# 第一章 初识GO语言

每个Go代码文件都是以package开头的，表明该Go代码所属的包。Go执行的起点是包名为main的main()函数。

1. 从命令行传入参数时， args[0]可能是执行的文件名，因此要从args[1]算起。

2. 工程项目编译： 将工程目录放到GOPATH的src下，go build [package]

编译： go build [package name] (package 下直接放与package同名的main函数)

执行：在bin目录下 [package name].exe 参数

# 第二章 顺序编程

## 2.1 变量

### 1. 变量声明

Go语言引入了关键字var，类型信息放在变量名之后：

|  |
| --- |
| **var** v1 **string**  **var** v2 **[**10**]int** // 数组  **var** v3 **[]int** // 数组切片  **var** **(**  v6 **\*int**  v7 **map[string]** **int**  **)** |

### 2. 变量初始化

出现在 := 左侧的变量是没有被声明过的变量，否则会导致编译错误

|  |
| --- |
| **var** v1 **int** **=** 10 // 正确的使用方式1  **var** v2 **=** 10 // 正确的使用方式2，编译器可以自动推导出v2的类型  v3 **:=** 10 // 正确的使用方式3，编译器可以自动推导出v3的类型 |

### 3. 变量赋值

Go语言支持多重赋值语句： i**,** j **=** j**,** i

## 2.2 常量

### 1. 常量定义

通过const关键字定义常量，常量定义可以限定常量类型，但不是必须的，常量定义的右值也可以是在编译期运算的常量表达式

|  |
| --- |
| **const** Pi **float64** **=** 3**.**14159265358979323846  **const** zero **=** 0**.**0 // 无类型浮点常量  **const** **(**  size **int64** **=** 1024  eof **=** -1 // 无类型整型常量  **)**  **const** u**,** v **float32** **=** 0**,** 3 // u = 0.0, v = 3.0，常量的多重赋值  **const** mask **=** 1 **<<** 3 |

### 2. 预定义常量：true、false和iota

iota在每一个const关键字出现时都会被重置为0，在下一个const出现之前，没出现一次iota，其所代表的数字就会自动增1.

|  |
| --- |
| **const** **(** // iota被重设为0  c0 **=** **iota** // c0 == 0  c1 **=** **iota** // c1 == 1  c2 **=** **iota** // c2 == 2  **)**  // 上边常量定义可以简写为  **const** **(**  a **=** 1 **<<iota** // a == 1 (iota在每个const开头被重设为0)  b // b == 2  c // c == 4  **)** |

### 3. 枚举类型

枚举指一系列相关常量，Go不支持enum，可以通过const结合iota实现

|  |
| --- |
| **const** **(**  Sunday **=** **iota**  Monday  Tuesday  Wednesday  Thursday  Friday  Saturday  numberOfDays // 这个常量没有导出  **)** |

## 2.3 类型

1. 布尔类型不接受其他类型的赋值，不支持自动或强制的类型转换。两种不同的类型不能直接比较

2. int和int32在go中被认为是两种不同的类型，编译器不会自动转换，必须使用强制转换

|  |
| --- |
| **var** value2 **int32**  value1 **:=** 64 // value1将会被自动推导为int类型  value2 **=** **int32(**value1**)** // 编译错误 |

3. 字符串在初始化后不能修改，可以通过下标的方式获取。

4. 字符串的遍历

|  |
| --- |
| // 两种遍历方式对汉字类型编码处理方式不同  str **:=** "Hello,世界"  // 以Unicode方式遍历  **for** i**,** ch **:=** **range** str **{**  fmt**.**Println**(**i**,** ch**)**//ch的类型为rune，单个Unicode字符  **}**  // 以byte方式遍历  n **:=** **len(**str**)**  **for** i **:=** 0; i **<** n; i**++** **{**  ch **:=** str**[**i**]** // 依据下标取字符串中的字符，类型为byte  fmt**.**Println**(**i**,** ch**)**  **}** |

## 2.4 数组

### 1. 数组定义

数组是一系列同一类型的数据集合，常规的数组声明方法如下：

|  |
| --- |
| **[**32**]byte** // 长度为32的数组，每个元素为一个字节  **[**2**\***N**]** **struct** **{** x**,** y **int32** **}** // 复杂类型数组  **[**1000**]\*float64** // 指针数组  **[**3**][**5**]int** // 二维数组  **[**2**][**2**][**2**]float64** // 等同于[2]([2]([2]float64)) |

数组长度在定义后就不能改变，长度可以是常量或常量表达式，长度获取方式：

arrLength **:=** **len(**arr**)**

### 2. 元素访问

可以使用数组下标来访问数组中的元素，也可以使用range关键字来遍历数组

|  |
| --- |
| **for** i **:=** 0; i **<** **len(**array**)**; i**++** **{**  fmt**.**Println**(**"Element "**,** i**,** " of array is "**,** array**[**i**])**  **}**  // i是下标，v是值  **for** i**,** v **:=** **range** array **{**  fmt**.**Println**(**"Array element["**,** i**,** "]="**,** v**)**  **}** |

### 3. 数组是一个值类型，作为参数传递时会产生一次复制动作，因此在函数体中无法修改传入的数组的内容。

## 2.5. 数组切片

数组切片添加了一系列管理功能，可以随时动态扩充存放空间，并且可以被随意传递而不会导致所管理的元素呗重复复制。

作为参数时，数组是值传递，会被复制；数组切片是引用传递。

### 1. 创建数组切片

* 基于数组创建：数组切片可以使用数组的一部分元素或者整个数组来创建

|  |
| --- |
| **var** myArray **[**10**]** **int** **=** **[**10**]** **int** **{**1**,**2**,**3**,**4**,**5**,**6**,**7**,**8**,**9**,**10**}**  **var** mySlice **[]** **int** **=** myArray**[**1**:**5**]** |

* 直接创建

|  |
| --- |
| // 创建一个初始元素个数为5的数组切片，初始值为0  mySlice3 **:=** **make([]int,** 5**)**  // 创建一个初始元素个数为5的数组切片，元素初始值为0，并预留10个元素的存储空间  mySlice1 **:=** **make([]int,** 5**,** 10**)**  mySlice2 **:=** **[]int{**1**,**2**,**3**,**4**,**5**}** |

### 2. 数组切片的其他操作

|  |
| --- |
| // 创建一个初始元素个数为5的数组切片，初始值为0  mySlice3 **:=** **make([]int,** 5**)**  // 创建一个初始元素个数为5的数组切片，元素初始值为0，并预留10个元素的存储空间  mySlice1 **:=** **make([]int,** 5**,** 10**)**  mySlice2 **:=** **[]int{**1**,**2**,**3**,**4**,**5**}** |

copy可以将一个数组切片复制到另一个数组切片，如果两个数组切片大小不同，就会按其中较小的那个数组切片的元素个数进行复制

|  |
| --- |
| slice1 **:=** **[]int{**1**,** 2**,** 3**,** 4**,** 5**}**  slice2 **:=** **[]int{**5**,** 4**,** 3**}**  **copy(**slice2**,** slice1**)** // 只会复制slice1的前3个元素到slice2中  **copy(**slice1**,** slice2**)** // 只会复制slice2的3个元素到slice1的前3个位置 |

## 2.6 map类型

map的声明：

|  |
| --- |
| // 声明  **var** myMap **map** **[string]** **string**  // 创建  myMap **=** **make(map[string]** **string)**  myMap2 **=** **make(map[string]** **string,** 100**)** // 创建初始存储能力为100 |

map类型的增删改查

|  |
| --- |
| // 赋值  myMap**[**"1234"**]** **=** "123454"  // 删除  delete**(**myMap**,** "1234"**)** // 如果传入的myMap为nil，会抛出异常  // 元素查找  value**,** ok :**=** myMap**[**"1234"**]**  **if** ok **{** // 含有该元素  **}** |

## 2.7 流程控制

### 1. 条件语句 –- if else

条件语句需要注意：

* 条件语句不需要使用圆括号包含
* 无论条件体内有几条语句，都需要用{}括起来，且左花括号必须与if/else在同一行
* 在if之后，条件语句之前可以添加变量初始化语句，使用；分割
* 在有返回值的函数中，不允许将最终的return语句放在if或else中

### 2. 选择语句 –- switch

switch需要注意：

* 条件表达式不限制为常量或整数；
* 单个case中，可以出现多个结果选项
* 不需要使用break退出case
* 只有在case中使用fallthrough关键字，才会执行紧跟的下一个case；
* 可以不设定switch之后的条件表达式，同多个if... else...

|  |
| --- |
| **switch** i **{**  **case** 0**:**  fmt**.**Printf**(**"0"**)**  **case** 1**:**  **fallthrough**  **case** 2**,**3**,**4**:**  fmt**.**Printf**(**"2,3,4"**)**  **default:**  fmt**.**Printf**(**"default"**)**  **}**  **switch** **{**  **case** 0 **<=** Num **&&** Num **<=** 3**:**  fmt**.**Printf**(**"0-3"**)**  **case** 4 **<=** Num **&&** Num **<=** 6**:**  fmt**.**Printf**(**"4-6"**)**  **case** 7 **<=** Num **&&** Num **<=** 9**:**  fmt**.**Printf**(**"7-9"**)**  **}** |

### 3. 循环语句

GO语言中的循环语句只支持for，Go语言不支持以逗号为间隔的多个赋值语句

|  |
| --- |
| // 无线循环  sum **:=** 0  **for** **{**  sum**++**  **if** sum **>** 100 **{**  **break**  **}**  **}**  // 初始化多个变量时，必须使用平行赋值  a **:=** **[]int{**1**,** 2**,** 3**,** 4**,** 5**,** 6**}**  **for** i**,** j **:=** 0**,** **len(**a**)** – 1; i **<** j; i**,** j **=** i **+** 1**,** j – 1 **{**  a**[**i**],** a**[**j**]** **=** a**[**j**],** a**[**i**]**  **}** |

GO语言的for支持continue和break来控制循环，break可以选择中断哪一个循环

|  |
| --- |
| **for** j **:=** 0; j **<** 5; j**++** **{**  **for** i **:=** 0; i **<** 10; i**++** **{**  **if** i **>** 5 **{**  **break** JLoop  **}**  **}**  **}**  JLoop **:**  **...** |

## 2.8 GO函数

函数的基本组成为： 关键字func、函数名、参数列表、返回值、函数体和返回语句

小写字母开头的函数只在本包内可见，大写字母开头的函数才能被其他包使用

### 1. 不定参数

将函数定义为接受不定参数类型（相当于一个int类型的数组切片）：

|  |
| --- |
| **func** myFunc**(**args **...int)** **{**  **for** **\_,** arg **:=** **range** args **{**  fmt**.**Println**(**arg**)**  **}**  **}** |

传入任意类型的不定参数

|  |
| --- |
| **func** Printf**(**format **string,** args **...interface{})** **{**  // ...  **}**  **func** MyPrintf**(**args **...interface{})** **{**  **for** **\_,** arg **:=** **range** args **{**  **switch** arg**.(type)** **{**  **case** **int:**  fmt**.**Println**(**arg**,** "is an int"**)**  **case** **string:**  fmt**.**Println**(**arg**,** "is a string value."**)**  **default:**  fmt**.**Println**(**arg**,** "is an unkown type: "**,** arg**.(type))**  **}**  **}**  **}** |

### 2. 匿名函数

匿名函数有一个不带函数名的函数声明和函数体组成；匿名函数可以直接赋值给一个变量或者直接执行

|  |
| --- |
| f **:=** **func(**x**,** y **int)** **int** **{**  **return** x **+** y  **}**  **func(**ch **chan** **int)** **{**  ch **<-** ACK  **}** **(**reply**\_chan)** // 花括号后直接跟参数列表表示函数调用 |

### 3. 闭包

闭包是可以包含自由变量的代码块，这些变量不在这个代码块内或者任何全局上下文重定义，而是在定义代码块的环境中定义。要执行的代码块（由于自由变量包含在代码块中，所以这些自由变量以及他们引用的对象并没有释放）为自由变量提供丙丁的计算环境

只要闭包还在使用，那么闭包引用函数外的变量会一直存在

|  |
| --- |
| **func** main**()** **{**  **var** j **int** **=** 5  // 返回类型是函数  a **:=** **func()(func())** **{**  **var** i **int** **=** 10  **return** **func()** **{**  fmt**.**Printf**(**"i, j: %d, %d\n"**,** i**,** j**)**  **}**  **}()**  a**()**  j **\*=** 2  a**()**  **}** |

## 2.9 异常处理

### 1. error接口

### 2. defer：关闭资源

即使出现异常，defer也会保证资源被正常关闭，defer后也可以使用匿名函数。一个函数中可以存在多个defer，defer语句的调用时遵照先进后出的原则，最后一个defer先执行

### 3. panic 和 recover

当一个函数执行过程中调用panic()时，正常的函数执行流程立即终止，但函数中之前使用defer关键字延迟执行的语句将正常执行，之后逐层向上执行panic流程，知道所属的goroutine中所有正在执行的函数被终止。

recover函数用于终止错误处理流程。recover应该在一个使用defer关键字的函数中执行以有效截取错误处理流程。如果没有在发生异常的goroutine中明确调用恢复过程，会导致该goroutine所属的进行打印异常信息后直接退出。

# 第三章 面向对象编程

类型系统是指一个语言的类型体系结构，一个典型的类型系统通常包含如下基本内容：

* 基础类型，如byte、int、bool、float等
* 复合类型，如数组、结构体、指针等
* 可以指向任意对象的类型
* 值语义和引用语义
* 面向对象，即具有面向对象特征的类型
* 接口

## 3.1 GO中的类型系统

GO语言的大多数类型都是值语义的，并且都包含相应的操作方法。可以给任何类型增加方法；实现某个接口时，只需要实现该接口要求的所有方法即可。Any类型是空接口，即interface{}

### 1. 为类型添加方法

|  |
| --- |
| // 定义类型 Integer  **type** Integer **int**  // 为Integer定义方法，a作为this  **func** **(**a Integer**)** Less**(**b Integer**)** **bool** **{**  **return** a **<** b  **}**  // 如果要修改对象，需要用指针  **func** **(**a **\***Integer**)** Add**(**b Integer**)** **{**  **\***a **+=** b  **}** |

### 2. 值语义和引用语义

GO语言中的数组、结构体和指针等都是值类型。

表达引用要： var b = &a

GO中有四种引用类型：数组切片、map、channel、接口

### 3. 结构体

|  |
| --- |
| // 定义结构体  **type** Rect **struct** **{**  x**,** y **float64**  width**,** height **float64**  **}**  // 为结构体定义成员方法  **func** **(**r **\***Rect**)** Area**()** **float64** **{**  **return** r**.**width **\*** r**.**height  **}** |

## 3.2 初始化

GO语言中，未初始化的变量都会被初始化为该类型的零值。对象的创建通常以一个全局的创建函数来完成，以NewXXX命名

|  |
| --- |
| // 初始化的方式  rect1 **:=** **new(**Rect**)**  rect2 **:=** **&**Rect**{}**  rect3 **:=** **&**Rect**{**0**,**0**,**100**,** 200**}**  rect4 **:=** **&**Rect**{**width**:**100**,** height**:**100**}**  // "构造函数"  **func** NewRect**(**x**,** y**,** widht**,** height **float64)** **\***Rect **{**  **return** **&**Rect**{**x**,** y**,** widht**,** height**}**  **}** |

## 3.3 匿名组合

GO语言中的继承，采用了组合文法

|  |
| --- |
| **type** Base **struct** **{**  Name **string**  **}**  **func** **(**base **\***Base**)** Foo**(){}**  **func** **(**base **\***Base**)** Bar**(){}**  // 将Base中的方法组合到Foo中， foo.Foo()  **type** Foo **struct** **{**  Base  **...**  **}**  // 需要外部提供一个Base类实例的指针  **type** Foo **struct** **{**  **\***Base  **...**  **}**  // 覆写Base中的Bar方法  **func** **(**foo **\***Foo**)** Bar**()** **{**  foo**.**Base**.**Bar**()**  **}** |

组合的类型和被组合的类型都包含一个Name成员时，组合类型的Name成员的访问只会访问到最外层的Name变量

## 3.4 可见性

要使某个符号对其他包课件，需要将该符号定义为以大写字母开头。**Go语言中符号的可访问性是包一级的而不是类型一级的。**

## 3.5 接口

GO语言中的接口是非侵入式的，一个类只需要实现了接口所要求的所有函数，我们就是这个类实现了该接口。

对接口的赋值有两种情况：

* 将对象实例赋值给接口

将对象实例赋值给接口时，该对象实例必须实现了接口要求的所有方法。将对象实例赋值给接口，一般采用 var b LessAddr = &a

* 将一个接口赋值给另一个接口

如果接口A的方法列表是接口B的方法列表的子集，那么接口B可以赋值给接口A

## 3.6 接口查询

|  |
| --- |
| **var** file1 Writer **=** **...**  **if** file5**,** ok **:=** file1**.(**two**.**IStream**)**; ok **{**  **...**  **}** |

该if语句检查file接口指向的对象实例是否实现了two.IStream接口，如果实现了，则执行特定的代码

## 3.7 类型查询

|  |
| --- |
| **var** v1 **interface{}** **=** **...**  **switch** v **:=** v1**.(type)** **{**  **case** **int:** // 现在v的类型是int  **case** **string:** // 现在v的类型是string  **...**  **}** |