Go语言编程

<https://www.shiyanlou.com/courses/11>

**第一节 Go 语言介绍**

**一 知识点**

* Go 语言的起源和特性
* Go 语言的基本语法与编译运行
* GO 语言的 goroutine 和 channel

**二 语言特性**

* Go 语言将静态语言的安全性和高效性与动态语言的易开发性进行有机结合；
* Go 语言是一门类型安全和内存安全的编程语言，虽然 Go 语言中仍有指针的存在，但并不允许进行指针运算；
* Go 语言对网络通信、并发和并行编程有着极佳的支持；
* 设计者通过 goroutine 这种轻量级线程的概念来实现这个目标，然后通过 channel 来实现各个 goroutine 之间的通信；
* Go 语言中另一个非常重要的特性就是它的构建速度（编译和链接到机器代码的速度），一般情况下构建一个程序的时间只需要数百毫秒到几秒；
* 内存问题，尽管 Go 语言像其它静态语言一样执行本地代码，但它依旧运行在某种意义上的虚拟机，以此来实现高效快速的垃圾回收；

**示例一 hello world**



保存为 hello.go

注释：//表示单行注释，到行尾结束，/\*...\*/表示多行注释。

所有的Go语言代码只能放置于一个包中，每一个Go程序都必须包含一个main包以及一个main()函数，main()函数作为整个程序的入口。

我们的第一个Go程序源代码中，第一行package main 表明当前源文件属于main包，import后面是导入的包，只有需要用到的包才需要导入，导入的包没有使用，则会导致编译失败。

Go语言语句是使用分号进行分隔的，但是在一般情况下我们不用手动添加，编译器会自动完成这些工作，除非需要在一行中写多条语句。

Go语言的函数和方法都以关键字func 进行定义。

以上代码中的main函数的第一行使用了 := 操作符，该符号声明并初始化了一个字符串变量。Go语言虽然是一门静态强类型的语言，但是在使用 := 操作符时，Go语言会根据符号右边的值推导出符号左边变量的类型。

这儿的例子中，'World'是一个字符串，所以Go语言将变量target创建为一个字符串类型，并且进行赋值。

代码中os.Args是一个字符串切片(slice)，切片内容是传递给Go程序的参数。切片是一个可以动态增长的数组，可以通过len()内置函数计算切片的长度，通过slice[n]的方式访问切片中的第n个元素，而slice[n:]则返回从第n个元素到最后一个元素的切片，像极了Python中的切片。

编译运行：

如果需要快速执行Go程序，使用以下命令：

$ go run hello.go

Hello World

go run hello.go命令直接让Go源文件运行，不会生成任何可执行文件，这可以快速查看源代码的执行效果。

一般情况下，我们将Go源代码编译链接成可执行文件，以方便随时运行。可以使用以下命令进行：

$ go build hello.go

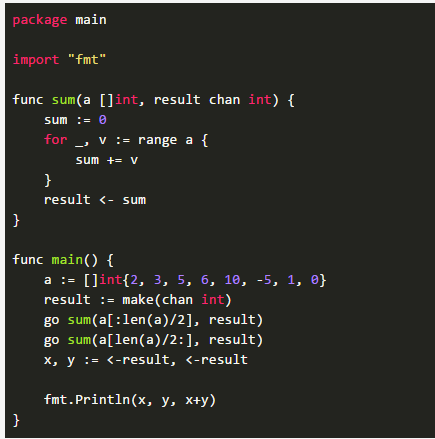
$ ./hello Aiden

Hello Aiden

以上的命令中，go build hello.go 编译和链接hello.go代码，并且生成了可执行文件hello。接着我们运行hello程序，并传递了一个参数给它，可以看到程序正常运行并且如期输出了结果。

**示例二 goroutine和channel**

分组求和的例子



goroutine是Go语言并行设计的核心。

goroutine是一种比线程更轻量的实现，十几个goroutine可能在底层就是几个线程。要使用goroutine只需要简单的在需要执行的函数前添加go关键字即可。当执行goroutine时候，go语言立即返回，接着执行剩余的代码，goroutine不阻塞主线程。channel就像一个管道，但是可以双向传输数据，通过它我们可以接收和发送数据，假如result 是一个channel那么：result <- value 是将数据发送到result, 而key <- result就是从result中接收一个数据，就如以上的代码所示，值得注意的地方是channel只能通过Go语言内建的函数make(chan type)创建，其中type指明了该channel能传递的数据类型。

下面我们来说说以上的代码。在main函数中，我们声明了一个int类型的切片a，然后通过内置函数make创建了一个能接收和发送int类型的channel。然后通过关键字go执行了两个goroutine，这两个goroutine的功能是分别计算切片a前半部分和后半部分的和。在这里main函数碰到go关键字，派发goroutine执行相应的函数后，立即返回执行剩余的代码，不会等待goroutine的返回。sum函数中，计算切片的和，然后将结果发送到channel。接下来main函数，从channel中获取结果，在这里，main函数会阻塞至直到能从channel result中接收到数据，最后我们打印出了结果。

**第二节 Go 语言基础和数值布尔类型**

**一 知识点**

* Go语言中的数值类型和变量类型
* Go语言中的标识符和关键字

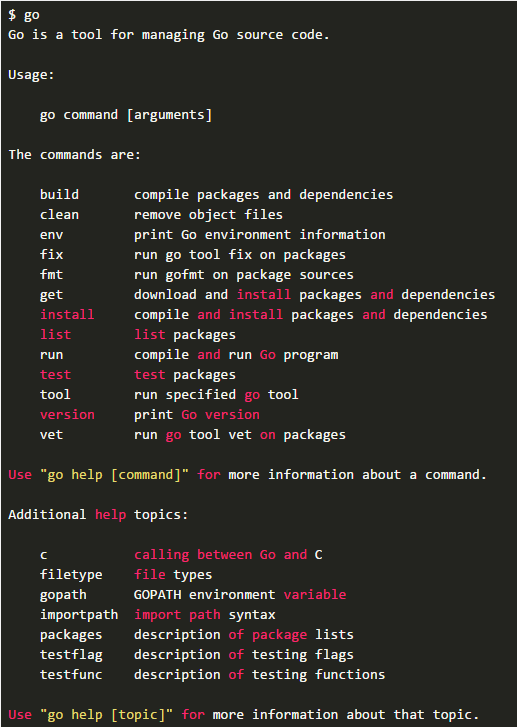
**二 环境变量和常用命令**

**2.1 Go语言环境变量**

Go语言开发环境依赖一些操作系统环境变量，常用的环境变量如下：

* $GOROOT 表示 Go 在你的电脑上的安装位置，它的值一般都是 $HOME/go，当然，你也可以安装在别的地方。
* $GOARCH 表示目标机器的处理器架构，它的值可以是 386，amd64 或 arm。
* $GOOS 表示目标机器的操作系统，它的值可以是 darwin，freebsd，linux 或 windows
* $GOBIN 表示编译器和链接器的安装位置，默认是 $GOROOT/bin，如果你使用的是 Go 1.0.3 及以后的版本，一般情况下你可以将它的值设置为空，Go 将会使用前面提到的默认值。
* $GOPATH 表示工作路径，允许包含多个目录。当有多个目录时，请注意分隔符，多个目录的时候Windows是分号，Linux系统是冒号，当有多个GOPATH时，默认会将go get命令的内容放在第一个目录下。$GOPATH 目录约定有三个子目录：
* src 存放源代码（比如：.go .c .h .s等）
* pkg 编译后生成的文件（比如：.a）
* bin 编译后生成的可执行文件（为了方便，可以把此目录加入到 \$PATH 变量中，如果有多个gopath，那么使用${GOPATH//://bin:}/bin添加所有的bin目录）很多Go命令都依赖于此变量，例如go get命令会将获取到的包放到GOPATH中。

**2.2 Go语言的命令行工具**



简单介绍下常用的命令。

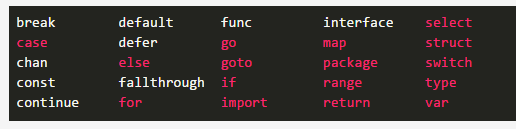
* go build 主要用于测试编译。在包的编译过程中，若有必要，会同时编译与之相关联的包。
* 如果是普通包，当你执行go build之后，它不会产生任何文件。如果你需要在\$GOPATH/pkg下生成相应的文件，那就得执行go install了
* 如果是main包，当你执行go build之后，它就会在当前目录下生成一个可执行文件。如果你需要在\$GOPATH/bin下生成相应的文件，需要执行go install。
* go build会忽略目录下以“\_”或“.”开头的go文件。
* go fmt
* 有过C/C++经验的读者会知道,一些人经常为代码采取K&R风格还是ANSI风格而争论不休。在go中，代码则有标准的风格。由于之前已经有的一些习惯或其它的原因我们常将代码写成ANSI风格或者其它更合适自己的格式，这将为人们在阅读别人的代码时添加不必要的负担，所以Go语言强制了代码格式（比如左大括号必须放在行尾），不按照此格式的代码将不能编译通过，为了减少浪费在排版上的时间，go工具集中提供了一个go fmt命令 它可以帮你格式化你写好的代码文件，使你写代码的时候不需要关心格式，你只需要在写完之后执行go fmt <文件名>.go，你的代码就被修改成了标准格式。
* go install
* 这个命令在内部实际上分成了两步操作：第一步是生成结果文件(可执行文件或者.a包)，第二步会把编译好的结果移到\$GOPATH/pkg或者\$GOPATH/bin。
* go test
* 执行这个命令，会自动读取源码目录下面名为\*\_test.go的文件，生成并运行测试用的可执行文件。详情请参考go help testflag。
* go run 编译并运行Go语言源代码，这对于我们快速运行测试代码非常方便。

Go语言还有其他一些命令参数，如go env， godoc，详细信息可以通过go help进行查看。

**三 Go语言基础**

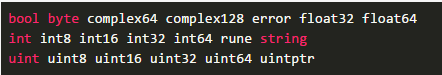
**3.1 标示符和关键字**

Go语言的标示符（变量名，函数名）是一个非空的字母或数字串，其中第一个字符必须是字母，该字符也不能是关键字的名字。Go语言一共有25个关键字，如下：



Go语言同样预定义了许多标示符：

* 类型：



* 常量



* 零值



* 函数

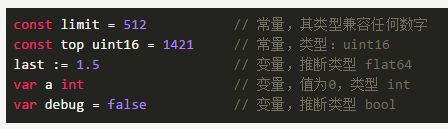


空标示符"\_"是一个占位符，它用于在赋值操作的时候将某个值赋值给空标示符号，从而达到丢弃该值的目的。空标示符不是一个新的变量，因此将它用于:=操作符号的时候，必须同时为至少另一个值赋值。下面有几个例子：

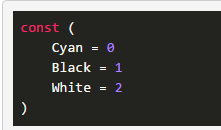


**3.2 常量和变量**

常量使用关键字const声明，变量可以使用关键字var声明，也可以通过使用快捷变量声明语法:=。Go语言可以自动推断出所声明变量的类型。对于没有显式初始化的变量，Go语言总是将零值赋值给该变量。在Go语言中，声明变量的时候类型名总是在变量名的后面。下面有几个例子：



当需要设置多个常量的时候，不必重复使用const关键字，可以使用以下语法(var声明同样可以使用):



**四 数值类型和布尔类型**

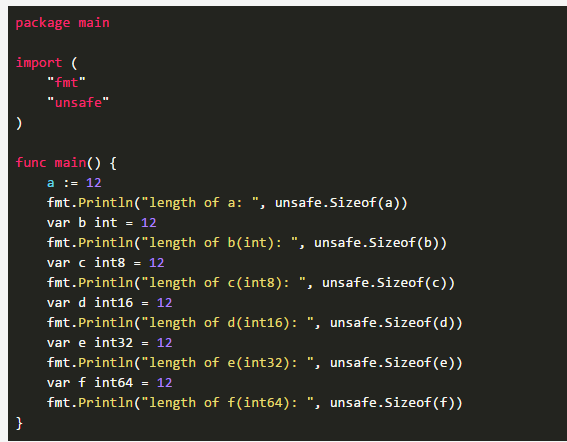
**4.1 整型**

Go语言提供了11种整型



在C语言中我们可以通过sizeof操作符查看类型的字节长度，在Go语言中可以通过unsafe.Sizeof函数进行。

例子：



**4.2 浮点类型**

Go语言提供了两种浮点类型和两种复数类型, 具体如下：



**4.3 布尔类型**

Go语言提供了内置的布尔值true和false。Go语言支持标准的逻辑和比较操作，这些操作的结果都是布尔值。值得注意的地方是可以通过!b的方式反转变量b的真假。

**第三节 Go 语言字符串**

**一 知识点**

* Go 中字符串处理的基本方式
* String 包 和 strconv 包

**二 字符串**

Go语言中的字符串是 UTF-8 字符的一个序列（当字符为 ASCII 码时则占用 1 个字节，其它字符根据需要占用 2-4 个字节）。UTF-8 是被广泛使用的编码格式，是文本文件的标准编码，其它包括 XML 和 JSON 在内，也都使用该编码。由于该编码对占用字节长度的不定性，Go 中的字符串也可能根据需要占用 1 至 4 个字节，这与其它语言如 C++、Java 或者 Python 不同。Go 这样做的好处是不仅减少了内存和硬盘空间占用，同时也不用像其它语言那样需要对使用 UTF-8 字符集的文本进行编码和解码。

Go语言中字符串的可以使用双引号( " )或者反引号( ` )来创建。双引号用来创建可解析的字符串字面量，所谓可解析的是指字符串中的一些符号可以被格式化为其他内容，如"\n"在在输出时候会被格式化成换行符， 如果需要按照原始字符输出必须进行转义。而反引号创建的字符串原始是什么样，那输出还是什么，不需要进行任何转义。以下是几个例子:



部分转义字符

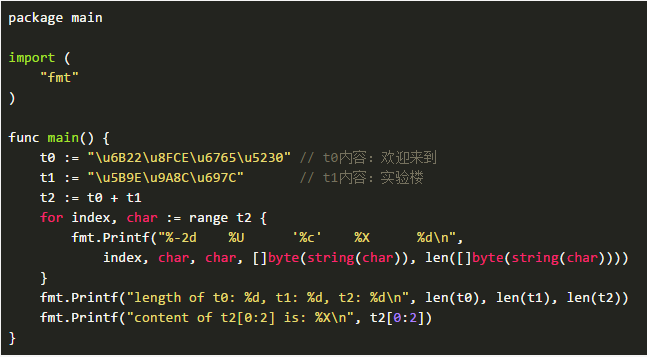


在Go语言中单个字符可以使用单引号( ' )来创建。之前的课程中，我们有学习过rune类型，它等同于unint32，在Go语言中，一个单一的字符可以用一个单一的rune来表示。这也是容易理解的，因为Go语言的字符串是UTF-8编码，其底层使用4个字节表示，也就是32 bit。

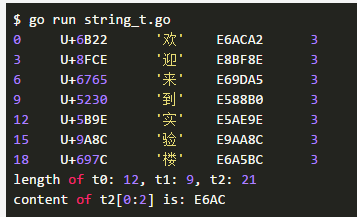
在Go语言中，字符串支持切片操作，但是需要注意的是如果字符串都是由ASCII字符组成，那可以随便使用切片进行操作，但是如果字符串中包含其他非ASCII字符，直接使用切片获取想要的单个字符时需要十分小心，因为对字符串直接使用切片时是通过字节进行索引的，但是非ASCII字符在内存中可能不是由一个字节组成。如果想对字符串中字符依次访问，可以使用range操作符。另外获取字符串的长度可能有两种含义，一种是指获取字符串的字节长度，一种是指获取字符串的字符数量。字符串支持以下操作：



例子



运行



说明：通过前面的课程我们知道通过\uhhhh的方式我们可以通过创建Unicod字符，在以上程序中，首先通过:=符号创建了变量t0，其值为\u6B22\u8FCE\u6765\u5230，是欢迎来到中文字符的unicode编码，然后以同样的方式创建了变量t1，其值为实验楼，然后通过+操作符将t0和t1拼接赋值给t2。然后我们通过range操作符号对unicode字符串t2中的每一个unicode字符依次操作，我们这里只是简单的打印出每个字符在t2中的位置，每个字符的unicode码值，每个字符的字面量，每个字符的十六进制值，以及每个字符的字节长度。这里我们使用fmt包种支持的格式指令，如果读者学习过C语言的话就一目了然。接着，我们通过len操作符计算出了每个字符串的字节长度。最后，我们使用切片访问了字符串t2的第0-1个字节，也就是前两个字节，其内容为E6AC。前面我们说到不能使用切片的方式访问非ASCII字符串中的字符，原因在这里一目了然。字符欢其底层使用了三个字节表示，内容是E6ACA2，如果只是简单的使用切片（只取切片中的一项)访问的是不能访问到整个字符的，因为字符的切片是通过字节数来索引的。

**三. 格式化字符串**

Go语言标准库中的fmt包提供了打印函数将数据以字符串形式输出到控制台，文件，其他满足io.Writer接口的值以及其他字符串。目前为止我们使用了fmt.Printf和fmt.Prinfln，对于前者的使用，就像C语言中的printf函数一样，我们可以提供一些格式化指令，让Go语言对输出的字符串进行格式化。同样的我们可以使用一些格式化修饰符，改变格式化指令的输出结果， 如左对齐等。常用的格式化指令如下：



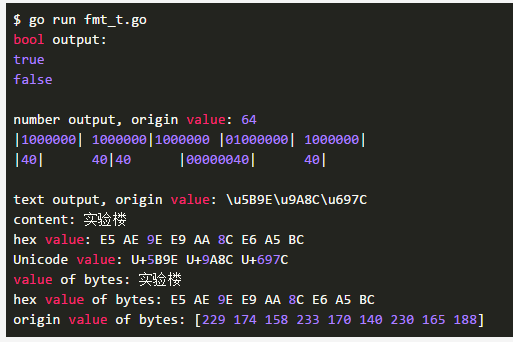
常用的格式化指令修饰符如下：

* 空白 如果输出的数字为负，则在其前面加上一个减号"-"。如果输出的是整数，则在前面加一个空格。使用%x或者%X格式化指令输出时，会在结果之间添加一个空格。例如fmt.Printf("% X", "实")输出E5 AE 9E
* #
* %#o 输出以0开始的八进制数据
* %#x 输出以0x开始的十六进制数据
* + 让格式化指令在数值前面输出+号或者-号，为字符串输出ASCII字符（非ASCII字符会被转义），为结构体输出其字段名
* - 让格式化指令将值向左对齐（默认值为像右对齐）
* 0 让格式指令以数字0而非空白进行填充

例子



结果如下



**四. 字符串处理相关的包**

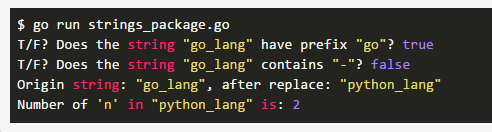
Go语言处理字符串的强大之处不仅限于对索引和切片的支持，很多官方包提供了大量的实用函数，可以对字符串进行很方便的操作。在这里我们简单的介绍几个常用的包。

**4.1 strings 包**

strings包提供了如查找字符串，分割字符串，判断前后缀，判断字符串包含，字符串替换，统计字符串出现的次数等常用操作，完整的方法列表可以参考官方包说明。下面我们通过一个小练习来感受下。使用vim创建文件strings\_package.go，输入以下源码：



结果



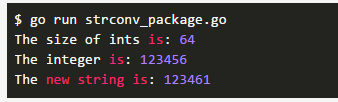
代码依然很简单，函数的功能就从函数名就可以看出。值得注意的地方是函数func Replace(s, old, new string, n int) string中的参数n指明了将字符串s中的前n个old字符串替换为new字符串，如果n = -1则提供所有匹配到的字符串。

**4.2 strconv 包**

strconv包提供了许多可以在字符串和其他类型的数据之间进行转换的函数。例如可以将数字转换为字符串，将数字样式的字符串转换为数值（将字符串"12345"转换int类型的整数）。我们还是直接通过例子学习，创建源文件strconv\_package.go，输入以下代码：



以上代码中，需要注意的地方是strconv.IntSize是一个常量，其值是int类型的所占的bit数，运行代码输出如下：



**第四节 Go 语言复合类型**

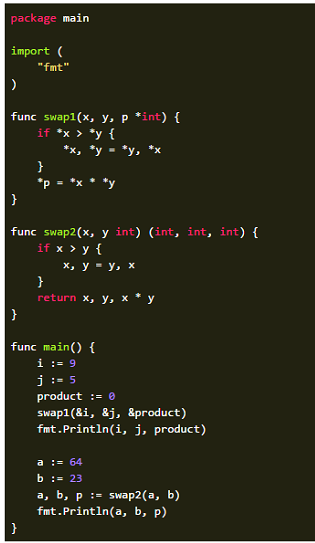
**一 知识点**

* Go语言中的 值 和 指针
* Go语言中的 数组 和 切片
* Go 语言中的映射

**二 值、指针和引用类型**

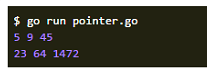
通常情况下Go语言中的变量持有相应的值。也就是说，我们可以将一个变量想象成它所持有的值来使用。其中有些例外，通道、函数、方法、映射、切片是引用变量，它们持有的都是引用，也即保存指针的变量。值在传递给函数或者方法的时候会被复制一次，对于布尔类型和数值类型来说这非常廉价，但是对于大型变量代价却非常大。而且复制传参的方式，修改值只是修改了副本，这能保证原始变量不被修改，但也一定程度上增加了修改原始值的麻烦。幸好在Go语言中有指针，使用指针时，我们每次传递给函数或者方法的只是变量的内存地址，这是非常廉价的。而且一个被指针指向的变量可以通过该指针来修改，这就很方便的在函数或者防止中通过指针修改原始变量。Go语言中的指针操作符也是使用&和\*操作符，其中&用于取地址，\*用于解引用，也就是获取指针指向的值。

使用VIM创建源文件pointer.go，输入以下源文件：



以上源码中，我们首先创建了swap1函数，其通过指针原地的交换值，同时swap2函数通过复制的方式交换了变量的值。

运行结果如下：



**三. 数组和切片**

**3.1 数组**

Go语言的数组是一个定长的序列，其中的元素类型相同。多维数组可以简单地使用自身为数组的元素来创建。数组的元素使用操作符号[ ]来索引，索引从0开始，到len(array)-1结束。数组使用以下语法创建：

* [length]Type
* [N]Type{value1, value2, ..., valueN}
* [...]Type{value1, value2, ..., valueN}

如果使用了...（省略符）操作符，Go语言会为我们自动计算数组的长度。在任何情况下，一个数组的长度都是固定的并且不可修改。数组的长度可以使用len()函数获得。由于数组的长度是固定的，因此数组的长度和容量都是一样的，因此对于数组而言cap()和len()函数返回值都是一样的。数组也可以使用和切片一样的语法进行切片，只是其结果为一个切片，而非数组。同样的，数组也可以使用range进行索引访问。

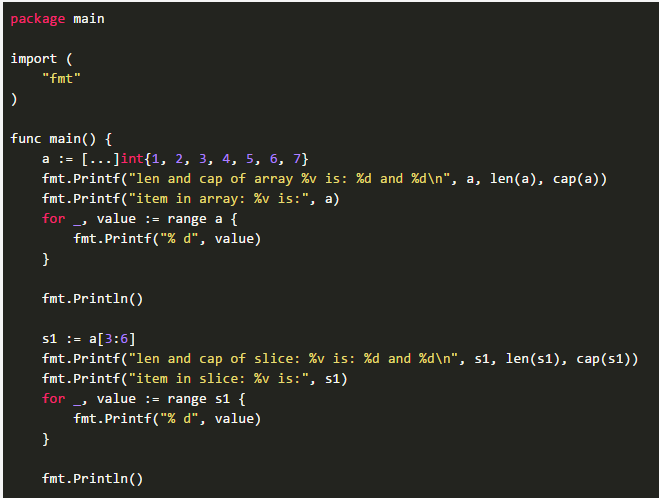
**3.2 切片**

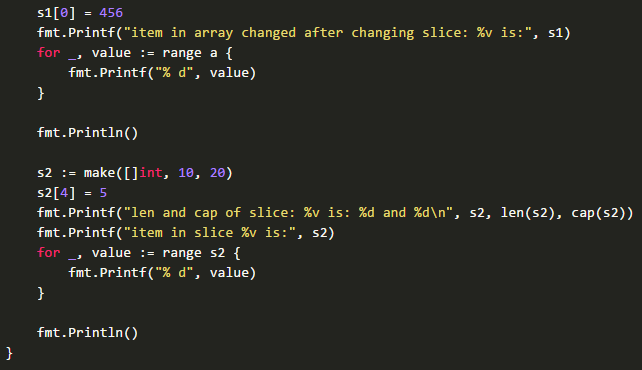
一般而言，Go语言的切片比数组更加灵活，强大而且方便。数组是按值传递的（即是传递的副本），而切片是引用类型，传递切片的成本非常小，而且是不定长的。而且数组是定长的，而切片可以调整长度。创建切片的语法如下：

* make([ ]Type, length, capacity)
* make([ ]Type, length)
* [ ]Type{}
* [ ]Type{value1, value2, ..., valueN}

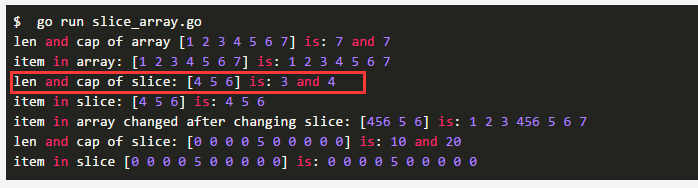
内置函数make()用于创建切片、映射和通道。当用于创建一个切片时，它会创建一个隐藏的初始化为零值的数组，然后返回一个引用该隐藏数组的切片。该隐藏的数组与Go语言中的所有数组一样，都是固定长度，如果使用第一种语法创建，那么其长度为切片的容量capacity；如果是第二种语法，那么其长度记为切片的长度length。一个切片的容量即为隐藏数组的长度，而其长度则为不超过该容量的任意值。另外可以通过内置的函数append()来增加切片的容量。切片可以支持以下操作：

我们练习下，使用VIM创建源文件slice\_array.go，输入以下代码：





以上代码中，我们首先创建了一个数组，数组的长度是由Go语言自动计算出的（省略号语法），然后通过切片操作从数组a中创建了切片s1，接着我们修改了该切片的第一个位置的数值，然后发现数组a中的值也发生了变化。最后我们通过make()函数创建了一个切片，该切片的长度和容量分别为10和20，还可以发现Go语言将未初始化的项自动赋予零值。运行代码输出如下：



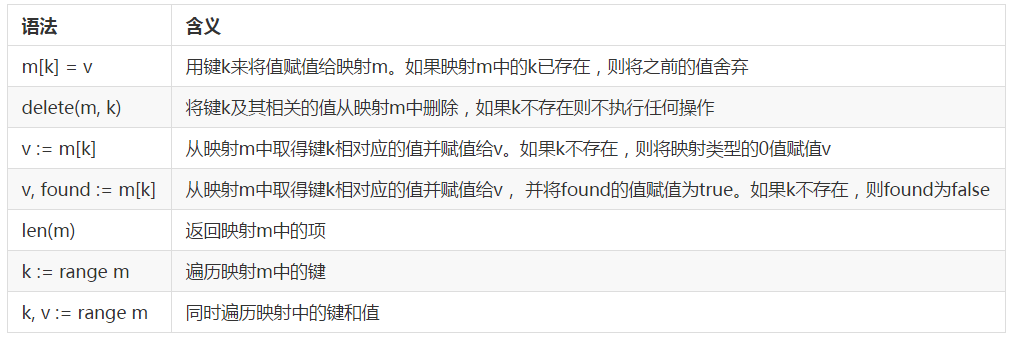
**四. 映射（map）**

Go语言中的映射（map）是一种内置的数据结构，保存键=值对的无序集合，它的容量只受到机器内存的限制，类似于Python中的字典。在一个映射中所有的键都是唯一的而且必须是支持==和!=操作符的类型，大部分Go语言的基本类型都可以作为映射的键，但是切片、不能用于比较的数组、结构体（这些类型的成员或者字段不支持==和!=操作）或者基于这些的自定义类型不能作为键。但是任意类型都可以作为值。映射是引用类型，所以传递非常廉价。

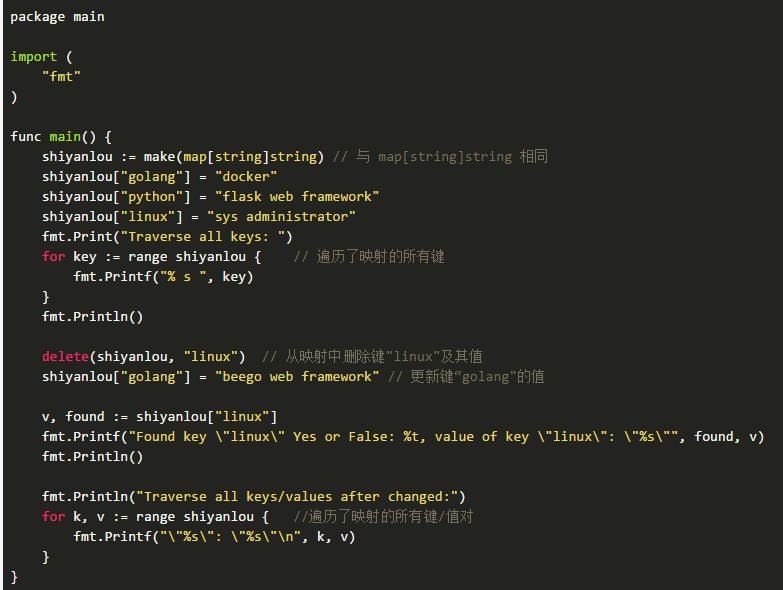
Go语言中的映射可以用以下用法创建：

* make(map[KeyType]VauleType, initialCapacity)
* make(map[KeyType]ValueType)
* map[KeyType]ValueType{ }
* map[KeyType]ValueType{key1: value1, key2: value2, ..., keyN: valueN}

内置的函数make()可以用来创建切片、映射和channel（通道）。当用make()来创建一个映射时候，实际上是得到一个空映射，如果指定了容量(initialCapacity)就会预先申请足够的内存，并随着加入的项越来越多，映射会字段扩容。映射支持以下操作：

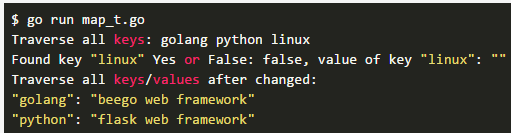


接下来我们练习下。使用VIM创建源文件map\_t.go输入以下代码:



以上代码中，我们首先创建了一个映射，然后赋值了3个键/值对，然后我们遍历了映射中的所有键，使用delete()函数删除了映射中的一个键，然后再次遍历打印了映射。映射的操作都非常简单，多多练习即可。

运行结果如下：



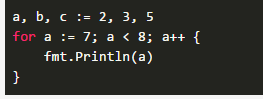
**第五节 Go 语言过程式编程**

**一 知识点**

* Go 语言中的 类型转换 和 类型断言
* Go 语言中的 分支 和 for 语句
* Go 语言中的 函数 和 goroutine

**二 Go语言语句基础**

之所以先学习过程式编程，是因为在Go语言中面向对象编程也是建立在面向过程的基础上的。形式上讲，Go语言需要使用分号(;) 来作为上下文语句的分隔结束符。实际上在前面的代码中我们可以看到在Go语言中很少使用分号，那是因为编译器会自动在需要分号的地方加上分号。但是有两个地方必须使用分号，第一个是需要在一个行中放入一条或多条语句时，或者是在使用原始的 for 循环时。Go语言也支持多重赋值，如a, b = b, a。另外在之前的课程中我们提到过快速声明操作符:=，它的作用是同时在一个语句中声明和赋值一个变量。当:=操作符用于多个逗号分隔的变量时，如果该变量已经存在，则只是简单的修改它的值。但是当:=操作符位于作用域的起始处时，Go语言会创建一个新的变量，不管该变量之前是否存在，如在if或者for语句中。下面有一个例子可以说明:



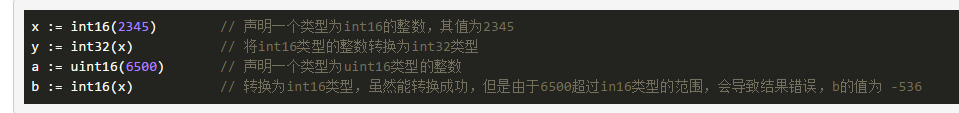
以上代码中，先使用:=声明并赋值了三个变量，Go会自动推导出变量的类型。然后再for语句处又一次使用:=操作符声明了变量a。需要注意的地方是，for语句代表了一个新的作用域，所以:=在这里新声明创建了一个变量a，这个变量和之前的变量a是完全不同的两个变量（内存地址不一样），所以是一个影子变量，因为覆盖了外部的同名变量。这是需要注意的一个地方。

**1. 类型转换**

Go语言提供了一种在不同但相互兼容的类型之间相互转换的方式，这种转换非常有用并且是安全的。但是需要注意的是在数值之间进行转换可能造成其他问题，如精度丢失或者错误的结果。以下是类型转换的语法:

* resultOfType := Type(expression)

几个例子：



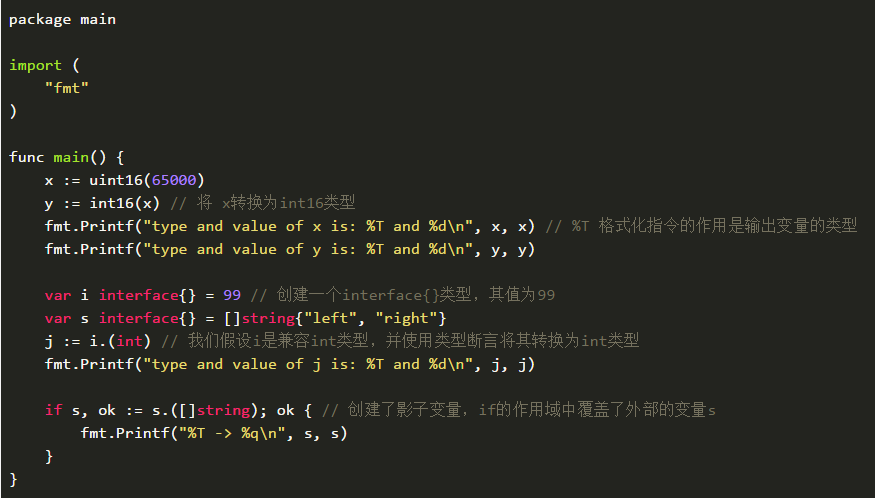
另外在Go语言中可以通过type关键字声明类型，如type StringsSlice []string 将[]string（string类型的切片）声明为StringSlice类型。

**2. 类型断言**

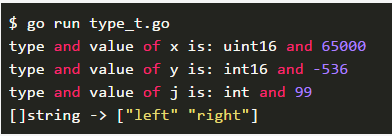
说到类型断言就需要先了解下Go语言中的接口。在Go语言中接口是一个自定义类型。它声明了一个或者多个方法。任何实现了这些方法的对象（类型）都满足这个接口。接口是完全抽象的，不能实例化。interface{}类型表示一个空接口，任何类型都满足空接口。也就是说interface{}类型的值可以用于表示任意Go语言类型的值。这里的空接口有点类似于python语言中的object实例。既然interface{} 可以用于表示任意类型，那有的时候我们需要将interface{}类型转换为我们需要的类型，这个操作类型断言。一般情况下只有我们希望表达式是某种特定类型的值时才使用类型断言。Go语言中可以使用以下语法：

* resultOfType, boolean := expression.(Type) // 安全的类型断言
* resultOfType := expression.(Type) // 非安全的类型断言，失败时程序会产生异常

例子



结果

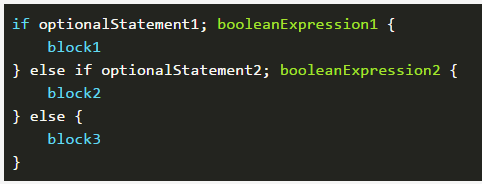


**三 分支和for语句**

Go语言提供了3种分支，即if、switch、select, 其中select用于监听channel（通道）在讲解通道的时候再详细介绍。

**1. if 分支**

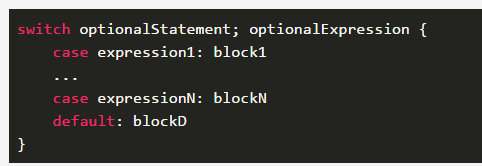
语法:

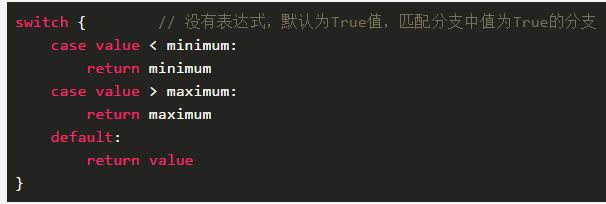


其中optionalStatement是可选的表达式，真正决定分支走向的是booleanExpression1的值。

**2. switch分支**

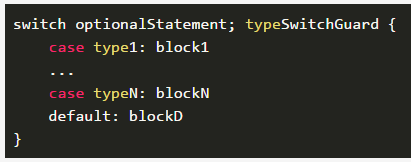
Go语言中switch分支既可用于常用的分支就象C语言中的switch一样，也可以用于类型开关，所谓类型开关就是用于判断变量属于什么类型。但是需要注意的是Go语言的switch语句不会自动贯穿，相反，如果想要贯穿需要添加fallthrough语句。表达式开关switch的语法如下：





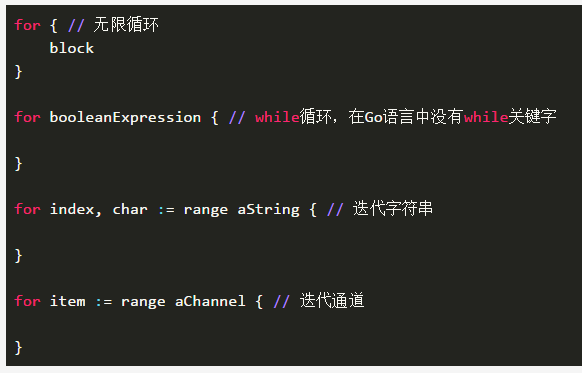
在上面的例子中，switch后面没有默认的表达式，这个时候Go语言默认其值为True。

在前面我们提到过类型断言，如果我们知道变量的类型就可以使用类型断言，但是当我们知道类型可能是许多类型中的一种时候，我们就可以使用类型开关。其语法如下：



**3. for循环语句**

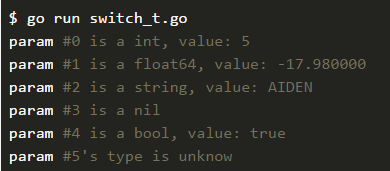
在前面的代码中我们已经遇到很多遍for语句了，它可以遍历数组，切片，映射等类型，也可以用于无限循环。以下是其语法：



例子



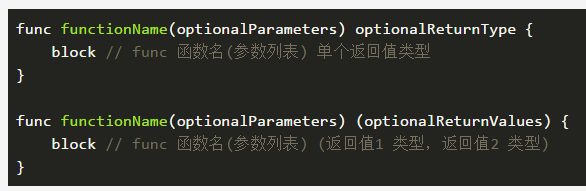
以上代码中我们首先创建了一个接收任意数量任意类型参数的函数，然后使用for ... range aSlice的语法迭代了每一个在切片items中的元素，接着使用了switch类型开关判断了每一个参数的类型，并打印了其值和类型。程序运行输出如下：



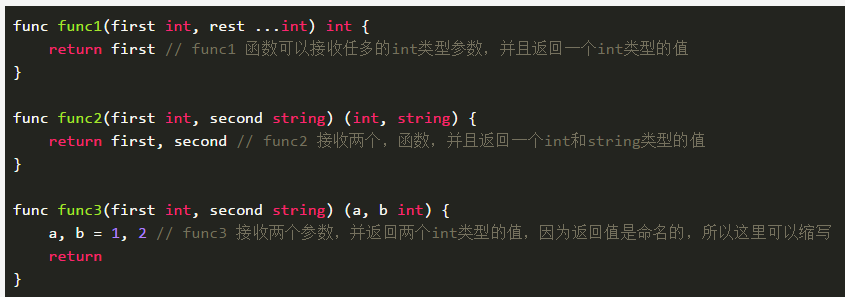
**四. 函数**

Go语言可以很方便的自定义函数，其中有特殊的函数main函数。main函数必须出现在main包里，且只能出现一次。当Go程序运行时候会自动调用main函数开始整个程序的执行。main函数不可接收任何参数，也不返回任何结果。

Go语言中函数的创建使用以下语法：



函数可以有任意多个参数，也可以有任意多个返回值，返回值可以是命名的。具体的看以下例子：



**五. 通信(channel)和并发(goroutine)语句**

在本课程开篇中我们就介绍过Go语言强大的并发功能，这些功能都是建立在通信和并发语句上的。所谓goroutine是程序中与其他goroutine完全独立而并发执行的函数或者方法调用。每一个Go程序都至少有一个goroutine，其中main()函数所在的goroutine是主goroutine。goroutine很像轻量级的线程，它们可以被大批量的创建。那goroutine之间怎么进行通信呢？Go语言中推荐的做法是使用channel(通道)。channel是一个双向的或者单向的通信管道，可以用于两个或者多个goroutine之间进行通信（即接收和发送）数据。

**1. 语法**

goroutine使用以下的go语句进行创建:

* go function(arguments)
* go func(parameters) { block } (arguments)

第二种方式中，我们是创建了一个临时的匿名函数，并马上在goroutine中执行。

当调用用go关键字执行函数时，函数会在另一个goroutine上马上执行，并且当前的goroutine的执行会从下一条语句马上恢复。因此执行一个go语句之后，当前程序中至少有两个goroutine在运行。

在大多数情况下，goroutine之间需要相互协作，最好的方式是通过channel来交换数据。使用下面语法创建channel（通道）：

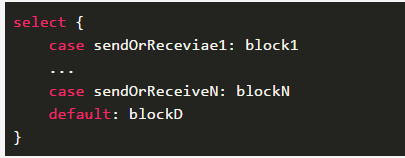
* make(chan Type)
* make(chan Type, capacity)

语法中的Type指明了通道能发送的数据类型。其中第一种语法中创建了一个同步的通道，一次只能发送一项数据，它会阻塞直到发送者准备好发送和接收者准备好接收。如果给定了capacity也就是缓冲区容量，在缓冲区容量未满之前通道都是异步无阻塞的。通道支持的操作如下:



**2. select语句**

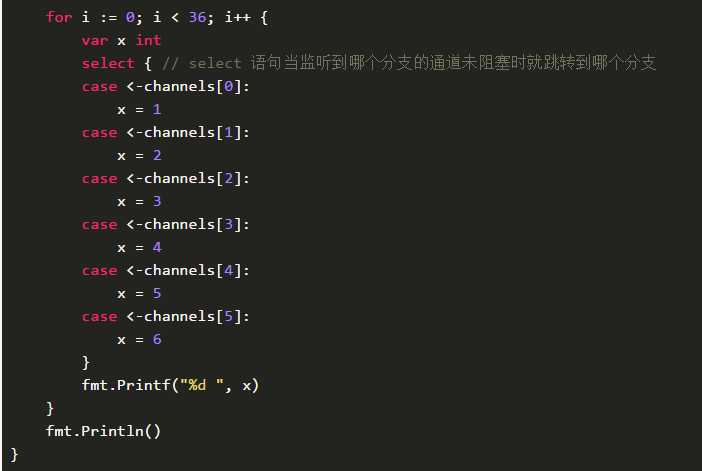
在前面的课程中我们提到过select语句，用于监听通道。其语法如下：



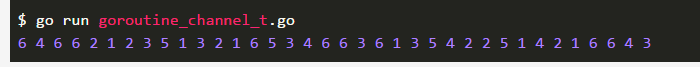
Go语言会从头至尾的判断每一个case中的发送和接收语句。如果其中任何一条语句可以执行（即没有被阻塞），那就从那些可执行的语句中任意选择一条来使用。如果所有的通道都被阻塞，那可能有两种情况。第一种，如果有default语句，那就会执行default 语句，同时程序的执行会从select语句恢复。第二种，如果没有default语句，则select语句会一直阻塞，直到有一个通道可用

例子





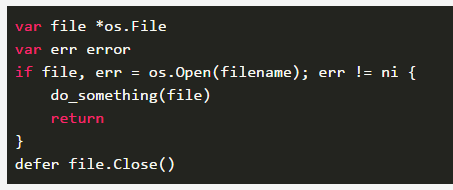
运行结果



**六. defer, panic和recover**

**1. defer**

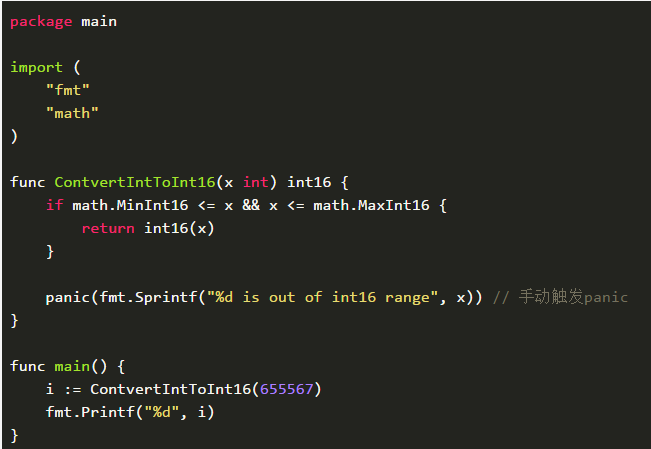
开发程序时，有的时候忘记关闭打开的文件导致程序执行失败，在python中可以很方便的使用with语句对这些资源进行自动管理。在Go中我们可以使用defer语句完成这项任务。defer语句用于延迟执行一个函数或者方法或者是当前创建的匿名函数，它会在外部函数或者方法返回之前但是其返回值计算之后执行。这样就可能在一个延迟执行的函数中修改函数的命名返回值。如果一个函数中又多个defer语句，它们会以后进先出的顺序执行。defer最常用的地方就是保证一个使用完成后的文件正常关闭。如下例子:



**2. panic和recover**

panic类似于其他程序中的异常，而recover 则用于恢复异常。当panic()函数被调用时，外围函数或者方法的执行会立即终止。然后任何延迟执行的函数都会被调用。这个过程一直在调用栈中层层发生，最后到达main函数，这个时候整个程序会终止，最终将最初的调用栈信息输出到stderr。但是当延迟执行函数中包含recover语句时，recover会捕捉到panic引发的异常，并停止panic的传播，这个时候我们能够以任何我们想用的方式处理panic。

Go语言将错误和异常两者区分对待。错误是指有可能出错的东西，程序中已经包含处理这些错误的优雅逻辑。而异常则是指不可能发生的事情。例如，一个永远为true的条件在实际环境中却是false。Go语言推荐使用错误，而不使用异常。通常情况下，我们可以在recover中阻止panic的传播，并将recover()的返回值转换成错误。



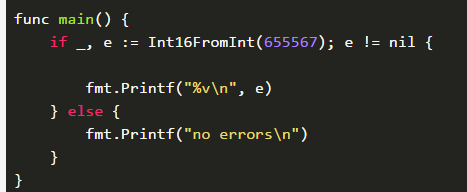
上面代码中为了演示panic，代码中手动促发了panic()的执行，但是我们没有使用recover进行捕捉，这会导致整个程序执行失败，下面执行程序验证下：



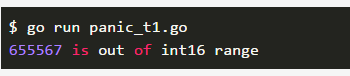
可以看到没有捕捉panic时，整个程序退出，并且打印出了调用栈的异常信息。

下面我们使用Go语言推荐的做法捕捉panic并将panic转换为error, 创建源文件panic\_t1.go，输入以下代码：



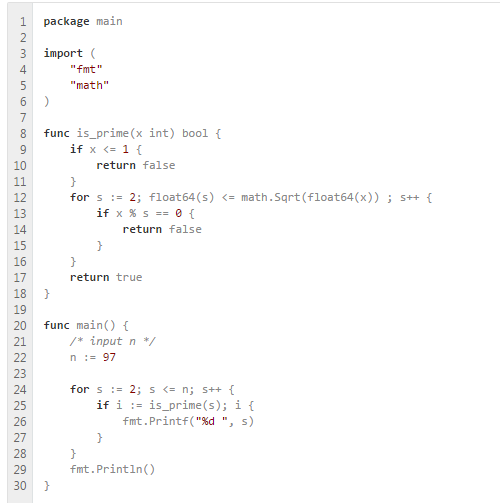


以上代码中，我们通过recover捕捉了异常，现在程序将异常转换成了错误，所以程序不会异常退出，执行验证如下：



值得注意的地方是，在以上代码中的Int16FromInt(x int) (i int16, err error)函数中，我们在defer语句的匿名函数中修改了命名的返回值err。该函数在被调用时，Go语言会自动的将其返回值设置为对应类型的零值，在Int16FromInt函数中，i被初始化为0，err被初始化为nil。当在defer语句中匿名函数执行时候，recover如果捕捉到异常，然后修改了命名返回值err，并保持i的值（零值 )不变。如果没有捕捉到异常，则程序正常返回i和nil。

练习，输出2-n之间的所有素数



**第六节 Go 语言面向对象编程**

**一 知识点**

* 自定义类型 和 结构体
* 接口

**二. 自定义类型以及结构体**

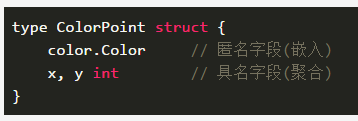
在讲解Go语言面向对象内容之前，需要说明下Go语言的代码是以包结构来组织的，且如果标示符(变量名，函数名，自定义类型等)如果以大写字母开头那么这些标示符是可以导出的，可以在任何导入了定义该标示符的包的包中直接使用。Go语言中的面向对象和C++，Java中的面向对象不同，因为Go语言不支持继承，Go语言只支持聚合。

**2.1 自定义类型**

在之前的课程中我们以及提到在Go语言中我们可以自定义类型，其语法如下:

* type typeName typeSpecification

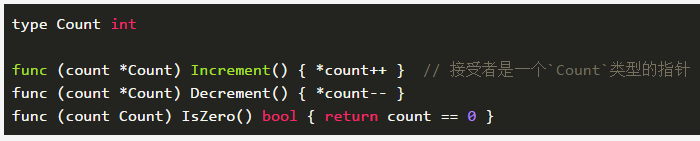
其中，typeName可以是一个包或者函数内唯一合法的Go标示符。typeSpecification 可以是任何内置的类型，一个接口或者是一个结构体。所谓结构体，它的字段是由其他类型或者接口组成。例如我们通过结构体定义了一下类型:



以上代码我们通过结构体自定义了类型ColorPoint，结构体中color.Color字段是Color包的类型color，这个字段没有名字，所以被称为匿名的，也是嵌入字段。字段x和y是有变量名的，所以被称为具名字段。假如我们创建了类型ColorPoint的一个值point（通过语法：point := ColorPoint{} 创建），那么这些字段可以通过point.Color、point.x、point.y访问。其他面向对象语言中的"类(class)"、"对象(object)"、"实例(instance)"在Go语言中我们完全避开使用。相反的我们使用"类型(type)"和其对应的"值"，其中自定义类型的值可以包含方法。

**2.2 方法**

方法是作用在自定义类型上的一类特殊函数，通常自定义类型的值会被传递给该函数，该值可能是以指针或者复制值的形式传递。定义方法和定义函数几乎相同，只是需要在func关键字和方法名之间必须写上接接受者。例如我们给类型Count定义了以下方法:



以上代码中，我们在内置类型int的基础上定义了自定义类型Count，然后给该类型添加了Increment()、Decrement()和IsZero()方法，其中前两者的接受者为Count类型的指针，后一个方法接收Count类型的值。

类型的方法集是指可以被该类型的值调用的所有方法的集合。

一个指向自定义类型的值的指针，它的方法集由该类型定义的所有方法组成，无论这些方法接受的是一个值还是一个指针。如果在指针上调用一个接受值的方法，Go语言会聪明地将该指针解引用。

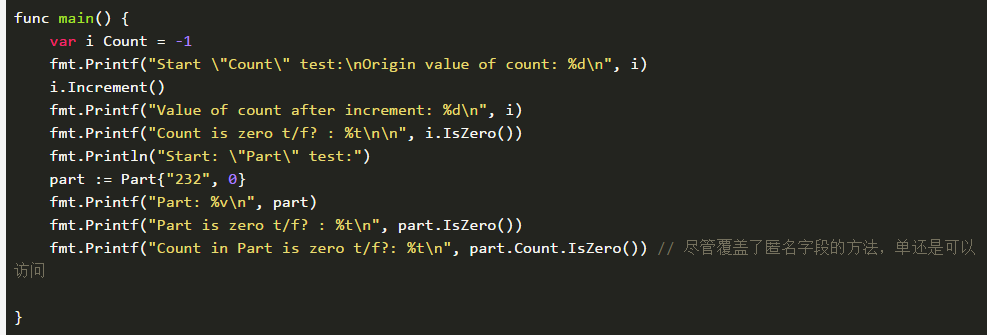
一个自定义类型值的方法集合则由该类型定义的接收者为值类型的方法组成，但是不包括那些接收者类型为指针的方法。

其实这些限制Go语言帮我们解决的非常好，结果就是我们可以在值类型上调用接收者为指针的方法。假如我们只有一个值，仍然可以调用一个接收者为指针类型的方法，这是因为Go语言会自动获取值的地址传递给该方法，前提是该值是可寻址的。

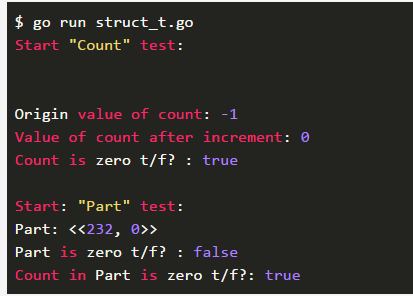
在以上定义的类型Count中，\*Count方法集是Increment(), Decrement()和IsZero()，Count的值的方法集是IsZero()。但是因为Count类型的是可寻址的，所以我们可以使用Count的值调用全部的方法。

另外如果结构体的字段也有方法，我们也可以直接通过结构体访问字段中的方法。下面让我们练习下，创建源文件struct\_t.go，输入以下代码:





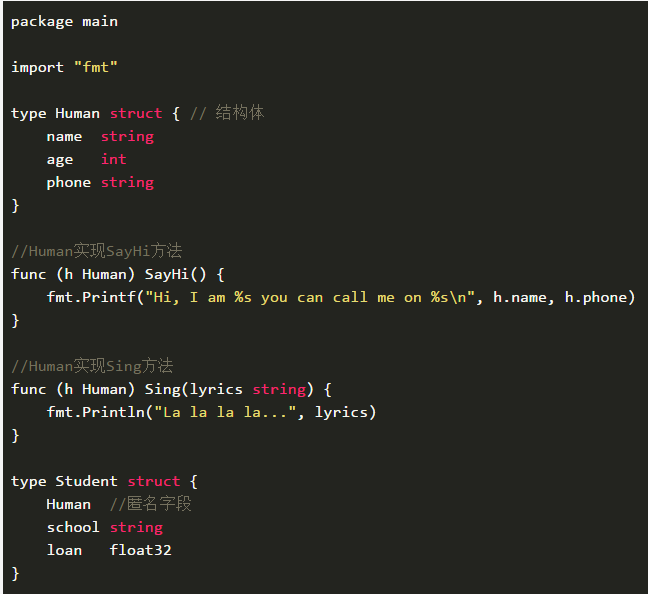
以上代码中，我们创建了Count类型，然后在其基础上又创建了结构体类型Part。我们为Count类型定义了3个方法，并在Part类型中创建了方法IsZero() 覆盖了其匿名字段Count中IsZero()方法。但是我们还是可以二次访问到匿名字段中被覆盖的方法。执行代码，输出如下：

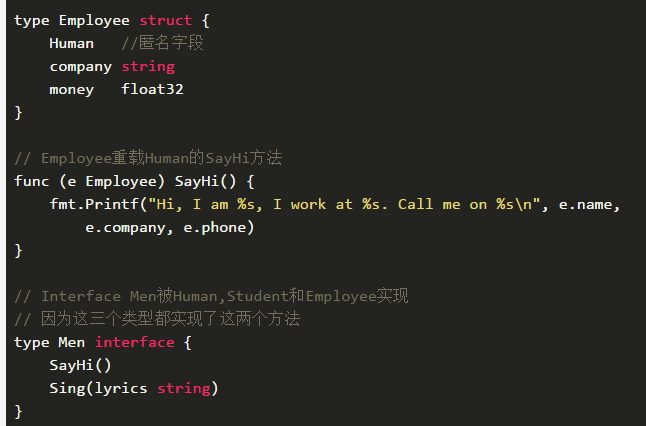


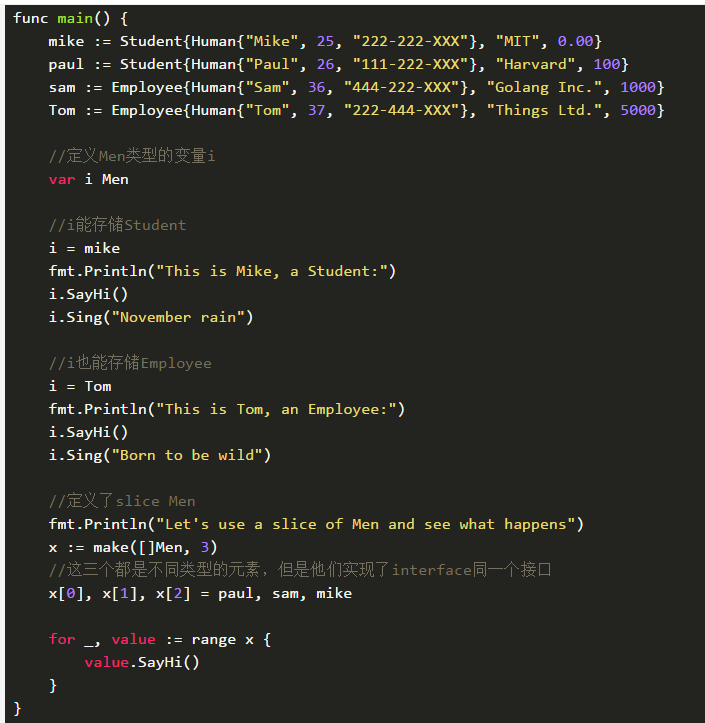
**三. 接口**

**3.1 接口基础**

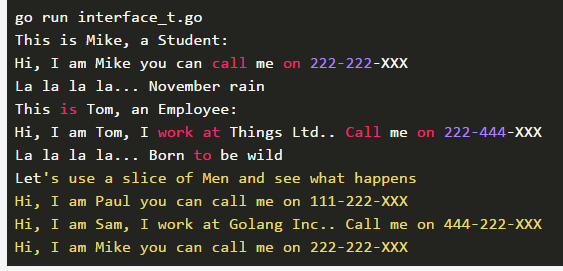
之所以说Go语言的面向对象很灵活，很大一部分原因是由于接口的存在。接口是一个自定义类型，它声明了一个或者多个方法签名，任何实现了这些方法的类型都实现这个接口。infterface{}类型是声明了空方法集的接口类型。任何一个值都满足interface{}类型，也就是说如果一个函数或者方法接收interface{}类型的参数，那么任意类型的参数都可以传递给该函数。接口是完全抽象的，不能实例化。接口能存储任何实现了该接口的类型。直接看例子吧，创建源文件interface\_t.go，输入以下代码：







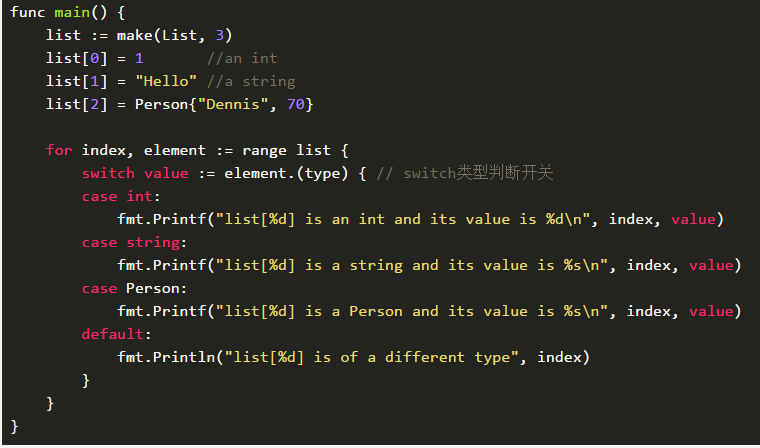
以上代码中，接口类型声明的变量能存储任何实现了该接口的类型的值。运行代码，输出如下:



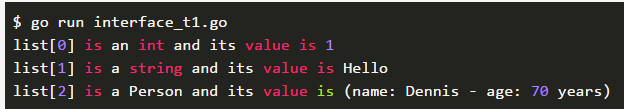
**3.2 接口变量值的类型**

我们知道接口类型声明的变量里能存储任何实现了该接口的类型的值。有的时候我们需要知道这个变量里的值的类型，那么需要怎么做呢？其实在之前的课程中我们就已经学习过了，可以使用类型断言，或者是switch类型判断分支。以下的例子interface\_t1.go我们使用了switch类型判断分支。





运行结果



**3.3 嵌入interface**

在前面的课程中我们已经知道在结构体中可以嵌入匿名字段，其实在接口里也可以再嵌入接口。如果一个interface1作为interface2的一个嵌入字段，那么interface2隐式的包含了interface1里的方法。如下例子中, Interface2包含了Interface1的所有方法。

