目录

[第一章 初识Go语言 6](#_Toc31873)

[1.1 Go语言的发展历史 6](#_Toc23036)

[1.1.1 知名编程语言发展简史 6](#_Toc9413)

[1.1.2 Go语言主要发展过程 7](#_Toc15405)

[1.2 Go语言的特点及优势 7](#_Toc27226)

[1.2.1 Go语言设计初衷 7](#_Toc12176)

[1.2.2 Go语言的优势 8](#_Toc10728)

[1.3 Go安装和配置 8](#_Toc29025)

[1.3.1 下载 8](#_Toc31267)

[1.3.2 windows系统下安装Go 10](#_Toc31302)

[1.3.3 Mac系统下安装及配置环境变量 10](#_Toc17110)

[1.4 Goland安装和配置 11](#_Toc1456)

[1.4.1 安装Goland开发工具 11](#_Toc16691)

[1.4.2 下载及安装 11](#_Toc13672)

[1.4.3 使用Goland 12](#_Toc3001)

[1.4.4 编写第一个程序HelloWorld 13](#_Toc14950)

[1.5 Go语言编码规范 13](#_Toc31278)

[1.5.1 HelloWorld程序解析 13](#_Toc30041)

[1.5.2 Go语言编码规范 14](#_Toc14974)

[1.5.3 Go语言关键字及保留字 15](#_Toc21651)

[1.5.4 Go 程序结构组成 15](#_Toc24956)

[第二章Go语言基础 16](#_Toc8756)

[2.1 变量 16](#_Toc32482)

[2.1.1 变量的概念 16](#_Toc19655)

[2.1.2 声明变量 16](#_Toc13818)

[2.1.3 变量初始化 17](#_Toc17840)

[2.1.4 匿名变量 17](#_Toc10448)

[2.2 数据类型 18](#_Toc11290)

[2.2.1 整型 18](#_Toc12783)

[2.2.2 浮点型 19](#_Toc28743)

[2.2.3 布尔型 20](#_Toc2282)

[2.2.4 字符串 20](#_Toc14959)

[2.2.5 字符 21](#_Toc4161)

[2.2.6 数组 22](#_Toc31127)

[2.2.7 结构体 23](#_Toc16230)

[2.2.8 指针 23](#_Toc25082)

[2.2.9 切片 25](#_Toc29161)

[2.2.10 映射（map） 25](#_Toc8200)

[2.2.11 函数 26](#_Toc27931)

[2.2.12 通道 26](#_Toc27648)

[2.3 打印格式化 26](#_Toc14244)

[2.4 数据类型转换 28](#_Toc19375)

[2.4.1 T(表达式) 28](#_Toc31451)

[2.4.2 float与int之间转换 28](#_Toc31454)

[2.4.3 int转string 28](#_Toc9195)

[2.4.4 string转int 29](#_Toc30834)

[2.5常量 29](#_Toc27396)

[2.5.1 声明方式 29](#_Toc25933)

[2.5.2 常量用于枚举（常量组） 30](#_Toc22822)

[2.5.3 iota 30](#_Toc2925)

[2.6类型别名（Type Alias） 31](#_Toc26962)

[2.6.1 概要 31](#_Toc22244)

[2.6.2 类型别名与类型定义 31](#_Toc2077)

[2.7 Go 语言运算符 32](#_Toc6037)

[2.7.1 算术运算符 （Arithmetic operator） 32](#_Toc3319)

[2.7.2 关系运算符（Relational operator） 33](#_Toc23007)

[2.7.3 逻辑运算符（Logical operator） 34](#_Toc25263)

[2.7.4 位运算符（Bitwise operator） 35](#_Toc21915)

[2.7.5 赋值运算符（Assignment operator） 38](#_Toc238)

[2.7.6 其他运算符 39](#_Toc11806)

[2.8 运算符优先级 40](#_Toc9926)

[第三章 Go语言流程控制 42](#_Toc21268)

[3.1 Go 语言流程控制 42](#_Toc1676)

[3.2 if条件判断语句 43](#_Toc4465)

[3.3 if嵌套语句 44](#_Toc21898)

[3.4 switch分支语句 45](#_Toc7382)

[3.5 Go循环语句 48](#_Toc1624)

[3.5.1 for循环语法结构 49](#_Toc23826)

[3.5.2 for循环使用案例 51](#_Toc27349)

[3.6 for嵌套循环语句 52](#_Toc26789)

[3.7 循环控制语句 55](#_Toc8988)

[3.7.1 break语句 55](#_Toc7283)

[3.7.2 continue语句 56](#_Toc10471)

[3.7.3 goto语句 58](#_Toc527)

[第四章 函数与指针 60](#_Toc828)

[4.1 函数 60](#_Toc314)

[4.1.1 声明函数 60](#_Toc19548)

[4.1.2 变量作用域 61](#_Toc32404)

[4.1.3 函数变量 62](#_Toc17479)

[4.1.4 匿名函数 64](#_Toc22666)

[4.1.5 闭包 65](#_Toc15451)

[4.1.6 可变参数 67](#_Toc32010)

[4.1.7 递归函数 68](#_Toc19660)

[4.2 指针 69](#_Toc25989)

[4.2.1 声明指针 70](#_Toc2831)

[4.2.2 空指针 72](#_Toc29760)

[4.2.3 操作指针改变变量的数值 73](#_Toc28783)

[4.2.4 使用指针作为函数的参数 73](#_Toc17679)

[4.2.5指针数组 75](#_Toc23131)

[4.2.6 指针的指针 75](#_Toc21730)

[4.3 函数的参数传递 76](#_Toc6715)

[4.3.1 值传递（传值） 76](#_Toc16084)

[4.3.2 引用传递（传引用） 76](#_Toc30921)

[4.3.3 小结 79](#_Toc668)

[第五章 Go语言内置容器 80](#_Toc28986)

[内置容器概述 80](#_Toc10124)

[5.1 数组(array) 80](#_Toc19140)

[5.1.1 数组简介 80](#_Toc26754)

[5.1.2 一维数组 80](#_Toc11075)

[5.1.3 多维数组 82](#_Toc14462)

[5.2 切片(Slice) 83](#_Toc21512)

[5.2.1 切片简介 83](#_Toc11440)

[5.2.2 切片的语法 83](#_Toc532)

[5.2.3 len() 和 cap() 函数 85](#_Toc66)

[5.2.4 append() 和 copy() 函数 87](#_Toc30038)

[5.2.5 切片高级操作 88](#_Toc18221)

[5.3 冒泡排序 89](#_Toc32417)

[5.3.1 概念： 89](#_Toc20684)

[5.3.2 冒泡排序算法的原理如下： 89](#_Toc12908)

[5.3.3 冒泡排序分析： 89](#_Toc22)

[5.3.4 冒泡排序核心代码 90](#_Toc3755)

[5.4 map 91](#_Toc20134)

[5.4.1 map简介 91](#_Toc5172)

[5.4.2 map的语法 91](#_Toc6392)

[5.4.4 delete() 函数 93](#_Toc6291)

[5.5 list 94](#_Toc1405)

[5.5.1 概述 94](#_Toc31548)

[5.5.2 声明list 95](#_Toc9597)

[5.5.3 element常用方法 95](#_Toc369)

[5.5.4 list常用方法 95](#_Toc9695)

[5.5.5 遍历list 96](#_Toc30310)

[5.5.6 list是值类型还是引用类型 96](#_Toc22347)

第一章 初识Go语言

本章目的是带领大家初步认识Go语言。先给大家讲解Go语言的发展历史，设计Go语言的初衷以及Go语言的主要特征。接下来带领大家安装Go语言的运行环境，以及Go语言集成开发工具（IDE）——Goland。之后便是Go语言版本的HelloWorld程序，以及通过对HelloWorld的分析，掌握Go语言的构成及编码规范。

本章重点为大家介绍如下的内容：

* Go语言发展历史
* Go语言核心特性
* Go的安装及环境变量配置
* 安装Goland开发环境及HelloWorld
* helloworld程序解读及Go语言编码规范

1.1 Go语言的发展历史

1.1.1 知名编程语言发展简史

B语言：Ken Thompson（肯.汤普森）。该语言得名于汤姆森的妻子Bonnie，一门剑桥于60年代中期开发的语言。

C语言：美国著名计算机专家Dennis Ritchie（丹尼斯·里奇）在1969-1973年期间发明了C语言和Unix操作系统。

C++：Bjarne Stroustrup（本贾尼·斯特劳斯特卢普）。1982年，美国贝尔实验室的Bjarne Stroustrup博士在C语言的基础上引入并扩充了面向对象的概念，发明了—种新的程序语言。为了表达该语言与C语言的渊源关系，它被命名为C++。

Python：Guido von Rossum（吉多.范.罗苏姆）。1989年，为了打发圣诞节假期，Guido开始写Python语言的编译/解释器。1994年发布1.0版本。

Java：James Gosling（詹姆斯.高斯林）。1991年开发Oak，1994年更名为Java。1995年5月正式发布。

Javascript：Brendan Eich（布兰登.艾奇)。艾奇对Java一点兴趣也没有，为了应付公司安排的任务，他于1995年5月只用10天时间就把Javascript设计出来了。

Go语言：Go的三个作者分别是：Robert Griesemer（罗伯特.格利茨默）, Rob Pike（罗伯.派克） 和 Ken Thompson（肯.汤普森）。

Robert在开发Go之前是Google V8、Chubby和HotSpot JVM的主要贡献者；

Rob主要是Unix、UTF-8、plan9的作者；

Ken主要是B语言、C语言的作者、Unix之父。



图 1.1

1.1.2 Go语言主要发展过程

2007年9月，Rob Pike（罗伯.派克） 正式命名为Go；

2008年5月，Google全力支持该项目；

2009年11月，Go将代码全部开源，它获得了当年的年度语言；

2012年3月28日，Go1.0发布；

2014年6月18日，Go语言Go 1.3版发布；

2015年8月19日，Go语言Go 1.5版发布，本次更新中移除了”最后残余的C代码”；

2018年2月16日，Go语言Go 1.10版发布。

2020年2月26日，Go1.14版本发布，Go Module已经具备在生产环境中使用的条件，并鼓励所有用户迁移到Go Module进行依赖关系管理。

1.2 Go语言的特点及优势

1.2.1 Go语言设计初衷

设计Go语言是为了解决当时Google开发遇到的问题：

* 大量的C++代码，同时又引入了Java和Python
* 成千上万的工程师
* 数以万计行的代码
* 分布式的编译系统
* 数百万的服务器

Google开发中的痛点：

* 编译慢
* 失控的依赖
* 每个工程师只是用了一个语言里面的一部分
* 程序难以维护（可读性差、文档不清晰等）
* 更新的花费越来越长
* 交叉编译困难

如何解决当前的问题和痛点？

Go希望成为互联网时代的C语言。多数系统级语言（包括Java和C#）的根本编程哲学来源于C++，将C++的面向对象进一步发扬光大。但是Go语言的设计者却有不同的看法，他们认为值得学习的是C语言。C语言经久不衰的根源是它足够简单。因此，Go语言也是足够简单。

他们当时设计Go的目标是为了消除各种缓慢和笨重、改进各种低效和扩展性。Go是由那些开发大型系统的人设计的，同时也是为了这些人服务的；它是为了解决工程上的问题，不是为了研究语言设计；它还是为了让我们的编程变得更舒适和方便。

但是结合Google当时内部的一些现实情况，如很多工程师都是C系的，所以新设计的语言一定要易学习，最好是类似C的语言；20年没有出新的语言了，所以新设计的语言必须是现代化的（例如内置GC）等情况。最后根据实战经验，他们向着目标设计了Go这个语言。

Go语言的特色：

* 没有继承多态的OO
* 强一致类型
* interface不需要显式声明(Duck Typing)
* 没有异常处理(Error is value)
* 基于首字母的可访问特性
* 不用的import或者变量引起编译错误
* 完整而卓越的标准库包

1.2.2 Go语言的优势

Go语言语法简单，包含了类C语法。它只提供了基本功能而没有多余的东西。Go引入了“defer”声明及带有goroutines和通道的并发性管理。

Go语言编译时间短，开发效率和运行效率高。开发过程中相较于 Java 和 C++呆滞的编译速度，Go 的快速编译时间是一个主要的效率优势。Go拥有接近C的运行效率和接近PHP的开发效率。Go语言可以说是开发效率和运行效率二者的完美融合，天生的并发编程支持。

Go语言支持当前所有的编程范式，包括过程式编程、面向对象编程、面向接口编程、函数式编程。程序员们可以各取所需、自由组合、想怎么玩就怎么玩。

Go 是一种非常高效的语言，高度支持并发性。Go是为大数据、微服务、并发而生的一种编程语言。作为一门语言致力于使事情简单化。它并未引入很多新概念，而是聚焦于打造一门简单的语言，它使用起来异常快速并且简单。其创新之处是 goroutines 和通道。goroutines 是 Go 面向线程的轻量级方法，而通道是 goroutines 之间通信的优先方式。

goroutines 的成本很低，只需几千个字节的额外内存，正由于此，才使得同时运行数百个甚至数千个 goroutines 成为可能。可以借助通道实现 goroutines 之间的通信。goroutines 以及基于通道的并发性方法使其非常容易使用所有可用的 CPU 内核，并处理并发的 IO。相较于 Python/Java，在一个 goroutine 上运行一个函数需要最小的代码。

Go拥有强大的编译检查、严格的编码规范和完整的软件生命周期工具，具有很强的稳定性，稳定压倒一切。

1.3 Go安装和配置

1.3.1 下载

在Mac、Windows和Linux三个平台上都支持Go语言。您可以从https://golang.org/dl/下载相应平台的二进制文件。该网站在国内不容易访问，所以可以访问https://www.studygolang.com/dl 进行安装软件的下载。

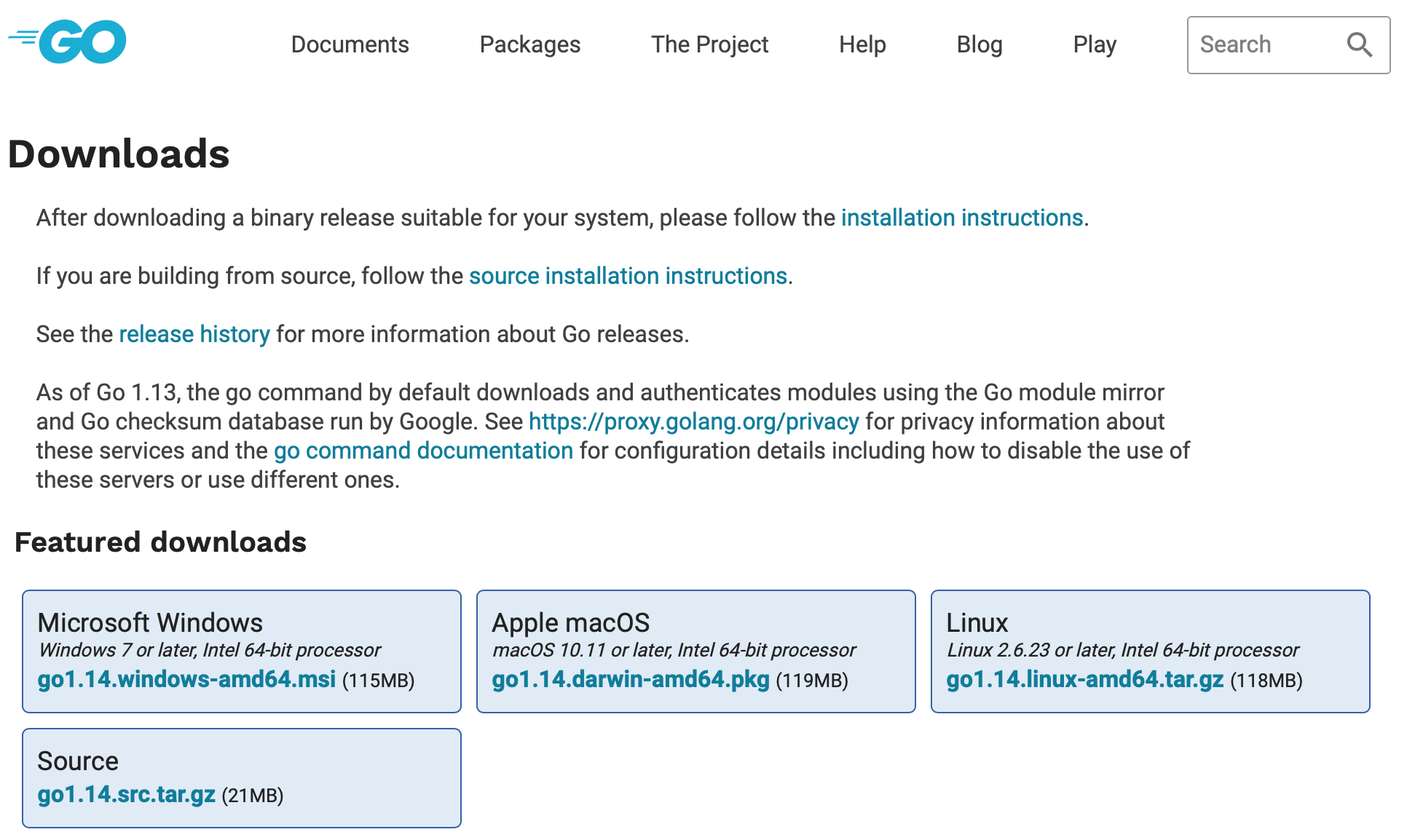


图 1.2



图 1.3

Mac OS 从https://golang.org/dl/下载osx安装程序。双击启动安装。按照提示，这应该在/usr/local/go中安装了Golang，并且还会将文件夹/usr/local/go/bin添加到您的PATH环境变量中。

Windows 从https://golang.org/dl/下载MSI安装程序。双击启动安装并遵循提示。这将在位置c中安装Golang:\Go，并且还将添加目录c:\Go\bin到您的path环境变量。

Linux 从https://golang.org/dl/下载tar文件，并将其解压到/usr/local。将/usr/local/go/bin添加到PATH环境变量中。

1.3.2 windows系统下安装Go

windows系统下安装步骤非常简单，双击.msi文件执行，一路next到底就可以完成安装，安装过程会自动配置好环境变量。

GOPATH之下主要包含三个目录: bin、pkg、src。bin目录主要存放可执行文件; pkg目录存放编译好的库文件, 主要是\*.a文件; src目录下主要存放go的源文件。

查看是否安装配置成功

使用快捷键win+R键，输入cmd，打开命令行提示符，在命令行中输入

go env # 查看得到go的配置信息

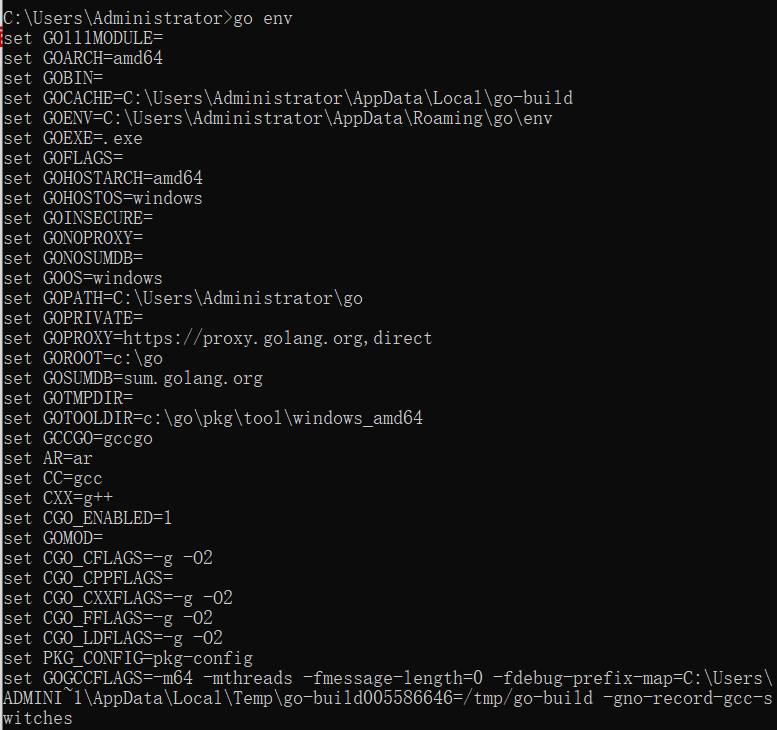


图1.4

go version # 查看go的版本号



图1.5

1.3.3 Mac系统下安装及配置环境变量

Mac系统下安装

双击pkg包，顺着指引，即可安装成功。 在命令行输入 go version，获取到go的版本号，则代表安装成功。

Mac平台下用户可以使用brew直接安装对应的go版本。

go env # 查看得到go的配置信息



图1.6

go version # 查看go的版本号



图1.7

1.4 Goland安装和配置

1.4.1 安装Goland开发工具

Goland是由JetBrains公司旨在为go开发者提供的一个符合人体工程学的新的商业IDE。这个IDE整合了IntelliJ平台的有关go语言的编码辅助功能和工具集成特点。它具有以下特点：

* 编码辅助功能
* 符合人体工程学的设计
* 工具的集成
* IntelliJ插件生态系统

1.4.2 下载及安装

官网下载地址：https://www.jetbrains.com/go/download/。下载完成后，在本地执行解压，安装。

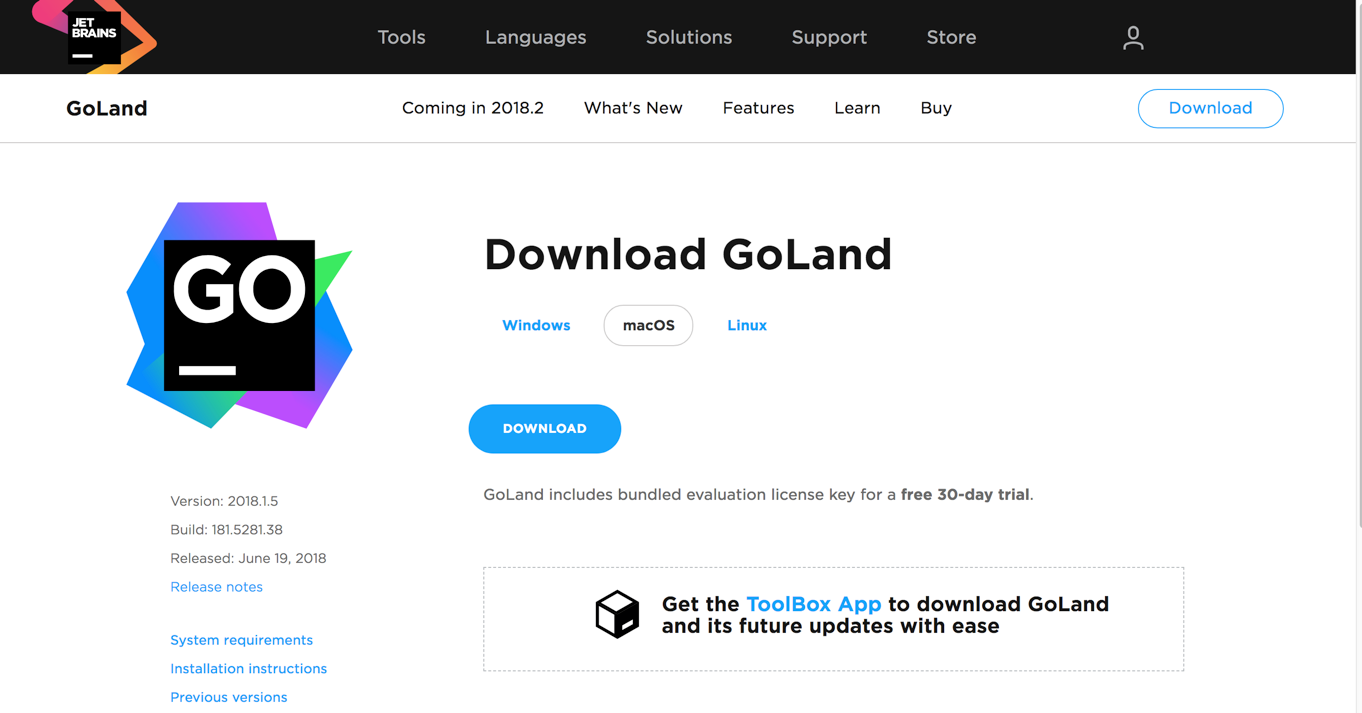


图 1.8

点击“next”按钮，选择要安装的路径，然后点击“next”，会出现安装选项。根据你自己电脑的型号，选择合适的版本后点击“next”按钮。接着保持默认的程序启动目录，点击“install”进行安装。整个安装过程很快，几乎一路next到底。

1.4.3 使用Goland

1.打开Goland工具

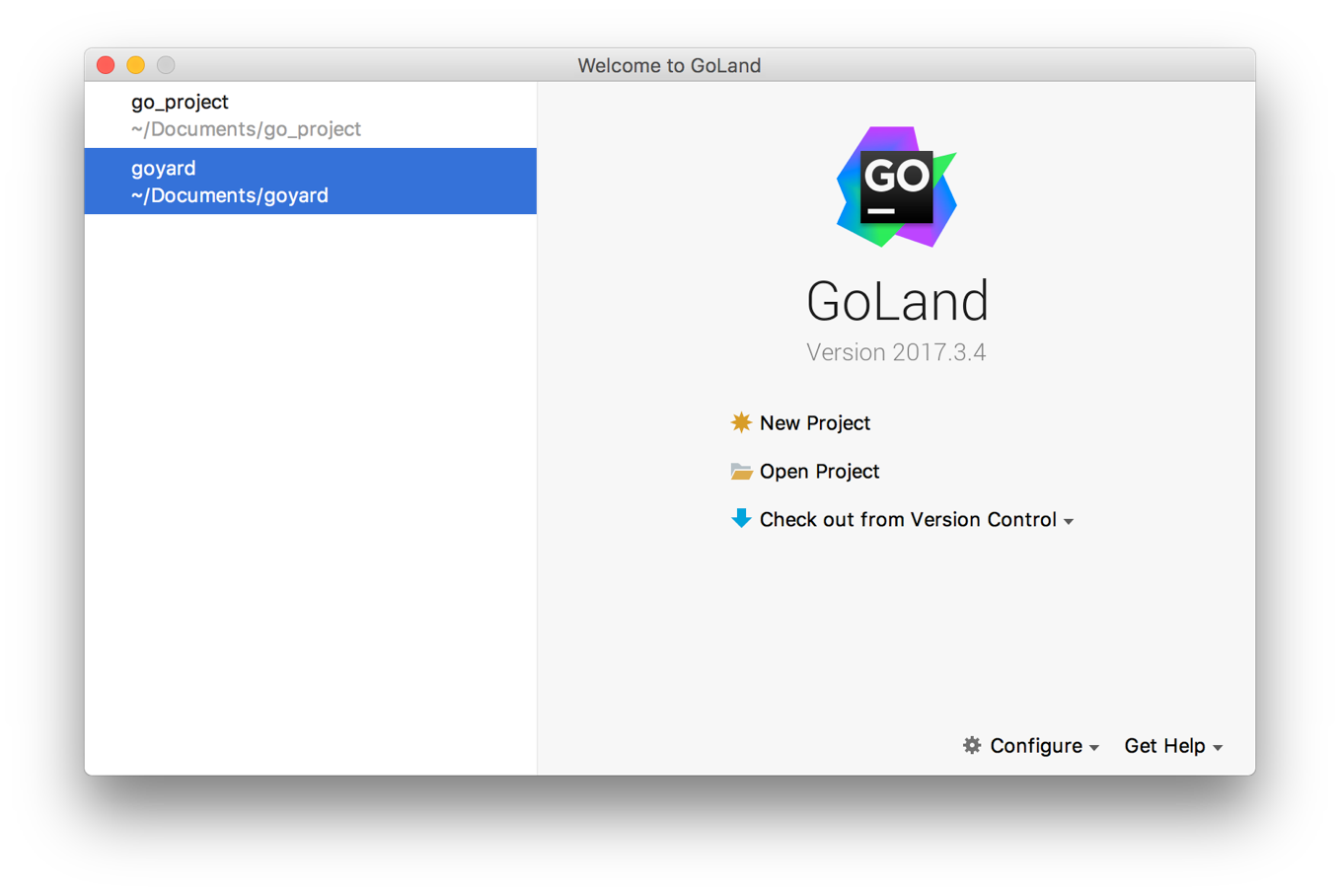


图 1.9

2.创建项目：

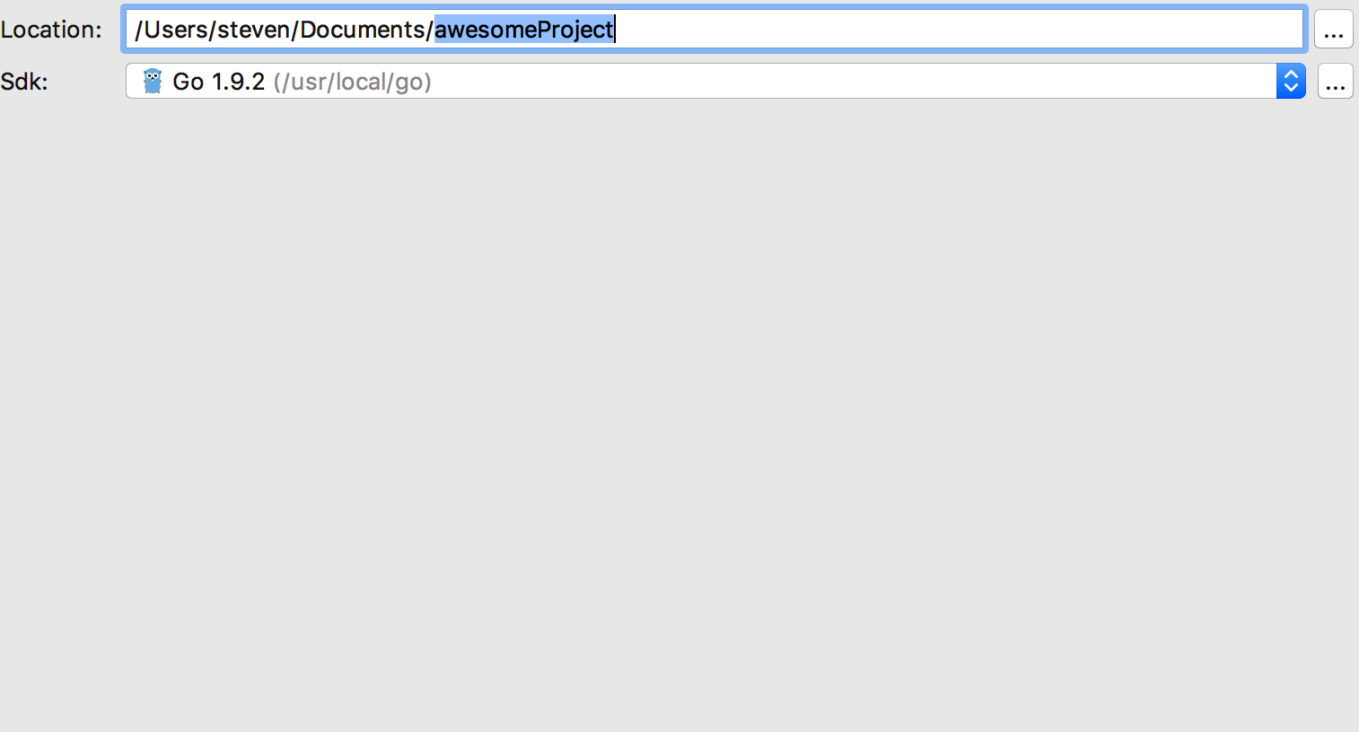


图 1.10

1.4.4 编写第一个程序HelloWorld

1.打开编辑器创建一个新的helloworld.go文件，并输入以下内容：

package main

import "fmt"

func main() {

/\* 输出 \*/

fmt.Println("Hello, World!")

}

2.执行go程序

执行go程序由几种方式

方式一：使用go run命令

step1：使用快捷键win+R，输入cmd打开命令行提示符

step2：进入helloworld.go所在的目录

step3：输入go run helloworld.go命令并观察运行结果。

方式二：使用go build命令

step1：使用快捷键win+R，输入cmd打开命令行提示符

step2：进入helloworld.go所在的目录

step3：输入go build helloworld.go命令进行编译，产生同名的helloworld.exe文件

step4：输入helloworld.exe，执行

1.5 Go语言编码规范

1.5.1 HelloWorld程序解析

package main

import "fmt"

func main() {

/\* 这是第一个简单的程序 \*/

fmt.Println("Hello, World!")

}

程序解析

1. package main 定义了包名。必须在源文件中非注释的第一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 应用程序都包含一个名为 main 包。

2. import "fmt" 告诉 Go 编译器这个程序需要使用 fmt 包，fmt 包实现了格式化 IO（输入/输出）的函数。

3. func main() 是程序入口。main 函数是每一个可执行程序所必须包含的，一般来说都是在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。

4. /\*...\*/ 是注释，在程序执行时将被忽略。单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 /\* 开头，并以 \*/ 结尾，且不可以嵌套使用，多行注释一般用于文档描述或代码片段。

5. fmt.Println(...) 可以将字符串输出到控制台，并在最后自动增加换行字符 \n。 使用 fmt.Print("hello, world\n") 可以得到相同的结果。

1.5.2 Go语言编码规范

1.注释

单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释

多行注释也叫块注释，均已以 /\* 开头，并以 \*/ 结尾，且不可以嵌套使用，多行注释一般用于文档描述或注释成块的代码片段

2.标识符

标识符用来命名变量、类型等程序实体。一个标识符实际上就是一个或是多个字母(A~Z和a~z)数字(0~9)、下划线\_组成的序列，但是第一个字符必须是字母或下划线而不能是数字。

Go不允许在标识符中使用@、$和%等标点符号。

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。

以下是无效的标识符：

* 1xy（以数字开头）
* case（Go 语言的关键字）
* chan（Go 语言的关键字）
* x+y（运算符是不允许的）

3.Go 语言的空格

Go 语言中变量的声明必须使用空格隔开，如：var age int;语句中适当使用空格能让程序更易阅读。在变量与运算符间加入空格，程序看起来更加美观，如：a = x + y;

4.语句的结尾

在 Go 程序中，一行代表一个语句结束。Go语言中是不需要类似于Java需要分号结尾，因为这些工作都将由 Go 编译器自动完成；

如果打算将多个语句写在同一行，它们则必须使用分号“;”人为区分，但在实际开发中并不鼓励这种做法。

5.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。

以一个大写字母开头，那么使用这种形式的标识符的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。

如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的，像面向对象语言中的 private 。

1.5.3 Go语言关键字及保留字

下面列举了 Go 代码中会使用到的 25 个关键字或保留字：

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

表1.1 Go语言关键字或保留字

除了以上介绍的这些关键字，Go 语言还有 36 个预定义标识符：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| append | bool | byte | cap | close | complex | complex64 | complex128 | uint16 |
| copy | false | float32 | float64 | imag | int | int8 | int16 | uint32 |
| int32 | int64 | iota | len | make | new | nil | panic | uint64 |
| print | println | real | recover | string | true | uint | uint8 | uintptr |

表1.2 Go语言预定义标识符

1.5.4 Go 程序结构组成

Go 文件的基本组成包括：包声明、引入包、函数、变量、语句 & 表达式、注释。

// 当前程序的包名

package main

// 导入其他包

import . "fmt"

// 常量定义

const PI = 3.14

// 全局变量的声明和赋值

var name = "gopher"

// 一般类型声明

type newType int

// 结构的声明

type gopher struct{}

// 接口的声明

type golang interface{}

// 由main函数作为程序入口点启动

func main() {

Println("Hello World!")

}

第二章Go语言基础

本章给大家讲解Go语言的基本语法，包括变量、常量、数据类型及运算符。学习任何一门编程语言，其基本语法无外乎这几部分。但是值得注意的是，本章包含了其他编程语言所没有的内容。首先是变量的多重赋值，其次是匿名变量，第三点是常量中iota的用法。

本章重点为大家介绍如下的内容：

* 变量声明、初始化及赋值
* 常量声明与初始化
* 数据类型
* 运算符
* 表达式

2.1 变量

2.1.1 变量的概念

变量是计算机语言中储存数据的抽象概念。变量的功能是存储数据。变量通过变量名访问；

变量的本质是计算机分配的一小块内存，专门用于存放指定数据，在程序运行过程中该数值可以发生改变；

变量的存储往往具有瞬时性，或者说是临时存储，当程序运行结束，存放该数据的内存就会释放，而该变量就会消失；

Go 语言的变量名由字母、数字、下划线组成，首个字符不能为数字;

Go语法规定，定义的局部变量若没有被调用则编译错误。

2.1.2 声明变量

标准变量声明格式

var <变量名> [变量类型]

例：

var str string

var关键字的另一种用法是将多个需要声明的变量放在一起，避免程序员重复写var关键字。

var (

<变量名> <变量类型>

<变量名> <变量类型>

<变量名> <变量类型>

)

例：

var (

a int

b string

c []float32

d func() bool

e struct {

x int

y string

}

)

与C语言不同，Go语言变量声明语句不需要使用分号作为结束符。

Go语言变量声明后默认值如下：

整形和浮点型变量默认值：0；

字符串默认值为空字符串；

布尔型默认值为false；

函数、指针变量初始值为nil。

2.1.3 变量初始化

初始化变量的标准格式：

var <变量名> [类型] = <表达式>

例：

var str string = "this is string"

初始化变量的编译器自动推断类型格式：

var <变量名> = <表达式>

例：

var str = "this is string"

初始化变量的简短声明格式（短变量声明格式）：

<变量名> := <表达式>

例：

s1 := "this is s1"

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。使用赋值操作符声明语句省略了var 关键字，变量类型将由编译器根据表达式自动推断。赋值操作符是声明变量的首选形式，要注意它只能被用在函数体内，不可以应用于全局变量的声明与赋值。

使用赋值操作符赋值的变量必须是没有定义过的变量，如果定义过，将会发生编译错误。在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，那么即便有其它变量名可能是重复声明的，编译器也不会报错。

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语言比其他语言更灵活、更简单。

曾经在不支持多重赋值的编程语言中，我们需要交换两个变量的值，使用如下方法：

t = i; i = j; j = t;

而在Go语言中，多重赋值帮我们解决了这个头痛的难题，我们只需要使用下面的代码就只可以实现变量互换。

i, j = j, i

多重赋值交换变量的原理是将两个被赋值的变量的值，都存储在临时变量里，然后再用临时变量去赋值。

2.1.4 匿名变量

Go语言的函数允许返回多个值，当我们只需要使用部分返回值的时候，可以使用匿名变量，用“\_”下划线替换即可。

匿名变量不占用命名空间，不会分配内存。

例

i , \_ := noname()

2.2 数据类型

Go 语言的数据类型分为基本数据类型与复合数据类型。

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）。

2.2.1 整型

整型就最基础的数据类型，其按长度可分为：int8、int16、int32、int64、int；无符号的整型数分为：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。

|  |  |
| --- | --- |
| 序号 | 类型和描述 |
| 1 | uint8 无符号 8 位整型 (0 到 255) |
| 2 | uint16 无符号 16 位整型 (0 到 65535) |
| 3 | uint32 无符号 32 位整型 (0 到 4294967295) |
| 4 | uint64 无符号 64 位整型 (0 到 18446744073709551615) |
| 5 | int8 有符号 8 位整型 (-128 到 127) |
| 6 | int16 有符号 16 位整型 (-32768 到 32767) |
| 7 | int32 有符号 32 位整型 (-2147483648 到 2147483647) |
| 8 | int64 有符号 64 位整型 (-9223372036854775808 到 9223372036854775807) |

表2.1 Go语言中整型

还有其他数字类型

|  |  |
| --- | --- |
| 序号 | 类型和描述 |
| 1 | byte 类似 uint8 |
| 2 | rune 类似 int32 |
| 3 | uint 32或64位 |
| 4 | int 与 uint 一样大小 |
| 5 | uintptr 无符号整型，用于存放一个指针 |

表2.2 Go语言中其他数字整型

int8与int是完全不同的两种数据类型，编译器不能做自动的类型转换。

例：

func main() {

var v int8

v1 := 10 //系统自动推断数据类型为int

v = v1 //此处报错

fmt.Println(v)

}

报错内容如下：

cannot use v1 (type int) as type int8 in assignment

错误提示告诉我们“不能使用int类型作为int8类型的参数”，所以如果要使之成立就需要使用强制类型转换解决此问题。修改后的代码如下：

例：

func main() {

var v int8

v1 := 10

v = int8(v1)

fmt.Println(v)

}

同样int和uint也是完全不同的两种数据类型，编译器同样不能做自动的类型转换。

在表2.1看到的uint8这个数据类型也就是byte类型，如下例子的方式是可行的：

例：

func main() {

var v byte

var v1 uint8

v1 = 10

v = v1

fmt.Println(v)

}

既然二者为同一类型，为什么要给它们提供不同的名字呢？

一般情况下我们使用byte类型来说明此变量是一个原始的数据而不是一个小的整型数。

uintptr是一种无符号整数类型，没有指定大小，但是足够容纳指针。此类型在底层编程中使用较多，与C语言函数库或者操作系统接口交互的地方会使用，其它情况使用较少。

2.2.2 浮点型

Go语言支持4种浮点型数：float32、float64、complex64（32 位实数和虚数）、complex128（64 位实数和虚数）。float32的最大范围是3.4e38，用常量定义是：math.MaxFloat32；float64的最大范围是1.8e308，用常量定义是：math.MaxFloat64。

例：

func main() {

fmt.Println(math.MaxFloat32) //输出：3.4028234663852886e+38

fmt.Println(math.MaxFloat64) //输出：1.7976931348623157e+308

}

通常情况下我们建议使用float64，毕竟flaot64的精度相对较大，累计计算的误差相对较小。float32的有效bit位只有23个，其它的用于指数和符号；当整数大于23bit能表达的范围时，float32的表示将出现误差，例如：

例：

func main() {

var f float32 = 1 << 24 // 1 << 24

fmt.Println(f == f+1) // "true"!

fmt.Println(f) // 1.6777216e+0.7

fmt.Println(f + 1) // 1.6777216e+0.7

}

上面的例子是显示指数形式的浮点数，我们平常在使用的时候使用的是类似3.1415926这样的浮点数，那我们需要如何输出呢？

例：

func main() {

var f float32 = 123453.1415926

fmt.Printf("%8.2f", f) // 123453.14

fmt.Printf("%3.1f", f) // 123453.1

fmt.Printf("%3.4f", f) // 123453.1406

}

注意，如果定义的时候不加小数点，会被推导为整型。

2.2.3 布尔型

声明方式：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。

例：

func main() {

var v1 bool

fmt.Println(v1) //false

v1 = true

fmt.Println(v1) //true

v1 = (1 == 2)

fmt.Println(v1) //false

v2 := (2 == 2)

fmt.Println(v2) //true

var a, b int

a = 2

b = 2

v3 := (a == b)

fmt.Println(v3) //true

}

由例子可见，定义了v1变量后，系统默认赋值为false，同时我们在v3变量可以看到，我们将一个逻辑判断表达式赋值给v3，v3会被自动推导为bool类型。

2.2.4 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。

例：

func main() {

var s string

s1 := "this is String"

ch := s1[0]

fmt.Println(s) //这里什么也不会显示

fmt.Println(s1) //这里输出this is String

fmt.Println(reflect.TypeOf(ch)) //这里输出uint8，也就是byte类型

fmt.Println(ch) //这里输出ascii码116

fmt.Printf("%c", ch) //这里输出字母t

}

从上面的例子我们可以看到，只定义一个字符串，在不赋值的情况下，输出的是空内容；我们可以使用s1[N]的方式，来获取第N个字符（程序员的世界数字从0开始哦^\_^）；s1[N]方式取出来的值类型是uint8类型，也就是我们之前说过的byte类型，其内容为字符串的ascii码，我们使用printf的“%c”模式，将其转换成字符串，就可以得到我们获取到的字符串了。

如果我们可以使用s1[N]的方式得到字符串的第N个字符，那么我们是不是可以通过s1[N] = “a”的方式来赋值呢？我们看下面的例子：

例：

func main() {

var s = "this is String"

ch := s[1]

fmt.Printf("%c", ch)

s[1] = "k" //cannot assign to s[1]

}

这个例子会报一个编译错误，告诉我们不能给s[1]赋值。

cannot assign to s[1]

字符串中可以使用的转义字符如下：

* \r 回车
* \n 换行符
* \t 制表符TAB
* \' 单引号
* \" 双引号
* \\ 反斜杠

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。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。

2.2.5 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：

1.byte型：其实是uint8的别名。代表了一个ASCII码的一个字符。

2.rune型：其实就是int32。代表一个UTF-8字符。当需要处理中文等unicode字符集时需要用到rune类型。

var a byte = 'a'

var b rune = '一'

例：

func main() {

var s1 rune

var s2 rune

var s3 rune

s1 = 'a'

s2 = "a"//此处报错cannot use "a" (type string) as type rune is assignment

s3 = 'abc'//此处报错 invalid character literal(more than one character)

fmt.Println(s1) //97

fmt.Printf("%c", s1) //a

}

上例中，我们给s2赋值为”a”的时候报错为不能使用字符串为字符类型赋值，而我们给s3赋值为‘abc’的时候，编译错误为无效的字符，超过了一个字符。所以我们可以得出结论rune类型在使用及赋值的时候，必须使用单引号包起来，并且只能是一个字符，不允许赋值多个。

我们给s1赋值为’a’,实际我们刚才说过rune是int32的等价类型，所以直接输出我们得到的是字符的ascii码97。

2.2.6 数组

数组是一个由固定长度的特定类型元素组成的序列，一个数组可以由N（N >= 0）个元素组成。数组的长度必须是固定的不可变的，所以很少会直接使用数组；与数组对应的两种引用类型切片类型与字典类型是可以增长和收缩的动态序列，它们的使用也更灵活，所以这两种数据类型也是未来我们最常用的数据类型。但是要理解这两种数据类型，我们需要先从理解数组类型开始。

数组的每个元素都可以通过索引下标来访问，还记得我们之前说过，程序员的世界从0开始。所以索引下标也是从0开始的。

大家可以回忆一下在字符串中，我们是如何使用下标获取字符串中单个字符的。数组的基础操作，也和字符串中获取字符的方式类似。我们通过例子看一下如何构建数组，如何对数组进行基本操作。

例：

func main() {

var a [8]int //定义长度为8整型数组

//定义长度为5的数组并初始化

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

fmt.Println(a[0]) //输出：0

fmt.Println(b[len(b)-1]) //输出:4

fmt.Println(b) //输出：[0 1 2 3 4]

b[0] = 100 //修改b数组第0个元素的值

fmt.Println(b[0]) //输出：100

}

数组遍历的方法有两种：

例：

func main() {

var b [5]int = [5]int{0, 1, 2, 3, 4} //定义长度为5的数组并初始化

//以下为第一种遍历方式

for i := 0; i < len(b); i++ {

fmt.Println(b[i])

//换行输出：

//0

//1

//2

//3

//4

}

//以下为第二种遍历方式

for key, value := range b {

fmt.Printf("下标是：%d 值是：%d\n", key, value)

//换行输出：

//下标是：0 值是：0

//下标是：1 值是：1

//下标是：2 值是：2

//下标是：3 值是：3

//下标是：4 值是：4

}

}

数组的长度必须是固定的，所以我们在给数组赋值的时候要注意a[8]int与b[5]int是完全不同的两种类型。

例：

func main() {

var a [8]int //定义长度为8整型数组

var b [5]int = [5]int{0, 1, 2, 3, 4} //定义长度为5的数组并初始化

a = b //编译错误：cannot use b(type [5]int) as type [8]int in assignment

}

当我们尝试将长度为5的数组赋值给长度为8的数组时，编译报错，告诉我们不允许这样的操作。

2.2.7 结构体

结构体是一种聚合的数据类型，由N（N >=0）个任意类型的值聚合成的实体，每个值被称为结构体的成员或属性。

结构体的基本定义与使用方法如下：

例：

type Object struct {

Hash string

length int

}

func main() {

var obj Object

obj.length = 10

obj.Hash = "45454afasf4asdf1a"

fmt.Println(obj.length) //输出：10

}

我们使用type关键字定义了一个Object的结构体（struct），然后将成员用“{”、“}”包含。任何数据类型都可以成为结构体的成员的类型，接下来的内容中，我们都将使用属性来替代成员这个词，大家只要看到属性，就一定要想到，这是和结构体有关。

当属性名的首字母大写时，我们就可以在任何位置上访问这个属性；如果属性首字母是小写，那这个属性就是私有的，只能在结构体声明的包下进行访问或赋值。上例中，结构体Object是在main名声明的，所以可以在main函数中直接访问操作。如果在别的包中，就不能访问length属性，只能访问Hash属性。

2.2.8 指针

Go语言中的指针相对于C语言中的指针简单了很多，对于开发者来说我们可以不必担心因为指针的错误使用而造成的各种各样奇奇怪怪的问题。

指针是一种指向变量内存地址的数据类型。一个指针变量可以指向任何一个变量的内存地址。

在Go语言中取地址的运算符为“&”，我将通过一个短例子查看地址与指针基本的样子。

例：

func main() {

//定义一个整型变量

var a int = 10

//定义一个指针变量

var p \*int

//指针p指向变量a的地址

p = &a

fmt.Println(a) //输出：10

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：10

\*p = 100

fmt.Println(a) //输出：100

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：100

fmt.Println(\*p == a) //输出：true

}

上面例子中我们使用指针p指向了变量a的地址（&a），通过输出我们发现指针p的值就是变量a的地址，而\*p的值是变量a的值；当我们改变了\*p的值的时候，变量a的值也随之改变，因此我们可以认为在这种情况下，\*p等价于变量a。

我们刚才说到，每个变量都有一个地址，指针本身也是一个变量，那么指针是不是也有地址呢？指针是不是也可以被另一个指针指向呢？我们看下面的例子。

例：

func main() {

//定义一个整型变量

var a int = 10

//定义一个指针变量

var p \*int

//指针p指向变量a的地址

p = &a

fmt.Println(a) //输出：10

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：10

fmt.Println(&p) //输出：0xc42000c028

//定义一个指向指针的指针

var ptr \*\*int

//将指针ptr指向指针p的地址

ptr = &p

fmt.Println(ptr) //输出：指针ptr的地址为0xc42000c028

fmt.Println(\*ptr) //输出：0xc420014100

fmt.Println(\*\*ptr) //输出：10

}

上例中我们定义了一个指向指针的指针ptr，并使其指向指针p的地址。

我们在输出的时候发现，ptr的值为指针p的地址；\*ptr的值为指针p的值，也就是变量a的地址；\*\*ptr的值为指针\*p的值，也就是变量a的值10。

2.2.9 切片

切片也叫数组切片，通过上面对数组的介绍，我们知道数组的长度一旦确定就不可以更改了。这种情况下是无法满足我们的真实需求的。

Go语言为我们提供了数组切片（slice）来补充数组的不足。

切片是一个可以随时动态扩充的存储空间，用于存放相同类型的数据；并且切片可以随意传递而不会导致所管理的元素被重复复制。

本书第5章将重点讲解切片的使用。

2.2.10 映射（map）

map是一种键/值（key/value）对形式的数据结构，它是一种未排序的集合，所有的key都是唯一的，通过key可以精准定位数据的位置，并对其进行相关操作。

在Go语言中，一个map就是一个哈希表的引用，所有的key的类型必须相同，value的类型也必须相同。要注意所有的的key都必须使用支持”==”比较运算符的类型。

需要注意的是浮点型最好不要使用在key中，一些误差会导致判断失败，影响程序的正常运行。value的类型则没有任何限制。

例：

func main() {

//创建一个map,key为string类型，value为string类型

m := make(map[string]string)

//给map增加值

m["username"] = "admin"

m["sex"] = "man"

m["age"] = "20"

fmt.Println(m) //输出：map[username:admin sex:man age:20]

//删除键值

delete(m, "age")

fmt.Println(m) //输出：map[username:admin sex:man]

//查询键值是否存在

value, ok := m["username"]

if ok {

fmt.Println(value) //输出：value的结果，此处为admin

} else {

fmt.Println("nil") //输出：如果ok返回false，则输出nil

}

}

通过上面的例子我们可以了解到如何创建map，如何给map增加数据、修改数据、删除数据以及查询数据是否存在 。这一个示例，已经基本上包含了map的大部分操作。

2.2.11 函数

在Go语言中，函数也作为一种数据类型。既然是数据类型那么函数也就可以当作一个值来传递和使用。

例：

func main() {

//定义一个匿名函数，赋值给fun变量

fun := func(a, b int) int {

if a > b {

return a

}

return b

}

//调用匿名函数并保存返回值

fmt.Println(fun(5, 10)) //输出：10

fmt.Println(reflect.TypeOf(fun)) //输出:func()int,int)int

}

在上面例子中，我们定义了一个匿名函数，将其赋值给变量fun，我们在程序结尾输出类型的时候，看到类型是func类型，也就是函数类型；函数类型的调用方式也与常规函数的方法一样，关于函数详细的使用方式，我们将在第4章函数中进行详细的解读。

2.2.12 通道

通道(channel)是Go语言中为我们提供的用于在进程间通信的一种方式，我们使用channel在多个不同的进程间传递消息。

channel本身有类型的限制，一个channel只能传递一种类型的数据，在声明的时候就必须指定好要传递的类型。本书第12章并发编程中将会对channel的使用进行详细的解释说明，需要提前了解的读者请移步第12章

2.3 打印格式化

Print、Println 、Printf 、Sprintf 、Fprintf都是fmt 包中的公共方法，在需要打印信息时需要用到这些函数，本节主要以Printf方法格式化输出为主。

|  |  |
| --- | --- |
| Print | 输出到控制台(不接受任何格式化，它等价于对每一个操作数都应用 %v) |
| Println | 输出到控制台并换行 |
| Printf | 只可以打印出格式化的字符串。只可以直接输出字符串类型的变量 |
| Sprintf | 格式化并返回一个字符串而不带任何输出。 |
| Fprintf | 格式化并输出到 io.Writers 而不是 os.Stdout。 |

表2.3 格式化打印介绍

1.通用

%v 值的默认格式表示；

%+v 类似%v，但输出结构体时会添加字段名；

%#v 值的Go语法表示；

%T 值的类型的Go语法表示。

2.布尔值

%t 单词true或false。

3.整数

%b 表示为二进制；

%c 该值对应的unicode码值；

%d 表示为十进制；

%8d 表示该整型长度是8，不足8则在数值前补空格。如果超出8，则以实际为准；

%08d 数字长度是8，不足8位的，在数字前补0。如果超出8，则以实际为准；

%o 表示为八进制；

%q 该值对应的单引号括起来的go语法字符字面值，必要时会采用安全的转义表示；

%x 表示为十六进制，使用a-f；

%X 表示为十六进制，使用A-F；

%U 表示为Unicode格式：U+1234，等价于"U+%04X"。

4.浮点数与复数的两个组分

%b 无小数部分、二进制指数的科学计数法，如-123456p-78；参见strconv.FormatFloat；

%e （=%.6e）有6位小数部分的科学计数法，如-1234.456e+78；

%E 科学计数法，如-1234.456E+78；

%f （=%.6f）有6位小数部分，如123.456123；

%F 等价于%f；

%g 根据实际情况采用%e或%f格式（以获得更简洁、准确的输出）；

%G 根据实际情况采用%E或%F格式（以获得更简洁、准确的输出）。

5.字符串和[]byte

%s 直接输出字符串或者[]byte；

%q 该值对应的双引号括起来的go语法字符串字面值，必要时会采用安全的转义表示；

%x 每个字节用两字符十六进制数表示（使用a-f）；

%X 每个字节用两字符十六进制数表示（使用A-F）。

6.指针

%p 表示为十六进制，并加上前导的0x。

没有%u。整数如果是无符号类型自然输出也是无符号的。类似的，也没有必要指定操作数的尺寸（int8，int64）。

宽度通过一个紧跟在百分号后面的十进制数指定，如果未指定宽度，则表示值时除必需之外不作填充。精度通过（可选的）宽度后跟点号后跟的十进制数指定。如果未指定精度，会使用默认精度；如果点号后没有跟数字，表示精度为0。举例如下：

%f: 默认宽度，默认精度；

%9f 宽度9，默认精度；

%.2f 默认宽度，精度2；

%9.2f 宽度9，精度2；

%9.f 宽度9，精度0。

7.其它flag

'+' 总是输出数值的正负号；对%q（%+q）会生成全部是ASCII字符的输出（通过转义）；

' ' 对数值，正数前加空格而负数前加负号；

'-' 在输出右边填充空白而不是默认的左边（即从默认的右对齐切换为左对齐）；

'#' 切换格式；

八进制数前加0（%#o），十六进制数前加0x（%#x）或0X（%#X），指针去掉前面的0x（%#p）；

对%q（%#q），如果strconv.CanBackquote返回真会输出反引号括起来的未转义字符串；

对%U（%#U），输出Unicode格式后，如字符可打印，还会输出空格和单引号括起来的go字面值；

对字符串采用%x或%X时（% x或% X）会给各打印的字节之间加空格；

'0' 使用0而不是空格填充，对于数值类型会把填充的0放在正负号后面。

2.4 数据类型转换

2.4.1 T(表达式)

采用数据类型前置加括号的方式进行类型转换。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。需要注意的是布尔型无法与其他类型进行转换。

例：

func main(){

var sum int = 17

var count int = 5

var mean float32

mean = float32(sum)/float32(count)

fmt.Printf("mean 的值为: %f\n",mean)//输出：mean 的值为: 3.400000

}

2.4.2 float与int之间转换

需要注意float转int时精度的损失，小数点后的数据会被舍弃。

例：

func main(){

var mean float32 = 3.6845822

fmt.Println(int(mean))//输出：3

}

2.4.3 int转string

int转string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”。

ASCII字符集中数字的10进制范围是[30 - 39]

ASCII字符集中大写字母的10进制范围是[65 - 90]

ASCII字符集中小写字母的10进制范围是[97 - 122]

Unicode字符集中汉字的范围是[4e00-9fa5]，10进制范围是[19968 - 40869]

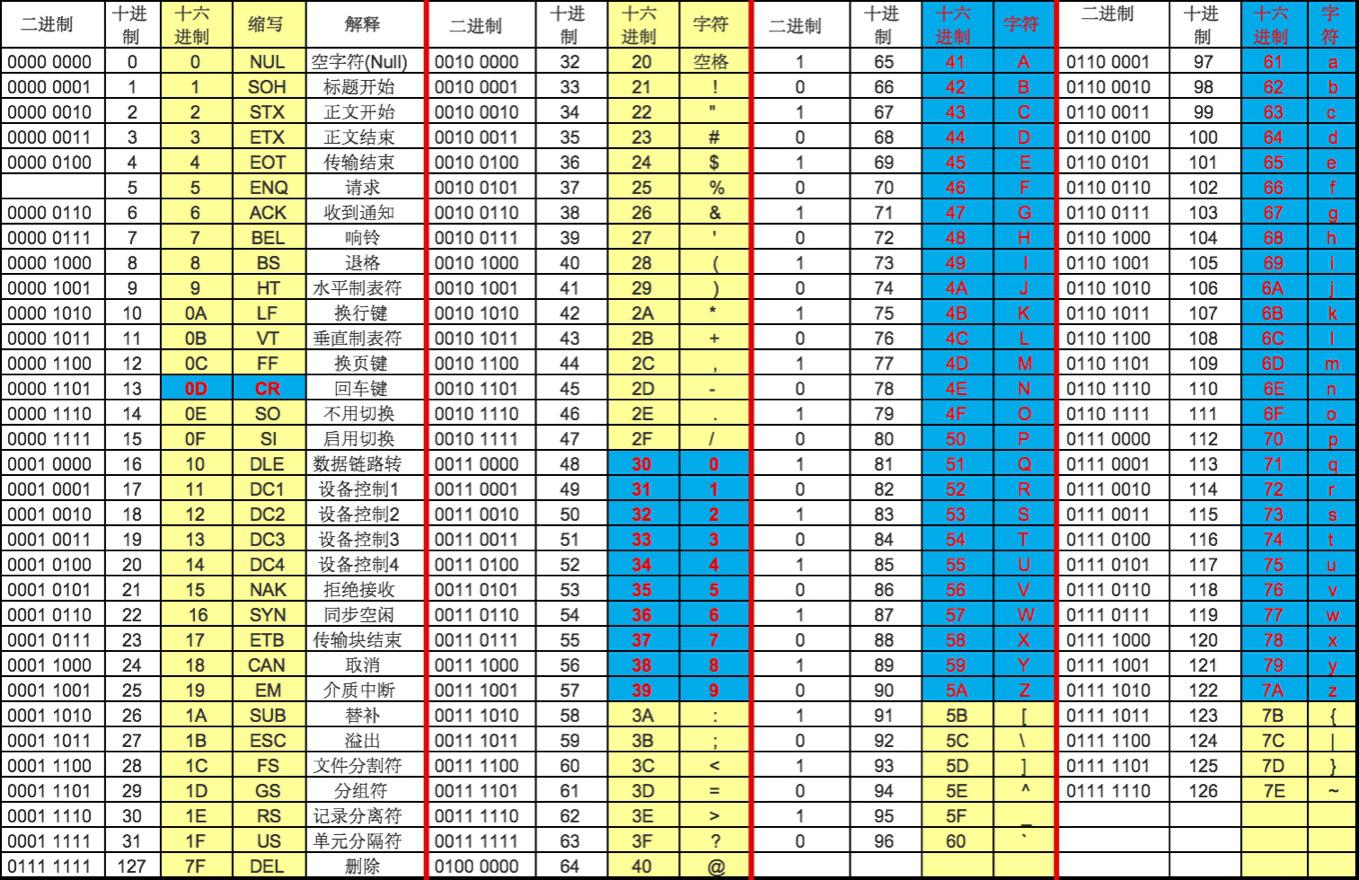


图 2.1

2.4.4 string转int

go语言中不允许string类型转换为int类型。

例：

func main(){

var mean string = "this is string"

fmt.Println(int(mean))

}

上面代码在编译过程中报错

cannot convert mean (type string) to type int

2.5常量

2.5.1 声明方式

相对于变量，常量是恒定不变的值，例如圆周率。常量是一个简单值的标识符，在程序运行时，不会被修改。常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

常量的定义格式：

const <标识符> [类型] = <值>

可以省略类型说明符 [type]，因为编译器可以根据变量的值来自动推断其类型。

显式类型定义：

const B string = "Steven"

隐式类型定义：

const C = "Steven"

多个相同类型的声明可以简写为：

const WIDTH , HEIGHT = value1, value2

常量定义后未被使用，不会在编译时出错。

2.5.2 常量用于枚举（常量组）

例如以下格式：

const (

Unknown = 0

Female = 1

Male = 2

)

数字 0、1 和 2 分别代表未知性别、女性和男性。

常量组中如果不指定类型和初始值，则与上一行非空常量的值相同。

const (

a = 10

b

c

)

打印a、b、c，输出：10 10 10。

2.5.3 iota

iota是特殊常量值，是一个系统定义的可以被编译器修改的常量值。iota只能用在常量赋值中。

在每一个const关键字出现时，被重置为0，然后每出现一个常量，iota所代表的数值会自动增加1。iota可以理解成常量组中常量的计数器，不论该常量的值是什么，只要有一个常量，那么iota就加1。

iota 可以被用作枚举值：

const (

a = iota

b = iota

c = iota

)

println(a, b, c)

打印输出：0 1 2

第一个 iota 等于 0，每当 iota 在新的一行被使用时，它的值都会自动加 1；所以 a=0, b=1, c=2

常量组中如果不指定类型和初始值，则与上一行非空常量的值相同。所以上述的枚举可以简写为如下形式：

const (

a = iota

b

c

)

println(a, b, c)

打印输出：0 1 2

例：

const (

i = 1<<iota

j = 3<<iota

k

l

)

func main() {

fmt.Println("i=",i)

fmt.Println("j=",j)

fmt.Println("k=",k)

fmt.Println("l=",l)

}

打印输出结果：

i= 1

j= 6

k= 12

l= 24

例：

const (

a1 = '一'

b1

c1 = iota

d1

)

func main() {

fmt.Println(a1, b1, c1, d1)

}

打印输出结果：

19968 19968 2 3

2.6类型别名（Type Alias）

2.6.1 概要

类型别名是Go1.9版本添加的新功能。主要用于代码升级、迁移中类型的兼容性问题。

在Go1.9版本前内建类型定义的代码是：

type byte uint8

type rune int32

而在Go1.9版本之后变更为：

type byte = uint8

type rune = int32

2.6.2 类型别名与类型定义

类型别名的语法格式：

type 类型别名 = 类型

定义类型的语法格式：

type 新的类型名 类型

例：

type NewString string

该语句是将NewString定义为string类型。通过type关键字，NewString会形成一种新的类型。NewString本身依然具备string的特性。

type StringAlias = string

该语句是将StringAlias定义为string的一个别名。使用StringAlias与string等效。别名类型只会在代码中存在，编译完成时，不会有别名类型。

2.7 Go 语言运算符

运算符用于在程序运行时执行数学或逻辑运算。

Go 语言内置的运算符有：

* 算术运算符
* 关系运算符
* 逻辑运算符
* 位运算符
* 赋值运算符
* 其他运算符

2.7.1 算术运算符 （Arithmetic operator）

下表列出了所有Go语言的算术运算符。假定 A 值为 10，B 值为 20。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| + | 相加 | A + B 输出结果 30 |
| - | 相减 | A - B 输出结果 -10 |
| \* | 相乘 | A \* B 输出结果 200 |
| / | 相除 | B / A 输出结果 2 |
| % | 求余 | B % A 输出结果 0 |
| ++ | 自增 | A++ 输出结果 11 |
| -- | 自减 | A-- 输出结果 9 |

表2.4 Go语言中算术运算符

以下实例演示了各个算术运算符的用法：

package main

import "fmt"

func main() {

var a int = 21

var b int = 10

var c int

c = a + b

fmt.Printf("第一行 - c 的值为 %d\n", c )

c = a - b

fmt.Printf("第二行 - c 的值为 %d\n", c )

c = a \* b

fmt.Printf("第三行 - c 的值为 %d\n", c )

c = a / b

fmt.Printf("第四行 - c 的值为 %d\n", c )

c = a % b

fmt.Printf("第五行 - c 的值为 %d\n", c )

a++

fmt.Printf("第六行 - a 的值为 %d\n", a )

a=21 // 为了方便测试，a 这里重新赋值为 21

a--

fmt.Printf("第七行 - a 的值为 %d\n", a )

}

以上实例运行结果：

第一行 - c 的值为 31

第二行 - c 的值为 11

第三行 - c 的值为 210

第四行 - c 的值为 2

第五行 - c 的值为 1

第六行 - a 的值为 22

第七行 - a 的值为 20

2.7.2 关系运算符（Relational operator）

下表列出了所有Go语言的关系运算符。假定 A 值为 10，B 值为 20。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| == | 检查两个值是否相等，如果相等返回 True 否则返回 False。 | (A == B) 为 False |
| != | 检查两个值是否不相等，如果不相等返回 True 否则返回 False。 | (A != B) 为 True |
| > | 检查左边值是否大于右边值，如果是返回 True 否则返回 False。 | (A > B) 为 False |
| < | 检查左边值是否小于右边值，如果是返回 True 否则返回 False。 | (A < B) 为 True |
| >= | 检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。 | (A >= B) 为 False |
| <= | 检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。 | (A <= B) 为 True |

表2.5 Go语言中的关系运算符

以下实例演示了关系运算符的用法：

package main

import "fmt"

func main() {

var a int = 21

var b int = 10

if( a == b ) {

fmt.Printf("第一行 - a 等于 b\n" )

} else {

fmt.Printf("第一行 - a 不等于 b\n" )

}

if ( a < b ) {

fmt.Printf("第二行 - a 小于 b\n" )

} else {

fmt.Printf("第二行 - a 不小于 b\n" )

}

if ( a > b ) {

fmt.Printf("第三行 - a 大于 b\n" )

} else {

fmt.Printf("第三行 - a 不大于 b\n" )

}

/\* Lets change value of a and b \*/

a = 5

b = 20

if ( a <= b ) {

fmt.Printf("第四行 - a 小于等于 b\n" )

}

if ( b >= a ) {

fmt.Printf("第五行 - b 大于等于 a\n" )

}

}

以上实例运行结果：

第一行 - a 不等于 b

第二行 - a 不小于 b

第三行 - a 大于 b

第四行 - a 小于等于 b

第五行 - b 大于等于 a

2.7.3 逻辑运算符（Logical operator）

下表列出了所有Go语言的逻辑运算符。假定 A 值为 True，B 值为 False。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| && | 逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。 | (A && B) 为 False |
| || | 逻辑 OR 运算符。 如果两边的操作数有一个 True，则条件 True，否则为 False。 | (A || B) 为 True |
| ! | 逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。 | !(A && B) 为 True |

表2.6 Go语言中的逻辑运算符

以下实例演示了逻辑运算符的用法：

package main

import "fmt"

func main() {

var a bool = true

var b bool = false

if ( a && b ) {

fmt.Printf("第一行 - 条件为 true\n" )

}

if ( a || b ) {

fmt.Printf("第二行 - 条件为 true\n" )

}

/\* 修改 a 和 b 的值 \*/

a = false

b = true

if ( a && b ) {

fmt.Printf("第三行 - 条件为 true\n" )

} else {

fmt.Printf("第三行 - 条件为 false\n" )

}

if ( !(a && b) ) {

fmt.Printf("第四行 - 条件为 true\n" )

}

}

以上实例运行结果：

第二行 - 条件为 true

第三行 - 条件为 false

第四行 - 条件为 true

2.7.4 位运算符（Bitwise operator）

位运算符对整数在内存中的二进制位进行操作。

位运算符比一般的算术运算符速度要快，而且可以实现一些算术运算符不能实现的功能。如果要开发高效率程序，位运算符是必不可少的。位运算符用来对二进制位进行操作，包括：按位与（&）、按位或（|）、按位异或（^）、按位左移（<<）、按位右移（>>）。

假定 A = 60; B = 13; 其二进制数转换为：

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

Go 语言支持的位运算符如下表所示。假定 A 为60，B 为13：

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| & | 按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。 | (A & B) 结果为 12, 二进制为 0000 1100 |
| | | 按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或 | (A | B) 结果为 61, 二进制为 0011 1101 |
| ^ | 按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1。 | (A ^ B) 结果为 49, 二进制为 0011 0001 |
| << | 左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。 | A << 2 结果为 240 ，二进制为 1111 0000 |
| >> | 右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。 | A >> 2 结果为 15 ，二进制为 0000 1111 |

表2.7 Go语言中的位运算符

按位与（&）：对两个数进行操作，然后返回一个新的数，这个数的每个位都需要两个输入数的同一位都为1 时才为1。简单说就是：同一位同时为1则为1。

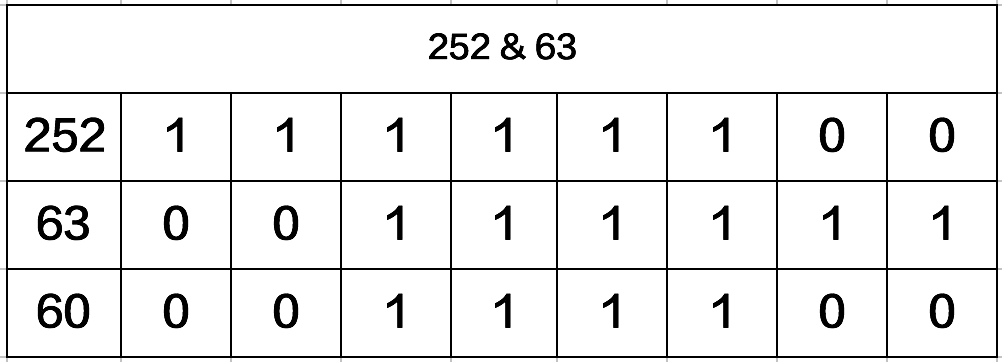


图 2.2

按位或（|）：比较两个数，然后返回一个新的数，这个数的每一位设置1的条件是任意一个数的同一位为1 则为1。简单说就是：同一位其中一个为1则为1。

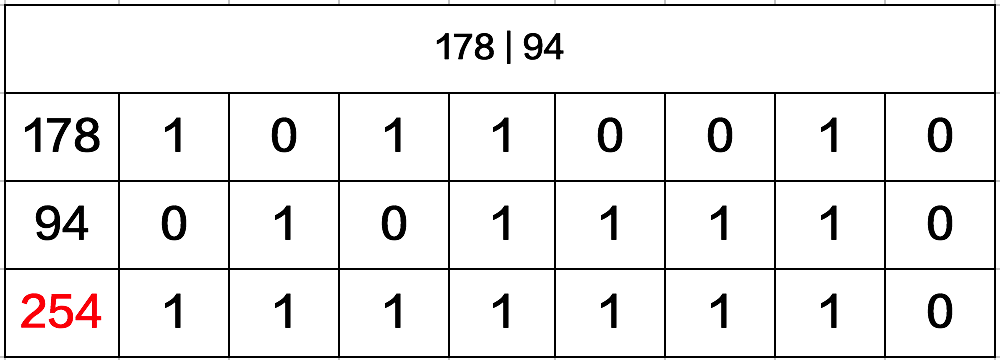


图 2.3

按位异或（^）：比较两个数，然后返回一个数，这个数的每个位设为1 的条件是两个输入数的同一位不同则为1，如果相同就设为0。简单说就是同一位不相同则为1。

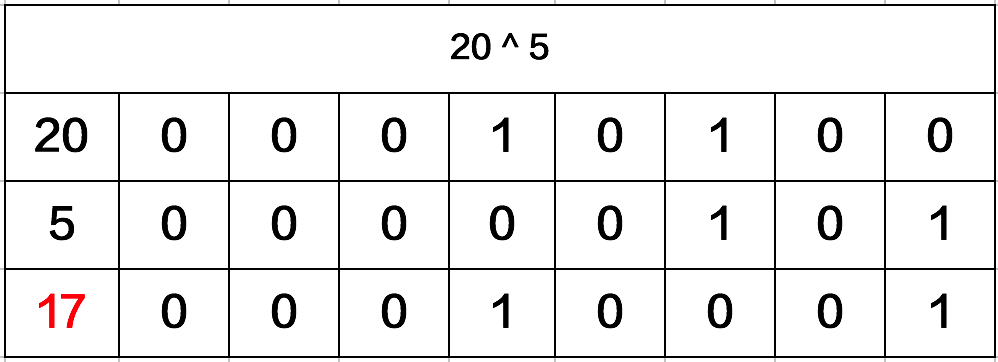


图 2.4

左移运算符（<<）是按二进制形式把所有的数字向左移动对应的位数，高位移出(舍弃)，低位的空位补零。

语法格式:

需要移位的数字 << 移位的次数

例如： 3 << 4，则是将数字3左移4位

计算过程：

3 << 4

首先把3转换为二进制数字0000 0000 0000 0000 0000 0000 0000 0011，然后把该数字高位(左侧)的两个零移出，其他的数字都朝左平移4位，最后在低位(右侧)的两个空位补零。则得到的最终结果是0000 0000 0000 0000 0000 0000 0011 0000，则转换为十进制是48。

在数字没有溢出的前提下，对于正数和负数，左移一位都相当于乘以2的1次方，左移n位就相当于乘以2的n次方。

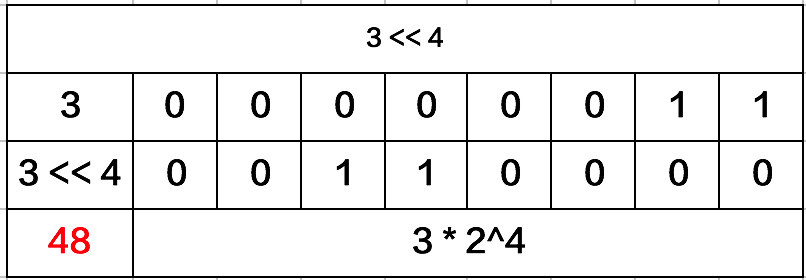


图 2.5

右移运算符（>>）是按二进制形式把所有的数字向右移动对应位移位数，低位移出(舍弃)，高位的空位补符号位，即正数补零，负数补1。

语法格式：

需要移位的数字 >> 移位的次数

例如11 >> 2，则是将数字11右移2位

计算过程：

11的二进制形式为：0000 0000 0000 0000 0000 0000 0000 1011，然后把低位的最后两个数字移出，因为该数字是正数，所以在高位补零。则得到的最终结果是0000 0000 0000 0000 0000 0000 0000 0010。转换为十进制是2。

右移一位相当于除2，右移n位相当于除以2的n次方。

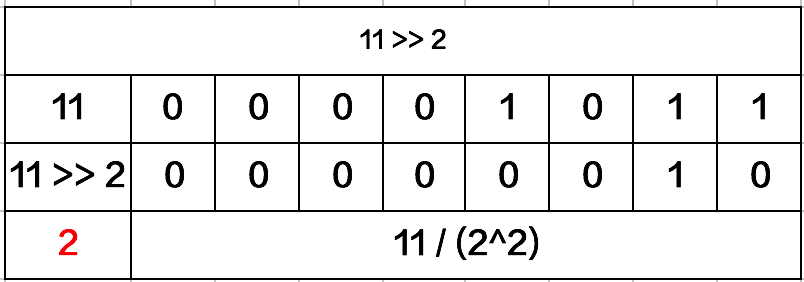


图 2.6

以下实例演示了位运算符的用法：

package main

import "fmt"

func main() {

var a uint = 60 /\* 60 = 0011 1100 \*/

var b uint = 13 /\* 13 = 0000 1101 \*/

var c uint = 0

c = a & b /\* 12 = 0000 1100 \*/

fmt.Printf("第一行 - c 的值为 %d\n", c )

c = a | b /\* 61 = 0011 1101 \*/

fmt.Printf("第二行 - c 的值为 %d\n", c )

c = a ^ b /\* 49 = 0011 0001 \*/

fmt.Printf("第三行 - c 的值为 %d\n", c )

c = a << 2 /\* 240 = 1111 0000 \*/

fmt.Printf("第四行 - c 的值为 %d\n", c )

c = a >> 2 /\* 15 = 0000 1111 \*/

fmt.Printf("第五行 - c 的值为 %d\n", c )

}

以上实例运行结果：

第一行 - c 的值为 12

第二行 - c 的值为 61

第三行 - c 的值为 49

第四行 - c 的值为 240

第五行 - c 的值为 15

2.7.5 赋值运算符（Assignment operator）

下表列出了所有Go语言的赋值运算符。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| = | 简单的赋值运算符，将一个表达式的值赋给一个左值 | C = A + B 将 A + B 表达式结果赋值给 C |
| += | 相加后再赋值 | C += A 等于 C = C + A |
| -= | 相减后再赋值 | C -= A 等于 C = C - A |
| \*= | 相乘后再赋值 | C \*= A 等于 C = C \* A |
| /= | 相除后再赋值 | C /= A 等于 C = C / A |
| %= | 求余后再赋值 | C %= A 等于 C = C % A |
| <<= | 左移后赋值 | C <<= 2 等于 C = C << 2 |
| >>= | 右移后赋值 | C >>= 2 等于 C = C >> 2 |
| &= | 按位与后赋值 | C &= 2 等于 C = C & 2 |
| ^= | 按位异或后赋值 | C ^= 2 等于 C = C ^ 2 |
| |= | 按位或后赋值 | C |= 2 等于 C = C | 2 |

表2.8 Go语言中的赋值运算符

以下实例演示了赋值运算符的用法：

package main

import "fmt"

func main() {

var a int = 21

var c int

c = a

fmt.Printf("第 1 行 - = 运算符实例，c 值为 = %d\n", c )

c += a

fmt.Printf("第 2 行 - += 运算符实例，c 值为 = %d\n", c )

c -= a

fmt.Printf("第 3 行 - -= 运算符实例，c 值为 = %d\n", c )

c \*= a

fmt.Printf("第 4 行 - \*= 运算符实例，c 值为 = %d\n", c )

c /= a

fmt.Printf("第 5 行 - /= 运算符实例，c 值为 = %d\n", c )

c = 200;

c <<= 2

fmt.Printf("第 6行 - <<= 运算符实例，c 值为 = %d\n", c )

c >>= 2

fmt.Printf("第 7 行 - >>= 运算符实例，c 值为 = %d\n", c )

c &= 2

fmt.Printf("第 8 行 - &= 运算符实例，c 值为 = %d\n", c )

c ^= 2

fmt.Printf("第 9 行 - ^= 运算符实例，c 值为 = %d\n", c )

c |= 2

fmt.Printf("第 10 行 - |= 运算符实例，c 值为 = %d\n", c )

}

以上实例运行结果：

第 1 行 - = 运算符实例，c 值为 = 21

第 2 行 - += 运算符实例，c 值为 = 42

第 3 行 - -= 运算符实例，c 值为 = 21

第 4 行 - \*= 运算符实例，c 值为 = 441

第 5 行 - /= 运算符实例，c 值为 = 21

第 6行 - <<= 运算符实例，c 值为 = 800

第 7 行 - >>= 运算符实例，c 值为 = 200

第 8 行 - &= 运算符实例，c 值为 = 0

第 9 行 - ^= 运算符实例，c 值为 = 2

第 10 行 - |= 运算符实例，c 值为 = 2

2.7.6 其他运算符

下表列出了Go语言的其他运算符。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| & | 返回变量存储地址 | &a; 将给出变量的实际地址。 |
| \* | 指针变量。 | \*a; 是一个指针变量 |

表2.9 Go语言中的其他运算符

以下实例演示了其他运算符的用法：

package main

import "fmt"

func main() {

var a int = 4

var b int32

var c float32

var ptr \*int

/\* 运算符实例 \*/

fmt.Printf("第 1 行 - a 变量类型为 = %T\n", a );

fmt.Printf("第 2 行 - b 变量类型为 = %T\n", b );

fmt.Printf("第 3 行 - c 变量类型为 = %T\n", c );

/\* & 和 \* 运算符实例 \*/

ptr = &a /\* 'ptr' 包含了 'a' 变量的地址 \*/

fmt.Printf("a 的值为 %d\n", a);

fmt.Printf("\*ptr 为 %d\n", \*ptr);

}

以上实例运行结果：

第 1 行 - a 变量类型为 = int

第 2 行 - b 变量类型为 = int32

第 3 行 - c 变量类型为 = float32

a 的值为 4

\*ptr 为 4

2.8 运算符优先级

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

|  |  |
| --- | --- |
| 优先级 | 运算符 |
| 7 | ^ ! |
| 6 | \* / % << >> & &^ |
| 5 | + - | ^ |
| 4 | == != < <= >= > |
| 3 | <- |
| 2 | && |
| 1 | || |

表2.10 Go语言中的运算符优先级

当然，你可以通过使用括号来临时提升某个表达式的整体运算优先级。

以上实例运行结果：

package main

import "fmt"

func main() {

var a int = 20

var b int = 10

var c int = 15

var d int = 5

var e int;

e = (a + b) \* c / d; // ( 30 \* 15 ) / 5

fmt.Printf("(a + b) \* c / d 的值为 : %d\n", e );

e = ((a + b) \* c) / d; // (30 \* 15 ) / 5

fmt.Printf("((a + b) \* c) / d 的值为 : %d\n" , e );

e = (a + b) \* (c / d); // (30) \* (15/5)

fmt.Printf("(a + b) \* (c / d) 的值为 : %d\n", e );

e = a + (b \* c) / d; // 20 + (150/5)

fmt.Printf("a + (b \* c) / d 的值为 : %d\n" , e );

}

以上实例运行结果：

(a + b) \* c / d 的值为 : 90

((a + b) \* c) / d 的值为 : 90

(a + b) \* (c / d) 的值为 : 90

a + (b \* c) / d 的值为 : 50

第三章 Go语言流程控制

本章学习Go语言的流程控制语句及循环语句。流程控制包括if条件判断语句、switch分支语句。循环语句包括for循环、for嵌套循环以及与循环语句相关的语句——break、continue、goto。

本章重点为大家介绍如下的内容：

* Go语言流程控制概述
* if条件判断语句
* switch分支语句
* Go语言中循环语句概述
* for循环语句及其多种语法形式
* for嵌套循环语句
* 循环控制语句（break、continue、goto）

3.1 Go 语言流程控制

流程控制是每种编程语言控制逻辑走向和执行次序的重要部分，流程控制是一门语言的经脉；流程控制语句包含条件判断语句、条件分支语句及循环语句。

Go语言的基本流程控制语句包括：

* if条件判断语句
* switch分支语句
* for循环语句
* goto跳转语句及break和continue循环控制语句

Go 语言提供了以下几种条件判断、分支语句

|  |  |
| --- | --- |
| 语句 | 描述 |
| if 语句 | if 语句由一个布尔表达式后紧跟一个或多个语句组成。 |
| if...else 语句 | if 语句后可以使用可选的else 语句, else 语句中的表达式在布尔表达式为 false 时执行。 |
| if 嵌套语句 | 你可以在if或else if语句中嵌入一个或多个if或else if语句。 |
| switch 语句 | switch语句用于基于不同条件执行不同动作。 |
| select 语句 | select语句类似于switch语句，但是select会随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。 |

表 3.1

Go 语言提供了以下几种循环语句

|  |  |
| --- | --- |
| 循环类型 | 描述 |
| for循环 | 重复执行语句块 |
| 循环嵌套 | 在 for 循环中嵌套一个或多个 for 循环 |

表 3.2

GO 语言支持以下几种循环控制语句

|  |  |
| --- | --- |
| 控制语句 | 描述 |
| break语句 | 经常用于中断当前 for 循环或跳出 switch 语句 |
| continue语句 | 跳过当前循环的剩余语句，然后继续进行下一轮循环。 |
| goto语句 | 将控制转移到被标记的语句。 |

表 3.3

3.2 if条件判断语句

Go 编程语言中 if 语句的语法如下：

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

}

if 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则不执行。

Go 编程语言中 if...else 语句的语法如下：

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

} else {

/\* 在布尔表达式为 false 时执行 \*/

}

if 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则执行 else 语句块。

Go 编程语言中 if...else if ... else 语句的语法如下：

if 布尔表达式1 {

/\* 在布尔表达式1为 true 时执行 \*/

} else if 布尔表达式2 {

/\* 在布尔表达式2为 true 时执行 \*/

} else {

/\* 在不满足以上条件时执行 \*/

}

if语句中的注意细节如下：

* 不需使用括号将条件包含起来;
* 大括号{}必须存在，即使只有一行语句;
* 左括号必须在if或else的同一行;
* 在if之后，条件语句之前，可以添加变量初始化语句，使用"；"进行分隔。

例：

1.用 if 语句判断数据奇数偶数

package main

import "fmt"

func main() {

num := 20

if num%2 == 0 {

fmt.Println(num, "偶数")

} else {

fmt.Println(num, "奇数")

}

}

2.判断学生平均成绩。有优、良、中等、及格、不及格等五档。

package main

import "fmt"

func main() {

score := 88

if score >= 90 {

fmt.Println("优秀")

} else if score >= 80 {

fmt.Println("良好")

} else if score >= 70 {

fmt.Println("中等")

} else if score >= 60 {

fmt.Println("及格")

} else if score < 60 {

fmt.Println("不及格")

}

}

if语句还有一个变体。它的语法是：

if statement; condition {

//代码块

}

例：

判断一个数是奇数还是偶数？

package main

import "fmt"

func main() {

if num := 10; num%2 == 0 {

fmt.Println(num, "偶数")

} else {

fmt.Println(num, "奇数")

}

}

需要注意的是，num的定义在if里，那么只能够在该if..else语句块中使用，否则编译器会报错的。

3.3 if嵌套语句

可以在 if 或 else if 语句中嵌入一个或多个 if 或 else if 语句。

Go 编程语言中 if...else 语句的语法如下：

if 布尔表达式 1 {

/\* 在布尔表达式 1 为 true 时执行 \*/

if 布尔表达式 2 {

/\* 在布尔表达式 2 为 true 时执行 \*/

}

}

可以以同样的方式在 if 语句中嵌套 else if...else 语句

例：

判断学生平均成绩。有优、良、中等、及格、不及格等五档。

package main

import "fmt"

func main() {

if score := 98; score >= 60 {

if score >= 70 {

if score >= 80 {

if score >= 90 {

fmt.Println("优")

} else {

fmt.Println("良")

}

} else {

fmt.Println("中等")

}

} else {

fmt.Println("及格")

}

} else {

fmt.Println("不及格")

}

}

3.4 switch分支语句

Go 编程语言中 switch 语句的语法如下：

switch var1 {

case val1:

...

case val2:

...

default:

...

}

switch 语句执行的过程自上而下，直到找到case匹配项，匹配项中无需使用break，因为Go语言中的switch默认给每个case自带break，因此匹配成功后不会向下执行其他的case分支，而是跳出整个switch。

变量 var1 可以是任何类型，而 val1 和 val2 则可以是同类型的任意值。类型不被局限于常量或整数，但必须是相同类型或最终结果为相同类型的表达式。

case后的值不能重复。

可以同时测试多个符合条件的值，也就是说case后可以有多个值，这些值之间使用逗号分割，例如：case val1, val2, val3。

Go语言中switch后的表达式可以省略，那么默认是switch true。

Go语言中的switch case因为自带break，所以匹配某个case后不会自动向下执行其他case，如需贯通后续的case，可以添加fallthrough（中文含义是：贯穿），强制执行后面的case分支。

fallthrough必须放在case分支的最后一行。如果它出现在中间的某个地方，编译器就会抛出错误（fallthrough statement out of place，含义是fallthrough不在合适的位置）。



图 3.1

例：

1.判断学生平均成绩。有优、良、中等、及格、不及格等五档。

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

grade := ""

score := 78.5

//思考：以下代码逻辑错误在哪里？

//switch {

//case score < 60:

// grade = "E"

//case score >= 60:

// grade = "D"

//case score >= 70:

// grade = "C"

//case score >= 80:

// grade = "B"

//case score >= 90:

// grade = "A"

//}

switch { //switch后面省略不写，默认相当于：switch true

case score >= 90:

grade = "A"

case score >= 80:

grade = "B"

case score >= 70:

grade = "C"

case score >= 60:

grade = "D"

default:

grade = "E"

}

fmt.Printf("你的等级是: %s\n", grade)

fmt.Print("最终评价是: ")

switch grade {

case "A":

fmt.Printf("优秀!\n")

case "B":

fmt.Printf("良好\n")

case "C":

fmt.Printf("中等\n")

case "D":

fmt.Printf("及格\n")

default:

fmt.Printf("差\n")

}

}

2.判断某年某月的天数

package main

import "fmt"

func main() {

/\* 定义局部变量:年、月、日 \*/

year := 2008

month := 2

days := 0

switch month {

case 1, 3, 5, 7, 8, 10, 12:

days = 31

case 4, 6, 9, 11:

days = 30

case 2:

if (year%4 == 0 && year%100 != 0) || year%400 == 0 {

days = 29

} else {

days = 28

}

default:

days = -1

}

fmt.Printf("%d 年 %d 月的天数为：%d\n", year, month, days)

}

switch 语句还可以被用于 type-switch 来判断某个 interface 变量中实际存储的变量类型。

语法结构如下：

switch x.(type){

case type:

statement(s);

case type:

statement(s);

/\* 你可以定义任意个数的case \*/

default: /\* 可选 \*/

statement(s);

}

例：

package main

import "fmt"

func main() {

var x interface{}

switch i := x.(type) {

case nil:

fmt.Printf(" x 的类型 :%T",i)

case int:

fmt.Printf("x 是 int 型")

case float64:

fmt.Printf("x 是 float64 型")

case func(int) float64:

fmt.Printf("x 是 func(int) 型")

case bool, string:

fmt.Printf("x 是 bool 或 string 型" )

default:

fmt.Printf("未知型")

}

}

结果

x 的类型 :<nil>

3.5 Go循环语句

在不少实际问题中有许多具有规律性的重复操作，因此在程序中就需要重复执行某些语句。循环语句包括循环处理语句及循环控制语句。

循环处理语句有：

|  |  |
| --- | --- |
| 循环类型 | 描述 |
| for 循环 | 重复执行语句块 |
| 循环嵌套 | 在 for 循环中嵌套一个或多个 for 循环 |

表 3.4

循环控制语句可以控制循环体内语句的执行过程。循环控制语句有：

|  |  |
| --- | --- |
| 控制语句 | 描述 |
| break语句 | 经常用于中断当前 for 循环或跳出 switch 语句 |
| continue语句 | 跳过当前循环的剩余语句，然后继续进行下一轮循环。 |
| goto语句 | 将控制转移到被标记的语句。 |

表 3.5

循环语句表示条件满足，可以反复的执行某段代码。for是Go语言中唯一的循环语句，Go没有while、do...while循环。按语法结构来分，Go语言的for循环有4种形式，只有其中第一种使用分号。for循环中for关键字后不能加小括号。

3.5.1 for循环语法结构

1.基本for循环语法结构：

for 初始语句init; 条件表达式condition; 结束语句post {

//循环体代码

}

三个组成部分，即初始化、条件表达式和post都是可选的。

因此这种基本的for循环语法结构又能演化出四种略有不同的写法。

例：

for i := 0; i <= 10; i++ {

fmt.Printf("%d ", i)

}

语法解释

（1）初始语句init：

初始语句是在第一次循环前执行的语句，一般为赋值表达式，给控制变量赋初始值。如果控制变量在此处被声明，其作用域将被局限在这个for的范围内；在for循环中声明的变量仅在循环范围内可用。初始语句可以省略不写，但是初始语句之后的分号必须要写。

i := 0

for ; i <= 10; i++ {

fmt.Printf("%d ", i)

}

（2）条件表达式condition：

条件表达式是控制循环与否的开关；如果表达式为true，则循环继续，否则结束循环；条件表达式可以省略不写，之后的分号必须要写；省略条件表达式默认形成无限循环。

i := 0

for ; ; i++ {

if (i > 20) {

break

}

fmt.Printf("%d ", i)

}

（3）结束语句post：

一般为赋值表达式，给控制变量递增或者递减；post语句将在循环的每次成功迭代之后执行。

for语句执行过程如下：

先执行初始化语句，对控制变量赋初始值。初始化语句只执行一次。

其次根据控制变量判断条件表达式的返回值，若其值为true，满足循环条件，则执行循环体内语句，之后执行 post语句，开始下一次循环。

执行post语句之后，将重新计算条件表达式的返回值，如果是true，循环将继续执行，否则循环终止。然后执行循环体外语句。

2.for关键字后只有一个条件表达式：

for 循环条件condition { }

效果类似其它编程语言中的while循环

例：

var i int

for i <= 10 {

fmt.Println(i)

i++

}

3.for关键字后无表达式：

for { }

效果与其它编程语言的for(;;) {}一致，此时for执行无限循环

例：

var i int

for {

if (i > 10) {

break

}

fmt.Println(i)

i++

}

4.for ... range：

for 循环的 range 格式

主要用于对字符串、slice、数组、map等进行迭代循环

for key, value := range oldMap {

newMap[key] = value

}

例：

遍历字符串，获得字符

func traverseString() {

str := "123ABCabc一丁丂"

for i, value := range str {

fmt.Printf("第 %d 位的ASCII值=%d ，字符是%c \n", i, value ,value)

}

}

打印结果：

第 0 位的ASCII值=49 ，字符是1

第 1 位的ASCII值=50 ，字符是2

第 2 位的ASCII值=51 ，字符是3

第 3 位的ASCII值=65 ，字符是A

第 4 位的ASCII值=66 ，字符是B

第 5 位的ASCII值=67 ，字符是C

第 6 位的ASCII值=97 ，字符是a

第 7 位的ASCII值=98 ，字符是b

第 8 位的ASCII值=99 ，字符是c

第 9 位的ASCII值=19968 ，字符是一

第 12 位的ASCII值=19969 ，字符是丁

第 15 位的ASCII值=19970 ，字符是丂

例：

遍历切片中元素

func traverseSlice() {

arr := []int{100, 200, 300}

for i, value := range arr {

fmt.Println(i, ":", value)

}

}

3.5.2 for循环使用案例

1.求1-100 的和

func summation() {

sum := 0

for i := 1; i <= 100; i++ {

sum += i;

}

fmt.Println(sum)

}

2.求1-100之间3的倍数的和

func summation2() {

i := 1

sum := 0

for i <= 100 {

if i%3 == 0 {

sum += i

fmt.Print(i)

if i < 99 {

fmt.Print("+")

} else {

fmt.Printf(" = %d \n", sum)

}

}

i++

}

}

3.截竹竿。32米竹竿，每次截1.5米，最快截几次之后能小于4米？

func cutBamboo() {

count := 0;

for i := 32.0; i >= 4; i -= 1.5 {

count++;

}

fmt.Println(count)

}

3.6 for嵌套循环语句

for [condition | ( init; condition; increment ) | Range] {

for [condition | ( init; condition; increment ) | Range] {

statement(s);

}

statement(s);

}

例：

1.打印直角三角形

func printRightTriangle() {

// 定义行数

lines := 8

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

for n := 0; n < 2\*i+1; n++ {

fmt.Print("\* ");

}

fmt.Println()

}

}

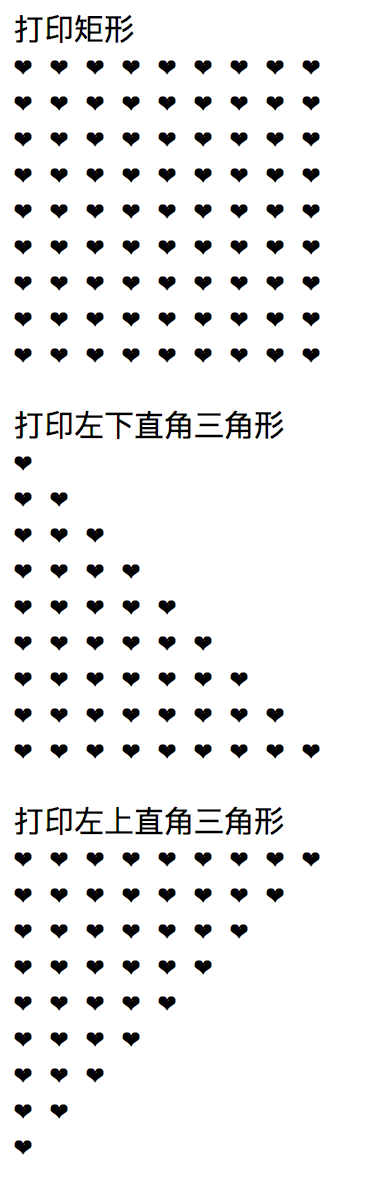


图 3.2

2.打印等腰三角形

func printTriangle() {

for i := 1; i <= 10; i++ {

//控制每一行符号前的空格的数量

for m := 10; m > i; m-- {

fmt.Print(" ")

}

//控制每一行符号的数量

for j := 1; j <= 2\*i-1; j++ {

fmt.Print("❤ ")

}

fmt.Println()

}

}

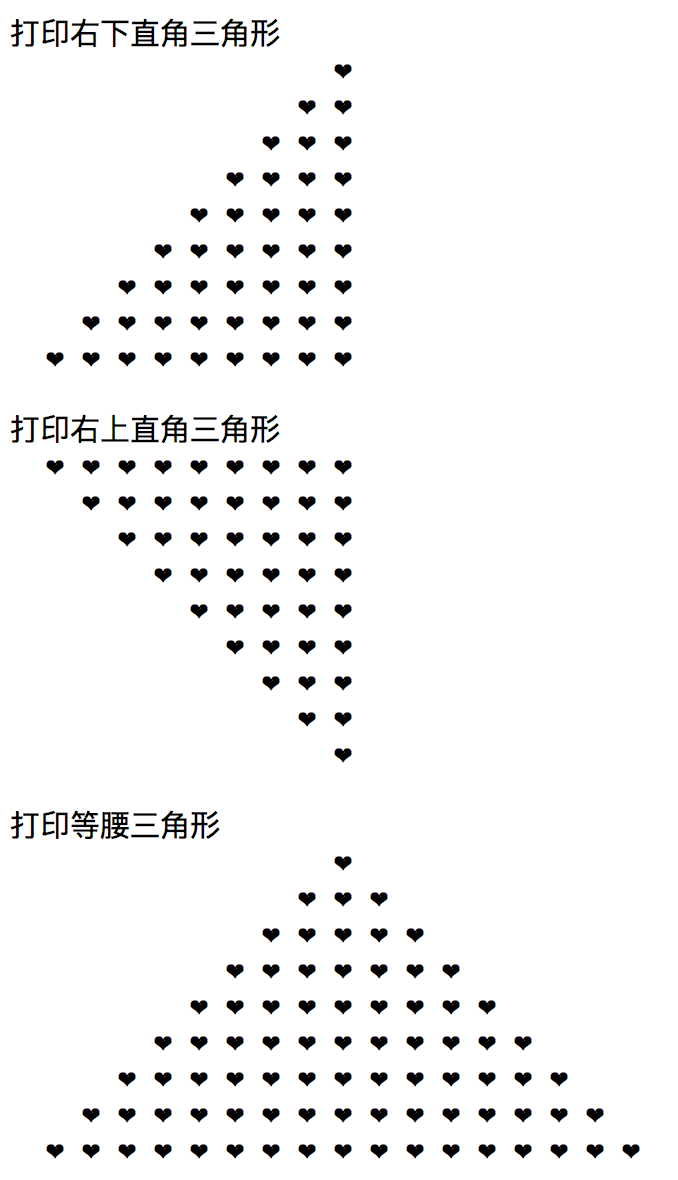


图 3.3

3.打印九九乘法表

func multiply99() {

for i := 1; i <= 9; i++ { // i 控制行数，是乘法中的第二个数。

for j := 1; j <= i; j++ { // j 控制每行的列数，是乘法中的第一个数。

fmt.Printf("%d\*%d=%d ", j, i, i\*j);

}

fmt.Println()

}

}

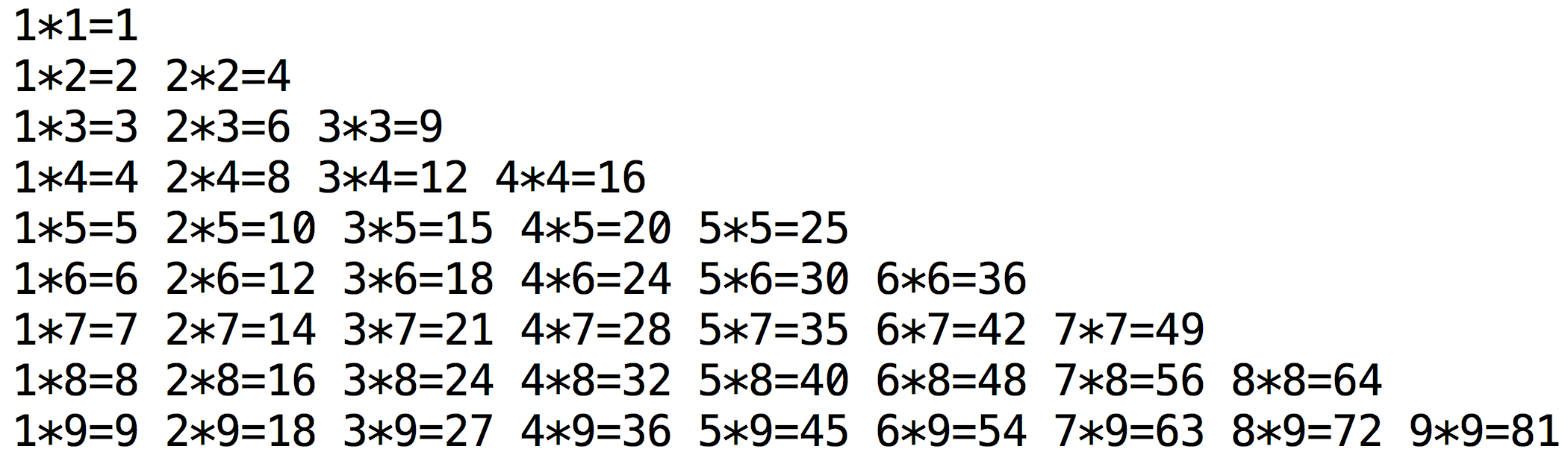


图 3.4

4.使用循环嵌套来输出 2 到 100 间的素数：

func printPrimeNumber() {

/\* 定义局部变量 \*/

fmt.Print("1-100的素数：")

var a, b int

for a = 2; a <= 100; a++ {

for b = 2; b <= (a / b); b++ {

if a%b == 0 {

break // 如果发现因子，则不是素数

}

}

if b > (a / b) {

fmt.Printf("%d\t", a)

}

}

}

3.7 循环控制语句

3.7.1 break语句

break：跳出循环体。break语句用于在结束其正常执行之前突然终止for循环，并开始执行循环之后的语句。

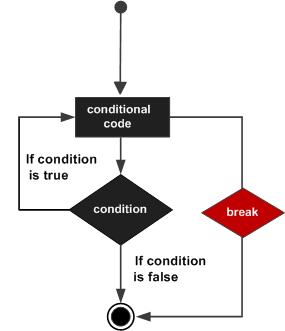


图 3.5

例：

func main() {

for i := 1; i <= 10; i++ {

if i > 5 {

break // 如果i > 5，则循环终止（loop is terminated ）

}

fmt.Printf("%d ", i)

}

fmt.Printf("\nline after for loop")

}

3.7.2 continue语句

Go 语言的 continue 语句 有点像 break 语句。但是 continue 不是跳出循环，而是跳过当前循环执行下一次循环语句。for 循环中，执行 continue 语句会触发for增量语句的执行。换句话说，continue语句用于跳过for循环的当前迭代，循环将继续到下一个迭代。

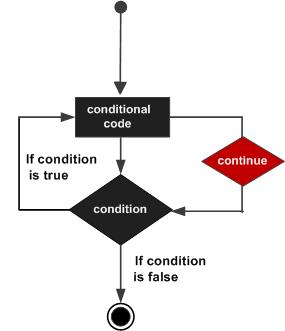


图 3.6

例：

func main() {

for i := 1; i <= 10; i++ {

if i%2 == 0 {

continue

}

fmt.Printf("%d ", i)

}

}

备注：

break，continue的区别：

● break语句将无条件跳出并结束当前的循环， 然后执行循环体后的语句;

● continue语句是跳过当前的循环， 而开始执行下一次循环。

// break终止循环

for i := 0; i < 10; i++ {

if i == 5 {

break

}

fmt.Print(i)

}

//结果是：01234

//continue跳过某次循环

for i := 0; i < 10; i++ {

if i == 5 {

continue

}

fmt.Print(i);

}

//结果是：012346789

例：输出1-50之间所有不包含数字4的数（continue实现）

func eludeFour() {

fmt.Println("\n输出1-50之间所有不包含数字4的数")

//定义局部变量

num := 0

//循环开始

for num < 50 {

num++

/\* 跳过迭代 \*/

if num%10 == 4 || num/10%10 == 4 {

continue

}

fmt.Printf("%d\t", num)

}

}

3.7.3 goto语句

Go 语言的 goto 语句允许无条件地转移到程序指定的行。goto语句通常与条件语句配合使用。可用来实现条件转移，构成循环，跳出循环体等功能。但是，在结构化程序设计中一般不主张使用goto语句， 以免造成程序流程的混乱，使理解和调试程序都产生困难。

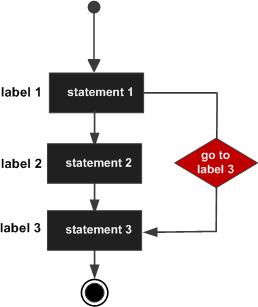


图 3.7

goto 语法格式如下：

LABEL: statement

goto LABEL

例：

输出1-50之间不包含4的数（goto实现）

func gotoTest() {

//定义局部变量

num := 0

/\* 跳过迭代 \*/

LOOP:

for num < 50 {

num++

if num%10 == 4 || num/10%10 == 4 {

goto LOOP

}

fmt.Printf("%d\t", num)

}

}

例：

求1-100的素数（借助goto跳转）

func printPrimeNumberGoto() {

var C, c int //声明变量

C = 1 /\*这里不写入FOR循环是因为For语句执行之初会将C的值变为1，当我们goto A时for语句会重新执行（不是重新一轮循环）\*/

LOOP:

for C < 100 {

C++ //C=1不能写入for这里就不能写入

for c = 2; c < C; c++ {

if C%c == 0 {

goto LOOP //若发现因子则不是素数

}

}

fmt.Printf("%d \t" , C)

}

}

第四章 函数与指针

本章。

本章重点为大家介绍如下的内容：

* 什么是函数
* 声明函数
* 变量作用域
* 函数变量（函数作为值）
* 匿名函数
* 闭包
* 可变参数
* 递归函数
* 指针
* 函数参数传递（值传递与引用传递）

4.1 函数

函数是组织好的、可重复使用的执行特定任务的代码块。它可以提高应用程序的模块性和代码的重复利用率。

Go语言支持普通函数、匿名函数和闭包，从设计上对函数进行了优化和改进，让函数使用起来更加方便。Go语言的函数本身可以作为值进行传递、支持匿名函数和闭包（closure）、函数可以满足接口。

4.1.1 声明函数

普通函数需要先声明才能调用，一个函数的声明包括参数和函数名等。编译器通过声明才能了解函数应该怎样在调用代码和函数体之间传递参数和返回参数。

语法格式：

func 函数名（参数列表）（返回参数列表）{

//函数体

}

例：

func funcName (parametername type1, parametername type2...) (output1 type1, output2 type2...){

//逻辑代码

//返回多个值

return value1, value2...

}

使用func关键字定义一个函数。每一个函数由 func 开始声明。

funcName：函数名。函数名和参数列表一起构成了函数签名。函数名由字母、数字和下划线组成。函数名的第一个字母不能为数字。在同一个包内，函数名称不能重名。

parametername type：参数列表。参数就像一个占位符，定义函数时的参数叫做形式参数，形参变量是函数的局部变量；当函数被调用时，你可以将值传递给参数，这个值被称为实际参数。

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，也就是说函数也可以不包含参数。在参数列表中，如果有多个参数变量，则以逗号分隔；如果相邻变量是同类型，则可以将类型省略。

例：

func add （a , b int） {}

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。

例：

func myfunc(arg ...int) {}

arg ...int告诉Go这个函数接受不定数量的参数。注意，这些参数的类型全部是int。在函数体中，变量arg是一个int的slice（切片）。

output1 type1, output2 type2：返回值列表。

返回值返回函数的执行结果，结束函数的执行。Go语言的函数可以返回多个值；返回值的类型可以是返回数据的数据类型，或者是变量名+变量类型的组合；函数声明时有返回值的，必须在函数体中使用return语句提供返回值列表；如果只有一个返回值且不声明返回值变量，那么可以省略包括返回值的括号；return后的数据，要保持和声明的返回值类型、数量、顺序一致。如果函数没有声明返回值，函数中也可以使用return关键字，用于强制结束函数。

4.1.2 变量作用域

作用域是指变量、常量、类型、函数的作用范围。

语言中变量可以在三个地方声明：

* 函数内定义的变量称为局部变量

在函数体内声明的变量称为局部变量，它们的作用域只在函数体内，参数和返回值变量也是局部变量。

* 函数外定义的变量称为全局变量

在函数体外声明的变量称之为全局变量，全局变量可以在整个包甚至外部包（被导出后）使用。全局变量可以在任何函数中使用。Go 语言程序中全局变量与局部变量名称可以相同，但是函数内的局部变量会被优先考虑。

* 函数中定义的参数称为形式参数

形式参数会作为函数的局部变量来使用。

例：

package main

import "fmt"

/\* 声明全局变量 \*/

var a1 int = 7

var b1 int = 9

func main() {

/\* main 函数中声明局部变量 \*/

a1, b1, c1 := 10, 20, 0

fmt.Printf("main()函数中 a1 = %d\n", a1) //10

fmt.Printf("main()函数中 b1 = %d\n", b1) //20

fmt.Printf("main()函数中 c1 = %d\n", c1) //0

c1 = sum(a1, b1)

fmt.Printf("main()函数中 c1 = %d\n", c1) //33

}

/\* 函数定义-两数相加 \*/

func sum(a1, b1 int) (c1 int) {

a1++

b1 += 2

c1 = a1 + b1

fmt.Printf("sum() 函数中 a1 = %d\n", a1) //11

fmt.Printf("sum() 函数中 b1 = %d\n", b1) //22

fmt.Printf("sum() 函数中 c1 = %d\n", c1) //33

return c1

}

输出结果：

main()函数中 a1 = 10

main()函数中 b1 = 20

main()函数中 c1 = 0

sum() 函数中 a1 = 11

sum() 函数中 b1 = 22

sum() 函数中 c1 = 33

main()函数中 c1 = 33

4.1.3 函数变量

在Go语言中，函数也是一种类型，可以和其它类型一样被保存在变量中。

可以通过type来定义一个自定义类型。函数的参数完全相同（包括：参数类型、个数、顺序），函数返回值相同。

例：

package main

import (

"fmt"

"strings"

)

func main() {

result := StringToLower("AbcdefGHijklMNOPqrstUVWxyz", processCase)

fmt.Println(result)

result = StringToLower2("AbcdefGHijklMNOPqrstUVWxyz", processCase)

fmt.Println(result)

}

//处理字符串，奇数偶数依次显示为大小写

func processCase(str string) string {

result := ""

for i, value := range str {

if i%2 == 0 {

result += strings.ToUpper(string(value))

} else {

result += strings.ToLower(string(value))

}

}

return result

}

func StringToLower(str string, f func(string) string) string {

fmt.Printf("%T \n", f)

return f(str)

}

type caseFunc func(string) string // 声明了一个函数类型。通过type关键字，caseFunc会形成一种新的类型。

func StringToLower2(str string, f caseFunc) string {

fmt.Printf("%T \n", f)

return f(str)

}

例：

package main

import "fmt"

type processFunc func(int) bool // 声明了一个函数类型

func main() {

slice := []int{1, 2, 3, 4, 5, 7}

fmt.Println("slice = ", slice)

odd := filter(slice, isOdd) // 函数当做值来传递

fmt.Println("奇数元素: ", odd)

even := filter(slice, isEven) // 函数当做值来传递

fmt.Println("偶数元素: ", even)

}

//判断元素是否是偶数

func isEven(integer int) bool {

if integer%2 == 0 {

return true

}

return false

}

//判断元素是否是奇数

func isOdd(integer int) bool {

if integer%2 == 0 {

return false

}

return true

}

//根据函数来处理切片，根据元素奇数偶数分组，返回新的切片

func filter(slice []int, f processFunc) []int {

var result []int

for \_, value := range slice {

if f(value) {

result = append(result, value)

}

}

return result

}

通过上面例子可以看到processFunc这个类型是一个函数类型，然后两个filter函数的参数和返回值与processFunc类型是一样的。用户可以实现很多种的逻辑，这样使得程序变得非常的灵活。

4.1.4 匿名函数

Go语言支持匿名函数，即在需要使用函数时，再定义函数，匿名函数没有函数名，只有函数体，函数可以被作为一种类型被赋值给变量，匿名函数也往往以变量方式被传递。匿名函数经常被用于实现回调函数、闭包等。

定义格式如下：

func(参数列表) （返回参数列表） {

//函数体

}

例：

1.在定义时调用匿名函数

package main

import "fmt"

func main() {

func(data int) {

fmt.Println("hello" , data)

}(100)

}

2.将匿名函数赋值给变量

package main

import "fmt"

func main() {

f:= func(data string) {

fmt.Println(data)

}

f("欢迎学习Go语言！")

}

3.匿名函数的用法——作回调函数

package main

import (

"fmt"

"math"

)

func main() {

//调用函数，对每个元素进行求平方根操作

arr := []float64{1, 9, 16, 25, 30}

visit(arr, func(v float64) {

v = math.Sqrt(v)

fmt.Printf("%.2f \n", v)

})

//调用函数，对每个元素进行求平方操作

visit(arr, func(v float64) {

v = math.Pow(v , 2)

fmt.Printf("%.0f \n", v)

})

}

//定义一个函数，遍历切片元素，对每个元素进行处理

func visit(list []float64, f func(float64)) {

for \_, value := range list {

f(value)

}

}

4.1.5 闭包

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。对闭包的具体定义有很多种说法，大体可以分为两类：

* 一种说法认为闭包是符合一定条件的函数，比如这样定义闭包：闭包是在其词法上下文中引用了自由变量的函数。
* 另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。比如这样的定义：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

上面的定义，一个认为闭包是函数，另一个认为闭包是函数和引用环境组成的整体。显然第二种说法更确切。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也被称为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。

闭包的价值在于：

* 加强模块化
* 抽象
* 简化代码

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。比如我们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呢？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，我们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。闭包是数据和行为的组合，这使得闭包具有较好抽象能力。

一个编程语言需要哪些特性来支持闭包呢？

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；

例：

1.没有使用闭包进行计数的代码

package main

import "fmt"

func main() {

for i := 0; i < 5; i++ {

fmt.Printf("i=%d \t", i)

fmt.Println(add2(i))

}

}

func add2(x int) int {

sum := 0

sum += x

return sum

}

运行结果：

i=0 0

i=1 1

i=2 2

i=3 3

i=4 4

for循环每执行一次，sum都会清零，没有实现sum累加计数。

2.使用闭包函数实现计数器：

package main

import "fmt"

func main() {

pos := adder()

for i := 0; i < 10; i++ {

fmt.Printf("i=%d \t", i)

fmt.Println(pos(i))

}

fmt.Println("---------------------")

for i := 0; i < 10; i++ {

fmt.Printf("i=%d \t", i)

fmt.Println(pos(i))

}

}

func adder() func(int) int {

sum := 0

return func(x int) int {

fmt.Printf("sum1=%d \t", sum)

sum += x

fmt.Printf("sum2=%d \t", sum)

return sum

}

}

运行结果为：

i=0 sum1=0 sum2=0 0

i=1 sum1=0 sum2=1 1

i=2 sum1=1 sum2=3 3

i=3 sum1=3 sum2=6 6

i=4 sum1=6 sum2=10 10

---------------------

i=0 sum1=10 sum2=10 10

i=1 sum1=10 sum2=11 11

i=2 sum1=11 sum2=13 13

i=3 sum1=13 sum2=16 16

i=4 sum1=16 sum2=20 20

例：

package main

import "fmt"

func main() {

myfunc := Counter()

//fmt.Printf("%T\n", myfunc)

fmt.Println("myfunc", myfunc)

/\* 调用 myfunc 函数，i 变量自增 1 并返回 \*/

fmt.Println(myfunc())

fmt.Println(myfunc())

fmt.Println(myfunc())

/\* 创建新的函数 nextNumber1，并查看结果 \*/

myfunc1 := Counter()

fmt.Println("myfunc1", myfunc1)

fmt.Println(myfunc1())

fmt.Println(myfunc1())

}

//计数器.闭包函数

func Counter() func() int {

i := 0

res := func() int {

i += 1

return i

}

//fmt.Printf("%T , %v \n" , res , res) //func() int , 0x1095af0

fmt.Println("Counter中的内部函数:", res) //0x1095af0

return res

}

4.1.6 可变参数

如果一个函数的参数类型一致，但个数不定，可以使用函数的可变参数。

语法格式:

func 函数名(参数名 ...类型) [(返回值列表)] {

//函数体

}

该语法格式定义了一个接受任何数目、任何类型参数的函数。这里特殊的语法是三个点“...”，在一个变量后面加上三个点后，表示从该处开始接受不定参数。

当要传递若干个值到不定参数函数中得时候，可以手动书写每个参数，也可以将一个slice传递给该函数，通过"..."可以将slice中的参数对应的传递给函数。

例：

计算学员考试总成绩及平均成绩

package main

import (

"fmt"

)

func main() {

//1传进n个参数

sum, avg, count := GetScore(90, 82.5, 73, 64.8)

fmt.Printf("学员共有%d门成绩，总成绩为：%.2f，平均成绩为：%.2f", count, sum, avg)

fmt.Println()

// 2传切片作为参数

scores := []float64{92, 72.5, 93, 74.5, 89, 87, 74}

sum, avg, count = GetScore(scores...)

fmt.Printf("学员共有%d门成绩，总成绩为：%.2f，平均成绩为：%.2f", count, sum, avg)

}

//累加求和，参数个数不定，参数个数从0-n

func GetScore(scores ...float64) (sum, avg float64, count int) {

for \_, value := range scores {

sum += value

count++

}

avg = sum / float64(count)

return

}

可变参数注意细节：

一个函数最多只能有一个可变参数

参数列表中还有其它类型参数，则可变参数写在所有参数的最后。

4.1.7 递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

递归函数必须满足以下两个条件：

1.在每一次调用自己时，必须是（在某种意义上）更接近于解；

2.必须有一个终止处理或计算的准则。

例：

求阶乘，计算阶乘n! = 1 x 2 x 3 x ... x n，用函数fact(n)表示，可以看出：fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n。所以，fact(n)可以表示为n x fact(n-1)，只有n=1时需要特殊处理。

package main

import "fmt"

func main() {

fmt.Println(factorial(5))

}

//通过递归实现阶乘

func factorial(n int) int {

if n == 0 {

return 1

}

return n \* factorial(n-1)

}

//通过循环实现阶乘

func getMultiple(num int) (result int) {

result = 1

for i:=1; i<= num; i++ {

result \*= i

}

return

}

递归的计算过程

===> factorial(5)

===> 5 \* factorial(4)

===> 5 \* (4 \* factorial(3))

===> 5 \* (4 \* (3 \* factorial(2)))

===> 5 \* (4 \* (3 \* (2 \* factorial(1))))

===> 5 \* (4 \* (3 \* (2 \* 1)))

===> 5 \* (4 \* (3 \* 2))

===> 5 \* (4 \* 6)

===> 5 \* 24

===> 120

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以用循环的方式实现，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈，每当函数返回，栈就会减一层。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

4.2 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。

如下图：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。



图 4.1

Go 语言的取地址符&，在一个变量前使用&，会返回该变量的内存地址。

例：

func main() {

a := 10

fmt.Printf("变量的地址: %x \n", &a)

}

运行结果：

变量的地址: c420014050

Go语言指针的最大特点是：指针不能运算（不同于C语言）。

在Go语言中如果对指针进行运算会报错：nvalid operation: p++ (non-numeric type \*int)

4.2.1 声明指针

声明指针，\*T是指针变量的类型，它指向T类型的值。

var 指针变量名 \*指针类型

\* 号用于指定变量是一个指针。

var ip \*int //指向整型的指针

var fp \*float32 //指向浮点型的指针

指针使用流程如下：

* 定义指针变量。
* 为指针变量赋值。
* 访问指针变量中指向地址的值。

获取指针的值时，在指针类型的变量前加上 \* 号（前缀）来获取指针所指向的内容。

获取一个指针意味着访问指针指向的变量的值。语法是：\*a

例：

func main() {

//声明实际变量

var a int = 120

//声明指针变量

var ip \*int

//给指针变量赋值，将变量a的地址赋值给ip

ip = &a

//打印a的类型和值

fmt.Printf("a 的类型是%T，值是%v \n", a, a)

//打印&a的类型和值

fmt.Printf("&a 的类型是%T，值是%v \n", &a, &a)

//打印ip的类型和值

fmt.Printf("ip 的类型是%T，值是%v \n", ip, ip)

//打印变量\*ip的类型和值

fmt.Printf("\*ip 变量的类型是%T，值是%v \n", \*ip, \*ip)

//打印变量\*&a的类型和值

fmt.Printf("\*&a 变量的类型是%T，值是%v \n", \*&a, \*&a)

fmt.Println(a, &a, \*&a)

fmt.Println(ip, &ip, \*ip, \*(&ip), &(\*ip))

}

运行结果：

a 的类型是int，值是120

&a 的类型是\*int，值是0xc420014050

ip 的类型是\*int，值是0xc420014050

\*ip 变量的类型是int，值是120

\*&a 变量的类型是int，值是120

120 0xc420014050 120

0xc420014050 0xc42000c028 120 0xc420014050 0xc420014050

例：

package main

import "fmt"

type Student struct {

name string

age int

married bool

sex int8

}

func main() {

var s1 = Student{"Steven", 35, true, 1}

var s2 = Student{"Sunny", 20, false, 0}

var a \*Student = &s1 //将s1的内存地址赋值给Student指针变量a

var b \*Student = &s2 //将s2的内存地址赋值给Student指针变量b

fmt.Println("\n---------------------")

fmt.Printf("s1类型为%T，值为%v \n", s1, s1)

fmt.Printf("s2类型为%T，值为%v \n", s2, s2)

fmt.Println("\n---------------------")

fmt.Printf("a类型为%T，值为%v \n", a, a)

fmt.Printf("b类型为%T，值为%v \n", b, b)

fmt.Println("\n---------------------")

fmt.Printf("\*a类型为%T，值为%v \n", \*a, \*a)

fmt.Printf("\*b类型为%T，值为%v \n", \*b, \*b)

fmt.Println("\n---------------------")

fmt.Println(s1.name, s1.age, s1.married, s1.sex)

fmt.Println(a.name, a.age, a.married, a.sex)

fmt.Println("\n---------------------")

fmt.Println(s2.name, s2.age, s2.married, s2.sex)

fmt.Println(b.name, b.age, b.married, b.sex)

fmt.Println("\n---------------------")

fmt.Println((\*a).name, (\*a).age, (\*a).married, (\*a).sex)

fmt.Println((\*b).name, (\*b).age, (\*b).married, (\*b).sex)

fmt.Println("\n---------------------")

fmt.Printf("&a类型为%T，值为%v\n", &a, &a)

fmt.Printf("&b类型为%T，值为%v\n", &b, &b)

fmt.Println("\n---------------------")

fmt.Println(&a.name, &a.age, &a.married, &a.sex)

fmt.Println(&b.name, &b.age, &b.married, &b.sex)

}

运行结果：

---------------------

s1类型为main.Student，值为{Steven 35 true 1}

s2类型为main.Student，值为{Sunny 20 false 0}

---------------------

a类型为\*main.Student，值为&{Steven 35 true 1}

b类型为\*main.Student，值为&{Sunny 20 false 0}

---------------------

\*a类型为main.Student，值为{Steven 35 true 1}

\*b类型为main.Student，值为{Sunny 20 false 0}

---------------------

Steven 35 true 1

Steven 35 true 1

---------------------

Sunny 20 false 0

Sunny 20 false 0

---------------------

Steven 35 true 1

Sunny 20 false 0

---------------------

&a类型为\*\*main.Student，值为0xc42000c028

&b类型为\*\*main.Student，值为0xc42000c030

---------------------

0xc42000a060 0xc42000a070 0xc42000a078 0xc42000a079

0xc42000a080 0xc42000a090 0xc42000a098 0xc42000a099

4.2.2 空指针

Go 空指针是指一个指针被定义后没有分配到任何变量的指针，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。一个指针变量通常缩写为 ptr。

空指针判断：

if(ptr != nil) // ptr 不是空指针

if(ptr == nil) // ptr 是空指针

4.2.3 操作指针改变变量的数值

package main

import (

"fmt"

)

func main() {

b := 3158

a := &b

fmt.Println("b 的地址：", a) //0xc420014050

fmt.Println("\*a 的值：", \*a) //3158

\*a++

fmt.Println("b 的新值：", b)//3159

}

运行结果

b 的地址： 0xc420014050

\*a 的值：3158

b 的新值： 3159

4.2.4 使用指针作为函数的参数

基本数据类型指针作为函数参数

例：

package main

import (

"fmt"

)

func main() {

a := 58

fmt.Println("函数调用之前a的值：", a)

fmt.Printf("%T \n", a)

fmt.Printf("%x \n", &a)

//b := &a

var b \*int = &a

change(b)

fmt.Println("函数调用之后的a的值：", a)

}

func change(val \*int) {

\*val = 15

}

运行结果

函数调用之前a的值： 58

int

c420014050

函数调用之后的a的值： 15

例：

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

a := 100

b := 200

//返回值的写法

a, b = swap0(a, b)

//指针作为参数的写法

swap(&a, &b)

fmt.Printf("交换后 a 的值 : %d\n", a)

fmt.Printf("交换后 b 的值 : %d\n", b)

}

//具有返回值的惯用写法

func swap0(x, y int) (int , int) {

return y, x

}

//指针作为参数的写法

func swap(x \*int, y \*int) {

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

}

例：

将一个指向切片的指针传递给函数。虽然将指针传递给一个切片作为函数的参数，可以实现对该切片中元素的修改，但这并不是实现这一目标的惯用方法。惯用做法是使用切片。

package main

import "fmt"

func main() {

a := [3]int{89, 90, 91}

modify(&a)

fmt.Println(a)

}

func modify(arr \*[3]int) {

(\*arr)[0] = 189

}

运行结果

[189 90 91]

建议做法是使用切片来实现更改元素数值，代码如下：

package main

import "fmt"

func main() {

a := []int{89, 90, 91}

modify(a[:])

fmt.Println(a)

}

func modify(sls []int) {

sls[0] = 190

}

4.2.5指针数组

指针数组是指元素为指针类型的数组。

定义一个指针数组，例如：var ptr [3]\*string ；

有一个元素个数相同的数组，将该数组中每个元素的地址赋值给该指针数组。也就是说该指针数组与某一个数组完全对应。

可以通过\*指针变量获取到该地址所对应的数值。

例：

package main

import "fmt"

const COUNT int = 4

func main() {

a := [COUNT]string{"abc", "ABC", "123", "一二三"}

i := 0

//定义指针数组

var ptr [COUNT]\*string

fmt.Printf("%T , %v \n", ptr, ptr)

for i = 0; i < COUNT; i++ {

//将数组中每个元素的地址赋值给指针数组

ptr[i] = &a[i]

}

fmt.Printf("%T , %v \n", ptr, ptr)

//获取指针数组中第一个值，其实就是一个地址

fmt.Println(ptr[0])

//根据数组元素的每个地址获取该地址所指向的元素的数值

for i = 0; i < COUNT; i++ {

fmt.Printf("a[%d] = %s \n", i, \*ptr[i])

}

}

4.2.6 指针的指针

如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。当定义一个指向指针的指针变量时，第一个指针存放第二个指针的地址，第二个指针存放变量的地址：



图 4.2

指向指针的指针变量声明格式如下：

var ptr \*\*int

指向指针的指针变量为整型，访问指向指针的指针变量值需要使用两个 \* 号。

例：

package main

import "fmt"

func main() {

var a int

var ptr \*int

var pptr \*\*int

a = 1234

/\* 指针 ptr 地址 \*/

ptr = &a

fmt.Println("ptr" , ptr)

/\* 指向指针 ptr 地址 \*/

pptr = &ptr

fmt.Println("pptr" , ptr)

/\* 获取 pptr 的值 \*/

fmt.Printf("变量 a = %d\n", a)

fmt.Printf("指针变量 \*ptr = %d\n", \*ptr)

fmt.Printf("指向指针的指针变量 \*\*pptr = %d\n", \*\*pptr)

}

运行结果：

ptr 0xc420014050

pptr 0xc420014050

变量 a = 1234

指针变量 \*ptr = 1234

指向指针的指针变量 \*\*pptr = 1234

4.3 函数的参数传递

函数如果使用参数，该参数变量称为函数的形参。形参就像定义在函数体内的局部变量。调用函数，可以通过两种方式来传递参数。即：值传递和引用传递，或者叫做传值和传引用。

4.3.1 值传递（传值）

值传递是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到原内容数据。

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到原内容数据。

每次调用函数，都将实参拷贝一份再传递到函数中。每次拷贝一份，性能是不是就下降了呢？其实Go语言中使用指针和值传递配合就避免了性能降低问题，也就是通过传指针参数来解决实参拷贝的问题。

4.3.2 引用传递（传引用）

引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到原内容数据。

严格来说Go语言只有值传递一种传参方式，Go语言是没有引用传递的。

Go语言中可以借助传指针来实现引用传递的效果。函数参数使用指针参数，传参时其实是在拷贝一份指针参数，也就是拷贝了一份变量地址。

函数的参数如果是指针，当函数调用时，虽然参数仍然是按拷贝传递的，但是此时仅仅只是拷贝一个指针，也就是一个内存地址，这样就不用担心实参拷贝造成的内存浪费、时间开销、性能降低的情况。

传指针使得多个函数能操作同一个对象。传指针更轻量级 (8bytes)，只需要传内存地址。如果参数是非指针参数，那么值传递的过程中，每次在拷贝上面就会花费相对较多的系统开销（内存和时间）。所以当要传递大的结构体的时候，用指针是一个明智的选择。

Go语言中slice、map、chan类型的实现机制都是类似指针，所以可以直接传递，而不必取地址后传递指针。

例：

1.函数传传int型参数

package main

import "fmt"

func main() {

a := 10

fmt.Printf("1、变量a的内存地址：%p ，值为：%v \n\n", &a, a) //10

fmt.Printf("========int型变量a的内存地址：%p \n\n", a) //？？？%!p

//传值

changeIntVal(a)

fmt.Printf("2、changeIntVal函数调用之后：变量a的内存地址：%p ，值为：%v \n\n", &a, a) //10

//传引用

changeIntPtr(&a)

fmt.Printf("3、changeIntPtr函数调用之后：变量a的内存地址：%p ，值为：%v \n\n", &a, a) //50

}

func changeIntVal(a int) {

fmt.Printf("--------changeIntVal函数内：值参数a的内存地址：%p ，值为：%v \n", &a, a) //10

a = 90

}

func changeIntPtr(a \*int) {

fmt.Printf("--------changeIntPtr函数内：指针参数a的内存地址：%p ，值为：%v \n", &a, a) //地址

\*a = 50

}

2.函数传值和传引用\_传slice型参数

package main

import "fmt"

func main() {

a := []int{1, 2, 3, 4}

fmt.Printf("1、变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[1,2,3,4]

fmt.Printf("切片型变量a内存地址是：%p \n\n", a)//可以获取到地址，类似：0xc420018080

//传值

changeSliceVal(a)

fmt.Printf("2、changeSliceVal函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[1,2,3,4]

//传引用

changeSlicePtr(&a)

fmt.Printf("3、changeSlicePtr函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[250,2,3,4]

}

func changeSliceVal(a []int) {

fmt.Printf("----------changeSliceVal函数内：值参数a的内存地址是：%p ，值为：%v \n", &a, a) //[1,2,3,4]

fmt.Printf("----------changeSlicePtr函数内：值参数a的内存地址是：%p \n", a)

a[0] = 99

}

func changeSlicePtr(a \*[]int) {

fmt.Printf("----------changeSlicePtr函数内：指针参数a的内存地址是：%p ，值为：%v \n", &a, a) //&[1,2,3,4]

(\*a)[1] = 250

}

3.函数传值和传引用\_传数组

package main

import "fmt"

func main() {

a := [4]int{1, 2, 3, 4}

fmt.Printf("1、变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[1,2,3,4]

fmt.Printf("数组型变量a内存地址是：%p \n\n", a)//可以获取到地址？ ❌

//传值

changeArrayVal(a)

fmt.Printf("2、changeArrayVal函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[99,2,3,4] ❌

//传引用

changeArrayPtr(&a)

fmt.Printf("3、changeArrayPtr函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[99,250,3,4] ❌

}

func changeArrayVal(a [4]int) {

fmt.Printf("----------changeArrayVal函数内：值参数a的内存地址是：%p ，值为：%v \n", &a, a) //[1,2,3,4]

fmt.Printf("----------changeArrayPtr函数内：值参数a的内存地址是：%p \n", a) //获取不到地址

a[0] = 99

}

func changeArrayPtr(a \*[4]int) {

fmt.Printf("----------changeArrayPtr函数内：指针参数a的内存地址是：%p ，值为：%v \n", &a, a) //&[1,2,3,4]

(\*a)[1] = 250

}

4.函数传值和传引用\_传struct结构体

package main

import "fmt"

type Teacher struct {

name string

age int

married bool

sex int8

}

func main() {

a := Teacher{"Steven", 35, true, 1}

fmt.Printf("1、变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//{Steven 35 true 1}

fmt.Printf("struct型变量a内存地址是：%p \n\n", a)//可以获取到地址？

//传值

changeStructVal(a)

fmt.Printf("2、changeArrayVal函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//{Steven 35 true 1}

//传引用

changeStructPtr(&a)

fmt.Printf("3、changeArrayPtr函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//

}

func changeStructVal(a Teacher) {

fmt.Printf("----------changeArrayVal函数内：值参数a的内存地址是：%p ，值为：%v \n", &a, a) //

fmt.Printf("----------changeArrayPtr函数内：值参数a的内存地址是：%p \n", a) //获取不到地址?

a.name = "Josh"

a.age = 29

a.married = false

}

func changeStructPtr(a \*Teacher) {

fmt.Printf("----------changeArrayPtr函数内：指针参数a的内存地址是：%p ，值为：%v \n", &a, a) //{Daniel 20 false 1}

(\*a).name = "Daniel"

(\*a).age = 20

(\*a).married = false

}

4.3.3 小结

Go语言中所有的传参都是值传递（传值），都是一个副本，一个拷贝。

拷贝的内容有时候是值类型（int、string、bool、数组、struct属于值类型），这样就在函数中就无法修改原内容数据；有的是引用类型（指针、slice、map、chan属于引用类型），这样就可以修改原内容数据。

是否可以修改原内容数据，和传值、传引用没有必然的关系。在C++中，传引用肯定是可以修改原内容数据的，在Go语言里，虽然只有传值，但是我们也可以修改原内容数据，因为参数可以是引用类型。

传引用和引用类型是两个概念。虽然Go语言只有传值一种方式，但是可以通过传引用类型变量达到跟传引用一样的效果。

第五章 Go语言内置容器

本章重点为大家介绍如下的内容：

* 数组的用法（数组声明、元素访问、值类型、多维数组）
* 切片的用法（创建切片、元素遍历、动态增减元素、内容复制）
* 冒泡排序什么是共识算法
* map（概念、声明、初始化和赋值、遍历map、键值对是否存在、map元素删除、清空map）
* 列表list（概念、声明、初始化、遍历list、插入元素、从list中删除元素）

内置容器概述

1、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）

2、复合数据类型（派生数据类型）：函数与指针、数组、切片、map、list、结构体、通道

3、本章讲解的核心知识点：

1. 数组的用法

2. 切片slice的用法

3. 冒泡排序

4. strings字符串处理包（string类型可以看成是一种特殊的slice类型）

5. strconv包

6. map集合的用法（声明、创建和遍历、map元素删除、查找）

7. list列表的用法

8. 深拷贝与浅拷贝(值类型和引用类型的区别)

9. 随机数及键盘输入

10. time包及math包

11. 利用所学知识练习封装函数

5.1 数组(array)

5.1.1 数组简介

Go 语言提供了数组类型的数据结构。数组是具有相同类型的一组长度固定的数据序列，这种类型可以是任意的基本数据类型或复合数据类型及自定义类型。

数组元素可以通过索引下标（位置）来读取或者修改元素数据。索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。数组的下标取值范围是从0开始，到长度减1。数组一旦定义后，大小不能更改。

5.1.2 一维数组

Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

var 变量名 [数组长度] 数据类型

数组长度必须是整数且大于 0，未初始化的数组不是nil，也就是说没有空数组（与切片不同）

例：

var lines [5]int

var titles [10]string

数组可以在声明的时候直接赋值，这种操作叫数组初始化。

例：

var lines = [5]int{2,4,6,8,10}

var titles = []string{"编号","部门","教研室","姓名"}

初始化数组中 {} 中的元素个数不能大于 [] 中的数字。如果忽略 [] 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小。

可以忽略声明中数组的长度并将其替换为…，编译器会自动计算长度。

例：

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

该实例没有设置数组的大小，与上面的实例是一样的效果。

数组元素可以通过索引（位置）来读取。格式为数组名后加中括号，中括号中为索引的值。例如：

balance[3] = 4

以上实例为数组第4个元素赋值。数组元素可以通过索引（位置）来读取（或者修改），索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。

如果已知一个数组，希望得到数组的长度，需要使用len()函数获得数组长度。通过将数组作为参数传递给len()函数，得到数组的长度。忽略声明中数组的长度并将其替换为…，编译器可以找到长度。

例：

package main

import (

"fmt"

)

func main() {

var titles = []string{"编号","部门","教研室","姓名"}

var persons = [...]string{"小明","小强","小亮"}

fmt.Print("titles:")

fmt.Println(len(titles))

fmt.Print("persons:")

fmt.Println(len(persons))

}

输出结果为：

titles:4

persons:3

通过循环语句与len()函数，可以实现遍历数组所有数据；通过for range也可以实现同样的功能。

例：

package main

import "fmt"

func main() {

a := [4]float64{67.7, 89.8, 21, 78}

b := [...]int{2, 3, 5}

//遍历数组方式1

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

fmt.Print(a[i], "\t")

}

fmt.Println()

// 遍历数组方式2

for \_, value := range b {

fmt.Print(value, "\t")

}

}

输出结果：

67.7 89.8 21 78

2 3 5

5.1.3 多维数组

Go 语言支持多维数组，以下为常用的多维数组声明方式：

var variable\_name [SIZE1][SIZE2]...[SIZEn] variable\_type

以下实例声明了三维的整型数组：

var threedim [5][10][4]int

二维数组是最简单的多维数组，二维数组本质上是由一维数组组成的。二维数组定义方式如下：

var arrayName [ x ][ y ] variable\_type

例：

a = [3][4]int{

{0, 1, 2, 3} , /\* 第一行索引为 0 \*/

{4, 5, 6, 7} , /\* 第二行索引为 1 \*/

{8, 9, 10, 11} /\* 第三行索引为 2 \*/

}

二维数组通过指定坐标来访问。如数组中的行索引与列索引。

例：

int val = a[2][3]

以上实例访问了二维数组 val 第三行的第四个元素。

二维数组可以使用循环嵌套的方式实现数组遍历。

例：

package main

import "fmt"

func main() {

/\* 数组 - 5 行 2 列\*/

var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}

fmt.Println(len(a))

fmt.Println(len(a[0]))

/\* 输出数组元素 \*/

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

for j := 0; j < len(a[0]); j++ {

fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )

}

}

}

Go中的数组是值类型，而不是引用类型。这意味着当它们被分配给一个新变量时，将把原始数组的副本分配给新变量。如果对新变量进行了更改，则不会在原始数组中反映。当将数组传递给函数作为参数时，它们将通过值传递，而原始数组将保持不变。

例：

package main

import "fmt"

func main() {

a := [...]string{"USA", "China", "India", "Germany", "France"}

b := a // a copy of a is assigned to b

b[0] = "Singapore"

fmt.Println("a ：", a)

fmt.Println("b ： ", b)

}

运行结果：

a ： [USA China India Germany France]

b ： [Singapore China India Germany France]

5.2 切片(Slice)

5.2.1 切片简介

Go 语言切片是对数组的抽象。Go语言中数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型切片("动态数组")；

与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。切片本身没有任何数据，它们只是对现有数组的引用。切片与数组相比，不需要设定长度，在[]中不用设定值，相对来说比较自由。从概念上面来说slice像一个结构体，这个结构体包含了三个元素：

* 指针，指向数组中slice指定的开始位置；
* 长度，即slice的长度；
* 最大长度，也就是slice开始位置到数组的最后位置的长度；

5.2.2 切片的语法

1.切片在声明时不需要说明长度，使用一个未指定长度的数组来定义切片如下：

var identifier []type

采用这种声明方式，且未初始化的切片称为空切片。这种切片默认为 nil，长度为 0。

2.使用make()函数来创建切片如下：

var slice1 []type = make([]type, len)

可以简写为：slice1 := make([]type, len)

可以指定容量，其中capacity为可选参数：make([]T, length, capacity)

package main

import "fmt"

func main() {

var numbers = make([]int,3,5)

fmt.Printf("%T\n" , numbers)

fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)

}

切片可以在声明的时候直接赋值，这种操作叫切片初始化。

1.直接初始化切片

例：

s :=[] int {1,2,3 }

2.通过对数组或切片截取来初始化切片

* s := arr[:]，切片中包含数组所有元素；
* s := arr[startIndex:endIndex]，将arr中从下标startIndex到endIndex-1 下的元素创建为一个新的切片（前闭后开），长度为endIndex-startIndex；
* s := arr[startIndex:]，缺省endIndex时将表示一直到arr的最后一个元素；
* s := arr[:endIndex]，缺省startIndex时将表示从arr的第一个元素开始。

例：

package main

import "fmt"

func main() {

arr := [5]int {1,2,3,4,5}

s := arr[:]//完全截取

fmt.Println(s)

s = arr[1:2]//截取从索引1到索引2

fmt.Println(s)

s = arr[3:]//从索引3开始截取到结束

fmt.Println(s)

s = arr[:2]//截取到索引2

fmt.Println(s)

}

输出结果：

[1 2 3 4 5]

[2]

[4 5]

[1 2]

3.通过切片截取来初始化切片：

可以通过设置下限及上限来设置截取切片 [lower-bound:upper-bound]

package main

import "fmt"

func main() {

/\* 创建切片 \*/

numbers := []int{0,1,2,3,4,5,6,7,8}

printSlice(numbers)

/\* 打印原始切片 \*/

fmt.Println("numbers ==", numbers)//[0 1 2 3 4 5 6 7 8]

/\* 打印子切片从索引1(包含) 到索引4(不包含)\*/

fmt.Println("numbers[1:4] ==", numbers[1:4])//[1 2 3 ]

/\* 默认下限为 0\*/

fmt.Println("numbers[:3] ==", numbers[:3])//[0 1 2]

/\* 默认上限为 len(s)\*/

fmt.Println("numbers[4:] ==", numbers[4:])//[4 5 6 7 8]

/\* 打印子切片从索引 0(包含) 到索引 2(不包含) \*/

number2 := numbers[:2]

printSlice(number2) //[0 1]

/\* 打印子切片从索引 2(包含) 到索引 5(不包含) \*/

number3 := numbers[2:5]

printSlice(number3) // [2 3 4]

}

func printSlice(x []int){

fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)

}

运行结果：

len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]

numbers == [0 1 2 3 4 5 6 7 8]

numbers[1:4] == [1 2 3]

numbers[:3] == [0 1 2]

numbers[4:] == [4 5 6 7 8]

len=2 cap=9 slice=[0 1]

len=3 cap=7 slice=[2 3 4]

5.2.3 len() 和 cap() 函数

切片的长度是切片中元素的数量。切片的容量是从创建切片的索引开始的底层数组中元素的数量。切片是可索引的，并且可以由 len() 方法获取长度。切片提供了计算容量的方法 cap()， 可以测量切片最长可以达到多少。切片实际的是获取数组的某一部分，len切片<=cap切片<=len数组。

package main

import "fmt"

func main() {

sliceCap()

}

func sliceCap() {

arr0 := [...]string{"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k"}

fmt.Println("cap(arr0)=", cap(arr0), arr0) //[a b c d e f g h i j k]

//截取数组，形成切片

s01 := arr0[2:8]

fmt.Printf("%T \n", s01) //[]string

fmt.Println("cap(s01)=", cap(s01), s01) //9，[c d e f g h]

s02 := arr0[4:7]

fmt.Println("cap(s02)=", cap(s02), s02) //7，[e f g]

//截取切片，形成切片

s03 := s01[3:9]

fmt.Println("截取s01[3:9]后形成s03：", s03) //[f g h i j k]

s04 := s02[4:7]

fmt.Println("截取s02[4:7]后形成s04：", s04) //[i j k]

//切片是引用类型

s04[0] = "x"

fmt.Print(arr0, s01, s02, s03, s04)

}

输出结果：

cap(arr0)= 11 [a b c d e f g h i j k]

[]string

cap(s01)= 9 [c d e f g h]

cap(s02)= 7 [e f g]

截取s01[3:9]后形成s03： [f g h i j k]

截取s02[4:7]后形成s04： [i j k]

[a b c d e f g h x j k] [c d e f g h] [e f g] [f g h x j k] [x j k]

slice没有自己的任何数据，它只是底层数组的一个引用。对slice所做的任何修改都将反映在底层数组中。数组是值类型，而切片是应用类型。

例：

package main

import "fmt"

func main() {

a := [4]float64{67.7, 89.8, 21, 78}

b := []int{2, 3, 5}

fmt.Printf("变量a —— 地址：%p ， 类型：%T，数值：%v，长度：%d \n", &a, a, a, len(a))

fmt.Printf("变量b —— 地址：%p ， 类型：%T，数值：%v，长度：%d \n", &b, b, b, len(b))

c := a

d := b

fmt.Printf("变量c —— 地址：%p ， 类型：%T，数值：%v，长度：%d \n", &c, c, c, len(c))

fmt.Printf("变量d —— 地址：%p ， 类型：%T，数值：%v，长度：%d \n", &d, d, d, len(d))

a[1] = 200

fmt.Println("a=", a, "c=", c)

d[0] = 100

fmt.Println("b=", b, "d=", d)

}

运行结果：

变量a —— 地址：0xc000010300 ， 类型：[4]float64，数值：[67.7 89.8 21 78]，长度：4

变量b —— 地址：0xc0000044a0 ， 类型：[]int，数值：[2 3 5]，长度：3

变量c —— 地址：0xc0000103a0 ， 类型：[4]float64，数值：[67.7 89.8 21 78]，长度：4

变量d —— 地址：0xc000004520 ， 类型：[]int，数值：[2 3 5]，长度：3

a= [67.7 200 21 78] c= [67.7 89.8 21 78]

b= [100 3 5] d= [100 3 5]

当多个片共享相同的底层数组时，每个元素所做的更改将在数组中反映出来。

例：

package main

import "fmt"

func main() {

//定义数组

arr := [3]int{1, 2, 3}

//根据数组截取切片

nums1 := arr[:]

nums2 := arr[:]

fmt.Println("arr=", arr)//[1 2 3]

nums1[0] = 100

fmt.Println("arr=", arr)//[100 2 3]

nums2[1] = 200

fmt.Println("arr=", arr)//[100 200 3]

}

运行结果：

arr= [1 2 3]

arr= [100 2 3]

arr= [100 200 3]

5.2.4 append() 和 copy() 函数

1.函数append()

append()函数实现往切片尾部追加新元素，函数可以向slice里面追加一个或者多个元素，也可以追加一个切片。

append函数会改变slice所引用的数组的内容，从而影响到引用同一数组的其它slice。当使用append追加元素到切片时，如果容量不够（也就是(cap-len) == 0），Go就会创建一个新的内存地址来储存元素。

2.函数copy()

copy()函数实现复制切片元素。将源切片中的元素复制到目标切片中，返回复制的元素的个数

copy函数是不会建立源切片与目标切片之间的联系。也就是两个切片不存在联系，一个修改不影响另一个。

例：

package main

import "fmt"

func main() {

numbers := make([]int , 0 , 20)

numbers = append(numbers, 2, 3, 4, 5, 6, 7)

numbers1 := make([]int, len(numbers), (cap(numbers))\*2)

/\* 拷贝 numbers 的内容到 numbers1 \*/

count := copy(numbers1, numbers)

fmt.Println("拷贝个数：", count)

printSlices("numbers1:", numbers1)

numbers[len(numbers)-1] = 99

numbers1[0] = 100

/\*numbers1与numbers两者不存在联系，numbers发生变化时，

numbers1是不会随着变化的。也就是说copy方法是不会建立两个切片的联系的

\*/

printSlices("numbers1:", numbers1)

printSlices("numbers:", numbers)

}

func printSlices(name string, x []int) {

fmt.Print(name, "\t")

fmt.Printf("addr:%p \t len=%d \t cap=%d \t slice=%v\n", x , len(x), cap(x), x)

}

输出结果：

拷贝个数： 6

numbers1: addr:0xc000108000 len=6 cap=40 slice=[2 3 4 5 6 7]

numbers1: addr:0xc000108000 len=6 cap=40 slice=[100 3 4 5 6 7]

numbers: addr:0xc000106000 len=6 cap=20 slice=[2 3 4 5 6 99]

5.2.5 切片高级操作

1.删除切片中的元素。

Go语言中并没有为切片删除元素及提供相应的函数，如果需要实现删除切片中的某个元素，可以利用切片截取及append()函数实现删除元素

删除第一个元素使用numbers = numbers[1:]；

删除最后一个元素使用numbers = numbers[:len(numbers)-1]；

删除任意元素n使用numbers = append(numbers[:n] , numbers[n+1:]...)

例：

package main

import "fmt"

func main() {

fmt.Println("1、------------------")

//var numbers []int

numbers := make([]int , 0 , 20)

printSlices("numbers:", numbers)

numbers = append(numbers, 0) //[0]

printSlices("numbers:", numbers)

/\* 向切片添加一个元素 \*/

numbers = append(numbers, 1) //[0 1]

printSlices("numbers:", numbers)

/\* 同时添加多个元素 \*/

numbers = append(numbers, 2, 3, 4, 5, 6, 7) //[0 1 2 3 4 5 6 7]

printSlices("numbers:", numbers)

fmt.Println("2、------------------")

//追加一个切片

s1 := []int{100, 200, 300, 400, 500, 600, 700}

numbers = append(numbers, s1...)

printSlices("numbers:", numbers)

fmt.Println("3、------------------")

//切片删除元素

//删除第一个元素

numbers = numbers[1:]

printSlices("numbers:", numbers)

//删除最后一个元素

numbers = numbers[:len(numbers)-1]

printSlices("numbers:", numbers)

//删除中间一个元素

a := int(len(numbers)/2)

fmt.Println("中间数：" , a)

numbers = append(numbers[:a] , numbers[a+1:]...)

printSlices("numbers:", numbers)

}

func printSlices(name string, x []int) {

fmt.Print(name, "\t")

fmt.Printf("addr:%p \t len=%d \t cap=%d \t slice=%v\n", x , len(x), cap(x), x)

}

运行结果：

1、------------------

numbers: addr:0xc000106000 len=0 cap=20 slice=[]

numbers: addr:0xc000106000 len=1 cap=20 slice=[0]

numbers: addr:0xc000106000 len=2 cap=20 slice=[0 1]

numbers: addr:0xc000106000 len=8 cap=20 slice=[0 1 2 3 4 5 6 7]

2、------------------

numbers: addr:0xc000106000 len=15 cap=20 slice=[0 1 2 3 4 5 6 7 100 200 300 400 500 600 700]

3、------------------

numbers: addr:0xc000106008 len=14 cap=19 slice=[1 2 3 4 5 6 7 100 200 300 400 500 600 700]

numbers: addr:0xc000106008 len=13 cap=19 slice=[1 2 3 4 5 6 7 100 200 300 400 500 600]

中间数： 6

numbers: addr:0xc000106008 len=12 cap=19 slice=[1 2 3 4 5 6 100 200 300 400 500 600]

5.3 冒泡排序

5.3.1 概念：

冒泡排序（Bubble Sort），是一种计算机科学领域的较简单的排序算法。

它重复地遍历要排序的数组元素，一次比较两个元素，如果他们的顺序错误就把他们交换过来。重复地进行直到没有再需要交换，也就是说该数组已经排序完成。

这个算法的名字由来是因为越大的元素会经由交换慢慢“浮”到数列的顶端，故名“冒泡排序”。

5.3.2 冒泡排序算法的原理如下：

1.比较相邻的元素。如果第一个比第二个大，就交换他们两个。

2.对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。

3.针对所有的元素重复以上的步骤，除了最后一个。

4.持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

5.3.3 冒泡排序分析：

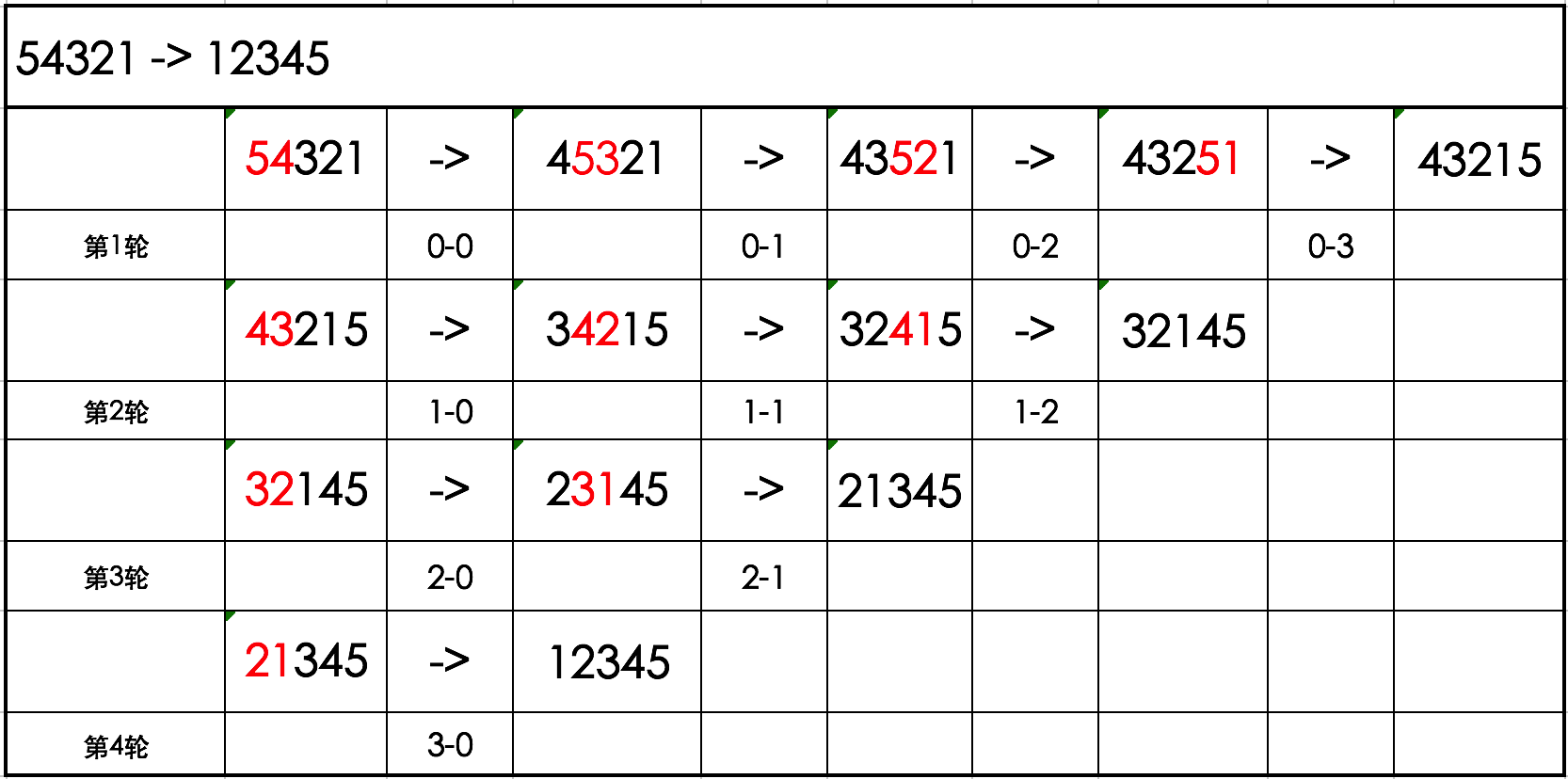


图5.1

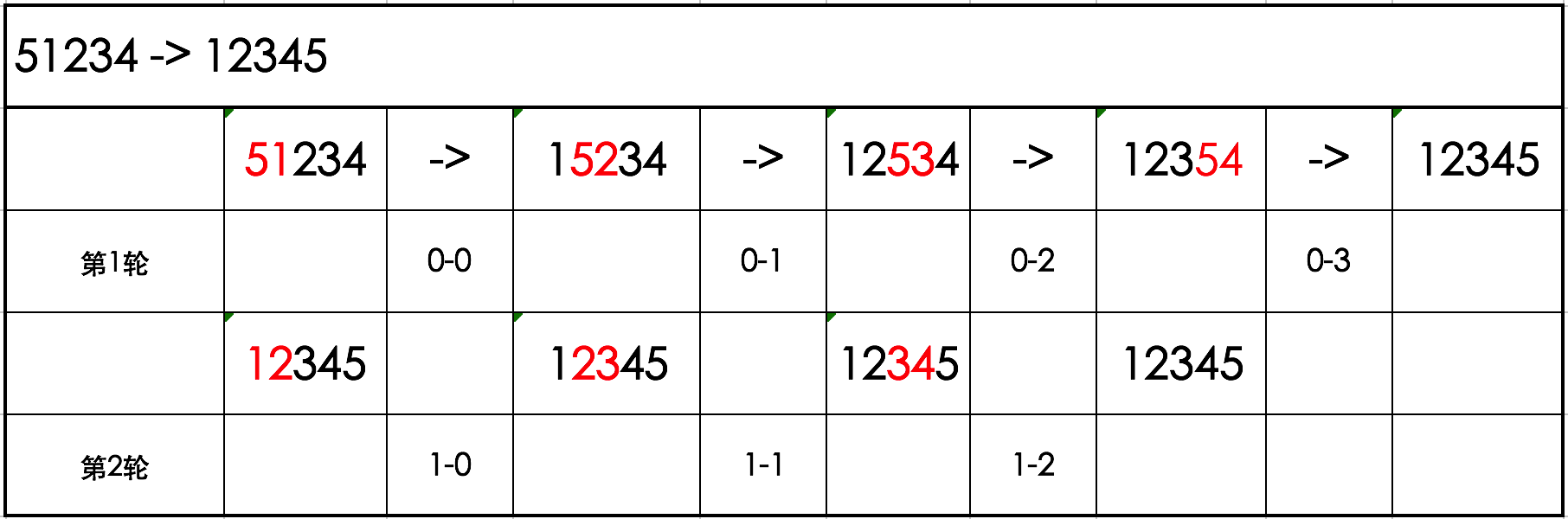


图5.2

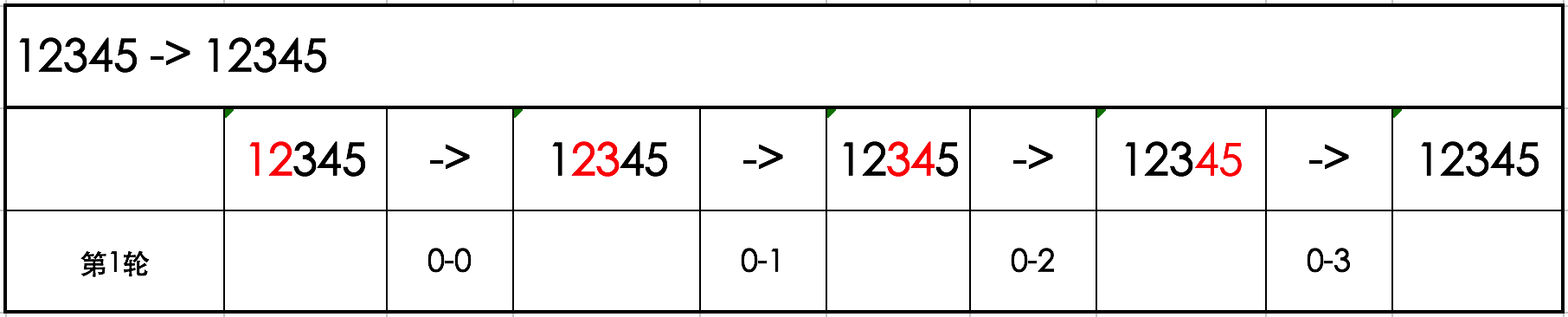


图5.3

5.3.4 冒泡排序核心代码

//从小到大排列

func BubbleSort(arr []int) {

iCount := 0 //记录内循环次数

jCount := 0 //记录外循环的次数

//双层for循环

for i := 0; i < len(arr)-1; i++ {

//定义一个标记，判断本轮是否有两两换位。如果没有换位，那说明排序结束，可以跳出循环。

//例如：12345，当执行第一轮循环后，所有相邻的两个数值都无需换位，那说明排序正常，无需排序。不用执行后续的循环。

flag := true

for j := 0; j < len(arr)-1-i; j++ {

//判断相邻两个数据的大小

if arr[j] > arr[j+1] {

//如果前者比后者大，则执行数据交换

arr[j], arr[j+1] = arr[j+1], arr[j]

//如果出现换位，说明排序还没有结束，需要继续循环执行。

flag = false

}

iCount++

}

jCount++

//如果本轮没有换位，那说明排序结束，则跳出循环

if (flag) {

break

}

}

fmt.Println("i循环次数=", jCount)

fmt.Println("j循环次数=", iCount)

}

5.4 map

5.4.1 map简介

map是Go中的内置类型，它将一个值与一个键关联起来。可以使用相应的键检索值。有资料翻译成地图、映射或字典。但是大多数习惯上翻译成集合。map 是一种无序的键值对（key-value pair ）的集合。map通过 key 来快速检索数据，key 类似于索引，指向相应的value值。

map 是一种集合，所以可以像遍历数组或切片那样去遍历它，不过map 是无序的，所以无法决定它的展示顺序，这是因为 map 是使用 hash 表来实现的。

map是无序的，每次打印出来的map都会不一样，它不能通过index获取，而必须通过key获取；

map的长度是不固定的，和slice一样可以扩展。内置的len()函数同样适用于map，返回map拥有的键值对的数量。但是map不能通过cap()函数计算容量（或者说cap()函数的参数不可以是map）；

同一个map中key必须保证唯一。key的数据类型必须是可参与比较运算的类型，也就是支持==或!=操作的类型。如布尔型、整数型、浮点型、字符串型、数组。对于切片、函数等引用类型则不能作为键的类型；(Invalid map key type: must not be must not be a function , map or slice)。

map的value可以是任何数据类型。和slice一样，map也是一种引用类型；

5.4.2 map的语法

1.使用map关键字定义map

var 变量名 map[key类型]value类型

var声明变量，默认 map 是 nil 。nil map 不能用来存放键值对。var声明后，要么声明时初始化，要么再使用make()函数分配到内存空间，这样才能在其中存放键值对。

例：

声明时初始化数值

var country = map[string]string{

"China": "Beijing",

"France": "Paris",

"Italy": "Rome",

"Japan": "Tokyo",

//"Japan": "Tokyo2",//key名不能重复

}

map1 := map[string]string{

"element": "div",

"width": "100px",

"height": "200px",

"border": "solid",

"color": "red",

"background": "none",

}

rating := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }

2.使用 make 函数

变量名 := make(map[key类型]value类型)

该声明方式，如果不初始化 map，那么map也不等于nil。

例：

定义时初始化

rating := map[string]float32{"C": 5, "Go": 4.5, "Python": 4.5, "C++": 2}

fmt.Println(rating)

创建集合后再赋值

countryCapitalMap := make(map[string]string)

/\* map 插入 key-value 对，各个国家对应的首都 \*/

countryCapitalMap["China"] = "Beijing"

countryCapitalMap["France"] = "Paris"

countryCapitalMap["Italy"] = "Rome"

countryCapitalMap["Japan"] = "Tokyo"

countryCapitalMap["India"] = "New Delhi"

3.map数值遍历

由于map是无序的数据集合，遍历时只能使用迭代遍历的方式，使用for range的方式迭代map集合。

例：

package main

import "fmt"

func main() {

countryCapitalMap := make(map[string]string)

countryCapitalMap["China"] = "Beijing"

countryCapitalMap["France"] = "Paris"

countryCapitalMap["Italy"] = "Rome"

countryCapitalMap["Japan"] = "Tokyo"

countryCapitalMap["India"] = "New Delhi"

for k, v := range countryCapitalMap {

fmt.Println("国家是：", k, "首都：", v)

}

}

输出结果：

国家是： China 首都： Beijing

国家是： France 首都： Paris

国家是： Italy 首都： Rome

国家是： Japan 首都： Tokyo

国家是： India 首都： New Delhi

4.查看元素在集合中是否存在

我们可以通过key获取map中对应的value值。语法为：map[key]

但是当key如果不存在的时候，会得到该value值类型的默认值，比如string类型得到空字符串，int类型得到0。但是程序不会报错。

所以可以通过 value, ok := map[key] ，获取key/value是否存在。ok是bool型，如果 ok 是 true, 则该键值对存在，否则不存在。

例：

captial, ok := countryCapitalMap["United States"]

/\* 如果 ok 是 true, 则存在，否则不存在 \*/

if ok {

fmt.Println("首都是：", captial)

} else {

fmt.Println("该国家的首都没有列出！")

}

5.4.4 delete() 函数

delete(map, key) 函数用于删除集合的某个元素, 参数为 map 和其对应的 key。删除函数不返回任何值。

例：

package main

import "fmt"

func main() {

map1 := map[string]string{

"element": "div",

"width": "100px",

"height": "200px",

"border": "solid",

"color": "red",

"background": "none",

}

fmt.Println("删除前：", map1)

if \_, ok := map1["background"]; ok {

delete(map1, "background")

}

fmt.Println("删除后：", map1)

map1 = make(map[string]string)

fmt.Println("清空后：", map1 , len(map1))

}

输出结果：

删除前： map[background:none border:solid color:red element:div height:200px width:100px]

删除后： map[border:solid color:red element:div height:200px width:100px]

清空后： map[] 0

Go语言没有为map提供任何清空所有元素的函数；清空map的唯一办法是重新make一个新的map；不用担心垃圾回收的效率，Go语言的垃圾回收比写一个清空函数更高效。

与切片相似，map是引用类型。当将map分配给一个新变量时，它们都指向相同的内部数据结构。因此，一个的变化会反映另一个。

例：

package main

import "fmt"

func main() {

personSalary := map[string]int{

"Steven": 18000,

"Daniel": 5000,

"Josh": 20000,

}

fmt.Println("原始薪资：", personSalary)

newPersonSalary := personSalary

newPersonSalary["Daniel"] = 8000

fmt.Println("修改后newPersonSalary：", newPersonSalary)

fmt.Println("personSalary受影响情况：", personSalary)

}

运行结果：

原始薪资： map[Daniel:5000 Josh:20000 Steven:18000]

修改后newPersonSalary： map[Daniel:8000 Josh:20000 Steven:18000]

personSalary受影响情况： map[Daniel:8000 Josh:20000 Steven:18000]

5.5 list

5.5.1 概述

list是一种非连续存储的容器，由多个节点组成，节点通过一些变量记录彼此之间的关系。list有多种实现方法，如单向链表、双向链表等。

假设A、B、C三个都有电话号码，如果A把号码告诉B，B把号码告诉C。这个过程就建立了一个单向链表结构；

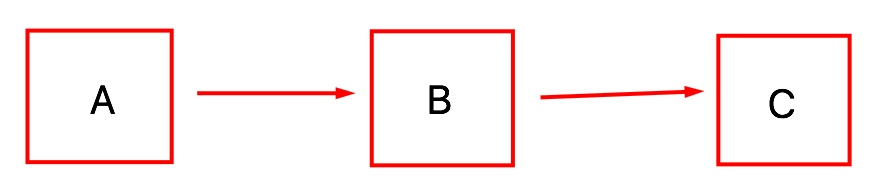


图 5.4

如果在单链表的基础上，再从C开始，将自己的号码给告诉自己号码的人，这样就形成了双向链表结构。

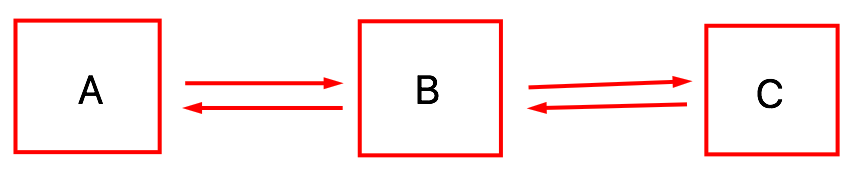


图 5.5

Go语言中list的实现原理是双向链表。list能高效地进行任意位置的元素插入和删除操作。

Golang的标准库提供了高级的数据结构List。具体在包container/list。

container/list包里主要有两个数据结构类型：“Element”、“List”；Element类型代表双向链表中的一个元素，相当于C++里面的"iterator"；List代表一个双向链表。List零值为一个空的、可用的链表。Element有Prev和Next方法用于得到前一个或者下一个Element，Element可以直接调用Value属性；

5.5.2 声明list

list的声明有两种方法：New和List声明。

1.通过container/list包的New方法声明list

变量名 := list.New()

2.通过var声明list

var 变量名 list.List

list与切片和map不同，没有具体元素类型的限制。list中的元素可以是任意类型。在CPP里面，list的成员必须是同一个数据类型，但是Go语言中却允许list中插入任意类型的成员。建议使用New()实现声明list。

5.5.3 element常用方法

1.func (e \*Element) Next() \*Element

Next返回链表的后一个元素或者nil。

2.func (e \*Element) Prev() \*Element

Prev返回链表的前一个元素或者nil。

5.5.4 list常用方法

1.func New() \*List

New创建一个链表。

2.func (l \*List) Init() \*List

Init清空链表。

3.func (l \*List) Len() int

Len返回链表中元素的个数，复杂度O(1)。

4.func (l \*List) Front() \*Element

Front返回链表第一个元素或nil。

5.func (l \*List) Back() \*Element

Back返回链表最后一个元素或nil。

6.func (l \*List) PushFront(v interface{}) \*Element

PushBack将一个值为v的新元素插入链表的第一个位置，返回生成的新元素。

7.func (l \*List) PushFrontList(other \*List)

添加另一个列表到开头。PushFrontList创建链表other的拷贝，并将拷贝的最后一个位置连接到链表l的第一个位置。

8.func (l \*List) PushBack(v interface{}) \*Element

PushBack将一个值为v的新元素插入链表的最后一个位置，返回生成的新元素。

9.func (l \*List) PushBackList(other \*List)

追加另一个列表到末尾。PushBack创建链表other的拷贝，并将链表l的最后一个位置连接到拷贝的第一个位置。

10.func (l \*List) InsertBefore(v interface{}, mark \*Element) \*Element

InsertBefore将一个值为v的新元素插入到mark前面，并返回生成的新元素。如果mark不是l的元素，l不会被修改。

11.func (l \*List) InsertAfter(v interface{}, mark \*Element) \*Element

InsertAfter将一个值为v的新元素插入到mark后面，并返回新生成的元素。如果mark不是l的元素，l不会被修改。

12.func (l \*List) MoveToFront(e \*Element)

MoveToFront将元素e移动到链表的第一个位置，如果e不是l的元素，l不会被修改。

13.func (l \*List) MoveToBack(e \*Element)

MoveToBack将元素e移动到链表的最后一个位置，如果e不是l的元素，l不会被修改。

14.func (l \*List) MoveBefore(e, mark \*Element)

MoveBefore将元素e移动到mark的前面。如果e或mark不是l的元素，或者e==mark，l不会被修改。

15.func (l \*List) MoveAfter(e, mark \*Element)

MoveAfter将元素e移动到mark的后面。如果e或mark不是l的元素，或者e==mark，l不会被修改。

16.func (l \*List) Remove(e \*Element) interface{}

Remove删除链表中的元素e，并返回e.Value。

5.5.5 遍历list

1.顺序遍历

for e := l.Front(); e != nil; e = e.Next() {

fmt.Print(e.Value, " ")

}

2.逆序遍历

for e := l.Back(); e != nil; e = e.Prev() {

fmt.Print(e.Value, " ")

}

5.5.6 list是值类型还是引用类型

1.list本质是什么？

type List struct {

root Element // sentinel list element, only &root, root.prev, and root.next are used

len int // current list length excluding (this) sentinel element

}

type Element struct {

next, prev \*Element

// The list to which this element belongs.

list \*List

// The value stored with this element.

Value interface{}

}

2.struct是值类型

例：

package main

import (

"container/list"

"fmt"

)

func main() {

copyList()

}

//list是值类型，不过采用New()方法声明的是一个指针。所以在拷贝操作和参数传递时具有引用类型的特征。

func copyList() {

//声明list1

list1 := list.New()

printListInfo2("刚声明的list1：", list1)

//给list1赋值

list1.PushBack("one")

list1.PushBack(2)

list1.PushBack("three")

list1.PushFront("first")

printListInfo2("赋值后的list1", list1)

iterateList2(list1)

//将list1拷贝给list2。其实拷贝的是地址

list2 := list1

printListInfo2("刚拷贝的list2", list2)

iterateList2(list2)

//list2修改后

list2.PushBack(250)

list2.PushBack(350)

list2.PushBack(450)

printListInfo2("修改后的list2", list2)

iterateList2(list2)

//list2的修改是否影响到list1？

printListInfo2("修改list2的list1", list1)

iterateList2(list1)

}

func printListInfo2(info string, l \*list.List) {

fmt.Println(info + "----------")

fmt.Printf("%T:%v \t ， 长度为：%d \n", l, l, l.Len())

fmt.Println("----------")

}

func iterateList2(l \*list.List) {

i := 0

for e := l.Front(); e != nil; e = e.Next() {

i++

fmt.Printf("%d:%v \t", i, e.Value)

}

fmt.Println("\n----------")

}

输出结果：

刚声明的list1：----------

\*list.List:&{{0xc00005e330 0xc00005e330 <nil> <nil>} 0} ， 长度为：0

----------

赋值后的list1----------

\*list.List:&{{0xc00005e420 0xc00005e3f0 <nil> <nil>} 4} ， 长度为：4

----------

1:first 2:one 3:2 4:three

----------

刚拷贝的list2----------

\*list.List:&{{0xc00005e420 0xc00005e3f0 <nil> <nil>} 4} ， 长度为：4

----------

1:first 2:one 3:2 4:three

----------

修改后的list2----------

\*list.List:&{{0xc00005e420 0xc00005e510 <nil> <nil>} 7} ， 长度为：7

----------

1:first 2:one 3:2 4:three 5:250 6:350 7:450

----------

修改list2的list1----------

\*list.List:&{{0xc00005e420 0xc00005e510 <nil> <nil>} 7} ， 长度为：7

----------

1:first 2:one 3:2 4:three 5:250 6:350 7:450

----------

Process finished with exit code 0

4.结论：

list是值类型，不过采用list的New()方法声明的是一个指针变量。所以在拷贝操作和参数传递时具有引用类型的特征。