Go语言筑基

工具篇

**序**

第一次与大家见面，如果看到有问题的地方，欢迎指正。同时也欢迎联系我，或者留言评论，发布论坛。不让错误的东西存在时间太长。

写程序过一段时间后发现有些内容还是容易忘记。有时会记录一些，有时便会忘记记录，过段时间再遇到相同的问题会很难受。要是基础问题，那就是丢人，脸疼了。因此本书是针对基础的工具书。

工作过小几年的朋友应该都清楚，再次遇到面试官。他们依然是要求基础要过硬，其它的都好说。基础不够硬，会让人家很为难的。老艾也是经常去网上各种搜索，又是看各种版本的书，去扫除脑袋里问号。有很多书词汇过于专业，咬文嚼字的过于多。有的更是非程序人员翻译的国外书，那看起来酸爽。

其中《GO语言圣经》是翻译的不错的基础类书籍。专业词翻译的也都非常好。Go现在的好书越来越多，而且大多自己创作，可谓是匠心著作。希望我也能是你们中的一员吧。

本书本着由浅入深的思路。所以新手一般看章节的前半部分即可，其它的回头再看。本书从定义，常规，特性，性能，内存，运行时太，以及一些Go隐形问题，以及建议所构成。

**Go简书**

创始人[Robert Griesemer](http://research.google.com/pubs/author96.html)，[Rob Pike](http://genius.cat-v.org/rob-pike/" \t "http://books.studygolang.com/gopl-zh/_blank)和[Ken Thompson](http://genius.cat-v.org/ken-thompson/" \t "http://books.studygolang.com/gopl-zh/_blank)三位大牛。他们就职于Google公司，于2007年9月开始设计和实现，然后于2009年的11月对外正式发布。

GO是一个开源的编程语言。有时会被喜欢的它的人称“21世纪的C”。相比于C和C++它易于构建简单、可靠和高效的软件。

Go的编译速度很快。不用在这点担心它。

Go的可移植性很高。不用担心在什么平台下写的。

Go的文化“简单哲学”。

Go的生态已成熟，标准库和第三方的库越来越多。

Go具有CSP并行特性，非常适合服务端编程。

Go有类C的语句，但它的来源先祖不仅仅是C。

Go是纯粹的函数式语言。

Go有GC内存垃圾自动回收。GC运行微秒级。

Go的构建工具go tool越来越完整。

Go少了面向对象类语言很多部件，像宏，异常，继承，运算符重载，构造析构函数等等它都没有要求在语言里。

Go原生支持Unicode。可以处理任何字符。

我们从Go1.11+ 开始，过去的就不多说了。平台我们选择amd64类unix系统。

**本书结构**

第一章是为了对go有兴趣的人而作，即使你没有太多其它语言基础，也可以轻松的运行第一个程序，从而敲响golang的大门。

**第一个部分**

目录

[第一章 新手入门 8](#_Toc67396338)

[1.1 Hello, world 8](#_Toc67396339)

[1.2 \*unix环境 9](#_Toc67396340)

[1.2.1 官方安装地址 9](#_Toc67396341)

[1.2.2 安装步骤 10](#_Toc67396342)

[1.2.3 运行hello, world! 12](#_Toc67396343)

[1.3 命令行参数 13](#_Toc67396344)

[1.4 类C语言 15](#_Toc67396345)

[1.5 感悟 15](#_Toc67396346)

[第二章 程序结构基础 17](#_Toc67396347)

[2.1 关键字和内置词 17](#_Toc67396348)

[2.2 包与go源码文件·简 17](#_Toc67396349)

[2.3 命名 19](#_Toc67396350)

[2.4 声明 19](#_Toc67396351)

[2.5 变量 20](#_Toc67396352)

[2.5.1 简短变量 21](#_Toc67396353)

[2.5.2 指针 21](#_Toc67396354)

[2.5.3 new 22](#_Toc67396355)

[2.5.4 变量的生命周期 23](#_Toc67396356)

[2.6 赋值 24](#_Toc67396357)

[2.7 作用域 25](#_Toc67396358)

[第三章 基础数据类型 27](#_Toc67396359)

[3.1 整数 27](#_Toc67396360)

[3.2 浮点数 30](#_Toc67396361)

[3.3 布尔值 30](#_Toc67396362)

[3.4 字符串 31](#_Toc67396363)

[3.4.1 字符串与切片·简 32](#_Toc67396364)

[3.4.2 字符串连接 32](#_Toc67396365)

[3.5 常量 33](#_Toc67396366)

[3.6 复数 35](#_Toc67396367)

[3.7类型转换 36](#_Toc67396368)

[3.7.1 数值型转换 36](#_Toc67396369)

[3.7.2 数字+布尔与字符串 38](#_Toc67396370)

[3.7.3 字节与字符串·精 41](#_Toc67396371)

[3.8 自定义类型 45](#_Toc67396372)

[第四章 表达式 46](#_Toc67396373)

[4.1 运算符 46](#_Toc67396374)

[4.2 流程语句 47](#_Toc67396375)

[4.2.1 IF 47](#_Toc67396376)

[4.2.2 Switch 48](#_Toc67396377)

[4.2.3 For 49](#_Toc67396378)

[4.2.4 For range 50](#_Toc67396379)

[4.2.5 Break, Continue, Goto 52](#_Toc67396380)

[第五章 复合结构数据类型 55](#_Toc67396381)

[5.1 数组 55](#_Toc67396382)

[5.2 切片 56](#_Toc67396383)

[5.2.1 初次相识 57](#_Toc67396384)

[5.2.2 内置函数copy 58](#_Toc67396385)

[5.2.3 内置append函数 58](#_Toc67396386)

[5.2.4 切片的删除插入操作 59](#_Toc67396387)

[5.2.4 切片申请的内存 60](#_Toc67396388)

[5.3 Map 61](#_Toc67396389)

[5.3.1 初次相识 61](#_Toc67396390)

[5.3.2 利用map分发简化流程 62](#_Toc67396391)

[5.4 结构体 64](#_Toc67396392)

[5.4.1 声明与定义 64](#_Toc67396393)

[5.4.2 体验结构体 66](#_Toc67396394)

[5.4.3 结构体嵌入 67](#_Toc67396395)

[5.4.4 看看结构体内存 68](#_Toc67396396)

[5.4.5 结构体大小 70](#_Toc67396397)

[第六章 函数与方法 71](#_Toc67396398)

[6.1 函数语法 71](#_Toc67396399)

[6.2 函数可作为值 73](#_Toc67396400)

[6.3 递归 74](#_Toc67396401)

[6.4 错误处理 75](#_Toc67396402)

[6.5 匿名函数 77](#_Toc67396403)

[6.6 控制流函数 78](#_Toc67396404)

[6.6.1 panic 78](#_Toc67396405)

[6.6.2 defer 80](#_Toc67396406)

[6.6.3 recover 82](#_Toc67396407)

[6.7 方法定义 84](#_Toc67396408)

[6.8 方法接收器 86](#_Toc67396409)

[6.9 方法与嵌入结构体 88](#_Toc67396410)

[第七章 接口 93](#_Toc67396411)

[7.1 接口定义与实现 93](#_Toc67396412)

[7.1.1 源码Reader接口定义 94](#_Toc67396413)

[7.1.2 接口实现 95](#_Toc67396414)

[7.2 接口嵌入组合 96](#_Toc67396415)

[7.3 接口类型 97](#_Toc67396416)

[7.3.1 接口类型 98](#_Toc67396417)

[7.3.2 nil接口(空接口) 98](#_Toc67396418)

[7.3.3 nil指针的接口与nil接口不同·深 99](#_Toc67396419)

[7.3.4 万能类型 102](#_Toc67396420)

[7.4 断言 103](#_Toc67396421)

[7.4.1 断言规则 103](#_Toc67396422)

[7.4.2 断言类型x.(type) 104](#_Toc67396423)

[7.4.3 编译时检查接口实现 104](#_Toc67396424)

[7.5 error接口 105](#_Toc67396425)

[7.5.1 errors源码 106](#_Toc67396426)

[7.5.2 Errno 107](#_Toc67396427)

[7.6 flag包+flag.Value接口 108](#_Toc67396428)

[7.6.1方式一·flag.T() 109](#_Toc67396429)

[7.6.2 方式二·flag.TVar() 111](#_Toc67396430)

[7.6.3 方式三·flag.Var() 112](#_Toc67396431)

[第八章 序列化数据结构 115](#_Toc67396432)

[8.1 JSON 115](#_Toc67396433)

[8.1.1 序列化和反序列化DEMO 116](#_Toc67396434)

[8.1.2 带标签的结构体序列化 117](#_Toc67396435)

[8.1.3 自定义解析 118](#_Toc67396436)

[8.2 Protobuf简介 120](#_Toc67396437)

[8.2.1 安装protoc 120](#_Toc67396438)

[8.2.2 定义proto消息 122](#_Toc67396439)

[8.2.3 使用protobuf 123](#_Toc67396440)

[8.2.4 protobuf类型介绍 125](#_Toc67396441)

[8.2.5 import 127](#_Toc67396442)

[8.2.6 service定义服务 128](#_Toc67396443)

[8.2.7 protoc工具 128](#_Toc67396444)

[8.2.8 资料 128](#_Toc67396445)

[8.3 XML 129](#_Toc67396446)

[第九章 并发 131](#_Toc67396447)

[9.1 Go语言并发 131](#_Toc67396448)

[9.2 goroutine 131](#_Toc67396449)

[9.2.1 不一样的并发 131](#_Toc67396450)

[9.2.2 Go并发模型M,P,G 132](#_Toc67396451)

[9.2.3一个简单的并发 132](#_Toc67396452)

# 第一章 新手入门

## 1.1 Hello, world

永恒的“hello, world”。现在各种程序语言都喜欢用 hello, world作为敲门砖。我们也看看Go的代码。

**CODE 1-1** ：

/\*\* 注释

\* Hello world.

\* module:github.com/aixgl/ch01/code1.1 module 是项目初始化的名称

\* source: https://github.com/aixgl/gobook/tree/master/basic.magic/ch01/code1.1

\*/

package main

import "fmt"

func main() {

fmt.Println("Hello,world!")

}

**编译运行**

如果你还没有安装go环境可以只看输出，下一节我们讲述安装环境。这里对执行的命令有个概念即可。

* **go run：**

Go run 可以对一个或者多个go后缀的文件进行编译, 不会生成执行文件。测试单个文件代码时用着不错。

$ go run helloworld.go

*输出*：

$ Hello,world!

* **go build：**

Go build 编译项目为一个可运行文件，生成一个可执行文件。

$ go build helloworld.go

$ ./helloworld

*输出*：

$ Hello,world!

只运行下面的命令是编译整个项目。可执行文件名是初始化的项目名。它们的运行结果是一样的。

$go build

至于编译运行命令详细介绍可以查工具链一章。

**程序介绍**

接下来我们讨论下这段小程序。

程序的文件必须是以.go为后缀。

* **package main**

Go是用包组织的，类似java包或者其它语言的库或者模块概念。Package就是开头定义包main就是包名。这里的main有些特殊，而是go工程入口包，同时也不能被引用。package详细内容在第二章第二节中再进行讲述。

* **import**

第二行import “fmt” 是引入标准库的fmt包。Import紧挨着package，它们之间不能写go的其它语句。Go的其它程序只能写在它的后面。Import要精确，不需要包的不用导入否则会编译报错。

* **func main**

定义主函数main，所有的go的项目入口都是这个函数。可以带参数，可以不带。func 是定义函数的关键字。

函数内的fmt.Println是引用fmt包的Println函数是按行打印字符串，后面自动跟回车符，它可以接收多个参数，这样写fmt.Println(“hello,”, “world”) 也是可以的。

在go语句结尾分号“;”是不用写了。同行写多个语句必须用分号分割。

程序块一般是用{}一对花括弧为界，作用域于此也是息息相关。

## 1.2 \*unix环境

### 1.2.1 官方安装地址

<https://golang.org/dl/>

<https://golang.google.cn/dl/>

地址1 经常需要高科技上网，地址2正常能上网即可打开



图1-1

### 1.2.2 安装步骤

1 下载：

用wget命令下载，可以从官网地址上选择一个对应系统的版本，右键复制链接，跟在wget后面即可。下面示例的命令可直接用于运行。

例：$ wget https://golang.google.cn/dl/go1.15.5.linux-amd64.tar.gz



图1-2

2 安装到指定目录

源码安装到指定目录。下面例子安装目录为 /usr/local/go

例：$ tar -C /usr/local -xzf go1.15.5.linux-amd64.tar.gz



图1-3

3 添加环境变量

**GOROOT添加**

* 配置文件修改

$ vi /etc/profile

此命令打开的是全局配置文件。将光标移动到文件最末尾（快捷键：shift+g），添加下面的变量（换行插入的命令是o）。

export GOROOT=/usr/local/go

export PATH=$PATH:$GOROOT/bin

* 保存并退出

按： ESC键

输入： :wq

* 导入

$ source /etc/profile

此命令是导入变量以及环境变量

* 判断是否安装成功，输入命令显示如图2-4类似，则OK。

$ go version



图1-4

**GOPATH添加**

此变量一般添加在开发者用户下，比如多人在同一台服务器上作业。又有各自的工作账号以及工作目录。

1 添加开发者用户

如果有用户里此步骤可以省略

例：$ useradd boy

添加用户

$ passwd 123456

设置用户密码

$ su boy

切换到此用户下

2 配置GOPATH

$ vi ~/.bash\_profile

在export PATH 前添加如下内容

export GOPATH=$HOME/go

export PATH=$PATH:$GOPATH/bin

export GOPROXY=https://goproxy.cn #代理

export GO111MODULE=on

保存并退出

导入： $ source ~/.bash\_profile

测试： $ echo $GOPATH

### 1.2.3 运行hello, world!

在boy用户下创建一个go项目文件夹hello，在这个工程下添加一个hello.go 的文件夹。将第一章的“hello，world”代码粘贴到这个文件里。Boy用户你可以替换成你自己的用户。

执行前先确认是否在项目文件下。

$ pwd

输出：

/home/boy/hello

**go run**

$ go run hello.go

输出：

Hello，world!

**go build**

*初始化项目*

若是第一次使用它，先用go mod 初始化我们的项目。下面的命令常用格式，并不是严格要求格式必须这样。命令的斜线部分是可以自定义的。

$ go mod init *github.com/xxx/hello*

github.com: 远程仓库地址

xxx: 你在github仓库的用户名

hello: 项目名

*执行*

$ go build && ./hello

同样输出：

Hello，world!

还会生成一个 hello 的可执行文件下次再运行 可以直接是用  
 $ ./hello

*注*：

以下命令都是有效的

$ go mod init github.com/xxx/path\_a/path\_b/hello  
 $ go mod init hello

## 1.3 命令行参数

这一节我们再举个小示例来解释，加深下go程序的运行，顺便解决部分人脑袋中的问号。如：命令行参数是个什么东东？ go的命令行参数是怎么解析的？

**CODE 1-2** ：

package main

/\*\*

\* Command Args.

\* module:github.com/aixgl/ch01/code1.2

\* source: https://github.com/aixgl/gobook/tree/master/basic.magic/ch01/code1.2

\*/

import (

"fmt"

"os"

)

func main() {

for i, arg := range os.Args {

fmt.Println("arg", i, "=", arg)

}

}

**编译运行 ：**

Init: 未初始化的先初始化。以后我们会省略不再提醒此步骤。

$ go mod init  *github.com/aixgl/ch02/code1.2*

Build: 生成可执行文件

$ go build

$ ./code1.2 721 boy mycode

上命令的 721, boy, mycode就是可执行文件code1.2的三个命令行参数

输出：



图1-5

由此可以看出命令行的参数是通过os.Args传递go程序。

**源码解析：**

* Import

fmt 标准库包专职格式化打印，这里只用了Fprintln按行打印输出，每行自动追加换行符。此函数可以支持任意多个参数，通常称为变长参数函数。它参数类型支持的很多，一些结构性复合变量也可以直接传个它且能被打印出来。如：fmt.Println(os.Args[1:])。

os 标准库包系统包，这里只用了Args包变量，类型是切片（可以理解为动态数组，但不一样）。用os.Args[0:]可以完全代替os.Args 且结果是一样的。

* Main函数代码

for range 是go循环遍历的一种语法格式，每次循环将键和值分别赋值给变量i和arg，fmt.Println接收它们后执行输出。这2个变量只能在for语句程序员块使用，作用域外也没定义的话，在for大括号外使用会报编译错误。

:= 是一种简易赋值方式，这里不详细介绍，在后续章节会专门讲解。

字符串 是用英文双引号括起来。

**扩展：**

通过以上结果以及分析，我们看到os.Args将执行文件名放在了第一个元素里了，若是我们只要参数应该怎么办，我们稍微改动下程序即可实现。

伪代码：

// 直接改动下os.Args，利用切片的特性

for i, arg := range os.Args[1:] {

fmt.Println("arg", i, "=", arg)

}

带有背景颜色的部分就是修改的部分

## 1.4 类C语言

C语言是最早的高级语言。汇编语言通常称为B类语言。机器语言01我们可以叫它A类吧。

很多编程语言底层要么全部或者部分都是C或者C++语言写的，尤其是脚本语言，像php，python，以及嵌入式语言lua。所以它们的语法部分都类C

它的近亲比如说c++, java, c#。它们的语法同样类C。

Go早期版本部分底层也是C 加 汇编写的，在1.5版本以前。很多关键字，语法等同样保留C的，它也同样是类C语言。并被很多喜欢它的人称为新一代的C语言。伴随它的还有go汇编以及自举（自己编译自己）。

**CODE OF GO:**

//赋值

a = 2

//if语句

If a== true {

//code to do something.

}

//for语句

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

}

...

看到上面的code估计你同样会有原来如此，嗯...的感觉

**对比**：

1、**简易**。 相较于C它更简单，有垃圾回收（GC），相同的实现用更少的代码且更稳定可靠易于维护。

2、**并行和异步**。 Go 语言的 Goroutine 和 Channel 这两个可以堪称为神器。使用同样简单，就像用一个普通函数和变量且高性能，这是具有开创性的。

3、**Go语言生态完善迅速**。Go语言的标准库库中有绝大多数常用的库，足够开发使用。第三方库与框架更是发展迅速。

4、**强悍**。相比于java,python,c#等。保证高性能的同时，同样适合比较多的领域。工具类开发如docker等，web领域，游戏等。

## 1.5 感悟

程序语言怎么也比英语好学。英语你学了多少年，程序语言学一个月就可以去工作了。

流水线式思路编程。

化繁为简，合理的利用封装，外部可见的一定是像1+1一样。

程序是可以抄会的，只要想着弄明白，多运行调试。

耐心是最重要的，bug,问题是你进阶的一个利器。总是给别人填坑就呵呵了。

# 第二章 程序结构基础

## 2.1 关键字和内置词

**关键字**

25个

不能用于自定义名字

不能命名成变量， 自定义类型

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

**预定义**

37个以上

与关键字不同

可以定义中重新使用它们

内建常量: true false iota nil

内建类型: int int8 int16 int32 int64

uint uint8 uint16 uint32 uint64 uintptr

float32 float64 complex128 complex64

bool byte rune string error

内建函数: make len cap new append copy close delete

complex real imag

panic recover

**注**

自定义的变量，函数，类型等尽量不用关键字和内置词做名称。

## 2.2 包与go源码文件·简

这里我们对包做个简单介绍，有助于阅读，详细的介绍见十三章。

**包简述**

包系统目的都是为了简化大型程序的设计和维护工作，模块化，封装性，共享以及重用。

每个包对应一个目录。

定义的关键字：package。声明文件首行如下

package flag

每个包还通过控制包内名字的可见性和是否导出来实现封装特性。依据名字的首字母大小写导出，大写的外部可见，小写的不可见。

修改了源码文件，必须从头构建该包以及依赖包。

Go闪电编译速度。

导入使用import

import (

"fmt"

"github.com/go-sql-driver/mysql"

)

**Go源码文件**

后缀是go

文件名命名一般是小写，多个词用下划线连接（推荐习惯）。

例：

源码文件：study\_args.go

/\*\*

\* Command Args.

\* module:github.com/aixgl/ch01/code1.2

\* https://github.com/aixgl/gobook/tree/master/basic.magic/ch01/code1.2

\*/

package main

import (

"fmt"

"os"

)

func main() {

for i, arg := range os.Args {

fmt.Println("arg", i, "=", arg)

}

}

## 2.3 命名

Go命名也通C语言类似，主要包含

函数名

变量名

常量名

类型名

**命名规范**

名称必须以一个字母（Unicode字母）或者下划线开头。

名称其它位置可以是数字字母或者下划线。

区分大小写，Head 和 head 是不同的。

名称不能使用关键字命名，也不用预定义的内置词。

名称尽量简短，名字的长度没有逻辑限制，理论上长短皆可。推荐：名称的作用域比较大，和业务逻辑的相关名称，尽量使用有意义的英文单词连接。

官方推荐 **驼峰式** 命名。如escapePackage。

## 2.4 声明

**声明语句**

变量声明 var

常量声明 const

类型声明 type

函数声明 func

**例**：

package main

/\*\*

\* 命名.

\* module:github.com/aixgl/ch02/code2.1

\* source: https://github.com/aixgl/gobook/tree/master/basic.magic/ch02/code2.1

\*/

import "fmt"

type myInt int32 //定义类型

const PI = 3.14 //定义常量

func main() { //定义函数

var r = 5.0 //定义变量

var cArea = PI \* r \* r

fmt.Println("circle R:=%f, C:=%f", r, cArea)

}

Output:

circle R:=%f, C:=%f 5 78.5

## 2.5 变量

**语法通用格式**：

var 变量名字 类型 = 表达式

类型 和 =表达式 我们分为2个部分，可以省略任意其中之一。

var 变量名字 类型 √

var 变量名字 = 表达式 √

var 变量名字 ❌

**默认值为零值:**

简单类型

var s string 零值=”” 字符串

var s int/float 零值=0 整形/浮点型

var s bool 零值=false 布尔值

接口引用类型的零值nil(包括slice、指针、map、chan和函数)

var s []byte/[]string/… 零值=nil 切片

var s map[string]string 零值=nil map

var s \*int 零值=nil 指针

var s chan int 零值=nil chan

var f func() 零值=nil 函数变量

**同行多个变量声明:**

可同类型或者不同类型皆可

var i, j, k int // int, int, int

var b, f, s = true, 3.14, "one" // bool, float64, string

### 2.5.1 简短变量

快速声明且初始化局部变量

作用范围：函数内

符号“:=”。 同一个作用域同一个变量只能使用一次该符号。

通用表达式：

变量名字 := 表达式

我们来看看具体的例子：

一个整数变量

i := 100

一个浮点数

f := 100.0

一个切片

s := []int{1,2,5,2,1}

同一行多变量

i, j := 1, 10

=(赋值) 和 := 的区别很大

i, j = j, i // 交换i 和 j 的值

### 2.5.2 指针

定义：指针的值是另一个变量的地址。

指针可以直接读或更新对应变量的值，而不需要知道该变量的名字

通用定义格式

var 变量名字 \*类型 //类型：go中常用或者自定义的类型

变量名字 := &其它变量

与指针相关的2个符号 \*（指针符号）和 &（取地址符号）

\*在部分情况下是可省略的，声明部分时它肯定是不能省略的

取指针的值 用 \*变量名

取指针的地址 用 变量名

Go不支持指针的指针，\*\*变量名

Go指针不支持指针计算，也就是不支持指针进行地址计算，C语言可以。

任何类型的指针的零值都是nil

DEMO:

x := 1

p := &x

fmt.Println(p, x, \*p) //output: *0xc0000160a8* 1 1

\*p = 10

fmt.Println(p, x, \*p) // output: *0xc0000160a8* 10, 10

\*p++

fmt.Println(x, p, p==&x) //output: 11 11 true

p为指针 变量p保存的x的地址p==&x为true，直接打印p为一个16进制内存地址的值。\*p取的是x的值，所以\*p == x为true，且修改\*p 可以改变x的值。

### 2.5.3 new

表达式new(T)将创建一个T类型的匿名变量，初始化一个零值，且返回一个指针\*T。

ptr := new(int) // p, \*int 类型, 指向匿名的 int 变量

fmt.Println(\*ptr) // output: 0

\*ptr = 2 // 设置 int 匿名变量的值为 2

fmt.Println(\*ptr) // output: 2

new是一个预定义函数，可被重定义。

用new创建变量和普通变量声明语句方式创建变量没有什么区别，所以这个是很少用的。

每次调用new函数都是返回一个新的变量的地址。

p1 := new(int)

p2 := new(int)

fmt.Println(p1 == p2) // output:false

如果两个类型都是空的，也就是说类型的大小是0，例如struct{}和[0]int，有可能有相同的地址（依赖具体的语言实现）（译注：请谨慎使用大小为0的类型，因为如果类型的大小为0的话，可能导致Go语言的自动垃圾回收器有不同的行为，具体请查看runtime.SetFinalizer函数相关文档）

### 2.5.4 变量的生命周期

变量的生命周期指的是在程序运行期间变量有效存在的时间段。主要指变量运行的过程。

包一级声明的变量生命周期：整个程序的运行周期

math.Pi // 整个程序运行期间内都可以访问这个常量

局部变量的生命周期：动态的，每次从创建一个新变量的声明语句开始，直到该变量不再被引用为止，然后变量的存储空间可能被回收，非是一定如此。

**不可达** ： 自动垃圾收集器依据每个包级的变量和每个当前运行函数的每一个局部变量开始，通过指针或引用的访问路径遍历，是否可以找到该变量。如果不存在这样的访问路径，那么说明该变量是不可达的。

局部变量存储空间分配在栈上还是堆上是由编译器决定的，不是声明字决定的。也就是与var和new等无关。

**局部变量逃逸**看下面的例子

var glavar \*int

func fHeap() {

var x int

x = 1

glavar = &x

}

注：fHeap的局部变量就是分配在堆上，fHeap执行完依然被glavar引用。x依然是可达状态。这是要额外分配内存的。

未发生逃逸行为的例子

func local() {

p := new(int) // var p \*int

\*p = 1

}

注：\*p是local的局部变量未发生逃逸，编译器分配空间既可以在栈上也可以在堆上。选择在堆上扔需额外分配内存。

Go是有垃圾回收机制的。编写代码一般不用考虑内存。要求高性能时，对生命周期理解也很重要。什么时候需要长声明周期的，什么时候需要短生命周期，gc内存操作是否频繁等，对程序性能是有很大的影响的。

## 2.6 赋值

**基础赋值语法**

更新变量的值。

Go赋值的通用格式

变量名字 = 表达式

示例：

num = 1 // 普通变量赋值整数

c = 2 \* 2 // 变量赋值一个计算表达式

\*p = 3 // 指针变量赋值整数

简写赋值，部分二元运算符与赋值符号可以组成一个简写的形式，运算符和=不能有空格。

变量名字 运算符= 表达式

示例：

inc += 10 //

dec -= 10 //

mcl \*= 10 //

div /= 10 //

语句与表达式不同，注意区分。在这里语句是有左值而表达式没有。++ 和 – 都是语句。

inc++ //正确 等同inc = inc + 1 或 inc += 1

dec-- //正确 等同dec = dec – 1 或 dec -= 1

c = inc++ //错误 不能编译通过

初始化赋值

nums := []int{1,2,45,52,4} // 等价 nums[0] = 1, nums[1] = 2 …

**元组赋值**

允许同行修改多个变量的值，而且可以是不同的类型。

b1, b2, i, s = 1, 2, false, "hello" // 多变量多类型赋值

b1, b2 = b2, b1 // 交换2个变量的值

斐波纳契数列（Fibonacci）

func fib(n int) int {

x, y := 0, 1

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

x, y = y, x+y

}

return x

}

## 2.7 作用域

声明语句的作用域是指源代码中可以有效使用这个变量名字的范围。

**作用域和生命周期对比：**

作用域：强调运行时，什么情况，范围可以使用，什么情况不能用。

生命周期：程序运行时变量存在的有效时间段。强调时间范围内，可以使用。

**句法块**

由花括弧所包含的一系列语句“{}”。句法块内部声明的名字是无法被外部块访问的，越里层的优先级越高。

示例：

var gla int

func f () {//作用域f函数内

fx, fy = 1, 2

gla = 3

// 作用域if\_1

if fx == 1 {

fx := 5 // if\_1的fx 与函数的fx局部变量是不同地址的变量，同名而已

ffx := 6 // 此变量只能在这个作用域下使用

//fy是函数作用下的变量，可以正确使用，gla是全局变量也可以正确使用

fmt.Println(fx, fy, gla) // output: 5 2 3

fmt.Println(ffx) // output: 6

}

fmt.Println(ffx) // 编译不通过 未定义

// 作用域 if\_2

if fy == 2 {

ffx := 7 // if\_2下的ffx只能在这里使用

fmt.Println(fx) // output: 1 //if\_1的fx重定义并未影响fx

fx := 4 // if\_2下的fx 与函数的fx是不同的变量，地址和值都不相同与 if\_1下的fx也不同

gla := 33 // if\_2下的gla与全局的gla是不同的变量，地址和值都不相同

fmt.Println(fx, fy , gla) // output: 4 2 33

fmt.Println(ffx) // output: 7

}

fmt.Println(gla) // output:3

}

分析：

为了方便区分我们将f函数作用域称为f，将第一个if域称为if\_1，将第二个if的语法块称为if\_2。我们要明确一个关系，if\_1 和if\_2是同级的且是f的子域。

**gla** 首先是包内全局变量，可以在各个语法块内使用。在f函数作用内赋值3，有效范围包内，可见且是3。在if\_2语法块内重新声明gla则在if\_2声明它以后的*语法块内变量gla*与全局的是不同的，且不影响*全局gla*的值，且此语法块相当于隐藏了*全局的gla*。

**fx** 是f函数的局部变量，并初始化值1，在if\_1下重新定义且赋值，不会影响*f的fx*，if\_1下相当于隐藏了*f的fx*。fx在f和*if\_2 重定义前作用域*皆可见。

**fy** 是f函数的局部变量，并初始化2，if\_1和if\_2是它的2个子域，这3个作用域都可以见fy。

**ffx**是if\_1 和if\_2的作用域的局部变量，且他们互不影响。因这2个作用域是同级别的。不能被它们以外的作用访问。

# 第三章 基础数据类型

所有的数据都是由比特(bit)组成。但计算机一般操作的是固定大小的数，如整数、浮点数、比特数组、内存地址等。进一步将这些数组织在一起，就可表达更多的对象。

Go语言将数据类型分为四类：基础类型、复合类型、引用类型和接口类型。本章着重讲解基础类型。



## 3.1 整数

Go语言的数值类型包括几种不同大小的整数。每种数值类型都决定了对应的大小范围和是否支持正负符号。

**整数类型**

有符号 int8 int16 int32 int64

无符号 uint8 uint16 uint32 uint64

大小 8 16 32 64 bit(单位)

int等价int32

Unicode字符rune类型是和int32等价的类型。

byte类型一般用于强调数值是一个原始的数据而不是一个小的整数。等价int8。

uintptr没有指定具体的bit大小但是足以容纳指针。uintptr类型只有在底层编程时才需要，特别是Go语言和C语言函数库或操作系统接口相交互的地方。

**声明与定义**

通用声明定义语法

var 变量名 整数类型 // 声明

var 变量名 整数类型 = 表达式 // 声明且赋值,可以省略类型

变量名 := 表达式 // 简写声明且赋值

示例：

var i int = 1 // 声明i 且赋值 1

k := 2 // 简写赋值整数2

var a, b int = 1, 2 // 多远定义与赋值

// 这样写也可以

var (

c = 3

d = 4

)

**整数类型转换**

只要类型的名字不同计算的时候都需要进行显示的转换。通用格式

整数类型(整数表达式/浮点型表达式)

示例：

var a, b int = 1, 2 // 定义且赋值整数 int可省略

c := int(a) + b // 正确

c = a + b // 错误 类型不同

**取值范围**

int8 [-128 -> 127]

int16 [-32768 -> 32767]

int32 [-2,147,483,648 -> 2,147,483,647]

int64 [-9,223,372,036,854,775,808 -> 9,223,372,036,854,775,807]

其中有符号整数采用2的补码形式表示，也就是最高bit位用来表示符号位，一个n-bit的有符号数的值域是从$-2^{n-1}$到$2^{n-1}-1$。无符号整数的所有bit位都用于表示非负数，值域是0到$2^n-1$。例如，int8类型整数的值域是从-128到127，而uint8类型整数的值域是从0到255。

示例：溢出

var u uint8 = 255

fmt.Println(u, u+1, u\*u) // output:"255 0 1"

var i int8 = 127

fmt.Println(i, i+1, i\*i) // output:"127 -128 1"

**运算符**

以下是整数可使用的运算符以及优先级递减顺序排列

\* / % << >> & &^

+ - | ^

== != < <= > >=

&&

||

**浮点型转整数**

整数类型(浮点型表达式)

结果只保留整数位

示例：

f := 3.141 // a float64

i := int(f)

fmt.Println(f, i) // output: "3.141 3"

c = 2.99

fmt.Println(int(c)) // output: "1"

## 3.2 浮点数

Go语言提供了两种精度的浮点数，float32和float64。

**类型**

float32 float64

获取类型的取值范围可以math标准包。math.Max类型(可替换float32/float64/… 等)。

float32取值范围 1.4e-45 - 3.4e38，大约6个十进制数的精度，

float64取值范围 4.9e-324 - 1.8e308，约15个十进制数的精度，

**声明与定义**

与上一节整数的语法格式是一致的。

小数点前面或后面的数字都可能被省略（例如.101或1.）

f1 := .101 // 等价 f1 := 0.101

f2 := 1. // 等价 f2 := 1.0

float32的有效bit位只有23个，其它的bit位用于指数和符号；当整数大于23bit能表达的范围时，float32的表示将出现误差）：

var f float32 = 16777216 // 1 << 24

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

**数值转换**

浮点数和整形的各个类型转换其实都是一个通用的格式

类型(数值)

## 3.3 布尔值

**类型：** bool

仅有2种值 true 和 false 。

**计算**

经常与if和for语句结合。

==，<=，>= 等操作符产生的结果也是布尔值。

逻辑运算 &&（AND）和||（OR） 和 ！（非）。

不支持隐式转换

**布尔转整形**

手动转换的示例

func btoi(b bool) int {

if b {

return 1

}

return 0

}

## 3.4 字符串

Go的字符串是一个不可改变的字节序列。字符串可以包含任意的数据，包括byte值0，文本字符串通常被解释为采用UTF8编码的Unicode码点（rune）序列，内部⽤指针指向 UTF-8 字节数组。 数据转换请看本章3.7一节。

**类型**：string

**声明与定义**

var s string // 只声明一个字符串 默认值""

var s1 string = "abc" // 声明且定义

var s2 = "abc" // 省略类型

s3 := "abc" // 简短声明初始化

s4 := `abc` // 简短声明初始化(反引号)

**细节**

不可修改，是一个整体。

字符串零值是""。

字符串相当于一个字节的数组(切片)。访问其中某个字节。s[i]，0<=i<len(s)。

不能获取字符串的字节指针，&s[i]错误。

Go提供的len函数，获取的是字符串的字节数。

字符串连接符号“+”，支持“+=”。

s = s1 + s2 // output:abcabc

s += s1 // output:abcabcabc

字符串字面值“"”（可转义）和“`”（不转义可跨行）。

s3 := "abc" // "" 括起来的字符串

s4 := `abc` // `` (反引号)括的字符串

### 3.4.1 字符串与切片·简

字符串可以用切片访问获取部分字符串或者字节。在字符串中间单的说切片s[i:j]就是切分字符串的，以左闭右开的规则截取字符串，结果还是字符串。可以赋值给新的变量。如果不熟悉切片，请看切片的章节。

切片访问字节

s := "hello, world"

fmt.Println(len(s)) // "12"

fmt.Println(s[0]) // output:104 ('h'== s[0])

fmt.Println(s[7]) // output:119 ('w'== s[7])

超出字符串索引范围的字节将会导致panic异常

c := s[len(s)] // panic: index out of range

切片获取部分串

s[0:5]) == "hello" // output:true

s[:5]) == "hello" // output:true

s[7:]) == "word" // output:true

### 3.4.2 字符串连接

**第一种“+”**

“+”，支持“+=”

s = s1 + s2 // output:abcabc

s += s1 // output:abcabcabc

性能相对一般，比较常用。

写业务肯定够用的。

**第二种[]byte**

先将字符串转[]byte切片， bs := []byte(s)。

上面的bs是可以按字节修改的。

转回字符串，改后后用 s = string(bs)。

通常使用在框架，公用库函数等地方。

[]byte方式

s := "hello, world"

bs := []byte(s) //

bs[7] = 'b'

bs[8] = 'a'

bs[9] = 'b'

bs[10] = 'y'

bs[11] = 0 // 这里不能写''，编译不过， byte的零值是0

s = string(bs)

fmt.Println(s) // output:hello, baby

这是个简单而难看的示例程序

标准库的bytes.Buffer它的变种，与[]bytes同一个原理。

import bytes //先引入标准库

var b bytes.Buffer

b.WriteString(s[:5])

b.WriteString(",")

b.WriteString("golang")

fmt.Println(b.String()) // output: hello, golang

以上是理论程序，直接用是加快不了多少。性能提升主要是看内存申请与维护情况。

**第三种 fmt.Sprintf**

s1,s2,s3,s4 := "hello", ",", "world", "!"

// output:hello,world!

result := fmt.Sprintf("%s %s %s %s", s1,

s2, s3, s4)

## 3.5 常量

**声明与定义**

语法： const 变量 类型 = 表达式

类型可省略

编译期计算，而不是在运行期。

类型都是基础类型：boolean、string或数字。自定义类型也要能直接转成上述几个基础类型，也可以做常量的类型。

示例：简单的

const pi = 3.14159

示例：另外一种写法

const (

a = 1

b

c = 2

d

)

fmt.Println(a, b, c, d) // "1 1 2 2"

**iota**

使用iota定义的常量，使用前最好先行确认，避免产生不是预期的结果。

iota定义常量组从0 开始按⾏计数的⾃增枚举值

const (

zero = iota // 0

one // 1，通常省略后续⾏表达式。

two // 2

three // 3

four // 4

)

iota自动带入表达式

const (

\_ = iota // iota = 0

KB int64 = 1 << (10 \* iota) // iota = 1, KB == 1024

MB // 带入KB表达式 1 << (10 \* 2)，但 iota = 2

GB // 同上 GB == (2的30次方)

TB // 同上

)

iota按列独立，各自增长，一个cost可以对应多个iota。很少见

const (

A, A2 = iota, 2\*iota // 0, 0 表达式分别是n和 2n)

B, B2 // 1, 2

C, C2

)

iota直接从非首行开始

const (

c1 = -1 // -1

c2 = iota // 1

c3 // 2

)

**常量的类型**

若是我们没有给常量定义类型。那么常量的类型是什么呢？答：untyped + (类型)

Go是静态语言且定义类型的变量之间也不能隐式转换。但是没有定义类型的常量是支持隐式转换类型的。它既能有默认转换的类型也能根据语义去隐式转换。这能大大减少计算时类型转换操作。

var a int = 1

var b int32 = 3

a += 2

b += 2

以上程序是正确的。2既能被转成int计算也能转成int32去计算。

由此结论：常量定义是否加类型也要思考，不要任性而为。

## 3.6 复数

**类型**

complex64 complex128

精度 float32(6) float64(15)

**定义**

定义函数 complex（实部，虚部）

real 和 imag 可以取出复数的实部和虚部的数字。

例：var x complex128 = complex(1, 2) // 1+2i

y := 3 + 4i

fmt.Println(real(y), imag(y)) // output: 3 4

**第三方库**

math/cmplx包提供了复数处理的许多函数

## 3.7类型转换

类型转换单独提出来的原因，主要是基础类型以及自定义类型基础类型做计算的时候有需要。Go中只要类型名字不同，哪怕是同一类的都是int32的计算也是需要类型转换的。

### 3.7.1 数值型转换

整数类的和浮点型数值转换的情况是一样的。

**转换语法公式**

类型(数值表达式)

num32 := int32(num64) //

**注意事项**

类型长度大的转小。

浮点型转整形小数部分直接砍掉。

浮点型转浮点型精度损失。

类型转换是会损失性能的。高频表达式需注意。

**CODE3.1: int64 转int32**

/\*\* 注释

\* 数值转换.

\* module:github.com/aixgl/ch03

\* filename code3.1.go

\* source: https://github.com/aixgl/gobook/tree/master/basic.magic/ch03/code3.1

\*/

// 长度大转长度小

func int64toint32() {

fmt.Println("=======int64toint32========")

//转换 长度从大到小

// 超过32位，转换后是0值。

num64 := int64(math.Pow(2, 38))

num32 := int32(num64)

// output: num64:=274877906944, num32:=0

fmt.Printf("num64:=%d, num32:=%d\n", num64, num32)

// 少于32位

num64 = int64(1024)

num32 = int32(num64)

// output: num64:=1024, num32:=1024

fmt.Printf("num64:=%d, num32:=%d\n", num64, num32)

}

Num64的值超过2的32次方时，int32(num64)为0。

Num64小于2的32次方，int32(num64)与原值等价。是预期结果。

**CODE3.1: int32 转int**

// 同长度不同类型名 也要转成相同的

func int32toint() {

fmt.Println("=======int32toint========")

num := 32

num2 := int32(2)

/\* 编译报错：invalid operation: num \* num2 (mismatched types int and int32)\*/

// fmt.Println(num \* num2) // int 和 int32 的类型不匹配

// output:64 //可以正确执行

fmt.Println(num \* int(num2))

}

Go数值类型不同不能参与计算。

相同位数不同类型也不能计算。

**CODE3.1: float32 转int**

// 同长度不同类型名 也要转成相同的

func float32toint() {

fmt.Println("=======float32toint========")

f := 32.8

num := int(f)

//output: f:=32.8, num:=32

fmt.Printf("f:=%v, num:=%d\n", f,num)

}

浮点转整形，砍掉了小数部分

**CODE3.1: float64 转float32**

// 高精度转低精度小数

func float64tofloat32() {

fmt.Println("=======float64tofloat32========")

f := 3.14182324232

// output: f:=3.14182324232, float32(f):=3.1418233 // 砍掉多余精度小数

fmt.Printf("f:=%v, float32(f):=%v\n", f, float32(f))

}

直接砍掉多余精度小数部分

### 3.7.2 数字+布尔与字符串

数字与字符串之间的转换，主要使用标准库的strconv库。使用前先import引入库。此小结内容我们称（数字+布尔）= T。

import "strconv"

i,\_ := strconv.Atoi("3") // 将字符串3转成 int 3

s := strconv.Itoa(3) // 将数字转成string 3

**strconv**

字符串转int：Atoi()

int转字符串: Itoa()

ParseT类函数将string转换为T类型：

ParseBool(str string)(bool, error)

ParseFloat(str，int)(float64, error)

ParseInt(str string，进制 int，转换位数 int )(int64, error)

ParseUint(str string，进制 int，转换位数 int)(int64, error)。

可能会失败，用第二个返回值error 成功(nil)，失败(error!=nil)。

FormatT类函数将其它类型转string：

FormatBool(bool)

FormatFloat(f float64, fmt byte, prec, bitSize int)

FormatInt(int64, 进制)

FormatUint(int64, 进制)

AppendT类函数用于将T转换成字符串后append到一个slice中：AppendBool()、AppendFloat()、AppendInt()、AppendUint()。

还有其他一些基本用不上的函数，见官方手册：go doc strconv或者https://golang.org/pkg/strconv/。或者可以查看源码。

**字符串转T Parse类函数**

**CODE3.2: stringtoT**

/\*\* 注释

\* 数值转换.

\* module:github.com/aixgl/ch03

\* filename code3.2.go

\* https://github.com/aixgl/gobook/tree/master/basic.magic/ch03/code3.2

\*/

package main

import (

"fmt"

"strcov"

)

// 简单的获取类型方法

func getType1(v interface{}) string {

return fmt.Sprintf("%T", v)

}

// 字符串转数字和bool

func stringtoT() {

fmt.Println("=======stringtoT========")

//

b, err := strconv.ParseBool("true")

// 转成float64，

f, err := strconv.ParseFloat("3.1415", 64)

// 转成int64，将"50"按16进制转成10进制值

i, err := strconv.ParseInt("50", 16, 64)

// 转成uint64，将"1024"按10进制转成10进制值。

u, err := strconv.ParseUint("1024", 10, 64)

// output: ParseBool 'true' type[bool] value[true] err[<nil>]

fmt.Printf("ParseBool type[%v]value[%v]err[%v]\n", getType1(b), b, err)

// output: ParseFloat type[float64] value[3.1415] err[<nil>]

fmt.Printf("ParseFloat type[%v]value[%v]err[%v]\n",getType1(f),f, err)

// output: ParseInt type[int64] value[80] err[<nil>]

fmt.Printf("ParseInt type[%v]value[%v]err[%v]\n",getType1(i), i, err)

// output: ParseUint type[uint64] value[1024] err[<nil>]

fmt.Printf("ParseUint type[%v]value[%v]err[%v]\n", getType1(u), u, err)

}

ParseFloat()只返回收float64类型的浮点数。

ParseInt()和ParseUint()有3个参数：

func ParseInt(s string, base int, bitSize int) (i int64, err error)

func ParseUint(s string, base int, bitSize int) (uint64, error)

s这第一个参数不用多说

bitSize参数表示转换多少位的int/uint，有效值为0、8、16、32、64。当bitSize=0的时候，默认32转字符串中表示32位的整数。例如bitSize=8表示类型是int64，只转2的8次方减一以内的值。

base参数表示以什么进制的方式去解析给定的字符串，有效值为0、2-64。当base=0的时候，表示根据string的前缀来判断以什么进制去解析：0x开头的以16进制的方式去解析，0开头的以8进制方式去解析，其它的以10进制方式解析。

**T转字符串 Format类函数**

**CODE3.2: Ttostringto**

// 其它基础类型转string

func Ttostring() {

fmt.Println("=======Ttostring========")

s := strconv.FormatBool(true)

fmt.Printf("ParseBool value[%v]\n", s )

// float转字符串 //output:FormatFloat value[3.14159E+00]

s = strconv.FormatFloat(3.14159, 'E', -1, 64)

fmt.Printf("FormatFloat value[%v]\n", s )

// int转字符串 //output:FormatInt value[-79d]

s = strconv.FormatInt(-1949, 16)

fmt.Printf("FormatInt value[%v]\n", s )

// uint转字符串 //output: FormatUint value[152]

s = strconv.FormatUint(152, 10)

fmt.Printf("FormatUint value[%v]\n", s )

}

2个int转string的函数

func FormatInt(i int64, base int) string

func FormatUint(i uint64, base int) string

第二个参数base指定将第一个参数转换为多少进制，有效值为2<=base<=36。当指定的进制位大于10的时候，超出10的数值以a-z字母表示。例如16进制时，10-15的数字分别使用a-f表示，17进制时，10-16的数值分别使用a-g表示。

1个转浮点数到string的函数

func FormatFloat(f float64, fmt byte, prec, bitSize int) string

bitSize表示f的来源类型（32：float32、64：float64），会据此进行舍入。

fmt表示格式：'f'（-ddd.dddd）、'b'（-ddddp±ddd，指数为二进制）、'e'（-d.dddde±dd，十进制指数）、'E'（-d.ddddE±dd，十进制指数）、'g'（指数很大时用'e'格式，否则'f'格式）、'G'（指数很大时用'E'格式，否则'f'格式）。

prec控制精度（排除指数部分）：对'f'、'e'、'E'，它表示小数点后的数字个数；对'g'、'G'，它控制总的数字个数。如果prec 为-1，则代表使用最少数量的、但又必需的数字来表示f。

### 3.7.3 字节与字符串·精

我们先简单说下字节byte。它与其它语言的byte是一样的。也可以进行计算。本文说的字节主要是[]byte切片与字符串的关系。

**byte转string**

string(byte)

s := string('a')

单向的，反过来用byte(string)则编译报错

CODE3.3:头部

/\*\* 注释

\* 字符串与字节.

\* module:github.com/aixgl/ch03

\* filename code3.3.go

\* source: https://github.com/aixgl/gobook/tree/master/basic.magic/ch03/code3.3

\*/

package main

import (

"fmt"

)

CODE3.3：字节

// 字节相当于int8的整数

func byteCal() {

fmt.Println("=======byteCal========")

b1 := 'H'

b2 := b1 + 32

// output: byte 运算 b1[H] b2[h]

fmt.Printf("byte 运算 b1[%v] b2[%v]", string(b1), string(b2))

}

**string与[]byte**

[]byte(string)

String([]byte)

双向的

a := []byte(s)

s := string(a)

**CODE3.3：字符串与字节数组**

// 字符串与字节数组

func stringAndBytes() {

fmt.Println("=======stringAndBytes========")

s := "hello"

bs := []byte(s)

fmt.Printf("s => bs s[%v] bs[%v]\n", s, bs)

bs[0] -= 32

// bs[0]是切片的第一个byte因此可以计算

fmt.Printf("string(bs)[%v]\n", string(bs))

}

转换后的[]byte与字符串长度一样，都是按自己算的。

bs可以修改，且每个元素是ascii。

**CODE3.3：遍历字符串**

// 使用字符串为切片循环

func stringWithFor() {

fmt.Println("=======stringWithFor========")

s := "Hello世界"

// 按[]byte解析

for i := 0; i < len(s); i++ { // byte

fmt.Printf("%c,", s[i])

}

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

// 按[]rune解析 这个类型在go中是比较特殊的存在。

rs :=[]rune{}

for \_, r := range s { // rune

rs = append(rs, r)

fmt.Printf("%c,", r)

}

// output: string([]rune)[Hello-世界]

fmt.Printf("\nstring([]rune)[%v]\n", string(rs))

// output: length []rune[7] string[11]

fmt.Printf("length []rune[%v] string[%v]\n", len(rs), len(s))

}

运行结果:

H,e,l,l,o,,,ä,¸,,ç,,,------

H,e,l,l,o,,,世,界,

output: string([]rune) value[Hello,世界] len[8]

rune 代表一个字符。

byte 代表一个字节。

转换方面[]rune 与[]byte 是一样的。

[]byte写文件，流数据传输等，元素长度uint8。

[]rune是个uint32的切片。更方便处理unicode和utf8编码，以及编码转换。

byte 和 rune 之间可以转换，byte 转向 rune 时不会出错

但是 rune 转向 byte 时会出现问题：

如果 rune 表示的字符只占用一个字符，不超过 uint8 时不会出错；超过时直接转换编译无法通过，可以通过引用转换，但是会舍去超出的位，出现错误结果

**高性能转换[]byte和string**

这个函数也是从网上，看到的，高频函数使用时性能确实好。原因主要是使用底层指针减少内存op（内存申请）。

// StringToBytes converts string to byte slice without a memory allocation.

func StringToBytes(s string) (b []byte) {

sh := \*(\*reflect.StringHeader)(unsafe.Pointer(&s))

bh := (\*reflect.SliceHeader)(unsafe.Pointer(&b))

bh.Data, bh.Len, bh.Cap = sh.Data, sh.Len, sh.Len

return b

}

// BytesToString converts byte slice to string without a memory allocation.

func BytesToString(b []byte) string {

return \*(\*string)(unsafe.Pointer(&b))

}

看情况使用就好。

项目落地才是王道。

## 3.8 自定义类型

与C语言类似，关键字是type。我们可以自定义任意类型，包括基础类型，结构体，函数类型，接口(interface)等。

type sex byte

var man sex = 1

println(man) // output: 1

至少看起来还是so easy。

与var和const语法很接近，也可以用小括号定义多个成组。

// 组

type (

bigint int64

student struct{ // student是一个结构体类型

name string

age uint8

}

handle func(int) bool // 函数类型

)

不支持隐式转换，必须是显性转换。

num := 1024

var kb bigint = bigint(num) // 必须显式转换，除非是无类型常量。

var kb2 int64 = int64(kb)

# 第四章 表达式

## 4.1 运算符

**全部符号**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| + | += | & | &= | && | == | != | ( | ) |
| - | -= | | | |= | || | < | <= | [ | ] |
| \* | \*= | ^ | ^= | <- | > | >= | { | } |
| / | /= | << | <<= | ++ | = | := | , | ; |
| % | %= | >> | >>= | -- | ! | … | . | : |
|  |  | &^ | &^= |  |  |  |  |  |

**优先级**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 优先级 | 运算符 | | | | | | |
| 高 | \* | / | & | << | >> | & | &^ |
|  | + | - | | | ^ |  |  |  |
| == | != | < | <= | > | >= |  |
| <- |  |  |  |  |  |  |
| && |  |  |  |  |  |  |
| 低 | || |  |  |  |  |  |  |

**运算符注释**

|  |  |
| --- | --- |
| 运算符 | 描述 |
| 算数运算符 | |
| + | 相加 |
| - | 减法 |
| \* | 乘法 |
| / | 除法 |
| % | 求余数 |
| ++ | a++ 等价a = a+1;++a 编译报错 |
| -- | a-- 等价a = a-1；--a编译报错 |
| 关系运算符 | |
| == | 两个值比较，相等:true，否则:false |
| != | 两个值比较，不相等:true，否则:false |
| > | 左值大于右值为true，否则false |
| >= | 左值大于等于右值为true，否则false |
| < | 左值小于右值为true，否则false |
| <= | 左值小于等于右值为true，否则false |
| 逻辑运算符 | |
| && | 逻辑与(and)，两边值都为true，则为true，否则false |
| || | 逻辑或(or)，两边值任意一个为true，为true，否则false |
| ! | 逻辑非(not)，条件为true，则为false，否则true |
| 位运算符 | |
| & | 按位与，相同位置都为1结果是1 |
| | | 按位或，相同位置其中一个是1结果是1 |
| ^ | 按位异或，一个为0一个为1则为1，否则0 |
| << | 左移n位就是乘以2的n次方，低位补0 |
| >> | 右移n位就是除以2的n次方 |
| 赋值运算符 | |
| = | 赋值 |
| += | a=a+b的缩写 |
| -= | a=a-b的缩写 |
| \*= | a=a\*b的缩写 |
| /= | a=a/b的缩写 |
| %= | a=a%b的缩写 |
| <<= | 将左值左移动n位再赋值给自己 |
| >>= | 左值右移动n位再赋值给自己 |
| &= | a=a&b |
| |= | a=a|b |
| ^= | a=a^b |

## 4.2 流程语句

### 4.2.1 IF

条件语句可省略小括号

它左大括号必须在条件表达式尾部。

条件语句支持if的局部初始化，表达式再给条件表达式

Else if

没有简写三元运算 condition?caseA:caseB

a := 1;

if a++; a==2 { // 条件为true

fmt.Println(a) // output: 2

}

//// 编译报错 条件有没有小括号都不行

//if (a > 0)

//{

// fmt.Println("a>0", a)

//}

if a < 0 {

// do somethings

} else if a > 0 {

} else {

// do somethings

}

### 4.2.2 Switch

条件比较多的时候，可替换if else。

省略break。

Case的比较值可以用逗号分隔

直接进入下一个case，可用fallthrough。

a := 3

switch a {

case 1, 3 :

fmt.Printf("case 1 value[%v]\n", a)

case 2 :

fmt.Printf("case 2 value[%v]\n", a)

fallthrough

default:

fmt.Printf("default value[%v]\n", a)

}

Output:

当a=3

case 1 value[3]

当a=2

case 2 value[2]

default value[2]

switch后面的变量a是可以省略的，只要作用域是可见的就行。那么case需要的值是就只能是bool值了。

switch a := 2; { //这里仅仅是初始化值，依然是省略参数的

case a < 0 : //case 只能用bool类型的表达式

/\* case 2 : //编译器报错switch的case 2 语句类型匹配错误

invalid case 2 in switch (mismatched types int and

bool) \*/

fmt.Printf("case a < 0 value[%v]\n", a)

default:

fmt.Printf("default value[%v]\n", a)

}

// fmt.Println(a) //编译报错

Output:

case a < 0 value[2]

这里case不用bool会编译不过。

### 4.2.3 For

Go的循环语句只有for，没有while。不用担心，for的各种变化也是够你玩的了。

没有条件的循环

for { // 替代 for(;;) {} 类似while (true)

// do somethings

}

只传递一个条件语句的

n := 10

for n <10 { // 等同 while (n < 10) {}

// do somethings

n++

}

常规的for循环

s := "hello"

for i, n := 0, len(s); i < n; i++ { // 常⻅的 for 循环，⽀持初始化语句。

//do somethings

}

// 下面这种做法是不推荐的

//for i := 0; **i < len(s)**; i++ {

// //do somethings

//}

For到到左括号间的语句是被2个”;”分割成了3个部分，

第一部分（init）是初始化语句，仅执行一次，这是声明的变量只能在for的作用域有效，这部分是可以省略的。

第二部分（condition）是条件语句是多次执行的，所以i<n 最好不要换成 i < len(s)，这样len将被多次执行。

第三部分（post） 一般为赋值表达式，给控制变量i增量或减量。

这也进一步提示我们，理解编译器跑程序的过程是很重要的，把“我想的”往后排一排。

### 4.2.4 For range

**初次见面**

For range 结构是Go语言特有的一种的迭代结构，非常有用， for range 语法上类似于其它语言中的 foreach 语句。本质就是for的语法糖。

它可以遍历数组、切片、字符串、map 及通道（channel）等。

语法：

for key, value := range collection {

// do somethings

}

value 始终为集合中对应索引的值拷贝，只读性质，对它所做的任何修改都不会影响到集合中原有的值。

若collection是一个字符串，则value对应的是rune，每个 rune 字符和索引在 for range 循环中是一一对应的，它能够自动根据 UTF-8 规则识别 Unicode 编码的字符。

**for range 遍历的返回值有一定的规律：**

数组、切片、字符串返回索引和值。

map 返回键和值。

通道（channel）只返回通道内的值。

**遍历切片的示例**

func forRangeSlice() {

for key, value := range []int{1, 2, 3, 4} {

fmt.Printf("key:=%d value:=%d\n", key, value)

}

}

Output:

key:=0 value:=1

key:=1 value:=2

key:=2 value:=3

key:=3 value:=4

**遍历一个字符串**

func forRangeString() {

fmt.Println("==========forRangeString==========")

s := "hello世界"

for key, value := range s {

fmt.Printf("key:=%d value:=%c\n", key, value)

}

}

Output:

key:=0 value:=h

key:=1 value:=e

key:=2 value:=l

key:=3 value:=l

key:=4 value:=o

key:=5 value:=世

key:=8 value:=界

**省略的可以“\_”**

for \_, value := range s {

//do somethings

}

其中value也同样可以被“\_”符号替代。

**遍历过程中用value取不到collection的指针**

for-range 其实是语法糖，内部调用还是 for 循环，初始化会拷贝带遍历的列表（如 array，slice，map）。每次遍历的value地址是不变的，若用value去地址，最终只会拿到一个地址。但换个方式还是可以取到地址的。

func forRangeGetPointer() {

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

for i,\_ := range arr {

fmt.Println("address:", &arr[i], "value:=", arr[i])

}

}

Output:

address: 0xc0000ae000 value:= 1

address: 0xc0000ae008 value:= 2

address: 0xc0000ae010 value:= 3

**对特定的大数组重置效率高**

Go 对这种重置元素值为默认值的遍历是有优化的。

### 4.2.5 Break, Continue, Goto

这3个保留字有着一样的语法格式，都可以跟一个标签参数。

Break可用于for,switch,select。

Continue仅能用于for语句内。

**先看一个简单的示例**

func BreakAndContinue() {

for i:=0; i < 100; i++ {

if i > 88 {

break // 若是有定义标签 break 标签名

}

if i % 3 == 0 {

continue // 若是有定义标签 continue 标签名

}

}

}

这是一个简单的例子，但也是最推荐的用法，关于break和continue小艾是不推荐结尾加标签的，代码可读性会有一定的降低，同时也破坏流水线形式。

**Break + 标签**

跳转标签(label)必须放在循环语句for前面，并且在break label跳出循环不再执行for循环里的代码。且break+标签只能用于for循环

func breakAndTag () {

fmt.Println("==========breakAndTag==========")

FOR1:

for i:=0; i < 3; i++ {

fmt.Printf("FOR1 第%d次循环\n", i)

// FOR2:

for j:=0; j < 3; j++ {

fmt.Printf("FOR2 第%d次循环\n", i)

break FOR1 //

}

}

}

Output:

FOR1 第0次循环

FOR2 第0次循环

上面代码可以看出，for+标签可以跳出好几层的for语句。若是将FOR1都注释掉且换成for2那么 此时的break+label 与break无标签是一样的。现实情况for还可能与select等嵌套使用。除了用break+label 还可以用封装简化代码去解决。

**Continue + label**

跳转标签(label)必须放在循环语句for前面，跳出循环后则将继续执行lable后面的代码。

不推荐使用。

**Goto + label**

**语法**

goto label;

..

.

label: statement;

Goto+label使用起来随性的多，直接跳转到定义好的标签域下。要是写在函数内，则不能跳转到其它函数中。相比c语言还是有了一定限制。这也避免代码流程混乱问题。

**Goto退出多重循环示例**

func gotoAndLabel() {

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

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

if x == 6 {

// 退出所有循环且跳转到标签

goto DONE\_ONE

}

}

}

// do somethings

// 手动返回, 避免执行进入标签

return

// 标签

DONE\_ONE:

fmt.Println("done one")

}

Goto也可以跳过return

若是goto出现早于label，就相当于打断调到某下一环节。

若是goto出现晚于label，有些类似递归了。

# 第五章 复合结构数据类型

在第三章我们讨论过基础的数据类型，仅有基础类型，使用起来还是有不够方便的地方，如描述一个人的特性等，做个队列，堆栈等都要写很多的代码。Go在此基础上又底层提供了一些复合类型，方便我们更加简练，快速的，落地我们的项目。

四个类型：数组，slice，map，结构体。

本章源码地址：

https://github.com/aixgl/gobook/tree/master/basic.magic/ch05

## 5.1 数组

老玩家可以直接跳过这一节也没关系。

数组的长度是固定的。

近亲slice且是动态变长的。

它很少被使用。多数都会选择slice。

访问数组元素用下标，索引下标从0开始。最后一个元素用a[len(a)-1]。

**初始化**

默认初始化的是对应元素类型的0值。

初始化的长度可用3个点代替，由初始化的元素数决定。

// 初始化 和 赋值

var a1 [3]int

var a2 [3]int = [3]int{1, 2, 3}

// 长度是根据初始化的值来确定的

a3 := [...]int {1, 2, 4}

数组的长度也是类型的部分，[3]int和[5]int是2个不一样的类型。

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

a = [4]int{1, 2, 3, 4} // 编译报错

**数组访问和修改**

var a2 [3]int = [3]int{1, 2, 3}

// 长度是根据初始化的值来确定的

a3 := [...]int {1, 2, 4}

for i, v := range a2 {

a2[i] = v \* 2

fmt.Printf("a2 k:=%d, v:=%d, a2[i]:=%v\n", i, v, a2[i])

}

for i, n := 0, len(a3); i < n; i++ {

fmt.Printf("a3[%d]:=%d \n", i, a3[i])

}

## 5.2 切片

Slice类型一般用[]T表示，T代表了go的数据类型，也可以是自定义类型。它是一个引用类型的数据结构。

**切片构成**

指针：指向第一个slice元素对应的底层数组元素的地址。

长度：长度小于等于容量，len(slice变量)。

容量：一般是从底层数据slice的开始到结尾位置，c:=cap(slice变量)。

**声明与初始化**

以下四种方式都是ok的。

var a []int

var a1 = []int{}

var b = make([]int, len) // len == cap

var b2 = make([]int,len, cap) //

**特性**

切片的长度的动态可变的，可粗暴的理解是变长的数组。

下标起始规则与数组一样。

slice的底层确实引用一个数组对象。

每个元素的类型不一定相同。([]interface{})这个特殊后续会讲到。

slice操作s[i:j]，其中0 ≤ i≤ j≤ cap(s)。获取的新slice有j-i个元素，从i开始到j-1结束，i或j可省略，都省略s[:]相当于原切片。没有变动。

slice的第一个元素并不一定就是数组的第一个元素

从同一个slice底层数据生成的多个slice之间是共享底层的数据。

索引值超过cap(s)的上限将导致一个panic异常。

零值为nil，且没有底层数据。

判断空用 len(s) == 0

### 5.2.1 初次相识

用一个程序和虚拟内存模型来分析切片

s1 := []string{"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"}

s2 := s1[3:8]

s3 := s1[5:7]

s4 := s1[4:10] // 等价s1[4:]

底层数据模拟示意图

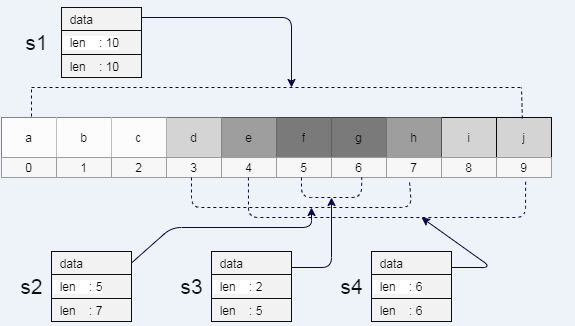


图5-1

上图中引用次数越多数据颜色块颜色越深，数据块下方是索引值。s1,s2,s3,s4是4个不同的切片，他们的长度和caps也各不相同的。底层都是互有折叠。因此这种新切片几乎没有内存op，slice无疑是一个高效的结构。

访问与遍历，支持for和for range，使用上与数组类似，通过索引即可访问与修改切片的元素。

简单遍历

s3 := s1[5:7]

for i, v := range s3 {

fmt.Printf("slice range k:%d, v:%s;\n", i, v)

}

Output: slice range k:0, v:f;

slice range k:1, v:g;

这是一个经典反转函数

func reverse(s1 []string) {

for i, j:=0, len(s1)-1; i<j; i, j = i+1, j-1 {

s1[i], s1[j] = s1[j], s1[i]

}

}

reverse(s1)

fmt.Println(s1)

output:

[j i h g f e d c b a]

### 5.2.2 内置函数copy

切片默认是引用类型，当需要不改原切片创建一个新的切片。需要用copy函数。

s3 := []int{10, 20, 30}

var s4 = make([]int, 3)

copy(s4, s3)

### 5.2.3 内置append函数

函数定义 func append(s []T, x …T) []T。追加或者删除元素且可以使用它。它是一个比较安全的操作切片的函数。Append是变长函数，可以有2个以上的参数。

但是有两点需要我们知道：

切片容量充足，在原切片基础上追加，无内存op。

切片容量不足，会分配新的切片空间，有内存op。性能相对低。

追加元素

var sRunes []rune

for \_, r := range "hello, world" {

sRunes = append(sRunes, r)

}

// ['h' 'e' 'l' 'l' 'o' ',' ' ' 'w' 'o' 'r' 'l' 'd']

fmt.Printf("append []rune%q\n", sRunes)

追加切片，需要切片参数追（slice…），用来讲切片的每个元素做参数值。

sRunes = append(sRunes, sRunes[0:5]...)

// append []rune['h' 'e' 'l' 'l' 'o' ',' ' ' 'w' 'o' 'r' 'l' 'd' 'h' 'e' 'l' 'l' 'o']

fmt.Printf("append []rune%q\n", sRunes)

### 5.2.4 切片的删除插入操作

通过一索引id删除元素

s2 := append(s1[:i], s1[i+1:]...) // 需要判断i小于len

删除索引 i-j 的元素

s2 := append(append([]int{}, s1[:i]...), s1[j:]...)

这个小示例是会产生内存op的，要是自己用for去操作可以避免的，是否值得就看你自己的了。

在索引 i 的位置插入元素

s2 := append(s1[:i], append([]int{2}, s1[i:]...)...)

在 s1 索引 i 的位置插入切片 s2

s1 = append(s1[:i], append(s2, s1[i:]...)...)

从以上几个例子中不难看出append的能力还是蛮强大的。这个函数也是蛮经典的。

Slice模拟栈

func sliceStack() {

fmt.Println("=======sliceStack======")

var sb = []int{1, 2, 4, 8, 16, 32}

// 入栈

sb = append(sb, 64)

fmt.Println(sb) //1 2 4 8 16 32 64]

// 出栈

k := sb[len(sb)-1]

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

fmt.Println(k, sb) //64 [1 2 4 8 16 32]

}

### 5.2.4 切片申请的内存

Go常规语法中没有明显的语句是，申请变量在栈还是在堆上。那么切片到底是申请内存是什么样的这是一个复杂的过程。我们在后续章节中会说，现在我们简单的知道一下有几个规则即可。能帮到我们基本判断出程序的语句与内存的使用情况。

一般初始化的全局变量分配在.data段(Section)内

一般未初始化的全局变量分配在.bss段(Section)内

局部变量在程序运行时分配内存地址。发生逃逸则在堆上申请

栈上操作比堆上还是快的多

大切片肯定是申请在堆上（总大小32kb,不是切片len）

局部变量切片，且确定没有被引用到外部，也没有return切片，在栈上

局部变量切片被其它函数作为参数直接使用也是在堆上申请。

查看是否逃逸或者分配在堆上的命令

// 结果中有 escapes to heap则为逃逸分配在堆上

go build -gcflags '-m -l' main.go

// 结果中有 runtime.newobject 则分配在堆上

go tool compile -S main.go

举个逃逸 escape to heap 的例子

func how() []int {

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

return s // 只要是指针就有可能逃逸

}

func main() {

\_ = how() // 如果没有接受不会发生逃逸

fmt.Println("escape testing")

}

即使发生了逃逸也不用担心，go编辑器都会处理，逻辑要求不高这些都也不用在意。

## 5.3 Map

可以简写成map[K]V，K是键，V是值。是我们常说的key/value结构型数据。操作简单，比切片容易的多。

**数据构成**

K: 索引map键，底层进行哈希计算

V: map的元素值。

Len: map长度

**声明与初始化**

scoreMap := make(map[string]int) // string和int可以换成其他类型

scoreMap := map[string]int{ // 声明且初始化键值

"a": 75, // 键：值

"b": 86,

}

**特性**

K有相同的类型。

V也有相同的类型，可以是聚合类型，结构体，函数等等。

K和V可以不同类型。

K必须是可比较的，也就是支持 == 运算。

查找复杂度O(1)。Map过大扔保持查询性能。

零值是nil。Map可以与nil做比较

尽量避免用浮点数做K。

Map长度计算也是O(1)时间复杂度，用len(map)就行了。

也通切片一样值for以及range。

删除用内置delete(map, K)。

无序。

Map之间不比较，没有== 操作。

该选用map还是slice，不看心情的话，可以根据数组的长度来决定，兼顾查询频度，修改频度，等所花费的时间。很多业务代码看心情决定也不会有任何问题。需要优化到语言上的时候，软妹币应该是拿到手软了。需要考虑这些一般都是框架，公共包的函数等。

### 5.3.1 初次相识

增

scoreMap["c"] = 90

fmt.Printf("%v\n", scoreMap) // output: map[a:75 b:86 c:90]

删

delete(scoreMap, "b")

fmt.Printf("%v\n", scoreMap) // output: map[a:75 c:90]

改

// 改

scoreMap["c"] = 92

scoreMap["a"] = 100

fmt.Printf("%v\n", scoreMap) // output: map[a:100 c:92]

查

// 查

fmt.Println("search:", scoreMap["a"], scoreMap["c"]) // output: search: 100 92

for k, v := range scoreMap {

fmt.Println(k,":=", v)

}

查，成功/失败

像上面那种霸道的直接复制就使用，并不多见。是否找到值，一般不会直接用值去判断，大多选择使用它返回的第二个返回值bool类型，true:值存在，false:值不存在。

// 安全查 ok:true/false

c, ok := scoreMap["c"]

if ok {

c += 25

}

### 5.3.2 利用map分发简化流程

有好几项任务用不同的函数在同一个请求中完成，除了用过多的if,或者switch来走不同的分支。我们还可以用map做一个简单粗暴的派发。当然这不是最好的选择。

import (

"bufio"

"errors"

"fmt"

"strings"

//"io"

"os"

)

func taskOne() {

// todo more

fmt.Println("To do one")

}

func taskTwo() {

fmt.Println("To do two")

}

func taskThree() {

fmt.Println("To do three")

}

func taskFour() {

fmt.Println("To do four")

}

// 利用map分发任务

func dispatch(choice string) error {

choice = strings.ToLower(choice)

tasks := map[string]func(){

"a": taskOne,

"b": taskTwo,

"c": taskThree,

"d": taskFour,

}

fn, ok := tasks[choice]

if !ok {

return errors.New("Not valid choice[" + choice + "]")

}

fn()

return nil

}

// 用终端输入命令行输入命令控制

func dialogWithStdin() {

fmt.Println("=======dialogWithStdin======")

in := bufio.NewReader(os.Stdin)

for {

r, \_, err := in.ReadRune() // returns rune, nbytes, error

if err != nil { // 跳出主goroutine

fmt.Println("===========\nWelcom use me!")

os.Exit(1)

}

dispatch(string([]rune{r}))

}

}

根据终端输入+回车测试。

这段代码的重点就是dispatch函数，尽量减少请求流程的分支。

其它的是辅助函数，使用了标准库的一些方法。

## 5.4 结构体

结构体是一种聚合类型。由其它类型的值组合的实体类型。

Go中用于描述实体主要用的结构体，描述人的个人信息，企业的组成，订单等。Go没有class。

### 5.4.1 声明与定义

首先定义一个结构，属于自定义一个类型。

type 结构体名 struct { // 结构体名是一个自定义类型

字段1 int // 定义一个结构体属性 没有逗号等符号

字段2 类型 //

…

}

**实例化，支持简短语法**

定义一个结构体

// 定义一个结构体

type User struct {

Uid int

Name string

Age int

Weight int

Height int

}

值类型初始化 3种情况

// 值类型初始化

func structInitValue() {

// 1 实例化

var u1 User // 声明User的实例

// 2 可以省略字段， 全部赋值值必须与结构体属性一一匹配

u2 := User{1, "joy", 20, 110, 170}

// 3 键值形式，可以部分赋值。

u3 := User{Name : "Game", Height: 169, Weight: 120}

fmt.Println("值类型初始化：", "\n", u1, "\n", u2, "\n",u3)

}

指针类结构体初始化

// 初始化成指针

func structInitPointer() {

fmt.Println("=======structInitPointer======")

// 1 只声明值== nil

var up1 \*User

// 2 new(T)一个有效的结构体指针 \*upNew的每项属性被初始化

var upNew \*User = new(User)

// 结构体前面加上"&"地址符号等同new

// 3 省略字段，全部赋值

up3 := &User{1, "p3", 20, 110, 170}

// 4 键值形式

up4 := &User{Uid: 2, Name:"p4", Height: 180}

fmt.Println(1, up1 == nil) // output:1 true

fmt.Println(2,up2, \*up2) // output:2 &{0 0 0 0} {0 0 0 0}

fmt.Println(3,up3, \*up3) // output:3 &{1 p3 20 110 170} {1 p3 20 110 170}

fmt.Println(4,up4, \*up4) // output:4 &{2 p4 0 0 180} {2 p4 0 0 180}

}

**特性**

结构体字段同行多字段定义，type T struct {x, y int}是合法的语法

结构体类型和字段的命名遵循可见性规则

结构体的字段可以是任何类型，甚至是结构体本身，也可以是函数或者接口

结构体指针\*可以使用时可以被省略

结构体支持嵌套

操作结构体字段用”.”，(结构体对象.字段)

结构体大小unsafe.Sizeof(结构体对象)，此值并不稳定，意义不大

字段默认值是各自类型的零值

较大的结构体通常使用指针传递

&User{1, …}写法可以直接在表达式中使用

若结构体所有成员是可比较的，那结构体也是可比较的，可用map的K

### 5.4.2 体验结构体

通过简单的玩家信息读取修改，我们看看结构体操作

func structGuild() {

p1 := User{Uid: 1, Name: "p1", Height: 180, Weight: 130}

p2 := User{Uid: 2, Name: "p2"}

// 用户1

// 改正用户p1的身高

p1.Height = 169

// 取值

fmt.Printf("p1[name:%v,height:%d,weight:%d]\n", p1.Name, p1.Height, p1.Weight)

// 用户2

p2.Height = p1.Height + 5

p2.Weight = p1.Weight - 10

fmt.Printf("p2[name:%v,height:%d,weight:%d]\n", p2.Name, p2.Height, p2.Weight)

}

Output:

p1[name:p1,height:169,weight:130]

p2[name:p2,height:174,weight:120]

**结构体是否可比较**

我们定义的User结构体的所有成员的只有int和string类型，因此结构体也是可比较的。

m1 := make(map[User]int) // 合法且ok

u1 == u2 // 同样是ok

这种应该不会有人喜欢用吧

**结构体可以定义的样式有很多**

type tree struct {

value int

left \*tree

right \*tree

}

这就是我们常见的二叉树结构体。

### 5.4.3 结构体嵌入

Go没有继承，嵌入则补充了这点，可以达到类似继承的效果。

可以直接访问嵌入结构体的字段。

匿名成员是嵌入时产生的。

嵌入是以指针的方式。

同一类型的只能嵌入一次，否则匿名成员名冲突。

匿名成员不仅仅限制于结构体用strins, int等基本类型也ok。

我们再定义一个**含匿名字段的结构体**

// 为嵌入User做准备的结构体

type Employee struct {

User // **嵌入User 结构体**

ManagerID int

Work int

}

**嵌入结构体注意初始化**格式。使用成员像使用它自己的一样。

func structExtend() {

fmt.Println("=======structExtend======")

e := Employee{

User:User {// 需要使用User结构体赋值

Uid: 2,

Name:"p4",

Height: 180,

},

Work:5,

}

// output: 2 p4 180 5 //使用成员像使用它自己的一样

fmt.Println(e.Uid, e.Name, e.Height, e.Work)

}

错误的初始化格式

//

e := Employee{Uid: 2, Name:"p4", Height: 180, Work: 5}

e.Height += 2 // 编译报错 cannot use promoted field

使用成员各自赋值

e := Employee{}

e.Height = 2

e.Uid = 1

e.Work = 2

这样也行

u := User {

Uid: 2,

Name:"p4",

Height: 180,

}

e := Employee{

User:u,

Work:5,

}

### 5.4.4 看看结构体内存

聚合类型内存变化，也没有想象的那么复杂。为了看起来清晰，我们重新定义一个简单的结构体。

// 一个点的坐标结构体

type Point struct {

X int

Y int

}

**先看普通值初始化**

p := Point{4, 5}

它的内存示意图

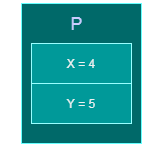


图5-4-1

**指针对象**

newPtr := new(Point)

refPtr := &Point{}

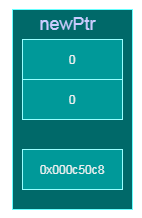
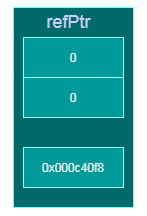
 

图5-4-2

p := Point{4, 5}

refPtr := &p // \*refPtr == p

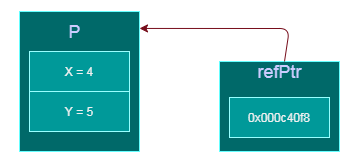


图5-4-3

上图中refPtr 的值存的p的地址。 \*refPtr根据refPtr存的内存地址取值，这个值也就是p。

### 5.4.5 结构体大小

结构体随是复合类型，但它里面的字段也是一维线性排列在内存里的，这是一种模糊的说法。即使有嵌套其它结构体或者复合类型也会一级一级的算下去。它也是可以计算大小的，**unsafe.Sizeof（结构体实例）**即可得到结构体的大小。

如果说结构体大小等于它的所有字段大小之和，多数情况下都不会是相等的，并不是绝对的错误。相等情况：所有字段的大小都相等；不等情况：存在长度不等的字段。

# 第六章 函数与方法

Go的函数也是把简短坚持到底了。连声明函数的关键字，都不用全名，它用func代替function。

相关源码：https://github.com/aixgl/gobook/tree/master/basic.magic/ch06

## 6.1 函数语法

函数由关键字func，函数名，形参，返回值定义以及函数体组成。函数体被大括号括起来，大括号也是函数作用域的分界线。

func 函数名(参数 类型, ...) (返回值 类型 ...) {

函数体

}

参数个数从0 到 n，函数可以无参数。

返回值类型也被定义，返回值变量可省略。

函数返回值用return 返回。

调用：函数名(参数…)

函数可以当类型来用

函数的零值是nil

支持递归

**无参，无返回值**

func a () {}

**带参数，带返回值**

// add(1, 2)

func add (x int, y int) int{

return x+y

}

// add2(2, 3)

func add2 (x int, y int) (c int) {

c = x + y

return

}

函数的参数类型要放在参数名后。

返回值只有一个的时候可以省略，括返回值类型的小括号。

若在函数头定义返回值变量(c int)，函数体内可以只写return即可。

以上2个函数是等价的，只是写法稍有不同而已。

**多参数，多返回值简写**

参数类型相同，或者返回值类型相同，以下2种写法也都ok。

func add3 (x , y int) ( c , d int) {

c = x + y

d = x \* y

return

}

func add4 (x , y int) ( int , int) {

c := x + y

d := x \* y

return c, d

}

**函数签名**

若是两个函数的形参列表和返回值列表，变量的类型一一对应，那么这两个函数是相同的类型或者签名。变量名是不会影响函数签名的，简写也不会影响签名。前面的add3和add4的签名是一样的。

**严格顺序和参数个数**

func stringAndInt5 (x string , y int) {

fmt.Println(x, y)

}

// stringAndInt5("next", 2) 这样使用是ok的

// stringAndInt5(2, "next") 编译错误

**函数变量**

var f func(int) int

f= func(x int) int {}

上面的写法是ok的。f零值是nil, 如果没有赋值就使用则会panic，崩掉程序(如果遇到recover处理程序并不会崩溃掉)。

## 6.2 函数可作为值

Go函数可看做一个值，函数名可以直接赋值给另外的变量。

先实现一个求幂值的函数。

// 实现一个求幂值的函数

func powMath (x, y int64) int64 {

if y == 0 {

return 1

}

r := x

for {

y--

if y == 0 {

break

}

r \*= x

}

return r

}

**函数名=值**用函数名可以直接赋值给其它的变量再去使用

f := powMath

fmt.Println(f(2, 2))

**函数名=实参**

func runeAdd(r rune) rune {

return r + 1

}

fmt.Println(strings.Map(runeAdd, "HAL-9001")) // IBM.:112

fmt.Println(strings.Map(runeAdd, "TSB")) // UTC

fmt.Println(strings.Map(runeAdd, "UUID")) // VVJE

这里稍微有点函数式编程的map函子的模型，这里基于包，别的语言一般用对象。

## 6.3 递归

函数体内，有调用本函数的语句为递归函数。

理论定义：对于某一函数f(x)，其定义域是集合A，那么若对于A集合中的某一个值X0，其函数值f(x0)由f(f(x0))决定，那么就称f(x)为递归函数。

斐波那契：递归

// 递归函数

func fib(n int) int {

if n ==0 || n == 1 {

return 1

}

return fib(n-1) + fib(n-2)

}

斐波那契：不用递归实现

func forFib(n int) int {

x, y := 0, 1

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

x, y = y, x+y

}

return x

}

同样实现的算法，第一种递归方式可以很容易造成栈溢出，第二种不会。

0,1,2,3,5,8,13 ... 数学函数n = f(n-1) + f(n -2)

阶乘：普通递归函数

func stepMul(n int, step int) int {

if n <= 0 {

return 1

}

if step == 0 {

step = 1

}

return n \* stepMul(n - step, step)

}

// stepMul(10, 1) 3628800

阶乘：goto递归

func gotoMul(n int) int {

r := n

WALK :

n--

if n <= 0 {

return r

}

r \*= n

goto WALK

}

// gotoMul (10) 3628800

这2个阶乘的函数运行的结果是一样的。算法也是一样的。

数学表达是 n\*(n-1)\*(n-2) ...2\*1

## 6.4 错误处理

程序处理过程中，总是有非法的情况出现。如果不处理就会发生panic。官方默认给与处理的包是errors

import "errors"

// EOF is the error returned by Read when no more input is available.

var EOF = errors.New("EOF")

func errFunc () error{

fmt.Println("=======errFunc======")

in := bufio.NewReader(os.Stdin)

for {

r, \_, err := in.ReadRune()

if err == io.EOF {

break // finished reading

}

if err != nil {

return errors.New("字符不是utf格式")

}

fmt.Println(r)

}

return nil

}

使用时读取错误信息，err== nil 则正常流程。

err := errFunc()

if err != nil {

// todo somethings

}

**自定义的错误码**

我们也可以不用errors，使用我们自己定义的错误码，用数字代表各个错误信息。

我们将上面的函数重写下

// 错误处理

type ERR\_CODE int // 我们也可以不用自定义类型

func errCodeFunc () ERR\_CODE {

fmt.Println("=======errFunc======")

in := bufio.NewReader(os.Stdin)

for {

r, \_, err := in.ReadRune()

if err == io.EOF {

break // finished reading

}

if err != nil {

return -10020

}

// ...use r…

}

return 0

}

**模拟try,catch**

使用recover和panic可以模拟try,catch语法，这种不推荐。

## 6.5 匿名函数

直接声明不带函数名的函数。也可以叫没有函数名的函数。

匿名函数的参数与返回值与带名称的函数一样。

func (参数列表) (返回值列表) {

函数体

}

**匿名函数与带名函数的区别**

带名字的函数只能在包一级作用域声明

匿名函数则不受此限制，你可以在包一级声明它，也可以在任何函数内声明或调用。

匿名函数可以直接在return后面声明且返回

这两种函数都可以当做值，赋值给变量，匿名函数在多次调用时一定要放在变量上然后去使用。避免一直生成不必要的匿名函数。

一个简单的匿名函数赋值给以一个变量上

sq := func (x int) int {

return x \* x

}

fmt.Println("anonymous sq1:=", sq(2)) // sq1:=4

在函数内定义的匿名函数，可以**引用该函数的变量**，有着完整的词法环境。如果再把这个**匿名函数传递**出去就有**闭包**的特性了。

// 一个求和加法器

func adder() func(int) int {

sum := 0// 试着修改sum的初始值为1

return func(n int) int {

sum += n

return sum

}

}

func anonymousEnv() {

fmt.Println("=======anonymousEnv======")

a := adder()

// adder的sum值变化， 对应的结果是不一样的。

// sum:=0 output:0 2 6

// sum:=1 output:1 3 7

fmt.Println(a(0), a(2), a(4))

}

匿名函数与普通函数都可以作为**实参**，也叫**回调**

net/http库的函数： http.HandleFunc("/",*func()*)

strings.Map(*func(r rune) rune { return r + 1 }*, "Hello")

## 6.6 控制流函数

在资源(数据库，缓存，文件，内存等)申请与使用过程中，可以安全的，快速高效的操作，很多其它语言是需要我们手动去做，一般还是在框架层面进行，避免与业务混淆产生不稳定因素。而go提供了辅助函数和关键字，让我们在这个过程安全，快速的完成这个过程。

实际项目中还有崩溃，异常等流程处理，go提供的内置函数，可以帮助我们完成这些。

### 6.6.1 panic

Go编译器编译时只能完成部分错误检查，而在运行时检查到的，将会引起异常(go自己调用panic函数)。

Go同样允许我们手动调用panic异常，就像调用一个函数一样。

panic(str string)

panic执行过程：当发生panic异常时，

1中断运行

2执行该groutine的defer 标志的函数

3是否有恢复函数

有：则恢复panic继续recover后执行

无：则退出程序，也称崩溃

Panic时go编辑器会调用堆栈跟踪信息。因此一般用于严重错误，不适合逻辑check。预料中的逻辑错误，分支处理，能用errors，或者自定义的错误码等更加。出现不可修复的错误，可以使用panic。

**用法**：语法上使用panic与调用一个函数是没什么区别

if err != nil {

panic("x is nil") // unnecessary!

}

**中断程序**

in := bufio.NewReader(os.Stdin)

for {

signal, \_, err := in.ReadRune()

if err != nil {

fmt.Println("input error")

}

switch signal {

case '9': // kill -9 的意思

panic("My program was held up!")

default :

fmt.Println("valid input")

}

}

这段程序根据标准输入的字符9触发panic中断程序

**信号量处理**

// 信号量处理

func signalHandle() {

for {

ch := make(chan os.Signal)

signal.Notify(ch, syscall.SIGINT, syscall.SIGUSR1, syscall.SIGUSR2,syscall.SIGHUP)

sig := <-ch

link.DEBUG("Signal received: %v", sig)

switch sig {

case syscall.SIGHUP:

link.DEBUG("get sighup\n")

case syscall.SIGINT:

os.Exit(1)

case syscall.SIGUSR1:

link.INFO("usr1\n")

case syscall.SIGUSR2:

link.INFO("usr2\n")

default:

link.INFO("get sig=%v\n",sig)

panic("we don't have the process")

}

}

}

注意区分panic和os.Exit:

Os.Exit立刻中断退出，直接清理延迟语句也就是清理堆栈，且可以返回错误码。不会运行延迟函数。

Panic中断，但会展开堆栈中的延迟函数，且执行它们，然后就是去走对出或者recover

### 6.6.2 defer

延迟执行命令。一般我们可以把它分成2个阶段：

1 注册调用阶段，但没有执行，将defer函数压栈

2 执行阶段，在return发生前一刻，或者panic发生前一刻。

Defer函数的执行顺序是，先进后出，栈结构。

语法：defer 函数(参数列表)

**资源型释放**

f, err := os.Open(filename)

if err != nil {

return err // 这里未注册过defer所以不会运行defer函数

}

defer f.Close() // 注册延迟

// do somthings // 早于defer函数执行

return nil // 先执行defer注册过的函数再返回

以上程序是打开文件，然后处理一些逻辑。仅仅使用defer函数便可有效，安全的读取文件。

若err不为nil执行return err 语句，因没有defer注册函数调用，不会执行f.Close()

若err为nil则会执行defer注册函数

**多个defer函数先进后出**

func deferStack() {

fmt.Println("=======deferStack======")

defer fmt.Println("defer 1")

defer fmt.Println("defer 2")

defer fmt.Println("defer 3")

}

Output:

defer 3

defer 2

defer 1

可以看到deferStack函数defer执行顺序正好与defer注册的顺序相反，也就是先入后出的顺序。

**Defer和return**

函数return语句后面可以跟随表达式，而defer是在这个表达式执行完事后才会执行。

Return语句并不是原子操作，对应汇编的ret命令，非原子命令，我们可以将它分为3个步骤。

1 return 接收参数

2 执行注册过的defer函数。

3 return 返回

主要区别是函数返回变量是否声明。未声明则defer语句无法改变返回值，已声明则defer有能力改变返回值，相当于修改了引用。

返回值不变的情况

func deferReturn() int {

i := 0

addi := func() {

i++

fmt.Println("defer", i)

}

defer addi()

defer addi()

// 此处return i 是值拷贝操作 将i理解为return函数的参数

// 等价于s:=i; return s;

return i

}

Output:

output:

defer 1

defer 2

return 0

分析：无声明的返回值函数return i 等价

1 s为return的直接操作的变量；将i拷贝到s中，也就是s := i

2然后执行defer操作的局变量i所以，defer的函数调用未能改变return的最终值s。

3 return s

返回值改变

func deferReturnRef()(i int) {

//i := 0 // 与deferReturn的区别

addi := func() {

i++

fmt.Println("defer", i)

}

defer addi()

defer addi()

// 此处return i 是值拷贝操作 将i理解为return函数的参数

// 可理解s := i; s++;s++; return s

return i

}

Output:

defer 1

defer 2

return 2

分析：返回变量提前声明了，return 等价

1 return 的操作变量已明确了，就是i

2 defer随后仍然操作i变量与return操作的是同一个变量。

3 defer执行完事，return立刻返回是已经修改的了的值。

### 6.6.3 recover

说起recover离不开panic，想让程序自救，不因预料中的panic而中断程序就要使用recover。但它也仅能处理部分的panic，针对go的错误处理情况如下。

1 常说的error或者自定义的错误机制（非panic），recover无效

2 系统级别错误，错误信息是fatal error 样子，如死锁，竞争错误等，recover也无效

3 模拟的try catch，用panic去throw recover去catch，非常不适合业务中去这样用，建议程序需要自救等情况去使用。

4 recover仅能回复本协程（groutine,也可理解多线程编程）内的panic，对于其它线程的panic是无能为力，未捕获的panic会中断崩溃程序。

**Recover示例**

func recoverPanic() {

// **do-one 包含一些在defer前的处理逻辑简称do-one**

defer func () {

if err := recover(); err == nil {

return

}

// do somethings

}()

// **do-two**

n := 0

n—

**// 其它函数调用语句 也可以有panic,且它们也能被恢复**

if n == -1 {

panic("n must bigger than -1")

}

// **do-return** // 无返回值定义，可将尾花括号看做return

// 我们称此处的return为**同级return**

}

以上是一个有效简单的例子，recover是放在了defer语句块中，panic发生时，它相当于return，执行defer链。由此可见recover与panic也是遵循一定规则的。

**总结recover规则**

1 recover与panic必须是同一个协程（goroutine，后续章节会讲到）

2 recover需要放在defer语句块内，若是直接写recover，没有defer则是被立即执行了，没有defer链，panic发生时也不会被捕捉到。

3 包含recover的函数或匿名临时函数必须在panic前面用defer注册且不被真正的执行，仅仅是注册。

4 可监控的panic一定是在defer-recover真正执行前发生，也就是注册defer的函数同级return之间发生的。从上程序可以看到Do-one处发生panic是无法被捕捉到的，仅仅是在**do-two到do-return之间的panic**才能被捕捉到，在这之间（two-return）调用的其它函数panic也是可以被捕捉的。

其中规则3和4可以理解为可以被捕捉的panic有效范围。所以设计使用panic还是要遵守这几项规则，否则recover无法按照思维完成工作。

## 6.7 方法定义

Go中将函数定义在类型上，这种函数叫方法。类型也主要指的**自定义类型**，接口除外，即使是接口也是与方法有着千丝万缕的联系。所以我们也可以将类型说成是type关键字定义的类型。

内置的类型不可以定义方法，但它的自定义别名可以。

接口不可以定义方法

也有其它说法，定义在**接受者(**或叫**接收器)**的函数叫方法。这个接受者也是指上面说的自定义类型。

Go中没有class抽象定义，而是用类型表示。它同样看成是抽象类型，同样可以实例化生成对象。

Go也不支持重载。

Go方法重点是定义在**结构体**类型上的。

**用示例说明**：

type NewString string

func (s NewString) Print() {

fmt.Println(s)

}

Print就是定义在NewString的方法， NewString是基于内置string定义的新的自定义类型。

func **(s NewString)** 黑体的小括号以及里面的内容都是必须的。比普通函数多了这个接收器，其余的与函数一致。

**调用示例**

1 “实例对象.方法名(方法参数列表)” 比较通用的调用方式

2 普通函数调用方式((类型/类型指针).方法名(实例对象，方法参数列表))

var s NewString = "new,beijing,shanghai,tianjin,NewYork "

s.Print() // 第一种调用方式

(NewString).Print(s2) // 第二种调用方式

**错误定义**

func (s string) Print() {

fmt.Println("string print", s)

}

// 编译报错 cannot define new methods on non-local type string

**切分字符串的示例**

func (s NewString) Split(sep rune) []string{

rtnSlice := []string{} // 返回的字符串切片定义

beforeKey := 0 // 切分起始key

sSrc := string(s) // 转换成string类型为了使用字符串切片特性

for i, r := range s {

if r == sep {

rtnSlice = append(rtnSlice, sSrc[beforeKey:i])

beforeKey = i + 1

}

}

if beforeKey >= len(s) {

rtnSlice = append(rtnSlice, "")

} else {

rtnSlice = append(rtnSlice, sSrc[beforeKey:])

}

return rtnSlice

}

我们就可以这样使用了

var s NewString = "nmgs,beijing,shanghai,"

ss := s.Split(',')

这里是用字符切割简易版，仅供学习使用。标准库strings包提供了strings.Split(s, sep string)。用字符串切割更好用。

综上所述

方法是可以用函数的方式来使用，函数确不可用方法的方式来使用。

结构体是定义方法的主要类型。后续小艾尽量用结构体定义方法。

## 6.8 方法接收器

接收器分为值和指针。函数与方法的值传递是拷贝行为，没有修改能力。指针传递是引用，可以修改。

方法可见性，与变量名一致，开头大写，包内外皆可用，小写只能包内使用。

**定义一个结构体**

type Person struct {

name string

sex byte

age int

uid int64

}

**值接收器**

接收器是值类型，非指针类型，权限：只能读取接受器。

name变量名开头字母用的是小写，我们定义一个包内外皆可以访问的方法

// 值方法

func (pv Person) ValGetName() string{

return pv.name

}

func (pv Person) ValSetName(name string) error{

pv.name = name

return nil

}

调用

返回Person的姓名

即使是值类型方法也可以用指针去访问其方法

pv := Person {name:"xiao li"}

pv.ValSetName("da ming") // 修改pv.name失败

// 用pv去访问是值，用(&pv)是指针

// output: xiao li xiao li

fmt.Println(pv.ValGetName(), (&pv).ValGetName())

**指针接收器**

接收器是指针类型，权限：可读可修改，且有引用的对象则同样可修改。

接收器访问成员可以**简写**，与值访问写法一样，go编辑器可以自动识别。

// 指针方法

func (ptr \*Person) PtrGetName() string {

return ptr.name // 等价(\*ptr).name // 错误\*ptr.name

}

func (ptr \*Person) PtrSetName(name string) error{

ptr.name = name

return nil

}

调用

ptr := &pv

ptr.PtrSetName("qiao feng") // 修改成功

// ouput: qiaofeng qiaofeng

fmt.Println(pv.ValGetName(), ptr.PtrGetName())

指针的方法将ptr.name成功进行了修改操作，因ptr引用的pv所以同样修改了pv.name的值。

ptr.name 等价(\*ptr).name

ptr.name 不等价 \*ptr.name

\*ptr.name 等价 \*(ptr.name)

**方法执行时**

/\* (ptr \*Person) PtrSetName(name string) error

\* 方法运行时可以看成定义的这个函数

\*/

func PtrSetName(ptr \*Person, name string) error {

ptr.name = name

return nil

}

PtrSetName在运行时，可以看成函数PtrSetName(ptr \*Person, name string)，但是不能反过来看待。它们2个并不等价，尤其是在以后用接口断言（详细在接口章节）等，对Person的影响很是巨大。

// SetName 与 上面函数PtrSetName是等价的

SetName := ptr.PtrSetName

SetName(ptr, "xiao gang ")

// func Callback(fn func (\*Person, string) error)

Callback(SetName)

将方法当做值，或者实参传递时，注意第一个参数是接收器对象。

**总结**：接收器与方法

1 方法(method)无论定义在指针/值类型，访问此方法时，既可以用值也可以用指针去调用，编译器会隐士转换。

2 值接收器调用方法时会产生一次对象拷贝。指针接收器调用方法时产生的引用，都指向同一个接收器的内存地址。

3 通过2不难看出，指针接收器定义的方法有修改接收器的能力，值定义的则只有访问能力。

## 6.9 方法与嵌入结构体

嵌入式结构体在5.4.3章节中我们有讲解，这里我们先写一个嵌入式结构体，然后说下它的特性。

type Teacher struct {

Person

School string

office int

}

在Teacher中嵌入Person结构体

**特性**

可见性：方法在结构体中可见性，与可调用性都与成员字段是一样的。也可叫权限。

继承：嵌入式可以理解为继承， Teacher中可以访问Person中可见成员以及成员方法(同包可见所有成员)。

无关多态：其它语言继承与多态有着紧密的联系，Go多态是用接口interface实现，可以说与嵌入式无直接关联。

组合：Go结构体可以嵌入多个其它的结构体，看似结构体可以随意组合，这一点比面向对象的继承是灵活的多。

覆盖：被嵌入的Person结构体，并没有能力直接修改Teacher结构体成员，若Teacher结构体重写方法，那么直接就是覆盖了。

**Teacher的方法**

func (t \*Teacher) GetSchool() string {

return t.School

}

func (t \*Teacher) SetSchool(name string) {

t.School = name

}

**调用示例**

同包的继承行，被组合的结构体Person全部成员都可见

t := &Teacher {

Person:Person {

name : "DuanYu",

age : 15,

},

School: "清华",

}

// 继承行，可见性 同包嵌入Person结构体的age也可以被访问

fmt.Println("Name :=", t.PtrGetName(), "Age :=", t.age)

// t即可访问Teacher的成员也可以访问Person的成员

t.PtrSetName("XuZhu")

// output: School:= 清华 Name := XuZhu

fmt.Println("School:=", t.GetSchool(), "Name :=", t.PtrGetName())

Teacher的实例t即可访问Person的方法也可以访问Teacher的方法，因为同包所以t同样有权限访问Person的私有成员。

**示例-覆盖**

我们先分别为Person和Teacher定义一个同名的方法PrintInfo

func (ptr \*Person) PrintInfo() {

fmt.Println("I am", ptr.name, "and", ptr.age, "year old.")

}

// 覆盖重写

func (ptr \*Teacher) PrintInfo() {

fmt.Println("My school is", ptr.School)

}

t.PrintInfo() //output:My school is 清华

依据输出我们可以看到，Teacher的实例t.PrintInfo() 执行的在Teacher接收器上定义的方法。

**Go结构体无关多态**

我们再次确认下关系Person，作为Teacher的父结构体，再与面向对象语言区别分析。这需要从2个方面说起：

1. 父子对象转换。面向对象语言：以父对象作为类型可直接传递或者使用操作子对象的技术，统一使用父对象成员，或者强制将父对象转子对象，转换成功后可以使用子对象独有的特性成员等。Go：结构体并不能做到这点，使用接口可以实现这一点。Go父结构体与子结构体是2个不同的完全不同的类型且不能直接转换。
2. 继承重写。面向对象语言：一个方法流程可以定义在父对象中，子对象可以不用再重复定义这个流程，仅需要重写部分方法即可，甚至是不做任何事情，子对象再使用这个流程方法时，子对象的重写都是生效的。Go: 做不到这点。

我们看段代码，先在Person中定义一个流程

// 检验多态，继承特性

func (ptr \*Person) CheckPolymorphism() {

ptr.PrintInfo()

}

再使用它

t := &Teacher{}

t.name = "Pobi"

t.age = 15

t.CheckPolymorphism()

output: I am Pobi and 15 year old.

若是面向对象语言此时应该使用的是Teacher（子）重写的PrintInfo()方法。而go使用的是Person（父）的PrintInfo()方法。

**组合+可见性**

简单的理解为，多个结构体合成一个结构体，这里的组合仅仅说的结构体组合。

子目录github.com/aixgl/gobook/tree/master/basic.magic/ch06/order

/\*\*

\* 结构体：组合

\* 包内所有成员字段和方法都是可见的

\* 包外首字母大写的成员字段和方法才是可见的

\*/

package order

type User struct{

Uid int64

Name string

Phone int64

}

type Order struct {

OrderSn int64

productID int64 // 私有属性 包外不可见

}

/\*\*-------------------------------------------

\* User

\*\*-------------------------------------------\*/

func (u \*User) GetName() string {

return u.Name

}

func (u \*User) SetName(name string) {

u.Name = name

}

// 私有方法，包外不可见

func (u \*User) save() error{

return nil

}

/\*\*-------------------------------------------

\* Order

\*\*-------------------------------------------\*/

func (o \*Order)GetOrderSn() int64 {

return o.OrderSn

}

func (o \*Order)SetOrderSn(sn int64) {

o.OrderSn = sn

}

func (o \*Order)GetProductID() int64 {

return o.productID

}

func (o \*Order)SetProductID(pid int64) error {

o.productID = pid

return nil

}

我们先在子包中定义2个结构体以及方法，去探索下结构体组合，和可见性。

我们引用它在其它的包中

import (

"fmt"

"github.com/aixgl/ch06/order"

)

定义组合结构体

type OrderInfo struct {

order.User

order.Order

}

上面是有效的，一个结构体可以由任意个不同结构体组合而成，也可以再定义任意个字段。

探索这结构体

oi := &OrderInfo{}

oi.Uid = 1

oi.SetName("god")

oi.SetOrderSn(10232000232)

// oi.productID = 20 // 编译错误，产品id首字母小写包外是不可见，

oi.SetProductID("20")

fmt.Println("CombineStruct", oi)

// oi.save() // 编译报错，方法同样包外首字母小写不可见

组合而成的OrderInfo结构体可以直接使用包中order.User和order.Order两个结构体的对外可见的成员字段和方法。当它使用不可见的字段会编译报错。这是属于结构体的基本特性。

# 第七章 接口

接口是其它类型行为的抽象与约束。这是一个很新颖且独特的设计。

**语法**

type Shape interface {

Test() error

...

}

**接口细节**

形式上，它是一组方法声明（函数签名）的集合，没有方法的定义实现。它只有方法，没有成员字段。

**解耦性**，其它类型的方法全部实现了此接口相应的方法签名，那么该类型就实现了该接口。它更像是一个**合约**没有直接的语法约束实体类型，签订完事就行了，与面向对象语言implements有着明显的不同。

某个类型可以实现多个接口。

接口可当go语言的**动态类型**理解。它本身是静态类型，值是可变的动态的其它类型。由此可推论**interface{} 是泛型**，可代表任意类型，因为它无具体方法约束，所以任意类型都算实现了这个接口。

接口实现Go的多态特性。

接口定义不易过多，暴露的接口越简单越好。若一个结构体或其它类型不准备用其它类型替代或者重定义，则没有设计接口会更好。

接口是一个虚拟的动态类型与实体类型转换使用**断言**。

接口也支持嵌入组合，这点很像结构体嵌入特性。

空接口值（**不能叫零值**）为nil。

上面说的**实体类型**指的就是非接口类型。

## 7.1 接口定义与实现

这里我们直接用标准库io的接口来说明，这是几个使用广泛且值得我们学习的接口。

源：github.com/aixgl/gobook/tree/master/basic.magic/ch07

引用标准库io包

import io

### 7.1.1 源码Reader接口定义

目测与struct的定义很是相似，结构体用struct关键字，接口用interface关键字。结构体可以有实体类型成员字段，接口则不能。结构体的方法是定义和实现，接口的方法仅仅是声明不能具体实现，也可以说不能有花括号包裹的函数实体。

通常接口类型命名用er作为结尾，也有的人在开头用大写I开头。

接口函数签名可省略形参，通常未省略写法主要用它表现更多的语义。

定义

type Reader interface {

Read(p []byte) (n int, err error)

// Read( []byte) (int, error) //等价上面写法,可省略形参

}

分析：Reader接口只有一个Read方法，Read的唯一形参[]byte切片引用类型，将读取的内容读取到此形参上；它有2个返回值，第一个表示当前读取的长度；第二个error表示是否出错，为nil则成功。

理解接口:接口本身是一个静态类型，这可以与其它实体类型一样的使用，可以作为变量类型，参数类型，返回值类型等等。接口的值是动态的，它可以是类型A的值,也可以是类型B的值,只要它们实现了此接口即可。这与其它静态语言，如c++,java等多态有点小相似。

当做普通类型去使用

// 测试空接口

func EmptyInterface() {

var r io.Reader

fmt.Printf("io.Reader type[%v] value[%T]\n", r, r)

}

Output: io.Reader type[] value[]

接口是可做普通类型去使用的，没有实体值，则为nil。若有实体值，类型则是实体值本身的静态类型。

### 7.1.2 接口实现

接口实现的必要条件是**实现某接口的全部方法**。

我们利用这个接口做一个流程处理，数据源使用接口Read方法。这个流程就可以处理多种类型，以及不同数据源。

实现Reader

type space struct {

str string

}

// 实现io.Reader.Read

func (sp \*space) Read(p []byte) (int, error){

n := len(sp.str)

// 循环读取str 此处禁止使用append

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

p[i] = sp.str[i]

}

// 等价上面循环的部分

// n = copy(p, sp.str[:])

return n, nil

}

此处实现需要注意参数p的长度，以及起始位置，使用append的话首先会浪费空间，然后可能有out of range 的编译报错。

简单使用

我们将读取来源用，统一使用的接口方法（Read）取出数据[]byte，做一个简单的字符串截取处理流程。这样此流程就可以使用不同的数据源，如reader注释的标准输入流，文件流，我们自己定义的结构体，等等。

func InterfaceCom() {

fmt.Println("======InterfaceCom======")

sp := &space {

str : "hello, pite\n hi sam",

}

// reader := bufio.NewReader(os.Stdin) // 从标准输入生成读对象

// reader := bufio.NewReader(fb) // 从文件中读取

reader := bufio.NewReader(sp)

text, \_ := reader.ReadString('\n') // 读到换行

text = strings.TrimSpace(text)

fmt.Printf("bufio.NewReader %#v\n", text)

}

Output: bufio.NewReader "hello, pite"

为了清晰，这里我们使用了标准库的bufio以及strings辅助完成这个小流程。使用io.Reader的标准库函数有fmt的Fscan，Fscanf，Fscanln等，还有很多第三方库的日志库，消息队列处理等。

这也算是我们初步体验接口的妙用。

## 7.2 接口嵌入组合

Package io 还有一个Writer接口定义。

此接口与Reader接口同样经典。

type Writer interface {

Write(p []byte) (n int, err error)

}

组合嵌入

很多时候我们同时需要Read和Write方法的接口，那么并不需要完全重新去定义它。

type ReadWriter interface {

Reader

Writer

}

混合型

type ReadWriteCloser interface {

Reader

Writer

Close() error

}

注：

这些特性与结构体是很相似的。同样也不会存在顺序的要求。

A,B接口嵌入到C，则实现了C接口的类型同样实现了A和B，很自然的一个包含关系。

实现接口io.Writer

我们继续用space结构体去实现io.Writer

func (sp \*space) Write(p []byte)(int, error) {

sp.str = string(p)

return len(sp.str), nil

}

先从理论上分析，space结构体实现了io.ReaderWriter，那么space则同时实现了io.Writer 和io.Reader。

测试code

var w io.Writer

var r io.Reader

var wr io.ReadWriter = &space{"hello"}

w = wr

r = wr

fmt.Println(w,r,wr)

Output: &{hello} &{hello} &{hello}

从输出结果可以看出，io.ReadWriter的变量可以直接赋值给io.Reader和io.Writer。且它们输出的结果是相同的。

## 7.3 接口类型

习惯于使用面向对象语言的，更习惯称为多态。习惯使用脚本语言的，加之go接口与实体类型松散彻底分离的特点，称之为动态类型。还有说go是面向接口的编程。无论哪一种称谓并不重要，仅仅是团队相互交流的词汇，同时小艾并不认为他们说错。

重要的是我们掌握接口的特性变化，习惯于接口编程的方式，能让它成为你手中的武器就好了。

这一节我们仍然继续使用使用io.Reader，io.Writer和io.ReadWriter等接口来继续细说一下。

### 7.3.1 接口类型

普通接口类型都需要使用type关键字，至于你是单独声明的还是使用嵌入式方法组合它，原理其实都是一样的。

Go接口是**虚类型**。运行过程中它实际代表的类型并不是由它本身决定的。

我们看下实例

var w io.Writer

w = os.Stdout

fmt.Printf("io.Writer 1 %T\n", w)

w = &space{}

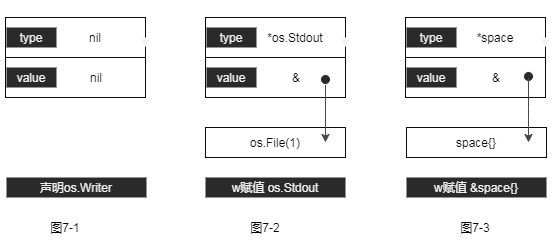
fmt.Printf("io.Writer 2 %T\n", w)

Output : io.Writer 1 \*os.File

io.Writer 2 \*main.space

我们定义的io.Writer，而在运行过程中它的类型是根据我们赋的值而决定的。所以我们更加可以确定go接口类型是一种虚类型与其它类型实体不同。

接口io.Writer变量w的值变化示意图



接口类型转化是隐式的

接口赋值会导致类型和值可能会变化

上图7-1 也许显得有些突兀，不过这就是空接口的示意图。

### 7.3.2 nil接口(空接口)

类型是nil，值也是nil才会是空接口。

上图7-1是就是空接口的示意图，它的类型是nil而不是io.Writer，它的值也是nil。

判断是空接口的语句 w == nil 或 w != nil。

空接口调用方法会产生panic，禁止使用空接口调用方法。

空接口panic示例

var w io.Writer

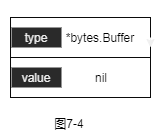
w.Write([]byte("hello")) // panic: nil pointer dereference

上面的代码无法运行，会报空指针引用错误。因此在使用接口变量或者参数的时候，我们首先就应该判断是否为nil。但**首先你要确认的是它是空接口**，加黑字其实有原因的，下一小节我们就说明了这个原因。

### 7.3.3 nil指针的接口与nil接口不同·深

我们先要确认什么是**nil指针的接口**。接口类型不是nil且值是nil，此种情况下它与nil并不相等，nil接口==nil 所以nil指针的接口不等于nil接口。此小节是一个很深的**陷阱**，编写代码时能正确的处理这个陷阱就是我们的重点。

**Nil指针的接口**



上图就是空指针的接口示意图。我们先看看空nil的接口是怎么产生的，为什么会发生这个情况？

我们从一段程序上去分析

var buf \*bytes.Buffer

fmt.Printf("[STEP1] buf T[%T] V[%v] buf==nil[%v]\n", buf, buf, buf == nil)

var w io.Writer

fmt.Printf("[STEP2]io.Writer T[%T] V[%v] w==nil[%v]\n", w, w, w == nil)

w = buf

fmt.Printf("[STEP3]w=buf T[%T] V[%v] w==nil[%v]\n", w, w, w == nil)

// STEP4

if w != nil {

w.Write([]byte("hello"))// 执行到Write 且 编译报错

}

Output:

[STEP1] buf T[\*bytes.Buffer] V[<nil>] buf==nil[true]

[STEP2]io.Writer T[<nil>] V[<nil>] w==nil[true]

[STEP3]w=buf T[\*bytes.Buffer] V[<nil>] w==nil[false]

程序在STEP1时，这是个指向bytes.Buffer的空指针，等于nil。

在STEP2时，刚声明完io.Writer接口，是个空接口==nil

从上面2个情况来看，buf和w都是nil，但是经过STEP3将buf赋值给w的时候为什么就不等于nil了？STEP4还会判空失败且去执行w.Write。

答案：空指针发生了类型转换，转换成接口类型io.Writer，但是w变量的类型发生了变化是\*bytes.Buffer 而值是nil，编译成了**nil指针的接口**与nil不等。

总结：

1**实体类型空指针转换接口类型**会发生这种情况。我们将上面的bytes.Buffer换成是我们前面定义的space效果同样如此。

2 go不能直接判断接口的**nil指针的接口（类型不为空，值为空）**这是一个很严重的陷阱。这一点的确实会给让我们程序复杂一些。

**解决问题**

问题的原因我们已经找到了，那么如何解决或者避免呢？

先重新定义一个比较接近项目上使用的函数

这与上面那段说明代码产生的问题是一样的。

func f2(w io.Writer) error {

if w == nil {

return errors.New("param nil")

}

w.Write([]byte("hello"))

fmt.Printf("[STEP]w=buf T[%T] V[%v] w==nil[%v]\n", w, w, w == nil)

return nil

}

* **方法1**

**避免实体类型空指针转换成接口**，实体类型转换是没有任何问题的。比如说传值给接口时先判断nil，或者初始化的变量本来就是接口避免是实体对象的指针。这个方法很多人会觉得别扭不太符合程序的简易性。

直接赋值在变量类型是接口类型

var w2 io.Writer

w2 = &space{}

f2(w2)

默认用new去初始化变量

var w2 io.Writer

w2 = new(space)

f2(w2)

* **方法2**

非接口类型先判断空指针

var w2 \*space

if w2 != nil {

f2(w2)

}

* **方法3**

使用映射，性能会低一些

func IsValidObject(value interface{}) bool {

val := reflect.ValueOf(value)

if val.Kind() == reflect.Ptr {

val = val.Elem()

}

return val.IsValid()

}

func f2(w io.Writer) error {

if !IsValidObject(w) {

return errors.New("param nil")

}

w.Write([]byte("hello"))

fmt.Printf("[STEP]w=buf T[%T] V[%v] w==nil[%v]\n", w, w, w == nil)

return nil

}

### 7.3.4 万能类型

Go中**interface{}**它没有规定任何方法，任何类型的值包括接口以及它本身都可以赋值给这种类型。这种类型几乎可以用于任何地方，变量，参数，返回值等等。

再7.3.2中我们专门讲述了空接口，所以我们在团队交流的时候避免说interface{}是空接口。这可是有着本质的区别的。

它的零值是nil。

不需要我们自定义，也就是不需要去使用type关键字。

变量中使用

var i interface{} = 2

i的值可以任何类型，无论是值，还是指针都可以。

我们定义个函数

func f3(i interface{}) {

// do somethings

}

此函数可以接收任何类型的值，传int，space,bytes.Buffer等等，无论什么类型都支持；无论是值类型还是指针类型也都支持。

从这一点上看，go算是动中有静，静中有动的一门语言了。一般大型项目静态语言会是比较好的选择，强制的类型限制，会让我们避免很多运行时的漏洞。从这一点就可以看出，go无论在大型项目还是在小型项目都可以使用。不过go官方一直是声称为大型项目准备的语言。

在前面我们说是推论接口无方法，从表面现象得出的结论，这其实是不正确的，go编译器对于这个类型是特殊处理的，至于如何做到万能类型我们可以不深究，可以很好的使用它就ok的，但这正好符合我们正常人的思维逻辑的语法不是吗？

## 7.4 断言

Go是静态类型的，那么接口类型与其它类型做计算的时候，需要类型转换。尤其是计算，或者使用实体类型的特有方法等。简单的说Go断言等价**接口类型的类型转换**。

断言为什么不叫类型转换呢？

1普通类型转换大部分都不需要做判断，即使转化失败也是相应的零值；

2接口断言更是让复合类型转换提供了可能。

3断言转换存在是失败的，失败时转化结果并不是转换的目标类型，不能用于后续语法。这点与我们通常的类型转换也存在了不一样的地方。

### 7.4.1 断言规则

1 断言就是将接口类型变量的值(x)，转换成类型(T)。格式为：**x.(T)**。

2 条件1的**x必是接口类型**，若不是接口类型则报错。

3 **T可以是任意类型**，

T是实体类型（非接口类型），若想转换成功，则T必须实现x的接口。

T是接口类型，成功转换，则x的动态类型也必须实现了T接口。或者说T中所有的方法都必须存在x接口内，或者是x包含T。

4 nil接口的断言总是失败的。

用法

var w io.Writer = os.Stdout

f, ok := **w.(\*os.File)**  // success: ok, f == os.Stdout

此处w相当于x，w的类型io.Writer是接口类型。T是\*os.File文件实体类型指针， \*os.File实现了io.Writer，因此此断言是成功的。变量ok的值等于true的布尔值。变量f是os.File的指针。

很少用的写法

f := w.(\*os.File)

这个看起是没有什么问题的，但运行时，断言失败后面使用\*os.File就会发生panic。程序不稳定显然不是我们想要的。

通常断言写法

if f, ok := w.(\*os.File); ok {

// do somethings with f …

}

分号前面的是初始化f和ok量个变量，f是目标类型(\*os.File)，ok是断言成败的布尔类型值。分号后根据ok的布尔值做if条件判断。断言成功则执行if作用域的代码。将断言成功，失败都支持了，程序就不会发生panic了。

### 7.4.2 断言类型x.(type)

断言可以获取动态变量的类型。只能配合switch使用，否则报错：“outside type switch”。

处理多种类型时，此种写法比if…else…多分枝情况简洁明了的多。

基本用法

func InterfaceConvType(arg1 interface{}) {

switch arg1.(type) {

case bool:

fmt.Println("Type of param is bool ", arg1)

case int, int32, int64:

fmt.Println("Type of param is int ", arg1)

case float32, float64:

fmt.Println("Type of param is float ", arg1)

default:

fmt.Println("Type of param is not valid\n", arg1)

}

}

此函数的参数是一个接口类型的万能类型。使用arg1.(type) 得出相应的类型，根据case的类型执行分支。但注意arg1.(type)不能脱离switch。

### 7.4.3 编译时检查接口实现

为了确认我们自定义类型的接口实现，要是在编译时就能确认，那无疑是能大大的提升编程效率。同时让我们更加确认了程序的行动路线，减少我们大脑中程序路线。而且相应类型的方法修改也不用担心接口实现。

也有运行时检查，这需要看我们项目的需要。

**编译时检查接**口实现的写法

var \_ io.ReadWriter = &space{}

var \_ io.ReadWriter = new(space)

以上2种写法都可以在编译时检查

**运行时检查**

var \_ io.ReadWriter = (\*space)(nil)

无论哪一种检查都不是必须的，习惯于使用此特性，能加快速，机械化比较安全的确认接口实现而已。

接口的特性细节我们基本介绍完了，后续小节，我们会介绍使用比较频繁的标准库接口。

## 7.5 error接口

前面章节中，我们也一直使用过error类型，这并不是关键字，它预定义的一个接口。零值nil一般代表没有错误发生。通常通过标准库的errors.New去创建。

很明显**错误处理**就是error接口解决的问题，error接口的设计目的很清晰。

源码中定义

type error interface {

Error() string

}

创建一个error

使用前记得import errors。

import "errors"

创建一个错误对象变量

err := errors.New("EOF")

返回值中创建错误对象

func doSome(args ...interface{}) error {

if len(args) == 0 {

return errors.New("Empty params!")

}

// todo somethings.

}

使用**fmt.Errorf**创建error对象

Errorf的源码在fmt包中，格式化创建error。

func Errorf(format string, args ...interface{}) error {

return errors.New(Sprintf(format, args...))

}

使用这个方法创建error的人可能会比较多，一个是格式化方便，另一个项目代码中可以少引用import errors。

### 7.5.1 errors源码

package errors

func New(text string) error {

return &errorString{text}

}

type errorString struct {

text string

}

func (e \*errorString) Error() string {

return e.text

}

源码中没有换行，所以仅仅有4行代码。

明显看出就是一个简单的结构体实现error接口。

这个包只要创建后就发现你没有办法修改它，它是只读的，对外也紧紧开放了一个Error只读的方法。errorString的成员text是私有成员不能被外面访问。

New创建的都是指针类型， 每次创建都分配在不同的地址和空间的新实例。因此errors.New("EOF") == errors.New("EOF") 返回的是false。

### 7.5.2 Errno

包syscall是Go语言底层系统调用API。在Unix平台上，Errno的Error方法会从一个字符串表中查找错误消息。字符串表其实就是个切片，**键**为错误码，**值**为错误信息。此方法明显比errors包更适合大型项目，易于维护，管理，切换多语言，以及高效等。

调用

var err error = syscall.Errno(2)

fmt.Println(err.Error()) // "no such file or directory"

fmt.Println(err) // 同上

若有疑问，请继续往下看

源码

package syscall

// 定义系统错误码类型

type Errno uintptr

// 错误码切片列表；键：错误码；值：错误信息串

var errors = [...]string{

1: "operation not permitted", // EPERM

2: "no such file or directory", // ENOENT

3: "no such process", // ESRCH

// ...

}

// 实现error接口

func (e Errno) Error() string {

if 0 <= int(e) && int(e) < len(errors) {

return errors[e]

}

return fmt.Sprintf("errno %d", e)

}

自定义整形类型Errno

errors定义错字符串列表，一个字符串切片

Error方法实现接口error，根据Errno的值直接读取errors切片

此种方式没用去额外创建内存空间，直接根据常量值去读取预先定义的字符串切片，性能是明显的高。

它毕竟只有系统部分的错误码，现实项目这些错误码，明显是不够用。我们可以自己可以按照这个方式自己设计一个错误码包，为了贴合业务，可以将errors切片换成map，以便支持不连续的错误码。

从新解读下这句代码

var err error = syscall.Errno(2)

首先var err error中error是接口类型。

err则是一个接口error类型的变量。

syscall.Errno是一个整形类型的别称且实现了error接口。

**syscall.Errno(2)**是将**常量2转换成syscall.Errno类型**，这里记住并不是初始化函数或者构造函数的东西。这里仅仅**整数的类型转换**。

所以syscall.Error(2)是可以正确赋值给err。

小结：从本小节内容，就可以看出error接口的使用：

1 简便只有一个Error()string方法

2 超过2个以上的实体类型实现了这个接口，分别是errorString和Errno。

3 一个整形和一个结构体，完全不同的类型，可以设计成同样的使用方式和流程。

4 error接口没有创建的方法，我猜也是总合Errno和errorString总合的考虑，简单的Errno并不需要额外的用法（脑回路）开销，减少复杂性。

## 7.6 flag包+flag.Value接口

介绍flag接口前我们同样，需要知道这个包是解决什么问题的。做服务端开发的都知道命令行参数，执行文件后面跟的参数。有的用，-参数名，—参数名等，就是命令行参数。我们就可以使用flag包去解析并使用这些命令行参数。

例如

查询一些软件或语言的帮助版本等功能

git --version

java -version

php -help

像上面那样的命令行参数解析我们就可以用flag包。若仅仅是按参数顺序取参数用os.args即可，他是参数的一个切片，这里我们就不去说明了，项目上并不实用。

本节重点是介绍flag的接口设计，为了我们更好的理解，我们介绍下flag包的具体应用吧。**7.6.1**和**7.6.2**是介绍flag内置的使用API解析命令行参数方式。**7.6.3**是利用flag包使用**自定义类型（实现flag.Value接口）**介绍。

**导入flag包**

import flag

**flag需要注意的细节**

Flag.Parse在程序生命周期仅会执行一次，调用第二次也不会再解析命令行。

### 7.6.1方式一·flag.T()

**T**是API函数名，首字母大写的3个基础类型关键字。

**API**

/\*----------------------------------------------------- -\*

\* 直接使用 flag.T(命令行变量名, 默认值, 帮助说明文案) \*

\* T为基础类型首字母大写 Bool, Int, String等 \*

\* API: flag.Bool, flag.Int, flag.String等 \*

\* flag.T()的返回值是指针变量 \*

\*-------------------------------------------------------\*/

**声明绑定变量**flag解析命令行参数的变量是用引用的方式，定义变量可以直接定义包级别指针。

var port \*int

var version \*bool

var name \*string

**绑定**命令参数到变量上

// 定义命令行指针变量和默认值

func initParseFlag() {

port = flag.Int("port", 808, "help message for port")

version = flag.Bool("version", false, "Check version of the soft!")

name = flag.String("name", "", "Name of project")

}

**处理**命令行逻辑

// 解析

func parseFlag() {

fmt.Println("====== parseFlag ======")

// 命令行指针port 端口非默认值打印出端口

if \*port != 808 {

fmt.Println("The progject port is", \*port)

}

// 命令行指针version 查看软件版本号

if \*version {

fmt.Println("Project version is 8.0.1")

}

// 命令行指针name 打印软件名

if \*name != "" {

fmt.Println("Project name is", \*name)

}

}

**解析命令流程**

func init() {

// 绑定命令行参数到变量

initParseFlag()

// 解析命令行

flag.Parse()

// 处理命令行参数

parseFlag()

}

命令行命令

./ch07 -version -port=707 -name=demo1

Output：

The progject port is 707

Project version is 8.0.1

Project name is demo1

### 7.6.2 方式二·flag.TVar()

我们以第二种方式同样实现方式一的功能。

API

/\*--------------------------------------------------\*

\* 直接使用 flag.TVar(存储指针变量, 命令行变量名, 默认值, 帮助说明文案)

\* T为基础类型首字母大写 Bool, Int, String 等

\* API: flag.BoolVar, flag.IntVar, flag.StringVar 等

\* flag.TVar将命令的值赋值给存储指针变量

\*--------------------------------------------------\*/

**绑定**命令参数到变量上

func initParseFlagVar() {

flag.IntVar(port, "p", 808, "help message for port")

flag.BoolVar(version, "v", false, "Check version of the soft!")

flag.StringVar(name, "nm", "", "Name of project")

}

**处理**命令行逻辑

func parseFlagVar() {

fmt.Println("====== parseFlagVar ======")

// 命令行指针port 端口非默认值打印出端口

if \*port != 808 {

fmt.Println("The progject port is", \*port)

}

// 命令行指针version 查看软件版本号

if \*version {

fmt.Println("Project version is 8.0.1")

}

// 命令行指针name 打印软件名

if \*name != "" {

fmt.Println("Project name is", name)

}

}

**解析命令流程**

func init() {

// 绑定命令行参数到变量

initParseFlagVar ()

// 解析命令行

flag.Parse()

// 处理命令行参数

parseFlagVar ()

}

执行

./ch07 -v -p=707 -nm=demo2

Output:

The progject port is 707

Project version is 8.0.1

Project name is demo2

### 7.6.3 方式三·flag.Var()

Flag.Var将命令行参数绑定在**自定义类型**且**实现flag.Value接口**的API。使用流程与前2种方式是一样的。

/\*--------------------------------------------------\*

\* 直接使用 flag.Var(T, 命令行变量名, 默认值, 帮助说明文案)

\* T为自定义类型

\* T 需要实现flag包的

// Value 接口

type Value interface {

String() string

Set(string) error

}

\* flag.TVar将命令的值赋值给存储指针变量

\*--------------------------------------------------\*/

**flag.Value接口**

此接口在flag包内，可直接使用。

type Value interface {

String() string

Set(string) error

}

**自定义类型**且实现**flag.Value**

type fint int

// 实现flag.Value.Set

func (i \*fint) Set(s string) error {

v, err := strconv.ParseInt(s, 0, strconv.IntSize)

\*i = fint(v)

return err

}

// 实现flag.Value.String

func (i \*fint) String() string {

return strconv.Itoa(int(\*i))

}

**绑定**命令参数到变量上

var vs = fint(0)

func initParseFlagDefine() {

flag.Var(&vs, "vs", "")

}

**处理**命令行逻辑

func parseFlagDefine() {

if vs != 0 {

fmt.Println("parse flag vs :", vs)

}

}

**解析命令流程**

func init() {

// 绑定命令行参数到变量

initParseFlagDefine ()

// 解析命令行

flag.Parse()

// 处理命令行参数

parseFlagDefine ()

}

**flag.Value的2个方法**：

Set(string)由flag.Parse()内执行，给绑定变量赋值。

String()string为不能做比较的类型做条件判定的方法。上例中vs != 0可以改写成vs.String() != "0"。

# 第八章 序列化数据结构

序列化是为跨进程，跨服务以及接口等等进行数据传输，简化双方通信。它们独立于语言平台，独立于语言。

例如：网站接口提供的多是json结构的数据流，前端和后端进行数据传输；游戏多是使用protobuf方式；还有很多使用xml格式进行数据交换；同样还有自己研发的其它格式的序列化格式。

Go在web以及rpc领域的强盛，对序列化当然也会进行强有力的支持。

本章主要介绍我们常用的json，protobuf，和xml格式，要是初学仅仅先熟悉json即可。

## 8.1 JSON

Go对json支持，主要使用标准库包encoding/json，还有第三方的包。我们主要以标准库为例进行说明。

Go是强类型语言，json可算是弱类型序列化，这之间是存在矛盾的。强类型语言与弱类型语言的序列化数据交换也会经常不稳定。

**引入包**

import "encoding/json"

**API**

// 编码成串

dataByte, \_ := **json.Marshal**(数据结构变量/指针)

// 解码成数据结构

**json.Unmarshal**([]byte(dataString), 数据结构指针)

编码：将结构化数据结构，编译成字节流，在go中是字节切片。可直接使用string(dataByte)转成字符串。

解码：将字节流转成go中的数据结构体。

### 8.1.1 序列化和反序列化DEMO

**序列化**

// Product 商品信息

type Product struct {

Name string

ProductID int64

Number int

Price float64

IsSale bool

model string

}

// 简单的序列化

func jsonSimpleMarshal() {

fmt.Println("=======jsonMarshal=======")

p := &Product{

Name:"ios",

ProductID :7239123023421,

Number:10,

Price: 20.5,

IsSale:false,

model:"TCL",

}

data, \_ := json.Marshal(p)

fmt.Println(string(data))

}

Output:

{"Name":"ios","ProductID":7239123023421,"Number":10,"Price":20.5,"IsSale":false}

**反序列化**

// 反序列化

func jsonSimpleUnmarshal() {

data := `{"Name":"ios","ProductID":7239123023421,"Number":10,"Price":20.5,"IsSale":false}`

p := &Product{}

err := json.Unmarshal([]byte(data), p)

fmt.Println("return:", err, "; struct:", \*p)

}

Output：

return: ; struct: {ios 7239123023421 10 20.5 false }

### 8.1.2 带标签的结构体序列化

标签tag就是给结构体每个字段打上标签，用反引号括起来，冒号前是标签类型，后面是标签名以及一些规则。

JSON库会按照标签的定义来更进一步的序列化。我们可以一步一步进化这个结构体。

**1 结构体字段名改名**

// Product 商品信息

type Product struct {

Name string `json:"name"`

ProductID int64 `json:"pid"`

Number int `json:"num"`

Price float64 `json:"price"`

IsSale bool `json:"is\_sale"`

model string `json:"m"`

}

Output：

{"name":"ios","pid":7239123023421,"num":10,"price":20.5,"is\_sale":false}

**2 忽略0值字段**

在标签里面加入omitempty关键字。

type Product struct {

Name string `json:"name"`

ProductID int64 `json:"pid"`

Number int `json:"num,omitempty"`

Price float64 `json:"price"`

IsSale bool `json:"is\_sale"`

model string `json:"m,omitempty"`

}

func jsonTagEmptyMarshal() {

fmt.Println("=======jsonTagEmptyMarshal=======")

p := &Product{

Name:"ios",

ProductID :7239123023421,

Number:0,

Price: 20.5,

IsSale:false,

model:"",

}

data, \_ := json.Marshal(p)

fmt.Println(string(data))

}

Output:

{"name":"ios","pid":7239123023421,"price":20.5,"is\_sale":false}

**3 类型转换**

将结构体字段进行类型转换，如json中经常将所有字段转成字符串等。

type Product struct {

Name string `json:"name"`

ProductID int64 `json:"pid"`

Number int `json:"num,omitempty"`

Price float64 `json:"price"`

IsSale bool `json:"is\_sale,string"`

model string `json:"m,omitempty"`

}

Output：

{"name":"ios","pid":7239123023421,"price":20.5,"is\_sale":"false"}

以上3种情况，序列化和反序列化，都是相互支持的。

### 8.1.3 自定义解析

有时，你想在解码或者序列化过程中进行，校验，转换等等操作。比如基本的类型转换等，或将远端传递的缺失的字段补充等等。那么这就是json库的自定义解析。

**需要实现的接口**

type Marshaler interface {

MarshalJSON() ([]byte, error)

}

type Unmarshaler interface {

UnmarshalJSON([]byte) error

}

**具体实现**

编码时，将map转化为int数组。

解码时，将数字数组转化为map。

type Order struct {

pids map[string]int64

}

func (o \*Order) MarshalJSON() ([]byte, error) {

l := []int64{}

for \_,v := range o.pids {

l = append(l,v)

}

return json.Marshal(l)

}

func (o \*Order) UnmarshalJSON(b []byte) error {

l := []int64{}

err := json.Unmarshal(b, &l)

if err != nil {

return err

}

for i,v := range l {

k := fmt.Sprintf("%d", i)

o.pids[k] = v

}

return nil

}

func jsonDefine() {

fmt.Println("=======jsonDefine=======")

m := map[string]int64{"1":234, "2":987,"3":110, "4":119}

o := &Order{pids:m}

byteOrder, err := json.Marshal(o)

fmt.Println("JSON defined", "error:",err, "order:", string(byteOrder))

mstr := `[234, 987, 110, 119]`

o1 := &Order{pids:map[string]int64{}}

err = json.Unmarshal([]byte(mstr), o1)

fmt.Println("JSON defined unmarshal", "error:",err, "order:", o1)

}

Output:

JSON defined error: order: [234,987,110,119]

JSON defined unmarshal error: order: &{map[0:234 1:987 2:110 3:119]}

## 8.2 Protobuf简介

Protobuf 性能和效率大幅度优于 JSON、XML 等其他的结构化数据格式。protobuf 是以二进制方式存储的，数据量小，传输快，但可读性差，想要直接读取数据流需要借助特定的工具。protobuf 在通信协议和数据存储等领域应用广泛。例如游戏领域，以及著名的分布式缓存工具 Memcached 的 Go 语言版本groupcache 就使用了 protobuf 作为其 RPC 数据格式。

Protobuf在.proto文件中定义数据结构和规则等，可以用protoc等工具将.proto转化成其它语言的规则文件。github.com/golang/protobuf/proto是利用.proto规则将二进制流与go语言结构性数据进行转换的包。

### 8.2.1 安装protoc

Protoc工具与语言无关，主流语言java,c++,python,php等都有将protobuf互译对应语言的识别的结构。

安装protobuf看你是否能科学上网，对应命令ping google.com如果能只需要一条命令即可；不能的话则需要执行多条命令。

**科学方式**

这条命令本质上先git clone下载，然后go install安装软件包。

go get github.com/golang/protobuf/proto

**仅能普通上网**

需要先从网站上下载安装压缩包，使用ftp, rz或者wget上传到服务器上。

下载网址

https://github.com/protocolbuffers/protobuf/releases

下载命令

wget https://github.com/protocolbuffers/protobuf/releases/download/v3.14.0/protobuf-all-3.14.0.tar.gz

解压

tar zxvf protobuf-all-3.14.0.tar.gz

进入目录

cd protobuf-3.14.0/

**安装**

检查

安装前检查依赖包autoconf,automake,libtool是否已安装。检查yum info xxx。安装依赖yum install -y xxx。

执行脚本

./autogen.sh

运行配置脚本

./configure

Make安装

make #要编译很久

make check #测试

make install #安装

检查是否安装成功

protoc --version #查看版本

注：configure未使用—prefix时，默认安装目录在/usr/local。使用则是自定义的安装目录。

### 8.2.2 定义proto消息

Protobuf需要建立proto消息结构模板。

syntax = "proto3";

package message;

message Person {

string name = 1;

int32 age = 2;

repeated string emails = 3;

repeated PhoneNumber phones = 4;

}

message PhoneNumber {

string number = 1;

PhoneType type = 2;

}

enum PhoneType {

MOBILE = 0;

HOME = 1;

WORK = 2;

}

**转proto-go**

在proto消息文件目录下执行命令生成go文件

#转化命令

$ protoc --go\_out=. \*.proto

# 确认转化是否OK

$ ls

entity.pb.go entity.proto

**解读entity.proto**

Protobuf有2个版本，分别是proto2和proto3。Proto2是默认版本，选择版本的语法：

syntax = "proto3";

package关键字是可选的，与go的package都有区分命名冲突的作用。在go中最好将它当成必填项，它对应go的package关键字。

message关键字定义消息类型。

字段修饰符，singular和repeated，默认都是singular，可省略不写的。repeated表示go数组

字段类型是常用的标量类型string,bool,int等，也可以是消息类型（使用message定义的类型）。Name,age,email是常用的基础类型，phones使用的PhoneNumber类型。

字段需要标识符也就是=号后面的数字，可以理解为此字段的数据存放的位置。二进制数据与语言结构化数据互相转化的依据。

Enum关键字是枚举

注释语法，单行用//，多行可以用/\*\*/

### 8.2.3 使用protobuf

package main

import (

"fmt"

"github.com/aixgl/ch08/message"

"github.com/golang/protobuf/proto"

)

func encodeProto() {

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

person := &message.Person{

Name: "xyj",

Age: 30,

Emails: []string{"00000000@qq.com", "88888888@163.com"},

Phones: []\*message.PhoneNumber{

&message.PhoneNumber{

Number: "011-23029321",

Type: message.PhoneType\_HOME,

},

&message.PhoneNumber{

Number: "13800000000",

Type: message.PhoneType\_MOBILE,

},

&message.PhoneNumber{

Number: "010-88888888",

Type: message.PhoneType\_WORK,

},

},

}

//编码：marshal: obj---[]byte

data, err := proto.Marshal(person)

if err != nil {

fmt.Println("error", err)

}

fmt.Println("data", data)

// 解码：decode

person1 := &message.Person{}

err = proto.Unmarshal(data, person1)

fmt.Println("error info:", err)

fmt.Println("person:", person1)

}

Output:

data [10 3 120 121 106 16 30 26 15 48 48 48 48 48 48 48 48 64 113 113 46 99 111 109 26 16 56 56 56 56 56 56 56 56 64 49 54 51 46 99 111 109 34 16 10 12 48 49 49 45 50 51 48 50 57 51 50 49 16 1 34 13 10 11 49 51 56 48 48 48 48 48 48 48 48 34 16 10 12 48 49 48 45 56 56 56 56 56 56 56 56 16 2]

error info: <nil>

person: name:"xyj" age:30 emails:"00000000@qq.com" emails:"88888888@163.com" phones:{number:"011-23029321" type:HOME} phones:{number:"13800000000"} phones:{number:"010-88888888" type:WORK}

### 8.2.4 protobuf类型介绍

#### 1 标量类型

Protobuf的字段类型也是类C的类型。常见的标量类型如string, byte, bool, int32, double等。

标量字段未赋值，则有赋值默认值。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **proto类型** | **go类型** | **备注** | **proto类型** | **go类型** | **备注** |
| double | float64 |  | float | float32 |  |
| int32 | int32 |  | int64 | int64 |  |
| uint32 | uint32 |  | uint64 | uint64 |  |
| sint32 | int32 |  | sint64 | int64 |  |
| fixed32 | uint32 | 固定长度 | fixed64 | uint64 | 固长编码 |
| sfixed32 | int32 | 固长编码 | sfixed64 | int64 | 固长编码 |
| bool | bool |  | string | string | UTF8 编码，长度不超过 2^32 |
| bytes | []byte | 任意字节序列，长度不超过 2^32 |  |  |  |

#### 2 枚举

适用于预定义的一组值，多选一。

message Teacher {

string name = 1;

enum Sex {

FEMALE = 0;

MALE = 1;

}

Sex sex = 2; //枚举类型

}

#### 3 自定义类型

适用定义的消息类型去定义字段。

message Person {

string name = 1;

int32 age = 2;

repeated string emails = 3;

// 自定义类型

repeated PhoneNumber phones = 4;

}

message PhoneNumber {

string number = 1;

}

嵌入式写法，一般不具有公用性的消息类型，会如此做。

message Person {

string name = 1;

int32 age = 2;

repeated string emails = 3;

// 嵌入式写法

message PhoneNumber {

string number = 1;

}

// 自定义类型

repeated PhoneNumber phones = 4;

}

#### 4 map类型

message MapList {

map<string, int32> scores = 1;

}

#### 5 one of

Oneof 字段类似于常规字段，除了Oneof共享内存的所有字段之外，最多可以同时设置一个字段。设置Oneof 的任何成员都会自动清除所有其他成员。您可以使用case()或WhichOneof()方法检查Oneof 中的哪个值被设置(如果有的话)，具体取决于您选择的语言。

message MainMessage {

oneof filed\_oneof {

string name = 4;

string alias = 9;

}

}

设置oneof字段将自动清除oneof字段的所有其他成员。因此，如果您设置了几个oneof字段，则只有最后一个字段仍然有值。

#### 6 any类型

可以简单的理解成万能类型或者泛型。但它的值需要符合可以被序列化bytes类型的任意消息，以及一个URL作为一个全局标识符和解析消息类型。

使用时需要先引入 import "google/protobuf/any.proto"。

import "google/protobuf/any.proto";

message ErrorStatus {

string message = 1;

repeated google.protobuf.Any details = 2;

}

### 8.2.5 import

Import关键字，主要的作用也就是导入。它导入其它的proto文件，从而引入需要的消息类型。

import "myproject/other\_protos.proto";

### 8.2.6 service定义服务

进程间接口通信，如想要将消息类型用在RPC(远程方法调用)系统中，可以在.proto文件中定义一个RPC服务接口，protocol buffer编译器将会根据所选择的不同语言生成服务接口代码及存根。如，想要定义一个RPC服务并具有一个方法，该方法能够接收 SearchRequest并返回一个SearchResponse，此时可以在.proto文件中进行如下定义：

service SearchService {

rpc Search (SearchRequest) returns (SearchResponse);

}

### 8.2.7 protoc工具

命令模板

protoc --proto\_path=<IMPORT\_PATH> --<lang>\_out=DST\_DIR path/to/file.proto

--proto\_path=<*IMPORT\_PATH>*：

IMPORT\_PATH需要替换成我们的import的目录

在.proto文件中若没有import其它proto则省略

Proto\_path是用来指定import的其它proto的目录的

--<lang>\_out=DST\_DIR：

指定生成代码的目标文件夹

例如 –go\_out=. 即生成 GO 代码在当前文件夹

Lang可以替换成go, cpp,java,python,ruby,objc,csharp,php等

### 8.2.8 资料

Github:

https://github.com/protocolbuffers/protobuf

https://github.com/protocolbuffers/protobuf/blob/master/docs/third\_party.md

## 8.3 XML

在go中Xml的编码与反编码可能会更简单一点。标准库中就有包同样是encoding库的。

**Import**

import "encoding/xml"

**API**

// 编码

xml.Marshal(struct)

xml.MarshalIndent(struct, sep string, sep2 string)

// 解码

xml.Unmarshal(buf []byte, struct)

**结构体转xml**

type Computer struct {

//XMLName xml.Name `xml:"computer"` // XMLName标签名

Id int `xml:"id"`

Memory int `xml:"memory"`

Cpu int `xml:"cpu"`

Keyboard bool `xml:"keyboard"`

Display string `xml:"display"`

}

结构体是有一定要求：

1 要导出的字段首字母需要大写

2 定义后的字段需要加上`xml:"字段名"`做xml说明

func XmlMarshal() {

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

c := &Computer {

Id:82032,

Memory:16,

Cpu:4,

Keyboard:true,

Display:"LCD",

}

output, err := xml.MarshalIndent(c, " ", " ")

fmt.Println("xml-marshal")

fmt.Printf("%v\n", string(output))

fmt.Println("error:", err)

}

Output:

<computer>

<id>82032</id>

<memory>16</memory>

<cpu>4</cpu>

<keyboard>true</keyboard>

<display>LCD</display>

</computer>

**Xml转结构体**

func XmlUnmarshal() {

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

data := `

<?xml version="1.0" encoding="UTF-8" ?>

<computer>

<id>82032</id>

<memory>16</memory>

<cpu>4</cpu>

<keyboard>true</keyboard>

<display>LCD</display>

</computer>

`

c := Computer{}

xml.Unmarshal([]byte(data), &c)

fmt.Println(c)

}

Output:

--------XmlUnmarshal--------

{82032 16 4 true LCD}

# 第九章 并发

并发指一段时间内cpu处理的任务量，任务开始时间可以有先有后。

并行指同一时刻可以处理多个任务，任务开始时间都是一样的。

并发在服务端开发尤其显得重要，也是经常在各种场合经常谈论的话题，也更是项目评审中的重要指标。

## 9.1 Go语言并发

Go并发靠的是协程，有的人也叫Go程，英文标识goroutine。协程类似于线程，但并不相同，区别也是很大的。

Goroutine是由go运行时调度完成。而线程是操作系统直接调度完成。Goroutine资源占用远远小于线程，创建成千上万个协程也是轻而易举。线程占用的资源多，操作起来显得笨重。

Go语言更是提供channel辅助goroutine进行通信。Go秉承着CSP(Communicating Sequential Process)并发模式。

Go并发虽然相对线程并发更容易些，但是同样存在着难题和陷阱。并发编程本质上比顺序编程要困难一些，若是没有并发经验与理解更是会加倍迷茫。但您并不需要忧心，办法总比困难多，多花点时间看看例子，多写写运行下程序，多思考章节的例子，多思考自己写的例子。

## 9.2 goroutine

### 9.2.1 不一样的并发

在我们使用主流的后端语言，如java,c++,python等并发编程时，首先需要维护一个线程池，并且需要手动包装各式各样的任务，接着去调度执行和维护上下文切换。这显然需要消耗我们宝贵的时间与精力。能不能省略或简化这一个繁复的过程，更容易的实现并发呢？goroutine会给我们惊喜，让我们看看Go是怎么做的。

Goroutine是go并发的一个基本单位。Go编译器会智能的将goroutine中的任务分配给cpu。省略了上述中的多数过程，内置话调度，上下文切换机制，从而被称为现代化编程语言。你不需要去自己写线程，协程只需要简单粗暴的使用goroutine技能（go f()）。

f() // call 普通的函数调度

go f() // concurrentcy 并行调度

### 9.2.2 Go并发模型M,P,G

**Gorountine并不等同于协程**，有时候我们会简单的把它比作协程，但是我们自己要清楚。

不要用共享内存的方式来通信。作为代替，应该以通信作为手段来共享内存

**并发模型3元素**

M（machine）: 一个M代表一个内核线程。

P（processer）: 上下文环境，一个P就是某Go代码片段G执行时需要的资源。

G（groutine）: 一个G代表一个代码片段，P和M来搞G。

**运行关系**

一个M和一个P形成一个运行环境（线程+上下文）。这个P会包含一个G的队列，每次运行一个G。该队列中的G会依次传递这个运行环境，并且依次去运行。

M与内核调度实体（KSE）一对一的关联，且非常稳固，runtime system用M表示一个调度实体。

M与P之间的关系是易变的，在调度过程中会发生改变。

P与G之间的关系是易变的，在调度过程中会发生改变。

### 9.2.3一个简单的并发