# **Go 语言介绍**

## **实验简介**

本节实验介绍Go语言，并编写了两个例子程序。

## **一、实验说明**

### **1. 环境登录**

无需密码自动登录，系统用户名shiyanlou，该用户具备sudo的权限，可以执行安装软件等管理员操作。

### **2. 环境介绍**

本实验环境采用Ubuntu Linux桌面环境，实验中会用到桌面上的程序：

1. Xfce终端: Linux命令行终端，打开后会进入Bash环境，可以使用Linux命令
2. Firefox及Opera：浏览器，可以用在需要前端界面的课程里，只需要打开环境里写的HTML/JS页面即可
3. gvim：非常好用的Vim编辑器，最简单的用法可以参考课程[Vim编辑器](http://www.shiyanlou.com/courses/2" \t "https://www.shiyanlou.com/courses/11/labs/49/_blank)
4. 其他编辑器：如果Vim不熟悉可以使用gedit或brackets，其中brackets比较适合开发前端代码。

### **3. 环境使用**

使用GVim编辑器输入实验所需的代码，然后使用XfceTerminal命令行环境进行编译运行，查看运行结果，运行后可以截图并分享自己的实验成果，实验楼提供的截图是后台截图，无法作弊，可以真实有效证明您已经完成了实验。

实验报告可以在个人主页中查看，其中含有每次实验的截图及笔记，以及每次实验的有效学习时间（指的是在实验桌面内操作的时间，如果没有操作，系统会记录为发呆时间）。这些都是您学习的真实性证明。

注：本课程所有源代码，可以在XfceTerminal中通过以下方式克隆到实验环境:

$ git clone http://git.shiyanlou.com/shiyanlou/Golang\_Programming

### **4. 参考文档**

本课程的文档部分参考了以下文档。

* [1]: [Go 语言官方文档](http://golang.org/doc/" \t "https://www.shiyanlou.com/courses/11/labs/49/_blank)
* [2]: [Go 入门指南](https://github.com/Unknwon/the-way-to-go_ZH_CN" \t "https://www.shiyanlou.com/courses/11/labs/49/_blank)
* [3]: [Go web编程](https://github.com/astaxie/build-web-application-with-golang/blob/master/LICENSE.md" \t "https://www.shiyanlou.com/courses/11/labs/49/_blank)

## **二. Go 语言介绍**

### **1. Go语言起源**

Go 语言起源 2007 年，并于 2009 年正式对外发布。它从 2009 年 9 月 21 日开始作为谷歌公司 20% 兼职项目，即相关员工利用 20% 的空余时间来参与 Go 语言的研发工作。该项目的三位领导者均是著名的计算机工程师：Robert Griesemer，参与开发 Java HotSpot 虚拟机；Rob Pike，Go 语言项目总负责人，贝尔实验室 Unix 团队成员，参与的项目包括 Plan 9，Inferno 操作系统和 Limbo 编程语言；Ken Thompson，贝尔实验室 Unix 团队成员，C 语言、Unix 和 Plan 9 的创始人之一，与 Rob Pike 共同开发了 UTF-8 字符集规范。2009 年 11 月 10 日，开发团队将 Go 语言项目以 BSD-style 授权（完全开源）正式公布了 Linux 和 Mac OS X 平台上的版本。Hector Chu 于同年 11 月 22 日公布了 Windows 版本。

作为一个开源项目，Go 语言借助开源社区的有生力量达到快速地发展，并吸引更多的开发者来使用并改善它。自该开源项目发布以来，超过 200 名非谷歌员工的贡献者对 Go 语言核心部分提交了超过 1000 个修改建议。在过去的 18 个月里，又有 150 开发者贡献了新的核心代码。这俨然形成了世界上最大的开源团队，并使该项目跻身 Ohloh 前 2% 的行列。大约在 2011 年 4 月 10 日，谷歌开始抽调员工进入全职开发 Go 语言项目。开源化的语言显然能够让更多的开发者参与其中并加速它的发展速度。在 Go 语言在 2010 年 1 月 8 日被 [Tiobe](http://www.tiobe.com/" \t "https://www.shiyanlou.com/courses/11/labs/49/_blank)（闻名于它的编程语言流行程度排名）宣布为“2009 年年度语言”后，引起各界很大的反响。目前 Go 语言在这项排名中的最高记录是在 2010 年 2 月创下的第13名，流行程度 1.778%。

### **2. Go语言的特性**

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，达到完美平衡，从而使编程变得更加有乐趣，而不是在艰难抉择中痛苦前行。因此，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，这能更好地利用大量的多核计算机。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。这个特性是 Go 语言最强有力的部分，不仅支持了日益重要的多核与多处理器计算机，也弥补了现存编程语言在这方面所存在的不足。

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。这不仅极大地提升了开发者的生产力，同时也使得软件开发过程中的代码测试环节更加紧凑，而不必浪费大量的时间在等待程序的构建上。

由于内存问题（通常称为内存泄漏）长期以来一直伴随着 C++ 的开发者们，Go 语言的设计者们认为内存管理不应该是开发人员所需要考虑的问题。因此尽管 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。

有许多活跃的开源项目都是基于Go语言进行开发，目前十分流行的[Docker](https://github.com/docker/docker" \t "https://www.shiyanlou.com/courses/11/labs/49/_blank)也是基于Go语言进行开发。在正式开始学习Go之前，让我们先尝试几个例子，直观感受下Go语言。

## **三. 示例一 Hello World**

### **1. 源代码**

第一个例子当然是HelloWorld了，使用[VIM](http://www.shiyanlou.com/courses/2" \t "https://www.shiyanlou.com/courses/11/labs/49/_blank)编辑源代码：

$ gvim hello.go

输入以下代码保存

package main

import (

"fmt"

"os"

)

func main() {

target := "World"

if len(os.Args) > 1 { /\* os.Args是一个参数切片 \*/

target = os.Args[1]

}

fmt.Println("Hello", target)

}

### **2. 说明**

Go语言使用C++风格的注释：//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。

### **3. 编译运行**

下面该对helloworld程序进行编译运行了。随着Go语言发布的Go命令行工具可以很方便的构建Go程序，包括编译连接Go源代码，管理Go 软件包等。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。

## **四. 示例二 goroutine和channel**

Go语言能之所以这么流行，goroutine和channel是两个很重要的因素，因为这两者让并发(并行)变的如此简单，具体请看如下分组求和的例子。

### **1. 源码**

使用[VIM](http://www.shiyanlou.com/courses/2" \t "https://www.shiyanlou.com/courses/11/labs/49/_blank)编写源代码：

$ gvim sum\_by\_group.go

输入以下源代码:

package main

import "fmt"

func sum(a []int, result chan int) {

sum := 0

for \_, v := range a {

sum += v

}

result <- sum

}

func main() {

a := []int{2, 3, 5, 6, 10, -5, 1, 0}

result := make(chan int)

go sum(a[:len(a)/2], result)

go sum(a[len(a)/2:], result)

x, y := <-result, <-result

fmt.Println(x, y, x+y)

}

### **2. 说明**

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。

### **3. 运行**

运行结果如下:

$ go run sum\_by\_group.go16 6 22

目前为止我们通过两个示例初步了解了Go语言，下一节开始我们正式开始学习Go语言。

## **作业**

请了解Go语言与C语言、Java语言和Python语言的差别。

# **Go 语言基础和数值布尔类型**

## **实验简介**

这一节中，我们详细介绍Go语言的一些基础知识，以及数值和布尔类型，下一节中我们会介绍字符串类型。

## **一、实验说明**

本课程所有源代码，可以在XfceTerminal中通过以下方式克隆到实验环境:

$ git clone http://git.shiyanlou.com/shiyanlou/Golang\_Programming

## **二. 环境变量和常用命令**

### **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. Go语言的命令行工具**

Go语言自带有一套完整的命令操作工具，你可以通过在shell中输入go来查看：

$ go

Go is a tool for managing Go source code.

Usage:

go command [arguments]

The commands are:

build compile packages and dependencies

clean remove object files

env print Go environment information

fix run go tool fix on packages

fmt run gofmt on package sources

get download and install packages and dependencies

install compile and install packages and dependencies

list list packages

run compile and run Go program

test test packages

tool run specified go tool

version print Go version

vet run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

c calling between Go and C

filetype file types

gopath GOPATH environment variable

importpath import path syntax

packages description of package lists

testflag description of testing flags

testfunc description of testing functions

Use "go help [topic]" for more information about that topic.

我们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。输出的信息类似

ok archive/tar 0.011s

FAIL archive/zip 0.022s

ok compress/gzip 0.033s

...

默认的情况下，不需要任何的参数，它会自动把你源码包下面所有test文件测试完毕，当然你也可以带上参数，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**

### **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：

break default func interface selectcase defer go map struct

chan else goto package switch

const fallthrough if range type

continue for import return var

Go语言同样预定义了许多标示符，可以很方便的使用，如下：

* 类型：

bool byte complex64 complex128 error float32 float64int int8 int16 int32 int64 rune stringuint uint8 uint16 uint32 uint64 uintptr

* 常量

true false iota

* 零值

nil

* 函数

append cap close complex copy delete imag len

make new panic print println real recover

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：

count, err = fmt.Println(x) // 获取打印的字节数以及相应的error值

count, \_ = fmt.Println(x) // 获取打印的字节数，并且丢弃error值

### **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：

const limit = 512 // 常量，其类型兼容任何数字const top uint16 = 1421 // 常量，类型：uint16

last := 1.5 // 变量，推断类型 flat64var a int // 变量，值为0，类型 intvar debug = false // 变量，推断类型 bool

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):

const (

Cyan = 0

Black = 1

White = 2

)

## **四. 数值类型和布尔类型**

Go语言提供了大量的内置数据类型，每一个数值类型都不同，意味着不能在不同类型之间进行二进制数值运算或者比较操作。无类型的数值常量可以兼容内置的任何类型的数值，在不同类型数值之间进行运算或者比较操作，需要进行类型转换。类型转换采用type(value)的方式，只要合法就能转换成功，即使会导致数据精度丢失。

### **1. 整型**

Go语言提供了11中整型，如下列表所示

| **类型** | **说明** |
| --- | --- |
| byte | 等同于uint8 |
| int | 依赖于不同平台下的实现，可以是int32或者int64 |
| int8 | [-128, 127] |
| int16 | [-32768, 32767] |
| int32 | [-2147483648, 2147483647] |
| int64 | [-9223372036854775808, 9223372036854775807] |
| rune | 等同于uint32 |
| uint | 依赖于不同平台下的实现，可以是uint32或者uint64 |
| uint8 | [0, 255] |
| uint16 | [0, 65535] |
| uint32 | [0, 4294967295] |
| uint64 | [0, 18446744073709551615] |
| uintptr | 一个可以恰好容纳指针值的无符号整型（对32位平台是uint32, 对64位平台是uint64） |

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，使用vim创建源文件type\_length.go，输入以下代码：

package main

import (

"fmt"

"unsafe"

)

func main() {

a := 12

fmt.Println("length of a: ", unsafe.Sizeof(a))

var b int = 12

fmt.Println("length of b(int): ", unsafe.Sizeof(b))

var c int8 = 12

fmt.Println("length of c(int8): ", unsafe.Sizeof(c))

var d int16 = 12

fmt.Println("length of d(int16): ", unsafe.Sizeof(d))

var e int32 = 12

fmt.Println("length of e(int32): ", unsafe.Sizeof(e))

var f int64 = 12

fmt.Println("length of f(int64): ", unsafe.Sizeof(f))

}

以上代码中，首先声明了目前的源文件属于main包，然后导入了fmt和unsafe包，fmt包用于格式化字符串，unsafe包含了用于获取Go语言类型信息的方法。然后在main函数中，我们分别声明了几种类型的整型变量，并通过unsafe.Sizeof方法获取该类型的字节长度。最后我们通过以下方法运行type\_length.go，同时打印出了输出：

$ go run type\_length.go length of a: 8length of b(int): 8length of c(int8): 1length of d(int16): 2length of e(int32): 4length of f(int64): 8

### **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：

| **类型** | **说明** |
| --- | --- |
| float32 | ±3.402 823 466 385 288 598 117 041 834 845 169 254 40x10<sup>38</sup> 计算精度大概是小数点后7个十进制数 |
| float64 | ±1.797 693 134 862 315 708 145 274 237 317 043 567 981x10<sup>38</sup> 计算精度大概是小数点后15个十进制数 |
| complex32 | 复数，实部和虚部都是float32 |
| complex64 | 复数，实部和虚部都是float64 |

### **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。

## **作业**

请使用上述讲的数值类型做一些基本算术运算，并思考结果。

# **Go 语言字符串**

## **实验简介**

本节介绍 Go 语言中的字符串以及常用的两个字符串处理包。

## **一、实验说明**

本课程所有源代码，可以在XfceTerminal中通过以下方式克隆到实验环境:

$ git clone http://git.shiyanlou.com/shiyanlou/Golang\_Programming

## **二. 字符串**

Go语言中的字符串是 [UTF-8](http://zh.wikipedia.org/wiki/UTF-8" \t "https://www.shiyanlou.com/courses/11/labs/58/_blank) 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:

t1 := "\"hello\"" //内容： "hello"

t2 := `"hello"` //内容：和t1一致

t3 := "\u6B22\u8FCE" //内容：你好

Go语言中的部分转义字符如下表所示：

| **转义字符** | **含义** |
| --- | --- |
| \\ | 表示反斜线 |
| \' | 单引号 |
| \" | 双引号 |
| \n | 换行符 |
| \uhhhh | 4个16进制数字给定的Unicode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：

| **语法** | **描述** |
| --- | --- |
| s += t | 将字符串t追加到s末尾 |
| s + t | 将字符串s和t级联 |
| s[n] | 从字符串s中索引位置为n处的原始字节 |
| s[n:m] | 从位置n到位置m-1处取得的字符（字节）串 |
| s[n:] | 从位置n到位置len(s)-1处取得的字符（字节）串 |
| s[:m] | 从位置0到位置m-1处取得的字符（字节）串 |
| len(s) | 字符串s中的字节数 |
| len([]rune(s)) | 字符串s中字符的个数，可以使用更快的方法utf8.RuneCountInString() |
| [ ]rune(s) | 将字符串s转换为一个unicode值组成的串 |
| string(chars) | chars类型是[]rune或者[]int32, 将之转换为字符串 |
| [ ]byte(s) | 无副本的将字符串s转换为一个原始的字节的切片数组，不保证转换的字节是合法的UTF-8编码字节 |

让我们尝试一个例子， 使用vim创建源文件string\_t.go，然后输入以下源代码：

package main

import (

"fmt")

func main() {

t0 := "\u6B22\u8FCE\u6765\u5230" // t0内容：欢迎来到

t1 := "\u5B9E\u9A8C\u697C" // t1内容：实验楼

t2 := t0 + t1

for index, char := range t2 {

fmt.Printf("%-2d %U '%c' %X %d\n",

index, char, char, []byte(string(char)), len([]byte(string(char))))

}

fmt.Printf("length of t0: %d, t1: %d, t2: %d\n", len(t0), len(t1), len(t2))

fmt.Printf("content of t2[0:2] is: %X\n", t2[0:2])

}

然后通过以下方式运行，在这里一起显示了程序的输出（由于console可能不支持中文显示，可能显示乱码）:

$ go run string\_t.go0 U+6B22 '欢' E6ACA2 33 U+8FCE '迎' E8BF8E 36 U+6765 '来' E69DA5 39 U+5230 '到' E588B0 312 U+5B9E '实' E5AE9E 315 U+9A8C '验' E9AA8C 318 U+697C '楼' E6A5BC 3

length of t0: 12, t1: 9, t2: 21

content of t2[0:2] is: E6AC

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：

| **格式化指令** | **含义** |
| --- | --- |
| %% | %字面量 |
| %b | 一个二进制整数，将一个整数格式化为二进制的表达方式 |
| %c | 一个Unicode的字符 |
| %d | 十进制数值 |
| %o | 八进制数值 |
| %x | 小写的十六进制数值 |
| %X | 大写的十六进制数值 |
| %U | 一个Unicode表示法表示的整形码值，默认是4个数字字符 |
| %s | 输出以原生的UTF-8字节表示的字符，如果console不支持UTF-8编码，则会输出乱码 |
| %t | 以true或者false的方式输出布尔值 |
| %v | 使用默认格式输出值，或者使用类型的String()方法输出的自定义值，如果该方法存在的话 |
| %T | 输出值的类型 |

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

让我们练习一下，使用vim创建源文件fmt\_t.go，输入以下源码：

package main

import (

"fmt"

)

func main() {

text := "\u5B9E\u9A8C\u697C"

fmt.Printf("bool output:\n%t\n%t\n\n", true, false)

fmt.Println("number output, origin value: 64")

fmt.Printf("|%b|%8b|%-8b|%08b|% 8b|\n", 64, 64, 64, 64, 64)

fmt.Printf("|%x|%8x|%-8x|%08X|% 8X|\n\n", 64, 64, 64, 64, 64)

fmt.Println(`text output, origin value: \u5B9E\u9A8C\u697C`)

fmt.Printf("content: %s\n", text)

fmt.Printf("hex value: % X\nUnicode value: ", text)

for \_, char := range text {

fmt.Printf("%U ", char)

}

fmt.Println()

bytes := []byte(text)

fmt.Printf("value of bytes: %s\n", bytes)

fmt.Printf("hex value of bytes: % X\n", bytes)

fmt.Printf("origin value of bytes: %v\n", bytes)

}

运行代码，输出如下：

$ go run fmt\_t.gobool output:truefalse

number output, origin value: 64

|1000000| 1000000|1000000 |01000000| 1000000|

|40| 40|40 |00000040| 40|

text output, origin value: \u5B9E\u9A8C\u697C

content: 实验楼

hex value: E5 AE 9E E9 AA 8C E6 A5 BC

Unicode value: U+5B9E U+9A8C U+697C value of bytes: 实验楼

hex value of bytes: E5 AE 9E E9 AA 8C E6 A5 BC

origin value of bytes: [229 174 158 233 170 140 230 165 188]

代码一目了然，就不详细解释了。

## **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。

### **1. strings 包**

[strings](http://golang.org/pkg/strings/" \t "https://www.shiyanlou.com/courses/11/labs/58/_blank)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：

package main

import (

"fmt"

"strings"

)

func main() {

var str string = "go\_lang"

fmt.Printf("T/F? Does the string \"%s\" have prefix \"%s\"? ", str, "go")

fmt.Printf("%t\n", strings.HasPrefix(str, "go"))

fmt.Printf("T/F? Does the string \"%s\" contains \"%s\"? ", str, "-")

fmt.Printf("%t\n", strings.Contains(str, "-"))

str\_new := strings.Replace(str, "go", "python", 1)

fmt.Printf("Origin string: \"%s\", after replace: \"%s\"\n", str, str\_new)

fmt.Printf("Number of 'n' in \"%s\" is: %d\n", str\_new, strings.Count(str\_new, "n"))

}

执行代码，输出如下：

$ go run strings\_package.go

T/F? Does the string "go\_lang" have prefix "go"? true

T/F? Does the string "go\_lang" contains "-"? false

Origin string: "go\_lang", after replace: "python\_lang"

Number of 'n' in "python\_lang" is: 2

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。

### **2. strconv 包**

[strconv](http://golang.org/pkg/strconv/" \t "https://www.shiyanlou.com/courses/11/labs/58/_blank)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：

package main

import (

"fmt"

"strconv")

func main() {

var ori string = "123456"

var i int

var s string

fmt.Printf("The size of ints is: %d\n", strconv.IntSize)

i, \_ = strconv.Atoi(ori)

fmt.Printf("The integer is: %d\n", i)

i = i + 5

s = strconv.Itoa(i)

fmt.Printf("The new string is: %s\n", s)

}

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：

$ go run strconv\_package.go

The size of ints is: 64

The integer is: 123456

The new string is: 123461

## **作业**

请通过查阅资料掌握字符串的常用处理包。

# **Go 语言复合类型**

## **实验简介**

这一节讲解Go语言的复合类型，包括数组、切片和映射等。

## **一、实验说明**

本课程所有源代码，可以在XfceTerminal中通过以下方式克隆到实验环境:

$ git clone http://git.shiyanlou.com/shiyanlou/Golang\_Programming

## **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：

package main

import (

"fmt"

)

func swap1(x, y, p \*int) {

if \*x > \*y {

\*x, \*y = \*y, \*x

}

\*p = \*x \* \*y

}

func swap2(x, y int) (int, int, int) {

if x > y {

x, y = y, x

}

return x, y, x \* y

}

func main() {

i := 9

j := 5

product := 0

swap1(&i, &j, &product)

fmt.Println(i, j, product)

a := 64

b := 23

a, b, p := swap2(a, b)

fmt.Println(a, b, p)

}

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：

$ go run pointer.go5 9 4523 64 1472

## **三. 数组和切片**

### **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。

### **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：

package main

import (

"fmt"

)

func main() {

a := [...]int{1, 2, 3, 4, 5, 6, 7}

fmt.Printf("len and cap of array %v is: %d and %d\n", a, len(a), cap(a))

fmt.Printf("item in array: %v is:", a)

for \_, value := range a {

fmt.Printf("% d", value)

}

fmt.Println()

s1 := a[3:6]

fmt.Printf("len and cap of slice: %v is: %d and %d\n", s1, len(s1), cap(s1))

fmt.Printf("item in slice: %v is:", s1)

for \_, value := range s1 {

fmt.Printf("% d", value)

}

fmt.Println()

s1[0] = 456

fmt.Printf("item in array changed after changing slice: %v is:", s1)

for \_, value := range a {

fmt.Printf("% d", value)

}

fmt.Println()

s2 := make([]int, 10, 20)

s2[4] = 5

fmt.Printf("len and cap of slice: %v is: %d and %d\n", s2, len(s2), cap(s2))

fmt.Printf("item in slice %v is:", s2)

for \_, value := range s2 {

fmt.Printf("% d", value)

}

fmt.Println()

}

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：

$ go run slice\_array.go

len and cap of array [1 2 3 4 5 6 7] is: 7 and 7

item in array: [1 2 3 4 5 6 7] is: 1 2 3 4 5 6 7

len and cap of slice: [4 5 6] is: 3 and 4

item in slice: [4 5 6] is: 4 5 6

item in array changed after changing slice: [456 5 6] is: 1 2 3 456 5 6 7

len and cap of slice: [0 0 0 0 5 0 0 0 0 0] is: 10 and 20

item in slice [0 0 0 0 5 0 0 0 0 0] is: 0 0 0 0 5 0 0 0 0 0

## **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：

| **语法** | **含义** |
| --- | --- |
| m[k] = v | 用键k来将值赋值给映射m。如果映射m中的k已存在，则将之前的值舍弃 |
| delete(m, k) | 将键k及其相关的值从映射m中删除，如果k不存在则不执行任何操作 |
| v := m[k] | 从映射m中取得键k相对应的值并赋值给v。如果k不存在，则将映射类型的0值赋值v |
| v, found := m[k] | 从映射m中取得键k相对应的值并赋值给v， 并将found的值赋值为false。如果k不存在，则found为false |
| len(m) | 返回映射m中的项 |
| k := range m | 遍历映射m中的键 |
| k, v := range m | 同时遍历映射中的键和值 |

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:

package main

import (

"fmt"

)

func main() {

shiyanlou := make(map[string]string) // 与 map[string]string 相同

shiyanlou["golang"] = "docker"

shiyanlou["python"] = "flask web framework"

shiyanlou["linux"] = "sys administrator"

fmt.Print("Traverse all keys: ")

for key := range shiyanlou { // 遍历了映射的所有键

fmt.Printf("% s ", key)

}

fmt.Println()

delete(shiyanlou, "linux") // 从映射中删除键"linux"及其值

shiyanlou["golang"] = "beego web framework" // 更新键“golang"的值

v, found := shiyanlou["linux"]

fmt.Printf("Found key \"linux\" Yes or False: %t, value of key \"linux\": \"%s\"", found, v)

fmt.Println()

fmt.Println("Traverse all keys/values after changed:")

for k, v := range shiyanlou { //遍历了映射的所有键/值对

fmt.Printf("\"%s\": \"%s\"\n", k, v)

}

}

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：

$ go run map\_t.go

Traverse all keys: golang python linux

Found key "linux" Yes or False: false, value of key "linux": ""

Traverse all keys/values after changed:"golang": "beego web framework""python": "flask web framework"

## **作业**

请使用映射实现统计文章中每个单词出现的次数。

# **Go 语言过程式编程**

## **实验简介**

这节讲解Go语言的各种语句，控制结构，以及如何使用这些语句进行过程式编程，课程的内容比较多，但是都比较重要，在编写Go程序的过程中会频繁使用到。

## **一、实验说明**

本课程所有源代码，可以在XfceTerminal中通过以下方式克隆到实验环境:

$ git clone http://git.shiyanlou.com/shiyanlou/Golang\_Programming

## **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:

a, b, c := 2, 3, 5 for a := 7; a < 8; a++ {

fmt.Println(a)

}

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：

x := int16(2345) // 声明一个类型为int16的整数，其值为2345

y := int32(x) // 将int16类型的整数转换为int32类型

a := uint16(6500) // 声明一个类型为uint16类型的整数

b := int16(x) // 转换为int16类型，虽然能转换成功，但是由于6500超过in16类型的范围，会导致结果错误，b的值为 -536

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

使用VIM创建源文件type\_t.go，输入以下源文件：

package main

import (

"fmt")

func main() {

x := uint16(65000)

y := int16(x) // 将 x转换为int16类型

fmt.Printf("type and value of x is: %T and %d\n", x, x) // %T 格式化指令的作用是输出变量的类型

fmt.Printf("type and value of y is: %T and %d\n", y, y)

var i interface{} = 99 // 创建一个interface{}类型，其值为99

var s interface{} = []string{"left", "right"}

j := i.(int) // 我们假设i是兼容int类型，并使用类型断言将其转换为int类型

fmt.Printf("type and value of j is: %T and %d\n", j, j)

if s, ok := s.([]string); ok { // 创建了影子变量，if的作用域中覆盖了外部的变量s

fmt.Printf("%T -> %q\n", s, s)

}

}

运行程序：

$ go run type\_t.go

type and value of x is: uint16 and 65000

type and value of y is: int16 and -536

type and value of j is: int and 99

[]string -> ["left" "right"]

## **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:

if optionalStatement1; booleanExpression1 {

block1

} else if optionalStatement2; booleanExpression2 {

block2

} else {

block3

}

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：

switch optionalStatement; optionalExpression {

case expression1: block1

...

case expressionN: blockN

default: blockD

}

下面是个例子：

switch { // 没有表达式，默认为True值，匹配分支中值为True的分支

case value < minimum:

return minimum

case value > maximum:

return maximum

default:

return value

}

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：

switch optionalStatement; typeSwitchGuard {

case type1: block1

...

case typeN: blockN

default: blockD

}

### **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：

for { // 无限循环

block

}

for booleanExpression { // while循环，在Go语言中没有while关键字

}

for index, char := range aString { // 迭代字符串

}

for item := range aChannel { // 迭代通道

}

说了这么多，让我们进行下练习，创建源文件switch\_t.go，输入以下代码：

package main

import (

"fmt")

func classchecker(items ...interface{}) { // 创建一个函数，该函数可以接受任意多的任意类型的参数

for i, x := range items {

switch x := x.(type) { // 创建了影子变量

case bool:

fmt.Printf("param #%d is a bool, value: %t\n", i, x)

case float64:

fmt.Printf("param #%d is a float64, value: %f\n", i, x)

case int, int8, int16, int32, int64:

fmt.Printf("param #%d is a int, value: %d\n", i, x)

case uint, uint8, uint16, uint32, uint64:

fmt.Printf("param #%d is a uint, value: %d\n", i, x)

case nil:

fmt.Printf("param #%d is a nil\n", i)

case string:

fmt.Printf("param #%d is a string, value: %s\n", i, x)

default:

fmt.Printf("param #%d's type is unknow\n", i)

}

}

}

func main() {

classchecker(5, -17.98, "AIDEN", nil, true, complex(1, 1))

}

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：

$ go run switch\_t.go

param #0 is a int, value: 5

param #1 is a float64, value: -17.980000

param #2 is a string, value: AIDEN

param #3 is a nil

param #4 is a bool, value: true

param #5's type is unknow

## **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：

func functionName(optionalParameters) optionalReturnType {

block // func 函数名(参数列表) 单个返回值类型

}

func functionName(optionalParameters) (optionalReturnValues) {

block // func 函数名(参数列表) (返回值1 类型，返回值2 类型)

}

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：

func func1(first int, rest ...int) int {

return first // func1 函数可以接收任多的int类型参数，并且返回一个int类型的值

}

func func2(first int, second string) (int, string) {

return first, second // func2 接收两个，函数，并且返回一个int和string类型的值

}

func func3(first int, second string) (a, b int) {

a, b := 1, 2 // func3 接收两个参数，并返回两个int类型的值，因为返回值是命名的，所以这里可以缩写

return

}

## **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:

| **语法** | **含义** |
| --- | --- |
| channel <- value | 发送value到通道中，有可能阻塞 |
| <-channel | 从通道中接收数据 |
| x := <-channel | 接收数据并赋值给x |
| x, ok := <-channel | 功能同上，同时检查通道是否已关闭或者是否为空 |

### **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：

select {

case sendOrReceviae1: block1

...

case sendOrReceiveN: blockN

default: blockD

}

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

下面让我们使用以上的相关知识进行下练习，使用VIM创建源文件goroutine\_channel\_t.go，输入如下源代码：

package main

import (

"fmt"

"math/rand")

func main() {

channels := make([]chan bool, 6) // 创建一个类型为chan bool的切片，每一项是能发送bool值的通道

for i := range channels { // 通过`range`初始化切片

channels[i] = make(chan bool)

}

go func() { // 在其他gouroutine中执行匿名函数

for {

channels[rand.Intn(6)] <- true // rand.Intn(n int)的用途是产生一个不大于n的随机数

} // 发送数据到随机出现的通道

}()

for i := 0; i < 36; i++ {

var x int

select { // select 语句当监听到哪个分支的同道未阻塞时就跳转到哪个分支

case <-channels[0]:

x = 1

case <-channels[1]:

x = 2

case <-channels[2]:

x = 3

case <-channels[3]:

x = 4

case <-channels[4]:

x = 5

case <-channels[5]:

x = 6

}

fmt.Printf("%d ", x)

}

fmt.Println()

}

通过以上注释可以很清晰的看到整个代码的执行流程，下面我们执行代码:

$ go run goroutine\_channel\_t.go6 4 6 6 2 1 2 3 5 1 3 2 1 6 5 3 4 6 6 3 6 1 3 5 4 2 2 5 1 4 2 1 6 6 4 3

## **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:

var file \*os.Filevar err errorif file, err = os.Open(filename); err != ni {

do\_something(file)

return

}

defer file.Close()

### **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。

使用VIM创建源文件panic\_t.go, 输入以下代码：

package main

import (

"fmt"

"math"

)

func ContvertIntToInt16(x int) int16 {

if math.MinInt16 <= x && x <= math.MaxInt16 {

return int16(x)

}

panic(fmt.Sprintf("%d is out of int16 range", x)) // 手动触发panic

}

func main() {

i := ContvertIntToInt16(655567)

fmt.Printf("%d", i)

}

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：

$ go run panic\_t.go

panic: 655567 is out of int16 range

goroutine 16 [running]:

runtime.panic(0x96bc0, 0x208178180)

/usr/local/go/src/pkg/runtime/panic.c:279 +0xf5

main.ContvertIntToInt16(0xa00cf, 0x3ec8f)

/Users/aiden/Project/golang/panic\_t.go:13 +0x10f

main.main()

/Users/aiden/Project/golang/panic\_t.go:17 +0x26

goroutine 17 [runnable]:

runtime.MHeap\_Scavenger()

/usr/local/go/src/pkg/runtime/mheap.c:507

runtime.goexit()

/usr/local/go/src/pkg/runtime/proc.c:1445

goroutine 18 [runnable]:

bgsweep()

/usr/local/go/src/pkg/runtime/mgc0.c:1976

runtime.goexit()

/usr/local/go/src/pkg/runtime/proc.c:1445

goroutine 19 [runnable]:

runfinq()

/usr/local/go/src/pkg/runtime/mgc0.c:2606

runtime.goexit()

/usr/local/go/src/pkg/runtime/proc.c:1445exit status 2

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：

package main

import (

"fmt"

"math"

)

func ContvertIntToInt16(x int) int16 {

if math.MinInt16 <= x && x <= math.MaxInt16 {

return int16(x)

}

panic(fmt.Sprintf("%d is out of int16 range", x)) // 手动触发panic

}

func Int16FromInt(x int) (i int16, err error) {

defer func() { // 延迟执行匿名函数，并使用recover捕捉了panic，并将panic转换为了error

if e := recover(); e != nil {

err = fmt.Errorf("%v", e)

}

}()

i = ContvertIntToInt16(x)

return i, nil

}

func main() {

if \_, e := Int16FromInt(655567); e != nil {

fmt.Printf("%v\n", e)

} else {

fmt.Printf("no errors\n")

}

}

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：

$ go run panic\_t1.go655567 is out of int16 range

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。

## **作业**

请使用本节所讲知识编写一个函数，实现输入一个正整数n，输出2-n之间的所有素数。

# **Go 语言面向对象编程**

## **实验简介**

区别于上一节实验，这一节实验中我们将学习 Go 语言的面向对象编程。

## **一、实验说明**

本课程所有源代码，可以在XfceTerminal中通过以下方式克隆到实验环境：

$ git clone http://git.shiyanlou.com/shiyanlou/Golang\_Programming

## **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。

### **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:

type ColorPoint struct {

color.Color // 匿名字段(嵌入)

x, y int // 具名字段(聚合)

}

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. 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:

type Count int

func (count \*Count) Increment() { \*count++ } // 接受者是一个`Count`类型的指针

func (count \*Count) Decrement() { \*count-- }

func (count Count) IsZero() bool { return count == 0 }

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:

package main

import "fmt"

type Count int // 创建自定义类型 Count

func (count \*Count) Increment() { \*count++ } // Count类型的方法

func (count \*Count) Decrement() { \*count-- }

func (count Count) IsZero() bool { return count == 0 }

type Part struct { // 基于结构体创建自定义类型 Part

stat string

Count // 匿名字段

}

func (part Part) IsZero() bool { // 覆盖了匿名字段Count的IsZero()方法

return part.Count.IsZero() && part.stat == "" // 调用了匿名字段的方法

}

func (part Part) String() string { // 定义String()方法，自定义了格式化指令%v的输出

return fmt.Sprintf("<<%s, %d>>", part.stat, part.Count)

}

func main() {

var i Count = -1

fmt.Printf("Start \"Count\" test:\nOrigin value of count: %d\n", i)

i.Increment()

fmt.Printf("Value of count after increment: %d\n", i)

fmt.Printf("Count is zero t/f? : %t\n\n", i.IsZero())

fmt.Println("Start: \"Part\" test:")

part := Part{"232", 0}

fmt.Printf("Part: %v\n", part)

fmt.Printf("Part is zero t/f? : %t\n", part.IsZero())

fmt.Printf("Count in Part is zero t/f?: %t\n", part.Count.IsZero()) // 尽管覆盖了匿名字段的方法，单还是可以访问

}

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：

$ go run struct\_t.goStart "Count" test:

Origin value of count: -1Value of count after increment: 0Count is zero t/f? : true

Start: "Part" test:

Part: <<232, 0>>

Part is zero t/f? : falseCount in Part is zero t/f?: true

## **三. 接口**

### **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：

package main

import "fmt"

type Human struct { // 结构体

name string

age int

phone string

}

//Human实现SayHi方法

func (h Human) SayHi() {

fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)

}

//Human实现Sing方法

func (h Human) Sing(lyrics string) {

fmt.Println("La la la la...", lyrics)

}

type Student struct {

Human //匿名字段

school string

loan float32

}

type Employee struct {

Human //匿名字段

company string

money float32

}

// Employee重载Human的SayHi方法

func (e Employee) SayHi() {

fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,

e.company, e.phone)

}

// Interface Men被Human,Student和Employee实现// 因为这三个类型都实现了这两个方法

type Men interface {

SayHi()

Sing(lyrics string)

}

func main() {

mike := Student{Human{"Mike", 25, "222-222-XXX"}, "MIT", 0.00}

paul := Student{Human{"Paul", 26, "111-222-XXX"}, "Harvard", 100}

sam := Employee{Human{"Sam", 36, "444-222-XXX"}, "Golang Inc.", 1000}

Tom := Employee{Human{"Tom", 37, "222-444-XXX"}, "Things Ltd.", 5000}

//定义Men类型的变量i

var i Men

//i能存储Student

i = mike

fmt.Println("This is Mike, a Student:")

i.SayHi()

i.Sing("November rain")

//i也能存储Employee

i = Tom

fmt.Println("This is Tom, an Employee:")

i.SayHi()

i.Sing("Born to be wild")

//定义了slice Men

fmt.Println("Let's use a slice of Men and see what happens")

x := make([]Men, 3)

//这三个都是不同类型的元素，但是他们实现了interface同一个接口

x[0], x[1], x[2] = paul, sam, mike

for \_, value := range x {

value.SayHi()

}

}

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:

go run interface\_t.go

This is Mike, a Student:

Hi, I am Mike you can call me on 222-222-XXX

La la la la... November rain

This is Tom, an Employee:

Hi, I am Tom, I work at Things Ltd.. Call me on 222-444-XXX

La la la la... Born to be wild

Let's use a slice of Men and see what happens

Hi, I am Paul you can call me on 111-222-XXX

Hi, I am Sam, I work at Golang Inc.. Call me on 444-222-XXX

Hi, I am Mike you can call me on 222-222-XXX

### **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。

package main

import (

"fmt"

"strconv")

type Element interface{}

type List []Element

type Person struct {

name string

age int

}

// 实现了fmt.Stringer接口

func (p Person) String() string {

return "(name: " + p.name + " - age: " + strconv.Itoa(p.age) + " years)"

}

func main() {

list := make(List, 3)

list[0] = 1 //an int

list[1] = "Hello" //a string

list[2] = Person{"Dennis", 70}

for index, element := range list {

switch value := element.(type) { // switch类型判断开关

case int:

fmt.Printf("list[%d] is an int and its value is %d\n", index, value)

case string:

fmt.Printf("list[%d] is a string and its value is %s\n", index, value)

case Person:

fmt.Printf("list[%d] is a Person and its value is %s\n", index, value)

default:

fmt.Println("list[%d] is of a different type", index)

}

}

}

运行结果：

$ go run interface\_t1.go

list[0] is an int and its value is 1

list[1] is a string and its value is Hello

list[2] is a Person and its value is (name: Dennis - age: 70 years)

### **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。

type Interface1 interface {

Send()

Receive()

}

type Interface2 interface {

Interface1

Close()

}

## **作业**

请使用上述知识写一个交通工具与汽车、飞机的例子程序。

# **Go 语言和词频统计**

## **实验简介**

本节实验是 Go 语言编程实验的最后一节，我们将编写一个词频统计程序，我们希望该词频统计应用有以下功能：

* 统计多个文件中英文单词出现的次数
* 按照词频从多到少排序输出
* 支持并发

同时我们会将该词频统计程序打成包，以便在其他程序中使用。

## **一、实验说明**

本课程所有源代码，可以在XfceTerminal中通过以下方式克隆到实验环境:

$ git clone http://git.shiyanlou.com/shiyanlou/Golang\_Programming

## **二. 包**

这一课内容作为本课程(Go语言基础课程)最后的课程，我们将学习“包”，同时我们将写一个英文词频统计程序作为本课程的结课练习。

前面我们了解过Go语言组织代码的方式是包，包是各种类型和函数的集合。在包中，如果标示符(类型名称，函数名称，方法名称)的首字母是大写，那这些标示符是可以被导出的，也就是说可以在包以外直接使用。前面我们也提到了$GOPATH环境变量（指向一个或多个目录），以及其子目录src目录的，当我们使用import关键字导入包的时候，Go语言会在$GOPATH，GOROOT目录中搜索包。

我们创建的自定义的包最好放在$GOPATH的src目录下，如果这个包只属于某个应用程序，可以直接放在应用程序源代码的子目录下，但如果我们希望这个包可以被其他的应用程序共享，那就应该放在$GOPATH的src目录下，每个包单独放在一个目录里，如果两个不同的包放在同一目录下，会出现名字冲突的编译错误。作为惯例，包的源代码应该放在一个同名的文件夹下面。同一个包可以有任意多的源文件，文件名的名字也没有任何规定。

## **三. 词频统计程序**

### **1. 准备工作**

作为最后本课程最后一门课，我们将编写一个词频统计程序，我们希望该词频统计应用有以下功能：

* 统计多个文件中英文单词出现的次数
* 按照词频从多到少排序输出
* 支持并发

同时我们会将该词频统计程序打成包，以便在其他程序中使用。在使用该包前，我们需要设置$GOPATH环境变量。在console中(lxterminal)中按照以下步骤操作。

$ cd $HOME

$ mkdir -p golang/src/wordcount

$ export GOPATH=$HOME/golang

$ cd $GOPATH/src/wordcount

以上linux命令中，我们创建了目录golang/src/wordcount，并把$GOPATH设置成$HOME/golang。如果linux当前用户为shiyanlou，那么$HOME和$GOPATH的值应该如下所示.

$ echo $HOME

/home/shiyanlou

$ echo $GOPATH

/home/shiyanlou/golang

### **2. 实现**

词频统计的程序逻辑很简单。我们首先会创建一个映射，然后读取文件的每一行，提取单词，然后更新映射中单词所对应的数量即可。

为了演示面向对象和goroutine的使用，我们将基础映射类型封装成了一个统计单词频率的包。我们在基础映射类型上创建额类型WordCound，然后为该类型了实现了关键方法UpdateFreq()和WordFreqCounter()，其中前者会读取一个文件并统计该文件中的所有单词的词频，后者通过goroutine实现了并发统计。其并发逻辑是：对于每一个文件，创建一个goroutine，在这个goroutine内部调用UpdateFreq()方法统计对应文件的词频，当统计完成以后会将映射中每一对键值转化为Pair结构发送到results通道，并在发送完成时候发送一个空结构体的值到done通道以表示自己的任务已经完成。由于map映射结构不支持并发写操作，所以我们通过result通道来保证每次只有一个goroutine能更新映射。又因为当所有的goroutine结束以后，有可能results通道中还有没来得及处理的数据，所以在WordFreqCounter()的结尾我们又开启了一个for循环处理results通道中的剩余数据。说了这么多，我们直接写代码吧。

在$GOPATH/src/wordcount目录中创建文件wordcount.go，输入以下源码

package wordcount

import (

"bufio"

"fmt"

"io"

"log"

"os"

"sort"

"strings"

"unicode"

"unicode/utf8")

type Pair struct {

Key string

Value int

}

// PariList实现了sort接口，可以使用sort.Sort对其排序

type PairList []Pair

func (p PairList) Swap(i, j int) { p[i], p[j] = p[j], p[i] }

func (p PairList) Len() int { return len(p) }

func (p PairList) Less(i, j int) bool { return p[j].Value < p[i].Value } // 逆序

// 提取单词func SplitOnNonLetters(s string) []string {

notALetter := func(char rune) bool { return !unicode.IsLetter(char) }

return strings.FieldsFunc(s, notALetter)

}

/\*

基于map实现了类型WordCount, 并对期实现了Merge(), Report(), SortReport(), UpdateFreq(),

WordFreqCounter() 方法

\*/

type WordCount map[string]int

// 用于合并两个WordCount

func (source WordCount) Merge(wordcount WordCount) WordCount {

for k, v := range wordcount {

source[k] += v

}

return source

}

// 打印词频统计情况

func (workdcount WordCount) Report() {

words := make([]string, 0, len(workdcount))

wordWidth, frequencyWidth := 0, 0

for word, frequency := range workdcount {

words = append(words, word)

if width := utf8.RuneCountInString(word); width > wordWidth {

wordWidth = width

}

if width := len(fmt.Sprint(frequency)); width > frequencyWidth {

frequencyWidth = width

}

}

sort.Strings(words)

gap := wordWidth + frequencyWidth - len("Word") - len("Frequency")

fmt.Printf("Word %\*s%s\n", gap, " ", "Frequency")

for \_, word := range words {

fmt.Printf("%-\*s %\*d\n", wordWidth, word, frequencyWidth,

workdcount[word])

}

}

// 从多到少打印词频

func (wordcount WordCount) SortReport() {

p := make(PairList, len(wordcount))

i := 0

for k, v := range wordcount { // 将wordcount map转换成PairList

p[i] = Pair{k, v}

i++

}

sort.Sort(p) // 因为PairList实现了排序接口，所以可以使用sort.Sort()对其排序

wordWidth, frequencyWidth := 0, 0

for \_, pair := range p {

word, frequency := pair.Key, pair.Value

if width := utf8.RuneCountInString(word); width > wordWidth {

wordWidth = width

}

if width := len(fmt.Sprint(frequency)); width > frequencyWidth {

frequencyWidth = width

}

}

gap := wordWidth + frequencyWidth - len("Word") - len("Frequency")

fmt.Printf("Word %\*s%s\n", gap, " ", "Frequency")

for \_, pair := range p {

fmt.Printf("%-\*s %\*d\n", wordWidth, pair.Key, frequencyWidth,

pair.Value)

}

}

// 从文件中读取单词，并更新其出现的次数

func (wordcount WordCount) UpdateFreq(filename string) {

var file \*os.File

var err error

if file, err = os.Open(filename); err != nil {

log.Println("failed to open the file: ", err)

return

}

defer file.Close() // 本函数退出之前时，关闭文件

reader := bufio.NewReader(file)

for {

line, err := reader.ReadString('\n')

for \_, word := range SplitOnNonLetters(strings.TrimSpace(line)) {

if len(word) > utf8.UTFMax ||

utf8.RuneCountInString(word) > 1 {

wordcount[strings.ToLower(word)] += 1

}

}

if err != nil {

if err != io.EOF {

log.Println("failed to finish reading the file: ", err)

}

break

}

}

}

// 并发统计单词频次

func (wordcount WordCount) WordFreqCounter(files []string) {

results := make(chan Pair, len(files)) // goroutine 将结果发送到该channel

done := make(chan struct{}, len(files)) // 每个goroutine工作完成后，发送一个空结构体到该channel，表示工作完成

for i := 0; i < len(files); { // 有多少个文件就开启多少个goroutine, 使用匿名函数的方式

go func(done chan<- struct{}, results chan<- Pair, filename string) {

wordcount := make(WordCount)

wordcount.UpdateFreq(filename)

for k, v := range wordcount {

pair := Pair{k, v}

results <- pair

}

done <- struct{}{}

}(done, results, files[i])

i++

}

for working := len(files); working > 0; { // 监听通道，直到所有的工作goroutine完成任务时才退出

select {

case pair := <-results: // 接收发送到通道中的统计结果

wordcount[pair.Key] += pair.Value

case <-done: // 判断工作goroutine是否全部完成

working--

}

}

DONE: // 再次启动for循环处理通道中还未处理完的值

for {

select {

case pair := <-results:

wordcount[pair.Key] += pair.Value

default:

break DONE

}

}

close(results)

close(done)

}

然后在$GOPATH目录中创建文件wordfreq.go，输入以下源码:

package main

import (

"fmt"

"os"

"path/filepath"

"wordcount"

)

func main() {

if len(os.Args) == 1 || os.Args[1] == "-h" || os.Args[1] == "--help" {

fmt.Printf("usage: %s <file1> [<file2> [... <fileN>]]\n",

filepath.Base(os.Args[0]))

os.Exit(1)

}

wordcounter := make(wordcount.WordCount)

// for \_, filename := range os.Args[1:] {

// wordcount.UpdateFreq(filename)

// }

wordcounter.WordFreqCounter(os.Args[1:])

wordcounter.SortReport()

}

### **3. 编译执行**

最后我们编译该程序，输入以下命令:

$ go build wordfreq.go

当运行以上命令后，当前目录已经有了一个可执行文件wordfreq。为了验证该程序，我们使用程序统计官方包os中的英文单词的词频。Go语言一门开源的语言，所有的官方包都可以在Go语言的安装目录下看到。首先输入命令:

$ go env

GOARCH="amd64"

GOBIN=""

GOCHAR="6"

GOEXE=""

GOHOSTARCH="amd64"

GOHOSTOS="linux"

GOOS="linux"

GOPATH=""

GORACE=""

GOROOT="/usr/lib/go"

GOTOOLDIR="/usr/lib/go/pkg/tool/linux\_amd64"

TERM="dumb"

CC="gcc"

GOGCCFLAGS="-g -O2 -fPIC -m64 -pthread"

CXX="g++"CGO\_ENABLED="1"

可以看到GOROOT的指向的目录为/usr/lib/go，则os包的源码路径为/usr/lib/go/src/pkg/os，下面让我们统计下该目录下所有源文件的词频率，为了方便输出我们只打印了排名前5的单词：

$ ./wordfreq /usr/lib/go/src/pkg/os/\*.go |head -n 6

Word Frequency

err 811if 722

the 576nil 545return 539

## **作业**

请使用Go语言实现一个小型计算器程序。

到目前为止，这门课程就结束了。在这门课程中我们学习了Go语言的基础知识，如数据类型、过程面层、面向对象编程等。可以看到Go语言的关键词非常少，使用起来相当灵活，相对于C、C++等语言很大程度上提高了开发效率。想要进一步熟练Go语言，多看别人的源码，多加练习，多做项目即可，贵在坚持。