尚硅谷 Go 语言核心编程

尚硅谷-韩顺平

第 1 章 GOLANG 开山篇 1
1.1 GOLANG 的学习方向 1
1.2 GOLANG 的应用领域 1
1.2.1 区块链的应用开发 1
1.2.2 后台的服务应用 2
1.2.3 云计算/云服务后台应用 2
1.3 学习方法的介绍 3
1.4 讲课的方式的说明 4
第 2 章 GOLANG 的概述 5
2.1 什么是程序 5
2.2 GO 语言的诞生小故事 5
2.2.1 Go 语言的核心开发团队-三个大牛 6
2.2.2 Google 创造 Golang 的原因 6
2.2.3 Golang 的发展历程 6
2.3 GOLANG 的语言的特点 7
2.4 GOLANG 的开发工具的介绍 8
2.4.1 工具介绍 8
2.4.2 工具选择： 9
2.4.3 VSCode 的安装和使用 10
2.4.4 小结 14
2.5 WINDOWS 下搭建 GO 开发环境-安装和配置 SDK 15
2.5.1 介绍了 SDK 15
2.5.2 下载 SDK 工具包 15
2.5.3 windows 下安装 sdk 16
2.5.4 windows 下配置 Golang 环境变量： 17
2.6 LINUX 下搭建 GO 开发环境-安装和配置 SDK 20
2.6.1 Linux 下安装 SDK： 20
2.6.2 Linux 下配置 Golang 环境变量 21
2.7 MAC 下搭建 GO 开发环境-安装和配置 SDK 21
2.7.1 mac 下安装 Go 的 sdk 21
2.7.2 Mac 下配置 Golang 环境变量： 22
2.8 GO 语言快速开发入门 22
2.8.1 需求 22
2.8.2 开发的步骤 22
2.8.3 linux 下如何开发 Go 程序 24
2.8.4 Mac 下如何开发 Go 程序 25

2.8.5 go 语言的快速入门的课堂练习.....26
2.8.6 Golang 执行流程分析.....27
2.8.7 编译和运行说明.....27
2.8.8 Go 程序开发的注意事项.....28
2.9 GO 语言的转义字符(ESCAPE CHAR).....29
2.10 GOLANG 开发常见问题和解决方法.....31
2.10.1 文件名或者路径错误.....31
2.10.2 小结和提示.....31
2.11 注释(COMMENT).....32
2.11.1 介绍注释.....32
2.11.2 在 Golang 中注释有两种形式.....32
2.12 规范的代码风格.....33
2.12.1 正确的注释和注释风格：.....33
2.12.2 正确的缩进和空白.....33
2.13 GOLANG 官方编程指南.....35
2.14 GOLANG 标准库 API 文档.....36
2.15 DOS 的常用指令(了解).....37
2.15.1 dos 的基本介绍.....37
2.15.2 dos 的基本操作原理.....37
2.15.3 目录操作指令.....38
2.15.4 文件的操作.....40
2.15.5 其它指令.....41
2.15.6 综合案例.....41
2.16 课后练习题的评讲.....41
2.17 本章的知识回顾.....42
第 3 章 GOLANG 变量.....44
3.1 为什么需要变量.....44
3.1.1 一个程序就是一个世界.....44
3.1.2 变量是程序的基本组成单位.....44
3.2 变量的介绍.....45
3.2.1 变量的概念.....45
3.2.2 变量的使用步骤.....45
3.3 变量快速入门案例.....45
3.4 变量使用注意事项.....46
3.5 变量的声明，初始化和赋值.....49
3.6 程序中 +号的使用.....50
3.7 数据类型的基本介绍.....50
3.8 整数类型.....50

3.8.1 基本介绍.....50
3.8.2 案例演示.....51
3.8.3 整数的各个类型.....51
3.8.4 整型的使用细节.....52
3.9 小数类型/浮点型.....53
3.9.1 基本介绍.....53
3.9.2 案例演示.....53
3.9.3 小数类型分类.....53
3.9.4 浮点型使用细节.....55
3.10 字符类型.....55
3.10.1 基本介绍.....55
3.10.2 案例演示.....55
3.10.3 字符类型使用细节.....56
3.10.4 字符类型本质探讨.....57
3.11 布尔类型.....57
3.11.1 基本介绍.....57
3.11.2 案例演示.....57
3.12 STRING 类型.....58
3.12.1 基本介绍.....58
3.12.2 案例演示.....58
3.12.3 string 使用注意事项和细节.....58
3.13 基本数据类型的默认值.....60
3.13.1 基本介绍.....60
3.13.2 基本数据类型的默认值如下.....60
3.14 基本数据类型的相互转换.....60
3.14.1 基本介绍.....60
3.14.2 基本语法.....60
3.14.3 案例演示.....61
3.14.4 基本数据类型相互转换的注意事项.....61
3.14.5 课堂练习.....62
3.15 基本数据类型和 STRING 的转换.....63
3.15.1 基本介绍.....63
3.15.2 基本类型转 string 类型.....63
3.15.3 string 类型转基本数据类型.....65
3.15.4 string 转基本数据类型的注意事项.....66
3.16 指针.....66
3.16.1 基本介绍.....66
3.16.2 案例演示.....67
3.16.3 指针的课堂练习.....68

3.16.4 指针的使用细节.....68
3.17 值类型和引用类型.....68
3.17.1 值类型和引用类型的说明.....68
3.17.2 值类型和引用类型的使用特点.....69
3.18 标识符的命名规范.....69
3.18.1 标识符概念.....70
3.18.2 标识符的命名规则.....70
3.18.3 标识符的案例.....70
3.18.4 标识符命名注意事项.....71
3.19 系统保留关键字.....73
3.20 系统的预定义标识符.....73
第 4 章 运算符.....74
4.1 运算符的基本介绍.....74
4.2 算术运算符.....74
4.2.1 算术运算符的一览表.....74
4.2.2 案例演示.....74
4.2.3 算术运算符使用的注意事项.....76
4.2.4 课堂练习 1.....76
4.2.5 课堂练习 2.....77
4.3 关系运算符(比较运算符).....77
4.3.1 基本介绍.....77
4.3.2 关系运算符一览图.....78
4.3.3 案例演示.....78
4.3.4 关系运算符的细节说明.....78
4.4 逻辑运算符.....79
4.4.1 基本介绍.....79
4.4.2 逻辑运算的说明.....79
4.4.3 案例演示.....79
4.4.4 注意事项和细节说明.....80
4.5 赋值运算符.....80
4.5.1 基本的介绍.....80
4.5.2 赋值运算符的分类.....81
4.5.3 赋值运算的案例演示.....81
4.5.4 赋值运算符的特点.....82
4.5.5 面试题.....82
4.6 位运算符.....83
4.7 其它运算符说明.....83
4.7.1 课堂案例.....84

4.8 特别说明.....85
4.9 运算符的优先级.....86
4.9.1 运算符的优先级的一览表.....86
4.9.2 对上图的说明.....86
4.10 键盘输入语句.....87
4.10.1 介绍.....87
4.10.2 步骤 ：.....87
4.10.3 案例演示：.....88
4.11 进制.....89
4.11.1 进制的图示.....89
4.11.2 进制转换的介绍.....90
4.11.3 其它进制转十进制.....91
4.11.4 二进制如何转十进制.....91
4.11.5 八进制转换成十进制示例.....92
4.11.6 16 进制转成 10 进制.....92
4.11.7 其它进制转 10 进制的课堂练习.....92
4.11.8 十进制如何转成其它进制.....92
4.11.9 十进制如何转二进制.....93
4.11.10 十进制转成八进制.....93
4.11.11 十进制转十六进制.....94
4.11.12 课堂练习.....94
4.11.13 二进制转换成八进制、十六进制.....94
4.11.14 二进制转换成八进制.....95
4.11.15 二进制转成十六进制.....95
4.11.16 八进制、十六进制转成二进制.....95
4.11.17 八进制转换成二进制.....96
4.11.18 十六进制转成二进制.....96
4.12 位运算.....96
4.12.1 位运算的思考题.....96
4.12.2 二进制在运算中的说明.....97
4.12.3 原码、反码、补码.....98
4.12.4 位运算符和移位运算符.....98

第 5 章 程序流程控制.....101

5.1 程序流程控制介绍.....101
5.2 顺序控制.....101
5.2.1 顺序控制的一个流程图.....101
5.2.2 顺序控制举例和注意事项.....102
5.3 分支控制.....102

5.3.1 分支控制的基本介绍.....102
5.3.2 单分支控制.....103
5.3.3 双分支控制.....104
5.3.4 单分支和双分支的案例.....106
5.3.5 多分支控制.....109
5.3.6 嵌套分支.....114
5.4 SWITCH 分支控制.....116
5.4.1 基本的介绍.....116
5.4.2 基本语法.....116
5.4.3 switch 的流程图.....117
5.4.4 switch 快速入门案例.....118
5.4.5 switch 的使用的注意事项和细节.....118
5.4.6 switch 的课堂练习.....122
5.4.7 switch 和 if 的比较.....124
5.5 FOR 循环控制.....124
5.5.1 基本介绍.....124
5.5.2 一个实际的需求.....124
5.5.3 for 循环的基本语法.....125
5.5.4 for 循环执行流程分析.....126
5.5.5 for 循环的使用注意事项和细节讨论.....127
5.5.6 for 循环的课堂练习.....129
5.6 WHILE 和 DO..WHILE 的实现.....130
5.6.1 while 循环的实现.....130
5.6.2 do..while 的实现.....131
5.7 多重循环控制(重点，难点).....132
5.7.1 基本介绍.....132
5.7.2 应用案例.....133
5.8 跳转控制语句-BREAK.....137
5.8.1 看一个具体需求，引出 break.....137
5.8.2 break 的快速入门案例.....137
5.8.3 基本介绍：.....138
5.8.4 基本语法：.....138
5.8.5 以 for 循环使用 break 为例,画出示意图.....138
5.8.6 break 的注意事项和使用细节.....139
5.8.7 课堂练习.....140
5.9 跳转控制语句-CONTINUE.....141
5.9.1 基本介绍：.....141
5.9.2 基本语法：.....141
5.9.3 continue 流程图.....141

5.9.4 案例分析 continue 的使用.....142
5.9.5 continu 的课堂练习.....142
5.10 跳转控制语句-GOTO.....143
5.10.1 goto 基本介绍.....144
5.10.2 goto 基本语法.....144
5.10.3 goto 的流程图.....144
5.10.4 快速入门案例.....144
5.11 跳转控制语句-RETURN.....145
5.11.1 介绍：.....145
第 6 章 函数、包和错误处理.....146
6.1 为什么需要函数.....146
6.1.1 请大家完成这样一个需求:.....146
6.1.2 使用传统的方法解决.....146
6.2 函数的基本概念.....147
6.3 函数的基本语法.....147
6.4 快速入门案例.....147
6.5 包的引出.....148
6.6 包的原理图.....148
6.7 包的基本概念.....149
6.8 包的三大作用.....149
6.9 包的相关说明.....149
6.10 包使用的快速入门.....149
6.11 包使用的注意事项和细节讨论.....151
6.12 函数的调用机制.....153
6.12.1 通俗易懂的方式的理解.....153
6.12.2 函数-调用过程.....153
6.12.3 return 语句.....154
6.13 函数的递归调用.....155
6.13.1 基本介绍.....155
6.13.2 递归调用快速入门.....156
6.13.3 递归调用的总结.....157
6.13.4 递归课堂练习题.....158
6.14 函数使用的注意事项和细节讨论.....160
6.15 函数的课堂练习.....165
6.16 INIT 函数.....166
6.16.1 基本介绍.....166
6.16.2 案例说明：.....166
6.16.3 inti 函数的注意事项和细节.....167

6.17 匿名函数.....168
6.17.1 介绍.....168
6.17.2 匿名函数使用方式 1.....169
6.17.3 匿名函数使用方式 2.....169
6.17.4 全局匿名函数.....169
6.18 闭包.....170
6.18.1 介绍.....170
6.18.2 案例演示：.....170
6.18.3 闭包的最佳实践.....171
6.19 函数的 DEFER.....173
6.19.1 为什么需要 defer.....173
6.19.2 快速入门案例.....173
6.19.3 defer 的注意事项和细节.....173
6.19.4 defer 的最佳实践.....174
6.20 函数参数传递方式.....175
6.20.1 基本介绍.....175
6.20.2 两种传递方式.....175
6.20.3 值类型和引用类型.....175
6.20.4 值传递和引用传递使用特点.....175
6.21 变量作用域.....176
6.21.1 变量作用域的课堂练习.....178
6.22 函数课堂练习(综合).....179
6.23 字符串常用的系统函数.....180
6.24 时间和日期相关函数.....184
6.24.1 基本的介绍.....184
6.24.2 时间和日期的课堂练习.....188
6.25 内置函数.....188
6.25.1 说明：.....188
6.26 错误处理.....189
6.26.1 看一段代码，因此错误处理.....189
6.26.2 基本说明.....190
6.26.3 使用 defer+recover 来处理错误.....190
6.26.4 错误处理的好处.....191
6.26.5 自定义错误.....191
6.26.6 自定义错误的介绍.....191
6.26.7 案例说明.....191

第 7 章 数组与切片.....193

7.1 为什么需要数组.....193

7.2 数组介绍 193
7.3 数组的快速入门 194
7.4 数组定义和内存布局 194
7.5 数组的使用 195
7.6 数组的遍历 196
7.6.1 方式 1-常规遍历: 196
7.6.2 方式 2-for-range 结构遍历 196
7.7 数组使用的注意事项和细节 197
7.8 数组的应用案例 200
7.9 为什么需要切片 202
7.10 切片的基本介绍 202
7.11 快速入门 203
7.12 切片在内存中形式(重要) 203
7.13 切片的使用 204
7.14 切片的遍历 206
7.15 切片的使用的注意事项和细节讨论 206
7.16 STRING 和 SLICE 209
7.17 切片的课堂练习题 211

第 8 章 排序和查找 213

8.1 排序的基本介绍 213
8.2 冒泡排序的思路分析 213
8.3 冒泡排序实现 214
8.4 课后练习 215
8.5 查找 215
8.6 二维数组的介绍 220
8.7 二维数组的应用场景 220
8.8 二维数组快速入门 220
8.9 使用方式 1: 先声明/定义,再赋值 221
8.10 使用方式 2: 直接初始化 222
8.11 二维数组的遍历 223
8.12 二维数组的应用案例 223

第 9 章 MAP 225

9.1 MAP 的基本介绍 225
9.2 MAP 的声明 225
9.2.1 基本语法 225
9.2.2 map 声明的举例 225
9.3 MAP 的使用 226

9.4 MAP 的增删改查操作 228
9.5 MAP 遍历： 230
9.6 MAP 切片 231
9.6.1 基本介绍 231
9.6.2 案例演示 231
9.7 MAP 排序 233
9.7.1 基本介绍 233
9.7.2 案例演示 233
9.8 MAP 使用细节 234
9.9 MAP 的课堂练习题 235
第 10 章 面向对象编程(上) 238
10.1 结构体 238
10.1.1 看一个问题 238
10.1.2 使用现有技术解决 238
10.1.3 现有技术解决的缺点分析 239
10.1.4 一个程序就是一个世界，有很多对象(变量) 239
10.1.5 Golang 语言面向对象编程说明 239
10.1.6 结构体与结构体变量(实例/对象)的关系示意图 240
10.1.7 快速入门-面向对象的方式(struct)解决养猫问题 240
10.1.8 结构体和结构体变量(实例)的区别和联系 241
10.1.9 结构体变量(实例)在内存的布局(重要!) 241
10.1.10 如何声明结构体 242
10.1.11 字段/属性 242
10.1.12 创建结构体变量和访问结构体字段 245
10.1.13 struct 类型的内存分配机制 247
10.1.14 结构体使用注意事项和细节 249
10.2 方法 252
10.2.1 基本介绍 252
10.2.2 方法的声明和调用 253
10.2.3 方法快速入门 255
10.2.4 方法的调用和传参机制原理：(重要！) 256
10.2.5 方法的声明(定义) 257
10.2.6 方法的注意事项和细节 258
10.2.7 方法的课堂练习题 260
10.2.8 方法的课后练习题 264
10.2.9 方法和函数区别 264
10.3 面向对象编程应用实例 266
10.3.1 步骤 266

10.3.2 学生案例：.....266
10.3.3 小狗案例 [学员课后练习].....268
10.3.4 盒子案例.....268
10.3.5 景区门票案例.....269
10.4 创建结构体变量时指定字段值.....270
10.5 工厂模式.....271
10.5.1 说明.....271
10.5.2 看一个需求.....272
10.5.3 工厂模式来解决问题.....272
10.5.4 思考题.....274
第 11 章 面向对象编程(下).....275
11.1 VSCODE 的使用.....275
11.1.1 VSCode 使用技巧和经验.....275
11.2 面向对象编程思想-抽象.....276
11.2.1 抽象的介绍.....276
11.2.2 代码实现.....276
11.3 面向对象编程三大特性-封装.....280
11.3.1 基本介绍.....280
11.3.2 封装介绍.....280
11.3.3 封装的理解和好处.....280
11.3.4 如何体现封装.....281
11.3.5 封装的实现步骤.....281
11.3.6 快速入门案例.....282
11.3.7 课堂练习(学员先做).....284
11.4 面向对象编程三大特性-继承.....288
11.4.1 看一个问题，引出继承的必要性.....288
11.4.2 继承基本介绍和示意图.....292
11.4.3 嵌套匿名结构体的基本语法.....293
11.4.4 快速入门案例.....293
11.4.5 继承给编程带来的便利.....297
11.4.6 继承的深入讨论.....297
11.4.7 课堂练习.....300
11.4.8 面向对象编程-多重继承.....301
11.5 接口(INTERFACE).....302
11.5.1 基本介绍.....302
11.5.2 为什么有接口.....302
11.5.3 接口快速入门.....303
11.5.4 接口概念的再说明.....306

11.5.5 基本语法.....306
11.5.6 接口使用的应用场景.....306
11.5.7 注意事项和细节.....307
11.5.8 课堂练习.....311
11.5.9 接口编程的最佳实践.....312
11.5.10 实现接口 vs 继承.....316
11.6 面向对象编程-多态.....319
11.6.1 基本介绍.....319
11.6.2 快速入门.....319
11.6.3 接口体现多态的两种形式.....320
11.7 类型断言.....322
11.7.1 由一个具体的需要，引出了类型断言.....322
11.7.2 基本介绍.....323
11.7.3 类型断言的最佳实践 1.....324
11.7.4 类型断言的最佳实践 2.....327
11.7.5 类型断言的最佳实践 3 【学员自己完成】.....329

第 12 章 项目 1-家庭收支记账软件项目.....330

12.1 项目开发流程说明.....330
12.2 项目需求说明.....330
12.3 项目的界面.....330
12.4 项目代码实现.....331
12.4.1 实现基本功能(先使用面向过程,后面改成面向对象).....331
12.4.2 项目代码实现改进.....334
12.4.3 对项目的扩展功能的练习.....341

第 13 章 项目 2-客户信息关系系统.....342

13.1 项目需求分析.....342
13.2 项目的界面设计.....342
13.3 客户关系管理系统的程序框架图.....344
13.4 项目功能实现-显示主菜单和完成退出软件功能.....344
13.5 项目功能实现-完成显示客户列表的功能.....349
13.6 项目功能实现-添加客户的功能.....353
13.7 项目功能实现-完成删除客户的功能.....356
13.8 项目功能实现-完善退出确认功能（课后作业）.....359
13.9 客户关系管理系统-课后练习.....360

第 14 章 文件操作.....361

14.1 文件的基本介绍.....361
14.2 打开文件和关闭文件.....362

14.3 读文件操作应用实例.....363
14.4 写文件操作应用实例.....365
14.4.1 基本介绍-os.OpenFile 函数.....365
14.4.2 基本应用实例-方式一.....366
14.4.3 基本应用实例-方式二.....371
14.4.4 判断文件是否存在.....372
14.5 文件编程应用实例.....373
14.5.1 拷贝文件.....373
14.5.2 统计英文、数字、空格和其他字符数量.....375
14.6 命令行参数.....377
14.6.1 看一个需求.....377
14.6.2 基本介绍.....378
14.6.3 举例说明.....378
14.6.4 flag 包用来解析命令行参数.....378
14.7 JSON 基本介绍.....379
14.8 JSON 数据格式说明.....380
14.9 JSON 数据在线解析.....381
14.10 JSON 的序列化.....382
14.11 JSON 的反序列化.....386

第 15 章 单元测试.....391

15.1 先看一个需求.....391
15.2 传统的方法.....391
15.2.1 传统的方式来进行测试.....391
15.2.2 传统方法的缺点分析.....392
15.3 单元测试-基本介绍.....392
15.4 单元测试-快速入门.....393
15.4.1 单元测试快速入门总结.....394
15.5 单元测试-综合案例.....395

第 16 章 GOROUTINE 和 CHANNEL.....400

16.1 GOROUTINE-看一个需求.....400
16.2 GOROUTINE-基本介绍.....400
16.2.1 进程和线程介绍.....400
16.2.2 程序、进程和线程的关系示意图.....400
16.2.3 并发和并行.....401
16.2.4 Go 协程和 Go 主线程.....402
16.3 GOROUTINE-快速入门.....403
16.3.1 案例说明.....403

16.3.2 快速入门小结.....405
16.4 GOROUTINE 的调度模型.....405
16.4.1 MPG 模式基本介绍.....405
16.4.2 MPG 模式运行的状态 1.....406
16.4.3 MPG 模式运行的状态 2.....406
16.5 设置 GOLANG 运行的 CPU 数.....407
16.6 CHANNEL(管道)-看个需求.....408
16.6.1 不同 goroutine 之间如何通讯.....410
16.6.2 使用全局变量加锁同步改进程序.....411
16.6.3 为什么需要 channel.....412
16.6.4 channel 的基本介绍.....412
16.6.5 定义/声明 channel.....413
16.6.6 管道的初始化，写入数据到管道，从管道读取数据及基本的注意事项.....413
16.6.7 channel 使用的注意事项.....415
16.6.8 读写 channel 案例演示.....416
16.7 管道的课后练习题.....419
16.8 CHANNEL 的遍历和关闭.....419
16.8.1 channel 的关闭.....419
16.8.2 channel 的遍历.....420
16.8.3 channel 遍历和关闭的案例演示.....420
16.8.4 应用实例 1.....421
16.8.5 应用实例 2-阻塞.....424
16.8.6 应用实例 3.....425
16.9 CHANNEL 使用细节和注意事项.....429

第 17 章 反射.....435

17.1 先看一个问题，反射的使用场景.....435
17.2 使用反射机制，编写函数的适配器，桥连接.....435
17.3 反射的基本介绍.....436
17.3.1 基本介绍.....436
17.3.2 反射的应用场景.....436
17.3.3 反射重要的函数和概念.....437
17.4 反射的快速入门.....439
17.4.1 快速入门说明.....439
17.5 反射的注意事项和细节.....442
17.6 反射课堂练习.....444
17.7 反射最佳实践.....445
17.8 课后作业.....449

第 18 章 TCP 编程.....450

18.1 看两个实际应用.....450
18.2 网络编程基本介绍.....450
18.2.1 网线,网卡,无线网卡.....450
18.2.2 协议(tcp/ip).....451
18.2.3 OSI 与 Tcp/ip 参考模型 (推荐 tcp/ip 协议 3 卷).....451
18.2.4 ip 地址.....452
18.2.5 端口(port)-介绍.....453
18.2.6 端口(port)-分类.....454
18.2.7 端口(port)-使用注意.....454
18.3 TCP SOCKET 编程的客户端和服务器端.....455
18.4 TCP SOCKET 编程的快速入门.....455
18.4.1 服务端的处理流程.....455
18.4.2 客户端的处理流程.....455
18.4.3 简单的程序示意图.....456
18.4.4 代码的实现.....456
18.5 经典项目-海量用户即时通讯系统.....461
18.5.1 项目开发流程.....461
18.5.2 需求分析.....462
18.5.3 界面设计.....462
18.5.4 项目开发前技术准备.....462
18.5.5 实现功能-显示客户端登录菜单.....463
18.5.6 实现功能-完成用户登录.....466
18.5.7 实现功能-完成注册用户.....501
18.5.8 实现功能-完成登录时能返回当前在线用户.....509
18.5.9 实现功能-完成登录用可以群聊.....520
18.5.10 聊天的项目的扩展功能要求.....529

第 19 章 REDIS 的使用.....530

19.1 REDIS 基本介绍.....530
19.1.1 Redis 的安装.....530
19.1.2 Redis 操作的基本原理图.....530
19.2 REDIS 的安装和基本使用.....531
19.2.1 Redis 的启动:.....531
19.3 REDIS 的操作指令一览.....531
19.3.1 Redis 的基本使用:.....532
19.4 REDIS 的 CRUD 操作.....533
19.4.1 Redis 的五大数据类型:.....533
19.4.2 String(字符串) -介绍.....533
19.4.3 String(字符串)-使用细节和注意事项.....534

19.4.4 Hash (哈希，类似 golang 里的 Map)-介绍.....535
19.4.5 Hash（哈希，类似 golang 里的 Map）-CRUD.....536
19.4.6 Hash-使用细节和注意事项.....537
19.4.7 课堂练习.....537
19.4.8 List（列表）-介绍.....538
19.4.9 List（列表）-CRUD.....538
19.4.10 List-使用细节和注意事项.....540
19.4.11 Set(集合) - 介绍.....540
19.4.12 Set(集合)- CRUD.....541
19.4.13 Set 课堂练习.....542
19.5 GOLANG 操作 REDIS.....542
19.5.1 安装第三方开源 Redis 库.....542
19.5.2 Set/Get 接口.....543
19.5.3 操作 Hash.....544
19.5.4 批量 Set/Get 数据.....548
19.5.5 给数据设置有效时间.....549
19.5.6 操作 List.....549
19.5.7 Redis 链接池.....549

第 20 章 数据结构.....554

20.1 数据结构(算法)的介绍.....554
20.2 数据结构和算法的关系.....554
20.3 看几个实际编程中遇到的问题.....554
20.4 稀疏 SPARSEARRAY 数组.....557
20.4.1 先看一个实际的需求.....557
20.4.2 基本介绍.....557
20.4.3 稀疏数组举例说明.....558
20.4.4 应用实例.....558
20.5 队列(QUEUE).....563
20.5.1 队列的应用场景.....563
20.5.2 队列介绍.....563
20.5.3 数组模拟队列.....564
20.5.4 数组模拟环形队列.....569
20.6 链表.....575
20.6.1 链表介绍.....575
20.6.2 单链表的介绍.....575
20.6.3 单链表的应用实例.....576
20.6.4 双向链表的应用实例.....582
20.6.5 单向环形链表的应用场景.....590

20.6.6 环形单向链表介绍.....591
20.6.7 环形的单向链表的案例.....591
20.6.8 环形单向链表的应用实例.....597
20.7 排序.....603
20.7.1 排序的介绍.....603
20.7.2 冒泡排序.....603
20.7.3 选择排序基本介绍.....603
20.7.4 选择排序思想:.....604
20.7.5 选择排序的示意图.....604
20.7.6 代码实现.....604
20.7.7 插入排序法介绍:.....605
20.7.8 插入排序法思想:.....606
20.7.9 插入排序的示意图.....606
20.7.10 插入排序法应用实例.....606
20.7.11 插入排序的代码实现.....607
20.7.12 快速排序法介绍.....607
20.7.13 快速排序法示意图.....608
20.7.14 快速排序法应用实例.....608
20.7.15 快速排序法的代码实现.....608
20.7.16 三种排序方法的速度的分析.....611
20.8 栈.....611
20.8.1 看一个实际需求.....611
20.8.2 栈的介绍.....612
20.8.3 栈的入栈和出栈的示意图.....612
20.8.4 栈的应用场景.....613
20.8.5 栈的案例.....613
20.8.6 栈实现综合计算器.....615
20.9 递归.....624
20.9.1 递归的一个应用场景[迷宫问题].....624
20.9.2 递归的概念.....624
20.9.3 递归快速入门.....624
20.9.4 递归用于解决什么样的问题.....625
20.9.5 递归需要遵守的重要原则.....625
20.9.6 举一个比较综合的案例,迷宫问题.....626
20.10 哈希表(散列).....630
20.10.1 实际的需求.....630
20.10.2 哈希表的基本介绍.....630
20.10.3 使用 hashtable 来实现一个雇员的管理系统[增删改查].....631

第 1 章 Golang 开山篇

1.1 Golang 的学习方向

Go 语言，我们可以简单的写成 Golang.

1.2 Golang 的应用领域

1.2.1区块链的应用开发

区块链应用

区块链技术，[1] 简称BT（Blockchain technology），也被称之为**分布式账本技术**，是一种互联网数据库技术，其特点是去**中心化**、**公开透明**，让**每个人均可参与数据库记录**

1.2.2后台的服务应用

后端服务器应用

美团后台流量支撑程序

支撑主站后台流量（**排序，推荐，搜索等**），提供**负载均衡，cache，容错，按条件分流**，统计运行指标（qps，latency）等功能-》Golang

仙侠道
产品网址：仙侠道官网 - 心动游戏
应用范围： 游戏服务端（通讯、逻辑、数据存储）

1.2.3云计算/云服务后台应用

云计算/云服务后台应用

盛大云CDN (内容分发网络)
网址：盛大云计算
应用范围：CDN的调度系统、分发系统、监控系统、短域名服务，CDN内部开放平台、运营报表系统以及其他一些小工具等

京东消息推送云服务/京东分布式文件系统
网址: 京东云
应用范围：后台所有服务全部用go实现
golang的计算能力强。

1.3 学习方法的介绍

我对学习Go编程方法的理解，**希望我们能够达成共识**:

金花鼠gordon

1) **高效**而**愉快**的学习
2) 先建立一个**整体框架**，然后**细节**
3) 在实际工作中，要培养用到什么，能够快速学习什么能力
4) 先know how ,再know why【工科】
5) 软件编程是一门 "做中学" 的学科 ,不是会了再做，而是做了才会.
6) 适当的**囫囵吞枣**
7) 学习软件编程是在琢磨别人怎么做，而不是我认为应该怎么做的过程

```
for i := 0; i < 10; i++ {
        fmt.Println("hello,world")
}
```

1.4 讲课的方式的说明

1) 努力做到通俗易懂

2) 注重 Go 语言体系，同时也兼顾技术细节

3) 在实际工作中，如何快速的掌握一个技术的分享，同时也是我们授课的思路(怎么讲解或者学习一个技术)。(很多学员反馈非常受用)

第 2 章 Golang 的概述

2.1 什么是程序

程序：就是完成某个功能的指令的集合。画一个图理解：

2.2 Go 语言的诞生小故事

2.2.1Go 语言的核心开发团队-三个大牛

Ken Thompson(肯·汤普森)：1983年图灵奖（Turing Award）和1998年美国国家技术奖（National Medal of Technology）得主。他与Dennis Ritchie是Unix的原创者。Thompson也发明了后来衍生出C语言的B程序语言，同时也是C语言的主要发明人。2000年

Rob Pike(罗布·派克)：曾是贝尔实验室（Bell Labs）的Unix团队，和Plan 9操作系统计划的成员。他与Thompson共事多年，并共创出广泛使用的UTF-8 字元编码。

Robert Griesemer：曾协助制作Java的HotSpot编译器，和Chrome浏览器的JavaScript引擎V8。

Ken Thompson(肯·汤普森)

Rob Pike(罗布·派克) 1980 奥运会 射箭 银牌 天文学家 【】

2.2.2Google 创造 Golang 的原因

1) 计算机硬**件技术更新频繁**，性能提高很快。目前主流的编程语言发展明显落后于硬件，不能合理利用**多核多CPU**的优势提升软件系统性能。

2) 软件系统复杂度越来越高，维护成本越来越高，目前**缺乏一个足够简洁高效**的编程语言。【现有的编程语言: 1. 风格不统一 2. 计算能力不够 3. 处理大并发不够好】

3) 企业运行维护很多c/c++的项目，c/c++程序运行速度虽然很快，但是编译速度确很慢，同时还存在**内存泄漏**的一系列的困扰需要解决。

2.2.3Golang 的发展历程

- 2007 年，谷歌工程师 Rob Pike, Ken Thompson 和 Robert Griesemer 开始设计一门全新的语言，这是 Go 语言的最初原型。

- **2009** 年 11 月 10 日，Google 将 Go 语言以开放源代码的方式向全球发布。
- **2015** 年 8 月 19 日，Go 1.5 版发布，本次更新中移除了”最后残余的 C 代码”

- 2017 年 2 月 17 日，Go 语言 Go 1.8 版发布。
- 2017 年 8 月 24 日，**Go 语言 Go 1.9 版发布**。 1.9.2 版本
- 2018 年 2 月 16 日，Go 语言 Go 1.10 版发布。

2.3 Golang 的语言的特点

- 简介：

Go 语言保证了既能到达**静态编译语言的安全和性能**，又达到了**动态语言开发维护的高效率**，使用一个表达式来形容 Go 语言：**Go = C + Python**，说明 Go 语言既有 C 静态语言程序的运行速度，又能达到 Python 动态语言的快速开发。

1) 从 C 语言中继承了很多理念，包括表达式语法，控制结构，基础数据类型，调用参数传值，**指针**等等，也保留了和 C 语言一样的编译执行方式及弱化的指针

举一个案例(体验)：

```
//go 语言的指针的使用特点(体验)
func testPtr(num *int) {
    *num = 20
}
```

2) 引入**包的概念**，用于组织程序结构，**Go 语言的一个文件都要归属于一个包**，而不能单独存在。

```go
package main //一个go文件需要在一个包

import "fmt"
func sayOk(){
    //输出一句话
    fmt.Println("ok")
}
```

3) 垃圾回收机制，内存自动回收，不需开发人员管理

4) 天然并发 (**重要特点**)

(1) 从语言层面支持并发，实现简单

(2) goroutine，轻量级线程，可实现大并发处理，高效利用多核。

(3) 基于 CPS 并发模型(Communicating Sequential Processes)实现

5) 吸收了管道通信机制，形成 Go 语言特有的管道 channel 通过管道 channel，可以实现不同的 goroute 之间的相互通信。

6) 函数可以返回多个值。举例：

```go
//写一个函数，实现同时返回 和，差
//go 函数支持返回多个值
func getSumAndSub(n1 int, n2 int) (int, int ) {
    sum := n1 + n2 //go 语句后面不要带分号.
    sub := n1 - n2
    return sum , sub
}
```

7) 新的创新：比如切片 slice、延时执行 defer

2.4 Golang 的开发工具的介绍

2.4.1工具介绍

工具介绍：

1) visual studio code Microsoft 产品(VSCode)：一个运行于 Mac OS X、Windows和 Linux 之上的，**默认提供Go 语言的语法高亮**，安装Go语言插件，还可以支持智能提示，编译运行等功能。
2) Sublime Text，可以免费使用，默认也支持Go代码语法高亮，只是保存次数达到一定数量之后就会提示是否购买，点击取消继续用，和正式注册版本没有任何区别
3) Vim：Vim是从vi发展出来的一个**文本编辑器**，代码补全、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用
4) Emacs：Emacs传说中的神器，她不仅仅是一个编辑器，因为功能强大，可称它为集成开发环境

5) Eclipse IDE工具，开源免费， 并提供GoEclipse插件
6) LiteIDE，LiteIDE是一款专门为**Go语言开发**的跨平台轻量级集成开发环境（IDE），是国人开发的。
7) JetBrains公司的产品：PhpStrom、WebStrom和PyCharm 等IDE工具，都需要安装Go插件。

2.4.2工具选择：

➢ 如何选择开发工具

我们先选择用 **visual studio code** 或者 vim 文本编辑器本，到大家对 Go 语言有一定了解后，我们再使用 Eclipse 等 IDE 开发工具。

➢ 这是为什么呢?

1) 更深刻的理解 Go 语言技术,培养代码感。->写代码的感觉。
2) 有利于公司面试。-> 给你纸，写程序

2.4.3 VSCode 的安装和使用

1) 先到下载地址去选择适合自己系统的 VSCode 安装软件

https://code.visualstudio.com/download

2) 演示如何在 windows 下安装 vscode 并使用

步骤 1: 把 vscode 安装文件准备好

步骤 2：双击安装文件，就可以一步一步安装，我个人的习惯是安装到 d:/programs 目录. 当看到如下界面时，就表示安装成功!

步骤 3: 简单的使用一下 vscode

在 d 盘创建了一个文件夹 gocode.

3) 演示如何在 Linux(ubuntu /centos)下安装 vscode 并使用

这里，我们介绍一下我的 linux 的环境:

步骤 1: 先下载 linux 版本的 vscode 安装软件.

步骤 2: 因为我这里使用的是虚拟机的 ubuntu, 因此我们需要先将 vscode 安装软件传输到 ubuntu 下,使用的 xftp5 软件上传。

步骤 3: 如果你是在 ubuntu 下做 go 开发，我们建议将 vscode 安装到 /opt 目录..

步骤 4：将安装软件拷贝到 /opt

步骤 5: cd /opt 【切换到 /opt】

步骤 6: 将安装文件解决即可

```
root@ubuntu:/opt# tar -zxvf code-stable-code_1.22.2-1523551015_amd64.tar.gz
```

步骤 7: 现在进入解压后的目录，即可运行我们的 vscode

4) 演示如何在 Mac 下安装 vscode 并使用

如果你使用的就是 **mac** 系统，也可以在该系统下进行 go 开发.

步骤 1：下载 mac 版本的 vscode 安装软件

步骤 2：把 vscode 安装软件，传输到 mac 系统

细节：在，默认情况下 mac 没有启动 ssh 服务，所以需要我们启动一下，才能远程传输文件.

mac 本身安装了 ssh 服务，默认情况下不会开机自启

1.启动 sshd 服务：

sudo launchctl load -w /System/Library/LaunchDaemons/ssh.plist

2.停止 sshd 服务：

sudo launchctl unload -w /System/Library/LaunchDaemons/ssh.plist

3 查看是否启动：

sudo launchctl list | grep ssh

如果看到下面的输出表示成功启动了：

————————————————

- 0 com.openssh.sshd

步骤 3：将安装软件解压后即可使用.

进入到这个解压后的文件夹(**图形界面**)，双击即可

步骤 4：编写解简单测试.

在用户所在的目录，创建了 gocode,然后将 test.go 写到这个文件夹下 ..

2.4.4 小结

我们会讲解在 windows， linux , mac 如何安装 vscode 开发工具，并且还会讲解如何在三个系统下安装 go 的 sdk 和如何开发 go 程序。

但是为了学习方便，我们前期选择 Windows 下开发 go。到我们开始讲项目和将区块链时，就会使用 linux 系统。

在实际开发中，也可以在 windows 开发好程序，然后部署到 linux 下。

2.5 Windows 下搭建 Go 开发环境-安装和配置 SDK

2.5.1 介绍了 SDK

1) SDK 的全称(**S**oftware **D**evelopment **K**it 软件开发工具包)
2) SDK 是提供给开发人员使用的，其中包含了对应开发语言的工具包

2.5.2 下载 SDK 工具包

1) Go 语言的官网为：golang.org，因为各种原因，可能无法访问。
2) SDK 下载地址：Golang 中国 https://www.golangtc.com/download
3) 如何选择对应的 sdk 版本

2.5.3windows 下安装 sdk

1) Windows 下 SDK 的各个版本说明：

Windows 下：根据自己系统是 32 位还是 64 位进行下载：

32 位系统：go1.9.2.windows-386.zip

64 位系统：**go1.9.2.windows-amd64.zip**

2) 请注意：安装路径不要有中文或者特殊符号如空格等

3) SDK 安装目录建议：windows 一般我安装在 d:/programs

4) 安装时，基本上是傻瓜式安装，解压就可以使用

5) 安装看老师的演示：

6) 解压后，我们会看到 d:/go 目录，这个是 sdk

如何测试我们的 go 的 sdk 安装成功。

2.5.4windows 下配置 Golang 环境变量：

➢ 为什么需要配置环境变量

➢ **为什么要配置环境变量**

1) **看一个现象**

在dos命令行中敲入go，出现错误提示

2) **原因分析**

错误原因：当前执行的程序在当前目录下如果不存在，windows系统会在系统中已有的一个名为path的环境变量指定的目录中查找。如果仍未找到，会出现以上的错误提示。所以进入到 go安装路径\bin目录下，执行go，会看到go参数提示信息

➢ 配置环境变量介绍

根据 windows 系统在查找可执行程序的原理，可以将 Go 所在路径定义到环境变量中，让系统帮我们去找运行执行的程序，这样在任何目录下都可以执行 go 指令。

➢ 在 Go 开发中，需要配置哪些环境变量

环境变量	说明
GOROOT	指定SDK的安装路径 d:/programs/go
Path	添加SDK的/bin目录
GOPATH	工作目录，将来我们的go项目的工作路径

➢ 看老师如何配置

步骤 1：先打开环境变量配置的界面

步骤 2: 配置我们的环境变量

对上图的一个说明：

1) Path 这个环境变量不需要在创建，因为系统本身就有，你后面增加即可

2) 增加 Go 的 bin： **;%GOROOT%\bin**

对上图的一个说明

1) GOPATH ：就是你以后 go 项目存放的路径，即工作目录

2) GOPATH：是一个新建的环境变量

➢ 测试一下我们的环境变量是否配置 ok

```
C:\Users\Administrator>go version
go version go1.9.2 windows/amd64
```

注意：配置环境变量后，需要重新打开一次 dos 的终端，这样环境变量才会生效。

2.6 Linux 下搭建 Go 开发环境-安装和配置 SDK

2.6.1 Linux 下安装 SDK：

1) Linux 下 SDK 的各个版本说明：

Linux 下：根据系统是 32 位还是 64 位进行下载：

32 位系统：go1.9.2.linux-386.tar.gz

64 位系统：**go1.9.2.linux-amd64.tar.gz**

如何确认你的 linux 是多少位：

```
atguigu@ubuntu:~$ uname -a
Linux ubuntu 4.13.0-38-generic #43~16.04.1-Ubuntu SMP Wed Ma
18 x86_64 x86_64 x86_64 GNU/Linux
atguigu@ubuntu:~$
```

2) 请注意：安装路径不要有中文或者特殊符号如空格等

3) SDK 安装目录建议： linux 放在 /opt 目录下

4) 安装时，解压即可，我们使用的是 tar.gz

5) 看老师演示

步骤 1: 将 go1.9.2.linux-amd64.tar.gz 传输到 ubuntu

步骤 2: 将 go1.9.2.linux-amd64.tar.gz 拷贝到 /opt 下

```
igu# cp go1.9.2.linux-amd64.tar.gz  /opt
igu#
```

步骤 3: cd /opt

步骤 4：tar -zxvf go1.9.2.linux-amd64.tar.gz [解压后，就可以看到一个 go 目录]

步骤 5: cd go/bin

步骤 6：./go version

```
root@ubuntu:/opt/go/bin# ./go version
go version go1.9.2 linux/amd64
root@ubuntu:/opt/go/bin#
```

2.6.2Linux 下配置 Golang 环境变量

步骤 1：使用 root 的权限来编辑 vim /etc/profile 文件

```
fi
export GOROOT=/opt/go
export PATH=$PATH:$GOROOT/bin
export GOPATH=$HOME/goproject
~
```

步骤 2: 如果需要生效的话，需要注销一下(重新登录)，再使用

```
atguigu@ubuntu:~$ go version
go version go1.9.2 linux/amd64
atguigu@ubuntu:~$
atguigu@ubuntu:~$
```

2.7 Mac 下搭建 Go 开发环境-安装和配置 SDK

2.7.1mac 下安装 Go 的 sdk

1) Mac 下 SDK 的各个版本说明：

Mac OS 下：只有 64 位的软件安装包

Mac OS 系统的安装包：go1.9.2.darwin-amd64.tar.gz

2) 请注意：安装路径不要有中文或者特殊符号如空格等

3) SDK 安装目录建议： Mac 一般放在用户目录下 go_dev/go 下

4) 安装时，解压即可

5) 看老师的演示步骤

步骤 1： 先将我们的安装文件 go1.9.2.darwin-amd64.tar.gz 上传到 mac

步骤 2: 先在用户目录下，创建一个目录 go_dev，将我们上传的文件 移动到 go_dev 目录

步骤 3： 解压 tar -zxvf go1.9.2.darwin-amd64.tar.gz

步骤 4: 解压后，我们会得到一个目录 go， 进入到 go/bin 就是可以使用

```
atguigudeMac:bin atguigu$ ./go version
go version go1.9.2 darwin/amd64
atguigudeMac:bin atguigu$
```

这里还是有一个问题，就是如果我们不做 bin 目录下，就使用不了 go 程序。因此我们仍然需要配置 go 的环境变量。

2.7.2Mac 下配置 Golang 环境变量：

步骤 1：使用 root 用户，修改 /etc/profile 增加环境变量的配置

```
fi
export GOROOT=$HOME/go_dev/go
export PATH=$PATH:$GOROOT/bin
export GOPATH=$HOME/goproject
```

步骤 2: 配置完后，需要重新注销用户，配置才会生效.

```
atguigudeMac:~ atguigu$ pwd
/Users/atguigu
atguigudeMac:~ atguigu$ go version
go version go1.9.2 darwin/amd64
atguigudeMac:~ atguigu$
```

2.8 Go 语言快速开发入门

2.8.1需求

要求开发一个 hello.go 程序，可以输出 "hello,world”

2.8.2开发的步骤

1) 开发这个程序/项目时，go 的目录结构怎么处理.

2) 代码如下：

对上图的说明

(1) go 文件的后缀是 .go

(2) package main

表示该 hello.go 文件所在的包是 main，在 go 中，每个文件都必须归属于一个包。

(3) import “fmt”

表示：引入一个包，包名 fmt，引入该包后，就可以使用 fmt 包的函数，比如：fmt.Println

(4) **func** main() {

}

func 是一个关键字，表示一个函数。

main 是函数名，是一个主函数，即我们程序的入口。

(5) fmt.Println("hello")

表示调用 fmt 包的函数 Println 输出 "hello,world"

3) 通过 go build 命令对该 go 文件进行编译，生成 .exe 文件.

```
D:\goproject\src\go_code\project01\main>go build hello.go

D:\goproject\src\go_code\project01\main>dir
 驱动器 D 中的卷是 新加卷
 卷的序列号是 D2AD-BC9F

 D:\goproject\src\go_code\project01\main 的目录

05/05  16:01    <DIR>          .
05/05  16:01    <DIR>          ..
05/05  16:01         1,940,480 hello.exe
05/05  15:53               146 hello.go
               2 个文件      1,940,626 字节
               2 个目录 23,630,262,272 可用字节

D:\goproject\src\go_code\project01\main>
```

4) 运行 hello.exe 文件即可

```
D:\goproject\src\go_code\project01\main>hello.exe
hello,world!
```

5) 注意：通过 go run 命令**可以**直接运行 hello.go 程序 [类似执行一个脚本文件的形式]

```
D:\goproject\src\go_code\project01\main>go run hello.go
hello,world!
```

2.8.3 linux 下如何开发 Go 程序

说明：linux 下开发 go 和 windows 开发基本是一样的。只是在运行可执行的程序时，

是以 **./文件名方式**

演示：在 linux 下开发 Go 程序。

编译和运行 hello.go

```
atguigu@ubuntu:~/goproject/src/go_code/project01/main$ go build hello.go
atguigu@ubuntu:~/goproject/src/go_code/project01/main$ ls
```

```
atguigu@ubuntu:~/goproject/src/go_code/project01/main$ ./hello
hello,go
```

也可以直接使用 go run hello.go 方式运行

```
atguigu@ubuntu:~/goproject/src/go_code/project01/main$ go run hello.go
hello,go
```

2.8.4Mac 下如何开发 Go 程序

- **说明**：在 mac 下开发 go 程序和 windows 基本一样。
- **演示一下**：如何在 mac 下开发一个 hello.go 程序

- 源代码的编写：hello.go

➢ 编译再运行，直接 go run 来运行

```
Pictures
atguigudeMac:~ atguigu$ cd goproject/src/go_code/project01/main/
atguigudeMac:main atguigu$ ls
hello.go
atguigudeMac:main atguigu$
atguigudeMac:main atguigu$ go build hello.go
atguigudeMac:main atguigu$ ls
hello            hello.go
atguigudeMac:main atguigu$ ./hello
hello,atguigu
atguigudeMac:main atguigu$
```

➢ 直接 go run 来运行

```
atguigudeMac:main atguigu$ go run hello.go
hello,atguigu
```

2.8.5go 语言的快速入门的课堂练习

课堂小练习

要求windows下开发一个hi.go 程序，可以输出 "hello,world!" (5min)

2.8.6Golang 执行流程分析

➢ 如果是对源码编译后，再执行，Go 的执行流程如下图

➢ 如果我们是对源码直接 执行 go run 源码，Go 的执行流程如下图

➢ 两种执行流程的方式区别

1) 如果我们先编译生成了可执行文件，那么我们可以将该可执行文件拷贝到没有 go 开发环境的机器上，仍然可以运行

2) 如果我们是直接 go run go 源代码，那么如果要在另外一个机器上这么运行，也需要 go 开发环境，否则无法执行。

3) 在编译时，编译器会将程序运行依赖的库文件包含在可执行文件中，所以，可执行文件变大了很多。

2.8.7编译和运行说明

1) 有了 go 源文件，通过编译器将其编译成机器可以识别的二进制码文件。

2) 在该源文件目录下，通过 go build 对 hello.go 文件进行编译。可以指定生成的可执行文件名，在 windows 下 必须是 .exe 后缀。

```
D:\goproject\src\go_code\project01\main>go build -o myhello.exe hello.go
```

3) 如果程序没有错误，没有任何提示，会在当前目录下会出现一个可执行文件(**windows 下是.exe Linux 下是一个可执行文件**)，该文件是二进制码文件，也是可以执行的程序。

4) 如果程序有错误，编译时，会在错误的那行报错。有助于程序员调试错误.

```
D:\goproject\src\go_code\project01\main>go build  hello.go
# command-line-arguments
.\hello.go:7:2: undefined: fmt.Printnl
```

5) 运行有两种形式

2.8.8Go 程序开发的注意事项

1) Go 源文件以 "go" 为扩展名。

2) Go 应用程序的执行入口是 main()函数。 这个是和其它编程语言（比如 java/c）

3) Go 语言严格区分大小写。

4) Go 方法由一条条语句构成，**每个语句后不需要分号**(Go 语言会在每行后自动加分号)，这也体

现出 Golang 的简洁性。

5) Go 编译器是一行行进行编译的，因此我们一行就写一条语句，不能把多条语句写在同一个，否则报错

```
package main
import "fmt"
func main() {

    fmt.Println("hello")
    fmt.Println("hello")
    fmt.Println("hello")
    fmt.Println("hello")
    fmt.Println("hello")
    fmt.Println("hello")
    fmt.Println("hello")
    fmt.Println("hello") fmt.Println("hello")
}
```

6) go 语言**定义的变量**或者 **import 的包**如果没有使用到，代码不能编译通过。

```
package main
import "fmt"
func main() {

    fmt.Println("hello")
    //定义一个变量,如果你没有使用这个变量，就会编译通不过
    var num = 10
}
```

7) 大括号都是成对出现的，缺一不可。

2.9 Go 语言的转义字符(escape char)

说明:常用的转义字符有如下:

1) \t : 表示一个制表符，通常使用它可以排版。

```go
import "fmt" //fmt包中提供格式化，输出，输入的函数.
func main() {
    //演示转义字符的使用 \t
    fmt.Println("tom\tjack")
}
```

2) \n ：换行符

3) \\ ：一个\

4) \" ：一个"

5) \r ：一个回车 fmt.Println("天龙八部雪山飞狐\r 张飞");

6) 案例截图

```go
package main

import "fmt" //fmt包中提供格式化，输出，输入的函数.
func main() {
    //演示转义字符的使用 \t
    fmt.Println("tom\tjack")

    fmt.Println("hello\nworld")
    fmt.Println("C:\\Users\\Administrator\\Desktop\\Go语言核心编程课程\\资料")
    fmt.Println("tom说\"i love you\"")
    // \r 回车,从当前行的最前面开始输出，覆盖掉以前内容
    fmt.Println("天龙八部雪山飞狐\r张飞厉害")
}
```

➢ 课堂练习

要求：请使用一**句输出语句**，达到输入如下图形的效果：

```
姓名    年龄    籍贯    住址
john    12      河北    北京
```

5min

作业评讲：

package main

```
import "fmt" //fmt 包中提供格式化，输出，输入的函数.
func main() {
    //要求：要求：请使用一句输出语句，达到输入如下图形的效果
    fmt.Println("姓名\t 年龄\t 籍贯\t 地址\njohn\t20\t 河北\t 北京")
}
```

2.10 Golang 开发常见问题和解决方法

2.10.1 文件名或者路径错误

```
D:\programfiles\gocode>go build hello100.go
CreateFile hello100.go: The system cannot find the file specified.
```

```
D:\programfiles\gocode>go run hello100.go
CreateFile hello100.go: The system cannot find the file specified.
```

```
D:\programfiles\gocode>hello02.exe
'hello02.exe' 不是内部或外部命令，也不是可运行的程序
或批处理文件。
```

解决方法:源文件名不存在或者写错，或者当前路径错误

2.10.2 小结和提示

学习编程最容易犯的错是语法错误 。Go 要求你必须按照语法规则编写代码。如果你的程序违反了语法规则，例如：忘记了大括号、引号，或者拼错了单词，Go 编译器都会报语法错误，**要求：尝试着去看懂编译器会报告的错误信息**。

```
D:\goproject\src\go_code\chapter02\escaptechar\pkgexec>go build exec01.go
# command-line-arguments
.\exec01.go:6:2: undefined: fmt.Println

D:\goproject\src\go_code\chapter02\escaptechar\pkgexec>
```

6：表错误的行

2.11 注释(comment)

2.11.1 介绍注释

用于注解说明解释程序的文字就是注释，**注释提高了代码的阅读性**；

注释是一个程序员必须要具有的良好编程习惯。将自己的思想通过注释先整理出来，再用代码去体现。

2.11.2 在 Golang 中注释有两种形式

1) 行注释

- 基本语法

// 注释内容

- 举例

```go
//这是一个main函数，是程序入口
func main() {
    //演示转义字符的使用 \t
    fmt.Println("tom\tjack")

    // 如果希望一次性注释 ctrl + / 第一次表示注释，第二次表示取消注释

    // fmt.Println("hello\nworld")
    // fmt.Println("C:\\Users\\Administrator\\Desktop\\Go语言核心编程课程\\资料")
    // fmt.Println("tom说\"i love you\"")

    // \r 回车,从当前行的最前面开始输出，覆盖掉以前内容
    fmt.Println("天龙八部雪山飞狐\r张飞厉害")
}
```

2) 块注释(多行注释)

- 基本语法

/*

注释内容

*/

- 举例说明

```
/*
fmt.Println("hello\nworld")
fmt.Println("C:\\Users\\Administrator\\Desktop\\Go语言核心编程课程\\资料")
fmt.Println("tom说\"i love you\"")
*/
```

- 使用细节

1) 对于行注释和块注释，被注释的文字，不会被 Go 编译器执行。

2) 块注释里面不允许有块注释嵌套 [注意一下]

2.12 规范的代码风格

2.12.1 正确的注释和注释风格：

1) Go 官方推荐使用行注释来注释整个方法和语句。

2) 带看 Go 源码

2.12.2 正确的缩进和空白

1) 使用一次 **tab 操作**，实现缩进,默认整体向右边移动，时候用 **shift+tab** 整体向左移

看老师的演示：

2) 或者使用 gofmt 来进行格式化 [演示]

```
D:\goproject\src\go_code\chapter02\escaptechar>gofmt main.go
package main

import "fmt" //fmt包中提供格式化，输出，输入的函数.

//这是一个main函数，是程序入口
func main() {
        //演示转义字符的使用 \t
        fmt.Println("tom\tjack")

        // 如果希望一次性注释 ctrl + / 第一次表示注释，第二次表示取消注释
        fmt.Println("hello\nworld")
        fmt.Println("C:\\Users\\Administrator\\Desktop\\Go语言核心编程课程\\资料")
        fmt.Println("tom说\"i love you\"")

        // \r 回车,从当前行的最前面开始输出，覆盖掉以前内容
        fmt.Println("天龙八部雪山飞狐\r张飞厉害")
}

D:\goproject\src\go_code\chapter02\escaptechar>gofmt -w main.go

D:\goproject\src\go_code\chapter02\escaptechar>
```

该指令可以将格式化后的内容重写写入到文件。当程序员重写打开该文件时，就会看到新的格式化后的文件。

3) 运算符两边习惯性各加一个空格。比如：2 + 4 * 5。

```
17
18      var num = 2 + 4 * 5
19  }
```

4) Go 语言的代码风格.

```
package main
import "fmt"
func main() {
    fmt.Println("hello,world!")
}
```

上面的写法是正确的.

```
package main
import "fmt"
func main()
{
    fmt.Println("hello,world!")
```

}

上面的写法不是正确，Go 语言不允许这样编写。 【Go 语言不允许这样写，是错误的！】

Go 设计者思想: **一个问题尽量只有一个解决方法**

5) 一行最长不超过 80 个字符，超过的请使用换行展示，尽量保持格式优雅

➢ 举例说明

```
// \r 回车,从当前行的最前面开始输出，覆盖掉以前内容
fmt.Println("天龙八部雪山飞狐\r张飞厉害")

fmt.Println("helloworldhelloworldhelloworldhelloworl\n",
    "dhelloworldhelloworldhelloworldhelloworldhelloworldhellowor\n",
    "ldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworl\n",
    "dhelloworldhelloworldhelloworldhelloworldhelloworldhelloworldhel\n",
    "loworldhelloworldhelloworldhelloworldhelloworldhelloworldhellowor\n",
    "ldhelloworldhelloworldhelloworldhelloworldhelloworldhelloworld")
```

2.13 Golang 官方编程指南

➢ 说明： Golang 官方网站 https://golang.org

➢ 点击上图的 tour -> 选择 简体中文就可以进入中文版的 Go 编程指南 。

➢ Golang 官方标准库 API 文档， https://golang.org/pkg 可以查看 Golang 所有包下的函数和使用

➢ 解释术语：API

api ： application program interface :应用程序编程接口。

就是我们 Go 的各个包的各个函数。

2.14 Golang 标准库 API 文档

1) API （Application Programming Interface,应用程序编程接口）是 Golang 提供的基本编程接口。

2) Go 语言提供了大量的标准库，因此 google 公司 也为这些标准库提供了相应的 API 文档，用于告诉开发者如何使用这些标准库，以及标准库包含的方法。

3) Golang 中文网 在线标准库文档： https://studygolang.com/pkgdoc

4) Golang 的包和源文件和函数的关系简图

5) 有一个离线版的 Golang_Manual_By_AstaXie_20120522.chm

2.15 Dos 的常用指令(了解)

2.15.1 dos 的基本介绍

Dos： Disk Operating System 磁盘操作系统，简单说一下 windows 的目录结构

2.15.2 dos 的基本操作原理

2.15.3 目录操作指令

- 查看当前目录是什么

- 切换到其他盘下：盘符号 F 盘

  案例演示：

- 切换到当前盘的其他目录下 (使用**相对路径**和**绝对路径**演示)

  案例演示：

```
D:\>cd d:\test100
d:\test100>cd abc100
d:\test100\abc100>cd d:\test100
d:\test100>cd d:\test100\abc100
d:\test100\abc100>
```

相对路径

绝对路径

- 切换到上一级：

  案例演示：

```
d:\test100\abc100>cd ..
d:\test100>
```

- 切换到根目录：

  案例演示：

```
d:\test100>cd \
d:\>
```

- 新建目录 md （make directory）

  新建一个目录：

```
d:\test200>md ok200
```

  新建多个目录：

```
d:\test200>md ok300 ok400
```

- 删除目录

  删除空目录

```
d:\test200>rd ok100
```

删除目录以及下面的子目录和文件，不带询问

```
d:\test200>rd /q/s ok200
```

删除目录以及下面的子目录和文件，带询问

```
d:\test200>rd /s ok300
ok300, 是否确认(Y/N)? y
```

2.15.4 文件的操作

- 新建或追加内容到文件

案例演示：

```
d:\test200>echo hello > d:\test100\abc100\abc.txt
```

```
d:\test100\abc100>echo atguigu > abc2.txt
```

- 复制或移动文件

复制

```
d:\test100\abc100>copy abc.txt d:\test200        ——► 拷贝时，使用原来文件名
已复制          1 个文件。

d:\test100\abc100>copy abc.txt d:\test200\ok.txt  拷贝重写指定名
已复制          1 个文件。
```

移动

```
d:\test100\abc100>move abc.txt f:\
移动了          1 个文件。
```

- 删除文件

  删除指定文件

```
d:\test100\abc100>del abc2.txt

d:\test100\abc100>dir
```

  删除所有文件

```
d:\test100\abc100>del *.txt
```

2.15.5 其它指令

- 清屏

  cls [苍老师]

- 退出 dos

  exit

2.15.6 综合案例

切换到E盘下，新建一个目录 atguigu100（文件夹），切换到该目录，新建a b c三个目录,然后切换到a ，新建文件news.txt,然后复制文件到b下，最后都删除

完成该案例： 10min

2.16 课后练习题的评讲

1) 独立编写 Hello,Golang!程序[评讲]

```go
package main
import "fmt"

func main() {
    fmt.Println("hello,Golang")
}
```

2) 将个人的基本信息（姓名、性别、籍贯、住址）打印到终端上输出。各条信息分别占一行

```go
package main
import "fmt"

func main() {
    fmt.Println("姓名\t性别\t籍贯\t住址\ntom\t男\t天津\t北京")
}
```

3) 在终端打印出如下图所示的效果

```
*                                   *
*     *       I love Golang     *     *
  *                               *
    *                           *
      *                       *
        *                   *
          *               *
            *     *
              *
```

2.17 本章的知识回顾

- Go 语言的 SDK 是什么?

  SDK 就是软件开发工具包。我们做 Go 开发，首先需要先安装并配置好 sdk.

- Golang 环境变量配置及其作用。

  GOROOT: 指定 go sdk 安装目录。

  Path: 指令 sdk\bin 目录：go.exe　godoc.exe　gofmt.exe

  GOPATH: 就是 golang 工作目录：我们的所有项目的源码都这个目录下。

- Golang 程序的编写、编译、运行步骤是什么? 能否一步执行?

  编写：就是写源码

  编译：go build 源码 =》 生成一个二进制的可执行文件

  运行：1. 对可执行文件运行 xx.exe　./可执行文件　2. go run 源码

- Golang 程序编写的规则。

  1) go 文件的后缀 .go

2) go 程序区分大小写

3) go 的语句后，不需要带分号

4) go 定义的变量，或者 import 包，必须使用，如果没有使用就会报错

5) go 中，不要把多条语句放在同一行。否则报错

6) go 中的大括号成对出现，而且风格

```
func main() {
    //语句
}
```

- 简述：在配置环境、编译、运行各个步骤中常见的错误

  对初学者而言，最容易错的地方**拼写错误**。比如文件名，路径错误。拼写错误

第 3 章 Golang 变量

3.1 为什么需要变量

3.1.1一个程序就是一个世界

3.1.2变量是程序的基本组成单位

不论是使用哪种高级程序语言编写程序,变量都是其程序的基本组成单位，比如一个示意图：

```go
func getVal(num1 int, num2 int) (int, int) {
    sum := num1 + num2
    sub := num2 - num2
    return sum, sub
}

func main() {

    sum, sub:= getVal(30, 30)
    fmt.Println("sum=", sum, "sub=", sub)
    sum2, _ := getVal(10, 30)//只取出第一个返回值
    fmt.Println("sum=", sum2)
}
```

比如上图的 sum,sub 都是变量。

3.2 变量的介绍

3.2.1变量的概念

变量相当于内存中一个数据存储空间的表示，你可以把变量看做是一个房间的门牌号，通过门牌号我们可以找到房间，同样的道理，通过变量名可以访问到变量(值)。

3.2.2变量的使用步骤

1) 声明变量(也叫:定义变量)
2) 非变量赋值
3) 使用变量

3.3 变量快速入门案例

看一个案例：

```go
package main
import "fmt"

func main() {
    //定义变量/声明变量
    var i int
    //给i 赋值
    i = 10
    //使用变量
    fmt.Println("i=", i)
}
```

输出：

```
D:\goproject\src\go_code\chapter03\demo01>go run main.go
i= 10
```

3.4 变量使用注意事项

1) 变量表示内存中的一个存储区域

2) 该区域有自己的名称（变量名）和类型（数据类型）

示意图：

3) Golang 变量使用的三种方式

(1) 第一种：指定变量类型，**声明后若不赋值，使用默认值**

```
package main
D:\goproject\src\go_code\test.go

func main() {
    //golang的变量使用方式1
    //第一种：指定变量类型，声明后若不赋值，使用默认值
    // int 的默认值是0 ，其它数据类型的默认值后面马上介绍
    var i int
    fmt.Println("i=", i)
}
```

(2) 第二种：根据值自行判定变量类型(**类型推导**)

```
//第二种：根据值自行判定变量类型(类型推导)
var num  = 10.11
fmt.Println("num=", num)
```

(3) 第三种：省略 var, 注意 :=左侧的变量不应该是已经声明过的，否则会导致编译错误

```
//第三种：省略var，注意 :=左侧的变量不应该是已经声明过的，否则会导致编译错误
//下面的方式等价 var name string   name = "tom"
// := 的 :不能省略，否则错误
name := "tom"
fmt.Println("name=", name)
```

4) 多变量声明

在编程中，有时我们需要一次性声明多个变量，Golang 也提供这样的语法

举例说明:

```go
package main
import "fmt"

func main() {

    //该案例演示golang如何一次性声明多个变量
    // var n1, n2, n3 int
    // fmt.Println("n1=",n1, "n2=", n2, "n3=", n3)

    //一次性声明多个变量的方式2
    // var n1, name , n3 = 100, "tom", 888
    // fmt.Println("n1=",n1, "name=", name, "n3=", n3)

    //一次性声明多个变量的方式3，同样可以使用类型推导
    n1, name , n3 := 100, "tom~", 888
    fmt.Println("n1=",n1, "name=", name, "n3=", n3)

}
```

如何一次性声明多个全局变量【在 go 中函数外部定义变量就是全局变量】：

```go

//定义全局变量
var n1 = 100
var n2 = 200
var name = "jack"
//上面的声明方式，也可以改成一次性声明
var (
    n3 = 300
    n4 = 900
    name2 = "mary"
)
```

5) 该区域的数据值可以在同一类型范围内不断变化(重点)

```go
package main
import "fmt"

//变量使用的注意事项
func main() {

    //该区域的数据值可以在同一类型范围内不断变化
    var i int = 10
    i = 30
    i = 50
    fmt.Println("i=", i)
    i = 1.2 //int ,原因是不能改变数据类型

}
```

6) 变量在**同一个作用域**(在一个函数或者在代码块)内不能重名

```go
//i = 1.2 //int ,原因是不能改变数据类型

//变量在同一个作用域(在一个函数或者在代码块)内不能重名
var i int = 59
i := 99
```

7) 变量=**变量名**+**值**+**数据类型**，这一点请大家注意，变量的三要素

8) Golang 的变量如果没有赋初值，编译器会使用默认值，比如 int 默认值 0 string 默认值为空串，小数默认为 0

3.5 变量的声明，初始化和赋值

声明变量

基本语法：var 变量名 数据类型

var a int 这就是声明了一个变量,变量名是 a

var num1 **float32** 这也声明了一个变量，表示一个单精度类型的小数,变量名是num1

初始化变量

在声明变量的时候，就给值。

var a int = 45 这就是初始化变量a

使用细节：如果声明时就直接赋值，可省略数据类型

var b = 400

给变量赋值

比如你先声明了变量: var num int // 默认0

然后，再给值 num = 780; 这就是给变量赋值.

3.6 程序中 +号的使用

1) 当左右两边都是数值型时，则做加法运算

2) 当左右两边都是字符串，则做字符串拼接

```go
package main
import "fmt"

//演示golang中+的使用
func main() {

    var i = 1
    var j = 2
    var r = i + j //做加法运算
    fmt.Println("r=", r)

    var str1 = "hello "
    var str2 = "world"
    var res = str1 + str2 //做拼接操作
    fmt.Println("res=", res)

}
```

3.7 数据类型的基本介绍

3.8 整数类型

3.8.1基本介绍

简单的说，就是**用于存放整数值**的，比如 0, -1, 2345 等等。

3.8.2案例演示

3.8.3整数的各个类型

整型的类型

类 型	有无符号	占用存储空间	表数范围	备注
int8	有	1字节	-128 ~ 127	
int16	有	2字节	$-2^{15} \sim 2^{15}-1$	
int32	有	4字节	$-2^{31} \sim 2^{31}-1$	
int64	有	8字节	$-2^{63} \sim 2^{63}-1$	

```go
//演示golang中整数类型使用
func main() {

    var i int = 1
    fmt.Println("i=", i)

    //测试一下int8的范围 -128~127,
    //其它的 int16, int32, int64,类推。。。
    var j int8 = 127
    fmt.Println("j=", j)

}
```

int 的无符号的类型：

整型的类型

类 型	有无符号	占用存储空间	表数范围	备注
uint8	无	1字节	0 ~ 255	
uint16	无	2字节	$0 \sim 2^{16}-1$	
uint32	无	4字节	$0 \sim 2^{32}-1$	
uint64	无	8字节	$0 \sim 2^{64}-1$	

说明一下：

```
//测试一下 uint8的范围(0-255),其它的 uint16, uint32, uint64类推即可
var k uint16 = 255
fmt.Println("k=", k)
```

int 的其它类型的说明:

整型的类型

类 型	有无符号	占用存储空间	表数范围	备注
int	有	32位系统4个字节 64位系统8个字节	$-2^{31} \sim 2^{31}-1$ $-2^{63} \sim 2^{63}-1$	
uint	无	32位系统4个字节 64位系统8个字节	$0 \sim 2^{32}-1$ $0 \sim 2^{64}-1$	
rune	有	与int32一样	$-2^{31} \sim 2^{31}-1$	等价 int32，表示一个 Unicode 码
byte	无	与 uint8 等价	0 ~ 255	当要存储字符时选用byte

```
//int , uint , rune , byte的使用
var a int = 8900
fmt.Println("a=", a)
var b uint = 1
var c byte = 255
fmt.Println("b=", b, "c=", c)
```

3.8.4整型的使用细节

1) Golang 各整数类型分：有符号和无符号，int uint 的大小和系统有关。

2) Golang 的整型默认声明为 int 型

```
//整型的使用细节
var n1 = 100 // ? n1 是什么类型
//这里我们给介绍一下如何查看某个变量的数据类型
//fmt.Printf() 可以用于做格式化输出。
fmt.Printf("n1 的 类型 %T \n", n1)
```

3) 如何在程序查看某个变量的字节大小和数据类型 （使用较多）

```
//如何在程序查看某个变量的占用字节大小和数据类型 （使用较多）
var n2 int64 = 10
//unsafe.Sizeof(n1) 是unsafe包的一个函数，可以返回n1变量占用的字节数
fmt.Printf("n2 的 类型 %T  n2占用的字节数是 %d ", n2, unsafe.Sizeof(n2))
```

4) Golang 程序中整型变量在使用时，遵守保小不保大的原则，即：在保证程序正确运行下，尽量使用占用空间小的数据类型。【如：年龄】

```
//Golang程序中整型变量在使用时，遵守保小不保大的原则，
//即：在保证程序正确运行下，尽量使用占用空间小的数据类型
var age byte = 90
```

5) bit: 计算机中的最小存储单位。byte:计算机中基本存储单元。[二进制再详细说] 1byte = 8 bit

3.9 小数类型/浮点型

3.9.1基本介绍

小数类型就是用于存放小数的，比如 1.2 0.23 -1.911

3.9.2案例演示

```
package main
import (
    "fmt"
)

//演示golang中小数类型使用
func main() {

    var price float32 = 89.12
    fmt.Println("price=", price)
}
```

3.9.3小数类型分类

类 型	占用存储空间	表数范围
单精度float32	4字节	-3.403E38 ~ 3.403E38
双精度float64	8字节	-1.798E308 ~ 1.798E308

对上图的说明:

1) 关于浮点数在机器中存放形式的简单说明,浮点数=符号位+指数位+尾数位

说明：浮点数都是有符号的.

```
//演示golang中小数类型使用
func main() {

    var price float32 = 89.12
    fmt.Println("price=", price)
    var num1 float32 = -0.00089
    var num2 float64 = -7809656.09
    fmt.Println("num1=", num1, "num2=", num2)
}
```

2) 尾数部分可能丢失，造成精度损失。 -123.0000901

```
//尾数部分可能丢失，造成精度损失。 -123.0000901
var num3 float32 = -123.0000901
var num4 float64 = -123.0000901
fmt.Println("num3=", num3, "num4=", num4)
```

```
D:\goproject\src\go_code\chapter03\floatdemo08>go run main.go
price= 89.12
num1= -0.00089 num2= -7.80965609e+06
num3= -123.00009 num4= -123.0000901
```

说明：float64 的精度比 float32 的要准确.

说明：如果我们要保存一个精度高的数，则应该选用 float64

3) 浮点型的存储分为三部分：符号位+指数位+尾数位 在存储过程中，精度会有丢失

3.9.4浮点型使用细节

1) Golang 浮点类型有固定的范围和字段长度，不受具体 OS(操作系统)的影响。

2) Golang 的浮点型默认声明为 float64 类型。

```
//Golang 的浮点型默认声明为float64 类型
var num5 = 1.1
fmt.Printf("num5的数据类型是 %T \n", num5)
```

3) 浮点型常量有两种表示形式

十进制数形式：如：5.12　　.512　(必须有小数点）

科学计数法形式:如：5.1234e2　= 5.12 * 10 的 2 次方　　5.12E-2　= 5.12/10 的 2 次方

```
//十进制数形式：如：5.12       .512   （必须有小数点）
num6 := 5.12
num7 := .123 //=> 0.123
fmt.Println("num6=", num6, "num7=", num7)

//科学计数法形式
num8 := 5.1234e2 // ? 5.1234 * 10的2次方
num9 := 5.1234E2 // ? 5.1234 * 10的2次方 shift+alt+向下的箭头
num10 := 5.1234E-2 // ? 5.1234 / 10的2次方 0.051234
```

4) 通常情况下，应该使用 float64 ，因为它比 float32 更精确。[开发中，**推荐使用 float64**]

3.10 字符类型

3.10.1 基本介绍

Golang 中没有专门的字符类型，如果要存储单个**字符**(字母)，一般使用 **byte 来**保存。

字符串就是一串固定长度的字符连接起来的字符序列。Go 的字符串是由单个字节连接起来的。也就是说对于传统的字符串是由字符组成的，而 **Go 的字符串不同**，它是由**字节**组成的。

3.10.2 案例演示

```go
//演示golang中字符类型使用
func main() {

    var c1 byte = 'a'
    var c2 byte = '0' //字符的0

    //当我们直接输出byte值，就是输出了的对应的字符的码值
    // 'a' ==>
    fmt.Println("c1=", c1)
    fmt.Println("c2=", c2)
    //如果我们希望输出对应字符，需要使用格式化输出
    fmt.Printf("c1=%c c2=%c\n", c1, c2)

    //var c3 byte = '北' //overflow溢出
    var c3 int = '北' //overflow溢出
    fmt.Printf("c3=%c c3对应码值=%d", c3, c3)

}
```

对上面代码说明

1) 如果我们保存的字符在 ASCII 表的,比如[0-1, a-z,A-Z..]直接可以保存到 byte

2) 如果我们保存的字符对应码值大于 255,这时我们可以考虑使用 int 类型保存

3) 如果我们需要安装字符的方式输出，这时我们需要格式化输出，即 fmt.Printf(“%c”, c1)..

3.10.3 字符类型使用细节

1) 字符常量是用单引号(')括起来的单个字符。例如：var c1 byte = 'a' var c2 int = '中' var c3 byte = '9'

2) Go 中允许使用转义字符 '\' 来将其后的字符转变为特殊字符型常量。例如：var c3 char = ‘\n’ // '\n'表示换行符

3) Go 语言的字符使用 UTF-8 编码，如果想查询字符对应的 utf8 码值 http://www.mytju.com/classcode/tools/encode_utf8.asp

英文字母-1 个字节 汉字-3 个字节

4) 在 Go 中，字符的本质是一个整数，直接输出时，是该字符对应的 UTF-8 编码的码值。

5) 可以直接给某个变量赋一个数字，然后按格式化输出时%c，会输出该数字对应的 unicode 字符

```
//可以直接给某个变量赋一个数字，然后按格式化输出时%c，会输出该数字对应的unicode 字符
var c4 int = 22269 // 22269 -> '国' 120->'x'
fmt.Printf("c4=%c\n", c4)
```

6) **字符类型是可以进行运算的**，相当于一个整数，因为它都对应有 Unicode 码.

```
//字符类型是可以进行运算的，相当于一个整数,运输时是按照码值运行
var n1 = 10 + 'a' //  10 + 97 = 107
fmt.Println("n1=", n1)
```

3.10.4 字符类型本质探讨

1) 字符型 存储到 计算机中，需要将字符对应的码值（整数）找出来

   存储：字符--->对应码值---->二进制-->存储

   读取：二进制----> 码值 ----> 字符 --> 读取

2) 字符和码值的对应关系是通过字符编码表决定的(是规定好)

3) Go 语言的编码都统一成了 utf-8。非常的方便，很统一，再也没有编码乱码的困扰了

3.11 布尔类型

3.11.1 基本介绍

1) 布尔类型也叫 bool 类型，bool 类型数据只允许取值 true 和 false

2) bool 类型占 1 个字节。

3) bool 类型适于**逻辑运算**，一般用于程序流程控制[注：这个后面会详细介绍]：

- if 条件控制语句；
- for 循环控制语句

3.11.2 案例演示

```go
//演示golang中bool类型使用
func main() {
    var b = false
    fmt.Println("b=", b)
    //注意事项
    //1. bool类型占用存储空间是1个字节
    fmt.Println("b 的占用空间 =", unsafe.Sizeof(b) )
    //2. bool类型只能取true或者false

}
```

3.12 string 类型

3.12.1 基本介绍

字符串就是一串固定长度的字符连接起来的字符序列。Go 的字符串是由单个字节连接起来的。Go 语言的字符串的字节使用 UTF-8 编码标识 Unicode 文本

3.12.2 案例演示

```go
//演示golang中string类型使用
func main() {
    //string的基本使用
    var address string = "北京长城 110 hello world!"
    fmt.Println(address)
}
```

3.12.3 string 使用注意事项和细节

1) Go 语言的字符串的字节使用 UTF-8 编码标识 Unicode 文本，这样 Golang 统一使用 UTF-8 编码,中文乱码问题不会再困扰程序员。

2) 字符串一旦赋值了，字符串就不能修改了：在 Go 中字符串是不可变的。

```go
//字符串一旦赋值了，字符串就不能修改了：在Go中字符串是不可变的
var str = "hello"
str[0] = 'a' //这里就不能去修改str的内容，即go中的字符串是不可变的。
```

3) 字符串的两种表示形式

(1) 双引号，会识别转义字符

(2) 反引号，以字符串的原生形式输出，包括换行和特殊字符，可以实现防止攻击、输出源代码等效果

【案例演示】

```
//输出源代码等效果 【案例演示】
str2 := "abc\nabc"
fmt.Println(str2)

//使用的反引号 ``
str3 := `
package main
import (
    "fmt"
    "unsafe"
)

//演示golang中bool类型使用
func main() {
    var b = false
    fmt.Println("b=", b)
    //注意事项
    //1. bool类型占用存储空间是1个字节
    fmt.Println("b 的占用空间 =", unsafe.Sizeof(b) )
    //2. bool类型只能取true或者false

}
`
fmt.Println(str3)
```

4) 字符串拼接方式

```
//字符串拼接方式
var str = "hello " + "world"
str += " haha!"
```

5) 当一行字符串太长时，需要使用到多行字符串，可以如下处理

```
fmt.Println(str)
//当一个拼接的操作很长时，怎么办，可以分行写,但是注意，需要将+保留在上一行.
str4 := "hello " + "world" + "hello " + "world" + "hello " +
"world" + "hello " + "world" + "hello " + "world" +
"hello " + "world"
fmt.Println(str4)
```

3.13 基本数据类型的默认值

3.13.1 基本介绍

在 go 中，数据类型都有一个默认值，当程序员没有赋值时，就会保留默认值，在 go 中，默认值又叫零值。

3.13.2 基本数据类型的默认值如下

数据类型	默认值
整型	0
浮点型	0
字符串	""
布尔类型	false

案例:

```
var a int // 0
var b float32 // 0
var c float64 // 0
var isMarried bool // false
var name string // ""
//这里的%v 表示按照变量的值输出
fmt.Printf("a=%d,b=%v,c=%v,isMarried=%v name=%v",a,b,c,isMarried, name)
```

3.14 基本数据类型的相互转换

3.14.1 基本介绍

Golang 和 java / c 不同，Go 在不同类型的变量之间赋值时**需要显式转换**。也就是说 Golang 中数据类型**不能自动转换**。

3.14.2 基本语法

表达式 T(v) 将值 v 转换为类型 T

T: 就是数据类型，比如 int32，int64，float32 等等

v: 就是需要转换的变量

3.14.3 案例演示

```go
package main
import (
    "fmt"
)

//演示golang中基本数据类型的转换
func main() {

    var i int32 = 100
    //希望将 i => float
    var n1 float32 = float32(i)
    var n2 int8 = int8(i)
    var n3 int64 = int64(i) //低精度->高精度

    fmt.Printf("i=%v n1=%v n2=%v n3=%v", i ,n1, n2, n3)

}
```

3.14.4 基本数据类型相互转换的注意事项

1) Go 中，数据类型的转换可以是从 表示范围小-->表示范围大，也可以 范围大--->范围小

2) 被转换的是变量存储的数据(即值)，变量本身的数据类型并没有变化！

```go
func main() {

    var i int32 = 100
    //希望将 i => float
    var n1 float32 = float32(i)
    var n2 int8 = int8(i)
    var n3 int64 = int64(i) //低精度->高精度

    fmt.Printf("i=%v n1=%v n2=%v n3=%v \n", i ,n1, n2, n3)


    //被转换的是变量存储的数据(即值)，变量本身的数据类型并没有变化
    fmt.Printf("i type is %T\n", i) // int32
```

3) 在转换中，比如将 int64 转成 int8 【-128---127】，编译时不会报错，只是转换的结果是**按溢出处理**，和我们希望的结果不一样。 因此在转换时，需要考虑范围.

```go
//在转换中，比如将 int64  转成 int8 【-128---127】 ，编译时不会报错，
//只是转换的结果是按溢出处理，和我们希望的结果不一样
var num1 int64 = 999999
var num2 int8 = int8(num1) //
fmt.Println("num2=", num2)  ——→ num2=63
```

3.14.5 课堂练习

➢ 练习 1

```go
func main() {

    //课堂练习 , tab键
    var n1 int32 = 12
    var n2 int64
    var n3 int8

    n2 = n1 + 20  //int32 ---> int64 错误 ✗
    n3 = n1 + 20  //int32 ---> int8 错误 ✗

}
```

如何修改上面的代码，就可以正确.

```go
func main() {

    //课堂练习 , tab键
    var n1 int32 = 12
    var n2 int64
    var n3 int8

    n2 = int64(n1) + 20 ✓ //int32 ---> int64 错误
    n3 = int8(n1) + 20 ✓ //int32 ---> int8 错误
    fmt.Println("n2=", n2, "n3=", n3)

}
```

➢ 练习 2

```go
var n1 int32 = 12
var n3 int8
var n4 int8
n4 = int8(n1) + 127  //【编译通过，但是 结果 不是127+12 ,按溢出处理】
n3 = int8(n1) + 128 //【编译不过】
fmt.Println(n4)
```

3.15 基本数据类型和 string 的转换

3.15.1 基本介绍

在程序开发中，我们经常将基本数据类型转成 string,或者将 string 转成基本数据类型。

3.15.2 基本类型转 string 类型

- 方式 1：fmt.Sprintf("%参数", 表达式) **【个人习惯这个，灵活】**

  函数的介绍：

  func Sprintf

  ```go
  func Sprintf(format string, a ...interface{}) string
  ```

  Sprintf根据format参数生成格式化的字符串并返回该字符串。

  参数需要和表达式的数据类型相匹配

  fmt.Sprintf().. 会返回转换后的字符串

- 案例演示

```go
//演示golang中基本数据练习转成string使用
func main() {

    var num1 int = 99
    var num2 float64 = 23.456
    var b bool = true
    var myChar byte = 'h'
    var str string //空的str

    //使用第一种方式来转换 fmt.Sprintf方法

    str = fmt.Sprintf("%d", num1)
    fmt.Printf("str type %T str=%q\n", str, str)

    str = fmt.Sprintf("%f", num2)
    fmt.Printf("str type %T str=%q\n", str, str)

    str = fmt.Sprintf("%t", b)
    fmt.Printf("str type %T str=%q\n", str, str)

    str = fmt.Sprintf("%c", myChar)
    fmt.Printf("str type %T str=%q\n", str, str)

}
```

➢ 方式 2：使用 strconv 包的函数

```go
func FormatBool(b bool) string
func FormatFloat(f float64, fmt byte, prec, bitSize int) string
func FormatInt(i int64, base int) string
func FormatUint(i uint64, base int) string
```

➢ 案例说明

```go
    //第二种方式 strconv 函数
    var num3 int = 99
    var num4 float64 = 23.456
    var b2 bool = true

    str = strconv.FormatInt(int64(num3), 10)
    fmt.Printf("str type %T str=%q\n", str, str)

    // strconv.FormatFloat(num4, 'f', 10, 64)
    // 说明： 'f' 格式 10：表示小数位保留10位 64 :表示这个小数是float64
    str = strconv.FormatFloat(num4, 'f', 10, 64)
    fmt.Printf("str type %T str=%q\n", str, str)

    str = strconv.FormatBool(b2)
    fmt.Printf("str type %T str=%q\n", str, str)
```

```go
    //strconv包中有一个函数Itoa
    var num5 int64 = 4567
    str = strconv.Itoa(int(num5))
    fmt.Printf("str type %T str=%q\n", str, str)
```

3.15.3 string 类型转基本数据类型

➢ 使用时 strconv 包的函数

```
func ParseBool(str string) (value bool, err error)
func ParseFloat(s string, bitSize int) (f float64, err error)
func ParseInt(s string, base int, bitSize int) (i int64, err error)
func ParseUint(s string, b int, bitSize int) (n uint64, err error)
```

➢ 案例演示

```go
func main() {

    var str string = "true"
    var b bool
    // b, _ = strconv.ParseBool(str)
    // 说明
    // 1. strconv.ParseBool(str) 函数会返回两个值 (value bool, err error)
    // 2. 因为我只想获取到 value bool ,不想获取 err 所以我使用_忽略
    b , _ = strconv.ParseBool(str)
    fmt.Printf("b type %T  b=%v\n", b, b)

    var str2 string = "1234590"
    var n1 int64
    var n2 int
    n1, _ = strconv.ParseInt(str2, 10, 64)
    n2 = int(n1)
    fmt.Printf("n1 type %T  n1=%v\n", n1, n1)
    fmt.Printf("n2 type %T n2=%v\n", n2, n2)

    var str3 string = "123.456"
    var f1 float64
    f1, _ = strconv.ParseFloat(str3, 64)
    fmt.Printf("f1 type %T f1=%v\n", f1, f1)

}
```

➢ 说明一下

note：因为返回的是 int64 或者 float64，如希望要得到int32 ,float32等如下处理：

```go
//如果希望将str->int32的可以这样处理
var num5 int32
num5 = int32(num)
```

int16 ...

3.15.4 string 转基本数据类型的注意事项

在将 String 类型转成 基本数据类型时，**要确保 String 类型能够转成有效的数据**，比如 我们可以把 "123"，转成一个整数，但是不能把 "hello" 转成一个整数，如果这样做，Golang 直接将其转成 0 ，其它类型也是一样的道理. float => 0 bool => false

案例说明：

```
//注意:
var str4 string = "hello"
var n3 int64 = 11
n3, _ = strconv.ParseInt(str4, 10, 64)
fmt.Printf("n3 type %T n3=%v\n", n3, n3)
```

如果没有转成功，n3 = 0 //默认值

3.16 指针

3.16.1 基本介绍

1) 基本数据类型，变量存的就是值，也叫值类型
2) 获取变量的地址，用&，比如： var num int, 获取 num 的地址：&num

   分析一下基本数据类型在内存的布局.

3) 指针类型，指针变量存的是一个地址，这个地址指向的空间存的才是值

   比如：var ptr *int = &num

   举例说明：指针在内存的布局.

4) 获取指针类型所指向的值，使用：*，比如：var ptr *int, 使用*ptr 获取 ptr 指向的值

```go
//演示golang中指针类型
func main() {

    //基本数据类型在内存布局
    var i int = 10
    // i 的地址是什么,&i
    fmt.Println("i的地址=", &i)

    //下面的 var ptr *int = &i
    //1. ptr 是一个指针变量
    //2. ptr 的类型 *int
    //3. ptr 本身的值&i
    var ptr *int = &i
    fmt.Printf("ptr=%v\n", ptr)
    fmt.Printf("ptr 的地址=%v", &ptr)
    fmt.Printf("ptr 指向的值=%v", *ptr)

}
```

5) 一个案例再说明

3.16.2 案例演示

1) 写一个程序，获取一个 int 变量 num 的地址，并显示到终端

2) 将 num 的地址赋给指针 ptr，并通过 ptr 去修改 num 的值.

3.16.3 指针的课堂练习

3.16.4 指针的使用细节

1) 值类型，都有**对应的指针类型**， 形式为 ***数据类型**，比如 int 的对应的指针就是 *int, float32 对应的指针类型就是 *float32, 依次类推。

2) 值类型包括：基本数据类型 **int 系列, float 系列, bool, string 、数组**和**结构体** struct

3.17 值类型和引用类型

3.17.1 值类型和引用类型的说明

1) 值类型：基本数据类型 int 系列, float 系列, bool, string 、数组和结构体 struct

2) 引用类型：指针、slice 切片、map、管道 chan、interface 等都是引用类型

3.17.2 值类型和引用类型的使用特点

1) 值类型：变量直接存储值，内存通常在栈中分配

示意图：

2) 引用类型：变量存储的是一个地址，这个地址对应的空间才真正存储数据(值)，内存通常在堆上分配，当没有任何变量引用这个地址时，该地址对应的数据空间就成为一个垃圾，由 GC 来回收

示意图：

3) 内存的栈区和堆区示意图

3.18 标识符的命名规范

3.18.1 标识符概念

1) Golang 对各种变量、方法、函数等命名时使用的字符序列称为标识符

2) 凡是自己可以起名字的地方都叫标识符

3.18.2 标识符的命名规则

1) 由 26 个英文字母大小写，0-9 ，_ 组成

2) 数字不可以开头。var num int //ok　　var 3num int //error

3) Golang 中严格区分大小写。

var num int

var Num int

说明：在 golang 中，num 和 Num 是两个不同的变量

4) 标识符不能包含空格。

```
//标识符不能包含空格
var ab c int = 30
```

5) 下划线"_"本身在 Go 中是一个特殊的标识符，称为空标识符。可以代表任何其它的标识符，但是它对应的值会被忽略(比如：忽略某个返回值)。所以**仅能被作为占位符使用，不能作为标识符使用**

```
//_ 是空标志符，用于占用
var _ int = 40 //error
fmt.Println(_)
```

6) 不能以系统**保留关键字**作为标识符（一共有 25 个），比如 break，if 等等...

3.18.3 标识符的案例

hello　// ok

hello12 //ok

1hello // error ,不能以数字开头

h-b　// error ,不能使用 -

x h　// error, 不能含有空格

h_4　// ok

_ab　// ok

int　// ok，我们要求大家不要这样使用

float32 // ok，我们要求大家不要这样使用

_　// error

Abc　// ok

3.18.4 标识符命名注意事项

1) 包名：保持 package 的名字和目录保持一致，尽量采取有意义的包名，简短，有意义，不要和标准库不要冲突 fmt

2) 变量名、函数名、常量名：采用驼峰法

举例：

var stuName string = “tom”　　形式: xxxYyyyyZzzz ...

var goodPrice float32 = 1234.5

3) 如果变量名、函数名、常量名首字母大写，则可以被其他的包访问；如果首字母小写，则只能在本包中使用（注：可以简单的理解成，**首字母大写是公开的**，首字母小写是私有的）,在 golang **没有** public , private 等关键字。

案例演示：

3.19 系统保留关键字

保留关键字介绍

在Go中，为了简化代码编译过程中对代码的解析，其定义的**保留关键字**只有25个。详见如下

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

3.20 系统的预定义标识符

预定义标识符介绍

除了保留关键字外，Go还提供了36个预定的标识符，其包括基础数据类型和**系统内嵌函数**

append	bool	byte	cap	close	complex
complex64	complex128	uint16	copy	false	float32
float64	imag	int	int8	int16	uint32
int32	int64	iota	len	make	new
nil	panic	uint64	print	println	real
recover	string	true	uint	uint8	uintprt

第 4 章 运算符

4.1 运算符的基本介绍

运算符是一种特殊的符号，用以表示数据的运算、赋值和比较等

运算符是一种特殊的符号，用以表示数据的运算、赋值和比较等

1) 算术运算符
2) 赋值运算符
3) 比较运算符/关系运算符
4) 逻辑运算符
5) 位运算符
6) 其它运算符

4.2 算术运算符

算术运算符是对数值类型的变量进行运算的，比如：加减乘除。在 Go 程序中使用的非常多

4.2.1算术运算符的一览表

运算符	运算	范例	结果
+	正号	+3	3
-	负号	-4	-4
+	加	5 + 5	10
-	减	6 - 4	2
*	乘	3 * 4	12
/	除	5 / 5	1
%	取模(取余)	7 % 5	2
++	自增	a=2 a++	a=3
- -	自减	a=2 a--	a=1
+	字符串相加	"He" + "llo"	"Hello"

4.2.2案例演示

- 案例演示算术运算符的使用。

  +, - , * , / , %, ++, --　，重点讲解 /、%

  自增：++

  自减：--

- 演示 / 的使用的特点

```
//重点讲解 /、%
//说明，如果运算的数都是整数，那么除后，去掉小数部分，保留整数部分
fmt.Println(10 / 4)

var n1 float32 = 10 / 4 //
fmt.Println(n1)

//如果我们希望保留小数部分，则需要有浮点数参与运算
var n2 float32 = 10.0 / 4
fmt.Println(n2)
```

- 演示 % 的使用特点

  // 演示　% 的使用

  // 看一个公式 a % b = a - a / b * b

  fmt.Println("10%3=", 10 % 3) // =1

  fmt.Println("-10%3=", -10 % 3) // = -10 - (-10) / 3 * 3 = -10 - (-9) = -1

  fmt.Println("10%-3=", 10 % -3) // =1

  fmt.Println("-10%-3=", -10 % -3) // =-1

- ++ 和 --的使用

```
// ++ 和 --的使用
var i int = 10
i++ // 等价 i = i + 1
fmt.Println("i=", i) // 11
i-- // 等价 i = i - 1
fmt.Println("i=", i) // 10
```

4.2.3算术运算符使用的注意事项

1) 对于除号 "/"，它的整数除和小数除是有区别的：整数之间做除法时，只保留整数部分而舍弃小数部分。 例如： x := 19/5 ,结果是 3

2) 当对一个数取模时，可以等价 a%b=a-a/b*b ， 这样我们可以看到 取模的一个本质运算。

3) Golang 的自增自减只能当做一个独立语言使用时，不能这样使用

```go
package main
import (
    "fmt"
)
func main() {

    //在golang中，++ 和 -- 只能独立使用.
    var i int = 8
    var a int
    a = i++ //错误，i++只能独立使用
    a = i-- //错误，i--只能独立使用

    if i++ > 0 {
        fmt.Println("ok")
    }
}
```

4) Golang 的++ 和 -- 只能写在变量的后面，不能写在变量的前面，即：只有 a++ a-- 没有 ++a --a

```go
var i int = 1
i++
++i // 错误，在golang没有 前++
i--
--i // 错误，在golang没有 前--
fmt.Println("i=", i)
```

5) Golang 的设计者去掉 c / java 中的 自增自减的容易混淆的写法，让 Golang 更加简洁，统一。(强制性的)

4.2.4课堂练习 1

```
var i int = 1
i = i++
fmt.Println(i);
//问：结果是多少？为什么？
//上面的代码时错误，编译不通过,i = i++
```

```
var i int = 10
if i++ > 10 {
fmt.Println("ok")
}
//问：结果是多少？为什么？
//上面的代码时错误，编译不通过,i++ >10
```

4.2.5课堂练习 2

1) 假如还有 97 天放假，问：xx 个星期零 xx 天

2) 定义一个变量保存华氏温度，华氏温度转换摄氏温度的公式为：5/9*(华氏温度-100),请求出华氏温度对应的摄氏温度。

```go
package main
import (
    "fmt"
)
func main() {

    //假如还有97天放假，问：xx个星期零xx天
    var days int = 97
    var week int = days / 7
    var day int = days % 7
    fmt.Printf("%d个星期零%d天\n", week, day)


    //定义一个变量保存华氏温度，华氏温度转换摄氏温度的公式为：
    //5/9*(华氏温度-100),请求出华氏温度对应的摄氏温度
    var huashi float32 = 134.2
    var sheshi float32 = 5.0 / 9 * (huashi - 100)
    fmt.Printf("%v 对应的摄氏温度=%v \n", huashi, sheshi)
}
```

4.3 关系运算符(比较运算符)

4.3.1基本介绍

1) 关系运算符的结果都是 bool 型，也就是要么是 true，要么是 false

2) 关系表达式 经常用在 **if 结构**的条件中或**循环结构**的条件中

4.3.2关系运算符一览图

运算符	运算	范例	结果
==	相等于	4==3	false
!=	不等于	4!=3	true
<	小于	4<3	false
>	大于	4>3	true
<=	小于等于	4<=3	false
>=	大于等于	4>=3	true

4.3.3案例演示

```go
package main
import (
    "fmt"
)
func main() {
    //演示关系运算符的使用
    var n1 int = 9
    var n2 int = 8
    fmt.Println(n1 == n2) //false
    fmt.Println(n1 != n2) //true
    fmt.Println(n1 > n2) //true
    fmt.Println(n1 >= n2) //true
    fmt.Println(n1 < n2) //flase
    fmt.Println(n1 <= n2) //flase
    flag := n1 > n2
    fmt.Println("flag=", flag)
}
```

4.3.4关系运算符的细节说明

细节说明

1) 关系运算符的结果都是 bool 型，也就是要么是 true，要么是 false。
2) 关系运算符组成的表达式，我们称为关系表达式： a > b
3) 比较运算符"=="不能误写成 "=" !!

4.4 逻辑运算符

4.4.1基本介绍

用于连接多个条件（一般来讲就是关系表达式），最终的结果也是一个 bool 值

4.4.2逻辑运算的说明

假定 A 值为 True，B 值为 False

运算符	描述	实例
&&	**逻辑与**运算符。如果两边的操作数**都是 True**，则为 True，否则为 False。	(A && B) 为 False
||	**逻辑或**运算符。如果两边的操作数有一个 True，则为 True，否则为 False。	(A || B) 为 True
!	**逻辑非**运算符。如果条件为 True，则逻辑 为 False，否则为 True。	!(A && B) 为 True

4.4.3案例演示

```go
//演示逻辑运算符的使用  &&
var age int = 40
if age > 30 && age < 50 {
    fmt.Println("ok1")
}

if age > 30 && age < 40 {
    fmt.Println("ok2")
}
```

```go
//演示逻辑运算符的使用  ||

if age > 30 || age < 50 {
    fmt.Println("ok3")
}

if age > 30 || age < 40 {
    fmt.Println("ok4")
}
```

```go
    //演示逻辑运算符的使用  !

    if age > 30 {
        fmt.Println("ok5")
    }

    if !(age > 30) {
        fmt.Println("ok6")
    }
```

4.4.4注意事项和细节说明

1) &&也叫**短路与**：**如果第一个条件为 false**，则**第二个条件不会判断**，最终结果为 **false**

2) ||也叫**短路或**：如果**第一个条件为 true**，则**第二个条件不会判断**，最终结果为 **true**

3) 案例演示

```go
//声明一个函数(测试)
func test() bool {
    fmt.Println("test....")
    return true
}

func main() {

    var i int = 10
    //短路与
    //说明 因为  i < 9 为 false ,因此后面的 test() 就不执行
    if i < 9 && test() {
        fmt.Println("ok...")
    }

    //说明 因为  i > 9 为 true ,因此后面的 test() 就不执行
    if i > 9 || test() {
        fmt.Println("hello...")
    }
```

4.5 赋值运算符

4.5.1基本的介绍

赋值运算符就是将某个运算后的值，赋给指定的变量。

4.5.2赋值运算符的分类

运算符	描述	实例
=	简单的赋值运算符，将一个表达式的值赋给一个左值	C = A + B 将 A + B 表达式结果赋值给 C
+=	相加后再赋值	C += A 等于 C = C + A
-=	相减后再赋值	C -= A 等于 C = C - A
*=	相乘后再赋值	C *= A 等于 C = C * A
/=	相除后再赋值	C /= A 等于 C = C / A
%=	求余后再赋值	C %= A 等于 C = C % A

运算符	描述	实例
<<=	左移后赋值	C <<= 2 等于 C = C << 2
>>=	右移后赋值	C >>= 2 等于 C = C >> 2
&=	按位与后赋值	C &= 2 等于 C = C & 2
^=	按位异或后赋值	C ^= 2 等于 C = C ^ 2
|=	按位或后赋值	C |= 2 等于 C = C | 2

说明：这部分的赋值运算涉及到二**进制**相关知识，我们放在讲二进制的时候再回头讲解

4.5.3赋值运算的案例演示

案例演示赋值运算符的基本使用。

1) 赋值基本案例
2) 有两个变量，a 和 b，要求将其进行交换，最终打印结果
3) += 的使用案例
4) 案例

```go
func main() {
    //赋值运算符的使用演示
    // var i int
    // i = 10 //基本赋值

    //有两个变量，a和b，要求将其进行交换，最终打印结果
    // a = 9 , b = 2 ==> a = 2 b = 9
    a := 9
    b := 2
    fmt.Printf("交换前的情况是 a = %v , b=%v \n", a, b)
    //定义一个临时变量
    t := a
    a = b //
    b = t //
    fmt.Printf("交换后的情况是 a = %v , b=%v \n", a, b)

    //复合赋值的操作
    a += 17 // 等价 a = a + 17
    fmt.Println("a=", a)

}
```

4.5.4赋值运算符的特点

1) 运算顺序从右往左

```go
var c int
c = a + 3 // 赋值运算的执行顺序是从右向左
```

2) 赋值运算符的左边 只能是**变量**,右边 可以是**变量、表达式、常量值**

```go
//2)赋值运算符的左边 只能是变量,右边 可以是变量、表达式、常量值
// 表达式：任何有值都可以看做表达式
var d int
d = a //
d = 8 + 2 * 8 // =的右边是表达式
d = test() + 90 //  =的右边是表达式
d = 890 // 890常量
fmt.Println(d)
```

3) 复合赋值运算符等价于下面的效果

比如：a += 3 等价于 a = a + 3

4.5.5面试题

有两个变量，a 和 b，要求将其进行交换，但是不允许使用中间变量，最终打印结果

```go
package main
import (
    "fmt"
)
func main() {

    //有两个变量，a和b，要求将其进行交换，但是不允许使用中间变量，最终打印结果
    var a int = 10
    var b int = 20

    a = a + b //
    b = a - b // b = a + b - b ==> b = a
    a = a - b // a = a + b - a ==> a = b

    fmt.Printf("a=%v b=%v", a, b)
}
```

4.6 位运算符

运算符	描述
&	按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。 运算规则是: 同时为1 , 结果为1, 否则为0
|	按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或 运算规则是: 有一个为1 , 结果为1, 否则为0
^	按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或. 运算规则是: 当二进位不同时, 结果为1, 否则为0
<<	左移运算符"<<"是双目运算符。 其功能把"<<"左边的运算数的各二进位全部左移若干位，高位丢弃，低位补0。左移n位就是乘以2的n次方。
>>	右移运算符">>"是双目运算符。 其功能是把">>"左边的运算数的各二进位全部右移若干位 右移n位就是除以2的n次方 **说明：因为位运算涉及到二进制相关知识，我们仍然放到讲二进制时，在详细讲解**

4.7 其它运算符说明

运算符	描述	实例
&	返回变量存储地址	&a; 将给出变量的实际地址。
*	指针变量	*a; 是一个指针变量

举例说明：

```
:\goproject\src\go_code\chapter04\exec\main.go
import (
    "fmt"
)
func main() {
    //演示一把 & 和 *的使用

    a := 100
    fmt.Println("a 的地址=", &a)

    var ptr *int = &a
    fmt.Println("ptr 指向的值是=", *ptr)
}
```

4.7.1课堂案例

案例 1：求两个数的最大值

```
func main() {

    //求两个数的最大值
    var n1 int = 10
    var n2 int = 40
    var max int
    if n1 > n2 {
        max = n1
    } else {
        max = n2
    }
    fmt.Println("max=", max)

}
```

案例 2：求三个数的最大值

```go
func main() {

    //求两个数的最大值
    var n1 int = 10
    var n2 int = 40
    var max int
    if n1 > n2 {
        max = n1
    } else {
        max = n2
    }
    fmt.Println("max=", max)

    //求出三个数的最大值思路：先求出两个数的最大值，
    //然后让这个最大值和第三数比较，在取出最大值
    var n3 = 45
    if n3 > max {
        max = n3
    }
    fmt.Println("三个数中最大值是=", max)

}
```

4.8 特别说明

特别说明：

Go语言明确不支持三**元运算符**，官方说明：

https://golang.org/doc/faq#Does_Go_have_a_ternary_form

Does Go have the ?: operator?

There is no ternary testing operation in Go. You may use the following to achieve the same result:

```go
if expr {
    n = trueVal
} else {
    n = falseVal
}
```

Golang的设计理念：

一种事情有且只有一种方法完成

举例说明，如果在 golang 中实现三元运算的效果。

```go
    var n int
    var i int = 10
    var j int = 12
    //传统的三元运算
    //n = i > j ? i : j
    if i > j {
        n = i
    } else {
        n = j
    }
    fmt.Println("n=", n) // 12
}
```

4.9 运算符的优先级

4.9.1 运算符的优先级的一览表

高

分类	描述	关联性
后缀	() [] -> . ++ --	左到右
单目	+ - ! ~ (type) * & sizeof	右到左
乘法	* / %	左到右
加法	+ -	左到右
移位	<< >>	左到右
关系	< <= > >=	左到右
相等（关系）	== !=	左到右
按位AND	&	左到右
按位XOR	^	左到右
按位OR	|	左到右
逻辑AND	&&	左到右
逻辑OR	||	左到右
赋值运算符	= += -= *= /= %= >>= <<= &= ^= |=	右到左
逗号	,	左到右

低

4.9.2 对上图的说明

1) 运算符有不同的优先级，所谓优先级就是表达式运算中的运算顺序。如右表，上一行运算符总

优先于下一行。

2) 只有单目运算符、赋值运算符是从右向左运算的。

3) 梳理了一个大概的优先级

1：括号，++, --

2: 单目运算

3：算术运算符

4：移位运算

5：关系运算符

6：位运算符

7：逻辑运算符

8：赋值运算符

9：逗号

4.10 键盘输入语句

4.10.1 介绍

在编程中，需要接收用户输入的数据，就可以使用键盘输入语句来获取。InputDemo.go

4.10.2 步骤 ：

1) 导入 fmt 包

2) 调用 fmt 包的 **fmt.Scanln()** 或者 **fmt.Scanf()**

func Scanln

```
func Scanln(a ...interface{}) (n int, err error)
```

Scanln类似Scan，但会在换行时才停止扫描。最后一个条目后必须有换行或者到达结束位置。

func Scanf

```
func Scanf(format string, a ...interface{}) (n int, err error)
```

Scanf从标准输入扫描文本，根据format 参数指定的格式将成功读取的空白分隔的值保存进成功传递给本函数的参数。返回成功扫描的条目个数和遇到的任何错误。

4.10.3 案例演示：

要求：可以从控制台接收用户信息，【姓名，年龄，薪水，是否通过考试 】。

1) 使用 fmt.Scanln() 获取

```go
func main() {

    //要求：可以从控制台接收用户信息，【姓名，年龄，薪水，是否通过考试 】。
    //方式1 fmt.Scanln
    //1先声明需要的变量
    var name string
    var age byte
    var sal float32
    var isPass bool
    fmt.Println("请输入姓名 ")
    //当程序执行到 fmt.Scanln(&name),程序会停止在这里，等待用户输入，并回车
    fmt.Scanln(&name)
    fmt.Println("请输入年龄 ")
    fmt.Scanln(&age)
    fmt.Println("请输入薪水 ")
    fmt.Scanln(&sal)

    fmt.Println("请输入是否通过考试 ")
    fmt.Scanln(&isPass)

    fmt.Printf("名字是 %v \n 年龄是 %v \n 薪水是 %v \n 是否通过考试 %v \n", name, age,

}
```

2) 使用 fmt.Scanf() 获取

```go
    //方式2:fmt.Scanf,可以按指定的格式输入
    fmt.Println("请输入你的姓名，年龄，薪水，是否通过考试， 使用空格隔开")
    fmt.Scanf("%s %d %f %t", &name, &age, &sal, &isPass)
    fmt.Printf("名字是 %v \n年龄是 %v \n 薪水是 %v \n 是否通过考试 %v \n", name, age, sal,
```

4.11 进制

对于整数，有四种表示方式：

1) 二进制：0,1 ，满 2 进 1。

   在 golang 中，不能直接使用二进制来表示一个整数，它沿用了 c 的特点。

2) 十进制：0-9 ，满 10 进 1。

3) 八进制：0-7 ，满 8 进 1. 以数字 0 开头表示。

4) 十六进制：0-9 及 A-F，满 16 进 1. 以 0x 或 0X 开头表示。

   此处的 A-F 不区分大小写。

```go

func main() {

    var i int = 5
    //二进制输出
    fmt.Printf("%b \n", i)

    //八进制：0-7 ，满8进1. 以数字0开头表示
    var j int = 011 // 011=> 9
    fmt.Println("j=", j)

    //0-9及A-F，满16进1. 以0x或0X开头表示
    var k int = 0x11 // 0x11=> 16 + 1 = 17
    fmt.Println("k=", k)
}
```

4.11.1 进制的图示

进制的图示

十进制	十六进制	八进制	二进制
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	8	10	1000

进制的图示

十进制	十六进制	八进制	二进制
9	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111
16	10	20	10000
17	11	21	10001

4.11.2 进制转换的介绍

- **第一组（其它进制转十进制）：**
  1) 二进制转十进制
  2) 八进制转十进制
  3) 十六进制转十进制
  4) 示意图
- **第二组：(十进制转其它进制)**
  1) 十进制转二进制
  2) 十进制转八进制
  3) 十进制转十六进制
  4) 示意图
- **第三组 (二进制转其它进制)**
  1) 二进制转八进制
  2) 二进制转十六进制
  3) 示意图
- **第四组 (其它进制转二进制)**
  1) 八进制转二进制
  2) 十六进制转二进制
  3) 示意图

4.11.3 其它进制转十进制

4.11.4 二进制如何转十进制

134 = 4 * 1 + 3 * 10 + 1 * 10 * 10 = 4 + 30 + 100 = 134

规则：从最低位开始（右边的），将每个位上的数提取出来，乘以2的(位数-1)次方，然后求和。

案例：请将二进制：1011 转成十进制的数

1011 = 1 * 1 + 1 * 2 + 0 * 2 * 2 + 1 * 2 * 2 * 2 = 1 + 2 + 0 + 8 = 11

4.11.5 八进制转换成十进制示例

规则：从最低位开始（右边的），将每个位上的数提取出来，乘以8的(位数-1)次方，然后求和。

案例：请将 0123 转成十进制的数

0123 = 3 * 1 + 2 * 8 + 1 * 8 * 8 = 3 + 16 + 64 = 83

4.11.6 16 进制转成 10 进制

规则：从最低位开始，将每个位上的数提取出来，乘以16的(位数-1)次方，然后求和。

案例：请将 0x34A 转成十进制的数

0x34A = 10 * 1 + 4 * 16 + 3 * 16 * 16 = 10 + 64 + 768 = 842

4.11.7 其它进制转 10 进制的课堂练习

- 课堂练习：请将

二进制： 110001100 转成 十进制

八进制： 02456 转成十进制

十六进制： 0xA45 转成十进制

4.11.8 十进制如何转成其它进制

4.11.9 十进制如何转二进制

规则：将该数不断除以2，直到商为0为止，然后将每步得到的余数倒过来，就是对应的二进制。

案例：请将 56 转成二进制

4.11.10 十进制转成八进制

规则：将该数不断除以8，直到商为0为止，然后将每步得到的余数倒过来，就是对应的八进制。

案例：请将 156 转成八进制

4.11.11 十进制转十六进制

规则：将该数不断除以16，直到商为0为止，然后将每步得到的余数倒过来，就是对应的十六进制。

案例：请将 356 转成十六进制

课堂练习：请将
123 转成 二进制
678 转成八进制
8912 转成十六进制

4.11.12 课堂练习

课堂练习：请将

123 转成 二进制

678 转成八进制

8912 转成十六进制

4.11.13 二进制转换成八进制、十六进制

4.11.14 二进制转换成八进制

规则：将二进制数每三位一组(从低位开始组合)，转成对应的八进制数即可。
案例：请将 二进制：11010101 转成八进制
11010101 = 0325

4.11.15 二进制转成十六进制

规则：将二进制数每四位一组(从低位开始组合)，转成对应的十六进制数即可。
案例：请将 二进制：11010101 转成十六进制
11010101 = 0xD5

- 课堂练习

课堂练习：请将

二进制：11100101 转成 八进制

二进制：1110010110 转成 十六进制

4.11.16 八进制、十六进制转成二进制

4.11.17 八进制转换成二进制

规则：将八进制数每1位，转成对应的一个3位的二进制数即可。
案例：请将 0237 转成二进制
0237 = 10011111

4.11.18 十六进制转成二进制

规则：将十六进制数每1位，转成对应的一个4位的二进制数即可。
案例：请将 0x237 转成二进制
0x237 = 1000110111

4.12 位运算

4.12.1 位运算的思考题

1) 请看下面的代码段，回答 a,b,c,d 结果是多少?

```
func main() {

    var a int = 1 >> 2
    var b int = -1 >> 2
```

```
    var c int = 1 << 2
    var d int = -1 << 2
    //a,b,c,d 结果是多少
    fmt.Println("a=", a)
    fmt.Println("b=", b)
    fmt.Println("c=", c)
    fmt.Println("d=", d)

}
```

2) 请回答在 Golang 中，下面的表达式运算的结果是:

```
func main() {

    fmt.Println(2&3)
    fmt.Println(2|3)
    fmt.Println(13&7)
    fmt.Println(5|4) //?
    fmt.Println(-3^3) //?

}
```

4.12.2 二进制在运算中的说明

二进制是逢 2 进位的进位制，0、1 是基本算符。

现代的电子计算机技术全部采用的是二进制，因为它只使用 0、1 两个数字符号，非常简单方便，

易于用电子方式实现。计算机内部处理的信息，都是采用二进制数来表示的。二进制（Binary）数用 0 和 1 两个数字及其组合来表示任何数。进位规则是“逢 2 进 1”，数字 1 在不同的位上代表不同的值，按从右至左的次序，这个值以二倍递增。

在计算机的内部，运行各种运算时，都是以二进制的方式来运行。

4.12.3 原码、反码、补码

网上对原码,反码,补码的解释过于复杂，我这里精简6句话:

对于有符号的而言:
1) 二进制的最高位是符号位: 0表示正数,1表示负数
1 ===》 [0000 0001] -1===>[1000 0001]

2) 正数的原码，反码，补码都一样
3) 负数的反码=它的原码符号位不变，其它位取反(0->1,1->0)
1===> 原码 [0000 0001] 反码 [0000 0001] **补码** [0000 0001]
-1===> 原码[1000 0001] 反码[1111 1110] **补码** [1111 1111]

4) 负数的补码=它的反码+1
5) 0的反码，补码都是0
6) 在**计算机运算的时候**，都是以**补码的方式来运算**的.
1+1 1-1 = 1 + (-1)

4.12.4 位运算符和移位运算符

- Golang 中有 3 个位运算

  分别是”按位与&、按位或|、按位异或^,它们的运算规则是：

  按位与&　　：　　两位全为 1，结果为 1，否则为 0

  按位或|　　：　　两位有一个为 1，结果为 1，否则为 0

  按位异或 ^　　：　两位一个为 0,一个为 1，结果为 1，否则为 0

- **案例练习**

  比如：2&3=?　2|3=?　2^3=?

```go
package main
import (
    "fmt"
)
func main() {

    //位运算的演示
    fmt.Println(2&3) // 2
    fmt.Println(2|3) // 3
    fmt.Println(2^3) // 3
    fmt.Println(-2^2) //-4

}
```

2&3

2 的补码 0000 0010
3 的补码 0000 0011
2&3 0000 0010 => 2

2|3=?
2 的补码 0000 0010
3 的补码 0000 0011
2|3 0000 0011 => 3

2^3
2 的补码 0000 0010
3 的补码 0000 0011
2^3 0000 0001 =>1

-2^2
-2 的原码 1000 0010 =》反码 1111 1101 => 补码 1111 1110
1111 1110
2 的补码 0000 0010

-2^2 1111 1100（补码）===》原码

1111 1100 =》反码 1111 1011 =》原码 1000 0100 ==》-4

- Golang 中有 2 个移位运算符：

  >>、<< 右移和左移,运算规则:

  右移运算符 >>：**低位溢出,符号位不变,并用符号位补溢出的高位**

  左移运算符 <<: **符号位不变,低位补 0**

- **案例演示**

  a := 1 >> 2 // 0000 0001 =>0000 0000 = 0

```
c := 1 << 2    // 0000 0001 ==> 0000 0100 => 4
```

第 5 章 程序流程控制

5.1 程序流程控制介绍

在程序中，程序运行的流程控制决定程序是如何执行的，是我们必须掌握的，主要有三大流程控制语句。

1) 顺序控制
2) 分支控制
3) 循环控制

5.2 顺序控制

程序从上到下逐行地执行，中间没有任何判断和跳转。

一个案例说明，必须下面的代码中，没有判断，也没有跳转.因此程序按照默认的流程执行，即顺序控制。

```go
//假如还有97天放假，问：xx个星期零xx天
var days int = 97
var week int = days / 7
var day int = days % 7
fmt.Printf("%d个星期零%d天\n", week, day)


//定义一个变量保存华氏温度，华氏温度转换摄氏温度的公式为:
//5/9*(华氏温度-100),请求出华氏温度对应的摄氏温度
var huashi float32 = 134.2
var sheshi float32 = 5.0 / 9 * (huashi - 100)
fmt.Printf("%v 对应的摄氏温度=%v \n", huashi, sheshi)
```

5.2.1 顺序控制的一个流程图

5.2.2顺序控制举例和注意事项

Golang 中定义变量时采用合法的前向引用。如：

```
func main() {
    var num1 int = 10 //声明了 num1
    var num2 int = num1 + 20 //使用 num1
    fmt.Println(num2)
}
```

错误形式：

```
func main() {
    var num2 int = num1 + 20 //使用 num1
    var num1 int = 10 //声明 num1 (×)
    fmt.Println(num2)
}
```

5.3 分支控制

5.3.1分支控制的基本介绍

分支控制就是让程序有选择执行。有下面三种形式

1) 单分支
2) 双分支

3) 多分支

5.3.2单分支控制

➢ 基本语法

```
if 条件表达式{
        执行代码块
}
```

说明：当条件表达式为ture时，就会执行{}的代码。
注意{}是必须有的，就算你只写一行代码。

➢ 应用案例

请大家看个案例[ifDemo.go]:

编写一个程序,可以输入人的年龄,**如果**该同志的年龄大于 18 岁,则输出 "你年龄大于 18,要对自己的行为负责!"　　需求---[分析]---->代码

代码:

```go
func main() {

    //请大家看个案例[ifDemo.go]:
    //编写一个程序,可以输入人的年龄,如果该同志的年龄大于18岁,则输出 "你年龄大
    //于18,要对自己的行为负责!"

    //分析
    //1.年龄 ==> var age int
    //2.从控制台接收一个输入 fmt.Scanln(&age)
    //3.if判断

    var age int
    fmt.Println("请输入年龄:")
    fmt.Scanln(&age)

    if age > 18 {
        fmt.Println("你年龄大于18,要对自己的行为负责!")
    }
}
```

输出的结果：

```
D:\goproject\src\go_code\chapter05\ifdemo>go run main.go
请输入年龄:
20
你年龄大于18,要对自己的行为负责!

D:\goproject\src\go_code\chapter05\ifdemo>go run main.go
请输入年龄:
1
```

- 单分支的流程图

流程图可以用**图形方式**来更加清晰的描述程序执行的流程。

- 单分支的细节说明

```golang
//golang支持在if中，直接定义一个变量，比如下面
if age := 20; age > 18 {
    fmt.Println("你年龄大于18,要对自己的行为负责!")
}
```

5.3.3双分支控制

- 基本语法

```
if 条件表达式 {
        执行代码块1
} else {
        执行代码块2
}
```

说明：当条件表达式成立，即执行代码块1，否则执行代码块2。{} 也是必须有的。

- 应用案例

请大家看个案例[IfDemo2.go]:

编写一个程序,可以输入人的年龄,**如果**该同志的年龄大于 18 岁,则输出 “你年龄大于 18,要对自己的行为负责!”。**否则** ,输出”你的年龄不大这次放过你了.”

```go
func main() {

    //请大家看个案例[IfDemo2.go]:
    //编写一个程序,可以输入人的年龄,如果该同志的年龄大于18岁,则输出 “你年龄大于18,要对
    //自己的行为负责!”。否则 ,输出”你的年龄不大这次放过你了.”

    //思路分析
    //1. 年龄 ===》 var age int
    //2. fmt.Scanln接收
    //3. if --- else

    //代码
    var age int
    fmt.Println("请输入年龄:")
    fmt.Scanln(&age)

    if age > 18 {
        fmt.Println("你年龄大于18....")
    } else {
        fmt.Println("你的年龄不大这次放过你了")
    }

}
```

- 双分支的流程图的分析

```
if age > 18 {
    fmt.Println("你年龄大于18....")
} else {
    fmt.Println("你的年龄不大这次放过你了")
}
```

对双分支的总结

1. 从上图看 条件表达式就是 age >18
2. 执行代码块 1 ===> fmt.Println("你的年龄大于 18") ..
3. 执行代码块 2 ===> fmt.Println("你的年龄不大....") .
4. **强调一下** 双分支只会执行其中的一个分支。

5.3.4单分支和双分支的案例

1) 对下列代码，若有输出，指出输出结果。

```
var x int = 4
var y int = 1
if (x > 2) {
    if (y > 2) {
        fmt.Println(x + y)
    }
        fmt.Println("atguigu")
} else {
    fmt.Println("x is = ", x)
}
//输出结果是 atguigu
```

2) 对下列代码，若有输出，指出输出结果。

```
var x int = 4
if x > 2
    fmt.Println("ok")
else
   fmt.Println("hello")

//编译错误，原因没有 {}
```

3) 对下列代码，若有输出，指出输出结果。

```
var x int = 4
if x > 2 {
    fmt.Println("ok")
}
else {
   fmt.Println("hello")
}
//编程错误，原因是 else 不能换行
```

4) 对下列代码，若有输出，指出输出结果。

```
var x int = 4
   if (x > 2) {
      fmt.Println("ok~")
   } else {
      fmt.Println("hello")
   }
//正确，输出 ok~
//虽然正确，但是我们要求大家 if (x > 2) 要求写成 if  x > 2 { ...
```

5) 编写程序，声明 2 个 int32 型变量并赋值。判断两数之和，如果大于等于 50，打印“hello world!

```go
func main() {
    //编写程序，声明2个int32型变量并赋值。判断两数之和，如果大于等于50，打印“hello world!”

    //分析
    //1. 变量
    //2. 单分支

    var n1 int32 = 10
    var n2 int32 = 50
    if n1 + n2 >= 50 {
        fmt.Println("hello,world")
    }
}
```

6) 编写程序，声明 2 个 float64 型变量并赋值。判断第一个数大于 10.0，且第 2 个数小于 20.0，打印两数之和。

```go
    //编写程序，声明2个float64型变量并赋值。判断第一个数大于10.0，
    //且第2个数小于20.0，打印两数之和

    var n3 float64 = 11.0
    var n4 float64 = 17.0
    if n3 > 10.0 && n4 < 20.0 {
        fmt.Println("和=", (n3 + n4) )
    }
```

7) 【选作】定义两个变量 int32，判断二者的和，是否能被 3 又能被 5 整除，打印提示信息

```go
var num1 int32 = 10
var num2 int32 = 5
if (num1 + num2) % 3 == 0 &&  (num1 + num2) % 5 == 0 {
    fmt.Println("能被3又能被5整除")
}
```

8) 判断一个年份是否是闰年，闰年的条件是符合下面二者之一：(1)年份能被 4 整除，但不能被 100 整除；(2)能被 400 整除

```
//8)判断一个年份是否是闰年，闰年的条件是符合下面二者之一：
//(1)年份能被4整除，但不能被100整除；(2)能被400整除
var year int = 2019
if (year % 4 == 0 && year % 100 !=0) || year % 400 == 0 {
    fmt.Println(year, "是润年~")
}
```

5.3.5 多分支控制

- 基本语法

```
if 条件表达式1 {
        执行代码块1
} else if 条件表达式2 {
        执行代码块2
}
.....
else {
        执行代码块n
}
```

对上面基本语法的说明

1) 多分支的判断流程如下:

(1) 先判断条件表达式 1 是否成立，如果为真，就执行代码块 1

(2) 如果条件表达式 1 如果为假，就去判断条件表达式 2 是否成立， 如果条件表达式 2 为真，就执行代码块 2

(3) 依次类推.

(4) 如果所有的条件表达式不成立，则执行 else 的语句块。

2) else 不是必须的。

3) **多分支只能有一个执行入口。**

- 看一个多分支的流程图(更加清晰)

➢ 多分支的快速入门案例

岳小鹏参加 Golang 考试，他和父亲岳不群达成承诺：

如果：

成绩为 100 分时，奖励一辆 BMW；

成绩为(80，99]时，奖励一台 iphone7plus；

当成绩为[60,80]时，奖励一个 iPad；

其它时，什么奖励也没有。

请从键盘输入岳小鹏的期末成绩，并加以判断

代码如下：

```go
// 其它时，什么奖励也没有。
// 请从键盘输入岳小鹏的期末成绩，并加以判断

//分析思路
//1. score 分数变量 int
//2. 选择多分支流程控制
//3. 成绩从键盘输入 fmt.Scanln

var score int
fmt.Println("请输入成绩:")
fmt.Scanln(&score)

//多分支判断
if score == 100 {
    fmt.Println("奖励一辆BMW")
} else if score > 80 && score <= 99 {
    fmt.Println("奖励一台iphone7plus")
} else if score >= 60 && score <= 80 {
    fmt.Println("奖励一个 iPad")
} else {
    fmt.Println("什么都不奖励")
}
```

对初学者而言，有一个使用陷阱.

```go
//使用陷阱.....只会输出ok1...

var n int = 10
if n > 9 {
    fmt.Println("ok1") //输出 ok1
} else if  n > 6 {
    fmt.Println("ok2")
} else if n > 3 {
    fmt.Println("ok3")
} else {
    fmt.Println("ok4")
}
```

- 多分支的课堂练习

➢ 案例演示2

```
func main() {
   var b bool = true
   if b == false {        //如果写成 b = false; 能编译通过吗？如果能，结果是？
      fmt.Println("a")
   } else if b {
      fmt.Println("b")
   } else if !b {
      fmt.Println("c")//c
   } else {
      fmt.Println("d")
   }
}
//输出结果是b ,
//如果写成 b = false; 能编译通过吗？如果能，结果是？[编程错误，if的条件表达式不能是 赋值语句]
```

案例 3:

➢ **案例演示3**

求$ax^2+bx+c=0$方程的根。a,b,c分别为函数的参数，如果：$b^2-4ac>0$，则有两个解；$b^2-4ac=0$，则有一个解；$b^2-4ac<0$，则无解；

提示1：$x_1=(-b+sqrt(b^2-4ac))/2a$
$x_2=(-b-sqrt(b^2-4ac))/2a$
提示2：math.Sqrt(num); 可以求平方根 需要引入 math包

测试数据: 3 ,100, 6

代码:

```go
    //分析思路
    //1. a,b,c 是三个float64
    //2. 使用到给出的数学公式
    //3. 使用到多分支
    //4. 使用math.Squr方法 =》手册

    //走代码
    var a float64 = 2.0
    var b float64 = 4.0
    var c float64 = 2.0

    m := b * b - 4 * a * c
    //多分支判断
    if m > 0 {
        x1 := (-b + math.Sqrt(m)) / 2 * a
        x2 := (-b - math.Sqrt(m)) / 2 * a
        fmt.Printf("x1=%v x2=%v", x1, x2)
    } else if m == 0 {
        x1 := (-b + math.Sqrt(m)) / 2 * a
        fmt.Printf("x1=%v", x1)
    } else {
        fmt.Println("无解...")
    }
```

➢ 案例演示4

大家都知道，男大当婚，女大当嫁。那么女方家长要嫁女儿，当然要提出一定的条件：高：180cm以上；富：财富1千万以上；帅：是。条件从控制台输入。

1) 如果这三个条件同时满足，则：“我一定要嫁给他!!!”
2) 如果三个条件有为真的情况，则：“嫁吧，比上不足，比下有余。”
3) 如果三个条件都不满足，则：“不嫁！”

var height int32 | var money float32 | var handsome bool

代码：

```go
//分析思路
//1. 应该设计三个变量 var height int32  | var money float32 | var handsome bool
//2. 而且需要从终端输入 fmt.Scanln
//3. 使用多分支if--else if -- else
var height int32
var money float32
var handsome bool

fmt.Println("请输入身高(厘米)")
fmt.Scanln(&height)
fmt.Println("请输入财富(千万)")
fmt.Scanln(&money)
fmt.Println("请输入是否帅(true/false)")
fmt.Scanln(&handsome)

if height > 180 && money > 1.0 && handsome {
    fmt.Println("我一定要嫁给他!!!")
} else if height > 180 || money > 1.0 || handsome {
    fmt.Println("嫁吧，比上不足，比下有余")
} else {
    fmt.Println("不嫁....")
}
```

5.3.6嵌套分支

- 基本介绍

  在一个分支结构中又完整的嵌套了另一个完整的分支结构，里面的分支的结构称为内层分支外面的分支结构称为外层分支。

- 基本语法

基本语法

```go
if 条件表达式{
    if 条件表达式{
    }else{
    }
}
```

说明：嵌套分支不宜过多，建议控制在3层内。

- 应用案例 1

  参加百米运动会，如果用时 8 秒以内进入决赛，否则提示淘汰。并且根据性别提示进入男子组或女

子组。【可以让学员先练习下】，输入成绩和性别。

代码:

```go
//分析思路
//1. 定义一个变量，来接收跑步使用秒数. float64
//2. 定义一个变量，来接收性别string
//3. 因为判断是嵌套的判断，因此我们会使用嵌套分支

var second float64

fmt.Println("请输入秒数")
fmt.Scanln(&second)

if second <= 8 {
    //进入决赛
    var gender string
    fmt.Println("请输入性别")
    fmt.Scanln(&gender)
    if gender == "男" {
        fmt.Println("进入决赛的男子组")
    } else {
        fmt.Println("进入决赛的女子组")
    }
} else {
    fmt.Println("out...")
}
```

- 应用案例 2

出票系统：根据淡旺季的月份和年龄，打印票价 [考虑学生先做]

4_10 旺季：

成人（18-60）：60

儿童（<18）:半价

老人（>60）:1/3

淡季：

成人：40

其他：20

代码：

```go
    //分析思路
    //1.month age 的两个变量 byte
    //2.使用嵌套分支

    var month byte
    var age byte
    var price float64 = 60.0
    fmt.Println("请输入游玩月份")
    fmt.Scanln(&month)
    fmt.Println("请输入游客的年龄")
    fmt.Scanln(&age)

    if month >= 4 && month <= 10 {
        if age > 60 {
            fmt.Printf("%v月 票价 %v 年龄 %v ", month, price / 3 ,  age)
        } else if age >= 18 {
            fmt.Printf("%v月 票价 %v 年龄 %v ", month, price,  age)
        } else {
            fmt.Printf("%v月 票价 %v 年龄 %v ", month, price / 2,  age)
        }
    } else {
        //淡季
        if age >= 18 && age < 60 {
            fmt.Println("淡季成人 票价 40")
        } else {
            fmt.Println("淡季儿童和老人  票价 20")
        }
    }
}
```

5.4 switch 分支控制

5.4.1基本的介绍

1) switch 语句用于基于不同条件执行不同动作，每一个 case 分支都是唯一的，从上到下逐一测试，直到匹配为止。

2) 匹配项后面也**不需要再加 break**

5.4.2基本语法

```
switch 表达式 {

case 表达式1, 表达式2, ... :
        语句块1

case 表达式3, 表达式4, ... :
        语句块2
//这里可以有多个case语句

default:
        语句块
}
```

5.4.3switch 的流程图

➢ 对上图的说明和总结

1) switch 的执行的流程是，先执行表达式，得到值，然后和 case 的表达式进行比较，如果相等，就匹配到，然后执行对应的 case 的语句块，然后退出 switch 控制。

2) 如果 switch 的表达式的值没有和任何的 case 的表达式匹配成功，则执行 default 的语句块。执行

后退出 switch 的控制.

3) golang 的 case 后的表达式可以有多个，使用 逗号 间隔.

4) golang 中的 case 语句块不需要写 break，因为默认会有,即在默认情况下，当程序执行完 case 语句块后，就直接退出该 switch 控制结构。

5.4.4switch 快速入门案例

➢ 案例：

请编写一个程序，该程序可以接收一个字符，比如: a,b,c,d,e,f,g a 表示星期一，b 表示星期二 … 根据用户的输入显示相依的信息.要求使用 switch 语句完成

➢ 代码

```go
    //分析思路
    //1. 定义一个变量接收字符
    //2. 使用switch完成
    var key byte
    fmt.Println("请输入一个字符 a,b,c,d,e,f,g")
    fmt.Scanf("%c", &key)

    switch key {
        case 'a':
            fmt.Println("周一, 猴子穿新衣")
        case 'b':
            fmt.Println("周二, 猴子当小二")
        case 'c':
            fmt.Println("周三, 猴子爬雪山")
        //...
        default:
            fmt.Println("输入有误...")

    }
```

5.4.5switch 的使用的注意事项和细节

1) case/switch 后是一个表达式(即：**常量值**、变量、一个有**返回值的函数**等都可以)

```go
//写一个非常简单的函数
func test(char byte) byte {
    return char + 1
}

func main() {

    //  案例:
    // 请编写一个程序，该程序可以接收一个字符，比如: a,b,c,d,e,f,g
    // a表示星期一，b表示星期二 …  根据用户的输入显示相依的信息.

    // 要求使用 switch 语句完成

    //分析思路
    //1. 定义一个变量接收字符
    //2. 使用switch完成
    var key byte
    fmt.Println("请输入一个字符 a,b,c,d,e,f,g")
    fmt.Scanf("%c", &key)

    switch test(key)+1 { //将语法现象
        case 'a':
            fmt.Println("周一，猴子穿新衣")
        case 'b':
            fmt.Println("周二，猴子当小二")
        case 'c':
            fmt.Println("周三，猴子爬雪山")
        //...
        default:
            fmt.Println("输入有误...")
    }



}
```

2) case 后的各个表达式的值的数据类型，必须和 switch 的表达式数据类型一致

```go
var n1 int32 = 20
var n2 int64 = 20
switch n1 {
    case n2 :  // 错误，原因是 n2的数据类型和n1不一致
        fmt.Println("ok1")
    default :
        fmt.Println("没有匹配到")
}
```

3) case 后面可以带多个表达式，使用逗号间隔。比如 case 表达式 1，表达式 2 ...

```go
var n1 int32 = 5
var n2 int32 = 20
switch n1 {
    case n2, 10, 5 :  // case 后面可以有多个表达式
        fmt.Println("ok1")
    default :
        fmt.Println("没有匹配到")
}
```

4) case 后面的表达式如果是常量值(字面量)，则要求不能重复

```go
var n1 int32 = 5
var n2 int32 = 20
switch n1 {
    case n2, 10, 5 :  // case 后面可以有多个表达式
        fmt.Println("ok1")
    case 5 : // 错误，因为前面我们有常量5，因此重复，就会报错
        fmt.Println("ok2~")
    default :
        fmt.Println("没有匹配到")
}
```

5) case 后面不需要带 break，程序匹配到一个 case 后就会执行对应的代码块，然后退出 switch，如果一个都匹配不到，则执行 default

6) default 语句不是必须的.

7) switch 后也可以不带表达式，类似 if --else 分支来使用。【案例演示】

```go
    //switch 后也可以不带表达式，类似 if --else分支来使用。【案例演示】
    var age int = 10

    switch {
        case age == 10 :
            fmt.Println("age == 10")
        case age == 20 :
            fmt.Println("age == 20")
        default :
            fmt.Println("没有匹配到")
    }

    //case 中也可以对 范围进行判断
    var score int = 90
    switch {
        case score > 90 :
            fmt.Println("成绩优秀..")
        case score >=70 && score <= 90 :
            fmt.Println("成绩优良...")
        case score >= 60 && score < 70 :
            fmt.Println("成绩及格...")
        default :
            fmt.Println("不及格")
    }
```

8) switch 后也可以直接声明/定义一个变量，分号结束，不推荐。 【案例演示】

```go
//switch 后也可以直接声明/定义一个变量，分号结束，不推荐

switch grade := 90; { // 在golang中，可以这样写
    case grade > 90 :
        fmt.Println("成绩优秀~..")
    case grade >=70 && grade <= 90 :
        fmt.Println("成绩优良~...")
    case grade >= 60 && grade < 70 :
        fmt.Println("成绩及格~...")
    default :
        fmt.Println("不及格~")
}
```

9) switch 穿透-fallthrough ，如果在 case 语句块后增加 fallthrough ,则会继续执行下一个 case，也叫 switch 穿透

```go
//switch 的穿透 fallthrought
var num int = 10
switch num {
    case 10:
        fmt.Println("ok1")
        fallthrough //默认只能穿透一层
    case 20:
        fmt.Println("ok2")
        fallthrough
    case 30:
        fmt.Println("ok3")
    default:
        fmt.Println("没有匹配到..")
}
```

10) Type Switch：switch 语句还可以被用于 type-switch 来判断某个 interface 变量中实际指向的变量类型 【还没有学 interface，先体验一把】

```go
var x interface{}
var y = 10.0
x = y
switch i := x.(type) {
    case nil:
    fmt.Printf(" x 的类型~ :%T",i)
    case int:
    fmt.Printf("x 是 int 型")
    case float64:
    fmt.Printf("x 是 float64 型")
    case func(int) float64:
    fmt.Printf("x 是 func(int) 型")
    case bool, string:
    fmt.Printf("x 是 bool 或 string 型" )
    default:
    fmt.Printf("未知型")
}
```

5.4.6switch 的课堂练习

1) 使用 switch 把小写类型的 char 型转为大写(键盘输入)。只转换 a, b, c, d, e. 其它的输出“other”。

```go
func main(){
    //1)使用 switch 把小写类型的 char型转为大写(键盘输入)。
    //只转换 a, b, c, d, e. 其它的输出 “other”。

    var char byte
    fmt.Println("请输入一个字符..")
    fmt.Scanf("%c", &char)

    switch char {
        case 'a':
            fmt.Println("A")
        case 'b':
            fmt.Println("B")
        case 'c':
            fmt.Println("C")
        case 'd':
            fmt.Println("D")
        case 'e':
            fmt.Println("E")
        default :
            fmt.Println("other")
    }

}
```

2) 对学生成绩大于 60 分的，输出“合格”。低于 60 分的，输出“不合格”。(注：输入的成绩不能大于 100)

```go
    //2)对学生成绩大于60分的，输出“合格”。低于60分的，输出“不合格”。
    //(注：输入的成绩不能大于100)

    var score float64
    fmt.Println("请输入成绩")
    fmt.Scanln(&score)

    switch int(score / 60) {
        case 1:
            fmt.Println("及格")
        case 0:
            fmt.Println("不及格")
        default:
            fmt.Println("输入有误..")
    }
```

3) 根据用户指定月份，打印该月份所属的季节。3,4,5 春季 6,7,8 夏季 9,10,11 秋季 12, 1, 2 冬季

```go
    //3)根据用户指定月份，
    //打印该月份所属的季节。3,4,5 春季 6,7,8 夏季  9,10,11 秋季 12, 1, 2 冬季

    var month byte
    fmt.Println("请输入月份")
    fmt.Scanln(&month)
    switch month {
        case 3, 4, 5 :
            fmt.Println("spring")
        case 6, 7, 8 :
            fmt.Println("summer")
        case 9, 10, 11 :
            fmt.Println("autumn")
        case 12, 1, 2 :
            fmt.Println("winter")
        default:
            fmt.Println("输入有误..")
    }
```

5.4.7switch 和 if 的比较

总结了什么情况下使用 switch ,什么情况下使用 if

1) 如果判断的**具体数值不多**，而且符合整数、浮点数、字符、字符串这几种类型。建议使用 **swtich** 语句，简洁高效。

2) 其他情况：对**区间判断**和结果为 **bool** 类型的判断，使用 if，**if 的使用范围更广**。

5.5 for 循环控制

5.5.1基本介绍

听其名而知其意。就是让我们的一段代码循环的执行。

5.5.2一个实际的需求

- 请大家看个案例 [forTest.go]:

  编写一个程序，可以打印 10 句

  "你好，尚硅谷!"。请大家想想怎么做?

- 使用传统的方式实现

```go
func main() {
    //输出10句 "你好，尚硅谷"

    // fmt.Println("你好，尚硅谷")
    // fmt.Println("你好，尚硅谷")
    // fmt.Println("你好，尚硅谷")
    // fmt.Println("你好，尚硅谷")
    // fmt.Println("你好，尚硅谷")
    // fmt.Println("你好，尚硅谷")
    // fmt.Println("你好，尚硅谷")
    // fmt.Println("你好，尚硅谷")
```

- for 循环的快速入门

```go
//golang中，有循环控制语句来处理循环的执行某段代码的方法->for循环
//for循环快速入门
for i := 1; i <= 10; i++ {
    fmt.Println("你好，尚硅谷", i)
}
```

5.5.3 for 循环的基本语法

- 语法格式

for 循环变量初始化; 循环条件; 循环变量迭代 {

循环操作(语句)

}

- 对上面的语法格式说明

1) 对 for 循环来说，有四个要素：
2) 循环变量初始化
3) 循环条件
4) 循环操作(语句) ,有人也叫循环体。
5) 循环变量迭代

- for 循环执行的顺序说明：

1) 执行循环变量初始化，比如 i := 1
2) 执行循环条件， 比如 i <= 10

3) 如果循环条件为真，就执行循环操作 :比如 fmt.Println(“....”)

4) 执行循环变量迭代 , 比如 i++

5) 反复执行 2, 3, 4 步骤，直到 循环条件为 False ，就退出 for 循环。

5.5.4for 循环执行流程分析

- for 循环的流程图

- 对照代码分析 for 循环的执行过程

```go
//golang中，有循环控制语句来处理循环的执
//for循环快速入门
for i := 1; i <= 10; i++ {
    fmt.Println("你好，尚硅谷", i)
}
```

内存
i [11]

终端

```
你好，尚硅谷 1
你好，尚硅谷 2
你好，尚硅谷 3
你好，尚硅谷 4
...
你好，尚硅谷 9
你好，尚硅谷 10
```

5.5.5 for 循环的使用注意事项和细节讨论

1) **循环条件**是返回一个布尔值的表达式
2) for 循环的第二种使用方式

for　循环判断条件　{

　　//循环执行语句

}

将变量初始化和变量迭代写到其它位置

➢ 案例演示：

```go
//for循环的第二种写法
j := 1 //循环变量初始化
for j <= 10 { //循环条件

    fmt.Println("你好，尚硅谷~", j)
    j++ //循环变量迭代
}
```

3) for 循环的第三种使用方式

for {

//循环执行语句

}

上面的写法**等价 for ; ; {}** 是一个**无限循环**， 通常需要**配合 break 语句**使用

```
//for循环的第三种写法，这种写法通常会配合break使用
k := 1
for {  // 这里也等价 for ; ; {
    if k <= 10 {
        fmt.Println("ok~~", k)
    } else {
        break //break就是跳出这个for循环
    }
    k++
}
```

4) Golang 提供 for-range 的方式，可以方便遍历字符串和数组(注: 数组的遍历，我们放到讲数组的时候再讲解) ，**案例说明如何遍历字符串**。

➢ 字符串遍历方式 1-传统方式

```
//字符串遍历方式1-传统方式
var str string = "hello,world!"
for i := 0; i < len(str); i++ {
    fmt.Printf("%c \n", str[i]) //使用到下标...
}
```

➢ 字符串遍历方式 2-for - range

```
fmt.Println()
//字符串遍历方式2-for-range
str = "abc~ok"
for index, val := range str {
    fmt.Printf("index=%d, val=%c \n", index, val)
}
```

➢ 上面代码的细节讨论

如果我们的字符串含有中文，那么传统的遍历字符串方式，就是错误，会出现乱码。原因是传统的对字符串的遍历是按照**字节来遍历**，而一个汉字在 utf8 编码是对应 **3 个字节**。

如何解决 需要要将 str 转成 []rune 切片.=> 体验一把

```
//字符串遍历方式1-传统方式
var str string = "hello,world!北京"
str2 := []rune(str) // 就是把 str 转成 []rune
for i := 0; i < len(str2); i++ {
    fmt.Printf("%c \n", str2[i]) //使用到下标...
}
```

对应 for-range 遍历方式而言，是按照字符方式遍历。因此如果有字符串有中文，也是 ok

```
fmt.Println()
//字符串遍历方式2-for-range
str = "abc~ok上海"
for index, val := range str {
    fmt.Printf("index=%d, val=%c \n", index, val)
}
```

5.5.6 for 循环的课堂练习

1) 打印 1~100 之间所有是 9 的倍数的整数的个数及总和

```go
func main() {
    //打印1~100之间所有是9的倍数的整数的个数及总和

    //分析思路
    //1. 使用for循环对 max 进行遍历
    //2. 当一个数%9 ==0 就是9的倍数
    //3. 我们需要声明两个变量 count 和 sum 来保存个数和总和
    var max uint64 = 100
    var count uint64 = 0
    var sum uint64 = 0
    var i uint64 = 1
    for ; i <= max; i++ {
        if i % 9 == 0 {
            count++
            sum += i
        }
    }
    fmt.Printf("count=%v sum=%v\n", count, sum)
}
```

2) 完成下面的表达式输出 ，6 是可变的。

```
0 + 6 = 6
1 + 5 = 6
2 + 4 = 6
3 + 3 = 6
4 + 2 = 6
5 + 1 = 6
6 + 0 = 6
```

```go
fmt.Println("--------------------------------")
//完成下面的表达式输出 ，6是可变的
var n int = 60
for i := 0; i <= n; i++ {
    fmt.Printf("%v + %v = %v \n", i, n - i, n)
}
```

5.6 while 和 do..while 的实现

Go 语言没有 while 和 do...while 语法，这一点需要同学们注意一下，如果我们需要使用类似其它语言(比如 java / c 的 while 和 do...while)，**可以通过 for 循环来实现其使用效果**。

5.6.1 while 循环的实现

```
循环变量初始化
for {
   if 循环条件表达式 {
      break //跳出for循环..
   }
   循环操作(语句)
   循环变量迭代
}
说明:
```

- 说明上图

  1) for 循环是一个无限循环

  2) break 语句就是跳出 for 循环

- 使用上面的 while 实现完成输出 10 句”hello,wrold”

```go
func main(){

    //使用while方式输出10句 "hello,world"
    //循环变量初始化
    var i int = 1
    for {
        if i > 10 { //循环条件
            break // 跳出for循环,结束for循环
        }
        fmt.Println("hello,world", i)
        i++ //循环变量的迭代
    }

    fmt.Println("i=", i)

}
```

5.6.2do..while 的实现

```
循环变量初始化
for {
   循环操作(语句)
   循环变量迭代
   if 循环条件表达式 {
      break //跳出for循环..
   }
}
说明:
```

➢ 对上图的说明

1) 上面的循环是先执行，在判断，因此至少执行一次。

2) 当循环条件成立后，就会执行 break, break 就是跳出 for 循环，结束循环.

➢ 案例演示

使用上面的 do...while 实现完成输出 10 句”hello,ok”

```go
    //使用的do...while实现完成输出10句"hello,ok"
    var j int = 1
    for {
        fmt.Println("hello,ok", j)
        j++ //循环变量的迭代
        if j > 10 {
            break //break 就是跳出for循环
        }
    }
```

5.7 多重循环控制(重点，难点)

5.7.1基本介绍

1) 将一个循环放在另一个循环体内，就形成了嵌套循环。在外边的 for 称为外层循环在里面的 for 循环称为内层循环。**【建议一般使用两层，最多不要超过 3 层】**

2) 实质上，嵌套循环就是把内层循环当成外层循环的循环体。当只有内层循环的循环条件为 false 时，才会完全跳出内层循环，才可结束外层的当次循环，开始下一次的循环。

3) 外层循环次数为 **m** 次，内层为 **n** 次，则内层循环体实际上需要执行 **m*n** 次

5.7.2应用案例

1) 统计 3 个班成绩情况，每个班有 5 名同学，求出各个班的平均分和所有班级的平均分[学生的成绩从键盘输入]

编程时两大绝招

(1) 先易后难， 即将一个复杂的问题分解成简单的问题。

(2) 先死后活

代码:

```go
package main
import "fmt"
func main(){

    //1)统计3个班成绩情况，每个班有5名同学，
    //求出各个班的平均分和所有班级的平均分[学生的成绩从键盘输入]

    //分析实现思路
    //1. 统计1个班成绩情况，每个班有5名同学，求出该班的平均分【学生的成绩从键盘输入】=》先易后
    //2. 学生数就是5个 [先死后活]
    //3. 声明一个sum 统计班级的总分

    //分析实现思路2
    //1. 统计3个班成绩情况，每个班有5名同学，求出每个班的平均分【学生的成绩从键盘输入】
    //2. j 表示第几个班级
    //3. 定义一个变量存放总成绩

    //分析实现思路3
    //1. 我们可以把代码做活
    //2. 定义两个变量，表示班级的个数和班级的人数


    //走代码实现
    var classNum int = 2
    var stuNum int = 5
    var totalSum float64 = 0.0
```

```go
    var totalSum float64 = 0.0
    for j := 1; j <= classNum; j ++ {
        sum := 0.0
        for i := 1; i <= stuNum; i++ {
            var score float64
            fmt.Printf("请输入第%d班 第%d个学生的成绩 \n", j, i)
            fmt.Scanln(&score)
            //累计总分
            sum += score
        }

        fmt.Printf("第%d个班级的平均分是%v\n", j, sum / float64(stuNum) )
        //将各个班的总成绩累计到totalSum
        totalSum += sum
    }

    fmt.Printf("各个班级的总成绩%v 所有班级平均分是%v\n", totalSum, totalSum / float64(stuN

}
```

2) 统计三个班及格人数，每个班有 5 名同学

对上面的代码进行了一点修改.

```go
    //统计三个班及格人数，每个班有5名同学
    //分析思路
    //1. 声明以变量 passCount 用于保存及格人数

    //走代码实现
    var classNum int = 2
    var stuNum int = 5
    var totalSum float64 = 0.0
    var passCount int = 0
    for j := 1; j <= classNum; j ++ {
        sum := 0.0
        for i := 1; i <= stuNum; i++ {
```

```go
    for j := 1; j <= classNum; j ++ {
        sum := 0.0
        for i := 1; i <= stuNum; i++ {
            var score float64
            fmt.Printf("请输入第%d班 第%d个学生的成绩 \n", j, i)
            fmt.Scanln(&score)
            //累计总分
            sum += score
            //判断分数是否及格
            if score >= 60 {
                passCount++
            }
        }
```

3) 打印金字塔经典案例

使用 for 循环完成下面的案例请编写一个程序，可以接收一个整数,表示层数，打印出金字

- 分析编程思路
- 走代码

```go
package main
import "fmt"
func main() {

    //使用 for 循环完成下面的案例请编写一个程序，可以接收一个整数,表示层数，打印出金字塔

    //编程思路
    //1. 打印一个矩形
    /*

        ***
        ***
        ***
    */

    //2. 打印半个金字塔
    /*
        *    1 个 *
        **   2 个 *
        ***  3 个 *
    */

    //3 打印整个金字塔
    /*
        *     1层 1 个*  规律: 2 * 层数 - 1   空格 2 规律 总层数-当前层数
       ***    2层 3 个*  规律: 2 * 层数 - 1   空格 1 规律 总层数-当前层数
```

```go
     *****   3层 5 个*  规律: 2 * 层数 - 1   空格 0 规律 总层数-当前层数
    */
    //4 将层数做成一个变量，先死后活
    //var totalLevel int

    //5 打印空心金字塔
    /*
        *
       * *
      *****
       分析：在我们给每行打印*号时，需要考虑是打印 * 还是打印 空格
       我们的分析的结果是，每层的第一个和最后一个是打印*，其它就应该是空的，即输出空格
       我们还分析到一个例外情况，最后层（底层）是全部打*

    */

    var totalLevel int = 20

    //i 表示层数
    for i := 1; i <= totalLevel; i++ {
        //在打印*前先打印空格
        for k := 1; k <= totalLevel - i; k++ {
            fmt.Print(" ")
        }

        //j 表示每层打印多少*
        for j :=1; j <= 2 * i - 1; j++ {
            if j == 1 || j == 2 * i - 1 || i == totalLevel {
                fmt.Print("*")
            } else {
                fmt.Print(" ")
            }

        }
        fmt.Println()
    }



}
```

4) 打印出九九乘法表

```
1 * 1 = 1
1 * 2 = 2     2 * 2 = 4
1 * 3 = 3     2 * 3 = 6     3 * 3 = 9
1 * 4 = 4     2 * 4 = 8     3 * 4 = 12    4 * 4 = 16
1 * 5 = 5     2 * 5 = 10    3 * 5 = 15    4 * 5 = 20    5 * 5 = 25
1 * 6 = 6     2 * 6 = 12    3 * 6 = 18    4 * 6 = 24    5 * 6 = 30    6 * 6 = 36
1 * 7 = 7     2 * 7 = 14    3 * 7 = 21    4 * 7 = 28    5 * 7 = 35    6 * 7 = 42    7 * 7 = 49
1 * 8 = 8     2 * 8 = 16    3 * 8 = 24    4 * 8 = 32    5 * 8 = 40    6 * 8 = 48    7 * 8 = 56    8 * 8 = 64
1 * 9 = 9     2 * 9 = 18    3 * 9 = 27    4 * 9 = 36    5 * 9 = 45    6 * 9 = 54    7 * 9 = 63    8 * 9 = 72    9 * 9 = 81
```

代码：

```go
//打印出九九乘法表
//i 表示层数
var num int = 9
for i := 1; i <= num; i++ {
    for j := 1; j <= i; j++ {
        fmt.Printf("%v * %v = %v \t", j, i, j * i)
    }
    fmt.Println()
}
```

5.8 跳转控制语句-break

5.8.1看一个具体需求，引出 break

随机生成 1-100 的一个数，直到生成了 99 这个数，看看你一共用了几次?

分析：编写一个无限循环的控制，然后**不停的随机生成数**，当生成了 99 时，就退出这个无限循环==》break 提示使用

这里我们给大家说一下，如下随机生成 1-100 整数.

```go
//在go中，需要生成一个随机额种子，否则返回的值总是固定的.
// time.Now().Unix() : 返回一个从 1970 1-1 0:0:0 到现在的一个秒数
rand.Seed(time.Now().Unix())
fmt.Println("n", rand.Intn(100)+1)
```

5.8.2break 的快速入门案例

```go
func main() {

    //我们为了生成一个随机数，还需要个rand设置一个种子.
    //time.Now().Unix() : 返回一个从1970:01:01 的0时0分0秒到现在的秒数
    //rand.Seed(time.Now().Unix())
    //如何随机的生成1-100整数
    //n := rand.Intn(100) + 1 // [0 100)
    //fmt.Println(n)

    //随机生成1-100的一个数，直到生成了99这个数，看看你一共用了几次
    //分析思路:
    //编写一个无限循环的控制，然后不停的随机生成数，当生成了99时，就退出这个无限循环==》break
    var count int = 0
    for {
        rand.Seed(time.Now().UnixNano())
        n := rand.Intn(100) + 1
        fmt.Println("n=", n)
        count++
        if (n == 99) {
            break //表示跳出for循环
        }
    }

    fmt.Println("生成 99 一共使用了 ", count)
}
```

5.8.3基本介绍：

break 语句用于终止某个语句块的执行，用于中断当前 for 循环或跳出 switch 语句。

5.8.4基本语法：

{ ……

break

……

}

5.8.5以 for 循环使用 break 为例,画出示意图

5.8.6break 的注意事项和使用细节

1) break 语句出现在多层嵌套的语句块中时，可以**通过标签**指明要终止的是哪一层语句块

2) 看一个案例

```
//这里演示一下指定标签的形式来使用 break
lable2:
for i := 0; i < 4; i++ {
    //lable1: // 设置一个标签
    for j := 0; j < 10; j++ {
        if j == 2 {
            //break // break 默认会跳出最近的for循环
            //break lable1
            break lable2 // j=0 j=1
        }
        fmt.Println("j=", j)
    }
}
```

3) 对上面案例的说明

(1) break 默认会跳出最近的 for 循环

(2) break 后面可以指定标签，跳出标签对应的 for 循环

5.8.7课堂练习

1) 100 以内的数求和，求出 当和 第一次大于 20 的当前数

```go
    //100以内的数求和，求出 当和 第一次大于20的当前数
    sum := 0
    for i := 1; i <= 100; i++ {
        sum += i //求和
        if sum > 20 {
            fmt.Println("当sum>20时，当前数是", i)
            break
        }
    }
```

2) 实现登录验证，有三次机会，如果用户名为”张无忌” ,密码”888”提示登录成功，否则提示还有几次机会.

```go
//实现登录验证，有三次机会，如果用户名为"张无忌" ,密码"888"提示登录成功，
//否则提示还有几次机会.

var name string
var pwd string
var loginChance = 3 //
for i := 1 ; i <= 3; i++ {
    fmt.Println("请输入用户名")
    fmt.Scanln(&name)
    fmt.Println("请输入密码")
    fmt.Scanln(&pwd)

    if name == "张无忌" && pwd == "888" {
        fmt.Println("恭喜你登录成功!")
        break
    } else {
        loginChance--
        fmt.Printf("你还有%v次登录机会，请珍惜\n", loginChance)
    }
}

if loginChance == 0 {
    fmt.Println("机会用完，没有登录成功!")
}
```

5.9 跳转控制语句-continue

5.9.1基本介绍：

continue 语句用于**结束本次循环**，继续**执行下一次循环**。

continue 语句出现在多层嵌套的循环语句体中时，**可以通过标签指明要跳过的是哪一层循环** ，这个和前面的 break 标签的使用的规则一样.

5.9.2基本语法：

```
{       ……
        continue
        ……
}
```

5.9.3continue 流程图

5.9.4案例分析 continue 的使用

```
//lable2:
for i := 0; i < 4; i++ {
    //lable1: // 设置一个标签
    for j := 0; j < 10; j++ {
        if j == 2 {
            continue
        }
        fmt.Println("j=", j)
    }
}
```

内存
i = 1
j = 3

终端
j = 0
j = 1
j = 3
j = 4..
j = 9

输出四次上面的结果，但是每次都没有 j = 2

5.9.5continu 的课堂练习

- 练习 1

```
for i := 0; i<13; i++ {
if i == 10 {
continue;
}
fmt.Println("i=", i)
}
```

0 1 2 3 4 5 6 7 8 9 11 12

```
for i := 0; i<2; i++ {
for j := 1; j<4; j++ {
if j == 2 {
continue;
}
fmt.Println( "i=",i,"j=",j)
}
}
```

i = 0 j = 1
i = 0 j = 3
i = 1 j =1
i = 1 j = 3

```
here:
for i:=0; i<2; i++ {
for j:=1; j<4; j++ {
if j==2 {
continue here
}
fmt.Println("i=",i,"j=",j)
}
}
```

i = 0 j = 1
i = 1 j = 1

- continue 实现 打印 1——100 之内的奇数[要求使用 for 循环+continue]

  代码：

```go
package main
import "fmt"
func main() {
    //continue实现 打印1—100之内的奇数[要求使用for循环+continue]

    for i := 1; i <= 100; i++ {
        if i % 2 == 0 {
            continue
        }
        fmt.Println("奇数是", i)
    }
}
```

➢ 从键盘读入个数不确定的整数，并判断读入的正数和负数的个数，输入为 0 时结束程序

```go
    //从键盘读入个数不确定的整数，并判断读入的正数和负数的个数，输入为0时结束程序

    var positiveCount int // 正数的个数
    var negativeCount int // 负数个数
    var num int
    for {
        fmt.Println("请输入一个整数")
        fmt.Scanln(&num)
        if num == 0 {
            break //终止for循环
        }

        if num > 0 {
            positiveCount++
            continue//结束本次循环，进入下次循环
        }
        negativeCount++
    }
    fmt.Printf("正数个数是%v 负数的个数是%v\n", positiveCount, negativeCount)
```

➢ 课后练习题(同学们课后自己完成)：

某人有 100,000 元,每经过一次路口，需要交费,规则如下:

当现金>50000 时,每次交 5%

当现金<=50000 时,每次交 1000

编程计算该人可以经过多少次路口,使用 for break 方式完成

5.10 跳转控制语句-goto

5.10.1 goto 基本介绍

1) Go 语言的 goto 语句可以无条件地转移到程序中指定的行。

2) goto 语句通常与条件语句配合使用。可用来实现条件转移，跳出循环体等功能。

3) 在 Go 程序设计中**一般不主张使用 goto 语句**， 以免造成程序流程的混乱，使理解和调试程序都产生困难

5.10.2 goto 基本语法

goto label

.. .

label: statement

5.10.3 goto 的流程图

5.10.4 快速入门案例

```go
func main() {

    var n int = 30
    //演示goto的使用
    fmt.Println("ok1")
    if n > 20 {
        goto label1
    }
    fmt.Println("ok2")
    fmt.Println("ok3")
    fmt.Println("ok4")
    label1:
    fmt.Println("ok5")
    fmt.Println("ok6")
    fmt.Println("ok7")
}
```

5.11 跳转控制语句-return

5.11.1 介绍：

return 使用在方法或者函数中，表示跳出所在的方法或函数，在**讲解函数的时候，会详细的介绍**。

```go
func main() {

    for i:=1; i<=10; i++ {

        if i==3 {
            return
        }
        fmt.Println("哇哇", i)
    }

    fmt.Println("Hello World!")
}
```

说明

1) 如果 return 是在普通的函数，则表示跳出该函数，即不再执行函数中 return 后面代码，也可以理解成终止函数。

2) 如果 return 是在 main 函数，表示终止 main 函数，也就是说终止程序。

第 6 章 函数、包和错误处理

6.1 为什么需要函数

6.1.1请大家完成这样一个需求:

输入两个数,再输入一个运算符(+,-,*,/)，得到结果.。

6.1.2使用传统的方法解决

➢ 走代码

```go
func main() {

    //请大家完成这样一个需求:
    //输入两个数,再输入一个运算符(+,-,*,/)，得到结果..

    //分析思路....
    var n1 float64 = 1.2
    var n2 float64 = 2.3
    var operator byte = '-'
    var res float64
    switch operator {
        case '+':
            res = n1 + n2
        case '-':
            res = n1 - n2
        case '*':
            res = n1 * n2
        case '/':
            res = n1 / n2
        default:
            fmt.Println("操作符号错误...")
    }
    fmt.Println("res=", res)
}
```

➢ 分析一下上面代码问题

1) 上面的写法是可以完成功能，但是代码冗余
2) 同时不利于代码维护
3) 函数可以解决这个问题

6.2 函数的基本概念

为完成某一功能的程序指令(语句)的集合,称为函数。

在 Go 中,函数分为: 自定义函数、系统函数(查看 Go 编程手册)

6.3 函数的基本语法

```
func 函数名（形参列表）（返回值列表）{
        执行语句...
        return 返回值列表
}
```

1) 形参列表：表示函数的输入
2) 函数中的语句：表示为了实现某一功能代码块
3) 函数可以有返回值,也可以没有

6.4 快速入门案例

使用函数解决前面的计算问题。

走代码:

```go
//将计算的功能，放到一个函数中，然后在需要使用，调用即可
func cal(n1 float64, n2 float64, operator byte) float64 {

    var res float64
    switch operator {
        case '+':
            res = n1 + n2
        case '-':
            res = n1 - n2
        case '*':
            res = n1 * n2
        case '/':
            res = n1 / n2
        default:
            fmt.Println("操作符号错误...")
    }
    return res
}
```

```go
func main() {
    //请大家完成这样一个需求:
    //输入两个数,再输入一个运算符(+,-,*,/), 得到结果..
    //分析思路....
    var n1 float64 = 1.2
    var n2 float64 = 2.3
    var operator byte = '+'
    result := cal(n1, n2, operator)
    fmt.Println("result=", result)

}
```

调用函数

6.5 包的引出

1) 在实际的开发中，我们往往需要在不同的文件中，去调用其它文件的定义的函数，比如 main.go 中，去使用 utils.go 文件中的函数，如何实现？ -》包

2) 现在有两个程序员共同开发一个 Go 项目,程序员 xiaoming 希望定义函数 Cal ,程序员 xiaoqiang 也想定义函数也叫 Cal。两个程序员为此还吵了起来,怎么办? -》包

6.6 包的原理图

包的本质实际上就是创建不同的文件夹，来存放程序文件。

画图说明一下包的原理

6.7 包的基本概念

说明：go 的每一个文件都是属于一个包的，也就是说 go 是以包的形式来管理文件和项目目录结构的

6.8 包的三大作用

区分相同名字的函数、变量等标识符

当程序文件很多时,可以很好的管理项目

控制函数、变量等访问范围，即作用域

6.9 包的相关说明

- 打包基本语法

  package 包名

- 引入包的基本语法

  import "包的路径"

6.10 包使用的快速入门

包快速入门-Go 相互调用函数，我们将 func Cal 定义到文件 utils.go，将 utils.go 放到一个包中，当其它文件需要使用到 utils.go 的方法时，可以 import 该包，就可以使用了.【为演示：新建项目目录结构】

代码演示：

utils.go 文件

```go
package utils
import (
    "fmt"
)
//将计算的功能，放到一个函数中，然后在需要使用，调用即可
//为了让其它包的文件使用Cal函数，需要将C大小类似其它语言的public
func Cal(n1 float64, n2 float64, operator byte) float64 {

    var res float64
    switch operator {
        case '+':
            res = n1 + n2
        case '-':
            res = n1 - n2
        case '*':
            res = n1 * n2
        case '/':
            res = n1 / n2
        default:
            fmt.Println("操作符号错误...")
    }
    return res
}
```

main.go 文件

```go
package main
import (
    "fmt"
    "go_code/chapter06/fundemo01/utils"
)


func main() {
    //请大家完成这样一个需求:
    //输入两个数,再输入一个运算符(+,-,*,/),得到结果..
    //分析思路....
    var n1 float64 = 1.2
    var n2 float64 = 2.3
    var operator byte = '+'
    result := utils.Cal(n1, n2 , operator)
    fmt.Println("result~=", result)
```

导入包

调用函数
包名.函数名()

6.11 包使用的注意事项和细节讨论

1) 在给一个文件打包时，该包对应一个文件夹，比如这里的 utils 文件夹对应的包名就是 utils, 文件的包名通常和文件所在的文件夹名一致，一般为小写字母。

2) 当一个文件要使用其它包函数或变量时，需要先引入对应的包

- 引入方式 1：import "包名"
- 引入方式 2：

import (

"包名"

"包名"

)

- package 指令在 文件第一行，然后是 import 指令。
- 在 import 包时，路径从 $GOPATH 的 src 下开始，不用带 src，编译器会自动从 src 下开始引入

3) 为了让其它包的文件，可以访问到本包的函数，则该**函数名的首字母需要大写**，类似其它语言的 public ,这样才能跨包访问。比如 utils.go 的

```go
func Cal(num1 int, num2 int, operator string) {
    result := 0
    switch operator {
        case "+":
            result = num1 + num2
        case "-":
```

4) 在访问其它包函数，变量时，其语法是 包名.函数名， 比如这里的 main.go 文件中

```go
utils.Cal(90, 80, "+")
utils.Cal(90, 80, "-")
```

5) 如果包名较长，Go 支持给包取别名， 注意细节：取别名后，原来的包名就不能使用了

```go
package main
import (
    "fmt"
    util "go_code/chapter06/fundemo01/utils"
)
```

说明: 如果给包取了别名，**则需要使用别名来访问该包的函数和变量**。

6) 在同一包下，不能有相同的函数名（也不能有相同的全局变量名），否则报重复定义

7) 如果你要编译成一个可执行程序文件，就需要将这个包声明为 main，即 package main .这个就是一个语法规范，如果你是写一个库 ，包名可以自定义

6.12 函数的调用机制

6.12.1 通俗易懂的方式的理解

6.12.2 函数-调用过程

介绍：为了让大家更好的理解函数调用过程, 看两个案例，并画出示意图，这个很重要

1) 传入一个数+1

对上图说明

(1) 在调用一个函数时，会给该函数分配一个新的空间，编译器会通过自身的处理让这个新的空间和其它的栈的空间区分开来

(2) 在每个函数对应的栈中，数据空间是独立的，不会混淆

(3) 当一个函数调用完毕(执行完毕)后，程序会销毁这个函数对应的栈空间。

2) 计算两个数,并返回

```go
package main
import (
    "fmt"
)

//一个函数 test
func test(n1 int) {

    n1 = n1 + 1
    fmt.Println("test() n1=", n1) //?输出结果= ?
}

//一个函数 getSum
//
func getSum(n1 int , n2 int) int {
    sum := n1 + n2
    fmt.Println("getSum sum = ", sum)  // 30
    //当函数有return语句时，就是将结果返回给调用者
    //即谁调用我，就返回给谁
    return sum
}

func main() {

    n1 := 10
    //调用test
```

```go
    test(n1)
    fmt.Println("main() n1=", n1)//?输出结果= ?


    sum := getSum(10, 20)
    fmt.Println("main sum = ", sum) // 30
}
```

6.12.3 return 语句

➢ 基本语法和说明

Go函数支持返回多个值，这一点是其它编程语言没有的。[案例演示]

```
func 函数名（形参列表）（返回值类型列表）{
        语句...
        return 返回值列表
}
```

1) 如果返回多个值时，在接收时，希望忽略某个返回值，则使用 _符号表示占位忽略
2) 如果返回值只有一个，（返回值类型列表） 可以不写 ()

➢ 案例演示 1

请编写要给函数，可以计算两个数的和和差，并返回结果。

```go
func main() {

    n1 := 10
    //调用test
    test(n1)
    fmt.Println("main() n1=", n1)//?输出结果= ?


    sum := getSum(10, 20)
    fmt.Println("main sum = ", sum) // 30

    //调用getSumAndSub
    res1, res2 := getSumAndSub(1, 2) //res1 = 3 res2 = -1
    fmt.Printf("res1=%v res2=%v\n", res1, res2)

}
```

```go
//请编写要给函数，可以计算两个数的和和差，并返回结果
func getSumAndSub(n1 int, n2 int) (int, int) {
    sum := n1 + n2
    sub := n1 - n2
    return sum, sub
}
```

➢ 案例演示 2

一个细节说明: 希望忽略某个返回值，则使用 _ 符号表示占位忽略

```go
//希望忽略某个返回值，则使用 _ 符号表示占位忽略
_, res3 =  getSumAndSub(3, 9)
fmt.Println("res3=", res3)
```

6.13 函数的递归调用

6.13.1 基本介绍

一个函数在**函数体内**又**调用了本身**，我们称为递归调用

6.13.2 递归调用快速入门

➢ 代码 1

```go
package main
import (
    "fmt"
)

func test(n int) {
    if n > 2 {
        n--
        test(n)
    }
    fmt.Println("n=", n)
}

func main() {

    //看一段代码
    test(4) // ?通过分析来看下递归调用的特点
}
```

上面代码的分析图:

➢ 代码 2

```go
func test2(n int) {
    if n > 2 {
        n--
        test2(n)
    } else {
        fmt.Println("n=", n)
    }
}

func main() {

    //看一段代码
    //test(4) // ?通过分析来看下递归调用的特点
    test2(4) // ?通过分析来看下递归调用的特点
}
```

对上面代码分析的示意图：

6.13.3 递归调用的总结

函数递归需要遵守的重要原则:

1) 执行一个函数时，就创建一个新的受保护的独立空间(新函数栈)
2) 函数的局部变量是独立的，不会相互影响
3) 递归必须向退出递归的条件逼近，否则就是无限递归，死龟了:)
4) 当一个函数执行完毕，或者遇到 return，就会返回，遵守谁调用，就将结果返回给谁，同时当

函数执行完毕或者返回时，该函数本身也会被系统销毁

6.13.4 递归课堂练习题

- 题 1：斐波那契数

  请使用递归的方式，求出斐波那契数 1,1,2,3,5,8,13...

  给你一个整数 n，求出它的斐波那契数是多少？

  思路:

  1) 当 n == 1 || n ==2 , 返回 1

  2) 当 n >= 2, 返回 前面两个数的和 f(n-1) + f(n-2)

  代码:

```go
/*
请使用递归的方式，求出斐波那契数1,1,2,3,5,8,13...
给你一个整数n，求出它的斐波那契数是多少？
*/
func fbn(n int) int {
    if (n == 1 || n == 2) {
        return 1
    } else {
        return fbn(n - 1) + fbn(n - 2)
    }
}

func main() {
    res := fbn(3)
    //测试
    fmt.Println("res=", res)
    fmt.Println("res=", fbn(4)) // 3
    fmt.Println("res=", fbn(5)) // 5
    fmt.Println("res=", fbn(6)) // 8
}
```

- 题 2：求函数值

  已知 **f(1)=3; f(n) = 2*f(n-1)+1**;

  请使用递归的思想编程，求出 f(n)的值?

  思路:

直接使用给出的表达式即可完成

代码：

```go
package main
import (
    "fmt"
)

/*
题2：求函数值已知 f(1)=3; f(n) = 2*f(n-1)+1; 请使用递归的思想编程，求出 f(n)的值?

*/
func f(n int) int {
    if n == 1 {
        return 3
    } else {
        return 2 * f(n - 1) + 1
    }
}
func main(){

    //测试一下
    fmt.Println("f(1)=", f(1))
    fmt.Println("f(5)=", f(5))
}
```

➢ 练习题 3

题 3：猴子吃桃子问题

有一堆桃子，猴子第一天吃了其中的一半，并再多吃了一个！以后每天猴子都吃其中的一半，然后再多吃一个。**当到第十天时，想再吃时（还没吃），发现只有 1 个桃子了**。问题：最初共多少个桃子？

思路分析：

1) 第 10 天只有一个桃子

2) 第 9 天有几个桃子 = (第 10 天桃子数量 + 1) * 2

3) 规律: 第 n 天的桃子数据 peach(n) = (peach(n+1) + 1) * 2

代码:

```go
package main
import (
    "fmt"
)

//题3：猴子吃桃子问题?有一堆桃子，猴子第一天吃了其中的一半，
//并再多吃了一个！以后每天猴子都吃其中的一半，然后再多吃一个。
//当到第十天时，想再吃时（还没吃），发现只有1个桃子了。问题：最初共多少个桃子？

//思路分析

/*
1)第10天只有一个桃子
2)第9天有几个桃子  =  (第10天桃子数量 + 1) * 2
3)规律: 第n天的桃子数据  peach(n) = (peach(n+1) + 1) * 2

*/
//n 范围是  1 -- 10 之间
func peach(n int) int {
    if n > 10 || n < 1 {
        fmt.Println("输入的天数不对")
        return 0 //返回0表示没有得到正确数量
    }
    if n == 10 {
        return 1
    } else {
        return (peach(n + 1) + 1) * 2
    }
}

func main() {

    fmt.Println("第1天桃子数量是=", peach(1)) //1534
}
```

6.14 函数使用的注意事项和细节讨论

1) 函数的形参列表可以是多个，返回值列表也可以是多个。

2) 形参列表和返回值列表的数据类型可以是值类型和引用类型。

3) 函数的命名遵循标识符命名规范，首字母不能是数字，首字母大写该函数可以被本包文件和其它包文件使用，类似 public，首字母小写，只能被本包文件使用，其它包文件不能使用，类似 privat

4) 函数中的变量是局部的，函数外不生效**【案例说明】**

```go
//函数中的变量是局部的，函数外不生效
func test() {
    //n1 是 test函数的局部变量，只能在test函数中使用
    var n1 int = 10
}

func main() {
    // 这里不能使用n1,因为n1是test函数的局部变量
    //fmt.Println("n1=", n1)

}
```

5) **基本数据类型**和**数组**默认都是**值传递的**，即进行值拷贝。在函数内修改，不会影响到原来的值。

```go
func test02(n1 int) {

    n1 = n1 + 10
    fmt.Println("test02() n1= ", n1)
}

func main() {
    num := 20
    test02(num)
    fmt.Println("main() num= ", num)
}
```

6) 如果希望函数内的变量能修改函数外的变量(指的是默认以值传递的方式的数据类型)，可以传入变量的地址&，函数内以指针的方式操作变量。从效果上看类似引用 。

7) Go 函数不支持函数重载

```
func test02(n1 int) {

    n1 = n1 + 10
    fmt.Println("test02() n1= ", n1)
}

func test02(n1 int , n2 int) {

}
```

Golang不支持传统的函数重载，会报函数重复定义

8) 在 Go 中，**函数也是一种数据类型**，可以赋值给一个变量，则该变量就是一个函数类型的变量了。通过该**变量可以对函数调**用

```
//在Go中，函数也是一种数据类型，
//可以赋值给一个变量，则该变量就是一个函数类型的变量了。通过该变量可以对函数调用

func getSum(n1 int, n2 int) int {
    return n1 + n2
}
func main() {

    a := getSum
    fmt.Printf("a的类型%T, getSum类型是%T\n", a, getSum)

    res := a(10, 40) // 等价  res := getSum(10, 40)
    fmt.Println("res=", res)
}
```

9) 函数既然是一种数据类型，因此在 Go 中，函数可以作为形参，并且调用

```
//看案例
res2 := myFun(getSum, 50, 60)
fmt.Println("res2=", res2)
```

```
//函数既然是一种数据类型，因此在Go中，函数可以作为形参，并且调用
func myFun(funvar func(int, int) int, num1 int, num2 int ) int {
    return funvar(num1, num2)
}
```

10) 为了简化数据类型定义，Go 支持自定义数据类型

基本语法：type 自定义数据类型名 数据类型 // 理解: 相当于一个别名

案例：type myInt int // 这时 myInt 就等价 int 来使用了.

案例：type mySum func (int, int) int // 这时 mySum 就等价 一个 函数类型 func (int, int) int

举例说明自定义数据类型的使用:

```go
// 给int取了别名 ， 在go中 myInt 和 int 虽然都是int类型，但是go认为myInt和int两个类型
type myInt int

var num1 myInt //
var num2 int
num1 = 40
num2 = int(num1) //各位，注意这里依然需要显示转换,go认为myInt和int两个类型
fmt.Println("num1=", num1, "num2=",num2)
```

```go
//再加一个案例
//这时 myFun 就是 func(int, int) int类型
type myFunType func(int, int) int

//函数既然是一种数据类型，因此在Go中，函数可以作为形参，并且调用
func myFun2(funvar myFunType, num1 int, num2 int ) int {
    return funvar(num1, num2)
}
```

```go
//看案例
res3 := myFun2(getSum, 500, 600)
fmt.Println("res3=", res3)
```

11) 支持对函数返回值命名

```go
//支持对函数返回值命名
func getSumAndSub(n1 int, n2 int) (sum int, sub int){
    sub = n1 - n2
    sum = n1 + n2
    return
}
```

```go
//看案例
a1, b1 := getSumAndSub(1, 2)
fmt.Printf("a=%v b=%v\n", a1, b1)
```

12) 使用 _ 标识符，忽略返回值

```
func cal(n1 int ,n2 int) (sum int , sub int) {
sum = n1 + n2
sub = n1 - n2
return
}
func main() {
res1, _ := cal(10, 20)
fmt.Printf("res1= %d ", res1)
}
```

13) Go 支持可变参数

```
//支持0到多个参数
func sum(args... int) sum int {
}
//支持1到多个参数
func sum(n1 int, args... int) sum int {
}
```

说明:

(1) **args 是slice 切片,通过 args[index] 可以访问到各个值。**

(2) 案例演示: 编写一个函数sum ,可以求出 1到多个int的和

(3) 如果一个函数的形参列表中有可变参数，则可变参数需要放在形参列表最后。

代码演示:

```go
//案例演示:  编写一个函数sum ,可以求出  1到多个int的和
//可以参数的使用
func sum(n1 int,args... int) int {
    sum := n1
    //遍历args
    for i := 0; i < len(args); i++ {
        sum += args[i]  //args[0] 表示取出args切片的第一个元素值，其它依次类推
    }
    return sum
}
```

```go
//测试一下可变参数的使用
res4 := sum(10, 0, -1, 90, 10,100)
fmt.Println("res4=", res4)
```

6.15 函数的课堂练习

- 题 1

```
func sum(n1, n2 float32 ) float32 {
fmt.Printf("n1 type = %T\n", n1)
// n1 type = float32
return n1 + n2
}

func main() {
fmt.Println("sum=", sum(1,2)) // sum = 3
}
//题1：代码有无错误，输出什么？
```

- 题 2

```
type mySum func (int, int) int

func sum(n1 int, n2 int) int {
return n1 + n2
}

func sum2(n1, n2, n3 int) int {
return n1 + n2
}
//使用type 自定义数据类型来简化定义
func myFunc(funcVar mySum, num1 int, num2 int) int {
return funcVar(num1, num2)
}
func main() {
a := sum
b := sum2
fmt.Println(myFunc(a, 1, 2)) //ok
fmt.Println(myFunc(b, 1, 2)) //error
}
//题2：代码有无错误，为什么?
fmt.Println(myFunc(b, 1, 2)) 错误，原因是类型不匹配.
因为不能把 func sum2(n1, n2, n3 int) int 赋给 func (int, int)
int
```

- 题 3:请编写一个函数 swap(n1 *int, n2 *int) 可以交换 n1 和 n2 的值

```go
//请编写一个函数 swap(n1 *int, n2 *int) 可以交换 n1 和 n2的值
func swap(n1 *int, n2 *int) {
    //定义一个临时变量
    t := *n1
    *n1 = *n2
    *n2 = t
}

func main() {

    a := 10
    b := 20
    swap(&a, &b) //传入的地址
    fmt.Printf("a=%v, b=%v", a, b)
}
```

6.16 init 函数

6.16.1 基本介绍

每一个源文件都可以包含一个 init 函数，该函数会在 main 函数执行前，被 Go 运行框架调用，也就是说 init 会在 main 函数前被调用。

6.16.2 案例说明：

```go
package main
import (
    "fmt"
)

//init函数,通常可以在init函数中完成初始化工作
func init() {
    fmt.Println("init()...")
}

func main() {
    fmt.Println("main()...")
}
```

输出的结果是:

```
D:\goproject\src\go_code
init()...
main()...
```

6.16.3 inti 函数的注意事项和细节

1) 如果一个文件同时包含**全局变量定义**，**init 函数**和 **main 函数**，则执行的流程**全局变量定义->init 函数->main 函数**

```go
package main
import (
    "fmt"
)

var age = test()

//为了看到全局变量是先被初始化的，我们这里先写函数
func test() int {
    fmt.Println("test()") //1
    return 90
}

//init函数,通常可以在init函数中完成初始化工作
func init() {
    fmt.Println("init()...") //2
}

func main() {
    fmt.Println("main()...age=", age) //3
}
```

2) init 函数最主要的作用，就是完成一些**初始化的工作**，比如下面的案例

```go
package main
import (
    "fmt"
    //引入包
    "go_code/chapter06/funcinit/utils"
)

var age = test()

//为了看到全局变量是先被初始化的，我们这里先写函数
func test() int {
    fmt.Println("test()") //1
    return 90
}

//init函数,通常可以在init函数中完成初始化工作
func init() {
    fmt.Println("init()...") //2
}

func main() {
    fmt.Println("main()...age=", age) //3
    fmt.Println("Age=", utils.Age, "Name=", utils.Name)
}
```

3) 细节说明：面试题：案例如果 main.go 和 utils.go 都含有 变量定义，init 函数时，执行的流程又是怎么样的呢？

6.17 匿名函数

6.17.1 介绍

Go 支持匿名函数，匿名函数就是没有名字的函数，如果我们某个函数只是希望使用一次，可以考虑使用匿名函数，匿名函数也可以实现多次调用。

6.17.2 匿名函数使用方式 1

在定义匿名函数时就直接调用，这种方式匿名函数只能调用一次。 【案例演示】

```go
func main() {
    //在定义匿名函数时就直接调用，这种方式匿名函数只能调用一次

    //案例演示,求两个数的和， 使用匿名函数的方式完成
    res1 := func (n1 int, n2 int) int {
        return n1 + n2
    }(10, 20)

    fmt.Println("res1=", res1)

}
```

6.17.3 匿名函数使用方式 2

将**匿名函数赋给一个变量**(函数变量)，再通过该**变量来调用匿名函数** 【案例演示】

```go
//将匿名函数func (n1 int, n2 int) int赋给 a变量
//则a 的数据类型就是函数类型 ，此时,我们可以通过a完成调用
a := func (n1 int, n2 int) int {
    return n1 - n2
}

res2 := a(10, 30)
fmt.Println("res2=", res2)
res3 := a(90, 30)
fmt.Println("res3=", res3)
```

6.17.4 全局匿名函数

如果将匿名函数赋给一个全局变量，那么这个匿名函数，就成为一个全局匿名函数，可以在程序有效。

```go
var (
    //fun1就是一个全局匿名函数
    Fun1 = func (n1 int, n2 int) int {
        return n1 * n2
    }
)
```

```go
//全局匿名函数的使用
res4 := Fun1(4, 9)
fmt.Println("res4=", res4)
```

6.18 闭包

6.18.1 介绍

基本介绍：闭包就是**一个函数**和与**其相关的引用环境**组合的**一个整体**(实体)

6.18.2 案例演示：

```go

//累加器
func AddUpper() func (int) int {
    var n int = 10
    return func (x int) int {
        n = n + x
        return n
    }
}

func main() {

    //使用前面的代码
    f := AddUpper()
    fmt.Println(f(1))// 11
    fmt.Println(f(2))// 13
    fmt.Println(f(3))// 16

}
```

➢ 对上面代码的说明和总结

1) AddUpper 是一个函数，返回的数据类型是 fun (int) int

2) 闭包的说明

```
var n int = 10
return func (x int) int {
    n = n + x
    return n
}
```

返回的是一个匿名函数，但是这个匿名函数引用到函数外的 n ,因此这个匿名函数就和 n 形成一个整体，构成闭包。

3) 大家可以这样理解: 闭包是类，函数是操作，n 是字段。函数和它使用到 n 构成闭包。

4) 当我们反复的调用 f 函数时，因为 n 是初始化一次，因此每调用一次就进行累计。

5) 我们要搞清楚闭包的关键，就是要分析出返回的函数它使用(引用)到哪些变量，因为函数和它引用到的变量共同构成闭包。

6) 对上面代码的一个修改，加深对闭包的理解

6.18.3 闭包的最佳实践

➢ 请编写一个程序，具体要求如下

1) 编写一个函数 makeSuffix(suffix string) 可以接收一个文件后缀名(比如.jpg)，并返回一个闭包

2) 调用闭包，可以传入一个文件名，如果该文件名没有指定的后缀(比如.jpg) ,则返回 文件名.jpg，如

果已经有.jpg 后缀，则返回原文件名。

3) 要求使用闭包的方式完成

4) strings.HasSuffix，该函数可以判断某个字符串是否有指定的后缀。

代码：

```
//
// 1)编写一个函数 makeSuffix(suffix string)  可以接收一个文件后缀名(比如.jpg)，并返回一个闭包
// 2)调用闭包，可以传入一个文件名，如果该文件名没有指定的后缀(比如.jpg) ,则返回 文件名.jpg , 如
// 3)要求使用闭包的方式完成
// 4)strings.HasSuffix , 该函数可以判断某个字符串是否有指定的后缀。

func makeSuffix(suffix string) func (string) string {

    return func (name string) string {
        //如果 name 没有指定后缀，则加上，否则就返回原来的名字
        if !strings.HasSuffix(name, suffix)  {
            return name + suffix
        }

        return name
    }
}
```

```
//测试makeSuffix 的使用
//返回一个闭包
f2 := makeSuffix(".jpg")
fmt.Println("文件名处理后=", f2("winter")) // winter.jgp
fmt.Println("文件名处理后=", f2("bird.jpg")) // bird.jpg
```

➢ 上面代码的总结和说明:

1) 返回的匿名函数和 makeSuffix (suffix string) 的 suffix 变量 组合成一个闭包,因为 返回的函数引用到 suffix 这个变量

2) 我们体会一下闭包的好处，如果使用传统的方法，也可以轻松实现这个功能，但是传统方法需要每次都传入 后缀名，比如 .jpg ,而闭包因为可以保留上次引用的某个值，所以我们传入一次就可以反复使用。大家可以仔细的体会一把！

6.19 函数的 defer

6.19.1 为什么需要 defer

在函数中，程序员经常需要创建资源(比如：数据库连接、文件句柄、锁等) ，为了在**函数执行完毕后，及时的释放资源**，Go 的设计者提供 defer (延时机制)。

6.19.2 快速入门案例

```go
func sum(n1 int, n2 int) int {

    //当执行到defer时，暂时不执行，会将defer后面的语句压入到独立的栈(defer栈)
    //当函数执行完毕后，再从defer栈，按照先入后出的方式出栈，执行
    defer fmt.Println("ok1 n1=", n1) //defer 3. ok1 n1 = 10
    defer fmt.Println("ok2 n2=", n2) //defer 2. ok2 n2= 20

    res := n1 + n2 // res = 30
    fmt.Println("ok3 res=", res) // 1. ok3 res= 30
    return res

}

func main() {
    res := sum(10, 20)
    fmt.Println("res=", res)  // 4. res= 30
}
```

执行后，输出的结果:

```
D:\goproject\src\go_code\chapter06
ok3 res= 30
ok2 n2= 20
ok1 n1= 10
res= 30
```

6.19.3 defer 的注意事项和细节

1) 当 go 执行到一个 defer 时，不会立即执行 defer 后的语句，而是将 defer 后的语句压入到一个栈中[我为了讲课方便，暂时称该栈为 defer 栈], 然后继续执行函数下一个语句。

2) 当函数执行完毕后，在从 defer 栈中，依次从栈顶取出语句执行(注：遵守栈 先入后出的机制)，

所以同学们看到前面案例输出的顺序。

3) 在 defer 将语句放入到栈时，也会将相关的值拷贝同时入栈。**请看一段代码：**

```go
func sum(n1 int, n2 int) int {

    //当执行到defer时，暂时不执行，会将defer后面的语句压入到独立的栈(defer栈)
    //当函数执行完毕后，再从defer栈，按照先入后出的方式出栈，执行
    defer fmt.Println("ok1 n1=", n1) //defer 3. ok1 n1 = 10
    defer fmt.Println("ok2 n2=", n2) //defer 2. ok2 n2= 20
    //增加一句话
    n1++ // n1 = 11
    n2++ // n2 = 21
    res := n1 + n2 // res = 32
    fmt.Println("ok3 res=", res) // 1. ok3 res= 32
    return res

}

func main() {
    res := sum(10, 20)
    fmt.Println("res=", res)  // 4. res= 32
}
```

上面代码输出的结果如下：

```
D:\goproject\src\go_code\chapter06\deferdemo>go run main.go
ok3 res= 32
ok2 n2= 20
ok1 n1= 10
res= 32
```

6.19.4 defer 的最佳实践

defer 最主要的价值是在，当函数执行完毕后，可以及时的释放函数创建的资源。看下**模拟代码**。。

```go
func test() {
    //关闭文件资源
    file = openfile(文件名)
    defer file.close()
    //其它代码
}
```

```go
func test() {
    //释放数据库资源
    connect = openDatabse()
    defer connect.close()
    //其它代码
}
```

说明

1) 在 golang 编程中的通常做法是，创建资源后，比如(打开了文件，获取了数据库的链接，或者是锁资源)， 可以执行 defer file.Close() defer connect.Close()

2) 在 defer 后，可以继续使用创建资源.

3) 当函数完毕后，系统会依次从 defer 栈中，取出语句，关闭资源.

4) 这种机制，非常简洁，程序员不用再为在什么时机关闭资源而烦心。

6.20 函数参数传递方式

6.20.1 基本介绍

我们在讲解函数注意事项和使用细节时，已经讲过值类型和引用类型了，这里我们再系统总结一下，因为这是重难点，值类型参数默认就是值传递，而引用类型参数默认就是引用传递。

6.20.2 两种传递方式

1) 值传递

2) 引用传递

其实，不管是值传递还是引用传递，传递给函数的都是变量的副本，不同的是，值传递的是值的拷贝，引用传递的是地址的拷贝，一般来说，地址拷贝效率高，因为数据量小，而值拷贝决定拷贝的数据大小，数据越大，效率越低。

6.20.3 值类型和引用类型

1) **值类型：**基本数据类型 int 系列, float 系列, bool, string 、数组和结构体 struct

2) **引用类型：**指针、slice 切片、map、管道 chan、interface 等都是引用类型

6.20.4 值传递和引用传递使用特点

1) **值类型默认是值传递** 变量直接存储值，内存通常在栈中分配【案例: 画出示意图】

2) **引用类型默认是引用传递** 变量存储的是一个地址，这个地址对应的空间才真正存储数据(值)，内存通常在堆上分配，当没有任何变量引用这个地址时，该地址对应的数据空间就成为一个垃圾，由GC来回收。【案例，并画出示意图】

3) 如果希望函数内的变量能修改函数外的变量，可以传入变量的地址&，函数内以指针的方式操作变量。从效果上看类似引用 。这个案例在前面详解函数使用注意事项的

6.21 变量作用域

1) 函数内部声明/定义的变量叫局部变量，**作用域仅限于函数内**部

```go
//函数
func test() {
    //age 和 Name的作用域就只在test函数内部
    age := 10
    Name := "tom~"
}

func main() {

}
```

2) 函数外部声明/定义的变量叫全局变量，作用域在整个包都有效，如果其首字母为大写，则作用域在整个程序有效

```go
package main
import (
    "fmt"
)

//函数外部声明/定义的变量叫全局变量，
//作用域在整个包都有效，如果其首字母为大写，则作用域在整个程序有效
var age int = 50
var Name string = "jack~"

//函数
func test() {
    //age 和 Name的作用域就只在test函数内部
    age := 10
    Name := "tom~"
    fmt.Println("age=", age) // 10
    fmt.Println("Name=", Name) // tom~
}

func main() {

    fmt.Println("age=", age) //  50
    fmt.Println("Name=", Name) // jack~
    test()
}
```

3) 如果变量是在一个代码块，比如 for / if 中，那么这个变量的的作用域就在该代码块

```go
//如果变量是在一个代码块，比如 for / if中，那么这个变量的的作用域就在该代码块

for i := 0; i <= 10; i++ {
    fmt.Println("i=", i)
}

var i int //局部变量
for i = 0; i <= 10; i++ {
    fmt.Println("i=", i)
}

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

6.21.1 变量作用域的课堂练习

```go
var name = "tom" //全局变量
func test01() {
fmt.Println(name) // tom tom
}
func test02() { //编译器采用就近原则
name := "jack"
fmt.Println(name) // jack
}
func main() {
fmt.Println(name) // tom
test01() //tom
test02() //jack
test01() //tom
}
```

输出的结果是: tom tom jack tom

思考：下面的代码输出什么内容？

：=是定义+赋值

```go
var Age int = 20 //ok
Name := "tom" // var Name string    Name = "tom"
func main() {
 fmt.Println("name", Name)
}
```

错误的原因：因为：
Name := "tom" 等价
var Name string
Name = "tom" 赋值语句不能在函数体外，所以错

6.22 函数课堂练习(综合)

1) 函数可以没有返回值案例，编写一个函数,从终端输入一个整数打印出对应的金子塔

分析思路：就是将原来写的打印金字塔的案例，使用函数的方式封装，在需要打印时，直接调用即可。

```go
//将打印金字塔的代码封装到函数中
func printPyramid(totalLevel int) {

    //i 表示层数
    for i := 1; i <= totalLevel; i++ {
        //在打印*前先打印空格
        for k := 1; k <= totalLevel - i; k++ {
            fmt.Print(" ")
        }

        //j 表示每层打印多少*
        for j :=1; j <= 2 * i - 1; j++ {
                fmt.Print("*")
        }
        fmt.Println()
    }

}
func main() {
    //调用printPyramid函数，就可以打印金字塔
    //从终端输入一个整数打印出对应的金子塔
    var n int
    fmt.Println("请输入打印金字塔的层数")
    fmt.Scanln(&n)
    printPyramid(n)
}
```

2) 编写一个函数,从终端输入一个整数(1—9),打印出对应的乘法表

分析思路：就是将原来写的调用九九乘法表的案例，使用函数的方式封装，在需要打印时，直接调用即可

代码：

```go
//编写一个函数调用九九乘法表
func printMulti(num int) {

    //打印出九九乘法表
    //i 表示层数
    for i := 1; i <= num; i++ {
        for j := 1; j <= i; j++ {
            fmt.Printf("%v * %v = %v \t", j, i, j * i)
        }
        fmt.Println()
    }

}

func main() {

    //从终端输入一个整数表示要打印的乘法表对应的数
    var num int
    fmt.Println("请输入九九乘法表的对应数")
    fmt.Scanln(&num)
    printMulti(num)
}
```

3) 编写函数,对给定的一个二维数组(3×3)转置，这个题讲数组的时候再完成

6.23 字符串常用的系统函数

说明：字符串在我们程序开发中，使用的是非常多的，常用的函数需要同学们掌握[带看手册或者官方编程指南]:

1) 统计字符串的长度，按字节 len(str)

```go
func main(){

    //统计字符串的长度，按字节 len(str)
    ////golang的编码统一为utf-8 (ascii的字符(字母和数字) 占一个字节，汉字占用3个字节)
    str := "hello北"
    fmt.Println("str len=", len(str)) // 8
}
```

2) 字符串遍历，同时处理有中文的问题 r := []rune(str)

```
str2 := "hello北京"
//字符串遍历，同时处理有中文的问题 r := []rune(str)
r := []rune(str2)
for i := 0; i < len(r); i++ {
    fmt.Printf("字符=%c\n", r[i])
}
```

3) 字符串转整数: n, err := strconv.Atoi("12")

```
//字符串转整数:  n, err := strconv.Atoi("12")
n, err := strconv.Atoi("hello")
if err != nil {
    fmt.Println("转换错误", err)
}else {
    fmt.Println("转成的结果是", n)
}
```

4) 整数转字符串 str = strconv.Itoa(12345)

```
//4)整数转字符串  str = strconv.Itoa(12345)
str = strconv.Itoa(12345)
fmt.Printf("str=%v, str=%T", str, str)
```

5) 字符串 转 []byte: var bytes = []byte("hello go")

```
//5)字符串 转 []byte:  var bytes = []byte("hello go")
var bytes = []byte("hello go")
fmt.Printf("bytes=%v\n", bytes)
```

6) []byte 转 字符串: str = string([]byte{97, 98, 99})

```
//6)[]byte 转 字符串: str = string([]byte{97, 98, 99})
str = string([]byte{97, 98, 99})
fmt.Printf("str=%v\n", str)
```

7) 10 进制转 2, 8, 16 进制: str = strconv.FormatInt(123, 2) // 2-> 8 , 16

```
//10进制转 2, 8, 16进制:  str = strconv.FormatInt(123, 2),返回对应的字符串
str = strconv.FormatInt(123, 2)
fmt.Printf("123对应的二进制是=%v\n", str)
str = strconv.FormatInt(123, 16)
fmt.Printf("123对应的16进制是=%v\n", str)
```

8) 查找子串是否在指定的字符串中: strings.Contains("seafood", "foo") //true

```
//查找子串是否在指定的字符串中: strings.Contains("seafood", "foo") //true
b := strings.Contains("seafood", "mary")
fmt.Printf("b=%v\n", b)
```

9) 统计一个字符串有几个指定的子串 ： strings.Count("ceheese", "e") //4

```
//统计一个字符串有几个指定的子串 ： strings.Count("ceheese", "e") //4
num := strings.Count("ceheese", "e")
fmt.Printf("num=%v\n", num)
```

10) 不区分大小写的字符串比较(==是区分字母大小写的): fmt.Println(strings.EqualFold("abc", "Abc")) // true

```
//10)不区分大小写的字符串比较(==是区分字母大小写的): fmt.Println(strin

b = strings.EqualFold("abc", "Abc")
fmt.Printf("b=%v\n", b) //true

fmt.Println("结果","abc" == "Abc") // false //区分字母大小写
```

11) 返回子串在字符串第一次出现的 index 值，如果没有返回-1 : strings.Index("NLT_abc", "abc") // 4

```
//11)返回子串在字符串第一次出现的index值，如果没有返回-1 :
//strings.Index("NLT_abc", "abc") // 4

index := strings.Index("NLT_abcabcabc", "abc") // 4
fmt.Printf("index=%v\n",index)
```

12) 返回子串在字符串最后一次出现的 index，如没有返回-1 : strings.LastIndex("go golang", "go")

```
//12)返回子串在字符串最后一次出现的index,
//如没有返回-1 : strings.LastIndex("go golang", "go")

index = strings.LastIndex("go golang", "go") //3
fmt.Printf("index=%v\n",index)
```

13) 将指定的子串替换成 另外一个子串: strings.Replace("go go hello", "go", "go 语言", n) n 可以指定你希望替换几个，如果 n=-1 表示全部替换

```
//将指定的子串替换成 另外一个子串: strings.Replace("go go hello", "go", "go语言", n)
//n可以指定你希望替换几个，如果n=-1表示全部替换

str2 = "go go hello"
str = strings.Replace(str2, "go", "北京", -1)
fmt.Printf("str=%v str2=%v\n", str, str2)
```

14) 按照指定的某个字符，为分割标识，将一个字符串拆分成**字符串数组**：strings.Split("hello,wrold,ok", ",")

```
//按照指定的某个字符，为分割标识，将一个字符串拆分成字符串数组：
//strings.Split("hello,wrold,ok", ",")
strArr := strings.Split("hello,wrold,ok", ",")
for i := 0; i < len(strArr); i++ {
    fmt.Printf("str[%v]=%v\n", i, strArr[i])
}
fmt.Printf("strArr=%v\n", strArr)
```

15) 将字符串的字母进行大小写的转换: strings.ToLower("Go") // go strings.ToUpper("Go") // GO

```
//15)将字符串的字母进行大小写的转换:
//strings.ToLower("Go") // go strings.ToUpper("Go") // GO

str = "goLang Hello"
str = strings.ToLower(str)
str = strings.ToUpper(str)
fmt.Printf("str=%v\n", str) //golang hello
```

16) 将字符串左右两边的空格去掉： strings.TrimSpace(" tn a lone gopher ntrn ")

```
//将字符串左右两边的空格去掉： strings.TrimSpace(" tn a lone gopher ntrn   ")
str = strings.TrimSpace(" tn a lone gopher ntrn   ")
fmt.Printf("str=%q\n", str)
```

17) 将字符串左右两边指定的字符去掉 ： strings.Trim("! hello! ", " !") // ["hello"] //将左右两边 ！和 " "去掉

```
//17)将字符串左右两边指定的字符去掉 :
//strings.Trim("! hello! ", " !")  // ["hello"] //将左右两边 ！ 和 " "去掉
str = strings.Trim("! he!llo! ", " !")
fmt.Printf("str=%q\n", str)
```

18) 将字符串左边指定的字符去掉 ： strings.TrimLeft("! hello! ", " !") // ["hello"] //将左边 ！ 和 " "去掉

19) 将字符串右边指定的字符去掉 ： strings.TrimRight("! hello! ", " !") // ["hello"] //将右边 ！ 和 " "去掉

20) 判断字符串是否以指定的字符串开头: strings.HasPrefix("ftp://192.168.10.1", "ftp") // true

```
//20)判断字符串是否以指定的字符串开头:
//strings.HasPrefix("ftp://192.168.10.1", "ftp") // true

b = strings.HasPrefix("ftp://192.168.10.1", "hsp") //true
fmt.Printf("b=%v\n", b)
```

21) 判断字符串是否以指定的字符串结束: strings.HasSuffix("NLT_abc.jpg", "abc") //false

6.24 时间和日期相关函数

6.24.1 基本的介绍

说明：在编程中，程序员会经常使用到日期相关的函数，比如：统计某段代码执行花费的时间等等。

1) 时间和日期相关函数，需要导入 time 包

Nanosecond

package time

import "time"

time包提供了时间的显示和测量用的函数。日历的计算采用的是公历。

Index

返回首页

- Constants
- type ParseError
  - func (e *ParseError) Error() string
- type Weekday
  - func (d Weekday) String() string
- type Month
  - func (m Month) String() string
- type Location
  - func LoadLocation(name string) (*Location, error)
  - func FixedZone(name string, offset int) *Location
  - func (l *Location) String() string

2) time.Time 类型，用于表示时间

```go
func main() {
    //看看日期和时间相关函数和方法使用
    //1. 获取当前时间
    now := time.Now()
    fmt.Printf("now=%v now type=%T", now, now)
}
```

```
D:\goproject\src\go_code\chapter06\timefunc>go run main.go
now=2018-05-29 16:19:21.1196972 +0800 CST m=+0.022001301 now type=time.Time
D:\goproject\src\go_code\chapter06\timefunc>
```

3) 如何获取到其它的日期信息

```go
package main
import (
    "fmt"
    "time"
)

func main() {
    //看看日期和时间相关函数和方法使用
    //1. 获取当前时间
    now := time.Now()
    fmt.Printf("now=%v now type=%T\n", now, now)

    //2.通过now可以获取到年月日，时分秒
    fmt.Printf("年=%v\n", now.Year())
    fmt.Printf("月=%v\n", now.Month())
    fmt.Printf("月=%v\n", int(now.Month()))
    fmt.Printf("日=%v\n", now.Day())
    fmt.Printf("时=%v\n", now.Hour())
    fmt.Printf("分=%v\n", now.Minute())
    fmt.Printf("秒=%v\n", now.Second())
}
```

4) 格式化日期时间

方式 1: 就是使用 Printf 或者 SPrintf

```
//格式化日期时间

fmt.Printf("当前年月日 %d-%d-%d %d:%d:%d \n", now.Year(),
now.Month(), now.Day(), now.Hour(), now.Minute(), now.Second())

dateStr := fmt.Sprintf("当前年月日 %d-%d-%d %d:%d:%d \n", now.Year(),
now.Month(), now.Day(), now.Hour(), now.Minute(), now.Second())

fmt.Printf("dateStr=%v\n", dateStr)
```

方式二：使用 time.Format() 方法完成：

```
//格式化日期时间的第二种方式
fmt.Printf(now.Format("2006-01-02 15:04:05"))
fmt.Println()
fmt.Printf(now.Format("2006-01-02"))
fmt.Println()
fmt.Printf(now.Format("15:04:05"))
fmt.Println()
```

对上面代码的说明：

"2006/01/02 15:04:05"　这个字符串的各个数字是固定的，必须是这样写。

"2006/01/02 15:04:05"　这个字符串各个数字可以自由的组合，这样可以按程序需求来返回时间和日期

5) 时间的常量

```
const (
    Nanosecond    Duration = 1 //纳秒
    Microsecond          = 1000 * Nanosecond    //微秒
    Millisecond          = 1000 * Microsecond //毫秒
    Second               = 1000 * Millisecond //秒
    Minute               = 60 * Second //分钟
    Hour                 = 60 * Minute //小时
)
```

常量的作用:**在程序中可用于获取指定时间单位的时间，比如想得到 100 毫秒**

100 * time. Millisecond

6) 结合 Sleep 来使用一下时间常量

```
//需求，每隔1秒中打印一个数字，打印到100时就退出
//需求2: 每隔0.1秒中打印一个数字，打印到100时就退出
i := 0
for {
    i++
    fmt.Println(i)
    //休眠
    //time.Sleep(time.Second)
    time.Sleep(time.Millisecond * 100)
    if i == 100 {
        break
    }
}
```

7) time 的 Unix 和 UnixNano 的方法

func (Time) Unix

```
func (t Time) Unix() int64
```

Unix将t表示为Unix时间，即从时间点January 1, 1970 UTC到时间点t所经过的时间（单位秒）。

func (Time) UnixNano

```
func (t Time) UnixNano() int64
```

UnixNano将t表示为Unix时间，即从时间点January 1, 1970 UTC到时间点t所经过的时间（单位纳秒）。如果纳秒为单位的unix时间超出了int64能表示的范围，结果是未定义的。注意这就意味着Time零值调用UnixNano方法的话，结果是未定义的。

```
//Unix和UnixNano的使用
fmt.Printf("unix时间戳=%v unixnano时间戳=%v\n", now.Unix(), now.UnixNano())
```

得到的结果是：

```
2018
unix时间戳=1527584269 unixnano时间戳=1527584269975756200
```

6.24.2 时间和日期的课堂练习

编写一段代码来统计 函数 test03 执行的时间

```go
package main
import (
    "fmt"
    "time"
    "strconv"
)

func test03() {

    str := ""
    for i := 0; i < 100000; i++ {
        str += "hello" + strconv.Itoa(i)
    }
}

func main() {
    //在执行test03前，先获取到当前的unix时间戳
    start := time.Now().Unix()
    test03()
    end := time.Now().Unix()
    fmt.Printf("执行test03()耗费时间为%v秒\n", end-start)
}
```

6.25 内置函数

6.25.1 说明：

Golang 设计者为了编程方便，提供了一些函数，这些函数可以直接使用，我们称为 Go 的内置函数。文档：https://studygolang.com/pkgdoc -> builtin

1) len：用来求长度，比如 string、array、slice、map、channel
2) new：用来分配内存，主要用来分配值类型，比如 int、float32,struct...返回的是指针

举例说明 new 的使用：

```go
func main() {

    num1 := 100
    fmt.Printf("num1的类型%T , num1的值=%v , num1的地址%v\n", num1, num1, &num1)

    num2 := new(int) // *int
    //num2的类型%T => *int
    //num2的值 = 地址 0xc04204c098 （这个地址是系统分配）
    //num2的地址%v = 地址 0xc04206a020 （这个地址是系统分配）
    //num2指向的值 = 100
    *num2  = 100
    fmt.Printf("num2的类型%T , num2的值=%v , num2的地址%v\n num2这个指针，指向的值=%v",
        num2, num2, &num2, *num2)
}
```

上面代码对应的内存分析图：

3) make：用来**分配内存**，主要用来**分配引用类型**，比如 channel、map、slice。这个我们后面讲解。

6.26 错误处理

6.26.1 看一段代码，因此错误处理

```go
package main
import (
    "fmt"
)

func test() {
    num1 := 10
    num2 := 0
    res := num1 / num2
    fmt.Println("res=", res)
}


func main() {

    //测试
    test()
    fmt.Println("main()下面的代码...")
}
```

对上面代码的总结

1) 在默认情况下，当发生错误后(panic) ,程序就会退出（崩溃.）

2) 如果我们希望：当发生错误后，可以捕获到错误，并进行处理，保证程序可以继续执行。还可以在捕获到错误后，给管理员一个提示(邮件,短信。。。）

3) 这里引出我们要将的错误处理机制

6.26.2 基本说明

1) Go 语言追求简洁优雅，所以，Go 语言不支持传统的 try…catch…finally 这种处理。

2) Go 中引入的处理方式为：**defer, panic, recover**

3) 这几个异常的使用场景可以这么简单描述：Go 中可以抛出一个 panic 的异常，然后在 defer 中通过 recover 捕获这个异常，然后正常处理

6.26.3 使用 defer+recover 来处理错误

```go
package main
import (
    "fmt"
    "time"
)

func test() {
    //使用defer + recover 来捕获和处理异常
    defer func() {
        err := recover() // recover()内置函数，可以捕获到异常
        if err != nil {  // 说明捕获到错误
            fmt.Println("err=", err)
        }
    }()
    num1 := 10
    num2 := 0
    res := num1 / num2
    fmt.Println("res=", res)
}


func main() {

    //测试
    test()
    for {
```

```go
        fmt.Println("main()下面的代码...")
        time.Sleep(time.Second)
    }

}
```

6.26.4 错误处理的好处

进行错误处理后，程序不会轻易挂掉，如果加入预警代码，就可以让程序更加的健壮。看一个案例演示：

```go
func test() {
    //使用defer + recover 来捕获和处理异常
    defer func() {
        err := recover()  // recover()内置函数，可以捕获到异常
        if err != nil {  // 说明捕获到错误
            fmt.Println("err=", err)
            //这里就可以将错误信息发送给管理员....
            fmt.Println("发送邮件给admin@sohu.com~")
        }
    }()
    num1 := 10
    num2 := 0
    res := num1 / num2
    fmt.Println("res=", res)
}
```

6.26.5 自定义错误

6.26.6 自定义错误的介绍

Go 程序中，也支持自定义错误， 使用 errors.New 和 panic 内置函数。

1) errors.New("错误说明") , 会返回一个 error 类型的值，表示一个错误

2) panic 内置函数 ,接收一个 interface{}类型的值（也就是任何值了）作为参数。可以接收 error 类型的变量，**输出错误信息，并退出程序**.

6.26.7 案例说明

```go
//函数去读取以配置文件init.conf的信息
//如果文件名传入不正确，我们就返回一个自定义的错误
func readConf(name string) (err error) {
    if name == "config.ini" {
        //读取...
        return nil
    } else {
        //返回一个自定义错误
        return errors.New("读取文件错误..")
    }
}

func test02() {

    err := readConf("config2.ini")
    if err != nil {
        //如果读取文件发送错误，就输出这个错误，并终止程序
        panic(err)
    }
    fmt.Println("test02()继续执行....")
}
```

```go
func main() {

    //测试
    // test()
    // for {
    //  fmt.Println("main()下面的代码...")
    //  time.Sleep(time.Second)
    // }

    //测试自定义错误的使用

    test02()
    fmt.Println("main()下面的代码...")
}
```

第 7 章 数组与切片

7.1 为什么需要数组

➢ 看一个问题

一个养鸡场有 6 只鸡，它们的体重分别是 3kg,5kg,1kg,3.4kg,2kg,50kg 。请问这六只鸡的总体重是多少?平均体重是多少? 请你编一个程序。=》数组

➢ 使用传统的方法来解决

```go
func main() {

    /*
    一个养鸡场有6只鸡，它们的体重分别是3kg,5kg,1kg,
    3.4kg,2kg,50kg 。请问这六只鸡的总体重是多少?平
    均体重是多少? 请你编一个程序。=》数组

    */

    //思路分析：定义六个变量，分别表示六只鸡的，然后求出和，然后求出平均值。
    hen1 := 3.0
    hen2 := 5.0
    hen3 := 1.0
    hen4 := 3.4
    hen5 := 2.0
    hen6 := 50.0

    totalWeight := hen1 + hen2 + hen3 + hen4 + hen5 + hen6
    //fmt.Sprintf("%.2f", totalWeight / 6) 将 totalWeight / 6 四舍五入保留到小数点2返回值
    avgWeight := fmt.Sprintf("%.2f", totalWeight / 6)
    fmt.Printf("totalWeight=%v avgWeight=%v", totalWeight, avgWeight)

}
```

对上面代码的说明

1) 使用传统的方法不利于数据的管理和维护.

2) 传统的方法不够灵活，因此我们引出需要学习的新的数据类型=>数组.

7.2 数组介绍

数组可以存放多个同一类型数据。数组也是一种数据类型，在 Go 中，数组是值类型。

7.3 数组的快速入门

我们使用数组的方法来解决养鸡场的问题.

```
//使用数组的方式来解决问题

//1.定义一个数组
var hens [7]float64
//2.给数组的每个元素赋值， 元素的下标是从0开始的  0-5
hens[0] = 3.0  //hens数组的第一个元素 hens[0]
hens[1] = 5.0  //hens数组的第2个元素 hens[1]
hens[2] = 1.0
hens[3] = 3.4
hens[4] = 2.0
hens[5] = 50.0
hens[6] = 150.0  //增加一只鸡
//3.遍历数组求出总体重
totalWeight2 := 0.0
for i := 0; i < len(hens); i++ {
    totalWeight2 += hens[i]
}

//4.求出平均体重
avgWeight2 := fmt.Sprintf("%.2f", totalWeight2 / float64(len(hens)))
fmt.Printf("totalWeight2=%v avgWeight2=%v", totalWeight2, avgWeight2)
```

对上面代码的总结

1) 使用数组来解决问题，程序的可维护性增加.

2) 而且方法代码更加清晰，也容易扩展。

7.4 数组定义和内存布局

- 数组的定义

  var 数组名 [数组大小]数据类型

  var a [5]int

  赋初值 a[0] = 1 a[1] = 30 ....

- 数组在内存布局(**重要**)

对上图的总结:

1) 数组的地址可以通过数组名来获取 &intArr

2) 数组的第一个元素的地址，就是数组的首地址

3) 数组的各个元素的地址间隔是依据数组的类型决定，比如 int64 -> 8 int32->4...

```go
func main() {

    var intArr [3]int //int占8个字节
    //当我们定义完数组后，其实数组的各个元素有默认值 0
    fmt.Println(intArr)
    intArr[0] = 10
    intArr[1] = 20
    intArr[2] = 30
    fmt.Println(intArr)
    fmt.Printf("intArr的地址=%p intArr[0] 地址%p intArr[1] 地址%p intArr[2] 地址%p",
        &intArr, &intArr[0], &intArr[1], &intArr[2])

}
```

7.5 数组的使用

➢ 访问数组元素

数组名[下标] 比如：你要使用 a 数组的第三个元素 a[2]

➢ 快速入门案例

从终端循环输入 5 个成绩，保存到 float64 数组,并输出.

```
//从终端循环输入5个成绩，保存到float64数组,并输出.
var score [5]float64

for i := 0; i < len(score); i++ {
    fmt.Printf("请输入第%d个元素的值\n", i+1)
    fmt.Scanln(&score[i])
}

//变量数组打印
for i := 0; i < len(score); i++ {
    fmt.Printf("score[%d]=%v\n", i, score[i])
}
```

➢ 四种初始化数组的方式

```
//四种初始化数组的方式
var numArr01 [3]int = [3]int{1, 2, 3}
fmt.Println("numArr01=", numArr01)

var numArr02 = [3]int{5, 6, 7}
fmt.Println("numArr02=", numArr02)
//这里的 [...] 是规定的写法
var numArr03 = [...]int{8, 9, 10}
fmt.Println("numArr03=", numArr03)

var numArr04 = [...]int{1: 800, 0: 900, 2:999}
fmt.Println("numArr04=", numArr04)

//类型推导
strArr05 := [...]string{1: "tom", 0: "jack", 2:"mary"}
fmt.Println("strArr05=", strArr05)
```

7.6 数组的遍历

7.6.1方式 1-常规遍历:

前面已经讲过了，不再赘述。

7.6.2方式 2-for-range 结构遍历

这是 Go 语言一种独有的结构，可以用来遍历访问数组的元素。

- for--range 的基本语法

```
for index, value := range array01 {
...
}
```

说明

1) 第一个返回值 index 是数组的下标
2) 第二个value是在该下标位置的值
3) 他们都是仅在 for 循环内部可见的局部变量
4) 遍历数组元素的时候，如果不想使用下标index，可以直接把下标index标为下划线_
5) index 和 value 的名称不是固定的，即程序员可以自行指定，一般命名为 index 和 value

- for-range 的案例

```go
package main
import (
    "fmt"
)

func main() {

    //演示for-range遍历数组
    heroes  := [...]string{"宋江", "吴用", "卢俊义"}
    //使用常规的方式遍历，我不写了..

    for i, v := range heroes {
        fmt.Printf("i=%v v=%v\n", i , v)
        fmt.Printf("heroes[%d]=%v\n", i, heroes[i])
    }

    for _, v := range heroes {
        fmt.Printf("元素的值=%v\n", v)
    }

}
```

7.7 数组使用的注意事项和细节

1) 数组是多个相同类型数据的组合,一个数组一旦声明/定义了,其长度是固定的, 不能动态变化

```go
package main
import (
    "fmt"
)

func main() {

    //数组是多个相同类型数据的组合,一个数组一旦声明/定义了,其长度是固定的, 不能动态变化。
    var arr01 [3]int
    arr01[0] = 1
    arr01[1] = 30
    //这里会报错
    arr01[2] = 1.1
    //其长度是固定的, 不能动态变化,否则报越界
    arr01[3] = 890

    fmt.Println(arr01)

}
```

因为数组的类型是int，就不能给 1.1

数组不能动态增长

2) var arr []int 这时 arr 就是一个 slice 切片，切片后面专门讲解，不急哈.

3) 数组中的元素可以是任何数据类型，包括值类型和引用类型，但是不能混用。

4) 数组创建后，如果没有赋值，有默认值(零值)

数值类型数组：默认值为 0

字符串数组： 默认值为 ""

bool 数组： 默认值为 false

```go
//数组创建后, 如果没有赋值, 有默认值(零值)
//1. 数值(整数系列, 浮点数系列) =>0
//2. 字符串 ==> ""
//3. 布尔类型 ==> false

var arr01 [3]float32
var arr02 [3]string
var arr03 [3]bool
fmt.Printf("arr01=%v arr02=%v arr03=%v\n", arr01, arr02, arr03)
```

5) 使用数组的步骤 1. 声明数组并开辟空间 2 给数组各个元素赋值(默认零值) 3 使用数组

6) 数组的下标是从 0 开始的

```go
//数组的下标是从0开始的

var arr04 [3]string // 0 - 2
var index int = 3
arr04[index] = "tom" // 因为下标是 0 - 2 ,因此arr04[3]就越界
```

7) 数组下标必须在指定范围内使用，否则报 panic：数组越界，比如

var arr [5]int 则有效下标为 0-4

8) Go 的数组属值类型， 在默认情况下是值传递， 因此会进行值拷贝。数组间不会相互影响

9) 如想在其它函数中，去修改原来的数组，可以使用引用传递(指针方式)

10) 长度是数组类型的一部分，在传递函数参数时 需要考虑数组的长度，看下面案例

不能将数组传递给切片

```
//默认值拷贝
func modify(arr []int){
arr[0] = 100
fmt.Println("modify 的arr ", arr)
}

func main() {
var arr = [...]int{1, 2, 3}
modify(arr)
}
```

题1
编译错误，因为不能把 [3]int 传递给 []int

```
//默认值拷贝
func modify(arr [4]int){
arr[0] = 100
fmt.Println("modify 的arr ", arr)
}

func main() {
var arr = [...]int{1, 2, 3}
modify(arr)
}
```

题2
编译错误，因为不能把 [3]int 传递给 [4]int

```
//默认值拷贝
func modify(arr [3]int){
arr[0] = 100
fmt.Println("modify 的arr ", arr)
}

func main() {
var arr = [...]int{1, 2, 3}
modify(arr)
}
```

题3, **这个正确**..

7.8 数组的应用案例

1) 创建一个 byte 类型的 26 个元素的数组，分别 放置'A'-'Z‘。使用 for 循环访问所有元素并打印出来。提示：字符数据运算 'A'+1 -> 'B'

```go
func main() {
    //1)创建一个byte类型的26个元素的数组，分别 放置'A'-'Z'。
    //使用for循环访问所有元素并打印出来。提示：字符数据运算 'A'+1 -> 'B'

    //思路
    //1. 声明一个数组 var myChars [26]byte
    //2. 使用for循环，利用 字符可以进行运算的特点来赋值 'A'+1 -> 'B'
    //3. 使用for打印即可
    //代码:
    var myChars [26]byte
    for i := 0; i < 26; i++ {
        myChars[i] = 'A' + byte(i) // 注意需要将 i => byte
    }

    for i := 0; i < 26; i++ {
        fmt.Printf("%c ", myChars[i])
    }

}
```

2) 请求出一个数组的最大值，并得到对应的下标。

```go
//请求出一个数组的最大值，并得到对应的下标

//思路
//1. 声明一个数组 var intArr[5] = [...]int {1, -1, 9, 90, 11}
//2. 假定第一个元素就是最大值，下标就0
//3. 然后从第二个元素开始循环比较，如果发现有更大，则交换

fmt.Println()
var intArr [6]int = [...]int {1, -1, 9, 90, 11, 9000}
maxVal := intArr[0]
maxValIndex := 0

for i := 1; i < len(intArr); i++ {
    //然后从第二个元素开始循环比较，如果发现有更大，则交换
    if maxVal < intArr[i] {
        maxVal = intArr[i]
        maxValIndex = i
    }
}
fmt.Printf("maxVal=%v maxValIndex=%v", maxVal, maxValIndex)
```

3) 请求出一个数组的和和平均值。for-range

```go
//请求出一个数组的和和平均值。for-range
//思路
//1. 就是声明一个数组  var intArr[5] = [...]int {1, -1, 9, 90, 11}
//2. 求出和sum
//3. 求出平均值
//代码
var intArr2 [5]int = [...]int {1, -1, 9, 90, 12}
sum := 0
for _, val := range intArr2 {
    //累计求和
    sum += val
}

//如何让平均值保留到小数.
fmt.Printf("sum=%v 平均值=%v", sum, float64(sum) / float64(len(intArr2)))
```

4) 要求：随机生成五个数，并将其反转打印 ，复杂应用.

```go
//要求：随机生成五个数，并将其反转打印
//思路
//1. 随机生成五个数 ，rand.Intn() 函数
//2. 当我们得到随机数后，就放到一个数组 int数组
//3. 反转打印 ，交换的次数是  len / 2，倒数第一个和第一个元素交换，倒数第2个和第2个元素交

var intArr3 [5]int
//为了每次生成的随机数不一样，我们需要给一个seed值
len := len(intArr3)

rand.Seed(time.Now().UnixNano())
for i := 0; i < len; i++ {
    intArr3[i] = rand.Intn(100) //  0<=n<100
}
fmt.Println("交换前~=", intArr3)
//反转打印 ，交换的次数是  len / 2,
//倒数第一个和第一个元素交换，倒数第2个和第2个元素交换
temp := 0  //做一个临时变量
for i := 0; i < len / 2; i++ {
    temp = intArr3[len - 1 - i]
    intArr3[len - 1 - i] = intArr3[i]
    intArr3[i] = temp
}
fmt.Println("交换后~=", intArr3)
```

7.9 为什么需要切片

先看一个需求：我们需要一个数组用于保存学生的成绩，但是**学生的个数**是不确定的，请问怎么办？解决方案：-》使用**切片**。

7.10 切片的基本介绍

1) 切片的英文是 slice
2) 切片是数组的一个引用，因此**切片是引用类型**，在进行传递时，遵守引用传递的机制。
3) 切片的**使用和数组类似**，遍历切片、访问切片的元素和求切片长度 len(slice)都一样。
4) 切片的长度是可以变化的，因此切片是一个**可以动态变化数组**。
5) 切片定义的基本语法:

   var 切片名 []类型

   比如：var a [] int

7.11 快速入门

演示一个切片的基本使用：

```go
func main() {

    //演示切片的基本使用
    var intArr [5]int = [...]int{1, 22, 33, 66, 99}
    //声明/定义一个切片
    //slice := intArr[1:3]
    //1. slice 就是切片名
    //2. intArr[1:3] 表示 slice 引用到intArr这个数组
    //3. 引用intArr数组的起始下标为 1 , 最后的下标为3(但是不包含3)
    slice := intArr[1:3]
    fmt.Println("intArr=", intArr)
    fmt.Println("slice 的元素是 =", slice) //  22, 33
    fmt.Println("slice 的元素个数 =", len(slice)) // 2
    fmt.Println("slice 的容量 =", cap(slice)) // 切片的容量是可以动态变化
}
```

运行结果是：

```
D:\goproject\src\go_code\chapter07\slicedemo01>g
intArr= [1 22 33 66 99]
slice 的元素是 = [22 33]
slice 的元素个数 = 2
slice 的容量 = 4
```

7.12 切片在内存中形式(重要)

➢ 基本的介绍：

为了让大家更加深入的理解切片，我们画图分析一下切片在内存中是如何布局的，这个是一个非常重要的知识点：(以前面的案例来分析)

➢ 画出前面的切片内存布局

```go
//演示切片的基本使用
var intArr [5]int = [...]int{1, 22, 33, 66, 99}
//声明/定义一个切片
//slice := intArr[1:3]
//1. slice 就是切片名
//2. intArr[1:3] 表示 slice 引用到intArr这个数组
//3. 引用intArr数组的起始下标为 1 , 最后的下标为3(但是不包含3)
slice := intArr[1:3]
fmt.Println("intArr=", intArr)
fmt.Println("slice 的元素是 =", slice) //  22, 33
fmt.Println("slice 的元素个数 =", len(slice)) // 2
fmt.Println("slice 的容量 =", cap(slice)) // 切片的容量是可以动态变化
```

➢ 对上面的分析图总结

1. slice 的确是一个引用类型

2. slice 从底层来说，其实就是一个数据结构(struct 结构体)

```
type slice struct {
    ptr    *[2]int
    len    int
    cap    int
}
```

7.13 切片的使用

➢ 方式 1

第一种方式：定义一个切片，然后让切片去引用一个已经创建好的数组，比如前面的案例就是这样的。

```go
func main() {

    var arr [5]int = [...]int {1,2,3,4,5}
    var slice = arr[1:3]
    fmt.Println("arr=", arr)
    fmt.Println("slice=", slice)
    fmt.Println("slice len =", len(slice))
    fmt.Println("slice cap =", cap(slice))

}
```

➢ 方式 2

第二种方式：通过 **make** 来创建切片.

基本语法：**var 切片名 []type = make([]type, len, [cap])**

参数说明: type: 就是数据类型　len：大小　cap ：指定切片容量，**可选， 如果你分配了 cap,则要求 cap>=len.**

案例演示：

对上面代码的小结:

1) 通过 make 方式创建切片可以指定切片的大小和容量

2) 如果没有给切片的各个元素赋值，那么就会使用默认值[int , float=> 0　string =>""　bool => false]

3) 通过 make 方式创建的切片对应的数组是由 make 底层维护，对外不可见，即只能通过 slice 去访问各个元素.

➢ 方式 3

第 3 种方式：定义一个切片，直接就指定具体数组，使用原理类似 make 的方式

案例演示：

```
//方式3
fmt.Println()
//第3种方式：定义一个切片，直接就指定具体数组，使用原理类似make的方式
var strSlice []string = []string{"tom", "jack", "mary"}
fmt.Println("strSlice=", strSlice)
fmt.Println("strSlice size=", len(strSlice)) //3
fmt.Println("strSlice cap=", cap(strSlice)) // ?
```

➢ 方式 1 和方式 2 的区别(面试)

方式1是直接引用数组，这个数组是事先存在的，程序员是可见的。
方式2是通过make来创建切片，make也会创建一个数组，是由切片在底层进行维护，程序员是看不见的。make创建切片的示意图：

7.14 切片的遍历

切片的遍历和数组一样，也有两种方式

- for 循环常规方式遍历
- for-range 结构遍历切片

```go
func main() {

    //使用常规的for循环遍历切片
    var arr [5]int = [...]int{10, 20, 30, 40, 50}
    slice := arr[1:4] // 20, 30, 40
    for i := 0; i < len(slice); i++ {
        fmt.Printf("slice[%v]=%v ", i, slice[i])
    }

    fmt.Println()
    //使用for--range 方式遍历切片
    for i, v := range slice {
        fmt.Printf("i=%v v=%v \n", i, v)
    }
}
```

7.15 切片的使用的注意事项和细节讨论

1) 切片初始化时 var slice = arr[startIndex:endIndex]

说明：从 arr 数组下标为 startIndex，取到 下标为 endIndex 的元素(不含 arr[endIndex])。

2) 切片初始化时，仍然不能越界。范围在 [0-len(arr)] 之间，但是可以动态增长.

var slice = arr[0:end] 可以简写 var slice = arr[:end]

var slice = arr[start:len(arr)] 可以简写： var slice = arr[start:]

var slice = arr[0:len(arr)] 可以简写: var slice = arr[:]

3) cap 是一个内置函数，用于统计切片的容量，即最大可以存放多少个元素。

4) 切片定义完后，还不能使用，因为本身是一个空的，需要让其引用到一个数组，或者 make 一个空间供切片来使用

5) 切片可以继续切片[案例演示]

```
//使用常规的for循环遍历切片
var arr [5]int = [...]int{10, 20, 30, 40, 50}
//slice := arr[1:4] // 20, 30, 40
slice := arr[1:4]
for i := 0; i < len(slice); i++ {
    fmt.Printf("slice[%v]=%v ", i, slice[i])
}

fmt.Println()
//使用for--range 方式遍历切片
for i, v := range slice {
    fmt.Printf("i=%v v=%v \n", i, v)
}

slice2 := slice[1:2] //  slice [ 20, 30, 40]    [30]
slice2[0] = 100  // 因为arr , slice 和slice2 指向的数据空间是同一个，因此slice2[0]=100,

fmt.Println("slice2=", slice2)
fmt.Println("slice=", slice)
fmt.Println("arr=", arr)
```

6) 用 append 内置函数，可以对切片进行动态追加

```
//用append内置函数，可以对切片进行动态追加
var slice3 []int = []int{100, 200, 300}
//通过append直接给slice3追加具体的元素
slice3 = append(slice3, 400, 500, 600)
fmt.Println("slice3", slice3) //100, 200, 300,400, 500, 600

//通过append将切片slice3追加给slice3
slice3 = append(slice3, slice3...) // 100, 200, 300,400, 500, 600 100, 200, 300,40
fmt.Println("slice3", slice3)
```

对上面代码的小结

切片 append 操作的底层原理分析:

切片 append 操作的本质就是对数组扩容

go 底层会创建一下新的数组 newArr(安装扩容后大小)

将 slice 原来包含的元素拷贝到新的数组 newArr

slice 重新引用到 newArr

注意 newArr 是在底层来维护的，程序员不可见.

7) 切片的拷贝操作

切片使用 copy 内置函数完成拷贝，举例说明

```
//切片的拷贝操作
//切片使用copy内置函数完成拷贝，举例说明
fmt.Println()
var slice4 []int = []int{1, 2, 3, 4, 5}
var slice5 = make([]int, 10)
copy(slice5, slice4)
fmt.Println("slice4=", slice4)// 1, 2, 3, 4, 5
fmt.Println("slice5=", slice5) // 1, 2, 3, 4, 5, 0 , 0 ,0,0,0
```

对上面代码的说明:

(1) copy(para1, para2) 参数的数据类型是切片

(2) 按照上面的代码来看, slice4 和 slice5 的数据空间是独立，相互不影响，也就是说 slice4[0]= 999, slice5[0] 仍然是 1

8) 关于拷贝的注意事项

思考题，下面的代码有没有错误：

```
var a []int = []int {1,2,3,4,5}
var slice = make([]int, 1)
fmt.Println(slice) // [0]
copy(slice, a)
fmt.Println(slice) // [1]
```

说明: 上面的代码没有问题，可以运行，最后输出的是 [1]

9) 切片是引用类型，所以在传递时，遵守引用传递机制。看两段代码，并分析底层原理

```
func main() {
var slice []int
var arr [5]int = [...]int {1, 2, 3, 4, 5}
slice = arr[:]
var slice2 = slice
slice2[0] = 10

fmt.Println("slice2", slice2) //? [10,2,3,4,5]
fmt.Println("slice", slice)//?[10,2,3,4,5]
fmt.Println("arr", arr)//?[10,2,3,4,5]
}
题1
```

```
func test(slice []int) {
slice[0] = 100 //这里修改slice[0],会改变实参
}

func main() {
var slice = []int {1, 2, 3, 4}
fmt.Println("slice=", slice) // [1,2,3,4]
test(slice)
fmt.Println("slice=", slice) // [100, 2, 3, 4]
}

//题2
```

7.16 string 和 slice

1) string 底层是一个 byte 数组，因此 string 也可以进行切片处理 案例演示：

```go
func main() {

    //string底层是一个byte数组，因此string也可以进行切片处理
    str := "hello@atguigu"
    //使用切片获取到 atguigu
    slice := str[6:]
    fmt.Println("slice=", slice)
}
```

2) string 和切片在内存的形式，以 "abcd" 画出内存示意图

3) string 是不可变的，也就说不能通过 str[0] = 'z' 方式来修改字符串

```go
//string是不可变的，也就说不能通过 str[0] = 'z' 方式来修改字符串
str[0] = 'z' [编译不会通过，报错，原因是string是不可变]
```

4) 如果需要修改字符串，可以先将 string -> []byte / 或者 []rune -> 修改 -> 重写转成 string

```go
//如果需要修改字符串，可以先将string -> []byte / 或者 []rune -> 修改 -> 重写转成string
// "hello@atguigu" =>改成 "zello@atguigu"
arr1 := []byte(str)
arr1[0] = 'z'
str = string(arr1)
fmt.Println("str=", str)

// 细节，我们转成[]byte后，可以处理英文和数字，但是不能处理中文
// 原因是 []byte 字节来处理 ，而一个汉字，是3个字节，因此就会出现乱码
// 解决方法是 将  string 转成 []rune 即可， 因为 []rune是按字符处理，兼容汉字

arr1 := []rune(str)
arr1[0] = '北'
str = string(arr1)
fmt.Println("str=", str)
```

7.17 切片的课堂练习题

说明：编写一个函数 fbn(n int) ，要求完成

1) 可以接收一个 n int
2) 能够将斐波那契的数列放到切片中
3) 提示，斐波那契的数列形式:

arr[0] = 1; arr[1] = 1; arr[2]=2; arr[3] = 3; arr[4]=5; arr[5]=8

代码+思路：

```go
package main
import (
    "fmt"
)

func fbn(n int) ([]uint64) {
    //声明一个切片，切片大小 n
    fbnSlice := make([]uint64, n)
    //第一个数和第二个数的斐波那契 为1
    fbnSlice[0] = 1
    fbnSlice[1] = 1
    //进行for循环来存放斐波那契的数列
    for i := 2; i < n; i++ {
        fbnSlice[i] = fbnSlice[i - 1] + fbnSlice[i - 2]
    }

    return fbnSlice
}

func main() {

    /*
    1)可以接收一个 n int
    2)能够将斐波那契的数列放到切片中
    3)提示，斐波那契的数列形式:
    arr[0] = 1; arr[1] = 1; arr[2]=2; arr[3] = 3; arr[4]=5; arr[5]=8
```

```go

    思路
    1. 声明一个函数 fbn(n int) ([]uint64)
    2. 编程fbn(n int) 进行for循环来存放斐波那契的数列  0 =》 1 1 =》 1
    */

    //测试一把看看是否好用
    fnbSlice := fbn(20)
    fmt.Println("fnbSlice=", fnbSlice)

}
```

第 8 章 排序和查找

8.1 排序的基本介绍

排序是将一组数据，依指定的顺序进行排列的过程。
排序的分类：
1）内部排序：
指将需要处理的所有数据都加载到内部存储器中进行排序。
包括(交换式排序法、选择式排序法和插入式排序法)；
2）外部排序法：
数据量过大，无法全部加载到内存中，需要借助外部存储进行排序。包括(合并排序法和直接合并排序法)。

冒泡排序（Bubble Sorting）的基本思想是：通过对待排序序列从后向前（从下标较大的元素开始），依次比较相邻元素的排序码，若发现逆序则交换，使排序码较小的元素逐渐从后部移向前部（从下标较大的单元移向下标较小的单元），就象水底下的气泡一样逐渐向上冒。

因为排序的过程中，各元素不断接近自己的位置，如果一趟比较下来没有进行过交换，就说明序列有序，因此要在排序过程中设置一个标志flag判断元素是否进行过交换。从而减少不必要的比较（优化）。

下图演示了一个冒泡过程的例子：

8.2 冒泡排序的思路分析

8.3 冒泡排序实现

```go
package main
import (
    "fmt"
)

//冒泡排序
func BubbleSort(arr *[5]int) {

    fmt.Println("排序前arr=", (*arr))
    temp := 0 //临时变量(用于做交换)


    //冒泡排序..一步一步推导出来的
    for i :=0; i < len(*arr) - 1; i++ {

        for j := 0; j < len(*arr) - 1 - i; j++ {
            if (*arr)[j] > (*arr)[j + 1] {
                //交换
                temp = (*arr)[j]
                (*arr)[j] = (*arr)[j + 1]
                (*arr)[j + 1] = temp
            }
        }

    }

```

```go

    fmt.Println("排序后arr=", (*arr))

}

func main() {

    //定义数组
    arr := [5]int{24,69,80,57,13}
    //将数组传递给一个函数，完成排序

    BubbleSort(&arr)

    fmt.Println("main arr=", arr) //有序？是有序的
}

```

8.4 课后练习

要求同学们能够，不看老师的代码，可以默写冒泡排序法(笔试题)

8.5 查找

- 介绍：

  在 Golang 中，我们常用的查找有两种:

  1) **顺序查找**

  2) **二分查找**(该数组是有序)

- 案例演示：

  1) 有一个数列：白眉鹰王、金毛狮王、紫衫龙王、青翼蝠王

  猜数游戏：从键盘中任意输入一个名称，判断数列中是否包含此名称【顺序查找】

  代码：

```go
package main
import (
    "fmt"
)

func main() {
    //有一个数列：白眉鹰王、金毛狮王、紫衫龙王、青翼蝠王
    //猜数游戏：从键盘中任意输入一个名称，判断数列中是否包含此名称【顺序查找】
    //思路
    //1 定义一个数组，白眉鹰王、金毛狮王、紫衫龙王、青翼蝠王 字符串数组
    //2.从控制台接收一个名字，依次比较，如果发现有，提示

    //代码
    names := [4]string{"白眉鹰王", "金毛狮王", "紫衫龙王", "青翼蝠王"}
    var heroName = ""
    fmt.Println("请输入要查找的人名...")
    fmt.Scanln(&heroName)

    //顺序查找:第一种方式
    for i := 0; i < len(names); i++ {
        if heroName == names[i] {
            fmt.Printf("找到%v , 下标%v \n", heroName, i)
            break
        } else if i == (len(names) - 1) {
            fmt.Printf("没有找到%v \n", heroName)
        }
```

```go
    }

    //顺序查找:第2种方式(推荐...)
    index := -1

    for i := 0; i < len(names); i++ {
        if heroName == names[i] {
            index = i //将找到的值对应的下标赋给 index
            break
        }
    }
    if index != -1 {
        fmt.Printf("找到%v , 下标%v \n", heroName, index)
    } else {
        fmt.Println("没有找到", heroName)
    }


}
```

2) 请对一个**有序数组**进行二分查找 {1,8, 10, 89, 1000, 1234} ，输入一个数看看该数组是否存在此数，并且求出下标，如果没有就提示"没有这个数"。【会使用到递归】

二分查找的思路分析：

二分查找的代码实现：

```
package main
import (
    "fmt"
)

//二分查找的函数
/*
二分查找的思路: 比如我们要查找的数是 findVal
1. arr 是一个有序数组，并且是从小到大排序
2.  先找到 中间的下标 middle = (leftIndex + rightIndex) / 2, 然后让 中间下标的值和 findVal 进行比较
2.1 如果 arr[middle] > findVal ,  就应该向  leftIndex ---- (middle - 1)
2.2 如果 arr[middle] < findVal ,  就应该向  middel+1---- rightIndex
2.3 如果 arr[middle] == findVal ,  就找到
2.4 上面的 2.1 2.2 2.3 的逻辑会递归执行
3. 想一下，怎么样的情况下，就说明找不到[分析出退出递归的条件!!]
if  leftIndex > rightIndex {
    // 找不到..
    return ..
}
*/
func BinaryFind(arr *[6]int, leftIndex int, rightIndex int, findVal int) {

    //判断 leftIndex 是否大于 rightIndex
    if leftIndex > rightIndex {
```

```
        fmt.Println("找不到")
        return
    }

    //先找到 中间的下标
    middle := (leftIndex + rightIndex) / 2

    if (*arr)[middle] > findVal {
        //说明我们要查找的数，应该在   leftIndex --- middel-1
        BinaryFind(arr, leftIndex, middle - 1, findVal)
    } else if (*arr)[middle] < findVal {
        //说明我们要查找的数，应该在   middel+1 --- rightIndex
        BinaryFind(arr, middle + 1, rightIndex, findVal)
    } else {
        //找到了
        fmt.Printf("找到了，下标为%v \n", middle)
    }
}

func main() {

    arr := [6]int{1,8, 10, 89, 1000, 1234}

    //测试一把
    BinaryFind(&arr, 0, len(arr) - 1, -6)
```

```
}
```

8.6 二维数组的介绍

多维数组我们只介绍二维数组

8.7 二维数组的应用场景

比如我们开发一个五子棋游戏，棋盘就是需要二维数组来表示。如图

8.8 二维数组快速入门

快速入门案例：

- 请用二维数组输出如下图形

  0 0 0 0 0 0

  0 0 1 0 0 0

  0 2 0 3 0 0

  0 0 0 0 0 0

- 代码演示

```go
package main
import (
    "fmt"
)

func main() {
    //二维数组的演示案例
    /*
    0 0 0 0 0 0
    0 0 1 0 0 0
    0 2 0 3 0 0
    0 0 0 0 0 0
    */

    //定义/声明二维数组
    var arr [4][6]int
    //赋初值
    arr[1][2] = 1
    arr[2][1] = 2
    arr[2][3] = 3

    //遍历二维数组，按照要求输出图形
    for i := 0; i < 4; i++ {
        for j := 0; j < 6; j++ {
            fmt.Print(arr[i][j], " ")
        }
        fmt.Println()
    }
}
```

8.9 使用方式 1: 先声明/定义,再赋值

- 语法: var 数组名 [大小][大小]类型
- 比如: var arr [2][3]int ， 再赋值。
- 使用演示
- 二维数组在内存的存在形式(重点)

```go
var arr2 [2][3]int //以这个为例来分析arr2在内存的布局!!
arr2[1][1] = 10
fmt.Println(arr2)

fmt.Printf("arr2[0]的地址%p\n", &arr2[0])
fmt.Printf("arr2[1]的地址%p\n", &arr2[1])

fmt.Printf("arr2[0][0]的地址%p\n", &arr2[0][0])
fmt.Printf("arr2[1][0]的地址%p\n", &arr2[1][0])
```

8.10 使用方式 2: 直接初始化

- 声明：var 数组名 [大小][大小]类型 =[大小][大小]类型{{初值..},{初值..}}
- 赋值(有默认值，比如 int 类型的就是 0)
- 使用演示

```go
fmt.Println()
var arr3 [2][3]int = [2][3]int{{1,2,3}, {4,5,6}}
fmt.Println("arr3=", arr3)
```

- 说明：二维数组在声明/定义时也对应有四种写法[和一维数组类似]

  var 数组名 [大小][大小]类型 =[大小][大小]类型{{初值..},{初值..}}

  var 数组名 [大小][大小]类型 =[...][大小]类型{{初值..},{初值..}}

  var 数组名 =[大小][大小]类型{{初值..},{初值..}}

var 数组名 =[...][大小]类型{{初值..},{初值..}}

8.11 二维数组的遍历

- 双层 for 循环完成遍历
- for-range 方式完成遍历

案例演示：

```go
func main() {

    //演示二维数组的遍历
    var arr3  = [2][3]int{{1,2,3}, {4,5,6}}

    //for循环来遍历
    for i := 0; i < len(arr3); i++ {
        for j := 0; j < len(arr3[i]); j++ {
            fmt.Printf("%v\t", arr3[i][j])
        }
        fmt.Println()
    }

    //for-range来遍历二维数组
    for i, v := range arr3 {
        for j, v2 := range v {
            fmt.Printf("arr3[%v][%v]=%v \t",i, j, v2)
        }
        fmt.Println()
    }
}
```

8.12 二维数组的应用案例

- 要求如下：

  定义二维数组，用于保存三个班，每个班五名同学成绩，

  并求出每个班级平均分、以及所有班级平均分

- 代码

```go
package main
import (
    "fmt"
)

func main() {

    /*
    定义二维数组，用于保存三个班，每个班五名同学成绩，
    并求出每个班级平均分、以及所有班级平均分
    */

    //1.定义一个二维数组
    var scores [3][5]float64
    //2.循环的输入成绩
    for i := 0; i < len(scores); i++ {
        for j := 0; j < len(scores[i]); j++ {
            fmt.Printf("请输入第%d班的第%d个学生的成绩\n", i+1, j+1)
            fmt.Scanln(&scores[i][j])
        }
    }

    //fmt.Println(scores)

    //3.遍历输出成绩后的二维数组，统计平均分
    totalSum := 0.0 // 定义一个变量，用于累计所有班级的总分
```

```go
    for i := 0; i < len(scores); i++ {
        sum := 0.0 //定义一个变量，用于累计各个班级的总分
        for j := 0; j < len(scores[i]); j++ {
            sum += scores[i][j]
        }
        totalSum += sum
        fmt.Printf("第%d班级的总分为%v ， 平均分%v\n", i+1, sum,
            sum / float64(len(scores[i])))
    }

    fmt.Printf("所有班级的总分为%v ， 所有班级平均分%v\n",
        totalSum, totalSum / 15 )
}
```

第 9 章 map

9.1 map 的基本介绍

map 是 key-value 数据结构，又称为字段或者关联数组。类似其它编程语言的集合，在编程中是经常使用到

9.2 map 的声明

9.2.1基本语法

var map 变量名 map[keytype]valuetype

- key 可以是什么类型

  golang 中的 map，的 key 可以是很多种类型，比如 bool, 数字，string, 指针, channel , 还可以是只包含前面几个类型的 接口, 结构体, 数组

  通常 key 为 int 、string

  注意: slice， map 还有 function 不可以，因为这几个没法用 == 来判断

- valuetype 可以是什么类型

  valuetype 的类型和 key 基本一样，这里我就不再赘述了

  通常为: 数字(整数,浮点数),string,map,struct

9.2.2map 声明的举例

- map 声明的举例：

  var a map[string]string

  var a map[string]int

  var a map[int]string

  var a map[string]map[string]string

  注意：声明是不会分配内存的，初始化需要 make ，分配内存后才能赋值和使用。

  案例演示：

```go
package main
import (
    "fmt"
)

func main() {
    //map的声明和注意事项
    var a map[string]string
    //在使用map前，需要先make ，make的作用就是给map分配数据空间
    a = make(map[string]string, 10)
    a["no1"] = "宋江" //ok?
    a["no2"] = "吴用" //ok?
    a["no1"] = "武松" //ok?
    a["no3"] = "吴用" //ok?
    fmt.Println(a)
}
```

➢ 对上面代码的说明

1) map 在使用前一定要 make
2) map 的 key 是不能重复，如果重复了，则以最后这个 key-value 为准
3) map 的 value 是可以相同的.
4) map 的 key-value 是无序
5) make 内置函数数目

func make

make 1/6

```go
func make(Type, size IntegerType) Type
```

内建函数make分配并初始化一个类型为切片、映射、或通道的对象。其第一个实参为类型，而非值。make的返回类型与其参数相同，而非指向它的指针。其具体结果取决于具体的类型：

切片：size指定了其长度。该切片的容量等于其长度。切片支持第二个整数实参可用来指定不同的容量；它必须不小于其长度，因此 make([]int, 0, 10) 会分配一个长度为0，容量为10的切片。
映射：初始分配的创建取决于size，但产生的映射长度为0。size可以省略，这种情况下就会分配一个小的起始大小。
通道：通道的缓存根据指定的缓存容量初始化。若 size为零或被省略，该信道即为无缓存的。

9.3 map 的使用

➢ 方式 1

```go
//第一种使用方式

var a map[string]string
//在使用map前，需要先make ，make的作用就是给map分配数据空间
a = make(map[string]string, 10)
a["no1"] = "宋江" //ok?
a["no2"] = "吴用" //ok?
a["no1"] = "武松" //ok?
a["no3"] = "吴用" //ok?
fmt.Println(a)
```

- 方式 2

```go
//第二种方式
cities := make(map[string]string)
cities["no1"] = "北京"
cities["no2"] = "天津"
cities["no3"] = "上海"
fmt.Println(cities)
```

- 方式 3

```go
//第三种方式
heroes := map[string]string{
    "hero1" : "宋江",
    "hero2" : "卢俊义",
    "hero3" : "吴用",
}
heroes["hero4"] = "林冲"
fmt.Println("heroes=", heroes)
```

- map 使用的课堂案例

课堂练习：演示一个 key-value 的 value 是 map 的案例

比如：我们要存放 3 个学生信息，每个学生有 name 和 sex 信息

思路: map[string]map[string]string

代码：

```go
//案例
/*
课堂练习：演示一个key-value 的value是map的案例
比如：我们要存放3个学生信息，每个学生有 name和sex 信息
思路：   map[string]map[string]string

*/
studentMap := make(map[string]map[string]string)

studentMap["stu01"] =  make(map[string]string, 3)
studentMap["stu01"]["name"] = "tom"
studentMap["stu01"]["sex"] = "男"
studentMap["stu01"]["address"] = "北京长安街~"

studentMap["stu02"] =  make(map[string]string, 3) //这句话不能少!!
studentMap["stu02"]["name"] = "mary"
studentMap["stu02"]["sex"] = "女"
studentMap["stu02"]["address"] = "上海黄浦江~"

fmt.Println(studentMap)
fmt.Println(studentMap["stu02"])
fmt.Println(studentMap["stu02"]["address"])
```

9.4 map 的增删改查操作

➢ map 增加和更新：

map["key"] = value //**如果 key 还没有，就是增加，如果 key 存在就是修改**。

```go
func main() {
    //第二种方式
    cities := make(map[string]string)
    cities["no1"] = "北京"
    cities["no2"] = "天津"
    cities["no3"] = "上海"
    fmt.Println(cities)

    //因为 no3这个key已经存在，因此下面的这句话就是修改
    cities["no3"] = "上海~"
    fmt.Println(cities)
}
```

- map 删除：

说明：

delete(map，"key") ，delete 是一个内置函数，如果 key 存在，就删除该 key-value,如果 key 不存在，不操作，但是也不会报错

func delete

```
func delete(m map[Type]Type1, key Type)
```

内建函数delete按照指定的键将元素从映射中删除。若m为nil或无此元素，delete不进行操作。

案例演示：

```
//演示删除
delete(cities, "no1")
fmt.Println(cities)
//当delete指定的key不存在时，删除不会操作，也不会报错
delete(cities, "no4")
fmt.Println(cities)
```

- 细节说明

如果我们要删除 map 的所有 key ,没有一个专门的方法一次删除，可以遍历一下 key, 逐个删除

或者 map = make(...)，make 一个新的，让原来的成为垃圾，被 gc 回收

```
//如果希望一次性删除所有的key
//1. 遍历所有的key,如何逐一删除 [遍历]
//2. 直接make一个新的空间
cities = make(map[string]string)
fmt.Println(cities)
```

- map 查找：

案例演示：

```go
//演示map的查找
val, ok := cities["no2"]
if ok {
    fmt.Printf("有no1 key 值为%v\n", val)
} else {
    fmt.Printf("没有no1 key\n")
}
```

对上面代码的说明:

说明：如果 heroes 这个 map 中存在 "no1" ， 那么 findRes 就会返回 true,否则返回 false

9.5 map 遍历：

案例演示相对复杂的 map 遍历：该 map 的 value 又是一个 map

说明：map 的遍历使用 **for-range** 的结构遍历

➢ 案例演示：

```go
package main
import (
    "fmt"
)

func main() {
    //使用for-range遍历map
    //第二种方式
    cities := make(map[string]string)
    cities["no1"] = "北京"
    cities["no2"] = "天津"
    cities["no3"] = "上海"

    for k, v := range cities {
        fmt.Printf("k=%v v=%v\n", k, v)
    }

    //使用for-range遍历一个结构比较复杂的map
    studentMap := make(map[string]map[string]string)

    studentMap["stu01"] =  make(map[string]string, 3)
    studentMap["stu01"]["name"] = "tom"
    studentMap["stu01"]["sex"] = "男"
    studentMap["stu01"]["address"] = "北京长安街~"

    studentMap["stu02"] =  make(map[string]string, 3) //这句话不能少!!
```

```go
    studentMap["stu02"]["name"] = "mary"
    studentMap["stu02"]["sex"] = "女"
    studentMap["stu02"]["address"] = "上海黄浦江~"

    for k1, v1 := range studentMap {
        fmt.Println("k1=", k1)
        for k2, v2 := range v1 {
            fmt.Printf("\t k2=%v v2=%v\n", k2, v2)
        }
        fmt.Println()
    }
}
```

➢ map 的长度：

func len

```go
func len(v Type) int
```

内建函数len返回 v 的长度，这取决于具体类型：

```
数组：v中元素的数量
数组指针：*v中元素的数量（v为nil时panic）
切片、映射：v中元素的数量；若v为nil，len(v)即为零
字符串：v中字节的数量
通道：通道缓存中队列（未读取）元素的数量；若v为 nil，len(v)即为零
```

案例演示： **fmt.Println(len(stus))**

9.6 map 切片

9.6.1基本介绍

切片的数据类型如果是 map，则我们称为 slice of map，**map 切片**，这样使用则 **map 个数就可以动态变化了**。

9.6.2案例演示

要求：使用一个 map 来记录 monster 的信息 name 和 age，也就是说一个 monster 对应一个 map,并且妖怪的个数可以动态的增加=>**map 切片**

代码:

```go
package main
import (
    "fmt"
)

func main() {
    //演示map切片的使用
    /*
    要求：使用一个map来记录monster的信息 name 和 age，也就是说一个
    monster对应一个map,并且妖怪的个数可以动态的增加=>map切片
    */
    //1. 声明一个map切片
    var monsters []map[string]string
    monsters = make([]map[string]string, 2) //准备放入两个妖怪
    //2. 增加第一个妖怪的信息
    if monsters[0] == nil {
        monsters[0] = make(map[string]string, 2)
        monsters[0]["name"] = "牛魔王"
        monsters[0]["age"] = "500"
    }

    if monsters[1] == nil {
        monsters[1] = make(map[string]string, 2)
        monsters[1]["name"] = "玉兔精"
        monsters[1]["age"] = "400"
    }
```

```go
    // 下面这个写法越界。
    // if monsters[2] == nil {
    //  monsters[2] = make(map[string]string, 2)
    //  monsters[2]["name"] = "狐狸精"
    //  monsters[2]["age"] = "300"
    // }

    //这里我们需要使用到切片的append函数，可以动态的增加monster
    //1. 先定义个monster信息
    newMonster := map[string]string{
        "name" : "新的妖怪~火云邪神",
        "age" : "200",
    }
    monsters = append(monsters, newMonster)

    fmt.Println(monsters)
}
```

9.7 map 排序

9.7.1基本介绍

1) golang 中没有一个专门的方法针对 map 的 key 进行排序

2) golang 中的 map 默认是无序的，注意也不是按照添加的顺序存放的，你每次遍历，得到的输出可能不一样. 【案例演示 1】

3) golang 中 map 的排序，是先将 key 进行排序，然后根据 key 值遍历输出即可

9.7.2案例演示

```go
package main
import (
    "fmt"
    "sort"
)

func main() {

    //map的排序
    map1 := make(map[int]int, 10)
    map1[10] = 100
    map1[1] = 13
    map1[4] = 56
    map1[8] = 90

    fmt.Println(map1)

    //如果按照map的key的顺序进行排序输出
    //1. 先将map的key 放入到 切片中
    //2. 对切片排序
    //3. 遍历切片，然后按照key来输出map的值

    var keys []int
    for k, _ := range map1 {
        keys = append(keys, k)
    }
```

```go
    //排序
    sort.Ints(keys)
    fmt.Println(keys)

    for _, k := range keys{
        fmt.Printf("map1[%v]=%v \n", k, map1[k])
    }

}
```

9.8 map 使用细节

1) map 是引用类型，遵守引用类型传递的机制，在一个函数接收 map，修改后，会直接修改原来的 map 【案例演示】

```go
package main
import (
    "fmt"
)
func modify(map1 map[int]int) {
    map1[10] = 900
}

func main() {

    //map是引用类型，遵守引用类型传递的机制，在一个函数接收map，
    //修改后，会直接修改原来的map

    map1 := make(map[int]int)
    map1[1] = 90
    map1[2] = 88
    map1[10] = 1
    map1[20] = 2
    modify(map1)
    // 看看结果， map1[10] = 900 ,说明map是引用类型
    fmt.Println(map1)

}
```

2) map 的容量达到后，再想 map 增加元素，会自动扩容，并不会发生 panic，也就是说 map 能动态的增长 键值对(key-value)

3) map 的 **value 也经常使用 struct 类型**，更适合管理复杂的数据(比前面 value 是一个 map 更好)，

比如 value 为 Student 结构体 【案例演示，因为还没有学结构体，**体验一下即可**】

```
//map的value 也经常使用struct 类型，
//更适合管理复杂的数据(比前面value是一个map更好)，
//比如value为 Student结构体 【案例演示，因为还没有学结构体，体验一下即可】
//1.map 的 key 为 学生的学号，是唯一的
//2.map 的 value为结构体，包含学生的 名字，年龄，地址

students := make(map[string]Stu, 10)
//创建2个学生
stu1 := Stu{"tom", 18, "北京"}
stu2 := Stu{"mary", 28, "上海"}
students["no1"] = stu1
students["no2"] = stu2

fmt.Println(students)

//遍历各个学生信息
for k, v := range students {
    fmt.Printf("学生的编号是%v \n", k)
    fmt.Printf("学生的名字是%v \n", v.Name)
    fmt.Printf("学生的年龄是%v \n", v.Age)
    fmt.Printf("学生的地址是%v \n", v.Address)
    fmt.Println()
}
```

9.9 map 的课堂练习题

➢ 课堂练习:

1) 使用 map[string]map[string]sting 的 map 类型

2) key: 表示用户名，是唯一的，不可以重复

3) 如果某个用户名存在，就将其密码修改"888888"，如果不存在就增加这个用户信息,（包括昵称 nickname 和 密码 pwd）。

4) 编写一个函数 modifyUser(users map[string]map[string]sting, name string) 完成上述功能

➢ 代码实现

```
package main
import (
    "fmt"
)

/*

1)使用  map[string]map[string]sting  的 map 类型
2)key:  表示用户名，是唯一的，不可以重复
3)如果某个用户名存在，就将其密码修改"888888"，如果不存在就增加这个用户信息,
（包括昵称 nickname  和  密码 pwd）。
4)编写一个函数  modifyUser(users map[string]map[string]sting, name string)  完成上述功能
*/

func modifyUser(users map[string]map[string]string, name string) {

    //判断 users 中是否有 name
    //v , ok := users[name]
    if users[name] != nil {
        //有这个用户
        users[name]["pwd"] = "888888"
    } else {
        //没有这个用户
        users[name] = make(map[string]string, 2)
        users[name]["pwd"] = "888888"
        users[name]["nickname"] = "昵称~" + name //示意
```

```
    }

}

func main() {

    users := make(map[string]map[string]string, 10)
    users["smith"] = make(map[string]string, 2)
    users["smith"]["pwd"] = "999999"
    users["smith"]["nickname"] = "小花猫"

    modifyUser(users, "tom")
    modifyUser(users, "mary")
    modifyUser(users, "smith")

    fmt.Println(users)

}
```

第 10 章面向对象编程(上)

10.1 结构体

10.1.1 看一个问题

张老太养了两只猫猫:一只名字叫小白,今年3岁,白色。还有一只叫小花,今年100岁,花色。请编写一个程序，当用户输入小猫的名字时，就显示该猫的名字，年龄，颜色。如果用户输入的小猫名错误，则显示 张老太没有这只猫猫。

10.1.2 使用现有技术解决

1) 单独的定义变量解决

代码演示：

```
//1. 使用变量的处理
var cat1Name string = "小白"
var cat1Age int = 3
var cat1Color string = "白色"

var cat2Name string = "小花"
var cat2Age int = 100
var cat2Color string = "花色"
```

2) 使用数组解决

代码演示：

```
//2. 使用数组解决
var catNames [2]string = [...]string{"小白", "小花"}
var catAges [2]int = [...]int{3, 100}
var catColors [2]string = [...]string{"白色", "花色"}
//... map[string]string
```

10.1.3 现有技术解决的缺点分析

1) 使用变量或者数组来解决养猫的问题，不利于数据的管理和维护。因为名字，年龄，颜色都是属于一只猫，但是这里是分开保存。

2) 如果我们希望对一只猫的属性（名字、年龄，颜色）进行操作(绑定方法)， 也不好处理。

3) 引出我们要讲解的技术-》**结构体**。

10.1.4 一个程序就是一个世界，有很多对象(变量)

10.1.5 Golang 语言面向对象编程说明

1) Golang 也支持面向对象编程(OOP)，但是和传统的面向对象编程有区别，并不是纯粹的面向对象语言。所以我们说 Golang 支持面向对象编程特性是比较准确的。

2) Golang 没有类(class)，Go 语言的结构体(struct)和其它编程语言的类(class)有同等的地位，你可以理解 Golang 是基于 struct 来实现 OOP 特性的。

3) Golang 面向对象编程非常简洁，去掉了传统 OOP 语言的继承、方法重载、构造函数和析构函数、隐藏的 this 指针等等

4) Golang 仍然有面向对象编程的**继承，封装和多态**的特性，只是实现的方式和其它 OOP 语言不一样，比如继承 ：Golang 没有 extends 关键字，继承是通过匿名字段来实现。

5) Golang 面向对象(OOP)很优雅，OOP 本身就是语言类型系统(type system)的一部分，通过接口(interface)关联，耦合性低，也非常灵活。后面同学们会充分体会到这个特点。也就是说在 Golang 中面向接口编程是非常重要的特性。

10.1.6 结构体与结构体变量(实例/对象)的关系示意图

➢ 对上图的说明

1) 将一类事物的特性提取出来(比如猫类)， 形成一个新的数据类型， 就是一个结构体。

2) 通过这个结构体，我们可以创建多个变量(实例/对象)

3) 事物可以猫类，也可以是 Person ， Fish 或是某个工具类。。。

➢ 上图说明
注意:从猫结构体到变量，就是创建一个Cat结构体变量，也可以说是定义一个Cat结构体变量.
当然：上面的猫也可是鱼、狗、人。

10.1.7 快速入门-面向对象的方式(struct)解决养猫问题

➢ 代码演示

```go
//定义一个Cat结构体，将Cat的各个字段/属性信息，放入到Cat结构体进行管理
type Cat struct {
    Name string
    Age int
    Color string
    Hobby string
}
```

```go
// 创建一个Cat的变量
var cat1 Cat  // var a int
cat1.Name = "小白"
cat1.Age = 3
cat1.Color = "白色"
cat1.Hobby = "吃<·)))><<"
fmt.Println("cat1=", cat1)

fmt.Println("猫猫的信息如下：")
fmt.Println("name=", cat1.Name)
fmt.Println("Age=", cat1.Age)
fmt.Println("color=", cat1.Color)
fmt.Println("hobby=", cat1.Hobby)
```

10.1.8 结构体和结构体变量(实例)的区别和联系

通过上面的案例和讲解我们可以看出:

1) 结构体是自定义的数据类型，代表一类事物.

2) 结构体变量(实例)是具体的，实际的，代表一个具体变量

10.1.9 结构体变量(实例)在内存的布局(重要!)

10.1.10 如何声明结构体

➢ 基本语法

```
type 结构体名称 struct {
    field1 type
    field2 type
}
```

➢ 举例:

```
type Student struct {
    Name string //字段
    Age int //字段
    Score float32
}
```

10.1.11 字段/属性

➢ 基本介绍

1) 从概念或叫法上看： 结构体字段 = 属性 = field （即授课中，统一叫字段)

2) 字段是结构体的一个组成部分，一般是**基本数据类型**、**数组**,也可是**引用类型**。比如我们前面定义猫结构体 的 Name string 就是属性

➢ 注意事项和细节说明

1) 字段声明语法同变量，示例：字段名 字段类型

2) 字段的类型可以为：基本类型、数组或引用类型

3) 在创建一个结构体变量后，如果没有给字段赋值，都对应一个零值(默认值)，规则同前面讲的一样:

布尔类型是 false ，数值是 0 ，字符串是 ""。

数组类型的默认值和它的元素类型相关，比如 score [3]int 则为[0, 0, 0]

指针，slice，和 map 的零值都是 nil ，即还没有分配空间。

案例演示：

```go
package main
import (
    "fmt"
)

//如果结构体的字段类型是: 指针，slice，和map的零值都是 nil ，即还没有分配空间
//如果需要使用这样的字段，需要先make，才能使用.

type Person struct{
    Name string
    Age int
    Scores [5]float64
    ptr *int //指针
    slice []int //切片
    map1 map[string]string //map
}


func main() {

    //定义结构体变量
    var p1 Person
    fmt.Println(p1)

    if p1.ptr == nil {
        fmt.Println("ok1")
```

```go
    }

    if p1.slice == nil {
        fmt.Println("ok2")
    }

    if p1.map1 == nil {
        fmt.Println("ok3")
    }

    //使用slice，再次说明，一定要make
    p1.slice = make([]int, 10)
    p1.slice[0] = 100 //ok

    //使用map，一定要先make
    p1.map1 = make(map[string]string)
    p1.map1["key1"] = "tom~"

    fmt.Println(p1)



}
```

4) 不同结构体变量的字段是独立，互不影响，一个结构体变量字段的更改，不影响另外一个，**结构体是值类型。**

案例：

```go
type Monster struct{
    Name string
    Age int
}
```

```go
//不同结构体变量的字段是独立，互不影响，一个结构体变量字段的更改，
//不影响另外一个，结构体是值类型
var monster1 Monster
monster1.Name = "牛魔王"
monster1.Age = 500

monster2 := monster1 //结构体是值类型，默认为值拷贝
monster2.Name = "青牛精"

fmt.Println("monster1=", monster1) //monster1= {牛魔王 500}
fmt.Println("monster2=", monster2) //monster2= {青牛精 500}
```

画出上面代码的内存示意图：

10.1.12 创建结构体变量和访问结构体字段

- 方式 1-直接声明

  案例演示: var person Person

  前面我们已经说了。

- 方式 2-{}

  案例演示: var person Person = Person{}

```go
//方式2
p2 := Person{"mary", 20}
// p2.Name = "tom"
// p2.Age = 18
fmt.Println(p2)
```

- 方式 3-&

案例: var person *Person = new (Person)

```go
//方式3-&
//案例: var person *Person = new (Person)

var p3 *Person= new(Person)
//因为p3是一个指针，因此标准的给字段赋值方式
//(*p3).Name = "smith" 也可以这样写 p3.Name = "smith"

//原因: go的设计者 为了程序员使用方便，底层会对 p3.Name = "smith" 进行处理
//会给 p3 加上 取值运算 (*p3).Name = "smith"
(*p3).Name = "smith"
p3.Name = "john" //

(*p3).Age = 30
p3.Age = 100
fmt.Println(*p3)
```

- 方式 4-{}

案例: var person *Person = &Person{}

```go
//方式4-{}
//案例: var person *Person = &Person{}

//下面的语句，也可以直接给字符赋值
//var person *Person = &Person{"mary", 60}
var person *Person = &Person{}

//因为person 是一个指针，因此标准的访问字段的方法
// (*person).Name = "scott"
// go的设计者为了程序员使用方便，也可以 person.Name = "scott"
// 原因和上面一样，底层会对 person.Name = "scott" 进行处理， 会加上 (*person)
(*person).Name = "scott"
person.Name = "scott~~"

(*person).Age = 88
person.Age = 10
fmt.Println(*person)
```

- 说明:

  1) 第 3 种和第 4 种方式返回的是 **结构体指针**。

  2) 结构体指针访问字段的标准方式应该是：(*结构体指针).字段名 ，比如 (*person).Name = "tom"

  3) 但 go 做了一个简化，**也支持 结构体指针.字段名**， 比如 person.Name = "tom"。更加符合程序员使用的习惯，**go 编译器底层 对 person.Name 做了转化 (*person).Name。**

10.1.13 struct 类型的内存分配机制

- 看一个思考题

我们定义一个Person结构体(包括 名字,年龄)。

我们看看下面一段代码,输出什么内容?

```
var p1 Person
p1.Age=10
p1.Name= "小明"
var p2 Person = p1

fmt.Println(p2.Age)
p2.Name = "tom"
fmt.Printf("p2.Name=%v p1.Name=%v", p2.Name, p1.Name)
```

输出的结果是: p2.Name = tom p1.Name = 小明

- 基本说明

变量总是存在内存中的,那么结构体变量在内存中究竟是怎样存在的?

这里我们给大家伙画一个图来说明一下结构体变量在内存中如何存在?

- 结构体在内存中示意图

➢ 看下面代码，并分析原因

```go
package main
import (
    "fmt"
)

type Person struct{
    Name string
    Age int
}
func main() {

    var p1 Person
    p1.Age=10
    p1.Name= "小明"
    var p2 *Person = &p1 //这里是关键-->画出示意图

    fmt.Println((*p2).Age)
    fmt.Println(p2.Age)
    p2.Name = "tom~"
    fmt.Printf("p2.Name=%v p1.Name=%v \n", p2.Name, p1.Name) // tom~ tom~
    fmt.Printf("p2.Name=%v p1.Name=%v \n", (*p2).Name, p1.Name) // tom~ tom~

    fmt.Printf("p1的地址%p\n", &p1)
    fmt.Printf("p2的地址%p p2的值%p\n", &p2, p2)

}
```

输出的结果是：

```
D:\goproject\src\go_code\chapter10\exercise>go run main.go
10
10
p2.Name=tom~ p1.Name=tom~
p2.Name=tom~ p1.Name=tom~
p1的地址0xc0420023e0
p2的地址0xc042004028 p2的值0xc0420023e0
```

上面代码对应的内存图的分析：

➢ 看下面代码，并分析原因

```
var p1 Person
p1.Age=10
p1.Name= "小明"
var p2 *Person = &p1

fmt.Println(*p2.Age)
//能不能这样写，不能这样写，会报错，原因是 .的运行符优先级比 * 高。
```

10.1.14 结构体使用注意事项和细节

1) 结构体的所有字段在内存中是连续的

```go
package main
import "fmt"

//结构体
type Point struct {
    x int
    y int
}

//结构体
type Rect struct {
    leftUp, rightDown Point
}

//结构体
type Rect2 struct {
    leftUp, rightDown *Point
}

func main() {

    r1 := Rect{Point{1,2}, Point{3,4}}

```

```go
    //r1有四个int，在内存中是连续分布
    //打印地址
    fmt.Printf("r1.leftUp.x 地址=%p r1.leftUp.y 地址=%p r1.rightDown.x 地址=%p r1.rightD
    &r1.leftUp.x, &r1.leftUp.y, &r1.rightDown.x, &r1.rightDown.y)

    //r2有两个 *Point类型，这个两个*Point类型的本身地址也是连续的，
    //但是他们指向的地址不一定是连续

    r2 := Rect2{&Point{10,20}, &Point{30,40}}

    //打印地址
    fmt.Printf("r2.leftUp 本身地址=%p r2.rightDown 本身地址=%p \n",
        &r2.leftUp, &r2.rightDown)

    //他们指向的地址不一定是连续...， 这个要看系统在运行时是如何分配
    fmt.Printf("r2.leftUp 指向地址=%p r2.rightDown 指向地址=%p \n",
        r2.leftUp, r2.rightDown)

}
```

对应的分析图:

2) 结构体是用户单独定义的类型，和其它类型进行转换时需要有完全相同的字段(名字、个数和类型)

```go
package main
D:\goproject\src\go_code\chapter10\exercise\main.go

type A struct {
    Num int
}
type B struct {
    Num int
}
func main() {
    var a A
    var b B
    a = A(b) // ? 可以转换，但是有要求，就是结构体的的字段要完全一样(包括:名字、个数和类型!)
    fmt.Println(a, b)
}
```

3) 结构体进行 type 重新定义(相当于取别名)，Golang 认为是新的数据类型，但是相互间可以强转

```go
type Student struct {
   Name string
   Age int
}

type Stu Student

func main() {
   var stu1 Student
   var stu2 Stu
   stu2 = stu1
  // 正确吗？错误，可以这样修改stu2 =Stu(stu1) //ok
   fmt.Println(stu1, stu2)
}
```

```go
type integer int

func main() {

   var i integer = 10
   var j int = 20
   j = i //正确吗？不可以！修改：j = int(i)
    fmt.Println(i, j)
}
```

4) struct 的每个字段上，可以写上一个 **tag**，该 tag 可以通过反射机制获取，常见的使用场景就是**序列化**和反序列化。

➢ 序列化的使用场景:

➢ 举例:

```go
type Monster struct{
    Name string `json:"name"` // `json:"name"` 就是 struct tag
    Age int `json:"age"`
    Skill string `json:"skill"`
}
```

```go
//1. 创建一个Monster变量
monster := Monster{"牛魔王", 500, "芭蕉扇~"}

//2. 将monster变量序列化为 json格式字串
//   json.Marshal 函数中使用反射，这个讲解反射时，我会详细介绍
jsonStr, err := json.Marshal(monster)
if err != nil {
    fmt.Println("json 处理错误 ", err)
}
fmt.Println("jsonStr", string(jsonStr))
```

10.2 方法

10.2.1 基本介绍

在某些情况下，我们要需要声明(定义)方法。比如 Person 结构体:除了有一些字段外(年龄，姓

名..),Person 结构体还有一些行为比如:可以说话、跑步..,通过学习，还可以做算术题。这时就要用方法才能完成。

Golang 中的方法是**作用在指定的数据类型上的**(即：和指定的数据类型绑定)，因此**自定义类型，都可以有方法**，而不仅仅是 **struct**。

10.2.2 方法的声明和调用

```
type A struct {
    Num int
}
func (a A) test() {
    fmt.Println(a.Num)
}
```

- 对上面的语法的说明

  1) func (**a A**) test()　{}　表示 A 结构体有一方法，方法名为 test

  2) (a A) 体现 test 方法是和 A 类型绑定的

- 举例说明

```go
package main

import (
    "fmt"
)

type Person struct{
    Name string
}

//给Person类型绑定一方法
func (p Person) test() {
    fmt.Println("test() name=", p.Name)
}

func main() {

    var p Person
    p.Name = "tom"
    p.test() //调用方法
}
```

➢ 对上面的总结

1) test 方法和 Person 类型绑定

2) test 方法只能通过 Person 类型的变量来调用，而不能直接调用，也不能使用其它类型变量来调用

```go
//下面的使用方式都是错误的
var dog Dog
dog.test()
test()
```

3) func (p Person) test() {}... p 表示哪个 Person 变量调用，这个 p 就是它的副本，这点和函数传参非常相似。

4) p 这个名字，有程序员指定，不是固定，比如修改成 person 也是可以

```go
//给Person类型绑定一方法
func (person Person) test() {
    person.Name = "jack"
    fmt.Println("test() name=", person.Name) // 输出jack
}
```

10.2.3 方法快速入门

1) 给 Person 结构体添加 speak 方法,输出 xxx 是一个好人

```
//给Person结构体添加speak 方法,输出  xxx是一个好人
func (p Person) speak() {
    fmt.Println(p.Name, "是一个goodman~")
}
```

2) 给 Person 结构体添加 jisuan 方法,可以计算从 1+..+1000 的结果，说明方法体内可以函数一样，进行各种运算

```
//给Person结构体添加jisuan 方法,可以计算从 1+..+1000的结果,
//说明方法体内可以函数一样，进行各种运算

func (p Person) jisuan() {
    res := 0
    for i := 1; i <= 1000; i++ {
        res += i
    }
    fmt.Println(p.Name, "计算的结果是=", res)
}
```

3) 给 Person 结构体 jisuan2 方法,该方法可以接收一个数 n，计算从 1+..+n 的结果

```
//给Person结构体jisuan2 方法,该方法可以接收一个参数n，计算从 1+..+n 的结果
func (p Person) jisuan2(n int) {
    res := 0
    for i := 1; i <= n; i++ {
        res += i
    }
    fmt.Println(p.Name, "计算的结果是=", res)
}
```

4) 给 Person 结构体添加 getSum 方法,可以计算两个数的和，并返回结果

```go
//给Person结构体添加getSum方法,可以计算两个数的和，并返回结果
func (p Person) getSum(n1 int, n2 int) int {
    return n1 + n2
}
```

5) 方法的调用

```go
//调用方法
p.speak()
p.jisuan()
p.jisuan2(20)
res := p.getSum(10, 20)
fmt.Println("res=", res)
```

10.2.4 方法的调用和传参机制原理：(重要！)

➢ 说明：

方法的调用和传参机制和函数基本一样，不一样的地方是方法调用时，会将调用方法的变量，当做实参也传递给方法。下面我们举例说明。

➢ 案例 1：

画出前面 getSum 方法的执行过程+说明

说明:

1) 在通过一个变量去调用方法时，其调用机制和函数一样

2) 不一样的地方时，变量调用方法时，该变量本身也会作为一个参数传递到方法(如果变量是值类型，则进行值拷贝，如果变量是引用类型，则进行地质拷贝)

➢ 案例 2

请编写一个程序，要求如下：

1) 声明一个结构体 Circle, 字段为 radius

2) 声明一个方法 area 和 Circle 绑定，可以返回面积。

3) 提示：画出 area 执行过程+说明

10.2.5 方法的声明(定义)

func (recevier type) methodName（参数列表） (返回值列表){

方法体

return 返回值

}

1) 参数列表：表示方法输入
2) recevier type : 表示这个方法和 type 这个类型进行绑定，或者说该方法作用于 type 类型
3) receiver type : type 可以是结构体，也可以其它的自定义类型
4) receiver : 就是 type 类型的一个变量(实例)，比如 ：Person 结构体 的一个变量(实例)
5) 返回值列表：表示返回的值，可以多个
6) 方法主体：表示为了**实现某一功能代码**块
7) return 语句不是必须的。

10.2.6 方法的注意事项和细节

1) 结构体类型是值类型，在方法调用中，遵守值类型的传递机制，是值拷贝传递方式
2) 如程序员希望在方法中，修改结构体变量的值，可以通过结构体指针的方式来处理

3) Golang 中的方法作用在指定的数据类型上的(即：和指定的数据类型绑定)，因此自定义类型，都可以有方法，而不仅仅是 struct， 比如 int , float32 等都可以有方法

```go
type integer int

func (i integer) print() {
    fmt.Println("i=", i)
}
//编写一个方法，可以改变i的值
func (i *integer) change() {
    *i = *i + 1
}

func main() {
    var i integer = 10
    i.print()
    i.change()
    fmt.Println("i=", i)

}
```

4) 方法的访问范围控制的规则，和函数一样。方法名首字母小写，只能在本包访问，方法首字母大写，可以在本包和其它包访问。[讲解]

5) 如果一个类型实现了 String()这个方法，那么 fmt.Println 默认会调用这个变量的 String()进行输出

```go
type Student struct {
    Name string
    Age int
}

//给*Student实现方法String()
func (stu *Student) String() string {
    str := fmt.Sprintf("Name=[%v] Age=[%v]", stu.Name, stu.Age)
    return str
}
```

```go
//定义一个Student变量
stu := Student{
    Name : "tom",
    Age : 20,
}
//如果你实现了 *Student 类型的 String方法，就会自动调用
fmt.Println(&stu)
```

10.2.7 方法的课堂练习题

1) 编写结构体(MethodUtils)，编程一个方法，方法不需要参数，在方法中打印一个 10*8 的矩形，在 main 方法中调用该方法。

```go
package main

import (
    "fmt"
)

type MethodUtils struct {
    //字段...
}

//给MethodUtils编写方法
func (mu MethodUtils) Print() {
    for i := 1; i <= 10; i++ {
        for j := 1; j <= 8; j++ {
            fmt.Print("*")
        }
        fmt.Println()
    }
}

func main() {
    /*
    1)编写结构体(MethodUtils)，编程一个方法，方法不需要参数，
    在方法中打印一个10*8 的矩形，在main方法中调用该方法。
    */
    var mu MethodUtils
    mu.Print()
}
```

2) 编写一个方法，提供 m 和 n 两个参数，方法中打印一个 m*n 的矩形

```go
//2)编写一个方法，提供m和n两个参数，方法中打印一个m*n的矩形
func (mu MethodUtils) Print2(m int, n int) {
    for i := 1; i <= m; i++ {
        for j := 1; j <= n; j++ {
            fmt.Print("*")
        }
        fmt.Println()
    }
}
```

3) 编写一个方法算该矩形的面积(可以接收长 len，和宽 width)， 将其作为方法返回值。在 main 方法中调用该方法，接收返回的面积值并打印。

```go
/*
编写一个方法算该矩形的面积(可以接收长len，和宽width)，
将其作为方法返回值。在main方法中调用该方法，接收返回的面积值并打印
*/

func (mu MethodUtils) area(len float64, width float64) (float64) {
    return len * width
}
```

4) 编写方法：判断一个数是奇数还是偶数

```go
/*
编写方法：判断一个数是奇数还是偶数

*/

func (mu *MethodUtils) JudgeNum(num int)  {
    if num % 2 == 0 {
        fmt.Println(num, "是偶数..")
    } else {
        fmt.Println(num, "是奇数..")
    }
}
```

5) 根据行、列、字符打印 对应行数和列数的字符，比如：行：3，列：2，字符*,则打印相应的效果

```
/*
根据行、列、字符打印 对应行数和列数的字符，
比如：行：3，列：2，字符*,则打印相应的效果

*/

func (mu *MethodUtils) Print3(n int, m int, key string)  {

    for i := 1; i <= n ; i++ {
        for j := 1; j <= m; j++ {
            fmt.Print(key)
        }
        fmt.Println()
    }
}
```

6) 定义小小计算器结构体(Calcuator)，实现加减乘除四个功能

实现形式 1：分四个方法完成:

实现形式 2：用一个方法搞定

```go
//实现形式1

type Calcuator struct{
    Num1 float64
    Num2 float64
}

func (calcuator *Calcuator) getSum() float64 {

    return calcuator.Num1 + calcuator.Num2
}

func (calcuator *Calcuator) getSub() float64 {

    return calcuator.Num1 - calcuator.Num2
}
```

```go
//实现形式2

func (calcuator *Calcuator) getRes(operator byte) float64 {
    res := 0.0
    switch operator {
    case '+':
        res = calcuator.Num1 + calcuator.Num2
    case '-':
        res = calcuator.Num1 - calcuator.Num2
    case '*':
        res = calcuator.Num1 * calcuator.Num2
    case '/':
        res = calcuator.Num1 / calcuator.Num2
    default:
        fmt.Println("运算符输入有误...")

    }
    return res
}
```

10.2.8 方法的课后练习题

1)在MethodUtils结构体编个方法,从键盘接收整数(1-9),打印对应乘法表:

```
1×1=1
1×2=2  2×2=4
1×3=3  2×3=6  3×3=9
1×4=4  2×4=8  3×4=12 4×4=16
1×5=5  2×5=10 3×5=15 4×5=20 5×5=25
1×6=6  2×6=12 3×6=18 4×6=24 5×6=30 6×6=36
1×7=7  2×7=14 3×7=21 4×7=28 5×7=35 6×7=42 7×7=49
1×8=8  2×8=16 3×8=24 4×8=32 5×8=40 6×8=48 7×8=56 8×8=64
1×9=9  2×9=18 3×9=27 4×9=36 5×9=45 6×9=54 7×9=63 8×9=72 9×9=81
请按任意键继续. . .
```

2)编写方法,使给定的一个二维数组(3×3)转置:

```
1 2 3          1 4 7
4 5 6   ==>    2 5 8
7 8 9          3 6 9
```

【上机练习】

强调: 一定自己要做，否则学习效果不好!!

10.2.9 方法和函数区别

1) 调用方式不一样

函数的调用方式: 函数名(实参列表)

方法的调用方式: 变量.方法名(实参列表)

2) 对于普通函数，接收者为值类型时，不能将指针类型的数据直接传递，反之亦然

```go
package main

import (
    "fmt"
)

type Person struct {
    Name string
}

//函数
//对于普通函数，接收者为值类型时，不能将指针类型的数据直接传递，反之亦然

func test01(p Person) {
    fmt.Println(p.Name)
}

func test02(p *Person) {
    fmt.Println(p.Name)
}
```

```go
func main() {

    p := Person{"tom"}
    test01(p)
    test02(&p)
```

3) 对于方法（如 struct 的方法），接收者为值类型时，可以直接用指针类型的变量调用方法，反过来同样也可以

```go
//对于方法（如struct的方法），
//接收者为值类型时，可以直接用指针类型的变量调用方法，反过来同样也可以

func (p Person) test03() {
    p.Name = "jack"
    fmt.Println("test03() =", p.Name) // jack
}

func (p *Person) test04() {
    p.Name = "mary"
    fmt.Println("test03() =", p.Name) // mary
}
```

```
p.test03()
fmt.Println("main() p.name=", p.Name) // tom

(&p).test03() // 从形式上是传入地址，但是本质仍然是值拷贝

fmt.Println("main() p.name=", p.Name) // tom


(&p).test04()
fmt.Println("main() p.name=", p.Name) // mary
p.test04() // 等价 (&p).test04 ，从形式上是传入值类型，但是本质仍然是地址拷贝
```

总结:

1) 不管调用形式如何，真正决定是值拷贝还是地址拷贝，看这个方法是和哪个类型绑定.

2) 如果是和值类型，比如 **(p Person)**，则是值拷贝， 如果和指针类型，比如是 **(p *Person)** 则是地址拷贝。

10.3 面向对象编程应用实例

10.3.1 步骤

1) 声明(定义)结构体，确定结构体名

2) 编写结构体的字段

3) 编写结构体的方法

10.3.2 学生案例：

1) 编写一个 Student 结构体，包含 name、gender、age、id、score 字段，分别为 string、string、int、int、float64 类型。

2) 结构体中声明一个 say 方法，返回 string 类型，方法返回信息中包含所有字段值。

3) 在 main 方法中，创建 Student 结构体实例(变量)，并访问 say 方法，并将调用结果打印输出。

4) 走代码

```
package main
```

```
import (
    "fmt"
)

/*
学生案例：
编写一个 Student 结构体，包含 name、gender、age、id、score 字段，分别为 string、string、int、int、
float64 类型。
结构体中声明一个 say 方法，返回 string 类型，方法返回信息中包含所有字段值。
在 main 方法中，创建 Student 结构体实例(变量)，并访问 say 方法，并将调用结果打印输出。

*/
type Student struct {
    name string
    gender string
    age int
    id int
    score float64
}

func (student *Student) say()   string {

    infoStr := fmt.Sprintf("student 的信息 name=[%v] gender=[%v], age=[%v] id=[%v] score=[%v]",
        student.name, student.gender, student.age, student.id, student.score)
```

```
    return infoStr
}



func main() {
    //测试
    //创建一个 Student 实例变量
    var stu = Student{
        name : "tom",
        gender : "male",
        age : 18,
        id : 1000,
        score : 99.98,
    }
    fmt.Println(stu.say())
}
```

10.3.3 小狗案例 [学员课后练习]

1) 编写一个 Dog 结构体，包含 name、age、weight 字段
2) 结构体中声明一个 say 方法，返回 string 类型，方法返回信息中包含所有字段值。
3) 在 main 方法中，创建 Dog 结构体实例(变量)，并访问 say 方法，将调用结果打印输出。

10.3.4 盒子案例

1) 编程创建一个 Box 结构体，在其中声明三个字段表示一个立方体的长、宽和高，长宽高要从终端获取

2) 声明一个方法获取立方体的体积。

3) 创建一个 Box 结构体变量，打印给定尺寸的立方体的体积

4) 走代码

```go
/*
1)编程创建一个Box结构体，在其中声明三个字段表示一个立方体的长、宽和高，长宽高要从终端获取
2)声明一个方法获取立方体的体积。
3)创建一个Box结构体变量，打印给定尺寸的立方体的体积
*/
type Box struct {
    len float64
    width float64
    height float64
}

//声明一个方法获取立方体的体积
func (box *Box) getVolumn() float64 {
    return box.len * box.width * box.height
}
```

```go
//测试代码
var box Box
box.len = 1.1
box.width = 2.0
box.height = 3.0
volumn := box.getVolumn()
fmt.Printf("体积为=%.2f", volumn)
```

10.3.5 景区门票案例

1) 一个景区根据游人的年龄收取不同价格的门票，比如年龄大于 18，收费 20 元，其它情况门票免费.

2) 请编写 Visitor 结构体，根据年龄段决定能够购买的门票价格并输出

3) 代码：

```go
// 景区门票案例

:\src\go_code\chapter10\structdetails2\main.go 收取不同价格的门票，比如年龄大于18，收费20元，其它情况门票免费.
// 请编写Visitor结构体，根据年龄段决定能够购买的门票价格并输出

type Visitor struct {
    Name string
    Age int
}

func (visitor *Visitor) showPrice() {
    if visitor.Age >= 90 || visitor.Age <=8 {
        fmt.Println("考虑到安全，就不要玩了")
        return
    }
    if visitor.Age > 18 {
        fmt.Printf("游客的名字为 %v 年龄为 %v 收费20元 \n", visitor.Name, visitor.Age)
    } else {
        fmt.Printf("游客的名字为 %v 年龄为 %v 免费 \n", visitor.Name, visitor.Age)
    }
}
```

```go
//测试 ——→ 这段代码是放在main函数中的..
var v Visitor
for {
    fmt.Println("请输入你的名字")
    fmt.Scanln(&v.Name)
    if v.Name == "n" {
        fmt.Println("退出程序....")
        break
    }
    fmt.Println("请输入你的年龄")
    fmt.Scanln(&v.Age)
    v.showPrice()

}
```

10.4 创建结构体变量时指定字段值

- 说明

  Golang 在创建结构体实例(变量)时，可以直接指定字段的值

- 方式 1

```go
package main

import (
    "fmt"
)
type Stu struct {
    Name string
    Age int
}
```

```go
//方式1
//在创建结构体变量时，就直接指定字段的值                    在main函数中。。。
var stu1 = Stu{"小明", 19} // stu1---> 结构体数据空间
stu2 := Stu{"小明~", 20}

//在创建结构体变量时，把字段名和字段值写在一起，这种写法，就不依赖字段的定义顺序.
var stu3 = Stu{
        Name :"jack",
        Age : 20,
    }
stu4 := Stu{
    Age : 30,
    Name : "mary",
}

fmt.Println(stu1, stu2, stu3, stu4)
```

➢ 方式 2

```go
//方式2，  返回结构体的指针类型(!!!)          在main函数中。。。
var stu5 *Stu = &Stu{"小王", 29}  // stu5--> 地址 ---》 结构体数据[xxxx,xxx]
stu6 := &Stu{"小王~", 39}

//在创建结构体指针变量时，把字段名和字段值写在一起，这种写法，就不依赖字段的定义顺序.
var stu7 = &Stu{
    Name : "小李",
    Age :49,
}
stu8 := &Stu{
    Age :59,
    Name : "小李~",
}
fmt.Println(*stu5, *stu6, *stu7, *stu8) //
```

10.5 工厂模式

10.5.1 说明

Golang 的结构体没有构造函数，通常可以使用工厂模式来解决这个问题。

10.5.2 看一个需求

一个结构体的声明是这样的:

package model

type Student struct {

  Name string...

}

因为这里的 Student 的首字母 S 是大写的，如果我们想在其它包创建 Student 的实例(比如 main 包)，引入 model 包后，就可以直接创建 Student 结构体的变量(实例)。**但是问题来了，如果首字母是小写的，比如 是 type student struct {....} 就不不行了，怎么办---> 工厂模式来解决.**

10.5.3 工厂模式来解决问题

➢ 使用工厂模式实现跨包创建结构体实例(变量)的案例:

如果 model 包的 **结构体变量首字母大写，引入后，直接使用，没有问题**

```go
package model

//定义一个结构体
type Student struct{
    Name string
    Score float64
}
```

```go
package main
import (
    "fmt"
    "go_code/chapter10/factory/model"
)

func main() {
    //创建要给Student实例
    var stu = model.Student{
        Name :"tom",
        Score : 78.9,
    }

    fmt.Println(stu)
}
```

如果 model 包的 结构体变量首字母小写，引入后，不能直接使用，可以**工厂模式解决**， 看老师演示， 代码:

student.go

```go
package model

//定义一个结构体
type student struct{
    Name string
    Score float64
}

//因为student结构体首字母是小写，因此是只能在model使用
//我们通过工厂模式来解决

func NewStudent(n string, s float64) *student {
    return &student{
        Name : n,
        Score : s,
    }
}
```

main.go

```go
package main
import (
    "fmt"
    "go_code/chapter10/factory/model"
)

func main() {
    //创建要给Student实例
    // var stu = model.Student{
    //  Name :"tom",
    //  Score : 78.9,
    // }

    //定student结构体是首字母小写，我们可以通过工厂模式来解决
    var stu = model.NewStudent("tom~", 88.8)

    fmt.Println(*stu) //&{....}
    fmt.Println("name=", stu.Name, " score=", stu.Score)
}
```

10.5.4 思考题

同学们思考一下，如果 model 包的 student 的结构体的字段 Score 改成 score，我们还能正常访问吗？又应该如何解决这个问题呢？[老师给出思路，学员自己完成]

➢ 解决方法如下：

```go
package model

//定义一个结构体
type student struct{
    Name string
    score float64
}

//因为student结构体首字母是小写，因此是只能在model使用
//我们通过工厂模式来解决

func NewStudent(n string, s float64) *student {
    return &student{
        Name : n,
        score : s,
    }
}

//如果score字段首字母小写，则，在其它包不可以直接方法，我们可以提供一个方法
func (s *student) GetScore() float64{
    return s.score //ok
}
```

```go
package main
import (
    "fmt"
    "go_code/chapter10/factory/model"
)

func main() {
    //创建要给Student实例
    // var stu = model.Student{
    //  Name :"tom",
    //  Score : 78.9,
    // }

    //定student结构体是首字母小写，我们可以通过工厂模式来解决
    var stu = model.NewStudent("tom~", 98.8)

    fmt.Println(*stu) //&{....}
    fmt.Println("name=", stu.Name, " score=", stu.GetScore())
}
```

第 11 章 面向对象编程(下)

11.1 VSCode 的使用

11.1.1 VSCode 使用技巧和经验

➢ 设置字体

文件->首选项->设置

➢ 快捷键的使用

自定义快捷配置：文件->首选项->键盘快捷方式

➢ 介绍几个常用的快捷键

1) 删除当前行 ctrl+shift+k [也可以自定义]
2) 向上/向下复制当前行 Shift+Alt + ↓ / ↑
3) 补全代码 alt + /
4) 添加注释和取消注释 ctrl + /
5) 快速修复 alt + /
6) 快速格式化代码 shift + alt + f
7) 还有很多其它的快捷键... 【参考 VScode 快捷键大全.doc】

特别说明:
1) VSCode的快捷键不要和输入法冲突，否则不会生效
2) 一些快捷键需要安装Go插件后才能生效

11.2 面向对象编程思想-抽象

11.2.1 抽象的介绍

我们在前面去定义一个结构体时候，实际上就是把一类事物的共有的**属性(字段)**和**行为(方法)**提取出来，形成一个**物理模型**(结构体)。这种研究问题的方法称为抽象。

11.2.2 代码实现

```
package main
```

```
import (
    "fmt"
)
//定义一个结构体 Account
type Account struct {
    AccountNo string
    Pwd string
    Balance float64
}

//方法
//1. 存款
func (account *Account) Deposite(money float64, pwd string)    {

    //看下输入的密码是否正确
    if pwd != account.Pwd {
        fmt.Println("你输入的密码不正确")
        return
    }

    //看看存款金额是否正确
    if money <= 0 {
        fmt.Println("你输入的金额不正确")
        return
    }
```

```
    account.Balance += money
    fmt.Println("存款成功~~")

}

//取款
func (account *Account) WithDraw(money float64, pwd string)    {

    //看下输入的密码是否正确
    if pwd != account.Pwd {
        fmt.Println("你输入的密码不正确")
        return
    }

    //看看取款金额是否正确
    if money <= 0    || money > account.Balance {
        fmt.Println("你输入的金额不正确")
        return
    }

    account.Balance -= money
    fmt.Println("取款成功~~")

}

//查询余额
```

```
func (account *Account) Query(pwd string)    {

    //看下输入的密码是否正确
    if pwd != account.Pwd {
        fmt.Println("你输入的密码不正确")
        return
    }

    fmt.Printf("你的账号为=%v  余额=%v \n", account.AccountNo, account.Balance)

}


func main() {

    //测试一把
    account := Account{
        AccountNo : "gs1111111",
        Pwd : "666666",
        Balance : 100.0,
    }

    //这里可以做的更加灵活，就是让用户通过控制台来输入命令...
    //菜单....
    account.Query("666666")
    account.Deposite(200.0, "666666")
```

```
        account.Query("666666")
        account.WithDraw(150.0, "666666")
        account.Query("666666")
    }
```

➢ 对上面代码的要求

1) 同学们自己可以独立完成
2) 增加一个控制台的菜单，可以让用户动态的输入命令和选项

11.3 面向对象编程三大特性-封装

11.3.1 基本介绍

Golang 仍然有面向对象编程的继承，封装和多态的特性，只是实现的方式和其它 OOP 语言不一样，下面我们一一为同学们进行详细的讲解 Golang 的三大特性是如何实现的。

11.3.2 封装介绍

封装(encapsulation)就是把抽象出的字段和对字段的操作封装在一起,数据被保护在内部,程序的其它包只有通过被授权的操作(方法),才能对字段进行操作

对电视机的操作就是典型封装

11.3.3 封装的理解和好处

1) 隐藏实现细节
2) 提可以对**数据进行验证**，保证安全合理(Age)

11.3.4 如何体现封装

1) 对结构体中的属性进行封装
2) 通过**方法**，**包** 实现封装

11.3.5 封装的实现步骤

1) 将结构体、字段(属性)的首字母小写(不能导出了，其它包不能使用，类似 private)
2) 给结构体所在包提供一个工厂模式的函数，首字母大写。类似一个构造函数
3) 提供一个首字母大写的 Set 方法(类似其它语言的 public)，用于对属性判断并赋值

```
func (var   结构体类型名) SetXxx(参数列表) (返回值列表) {
    //加入数据验证的业务逻辑
    var.字段 = 参数
}
```

4) 提供一个首字母大写的 Get 方法(类似其它语言的 public)，用于获取属性的值

```
func (var  结构体类型名) GetXxx() {
    return var.age;
}
```

特别说明：在 Golang 开发中并没有特别强调封装，这点并不像 Java. 所以提醒学过 java 的朋友，不用总是用 java 的语法特性来看待 Golang, Golang 本身对面向对象的特性做了简化的.

11.3.6 快速入门案例

➢ 看一个案例

请大家看一个程序(person.go),不能随便查看**人的年龄,工资**等隐私，并对输入的年龄进行合理的验证。设计: model 包(person.go) main 包(main.go 调用 Person 结构体)

➢ 代码实现

model/person.go

```go
package model
import "fmt"

type person struct {
    Name string
    age int   //其它包不能直接访问..
    sal float64
}

//写一个工厂模式的函数，相当于构造函数
func NewPerson(name string) *person {
    return &person{
        Name : name,
    }
}

//为了访问age 和 sal 我们编写一对SetXxx的方法和GetXxx的方法
func (p *person) SetAge(age int) {
    if age >0 && age <150 {
        p.age = age
    } else {
        fmt.Println("年龄范围不正确..")
        //给程序员给一个默认值
```

```go
            //给程序员给一个默认值
        }
    }

    func (p *person) GetAge() int {
        return p.age
    }


    func (p *person) SetSal(sal float64) {
        if sal >= 3000 && sal <= 30000 {
            p.sal = sal
        } else {
            fmt.Println("薪水范围不正确..")

        }
    }

    func (p *person) GetSal() float64 {
        return p.sal
    }
```

main/main.go

```go
package main
import (
    "fmt"
    "go_code/chapter11/encapsulate/model"
)

func main() {

    p := model.NewPerson("smith")
    p.SetAge(18)
    p.SetSal(5000)
    fmt.Println(p)
    fmt.Println(p.Name, " age =", p.GetAge(), " sal = ", p.GetSal())

}
```

11.3.7 课堂练习(学员先做)

➢ 要求

1) 创建程序,在 model 包中定义 Account 结构体：在 main 函数中体会 Golang 的封装性。
2) Account 结构体要求具有字段：账号（长度在 6-10 之间）、余额(必须>20)、密码（必须是六
3) 通过 SetXxx 的方法给 Account 的字段赋值。(同学们自己完成
4) 在 main 函数中测试

➢ 代码实现

model/account.go

```
package model

import (
    "fmt"
)
//定义一个结构体 account
type account struct {
    accountNo string
    pwd string
    balance float64
}

//工厂模式的函数-构造函数
func NewAccount(accountNo string, pwd string, balance float64) *account {

    if len(accountNo) < 6 || len(accountNo) > 10 {
```

```
        fmt.Println("账号的长度不对...")
        return nil
    }

    if len(pwd) != 6 {
        fmt.Println("密码的长度不对...")
        return nil
    }

    if balance < 20 {
        fmt.Println("余额数目不对...")
        return nil
    }

    return &account{
        accountNo : accountNo,
        pwd : pwd,
        balance : balance,
    }

}

//方法
//1. 存款
func (account *account) Deposite(money float64, pwd string)    {
```

```
    //看下输入的密码是否正确
    if pwd != account.pwd {
        fmt.Println("你输入的密码不正确")
        return
    }

    //看看存款金额是否正确
    if money <= 0 {
        fmt.Println("你输入的金额不正确")
        return
    }

    account.balance += money
    fmt.Println("存款成功~~")

}

//取款
func (account *account) WithDraw(money float64, pwd string)    {

    //看下输入的密码是否正确
    if pwd != account.pwd {
        fmt.Println("你输入的密码不正确")
        return
    }
```

```
    //看看取款金额是否正确
    if money <= 0   || money > account.balance {
        fmt.Println("你输入的金额不正确")
        return
    }

    account.balance -= money
    fmt.Println("取款成功~~")

}

//查询余额
func (account *account) Query(pwd string)   {

    //看下输入的密码是否正确
    if pwd != account.pwd {
        fmt.Println("你输入的密码不正确")
        return
    }

    fmt.Printf("你的账号为=%v  余额=%v \n", account.accountNo, account.balance)

}
```

main/main.go

```
package main
```

```
import (
    "fmt"
    "go_code/chapter11/encapexercise/model"
)

func main() {
    //创建一个 account 变量
    account := model.NewAccount("jzh11111", "000", 40)
    if account != nil {
        fmt.Println("创建成功=", account)
    } else {
        fmt.Println("创建失败")
    }
}
```

➢ 说明：在老师的代码基础上**增加如下功能**：

通过 SetXxx 的方法给 Account 的字段赋值 通过 GetXxx 方法获取字段的值。(同学们自己完成)

在 main 函数中测试

11.4 面向对象编程三大特性-继承

11.4.1 看一个问题，引出继承的必要性

一个小问题,看个学生考试系统的程序 extends01.go，提出代码复用的问题

➢ 走一下代码

```
package main

import (
    "fmt"
)

//编写一个学生考试系统

//小学生
type Pupil struct {
    Name string
    Age int
    Score int
}

//显示他的成绩
```

```
func (p *Pupil) ShowInfo() {
    fmt.Printf("学生名=%v  年龄=%v  成绩=%v\n", p.Name, p.Age, p.Score)
}

func (p *Pupil) SetScore(score int) {
    //业务判断
    p.Score = score
}

func (p *Pupil) testing() {
    fmt.Println("小学生正在考试中.....")
}

//大学生，研究生。。



//大学生
type Graduate struct {
    Name string
    Age int
    Score int
}

//显示他的成绩
func (p *Graduate) ShowInfo() {
    fmt.Printf("学生名=%v  年龄=%v  成绩=%v\n", p.Name, p.Age, p.Score)
```

```
}

func (p *Graduate) SetScore(score int) {
    //业务判断
    p.Score = score
}

func (p *Graduate) testing() {
    fmt.Println("大学生正在考试中.....")
}

//代码冗余..  高中生....


func main() {

    //测试
    var pupil = &Pupil{
        Name :"tom",
        Age : 10,
    }
    pupil.testing()
    pupil.SetScore(90)
    pupil.ShowInfo()

    //测试
```

```
    var graduate = &Graduate{
        Name :"mary",
        Age : 20,
    }
    graduate.testing()
    graduate.SetScore(90)
    graduate.ShowInfo()
}
```

➢ 对上面代码的小结

1) Pupil 和 Graduate 两个结构体的字段和方法几乎，但是我们却写了相同的代码， 代码复用性不强

2) 出现代码冗余，而且代码**不利于维护**，同时**也不利于功能的扩展**。

3) 解决方法-通过**继承**方式来解决

11.4.2 继承基本介绍和示意图

继承可以解决代码复用,让我们的编程更加靠近人类思维。

当多个结构体存在相同的属性(字段)和方法时,可以从这些结构体中抽象出结构体(比如刚才的 Student),在该结构体中定义这些相同的属性和方法。

其它的结构体不需要重新定义这些属性(字段)和方法，只需嵌套一个 Student 匿名结构体即可。 **[画出示意图]**

也就是说：在 Golang 中，如果一个 struct 嵌套了另一个匿名结构体，那么这个结构体可以直接访问匿名结构体的字段和方法，从而实现了继承特性。

11.4.3 嵌套匿名结构体的基本语法

```
type Goods struct {
    Name string
    Price int
}
type Book struct {
    Goods   //这里就是嵌套匿名结构体 Goods
    Writer string
}
```

11.4.4 快速入门案例

➢ 案例

我们对 extends01.go 改进，使用嵌套匿名结构体的方式来实现继承特性,请大家注意体会这样编程

的好处

➢ 代码实现

```
package main

import (
    "fmt"
)

//编写一个学生考试系统

type Student struct {
    Name string
    Age int
    Score int
}

//将 Pupil 和 Graduate 共有的方法也绑定到 *Student
func (stu *Student) ShowInfo() {
```

```
    fmt.Printf("学生名=%v  年龄=%v  成绩=%v\n", stu.Name, stu.Age, stu.Score)
}
func (stu *Student) SetScore(score int) {
    //业务判断
    stu.Score = score
}

//小学生
type Pupil struct {
    Student //嵌入了 Student 匿名结构体
}

//显示他的成绩

//这时 Pupil 结构体特有的方法，保留
func (p *Pupil) testing() {
    fmt.Println("小学生正在考试中.....")
}

//大学生，研究生。。



//大学生
type Graduate struct {
    Student //嵌入了 Student 匿名结构体
}
```

```
//显示他的成绩
//这时 Graduate 结构体特有的方法，保留
func (p *Graduate) testing() {
    fmt.Println("大学生正在考试中.....")
}

//代码冗余.. 高中生....

func main() {

    //当我们对结构体嵌入了匿名结构体使用方法会发生变化
    pupil := &Pupil{}
    pupil.Student.Name = "tom~"
    pupil.Student.Age = 8
    pupil.testing()
    pupil.Student.SetScore(70)
    pupil.Student.ShowInfo()


    graduate := &Graduate{}
    graduate.Student.Name = "mary~"
    graduate.Student.Age = 28
    graduate.testing()
    graduate.Student.SetScore(90)
    graduate.Student.ShowInfo()
```

```
}
```

11.4.5 继承给编程带来的便利

1) 代码的复用性提高了
2) 代码的扩展性和维护性提高了

11.4.6 继承的深入讨论

1) 结构体可以**使用嵌套匿名结构体所有的字段和方法**，即：首字母大写或者小写的字段、方法，都可以使用。【举例说明】

```go
package main

import (
    "fmt"
)

type A struct {
    Name string
    age int
}

func (a *A) SayOk() {
    fmt.Println("A SayOk", a.Name)
}

func (a *A) hello() {
    fmt.Println("A hello", a.Name)
}

type B struct {
    A
}

func main() {

    var b B
    b.A.Name = "tom"
    b.A.age = 19
    b.A.SayOk()
    b.A.hello()
}
```

2) 匿名结构体字段访问可以简化，如图

```go
func main() {

    var b B
    b.A.Name = "tom"
    b.A.age = 19
    b.A.SayOk()
    b.A.hello()

    //上面的写法可以简化

    b.Name = "smith"
    b.age = 20
    b.SayOk()
    b.hello()

}
```

对上面的代码小结

(1) **当我们直接通过 b 访问字段或方法时，其执行流程如下**比如 b.Name

(2) 编译器会先看 b 对应的类型有没有 Name, 如果有，则直接调用 B 类型的 Name 字段

(3) 如果没有就去看B 中嵌入的匿名结构体 A 有没有声明 Name 字段,如果有就调用,如果没有继续查找..如果都找不到就报错.

3) 当**结构体**和**匿名结构体**有相同的字段或者方法时，**编译器采用就近访问原则访问**，如希望访问匿名结构体的字段和方法，可以通过匿名结构体名来区分【举例说明】

```go
func main() {

    var b B
    b.name = "tom" //这时就近原则，会访问B结构体的name字段
    //b.A.name 就明确指定访问A匿名结构体的字段name
    b.A.name = "jack"
    b.Age = 78
    b.say()  // 这时就近原则，会访问B结构体的say函数
    b.Hello()
    //b.A.Hello() 就明确指定访问A匿名结构体的方法Hello()
    b.A.Hello()
}
```

4) 结构体嵌入两个(或多个)匿名结构体，如两个匿名结构体有相同的字段和方法(**同时结构体本身没有同名的字段和方法**)，在访问时，就必须明确指定匿名结构体名字，否则编译报错。【举例说明】

```go
type A struct {
    Name string
    age int
}
type B struct {
    Name string
    Score float64
}
type C struct {
    A
    B
    //Name string
}
func main() {
    var c C
    //如果c 没有Name字段，而A 和 B有Name，这时就必须通过指定匿名结构体名字来区分
    //所以 c.Name 就会包编译错误， 这个规则对方法也是一样的！
    c.A.Name = "tom" // error
    fmt.Println("c")
}
```

5) 如果一个struct 嵌套了一个有名结构体，这种模式就是组合，如果是组合关系，那么在访问组合的结构体的字段或方法时，必须带上结构体的名字

```go
type D struct {
    a A //有名结构体
}
```

组合关系

```go
//如果D 中是一个有名结构体，则访问有名结构体的字段时，就必须带上有名结构体的名字
//比如 d.a.Name
var d D
d.a.Name = "jack"
```

6) 嵌套匿名结构体后，也可以在创建结构体变量(实例)时，直接指定各个**匿名结构体字段的**值

```go
type Goods struct {
    Name string
    Price float64
}

type Brand struct {
    Name string
    Address string
}

type TV struct {
    Goods
    Brand
}

type TV2 struct {
    *Goods
    *Brand
}
```

```go
    //嵌套匿名结构体后，也可以在创建结构体变量(实例)时，直接指定各个匿名结构体字段的值
    tv := TV{ Goods{"电视机001", 5000.99},  Brand{"海尔", "山东"}, }

    tv2 := TV{
        Goods{
            Price : 5000.99,
            Name : "电视机002",
        },
        Brand{
            Name : "夏普",
            Address :"北京",
        },
    }

    fmt.Println("tv", tv)
    fmt.Println("tv2", tv2)

    tv3 := TV2{ &Goods{"电视机003", 7000.99},  &Brand{"创维", "河南"}, }

    tv4 := TV2{
            &Goods{
                Name : "电视机004",
                Price : 9000.99,
            },
            &Brand{
```

```go
                Name : "长虹",
                Address : "四川",
            },
        }

    fmt.Println("tv3", *tv3.Goods, *tv3.Brand)
    fmt.Println("tv4", *tv4.Goods, *tv4.Brand)

}
```

11.4.7 课堂练习

结构体的匿名字段是基本数据类型，如何访问，下面代码输出什么

```
type Monster struct  {
    Name string
    Age int
}

type E struct {
    Monster
    int        匿名字段时基本数据类型
    n int
}
```

```
//演示一下匿名字段时基本数据类型的使用
var e E
e.Name = "狐狸精"
e.Age = 300
e.int = 20
e.n = 40
fmt.Println("e=", e)
```

说明

1) 如果一个结构体有 int 类型的匿名字段，就不能第二个。

2) 如果需要有多个 int 的字段，则必须给 int 字段指定名字

11.4.8 面向对象编程-多重继承

➢ 多重继承说明

如**一个 struct 嵌套了多个匿名结构体**，那么该结构体可以直接访问嵌套的匿名结构体的字段和方法，**从而实现了多重继承**。

➢ 案例演示

通过一个案例来说明多重继承使用

```go
type Goods struct {
    Name string
    Price float64
}

type Brand struct {
    Name string
    Address string
}

type TV struct {
    Goods
    Brand
}
```

➢ 多重继承细节说明

1) 如嵌入的匿名结构体有相同的字段名或者方法名，则在访问时，需要通过匿名结构体类型名来区分。【案例演示】

```go
//演示访问Goods的Name
fmt.Println(tv.Goods.Name)
fmt.Println(tv.Price)
```

2) 为了保证代码的简洁性，建议大家尽量不使用多重继承

11.5 接口(interface)

11.5.1 基本介绍

按顺序,我们应该讲解多态,但是在讲解多态前,我们需要讲解接口(interface),因为在 Golang 中 多态特性主要是通过接口来体现的。

11.5.2 为什么有接口

请大家先看一张图:

usb插槽就是现实中的接口。
你可以把手机, 相机, u盘都插在usb插槽上, 而不用担心那个插槽是专门插哪个的, 原因是做usb插槽的厂家和做各种设备的厂家都遵守了统一的规定包括尺寸，排线等等。

11.5.3 接口快速入门

这样的设计需求在 Golang 编程中也是会大量存在的,我曾经说过,一个程序就是一个世界,在现实世界存在的情况，在程序中也会出现。**我们用程序来模拟一下前面的应用场景**。

➢ 代码实现

```
package main
import (
    "fmt"
)

//声明/定义一个接口
type Usb interface {
    //声明了两个没有实现的方法
    Start()
    Stop()
}
```

```
type Phone struct {

}

//让 Phone  实现  Usb 接口的方法
func (p Phone) Start() {
    fmt.Println("手机开始工作。。。")
}
func (p Phone) Stop() {
    fmt.Println("手机停止工作。。。")
}

type Camera struct {

}
//让 Camera  实现     Usb 接口的方法
func (c Camera) Start() {
    fmt.Println("相机开始工作。。。")
}
func (c Camera) Stop() {
    fmt.Println("相机停止工作。。。")
}


//计算机
```

```
type Computer struct {

}

//编写一个方法 Working 方法，接收一个 Usb 接口类型变量
//只要是实现了 Usb 接口 （所谓实现 Usb 接口，就是指实现了 Usb 接口声明所有方法）
func (c Computer) Working(usb Usb) { //usb 变量会根据传入的实参，来判断到底是 Phone,还是 Camera

    //通过 usb 接口变量来调用 Start 和 Stop 方法
    usb.Start()
    usb.Stop()
}

func main() {

    //测试
    //先创建结构体变量
    computer := Computer{}
    phone := Phone{}
    camera := Camera{}

    //关键点
    computer.Working(phone)
    computer.Working(camera) //
}
```

说明: 上面的代码就是一个接口编程的快速入门案例。

11.5.4 接口概念的再说明

interface 类型可以定义一组方法，但是这些不需要实现。并且 interface 不能包含任何变量。到某个自定义类型(比如结构体 Phone)要使用的时候,在根据具体情况把这些方法写出来(实现)。

11.5.5 基本语法

➢ 小结说明:

1) 接口里的**所有方法都没有方法体**，即接口的方法都是没有实现的方法。接口体现了程序设计的多态和高内聚低偶合的思想。

2) Golang 中的接口，不需要**显式的实现**。只要一个变量，含有接口类型中的所有方法，那么这个变量就实现这个接口。因此，Golang 中**没有 implement** 这样的关键字

11.5.6 接口使用的应用场景

对初学者讲，理解接口的概念不算太难，难的是不知道什么时候使用接口，下面我例举几个应用场景：

1. 说现在美国要制造轰炸机，武装直升机. 专家只需把飞机需要的功能/规格定下来即可，然后让别的人具体实现就可。

2. 说现在有一个项目经理，管理三个程序员，开发一个软件，为了控制和管理软件，项目经理可以定义一些接口，然后由程序员具体实现。

11.5.7 注意事项和细节

1) 接口本身**不能创建实例**,但是**可以指向一个实现了该接口的自定义类型的变量**(实例)

```go
package main
import (
    "fmt"
)

type AInterface interface {
    Say()
}

type Stu struct {
    Name string
}

func (stu Stu) Say() {
    fmt.Println("Stu Say()")
}

func main() {
    var stu Stu //结构体变量，实现了 Say() 实现了 AInterface
    var a AInterface = stu
    a.Say()
}
```

2) 接口中所有的方法都没有方法体,即都是没有实现的方法。

3) 在 Golang 中，一个自定义类型需要将某个接口的所有方法都实现，我们说这个自定义类型实现了该接口。

4) 一个自定义类型只有实现了某个接口，才能将该自定义类型的实例(变量)赋给接口类型

5) 只要是自定义数据类型，就可以实现接口，不仅仅是结构体类型。

```go
type integer int

func (i integer) Say() {
    fmt.Println("integer Say i =" ,i )
}
```

```go
var i integer = 10
var b AInterface = i
b.Say() // integer Say i = 10
```

6) 一个自定义类型可以实现多个接口

```go
type AInterface interface {
    Say()
}

type BInterface interface {
    Hello()
}
type Monster struct {

}
func (m Monster) Hello() {
    fmt.Println("Monster Hello()~~")
}

func (m Monster) Say() {
    fmt.Println("Monster Say()~~")
}
```

```go
//Monster实现了AInterface 和 BInterface
var monster Monster
var a2 AInterface = monster
var b2 BInterface = monster
a2.Say()
b2.Hello()
```

7) Golang 接口中不能有任何变量

```go
type AInterface interface {
    Name string
    Test01()
    Test02()
}
```

8) 一个接口(比如 A 接口)可以继承多个别的接口(比如 B,C 接口)，这时如果要实现 A 接口，也必须将 B,C 接口的方法也全部实现。

```go
package main
import (
    _ "fmt"
)

type BInterface interface {
    test01()
}

type CInterface interface {
    test02()
}

type AInterface interface {
    BInterface
    CInterface
    test03()
}

//如果需要实现AInterface,就需要将BInterface CInterface的方法都实现
type Stu struct {
}
func (stu Stu) test01() {

}
func (stu Stu) test02() {
```

```go

}
func (stu Stu) test03() {

}

func main() {
    var stu Stu
    var a AInterface = stu
    a.test01()
}
```

9) interface 类型默认是一个指针(引用类型)，如果没有对 interface 初始化就使用，那么会输出 nil

10) 空接口 interface{} 没有任何方法，*所以所有类型都实现了空接口*，即我们可以**把任何一个变量赋给空接口**。

```go
type T  interface{

}
```

空接口

```go
var t T = stu //ok
fmt.Println(t)
var t2 interface{} = stu
var num1 float64 = 8.8
t2 = num1
t = num1
fmt.Println(t2, t)
```

11.5.8 课堂练习

```go
//下面代码，有没有错误，你能得出什么结论?
type AInterface interface{
    Test01()
    Test02()
}
type BInterface interface{
    Test01()
    Test03()
}
type Stu struct{
}
func (stu Stu) Test01(){
}
func (stu Stu) Test02(){
}
func (stu Stu) Test03(){
}
func main(){
    stu := Stu{}
    var a AInterface = stu
    var b BInterface = stu
    fmt.Println("ok~", a, b)
}
```

ok

```go
// 下面代码，有没有错误，你能得出什么结论?
type AInterface interface {
   Test01()
   Test02()
}
type BInterface interface {
   Test01()
   Test03()
}
type CInterface interface {
   AInterface
   BInterface
}
func main() {

}
//这里编译错误，因为CInterface有两个Test01()，
编译器不能通过!报告重复定义
```

```go
package main
import "fmt"
type Usb interface {
    Say()
}
type Stu struct {
}
func (this *Stu) Say() {
    fmt.Println("Say()")
}
func main() {
    var stu Stu = Stu{}
    // 错误！ 会报 Stu类型没有实现Usb接口 ，
    // 如果希望通过编译， var u Usb = &stu
    var u Usb = stu
    u.Say()
    fmt.Println("here", u)
}
```

错误的写法，应该改成:
var u Usb = &stu

11.5.9 接口编程的最佳实践

➢ 实现对 Hero 结构体切片的排序: sort.Sort(data Interface)

```
package main
```

```
import (
	"fmt"
	"sort"
	"math/rand"
)

//1.声明 Hero 结构体
type	Hero struct{
	Name string
	Age int
}

//2.声明一个 Hero 结构体切片类型
type HeroSlice []Hero

//3.实现 Interface  接口
func (hs HeroSlice) Len() int {
	return len(hs)
}

//Less 方法就是决定你使用什么标准进行排序
//1.  按 Hero 的年龄从小到大排序!!
func (hs HeroSlice) Less(i, j int) bool {
	return hs[i].Age < hs[j].Age
	//修改成对 Name 排序
	//return hs[i].Name < hs[j].Name
```

```
}

func (hs HeroSlice) Swap(i, j int) {
    //交换
    // temp := hs[i]
    // hs[i] = hs[j]
    // hs[j] = temp
    //下面的一句话等价于三句话
    hs[i], hs[j] = hs[j], hs[i]
}


//1.声明 Student 结构体
type   Student struct{
    Name string
    Age int
    Score float64
}

//将 Student 的切片，安 Score 从大到小排序!!

func main() {

    //先定义一个数组/切片
    var intSlice = []int{0, -1, 10, 7, 90}
    //要求对  intSlice 切片进行排序
```

```
//1. 冒泡排序...
//2. 也可以使用系统提供的方法
sort.Ints(intSlice)
fmt.Println(intSlice)

//请大家对结构体切片进行排序
//1. 冒泡排序...
//2. 也可以使用系统提供的方法

//测试看看我们是否可以对结构体切片进行排序
var heroes HeroSlice
for i := 0; i < 10 ; i++ {
    hero := Hero{
        Name : fmt.Sprintf("英雄|%d", rand.Intn(100)),
        Age : rand.Intn(100),
    }
    //将 hero append 到 heroes 切片
    heroes = append(heroes, hero)
}

//看看排序前的顺序
for _ , v := range heroes {
    fmt.Println(v)
}

//调用 sort.Sort
```

```
    sort.Sort(heroes)
    fmt.Println("-----------排序后------------")
    //看看排序后的顺序
    for _ , v := range heroes {
        fmt.Println(v)
    }

    i := 10
    j := 20
    i, j = j, i
    fmt.Println("i=", i, "j=", j) // i=20 j = 10
}
```

- 接口编程的课后练习

```
//1.声明 Student 结构体
type Student struct{
   Name string
   Age int
   Score float64
}
//将 Student 的切片，安 Score 从大到小排序!!
```

11.5.10 实现接口 vs 继承

- 大家听到现在,可能会对实现接口和继承比较迷茫了，这个问题,那么他们究竟有什么区别呢

代码说明:

```go
package main
import (
    "fmt"
)

//Monkey结构体
type Monkey struct {
    Name string
}

//声明接口
type BirdAble interface {
    Flying()
}

type FishAble interface {
    Swimming()
}

func (this *Monkey) climbing() {
    fmt.Println(this.Name, " 生来会爬树..")
}

```

```go
//LittleMonkey结构体
type LittleMonkey struct {
    Monkey //继承
}


//让LittleMonkey实现BirdAble
func (this *LittleMonkey) Flying() {
    fmt.Println(this.Name, " 通过学习，会飞翔...")
}

//让LittleMonkey实现FishAble
func (this *LittleMonkey) Swimming() {
    fmt.Println(this.Name, " 通过学习，会游泳..")
}

func main() {

    //创建一个LittleMonkey 实例
    monkey := LittleMonkey{
        Monkey {
            Name : "悟空",
        },
```

```go
    }
    monkey.climbing()
    monkey.Flying()
    monkey.Swimming()

}
```

➢ 对上面代码的小结

1) 当 A 结构体继承了 B 结构体，那么 A 结构就自动的继承了 B 结构体的字段和方法，并且可以直接使用

2) 当 A 结构体需要扩展功能，同时不希望去破坏继承关系，则可以去实现某个接口即可，因此我们可以认为：实现接口是对继承机制的补充.

➢ 实现接口可以看作是对 继承的一种补充

- **接口和继承解决的解决的问题**不同

继承的价值主要在于：解决代码的**复用性**和**可维护性**。

接口的价值主要在于：**设计**，设计好各种规范(方法)，让其它自定义类型去实现这些方法。

- 接口比继承更加灵活　Person　Student　　BirdAble LittleMonkey

接口比继承更加灵活，继承是满足 is - a 的关系，而接口只需满足 like - a 的关系。

- 接口在一定程度上实现**代码**解耦

11.6 面向对象编程-多态

11.6.1 基本介绍

变量(实例)具有多种形态。面向对象的第三大特征，在 Go 语言，多态特征是通过接口实现的。可以按照统一的接口来调用不同的实现。这时接口变量就呈现不同的形态。

11.6.2 快速入门

在前面的 Usb 接口案例，Usb usb ，既可以接收手机变量，又可以接收相机变量，就体现了 Usb 接口 多态特性。[点明]

```
//编写一个方法 Working 方法，接收一个 Usb 接口类型变量
//只要是实现了 Usb 接口 （所谓实现 Usb 接口，就是指实现了 Usb 接口声明所有方法）
func (c Computer) Working(usb Usb) { //usb 变量会根据传入的实参，来判断到底是 Phone,还是 Camera

    //通过 usb 接口变量来调用 Start 和 Stop 方法
    usb.Start()
    usb.Stop()
}
```

usb 接口变量就体现出多态的特点

11.6.3 接口体现多态的两种形式

- 多态参数

在前面的 Usb 接口案例，Usb usb ，即可以接收手机变量，又可以接收相机变量，就体现了 Usb 接口 多态。

- **多态**数组

演示一个案例：给 Usb 数组中，存放 Phone 结构体 和 Camera 结构体变量

案例说明:

```
package main
import (
    "fmt"
)

//声明/定义一个接口
```

```
type Usb interface {
    //声明了两个没有实现的方法
    Start()
    Stop()
}

type Phone struct {
    name string
}

//让 Phone  实现  Usb 接口的方法
func (p Phone) Start() {
    fmt.Println("手机开始工作。。。")
}
func (p Phone) Stop() {
    fmt.Println("手机停止工作。。。")
}

type Camera struct {
    name string
}
//让 Camera  实现     Usb 接口的方法
func (c Camera) Start() {
    fmt.Println("相机开始工作。。。")
}
func (c Camera) Stop() {
```

```
    fmt.Println("相机停止工作。。。")
}

func main() {
    //定义一个 Usb 接口数组，可以存放 Phone 和 Camera 的结构体变量
    //这里就体现出多态数组
    var usbArr [3]Usb
    usbArr[0] = Phone{"vivo"}
    usbArr[1] = Phone{"小米"}
    usbArr[2] = Camera{"尼康"}


    fmt.Println(usbArr)

}
```

11.7 类型断言

11.7.1 由一个具体的需要，引出了类型断言.

看一段代码

```
type Point struct {
        x int
        y int
}
func main() {
        var a interface{}
        var point Point = Point{1, 2}
        a = point  //oK
        // 如何将 a 赋给一个Point变量?
        var b Point
        b = a // 可以吗=>error
        fmt.Println(b)
}
```

解决方法

```
func main() {
        var a interface{}
        var point Point = Point{1, 2}
        a = point
        var b Point
        b = a.(Point)//类型断言
        fmt.Println(b)
}
```

b=a.(Point) 就是类型断言，表示判断a是否指向Point类型的变量，如果是就转成Point类型并赋给b变量，否则报错。

需求：如何将一个接口变量，赋给自定义类型的变量. => 引出类型断言

11.7.2 基本介绍

类型断言，由于接口是一般类型，不知道具体类型，如果要转成具体类型，就需要使用类型断言，具体的如下:

```go
//类型断言的其它案例
var x interface{}
var b2 float32 = 1.1
x = b2  //空接口，可以接收任意类型
// x=>float32 [使用类型断言]
y := x.(float32)
fmt.Printf("y 的类型是 %T 值是=%v", y, y)
```

➢ 对上面代码的说明:

在进行类型断言时，如果类型不匹配，就会报 panic, 因此进行类型断言时，要确保原来的空接口指向的就是断言的类型.

➢ 如何在进行断言时，带上检测机制，如果成功就 ok,否则也不要报 panic

```go
//类型断言(带检测的)
var x interface{}
var b2 float32 = 2.1
x = b2  //空接口，可以接收任意类型
// x=>float32 [使用类型断言]

//类型断言(带检测的)
if y, ok := x.(float32); ok {
    fmt.Println("convert success")
    fmt.Printf("y 的类型是 %T 值是=%v", y, y)
} else {
    fmt.Println("convert fail")
}
fmt.Println("继续执行...")
```

11.7.3 类型断言的最佳实践 1

➢ 在前面的 Usb 接口案例做改进：

给 Phone 结构体增加一个特有的方法 call()，当 Usb 接口接收的是 Phone 变量时，还需要调用 call 方法，走代码:

```go
package main
import (
    "fmt"
)

//声明/定义一个接口
type Usb interface {
    //声明了两个没有实现的方法
    Start()
    Stop()
}
```

```
type Phone struct {
    name string
}

//让 Phone 实现 Usb 接口的方法
func (p Phone) Start() {
    fmt.Println("手机开始工作。。。")
}
func (p Phone) Stop() {
    fmt.Println("手机停止工作。。。")
}

func (p Phone) Call() {
    fmt.Println("手机 在打电话..")
}


type Camera struct {
    name string
}
//让 Camera 实现    Usb 接口的方法
func (c Camera) Start() {
    fmt.Println("相机开始工作。。。")
}
func (c Camera) Stop() {
```

```
    fmt.Println("相机停止工作。。。")
}

type Computer struct {

}

func (computer Computer) Working(usb Usb) {
    usb.Start()
    //如果 usb 是指向 Phone 结构体变量，则还需要调用 Call 方法
    //类型断言..[注意体会!!!]
    if phone, ok := usb.(Phone); ok {
        phone.Call()
    }
    usb.Stop()
}

func main() {
    //定义一个 Usb 接口数组，可以存放 Phone 和 Camera 的结构体变量
    //这里就体现出多态数组
    var usbArr [3]Usb
    usbArr[0] = Phone{"vivo"}
    usbArr[1] = Phone{"小米"}
    usbArr[2] = Camera{"尼康"}

    //遍历 usbArr
```

```
    //Phone 还有一个特有的方法 call()，请遍历 Usb 数组，如果是 Phone 变量，
    //除了调用 Usb 接口声明的方法外，还需要调用 Phone 特有方法 call. =》类型断言
    var computer Computer
    for _, v := range usbArr{
        computer.Working(v)
        fmt.Println()
    }
    //fmt.Println(usbArr)
}
```

11.7.4 类型断言的最佳实践 2

写一函数，循环判断传入参数的类型:

```go
package main
import (
    "fmt"
)
//编写一个函数，可以判断输入的参数是什么类型
func TypeJudge(items... interface{}) {

    for index, x := range items {

        switch x.(type) {
            case bool :
                fmt.Printf("第%v个参数是 bool 类型，值是%v\n", index, x)
            case float32 :
                fmt.Printf("第%v个参数是 float32 类型，值是%v\n", index, x)
            case float64 :
                fmt.Printf("第%v个参数是 float64 类型，值是%v\n", index, x)
            case int, int32, int64 :
                fmt.Printf("第%v个参数是 整数 类型，值是%v\n", index, x)
            case string :
                fmt.Printf("第%v个参数是 string 类型，值是%v\n", index, x)
            default :
                fmt.Printf("第%v个参数是  类型 不确定，值是%v\n", index, x)
        }
```

```go
    }
}

func main() {

    var n1 float32 = 1.1
    var n2 float64 = 2.3
    var n3 int32 = 30
    var name string = "tom"
    address := "北京"
    n4 := 300

    TypeJudge(n1, n2, n3, name, address, n4)


}
```

11.7.5 类型断言的最佳实践 3 【学员自己完成】

在前面代码的基础上，增加判断 Student 类型和 *Student 类型

```go
//编写一个函数，可以判断输入的参数是什么类型
func TypeJudge(items... interface{}) {
    for index, x := range items {
        switch x.(type) {
            case bool :
                fmt.Printf("第%v个参数是 bool 类型，值是%v\n", index, x)
            case float32 :
                fmt.Printf("第%v个参数是 float32 类型，值是%v\n", index, x)
            case float64 :
                fmt.Printf("第%v个参数是 float64 类型，值是%v\n", index, x)
            case int, int32, int64 :
                fmt.Printf("第%v个参数是 整数 类型，值是%v\n", index, x)
            case string :
                fmt.Printf("第%v个参数是 string 类型，值是%v\n", index, x)
            case Student :
                fmt.Printf("第%v个参数是 Student 类型，值是%v\n", index, x)
            case *Student :
                fmt.Printf("第%v个参数是 *Student 类型，值是%v\n", index, x)
            default :
                fmt.Printf("第%v个参数是  类型 不确定，值是%v\n", index, x)
        }
    }
```

第 12 章项目 1-家庭收支记账软件项目

12.1 项目开发流程说明

12.2 项目需求说明

1) 模拟实现基于文本界面的《家庭记账软件》
2) 该软件能够记录家庭的收入、支出，并能够打印收支明细表

12.3 项目的界面

```
-----------------家庭收支记账软件-----------------

                1 收支明细
                2 登记收入
                3 登记支出
                4 退     出

                请选择(1-4)：1

-----------------当前收支明细记录-----------------
收支      账户金额          收支金额          说     明
```

```
-----------------家庭收支记账软件-----------------

                    1 收支明细
                    2 登记收入
                    3 登记支出
                    4 退      出

                    请选择(1-4)：2

本次收入金额：1000
本次收入说明：有人发红包
```

```
-----------------家庭收支记账软件-----------------

                    1 收支明细
                    2 登记收入
                    3 登记支出
                    4 退      出

                    请选择(1-4)：1

-----------------当前收支明细记录-----------------
收支      账户金额          收支金额          说      明
收入      11000             1000              有人发红包
```

其它的界面，我们就直接参考 项目效果图.txt

12.4 项目代码实现

12.4.1 实现基本功能(先使用面向过程,后面改成面向对象)

- 功能 1:**先完成可以显示主菜单**，并且**可以退出**

  思路分析：

  更加给出的界面完成，主菜单的显示, 当用户输入 4 时，就退出该程序

  走代码:

```go
package main
import (
    "fmt"
)
func main() {
    //声明一个变量，保存接收用户输入的选项
    key := ""
    //声明一个变量，控制是否退出for
    loop := true
    //显示这个主菜单
    for {
        fmt.Println("-----------------家庭收支记账软件-----------------")
        fmt.Println("                    1 收支明细")
        fmt.Println("                    2 登记收入")
        fmt.Println("                    3 登记支出")
        fmt.Println("                    4 退出软件")
        fmt.Print("请选择(1-4): ")
        fmt.Scanln(&key)

        switch key {
            case "1":
                fmt.Println("-----------------当前收支明细记录-----------------")
            case "2":
            case "3":
                fmt.Println("登记支出..")
            case "4":
                loop = false
            default :
                fmt.Println("请输入正确的选项..")
        }

        if !loop {
            break
        }
    }
    fmt.Println("你退出家庭记账软件的使用...")
}
```

➢ 功能 2:完成可以显示明细和登记收入的功能

思路分析：

1) 因为需要显示明细，我们定义一个变量 details string 来记录

2) 还需要定义变量来记录余额(balance)、每次收支的金额(money)，每次收支的说明(note)

走代码：

```go
func main() {
    //声明一个变量，保存接收用户输入的选项
    key := ""
    //声明一个变量，控制是否退出for
    loop := true

    //定义账户的余额 []
    balance := 10000.0
    //每次收支的金额
    money := 0.0
    //每次收支的说明
    note := ""
    //收支的详情使用字符串来记录
    //当有收支时，只需要对details 进行拼接处理即可
    details := "收支\t账户金额\t收支金额\t说    明"
    //显示这个主菜单
```

增加了必要的变量

```go
switch key {
    case "1":
        fmt.Println("-----------------当前收支明细记录-----------------")
        fmt.Println(details)
    case "2":
        fmt.Println("本次收入金额:")
        fmt.Scanln(&money)
        balance += money // 修改账户余额
        fmt.Println("本次收入说明:")
        fmt.Scanln(&note)
        //将这个收入情况，拼接到details变量
        //收入    11000          1000           有人发红包
        details += fmt.Sprintf("\n收入\t%v\t%v\t%v", balance, money, note)
```

增加了登记收入的处理

- 功能 3:完成了登记支出的功能

思路分析：

登记支出的功能和登录收入的功能类似，做些修改即可

走代码：

```
    case "3":
        fmt.Println("本次支出金额:")
        fmt.Scanln(&money)
        //这里需要做一个必要的判断
        if money > balance {
            fmt.Println("余额的金额不足")
            break
        }
        balance -= money
        fmt.Println("本次支出说明:")
        fmt.Scanln(&note)
        details += fmt.Sprintf("\n支出\t%v\t%v\t%v", balance, money, note)
    case "4":
```

12.4.2 项目代码实现改进

1) 用户输入 4 退出时，给出提示"你确定要退出吗? y/n"，必须输入正确的 y/n ，否则循环输入指令，直到输入 y 或者 n

```
        case "4":
            fmt.Println("你确定要退出吗? y/n")
            choice := ""
            for {

                fmt.Scanln(&choice)
                if choice == "y" || choice == "n" {
                    break
                }
                fmt.Println("你的输入有误，请重新输入 y/n")
            }

            if choice == "y" {
                loop = false
            }
```

2) 当没有任何收支明细时，提示 "当前没有收支明细... 来一笔吧!"

```
    note := ""
    //定义个变量，记录是否有收支的行为
    flag := false
    //收支的详情使用字符串来记录
```

```go
switch key {
    case "1":
        fmt.Println("-----------------当前收支明细记录-----------------")
        if flag {
            fmt.Println(details)
        } else {
            fmt.Println("当前没有收支明细... 来一笔吧!")
        }
```

3) 在支出时，判断余额是否够，并给出相应的提示

```go
case "3":
    fmt.Println("本次支出金额:")
    fmt.Scanln(&money)
    //这里需要做一个必要的判断
    if money > balance {
        fmt.Println("余额的金额不足")
        break
    }
    balance -= money
    fmt.Println("本次支出说明:")
    fmt.Scanln(&note)
    details += fmt.Sprintf("\n支出\t%v\t%v\t%v", balance, money, note)
    flag = true
```

4) 将**面向过程**的代码修改成**面向对象**的方法，编写 myFamilyAccount.go ，并使用 testMyFamilyAccount.go 去完成测试

思路分析：

把记账软件的功能，封装到一个结构体中，然后调用该结构体的方法，来**实现记账**，**显示明细**。结构体的名字 **FamilyAccount** .

在通过在 **main** 方法中，创建一个结构体 FamilyAccount 实例，实现记账即可.

代码实现:

代码不需要重写，只需要重写组织一下.

familyaccount/main/main.go

```go
package main
import (
    "go_code/familyaccount/utils"
    "fmt"
)
func main() {

    fmt.Println("这个是面向对象的方式完成~~")
    utils.NewFamilyAccount().MainMenu() //思路非常清晰
}
```

familyaccount/utils/familyAccount.go

```go
package utils
import (
    "fmt"
)

type FamilyAccount struct {
    //声明必须的字段.

    //声明一个字段，保存接收用户输入的选项
    key    string
    //声明一个字段，控制是否退出 for
    loop bool
    //定义账户的余额 []
    balance float64
    //每次收支的金额
    money float64
    //每次收支的说明
    note string
```

```
    //定义个字段，记录是否有收支的行为
    flag bool
    //收支的详情使用字符串来记录
    //当有收支时，只需要对 details  进行拼接处理即可
    details string
}

//编写要给工厂模式的构造方法，返回一个*FamilyAccount 实例
func NewFamilyAccount() *FamilyAccount {

    return &FamilyAccount{
        key : "",
        loop : true,
        balance : 10000.0,
        money : 0.0,
        note : "",
        flag : false,
        details : "收支\t 账户金额\t 收支金额\t 说      明",
    }

}

//将显示明细写成一个方法
func (this *FamilyAccount) showDetails() {
    fmt.Println("-----------------当前收支明细记录-----------------")
    if this.flag {
```

```
        fmt.Println(this.details)
    } else {
        fmt.Println("当前没有收支明细... 来一笔吧!")
    }
}

//将登记收入写成一个方法，和*FamilyAccount 绑定
func (this *FamilyAccount) income() {

    fmt.Println("本次收入金额:")
    fmt.Scanln(&this.money)
    this.balance += this.money //  修改账户余额
    fmt.Println("本次收入说明:")
    fmt.Scanln(&this.note)
    //将这个收入情况，拼接到 details 变量
    //收入      11000                1000                有人发红包
    this.details += fmt.Sprintf("\n 收入\t%v\t%v\t%v", this.balance, this.money, this.note)
    this.flag = true
}

//将登记支出写成一个方法，和*FamilyAccount 绑定
func (this *FamilyAccount) pay() {
    fmt.Println("本次支出金额:")
    fmt.Scanln(&this.money)
    //这里需要做一个必要的判断
    if this.money > this.balance {
```

```
            fmt.Println("余额的金额不足")
            //break
        }
        this.balance -= this.money
        fmt.Println("本次支出说明:")
        fmt.Scanln(&this.note)
        this.details += fmt.Sprintf("\n 支出\t%v\t%v\t%v", this.balance, this.money, this.note)
        this.flag = true
}

//将退出系统写成一个方法,和*FamilyAccount 绑定
func (this *FamilyAccount) exit() {

        fmt.Println("你确定要退出吗? y/n")
        choice := ""
        for {

            fmt.Scanln(&choice)
            if choice == "y" || choice == "n" {
                break
            }
            fmt.Println("你的输入有误，请重新输入  y/n")
        }

        if choice == "y" {
            this.loop = false
```

```
    }
}


//给该结构体绑定相应的方法
//显示主菜单
func (this *FamilyAccount) MainMenu() {

    for {
        fmt.Println("\n-----------------家庭收支记账软件-----------------")
        fmt.Println("                    1 收支明细")
        fmt.Println("                    2 登记收入")
        fmt.Println("                    3 登记支出")
        fmt.Println("                    4 退出软件")
        fmt.Print("请选择(1-4)：")
        fmt.Scanln(&this.key)
        switch this.key {
            case "1":
                this.showDetails()
            case "2":
                this.income()
            case "3":
                this.pay()
            case "4":
                this.exit()
            default :
```

```
                fmt.Println("请输入正确的选项..")
        }

        if !this.loop {
            break
        }

    }
}
```

12.4.3 对项目的扩展功能的练习

1) 对上面的项目完成一个转账功能
2) 在使用该软件前，有一个登录功能，只有输入正确的**用户名**和**密码**才能操作.

第 13 章项目 2-客户信息关系系统

13.1 项目需求分析

1) 模拟实现基于文本界面的《**客户信息管理软件**》。

2) 该软件能够实现对客户对象的插入、修改和删除（用切片实现），并能够打印客户明细表

13.2 项目的界面设计

➢ 主菜单界面

```
----------------客户信息管理软件----------------

            1 添 加 客 户
            2 修 改 客 户
            3 删 除 客 户
            4 客 户 列 表
            5 退       出

            请选择(1-5): _
```

➢ 添加客户界面

```
……
        请选择(1-5)：1

----------------------添加客户----------------------
姓名：张三
性别：男
年龄：30
电话：010-56253825
邮箱：zhang@abc.com
----------------------添加完成----------------------
```

➢ 修改客户界面

```
……
请选择(1-5)：2

----------------------修改客户----------------------
请选择待修改客户编号(-1退出)：1
姓名(张三)：<直接回车表示不修改>
性别(男)：
年龄(30)：
电话(010-56253825)：
邮箱(zhang@abc.com)：zsan@abc.com
----------------------修改完成----------------------
```

➢ 删除客户界面

```
……
请选择(1-5)：3

----------------------删除客户----------------------
请选择待删除客户编号(-1退出)：1
确认是否删除(Y/N)：y
----------------------删除完成----------------------
```

➢ 客户列表界面

```
……
请选择(1-5)：4

----------------------------客户列表----------------------------
编号  姓名    性别  年龄  电话          邮箱
 1    张三    男    30    010-56253825  abc@email.com
 2    李四    女    23    010-56253825  lisi@ibm.com
 3    王芳    女    26    010-56253825  wang@163.com
--------------------------客户列表完成--------------------------
```

13.3 客户关系管理系统的程序框架图

13.4 项目功能实现-显示主菜单和完成退出软件功能

- 功能的说明

  当用户运行程序时，可以看到主菜单，当输入 5 时，可以退出该软件.

- 思路分析

  编写 customerView.go ,另外可以把 customer.go 和 customerService.go 写上.

- 代码实现

  customerManage/model/customer.go

```
package model


//声明一个 Customer 结构体，表示一个客户信息
```

```
type Customer struct {
    Id int
    Name string
    Gender string
    Age int
    Phone string
    Email string
}

//使用工厂模式，返回一个 Customer 的实例

func NewCustomer(id int, name string, gender string,
    age int, phone string, email string ) Customer {
    return Customer{
        Id : id,
        Name : name,
        Gender : gender,
        Age : age,
        Phone : phone,
        Email : email,
    }
}
```

customerManage/service/customerService.go

```
package service
import (
```

```
    "go_code/customerManage/model"
)

//该 CustomerService，  完成对 Customer 的操作,包括
//增删改查
type CustomerService struct {
    customers []model.Customer
    //声明一个字段，表示当前切片含有多少个客户
    //该字段后面，还可以作为新客户的 id+1
    customerNum int
}
```

customerManage/view/customerView.go

```
package main

import (
    "fmt"
)

type customerView struct {

    //定义必要字段
    key string //接收用户输入...
    loop bool   //表示是否循环的显示主菜单

}
```

```
//显示主菜单
func (this *customerView) mainMenu() {

    for {

        fmt.Println("-----------------客户信息管理软件-----------------")
        fmt.Println("                    1 添 加 客 户")
        fmt.Println("                    2 修 改 客 户")
        fmt.Println("                    3 删 除 客 户")
        fmt.Println("                    4 客 户 列 表")
        fmt.Println("                    5 退        出")
        fmt.Print("请选择(1-5)：")

        fmt.Scanln(&this.key)
        switch this.key {
            case "1" :
                fmt.Println("添 加 客 户")
            case "2" :
                fmt.Println("修 改 客 户")
            case "3" :
                fmt.Println("删 除 客 户")
            case "4" :
                fmt.Println("客 户 列 表")
            case "5" :
                this.loop = false
```

```
            default :
                fmt.Println("你的输入有误，请重新输入...")
        }

        if !this.loop {
            break
        }

    }

    fmt.Println("你退出了客户关系管理系统...")

}

func main() {
    //在 main 函数中，创建一个 customerView,并运行显示主菜单..
    customerView := customerView{
        key : "",
        loop : true,
    }
    //显示主菜单..
    customerView.mainMenu()

}
```

13.5 项目功能实现-完成显示客户列表的功能

➢ 功能说明

```
-------------------------客户列表-------------------------
编号  姓名      性别    年龄    电话          邮箱
 1    张三      男      30      010-56253825  abc@email.com
 2    李四      女      23      010-56253825   lisi@ibm.com
 3    王芳      女      26      010-56253825  wang@163.com
-----------------------客户列表完成-----------------------
```

➢ 思路分析

➢ 代码实现

customerManage/model/customer.go

```go
//返回用户的信息,格式化的字符串
func (this Customer) GetInfo()  string {
    info := fmt.Sprintf("%v\t%v\t%v\t%v\t%v\t%v\t", this.Id,
        this.Name, this.Gender,this.Age, this.Phone, this.Email)
    return info

}
```

customerManage/service/customerService.go[增加了两个方法]

```go
//编写一个方法，可以返回 *CustomerService
func NewCustomerService() *CustomerService {
    //为了能够看到有客户在切片中，我们初始化一个客户
    customerService := &CustomerService{}
    customerService.customerNum = 1
    customer := model.NewCustomer(1, "张三", "男", 20, "112", "zs@sohu.com")
    customerService.customers = append(customerService.customers, customer)
    return customerService
}

//返回客户切片
func (this *CustomerService) List() []model.Customer {
    return this.customers
}
```

customerManage/view/customerView.go

```go
package main

import (
    "fmt"
    "go_code/customerManage/service"
)

type customerView struct {

    //定义必要字段
    key string //接收用户输入...
    loop bool   //表示是否循环的显示主菜单
    //增加一个字段 customerService
    customerService *service.CustomerService

}
```

```
//显示所有的客户信息
func (this *customerView) list() {

    //首先，获取到当前所有的客户信息(在切片中)
    customers := this.customerService.List()
    //显示
    fmt.Println("--------------------------客户列表--------------------------")
    fmt.Println("编号\t 姓名\t 性别\t 年龄\t 电话\t 邮箱")
    for i := 0; i < len(customers); i++ {
        //fmt.Println(customers[i].Id,"\t", customers[i].Name...)
        fmt.Println(customers[i].GetInfo())
    }
    fmt.Printf("\n-------------------------客户列表完成-------------------------\n\n")
}

//显示主菜单
func (this *customerView) mainMenu() {

    for {

        fmt.Println("-----------------客户信息管理软件-----------------")
        fmt.Println("                    1 添 加 客 户")
        fmt.Println("                    2 修 改 客 户")
        fmt.Println("                    3 删 除 客 户")
        fmt.Println("                    4 客 户 列 表")
```

```
        fmt.Println("                    5 退          出")
        fmt.Print("请选择(1-5)：")

        fmt.Scanln(&this.key)
        switch this.key {
            case "1" :
                fmt.Println("添 加 客 户")
            case "2" :
                fmt.Println("修 改 客 户")
            case "3" :
                fmt.Println("删 除 客 户")
            case "4" :
                this.list()
            case "5" :
                this.loop = false
            default :
                fmt.Println("你的输入有误，请重新输入...")
        }

        if !this.loop {
            break
        }

    }

    fmt.Println("你退出了客户关系管理系统...")
```

```
}

func main() {
    //在 main 函数中，创建一个 customerView,并运行显示主菜单..
    customerView := customerView{
        key : "",
        loop : true,
    }
    //这里完成对 customerView 结构体的 customerService 字段的初始化
    customerView.customerService = service.NewCustomerService()
    //显示主菜单..
    customerView.mainMenu()

}
```

13.6 项目功能实现-添加客户的功能

- 功能说明

```
---------------------添加客户---------------------
姓名：张三
性别：男
年龄：30
电话：010-56253825
邮箱：zhang@abc.com
---------------------添加完成---------------------
```

- 思路分析

➢ 代码实现

customerManage/model/customer.go

```
//第二种创建Customer实例方法，不带id
func NewCustomer2(name string, gender string,
    age int, phone string, email string ) Customer {
    return Customer{
        Name : name,
        Gender : gender,
        Age : age,
        Phone : phone,
        Email : email,
    }
}
```

customerManage/service/customerService.go

```go
//添加客户到customers切片
//!!!
func (this *CustomerService) Add(customer model.Customer) bool {

    //我们确定一个分配id的规则,就是添加的顺序
    this.customerNum++
    customer.Id = this.customerNum
    this.customers = append(this.customers, customer)
    return true
}
```

customerManage/service/customerView.go

```go
//得到用户的输入，信息构建新的客户，并完成添加
func (this *customerView) add() {
    fmt.Println("---------------------添加客户---------------------")
    fmt.Println("姓名:")
    name := ""
    fmt.Scanln(&name)
    fmt.Println("性别:")
    gender := ""
    fmt.Scanln(&gender)
    fmt.Println("年龄:")
    age := 0
    fmt.Scanln(&age)
    fmt.Println("电话:")
    phone := ""
    fmt.Scanln(&phone)
    fmt.Println("电邮:")
    email := ""
```

```
    fmt.Scanln(&email)
    //构建一个新的 Customer 实例
    //注意: id 号，没有让用户输入，id 是唯一的，需要系统分配
    customer := model.NewCustomer2(name, gender, age, phone, email)
    //调用
    if this.customerService.Add(customer) {
        fmt.Println("---------------------添加完成---------------------")
    } else {
        fmt.Println("---------------------添加失败---------------------")
    }
}
```

```
func (this *customerView) mainMenu() {

    for {

        fmt.Println("-----------------客户信息管理软件-----------------")
        fmt.Println("                 1 添 加 客 户")
        fmt.Println("                 2 修 改 客 户")
        fmt.Println("                 3 删 除 客 户")
        fmt.Println("                 4 客 户 列 表")
        fmt.Println("                 5 退       出")
        fmt.Print("请选择(1-5): ")

        fmt.Scanln(&this.key)
        switch this.key {
            case "1" :
                this.add()
            case "2" :
```

调用add方法

13.7 项目功能实现-完成删除客户的功能

➢ 功能说明

```
----------------------删除客户----------------------
请选择待删除客户编号(-1退出)：1
确认是否删除(Y/N)：y
----------------------删除完成----------------------
```

➢ 思路分析

➢ 代码实现

customerManage/model/customer.go [没有变化]

customerManage/service/customerService.go

customerView.go customer.go customerService.go main.go

```go
func (this *CustomerService) Delete(id int) bool {
    index := this.FindById(id)
    //如果index == -1, 说明没有这个客户
    if index == -1 {
        return false
    }
    //如何从切片中删除一个元素
    this.customers = append(this.customers[:index], this.customers[index+1:]...)
    return true
}

//根据id查找客户在切片中对应下标,如果没有该客户, 返回-1
func (this *CustomerService) FindById(id int)  int {
    index := -1
    //遍历this.customers 切片
    for i := 0; i < len(this.customers); i++ {
        if this.customers[i].Id == id {
            //找到
            index = i
        }
    }
    return index
}
```

customerManage/view/customerView.go

```go
//得到用户的输入id，删除该id对应的客户
func (this *customerView) delete() {
    fmt.Println("---------------------删除客户---------------------")
    fmt.Println("请选择待删除客户编号(-1退出)：")
    id := -1
    fmt.Scanln(&id)
    if id == -1 {
        return //放弃删除操作
    }
    fmt.Println("确认是否删除(Y/N)：")
    //这里同学们可以加入一个循环判断，直到用户输入 y 或者 n,才退出..
    choice := ""
    fmt.Scanln(&choice)
    if choice == "y" || choice == "Y" {
        //调用customerService 的 Delete方法
        if this.customerService.Delete(id) {
            fmt.Println("---------------------删除完成---------------------")
        } else {
            fmt.Println("---------------------删除失败，输入的id号不存在----")
        }
    }
}
```

13.8 项目功能实现-完善退出确认功能（课后作业）

- 功能说明：

  要求用户在退出时提示 " 确认是否退出(Y/N)："，用户必须输入 y/n，否则循环提示。

- 思路分析：

  需要编写 customerView.go

- 代码实现：

```go
//退出软件
func (this *customerView) exit() {

    fmt.Println("确认是否退出(Y/N): ")
    for {
        fmt.Scanln(&this.key)
        if this.key == "Y" || this.key == "y" || this.key == "N" || this.key == "n" {
            break
        }

        fmt.Println("你的输入有误，确认是否退出(Y/N): ")
    }

    if this.key == "Y" || this.key == "y" {
        this.loop = false
    }

}
```

13.9 客户关系管理系统-课后练习

项目功能实现-完成修改客户的功能（课后作业）

功能说明：

```
---------------------修改客户---------------------
请选择待修改客户编号(-1退出)：1
姓名(张三)：<直接回车表示不修改>
性别(男)：
年龄(30)：
电话(010-56253825)：
邮箱(zhang@abc.com)：zsan@abc.com
---------------------修改完成---------------------
```

思路分析：

需要编写 CustomerView.go 和 CustomerService.go

代码实现：

第 14 章文件操作

14.1 文件的基本介绍

➢ 文件的概念

文件,对我们并不陌生,文件是数据源(保存数据的地方)的一种,比如大家经常使用的 word 文档,txt 文件,excel 文件...都是文件。文件最主要的作用就是保存数据,它既可以保存一张图片,也可以保持视频,声音...

➢ 输入流和输出流

文件在程序中是以流的形式来操作的。

流：数据在数据源(文件)和程序(内存)之间经历的路径
输入流：数据从数据源(文件)到程序(内存)的路径
输出流：数据从程序(内存)到数据源(文件)的路径

➢ **os**.File 封装所有文件相关操作，File 是一个结构体

type File

```
type File struct {
    // 内含隐藏或非导出字段
}
```

func (*File) Read

```
func (f *File) Read(b []byte) (n int, err error)
```

Read方法从f中读取最多len(b)字节数据并写入b。它返回读取的字节数和可能遇到的任何错误。文件终止标志是读取0个字节且返回值err为io.EOF。

func (*File) ReadAt

```
func (f *File) ReadAt(b []byte, off int64) (n int, err error)
```

ReadAt从指定的位置（相对于文件开始位置）读取len(b)字节数据并写入b。它返回读取的字节数和可能遇到的任何错误。当n<len(b)时，本方法总是会返回错误；如果是因为到达文件结尾，返回值err会是io.EOF。

func (*File) Write

```
func (f *File) Write(b []byte) (n int, err error)
```

Write向文件中写入len(b)字节数据。它返回写入的字节数和可能遇到的任何错误。如果返回值n!=len(b)，本方法会返回一个非nil的错误。

func (*File) WriteString

```
func (f *File) WriteString(s string) (ret int, err error)
```

func (*File) WriteAt

```
func (f *File) WriteAt(b []byte, off int64) (n int, err error)
```

WriteAt在指定的位置（相对于文件开始位置）写入len(b)字节数据。它返回写入的字节数和可能遇到的任何错
如果返回值n!=len(b)，本方法会返回一个非nil的错误。

func (*File) Seek

```
func (f *File) Seek(offset int64, whence int) (ret int64, err error)
```

Seek设置下一次读/写的位置。offset为相对偏移量，而whence决定相对位置：0为相对文件开头，1为相对当
置，2为相对文件结尾。它返回新的偏移量（相对开头）和可能的错误。

func (*File) Sync

```
func (f *File) Sync() (err error)
```

Sync通交文件的当前内容进行稳定的存储。一般来说，这表示将文件系统的最近写入的数据在内存中的拷贝
到硬盘中稳定保存。

func (*File) Close

```
func (f *File) Close() error
```

Close关闭文件f，使文件不能用于读写。它返回可能出现的错误。

总结：后面我们操作文件，会经常使用到 os.File 结构体.

14.2 打开文件和关闭文件

➢ 使用的函数和方法

func Open

```
func Open(name string) (file *File, err error)
```

Open打开一个文件用于读取。如果操作成功，返回的文件对象的方法可用于读取数据；对应的文件描述符具有O_RDONLY模式。如果出错，错误底层类型是*PathError。

func (*File) Close

```
func (f *File) Close() error
```

Close关闭文件f，使文件不能用于读写。它返回可能出现的错误。

➢ 案例演示

```go
package main
import (
    "fmt"
    "os"
)
func main() {
    //打开文件
    //概念说明: file 的叫法
    //1. file 叫 file对象
    //2. file 叫 file指针
    //3. file 叫 file 文件句柄
    file , err := os.Open("d:/test.txt")
    if err != nil {
        fmt.Println("open file err=", err)
    }
    //输出下文件，看看文件是什么，看出file 就是一个指针 *File
    fmt.Printf("file=%v", file)
    //关闭文件
    err = file.Close()
    if err != nil {
        fmt.Println("close file err=", err)
    }
}
```

14.3 读文件操作应用实例

1) **读取文件的内容**并**显示在终端**(带**缓冲区**的方式)，使用 os.Open, file.Close, bufio.NewReader(), reader.ReadString 函数和方法.

代码实现：

```
package main
import (
    "fmt"
    "os"
    "bufio"
    "io"
)
```

```
func main() {
    //打开文件
    //概念说明: file  的叫法
    //1. file  叫  file 对象
    //2. file  叫  file 指针
    //3. file  叫  file  文件句柄
    file , err := os.Open("d:/test.txt")
    if err != nil {
        fmt.Println("open file err=", err)
    }

    //当函数退出时，要及时的关闭 file
    defer file.Close() //要及时关闭 file 句柄，否则会有内存泄漏.

    //  创建一个  *Reader    ，是带缓冲的
    /*
    const (
    defaultBufSize = 4096 //默认的缓冲区为 4096
    )
    */
    reader := bufio.NewReader(file)
    //循环的读取文件的内容
    for {
        str, err := reader.ReadString('\n') //  读到一个换行就结束
        if err == io.EOF { // io.EOF 表示文件的末尾
            break
```

```
        }
        //输出内容
        fmt.Print(str)
    }

    fmt.Println("文件读取结束...")
}
```

2) 读取文件的内容并显示在终端(使用 ioutil 一次将整个文件读入到内存中)，这种方式适用于文件不大的情况。相关方法和函数(ioutil.ReadFile)

代码演示：

```go
package main
import (
    "fmt"
    "io/ioutil"
)
func main() {

    //使用ioutil.ReadFile一次性将文件读取到位
    file := "d:/test.txt"
    content, err := ioutil.ReadFile(file)
    if err != nil {
        fmt.Printf("read file err=%v", err)
    }
    //把读取到的内容显示到终端
    //fmt.Printf("%v", content) // []byte
    fmt.Printf("%v", string(content)) // []byte

    //我们没有显式的Open文件，因此也不需要显式的Close文件
    //因为，文件的Open和Close被封装到 ReadFile 函数内部
}
```

14.4 写文件操作应用实例

14.4.1 基本介绍-os.OpenFile 函数

func OpenFile(name string, flag int, perm FileMode) (file *File, err error)

说明: **os.OpenFile**是一个更一般性的文件打开函数，它会使用指定的选项（如O_RDONLY等）、指定的模式（如0666等）打开指定名称的文件。如果操作成功，返回的文件对象可用于I/O。如果出错，错误底层类型是*PathError。

第二个参数：文件打开模式(可以组合):

```
const (
    O_RDONLY int = syscall.O_RDONLY // 只读模式打开文件
    O_WRONLY int = syscall.O_WRONLY // 只写模式打开文件
    O_RDWR   int = syscall.O_RDWR   // 读写模式打开文件
    O_APPEND int = syscall.O_APPEND // 写操作时将数据附加到文件尾部
    O_CREATE int = syscall.O_CREAT  // 如果不存在将创建一个新文件
    O_EXCL   int = syscall.O_EXCL   // 和O_CREATE配合使用，文件必须不存在
    O_SYNC   int = syscall.O_SYNC   // 打开文件用于同步I/O
    O_TRUNC  int = syscall.O_TRUNC  // 如果可能，打开时清空文件
)
```

第三个参数：权限控制(linux) /Un 区

r->4

w->2

x->1

详情：参考尚硅谷-linux课程

14.4.2 基本应用实例-方式一

1) 创建一个新文件，写入内容 5 句 "hello, Gardon"

代码实现：

```go
package main
import (
    "fmt"
    "bufio"
    "os"
)
func main() {
    //创建一个新文件，写入内容 5句 "hello, Gardon"
    //1 .打开文件 d:/abc.txt
    filePath := "d:/abc.txt"
    file, err := os.OpenFile(filePath, os.O_WRONLY | os.O_CREATE, 0666)
    if err != nil {
        fmt.Printf("open file err=%v\n", err)
        return
    }
    //及时关闭file句柄
    defer file.Close()
    //准备写入5句 "hello, Gardon"
    str := "hello,Gardon\n" // \n 表示换行
    //写入时，使用带缓存的 *Writer
    writer := bufio.NewWriter(file)
    for i := 0; i < 5; i++ {
        writer.WriteString(str)
```

```
    }
    //因为writer是带缓存，因此在调用WriterString方法时，其实
    //内容是先写入到缓存的,所以需要调用Flush方法，将缓冲的数据
    //真正写入到文件中，  否则文件中会没有数据!!!
    writer.Flush()
}
```

2) 打开一个存在的文件中，将原来的内容覆盖成新的内容 10 句 "你好，尚硅谷!"

```
package main
import (
    "fmt"
    "bufio"
    "os"
)

func main() {
    //打开一个存在的文件中，将原来的内容覆盖成新的内容 10 句 "你好，尚硅谷!"

    //创建一个新文件，写入内容 5 句 "hello, Gardon"
    //1 .打开文件已经存在文件, d:/abc.txt
    filePath := "d:/abc.txt"
    file, err := os.OpenFile(filePath, os.O_WRONLY | os.O_TRUNC, 0666)
    if err != nil {
        fmt.Printf("open file err=%v\n", err)
        return
    }
    //及时关闭 file 句柄
    defer file.Close()
```

```
    //准备写入 5 句 "你好,尚硅谷!"
    str := "你好,尚硅谷!\r\n" // \r\n 表示换行
    //写入时，使用带缓存的 *Writer
    writer := bufio.NewWriter(file)
    for i := 0; i < 10; i++ {
        writer.WriteString(str)
    }
    //因为 writer 是带缓存，因此在调用 WriterString 方法时，其实
    //内容是先写入到缓存的,所以需要调用 Flush 方法，将缓冲的数据
    //真正写入到文件中， 否则文件中会没有数据!!!
    writer.Flush()

}
```

3) 打开一个存在的文件，在原来的内容追加内容 'ABC! ENGLISH!'

代码实现：

```
package main
import (
    "fmt"
    "bufio"
    "os"
)

func main() {
```

```
    //打开一个存在的文件，在原来的内容追加内容 'ABC! ENGLISH!'
    //1 .打开文件已经存在文件, d:/abc.txt
    filePath := "d:/abc.txt"
    file, err := os.OpenFile(filePath, os.O_WRONLY | os.O_APPEND, 0666)
    if err != nil {
        fmt.Printf("open file err=%v\n", err)
        return
    }
    //及时关闭 file 句柄
    defer file.Close()
    //准备写入 5 句 "你好,尚硅谷!"
    str := "ABC,ENGLISH!\r\n" // \r\n 表示换行
    //写入时，使用带缓存的 *Writer
    writer := bufio.NewWriter(file)
    for i := 0; i < 10; i++ {
        writer.WriteString(str)
    }
    //因为 writer 是带缓存，因此在调用 WriterString 方法时，其实
    //内容是先写入到缓存的,所以需要调用 Flush 方法，将缓冲的数据
    //真正写入到文件中， 否则文件中会没有数据!!!
    writer.Flush()

}
```

4) 打开一个存在的文件，将原来的内容**读出**显示在终端，并且**追加** 5 句"hello,北京!"

代码实现：

```
package main
import (
    "fmt"
    "bufio"
    "os"
    "io"
)

func main() {

    //打开一个存在的文件，将原来的内容读出显示在终端，并且追加 5 句"hello,北京!"
    //1 .打开文件已经存在文件, d:/abc.txt
    filePath := "d:/abc.txt"
    file, err := os.OpenFile(filePath, os.O_RDWR | os.O_APPEND, 0666)
    if err != nil {
        fmt.Printf("open file err=%v\n", err)
        return
    }
    //及时关闭 file 句柄
    defer file.Close()

    //先读取原来文件的内容，并显示在终端.
    reader := bufio.NewReader(file)
    for {
        str, err := reader.ReadString('\n')
        if err == io.EOF { //如果读取到文件的末尾
```

```
            break
        }
        //显示到终端
        fmt.Print(str)
    }



    //准备写入 5 句 "你好,尚硅谷!"
    str := "hello,北京!\r\n" // \r\n 表示换行
    //写入时，使用带缓存的 *Writer
    writer := bufio.NewWriter(file)
    for i := 0; i < 5; i++ {
        writer.WriteString(str)
    }
    //因为 writer 是带缓存，因此在调用 WriterString 方法时，其实
    //内容是先写入到缓存的,所以需要调用 Flush 方法，将缓冲的数据
    //真正写入到文件中，  否则文件中会没有数据!!!
    writer.Flush()

}
```

14.4.3 基本应用实例-方式二

编程一个程序，将一个文件的内容，写入到另外一个文件。注：这两个文件已经存在了.

说明：使用 ioutil.ReadFile / ioutil.WriteFile 完成写文件的任务.

代码实现：

```go
package main
import (
    "fmt"
    "io/ioutil"
)
func main() {
    //将d:/abc.txt 文件内容导入到  e:/kkk.txt
    //1. 首先将  d:/abc.txt 内容读取到内存
    //2. 将读取到的内容写入 e:/kkk.txt
    file1Path := "d:/abc.txt"
    file2Path := "e:/kkk.txt"
    data, err := ioutil.ReadFile(file1Path)
    if err != nil {
        //说明读取文件有错误
        fmt.Printf("read file err=%v\n", err)
        return
    }
    err = ioutil.WriteFile(file2Path, data, 0666)
    if err != nil {
        fmt.Printf("write file error=%v\n", err)
    }
}
```

14.4.4 判断文件是否存在

golang判断**文件或文件夹是否**存在的方法为使用os.Stat()函数返回的错误值进行判断:

1) 如果返回的错误为nil,说明文件或文件夹存在
2) 如果返回的错误类型使用os.IsNotExist()判断为true,说明文件或文件夹不存在
3) 如果返回的错误为其它类型,则不确定是否在存在

//自己写了一个函数

```go
func PathExists(path string) (bool, error) {
        _, err := os.Stat(path)
        if err == nil { // 文件或者目录存在
                return true, nil
        }
        if os.IsNotExist(err) {
                return false, nil
        }
        return false, err
}
```

14.5 文件编程应用实例

14.5.1 拷贝文件

说明：将一张图片/电影/mp3 拷贝到另外一个文件 e:/abc.jpg io 包

func Copy(dst Writer, src Reader) (written int64, err error)

注意; Copy 函数是 io 包提供的.

代码实现：

```
package main
import (
    "fmt"
    "os"
    "io"
    "bufio"
)

//自己编写一个函数，接收两个文件路径 srcFileName dstFileName
func CopyFile(dstFileName string, srcFileName string) (written int64, err error) {

    srcFile, err := os.Open(srcFileName)
    if err != nil {
        fmt.Printf("open file err=%v\n", err)
    }
    defer srcFile.Close()
    //通过 srcfile ,获取到 Reader
    reader := bufio.NewReader(srcFile)
```

```
    //打开 dstFileName
    dstFile, err := os.OpenFile(dstFileName, os.O_WRONLY | os.O_CREATE, 0666)
    if err != nil {
        fmt.Printf("open file err=%v\n", err)
        return
    }

    //通过 dstFile, 获取到 Writer
    writer := bufio.NewWriter(dstFile)
    defer dstFile.Close()

    return io.Copy(writer, reader)


}

func main() {

    //将 d:/flower.jpg 文件拷贝到 e:/abc.jpg

    //调用 CopyFile 完成文件拷贝
    srcFile := "d:/flower.jpg"
    dstFile := "e:/abc.jpg"
    _, err := CopyFile(dstFile, srcFile)
    if err == nil {
        fmt.Printf("拷贝完成\n")
```

```
    } else {
        fmt.Printf("拷贝错误  err=%v\n", err)
    }

}
```

14.5.2 统计英文、数字、空格和其他字符数量

说明：统计一个文件中含有的英文、数字、空格及其它字符数量

代码实现：

```
package main
import (
    "fmt"
    "os"
    "io"
    "bufio"
)

//定义一个结构体，用于保存统计结果
type CharCount struct {
    ChCount int //  记录英文个数
    NumCount int //  记录数字的个数
    SpaceCount int //  记录空格的个数
    OtherCount int //  记录其它字符的个数
}
```

```
func main() {

    //思路：打开一个文件，创一个 Reader
    //每读取一行，就去统计该行有多少个 英文、数字、空格和其他字符
    //然后将结果保存到一个结构体
    fileName := "e:/abc.txt"
    file, err := os.Open(fileName)
    if err != nil {
        fmt.Printf("open file err=%v\n", err)
        return
    }
    defer file.Close()
    //定义个 CharCount 实例
    var count CharCount
    //创建一个 Reader
    reader := bufio.NewReader(file)

    //开始循环的读取 fileName 的内容
    for {
        str, err := reader.ReadString('\n')
        if err == io.EOF { //读到文件末尾就退出
            break
        }
        //为了兼容中文字符，可以将 str 转成 []rune
        str = []run(str)
        //遍历 str ，进行统计
```

```
        for _, v := range str {

            switch {
                case v >= 'a' && v <= 'z':
                        fallthrough //穿透
                case v >= 'A' && v <= 'Z':
                        count.ChCount++
                case v == ' ' || v == '\t':
                        count.SpaceCount++
                case v >= '0' && v <= '9':
                        count.NumCount++
                default :
                        count.OtherCount++
            }
        }
    }

    //输出统计的结果看看是否正确
    fmt.Printf("字符的个数为=%v 数字的个数为=%v 空格的个数为=%v 其它字符个数=%v",
        count.ChCount, count.NumCount, count.SpaceCount, count.OtherCount)
}
```

14.6 命令行参数

14.6.1 看一个需求

我们希望能够获取到命令行输入的各种参数，该如何处理？如图：=> 命令行参数

```
D:\goproject\src\go_code>test.exe tom c:/aa/bb/config.init 88
```

14.6.2 基本介绍

os.Args 是一个 string 的切片，用来存储所有的命令行参数

14.6.3 举例说明

请编写一段代码，可以获取命令行各个参数

```
D:\goproject\src\go_code>test.exe tom c:/aaa/bbb/test.log 99
参数个数=:4
args[0]=test.exe
args[1]=tom
args[2]=c:/aaa/bbb/test.log
args[3]=99
```

代码实现:

```go
package main
import (
    "fmt"
    "os"
)

func main() {

    fmt.Println("命令行的参数有", len(os.Args))
    //遍历os.Args切片，就可以得到所有的命令行输入参数值
    for i, v := range os.Args {
        fmt.Printf("args[%v]=%v\n", i, v)
    }
}
```

14.6.4 flag 包用来解析命令行参数

说明: 前面的方式是比较原生的方式，对解析参数不是特别的方便，特别是带有指定参数形式的命令行。

比如：cmd>main.exe -f c:/aaa.txt -p 200 -u root 这样的形式命令行，go 设计者给我们提供了 flag 包，可以方便的解析命令行参数，而且参数顺序可以随意

请编写一段代码，可以获取命令行各个参数.

```
D:\goproject\src\go_code>test.exe -u root -pwd root -h 192.168.0.1 -port 3306
user= root
pwd= root
host= 192.168.0.1
port= 3306
```

代码实现：

```go
package main
import (
    "fmt"
    "flag"
)

func main() {

    //定义几个变量，用于接收命令行的参数值
    var user string
    var pwd string
    var host string
    var port int

    //&user 就是接收用户命令行中输入的 -u 后面的参数值
    //"u" ,就是 -u 指定参数
    //"" , 默认值
    //"用户名,默认为空" 说明
    flag.StringVar(&user, "u", "", "用户名,默认为空")
    flag.StringVar(&pwd, "pwd", "", "密码,默认为空")
    flag.StringVar(&host, "h", "localhost", "主机名,默认为localhost")
    flag.IntVar(&port, "port", 3306, "端口号，默认为3306")
    //这里有一个非常重要的操作,转换， 必须调用该方法
    flag.Parse()

    //输出结果
    fmt.Printf("user=%v pwd=%v host=%v port=%v",
        user, pwd, host, port)

}
```

14.7 json 基本介绍

➢ 概述

JSON(JavaScript Object Notation) 是一种**轻量级的数据交换格式**。易于人阅读和编写。同时也易于机器解析和生成。key-val

JSON是在2001年开始推广使用的数据格式，目前已经成为**主流的数据格式**。

JSON易于机器解析和生成，并有效地提升**网络传输效率,通常程序在网络传输时会先将数据(结构体、map等)序列化成json字符串,到接收方得到json字符串时，在反序列化恢复成原来的数据类型(结构体、map等)。这种方式已然成为各个语言的标准。**

➢ 应用场景(示意图)

14.8 json 数据格式说明

在 JS 语言中，一切都是对象。因此，任何的数据类型都可以通过 JSON 来表示，例如字符串、数字、对象、数组, map, 结构体等.

任何数据类型都可以转成json格式..

JSON **键值**对是用来保存 数据一种方式，
键/值对组合中的键名写在前面并用双引号 "" 包裹，使用冒号 : 分隔，然后紧接着值：
[{"key1":val1, "key2":val2, "key3" : val3, "key4":[val4, val5]},
{"key1":val1, "key2":val2, "key3" : val3, "key4":[val4, val5]}]

比如:
{"firstName": "Json"}
比如:
{"name": "tom", "age": 18, "address":["北京", "上海"]}
比如:
[{"name": "tom", "age": 18, "address":["北京", "上海"]},
{"name": "mary", "age": 28, "address":["广州", "深圳"]}]

14.9 json 数据在线解析

https://www.json.cn/ 网站可以验证一个 json 格式的数据是否正确。尤其是在我们编写比较复杂的 json 格式数据时，很有用。

14.10json 的序列化

➢ 介绍

json 序列化是指，将有 **key-value** 结构的数据类型(比如**结构体**、**map**、**切片**)序列化成 json 字符串的操作。

➢ 应用案例

这里我们介绍一下**结构体**、**map 和切片**的序列化，其它数据类型的序列化类似。

➢ 代码演示

```
package main
import (
    "fmt"
    "encoding/json"
)

//定义一个结构体
type Monster struct {
    Name string
    Age int
    Birthday string
    Sal float64
    Skill string
}

func testStruct() {
    //演示
    monster := Monster{
```

```
        Name :"牛魔王",
        Age : 500,
        Birthday : "2011-11-11",
        Sal : 8000.0,
        Skill : "牛魔拳",
    }

    //将 monster  序列化
    data, err := json.Marshal(&monster)
    if err != nil {
        fmt.Printf("序列号错误  err=%v\n", err)
    }
    //输出序列化后的结果
    fmt.Printf("monster 序列化后=%v\n", string(data))

}

//将 map 进行序列化
func testMap() {
    //定义一个 map
    var a map[string]interface{}
    //使用 map,需要 make
    a = make(map[string]interface{})
    a["name"] = "红孩儿"
    a["age"] = 30
    a["address"] = "洪崖洞"
```

```
    //将 a 这个 map 进行序列化
    //将 monster 序列化
    data, err := json.Marshal(a)
    if err != nil {
        fmt.Printf("序列化错误 err=%v\n", err)
    }
    //输出序列化后的结果
    fmt.Printf("a map 序列化后=%v\n", string(data))

}

//演示对切片进行序列化，我们这个切片 []map[string]interface{}
func testSlice() {
    var slice []map[string]interface{}
    var m1 map[string]interface{}
    //使用 map 前，需要先 make
    m1 = make(map[string]interface{})
    m1["name"] = "jack"
    m1["age"] = "7"
    m1["address"] = "北京"
    slice = append(slice, m1)

    var m2 map[string]interface{}
    //使用 map 前，需要先 make
    m2 = make(map[string]interface{})
```

```
    m2["name"] = "tom"
    m2["age"] = "20"
    m2["address"] = [2]string{"墨西哥","夏威夷"}
    slice = append(slice, m2)

    //将切片进行序列化操作
    data, err := json.Marshal(slice)
    if err != nil {
        fmt.Printf("序列化错误  err=%v\n", err)
    }
    //输出序列化后的结果
    fmt.Printf("slice  序列化后=%v\n", string(data))

}

//对基本数据类型序列化，对基本数据类型进行序列化意义不大
func testFloat64() {
    var num1 float64 = 2345.67

    //对 num1 进行序列化
    data, err := json.Marshal(num1)
    if err != nil {
        fmt.Printf("序列化错误  err=%v\n", err)
    }
    //输出序列化后的结果
    fmt.Printf("num1  序列化后=%v\n", string(data))
```

```
}

func main() {
    //演示将结构体, map , 切片进行序列号
    testStruct()
    testMap()
    testSlice()//演示对切片的序列化
    testFloat64()//演示对基本数据类型的序列化
}
```

➢ 注意事项

对于结构体的序列化，如果我们希望序列化后的 key 的名字，又我们自己重新制定，那么可以给 struct 指定一个 tag 标签.

```go
//定义一个结构体
type Monster struct {
    Name string `json:"monster_name"` //反射机制
    Age int `json:"monster_age"`
    Birthday string //....
    Sal float64
    Skill string
}
```

序列化后：

{"**monster_name**":"牛魔王","**monster_age**":500,"Birthday":"2011-11-11","Sal":8000,"Skill":"牛魔拳"}

14.11json 的反序列化

➢ 基本介绍

json 反序列化是指，将 json 字符串反序列化成对应的数据类型(比如结构体、map、切片)的操作

- 应用案例

这里我们介绍一下将 json 字符串反序列化成结构体、map 和切片

代码演示：

```
package main
import (
    "fmt"
    "encoding/json"
)

//定义一个结构体
type Monster struct {
    Name string
    Age int
    Birthday string //....
    Sal float64
    Skill string
}

//演示将 json 字符串，反序列化成 struct
func unmarshalStruct() {
    //说明 str 在项目开发中，是通过网络传输获取到.. 或者是读取文件获取到
    str := "{\"Name\":\"牛魔王\",\"Age\":500,\"Birthday\":\"2011-11-11\",\"Sal\":8000,\"Skill\":\"牛魔拳\"}"

    //定义一个 Monster 实例
    var monster Monster
```

```
    err := json.Unmarshal([]byte(str), &monster)
    if err != nil {
        fmt.Printf("unmarshal err=%v\n", err)
    }
    fmt.Printf("反序列化后  monster=%v monster.Name=%v \n", monster, monster.Name)

}

//演示将 json 字符串，反序列化成 map
func unmarshalMap() {
    str := "{\"address\":\"洪崖洞\",\"age\":30,\"name\":\"红孩儿\"}"

    //定义一个 map
    var a map[string]interface{}

    //反序列化
    //注意：反序列化 map,不需要 make,因为 make 操作被封装到 Unmarshal 函数
    err := json.Unmarshal([]byte(str), &a)
    if err != nil {
        fmt.Printf("unmarshal err=%v\n", err)
    }
    fmt.Printf("反序列化后  a=%v\n", a)

}
```

```
//演示将 json 字符串，反序列化成切片
func unmarshalSlice() {
    str := "[{\"address\":\"北京\",\"age\":\"7\",\"name\":\"jack\"}," +
        "{\"address\":[\"墨西哥\",\"夏威夷\"],\"age\":\"20\",\"name\":\"tom\"}]"

    //定义一个 slice
    var slice []map[string]interface{}
    //反序列化，不需要 make,因为 make 操作被封装到 Unmarshal 函数
    err := json.Unmarshal([]byte(str), &slice)
    if err != nil {
        fmt.Printf("unmarshal err=%v\n", err)
    }
    fmt.Printf("反序列化后 slice=%v\n", slice)
}

func main() {

    unmarshalStruct()
    unmarshalMap()
    unmarshalSlice()
}
```

- 对上面代码的小结说明
  1) 在反序列化一个 json 字符串时，要确保**反序列化后的数据类型**和原来**序列化前的数据类型**一致。
  2) 如果 json 字符串是通过程序获取到的，则不需要再对“ 转义处理。

```
//演示将json字符串，反序列化成map
func unmarshalMap() {
    //str := "{\"address\":\"洪崖洞\",\"age\":30,\"name\":\"红孩儿\"}"
    str := testMap()
    //定义一个map
    var a map[string]interface{}
```

直接写str是需要转义

通过程序获取的，不需要转义.因为内部已经转义

第 15 章单元测试

15.1 先看一个需求

在我们工作中，我们会遇到这样的情况，就是去确认一个函数，或者一个模块的结果是否正确，如：

```go
func addUpper(n int) int {
    res := 0
    for i := 1; i <= n; i++ {
        res += i
    }
    return res
}
```

15.2 传统的方法

15.2.1 传统的方式来进行测试

在 main 函数中，调用 addUpper 函数，看看实际输出的结果是否和预期的结果一致，如果一致，则说明函数正确，否则函数有错误，然后修改错误

代码实现：

```go
package main
import (
    "fmt"
)

//一个被测试函数
func addUpper(n int)  int {
    res := 0
    for i := 1; i <= n - 1; i++ {
        res += i
    }
    return res
}

func main() {

    //传统的测试方法，就是在main函数中使用看看结果是否正确
    res := addUpper(10) // 1.+ 10 = 55
    if res != 55 {
        fmt.Printf("addUpper错误 返回值=%v 期望值=%v\n", res, 55)
    } else {
        fmt.Printf("addUpper正确 返回值=%v 期望值=%v\n", res, 55)
    }
}
```

15.2.2 传统方法的缺点分析

1) 不方便，我们需要在 main 函数中去调用，这样就需要去修改 main 函数，如果现在项目正在运行，就可能去停止项目。

2) 不利于管理，因为当我们测试多个函数或者多个模块时，都需要写在 main 函数，不利于我们管理和清晰我们思路

3) 引出单元测试。-> testing 测试框架 可以很好解决问题。

15.3 单元测试-基本介绍

Go 语言中自带有一个轻量级的测试框架 testing 和自带的 go test 命令来实现单元测试和性能测试，

testing 框架和其他语言中的测试框架类似，可以基于这个框架写针对相应函数的测试用例，也可以基于该框架写相应的压力测试用例。通过单元测试，可以解决如下问题:

1) 确保**每个函数是可运行，并且运行结果是正确**的

2) 确保写出来的代码**性能是好**的，

3) 单元测试能及时的发现程序设计或实现的**逻辑错误**，使问题及早暴露，便于问题的定位解决，而**性能测试**的重点在于发现程序设计上的一些问题，让程序能够在高并发的情况下还能保持稳定

15.4 单元测试-快速入门

使用 Go 的单元测试，对 addUpper 和 sub 函数进行测试。

特别说明: 测试时，可能需要暂时退出 360。(因为 360 可能会认为生成的测试用例程序是木马)

演示如何进行单元测试:

单元测试的运行原理示意图:

15.4.1 单元测试快速入门总结

1) 测试用例文件名必须以 _test.go 结尾。 比如 cal_test.go , cal 不是固定的。
2) 测试用例函数必须以 Test 开头，一般来说就是 Test+被测试的函数名，比如 TestAddUpper
3) TestAddUpper(t *tesing.T) 的形参类型必须是 *testing.T 【看一下手册】
4) 一个测试用例文件中，可以有多个测试用例函数，比如 TestAddUpper、TestSub
5) 运行测试用例指令

(1) cmd>go test [如果运行正确，无日志，错误时，会输出日志]

(2) cmd>go test -v [运行正确或是错误，都输出日志]

6) 当出现错误时，可以使用 t.Fatalf 来格式化输出错误信息，并退出程序
7) t.Logf 方法可以输出相应的日志
8) 测试用例函数，并没有放在 main 函数中，也执行了，这就是测试用例的方便之处[原理图].
9) PASS 表示测试用例运行成功，FAIL 表示测试用例运行失败
10) 测试单个文件，一定要带上被测试的原文件

    go test -v cal_test.go cal.go

11) 测试单个方法

    go test -v -test.run TestAddUpper

15.5 单元测试-综合案例

单元测试综合案例要求:

1) 编写一个Monster结构体, 字段 Name, Age, Skill
2) 给Monster绑定方法Store, 可以将一个Monster变量(对象),序列化后保存到文件中
3) 给Monster绑定方法ReStore, 可以将一个序列化的Monster,从文件中读取，并反序列化为Monster对象,检查反序列化，名字正确
4) 编程测试用例文件 store_test.go , 编写测试用例函数 TestStore 和 TestRestore进行测试。

代码实现：

monster/monster.go

```
package monster
import (
    "encoding/json"
    "io/ioutil"
    "fmt"
)

type Monster struct {
    Name string
    Age int
    Skill string
}
```

```
//给 Monster 绑定方法 Store，可以将一个 Monster 变量(对象),序列化后保存到文件中

func (this *Monster) Store() bool {

    //先序列化
    data, err := json.Marshal(this)
    if err != nil {
        fmt.Println("marshal err =", err)
        return false
    }

    //保存到文件
    filePath := "d:/monster.ser"
    err = ioutil.WriteFile(filePath, data, 0666)
    if err != nil {
        fmt.Println("write file err =", err)
        return false
    }
    return true
}


//给 Monster 绑定方法 ReStore，可以将一个序列化的 Monster,从文件中读取，
//并反序列化为 Monster 对象,检查反序列化，名字正确
func (this *Monster) ReStore() bool {
```

```
    //1. 先从文件中，读取序列化的字符串
    filePath := "d:/monster.ser"
    data, err := ioutil.ReadFile(filePath)
    if err != nil {
        fmt.Println("ReadFile err =", err)
        return false
    }

    //2.使用读取到 data []byte ,对反序列化
    err = json.Unmarshal(data, this)
    if err != nil {
        fmt.Println("UnMarshal err =", err)
        return false
    }
    return true
}
```

monster/monster_test.go

```
package monster

import (
    "testing"
)
```

```
//测试用例,测试 Store 方法
func TestStore(t *testing.T) {

    //先创建一个 Monster 实例
    monster := &Monster{
        Name : "红孩儿",
        Age :10,
        Skill : "吐火.",
    }
    res := monster.Store()
    if !res {
        t.Fatalf("monster.Store() 错误，希望为=%v 实际为=%v", true, res)
    }
    t.Logf("monster.Store() 测试成功!")
}

func TestReStore(t *testing.T) {

    //测试数据是很多，测试很多次，才确定函数，模块..

    //先创建一个 Monster 实例 ， 不需要指定字段的值
    var monster = &Monster{}
    res := monster.ReStore()
    if !res {
        t.Fatalf("monster.ReStore() 错误，希望为=%v 实际为=%v", true, res)
    }
```

```
    //进一步判断
    if monster.Name != "红孩儿" {
        t.Fatalf("monster.ReStore() 错误，希望为=%v 实际为=%v", "红孩儿", monster.Name)
    }

    t.Logf("monster.ReStore() 测试成功!")
}
```

第 16 章goroutine 和 channel

16.1 goroutine-看一个需求

➢ 需求：要求统计 1-9000000000 的数字中，哪些是素数？

➢ 分析思路：

1) 传统的方法，就是使用一个循环，循环的判断各个数是不是素数。[很慢]

2) 使用并发或者并行的方式，将统计素数的任务分配给多个 goroutine 去完成，这时就会使用到 goroutine.【速度提高 4 倍】

16.2 goroutine-基本介绍

16.2.1 进程和线程介绍

1) 进程就是程序程序在操作系统中的一次执行过程，是系统进行资源分配和调度的基本单位
2) 线程是进程的一个执行实例，是程序执行的最小单元，它是比进程更小的能独立运行的基本单位。
3) 一个进程可以创建核销毁多个线程，同一个进程中的多个线程可以并发执行。
4) 一个程序至少有一个进程，一个进程至少有一个线程

16.2.2 程序、进程和线程的关系示意图

16.2.3 并发和并行

- 并发和并行

1) 多线程程序在单核上运行，就是并发
2) 多线程程序在多核上运行，就是并行
3) 示意图:

- 小结

并发：因为是在一个cpu上，比如有10个线程，每个线程执行10毫秒(进行轮询操作)，从人的角度看，好像这10个线程都在运行，但是从微观上看，在某一个时间点看，其实只有一个线程在执行，这就是并发。

并行：因为是在多个cpu上(比如有10个cpu)，比如有10个线程，每个线程执行10毫秒(各自在不同cpu上执行)，从人的角度看，这10个线程都在运行，但是从微观上看，在某一个时间点看，也同时有10个线程在执行，这就是并行

16.2.4 Go 协程和 Go 主线程

- Go 主线程(有程序员直接称为线程/也可以理解成进程): 一个 Go 线程上，可以起多个协程，你可以这样理解，协程是轻量级的线程[编译器做优化]。

- Go 协程的特点
  1) 有独立的栈空间
  2) 共享程序堆空间
  3) 调度由用户控制
  4) 协程是轻量级的线程

- 一个示意图

16.3 goroutine-快速入门

16.3.1 案例说明

- 请编写一个程序，完成如下功能:
  1) 在主线程(可以理解成进程)中，开启一个 goroutine，该协程每隔 1 秒输出 "hello,world"
  2) 在主线程中也每隔一秒输出"hello,golang"，输出 10 次后，退出程序
  3) 要求主线程和 goroutine 同时执行.
  4) 画出**主线程和协程执行流程图**

➢ 代码实现

```go
//编写一个函数，每隔1秒输出 "hello,world"
func test() {
    for i := 1; i <= 10; i++ {
        fmt.Println("tesst () hello,world " + strconv.Itoa(i))
        time.Sleep(time.Second)
    }
}

func main() {

    go test() // 开启了一个协程

    for i := 1; i <= 10; i++ {
        fmt.Println(" main() hello,golang" + strconv.Itoa(i))
        time.Sleep(time.Second)
    }
}
```

输出的效果说明， main 这个主线程和 test 协程同时执行.

```
D:\goproject\src\go_code\chapter16\goroutinedemo>
 main() hello,golang1
tesst () hello,world 1
 main() hello,golang2
tesst () hello,world 2
 main() hello,golang3
tesst () hello,world 3
tesst () hello,world 4
 main() hello,golang4
tesst () hello,world 5
 main() hello,golang5
tesst () hello,world 6
 main() hello,golang6
tesst () hello,world 7
 main() hello,golang7
tesst () hello,world 8
 main() hello,golang8
```

➢ 主线程和协程执行流程图

16.3.2 快速入门小结

1) 主线程是一个物理线程，直接作用在 cpu 上的。是重量级的，非常耗费 cpu 资源。

2) 协程从主线程开启的，是轻量级的线程，是逻辑态。对资源消耗相对小。

3) Golang 的协程机制是重要的特点，可以轻松的**开启上万个协程**。其它编程语言的并发机制是一般基于线程的，开启过多的线程，资源耗费大，这里就突显 Golang 在并发上的优势了

16.4 goroutine 的调度模型

16.4.1 MPG 模式基本介绍

1) M: 操作系统的主线程 (是物理线程)
2) P: 协程执行需要的上下文
3) G: 协程

16.4.2 MPG 模式运行的状态 1

1) 当前程序有三个M, 如果三个M都在一个cpu运行，就是并发，如果在不同的cpu运行就是并行
2) M1,M2,M3正在执行一个G，M1的协程队列有三个，M2的协程队列有3个， M3协程队列有2个
3) 从上图可以看到: Go的协程是轻量级的线程，是逻辑态的，Go可以容易的起上万个协程。
4) 其它程序c/java的多线程，往往是内核态的，比较重量级，几千个线程可能耗光cpu

16.4.3 MPG 模式运行的状态 2

1) 分成两个部分来看
2) 原来的情况是 M0 主线程正在执行G0协程，另外有三个协程在队列等待
3) 如果G0协程阻塞，比如读取文件或者数据库等
4) 这时就会**创建**M1主线程(**也可能**是从已有的线程池中**取出M1**)，并且将等待的3个协程挂到M1下开始执行， M0的主线程下的G0仍然执行文件io的读写。
5) 这样的MPG调度模式，可以既让G0执行，同时也不会让队列的其它协程一直阻塞，仍然可以并发/并行执行。
6) 等到G0不阻塞了，M0会被放到空闲的主线程继续执行(从已有的线程池中取)，同时G0又会被唤醒。

16.5 设置 Golang 运行的 cpu 数

➢ 介绍：为了充分了利用多 cpu 的优势，在 Golang 程序中，设置运行的 cpu 数目

```go
package main

import "fmt"
import "runtime"

func main() {
    //获取当前系统cpu的数量
    num := runtime.NumCPU()
    //我这里设置num-1的cpu运行go程序
    runtime.GOMAXPROCS(num)
    fmt.Println("num=", num)
}
```

1) go1.8后，默认让程序运行在多个核上,可以不用设置了
2) go1.8前，还是要设置一下，可以更高效的利益cpu

16.6 channel(管道)-看个需求

需求：现在要计算 1-200 的各个数的阶乘，并且把各个数的阶乘放入到 map 中。最后显示出来。要求使用 goroutine 完成

- 分析思路：
  1) 使用 goroutine 来完成，效率高，但是会出现并发/并行安全问题.
  2) 这里就提出了不同 goroutine 如何通信的问题

- 代码实现
  1) 使用 goroutine 来完成(看看使用 gorotine 并发完成会出现什么问题？然后我们会去解决)
  2) 在运行某个程序时，如何知道是否存在资源竞争问题。 方法很简单，在编译该程序时，增加一个参数 -race 即可 [示意图]
  3) 代码实现：

```
package main
import (
    "fmt"
    "time"
)

// 需求：现在要计算 1-200 的各个数的阶乘，并且把各个数的阶乘放入到 map 中。
// 最后显示出来。要求使用 goroutine 完成

// 思路
// 1. 编写一个函数，来计算各个数的阶乘，并放入到 map 中.
// 2. 我们启动的协程多个，统计的将结果放入到 map 中
```

```
// 3. map 应该做出一个全局的.

var (
    myMap = make(map[int]int, 10)
)

// test 函数就是计算 n!, 让将这个结果放入到 myMap
func test(n int) {

    res := 1
    for i := 1; i <= n; i++ {
        res *= i
    }

    //这里我们将 res 放入到 myMap
    myMap[n] = res //concurrent map writes?
}

func main() {

    // 我们这里开启多个协程完成这个任务[200 个]
    for i := 1; i <= 200; i++ {
        go test(i)
    }
```

```
    //休眠 10 秒钟【第二个问题 】
    time.Sleep(time.Second * 10)

    //这里我们输出结果,变量这个结果
    for i, v := range myMap {
        fmt.Printf("map[%d]=%d\n", i, v)
    }


}
```

4) 示意图：

16.6.1 不同 goroutine 之间如何通讯

1) 全局变量的互斥锁

2) 使用管道 channel 来解决

16.6.2 使用全局变量加锁同步改进程序

- 因为没有对全局变量 m 加锁，因此会出现资源争夺问题，代码会出现错误，提示 concurrent map writes
- 解决方案：加入互斥锁
- 我们的数的阶乘很大，结果会越界,可以将求阶乘改成 sum += uint64(i)
- 代码改进

```go
var (
    myMap = make(map[int]int, 10)
    //声明一个全局的互斥锁
    //lock 是一个全局的互斥锁,
    //sync 是包: synchornized 同步
    //Mutex : 是互斥
    lock sync.Mutex
)
```

```go
// test 函数就是计算 n!, 让将这个结果放入到 myMap
func test(n int) {

    res := 1
    for i := 1; i <= n; i++ {
        res *= i
    }

    //这里我们将 res 放入到myMap
    //加锁
    lock.Lock()
    myMap[n] = res //concurrent map writes?
    //解锁
    lock.Unlock()
}
```

```go
//这里我们输出结果,变量这个结果
lock.Lock()
for i, v := range myMap {
    fmt.Printf("map[%d]=%d\n", i, v)
}
lock.Unlock()
```

16.6.3 为什么需要 channel

1) 前面使用全局变量加锁同步来解决 goroutine 的通讯，但不完美

2) 主线程在等待所有 goroutine 全部完成的时间很难确定，我们这里设置 10 秒，仅仅是估算。

3) 如果主线程休眠时间长了，会加长等待时间，如果等待时间短了，可能还有 goroutine 处于工作状态，这时也会随主线程的退出而销毁

4) 通过全局变量加锁同步来实现通讯，也并不利用多个协程对全局变量的读写操作。

5) 上面种种分析都在呼唤一个新的通讯机制-channel

16.6.4 channel 的基本介绍

1) channle 本质就是一个数据结构-队列【示意图】

2) 数据是先进先出【FIFO : first in first out】

3) 线程安全，多 goroutine 访问时，不需要加锁，就是说 channel 本身就是线程安全的

4) channel 有类型的，一个 string 的 channel 只能存放 string 类型数据。

5) 示意图：

16.6.5 定义/声明 channel

- var 变量名 chan 数据类型
- 举例：

  var intChan chan int (intChan 用于存放 int 数据)

  var mapChan chan map[int]string (mapChan 用于存放 map[int]string 类型)

  var perChan chan Person

  var perChan2 chan *Person

  ...
- 说明

  channel 是引用类型

  channel 必须初始化才能写入数据，即 make 后才能使用

  管道是有类型的，intChan 只能写入 整数 int

16.6.6 管道的初始化，写入数据到管道，从管道读取数据及基本的注意事项

```
package main
import (
    "fmt"
)

func main() {

    //演示一下管道的使用
    //1. 创建一个可以存放 3 个 int 类型的管道
    var intChan chan int
    intChan = make(chan int, 3)

    //2. 看看 intChan 是什么
    fmt.Printf("intChan 的值=%v intChan 本身的地址=%p\n", intChan, &intChan)


    //3. 向管道写入数据
    intChan<- 10
    num := 211
    intChan<- num
    intChan<- 50
    // intChan<- 98//注意点，当我们给管写入数据时，不能超过其容量


    //4. 看看管道的长度和 cap(容量)
    fmt.Printf("channel len= %v cap=%v \n", len(intChan), cap(intChan)) // 3, 3
```

```
    //5. 从管道中读取数据

    var num2 int
    num2 = <-intChan
    fmt.Println("num2=", num2)
    fmt.Printf("channel len= %v cap=%v \n", len(intChan), cap(intChan))    // 2, 3

    //6. 在没有使用协程的情况下，如果我们的管道数据已经全部取出，再取就会报告 deadlock

    num3 := <-intChan
    num4 := <-intChan
    num5 := <-intChan

    fmt.Println("num3=", num3, "num4=", num4, "num5=", num5)

}
```

16.6.7 channel 使用的注意事项

1) channel 中只能存放指定的数据类型
2) channle 的数据放满后，就不能再放入了
3) 如果从 channel 取出数据后，可以继续放入
4) 在没有使用协程的情况下，如果 channel 数据取完了，再取，就会报 dead lock

16.6.8 读写 channel 案例演示

1) 创建一个intChan, 最多可以存放3个int，演示存3数据到intChan, 然后再取出这三个int

```go
func main() {

    var  intChan  chan  int
    intChan = make(chan int, 3)
    intChan <- 10
    intChan <- 20
    intChan <- 10
    //因为 intChan 的容量为3 ,再存放会报告deadlock
    //intChan <- 50
    num1 := <- intChan
    num2 := <- intChan
    num3 := <- intChan
    //因为 intChan 这时已经没有数据了，再取就会报告deadlock
    //num4 := <- intChan
    fmt.Printf("num1=%v num2=%v num3=%v", num1, num2, num3)
}
```

2) 创建一个mapChan, 最多可以存放10个map[string]string的key-val，演示写入和读取。

```go
func main() {

    var  mapChan  chan  map[string]string
    mapChan = make(chan map[string]string, 10)
    m1 := make(map[string]string, 20)
    m1["city1"] = "北京"
    m1["city2"] = "天津"

    m2 := make(map[string]string, 20)
    m2["hero1"] = "宋江"
    m2["hero2"] = "武松"
    //...
    mapChan <- m1
    mapChan <- m2
```

3) 创建一个catChan, 最多可以存放10个Cat结构体变量，演示写入和读取的用法

```go
func main() {

    var  catChan  chan  Cat
    catChan = make(chan Cat, 10)

    cat1 := Cat{Name:"tom", Age:18,}
    cat2 := Cat{Name:"tom~", Age:180,}
    catChan <- cat1
    catChan <- cat2

    //取出
    cat11 := <- catChan
    cat22 := <- catChan

    fmt.Println(cat11, cat22)
}
```

4) 创建一个catChan2, 最多可以存放10个*Cat 变量，演示写入和读取的用法

```go
func main() {

    var  catChan  chan  *Cat
    catChan = make(chan *Cat, 10)

    cat1 := Cat{Name:"tom", Age:18,}
    cat2 := Cat{Name:"tom~", Age:180,}
    catChan <- &cat1
    catChan <- &cat2

    //取出
    cat11 := <- catChan
    cat22 := <- catChan

    fmt.Println(cat11, cat22)
}
```

5) 创建一个allChan, 最多可以存放10个 **任意数据类型变量**, 演示写入和读取的用法

```go
func main() {

    var  allChan  chan  interface{}
    allChan = make(chan interface{}, 10)

    cat1 := Cat{Name:"tom", Age:18,}
    cat2 := Cat{Name:"tom~", Age:180,}
    allChan <- cat1
    allChan <- cat2
    allChan <- 10
    allChan <- "jack"
    //取出
    cat11 := <- allChan
    cat22 := <- allChan
    v1 := <- allChan
    v2 := <- allChan
    fmt.Println(cat11, cat22, v1, v2)
}
```

6) 看下面的代码, 会输出什么?

```go
func main() {

    var  allChan  chan  interface{}
    allChan = make(chan interface{}, 10)

    cat1 := Cat{Name:"tom", Age:18,}
    cat2 := Cat{Name:"tom~", Age:180,}
    allChan <- cat1
    allChan <- cat2
    allChan <- 10
    allChan <- "jack"
    //取出
    cat11 := <- allChan
    fmt.Println(cat11.Name)
}
```

```
newCat := <-allChan //从管道中取出的Cat是什么?

fmt.Printf("newCat=%T , newCat=%v\n", newCat, newCat)
//下面的写法是错误的!编译不通过
//fmt.Printf("newCat.Name=%v", newCat.Name)
//使用类型断言
a := newCat.(Cat)
fmt.Printf("newCat.Name=%v", a.Name)
```

16.7 管道的课后练习题

读写channel课堂练习/课后练习

说明：请完成如下案例

1) 创建一个 Person 结构体 [Name, Age, Address]
2) 使用rand方法配合随机创建10个Person 实例，并放入到channel中.
3) 遍历channel ，将各个Person实例的信息显示在终端...

16.8 channel 的遍历和关闭

16.8.1 channel 的关闭

使用内置函数 close 可以关闭 channel，当 channel 关闭后，就不能再向 channel 写数据了，但是仍然可以从该 channel 读取数据

案例演示：

```go
package main
import (
    "fmt"
)

func main() {

    intChan := make(chan int, 3)
    intChan<- 100
    intChan<- 200
    close(intChan) // close
    //这是不能够再写入数到channel
    //intChan<- 300
    fmt.Println("okook~")
    //当管道关闭后，读取数据是可以的
    n1 := <-intChan
    fmt.Println("n1=", n1)
}
```

16.8.2 channel 的遍历

channel 支持 for--range 的方式进行遍历，请注意两个细节

1) 在遍历时，如果 channel 没有关闭，则回出现 deadlock 的错误

2) 在遍历时，如果 channel 已经关闭，则会正常遍历数据，遍历完后，就会退出遍历。

16.8.3 channel 遍历和关闭的案例演示

看代码演示：

```go
//遍历管道
intChan2 := make(chan int, 100)
for i := 0; i < 100; i++ {
    intChan2<- i * 2  //放入100个数据到管道
}

//遍历管道不能使用普通的 for 循环
// for i := 0; i < len(intChan2); i++ {

// }
//在遍历时，如果channel没有关闭，则会出现deadlock的错误
//在遍历时，如果channel已经关闭，则会正常遍历数据，遍历完后，就会退出遍历
close(intChan2)
for v := range intChan2 {
    fmt.Println("v=", v)
}
```

16.8.4 应用实例 1

请完成goroutine和channel协同工作的案例，具体要求：

1) 开启一个writeData协程，向管道intChan中写入50个整数.
2) 开启一个readData协程，从管道intChan中读取writeData写入的数据。
3) 注意: writeData和readDate操作的是同一个管道
4) 主线程需要等待writeData和readDate协程都完成工作才能退出【管道】

➢ 思路分析：

➢ 代码的实现:

```
package main
import (
    "fmt"
    _ "time"
)



//write Data
func writeData(intChan chan int) {
    for i := 1; i <= 50; i++ {
        //放入数据
        intChan<- i
        fmt.Println("writeData ", i)
```

```
        //time.Sleep(time.Second)
    }
    close(intChan) //关闭
}

//read data
func readData(intChan chan int, exitChan chan bool) {

    for {
        v, ok := <-intChan
        if !ok {
            break
        }
        //time.Sleep(time.Second)
        fmt.Printf("readData 读到数据=%v\n", v)
    }
    //readData 读取完数据后，即任务完成
    exitChan<- true
    close(exitChan)

}

func main() {

    //创建两个管道
    intChan := make(chan int, 50)
```

```
    exitChan := make(chan bool, 1)


    go writeData(intChan)
    go readData(intChan, exitChan)


    //time.Sleep(time.Second * 10)
    for {
        _, ok := <-exitChan
        if !ok {
            break
        }
    }


}
```

16.8.5 应用实例 2-阻塞

```
func main() {
    intChan := make(chan int, 10) // 10->50 的话数据一下就放入了
    exitChan := make(chan bool, 1)
    go writeData(intChan)
    //go readData(intChan, exitChan)

    //就是为了等待..readData 协程 完成
    for _ = range exitChan {
        fmt.Println("ok...")
    }
}
```

如果，编译器(运行)，发现一个管道只有写，而没有读，则该管道，会阻塞。

写管道和读管道的频率不一致，无所谓。

问题：如果注销掉 go readData(intChan, exitChan)，程序会怎么样?

答：如果只是向管道写入数据，**而没有读取**，就会出现阻塞而dead lock，原因是intChan容量是10, 而代码writeData会写入 50个数据,因此会阻塞在 writeData的 **ch <- i**

16.8.6 应用实例 3

➢ 需求：

要求统计 1-200000 的数字中，哪些是素数？这个问题在本章开篇就提出了，现在我们有 goroutine 和 channel 的知识后，就可以完成了 [测试数据: 80000]

➢ 分析思路：

传统的方法，就是使用一个循环，循环的判断各个数是不是素数【ok】。

使用并发/并行的方式，将统计素数的任务分配给多个(4 个)goroutine 去完成，完成任务时间短。

➢ **画出分析思路**

➢ 代码实现

```
package main
import (
    "fmt"
    "time"
)



//向 intChan 放入 1-8000 个数
func putNum(intChan chan int) {

    for i := 1; i <= 8000; i++ {
        intChan<- i
    }

    //关闭 intChan
    close(intChan)
}

// 从 intChan 取出数据，并判断是否为素数,如果是，就
//   //放入到 primeChan
func primeNum(intChan chan int, primeChan chan int, exitChan chan bool) {

    //使用 for 循环
    // var num int
    var flag bool //
```

```
for {
    time.Sleep(time.Millisecond * 10)
    num, ok := <-intChan

    if !ok { //intChan 取不到..
        break
    }
    flag = true //假设是素数
    //判断 num 是不是素数
    for i := 2; i < num; i++ {
        if num % i == 0 {//说明该 num 不是素数
            flag = false
            break
        }
    }

    if flag {
        //将这个数就放入到 primeChan
        primeChan<- num
    }
}

fmt.Println("有一个 primeNum 协程因为取不到数据，退出")
//这里我们还不能关闭 primeChan
//向 exitChan 写入 true
exitChan<- true
```

```
}

func main() {

    intChan := make(chan int , 1000)
    primeChan := make(chan int, 2000)//放入结果
    //标识退出的管道
    exitChan := make(chan bool, 4) // 4 个

    //开启一个协程，向 intChan 放入 1-8000 个数
    go putNum(intChan)
    //开启 4 个协程，从 intChan 取出数据，并判断是否为素数,如果是，就
    //放入到 primeChan
    for i := 0; i < 4; i++ {
        go primeNum(intChan, primeChan, exitChan)
    }

    //这里我们主线程，进行处理
    //直接
    go func(){
        for i := 0; i < 4; i++ {
            <-exitChan
        }
        //当我们从 exitChan 取出了 4 个结果，就可以放心的关闭 prprimeChan
        close(primeChan)
```

```
    }()


    //遍历我们的 primeChan ,把结果取出
    for {
        res, ok := <-primeChan
        if !ok{
            break
        }
        //将结果输出
        fmt.Printf("素数=%d\n", res)
    }

    fmt.Println("main 线程退出")



}
```

结论：使用 go 协程后，执行的速度，比普通方法提高至少 4 倍

16.9 channel 使用细节和注意事项

1) channel 可以声明为只读，或者只写性质【案例演示】

```go
package main
import (
    "fmt"
)

func main() {
    //管道可以声明为只读或者只写

    //1. 在默认情况下下，管道是双向
    //var chan1 chan int //可读可写

    //2 声明为只写
    var chan2 chan<- int
    chan2 = make(chan int, 3)
    chan2<- 20
    //num := <-chan2 //error

    fmt.Println("chan2=", chan2)

    //3. 声明为只读
    var chan3 <-chan int
    num2 := <-chan3
    //chan3<- 30 //err
    fmt.Println("num2", num2)

}
```

2) channel 只读和只写的最佳实践案例

```go
func main() {
    var ch chan int
    ch = make(chan int, 10)
    exitChan := make(chan struct{}, 2)
    go send(ch, exitChan)
    go recv(ch, exitChan)

    var total = 0
    for _ = range exitChan {
        total++
        if total == 2 {
            break
        }
    }
    fmt.Println("结束....")
}
```

```go
// ch chan<- int ,这样ch就只能写操作了
func send(ch chan<- int, exitChan chan struct{}) {
    for i := 0; i < 10; i++ {
        ch <- i
    }
    close(ch)
    var a struct{}
    exitChan <- a
}
```

```go
// ch <-chan int ,这样ch就只能读操作了
func recv(ch <-chan int, exitChan chan struct{}) {
    for {
        v, ok := <-ch
        if !ok {
            break
        }
        fmt.Println(v)
    }
    var a struct{}
    exitChan <- a
}
```

3) 使用 select 可以解决从管道取数据的阻塞问题

```go
package main
import (
    "fmt"
    "time"
)

func main() {

    //使用 select 可以解决从管道取数据的阻塞问题

    //1.定义一个管道 10 个数据 int
    intChan := make(chan int, 10)
    for i := 0; i < 10; i++ {
        intChan<- i
    }
    //2.定义一个管道 5 个数据 string
    stringChan := make(chan string, 5)
```

```
for i := 0; i < 5; i++ {
    stringChan <- "hello" + fmt.Sprintf("%d", i)
}

//传统的方法在遍历管道时，如果不关闭会阻塞而导致 deadlock

//问题，在实际开发中，可能我们不好确定什么关闭该管道.
//可以使用 select 方式可以解决
//label:
for {
    select {
        //注意: 这里，如果 intChan 一直没有关闭，不会一直阻塞而 deadlock
        //，会自动到下一个 case 匹配
        case v := <-intChan :
            fmt.Printf("从 intChan 读取的数据%d\n", v)
            time.Sleep(time.Second)
        case v := <-stringChan :
            fmt.Printf("从 stringChan 读取的数据%s\n", v)
            time.Sleep(time.Second)
        default :
            fmt.Printf("都取不到了，不玩了，程序员可以加入逻辑\n")
            time.Sleep(time.Second)
            return
            //break label
    }
}
```

```
}
```

4) goroutine 中使用 recover，解决协程中出现 panic，导致程序崩溃问题

说明: 如果我们起了一个协程，但是这个协程出现了panic，如果我们没有捕获这个panic,就会造成整个程序崩溃，这时我们可以在goroutine中使用recover来捕获panic，进行处理，这样即使这个协程发生的问题，但是主线程仍然不受影响，可以继续执行。

var map1 map[string]string
map1["no1"] = "tom"

代码实现：

```
package main
import (
    "fmt"
    "time"
)

//函数
func sayHello() {
    for i := 0; i < 10; i++ {
        time.Sleep(time.Second)
        fmt.Println("hello,world")
    }
}
//函数
func test() {
    //这里我们可以使用 defer + recover
    defer func() {
        //捕获 test 抛出的 panic
```

```
        if err := recover(); err != nil {
            fmt.Println("test()  发生错误", err)
        }
    }()
    //定义了一个 map
    var myMap map[int]string
    myMap[0] = "golang" //error
}

func main() {

    go sayHello()
    go test()



    for i := 0; i < 10; i++ {
        fmt.Println("main() ok=", i)
        time.Sleep(time.Second)
    }

}
```

第 17 章反射

17.1 先看一个问题，反射的使用场景

```go
package main
import (
    "fmt"
    "encoding/json"
)
type Monster struct {
    Name  string `json:"monsterName"`
    Age   int    `json:"monsterAge"`
    Sal   float64 `json:"monsterSal"`
    Sex   string `json:"monsterSex"`
}
func main() {
    m := Monster{
        Name : "玉兔精",
        Age : 20,
        Sal : 888.99,
        Sex : "female",
    }
    data, _ := json.Marshal(m)
    fmt.Println("json result:", string(data))
}
```

输出结果:
json result: {"monsterName":"玉兔精","monsterAge":20,"monsterSal":888.99,"monsterSex":"female"}

思考问题:
为什么序列化后，key-val 的key值是 结构体 Tag的值，而不是字段的名称，比如：不是Name 而是："monsterName":"玉兔精"

引出反射： (note:学习reflect后，回头来解决)

17.2 使用反射机制，编写函数的适配器，桥连接

要求如下：

1) 定义了两个匿名函数

```go
test1 := func(v1 int, v2 int) {
        t.Log(v1, v2)
}
test2 := func(v1 int, v2 int, s string) {
        t.Log(v1, v2, s)
}
```

2) 定义一个适配器函数用作统一处理接口，其大致结构如下：

```go
bridge := func(call interface{}, args ...interface{}) {
 //内容
}
//实现调用test1对应的函数
bridge(test1, 1, 2)
//实现调用test2对应的函数
bridge(test2, 1, 2, "test2")
```

3) 要求使用反射机制完成(note:学习reflect后，回头来解决)

17.3 反射的基本介绍

17.3.1 基本介绍

1) 反射可以在运行时**动态获取变量的各种信息**，比如变量的类型(type)，类别(kind)
2) 如果是结构体变量，还可以获取到结构体本身的信息(包括结构体的**字段**、**方法**)
3) 通过反射，可以修改变量的值，可以调用关联的方法。
4) 使用反射，需要 import (“**reflect**”)
5) 示意图

17.3.2 反射的应用场景

反射常见应用场景有以下两种

1) 不知道接口调用哪个函数，根据传入参数在运行时确定调用的具体接口，这种需要对函数或方法反射。例如以下这种桥接模式，比如我前面提出问题。

```
func bridge(funcPtr interface{}, args ...interface{})
```

第一个参数funcPtr以接口的形式传入函数指针，函数参数args以可变参数的形式传入，bridge函数中可以用反射来动态执行funcPtr函数

2) 对结构体序列化时，如果结构体有指定Tag，也会使用到反射生成对应的字符串。

```go
package main
import (
    "fmt"
    "encoding/json"
)
type Monster struct {
    Name  string `json:"monsterName"`
    Age   int    `json:"monsterAge"`
    Sal   float64 `json:"monsterSal"`
    Sex   string `json:"monsterSex"`
}
func main() {
    m := Monster{
        Name : "玉兔精",
        Age : 20,
        Sal : 888.99,
        Sex : "female",
    }
    data, _ := json.Marshal(m)
    fmt.Println("json result:", string(data))
}
```

17.3.3 反射重要的函数和概念

1) reflect.TypeOf(变量名)，获取变量的类型，返回reflect.Type类型
2) reflect.ValueOf(变量名)，获取变量的值，返回reflect.Value类型reflect.Value 是一个结构体类型。【看文档】，通过reflect.Value，可以获取到关于该变量的很多信息。

type Value

```
type Value struct {
    // 内含隐藏或非导出字段
}
```

Value为go值提供了反射接口。

- func (v Value) IsValid() bool
- func (v Value) IsNil() bool
- func (v Value) Kind() Kind
- func (v Value) Type() Type
- func (v Value) Convert(t Type) Value
- func (v Value) Elem() Value
- func (v Value) Bool() bool
- func (v Value) Int() int64
- func (v Value) OverflowInt(x int64) bool
- func (v Value) Uint() uint64
- func (v Value) OverflowUint(x uint64) bool
- func (v Value) Float() float64
- func (v Value) OverflowFloat(x float64) bool
- func (v Value) Complex() complex128

3) 变量、interface{} 和 reflect.Value 是可以相互转换的，这点在实际开发中，会经常使用到。画出示意图

变量、interface{}和reflect.Value是可以相互转换的
，在使用反射的过程中，通常的方式是

```
var student Stu

var num int
```

```
//专门用于做反射
func test(b interface{}) {

    //1. 如何将 Interface{} 转成 reflect.Value
    rVal := reflect.ValueOf(b)

    //2. 如何将 reflect.Value -> interface{}
    iVal := rVal.Interface()
    //3. 如何将 inteface{} 转成 原来的变量类型,使用类型断言
    v := iVal.(Stu)
}
```

17.4 反射的快速入门

17.4.1 快速入门说明

- 请编写一个案例，演示对(基本数据类型、interface{}、reflect.Value)进行反射的基本操作

  代码演示，见下面的表格：
- 请编写一个案例，演示对(结构体类型、interface{}、reflect.Value)进行反射的基本操作

  代码演示：

```
package main
import (
    "reflect"
    "fmt"
)


//专门演示反射
func reflectTest01(b interface{}) {

    //通过反射获取的传入的变量的  type , kind,  值
```

```
    //1. 先获取到 reflect.Type
    rTyp := reflect.TypeOf(b)
    fmt.Println("rType=", rTyp)

    //2. 获取到 reflect.Value
    rVal := reflect.ValueOf(b)

    n2 := 2 + rVal.Int()
    fmt.Println("n2=", n2)

    fmt.Printf("rVal=%v rVal type=%T\n", rVal, rVal)

    //下面我们将 rVal 转成 interface{}
    iV := rVal.Interface()
    //将 interface{} 通过断言转成需要的类型
    num2 := iV.(int)
    fmt.Println("num2=", num2)


}

//专门演示反射[对结构体的反射]
func reflectTest02(b interface{}) {

    //通过反射获取的传入的变量的 type , kind, 值
    //1. 先获取到 reflect.Type
```

```
    rTyp := reflect.TypeOf(b)
    fmt.Println("rType=", rTyp)

    //2. 获取到 reflect.Value
    rVal := reflect.ValueOf(b)




    //下面我们将 rVal 转成 interface{}
    iV := rVal.Interface()
    fmt.Printf("iv=%v iv type=%T \n", iV, iV)
    //将 interface{} 通过断言转成需要的类型
    //这里，我们就简单使用了一带检测的类型断言.
    //同学们可以使用 swtich 的断言形式来做的更加的灵活
    stu, ok := iV.(Student)
    if ok {
        fmt.Printf("stu.Name=%v\n", stu.Name)
    }

}

type Student struct {
    Name string
    Age int
}
```

```
type Monster struct {
    Name string
    Age int
}

func main() {

    //请编写一个案例，
    //演示对(基本数据类型、interface{}、reflect.Value)进行反射的基本操作

    //1. 先定义一个 int
    // var num int = 100
    // reflectTest01(num)

    //2. 定义一个 Student 的实例
    stu := Student{
        Name : "tom",
        Age : 20,
    }
    reflectTest02(stu)


}
```

17.5 反射的注意事项和细节

1) reflect.Value.Kind，获取变量的类别，返回的是一个常量

type Kind

```
type Kind uint
```

Kind代表Type类型值表示的具体分类。零值表示非法分类。

```
const (
    Invalid Kind = iota
    Bool
    Int
    Int8
    Int16
    Int32
    Int64
    Uint
    Uint8
    Uint16
    Uint32
    Uint64
    Uintptr
    Float32
    Float64
    Complex64
    Complex128
```

2) Type 和 Kind 的区别

Type 是类型, Kind 是类别， Type 和 Kind **可能是相同的**，也**可能是不同的.**

比如: var num int = 10 num 的 Type 是 int , Kind 也是 int

比如: var stu Student stu 的 Type 是 pkg1.Student , Kind 是 struct

3) 通过反射可以在让变量在 interface{} 和Reflect.Value 之间相互转换, 这点在前面画过示意图并在快速入门案例中讲解过, 这里我们看下是**如何在代码中体现**的.

变量<------->interface{}<------->reflect.Value

```
//将reflect.Value 转成 interface()
iv := v.Interface()
//下面的printf 不能使用 iv.Name
fmt.Printf("iv=%v iv=%T \n", iv, iv)
//将iv 转成Student, 使用了类型断言
stu, ok := iv.(Student)
if ok {
        fmt.Printf("%v %T stu.Name=%v\n", stu, stu, stu.Name)
}
```

4) 使用反射的方式来获取变量的值(并返回对应的类型)，要求数据类型匹配，比如x是int, 那么就应该使用 reflect.Value(x).Int(), 而不能使用其它的，否则报panic

func (Value) Int

```
func (v Value) Int() int64
```

返回v持有的有符号整数（表示为int64），如果v的Kind不是Int、Int8、Int16、Int32、Int64会panic

```
func testInt(b interface{}) {
val := reflect.ValueOf(b)
fmt.Printf("v=%v\n", val.Int())
fmt.Printf("v=%v\n", val.Float())//
}
```

5) 通过反射的来修改变量, 注意当使用 SetXxx 方法来设置需要通过对应的指针类型来完成, 这样

才能改变传入的变量的值，同时需要使用到 reflect.Value.Elem()方

```
func testInt(b interface{}) {
    val := reflect.ValueOf(b)
    fmt.Printf("val type=%T\n", val)
    val.Elem().SetInt(110)
    fmt.Printf("val=%v\n", val)
}
func main() {
    var num int = 20
    testInt(&num)
    fmt.Println("num=", num)
}
```

6) reflect.Value.Elem() 应该如何理解？

```
func main() {
    var num int = 100
    fn := reflect.ValueOf(&num)
    fn.Elem().SetInt(200)
    fmt.Printf("%v\n", num)
}
```

```
//fn.Elem() 用于获取指针指向变量，类似
var num = 10
var b *int = &num
*b = 3
```

17.6 反射课堂练习

1) 给你一个变量 var v float64 = 1.2，请使用反射来得到它的 reflect.Value，然后获取对应的 Type，Kind 和值，并将 reflect.Value 转换成 interface{}，再将 interface{} 转换成 float64. [不说：]

2) 看段代码，判断是否正确，为什么

```
package main
import (
    "fmt"
    "reflect"
)
func main() {
    var str string = "tom"    //ok
```

```
    fs := reflect.ValueOf(str) //ok fs -> string
    fs.SetString("jack") //error
    fmt.Printf("%v\n", str)
}
```

修改如下：

```
package main
import (
    "fmt"
    "reflect"
)
func main() {
    var str string = "tom"   //ok
    fs := reflect.ValueOf(&str) //ok fs -> string
    fs.Elem().SetString("jack") //ok
    fmt.Printf("%v\n", str) // jack
}
```

ok!

17.7 反射最佳实践

1) 使用**反射来遍历结构体的字段，调用结构体的方法，**并**获取结构体标签**的值

```
package main
import (
    "fmt"
    "reflect"
)
```

```
//定义了一个 Monster 结构体
type Monster struct {
    Name    string `json:"name"`
    Age     int `json:"monster_age"`
    Score float32 `json:"成绩"`
    Sex     string

}

//方法，返回两个数的和
func (s Monster) GetSum(n1, n2 int) int {
    return n1 + n2
}
//方法，  接收四个值，给 s 赋值
func (s Monster) Set(name string, age int, score float32, sex string) {
    s.Name = name
    s.Age = age
    s.Score = score
    s.Sex = sex
}

//方法，显示 s 的值
func (s Monster) Print() {
    fmt.Println("---start~-----")
    fmt.Println(s)
    fmt.Println("---end~-----")
```

```
}
func TestStruct(a interface{}) {
    //获取 reflect.Type 类型
    typ := reflect.TypeOf(a)
    //获取 reflect.Value 类型
    val := reflect.ValueOf(a)
    //获取到 a 对应的类别
    kd := val.Kind()
    //如果传入的不是 struct，就退出
    if kd !=  reflect.Struct {
        fmt.Println("expect struct")
        return
    }

    //获取到该结构体有几个字段
    num := val.NumField()

    fmt.Printf("struct has %d fields\n", num) //4
    //变量结构体的所有字段
    for i := 0; i < num; i++ {
        fmt.Printf("Field %d:  值为=%v\n", i, val.Field(i))
        //获取到 struct 标签，注意需要通过 reflect.Type 来获取 tag 标签的值
        tagVal := typ.Field(i).Tag.Get("json")
        //如果该字段于 tag 标签就显示，否则就不显示
        if tagVal != "" {
            fmt.Printf("Field %d: tag 为=%v\n", i, tagVal)
```

```
        }
    }

    //获取到该结构体有多少个方法
    numOfMethod := val.NumMethod()
    fmt.Printf("struct has %d methods\n", numOfMethod)

    //var params []reflect.Value
    //方法的排序默认是按照 函数名的排序（ASCII 码）
    val.Method(1).Call(nil) //获取到第二个方法。调用它


    //调用结构体的第 1 个方法 Method(0)
    var params []reflect.Value   //声明了 []reflect.Value
    params = append(params, reflect.ValueOf(10))
    params = append(params, reflect.ValueOf(40))
    res := val.Method(0).Call(params) //传入的参数是 []reflect.Value, 返回[]reflect.Value
    fmt.Println("res=", res[0].Int()) //返回结果, 返回的结果是 []reflect.Value*/

}
func main() {
    //创建了一个 Monster 实例
    var a Monster = Monster{
        Name:   "黄鼠狼精",
        Age:    400,
        Score: 30.8,
```

```
    }
    //将 Monster 实例传递给 TestStruct 函数
    TestStruct(a)
}
```

2) 使用反射的方式来获取结构体的 tag 标签，遍历字段的值，修改字段值，调用结构体方法(要求：通过传递地址的方式完成，在前面案例上修改即可)

3) 定义了两个函数 test1 和 test2，定义一个适配器函数用作统一处理接口【了解】

4) 使用反射操作任意结构体类型：【了解】

5) 使用反射创建并操作结构体

17.8 课后作业

课堂练习

要求

1) 编写一个Cal 结构体，有 两个字段 Num1, 和Num2 。
2) 方法 GetSub(name string)
3) 使用反射遍历Cal结构体所有的字段信息.
4) 使用反射机制完成对GetSub 的调用，输出形式为 "tom 完成了减法运行，8 - 3 = 5"

第 18 章tcp 编程

18.1 看两个实际应用

➢ QQ,迅雷,百度网盘客户端. 新浪网站,京东商城,淘宝..

18.2 网络编程基本介绍

Golang 的主要设计目标之一就是面向大规模后端服务程序，网络通信这块是服务端 程序必不可少也是至关重要的一部分。

➢ 网络编程有两种:

1) TCP socket 编程，是网络编程的主流。之所以叫 Tcp socket 编程，是因为底层是基于 Tcp/ip 协议的. 比如: QQ 聊天 [示意图]

2) b/s 结构的 http 编程，我们使用浏览器去访问服务器时，使用的就是 http 协议，而 http 底层依旧是用 tcp socket 实现的。[示意图] 比如: 京东商城 【这属于 go web 开发范畴 】

18.2.1 网线,网卡,无线网卡

计算机间要相互通讯,必须要求网线,网卡,或者是无线网卡.

18.2.2 协议(tcp/ip)

TCP/IP（Transmission Control Protocol/Internet Protocol)的简写,中文译名为传输控制协议/因特网互联协议，又叫网络通讯协议，这个协议是 Internet 最基本的协议、Internet 国际互联网络的基础，简单地说，就是由网络层的 IP 协议和传输层的 TCP 协议组成的。

18.2.3 OSI 与 Tcp/ip 参考模型 (推荐 tcp/ip 协议 3 卷)

18.2.4 ip 地址

概述：每个 internet 上的主机和路由器都有一个 ip 地址，它包括网络号和主机号，ip 地址有 ipv4(32 位)或者 ipv6(128 位). 可以通过 **ipconfig** 来查看

```
C:\Users\Administrator>ipconfig

Windows IP 配置


无线局域网适配器 无线网络连接 2:

   媒体状态  . . . . . . . . . . . . : 媒体已断开
   连接特定的 DNS 后缀 . . . . . . . :

无线局域网适配器 无线网络连接:

   连接特定的 DNS 后缀 . . . . . . . :
   本地链接 IPv6 地址. . . . . . . . : fe80::3c46:ea3c:8703:de88%14
   IPv4 地址 . . . . . . . . . . . . : 192.168.1.7
   子网掩码  . . . . . . . . . . . . : 255.255.255.0
   默认网关. . . . . . . . . . . . . : fe80::1%14
                                       192.168.1.1
```

18.2.5 端口(port)-介绍

我们这里所指的端口不是指物理意义上的端口，而是特指 TCP/IP 协议中的端口，是逻辑意义上的端口。

如果把 IP 地址比作一间房子，端口就是出入这间房子的门。真正的房子只有几个门，但是一个 IP 地址的端口 可以有 65536（即：256×256）个之多！端口是通过端口号来标记的，端口号只有整数，范围是从 0 到 65535（256×256-1）

18.2.6 端口(port)-分类

- 0 号是保留端口.

- 1-1024 是固定端口(程序员不要使用)

  又叫**有名端口**,即被某些程序固定使用,一般程序员不使用.

  22: SSH 远程登录协议　23: telnet 使用　21: ftp 使用

  25: smtp 服务使用　　80: iis 使用 7: echo 服务

- 1025-65535 是**动态端口**

  这些端口，程序员可以使用.

18.2.7 端口(port)-使用注意

1) 在计算机(尤其是做服务器)要尽可能的少开端口

2) 一个端口只能被一个程序监听

3) 如果使用 netstat －an 可以查看本机有哪些端口在监听

4) 可以使用 netstat －anb 来查看监听端口的 pid,在结合任务管理器关闭不安全的端口

18.3 tcp socket 编程的客户端和服务器端

为了授课方法，我们将 tcp socket 编程，简称 socket 编程.下图为 Golang socket 编程中客户端和服务器的网络分布

18.4 tcp socket 编程的快速入门

18.4.1 服务端的处理流程

1) 监听端口 8888

2) 接收客户端的 tcp 链接，建立客户端和服务器端的链接.

3) 创建 **goroutine**，处理该链接的请求(通常客户端会通过链接发送请求包)

18.4.2 客户端的处理流程

1) 建立与服务端的链接
2) 发送请求数据[终端]，接收服务器端返回的结果数据
3) 关闭链接

18.4.3 简单的程序示意图

18.4.4 代码的实现

- 程序框架图示意图
- 服务器端功能:

  编写一个服务器端程序，在 8888 端口监听

  可以和多个客户端创建链接

  链接成功后，客户端可以发送数据，服务器端接受数据，并显示在终端上.

  先使用 telnet 来测试，然后编写客户端程序来测试
- 服务端的代码:

```
package main
import (
    "fmt"
    "net" //做网络 socket 开发时,net 包含有我们需要所有的方法和函数
    _"io"
)

func process(conn net.Conn) {

    //这里我们循环的接收客户端发送的数据
    defer conn.Close() //关闭 conn

    for {
        //创建一个新的切片
        buf := make([]byte, 1024)
        //conn.Read(buf)
        //1. 等待客户端通过 conn 发送信息
        //2. 如果客户端没有 wrtie[发送]，那么协程就阻塞在这里
        fmt.Printf("服务器在等待客户端%s  发送信息\n", conn.RemoteAddr().String())
        n , err := conn.Read(buf) //从 conn 读取
        if err != nil {

            fmt.Printf("客户端退出  err=%v", err)
            return //!!!
        }
        //3. 显示客户端发送的内容到服务器的终端
```

```
        fmt.Print(string(buf[:n]))
    }

}

func main() {

    fmt.Println("服务器开始监听....")
    //net.Listen("tcp", "0.0.0.0:8888")
    //1. tcp  表示使用网络协议是 tcp
    //2. 0.0.0.0:8888  表示在本地监听  8888 端口
    listen, err := net.Listen("tcp", "0.0.0.0:8888")
    if err != nil {
        fmt.Println("listen err=", err)
        return
    }
    defer listen.Close() //延时关闭 listen

    //循环等待客户端来链接我
    for {
        //等待客户端链接
        fmt.Println("等待客户端来链接....")
        conn, err := listen.Accept()
        if err != nil {
            fmt.Println("Accept() err=", err)
```

```
        } else {
            fmt.Printf("Accept() suc con=%v  客户端 ip=%v\n", conn, conn.RemoteAddr().String())
        }
        //这里准备其一个协程，为客户端服务
        go process(conn)
    }

    //fmt.Printf("listen suc=%v\n", listen)
}
```

➢ 客户端功能:

1. 编写一个客户端端程序，能链接到 服务器端的 8888 端口
2. 客户端可以发送单行数据，然后就退出
3. 能通过终端输入数据(输入一行发送一行), 并发送给服务器端 []
4. 在终端输入 exit,表示退出程序.
5. 代码:

```
package main
import (
    "fmt"
    "net"
    "bufio"
    "os"
)

func main() {
```

```
    conn, err := net.Dial("tcp", "192.168.20.253:8888")
    if err != nil {
        fmt.Println("client dial err=", err)
        return
    }
    //功能一：客户端可以发送单行数据，然后就退出
    reader := bufio.NewReader(os.Stdin) //os.Stdin 代表标准输入[终端]

    //从终端读取一行用户输入，并准备发送给服务器
    line, err := reader.ReadString('\n')
    if err != nil {
        fmt.Println("readString err=", err)
    }

    //再将 line 发送给 服务器
    n, err := conn.Write([]byte(line))
    if err != nil {
        fmt.Println("conn.Write err=", err)
    }
    fmt.Printf("客户端发送了 %d 字节的数据，并退出", n)

}
```

2) 客户端可以发送单行数据，然后就退出
3) 能通过终端输入数据(输入一行发送一行)，并发送给服务器端 []
4) 在终端输入exit,表示退出程序.

```
if err != nil {
    fmt.Println("readString
    return
}
n, err := conn.Write([]byte(
if err != nil {
    fmt.Println("发送失败 err
    return
}
fmt.Println("发送了", n, "字
}
```

对 client.go 做了改进：

```go
    for {

        //从终端读取一行用户输入，并准备发送给服务器
        line, err := reader.ReadString('\n')
        if err != nil {
            fmt.Println("readString err=", err)
        }
        //如果用户输入的是 exit就退出
        line = strings.Trim(line, " \r\n")
        if line == "exit" {
            fmt.Println("客户端退出..")
            break
        }

        //再将line 发送给 服务器
        _, err = conn.Write([]byte(line + "\n"))
        if err != nil {
            fmt.Println("conn.Write err=", err)
        }
    }


}
```

18.5 经典项目-海量用户即时通讯系统

18.5.1 项目开发流程

需求分析--> 设计阶段---> 编码实现 --> 测试阶段-->实施

18.5.2 需求分析

1) 用户注册
2) 用户登录
3) 显示在线用户列表
4) 群聊(广播)
5) 点对点聊天
6) 离线留言

18.5.3 界面设计

```
-----------------欢迎登陆多人聊天系统:----------------
                          1 登录聊天系统
                          2 注册用户
                          3 退出系统
请选择(1-3):
------------------------------
1
登录...
请输入用户id:
100
请输入用户密码:
200

你输入的 userid=100 pwd=200
```

18.5.4 项目开发前技术准备

项目要保存用户信息和消息数据，因此我们需要学习数据库(Redis 或者 Mysql)，这里我们选择 Redis， 所以先给同学们讲解如何在 **Golang 中使用 Redis**.

18.5.5 实现功能-显示客户端登录菜单

功能：能够正确的显示客户端的菜单。

界面：

```
-----------------欢迎登陆多人聊天系统:----------------
                          1 登录聊天系统
                          2 注册用户
                          3 退出系统
请选择(1-3):
------------------------------
1
登录...
请输入用户id:
100
请输入用户密码:
200

你输入的 userid=100 pwd=200
```

```
1. 显示在线用户列表
2. 发送信息
3. 信息列表
4. 退出系统
```

思路分析：这个非常简单，直接写.

代码实现：

client/main.go

```go
package main
import (
    "fmt"
    "os"
)

//定义两个变量，一个表示用户id，一个表示用户密码
var userId int
var userPwd string

func main() {

    //接收用户的选择
    var key int
    //判断是否还继续显示菜单
    var loop = true

    for loop {
        fmt.Println("----------------欢迎登陆多人聊天系统------------")
        fmt.Println("\t\t\t 1 登陆聊天室")
        fmt.Println("\t\t\t 2 注册用户")
        fmt.Println("\t\t\t 3 退出系统")
        fmt.Println("\t\t\t 请选择(1-3):")

        fmt.Scanf("%d\n", &key)
        switch key {
```

```go
            case 1 :
                fmt.Println("登陆聊天室")
                loop = false
            case 2 :
                fmt.Println("注册用户")
                loop = false
            case 3 :
                fmt.Println("退出系统")
                //loop = false
                os.Exit(0)
            default :
                fmt.Println("你的输入有误，请重新输入")
        }

    }
    //更加用户的输入，显示新的提示信息
    if key == 1 {
        //说明用户要登陆
        fmt.Println("请输入用户的id")
        fmt.Scanf("%d\n", &userId)
        fmt.Println("请输入用户的密码")
        fmt.Scanf("%s\n", &userPwd)
        //先把登陆的函数，写到另外一个文件， 比如login.go
        err := login(userId, userPwd)
        if err != nil {
            fmt.Println("登录失败")
        } else {
            fmt.Println("登录成功")
        }

    } else if key == 2 {
        fmt.Println("进行用户注册的逻辑....")
    }
}
```

client/login.go

```go
package main
import (
    "fmt"
)

//写一个函数，完成登录
func login(userId int, userPwd string) (err error) {

    //下一个就要开始定协议..
    fmt.Printf(" userId = %d userPwd=%s\n", userId, userPwd)

    return nil
}
```

18.5.6 实现功能-完成用户登录

- 要求：先完成指定用户的验证，用户 id=100，密码 pwd=123456 可以登录，其它用户不能登录
- 这里需要先说明一个 Message 的组成(示意图)，并**发送一个 Message 的流**程

- 1.**完成客户端可以发送消息长度**，服**务器端可以正常收到该长度**值

  分析思路

(1) 先确定消息 Message 的格式和结构

(2) 然后根据上图的分析完成代码

(3) 示意图

代码实现：

server/main.go

```go
package main
import (
    "fmt"
    "net"
)

//处理和客户端的通讯
func process(conn net.Conn) {
    //这里需要延时关闭conn
    defer conn.Close()

    //循环的客户端发送的信息
    for {
        buf := make([]byte, 8096)
        fmt.Println("读取客户端发送的数据...")
        n, err := conn.Read(buf[:4])
        if n != 4 || err != nil {
            fmt.Println("conn.Read err=", err)
            return
        }
        fmt.Println("读到的buf=", buf[:4])
    }

}

func main() {
```

```go

    //提示信息
    fmt.Println("服务器在8889端口监听....")
    listen, err := net.Listen("tcp", "0.0.0.0:8889")
    defer listen.Close()
    if err != nil {
        fmt.Println("net.Listen err=", err)
        return
    }
    //一旦监听成功，就等待客户端来链接服务器
    for {
        fmt.Println("等待客户端来链接服务器.....")
        conn, err := listen.Accept()
        if err != nil {
            fmt.Println("listen.Accept err=" ,err)
        }

        //一旦链接成功，则启动一个协程和客户端保持通讯。。
        go process(conn)
    }
}
```

common/message/message.go

```go
package message

const (
    LoginMesType        = "LoginMes"
    LoginResMesType     = "LoginResMes"
)


type Message struct {
    Type string `json:"type"`  //消息类型
    Data string `json:"data"` //消息的类型
}


//定义两个消息..后面需要再增加

type LoginMes struct {
    UserId int `json:"userId"` //用户id
    UserPwd string `json:"userPwd"` //用户密码
    UserName string `json:"userName"` //用户名
}

type LoginResMes struct {
    Code int  `json:"code"` // 返回状态码 500 表示该用户未注册 200表示登录成功
    Error string `json:"error"` // 返回错误信息
}
```

client/main.go

和前面的代码一样，没有修改

client/login.go

```go
package main
import (
    "fmt"
    "encoding/json"
    "encoding/binary"
    "net"
    "go_code/chatroom/common/message"
)

//写一个函数，完成登录
func login(userId int, userPwd string) (err error) {

    //下一个就要开始定协议..
    // fmt.Printf(" userId = %d userPwd=%s\n", userId, userPwd)

    // return nil

    //1. 链接到服务器
    conn, err := net.Dial("tcp", "localhost:8889")
    if err != nil {
        fmt.Println("net.Dial err=", err)
        return
    }
    //延时关闭
    defer conn.Close()

```

```go
    //2. 准备通过conn发送消息给服务
    var mes message.Message
    mes.Type = message.LoginMesType
    //3. 创建一个LoginMes 结构体
    var loginMes message.LoginMes
    loginMes.UserId = userId
    loginMes.UserPwd = userPwd

    //4. 将loginMes 序列化
    data, err := json.Marshal(loginMes)
    if err != nil {
        fmt.Println("json.Marshal err=", err)
        return
    }
    // 5. 把data赋给 mes.Data字段
    mes.Data = string(data)

    // 6. 将 mes进行序列化化
    data, err = json.Marshal(mes)
    if err != nil {
        fmt.Println("json.Marshal err=", err)
        return
    }

    // 7. 到这个时候 data就是我们要发送的消息
```

```go
    // 7.1 先把 data的长度发送给服务器
    // 先获取到 data的长度->转成一个表示长度的byte切片
    var pkgLen uint32
    pkgLen = uint32(len(data))
    var buf [4]byte
    binary.BigEndian.PutUint32(buf[0:4], pkgLen)
    // 发送长度
    n, err := conn.Write(buf[:4])
    if n != 4 || err != nil {
        fmt.Println("conn.Write(bytes) fail", err)
        return
    }

    fmt.Printf("客户端，发送消息的长度=%d 内容=%s", len(data), string(data))
    return
}
```

- 2.完成客户端可以发送消息本身，服务器端可以正常接收到消息,并根据客户端发送的消息(LoginMes), 判断用户的合法性，并返回相应的 LoginResMes

思路分析：

(1) 让客户端发送消息本身

(2) 服务器端接受到消息， 然后反序列化成对应的消息结构体.

(3) 服务器端根据反序列化成对应的消息， 判断是否登录用户是合法, 返回 LoginResMes

(4) 客户端解析返回的 LoginResMes,显示对应界面

(5) 这里我们需要做函数的封装

代码实现：

client/login.go 做了修改

```go
    // 发送消息本身
    _, err = conn.Write(data)
    if err != nil {
        fmt.Println("conn.Write(data) fail", err)
        return
    }

    //休眠20
    time.Sleep(20 * time.Second)
    fmt.Println("休眠了20..")
    // 这里还需要处理服务器端返回的消息.
    return
}
```

server/main.go 修改

```go
func process(conn net.Conn) {
    //这里需要延时关闭conn
    defer conn.Close()

    //循环的客户端发送的信息
    for {
        //这里我们将读取数据包，直接封装成一个函数readPkg(), 返回Message, Err
        mes, err := readPkg(conn)
        if err != nil {
            if err == io.EOF {
                fmt.Println("客户端退出，服务器端也退出..")
                return
            } else {
                fmt.Println("readPkg err=", err)
                return
            }

        }
        fmt.Println("mes=", mes)

    }

}
```

将读取包的任务封装到了一个函数中.readPkg()

```go
func readPkg(conn net.Conn) (mes message.Message, err error) {

    buf := make([]byte, 8096)
    fmt.Println("读取客户端发送的数据...")
    //conn.Read 在conn没有被关闭的情况下，才会阻塞
    //如果客户端关闭了 conn 则，就不会阻塞
    _, err = conn.Read(buf[:4])
    if err != nil {
        //err = errors.New("read pkg header error")
        return
    }
    //根据buf[:4] 转成一个 uint32类型
    var pkgLen uint32
    pkgLen = binary.BigEndian.Uint32(buf[0:4])
    //根据 pkgLen 读取消息内容
    n, err := conn.Read(buf[:pkgLen])
    if n != int(pkgLen) || err != nil {
        //err = errors.New("read pkg body error")
        return
    }
    //把pkgLen 反序列化成 -> message.Message
    // 技术就是一层窗户纸 &mes！！
    err = json.Unmarshal(buf[:pkgLen], &mes)
    if err != nil {
        fmt.Println("json.Unmarsha err=", err)
        return
    }
    return
}
```

➢ 能够完成登录，并提示相应信息

server/main.go 修改

```go
func process(conn net.Conn) {
    //这里需要延时关闭conn
    defer conn.Close()

    //循环的客户端发送的信息
    for {
        //这里我们将读取数据包，直接封装成一个函数readPkg(), 返回Message, Err
        mes, err := readPkg(conn)
        if err != nil {
            if err == io.EOF {
                fmt.Println("客户端退出，服务器端也退出..")
                return
            } else {
                fmt.Println("readPkg err=", err)
                return
            }

        }
        err = serverProcessMes(conn, &mes)
        if err != nil {
            return
        }
    }
}
```

```go
//编写一个ServerProcessMes 函数
//功能：根据客户端发送消息种类不同，决定调用哪个函数来处理
func serverProcessMes(conn net.Conn, mes *message.Message) (err error) {

    switch mes.Type {
        case message.LoginMesType :
            //处理登录登录
            err = serverProcessLogin(conn, mes)
        case message.RegisterMesType :
            //处理注册
        default :
            fmt.Println("消息类型不存在，无法处理...")
    }
    return
}
```

```go
//编写一个函数serverProcessLogin函数，专门处理登录请求
func serverProcessLogin(conn net.Conn, mes *message.Message) (err error) {
    //核心代码...
    //1. 先从mes 中取出 mes.Data ，并直接反序列化成LoginMes
    var loginMes message.LoginMes
    err = json.Unmarshal([]byte(mes.Data), &loginMes)
    if err != nil {
        fmt.Println("json.Unmarshal fail err=", err)
        return
    }
    //1先声明一个 resMes
    var resMes message.Message
    resMes.Type = message.LoginResMesType
    //2在声明一个 LoginResMes，并完成赋值
    var loginResMes message.LoginResMes
    //如果用户id= 100，  密码=123456，认为合法，否则不合法

    if loginMes.UserId == 100 && loginMes.UserPwd == "123456" {
        //合法
        loginResMes.Code = 200

    } else {
        //不合法
        loginResMes.Code = 500 // 500 状态码，表示该用户不存在
        loginResMes.Error = "该用户不存在，请注册再使用..."
    }

    //3将 loginResMes 序列化
    data, err := json.Marshal(loginResMes)
    if err != nil {
        fmt.Println("json.Marshal fail", err)
        return
    }
```

client/utils.go 新增

```go
package main
import (
    "fmt"
    "net"
    "go_code/chatroom/common/message"
    "encoding/binary"
    "encoding/json"
)

func readPkg(conn net.Conn) (mes message.Message, err error) {

    buf := make([]byte, 8096)
    fmt.Println("读取客户端发送的数据~~~...")
    //conn.Read 在conn没有被关闭的情况下，才会阻塞
    //如果客户端关闭了 conn 则，就不会阻塞
    _, err = conn.Read(buf[:4])
    if err != nil {
        //err = errors.New("read pkg header error")
        return
    }
    //根据buf[:4] 转成一个 uint32类型
    var pkgLen uint32
    pkgLen = binary.BigEndian.Uint32(buf[0:4])
    //根据 pkgLen 读取消息内容
    n, err := conn.Read(buf[:pkgLen])
    if n != int(pkgLen) || err != nil {
```

```go
        //err = errors.New("read pkg body error")
        return
    }
    //把pkgLen 反序列化成 -> message.Message
    // 技术就是一层窗户纸 &mes！！
    err = json.Unmarshal(buf[:pkgLen], &mes)
    if err != nil {
        fmt.Println("json.Unmarsha err=", err)
        return
    }
    return
}

func writePkg(conn net.Conn, data []byte) (err error) {

    //先发送一个长度给对方
    var pkgLen uint32
    pkgLen = uint32(len(data))
    var buf [4]byte
    binary.BigEndian.PutUint32(buf[0:4], pkgLen)
    // 发送长度
    n, err := conn.Write(buf[:4])
    if n != 4 || err != nil {
        fmt.Println("conn.Write(bytes) fail", err)
        return
    }

    //发送data本身
    n, err = conn.Write(data)
    if n != int(pkgLen) || err != nil {
        fmt.Println("conn.Write(bytes) fail", err)
        return
    }
    return
}
```

client/login.go 增加代码

```go
    //休眠20
    // time.Sleep(20 * time.Second)
    // fmt.Println("休眠了20..")
    // 这里还需要处理服务器端返回的消息.
    mes, err = readPkg(conn) // mes 就是

    if err != nil {
        fmt.Println("readPkg(conn) err=", err)
        return
    }

    //将mes的Data部分反序列化成 LoginResMes
    var loginResMes message.LoginResMes
    err = json.Unmarshal([]byte(mes.Data), &loginResMes)
    if loginResMes.Code == 200 {
        fmt.Println("登录成功")
    } else if loginResMes.Code == 500 {
        fmt.Println(loginResMes.Error)
    }

    return
}
```

- 程序结构的改进， 前面的程序虽然完成了功能，**但是没有结构**，系统的可读性、扩展性和维护性都不好，因此需要对程序的结构进行改进。

  1) 先改进服务端, 先画出**程序的框架图[思路]**，再写代码.

2) 步骤

[1] . 先把分析出来的文件，创建好，然后放到相应的文件夹[包]

[2] 现在根据各个文件，完成的任务不同，将 main.go 的代码剥离到对应的文件中即可。

[3] 先修改了 utils/utils.go

```go
package utils
import (
    "fmt"
    "net"
    "go_code/chatroom/common/message"
    "encoding/binary"
    "encoding/json"
)

//这里将这些方法关联到结构体中
type Transfer struct {
    //分析它应该有哪些字段
    Conn net.Conn
    Buf [8096]byte //这时传输时，使用缓冲
}


func (this *Transfer) ReadPkg() (mes message.Message, err error) {

    //buf := make([]byte, 8096)
    fmt.Println("读取客户端发送的数据...")
    //conn.Read 在conn没有被关闭的情况下，才会阻塞
    //如果客户端关闭了 conn 则，就不会阻塞
    _, err = this.Conn.Read(this.Buf[:4])
    if err != nil {
        //err = errors.New("read pkg header error")
```

```go
        return
    }
    //根据buf[:4] 转成一个 uint32类型
    var pkgLen uint32
    pkgLen = binary.BigEndian.Uint32(this.Buf[0:4])
    //根据 pkgLen 读取消息内容
    n, err := this.Conn.Read(this.Buf[:pkgLen])
    if n != int(pkgLen) || err != nil {
        //err = errors.New("read pkg body error")
        return
    }
    //把pkgLen 反序列化成 -> message.Message
    // 技术就是一层窗户纸 &mes！！
    err = json.Unmarshal(this.Buf[:pkgLen], &mes)
    if err != nil {
        fmt.Println("json.Unmarsha err=", err)
        return
    }
    return
}


func (this *Transfer) WritePkg(data []byte) (err error) {

    //先发送一个长度给对方
```

```go
    var pkgLen uint32
    pkgLen = uint32(len(data))
    //var buf [4]byte
    binary.BigEndian.PutUint32(this.Buf[0:4], pkgLen)
    // 发送长度
    n, err := this.Conn.Write(this.Buf[:4])
    if n != 4 || err != nil {
        fmt.Println("conn.Write(bytes) fail", err)
        return
    }

    //发送data本身
    n, err = this.Conn.Write(data)
    if n != int(pkgLen) || err != nil {
        fmt.Println("conn.Write(bytes) fail", err)
        return
    }
    return
}
```

[4] 修改了 process2/userProcess.go

```go
package process2
import (
    "fmt"
    "net"
    "go_code/chatroom/common/message"
    "go_code/chatroom/server/utils"
    "encoding/json"
)

type UserProcess struct {
    //字段
    Conn net.Conn
    //
}

//编写一个函数serverProcessLogin函数， 专门处理登录请求
func (this *UserProcess) ServerProcessLogin(mes *message.Message) (err error) {
    //核心代码...
    //1. 先从mes 中取出 mes.Data ，并直接反序列化成LoginMes
    var loginMes message.LoginMes
    err = json.Unmarshal([]byte(mes.Data), &loginMes)
    if err != nil {
        fmt.Println("json.Unmarshal fail err=", err)
        return
    }
    //1先声明一个 resMes
```

```go
    var resMes message.Message
    resMes.Type = message.LoginResMesType
    //2在声明一个 LoginResMes，并完成赋值
    var loginResMes message.LoginResMes
    //如果用户id= 100，  密码=123456，认为合法，否则不合法

    if loginMes.UserId == 100 && loginMes.UserPwd == "123456" {
        //合法
        loginResMes.Code = 200

    } else {
        //不合法
        loginResMes.Code = 500 // 500 状态码，表示该用户不存在
        loginResMes.Error = "该用户不存在，请注册再使用..."
    }

    //3将 loginResMes 序列化
    data, err := json.Marshal(loginResMes)
    if err != nil {
        fmt.Println("json.Marshal fail", err)
        return
    }

    //4. 将data 赋值给 resMes
    resMes.Data = string(data)
```

```go

    //5. 对resMes 进行序列化，准备发送
    data, err = json.Marshal(resMes)
    if err != nil {
        fmt.Println("json.Marshal fail", err)
        return
    }
    //6. 发送data，我们将其封装到writePkg函数
    //因为使用分层模式(mvc)，我们先创建一个Transfer 实例，然后读取
    tf := &utils.Transfer{
        Conn : this.Conn,
    }
    err = tf.WritePkg(data)
    return
}
```

[5] 修改了 main/processor.go

```go
package main

import (
    "fmt"
    "net"
    "go_code/chatroom/common/message"
    "go_code/chatroom/server/utils"
    "go_code/chatroom/server/process"
    "io"
)

//先创建一个Processor 的结构体体
type Processor struct {
    Conn net.Conn
}

//编写一个ServerProcessMes 函数
//功能：根据客户端发送消息种类不同，决定调用哪个函数来处理
func (this *Processor) serverProcessMes(mes *message.Message) (err error) {

    switch mes.Type {
        case message.LoginMesType :
            //处理登录登录
            //创建一个UserProcess实例
            up := &process2.UserProcess{
                Conn : this.Conn,
```

```go
            }
            err = up.ServerProcessLogin(mes)
        case message.RegisterMesType :
            //处理注册
        default :
            fmt.Println("消息类型不存在，无法处理...")
    }
    return
}

func (this *Processor) process2() (err error) {

    //循环的客户端发送的信息
    for {
        //这里我们将读取数据包，直接封装成一个函数readPkg(), 返回Message, Err
        //创建一个Transfer 实例完成读包任务
        tf := &utils.Transfer{
            Conn : this.Conn,
        }
        mes, err := tf.ReadPkg()
        if err != nil {
            if err == io.EOF {
                fmt.Println("客户端退出，服务器端也退出..")
                return err
            } else {
```

```go
                fmt.Println("readPkg err=", err)
                return err
            }

        }
        err = this.serverProcessMes(&mes)
        if err != nil {
            return err
        }
    }

}
```

[6] 修改 main/main.go

```go
//处理和客户端的通讯
func process(conn net.Conn) {
    //这里需要延时关闭conn
    defer conn.Close()

    //这里调用总控，创建一个
    processor := &Processor{
        Conn : conn,
    }
    err := processor.process2()
    if err != nil {
        fmt.Println("客户端和服务器通讯协程错误=err", err)
        return
    }
}
```

- 修改客户端，先画出程序的框架图[思路]，再写代码

  [1] 步骤 1-画出示意图

[2] 先把各个文件放到对应的文件夹[包]

[3] 将 server/utils.go 拷贝到 client/utils/utils.go

[4] 创建了 server/process/userProcess.go

```go
type UserProcess struct {
    //暂时不需要字段..
}

//给关联一个用户登录的方法
//写一个函数，完成登录
func (this *UserProcess) Login(userId int, userPwd string) (err error) {

    //下一个就要开始定协议..
```

说明：该文件就是在原来的 login.go 做了一个改进，即封装到 UserProcess 结构体

[5] 创建了 server/process/server.go

```go
package process

import (
    "fmt"
    "os"
    "go_code/chatroom/client/utils"
    "net"
)

//显示登录成功后的界面..
func ShowMenu() {

    fmt.Println("-------恭喜xxx登录成功---------")
    fmt.Println("-------1. 显示在线用户列表---------")
    fmt.Println("-------2. 发送消息---------")
    fmt.Println("-------3. 信息列表---------")
    fmt.Println("-------4. 退出系统---------")
    fmt.Println("请选择(1-4):")
    var key int
    fmt.Scanf("%d\n", &key)
    switch key {
        case 1:
            fmt.Println("显示在线用户列表-")
        case 2:
            fmt.Println("发送消息")
        case 3:
```

```go
            fmt.Println("信息列表")
        case 4:
            fmt.Println("你选择退出了系统...")
            os.Exit(0)
        default :
            fmt.Println("你输入的选项不正确..")
    }

}

//和服务器保持通讯
func serverProcessMes(conn net.Conn) {
    //创建一个transfer实例，不停的读取服务器发送的消息
    tf := &utils.Transfer{
        Conn: conn,
    }
    for {
        fmt.Println("客户端正在等待读取服务器发送的消息")
        mes, err := tf.ReadPkg()
        if err != nil {
            fmt.Println("tf.ReadPkg err=", err)
            return
        }
        //如果读取到消息，又是下一步处理逻辑
        fmt.Printf("mes=%v\n", mes)
```

```go
    }
}
```

[6] server/main/main.go 修改

```go
        fmt.Scanf("%d\n", &key)
        switch key {
            case 1 :
                fmt.Println("登陆聊天室")
                fmt.Println("请输入用户的id")
                fmt.Scanf("%d\n", &userId)
                fmt.Println("请输入用户的密码")
                fmt.Scanf("%s\n", &userPwd)
                // 完成登录
                //1. 创建一个UserProcess的实例
                up := &process.UserProcess{}
                up.Login(userId, userPwd)
            case 2 :
                fmt.Println("注册用户")
                //loop = false
            case 3 :
                fmt.Println("退出系统")
                //loop = false
                os.Exit(0)
            default :
                fmt.Println("你的输入有误，请重新输入")
        }

    }
```

- 在 Redis 手动添加测试用户,并画图+说明注意. (后面通过程序注册用户)

手动直接在 redis 增加一个用户信息：

```
127.0.0.1:6379> hget users 100
"{\"userId\":100,\"userPwd\":\"123456\",\"userName\":\"\"}"
127.0.0.1:6379> hset users 100 "{\"userId\":100,\"userPwd\":\"123456\",\"userName\":\"scott\"}"
(integer) 0
127.0.0.1:6379> hget users 100
"{\"userId\":100,\"userPwd\":\"123456\",\"userName\":\"scott\"}"
127.0.0.1:6379>
```

- 如输入的用户名密码在 Redis 中存在则登录，否则退出系统，并给出相应的提示信息:

  1. 用户不存在,你也可以重新注册，再登录
  2. 你密码不正确。。

代码实现：

[1] 编写 model/user.go

```
D:\goproject\src\go_code\chatroom\client\process\userProcess.go

//定义一个用户的结构体

type User struct {
    //确定字段信息
    //为了序列化和反序列化成功，我们必须保证
    //用户信息的json字符串的key 和 结构体的字段对应的 tag 名字一致!!!
    UserId int `json:"userId"`
    UserPwd string `json:"userPwd"`
    UserName string `json:"userName"`
}
```

[2] 编写 model/error.go

```
package model
import (
    "errors"
)
//根据业务逻辑需要，自定义一些错误.
var (
    ERROR_USER_NOTEXISTS = errors.New("用户不存在..")
    ERROR_USER_EXISTS = errors.New("用户已经存在...")
    ERROR_USER_PWD = errors.New("密码不正确")
)
```

[3] 编写 model/userDao.go

```
package model


import (
    "fmt"
    "github.com/garyburd/redigo/redis"
    "encoding/json"
```

```
)

//我们在服务器启动后，就初始化一个 userDao 实例，
//把它做成全局的变量，在需要和 redis 操作时，就直接使用即可
var (
    MyUserDao *UserDao
)

//定义一个 UserDao  结构体体
//完成对 User  结构体的各种操作.

type UserDao struct {
    pool   *redis.Pool
}

//使用工厂模式，创建一个 UserDao 实例
func NewUserDao(pool *redis.Pool) (userDao *UserDao) {

    userDao = &UserDao{
        pool: pool,
    }
    return
}

//思考一下在 UserDao  应该提供哪些方法给我们
//1. 根据用户 id  返回  一个 User 实例+err
```

```
func (this *UserDao) getUserById(conn redis.Conn, id int) (user *User, err error) {

    //通过给定 id 去 redis 查询这个用户
    res, err := redis.String(conn.Do("HGet", "users", id))
    if err != nil {
        //错误!
        if err == redis.ErrNil { //表示在 users 哈希中，没有找到对应 id
            err = ERROR_USER_NOTEXISTS
        }
        return
    }
    user = &User{}
    //这里我们需要把 res 反序列化成 User 实例
    err = json.Unmarshal([]byte(res), user)
    if err != nil {
        fmt.Println("json.Unmarshal err=", err)
        return
    }
    return
}

//完成登录的校验 Login
//1. Login 完成对用户的验证
//2. 如果用户的 id 和 pwd 都正确，则返回一个 user 实例
//3. 如果用户的 id 或 pwd 有错误，则返回对应的错误信息
func (this *UserDao) Login(userId int, userPwd string) (user *User, err error) {
```

```
    //先从 UserDao 的连接池中取出一根连接
    conn := this.pool.Get()
    defer conn.Close()
    user, err = this.getUserById(conn, userId)
    if err != nil {
        return
    }
    //这时证明这个用户是获取到.
    if user.UserPwd != userPwd {
        err = ERROR_USER_PWD
        return
    }
    return
}
```

[4] main/redis.go

```go
package main
import (
    "github.com/garyburd/redigo/redis"
    "time"
)

//定义一个全局的pool
var pool *redis.Pool

func initPool(address string, maxIdle, maxActive int, idleTimeout time.Duration)

    pool = &redis.Pool{
        MaxIdle: maxIdle, //最大空闲链接数
        MaxActive: maxActive, // 表示和数据库的最大链接数， 0 表示没有限制
        IdleTimeout: idleTimeout, // 最大空闲时间
        Dial: func() (redis.Conn, error) { // 初始化链接的代码， 链接哪个ip的redis
        return redis.Dial("tcp", address)
        },
    }
}
```

[5] main/main.go

```go

//这里我们编写一个函数，完成对UserDao的初始化任务
func initUserDao() {
    //这里的pool 本身就是一个全局的变量
    //这里需要注意一个初始化顺序问题
    //initPool, 在 initUserDao
    model.MyUserDao = model.NewUserDao(pool)
}

func main() {
    //当服务器启动时，我们就去初始化我们的redis的连接池
    initPool("localhost:6379", 16, 0, 300 * time.Second)
    initUserDao()
    //提示信息
    fmt.Println("服务器[新的结构]在8889端口监听....")
    listen, err := net.Listen("tcp", "0.0.0.0:8889")
```

[6] 在 process/userProcess.go 使用到 redis 验证的功能

```go
//我们需要到redis数据库去完成验证.
//1.使用model.MyUserDao 到redis去验证
user, err := model.MyUserDao.Login(loginMes.UserId, loginMes.UserPwd)

if err != nil {

    if err == model.ERROR_USER_NOTEXISTS {
        loginResMes.Code = 500
        loginResMes.Error = err.Error()
    } else if err == model.ERROR_USER_PWD  {
        loginResMes.Code = 403
        loginResMes.Error = err.Error()
    } else {
        loginResMes.Code = 505
        loginResMes.Error = "服务器内部错误..."
    }

} else {
    loginResMes.Code = 200
    fmt.Println(user, "登录成功")
}
```

18.5.7 实现功能-完成注册用户

1) 完成注册功能，将用户信息录入到 Redis 中
2) 思路分析，并完成代码
3) 思路分析的示意图

➢ 实现功能-完成注册用户

[1] common/message/user.go

```go
package message
D:\goproject\src\go_code\chatroom\common\message\user.go

//定义一个用户的结构体

type User struct {
    //确定字段信息
    //为了序列化和反序列化成功，我们必须保证
    //用户信息的json字符串的key 和 结构体的字段对应的 tag 名字一致!!!
    UserId int `json:"userId"`
    UserPwd string `json:"userPwd"`
    UserName string `json:"userName"`
}
```

[2] common/message/message.go

```go
type RegisterMes struct {
    User User `json:"user"`//类型就是User结构体.
}
type RegisterResMes struct {
    Code int  `json:"code"` // 返回状态码 400 表示该用户已经占有 200表
    Error string `json:"error"` // 返回错误信息
}
```

[3] client/process/userProcess.go

```go
func (this *UserProcess) Register(userId int,
    userPwd string, userName string) (err error) {



    //1. 链接到服务器
    conn, err := net.Dial("tcp", "localhost:8889")
```

```
if err != nil {
    fmt.Println("net.Dial err=", err)
    return
}
//延时关闭
defer conn.Close()

//2. 准备通过 conn 发送消息给服务
var mes message.Message
mes.Type = message.RegisterMesType
//3. 创建一个 LoginMes 结构体
var registerMes message.RegisterMes
registerMes.User.UserId = userId
registerMes.User.UserPwd = userPwd
registerMes.User.UserName = userName

//4.将 registerMes 序列化
data, err := json.Marshal(registerMes)
if err != nil {
    fmt.Println("json.Marshal err=", err)
    return
}

// 5. 把 data 赋给 mes.Data 字段
mes.Data = string(data)
```

```
// 6. 将 mes 进行序列化化
data, err = json.Marshal(mes)
if err != nil {
    fmt.Println("json.Marshal err=", err)
    return
}

//创建一个 Transfer 实例
tf := &utils.Transfer{
    Conn : conn,
}

//发送 data 给服务器端
err = tf.WritePkg(data)
if err != nil {
    fmt.Println("注册发送信息错误 err=", err)
}

mes, err = tf.ReadPkg() // mes 就是 RegisterResMes

if err != nil {
    fmt.Println("readPkg(conn) err=", err)
    return
}

//将 mes 的 Data 部分反序列化成 RegisterResMes
```

```
    var registerResMes message.RegisterResMes
    err = json.Unmarshal([]byte(mes.Data), &registerResMes)
    if registerResMes.Code == 200 {
        fmt.Println("注册成功，你重新登录一把")
        os.Exit(0)
    } else   {
        fmt.Println(registerResMes.Error)
        os.Exit(0)
    }
    return
}
```

[4] 在 client/main/main.go 增加了代码

```
        fmt.Println("请输入用户名字(nickname):")
        fmt.Scanf("%s\n", &userName)
        //2. 调用UserProcess，完成注册的请求、
        up := &process.UserProcess{}
        up.Register(userId, userPwd, userName)
    case 3 :
        fmt.Println("退出系统")
```

[5] 在 server/model/userDao.go 增加方法

```go
func (this *UserDao) Register(user *message.User) (err error) {

    //先从UserDao 的连接池中取出一根连接
    conn := this.pool.Get()
    defer conn.Close()
    _, err = this.getUserById(conn, user.UserId)
    if err == nil {
        err = ERROR_USER_EXISTS
        return
    }
    //这时，说明id在redis还没有，则可以完成注册
    data, err := json.Marshal(user) //序列化
    if err != nil {
        return
    }
    //入库
    _, err = conn.Do("HSet", "users", user.UserId, string(data))
    if err != nil {
        fmt.Println("保存注册用户错误 err=", err)
        return
    }
    return
}
```

[6] 在 server/process/userProcess.go 增加了方法，处理注册

```go
func (this *UserProcess) ServerProcessRegister(mes *message.Message) (err error) {



    //1.先从 mes 中取出 mes.Data ，并直接反序列化成 RegisterMes
    var registerMes message.RegisterMes
    err = json.Unmarshal([]byte(mes.Data), &registerMes)
    if err != nil {
```

```
        fmt.Println("json.Unmarshal fail err=", err)
        return
    }

    //1 先声明一个 resMes
    var resMes message.Message
    resMes.Type = message.RegisterResMesType
    var registerResMes message.RegisterResMes

    //我们需要到 redis 数据库去完成注册.
    //1.使用 model.MyUserDao 到 redis 去验证
    err = model.MyUserDao.Register(&registerMes.User)

    if err != nil {
        if err == model.ERROR_USER_EXISTS {
            registerResMes.Code = 505
            registerResMes.Error = model.ERROR_USER_EXISTS.Error()
        } else {
            registerResMes.Code = 506
            registerResMes.Error = "注册发生未知错误..."
        }
    } else {
        registerResMes.Code = 200
    }

    data, err := json.Marshal(registerResMes)
```

```
    if err != nil {
        fmt.Println("json.Marshal fail", err)
        return
    }


    //4. 将 data 赋值给 resMes
    resMes.Data = string(data)

    //5. 对 resMes 进行序列化，准备发送
    data, err = json.Marshal(resMes)
    if err != nil {
        fmt.Println("json.Marshal fail", err)
        return
    }
    //6. 发送 data, 我们将其封装到 writePkg 函数
    //因为使用分层模式(mvc), 我们先创建一个 Transfer 实例，然后读取
    tf := &utils.Transfer{
        Conn : this.Conn,
    }
    err = tf.WritePkg(data)
    return

}
```

[7] server/main/processor.go 调用了

```go
//功能：根据客户端发送消息种类不同，决定调用哪个函数来处理
func (this *Processor) serverProcessMes(mes *message.Message) (err error) {

    switch mes.Type {
        case message.LoginMesType :
            //处理登录登录
            //创建一个UserProcess实例
             up := &process2.UserProcess{
                 Conn : this.Conn,
             }
             err = up.ServerProcessLogin(mes)
        case message.RegisterMesType :
            //处理注册
            up := &process2.UserProcess{
                 Conn : this.Conn,
             }
             err = up.ServerProcessRegister(mes) // type : data
        default :
            fmt.Println("消息类型不存在，无法处理...")
    }
    return
}
```

18.5.8 实现功能-完成登录时能返回当前在线用户

- 用户登录后，可以得到当前在线用户列表思路分析、示意图、代码实现

思路分析：

代码实现：

[1] 编写了 server/process/userMgr.go

```go
package process2
import (
    "fmt"
)
//因为UserMgr 实例在服务器端有且只有一个
//因为在很多的地方，都会使用到，因此，我们
//将其定义为全局变量
var (
    userMgr *UserMgr
)
type UserMgr struct {
    onlineUsers map[int]*UserProcess
}
//完成对userMgr初始化工作
func init() {
    userMgr = &UserMgr{
        onlineUsers : make(map[int]*UserProcess, 1024),
    }
}
//完成对onlineUsers添加
func (this *UserMgr) AddOnlineUser(up *UserProcess) {
    this.onlineUsers[up.UserId] = up
}
//删除
func (this *UserMgr) DelOnlineUser(userId int) {
    delete(this.onlineUsers, userId)
```

```go
}
//返回当前所有在线的用户
func (this *UserMgr) GetAllOnlineUser() map[int]*UserProcess {
    return this.onlineUsers
}
//根据id返回对应的值
func (this *UserMgr) GetOnlineUserById(userId int) (up *UserProcess, err error) {

    //如何从map取出一个值，带检测方式
    up, ok := this.onlineUsers[userId]
    if !ok { //说明，你要查找的这个用户，当前不在线。
        err = fmt.Errorf("用户%d 不存在", userId)
        return
    }
    return
}
```

[2] server/process/userProcess.go

```go
} else {
    loginResMes.Code = 200
    //这里，因为用户登录成功，我们就把该登录成功的用放入到userMgr中
    //将登录成功的用户的userId 赋给 this
    this.UserId = loginMes.UserId
    userMgr.AddOnlineUser(this)
    //将当前在线用户的id 放入到loginResMes.UsersId
    //遍历 userMgr.onlineUsers
    for id, _ := range userMgr.onlineUsers {
        loginResMes.UsersId = append(loginResMes.UsersId, id)
    }
    fmt.Println(user, "登录成功")
}
```

[3] common/message/message.go

```go
type LoginResMes struct {
    Code int  `json:"code"` // 返回状态码 500 表示该用户未
    UsersId []int          // 增加字段，保存用户id的切片
    Error string `json:"error"` // 返回错误信息
}
```

[4] client/process/userProcess.go

```go
if loginResMes.Code == 200 {

    //fmt.Println("登录成功")
    //可以显示当前在线用户列表,遍历loginResMes.UsersId
    fmt.Println("当前在线用户列表如下:")
    for _, v := range loginResMes.UsersId {
        //如果我们要求不显示自己在线,下面我们增加一个代码
        if v == userId {
            continue
        }

        fmt.Println("用户id:\t", v)
    }
    fmt.Print("\n\n")
```

- 当一个**新的用户上线后**，其它已经登录的用户也能获取最新在线用户列表,思路分析、示意图、代码实现

[1] server/process/userProcess.go

```go
//这里我们编写通知所有在线的用户的方法
//userId 要通知其它的在线用户，我上线
func (this *UserProcess) NotifyOthersOnlineUser(userId int) {

    //遍历 onlineUsers，然后一个一个的发送 NotifyUserStatusMes
    for id, up := range userMgr.onlineUsers {
        //过滤到自己
        if id == userId {
            continue
        }
        //开始通知【单独的写一个方法】
        up.NotifyMeOnline(userId)
    }
}
```

```go
func (this *UserProcess) NotifyMeOnline(userId int) {

    //组装我们的NotifyUserStatusMes
    var mes message.Message
    mes.Type = message.NotifyUserStatusMesType

    var notifyUserStatusMes message.NotifyUserStatusMes
    notifyUserStatusMes.UserId = userId
    notifyUserStatusMes.Status = message.UserOnline

    //将notifyUserStatusMes序列化
    data, err := json.Marshal(notifyUserStatusMes)
    if err != nil {
        fmt.Println("json.Marshal err=", err)
        return
    }
    //将序列化后的notifyUserStatusMes赋值给 mes.Data
    mes.Data = string(data)

    //对mes再次序列化，准备发送.
    data, err = json.Marshal(mes)
    if err != nil {
        fmt.Println("json.Marshal err=", err)
        return
    }
```

```go
    //发送,创建我们Transfer实例，发送
    tf := &utils.Transfer{
        Conn : this.Conn,
    }

    err = tf.WritePkg(data)
    if err != nil {
        fmt.Println("NotifyMeOnline err=", err)
        return
    }
}
```

[2] sever/proces/userProcess.go [的 Login]

```go
} else {
    loginResMes.Code = 200
    //这里，因为用户登录成功，我们就把该登录成功的用放入到userMgr中
    //将登录成功的用户的userId 赋给 this
    this.UserId = loginMes.UserId
    userMgr.AddOnlineUser(this)
    //通知其它的在线用户， 我上线了
    this.NotifyOthersOnlineUser(loginMes.UserId)
    //将当前在线用户的id 放入到loginResMes.UsersId
    //遍历 userMgr.onlineUsers
```

[3] common/mesage/message.go

```go
//为了配合服务器端推送用户状态变化的消息
type NotifyUserStatusMes struct {
    UserId int `json:"userId"` //用户id
    Status int `json:"status"` //用户的状态
}
```

[4] client/process/userMgr.go

```go
package process
import (
    "fmt"
    "go_code/chatroom/common/message"
)

//客户端要维护的map
var onlineUsers map[int]*message.User = make(map[int]*message.User, 10)

//在客户端显示当前在线的用户
func outputOnlineUser() {
    //遍历一把 onlineUsers
    fmt.Println("当前在线用户列表:")
    for id, _ := range onlineUsers{
        //如果不显示自己.
        fmt.Println("用户id:\t", id)
    }
}

//编写一个方法，处理返回的NotifyUserStatusMes
func updateUserStatus(notifyUserStatusMes *message.NotifyUserStatusMes) {

    //适当优化
    user, ok := onlineUsers[notifyUserStatusMes.UserId]
    if !ok { //原来没有
        user = &message.User{
            UserId : notifyUserStatusMes.UserId,
        }
    }
    user.UserStatus = notifyUserStatusMes.Status
    onlineUsers[notifyUserStatusMes.UserId] = user

    outputOnlineUser()
}
```

[5] client/process/server.go

```go
    }
    for {
        fmt.Println("客户端正在等待读取服务器发送的消息")
        mes, err := tf.ReadPkg()
        if err != nil {
            fmt.Println("tf.ReadPkg err=", err)
            return
        }
        //如果读取到消息，又是下一步处理逻辑
        switch mes.Type {
            case message.NotifyUserStatusMesType : // 有人上线了

                //1. 取出.NotifyUserStatusMes
                var notifyUserStatusMes message.NotifyUserStatusMes
                json.Unmarshal([]byte(mes.Data), &notifyUserStatusMes)
                //2. 把这个用户的信息，状态保存到客户map[int]User中
                updateUserStatus(&notifyUserStatusMes)
                //处理
            default :
                fmt.Println("服务器端返回了未知的消息类型")
        }
        //fmt.Printf("mes=%v\n", mes)

    }
}
```

[6] client/process/server.go

```go
func ShowMenu() {

    fmt.Println("-------恭喜xxx登录成功---------")
    fmt.Println("-------1. 显示在线用户列表---------")
    fmt.Println("-------2. 发送消息---------")
    fmt.Println("-------3. 信息列表---------")
    fmt.Println("-------4. 退出系统---------")
    fmt.Println("请选择(1-4):")
    var key int
    fmt.Scanf("%d\n", &key)
    switch key {
        case 1:
            //fmt.Println("显示在线用户列表-")
            outputOnlineUser()
        case 2:
            fmt.Println("发送消息")
        case 3:
```

```go
    //fmt.Println("登录成功")
    //可以显示当前在线用户列表,遍历loginResMes.UsersId
    fmt.Println("当前在线用户列表如下:")
    for _, v := range loginResMes.UsersId {
        //如果我们要求不显示自己在线,下面我们增加一个代码
        if v == userId {
            continue
        }

        fmt.Println("用户id:\t", v)
        //完成 客户端的 onlineUsers 完成初始化
        user := &message.User{
            UserId : v,
            UserStatus : message.UserOnline,
        }
        onlineUsers[v]= user
    }
    fmt.Print("\n\n")
```

18.5.9 实现功能-完成登录用可以群聊

- 步骤 1：步骤 1:当一个用户上线后，可以将群聊消息发给服务器，服务器可以接收到

  思路分析:

代码实现：

[1] common/message/messag.go

```
//增加一个SmsMes //发送的消息
type SmsMes struct {
    Content string `json:"content"` //内容
    User //匿名结构体，继承
}
```

[2] client/model/curUser.go

```go
package model

import (
    "net"
    "go_code/chatroom/common/message"
)
//因为在客户端，我们很多地方会使用到curUser,我们将其作为一个全局
type CurUser struct {
    Conn net.Conn
    message.User
}
```

[3] client/process/smsProcess.go 增加了发送群聊消息

```go
type SmsProcess struct {
}

//发送群聊的消息
func (this *SmsProcess) SendGroupMes(content string) (err error) {

    //1 创建一个Mes
    var mes message.Message
    mes.Type = message.SmsMesType

    //2 创建一个SmsMes 实例
    var smsMes message.SmsMes
    smsMes.Content = content //内容.
    smsMes.UserId = CurUser.UserId //
    smsMes.UserStatus = CurUser.UserStatus //

    //3.序列化 smsMes
    data, err := json.Marshal(smsMes)
    if err != nil {
        fmt.Println("SendGroupMes json.Marshal fail =", err.Error())
        return
    }

    mes.Data = string(data)

    //4. 对mes再次序列化
```

```go
    //4. 对mes再次序列化
    data, err = json.Marshal(mes)
    if err != nil {
        fmt.Println("SendGroupMes json.Marshal fail =", err.Error())
        return
    }

    //5. 将mes发送给服务器。。
    tf := &utils.Transfer{
        Conn : CurUser.Conn,
    }
    //6.发送
    err = tf.WritePkg(data)
    if err != nil {
        fmt.Println("SendGroupMes err=", err.Error())
        return
    }

    return
}
```

```go
if loginResMes.Code == 200 {
    //初始化CurUser
    CurUser.Conn = conn
    CurUser.UserId = userId
    CurUser.UserStatus = message.UserOnline

    //fmt.Println("登录成功")
    //可以显示当前在线用户列表，遍历loginResMes.UsersId
```

[4] 测试

```
等待客户端来链接服务器.....
读取客户端发送的数据...
mes= &{LoginMes {"userId":100,"userPwd":"123456","userName":""}}
&{100 123456 tom} 登录成功
读取客户端发送的数据...
mes= &{SmsMes {"content":"hello,body!","userId":100,"userPwd":"","userName":"","userStatus":
消息类型不存在，无法处理...
读取客户端发送的数据...

        半:
```

➢ 步骤 2:服务器可以将接收到的消息，群发给所有在线用户(发送者除外)

思路分析：

代码实现：

[1] server/process/smsProcess.go

```go
package process2
import (
    "fmt"
    "net"
    "go_code/chatroom/common/message"
    "go_code/chatroom/server/utils"

    "encoding/json"
)

type SmsProcess struct {
    //..[暂时不需字段]
}
//写方法转发消息
func (this *SmsProcess) SendGroupMes(mes *message.Message) {

    //遍历服务器端的onlineUsers map[int]*UserProcess,
    //将消息转发取出.
    //取出mes的内容 SmsMes
    var smsMes message.SmsMes
    err := json.Unmarshal([]byte(mes.Data), &smsMes)
    if err != nil {
        fmt.Println("json.Unmarshal err=", err)
        return
    }

```

```go
    data, err := json.Marshal(mes)
    if err != nil {
        fmt.Println("json.Marshal err=", err)
        return
    }

    for id, up := range userMgr.onlineUsers {
        //这里，还需要过滤到自己,即不要再发给自己
        if id == smsMes.UserId {
            continue
        }
        this.SendMesToEachOnlineUser(data, up.Conn)
    }
}
func (this *SmsProcess) SendMesToEachOnlineUser(data []byte , conn net.Conn) {

    //创建一个Transfer 实例，发送data
    tf := &utils.Transfer{
        Conn : conn, //
    }
    err := tf.WritePkg(data)
    if err != nil {
        fmt.Println("转发消息失败 err=", err)
    }
}
```

[2] server/main/processor.go

```go
        //处理注册
        up := &process2.UserProcess{      server/main  /processor.go
            Conn : this.Conn,
        }
        err = up.ServerProcessRegister(mes) // type : data
    case message.SmsMesType :
        //创建一个SmsProcess实例完成转发群聊消息.
        smsProcess := &process2.SmsProcess{}
        smsProcess.SendGroupMes(mes)
    default :
        fmt.Println("消息类型不存在，无法处理...")
}
```

[3] client/process/smsMgr.go

```go
package process
import (
    "fmt"
    "go_code/chatroom/common/message"
    "encoding/json"
)

func outputGroupMes(mes *message.Message) { //这个地方mes一定SmsMes
    //显示即可
    //1. 反序列化mes.Data
    var smsMes message.SmsMes
    err := json.Unmarshal([]byte(mes.Data), &smsMes)
    if err != nil {
        fmt.Println("json.Unmarshal err=", err.Error())
        return
    }

    //显示信息
    info := fmt.Sprintf("用户id:\t%d 对大家说:\t%s",
        smsMes.UserId, smsMes.Content)
    fmt.Println(info)
    fmt.Println()


}
```

[4] client/process/server.go

```go
            //2. 把这个用户的信息，状态保存到客户map[int]User中
            updateUserStatus(&notifyUserStatusMes)
            //处理
        case message.SmsMesType : //有人群发消息
            outputGroupMes(&mes)
        default :
            fmt.Println("服务器端返回了未知的消息类型")
    }
    //fmt.Printf("mes=%v\n", mes)

}
```

18.5.10 聊天的项目的扩展功能要求

1. **实现私聊**.[点对点聊天]
2. 如果一个登录用户离线，就把这个人从在线列表去掉【】
3. 实现离线留言，在群聊时，如果某个用户没有在线，当登录后，可以接受离线的消息
4. 发送一个文件.

第 19 章Redis 的使用

19.1 Redis 基本介绍

1.Redis 是 NoSQL数据库， 不是传统的关系型数据库
官网: https://redis.io/ 和 http://www.redis.cn/

2.Redis:REmote DIctionary Server(远程字典服务器)，Redis性能非常高，单机能够达到15w qps，通常适合做缓存，也可以持久化。

3.是完全开源免费的，高性能的(key/value)分布式内存数据库，基于内存运行并支持持久化的NoSQL数据库，是最热门的NoSql数据库之一,也称为**数据结构**服务器

19.1.1 Redis 的安装

下载后直接解压就有Redis的服务器端程序(redis-server.exe)和客户端程序(redis-cli.exe)，直接双击即可运行,并不需要安装。

Redis-x64-3.2.100.zip

19.1.2 Redis 操作的基本原理图

19.2 Redis 的安装和基本使用

19.2.1 Redis 的启动:

19.3 Redis 的操作指令一览

19.3.1 Redis 的基本使用:

说明: Redis 安装好后，默认有 16 个数据库，初始默认使用 0 号库，编号是 0...15

1. 添加 key-val [set]
2. 查看当前 redis 的 所有 key [keys *]
3. 获取 key 对应的值. [get key]
4. 切换 redis 数据库 [select index]
5. 如何查看当前数据库的 key-val 数量 [dbsize]
6. 清空当前数据库的 key-val 和清空所有数据库的 key-val [flushdb flushall]

```
127.0.0.1:6379> set key1 hello
OK
127.0.0.1:6379> get key1
"hello"
127.0.0.1:6379> get key1
"hello"
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> get key1
(nil)
127.0.0.1:6379[1]> select 0
OK
127.0.0.1:6379> get key1
"hello"
127.0.0.1:6379> dbsize
(integer) 1
127.0.0.1:6379> set key2 sgg
OK
127.0.0.1:6379> dbsize
(integer) 2
127.0.0.1:6379>
```

19.4 Redis 的 Crud 操作

19.4.1 Redis 的五大数据类型:

Redis 的五大数据类型是: String(字符串) 、Hash (哈希)、List(列表)、Set(集合)和 zset(sorted set：有序集合)

19.4.2 String(字符串) -介绍

string 是 redis 最基本的类型，一个 key 对应一个 value。

string 类型是二进制安全的。除普通的字符串外，也可以存放图片等数据。

redis 中字符串 value 最大是 512M

- 举例,存放一个地址信息:

  address 北京天安门

  说明 ：

  key : address

  value: 北京天安门

```
127.0.0.1:6379> set address beijing
OK
127.0.0.1:6379> get address
"beijing"
127.0.0.1:6379>
```

- String(字符串) -CRUD

  举例说明 Redis 的 String 字符串的 CRUD 操作.

  set[如果存在就相当于修改，不存在就是添加]/get/del

```
"beijing"
127.0.0.1:6379> del address
(integer) 1
127.0.0.1:6379> get address
(nil)
127.0.0.1:6379>
```

19.4.3 String(字符串)-使用细节和注意事项

- setex(set with expire)键秒值

```
(nil)
127.0.0.1:6379> setex mess01 10 hello,you
OK
127.0.0.1:6379> get mess01
"hello,you"
127.0.0.1:6379> get mess01
"hello,you"
127.0.0.1:6379> get mess01
"hello,you"
127.0.0.1:6379> get mess01
(nil)
127.0.0.1:6379>
```

10s

- mset[同时设置一个或多个 key-value 对]
- mget[同时获取多个 key-val]

```
(nil)
127.0.0.1:6379> mset worker01 tom worker02 scott
OK
127.0.0.1:6379> get worker02
"scott"
127.0.0.1:6379> mget worker01 worker02
1) "tom"
2) "scott"
127.0.0.1:6379>
```

19.4.4 Hash (哈希，类似 golang 里的 Map)-介绍

- 基本的介绍

  Redis hash 是一个键值对集合。var user1 map[string]string

  Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。

- 举例,存放一个 User 信息:(user1)

  user1 name "smith" age 30 job "golang coder"

  说明 ：

key : user1

name 张三 和 age 30 就是两对 field-value

```
1) "tom"
2) "scott"
127.0.0.1:6379> hset user1 name "smith"
(integer) 1
127.0.0.1:6379> hset user1 age  30
(integer) 1
127.0.0.1:6379> hset user1 job "golang coder"
(integer) 1
127.0.0.1:6379> hget user1 name
"smith"
127.0.0.1:6379> hget user1 age
"30"
127.0.0.1:6379> hget user1 job
"golang coder"
127.0.0.1:6379>
```

19.4.5 Hash（哈希，类似 golang 里的 Map）-CRUD

举例说明 Redis 的 Hash 的 CRUD 的基本操作.

hset/hget/**hgetall/hdel**

演示添加 user 信息的案例 (name,age)

```
127.0.0.1:6379> hset user1 name "smith"
(integer) 1
127.0.0.1:6379> hset user1 age  30
(integer) 1
127.0.0.1:6379> hset user1 job "golang coder"
(integer) 1
127.0.0.1:6379> hget user1 name
"smith"
127.0.0.1:6379> hget user1 age
"30"
127.0.0.1:6379> hget user1 job
"golang coder"
127.0.0.1:6379> hgetall user1
1) "name"
2) "smith"
3) "age"
4) "30"
5) "job"
6) "golang coder"
```

19.4.6 Hash-使用细节和注意事项

- 在给 user 设置 name 和 age 时，前面我们是一步一步设置,使用 hmset 和 hmget 可以一次性来设置多个 filed 的值和返回多个 field 的值 。

- hlen 统计一个 hash 有几个元素.
- hexists key field

  查看哈希表 key 中，给定域 field 是否存在

```
2) "110"
3) "\xa1java coder"
127.0.0.1:6379> hmset user2 name jerry age 110 job "java coder"
OK
127.0.0.1:6379> hmget user2 name age job
1) "jerry"
2) "110"
3) "java coder"
127.0.0.1:6379> hlen user2
(integer) 3
127.0.0.1:6379> hexists user2 name
(integer) 1
127.0.0.1:6379> hexists user2 name2
(integer) 0
127.0.0.1:6379>
```

19.4.7 课堂练习

Hash课堂练习

举例,存放一个Student信息:

stu1 name 张三 age 30 score 80 address 北京

说明：

通过相关指令，完成对Student 的crud操作

19.4.8 List（列表）-介绍

列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

List 本质是个链表, List 的元素 是有序的，元素的值可以重复.

举例,**存放多个地址信息**:

city 北京 天津 上海

说明 ：

key : city

北京 天津 上海 就是三个元素

- 入门的案例

19.4.9 List（列表）-CRUD

举例说明 Redis 的 List 的 CRUD 操作。

lpush/rpush/lrange/lpop/rpop/del/

➢ 说明:

List 画图帮助学员理解(可以把 l 想象成一根管道.)

herosList 的演示

```
127.0.0.1:6379> lpush city beijing shanghai tianjing
(integer) 3
127.0.0.1:6379> lrange city 0 -1
1) "tianjing"
2) "shanghai"
3) "beijing"
127.0.0.1:6379> lpush herolist aaa bbb ccc
(integer) 3
127.0.0.1:6379> lrange herolist 0 -1
1) "ccc"
2) "bbb"
3) "aaa"
127.0.0.1:6379> rpush herolist ddd eee
(integer) 5
127.0.0.1:6379> lrange herolist 0 -1
1) "ccc"
2) "bbb"
3) "aaa"
4) "ddd"
5) "eee"
127.0.0.1:6379> lpop herolist
"ccc"
127.0.0.1:6379> lrange herolist 0 -1
1) "bbb"
2) "aaa"
3) "ddd"
4) "eee"
127.0.0.1:6379> rpop herolist
```

```
127.0.0.1:6379> lrange herolist 0 -1
1) "bbb"
2) "aaa"
3) "ddd"
127.0.0.1:6379> del herolist
(integer) 1
127.0.0.1:6379> lrange herolist 0 -1
(empty list or set)
127.0.0.1:6379>
```

19.4.10 List-使用细节和注意事项

(1) lindex，按照索引下标获得元素(从左到右，编号从0开始.)
(2) **LLEN key**
返回列表 key 的长度,如果 key 不存在，则 key 被解释为一个空列表，返回 0

(3) List的其它说明

- List 数据，可以从左或者右 插入添加；
- 如果值全移除，对应的键也就消失了。

19.4.11 Set(集合) - 介绍

- Redis 的 Set 是 string 类型的无序集合。
- 底层是 HashTable 数据结构, Set 也是存放很多字符串元素，字符串元素是无序的，而且元素的值不能重复
- 举例,存放多个邮件列表信息:

  email sgg@sohu.com tom@sohu.com

  说明 ：

  key : email

tn@sohu.com tom@sohu.com　就是二个元素

redis>sadd email　xx xxx

```
127.0.0.1:6379>
127.0.0.1:6379> sadd emails tom@sohu.com jack@qq.com
(integer) 2
127.0.0.1:6379> smembers emails
1) "tom@sohu.com"
2) "jack@qq.com"
127.0.0.1:6379> smembers emails
1) "tom@sohu.com"
2) "jack@qq.com"
127.0.0.1:6379> sadd emails kk@yy.com yy@sohu.com
(integer) 2
127.0.0.1:6379> smembers emails
1) "kk@yy.com"
2) "tom@sohu.com"
3) "jack@qq.com"
4) "yy@sohu.com"
127.0.0.1:6379> sadd emails tom@sohu.com
(integer) 0
127.0.0.1:6379>
```

从集合中取出所有的元素

19.4.12 Set(集合)- CRUD

- 举例说明 Redis 的 Set 的 CRUD 操作.

  sadd

  smembers[取出所有值]

  sismember[判断值是否是成员]

  srem [删除指定值]

- 演示添加多个电子邮件信息的案例

```
4) "yy@sohu.com"
127.0.0.1:6379> sadd emails tom@sohu.com
(integer) 0
127.0.0.1:6379> sismember emails tom@sohu.com
(integer) 1
127.0.0.1:6379> sismember emails tom~@sohu.com
(integer) 0
127.0.0.1:6379> srem emails tom@sohu.com
(integer) 1
127.0.0.1:6379> smembers emails
1) "yy@sohu.com"
2) "kk@yy.com"
3) "jack@qq.com"
127.0.0.1:6379>
```

19.4.13 Set 课堂练习

- 举例,存放一个商品信息:

  包括 商品名、价格、生产日期。

  完成对应的 crud 操作

19.5 Golang 操作 Redis

19.5.1 安装第三方开源 Redis 库

1) 使用第三方开源的 redis 库: github.com/garyburd/redigo/redis
2) 在使用 Redis 前，先安装第三方 Redis 库，在 GOPATH 路径下执行安装指令:

D:\goproject>go get github.com/garyburd/redigo/redis

3) 安装成功后,可以看到如下包

➢ 特别说明：在安装 Redis 库前，确保已经安装并配置了 Git，因为 是从 github 下载安装 Redis 库的，需要使用到 Git。 如果没有安装配置过 Git，请参考：如何安装配置 Git

19.5.2 Set/Get 接口

说明：通过 Golang 添加和获取 key-value 【比如 name-tom~ 】

```
package main
import (
    "fmt"
    "github.com/garyburd/redigo/redis" //引入 redis 包
)

func main() {
    //通过 go 向 redis 写入数据和读取数据
    //1. 链接到 redis
    conn, err := redis.Dial("tcp", "127.0.0.1:6379")
    if err != nil {
        fmt.Println("redis.Dial err=", err)
        return
    }
```

```
    defer conn.Close() //关闭..

    //2. 通过 go 向 redis 写入数据 string [key-val]
    _, err = conn.Do("Set", "name", "tomjerry 猫猫")
    if err != nil {
        fmt.Println("set  err=", err)
        return
    }

    //3. 通过 go 向 redis 读取数据 string [key-val]

    r, err := redis.String(conn.Do("Get", "name"))
    if err != nil {
        fmt.Println("set  err=", err)
        return
    }

    //因为返回 r 是 interface{}
    //因为 name 对应的值是 string ,因此我们需要转换
    //nameString := r.(string)

    fmt.Println("操作 ok ", r)
}
```

19.5.3 操作 Hash

说明: 通过 Golang 对 Redis 操作 Hash 数据类型

对 hash 数据结构，field-val 是一个一个放入和读取

代码：

```
package main
import (
    "fmt"
    "github.com/garyburd/redigo/redis" //引入 redis 包
)

func main() {
    //通过 go 向 redis 写入数据和读取数据
    //1. 链接到 redis
    conn, err := redis.Dial("tcp", "127.0.0.1:6379")
    if err != nil {
        fmt.Println("redis.Dial err=", err)
        return
    }
    defer conn.Close() //关闭..

    //2. 通过 go 向 redis 写入数据 string [key-val]
    _, err = conn.Do("HSet", "user01", "name", "john")
    if err != nil {
        fmt.Println("hset    err=", err)
        return
    }
```

```
_, err = conn.Do("HSet", "user01", "age", 18)
if err != nil {
    fmt.Println("hset    err=", err)
    return
}

//3. 通过 go 向 redis 读取数据

r1, err := redis.String(conn.Do("HGet","user01", "name"))
if err != nil {
    fmt.Println("hget    err=", err)
    return
}

r2, err := redis.Int(conn.Do("HGet","user01", "age"))
if err != nil {
    fmt.Println("hget    err=", err)
    return
}

//因为返回 r 是 interface{}
//因为 name 对应的值是 string ,因此我们需要转换
//nameString := r.(string)

fmt.Printf("操作 ok r1=%v r2=%v \n", r1, r2)
```

```
    }
```

对 hash 数据结构，field-val 是批量放入和读取

```
package main
import (
    "fmt"
    "github.com/garyburd/redigo/redis" //引入 redis 包
)

func main() {
    //通过 go 向 redis 写入数据和读取数据
    //1. 链接到 redis
    conn, err := redis.Dial("tcp", "127.0.0.1:6379")
    if err != nil {
        fmt.Println("redis.Dial err=", err)
        return
    }
    defer conn.Close() //关闭..

    //2. 通过 go 向 redis 写入数据 string [key-val]
    _, err = conn.Do("HMSet", "user02", "name", "john", "age", 19)
    if err != nil {
        fmt.Println("HMSet    err=", err)
        return
    }
```

```
    //3. 通过 go 向 redis 读取数据

    r, err := redis.Strings(conn.Do("HMGet","user02", "name", "age"))
    if err != nil {
        fmt.Println("hget   err=", err)
        return
    }
    for i, v := range r {
        fmt.Printf("r[%d]=%s\n", i, v)
    }

}
```

19.5.4 批量 Set/Get 数据

说明: 通过 Golang 对 Redis 操作，一次操作可以 Set / Get 多个 key-val 数据

核心代码:

```
    _, err = c.Do("MSet", "name", "尚硅谷", "address", "北京昌平~")
    r, err := redis.Strings(c.Do("MGet", "name", "address"))

    for _, v := range r {
        fmt.Println(v)
```

}

19.5.5 给数据设置有效时间

说明: 通过 Golang 对 Redis 操作，给 key-value 设置有效时间

核心代码:

```
//给 name 数据设置有效时间为 10s
  _, err = c.Do("expire", "name", 10)
```

19.5.6 操作 List

说明: 通过 Golang 对 Redis 操作 List 数据类型

核心代码:

```
_, err = c.Do("lpush", "heroList", "no1:宋江", 30, "no2:卢俊义", 28)
r, err := redis.String(c.Do("rpop", "heroList"))
```

19.5.7 Redis 链接池

- 说明: 通过 Golang 对 Redis 操作， 还可以通过 Redis 链接池, 流程如下：
  1) 事先初始化一定数量的链接，放入到链接池
  2) 当 Go 需要操作 Redis 时，**直接从 Redis 链接池取出链接**即可。
  3) 这样可以节省临时**获取 Redis 链接**的时间，从而提高效率.
  4) 示意图

5) 链接池使用的案例

```
package main
import (
    "fmt"
    "github.com/garyburd/redigo/redis"
)

//定义一个全局的 pool
var pool *redis.Pool

//当启动程序时，就初始化连接池
func init() {

    pool = &redis.Pool{
```

```
            MaxIdle: 8, //最大空闲链接数
            MaxActive: 0, //  表示和数据库的最大链接数，  0  表示没有限制
            IdleTimeout: 100, //  最大空闲时间
            Dial: func() (redis.Conn, error) { //  初始化链接的代码，  链接哪个 ip 的 redis
            return redis.Dial("tcp", "localhost:6379")
            },
        }

}

func main() {
        //先从 pool  取出一个链接
        conn := pool.Get()
        defer conn.Close()

        _, err := conn.Do("Set", "name", "汤姆猫~~")
        if err != nil {
            fmt.Println("conn.Do err=", err)
            return
        }

        //取出
        r, err := redis.String(conn.Do("Get", "name"))
        if err != nil {
            fmt.Println("conn.Do err=", err)
            return
```

```
}

fmt.Println("r=", r)

//如果我们要从 pool 取出链接，一定保证链接池是没有关闭
//pool.Close()
conn2 := pool.Get()

_, err = conn2.Do("Set", "name2", "汤姆猫~~2")
if err != nil {
    fmt.Println("conn.Do err~~~~=", err)
    return
}

//取出
r2, err := redis.String(conn2.Do("Get", "name2"))
if err != nil {
    fmt.Println("conn.Do err=", err)
    return
}

fmt.Println("r=", r2)

//fmt.Println("conn2=", conn2)
```

```
}
```

第 20 章数据结构

20.1 数据结构(算法)的介绍

➢ 数据结构的介绍

1) 数据结构是一门研究算法的学科，只从有了编程语言也就有了数据结构.学好数据结构可以编写出更加漂亮,更加有效率的代码。

2) 要学习好数据结构就要多多考虑如何将生活中遇到的问题,用程序去实现解决.

3) 程序 = 数据结构 + 算法

20.2 数据结构和算法的关系

➢ 算法是程序的灵魂，为什么有些网站能够在高并发，和海量吞吐情况下依然坚如磐石,大家可能会说: 网站使用了服务器群集技术、数据库读写分离和缓存技术(比如 Redis 等),那如果我再深入的问一句，这些优化技术又是怎样被那些天才的技术高手设计出来的呢?

➢ 大家请思考一个问题，是什么让不同的人写出的代码从功能看是一样的，但从效率上却有天壤之别,拿在公司工作的实际经历来说, 我是做服务器的，环境是 UNIX，功能是要支持上千万人同时在线，并保证数据传输的稳定, 在服务器上线前，做内测，一切 OK,可上线后，服务器就支撑不住了, 公司的 CTO 对我的代码进行优化，再次上线，坚如磐石。那一瞬间,我认识到程序是有灵魂的，就是算法。如果你不想永远都是代码工人,那就花时间来研究下算法吧!

➢ 本章着重讲解算法的基石-数据结构。

20.3 看几个实际编程中遇到的问题

Golang代码:

```
func main() {
var str string = "go,go, hello,world!"
str = strings.Replace(str, "go", "尚硅谷", -1)
fmt.Println(str)
}
```

?试写出用单链表表示的字符串类及字符串结点类的定义，并依次实现它的构造函数、以及计算串长度、串赋值、判断两串相等、求子串、两串连接、求子串在串中位置等7个成员函数。

引入清华课程,对学生要求，看:第3章习题(链表)

一个五子棋程序:

?如何判断游戏的输赢，并可以完成存盘退出和继续上局的功能

约瑟夫问题(丢手帕问题)

1. Josephu 问题

Josephu 问题为：设编号为1，2，… n的n个人围坐一圈，约定编号为k（1<=k<=n）的人从1开始报数，数到m 的那个人出列，它的下一位又从1开始报数，数到m的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

提示：用一个不带头结点的循环链表来处理Josephu 问题：先构成一个有n个结点的单循环链表，然后由k结点起从1开始计数，计到m时，对应结点从链表中删除，然后再从被删除结点的下一个结点又从1开始计数，直到最后一个结点从链表中删除算法结束。

- 邮差问题
- 最短路径问题
- **汉诺塔**
- 八皇后问题

其它常见算法问题:

- 邮差问题
- 最短路径问题
- 汉诺塔
- 八皇后问题

20.4 稀疏 sparsearray 数组

20.4.1 先看一个实际的需求

- 编写的五子棋程序中，有存盘退出和续上盘的功能

- 分析按照原始的方式来的二维数组的问题

  因为该二维数组的很多值是默认值 0, 因此记录了很多没有意义的数据

20.4.2 基本介绍

当一个数组中大部分元素为 0 ，或者为同一个值的数组时，可以使用稀疏数组来保存该数组。

稀疏数组的处理方法是:

1) 记录数组一共有几行几列，有多少个不同的值

2) 思想：把具有不同值的元素的行列及值记录在一个小规模的数组中，从而**缩小程序**的规模

20.4.3 稀疏数组举例说明

基本介绍

➢ 稀疏数组举例说明

$$\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	行 (row)	列 (col)	值 (value)
[0]	0	3	22
[1]	0	6	15
[2]	1	1	11
[3]	1	5	17
[4]	2	3	-6
[5]	3	5	39
[6]	4	0	91
[7]	5	2	28

20.4.4 应用实例

1) 使用稀疏数组，来保留类似前面的二维数组(棋盘、地图等等)

2) 把稀疏数组存盘，并且可以从新恢复原来的二维数组数

3) 整体思路分析

4) 代码实现

```
package main
import (
    "fmt"
)

type ValNode struct {
    row int
    col int
    val int
}

func main() {

    //1. 先创建一个原始数组
    var chessMap [11][11]int
    chessMap[1][2] = 1 //黑子
    chessMap[2][3] = 2 //蓝子
```

```
//2. 输出看看原始的数组
for _, v := range chessMap {
    for _, v2 := range v {
        fmt.Printf("%d\t", v2)
    }
    fmt.Println()
}

//3. 转成稀疏数组。想-> 算法
// 思路
//(1). 遍历 chessMap, 如果我们发现有一个元素的值不为 0，创建一个 node 结构体
//(2). 将其放入到对应的切片即可

var sparseArr []ValNode

//标准的一个稀疏数组应该还有一个 记录元素的二维数组的规模(行和列，默认值)
//创建一个 ValNode 值结点
valNode := ValNode{
    row : 11,
    col : 11,
    val : 0,
}

sparseArr = append(sparseArr, valNode)
```

```
for i, v := range chessMap {
    for j, v2 := range v {
        if v2 != 0 {
            //创建一个 ValNode 值结点
            valNode := ValNode{
                row : i,
                col : j,
                val : v2,
            }
            sparseArr = append(sparseArr, valNode)
        }
    }

}

//输出稀疏数组
fmt.Println("当前的稀疏数组是:::::")
for i, valNode := range sparseArr {
    fmt.Printf("%d: %d %d %d\n", i, valNode.row, valNode.col, valNode.val)
}

//将这个稀疏数组，存盘 d:/chessmap.data

//如何恢复原始的数组

//1. 打开这个 d:/chessmap.data => 恢复原始数组.
```

```
    //2. 这里使用稀疏数组恢复



    // 先创建一个原始数组
    var chessMap2 [11][11]int

    // 遍历 sparseArr [遍历文件每一行]
    for i, valNode := range sparseArr {
        if i != 0 { //跳过第一行记录值
            chessMap2[valNode.row][valNode.col] = valNode.val
        }
    }

    // 看看 chessMap2 是不是恢复.
    fmt.Println("恢复后的原始数据.....")
    for _, v := range chessMap2 {
        for _, v2 := range v {
            fmt.Printf("%d\t", v2)
        }
        fmt.Println()
    }
}
```

- 对老师的稀疏数组的改进

  1) 将构建的稀疏数组，存盘 chessmap.data

2) 在恢复原始二维数组，要求从文件 chessmap.data 读取。

20.5 队列(queue)

20.5.1 队列的应用场景

银行排队的案例:

20.5.2 队列介绍

- 队列是一个有序列表，可以用**数组**或是**链表**来实现。
- 遵循**先入先出**的原则。即：先存入队列的数据，要先取出。后存入的要后取出
- 示意图：(使用数组模拟队列示意图)

20.5.3 数组模拟队列

- 队列本身是有序列表，若使用数组的结构来存储队列的数据，则队列数组的声明如下 其中 maxSize 是该队列的最大容量。

- 因为队列的输出、输入是分别从前后端来处理，因此需要两个变量 front及 rear分别记录**队列前后端**的下标，front 会随着数据输出而改变，而 rear则是随着数据输入而改变，如图所示:

- 先完成一个非环形的队列(数组来实现)

➢ 当我们将数据存入队列时称为”addqueue”，addqueue 的处理需要有两个步骤：

1) 将尾指针往后移：rear+1 , front == rear 【空】
2) 若尾指引 rear 小于等于队列的最大下标 MaxSize-1，则将数据存入 rear所指的数组元素中，否则无法存入数据。 rear == MaxSize - 1[队列满]

思路分析：

代码实现：

```
package main
import (
    "fmt"
    "os"
    "errors"
```

```
)

//使用一个结构体管理队列
type Queue struct {
    maxSize int
    array [5]int //  数组=>模拟队列
    front int //  表示指向队列首
    rear int //  表示指向队列的尾部
}

//添加数据到队列
func (this *Queue) AddQueue(val int) (err error) {

    //先判断队列是否已满
    if this.rear == this.maxSize - 1 { //重要重要的提示; rear  是队列尾部(含最后元素)
        return errors.New("queue full")
    }

    this.rear++ //rear  后移
    this.array[this.rear] = val
    return
}

//从队列中取出数据
func (this *Queue) GetQueue() (val int, err error) {
    //先判断队列是否为空
```

```
    if this.rear == this.front { //队空
        return -1, errors.New("queue empty")
    }
    this.front++
    val = this.array[this.front]
    return val ,err
}

//显示队列, 找到队首，然后到遍历到队尾
//
func (this *Queue) ShowQueue() {
    fmt.Println("队列当前的情况是:")
    //this.front  不包含队首的元素
    for i := this.front + 1; i <= this.rear; i++ {
        fmt.Printf("array[%d]=%d\t", i, this.array[i])
    }
    fmt.Println()
}



//编写一个主函数测试，测试
func main() {

    //先创建一个队列
    queue := &Queue{
```

```
        maxSize : 5,
        front : -1,
        rear : -1,

    }

    var key string
    var val int
    for {
        fmt.Println("1. 输入 add 表示添加数据到队列")
        fmt.Println("2. 输入 get 表示从队列获取数据")
        fmt.Println("3. 输入 show 表示显示队列")
        fmt.Println("4. 输入 exit 表示显示队列")

        fmt.Scanln(&key)
        switch key {
            case "add":
                fmt.Println("输入你要入队列数")
                fmt.Scanln(&val)
                err := queue.AddQueue(val)
                if err != nil {
                    fmt.Println(err.Error())
                } else {

                    fmt.Println("加入队列 ok")
                }
```

```
            case "get":
                val, err := queue.GetQueue()
                if err != nil {
                    fmt.Println(err.Error())
                } else {
                    fmt.Println("从队列中取出了一个数=", val)
                }
            case "show":
                queue.ShowQueue()
            case "exit":
                os.Exit(0)
        }
    }
}
```

对上面代码的小结和说明：

1） 上面代码实现了基本队列结构，但是**没有有效的利用数组**空间

2） 请思考，如何使用数组 实现一个环形的队列

20.5.4 数组模拟环形队列

对前面的数组模拟队列的优化，充分利用数组.因此将数组看做是一个环形的。(通过取模的方式来实现即可)

提醒：

1) 尾索引的下一个为头索引时表示队列满，即将**队列容量空出一个作为约定**,这个在做判断队列满的时候需要注意 (tail + 1) % maxSize == head 满]
2) tail == head [空]
3) 测试示意图:

```go
}

func (cq *CircleQueue) Push(val int) (err error) {

    if cq.IsFull() {
        return errors.New("queue full")
    }
    cq.array[cq.tail] = val
    cq.tail = (cq.tail + 1) % cq.maxSize
    return
}
```

```go
func (cq *CircleQueue) Pop() (val int, err error) {

    if cq.IsEmpty() {
        return -1, errors.New("queue empty")
    }
    val = cq.array[cq.head]
    cq.head = (cq.head + 1) % cq.maxSize
```

分析思路:

1) 什么时候表示队列满 **(tail + 1) % maxSize** = hed
2) **tail == head** 表示空
3) 初始化时， tail = 0 head = 0
4) 怎么统计该队列有多少个元素 (tail + maxSize - head) % maxSize

代码实现：

```go
package main
import (
    "fmt"
    "errors"
    "os"
)

//使用一个结构体管理环形队列
type CircleQueue struct {
    maxSize int // 4
    array [5]int //  数组
    head    int //指向队列队首  0
```

```
    tail int    //指向队尾 0
}



//如队列 AddQueue(push)    GetQueue(pop)
//入队列
func (this *CircleQueue) Push(val int)    (err error) {
    if this.IsFull() {
        return errors.New("queue full")
    }
    //分析出 this.tail 在队列尾部，但是包含最后的元素
    this.array[this.tail] = val //把值给尾部
    this.tail = (this.tail + 1) % this.maxSize
    return
}

//出队列
func (this *CircleQueue) Pop() (val int, err error) {

    if this.IsEmpty() {
        return 0, errors.New("queue empty")
    }
    //取出,head 是指向队首，并且含队首元素
    val = this.array[this.head]
    this.head = (this.head + 1) % this.maxSize
    return
```

```
}

//显示队列
func (this *CircleQueue) ListQueue() {

    fmt.Println("环形队列情况如下：")
    //取出当前队列有多少个元素
    size := this.Size()
    if size == 0 {
        fmt.Println("队列为空")
    }

    //设计一个辅助的变量，指向 head
    tempHead := this.head
    for i := 0; i < size; i++ {
        fmt.Printf("arr[%d]=%d\t", tempHead, this.array[tempHead])
        tempHead = (tempHead + 1) % this.maxSize
    }
    fmt.Println()
}

//判断环形队列为满
func (this *CircleQueue) IsFull() bool {
    return (this.tail + 1) % this.maxSize == this.head
}
//判断环形队列是空
```

```
func (this *CircleQueue) IsEmpty() bool {
    return this.tail == this.head
}
//取出环形队列有多少个元素
func (this *CircleQueue) Size() int {
    //这是一个关键的算法.
    return (this.tail + this.maxSize - this.head ) % this.maxSize
}



func main() {

    //初始化一个环形队列
    queue := &CircleQueue{
        maxSize : 5,
        head : 0,
        tail : 0,
    }

    var key string
    var val int
    for {
        fmt.Println("1. 输入 add 表示添加数据到队列")
        fmt.Println("2. 输入 get 表示从队列获取数据")
        fmt.Println("3. 输入 show 表示显示队列")
        fmt.Println("4. 输入 exit 表示显示队列")
```

```
    fmt.Scanln(&key)
    switch key {
        case "add":
            fmt.Println("输入你要入队列数")
            fmt.Scanln(&val)
            err := queue.Push(val)
            if err != nil {
                fmt.Println(err.Error())
            } else {

                fmt.Println("加入队列 ok")
            }
        case "get":
            val, err := queue.Pop()
            if err != nil {
                fmt.Println(err.Error())
            } else {
                fmt.Println("从队列中取出了一个数=", val)
            }
        case "show":
            queue.ListQueue()
        case "exit":
            os.Exit(0)
    }
}
```

```
}
```

20.6 链表

20.6.1 链表介绍

➢ 链表是有序的列表，但是它在内存中是存储如下

20.6.2 单链表的介绍

➢ 单链表的示意图：

- 说明：一般来说，为了比较好的对单链表进行增删改查的操作，我们都会给他设置一个头结点，头结点的作用主要是用来标识链表头， 本身这个结点不存放数据。

20.6.3 单链表的应用实例

- 案例的说明：

  使用带 head 头的单向链表实现 – 水浒英雄排行榜管理

  完成对英雄人物的增删改查操作， 注：删除和修改,查找可以考虑学员独立完成
- 第一种方法在添加英雄时，直接添加到链表的尾部
- 代码实现：

```
package main
import (
    "fmt"
)

//定义一个 HeroNode
type HeroNode struct {
    no        int
    name      string
    nickname string
```

```
    next        *HeroNode //这个表示指向下一个结点
}

//给链表插入一个结点
//编写第一种插入方法，在单链表的最后加入.[简单]
func InsertHeroNode(head *HeroNode, newHeroNode *HeroNode) {
    //思路
    //1. 先找到该链表的最后这个结点
    //2. 创建一个辅助结点[跑龙套，帮忙]
    temp := head
    for {
        if temp.next == nil { //表示找到最后
            break
        }
        temp = temp.next //  让 temp 不断的指向下一个结点
    }

    //3. 将 newHeroNode 加入到链表的最后
    temp.next = newHeroNode
}

//给链表插入一个结点
//编写第 2 种插入方法，根据 no  的编号从小到大插入..【实用】
func InsertHeroNode2(head *HeroNode, newHeroNode *HeroNode) {
    //思路
    //1. 找到适当的结点
```

```
//2. 创建一个辅助结点[跑龙套，帮忙]
temp := head
flag := true
//让插入的结点的 no，和 temp 的下一个结点的 no 比较
for {
    if temp.next == nil {//说明到链表的最后
        break
    } else if temp.next.no >= newHeroNode.no {
        //说明 newHeroNode 就应该插入到 temp 后面
        break
    } else if temp.next.no == newHeroNode.no {
        //说明我们额链表中已经有这个 no,就不然插入.
        flag = false
        break
    }
    temp = temp.next
}

if !flag {
    fmt.Println("对不起，已经存在 no=", newHeroNode.no)
    return
} else {
    newHeroNode.next = temp.next
    temp.next = newHeroNode
}
```

```
}

//显示链表的所有结点信息
func ListHeroNode(head *HeroNode) {

    //1. 创建一个辅助结点[跑龙套，帮忙]
    temp := head

    // 先判断该链表是不是一个空的链表
    if temp.next == nil {
        fmt.Println("空空如也。。。。")
        return
    }
    //2. 遍历这个链表
    for {
        fmt.Printf("[%d , %s , %s]==>", temp.next.no,
            temp.next.name, temp.next.nickname)
        //判断是否链表后
        temp = temp.next
        if temp.next == nil {
            break
        }
    }
}
```

```
func main() {

    //1. 先创建一个头结点,
    head := &HeroNode{}

    //2. 创建一个新的 HeroNode
    hero1 := &HeroNode{
        no : 1,
        name : "宋江",
        nickname : "及时雨",
    }

    hero2 := &HeroNode{
        no : 2,
        name : "卢俊义",
        nickname : "玉麒麟",
    }

    hero3 := &HeroNode{
        no : 3,
        name : "林冲",
        nickname : "豹子头",
    }
```

```
    hero4 := &HeroNode{
        no : 3,
        name : "吴用",
        nickname : "智多星",
    }

    //3. 加入
    InsertHeroNode2(head, hero3)
    InsertHeroNode2(head, hero1)
    InsertHeroNode2(head, hero2)
    InsertHeroNode2(head, hero4)

    // 4. 显示
    ListHeroNode(head)
}
```

- 删除结点：

```go
//删除一个结点
func DelHerNode(head *HeroNode, id int) {
    temp := head
    flag := false
    //找到要删除结点的no，和temp的下一个结点的no比较
    for {
        if temp.next == nil {//说明到链表的最后
            break
        } else if temp.next.no == id {
            //说明我们找到了.
            flag = true
            break
        }
        temp = temp.next
    }
    if flag {//找到，删除

        temp.next = temp.next.next
    } else {
        fmt.Println("sorry，要删除的id不存在")
    }
}
```

20.6.4 双向链表的应用实例

使用带head头的**双向链表**实现 –水浒英雄排行榜管理

单向链表的缺点分析:

1) 单向链表，查找的方向只能是一个方向，而双向链表可以向前或者向后查找。
2) 单向链表不能自我删除，需要靠辅助节点 ，而双向链表，则可以**自我删除**，所以前面我们单链表删除时节点，总是找到temp的下一个节点来删除的(认真体会).

```go
func DelHerc
    temp :=
    flag :=
    for {
        if t

        }
        if t

        }
```

➢ 示意图

➢ 代码实现

```
package main
import (
    "fmt"
)

//定义一个 HeroNode
type HeroNode struct {
    no       int
    name     string
    nickname string
    pre      *HeroNode //这个表示指向前一个结点
    next     *HeroNode //这个表示指向下一个结点
}

//给双向链表插入一个结点
//编写第一种插入方法，在单链表的最后加入.[简单]
func InsertHeroNode(head *HeroNode, newHeroNode *HeroNode) {
```

```
    //思路
    //1. 先找到该链表的最后这个结点
    //2. 创建一个辅助结点[跑龙套，帮忙]
    temp := head
    for {
        if temp.next == nil { //表示找到最后
            break
        }
        temp = temp.next //  让 temp 不断的指向下一个结点
    }

    //3. 将 newHeroNode 加入到链表的最后
    temp.next = newHeroNode
    newHeroNode.pre = temp
}

//给双向链表插入一个结点
//编写第 2 种插入方法，根据 no  的编号从小到大插入..【实用】
func InsertHeroNode2(head *HeroNode, newHeroNode *HeroNode) {
    //思路
    //1. 找到适当的结点
    //2. 创建一个辅助结点[跑龙套，帮忙]
    temp := head
    flag := true
    //让插入的结点的 no，和 temp 的下一个结点的 no 比较
    for {
```

```
        if temp.next == nil {//说明到链表的最后
            break
        } else if temp.next.no >= newHeroNode.no {
            //说明 newHeroNode  就应该插入到 temp 后面
            break
        } else if temp.next.no == newHeroNode.no {
            //说明我们额链表中已经有这个 no,就不然插入.
            flag = false
            break
        }
        temp = temp.next
    }

    if !flag {
        fmt.Println("对不起，已经存在 no=", newHeroNode.no)
        return
    } else {
        newHeroNode.next = temp.next //ok
        newHeroNode.pre = temp//ok
        if temp.next != nil {
            temp.next.pre = newHeroNode //ok
        }
        temp.next = newHeroNode //ok
    }
```

```
}
//删除一个结点[双向链表删除一个结点]
func DelHerNode(head *HeroNode, id int) {
    temp := head
    flag := false
    //找到要删除结点的 no，和 temp 的下一个结点的 no 比较
    for {
        if temp.next == nil {//说明到链表的最后
            break
        } else if temp.next.no == id {
            //说明我们找到了.
            flag = true
            break
        }
        temp = temp.next
    }
    if flag {//找到，删除
        temp.next = temp.next.next //ok
        if temp.next != nil {
            temp.next.pre = temp
        }
    } else {
        fmt.Println("sorry，要删除的 id 不存在")
    }
}
```

```
//显示链表的所有结点信息
//这里仍然使用单向的链表显示方式
func ListHeroNode(head *HeroNode) {

    //1. 创建一个辅助结点[跑龙套，帮忙]
    temp := head

    // 先判断该链表是不是一个空的链表
    if temp.next == nil {
        fmt.Println("空空如也。。。。")
        return
    }
    //2. 遍历这个链表
    for {
        fmt.Printf("[%d , %s , %s]==>", temp.next.no,
            temp.next.name, temp.next.nickname)
        //判断是否链表后
        temp = temp.next
        if temp.next == nil {
            break
        }
    }
}


func ListHeroNode2(head *HeroNode) {
```

```
//1. 创建一个辅助结点[跑龙套，帮忙]
temp := head

// 先判断该链表是不是一个空的链表
if temp.next == nil {
    fmt.Println("空空如也。。。。")
    return
}

//2. 让 temp 定位到双向链表的最后结点
for {
    if temp.next == nil {
        break
    }
    temp = temp.next
}


//2. 遍历这个链表
for {
    fmt.Printf("[%d , %s , %s]==>", temp.no,
        temp.name, temp.nickname)
    //判断是否链表头
    temp = temp.pre
    if temp.pre == nil {
```

```
                break
            }
        }
}




func main() {

    //1. 先创建一个头结点,
    head := &HeroNode{}

    //2. 创建一个新的 HeroNode
    hero1 := &HeroNode{
        no : 1,
        name : "宋江",
        nickname : "及时雨",
    }

    hero2 := &HeroNode{
        no : 2,
        name : "卢俊义",
        nickname : "玉麒麟",
    }

    hero3 := &HeroNode{
```

```
        no : 3,
        name : "林冲",
        nickname : "豹子头",
    }


    InsertHeroNode(head, hero1)
    InsertHeroNode(head, hero2)
    InsertHeroNode(head, hero3)
    ListHeroNode(head)
    fmt.Println("逆序打印")
    ListHeroNode2(head)

}
```

20.6.5 单向环形链表的应用场景

丢手帕问题

Josephu 问题

Josephu 问题为：设编号为1，2，… n的n个人围坐一圈，约定编号为k（1<=k<=n）的人从1开始报数，数到m 的那个人出列，它的下一位又从1开始报数，数到m的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

提示：用一个不带头结点的循环链表来处理Josephu 问题：先构成一个有n个结点的单循环链表，然后由k结点起从1开始计数，计到m时，对应结点从链表中删除，然后再从被删除结点的下一个结点又从1开始计数，直到最后一个结点从链表中删除算法结束。

20.6.6 环形单向链表介绍

20.6.7 环形的单向链表的案例

完成对单向环形链表的**添加结点**，**删除结点**和**显示**.

```
package main
import (
    "fmt"
)

//定义猫的结构体结点
type CatNode struct {
    no int //猫猫的编号
    name string
    next *CatNode
}

func InsertCatNode(head *CatNode, newCatNode *CatNode) {
```

```
    //判断是不是添加第一只猫
    if head.next == nil {
        head.no = newCatNode.no
        head.name = newCatNode.name
        head.next = head //构成一个环形
        fmt.Println(newCatNode, "加入到环形的链表")
        return
    }

    //定义一个临时变量，帮忙,找到环形的最后结点
    temp := head
    for {
        if temp.next == head {
            break
        }
        temp = temp.next
    }
    //加入到链表中
    temp.next = newCatNode
    newCatNode.next = head

}

//输出这个环形的链表
func ListCircleLink(head *CatNode) {
```

```
    fmt.Println("环形链表的情况如下：")
    temp := head
    if temp.next == nil {
        fmt.Println("空空如也的环形链表...")
        return
    }
    for {
        fmt.Printf("猫的信息为=[id=%d name=%s] ->\n", temp.no, temp.name)
        if temp.next == head {
            break
        }
        temp = temp.next
    }
}

//删除一只猫
func DelCatNode(head *CatNode, id int) *CatNode {

    temp := head
    helper := head
    //空链表
    if temp.next == nil {
        fmt.Println("这是一个空的环形链表，不能删除")
        return head
    }
```

```
//如果只有一个结点
if temp.next == head { //只有一个结点
    if temp.no == id {
        temp.next = nil
    }
    return head
}

//将 helper  定位到链表最后
for {
    if helper.next == head {
        break
    }
    helper = helper.next
}

//如果有两个包含两个以上结点
flag := true
for {
    if temp.next == head { //如果到这来，说明我比较到最后一个【最后一个还没比较】
        break
    }
    if temp.no ==id {
        if temp == head { //说明删除的是头结点
            head = head.next
        }
```

```
                //恭喜找到., 我们也可以在直接删除
                helper.next = temp.next
                fmt.Printf("猫猫=%d\n", id)
                flag = false
                break
            }
            temp = temp.next //移动 【比较】
            helper = helper.next //移动 【一旦找到要删除的结点 helper】
        }
        //这里还有比较一次
        if flag { //如果 flag 为真，则我们上面没有删除
            if temp.no == id {
                helper.next = temp.next
                fmt.Printf("猫猫=%d\n", id)
            }else {
                fmt.Printf("对不起，没有 no=%d\n", id)
            }
        }
        return head
    }

    func main() {

        //这里我们初始化一个环形链表的头结点
        head := &CatNode{}
```

```
    //创建一只猫
    cat1 := &CatNode{
        no : 1,
        name : "tom",
    }
    cat2 := &CatNode{
        no : 2,
        name : "tom2",
    }
    cat3 := &CatNode{
        no : 3,
        name : "tom3",
    }
    InsertCatNode(head, cat1)
    InsertCatNode(head, cat2)
    InsertCatNode(head, cat3)
    ListCircleLink(head)

    head = DelCatNode(head, 30)

    fmt.Println()
    fmt.Println()
    fmt.Println()
    ListCircleLink(head)

}
```

作业:

20.6.8 环形单向链表的应用实例

➢ Josephu 问题

Josephu 问题为：设编号为 1，2，… n 的 n 个人围坐一圈，约定编号为 k（1<=k<=n）的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

➢ 提示

用一个不带头结点的循环链表来处理 Josephu 问题：先构成一个有 n 个结点的单循环链表，然后由 k 结点起从 1 开始计数，计到 m 时，对应结点从链表中删除，然后再从被删除结点的下一个结点又从 1 开始计数，直到最后一个结点从链表中删除算法结束。

➢ 示意图说明

➢ 走代码：

```
package main
import (
    "fmt"
)


//小孩的结构体
type Boy struct {
    No int //  编号
    Next *Boy //  指向下一个小孩的指针[默认值是 nil]
}

//  编写一个函数，构成单向的环形链表
// num  ：表示小孩的个数
// *Boy :  返回该环形的链表的第一个小孩的指针
func AddBoy(num int) *Boy {

    first := &Boy{} //空结点
    curBoy := &Boy{} //空结点

    //判断
    if num < 1    {
        fmt.Println("num 的值不对")
        return first
    }
```

```
    //循环的构建这个环形链表
    for i := 1; i <= num; i++ {
        boy := &Boy{
            No : i,
        }
        //分析构成循环链表，需要一个辅助指针[帮忙的]
        //1. 因为第一个小孩比较特殊
        if i == 1 { //第一个小孩
            first = boy //不要动
            curBoy = boy
            curBoy.Next = first //
        } else {
            curBoy.Next = boy
            curBoy = boy
            curBoy.Next = first //构造环形链表
        }
    }
    return first

}

//显示单向的环形链表[遍历]
func ShowBoy(first *Boy) {

    //处理一下如果环形链表为空
    if first.Next == nil {
```

```
        fmt.Println("链表为空，没有小孩...")
        return
    }

    //创建一个指针，帮助遍历.[说明至少有一个小孩]
    curBoy := first
    for {
        fmt.Printf("小孩编号=%d ->", curBoy.No)
        //退出的条件?curBoy.Next == first
        if curBoy.Next == first {
            break
        }
        //curBoy 移动到下一个
        curBoy = curBoy.Next
    }
}

/*
设编号为 1，2，… n 的 n 个人围坐一圈，约定编号为 k（1<=k<=n）
的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，
数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列
*/

//分析思路
//1. 编写一个函数，PlayGame(first *Boy, startNo int, countNum int)
//2. 最后我们使用一个算法，按照要求，在环形链表中留下最后一个人
```

```
func PlayGame(first *Boy, startNo int, countNum int) {

    //1. 空的链表我们单独的处理
    if first.Next == nil {
        fmt.Println("空的链表，没有小孩")
        return
    }
    //留一个，判断 startNO <= 小孩的总数
    //2. 需要定义辅助指针，帮助我们删除小孩
    tail := first
    //3. 让 tail 执行环形链表的最后一个小孩,这个非常的重要
    //因为 tail 在删除小孩时需要使用到.
    for {
        if tail.Next == first { //说明 tail 到了最后的小孩
            break
        }
        tail = tail.Next
    }
    //4. 让 first 移动到 startNo [后面我们删除小孩，就以 first 为准]
    for i := 1; i <= startNo - 1; i++ {
        first = first.Next
        tail = tail.Next
    }
    fmt.Println()
    //5. 开始数 countNum, 然后就删除 first 指向的小孩
    for {
```

```
        //开始数 countNum-1 次
        for i := 1; i <= countNum -1; i++ {
            first = first.Next
            tail = tail.Next
        }
        fmt.Printf("小孩编号为%d  出圈  \n", first.No)
        //删除 first 执行的小孩
        first = first.Next
        tail.Next = first
        //判断如果  tail == first,  圈子中只有一个小孩.
        if tail == first {
            break
        }
    }
    fmt.Printf("小孩小孩编号为%d  出圈  \n", first.No)

}

func main() {

    first := AddBoy(500)
    //显示
    ShowBoy(first)
    PlayGame(first, 20, 31)

}
```

20.7 排序

20.7.1 排序的介绍

排序是将一组数据，依指定的顺序进行排列的过程，常见的排序：

1) 冒泡排序
2) 选择排序
3) 插入排序
4) 快速排序

20.7.2 冒泡排序

冒泡排序（Bubble Sorting）的基本思想是：通过对待排序序列从后向前（从下标较大的元素开始），依次比较相邻元素的排序码，若发现逆序则交换，使排序码较小的元素逐渐从后部移向前部（从下标较大的单元移向下标较小的单元），就象水底下的气泡一样逐渐向上冒。

因为排序的过程中，各元素不断接近自己的位置，如果一趟比较下来没有进行过交换，就说明序列有序，因此要在排序过程中设置一个标志flag判断元素是否进行过交换。从而减少不必要的比较。

冒泡排序，在讲解数组时，已经讲过了。

20.7.3 选择排序基本介绍

选择式排序也属于内部排序法，是从欲排序的数据中，按指定的规则选出某一元素，经过和其他元素重整，再依原则交换位置后达到排序的目的。

20.7.4 选择排序思想:

选择排序（select sorting）也是一种简单的排序方法。它的基本思想是：第一次从 R[0]~R[n-1]中选取最小值，与 R[0]交换，第二次从 R[1]~R[n-1]中选取最小值，与 R[1]交换，第三次从 R[2]~R[n-1]中选取最小值，与 R[2]交换，…，第 i 次从 R[i-1]~R[n-1]中选取最小值，与 R[i-1]交换，…，第 n-1 次从 R[n-2]~R[n-1]中选取最小值，与 R[n-2]交换，总共通过 n-1 次，得到一个按排序码从小到大排列的有序序列。

20.7.5 选择排序的示意图

20.7.6 代码实现

```go
func SelectSort(arr *[6]int) {

    //标准的访问方式
    //(*arr)[1] = 600 等价于 arr[1] = 900
    //arr[1] = 900
    //1. 先完成将第一个最大值和 arr[0] => 先易后难

    //1 假设  arr[0] 最大值

    for j := 0; j < len(arr) - 1; j++ {

        max := arr[j]
        maxIndex := j
        //2. 遍历后面 1---[len(arr) -1] 比较
        for i := j + 1; i < len(arr); i++ {
            if max < arr[i] { //找到真正的最大值
                max = arr[i]
                maxIndex = i
            }
        }
        //交换
        if maxIndex != j {
            arr[j], arr[maxIndex] = arr[maxIndex], arr[j]
        }

        fmt.Printf("第%d次 %v\n  ", j+1 ,*arr)
```

```go
    }


```

20.7.7 插入排序法介绍:

插入式排序属于**内部排序法**，是对于欲排序的元素以插入的方式找寻该元素的适当位置，以达到排序的目的。

20.7.8 插入排序法思想:

插入排序（Insertion Sorting）的基本思想是：把 n 个待排序的元素看成为一个有序表和一个无序表，开始时有序表中只包含一个元素，无序表中包含有 n-1 个元素，排序过程中每次从无序表中取出第一个元素，把它的排序码依次与有序表元素的排序码进行比较，将它插入到有序表中的适当位置，使之成为新的有序表。

20.7.9 插入排序的示意图

20.7.10 插入排序法应用实例

有一群小牛，考试成绩分别是 23，0，12，56，34 请从大到小排序

```
for i := 1; i < len(arr); i++ {

    insertVal := arr[i]
    insertIndex := i - 1
    for insertIndex >= 0 && arr[insertIndex]
        //后移
        arr[insertIndex + 1] = arr[insertInd
        insertIndex--
    }
    if insertIndex + 1 != i {
        //插入到insertIndex + 1这个位置
```

20.7.11 插入排序的代码实现

```go
func InsertSort(arr *[7]int) {

    //完成第一次，给第二个元素找到合适的位置并插入

    for i := 1; i < len(arr); i++ {

        insertVal := arr[i]
        insertIndex := i - 1 // 下标

        //从大到小
        for insertIndex >= 0 && arr[insertIndex] < insertVal {
            arr[insertIndex + 1] = arr[insertIndex] // 数据后移
            insertIndex--
        }
        //插入
        if insertIndex + 1 != i {
            arr[insertIndex + 1] = insertVal
        }
        fmt.Printf("第%d次插入后 %v\n",i, *arr)
    }

```

20.7.12 快速排序法介绍

快速排序（Quicksort）是对冒泡排序的一种改进。基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这

两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列

20.7.13 快速排序法示意图

20.7.14 快速排序法应用实例

快速排序法应用实例：

要求：对 [-9,78,0,23,-567,70] 进行**从小到大**的排序，要求使用快速排序法。【测试8w和800w】

➢ **说明[验证分析]：**

1) **如果取消左右递归，结果是 -9 -567 0 23 78 70**
2) **如果取消右递归,结果是 -567 -9 0 23 78 70**
3) **如果取消左递归,结果是 -9 -567 0 23 70 78**

20.7.15 快速排序法的代码实现

```
package main
import (
```

```
    "fmt"
)

//快速排序
//说明
//1. left 表示 数组左边的下标
//2. right 表示数组右边的下标
//3   array 表示要排序的数组
func QuickSort(left int, right int, array *[9]int) {
    l := left
    r := right
    // pivot 是中轴，  支点
    pivot := array[(left + right) / 2]
    temp := 0

    //for 循环的目标是将比 pivot 小的数放到 左边
    //   比 pivot 大的数放到 右边
    for ; l < r; {
        //从   pivot 的左边找到大于等于 pivot 的值
        for ; array[l] < pivot; {
            l++
        }
        //从   pivot 的右边边找到小于等于 pivot 的值
        for ; array[r] > pivot; {
            r--
        }
```

```
        // l >= r  表明本次分解任务完成, break
        if l >= r {
            break
        }
        //交换
        temp = array[l]
        array[l] = array[r]
        array[r] = temp
        //优化
        if array[l]== pivot    {
            r--
        }
        if array[r]== pivot {
            l++
        }
    }
    //  如果    l== r,  再移动下
    if l == r {
        l++
        r--
    }
    //  向左递归
    if left < r {
        QuickSort(left, r, array)
    }
    //  向右递归
```

```
        if right > l {
            QuickSort(l, right, array)
        }
    }


    func main() {

        arr := [9]int {-9,78,0,23,-567,70, 123, 90, -23}
        fmt.Println("初始", arr)
        //调用快速排序
        QuickSort(0, len(arr) - 1, &arr)
        fmt.Println("main..")
        fmt.Println(arr)

    }
```

20.7.16 三种排序方法的速度的分析

20.8 栈

20.8.1 看一个实际需求

请输入一个表达式
计算式:[7*2*2-5+1-5+3-3] 点击计算

请问: 计算机底层是如何运算得到结果的？ 注意不是简单的把算式列出运算,因为我们看这个算式 7 * 2 * 2 - 5, **但是计算机怎么理解这个算式的(对计算机而言，它接收到的就是一个字符串)，我们讨论的是这个问题。-> 栈**

20.8.2 栈的介绍

有些程序员也把栈称为堆栈, 即栈和堆栈是同一个概念

1) 栈的英文为(stack)

2) 栈是一个**先入后出**(FILO-First In Last Out)的有序列表。

3) 栈(stack)是限制线性表中元素的插入和删除只能在线性表的同一端进行的一种特殊线性表。允许插入和删除的一端，为变化的一端，称为**栈顶**(Top)，**另一端为固定的一端，称为栈底**(Bottom)。

4) 根据堆栈的定义可知，最先放入栈中元素在栈底，最后放入的元素在栈顶，而删除元素刚好相反，最后放入的元素最先删除，最先放入的元素最后删除

20.8.3 栈的入栈和出栈的示意图

20.8.4 栈的应用场景

1) 子程序的调用：在跳往子程序前，会先将下个指令的地址存到堆栈中，直到子程序执行完后再将地址取出，以回到原来的程序中。

2) 处理递归调用：和子程序的调用类似，只是除了储存下一个指令的地址外，也将参数、区域变量等数据存入堆栈中。

3) 表达式的转换与求值。

4) 二叉树的遍历。

5) 图形的深度优先(depth 一 first)搜索法。

20.8.5 栈的案例

1) 用数组模拟栈的使用，由于堆栈是一种有序列表，当然可以使用数组的结构来储存栈的数据内容，下面我们就用数组模拟栈的出栈，入栈等操作。

2) 实现思路分析,并画出示意图

```go
func (this *Stack) Push(val int) (err error) {
    if this.top == this.maxTop - 1 {
        fmt.Println("stack full")
        return
    }
    this.top++
    this.arr[this.top] = val
    return
}
func (this *Stack) List() {
    if this.top == -1 {
        fmt.Println("stack empty")
        return
    }
    fmt.Println("栈的情况如下:")
    for i := this.top; i > -1; i-- {
        fmt.Printf("arr[%d]=%d\n", i, this.arr[i])
    }
}
```

```go
func (this *Stack) Pop() (val int, err
    if this.top == -1 {
        fmt.Println("stack empty")
        return -1, errors.New("stack em
    }
    val = this.arr[this.top]
    this.top--
    return val, nil
}
```

3) 对同学们加深栈的理解非常有帮助

4) **课后练习**，将老师写的程序改成可以通过终端来入栈、出栈和显示栈数据

➢ 代码实现

```go
package main
import (
    "fmt"
    "errors"
)

//使用数组来模拟一个栈的使用
type Stack struct {
    MaxTop int  // 表示我们栈最大可以存放数个数
    Top int // 表示栈顶，因为栈顶固定，因此我们直接使用Top
    arr [5]int // 数组模拟栈
}
//入栈
func (this *Stack) Push(val int) (err error) {

    //先判断栈是否满了
    if this.Top == this.MaxTop - 1 {
        fmt.Println("stack full")
        return errors.New("stack full")
    }
    this.Top++
    //放入数据
    this.arr[this.Top] = val
    return
}

```

```go
//出栈
func (this *Stack) Pop() (val int, err error) {
    //判断栈是否空
    if this.Top == -1 {
        fmt.Println("stack empty!")
        return 0, errors.New("stack empty")
    }

    //先取值，再 this.Top--
    val =  this.arr[this.Top]
    this.Top--
    return val, nil

}
//遍历栈，注意需要从栈顶开始遍历
func (this *Stack) List() {
    //先判断栈是否为空
    if this.Top == -1 {
        fmt.Println("stack empty")
        return
    }
    fmt.Println("栈的情况如下：")
    for i := this.Top; i >= 0; i-- {
        fmt.Printf("arr[%d]=%d\n", i, this.arr[i])
    }
```

20.8.6 栈实现综合计算器

- 分析了实现的思路

➢ 代码实现

```
package main
import (
    "fmt"
    "errors"
    "strconv"
)

//使用数组来模拟一个栈的使用
type Stack struct {
    MaxTop int   // 表示我们栈最大可以存放数个数
    Top int // 表示栈顶, 因为栈顶固定，因此我们直接使用 Top
    arr [20]int // 数组模拟栈
}
//入栈
```

```
func (this *Stack) Push(val int) (err error) {

    //先判断栈是否满了
    if this.Top == this.MaxTop - 1 {
        fmt.Println("stack full")
        return errors.New("stack full")
    }
    this.Top++
    //放入数据
    this.arr[this.Top] = val
    return
}

//出栈
func (this *Stack) Pop() (val int, err error) {
    //判断栈是否空
    if this.Top == -1 {
        fmt.Println("stack empty!")
        return 0, errors.New("stack empty")
    }

    //先取值，再 this.Top--
    val =   this.arr[this.Top]
    this.Top--
    return val, nil
```

```
}
//遍历栈，注意需要从栈顶开始遍历
func (this *Stack) List() {
    //先判断栈是否为空
    if this.Top == -1 {
        fmt.Println("stack empty")
        return
    }
    fmt.Println("栈的情况如下：")
    for i := this.Top; i >= 0; i-- {
        fmt.Printf("arr[%d]=%d\n", i, this.arr[i])
    }

}
//判断一个字符是不是一个运算符[+, - , * , /]
func (this *Stack) IsOper(val int) bool {

    if val == 42 || val == 43 || val == 45 || val == 47 {
        return true
    } else {
        return false
    }
}

//运算的方法
func (this *Stack) Cal(num1 int, num2 int, oper int) int{
```

```
    res := 0
    switch oper {
        case 42 :
            res = num2 * num1
        case 43 :
            res = num2 + num1
        case 45 :
            res = num2 - num1
        case 47 :
            res = num2 / num1
        default :
            fmt.Println("运算符错误.")
    }
    return res
}

//编写一个方法，返回某个运算符的优先级[程序员定义]
//[* / => 1 + - => 0]
func (this *Stack) Priority(oper int) int {
    res := 0
    if oper == 42 || oper == 47 {
        res = 1
    } else if oper == 43 || oper == 45 {
        res = 0
    }
    return res
```

```
}

func main() {

    //数栈
    numStack := &Stack{
        MaxTop : 20,
        Top : -1,
    }
    //符号栈
    operStack := &Stack{
        MaxTop : 20,
        Top : -1,
    }

    exp := "30+30*6-4-6"
    //定义一个 index ，帮助扫描 exp
    index := 0
    //为了配合运算，我们定义需要的变量
    num1 := 0
    num2 := 0
    oper := 0
    result := 0
    keepNum := ""

    for {
```

```
//这里我们需要增加一个逻辑，
//处理多位数的问题
ch := exp[index:index+1] //  字符串.
//ch ==>"+" ===> 43
temp := int([]byte(ch)[0]) //  就是字符对应的 ASCiI 码
if operStack.IsOper(temp) { //  说明是符号

    //如果 operStack    是一个空栈，   直接入栈
    if operStack.Top == -1 { //空栈
        operStack.Push(temp)
    }else {
        //如果发现 opertStack 栈顶的运算符的优先级大于等于当前准备入栈的运算符的优先级
        //，就从符号栈 pop 出，并从数栈也 pop  两个数，进行运算，运算后的结果再重新入栈
        //到数栈，   当前符号再入符号栈
        if operStack.Priority(operStack.arr[operStack.Top]) >=
            operStack.Priority(temp) {
                num1, _ = numStack.Pop()
                num2, _ = numStack.Pop()
                oper, _ = operStack.Pop()
                result = operStack.Cal(num1,num2, oper)
                //将计算结果重新入数栈
                numStack.Push(result)
                //当前的符号压入符号栈
                operStack.Push(temp)

        }else {
```

```
                operStack.Push(temp)
            }

        }


    } else { //说明是数

        //处理多位数的思路
        //1.定义一个变量 keepNum string, 做拼接
        keepNum += ch
        //2.每次要向 index 的后面字符测试一下，看看是不是运算符，然后处理
        //如果已经到表达最后，直接将 keepNum
        if index == len(exp) - 1 {
            val, _ := strconv.ParseInt(keepNum, 10, 64)
            numStack.Push(int(val))
        } else {
            //向 index 后面测试看看是不是运算符 [index]
            if operStack.IsOper(int([]byte(exp[index+1:index+2])[0])) {
                val, _ := strconv.ParseInt(keepNum, 10, 64)
                numStack.Push(int(val))
                keepNum = ""
            }
        }
    }
```

```
        //继续扫描
        //先判断 index 是否已经扫描到计算表达式的最后
        if index + 1 == len(exp) {
            break
        }
        index++

    }

    //如果扫描表达式 完毕，依次从符号栈取出符号，然后从数栈取出两个数，
    //运算后的结果，入数栈，直到符号栈为空
    for {
        if operStack.Top == -1 {
            break //退出条件
        }
        num1, _ = numStack.Pop()
        num2, _ = numStack.Pop()
        oper, _ = operStack.Pop()
        result = operStack.Cal(num1,num2, oper)
        //将计算结果重新入数栈
        numStack.Push(result)

    }

    //如果我们的算法没有问题，表达式也是正确的，则结果就是 numStack 最后数
    res, _ := numStack.Pop()
```

```
    fmt.Printf("表达式%s = %v", exp, res)
}
```

20.9 递归

20.9.1 递归的一个应用场景[迷宫问题]

20.9.2 递归的概念

简单的说：第归就是函数/方法**自己调用自己**,每次调用时传入不同的变量.第归有助于编程者解决复杂的问题,同时可以让代码变得简洁。

20.9.3 递归快速入门

我列举两个小案例,来帮助大家理解递归，递归在讲函数时已经讲过(当时讲的相对比较简单)，这里在给大家回顾一下递归调用机制

1) 打印问题

2) 阶乘问题

3) 快速入门的示意图

20.9.4 递归用于解决什么样的问题

1） 各种数学问题如: 8 皇后问题 ，汉诺塔, 阶乘问题, 迷宫问题, 球和篮子的问题(google 编程大赛)

2） 将用栈解决的问题-->第归代码比较简洁

20.9.5 递归需要遵守的重要原则

1) 执行一个函数时，就创建一个新的受保护的**独立空间**(新函数栈)

2) 函数的局部变量是独立的，不会相互影响, 如果希望各个函数栈使用同一个数据，使用引用传递

3) 递归必须向**退出递归的条件逼近【程序员自己必须分析】**，否则就是无限递归，死龟了:)

4) 当一个函数执行完毕，或者遇到 return，就会返回，遵守谁调用，就将结果返回给谁，同时当函

数执行完毕或者返回时，该函数本身也会被系统销毁

20.9.6 举一个比较综合的案例,迷宫问题

➢ 走代码：

```
package main
import (
    "fmt"
)

//编写一个函数，完成老鼠找路
//myMap *[8][7]int:地图，保证是同一个地图，使用引用
//i,j 表示对地图的哪个点进行测试
func SetWay(myMap *[8][7]int, i int, j int) bool {

    //分析出什么情况下，就找到出路
    //myMap[6][5] == 2
    if myMap[6][5] == 2 {
        return true
```

```
    } else {
        //说明要继续找
        if myMap[i][j] == 0 { //如果这个点是可以探测

            //假设这个点是可以通, 但是需要探测 上下左右
            //换一个策略 下右上左
            myMap[i][j] = 2
            if SetWay(myMap, i + 1, j) { //下
                return true
            } else if SetWay(myMap, i , j + 1) { //右
                return true
            } else if SetWay(myMap, i - 1, j) { //上
                return true
            } else if SetWay(myMap, i , j - 1) { //左
                return true
            } else { // 死路
                myMap[i][j] = 3
                return false
            }

        } else { // 说明这个点不能探测，为 1，是强
            return false
        }

    }
}
```

```
func main() {
    //先创建一个二维数组，模拟迷宫
    //规则
    //1. 如果元素的值为 1 ，就是墙
    //2. 如果元素的值为 0, 是没有走过的点
    //3. 如果元素的值为 2, 是一个通路
    //4. 如果元素的值为 3，  是走过的点，但是走不通
    var myMap [8][7]int

    //先把地图的最上和最下设置为 1
    for i := 0 ; i < 7 ; i++ {
        myMap[0][i] = 1
        myMap[7][i] = 1
    }

    //先把地图的最左和最右设置为 1
    for i := 0 ; i < 8 ; i++ {
        myMap[i][0] = 1
        myMap[i][6] = 1
    }

    myMap[3][1] = 1
    myMap[3][2] = 1
    myMap[1][2] = 1
    myMap[2][2] = 1
```

```
    //输出地图
    for i := 0; i < 8; i++ {
        for j := 0; j < 7; j++ {
            fmt.Print(myMap[i][j], " ")
        }
        fmt.Println()
    }

    //使用测试
    SetWay(&myMap, 1, 1)
    fmt.Println("探测完毕....")
    //输出地图
    for i := 0; i < 8; i++ {
        for j := 0; j < 7; j++ {
            fmt.Print(myMap[i][j], " ")
        }
        fmt.Println()
    }

}
```

- 课后思考题：

  思考：如何求出最短路径?

20.10哈希表(散列)

20.10.1 实际的需求

google 公司的一个上机题:

有一个公司,当有新的员工来报道时,要求将该员工的信息加入(id,性别,年龄,住址..),当输入该员工的 id 时,要求查找到该员工的 所有信息.

要求: 不使用数据库,尽量节省内存,速度越快越好=>哈希表(散列)

20.10.2 哈希表的基本介绍

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

20.10.3 使用 hashtable 来实现一个雇员的管理系统[增删改查]

➢ 应用实例 google 公司的一个上机题:

有一个公司,当有新的员工来报道时,要求将该员工的信息加入(id,性别,年龄,住址..),当输入该员工的 id 时,要求查找到该员工的 所有信息.

➢ 要求:

1) 不使用数据库,尽量节省内存,**速度越快越好**=>哈希表(散列)

2) 添加时，保证按照雇员的 id 从低到高插入

➢ 思路分析

1) 使用链表来实现哈希表, 该**链表不带**表头

[即: 链表的第一个结点就存放雇员信息]

2) 思路分析并画出示意图

3) 代码实现[**增删改查(显示所有员工，按 id 查询)**]

```
package main
import (
    "fmt"
)

//定义 emp
type Emp struct {
    Id int
    Name string
    Next *Emp
}
//方法待定..
```

```
//定义 EmpLink
//我们这里的 EmpLink 不带表头,即第一个结点就存放雇员
type EmpLink struct {
    Head *Emp
}
//方法待定..
//1. 添加员工的方法, 保证添加时，编号从小到大
func (this *EmpLink) Insert(emp *Emp) {

    cur := this.Head // 这是辅助指针
    var pre *Emp = nil // 这是一个辅助指针 pre 在 cur 前面
    //如果当前的 EmpLink 就是一个空链表
    if cur == nil {
        this.Head = emp //完成
        return
    }
    //如果不是一个空链表,给 emp 找到对应的位置并插入
    //思路是 让 cur 和 emp 比较，然后让 pre 保持在 cur 前面
    for {
        if cur != nil {
            //比较
            if cur.Id > emp.Id {
                //找到位置
                break
            }
```

```
                pre = cur //保证同步
                cur = cur.Next
            }else {
                break
            }
        }
        //退出时，我们看下是否将 emp 添加到链表最后
        pre.Next = emp
        emp.Next = cur

    }
    //显示链表的信息
    func (this *EmpLink) ShowLink(no int) {
        if this.Head == nil {
            fmt.Printf("链表%d  为空\n", no)
            return
        }

        //变量当前的链表，并显示数据
        cur := this.Head //  辅助的指针
        for {
            if cur != nil {
                fmt.Printf("链表%d  雇员 id=%d  名字=%s ->", no, cur.Id, cur.Name)
                cur = cur.Next
            } else {
                break
```

```
            }
        }
        fmt.Println() //换行处理
    }

    //定义 hashtable ,含有一个链表数组
    type HashTable struct {
        LinkArr [7]EmpLink
    }

    //给 HashTable  编写 Insert  雇员的方法.
    func (this *HashTable) Insert(emp *Emp) {
        //使用散列函数，确定将该雇员添加到哪个链表
        linkNo := this.HashFun(emp.Id)
        //使用对应的链表添加
        this.LinkArr[linkNo].Insert(emp) //
    }

    //编写方法，显示 hashtable 的所有雇员
    func (this *HashTable) ShowAll() {
        for i := 0; i < len(this.LinkArr); i++ {
            this.LinkArr[i].ShowLink(i)
        }
    }

    //编写一个散列方法
```

```
func (this *HashTable) HashFun(id int) int {
    return id % 7 //得到一个值，就是对于的链表的下标
}


func main() {

    key := ""
    id := 0
    name := ""
    var hashtable HashTable
    for {
        fmt.Println("===============雇员系统菜单============")
        fmt.Println("input  表示添加雇员")
        fmt.Println("show   表示显示雇员")
        fmt.Println("find   表示查找雇员")
        fmt.Println("exit   表示退出系统")
        fmt.Println("请输入你的选择")
        fmt.Scanln(&key)
        switch key {
            case "input":
                fmt.Println("输入雇员 id")
                fmt.Scanln(&id)
                fmt.Println("输入雇员 name")
                fmt.Scanln(&name)
                emp := &Emp{
```

```
                Id : id,
                Name : name,
            }
            hashtable.Insert(emp)
        case "show":
            hashtable.ShowAll()
        case "exit":
        default :
            fmt.Println("输入错误")
    }
}

}
```