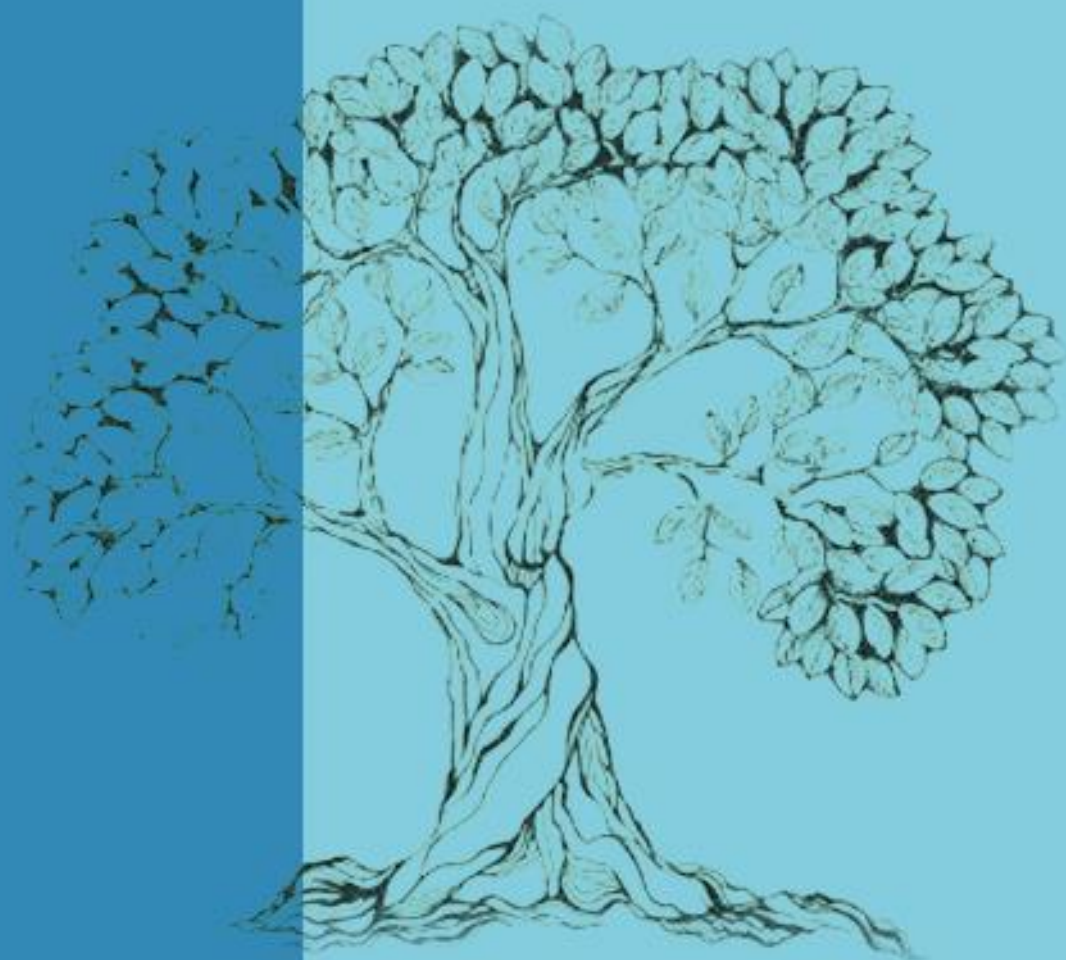


GO语言筑基

```
func main() {  
    fhs := []int{3, 4, 5}  
    var wg sync.WaitGroup  
    for idx, v := range fhs {  
        wg.Add( delta: 1)  
        // ...  
    }  
    wg.Wait()  
}
```



许亚军 著

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。

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，以及一些 Go 隐形问题，以及建议所构成。

Go 简书

创始人 [Robert Griesemer](#)，[Rob Pike](#) 和 [Ken Thompson](#) 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。

若您无基础，可以将标注“深”，“精”的小节跳过，通读后再返回继续研读。为了读着方便加了一个目录简介，快速定位需要的内容。

程序的运行环境若无特别都是建立在 AMD64 环境说明，默认是 64 位机，32 位我们很少说明讨论。

目录简介

第一部分 · 基础后天篇

第一章	新手入门	14
第二章	程序结构基础	26
第三章	基础数据类型	37
第四章	表达式	58
第五章	复合结构数据类型	68
第六章	函数与方法	90
第七章	接口	115
第八章	编码、解码	137

第二部分 · 进阶练气篇

第九章	并发	154
第十章	锁和共享内存	183
第十一章	包	198
第十二章	工具链	206
第十三章	测试姿势	240
第十四章	反射	263
第十五章	设计模式	278

第三部分 · 深度筑基篇

第十六章	runtime	293
第十七章	内存与 GC	302
第十八章	unsafe	320
第十九章	go 汇编	328
第二十章	CGO	351
第二十一章	网络编程	360
第二十二章	生态圈	376

目录

第一章 新手入门.....	16
1.1 Hello, world.....	16
1.2 *unix 环境.....	17
1.2.1 官方安装地址	17
1.2.2 安装步骤	18
1.2.3 运行 hello, world!.....	20
1.3 命令行参数	21
1.4 对话小程序	23
1.5 类 C 语言	24
1.6 感悟	25
第二章 程序结构基础.....	26
2.1 关键字和内置词	26
2.2 包与 go 源码文件·简	27
2.3 命名	28
2.4 声明	28
2.5 变量	30
2.5.1 简短变量	31
2.5.2 指针	32
2.5.3 new.....	32
2.5.4 变量的生命周期	33
2.6 赋值	34
2.7 作用域	35
第三章 基础数据类型.....	38
3.1 整数	39
3.2 浮点数	41
3.3 布尔值	42

3.4 字符串	42
3.4.1 字符串与切片	43
3.4.2 字符串连接	44
3.4.3 字符串底层实现·深	45
3.5 常量	46
3.6 复数	48
3.7 类型转换	49
3.7.1 数值型转换	49
3.7.2 数字+布尔与字符串	51
3.7.3 字节与字符串·精	54
3.8 自定义类型	58
第四章 表达式与控制流.....	59
4.1 运算符	59
4.2 流程语句	60
4.2.1 IF.....	61
4.2.2 Switch.....	61
4.2.3 For	62
4.2.4 For range	63
4.2.5 Break, Continue, Goto.....	66
第五章 复合结构数据类型.....	69
5.1 数组	69
5.2 切片	70
5.2.1 初次相识	71
5.2.2 内置函数 copy.....	72
5.2.3 内置 append 函数	72
5.2.4 切片的删除插入操作	73
5.2.4 切片申请的内存	74
5.2.5 切片内存布局·深	75

5.3 Map	77
5.3.1 初次相识	78
5.3.2 利用 map 分发简化流程	79
5.4 结构体	80
5.4.1 声明与定义	81
5.4.2 体验结构体	82
5.4.3 结构体嵌入	84
5.4.4 看看结构体内存	85
5.4.5 结构体大小	86
5.4.6 内存对齐	87
第六章 函数与方法.....	91
6.1 函数语法	91
6.2 函数可作为值	93
6.3 递归	94
6.4 错误处理	95
6.5 匿名函数	97
6.6 控制流函数	98
6.6.1 panic.....	98
6.6.2 defer	100
6.6.3 recover	104
6.7 方法定义	105
6.8 方法接收器	107
6.9 方法与嵌入结构体	109
6.10 方法与函数之间的差异总结	113
6.11 函数类型·精	114
第七章 接口.....	117
7.1 接口定义与实现	117
7.1.1 源码 Reader 接口定义	118

7.1.2 接口实现	119
7.2 接口嵌入组合	120
7.3 接口类型	121
7.3.1 接口类型	122
7.3.2 nil 接口(空接口)	122
7.3.3 nil 指针的接口与 nil 接口不同·深.....	123
7.3.4 万能类型	126
7.4 断言	127
7.4.1 断言规则	127
7.4.2 断言类型 x.(type)	128
7.4.3 编译时检查接口实现	128
7.5 error 接口	129
7.5.1 errors 源码.....	130
7.5.2 Errno.....	131
7.6 flag 包+flag.Value 接口	132
7.6.1 方式一·flag.T()	133
7.6.2 方式二·flag.TVar()	135
7.6.3 方式三·flag.Var()	136
第八章 序列化·编码/解码	错误!未定义书签。
8.1 JSON	错误!未定义书签。
8.1.1 序列化和反序列化 DEMO	错误!未定义书签。
8.1.2 带标签的结构体序列化	错误!未定义书签。
8.1.3 自定义解析	错误!未定义书签。
8.2 Protobuf 简介	错误!未定义书签。
8.2.1 安装 protoc.....	错误!未定义书签。
8.2.2 定义 proto 消息	错误!未定义书签。
8.2.3 使用 protobuf	错误!未定义书签。
8.2.4 protobuf 类型介绍	错误!未定义书签。

8.2.5 import	错误!未定义书签。
8.2.6 service 定义服务	错误!未定义书签。
8.2.7 protoc 工具	错误!未定义书签。
8.2.8 资料	错误!未定义书签。
8.3 XML	错误!未定义书签。
第九章 并发.....	错误!未定义书签。
9.1 Go 语言并发	错误!未定义书签。
9.2 goroutine.....	错误!未定义书签。
9.2.1 不一样的并发	错误!未定义书签。
9.2.2 一个简单的并发	错误!未定义书签。
9.3 Channel	错误!未定义书签。
9.3.1 Channel 要素	错误!未定义书签。
9.3.2 无缓存 Channel	错误!未定义书签。
9.2.3 带缓存的 channel.....	错误!未定义书签。
9.2.4 单向 channel.....	错误!未定义书签。
9.3 并发编程 WaitGroup.....	错误!未定义书签。
9.3.1 使用通道控制并发	错误!未定义书签。
9.3.2 利用 sync.WaitGroup 实现.....	错误!未定义书签。
9.3.3 WaitGroup 对象方法.....	错误!未定义书签。
9.3.4 WaitGroup 注意事项.....	错误!未定义书签。
9.4 多路复用 select.....	错误!未定义书签。
9.4.1 Select 实现 timeout.....	错误!未定义书签。
9.4.2 Select 随机选择.....	错误!未定义书签。
9.4.3 利用 for{}监听	错误!未定义书签。
9.4.4 阻塞 goroutine.....	错误!未定义书签。
9.5 Goroutine 退出	错误!未定义书签。
9.5.1 退出信号-关闭通道	错误!未定义书签。
9.5.2 退出并发-Context 包.....	错误!未定义书签。

9.6 Go 并发模型	错误!未定义书签。
9.6.1 栈 (stack)	错误!未定义书签。
9.6.2 M,P,G	错误!未定义书签。
9.7 简单的聊天服务	错误!未定义书签。
9.7.1 服务端代码	错误!未定义书签。
9.7.2 客户端	错误!未定义书签。
第十章 锁和共享内存.....	错误!未定义书签。
10.1 竞争	错误!未定义书签。
10.2 互斥锁·sync.Mutex.....	错误!未定义书签。
10.2.1 锁的注意事项	错误!未定义书签。
10.2.2 使用 channel 模拟锁.....	错误!未定义书签。
10.2.3 互斥锁仅是个君子协定	错误!未定义书签。
10.3 读写锁·sync.RWMutex	错误!未定义书签。
10.4 并发安全工具与缓存	错误!未定义书签。
10.4.1 Sync.Once 懒初始化.....	错误!未定义书签。
10.4.2 Sync.Pool 内存池	错误!未定义书签。
10.4.3 小小的并发安全的缓存	错误!未定义书签。
第十一章 包.....	错误!未定义书签。
11.1 导入	错误!未定义书签。
11.2 导入路径	错误!未定义书签。
11.3 包引用方式	错误!未定义书签。
11.4 包加载	错误!未定义书签。
11.5 导出可见性	错误!未定义书签。
第十二章 工具链.....	错误!未定义书签。
12.1 Go build	错误!未定义书签。
12.2 Go run	错误!未定义书签。
12.3 go install	错误!未定义书签。
12.4 go mod	错误!未定义书签。

12.4.1 go mod 的基本操作	错误!未定义书签。
12.4.2 go1.13 后修改配置	错误!未定义书签。
12.4.3 go get 变化	错误!未定义书签。
12.4.4 高级操作	错误!未定义书签。
12.5 go get	错误!未定义书签。
12.6 go env.....	错误!未定义书签。
12.7 go clean	错误!未定义书签。
12.8 go list	错误!未定义书签。
12.9 go test.....	错误!未定义书签。
12.10 go fix.....	错误!未定义书签。
12.11 go doc 和 godoc.....	错误!未定义书签。
12.11.1 go doc.....	错误!未定义书签。
12.11.2 godoc.....	错误!未定义书签。
12.12 go tool pprof 性能分析杀器	错误!未定义书签。
12.12.1 程序栗子	错误!未定义书签。
12.12.2 Web 分析	错误!未定义书签。
12.12.3 通过终端	错误!未定义书签。
12.12.4 可视化	错误!未定义书签。
12.12.5 火焰图	错误!未定义书签。
12.13 go vet 和 go tool vet	错误!未定义书签。
12.13.1 Printf format 错误	错误!未定义书签。
12.13.2 Bool 问题	错误!未定义书签。
12.13.3 goroutine 在主线程退出后还未执行.....	错误!未定义书签。
12.13.4 unreachable code.....	错误!未定义书签。
12.13.5 小整数向右位移过多	错误!未定义书签。
12.13.6 使用前未检测 err.....	错误!未定义书签。
12.13.7 性能	错误!未定义书签。
第十三章 测试姿势.....	错误!未定义书签。

13.1 Go 测试能力	错误!未定义书签。
13.2 测试约定	错误!未定义书签。
13.3 功能测试	错误!未定义书签。
13.3.1 结合命令完整测试	错误!未定义书签。
13.3.2 致命错误	错误!未定义书签。
13.3.3 失败标记	错误!未定义书签。
13.3.4 并发加速测试	错误!未定义书签。
13.4 测试覆盖率	错误!未定义书签。
13.5 基准测试	错误!未定义书签。
13.5.1 基准测试工作原理	错误!未定义书签。
13.5.2 提高基准测试精度	错误!未定义书签。
13.5.3 基准函数-部分测试	错误!未定义书签。
13.5.4 基准测试-内存	错误!未定义书签。
13.5.5 警告	错误!未定义书签。
13.6 第三方测试框架	错误!未定义书签。
第十四章 反射.....	错误!未定义书签。
14.1 简书	错误!未定义书签。
14.2 反射两大基本接口类型	错误!未定义书签。
14.3 反射 reflect.Type.....	错误!未定义书签。
14.3.1 方法 doc	错误!未定义书签。
14.3.2 反射结构体	错误!未定义书签。
14.3.3 反射结构体 func (t *commonType) Elem() Type.....	错误!未定义书签。
14.3.4 reflect.Type 的两个获取类型的方法 Name 和 Kind 纠葛.....	错误!未定义书签。
14.4 reflect.Value	错误!未定义书签。
14.4.1 方法 doc	错误!未定义书签。
14.4.2 demo	错误!未定义书签。
14.4.3 修改引用的值	错误!未定义书签。

14.4.5 反射调用方法	错误!未定义书签。
14.5 反射缺陷	错误!未定义书签。
第十五章 常用设计模式.....	错误!未定义书签。
15.1 单例模式	错误!未定义书签。
15.2 工厂模式	错误!未定义书签。
15.3 观察者模式	错误!未定义书签。
15.4 策略模式	错误!未定义书签。
15.5 函数式编程·精	错误!未定义书签。
15.5.1 因子	错误!未定义书签。
15.5.2 柯里化	错误!未定义书签。
15.5.3 尾化递归	错误!未定义书签。
15.5.4 闭包	错误!未定义书签。
15.6 注入	错误!未定义书签。
15.6.1 net/http 注入简书.....	错误!未定义书签。
15.6.2 依赖注入总结	错误!未定义书签。
15.6.3 这种类型注入优势	错误!未定义书签。
第十六章 runtime.....	错误!未定义书签。
16.1 Runtime 版本历史.....	错误!未定义书签。
16.2 GMP 并发模型.....	错误!未定义书签。
16.3 Goroutine 状态流转	错误!未定义书签。
16.4 调度 schedule.....	错误!未定义书签。
16.5 sysmon 协程	错误!未定义书签。
16.6 网络 IO 不阻塞线程.....	错误!未定义书签。
第十七章 go 内存与 GC.....	错误!未定义书签。
17.1 go 内存.....	错误!未定义书签。
17.2 go 内存结构思想.....	错误!未定义书签。
17.3 go 内存结构.....	错误!未定义书签。
17.3.1 arena	错误!未定义书签。

17.3.2 spans	错误!未定义书签。
17.3.3 bitmap	错误!未定义书签。
17.4 Go 内存单元 mspan	错误!未定义书签。
17.5 内存管理组件	错误!未定义书签。
17.5.1 mcache	错误!未定义书签。
17.5.2 mcentral	错误!未定义书签。
17.5.3 mheap	错误!未定义书签。
17.6 分配	错误!未定义书签。
17.6.1 三种对象分配	错误!未定义书签。
17.6.2 分配顺序	错误!未定义书签。
17.7 GC 垃圾回收	错误!未定义书签。
17.7.1 GC 常用算法	错误!未定义书签。
17.7.2 STW	错误!未定义书签。
17.7.3 gc 触发条件	错误!未定义书签。
17.7.4 写屏障	错误!未定义书签。
总结	错误!未定义书签。
第十八章 底层编程 unsafe	错误!未定义书签。
18.1 简书	错误!未定义书签。
18.1.1 unsafe 包元素	错误!未定义书签。
18.1.2 uintptr	错误!未定义书签。
18.1.2 unsafe.Pointer	错误!未定义书签。
18.2 将 T1 转成 T2	错误!未定义书签。
18.3 unsafe.Pointer 和 uintptr	错误!未定义书签。
18.4 unsafe 包函数	错误!未定义书签。
第十九章 go 汇编	错误!未定义书签。
19.1 人生若只如初见	错误!未定义书签。
19.2 计算机结构	错误!未定义书签。
19.2.1 X86-64 结构	错误!未定义书签。

19.2.2 通用寄存器和 plan9	错误!未定义书签。
19.2.3 X86-64 指令集	错误!未定义书签。
19.3 GO 汇编语法	错误!未定义书签。
19.3.1 整形包变量	错误!未定义书签。
19.3.2 字符串包变量	错误!未定义书签。
19.3.3 常量	错误!未定义书签。
19.3.4 bool 布尔类型	错误!未定义书签。
19.3.5 数组与切片	错误!未定义书签。
19.4 Go 汇编函数	错误!未定义书签。
19.4.1 函数参数、返回值	错误!未定义书签。
19.4.2 函数局部变量	错误!未定义书签。
19.4.3 函数调用	错误!未定义书签。
19.4.4 PCDATA 和 FUNCDATA	错误!未定义书签。
19.5 条件与跳转	错误!未定义书签。
19.5.1 IF	错误!未定义书签。
19.5.2 FOR	错误!未定义书签。
19.6 未完成	错误!未定义书签。
第二十章 CGO	错误!未定义书签。
20.1 基础知识	错误!未定义书签。
20.2 从 C 中访问 Go 函数	错误!未定义书签。
20.2.1 导出全局函数	错误!未定义书签。
20.2.2 导出函数变量	错误!未定义书签。
20.2.3 函数指针回调	错误!未定义书签。
20.3 Go 字符串(string) 和 C 字符串(string)	错误!未定义书签。
20.4 C 数组转 Go 切片 Turning C arrays into Go slices	错误!未定义书签。
20.5 常见陷阱	错误!未定义书签。
注语	错误!未定义书签。
第二十一章 网络编程	错误!未定义书签。

21.1 TCP 协议	错误!未定义书签。
21.2 UDP 通信	错误!未定义书签。
21.3 HTTP·精	错误!未定义书签。
21.3.1 HTTP 客户端	错误!未定义书签。
21.3.2 HTTP 服务端	错误!未定义书签。
21.4 websocket	错误!未定义书签。
21.5 常见 web 框架.....	错误!未定义书签。
第二十二章 GO 生态圈	错误!未定义书签。
22.1 书籍	错误!未定义书签。
22.2 社区	错误!未定义书签。
22.3 资料	错误!未定义书签。
22.4 Go 项目	错误!未定义书签。

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

Microsoft Windows
Windows 7 or later, Intel 64-bit processor
[go1.15.5.windows-amd64.msi](#) (115MB)

Apple macOS
macOS 10.12 or later, Intel 64-bit processor
[go1.15.5.darwin-amd64.pkg](#) (117MB)

Linux
Linux 2.6.23 or later, Intel 64-bit processor
[go1.15.5.linux-amd64.tar.gz](#) (115MB)

Source
[go1.15.5.src.tar.gz](#) (22MB)

Stable versions

go1.15.5 ▾

File name	Kind	OS	Arch	Size	SHA256 Checksum
go1.15.5.src.tar.gz	Source			22MB	c1076b90c94b73e4d62a814802c484443d024e8c07abd4c922c57a071c84f1
go1.15.5.darwin-amd64.tar.gz	Archive	macOS	x86-64	117MB	359a4334b8c8f5a3067a5a76f16419791ac3fe461348e8e1a9c0b9719915f5d
go1.15.5.darwin-amd64.pkg	Installer	macOS	x86-64	117MB	428b251143315728277b98a104387052757b63ac324ef444c854a289606101e
go1.15.5.linux-386.tar.gz	Archive	Linux	x86	96MB	4c81794406136979724c711732009c7e2e7c794dbaaa2a043c00da34d4be0559
go1.15.5.linux-amd64.tar.gz	Archive	Linux	x86-64	115MB	9a58494e8da722c3aef248c9227b0e9c528c7318309827780f16220998180a0d
go1.15.5.linux-arm64.tar.gz	Archive	Linux	ARMv8	93MB	a72a0b036be4193a0214bca3fca4c5468a38a4ccef098c909f7ce8bfe08567c48
go1.15.5.linux-armv6l.tar.gz	Archive	Linux	ARMv6	93MB	5ea645662043efa85d4a99238c7f23866eafda915a5348736a631bc283c0238a
go1.15.5.windows-386.zip	Archive	Windows	x86	113MB	8b12436c7e3482b3c97172e4d26afa35ac60a5621ff4a5f8a08386505ab9c
go1.15.5.windows-386.msi	Installer	Windows	x86	98MB	28fa444b732033302d14e10a2cfe133f435a02c0089f5b1b0d18572ab39aee2a
go1.15.5.windows-amd64.zip	Archive	Windows	x86-64	132MB	1a24b3a200201a74ba25e4134fbae467750a834a84e9c7789a9fc13248c5507
go1.15.5.windows-amd64.msi	Installer	Windows	x86-64	115MB	c20eb1e9e2976b59a379096867aef7c632f3a3a3045b0c666ff037463e7fa0f65f
Other Ports					
go1.15.5.freebsd-386.tar.gz	Archive	FreeBSD	x86	96MB	44e6b4a4b22a00ab9048fea6b44aee8626345cbcf5a63325872b357d8eb11baefb
go1.15.5.freebsd-amd64.tar.gz	Archive	FreeBSD	x86-64	115MB	aa25a356167b6f30468205c01b88f8319f07a6b127405a3f7bb6af9ab1ca27b
go1.15.5.linux-ppc64le.tar.gz	Archive	Linux	ppc64le	92MB	86f209a66ba80274a07c4fa464c72b5b21dda1bf18347b93d13b98cbf546ecb1
go1.15.5.linux-s390x.tar.gz	Archive	Linux	s390x	96MB	c0b572c26bbef47d9da09f4609dcdfb8948424b8a0bb2298baacc9da48b18482ed

go1.14.12 ▾

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`

```
--2020-11-17 10:55:06-- https://golang.google.cn/dl/go1.15.5.linux-amd64.tar.gz
Resolving golang.google.cn (golang.google.cn)... 203.208.43.98
Connecting to golang.google.cn (golang.google.cn)|203.208.43.98|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://dl.google.com/go/go1.15.5.linux-amd64.tar.gz [following]
--2020-11-17 10:55:06-- https://dl.google.com/go/go1.15.5.linux-amd64.tar.gz
Resolving dl.google.com (dl.google.com)... 203.208.50.65
Connecting to dl.google.com (dl.google.com)|203.208.50.65|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 120900442 (115M) [application/octet-stream]
Saving to: 'go1.15.5.linux-amd64.tar.gz'

100%[=====>] 120,900,442 10.2MB/s in 11s

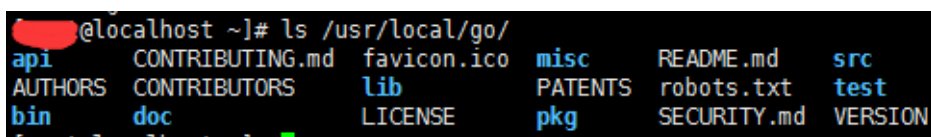
2020-11-17 10:55:17 (10.4 MB/s) - 'go1.15.5.linux-amd64.tar.gz' saved [120900442/120900442]
```

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



```

[go@localhost ~]$ ls /usr/local/go/
api      CONTRIBUTING.md  favicon.ico  misc      README.md  src
AUTHORS  CONTRIBUTORS     lib          PATENTS   robots.txt test
bin      doc              LICENSE      pkg       SECURITY.md VERSION

```

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

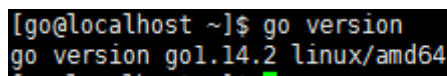
✧ 导入

\$ 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



```

[go@localhost ~]$ go version
go version go1.14.2 linux/amd64

```

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:$GOPATH/bin
export GOPROXY=https://goproxy.cn #代理
export GOMODCACHE=$GOPATH/pkg/mod
```

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
```

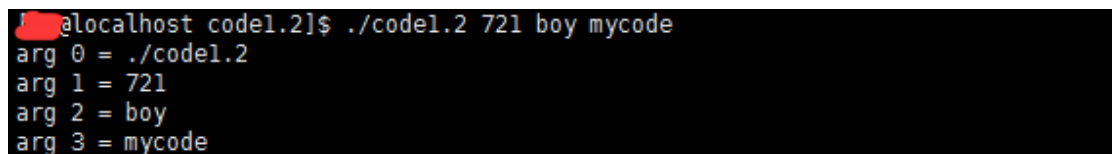
Build: 生成可执行文件

```
$ 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:



```

@localhost code1.2]$ ./code1.2 721 boy mycode
arg 0 = ./code1.2
arg 1 = 721
arg 2 = boy
arg 3 = mycode

```

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 对话小程序

CODE1-3

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     for {
9         msg := ""
10        fmt.Print("问:")
11        fmt.Scan(&msg)
12        fmt.Println("答:", handle(msg))
13        if msg == "q" || msg == "Q" {
14            return
15        }
16    }
17 }
18
19 func handle(msg string) string {
20     ni := []rune("您你我")
21     ma := []rune("吗")
22     msgArr := []rune(msg)
23     for i, v := range msgArr {
```



```

24         if v == ni[0] || v == ni[1] {
25             msgArr[i] = ni[2]
26         }
27
28         if v == ma[0] {
29             msgArr = msgArr[:i]
30             break
31         }
32     }
33     msg = string(msgArr)
34     return msg
35 }

```

运行\$ go run main.go

问:你好吗?

答:我好

问:你吃饭了吗

答:我吃饭了

问:你是小傻瓜吗

答:我是小傻瓜

主要使用 `fmt` 以及标准输入输出终端

从标准输入读取 `fmt.Scan(*string)`，将输入内容读取到字符串上

使用 `fmt.Print` 和 `fmt.Println` 打印字符串到终端

`Handle` 函数将问句处理成答句子

利用 `go` 特性语法可以将字符串 `string` 和 `[]rune` 之间进行转换,把字符串转成数组，再处理。因字符串是只读的

1.5 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++ {

}
// 并发
go 函数()
```

...

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。

1.6 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。

化繁为简，合理的利用封装，外部可见的一定是像 1+1 一样。

编程思想追求工业化，流水线式，设计模式也是辅助实现此过程。

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，那自己也就呵呵（玩完）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。

对于老程序员一般都会先看看有哪些内置关键字等，这里我们就是简单列举，不做过多介绍。以后会逐个涉及到的。

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，详细的介绍见十一章。

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

go 源码文件的后缀是 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 命名也同 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

var 语法

var variable

var variable go 类型/int/string/等（包含自定义类型）

以上两种写法都是正确的声明变量的方式；Go 中的变量再使用前必须声明，这也算是强类型语言的一种标志。

变量在声明时还可以直接赋值，就是等号后面直接跟表达式；但 go 还提供一个简短声明且赋值的写法，同时也省略了 var 关键字在 2.5 变量小节中会有介绍。

const 语法

```
const variable = value
```

```
const variable go 类型 = value
```

常量的声明一般都是直接赋值，只有一些特定的语法糖会省略，但实际也是赋值的；第一种情况使用的较多，一般声明常量为了使用方便都不会去设置类型。

type 语法

```
type variable Go 类型/结构体 struct
```

```
type variable = Go 类型/结构体 struct/函数
```

以上两种情况有很大的区别：

第一种是声明另一种类型，也可以叫自定义类型；一般可理解成在这个包中多了一个新的类型。

第二种是仅仅是别称，`variable` 等价于等号后面的类型或者函数。

func 语法

声明函数本节不做重点细节介绍，详细请看第六章

```
func variable([ar1 go 类型, ...])(rtn, ...)
```

go 函数声明关键字是 `func`，不像其它语言用 `function` 全写；`func` 后面的一般可以称为函数标签；这里的 `variable` 一般叫做函数名；函数是可以支持多参数，变参，不同类型的参数，也可以支持多个返回值，无论函数还是返回值的个数都可以是 0 个。

最简短的写法就是 `func variable()`。

最特殊的就是 `func main() {}`，这是主函数，一个项目的应用运行都是从这个函数作为入口。

还有一个比较特殊的就是 `func init()`，这在 go 包中是加载时候执行的函数，也就是 `import` 语句引起的，在 `main` 函数前执行。

使用示例：

```

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:=5.0, C:=%f 5 78.5
```

有注释的行，结合注释和前面介绍的概念就可以明白意思

引入 `fmt` 包 `import "fmt"` 行；因在函数内使用了 `fmt` 的导出函数

使用 `fmt.Println` 包的导出函数打印字符串到标准输出终端上

至于 `main` 函数，声明函数哪里我们介绍的很清楚了。

这里我们没有一一列举各个声明的所有情况，到这就可以了，详细的后面会根据示例使用说明的。

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          ✗
```

变量的声明和赋值与其它编程语言的差异比较大，所以赋值和声明放在一起才能说的清楚。

上面的语句中打对号的正确，打差的是不能编译通过。

变量还可以根据首字母大写可以包导出可见，否则为包私有的变量，这一点在第十一章中会使用 and 说明，这里略过即可。

默认值即是零值

简单类型

<code>var s string</code>	零值=""	字符串
<code>var s int/float</code>	零值=0	整形/浮点型
<code>var s bool</code>	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)

<code>var s []byte/[]string/...</code>	零值= <code>nil</code>	切片
<code>var s map[string]string</code>	零值= <code>nil</code>	<code>map</code>
<code>var s *int</code>	零值= <code>nil</code>	指针
<code>var s chan int</code>	零值= <code>nil</code>	<code>chan</code>
<code>var f func()</code>	零值= <code>nil</code>	函数变量

同行多个变量声明

使用 `var` 声明的两种用法，结合了同一行可声明多个变量；同一行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，简单的先当数组理解，后续有详细使用介绍

```
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
```

`=` 它前面的变量必须是已经声明过

`:=` 它前面的变量一定没有被声明过

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
```

指针相关符号：

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
```

先初始化 `x` 等于 1

用 `p` 变量赋值 `x` 的内存地址，一般也可以叫引用，在 `go` 中多数还是叫指针

`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`。

示例说明：

```
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，所以这个是很很少用的。

每次调用 `new` 函数都是返回一个新的变量的地址，对应的变量叫指针。

指针 0 值：

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

此种情况多数 `p1` 和 `p2` 两个指针的是不同的

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，运行时长。

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

```
var glavar *int

func fHeap() {
```

```
var x int
x = 1
glavar = &x
}
```

注：fHeap 的局部变量 x 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。

函数内变量大多是优先分配在栈上的。

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

等号赋值 DEMO

```
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

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

CODE:

```
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。

CODE:

```
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** 函数作用域内赋值 **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，8 个 bit 组成一个 byte。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。进一步将这些数字组织在一起，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。下表列举出 go 中的基础类型。

类型	长度/字节	默认值	说明
bool	1	false	一个字节内非 0: true; 否则: false
byte	1	0	uint8
rune	4	0	Unicode pointer, type int32
Int,uint	4/8	0	32/64 位机 = 4*8bit/8*8bit/
Int8,uint8	1	0	-128~127, 0~255
Int16,uint16	2	0	-32768~32767, 0~65535
Int32,uint32	4	0	-21 亿~21 亿, 0~42 亿
Int64,uint64	8	0	
float32	4	0.0	
float64	8	0.0	
complex32	4		
complex64	8		
complex128	16		
uintptr	4/8		内存指针数字
array			值类型，内存一段数据
struct			默认值类型，&struct 指针
string		""	值类型，内部是用长度和字符串首地址
slice		nil	引用类型
map		nil	应用类型
channel		nil	引用类型
interface		nil	接口
func		nil	函数定义关键字

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。大小单位是 `byte`，`size(int8)=size(byte)`。整数的 `int` 类型是根据机器操作系统位数和 `cpu` 决定。近些年服务器常用的 AMD64 架构，也就是一个字等于 `8byte`（字节），也就是 `64` 位。

整数类型

有符号	<code>int8</code>	<code>int16</code>	<code>int32</code>	<code>int64</code>	
无符号	<code>uint8</code>	<code>uint16</code>	<code>uint32</code>	<code>uint64</code>	
位大小	<code>8</code>	<code>16</code>	<code>32</code>	<code>64</code>	bit(单位)
大小	<code>1</code>	<code>2</code>	<code>4</code>	<code>8</code>	byte

`Int` 根据计算位数有关系在 AMD64 是 `int64`

`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。可理解 `uintptr` 指针的地址，可进行指针计算。

声明与定义

通用声明定义语法，再简单整理下

```
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
```

CODE:

下面的代码又多了一种，声明赋值的格式，也是正确的，注意看注释解释。

```
var i int = 1           // 声明 i 且赋值 1
k := 2                  // 简写赋值整数 2，不能是全局变量
var a, b int = 1, 2     // 多远定义与赋值

// 这样写也可以，就是减少不必要的 var 关键字，让代码清爽
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。即使是用 `type` 自定义的整数类型也一样。下面的是通用格式：

整数类型(整数表达式/浮点型表达式)

CODE:

```
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
```

第二行就用了类型 (`int`) 强制转换

对于最后一行，也是从网上看到的才实验一下确认执行不通过。这也是我们实践中需要注意的一个点。

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。

溢出 DEMO:

溢出时，所得到的结果就不是那么好确认了，虽然可以算出来，可大多情况是没有啥意义的；一般**位移溢出**的手段是比较常用的，如清零补位。

```
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

1 size (byte)

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

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:abcbc
s += s1                    // output:abcbabc
```

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, go!ang
```

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)
```

3.4.3 字符串底层实现 · 深

源码

位置：src/runtime/string.go:stringStruct 定义了 string 的数据结构

```
type stringStruct struct {
    str unsafe.Pointer
    len int
}
```

源码中可以发现结构体是私有的

所以很多介绍源码都用 reflect 反射包中 reflect.StringHeader 相同的结构体。

对应源码反射包结构体

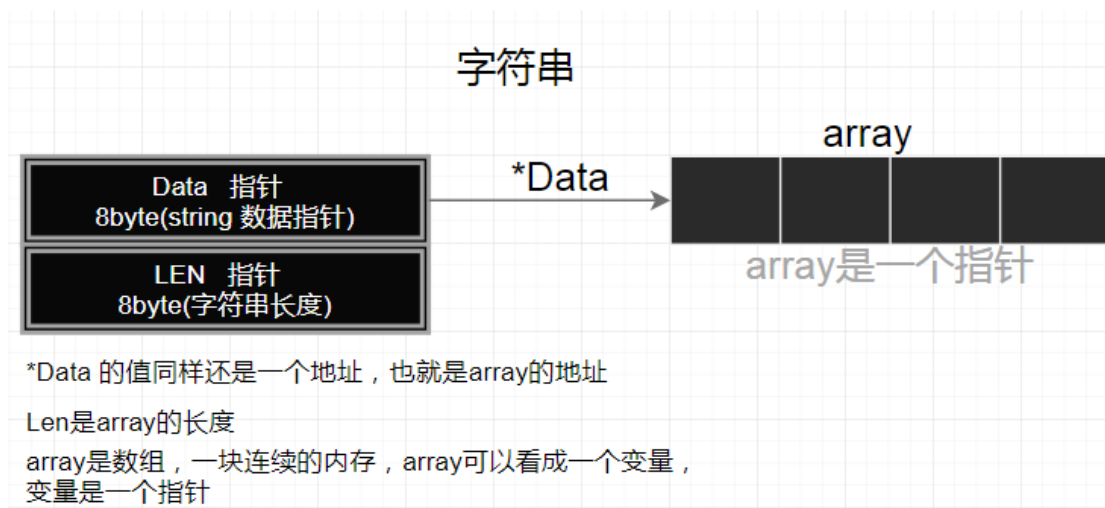
```
type StringHeader struct {
    Data uintptr
    Len int
}
```

猛然间看见确实不一样，本质是一样的。

2 个结构体第二个字段相同，不用多说

2 个结构体第一个字段，类型虽然不同，但却可以直接转换，暂时理解成一个东西就行了

内存结构体示意图



Go 字符串用常量值赋值的，数据多存于 rodata 中，不能修改

直接内存地址访问字符串

直接使用地址访问内存数据

```

1   s := "0"
    // 获取数据指针
2   s_d_ptr := (*int64)(unsafe.Pointer(&s))
    // 数据指针中存放的是一个数组，可以获取字符串第一个 byte 等价 s[0]
3   s_d := (*byte)(unsafe.Pointer(uintptr(*s_d_ptr)))
    // 报错: 不能被修改 unexpected fault address 0x6b82c4
4   *s_d = b

```

*s_d 就是此字符串第一个字节的值，s_d 是内存地址-指针，相当于数组的变量，只读的指针，写操作是不安全的

变量 s_d_ptr 是 go string 的数据指针，相当于是存储数组的引用指针。

此处示例仅仅是让我们理解字符串内存结构，并没有多少应用价值

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。

CODE：简单的

```
const pi = 3.14159
```

CODE: 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，毕竟 go 是一个强类型静态语言。

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。

CODE: 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)
}

```

浮点强制转整形，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。此小节内容我们将（数字+布尔）称为 T 类型；因此本节内容主要就是 T 与字符串的转换关系了。

```

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](https://golang.org/pkg/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"
    "strconv"
)

// 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 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:

```

// 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 $2 \leq \text{base} \leq 36$ 。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。新手可以先跳过此节内容，通读后回头再看。

byte 转 string

```
string(byte)
```

```
s := string('a')
```

单向的，反过来用 `byte(string)` 则编译报错

字节简书

字节一般虽然用单引号表示，但是实际是一个 `int8` 的数字，是可以算数运算的。同理字节的零值是 `0` 而不是 `''`。

CODE3.3:

```
/** 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"
)

// 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))
}
```

Package 前面的注释可以在 `github` 上找到源码位置

函数 `byteCal` 就是将大写的 `H` 字节转小写的 `h` 字节

将大写的 `H` 字节值放在 `b1` 变量上

将 `b1` 的值加 `32` 放在 `b2` 变量上，大写 `H` 转小写 `h`

`Fmt.Printf` 上的注释也就是输出的是内容。字节我们不做过多的介绍，切实感受下就好了

string 与 []byte

```
[]byte(string)
```

```
String([]byte)
```


双向的

切片这里有点突兀，在切片章节中会讲到。这里会涉及到主要是因为它们的内存结构布局比较相似。

```
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。

学习了 unsafe 包后，便可以清晰的理解，写出类似的高性能函数。

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
)
```

自定义类型，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 表达式与控制流

为了复合类型可以更好的被理解，会有不少例子，使用到此章内容，因此在复合类型章节前插队了。

本章主要介绍，运算符，以及运算符对应的表达式，再加上条件跳转等基础控制流语言。

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	

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

4.2 流程语句

每个语言都会有自己独特的控制流语句, 就是汇编 B 类语言同样是有。go 的控制流和其它高级语言很不相同, 整体来说比较唯一, 比较简化写法且强化功能。

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 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 $i < \text{len}(s)$ ，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“我想要的”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
}
```

特性：

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。

因 value 是值类型，且只循环体内，只需要声明一次，则不能使用&取地址符，即使用，也只能指向循环体内的最后一个值。达不到想要的效果。

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

```

感觉是想要的挺好哈

这里重点就是 `range` 对应字符串遍历比较特殊，`value` 会对应一个 `rune` 类型；而不是一个 `byte`。

`rune` 类型是 4byte，因此他有足够的长度表示中文，且可以把字符串按有效想要的 `n` 个 `byte` 长度分割好。

如果想按 `byte` 轮询，不使用 `range` 而用计数器的 `for`；或先将字符串转换成 `[]byte` 切片。

省略的可以用符号 “`_`” 代替

```

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])  
}  
}
```

循环体内用引用需要小心

当然这个并不仅仅是 go 语言是这样

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。

Break 和 continue 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 label 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]`。

虽然是一个指针但确是个值类型的。

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])
}

```

数组访问，修改，用数组变量和下标

下标是用[]访问，**array[下标索引]**

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 $0 \leq i \leq j \leq \text{cap}(s)$ 。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

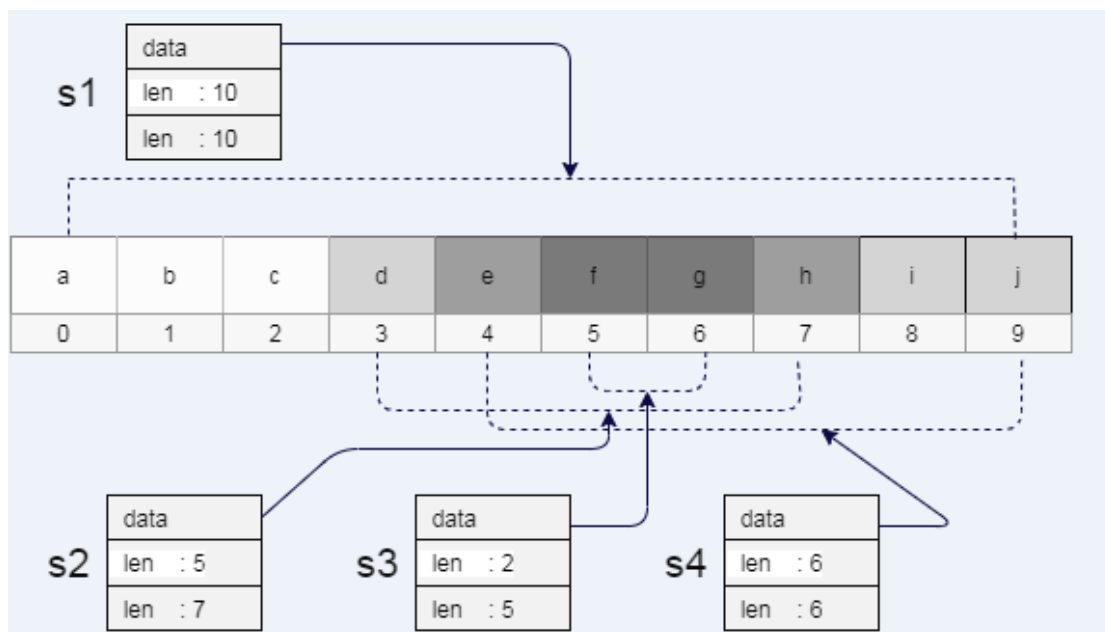


图 5-1

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，需要切片参数追加（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。

逃逸命令

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
```

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")
}
```

函数 `how` 返回的切片 `s` 是引用类型, 局部变量, 这是正常的外部有接收 `how` 的引用类型返回值, 则会发生逃逸

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在意。

5.2.5 切片内存布局 · 深

底层结构体

在反射包中切片结构体

```
type SliceHeader struct {
    Data uintptr // 数据指针, 指向内存数组
    Len  int      // 有值的切片长度
    Cap  int      // 切片容量值, 理解成数组的长度
}
```

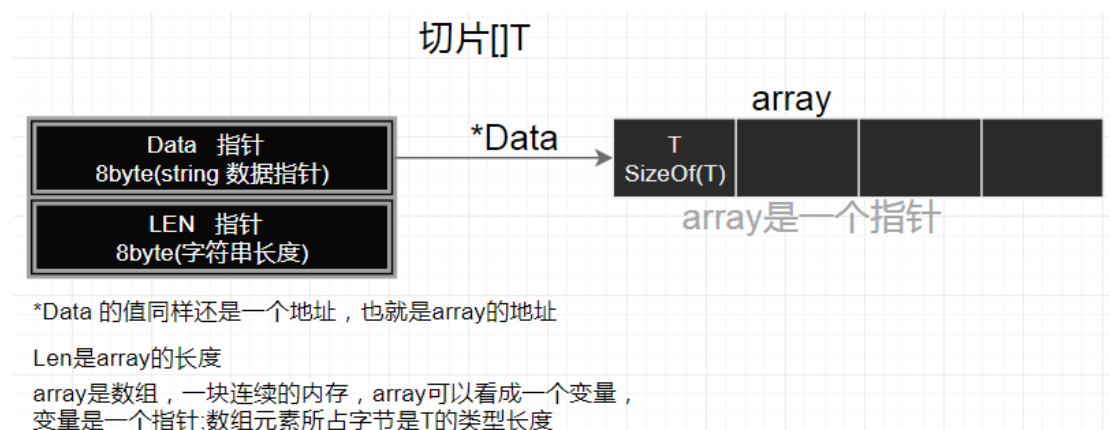
与字符串的反射结构体对比, 仅仅是多了个 `Cap` 容器大小的字段

字段 `Data` 是一个指针, 指针的值是内存数组

内存数组是同 `C` 语言的一样, 同样是一个指针; 所以 `Data` 可理解成指针的指针。

从以上数据类型分析可以看出字符串与 `[]byte` 切片的转换可以是指针的变动, 而无需对数据内容进行改变。在字符串与 `[]byte` 小节中有相关的函数。

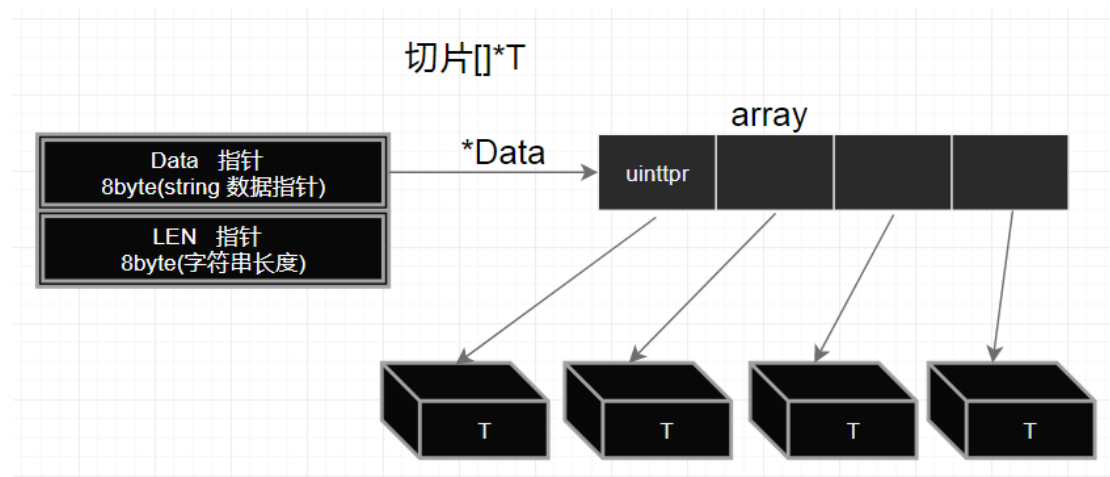
内存示意图



通过上图分析；AMD64 环境下：

1. []byte 切片，内存数组是一个 byte 类型的数组，每个元素大小是 1byte
2. []int32 切片，内存数组是一个 int32 的数组，每个元素大小是 4byte
3. []int64 切片，内存数组是一个 int64 的数组，每个元素大小是 8byte
4. []string 切片，内存数组是一个 go string 的数组，每个元素的大小是 16byte；前面字符串章节中字符串底层是一个结构体，大小是 16 字节。
5. []*string 切片，内存数组中每个元素是指针，那么每个元素大小是 8byte；其实只要是*T 类型的切片每个元素的大小都是 8byte。上面这个图还表示不了这种类型的切片。针对这种情况我们再画下示意图

指针切片[]*T 示意图



切片[]interface{}

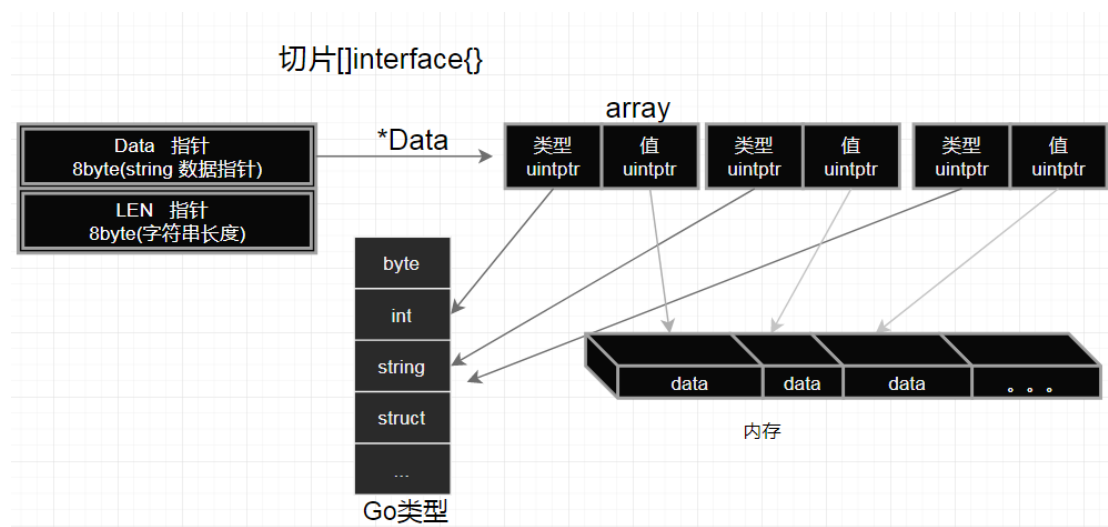
此部分我也还没有确认到底是如何的。

Interface {} 类型是空接口，所有类型都实现了空接口。不光指针，还有值同样实现了。

一个接口值由两个字（32 位机器一个字是 32 bits，64 位机器一个字是 64

bits) 组成; 一个字用于指向该值底层类型的方法表, 另一个字用于指向实际数据。

因此我只给出个内存布局参考图, 是否正确还待商榷



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 过大仍保持查询性能。

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，如描述人的个人信息，企业的组成，订单等。Go 没有 `class`。

5.4.1 声明与定义

语法 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
}
```

初始化

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

结构体可以定义的风格有很多

```

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 strings, 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

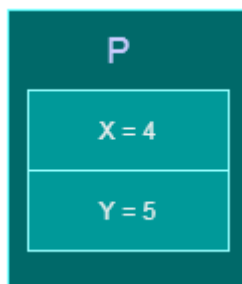


图 5-4-1

指针对象

```
newPtr := new(Point)
refPtr := &Point{}
```

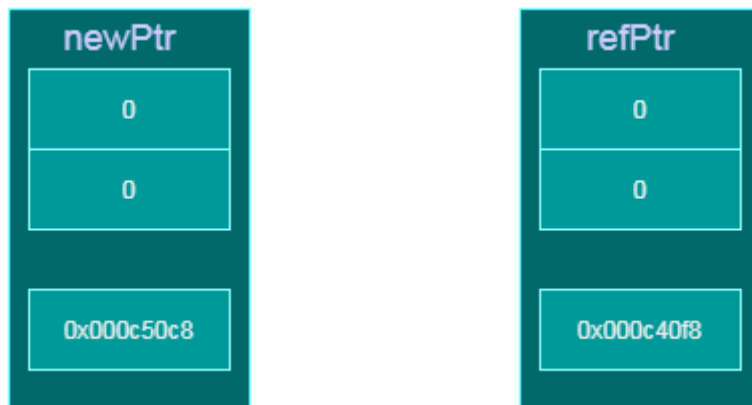


图 5-4-2

```
p := Point{4, 5}
refPtr := &p      // *refPtr == p
```

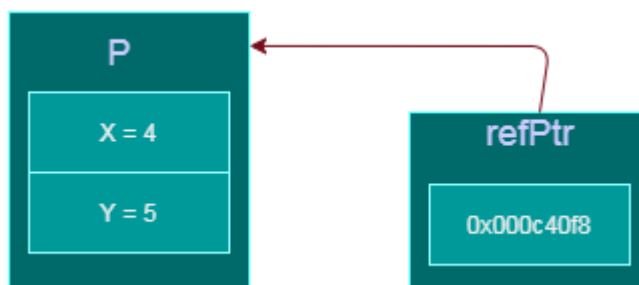


图 5-4-3

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。因内存对齐，导致内存空洞。

```
unsafe.Sizeof(Point{0, 0})
```

5.4.6 内存对齐

DEMO-1

```
import (
    "fmt"
    "unsafe"
)

type SizeStruct struct {
    b  byte
    in int64
}

func demoOne() {
    s := SizeStruct{'1', 23}
    fmt.Println("SizeStruct.b:", unsafe.Sizeof(s.b),
        "SizeStruct.in:", unsafe.Sizeof(s.in), "SizeStruct:", unsafe.Sizeof(s) )
    fmt.Println("SizeStruct Align:", unsafe.Alignof(s))
}
```

Output:

```
SizeStruct.b: 1 SizeStruct.in: 8 SizeStruct: 16
```

```
SizeStruct Align: 8
```

首先引入 `unsafe` 包

结构体大小并不是所有字段 `Size` 之和； 并不是 $1+8 = 9$ 而是 16

这是因为有内存对齐

`unsafe.Alignof` 返回的就是结构体内存对齐的值，也就结构体是这个值的倍数；现在的值是 8 那么结构体的大小一定是 $n*8$

内存对齐

为了快速访问内存数据，cpu 需要尽可能的少访问内存，对齐内存是一个很好的策略。对齐内存的值一般是 1, 2, 4, 8 等；结构体的大小一般是这些数字的倍

数。

内存对齐会产生内存空洞，其实也就是冗余的，没有数据，访问不到的内存。所以结构体实际大小一般是每个字段大小和这些内存空洞加在一起的大小。对齐保证程序兼容性，既能在 64 下编译成功，也能在 32 位机器上编译成功。

SizeStruct 的内存图形

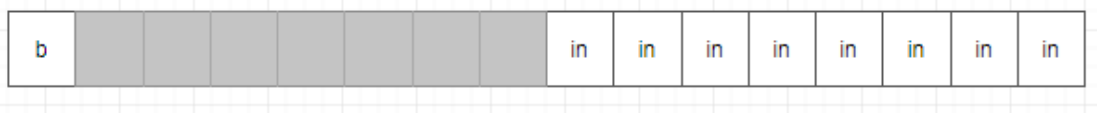


图 5-4-4

DEMO-2

```
type Size2 struct {
    A byte
    B int64
    C int16
    D int32
}

func demoSize2() {
    s := Size2{}
    fmt.Println(unsafe.Sizeof(s))
    fmt.Println(
        unsafe.Offsetof(s.A),
        unsafe.Offsetof(s.B),
        unsafe.Offsetof(s.C),
        unsafe.Offsetof(s.D))
}
```

Output:

```
24
0 8 16 20
```

Unsafe.Offsetof: 结构体起始地址到字段的起始地址的字节数

内存图

A							
B	B	B	B	B	B	B	B
C	C			D	D	D	D

图 5-4-5

从上图和数据中可以分析出：

内存分布是与结构体字段顺序是关系的。

Size2 结构体的对齐值是 8

若多个字节之和小于 8，则他们可以在一次对齐内排列

先申请 8 字节内存 A 类型是 byte 占一个字节占第一个字节

B 是 8 字节与对齐值相等；申请 8 字节内存，B 会独占 8 字节； A 与 B 之间会产生 7 个空洞字节。

再申请 8 字节；C 大小是 2 字节，

D 大小是 4 字节，D 与 C 对齐会产生 2 个空洞字节。D 和 C 在一个对齐内存。

DEMO-3

我们将 DEMO 稍微改动下；将 A 和 B 两个字段互换一下

```
type Size2 struct {
    B int64
    A byte
    C int16
    D int32
}

func demoSize2() {
    s := Size2{}
    fmt.Println(unsafe.Sizeof(s))
    fmt.Println(
        unsafe.Offsetof(s.B),
        unsafe.Offsetof(s.A),
        unsafe.Offsetof(s.C),
        unsafe.Offsetof(s.D))
}
```

Output:

16

0 8 10 12

结构体大小少变为了 16 字节

对齐值仍然是 8

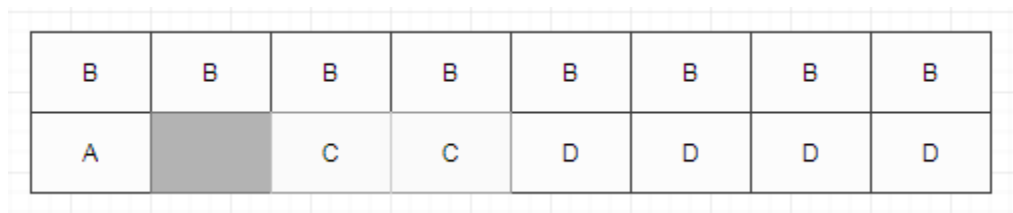


图 5-4-6

总结：

前提这是 64 位机器的结果，32 位或许不一样

64 位机器一个字等于 8 字节。因此基本类型最大值就是 8 字节。内存对齐与这个有着直接的关系。

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`。Go 函数支持多返回值，多数其它高级语言并不支持这个。一般常用引用参数替代。还有很多特殊内置函数，函数特性等都在本章详细说一下。

Go 中方法并没有太多的特性和注意事项，也仅仅总结出三个小节内容，就没有再单独列一章出来。

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
}
```

以上两个函数的作用是相同的

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

定义时参数顺序，决定了在调用时同样要保持一致；其实返回值，也是一样需要遵守这个规则。

```
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

函数可以赋值给变量

```
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：y 个 x 相乘。

```
// 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 x_0 ，其函数值 $f(x_0)$ 由 $f(f(x_0))$ 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)
}
```

缺点，栈空间使用容易过多；容易发生 overflow。

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，此种算法不会发生栈溢出。

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 $n*(n-1)*(n-2) \dots 2*1$

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))
}

```

通过注释中的输出的值，我们可以明显看到，sum 一直在累加；

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("/", funcName)`

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

还有大多数的 go 语言实现的 web 框架，给路由绑定控制器也是作为回调传递的。

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；同样可以中断程序。

```
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 防止线上 bug。

用法：手动上使用 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

通过标准终端输入字符串 9 来，调用 panic 函数。其实 panic 调用没什么，关键是可以根据这段小程序熟悉，bufio 以及 os 标准库包。

```
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")
    }
}
```

第一行用 `bufio.NewReader` 创建一个 reader，且与标准输入绑定

其中给 `signal` 变量赋值的 `in.ReadRune()` 是具有 `epoll wait` 的效果(IO 阻塞)，接到输入字符串继续向下执行

接下来就用 `switch` 语句判断执行分支，是否执行到 `panic`

信号量处理

简单解释下信号量，操作系统和运行中的应用程序通信的信号。执行 `kill 9 pid` 等，OS 根据 `pid` 找到应用程序且发送自杀命令到这个应用程序。目的是应用程序需要根据这些信号做一系列处理。

```
// 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()  // 注册太晚了，不太安全
// 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

同一个函数出现多个 `defer` 也不是什么过激的事情，既然合理就需要知道他们是如何执行的

```

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:

```
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了的了的值。

Defer 和参数

请看下面代码的执行结果

```
func deferRun(){
    a := 1
    b := 2
    defer deferCall(a, deferCall(a,b))
    a = 0
    defer deferCall(a, deferCall(a,b))
}
func deferCall(x,y int) int{
    fmt.Println(x,y,x+y)
    return x+y
}
```

Output:

```
1 2 3    // b:= 2 下面 deferCall 函数参数里的 deferCall (1, 2) 所执行的结果
0 2 2
0 2 2
1 3 4
```

1 defer 命令是先入后执行，栈调用。

2 defer 执行的函数的参数是立即就执行的。

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（goroutine, 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 也是可以捕捉到的。

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，只能声明函数签名；就是函数不能有实体。

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。

方法声明示例

```
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

我们不能在系统给的内置类型上，再去多定义方法，这是编译不通过的

```
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

我们在用这个自定义的 NewString 实现个 Split 方法对应下标准库中的 Split 函数。顺便感受下 strings 库中的函数运行线路。

```
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
// output: 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（READ）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)，重名的话就只能访问 Teacher 的了。

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，比较松散的多态。可以说与嵌入式无直接关联，但还是可以利用的，主要看这 2 个结构体所有实现的方法是否完成接口定义的方法。

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 Person 的方法或成员，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

在同一个包的继承行，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() 执行的是定义在 Teacher 接收器上的方法。

Person 上的 PrintInfo() 直接被覆盖掉，不会执行

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。

关于继承：我们看段代码，先在 Person 中定义一个简单的打印流程

```
// 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。

6.10 方法与函数之间的差异总结

方法	函数
签名上包含接收器，方法是嵌入在接收器上的函数	无接收器，都是普通参数
一个包中在不同的接收器上可以定义多个相同的函数名的方法	一个包中，一个函数名只能有一个

方法需要注意区分接收器，是指针还是值类型。

6.11 函数类型 · 精

其实在 Go 中函数也是有类型的，这一点熟悉好了后，可以明显感觉到用 go 写程序时，更加灵活，精简；读源码时也更轻松些。那我们就先看下到底什么是函数的类型。

答：

最基本的函数类型就是函数标签去掉函数名；

要是去掉具体的参数名和返回值，只留下类型更好；显得函数类型唯一

0 参-函数

无参数无返回值我们才会称为 0 参函数，在汇编中返回值和参数是一样的结构。

```
func anEmptyFunction (){
    fmt.Println("print empty")
    // TODO somethings
}
```

依据定义我们可以得到基础类型如下

```
func ()
```

我们先将函数放在一个 interface{} 类型上

```
var fn interface{} = anEmptyFunction
f2, e2 := fn.(func ())
if e2 {
    f2()
}
```

上面代码是可以通过的，e2 的值会是 true
就是一个类型断言

多参-函数

还是根据汇编参数内存分配等，参数和返回值是无分别的

```
func forFuncType(str string, i int, many *int64, bs []byte)(int, error) {
```

```
// TODO
}
```

根据上面的步骤

```
var fn interface{} = forFuncType
f2, e2 := fn.(func( string, int, *int64, []byte)(int, error) )
if e2 {
    f2()
}
```

同样可以通过；`e2=true`
还是可以断言的

无参数的函数，还不觉得什么，要是参数，返回值多了，那会很麻烦且容易出错；对了你或许也想到为函数类型定义别名；前面我们不是讲述了 `type` 吗？

自定义·函数类型

```
type manyArgs func( string, int, *int64, []byte)(int, error)
```

在试试

```
var fn interface{} = forFuncType
f2, e2 := fn.(manyArgs )
if e2 {
    f2()
}
```

无法执行到 `f2`

无法断言的，这点与普通的基础类型是大为不同的
但也不是一无是处，函数格式一样可以强制类型转换

强制转换

```
manyArgs(forFuncType)           // OK
manyArgs(anEmptyFunction )      // panic
```

这是可以正确执行的

若有参数是 `manyArgs` 类型的，可以将符合此类型的函数强制转换过去，传递过去

但函数标签不符合这是一点情面不会给，直接中断 `panic`。

函数别称

```
type manyArgsAlias = func( string, int, *int64, []byte)(int, error)
```

```
var fn interface{} = forFuncType
f2, e2 := fn.(manyArgsAlias )
if e2 {
    f2()
}
```

可以执行到 f2

断言成功，即使不成功，也不会 panic，与正常断言无区别

多用在函数注册，注入等实现上比较有用，可以美化代码

注

以上内容，我没有在书本上，明确的学习过，都是从各种源码中总结出来的。若有错误希望指正。希望对您也有帮助吧。不理解以上内容也是没关系的，我写过的一些包并没有用到此小节的特性也是能够完美胜任的。

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/aixg1/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.ReadWriter`, 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

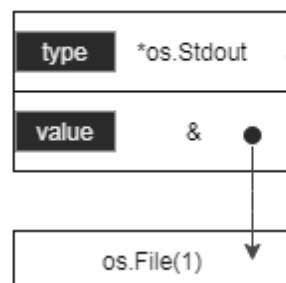
我们定义的 `io.Writer`，而在运行过程中它的类型是根据我们赋的值而决定的。所以我们更加可以确定 `go` 接口类型是一种虚类型与其它类型实体不同。

接口 `io.Writer` 变量 `w` 的值变化示意图



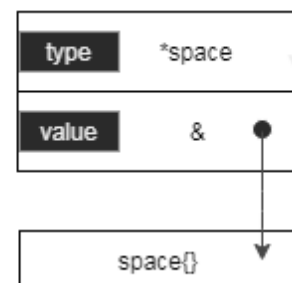
声明 `os.Writer`

图7-1



`w`赋值 `os.Stdout`

图7-2



`w`赋值 `&space{}`

图7-3

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

type	*bytes.Buffer
value	nil

图7-4

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` 两个变量，`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.Errno(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 project 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"`。