# 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  **...**  **}** |

# 第四章 并发编程

并发的主流的实现模型：

1. 多进程：多进程是在操作系统层面进行并发的基本模式，也是开销最大的模式。这种方法的好处在于简单、进程间互不影响，坏处在于系统开销大
2. 多线程：多线程属于系统层面的并发模式。他比多进程的开销小很多，但是开销依旧比较大，且在高并发模式下，效率会有影响。在高并发服务器开发的实践中，使用多线程模式会很快耗尽服务器的内存和CPU资源
3. 基于回调的非阻塞/异步IO：在Node.js中得到了很好的实践
4. 协程：协程可以通过yield来调用其他协程，通过yield方式转移执行权的协程之间不是调用者和被调用者的关系，而是彼此对称、平等的。协程本质上是用户态线程，真正的实现寄存于线程中，系统开销小。

GO语言目前并不知道多核并行化，需要设置其环境变量，通过runtime.GOMAXPROCS(runtime.NumCPU())来设置

## 4.1 goroutine

goroutine是GO语言中的轻量级线程实现，在函数调用之前加上“go”关键字，函数就会并发执行。如果这个函数有返回值，会被丢弃。

GO程序从初始化main package并执行main()开始，当main()函数返回时，程序退出，且程序不等待其他goroutine结束。

## 4.2 并发通信

在工程上，有两种最常见的并发通信模型：共享数据和消息。GO语言中是通过通信模型来传递消息的，其机制被称为channel。

channel是类型相关的，一个channel只能传递一种类型的值。

|  |
| --- |
| **func** Count**(**ch **chan** **int)** **{**  ch **<-** 1 // 向channel中写入一个数据  fmt**.**Println**(**"Counting"**)**  **}**  **func** main**()** **{**  chs **:=** **make([]chan** **int,** 10**)**  **for** i **:=** 0; i **<** 10; i**++** **{**  chs**[**i**]** **=** **make(chan** **int)**  **go** Count**(**chs**[**i**])**  **}**  **for** **\_,** ch **:=** **range** chs **{**  **<-**ch // 从chann中读取数据  **}**  **}** |

### 1. 基本语法

一般channel的声明形式如下：

**var** chanName **chan** ElementType

ElementType指定改channel所能传递的元素类型。可以使用make函数定义一个channel：

ch **:=** **make(chan** **int)**

将一个数据写入到channel: ch <- value; 向channel写入数据通常会导致程序阻塞，知道有其他goroutine从channel读取数据：value := <-ch

### 2. select

在Unix上，通过调用select()函数来监控一系列的文件句柄，一旦其中一个文件句柄发生了IO动作，该select()调用就会被返回。后来该机制也被用于实现高并发的Socket服务器程序。

select同switch非常类似，不过select有较多的限制，每个case语句里必须是一个IO操作。

|  |
| --- |
| **select** **{**  **case** **<-**chan1**:**  // 如果chan1成功读到数据，则进行该case处理语句  **case** chan2 **<-** 1**:**  // 如果成功向chan2写入数据，则进行该case处理语句  **default:**  // 如果上面都没有成功，则进入default处理流程  **}** |

### 3. 缓冲机制

创建一个带缓冲的channel，如下所示：

c **:=** **make(chan** **int,** 1024**)**

上面的例子创建了一个大小为1024的int类型的channel，即使没有读取方，写入方也可以一直写数据。

从缓冲的channel中读取数据使用与非缓冲channel相同，也可以使用range关键字来读取：

**for** i **:=** **range** c **{**

// handle i

**}**

### 4.超时机制

GO语言中没有直接提供超时的处理机制，可以用select实现超时处理方案

|  |
| --- |
| // 首先，实现并执行一个匿名的超时等待函数  timeout **:=** **make(chan** **bool,** 1**)**  **go** **func()** **{**  time**.**Sleep**(**1e9**)** // 等待1s  timeout **<-** **true**  **}**  **select{**  **case** **<-**ch**:**  // 从ch中读取到数据  **case** **<-**timout**:**  // 一直没有从ch中读取到数据，但从timeout中读取到了数据  **}** |

### 5. 单向channel

单向channel只能用于发送或者接收数据。单向channel是为了遵循最小权限原则

|  |
| --- |
| // 声明单向channel  **var** ch1 **chan** **int** // 正常的channel  **var** ch2 **chan** **<-float64** // ch2是单向channel，只用于写float64数据  **var** ch3 **<-chan** **int** // ch3 是单向channel，只用于读取int数据  // 初始化channel, 通过类型转换初始化了两个单向channel  ch4 **:=** **make(chan** **int)**  ch5 **:=** **<-chan** **int(**ch4**)** // ch5就是一个单向的读取channel  ch6 **:=** **chan<-** **int(**ch4**)** // ch6 是一个单向的写入channel |

### 6. 关闭channel

直接使用GO内置的close()函数关闭channel： close(ch)

可以通过 x, ok : <-ch 来检测channel是否关闭

## 4.2 出让时间片

可以使用runtime.GoSched()函数来让goroutine主动出让时间片给其他goroutine。

## 4.3 同步

GO语言包中的sync包提供了两种锁类型：sync.Mutex和sync.RWMutex.

|  |
| --- |
| **var** l sync**.**Mutex  **func** foo**()** **{**  l**.**Lock**()**  **defer** l**.**Unlock**()**  //...  **}** |

## 4.4 全局唯一性操作

对于全局只运行一次的代码，GO语言提供了一个Once类型来保证全局的唯一性操作。Once的Do()方法可以保证在全局范围内只调用指定的函数一次。

# 第五章 网络编程

GO语言标准库里提供的net包，支持基于IP层、TCP/UDP层及更高层的网络操作，其中IP层称为Raw Socket

## 5.1 Socket编程

使用net.Dial()建立各种各样的链接。

TCP链接： conn, err := net.Dial("tcp", "192.168.0.10:2100")

UDP链接：conn, err := net.Dial("udp", "192.168.0.12:975")

ICMP（协议名称）： conn, err := net.Dial("ip4:icmp", "www.baidu.com")

ICMP（使用协议编号）: conn, err := net.Dial("ip4:1", "10.0.0.3")

协议编号含义：<http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml>。

成功建立连接后，可以使用conn的Write()方法发送数据，使用Read()方法接收数据。

## 5.2 HTTP编程

使用GO语言提供的net/http包来编写HTTP客户端或服务端程序，内置的基本方法：

|  |
| --- |
| **func** **(**c **\***Client**)** Get**(**url **string)** **(**r **\***Response**,** err error**)**  **func** **(**c **\***Client**)** Post**(**url **string,** bodyType **string,** body io**.**Reader**)** **(**r **\***Response**,** err error**)**  **func** **(**c **\***Client**)** PostForm**(**url **string,** data url**.**Values**)** **(**r **\***Response**,** err error**)**  **func** **(**c **\***Client**)** Head**(**url **string)** **(**r **\***Response**,** err error**)**  **func** **(**c **\***Client**)** Do**(**req **\***Request**)** **(**resp **\***Response**,** err error**)** |

### 1. 使用 Get

|  |
| --- |
| **import** **(**  "fmt"  "io/ioutil"  "net/http"  **)**  **func** main**()** **{**  resp**,** err **:=** http**.**Get**(**"http://m.sogou.com/"**)**  **if** err **!=** **nil** **{**  fmt**.**Println**(**"err"**)**  **return**  **}**  **defer** resp**.**Body**.**Close**()**  by**,** **\_** **:=** ioutil**.**ReadAll**(**resp**.**Body**)**  fmt**.**Println**(string(**by**))**  **}** |

### 2. 使用Post

调用http.Post()方法，需要三个参数：URL，POST数据的资源类型，数据的byte流。

|  |
| --- |
| resp**,** err **:=** http**.**Post**(**"http://example.com/upload"**,** "image/jpeg"**,** **&**imageDataBuf**)** |

### 3. http.PostForm()

http.PostForm方法实现了标准编码格式为application/x-www-form-urlencoded的表单提交：

|  |
| --- |
| resp**,** err **:=** http**.**PostForm**(**"http://example.com/posts"**,** url**.**Values**{**"title"**:{**"article title"**},** "content"**:** **{**"article body"**}})** |

### 4. http.Head()

HTTP中的Head请求方式表明只请求目标URL的头部信息，HTTP不返回Body数据，只需要一个url参数

### 5. (\*http.Client).Do()

HTTP请求需要更多的定制信息时，比如设定自定义的Http Header字段。

|  |
| --- |
| // 含有自定义信息的请求  req**,** err **:=** http**.**NewRequest**(**"GET"**,** "http://example.com"**,** **nil)**  // ...  req**.**Header**.**Add**(**"User-Agent"**,** "Gobook Custom User-Agent"**)**  // ...  client **:=** **&**http**.**Client**{** //... }  resp**,** err **:=** client**.**Do**(**req**)** |

## 5.3 HTTP的高级封装

前面使用的方法都是在http.DefaultClient的基础上调用的。我们可以自定义http.Client

|  |
| --- |
| **type** Client **struct** **{**  // Transport用于确定HTTP请求的创建机制。  // 如果为空，将会使用DefaultTransport  Transport RoundTripper  // CheckRedirect定义重定向策略。  // 如果CheckRedirect不为空，客户端将在跟踪HTTP重定向前调用该函数。  // 两个参数req和via分别为即将发起的请求和已经发起的所有请求，最早的  // 已发起请求在最前面。  // 如果CheckRedirect返回错误，客户端将直接返回错误，不会再发起该请求。  // 如果CheckRedirect为空，Client将采用一种确认策略，将在10个连续  // 请求后终止  CheckRedirect **func(**req **\***Request**,** via **[]\***Request**)** error  // 如果Jar为空，Cookie将不会在请求中发送，并会  // 在响应中被忽略  Jar CookieJar  **}** |

CheckRedirect函数指定处理重定向的策略，HTTP Client会在遵循跳转规则之前先调用这个函数。

Jar用于在HTTP Client中设定Cookie，Jar的类型必须是实现了http.CookieJar接口，该接口预定义了SetCookies和Cookies两个方法，一般通过http.SetCookie()来设定Cookie

|  |
| --- |
| client **:=** **&**http**.**Client **{**  CheckRedirect **:** redirectPolicyFunc**,**  **}**  resp**,** err **:=** client**.**Get**(**"http://example.com"**)** |

在Client的结构体中http.Transport对象指定执行一个HTTP请求时的运行规则。

|  |
| --- |
| **type** Transport **struct** **{**  // Proxy指定用于针对特定请求返回代理的函数。  // 如果该函数返回一个非空的错误，请求将终止并返回该错误。  // 如果Proxy为空或者返回一个空的URL指针，将不使用代理  Proxy **func(\***Request**)** **(\***url**.**URL**,** error**)**  // Dial指定用于创建TCP连接的dail()函数。  // 如果Dial为空，将默认使用net.Dial()函数  Dial **func(**net**,** addr **string)** **(**c net**.**Conn**,** err error**)**  // TLSClientConfig指定用于tls.Client的TLS配置。  // 如果为空则使用默认配置  TLSClientConfig **\***tls**.**Config  DisableKeepAlives **bool**  DisableCompression **bool**  // 如果MaxIdleConnsPerHost为非零值，它用于控制每个host所需要  // 保持的最大空闲连接数。如果该值为空，则使用DefaultMaxIdleConnsPerHost  MaxIdleConnsPerHost **int**  // ...  **}** |

自定义http.Transport

|  |
| --- |
| tr **:=** **&**http**.**Transport**{**  TLSClientConfig**:** **&**tls**.**Config**{**RootCAs**:** pool**},**  DisableCompression**:** **true,**  **}**  client **:=** **&**http**.**Client**{**Transport**:** tr**}**  resp**,** err **:=** client**.**Get**(**"https://example.com"**)** |

## 5.4 HTTP服务端

使用net/http包提供的http.ListenAndServe()方法，可以在指定的地址进行监听，开启一个HTTP服务，方法原型如下：

**func** ListenAndServe(addr *string*, handler Handler) error

该方法用于在指定的TCP网络地址addr进行监听，然后调用服务端处理程序来处理传入的连接请求。第一个参数addr即监听地址；第二个参数表示服务端处理程序，通常为空，这意味着服务端调用http.DefaultServeMux进行处理，而服务端编写的业务逻辑处理程序http.Handle()或http.HandleFunc会默认注入到http.DefaultServeMux中

|  |
| --- |
| http**.**Handle**(**"/foo"**,** fooHandler**)**  http**.**HandleFunc**(**"/bar"**,** **func(**w http**.**ResponseWriter**,** r **\***http**.**Request**)** **{**  fmt**.**Fprintf**(**w**,** "Hello, %q"**,** html**.**EscapeString**(**r**.**URL**.**Path**))**  **})**  log**.**Fatal**(**http**.**ListenAndServe**(**":8080"**,** **nil))** |

如果想更多地控制服务端的行为，可以自定义http.Server:

|  |
| --- |
| s **:=** **&**http**.**Server**{**  Addr**:** ":8080"**,**  Handler**:** myHandler**,**  ReadTimeout**:** 10 **\*** time**.**Second**,**  WriteTimeout**:** 10 **\*** time**.**Second**,**  MaxHeaderBytes**:** 1 **<<** 20**,**  **}**  log**.**Fatal**(**s**.**ListenAndServe**())** |

还可以处理https请求，调用为：

**func** ListenAndServeTLS(addr *string*, certFile *string*, keyFile *string*, handler Handler) error

## 5.5 JSON处理

GO语言内置的encoding/json标准库使得GO程序可以轻松生成和解析JSON格式的数据。

### 1. 编码为JSON格式

使用json.Marshal()函数可以对一组数据进行JSON格式的编码：

**func** Marshal**(**v **interface{})** **([]byte,** error**)**

GO语言中除了channel、complex和函数类型除外，其他类型都可以转化为有效的JSON文本。

JSON转化前后的数据类型映射如下：

* 布尔类型转化为JSON后还是布尔类型
* 浮点数和整型转化为JSON里边的常规数字
* 字符串将以UTF-8编码转化输出为Unicode字符集的字符串
* 数组和切片会转化为JSON里边的数组，但[]byte类型的值会被转化为Base64编码后的字符串
* 结构体会转化为JSON对象，并且只有结构体里边以大写字母开头的可被导出的字段才会被转化输出，而这些可导出的字段会作为JSON对象的字符串索引
* 转化一个map类型的数据结构时，该数据的类型必须是map[string]T(T 可以是encoding/json包支持的任意数据类型

### 2. 解码JSON数据

使用json.Unmarshal()函数将JSON格式的文本解码为GO中的数据结构

**func** Unmarshal**(**data **[]byte,** v **interface{})** error

该函数的第一个参数是输入，即JSON格式的文本，第二个参数表示目标输出容器，用于存放解码后的值。

解码已知结构的JSON数据时，首先创建一个目标类型的实例对象，用于存放解码的值。json.Unmarshal()会根据一个约定的顺序查找目标结构中的字段，如果找到一个就发生匹配；如果JSON中的字段在GO目标类型中不存在，该字段将会被丢弃。

### 3. 解码未知结构的JSON数据

对于未知结构的JSON数据，可以将其解码输出到一个空接口。在解码JSON数据的过程中，JSON数据中的元素类型将做如下转换：

* JSON中的布尔值将会转换成bool类型
* 数值将会转换成float64类型
* 字符串转换后还是string类型
* JSON数组会转换成[]interface{}类型
* JSON对象会转换成map[string]interface{}类型
* null值会转换成nil

|  |
| --- |
| **var** r **interface{}**  err **=** json**.**Unmarshal**(**b**,** **&**r**)**  **if** err **!=** **nil** **{**  fmt**.**Println**(**err**.**Error**())**  **}**  unGobook**,** ok **:=** r**.(map[string]interface{})**  **if** ok **{**  **for** k**,** v **:=** **range** unGobook **{**  **switch** v2 **:=** v**.(type)** **{**  **case** **string:**  fmt**.**Println**(**k**,** "is string"**,** v2**)**  **case** **int:**  fmt**.**Println**(**k**,** "is int"**,** v2**)**  **case** **bool:**  fmt**.**Println**(**k**,** "is bool"**,** v2**)**  **case** **[]interface{}:**  fmt**.**Println**(**k**,** "is an array:"**)**  **for** i**,** iv **:=** **range** v2 **{**  fmt**.**Println**(**i**,** iv**)**  **}**  **default:**  fmt**.**Println**(**k**,** "is another type not handle"**)**  **}**  **}**  **}** |

### 4. JSON流式读写

encoding/json包提供了Decoder和Encoder两个类型，用于支持JSON数据的流式读写，并提供了两个函数来具体实现

|  |
| --- |
| **func** NewDecoder**(**r io**.**Reader**)** **\***Decoder  **func** NewEncoder**(**w io**.**Writer**)** **\***Encoder |

使用Decoder和Encoder对数据流进行处理还可以用来HTTP连接、WebSocket或文件等。

## 5.6 网站开发

GO提供了渲染网页的包html/template，可以让我们将HTML从业务逻辑中分离出来，详见代码。

# 第六章 安全相关

## 6.1 GO语言的Hash函数

GO提供了MD5、SHA-1等几种哈希哈数

func main() {  
 TestString := "Hi, pandam."  
 Md5Inst := md5.New()  
 Md5Inst.Write([]byte(TestString))  
 Result := Md5Inst.Sum([]byte(""))  
 fmt.Printf("%x\n\n", Result)  
  
 Sha1Inst := sha1.New()  
 Sha1Inst.Write([]byte(TestString))  
 Result = Sha1Inst.Sum([]byte(""))  
 fmt.Printf("%x\n\n", Result)  
  
 // 对文件计算hash  
 TestFile := "123.txt"  
 infile, inerr := os.Open(TestFile)  
 if inerr == nil {  
 md5h := md5.New()  
 io.Copy(md5h, infile)  
 fmt.Printf("%x %s\n",md5h.Sum([]byte("")), TestFile)  
 sha1h := sha1.New()  
 io.Copy(sha1h, infile)  
 fmt.Printf("%x %s\n",sha1h.Sum([]byte("")), TestFile)  
 } else {  
 fmt.Println(inerr)  
 os.Exit(1)  
 }  
}

# 第七章 学习总结

到此为止，GO语言的学习也告一段落了。总体来说，GO有其优势，也有其缺点。GO更改了异常的处理方式，抛出-捕获。每次都要根据err == nil 来判断是否有异常，让人比较恼火。好处是其执行速度快，接口灵活，不过要在定义type时，注明其实现的接口。