第 3 章

CHAPTER 3

变量与基本数据类型

由于 Go 语言是静态类型语言，因此变量(Variable)是有明确类型的，编译器也会检查变量类型的正确性。

3.1 什么是变量

变量这个词的定义来源于数学，例如，$f(x)=2x+1$，其中 x 就是变量。

在编程中，变量(Variable)是程序在运行的过程中内容可以变化(修改)的量。变量是程序的基本组成单位，是内存中一个数据存储空间的表示，可以通过变量名找到变量值。它的功能是存储用户的数据，是计算机语言中能储存计算结果或能表示值的抽象概念，其内存模拟演示如图 3-1 所示。

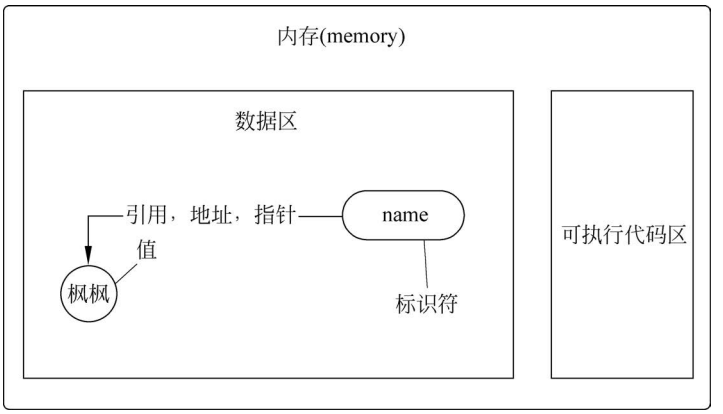

图 3-1 变量的内存模拟

3.2 变量的声明、初始化和赋值

在 Go 语言中，变量的标准声明如下：

```
var name string = "枫枫"
```

其中，var 是关键字，name 是变量名，name 后面的 string 是变量类型，“=”是赋值符号，“枫枫”就是变量对应的值。

对于上面这段代码，标准的读法是将“枫枫”这个字符串赋值给 name 这个变量，而不是 name 变量等于“枫枫”。需要牢记编程中的“=”和数学中的“=”含义是不一样的。

除了标准声明之外，还可以引申以下几种写法。

（1）先声明，再赋值，代码如下：

```
var name string
name = "枫枫"
```

（2）自动类型推导，代码如下：

```
var name = "枫枫"
```

（3）简短赋值，代码如下：

```
name := "枫枫"
```

简短赋值还能用于声明多个变量，代码如下：

```
name, age := "枫枫", 25
```

（4）多行声明，代码如下：

```
var(
    name string
    age int
    )
name = "枫枫"
age = 23
```

变量的类型一旦确定之后，就不能再进行更改了，例如将字符串的值赋给 int 类型的变量，这是不被允许的。

3.3　常量

与变量对应的是常量，常量的值一旦被确定就不能再被修改了，它的定义如下：

```
const NAME string = "枫枫"
```

常量的定义只有两种写法。

（1）自动类型推导，代码如下：

```
const NAME = "枫枫"
const AGE = 23
```

(2) 多行定义,代码如下:

```
const (
    NAME = "枫枫"
    AGE = 23
)
```

一般定义常量使用全大写命名。

iota 是 Go 语言中的一个预定义标识符,它用于创建自增的无类型整数常量。iota 的行为类似于一个计数器,每次在常量声明中出现时自增一次,以下是 iota 的基本用法:

```
const (
    Apple = iota                //0
    Banana                      //1
    Cherry                      //2
)
```

在上述示例中,使用 iota 创建了一组常量。第 1 个常量 Apple 的值为 iota,也就是 0。接下来的常量 Banana 和 Cherry 会自动递增,分别为 1 和 2。

iota 在每个常量声明中按顺序递增,但有一些可以影响它的规则:

(1) iota 从 0 开始递增。

(2) 每个常量声明中的 iota 值会自动递增,不需要显式操作。

(3) 在同一个 const 块中,每次遇到 iota 都会自增。

(4) 在不同的 const 块中,iota 会重新从 0 开始递增。

以下代码用于演示递增规则,代码如下:

```
//unit3/0.常量.go
package main

const (
    One = iota                  //0
    Two                         //1
    Three = 5                   //5
    Four                        //5 (与上一个值相同)
    Five = iota + 1             //5 (显式操作)
    Six                         //6 (继续自增)
)

const (
    _ = iota                    //忽略第 1 个 iota 值 0
    January                     //1
    February                    //2
    March = iota                //3
)
```

在上述示例中展示了多个常量声明块中 iota 的递增行为。可以看到,iota 在每个常量

声明中自动递增，并受到前一个常量值的影响。通过显式操作，可以在递增过程中进行调整或跳过。

3.4　局部变量和全局变量

在函数体内定义的变量称为局部变量，它的作用域是从它定义那一行开始，直到遇到“}”结束或者遇到 return 为止，代码如下：

```
//unit3/1.局部变量.go
package main

import "fmt"

func main() {
    {
        //定义一个局部变量,它的作用域为当前的花括号内
        var name = "枫枫"
        fmt.Println(name)          //可以正常打印
    }
    fmt.Println(name)              //错误,name 作用域只在代码块内
}
```

局部变量还有一个特性，那就是定义的变量必须被使用，例如将上方的第 1 个 print 语句进行注释会看到 name 变量下方会出现一个红色波浪线，鼠标悬浮上去会得到一个变量未使用的错误信息，如图 3-2 所示。能正确地观察错误是一个合格程序员的必备技能。

图 3-2　变量未使用的错误信息

在函数体外定义的变量称为全局变量，只要定义了，在定义前后都能使用，并且在当前包中的任意函数体内也都能使用，代码如下：

```
//unit3/2.全局变量.go
package main

import "fmt"

//变量 className 在定义之前都能使用
var classStudent = className + "的学生"

var className = "三年级二班"

func main() {
    fmt.Println(className)              //三年级二班
    fmt.Println(classStudent)           //三年级二班的学生
}
```

全局变量的定义只能使用关键字声明,也就是不能使用简短赋值方式声明全局变量。全局变量还有一个特性,就是定义之后可以不进行使用。

3.5 变量可见性

在声明变量时,如果该变量的首字母是大写,则表示该变量对外可见;反之,如果该变量的首字母是小写,则表示该变量对外不可见。

这个规则是 Go 语言中最重要的一条规则,也是初学者最容易忽略的问题,要理解这个问题,先看示例:

```
//unit3/pkg/version.go
package pkg

var Version = "1.0.0"
var commit = "第 1 次提交"
```

在 unit3 目录下创建了一个 pkg 目录,然后创建了一个文件名为 version 的 Go 文件,指定这个文件的包名为 pkg,一般和目录名同名。在包中定义了两个全局变量 Version 和 commit,其中 Version 的首字母大写,此变量将在外包中可见。

然后在其他包中导入 pkg 这个包进行使用,代码如下:

```
//unit3/3.变量可见性.go
package main

import (
    "code/unit3/pkg"
    "fmt"
)

func main() {
```

```
    fmt.Println(pkg.Version)            //可以使用
    fmt.Println(pkg.commit)             //不能使用
}
```

代码编辑器也能在使用的过程中显示对外可见的变量，如图 3-3 所示。

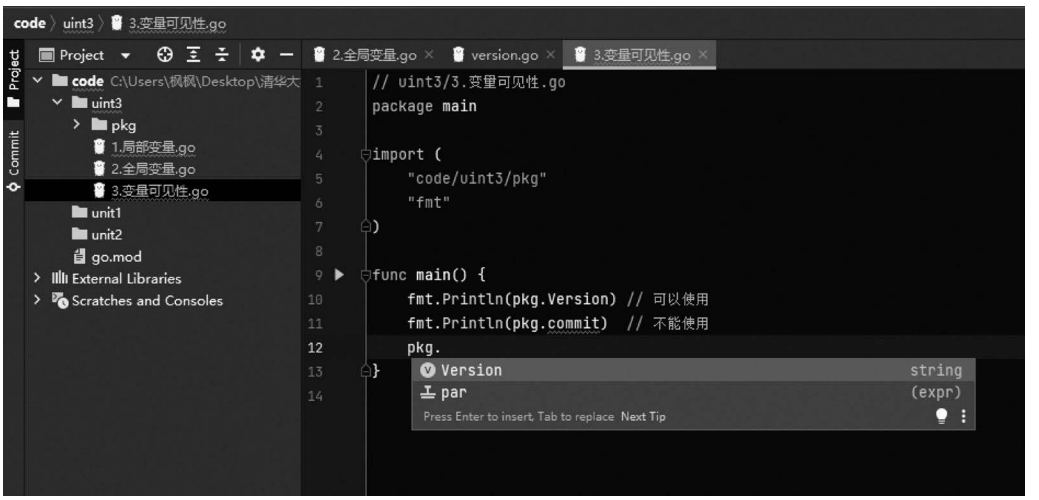

图 3-3　代码编辑器显示可见的变量

该规则不仅针对变量名有效，对于未来学习的函数名、结构体命名、结构体属性命名同样有效，需要牢记此规则。

3.6　注释

在着手编写 Go 代码时，是否考虑过，该编写什么样的代码注释才会使代码读起来易懂呢？不会出现“过了几个月，自己写的代码都不认识了”的情况呢？

注释就是不被机器所执行的代码，是给开发人员看的提示内容。

3.6.1　注释的格式

在 Go 语言中，使用注释有两种写法：

```
//这是单行注释

/*
    这是多行
    注释
*/
```

对于大部分编辑器，可按快捷键 Ctrl+/快速添加注释。

3.6.2 包注释

每个程序包(Package)都应该有一个包注释,该注释用于介绍整个 Package 相关的信息,并且通常设定了对 Package 的期望效果。

包注释不仅可以使用块注释的格式,当然也可以使用行注释的格式,这两种格式在 Go 语言中都非常常用,例如下方是 Go 内置的 path 包的包注释。

```
//Package path implements utility routines for manipulating slash-separated
//paths.
//
//The path package should only be used for paths separated by forward
//slashes, such as the paths in Urls. This package does not deal with
//Windows paths with drive letters or backslashes; to manipulate
//operating system paths, use the [path/filepath] package.
package path
```

解释一下示例中的包注释,第 1 段用于描述接下来的代码文件是一个名为 path 的包。这是 Go 语言约定的一部分,有助于在整个代码库中提供一致的文档。也就是说开头必须声明这个 Package,Package 后面接着是包的名称。

第 2 段用于对包功能进行简要描述。它说明了该包的目的,即实现用于处理斜杠分隔路径的实用程序例程。

3.6.3 命令注释

命令(Command)注释与包注释不同,它描述的是程序的行为,而不是程序包中的功能特征。注释的第一句话的开头通常是 Command 的名称,需要首字母大写(因为是一行的开头)。

```
//unit3/4.命令注释.go
/*
Gofmt formats Go programs.
It uses tabs for indentation and blanks for alignment.
Alignment assumes that an editor is using a fixed-width font.
...
Usage:

    gofmt [flags] [path ...]

The flags are:

    -d
        Do not print reformatted sources to standard output.
        If a file's formatting is different than gofmt's, print diffs
        to standard output.
    -w
        Do not print reformatted sources to standard output.
```

```
        If a file's formatting is different from gofmt's, overwrite it
        with gofmt's version. If an error occurred during overwriting,
        the original file is restored from an automatic backup.
...
*/
package main

func main() {

}
```

注意：命令注释通常使用块注释来表示，内容主要包括命令的功能、命令的用法及参数说明等。

3.6.4　变量注释

变量注释是最常用的注释，用于解释每个变量的作用，防止过段时间看不懂代码的情况发生。

（1）分组注释：可以对常量（const）、变量（variable）进行分组表示，同时一般使用单行注释来说明，示例代码如下：

```
package scanner //import "text/scanner"

//The result of Scan is one of these tokens or a Unicode character.
const (
    EOF = -(iota + 1)
    Ident
    Int
    Float
    Char
    ...
)
```

（2）组内注释：有时，常量、变量里面的每个元素都需要记录其作用，示例代码如下：

```
package unicode //import "unicode"

const (
    MaxRune         = '\U0010FFFF'  //maximum valid Unicode code point.
    ReplacementChar = '\uFFFD'      //represents invalid code points.
    MaxASCII        = '\u007F'      //maximum ASCII value.
    MaxLatin1       = '\u00FF'      //maximum Latin-1 value.
)
```

（3）未分组元素注释：未分组的常量、变量的注释开头通常为名称，示例代码如下：

```
package unicode

//Version is the Unicode edition from which the tables are derived.
const Version = "13.0.0"
```

3.7 规范的代码风格

命名是代码规范中很重要的一部分,统一的命名规则有利于提高代码的可读性,好的命名仅仅通过命名就可以获取足够多的信息。

在命名变量名时,可以使用的字符只能是小写字母、大写字母、数字及下画线,并且首字符不能是数字。

除此之外,Go 语言变量的命名还影响着变量的外部访问。它们遵循如下规则:

(1) 当命名(包括常量、变量、类型、函数名、结构字段等)以一个大写字母开头时,如 Group1,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(客户端程序需要先导入这个包),这被称为导出。

(2) 如果命名以小写字母开头,则对包外是不可见的,但是在整个包的内部是可见的,并且是可用的。

1. 包命名

保持 package 的名字和目录一致,尽量采取简短且有意义的包名,尽量不要和标准库冲突。包名应该为小写单词,不要使用下画线或者混合大小写。

```
package main
package demo
```

2. 文件命名

文件命名规则应该为小写单词,使用下画线分隔各个单词。

```
my_test.go
```

3. 结构体命名

结构体采用驼峰命名法,首字母根据访问控制大写或者小写,struct 声明和初始化格式采用多行,代码如下:

```
type User struct {
    Username string
    Email string
}
u: = User{
    Username: "zal"
    Email: "zal@qq.com"
}
```

4. 接口命名

接口命名规则基本和结构体命名规则一致。

单个函数的结构名以 er 作为后缀,例如 Reader、Writer,代码如下:

```
type Reader interface {
    Read(p []byte)(n int ,err error)
}
```

5. 变量命名

若变量类型为 bool 类型，则名称应以 Has、Is、Can 或 Allow 开头，代码如下：

```
var isExist bool
var hasconflict bool
var canManage bool
var allowGitHook bool
```

和结构体类似，变量名称一般遵循驼峰命令法，首字母根据访问控制原则采用大写或者小写，但当遇到特有名词时，需要遵循以下规则。

(1) 如果变量为私有，并且特有名词为首个单词，则使用小写，如 apiClient。

(2) 其他情况都应当使用该名词原有的写法，如 APIClient、repolD、UserlDo。

(3) 错误示例：UrlArray，应该写成 urlArray 或者 URLArray。

6. 常量命名

常量由全部大写字母组成，并使用下画线分词，代码如下：

```
const APP_VER = "1.0"
```

如果是枚举类型的常量，则需要先创建相应类型，代码如下：

```
type Scheme string
const(
    HTTP Scheme = "http"
    HTTPS Scheme = "https"
)
```

3.8　数据类型的基本介绍

Go 语言有一系列的基本数据类型，包括数字类型、布尔类型和字符串类型。

3.8.1　数字类型

Go 语言中的数字类型较多，包含 int8、int16、int32、int64、int、uint8、uint16、unit32、uint64、uint。这么多的数字类型，难道都需要记住吗？

其实数字类型可分为两大类，一类是有符号类型，其中的符号是指正负号，另一类是无符号类型，后面的数字表示长度，例如 int8 表示有符号 8 位整型，取值范围是－128～127，uint8 表示无符号 8 位整型，取值范围是 0～255。对于没有数字的类型，例如 int 和 uint 类型，则根据操作系统决定数字长度。

这里简单地介绍数字类型的取值范围如何计算，以 int8 为例，它是有符号的，那么需要取出一位来存储符号，并且正向的最后一个是取不到的，所以实际取值范围就是 -2^{8-1}～$2^{8-1}-1$。也就是－128～127。那么 uint8 的取值范围就是 0～2^8-1。

1. 不同进制的表示方法

出于习惯，在初始化数据类型为整型的变量时会使用十进制表示法，因为它最直观，但是也可以使用其他进制表示整数，代码如下：

```
//unit3/5.数字类型.go
package main

import "fmt"

func main() {
    var n1 = 2                  //十进制
    fmt.Println(n1)
    var n2 = 0b10               //二进制
    fmt.Println(n2)
    var n3 = 0o02               //八进制
    fmt.Println(n3)
    var n4 = 0x02               //十六进制
    fmt.Println(n4)
}
```

2. 二进制

由于二进制在计算机中被广泛使用，所以需要简单了解什么是二进制。

二进制，最简单的理解就是一种仅用“1”和“0”的数列组合来表示具体数值的记数方法。它的基数为 2，进位规则是“逢二进一”，借位规则是“借一当二”。

二进制数据是用 0 和 1 这两个数码来表示的数，这两个数码可以对应计算机中的开(1)和关(0)两种状态，因此计算机中的所有数据都是以二进制的形式存储和运算的。

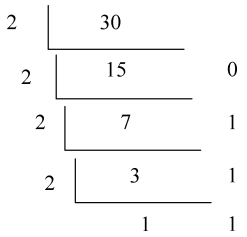

图 3-4　除二取余法

此外，二进制也可以用来表示字母、颜色、图像、声音等多种信息，是计算机技术中广泛采用的一种数制。

如何将十进制数转换为二进制？可以使用除二取余法，以 30 为例，解答步骤如图 3-4 所示。最终 30 的二进制数为 11110。

如何将二进制数转换为十进制？将对应位置的数乘以对应 2 的 n 次方，最后相加即可，还是以 30 的二进制 11110 为例，转换为十进制的代码如下：

```
//unit3/二进制.go
package main

func main() {
```

```
    //1          1          1          1          0
    //1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 0*2^0
    //16 + 8 + 4 + 2 + 0
    //30
}
```

3. 十六进制

十六进制是一种以16为基数的计数系统，使用数字0～9和字母A～F表示数值。它比十进制更简洁，适合在二进制和计算机科学中使用。

十六进制在计算机领域应用普遍，常见的有HTML、CSS的颜色表、MAC地址、字符编码等，例如计算机的物理地址使用十六进制表示，如图3-5所示。

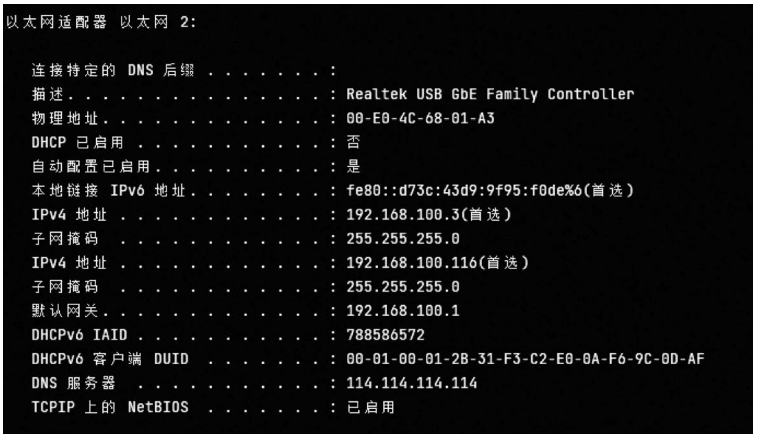

图3-5　计算机的物理地址使用十六进制表示

3.8.2　浮点数类型

在Go语言中，浮点型包括两种类型：float32和float64，分别占用32位和64位内存空间，并用于表示单精度和双精度浮点数。声明和初始化浮点型变量的示例代码如下：

```
var num1 float32      //声明一个float32类型的变量num1,默认初始化为0.0
num2 := 3.14          //使用短变量声明方式声明一个float64类型的变量num2,并初始化为3.14
var num3 float64 = 2.71828          //声明一个float64类型的变量num3,并初始化为2.71828
```

浮点型的精度指的是它可以表示的小数部分的位数。在Go语言中，float32类型的精度约为6位小数，而float64类型的精度约为15位小数。需要注意的是，由于浮点数使用二进制表示，所以浮点数的精确度在进行运算时可能会有一定的损失。

以下是一个示例，展示浮点型精度损失的情况，代码如下：

```
package main

import "fmt"
```

```
func main() {
    num1 := 0.1
    num2 := 0.2
    sum := num1 + num2

    fmt.Println(sum)      //输出结果可能会有精度损失,显示为 0.30000000000000004
}
```

在上述示例中,由于浮点数的精度损失,num1 和 num2 的和可能会显示为一个不太准确的结果。

1. 特殊的浮点数——无穷大

在一些特定的场合中,常常需要用到一个“无穷大”的值,如果直接设置一个特定的值,例如 999 999,则不太合适。在 Go 语言中,可以很方便地实现这一操作,代码如下:

```
//无穷大
num := math.Inf(1)
fmt.Println(num > 99999999)                    //true
fmt.Println(num > 999999999999999999999)       //true
```

2. 特殊的浮点数——NaN

NaN(Not a Number)是一种特殊的浮点数值,在计算中表示无效或未定义的结果。当进行一些数学运算时,如果结果无法确定或不可表达,就会产生 NaN 值。

NaN 值主要具有以下特点:

(1) NaN 不等于任何值,包括自身,因此,NaN !=NaN 会返回 true。

(2) NaN 参与任何数学运算(除了一些特殊的比较操作)都会返回 NaN,例如,NaN+1、NaN * 2、sqrt(NaN) 都会得到 NaN。

(3) NaN 可以通过调用 math.NaN()函数来生成。

代码如下:

```
//创建一个 NaN
nan := math.NaN()

//判断一个数是否是 NaN
isNaN := math.IsNaN(nan)
fmt.Println(isNaN)          //输出:true
```

3.8.3 字符串类型

Go 语言中的字符串是一组只读的字节切片(Slice of bytes),每个字符串都使用一字节或多字节表示。当字符为 ASCII 码表上的字符时占用 1 字节,例如英文字母,其他字符根据需要占用 2～4 字节,例如汉语、日语中的汉字、平假名、片假名等。

字符串的定义是使用双引号进行包裹的，代码如下：

```
var name = "这是一个字符串"
```

如果要设置多行字符串，则可以使用反引号，代码如下：

```
var name = `可以
定义
多行
字符串`
```

1. 转义字符

使用双引号定义的字符串，如何在字符串里面表示双引号本身呢？答案是使用转义字符，示例代码如下：

```
var result = "我的名字是:\"枫枫\""
```

除此之外，还有许多转义字符，只需简单记住常用转义字符，如表 3-1 所示。

表 3-1　赋值运算符及其含义

转　义　符	含　　义	转　义　符	含　　义
\n	换行符	\'	单引号
\r	回车符	\"	双引号
\t	制表符	\\	反斜杠

这些转义字符的使用效果如图 3-6 所示。

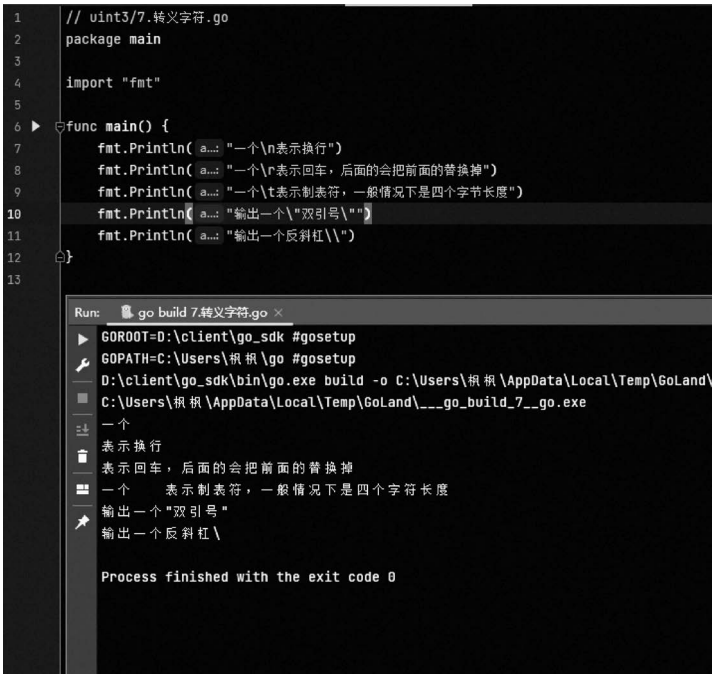

图 3-6　转义字符的使用效果

2. 字符串函数

在Go语言中,字符串处理非常重要,因为字符串在编程中扮演着关键的角色,因此Go语言内置了很多处理字符串的内置函数。

1) len()函数

Go语言的内置函数len()可以用来获取切片、字符串、通道(Channel)等的长度,代码如下:

```
str01 := "Hello World!"
str02 := "你好"
fmt.Println(len(str01))                    //12
fmt.Println(len(str02))                    //6
```

len()函数返回一个整数,表示字符串的ASCII字符个数或字节长度。

这里需要注意的一点是,由于在计算机中,中文是以UTF-8格式保存的,每个中文占用3字节,因此使用len()函数获得两个中文文字对应的6字节。

2) RuneCountInString()函数

如果希望按习惯上的字符个数来计算,就需要使用Go语言中UTF-8包提供的RuneCountInString()函数,统计Unicode字符数量,代码如下:

```
name := "枫枫知道"
fmt.Println(len(name))                     //12
fmt.Println(utf8.RuneCountInString(name))  //4
```

3) Contains()函数

使用strings.Contains()函数检查字符串是否包含指定的子串。strings.Contains()函数用于检查一个字符串是否包含另一个子串。它返回一个布尔值,表示是否找到了子串,代码如下:

```
str1 := "hello world"
fmt.Println(strings.Contains(str1, "world"))    //true
fmt.Println(strings.Contains(str1, "abc"))      //false
```

4) Count()函数

strings.Count()函数用于统计一个字符串中子串出现的次数。它返回一个整数,表示子串在字符串中出现的次数,示例代码如下:

```
str2 := "hello world"
fmt.Println(strings.Count(str2, "l"))      //3
fmt.Println(strings.Count(str2, "a"))      //0
```

5) Split()函数

strings.Split()函数用于对一个字符串按照指定的分隔符进行分割,得到一个字符串切片。它返回一个切片,其中包含被分割后的子串,代码如下:

```
str := "apple, banana, orange"
fruits := strings.Split(str, ",")
fmt.Println("fruits:", fruits)     //fruits: [apple banana orange]
```

6）HasPrefix()函数

strings.HasPrefix()函数用于检查一个字符串是否以指定的前缀开头。它返回一个布尔值，表示字符串是否以指定的前缀开头，代码如下：

```
str3 := "this is a apple"
fmt.Println(strings.HasPrefix(str3, "thi"))          //true
fmt.Println(strings.HasPrefix(str3, "abc"))          //false
```

7）HasSuffix()函数

strings.HasSuffix()函数用于检查一个字符串是否以指定的后缀结尾。它返回一个布尔值，表示字符串是否以指定后缀结尾，代码如下：

```
fileName := "test.go"
fmt.Println(strings.HasSuffix(fileName, ".go"))       //true
fmt.Println(strings.HasSuffix(fileName, ".txt"))      //false
```

8）Replace()函数

strings.Replace()函数用于在一个字符串中将指定的子串替换为新的子串。它返回一个新的字符串，其中完成了替换操作，代码如下：

```
str4 := "今天中午吃橘子,橘子太好吃了"
fmt.Println(strings.Replace(str4, "橘子", "红烧肉", 1))          //今天中午吃红烧肉,橘子
                                                                 //太好吃了
fmt.Println(strings.Replace(str4, "橘子", "红烧肉", 2))          //今天中午吃红烧肉,红烧
                                                                 //肉太好吃了
fmt.Println(strings.Replace(str4, "橘子", "红烧肉", -1))         //今天中午吃红烧肉,红烧
                                                                 //肉太好吃了
```

第4个参数表示替换的次数，可以使用－1表示全部替换，如果要表示全部替换，则可以直接使用strings.ReplaceAll()函数，在其内部也是使用－1来实现的，代码如下：

```
fmt.Println(strings.ReplaceAll(str4, "橘子", "红烧肉"))       //今天中午吃红烧肉,红烧肉
                                                              //太好吃了
```

3. 格式化输出

有时，需要输出该变量的类型，或者将浮点数保留两位小数进行输出，可使用Go语言的%占位符进行格式化输出。

Go语言的占位符比较多，归纳为四大类：

（1）通用类占位符，如表3-2所示。

表 3-2 通用类占位符

占 位 符	含 义
%T	获取数据类型
%v	获取数据的值
%+v	获取数据的值,如果是结构体,则会携带字段名
%#v	获取数据的值,如果是结构体,则会携带结构体名和字段名
%%	字面上的一个%

代码如下:

```
//unit3/格式化输出/1.通用类.go
package main

import "fmt"

func main() {
    name := "枫枫知道"
    age := 18
    type Info struct {
        Name string
        Age int
    }
    var info = Info{"枫枫", 25}

    fmt.Printf("%T %T %T\n", name, age, info)          //string int main.Info
    fmt.Printf("%v %v %v\n", name, age, info)          //枫枫知道 18 {枫枫 25}
    fmt.Printf("%+v %+v %+v\n", name, age, info)       //枫枫知道 18 {枫枫 25}
    fmt.Printf("%#v %#v %#v\n", name, age, info)       //"枫枫知道" 18
main.Info{Name: "枫枫", Age:25}
    fmt.Printf("单独表示一个%%号\n")
}
```

(2) 整数类占位符,如表 3-3 所示。

表 3-3 整数类占位符

占 位 符	含 义
%b	二进制表示
%c	相应 Unicode 码点所表示的字符
%d	十进制表示
%o	八进制表示
%q	单引号围绕的字符字面值,由 Go 语法安全地转义
%x	十六进制表示,字母形式为小写 a-f
%X	十六进制表示,字母形式为大写 A-F
%U	Unicode 格式:U+1234,等同于“U+%04X”

代码如下：

```
//unit3/格式化输出/2.整数类.go
package main

import "fmt"

func main() {
    num := 97
    fmt.Printf("%b\n", num)          //1100001
    fmt.Printf("%c\n", num)          //a
    fmt.Printf("%d\n", num)          //97
    fmt.Printf("%o\n", num)          //141
    fmt.Printf("%q\n", num)          //'a'
    fmt.Printf("%x\n", num)          //61
    fmt.Printf("%X\n", num)          //61
    fmt.Printf("%U\n", num)          //U+0061
}
```

(3) 浮点数类占位符，如表 3-4 所示。

表 3-4　浮点数类占位符

占　位　符	含　　义
%e	科学记数法，例如－1234.456e＋78
%E	科学记数法，例如－1234.456E＋78
%f	有小数点而无指数，例如 123.456
%g	根据情况选择%e 或%f 以产生更紧凑的浮点数(无末尾的 0)
%G	根据情况选择%E 或%f 以产生更紧凑的浮点数(无末尾的 0)

代码如下：

```
//unit3/格式化输出/3.浮点数类.go
package main

import "fmt"

func main() {
    var f = 314.15926000
    fmt.Printf("%e\n", f)          //3.141593e+02
    fmt.Printf("%E\n", f)          //3.141593E+02
    fmt.Printf("%f\n", f)          //314.159260
    fmt.Printf("%.2f\n", f)        //314.16
    fmt.Printf("%.0f\n", f)        //314
    fmt.Printf("%g\n", f)          //314.15926
    fmt.Printf("%G\n", f)          //314.15926
}
```

(4) 字符串类占位符,如表 3-5 所示。

表 3-5 字符串类占位符

占 位 符	含 义
%s	输出字符串表示(string 类型或[]byte)
%q	双引号围绕的字符串,由 Go 语法安全地转义
%x	十六进制,小写字母,每字节两个字符
%X	十六进制,大写字母,每字节两个字符

代码如下:

```
//unit3/格式化输出/4.字符串类.go
package main

import "fmt"

func main() {
    name := "枫枫知道"
    fmt.Printf("%s\n", name)          //枫枫知道
    fmt.Printf("%q\n", name)          //"枫枫知道"
    fmt.Printf("%x\n", name)          //e69eabe69eabe79fa5e98193
    fmt.Printf("%X\n", name)          //E69EABE69EABE79FA5E98193
}
```

(5) 布尔占位符,如表 3-6 所示。

表 3-6 布尔占位符

占 位 符	含 义
%t	true 或 false

3.8.4 布尔类型

布尔类型(Bool)是一种基本数据类型,用于表示逻辑值,即真或假、是或否。它主要用于条件判断和逻辑运算。

Go 语言中的布尔类型关键字为 bool,布尔类型的变量只有两个取值:true 和 false,在默认情况下,未初始化的布尔类型的变量值为 false。

声明布尔变量,代码如下:

```
var isDone bool             //未初始化,默认为 false
var isActive = true         //初始化为 true
var isReady := false        //简短声明方式,初始化为 false
```

3.8.5 字符类型

定义一个字符串需要使用双引号包裹,那么使用单引号包裹会怎样?

在 Go 语言中，使用单引号赋值的就是字符类型，示例代码如下：

```
var char byte = 'a'
```

需要注意的是，不是所有字符都可以直接赋值，在 ASCII 码表中的字符才能进行赋值。如果是单个中文，则需要使用 rune 类型赋值，代码如下：

```
var char rune = '中'
```

其中的根本原因是，byte 的实际类型是无符号 8 位 int 类型，简称 uint8，它的实际长度就是 0～255，所以无法存下中文、日文、韩文等非 ASCII 的字符。rune 类型的实际类型是有符号的 32 位整型，简称 int32，有 4 字节长度。

3.9　扩展——字符编码

上文多次提到关于中文的存储和英文的存储不太一样，例如使用 len()函数得到的一个中文长度为 3，以及 byte 类型不能存储中文等问题，有没有想过这是为什么呢？中文字符在计算机中究竟是如何进行存储的？

3.9.1　ASCII 第 1 个编码集合的诞生及发展

起初计算机刚发展时，发明这种机器的人使用 8 个不同位数的开关（只能是 0 和 1）表示不同的状态，称为“字节”，字节是所有编码的基础，所有的编码都由字节组成。后来根据这些字节发明出了一种处理字节的机器，发明者将其称为计算机。

最开始计算机被发明于美国，8 个 0 和 1 可以组成 256 种组合，于是发明者约定每个组合状态代表一个用途，例如，如果遇上 0×10，终端就换行，如果遇上 0×07，终端就向人们嘟嘟叫，如果遇上 0x1b，打印机就打印反白的字，或者终端就用彩色显示字母。由于 0x20 以下的组合基本是用来控制计算机的，因此被称为“控制码”。后续又把 0x20～127 之间的组合全部用字符、符号和数字等填入，这样这些规则就可以支撑起起初的需求了。于是，被称作为美国信息交换标准代码（American Standard Code for Information Interchange，ASCII）诞生，如图 3-7 所示。

但随着计算机的发展，很多国家的语言使用的不是英文，于是美国国家标准协会把 127 后面剩下的组合使用画表格时需要用到的横线、竖线、交叉等形状及带音标的字母等全部填入，127 后面的字符集就被称为“扩展字符集”。

3.9.2　GBK 编码的诞生及发展

随着计算机的进一步发展，中国引入了计算机，但当计算机传到中国时，起初 256 个位置已经全部被填满了，况且中国汉字众多，就算 256 个位置全部未被使用，也完全不够 6000 多个常用汉字使用。于是定义了一个新的编码规则：一个小于 127 的字符的意义与原来相同，

高四位 / 低四位		ASCII非打印控制字符										ASCII打印字符												
		0000					0001					0010		0011		0100		0101		0110		0111		
		0					1					2		3		4		5		6		7		
		十进制	字符	ctrl	代码	字符解释	十进制	字符	ctrl	代码	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	ctrl
0000	0	0	BLANK NULL	^@	NUL	空	16	▶	^P	DLE	数据链路转意	32		48	0	64	@	80	P	96	`	112	p	
0001	1	1	☺	^A	SOH	头标开始	17	◀	^Q	DC1	设备控制1	33	!	49	1	65	A	81	Q	97	a	113	q	
0010	2	2	☻	^B	STX	正文开始	18	↕	^R	DC2	设备控制2	34	"	50	2	66	B	82	R	98	b	114	r	
0011	3	3	♥	^C	ETX	正文结束	19	‼	^S	DC3	设备控制3	35	#	51	3	67	C	83	S	99	c	115	s	
0100	4	4	♦	^D	EOT	传输结束	20	¶	^T	DC4	设备控制4	36	$	52	4	68	D	84	T	100	d	116	t	
0101	5	5	♣	^E	ENQ	查询	21	§	^U	NAK	反确认	37	%	53	5	69	E	85	U	101	e	117	u	
0110	6	6	♠	^F	ACK	确认	22	■	^V	SYN	同步空闲	38	&	54	6	70	F	86	V	102	f	118	v	
0111	7	7	•	^G	BEL	震铃	23	↨	^W	ETB	传输块结束	39	'	55	7	71	G	87	W	103	g	119	w	
1000	8	8	◘	^H	BS	退格	24	↑	^X	CAN	取消	40	(	56	8	72	H	88	X	104	h	120	x	
1001	9	9	○	^I	TAB	水平制表符	25	↓	^Y	EM	媒体结束	41	)	57	9	73	I	89	Y	105	i	121	y	
1010	A	10	◙	^J	LF	换行/新行	26	→	^Z	SUB	替换	42	*	58	:	74	J	90	Z	106	j	122	z	
1011	B	11	♂	^K	VT	竖直制表符	27	←	^[	ESC	转意	43	+	59	;	75	K	91	[	107	k	123	{	
1100	C	12	♀	^L	FF	换页/新页	28	∟	^\	FS	文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|	
1101	D	13	♪	^M	CR	回车	29	↔	^]	GS	组分隔符	45	-	61	=	77	M	93	]	109	m	125	}	
1110	E	14	♫	^N	SO	移出	30	▲	^6	RS	记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~	
1111	F	15	☼	^O	SI	移入	31	▼	^-	US	单元分隔符	47	/	63	?	79	O	95	_	111	o			

图 3-7　ASCII 字符表

但两个大于127的字符连在一起时，就表示一个汉字，前面的一字节（称为高字节）从0xA1用到0xF7，后面一字节（称为低字节）从0xA1到0xFE，这样就可以组合出7000多个简体汉字了。不仅包含了中国汉字，还把数学符号、罗马数字、希腊字母、日文的假名都编进去了，连在ASCII里本来就有的数字、标点、字母都重新编了两字节长的编码，这就是常说的“全角”字符，而原来在127号以下的那些就叫“半角”字符了。于是GB2312诞生了，GB2312是对ASCII中文字符集的扩展。

但随着计算机进一步地在中国发展，使用范围越来越广，很多人发现自己的名字打不出来，于是不得不继续把GB2312没有用到的码位找出来。后来还是不够用，于是不再要求低字节一定是127号之后的内码，只要第1字节是大于127就固定表示这是一个汉字的开始，不管后面跟的是不是扩展字符集里的内容。结果扩展之后的编码方案被称为GBK标准，GBK包括GB2312的所有内容，同时又增加了近20 000个新的汉字（包括繁体字）和符号。后来少数民族也要用计算机了，于是再扩展，又加了几千个新的少数民族的字，GBK扩成了GB18030。

于是DBCS（Double Byte Character Set，双字节字符集）标准诞生了。在DBCS系列标准里，最大的特点是两字节长的汉字字符和一字节长的英文字符并存于同一套编码方案里，因此为了支持中文处理，必须注意字串里的每字节的值，如果这个值是大于127的，就认为一个双字节字符集里的字符出现了。此时，1中文字符＝2英文字符。

3.9.3　Unicode编码的诞生

双字节字符集诞生了，这意味着其他国家也能像中国一样构造自己的编码体系，但随着发展出现了新的问题：每个国家都有自己的编码，要正常地显示每个国家的编码，就必须装上相应编码的字符系统，但这么多国家，每个国家都装一套字符系统既容易出错，也浪费空间。

随着这种问题的产生及发展，ISO（国际标准化组织）决定解决这个问题，于是废除了所有的地区性编码方案，重新制定一个包括地球上所有文字、所有字母和符号的编码。称为Universal Multiple-Octet Coded Character Set，简称UCS，俗称Unicode。

Unicode开始制定时，计算机的存储器容量极大地发展了，空间再也不成问题了。于是ISO就直接规定必须用两字节，也就是16位来统一表示所有的字符，对于ASCII里的那些“半角”字符，Unicode保持其原编码不变，只是将其长度由原来的8位扩展为16位，而其他文化和语言的字符则全部重新统一编码。由于“半角”英文符号只需用到低8位，所以其高8位永远是0，因此这种方案在保存英文文本时会浪费空间。

3.9.4　UTF系列编码的诞生及发展

Unicode在很长一段时间内无法推广，直到互联网的出现，为了解决Unicode如何在网络上传输的问题，于是面向传输的众多UTF（UCS Transfer Format）标准出现了，顾名思

义,UTF-8 就是每次采用 8 位传输数据,而 UTF-16 就是每次采用 16 位传输数据。UTF-8 是互联网上使用最广的一种 Unicode 的实现方式,这是为传输而设计的编码,并使编码无国界,这样就可以显示全世界所有文化的字符了。

UTF-8 最大的一个特点就是它是一种变长的编码方式。它可以使用 1～4 字节表示一个符号,根据不同的符号而变化字节长度,当字符在 ASCII 码的范围时,就用一字节表示,保留了 ASCII 字符一字节的编码作为它的一部分,需要注意的是 Unicode 一个中文字符占 2 字节,而 UTF-8 一个中文字符占 3 字节。从 Unicode 到 UTF-8 并不是直接对应的,而是要通过一些算法和规则来转换。

3.9.5 扩展——乱码之王锟斤拷

乱码通常是由于编码时使用的字符集和解码时使用的字符集不相同。锟斤拷通常在 UTF-8 与中文编码的转换过程中出现。

Unicode 字符集有一个专门用于提示用户字符无法识别或展示的替换符号,如图 3-8 所示。

图 3-8 替换符号

如果有 UTF-8 无法识别的字符,则会用这个问号替换,在 UTF-8 中对应的十六进制为“EF BF BD”,如果有两个连着的问号替换符,十六进制则为“EF BF BD EF BF BD”。

这时再用 GBK 中文编码解码则会出现锟斤拷,因为 GBK 编码中,每个汉字用两字节。

“EF BF”对应拷,“BD EF”对应斤,“BF BD”对应拷,乱码之王锟斤拷就是这样来的。

3.10 基本数据类型的默认值

在定义变量时,有一种情况是先声明再赋值。

如果变量只声明不赋值,则变量对应的值是多少,在 Go 语言中,一旦变量声明,就会给变量一个对应类型的默认值。

常见的基本数据类型对应的默认值如表 3-7 所示。

表 3-7 常见的基本数据类型对应的默认值

数据类型	默认值
int	0
float	0.000000
string	空字符串
结构体	根据结构体内部的基础数据类型进行初始化赋值
数组、切片	空数组
指针	nil

代码如下：

```
//unit3/9.数据类型默认值.go
package main

import "fmt"

type UserInfo struct {
    Name string
    Age int
    Sex string
    Flag bool
}

//main 函数
func main() {
    var a int
    var b bool
    var c float64
    var d byte
    var e string
    var f UserInfo
    var g *UserInfo
    var ip *int
    var bp *bool
    var fp *float64
    var sp *string
    var ssp *byte
    var iArray []int

    fmt.Println("-------- 默认值列表 --------- ")
    fmt.Printf("int 的默认值为 %d\n", a)                          //0
    fmt.Printf("bool 的默认值为 %t\n", b)                         //false
    fmt.Printf("float64 的默认值为 %f\n", c)                      //0.000000
    fmt.Printf("byte 的默认值为 %b\n", d)                         //0
    fmt.Printf("string 的默认值为 %s\n", e)                       //
    fmt.Printf("结构体 UserInfo 的默认值为 %v\n", f)              //{0 false}
    fmt.Printf("结构体指针 UserInfo 的默认值为 %v\n", g)          //<nil>
    fmt.Printf("int 切片的默认值为 %v\n", iArray)                 //[]
    fmt.Printf("int 指针的默认值为 %p\n", ip)                     //0x0
    fmt.Printf("byte 指针的默认值为 %p\n", bp)                    //0x0
    fmt.Printf("string 指针的默认值为 %p\n", fp)                  //0x0
    fmt.Printf("float64 指针的默认值为 %p\n", sp)                 //0x0
    fmt.Printf("byte 指针的默认值为 %p\n", ssp)                   //0x0
}
```

3.11　类型转换

类型转换是将一种类型通过一定方法变换到其他类型，例如将数字转换为字符串。

3.11.1 显式类型转换

显式类型转换可以通过类型转换操作符将一个值转换为指定的类型,其语法如下:

```
type_name(expression)
```

其中,type_name 为数据类型,expression 为表达式或变量值。注意:类型转换可能会导致精度损失或溢出,因此,在进行类型转换时,需要确保目标类型能够容纳原始值的范围,否则可能会产生不正确的结果。

在下面的示例代码中,将一个 float64 类型的值 66.66 显式地转换为 int 类型,并将结果赋给变量 y。由于 int 类型无法容纳小数部分,故转换后的值为 66,代码如下:

```
package main

import "fmt"

func main() {
    var x float64 = 66.66
    //将 float64 类型的变量显式地转换为 int 类型
    var y int = int(x)
    //输出: 66
    fmt.Println(y)
}
```

3.11.2 隐式类型转换

隐式类型转换是指在表达式中自动进行的类型转换,这种类型转换通常发生在不同类型的数据之间进行运算时。Go 语言会自动将它们转换成相同的类型,以确保表达式的合法性和正确性。如果要真正理解隐式类型转换,则需要先了解常量。

常量是指不能改变值的特殊变量,分为两种:未命名常量和命名常量。未命名常量只在编译期间存在,不会存储在内存中。命名常量存在于内存静态区,不允许修改,例如,const a=66 这条语句,66 是未命名常量,a 是命名常量;编译后,a 的值固定为 66,而等号右边的 66 不再存在。

除了位运算、未命名常量外,运算符两边的操作数类型必须相同,否则会发生编译错误。如果运算符两边是不同类型的未命名常量,则隐式转换的优先级为整数(int)<字符型(rune)<浮点数(float)<复数(Complex),代码如下:

```
package main

import "fmt"

func main() {
```

```
    //由于 66 和 100 的类型相同,所以不需要隐式类型转换
    const num1 = 66 / 100
    //整数会被优先转换为浮点数 100.0, 结果为 0.66
    const num2 = 66 / 100.0

    const num3 int = 88
    //num3 为命名常量,与 1.5 的数据类型不同,由于无法进行隐式类型转换,所以会发生编译错误
    const num4 = num3 * 1.5
    fmt.Println(num1, num2, num3, num4)
}
```

3.11.3　strconv 包

strconv 是 Go 语言标准库中的一个包,用于字符串与基本数据类型之间的相互转换。它提供了一系列函数,可以用于将字符串解析为各种基本数据类型,或将基本数据类型格式化为字符串。strconv 包在处理用户输入、配置文件解析等场景中非常有用。

1. 字符串转整数(Atoi)

Atoi 函数的语法如下:

```
func Atoi(s string) (int, error)
```

它接受一个字符串参数 s,并返回两个值:一个 int 类型的整数和一个 error 类型的错误。如果转换成功,则函数返回转换后的整数和 nil 的错误。如果转换失败,则函数返回一个非 nil 的错误,其中错误信息描述了转换失败的原因。

注意:如果输入的字符串无法解析为整数,或者包含了超出整数范围的值,则转换将失败,函数会返回一个错误。

代码如下:

```
//unit3/类型转换/1.atoi.go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    num, err := strconv.Atoi("123")
    fmt.Println(num, err) //123 <nil>
    num, err = strconv.Atoi("abc")
    fmt.Println(num, err) //0 strconv.Atoi: parsing "abc": invalid syntax
}
```

2. 整数转字符串(Itoa)

Itoa 函数的语法如下：

```
func strconv.Itoa(i int)
```

Itoa 函数接收一个整数参数 i,并返回一个对应的字符串表示,代码如下：

```
//unit3/类型转换/2.itoa.go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    fmt.Println(strconv.Itoa(123))          //123
    fmt.Println(strconv.Itoa(0b111))        //7
    fmt.Println(strconv.Itoa(0xaa))         //170
}
```

3.12 输入函数

当编写程序时,在很多情况下需要从键盘将数据输入程序中。这种用户输入可以是任何类型的数据,例如字符串、整数、浮点数等。用户输入的数据可以用于程序的逻辑运算、计算、条件判断等。

在 Go 语言中,可以使用标准库中的 fmt 包提供的输入函数来读取用户的输入。这些函数提供了不同的方式来获取用户输入,并将其存储到指定的变量中。

常见的输入函数有以下几种。

(1) Scan：从标准输入中读取一行文本,并将其存储到指定的变量中。它以空格为分隔符,将输入的文本分隔成多个字符串,并将它们依次存储到指定的变量中。

(2) Scanf：从标准输入中读取格式化的文本,并将其存储到指定的变量中。它可以根据指定的格式字符串解析输入的文本,并将其存储到指定的变量中。

(3) Scanln：类似于 Scan,但是它会读取一整行文本,直到遇到换行符为止,并将其存储到指定的变量中。

这些输入函数可以根据需要读取不同类型的数据,并将其转换为相应的数据类型。

需要注意的是,用户输入通常需要进行错误处理和数据验证,以确保输入的数据符合预期,例如,通过检查输入的数据类型、范围或格式等来验证用户输入的有效性,以避免潜在的错误或异常。

3.12.1　Scanf 函数

使用占位符，识别对应类型的内容，Scanf 常见占位符如表 3-8 所示。

表 3-8　Scanf 常见占位符

占　位　符	介　　绍	占　位　符	介　　绍
%v	按数据原格式	%t	布尔型
%d	十进制整型	%c	字符型
%f	浮点数	%s	字符串型

代码如下：

```
//unit3/scanf.go
package main

import "fmt"

func main() {
    var a int
    var b float64
    var str string
    var c bool
    //注意:在 Go 语言中字符类型不是 char,而是 byte
    var d byte

    //测试的是这个
    fmt.Scanf("%d %f %s %t %c", &a, &b, &str, &c, &d)
    fmt.Printf("%d %f %s %t %c\n", a, b, str, c, d)

    //输入整型
    fmt.Scanf("%d", &a)
    fmt.Printf("整型:%d\n", a)
    //输入浮点型
    fmt.Scanf("%f", &b)
    fmt.Printf("浮点型:%f\n", b)
    //输入字符串
    fmt.Scanf("%s", &str)
    fmt.Printf("字符串:%s\n", str)
    //输入布尔类型
    fmt.Scanf("%t", &c)
    fmt.Printf("布尔:%t\n", c)
    //输入字符类型
    fmt.Scanf("%c", &d)
    fmt.Printf("字符:%c\n", d)
}
```

3.12.2 Scanln 函数

此函数是一行一行地读取数据,代码如下:

```
//unit3/scanln.go
package main

import "fmt"

func main() {
    var a int
    var b float64
    var str string
    var c bool

    fmt.Scanln(&a)
    fmt.Scanln(&b)
    fmt.Scanln(&str)
    fmt.Scanln(&c)
    fmt.Printf("%d %f %s %t", a, b, str, c)
}
```

3.12.3 Scan 函数

Scan 函数的用法与 Scanln 函数类似,可以写在一行,代码如下:

```
//unit3/scan.go
package main

import "fmt"

func main() {
    var a int
    var b float64
    var c bool
    var str string
    //使用 Scan 输入
    fmt.Scan(&a, &b, &c, &str)
    fmt.Printf("%d %.2f %t %s", a, b, c, str)
}
```