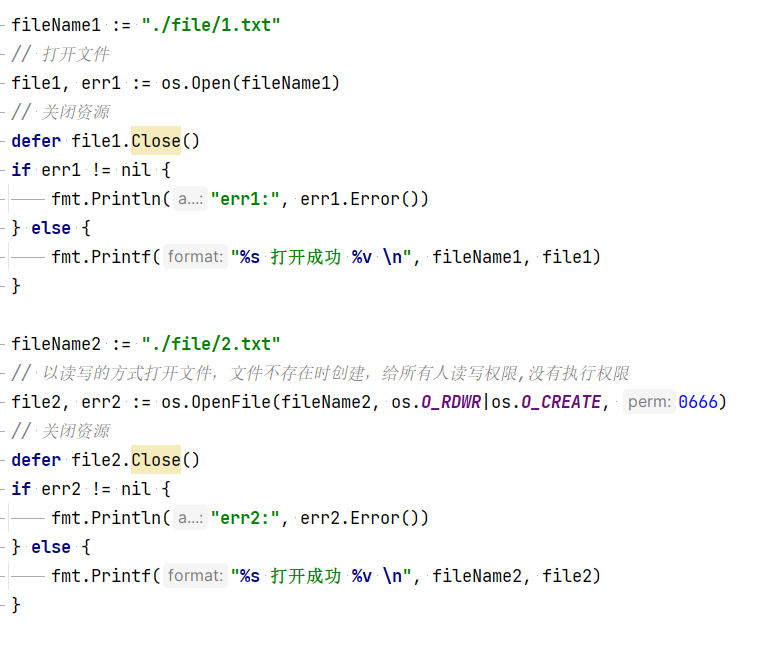


第三个参数: 指定权限

如:



示例：

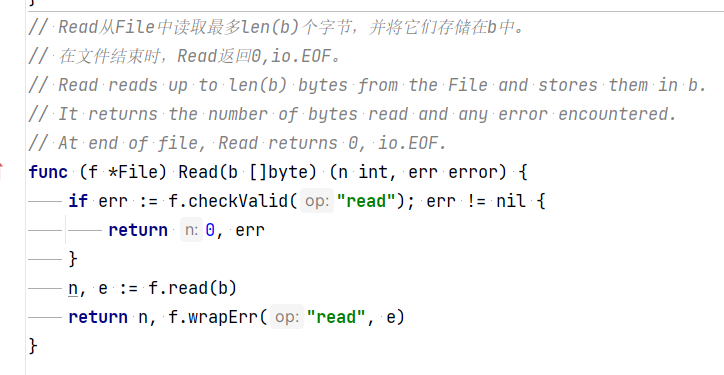


## 删除文件/目录

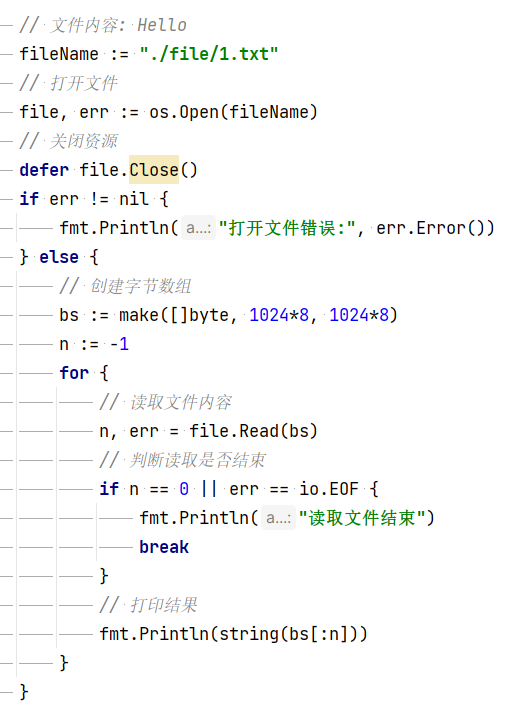


## 读取文件

从文件中开始读取数据，返回值n是实际读取的字节数。如果读取到文件末尾，n为0，err为EOF（end of file）



示例:



## 写入文件

写入字节数组，返回写入字节大小和一个error



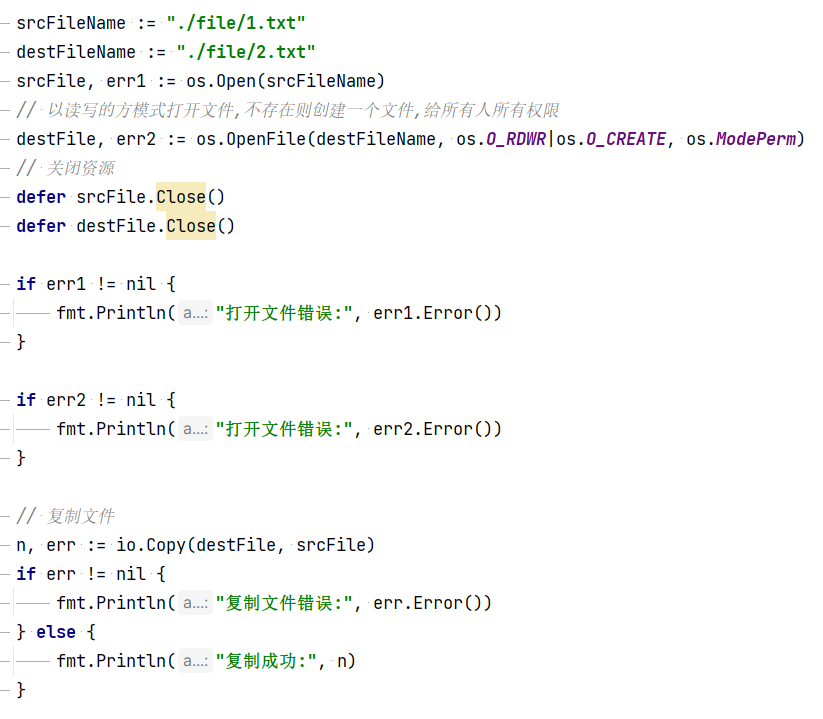
写入字符串，返回写入字节大小和一个error



示例:



## 复制文件



# JSON

## 结构体转json

转json，但是无缩进

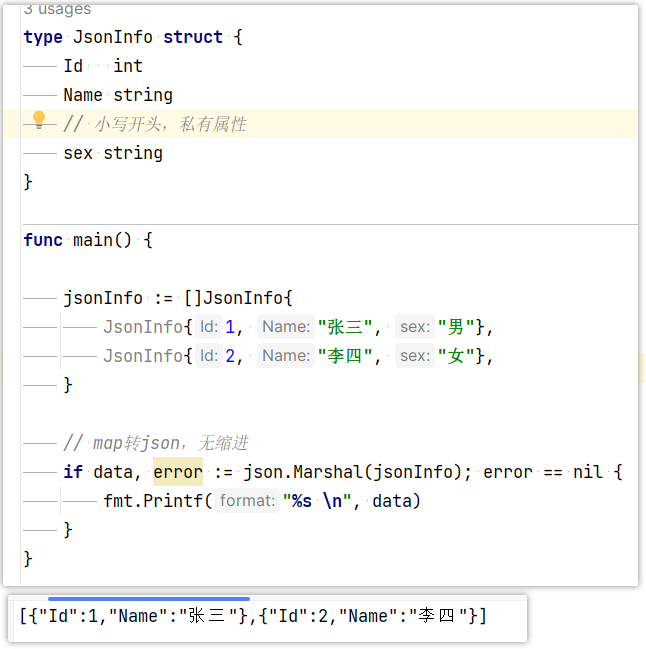


转json，有缩进，第二个参数是每一行的前缀，第二个参数是每行的缩进方式

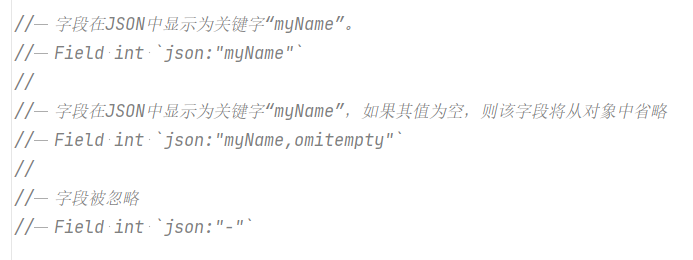




Go中没有私有、公有关键字， 都是通过大小写来实现(大写开头的为公有，小写开头的为私有)。json包是通过反射机制来实现编解码的，因此结构体必须对指定字段设置大写开头，没有大写开头的字段不会被json包解析。



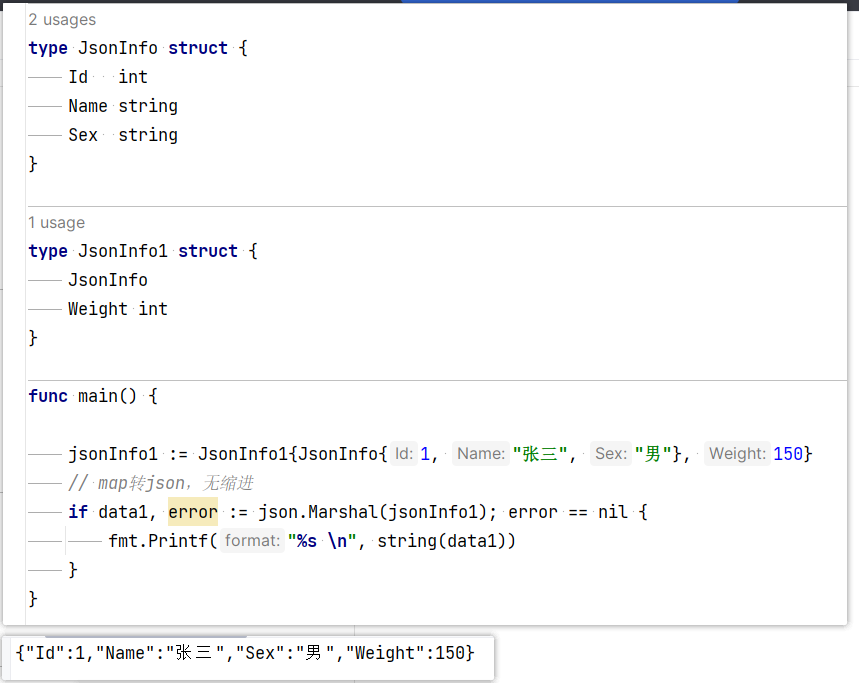
json包在解析结构体时，如果遇到key为JSON的字段标签，则会按照一定规则解析该标签





匿名字段

json包在解析匿名字段时，会将匿名字段的字段当成该结构体的字段处理



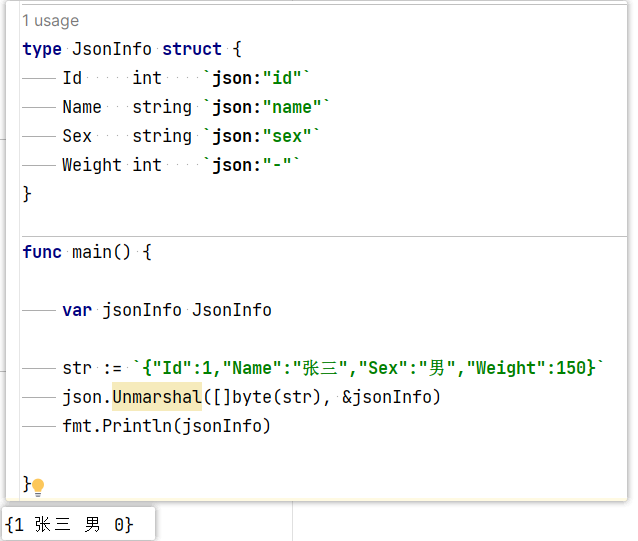
Marshal函数只有在转换成功的时候才会返回数据，在转换的过程中需要注意如下几点。

* JSON对象只支持string作为key，所以要编码一个map，必须是map[string]T这种类型(T是Go语言中任意的类型)。
* Channel, complex和function是不能被编码成JSON的。
* 指针在编码的时候会输出指针指向的内容，而空指针会输出null。

## Json转结构体

将data的JSON转换成v





数据库操作

## 安装模块

安装mysql

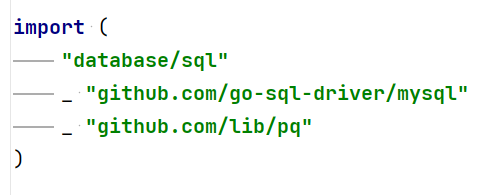
在项目路径下输入：go get -u github.com/go-sql-driver/mysql

如果无法对外网访问可以使用代理网站:

输入go env -w GOPROXY=https://goproxy.cn,direct

安装postgresql：go get github.com/lib/pq

## 导入包



匿名导包——只导入包但是不使用包内的类型和数据，使用匿名导包的方式(在包路径前添加下划线 “\_” )

## 连接数据库



* driverName使用的驱动名。这个名字其实就是数据库驱动注册到 database/sql 时所使用的名字.
* dataSourceName：数据库连接信息，这个连接包含了数据库的用户名，密码，数据库主机以及需要连接的数据库名等信息。



sql.Open()返回的sql.DB对象是协程并发安全的。sql.DB 通过数据库驱动为开发者提供管理底层数据库连接的打开和关闭操作。sql.DB 帮助开发者管理数据库连接池。正在使用的连接被标记为繁忙，用完后回到连接池等待下次使用。所以，如果开发者没有把连接释放回连接池，会导致过多连接使系统资源耗尽。

sql.DB的设计就是用来作为长连接（一次链接多次数据交互）使用。不宜频繁开关。比较好的做法是，为每个不同的datastore建一个DB对象，保持这些对象打开。如果需要短连接（一次链接一次数据交互），就把DB作为参数传入function，而不要在function中开关。

## 增删改数据

直接调用DB对象的Exec()方法执行sql

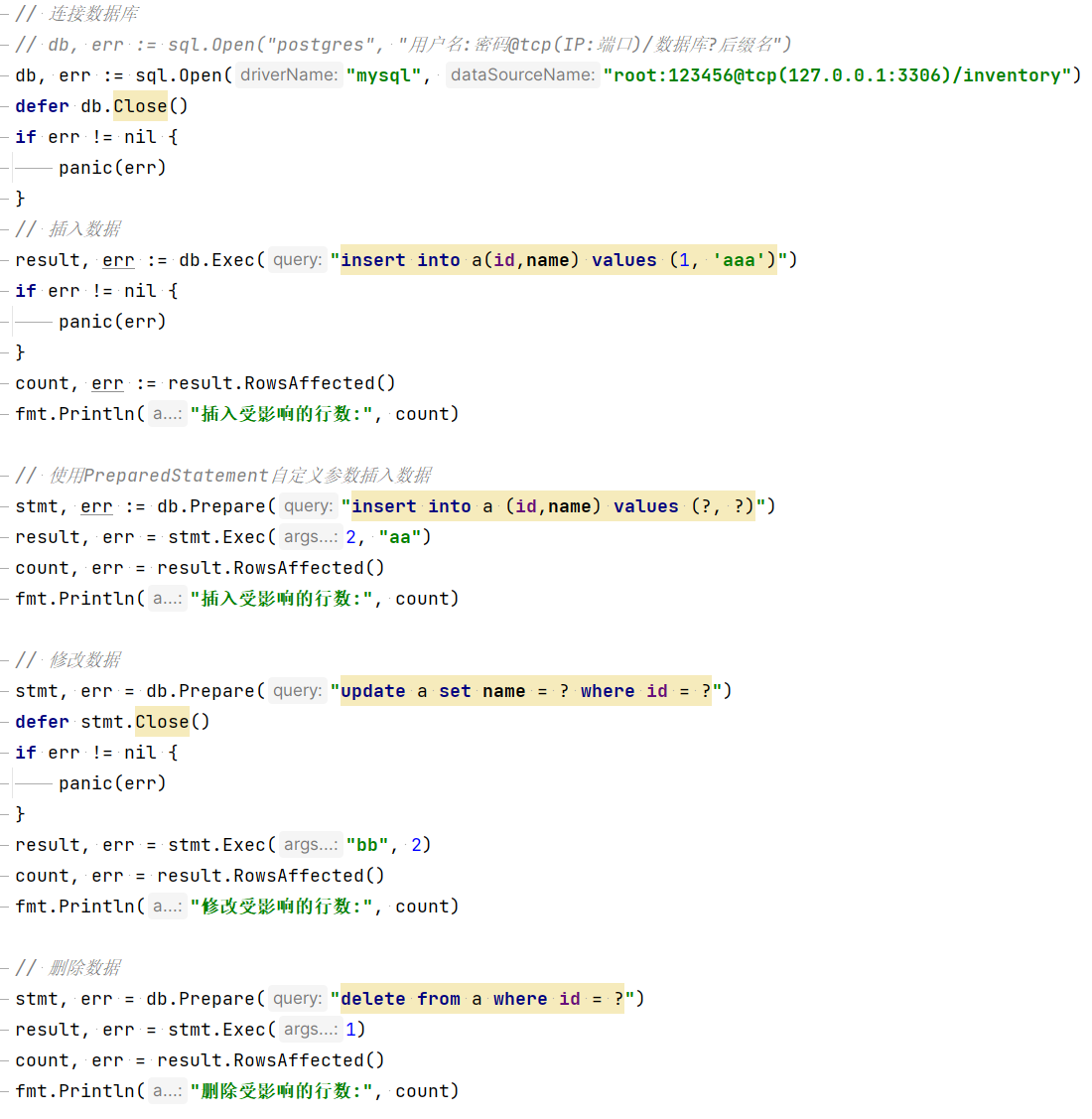


通过返回的结果Result可以进一步查询本次插入数据影响的行数（RowsAffected）和最后插入的ID(如果数据库支持查询最后插入ID)。

预编译语句(PreparedStatement)提供了诸多好处,，PreparedStatement 可以实现自定义参数的查询，通常来说，比手动拼接字符串 SQL 语句高效；PreparedStatement 还可以防止SQL注入攻击，在开发中尽量使用它。







## 查询数据

调用 db.Query()方法执行SQL语句，此方法返回一个Rows作为查询结果

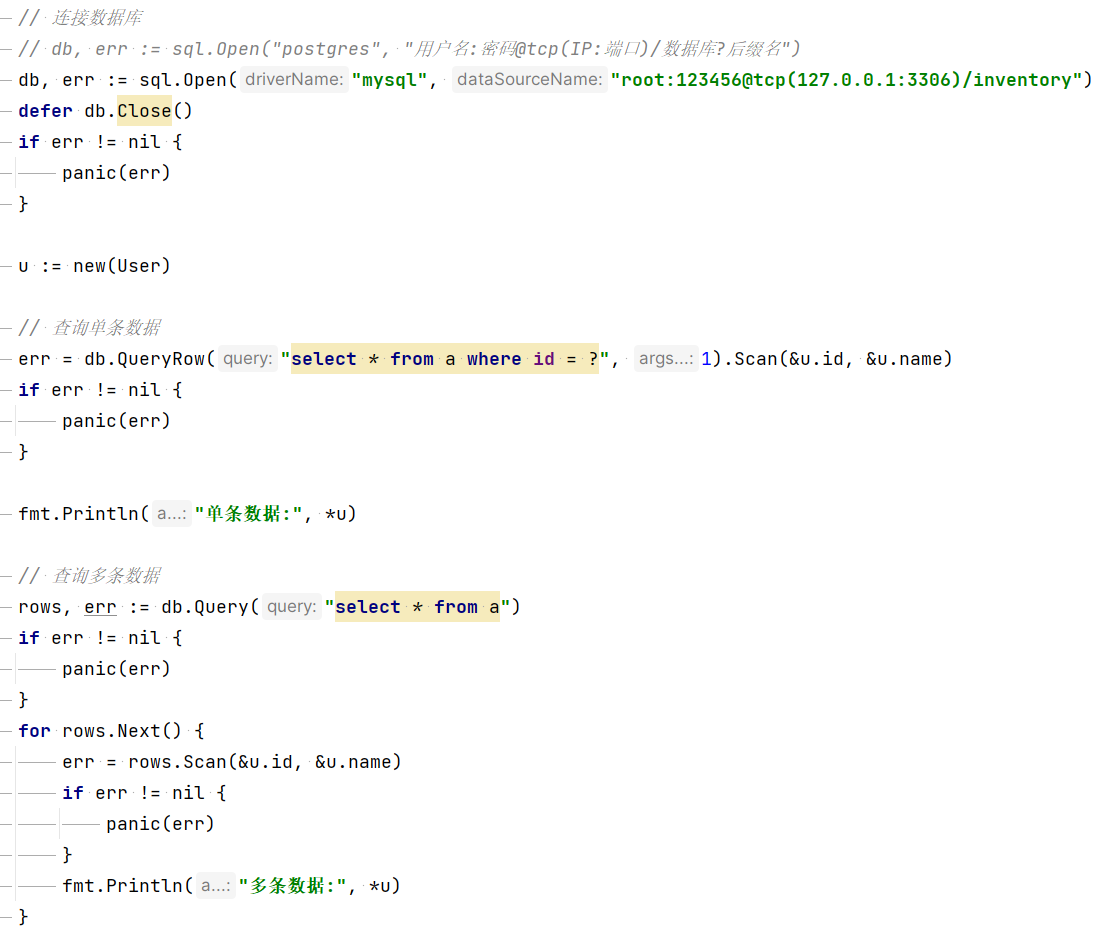


将rows.Next()方法的返回值作为for循环的条件，迭代查询数据



在循环中，通过 rows.Scan()方法读取每一行数据





并发编程

## 协程

协程（Coroutine），最初在1963年被提出。又称为微线程。是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。如图所示。

协程是编译器级的，进程和线程是操作系统级的。协程不被操作系统内核管理，而完全是由程序控制。因此没有线程切换的开销。和多线程比，线程数量越多，协程的性能优势就越明显。协程的最大优势在于其轻量级，可以轻松创建上万个而不会导致系统资源衰竭。

### Go语言中的协程

Go语言中的协程叫做Goroutine，Goroutine由Go程序运行时（runtime）调度和管理，Go程序会智能地将Goroutine中的任务合理地分配给每个CPU。创建Goroutine的成本很小。每个Goroutine的堆栈只有几kb，且堆栈可以根据应用程序的需要增长和收缩。

### Coroutine和Goroutine

Goroutine可能并行执行；但是Coroutine只能顺序执行；Goroutine可在多线程环境产生，Coroutine只能发生在单线程，Coroutine程序需要主动交出控制权，系统才能获得控制权并将控制权交给其他Coroutine。

Coroutine的运行机制属于协作式任务处理，应用程序在不使用CPU时，需要主动交出CPU使用权。如果开发者无意间让应用程序长时间占用CPU，操作系统也无能为力，计算机很容易失去响应或者死机。

Goroutine属于抢占式任务处理，和现有的多线程和多进程任务处理非常类似。应用程序对CPU的控制最终需要有操作系统来管理，如果操作系统如何发现一个应用程序常时间占用CPU，那么用户有权终止这个任务。

### Goroutine

一个 goroutine 必定对应一个函数，可以创建多个 goroutine 去执行相同的函数。

在 go中使用 goroutine 很方便，在调用函数时在前面加上go关键字，就可以为一个函数创建一个 goroutine 。

Go程序的执行过程是：创建和启动主Goroutine，初始化操作，执行main()函数，当main()函数结束，主Goroutine随之结束，程序结束。

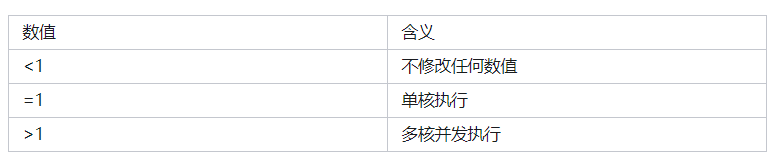


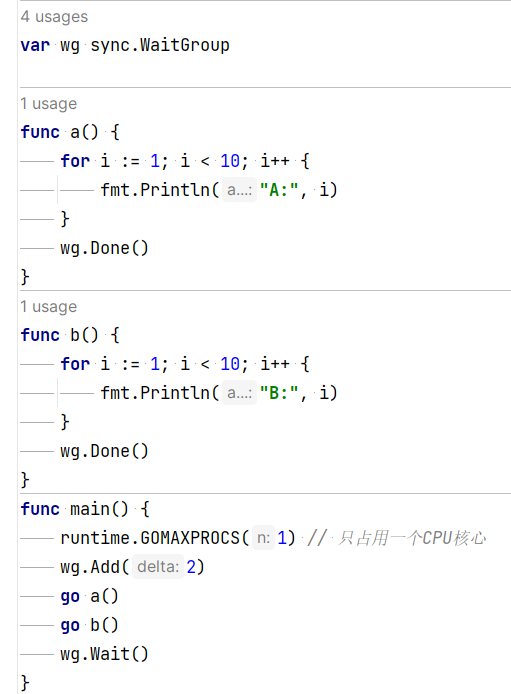
被启动的Goroutine叫做子Goroutine。如果main()的Goroutine终止了，程序将被终止，而其他Goroutine将不再运行。换句话说，所有Goroutine在main()函数结束时会一同结束。如上例所示。如果main()的Goroutine比子Goroutine先终止，运行的结果就不会打印Hello hello。

### GOMAXPROCS

Go语言中可以通过runtime.GOMAXPROCS()函数设置当前程序并发时占用的CPU逻辑核心数。

Go1.5版本之前，默认使用的是单核心执行。Go1.5版本之后，默认使用全部的CPU逻辑核心数。





## Channel

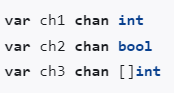
Channels即Go的通道，是协程之间的通信机制。一个channel是一条通信管道，它可以让一个协程通过它给另一个协程发送数据。每个channel都需要指定数据类型，即channel可发送数据的类型。如果使用channel发送int类型数据，可以写成chan int。数据发送的方式如同水在管道中的流动。

传统的线程之间可以通过共享内存进行数据交互，不同的线程之间对共享内存的同步问题需要使用锁来解决，这样会导致性能低下。Go语言中提倡使用channel的方式代替共享内存。换言之，Go语言主张通过数据传递来实现共享内存，而不是通过共享内存来实现消息传递。

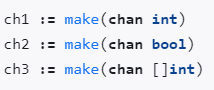
### channel创建方式

channel 是一种类型，一种引用类型，声明通道类型的格式如下：

var 变量 chan 元素类型



通道是引用类型，通道类型的空值是 nil。因为是引用类型所以声明通道后需要使用 make 函数进行初始化后才能使用。



### channel 操作

通道有 发送、接收和关闭 三种操作。

发送和接收都使用 <- 符号。

* 发送

将一个值发送到通道中



* 接收

从一个通道中接收值



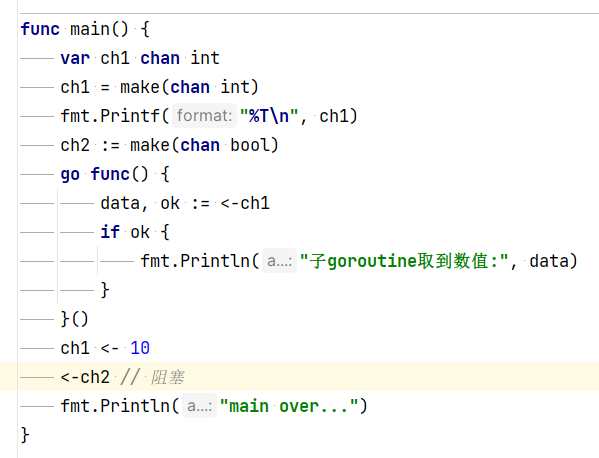
* 关闭

我们通常调用内置的 close 函数来关闭通道



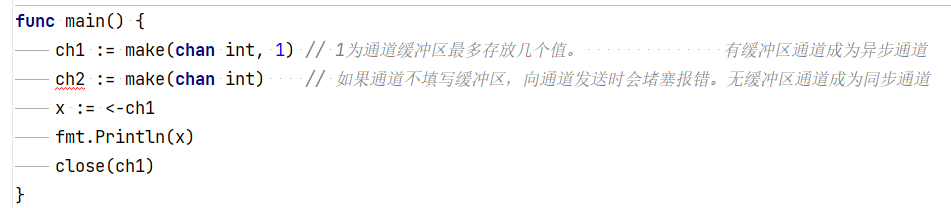
### 阻塞

channel默认是阻塞的。当数据被发送到channel时会发生阻塞，直到有其他Goroutine从该channel中读取数据。当从channel读取数据时，读取也会被阻塞，直到其他Goroutine将数据写入该channel。这些channel的特性是帮助Goroutines有效地通信，而不需要使用其他语言中的显式锁或条件变量。



### 缓冲channel

默认创建的都是非缓冲channel，读写都是即时阻塞。缓冲channel自带一块缓冲区，可以暂时存储数据，如果缓冲区满了，就会发生阻塞。



关于关闭通道需要注意的是：只有在通知接收方 goroutine 所有的数据都发送完毕的时候才需要关闭通道。通道是可以被垃圾回收机制回收的，他和关闭文件是不一样的，在操作结束之后关闭文件是必须要做的，但关闭通道不是必须的。

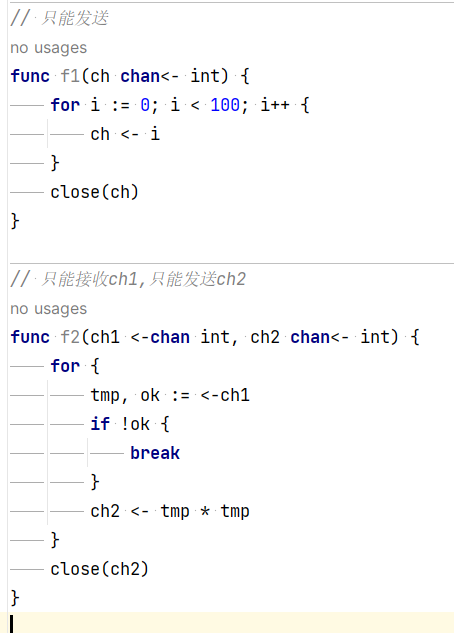
关于关闭通道后有以下特点：

1. 对一个关闭的通道再发送值就会导致panic。
2. 对一个关闭的通道进行接收会一直获取值直到通道为空。
3. 对一个关闭的并且没有值的通道执行接收操作会得到对应类型的零值。
4. 关闭一个已经关闭的通道会导致panic。



### 单向通道

限制通道在函数中只能发送或只能接收。



* chan<- int是一个只写单向通道（只能对其写入int类型值），可以对其执行发送操作但是不能执行接收操作；
* <-chan int是一个只读单向通道（只能从其读取int类型值），可以对其执行接收操作但是不能执行发送操作。

## select多路复用

在某些场景下我们需要同时从多个通道接收数据。

select 的使用类似于switch语句，它有一系列case分支和一个默认的分支。每个case会对应一个通道的通信（接收或发送）过程。select 会一直等待，直到某个 case 的通信操作完成时，就会执行 case 分支对应的语句。具体格式如下：

**select**{

**case** **<-**ch1:

**...**

**case** data **:=** **<-**ch2:

**...**

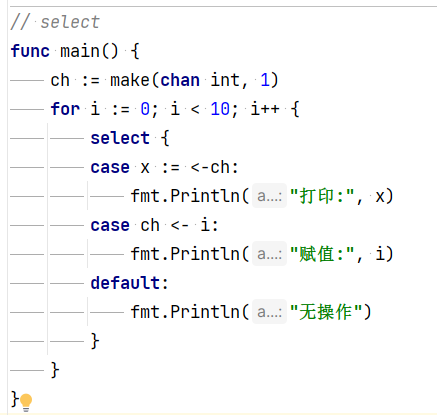
**case** ch3**<-**data:

**...**

**default**:

默认操作

}

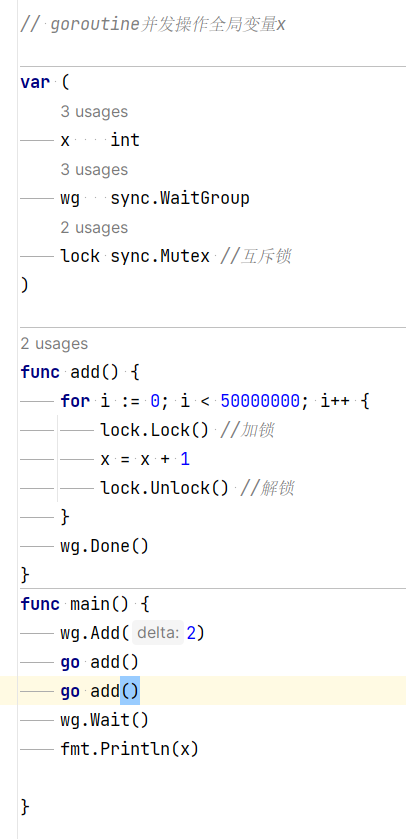


## 并发安全和锁

有时候在Go代码中可能会有多个 goroutine同时操作一个资源，这种情况就发生竞态问题（数据竞态）

### 互斥锁

互斥锁是一种常用的控制共享资源访问的方法，它能够保证同时只有一个goroutine可以访问共享资源。Go语言中使用 sync 包的 Mutex 类型来实现互斥锁。

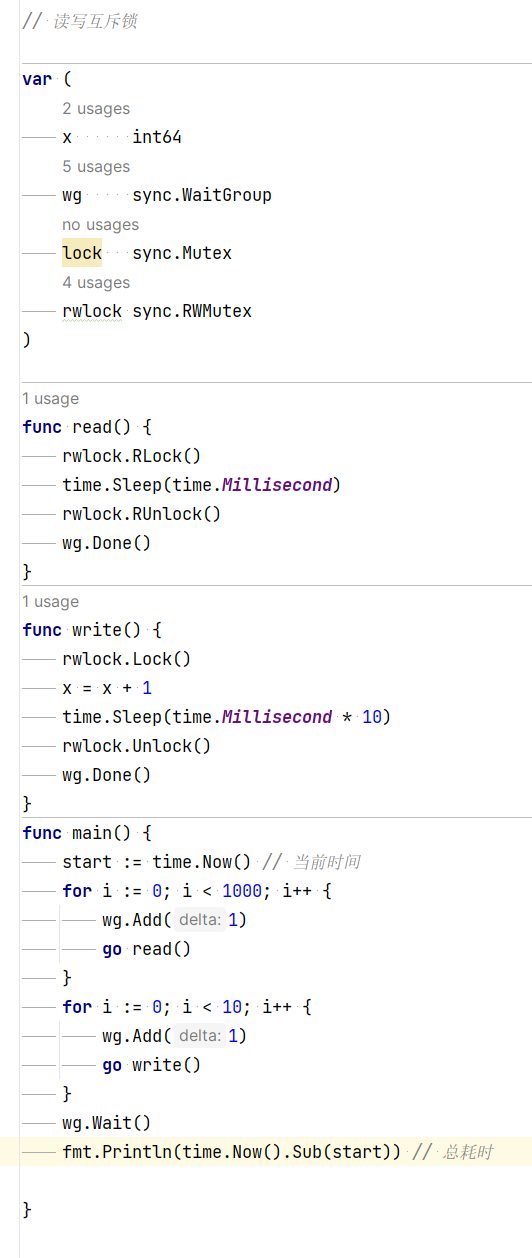


使用互斥锁能够保证同一时间有且只有一个goroutine进入临界区，其他的goroutine则在等待锁；当互斥锁释放后，等待的goroutine才可以获取锁进入临界区，多个goroutine同时等待一个锁时，唤醒的策略是随机的。

### 读写互斥锁

互斥锁是完全互斥的，但是有很多实际的场景下是读多写少的，当我们并发的去读取一个资源不涉及资源修改的时候是没有必要加锁的，这种场景下使用读写锁是更好的一种选择。读写锁在Go语言中使用sync包中的RWMutex类型。

读写锁分为两种：读锁和写锁。当一个goroutine获取读锁之后，其他的goroutine如果是获取读锁会继续获得锁，如果是获取写锁就会等待；当一个goroutine获取写锁之后，其他的goroutine无论是获取读锁还是写锁都会等待。



读写锁非常适合读多写少的场景，如果读和写的操作差别不大，读写锁的优势就发挥不出来

如果读写耗时次数差别大用读写互斥锁，如果读写次数相差无几用互斥锁