目录

[第一章 初识Go语言 13](#_Toc1617657359)

[1.1 Go语言的发展历史 13](#_Toc824261693)

[1.1.1 知名编程语言发展简史 13](#_Toc2096751101)

[1.1.2 Go语言主要发展过程 14](#_Toc2036590884)

[1.2 Go语言的特点及优势 14](#_Toc241137855)

[1.2.1 Go语言设计初衷 14](#_Toc502287096)

[1.2.2 Go语言的优势 15](#_Toc181006115)

[1.3 Go安装和配置 15](#_Toc1332930653)

[1.3.1 下载 15](#_Toc16079467)

[1.3.2 windows系统下安装Go 17](#_Toc1812145994)

[1.3.3 Mac系统下安装及配置环境变量 17](#_Toc1124639404)

[1.4 Goland安装和配置 18](#_Toc1810885781)

[1.4.1 安装Goland开发工具 18](#_Toc1419075983)

[1.4.2 下载及安装 18](#_Toc456662699)

[1.4.3 使用Goland 19](#_Toc23427715)

[1.4.4 编写第一个程序HelloWorld 20](#_Toc760098604)

[1.4.5 Go Module基本使用说明 20](#_Toc1744505072)

[1.5 Go语言编码规范 20](#_Toc302512613)

[1.5.1 HelloWorld程序解析 20](#_Toc1235694242)

[1.5.2 Go语言编码规范 21](#_Toc2146258804)

[1.5.3 Go语言关键字及保留字 22](#_Toc888900169)

[1.5.4 Go 程序结构组成 22](#_Toc1848891851)

[1.6 小结 22](#_Toc236967667)

[第二章Go语言基础 24](#_Toc1280897731)

[2.1 变量 24](#_Toc1672087389)

[2.1.1 变量的概念 24](#_Toc801742281)

[2.1.2 声明变量 24](#_Toc1570115489)

[2.1.3 变量初始化 25](#_Toc651969287)

[2.1.4 匿名变量 25](#_Toc1186239615)

[2.2 数据类型 25](#_Toc2038514204)

[2.2.1 整型 26](#_Toc354122390)

[2.2.2 浮点型 27](#_Toc1057822893)

[2.2.3 布尔型 28](#_Toc1959732785)

[2.2.4 字符串 28](#_Toc1272223456)

[2.2.5 字符 29](#_Toc1912435460)

[2.2.6 数组 30](#_Toc915031571)

[2.2.7 结构体 31](#_Toc805217630)

[2.2.8 指针 31](#_Toc1998247663)

[2.2.9 切片 33](#_Toc51716608)

[2.2.10 映射（map） 33](#_Toc1617637268)

[2.2.11 函数 33](#_Toc486592256)

[2.2.12 通道 34](#_Toc538318816)

[2.3 打印格式化 34](#_Toc175735701)

[2.4 数据类型转换 35](#_Toc799912082)

[2.4.1 T(表达式) 36](#_Toc874731954)

[2.4.2 float与int之间转换 36](#_Toc2094387163)

[2.4.3 int转string 36](#_Toc960590564)

[2.4.4 string转int 37](#_Toc2011034649)

[2.5常量 37](#_Toc214225610)

[2.5.1 声明方式 37](#_Toc1307234898)

[2.5.2 常量用于枚举（常量组） 38](#_Toc1939221876)

[2.5.3 iota 38](#_Toc142759413)

[2.6类型别名（Type Alias） 39](#_Toc618220592)

[2.6.1 概要 39](#_Toc907605558)

[2.6.2 类型别名与类型定义 39](#_Toc550268665)

[2.7 Go 语言运算符 40](#_Toc1300868673)

[2.7.1 算术运算符 （Arithmetic operator） 40](#_Toc168777004)

[2.7.2 关系运算符（Relational operator） 41](#_Toc1956692188)

[2.7.3 逻辑运算符（Logical operator） 42](#_Toc1708517205)

[2.7.4 位运算符（Bitwise operator） 43](#_Toc1044820398)

[2.7.5 赋值运算符（Assignment operator） 46](#_Toc322647667)

[2.7.6 其他运算符 47](#_Toc343130594)

[2.8 运算符优先级 48](#_Toc1002301163)

[2.8 小结 49](#_Toc813919473)

[第三章 Go语言流程控制 50](#_Toc73751321)

[3.1 Go 语言流程控制 50](#_Toc440387728)

[3.2 if条件判断语句 51](#_Toc1367896934)

[3.3 if嵌套语句 52](#_Toc1431328603)

[3.4 switch分支语句 53](#_Toc228016927)

[3.5 Go循环语句 56](#_Toc1169665841)

[3.5.1 for循环语法结构 57](#_Toc508485049)

[3.5.2 for循环使用案例 59](#_Toc1270787130)

[3.6 for嵌套循环语句 60](#_Toc1394424495)

[3.7 循环控制语句 63](#_Toc603447754)

[3.7.1 break语句 63](#_Toc1728620344)

[3.7.2 continue语句 64](#_Toc1763344992)

[3.7.3 goto语句 66](#_Toc1264951944)

[3.8 小结 67](#_Toc2106701155)

[第四章 函数与指针 68](#_Toc1763423996)

[4.1 函数 68](#_Toc445288525)

[4.1.1 声明函数 68](#_Toc2131213527)

[4.1.2 变量作用域 69](#_Toc1425999976)

[4.1.3 函数变量 70](#_Toc864096112)

[4.1.4 匿名函数 71](#_Toc1578933370)

[4.1.5 闭包 73](#_Toc677723611)

[4.1.6 可变参数 75](#_Toc247466389)

[4.1.7 递归函数 76](#_Toc1639259331)

[4.2 指针 77](#_Toc963868754)

[4.2.1 声明指针 78](#_Toc1272999157)

[4.2.2 空指针 80](#_Toc2064740285)

[4.2.3 操作指针改变变量的数值 81](#_Toc901718122)

[4.2.4 使用指针作为函数的参数 81](#_Toc384379575)

[4.2.5指针数组 83](#_Toc636706849)

[4.2.6 指针的指针 83](#_Toc220998142)

[4.3 函数的参数传递 84](#_Toc1316546931)

[4.3.1 值传递（传值） 84](#_Toc1680254276)

[4.3.2 引用传递（传引用） 84](#_Toc623658682)

[4.4 小结 87](#_Toc2111271014)

[第五章 Go语言内置容器 88](#_Toc1259632917)

[5.1 数组(array) 88](#_Toc756643893)

[5.1.1 数组简介 88](#_Toc1663235764)

[5.1.2 一维数组 88](#_Toc208852549)

[5.1.3 多维数组 89](#_Toc1196511845)

[5.2 切片(Slice) 90](#_Toc737708407)

[5.2.1 切片简介 90](#_Toc1242102318)

[5.2.2 切片的语法 91](#_Toc325126139)

[5.2.3 len() 和 cap() 函数 92](#_Toc1196620205)

[5.2.4 append() 和 copy() 函数 94](#_Toc411431280)

[5.2.5 切片高级操作 95](#_Toc28179620)

[5.3 冒泡排序 97](#_Toc1168471000)

[5.3.1 概念： 97](#_Toc1901628832)

[5.3.2 冒泡排序算法的原理如下： 97](#_Toc1824144770)

[5.3.3 冒泡排序分析： 97](#_Toc924604818)

[5.3.4 冒泡排序核心代码 98](#_Toc641506434)

[5.4 map 99](#_Toc1430728298)

[5.4.1 map简介 99](#_Toc876109027)

[5.4.2 map的语法 99](#_Toc1616532957)

[5.4.4 delete() 函数 101](#_Toc1253790102)

[5.5 list 102](#_Toc1340699950)

[5.5.1 概述 102](#_Toc1745635326)

[5.5.2 声明list 102](#_Toc2118822415)

[5.5.3 element常用方法 103](#_Toc1474494351)

[5.5.4 list常用方法 103](#_Toc2012754524)

[5.5.5 遍历list 104](#_Toc1202877324)

[5.5.6 list是值类型还是引用类型 104](#_Toc348131610)

[5.6 小结 106](#_Toc1302514842)

[第六章 strings包和time及math包 106](#_Toc2066135623)

[6.1 字符串处理包概述 107](#_Toc730843771)

[6.2 strings包中的字符串处理函数 108](#_Toc1832282004)

[6.2.1 检索字符串 108](#_Toc248143248)

[6.2.2 分割字符串： 109](#_Toc130326662)

[6.2.3 大小写转换： 110](#_Toc2114371941)

[6.2.4 trim修剪函数： 110](#_Toc1837305478)

[6.2.5 比较字符串 111](#_Toc925808533)

[6.3 strconv包中常用函数 111](#_Toc1544991616)

[6.3.1 Parse类函数（将字符串转其它类型） 111](#_Toc1449314235)

[6.3.2 Format类函数 112](#_Toc1864823371)

[6.4 time包 113](#_Toc1710052079)

[6.4.1 time包中核心方法介绍 113](#_Toc1071643952)

[6.5 math包 115](#_Toc174553875)

[6.5.1 math包中核心函数介绍 115](#_Toc264315323)

[6.6 随机数(math/rand包) 116](#_Toc1351451665)

[6.6.1 rand包中核心方法介绍 116](#_Toc2061082983)

[6.7 键盘输入 117](#_Toc1710469171)

[6.7.1 scanln 117](#_Toc1639258255)

[6.7.2 随机数+键盘输入案例——猜数字游戏 117](#_Toc945784422)

[6.8 小结 119](#_Toc124825460)

[第七章 Go语言面向对象编程 119](#_Toc1997466748)

[7.1 面向对象思想的概述 120](#_Toc1959263732)

[7.1.1 OOP概述 120](#_Toc1978784273)

[7.1.2 Go语言面向对象 120](#_Toc1495518869)

[7.2 结构体 121](#_Toc1037026795)

[7.2.1 定义结构体 121](#_Toc332064513)

[7.2.2 实例化结构体——为结构体分配内存并初始化 121](#_Toc1845755085)

[7.2.3 结构体中的语法糖 122](#_Toc1204432680)

[7.2.4 结构体是值类型 124](#_Toc719196138)

[7.2.5 结构体的深拷贝和浅拷贝 125](#_Toc1491526050)

[7.2.6 结构体对象或指针作为函数的参数及函数返回值 127](#_Toc501710919)

[7.2.7 匿名结构体和匿名字段 129](#_Toc1234617511)

[7.2.8 结构体嵌套 131](#_Toc1229510063)

[7.3 方法 136](#_Toc1287977407)

[7.3.1 什么是方法？ 136](#_Toc401117689)

[7.3.2 方法的语法格式： 136](#_Toc633831090)

[7.3.3 方法和函数 137](#_Toc1280240510)

[7.3.4 指针作为接受者 139](#_Toc1363592277)

[7.3.5 method继承 140](#_Toc2097402402)

[7.3.6 method重写 141](#_Toc98104909)

[7.4 什么是接口? 142](#_Toc1729248314)

[7.4.1 概念 143](#_Toc1580218547)

[7.4.2 接口的定义语法及示例 143](#_Toc802856980)

[7.4.3 duck typing 145](#_Toc977508759)

[7.4.4 多态 147](#_Toc739812963)

[7.4.5 空接口 150](#_Toc106153011)

[7.4.6 接口对象转型 152](#_Toc1702228867)

[7.5 小结 155](#_Toc583422335)

[第八章 Go语言异常处理 155](#_Toc168852143)

[8.1 错误处理 156](#_Toc1072069714)

[8.1.1 错误是什么? 156](#_Toc887884868)

[8.1.2 创建error对象的方式 156](#_Toc1964597120)

[8.1.3 自定义错误 157](#_Toc1422723215)

[8.2 延迟是什么? 158](#_Toc1626148807)

[8.2.1 延迟函数 158](#_Toc1806107527)

[8.2.2 延迟方法 159](#_Toc567855944)

[8.2.3 延迟参数 159](#_Toc537523540)

[8.2.4 堆栈的推迟 160](#_Toc1841917498)

[8.2.5 延迟的应用 160](#_Toc1130617381)

[8.3 panic和recover（宕机和宕机恢复） 162](#_Toc1351013811)

[8.3.1 panic和recover机制 162](#_Toc1144521746)

[8.3.2 案例 163](#_Toc965958843)

[8.4 小结 164](#_Toc2041386628)

[第九章 Go语言的IO操作 164](#_Toc1386312324)

[9.1 文件信息 165](#_Toc1701143165)

[9.1.1 FileInfo接口 165](#_Toc1663381644)

[9.1.2 文件路径 166](#_Toc513174062)

[9.2 文件常规操作 167](#_Toc622133682)

[9.2.1 创建目录 167](#_Toc102916131)

[9.2.2 创建文件 167](#_Toc987077882)

[9.2.3 打开文件 167](#_Toc506789699)

[9.2.4 关闭文件 167](#_Toc694327091)

[9.2.5 删除 167](#_Toc129280639)

[9.3 读写文件及复制文件 168](#_Toc1713732556)

[9.3.1 读取文件 168](#_Toc652395128)

[9.3.2 写入文件 168](#_Toc1900898361)

[9.3.3 复制文件 169](#_Toc284536908)

[9.4 ioutil包 170](#_Toc1913214534)

[9.4.1 ioutil包核心函数 170](#_Toc1124026407)

[9.4.2 示例代码 170](#_Toc98179790)

[9.5 bufio包 171](#_Toc840289634)

[9.5.1 bufio的原理 171](#_Toc895415966)

[9.5.2 bufio.Reader结构体 172](#_Toc1838226033)

[9.5.3 bufio.Writer结构体 174](#_Toc1365190889)

[9.5.4 Scanner 175](#_Toc1047986875)

[9.6 小结 176](#_Toc2002019078)

[第十章 Go语言数据库编程 176](#_Toc1160862750)

[10.1 数据库介绍 177](#_Toc731306255)

[10.2 MySQL数据库介绍 177](#_Toc1015316004)

[10.3 Go程序操作MySQL数据库 177](#_Toc511020166)

[10.3.1 安装mysql模块 178](#_Toc928825609)

[10.3.2 连接数据库 178](#_Toc713380420)

[10.3.3 增删改数据 179](#_Toc383517739)

[10.3.4 查询数据 179](#_Toc1184214726)

[10.3.5 示例代码 180](#_Toc218459486)

[10.4 gorm库 186](#_Toc1599028479)

[10.4.1 获取gorm库 186](#_Toc1261287995)

[10.4.2 gorm基本操作 186](#_Toc656252428)

[10.4.3 gorm模型 188](#_Toc158546404)

[10.4.3 gorm读写数据 191](#_Toc1809689748)

[10.4.4 206](#_Toc644702175)

[10.4.5 208](#_Toc1454456110)

[10.5 小结 210](#_Toc237486969)

[第十一章 Go语言操作缓存 210](#_Toc1418871857)

[11.1 Go语言操作Memcached 211](#_Toc1320884311)

[11.1.1 安装MemCached驱动 211](#_Toc1564155938)

[11.1.2 Go语言操作Memcached 212](#_Toc1421527039)

[11.1.3 Go语言Memcached库get操作 213](#_Toc849371598)

[11.1.4 Go语言Memcached库add操作 213](#_Toc1064645977)

[11.1.5 Go语言Memcached库 replace操作 214](#_Toc671188635)

[11.1.6 Go语言Memcached库 delete操作 214](#_Toc2083274401)

[11.1.7 Go语言Memcached库incrby操作 215](#_Toc1019476919)

[11.1.8 Go语言Memcached库decrby操作 216](#_Toc1724041867)

[11.2 GO语言操作Redis 216](#_Toc2122293345)

[11.2.1 安装redigo 216](#_Toc1828356392)

[11.2.2 Go语言Redis基本操作 217](#_Toc842375421)

[11.2.3 Go语言操作Redis连接池管理 221](#_Toc1591499723)

[11.3 Go语言操groupcache 222](#_Toc1427021076)

[11.3.1 groupcache简介 222](#_Toc845854636)

[11.3.2 groupcache运行机制 222](#_Toc2084607759)

[11.3.3 使用groupcache构建服务端 222](#_Toc1954388355)

[11.4 小结 224](#_Toc1642701620)

[第十二章 Go语言并发编程 224](#_Toc836361508)

[12.1 并发与并行 225](#_Toc1447395341)

[12.1.1 Go是并发语言，而不是并行语言 225](#_Toc1826226618)

[12.1.2 什么是并发？ 226](#_Toc1554485802)

[12.2 进程和线程 226](#_Toc2104308459)

[12.2.1 基本概念 226](#_Toc204087970)

[12.2.2 进程与线程的区别 227](#_Toc575127531)

[12.2.3 相关背景知识 227](#_Toc344518370)

[12.3 轻量级线程——协程coroutine 228](#_Toc704332278)

[12.3.1 什么是协程？ 228](#_Toc782734082)

[12.3.2 协程的应用 229](#_Toc2074378299)

[12.4 Go语言中协程——goroutine 229](#_Toc1826545895)

[12.4.1 什么是Goroutine？ 229](#_Toc478123400)

[12.4.2 Goroutine在线程上的优势 229](#_Toc2083660373)

[12.4.3 使用普通函数创建goroutine 230](#_Toc1064057382)

[12.4.4 调整并发的运行性能Gomaxprocs 232](#_Toc1516090705)

[12.5 多个goroutine间通信的通道channel 232](#_Toc1043007280)

[12.5.1 通道的概述 232](#_Toc2061828146)

[12.5.2 声明通道类型 233](#_Toc1349521830)

[12.5.3 通道发送数据 233](#_Toc1838600843)

[12.5.4 阻塞 234](#_Toc1222171618)

[12.5.5 通道接收数据 234](#_Toc357300171)

[12.5.6 关闭通道 236](#_Toc779696985)

[12.6 缓冲通道和定向通道 237](#_Toc422012901)

[12.6.1 缓冲通道 237](#_Toc1779824713)

[12.6.2 定向通道 239](#_Toc1214232328)

[12.7 time包中跟通道相关的函数 240](#_Toc65639255)

[12.7.1 Timer结构体 240](#_Toc1539847874)

[12.7.2 NewTimer函数 240](#_Toc897788321)

[12.7.3 After()函数 241](#_Toc908207225)

[12.8 select分支语句 241](#_Toc2072551346)

[12.9 sync包 243](#_Toc1185717882)

[12.9.1 WaitGroup同步等待组 243](#_Toc1859682261)

[12.9.2 互斥锁 244](#_Toc1202762189)

[12.9.3 读写互斥锁RWMutex 245](#_Toc560541312)

[12.9.4 条件变量Cond 247](#_Toc7071395)

[第十三章 Go语言加密算法 248](#_Toc737335180)

[13.1 哈希算法 249](#_Toc1411727070)

[13.1.1 Hash的定义 249](#_Toc1497533434)

[13.1.2 流行的哈希算法 249](#_Toc536082398)

[13.1.3 Hash与加密解密的区别 250](#_Toc1242964021)

[13.1.4 SHA256 253](#_Toc1922866578)

[13.1.5 核心代码 253](#_Toc137172743)

[13.2 对称加密算法 255](#_Toc1212338370)

[13.2.1 对称加密的概述 255](#_Toc446141854)

[13.2.2 DES和TripleDES算法 256](#_Toc1440728501)

[13.2.3 AES加密 256](#_Toc1445796382)

[13.2.4 AES的加密模式 257](#_Toc722326469)

[13.2.5 核心代码 257](#_Toc415907992)

[13.3 非对称加密算法 262](#_Toc106350559)

[13.3.1 非对称加密算法的发展史 262](#_Toc727450809)

[13.3.2 非对称加密（Asymmetric Cryptography） 262](#_Toc641344492)

[13.3.3 对称加密与非对称加密的区别 262](#_Toc856452751)

[13.3.4 核心代码 263](#_Toc1965983863)

[13.4 椭圆曲线加密算法ECC和椭圆曲线数字签名算法ECDSA 266](#_Toc1107392699)

[13.4.1 椭圆曲线密码学概述 266](#_Toc1855807191)

[13.4.2 数字签名 266](#_Toc498970109)

[13.4.3 ECC数字签名(ECDSA)核心代码 268](#_Toc266980428)

[13.5 字符编码/解码 270](#_Toc1046714813)

[13.5.1 Base64 270](#_Toc2097309514)

[13.5.2 Base58 271](#_Toc684419940)

[13.5.3 Base58Check 272](#_Toc1123518248)

[13.5.4 核心代码 272](#_Toc147486065)

[第十四章Socket编程 273](#_Toc602165817)

[14.1 Dial()函数 274](#_Toc1657941655)

[14.2 TCP Socket 通信 276](#_Toc1425075760)

[14.3 聊天室案例 278](#_Toc363183329)

[第十五章 HTTP编程 284](#_Toc873685729)

[15.1 HTTP协议 285](#_Toc1690352764)

[15.1.1 HTTP协议介绍 285](#_Toc697738385)

[15.1.2HTTP状态码 285](#_Toc1628324075)

[15.2 HTTP客户端 287](#_Toc1858614804)

[15.2.1 GET方法示例 287](#_Toc441881566)

[15.2.2 http.NewRequest示例 288](#_Toc705028436)

[15.2.3 http.clinet.Get示例 289](#_Toc1745643353)

[15.2.4 POST方法： 289](#_Toc106248557)

[15.2.5 http.NewRequest示例： 291](#_Toc1160586842)

[15.2.6 http.client示例： 292](#_Toc389087793)

[15.2.7 http.PostForm方法示例 292](#_Toc310831836)

[15.2.8 复杂的请求 293](#_Toc1470438148)

[15.2.9 GET方法和POST方法区别 293](#_Toc412143760)

[15.3 HTTP服务端 294](#_Toc1265412745)

[15.4 案例 297](#_Toc1261448974)

[第十六章 JSON处理 304](#_Toc1214342834)

[16.1 JSON简介 305](#_Toc1922913597)

[16.2 对象转换为JSON 305](#_Toc927421076)

[16.3 JSON转换为对象 308](#_Toc729714406)

[16.4 JSON转换为map 310](#_Toc30913625)

[16.5 JSON流解析 311](#_Toc2021736448)

第一章 初识Go语言

本章重点为大家介绍如下的内容：

* Go语言发展历史
* Go语言核心特性
* Go的安装及环境变量配置
* 安装Goland开发环境及HelloWorld
* helloworld程序解读及Go语言编码规范

1.1 Go语言的发展历史

1.1.1 知名编程语言发展简史

B语言：Ken Thompson（肯.汤普森）。该语言得名于汤姆森的妻子Bonnie，一门剑桥于60年代中期开发的语言。

C语言：美国著名计算机专家Dennis Ritchie（丹尼斯·里奇）在1969-1973年期间发明了C语言和Unix操作系统。

C++：Bjarne Stroustrup（本贾尼·斯特劳斯特卢普）。1982年，美国贝尔实验室的Bjarne Stroustrup博士在C语言的基础上引入并扩充了面向对象的概念，发明了—种新的程序语言。为了表达该语言与C语言的渊源关系，它被命名为C++。

Python：Guido von Rossum（吉多.范.罗苏姆）。1989年，为了打发圣诞节假期，Guido开始写Python语言的编译/解释器。1994年发布1.0版本。

Java：James Gosling（詹姆斯.高斯林）。1991年开发Oak，1994年更名为Java。1995年5月正式发布。

Javascript：Brendan Eich（布兰登.艾奇)。艾奇对Java一点兴趣也没有，为了应付公司安排的任务，他于1995年5月只用10天时间就把Javascript设计出来了。

Go语言：Go的三个作者分别是：Robert Griesemer（罗伯特.格利茨默）, Rob Pike（罗伯.派克） 和 Ken Thompson（肯.汤普森）。

Robert在开发Go之前是Google V8、Chubby和HotSpot JVM的主要贡献者；

Rob主要是Unix、UTF-8、plan9的作者；

Ken主要是B语言、C语言的作者、Unix之父。



图 1.1

1.1.2 Go语言主要发展过程

2007年9月，Rob Pike（罗伯.派克） 正式命名为Go；

2008年5月，Google全力支持该项目；

2009年11月，Go将代码全部开源，它获得了当年的年度语言；

2012年3月28日，Go1.0发布；

2014年6月18日，Go语言Go 1.3版发布；

2015年8月19日，Go语言Go 1.5版发布，本次更新中移除了”最后残余的C代码”；

2018年2月16日，Go语言Go 1.10版发布。

2020年2月26日，Go1.14版本发布，Go Module已经具备在生产环境中使用的条件，并鼓励所有用户迁移到Go Module进行依赖关系管理。

1.2 Go语言的特点及优势

1.2.1 Go语言设计初衷

设计Go语言是为了解决当时Google开发遇到的问题：

* 大量的C++代码，同时又引入了Java和Python
* 成千上万的工程师
* 数以万计行的代码
* 分布式的编译系统
* 数百万的服务器

Google开发中的痛点：

* 编译慢
* 失控的依赖
* 每个工程师只是用了一个语言里面的一部分
* 程序难以维护（可读性差、文档不清晰等）
* 更新的花费越来越长
* 交叉编译困难

如何解决当前的问题和痛点？

Go希望成为互联网时代的C语言。多数系统级语言（包括Java和C#）的根本编程哲学来源于C++，将C++的面向对象进一步发扬光大。但是Go语言的设计者却有不同的看法，他们认为值得学习的是C语言。C语言经久不衰的根源是它足够简单。因此，Go语言也是足够简单。

他们当时设计Go的目标是为了消除各种缓慢和笨重、改进各种低效和扩展性。Go是由那些开发大型系统的人设计的，同时也是为了这些人服务的；它是为了解决工程上的问题，不是为了研究语言设计；它还是为了让我们的编程变得更舒适和方便。

但是结合Google当时内部的一些现实情况，如很多工程师都是C系的，所以新设计的语言一定要易学习，最好是类似C的语言；20年没有出新的语言了，所以新设计的语言必须是现代化的（例如内置GC）等情况。最后根据实战经验，他们向着目标设计了Go这个语言。

Go语言的特色：

* 没有继承多态的OO
* 强一致类型
* interface不需要显式声明(Duck Typing)
* 没有异常处理(Error is value)
* 基于首字母的可访问特性
* 不用的import或者变量引起编译错误
* 完整而卓越的标准库包

1.2.2 Go语言的优势

Go语言语法简单，包含了类C语法。它只提供了基本功能而没有多余的东西。Go引入了“defer”声明及带有goroutines和通道的并发性管理。

Go语言编译时间短，开发效率和运行效率高。开发过程中相较于 Java 和 C++呆滞的编译速度，Go 的快速编译时间是一个主要的效率优势。Go拥有接近C的运行效率和接近PHP的开发效率。Go语言可以说是开发效率和运行效率二者的完美融合，天生的并发编程支持。

Go语言支持当前所有的编程范式，包括过程式编程、面向对象编程、面向接口编程、函数式编程。程序员们可以各取所需、自由组合、想怎么玩就怎么玩。

Go 是一种非常高效的语言，高度支持并发性。Go是为大数据、微服务、并发而生的一种编程语言。作为一门语言致力于使事情简单化。它并未引入很多新概念，而是聚焦于打造一门简单的语言，它使用起来异常快速并且简单。其创新之处是 goroutines 和通道。goroutines 是 Go 面向线程的轻量级方法，而通道是 goroutines 之间通信的优先方式。

goroutines 的成本很低，只需几千个字节的额外内存，正由于此，才使得同时运行数百个甚至数千个 goroutines 成为可能。可以借助通道实现 goroutines 之间的通信。goroutines 以及基于通道的并发性方法使其非常容易使用所有可用的 CPU 内核，并处理并发的 IO。相较于 Python/Java，在一个 goroutine 上运行一个函数需要最小的代码。

Go拥有强大的编译检查、严格的编码规范和完整的软件生命周期工具，具有很强的稳定性，稳定压倒一切。

1.3 Go安装和配置

1.3.1 下载

在Mac、Windows和Linux三个平台上都支持Go语言。您可以从https://golang.org/dl/下载相应平台的二进制文件。该网站在国内不容易访问，所以可以访问https://www.studygolang.com/dl 进行安装软件的下载。

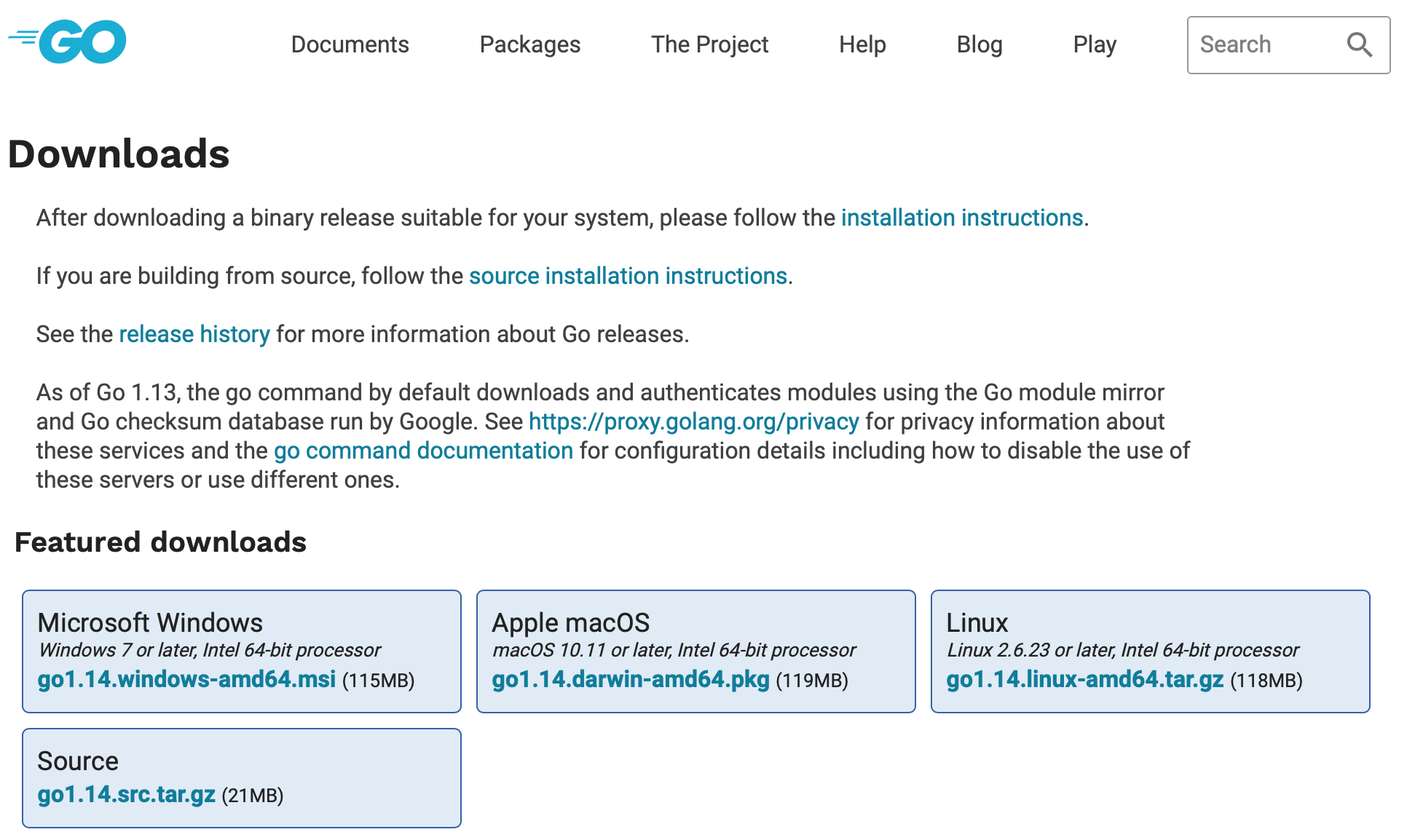


图 1.2



图 1.3

Mac OS 从https://golang.org/dl/下载osx安装程序。双击启动安装。按照提示，这应该在/usr/local/go中安装了Golang，并且还会将文件夹/usr/local/go/bin添加到您的PATH环境变量中。

Windows 从https://golang.org/dl/下载MSI安装程序。双击启动安装并遵循提示。这将在位置c中安装Golang:\Go，并且还将添加目录c:\Go\bin到您的path环境变量。

Linux 从https://golang.org/dl/下载tar文件，并将其解压到/usr/local。将/usr/local/go/bin添加到PATH环境变量中。

1.3.2 windows系统下安装Go

windows系统下安装步骤非常简单，双击.msi文件执行，一路next到底就可以完成安装，安装过程会自动配置好环境变量。

GOPATH之下主要包含三个目录: bin、pkg、src。bin目录主要存放可执行文件; pkg目录存放编译好的库文件, 主要是\*.a文件; src目录下主要存放go的源文件。

查看是否安装配置成功

使用快捷键win+R键，输入cmd，打开命令行提示符，在命令行中输入

go env # 查看得到go的配置信息

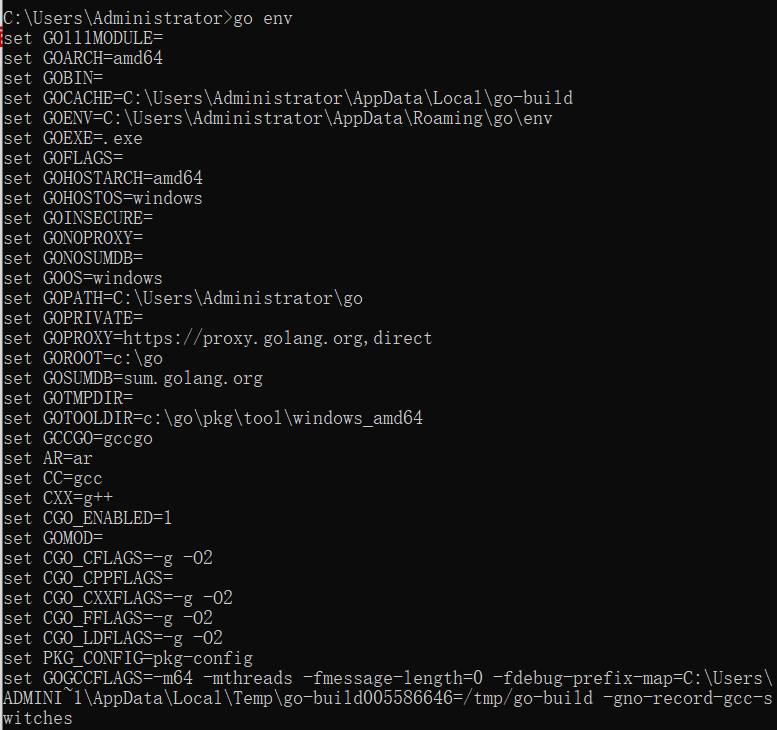


图1.4

go version # 查看go的版本号



图1.5

1.3.3 Mac系统下安装及配置环境变量

Mac系统下安装

双击pkg包，顺着指引，即可安装成功。 在命令行输入 go version，获取到go的版本号，则代表安装成功。

Mac平台下用户可以使用brew直接安装对应的go版本。

go env # 查看得到go的配置信息



图1.6

go version # 查看go的版本号



图1.7

1.4 Goland安装和配置

1.4.1 安装Goland开发工具

Goland是由JetBrains公司旨在为go开发者提供的一个符合人体工程学的新的商业IDE。这个IDE整合了IntelliJ平台的有关go语言的编码辅助功能和工具集成特点。它具有以下特点：

* 编码辅助功能
* 符合人体工程学的设计
* 工具的集成
* IntelliJ插件生态系统

1.4.2 下载及安装

官网下载地址：https://www.jetbrains.com/go/download/。下载完成后，在本地执行解压，安装。

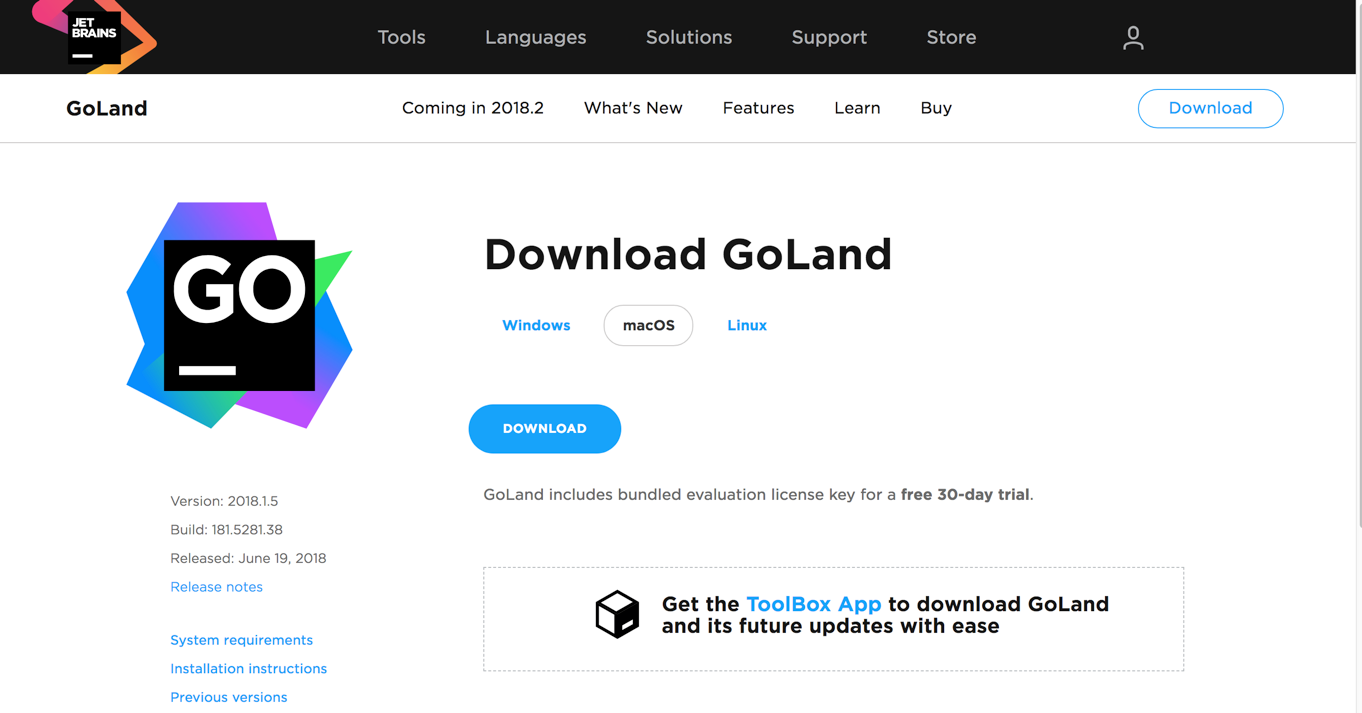


图 1.8

点击“next”按钮，选择要安装的路径，然后点击“next”，会出现安装选项。根据你自己电脑的型号，选择合适的版本后点击“next”按钮。接着保持默认的程序启动目录，点击“install”进行安装。整个安装过程很快，几乎一路next到底。

1.4.3 使用Goland

1.打开Goland工具

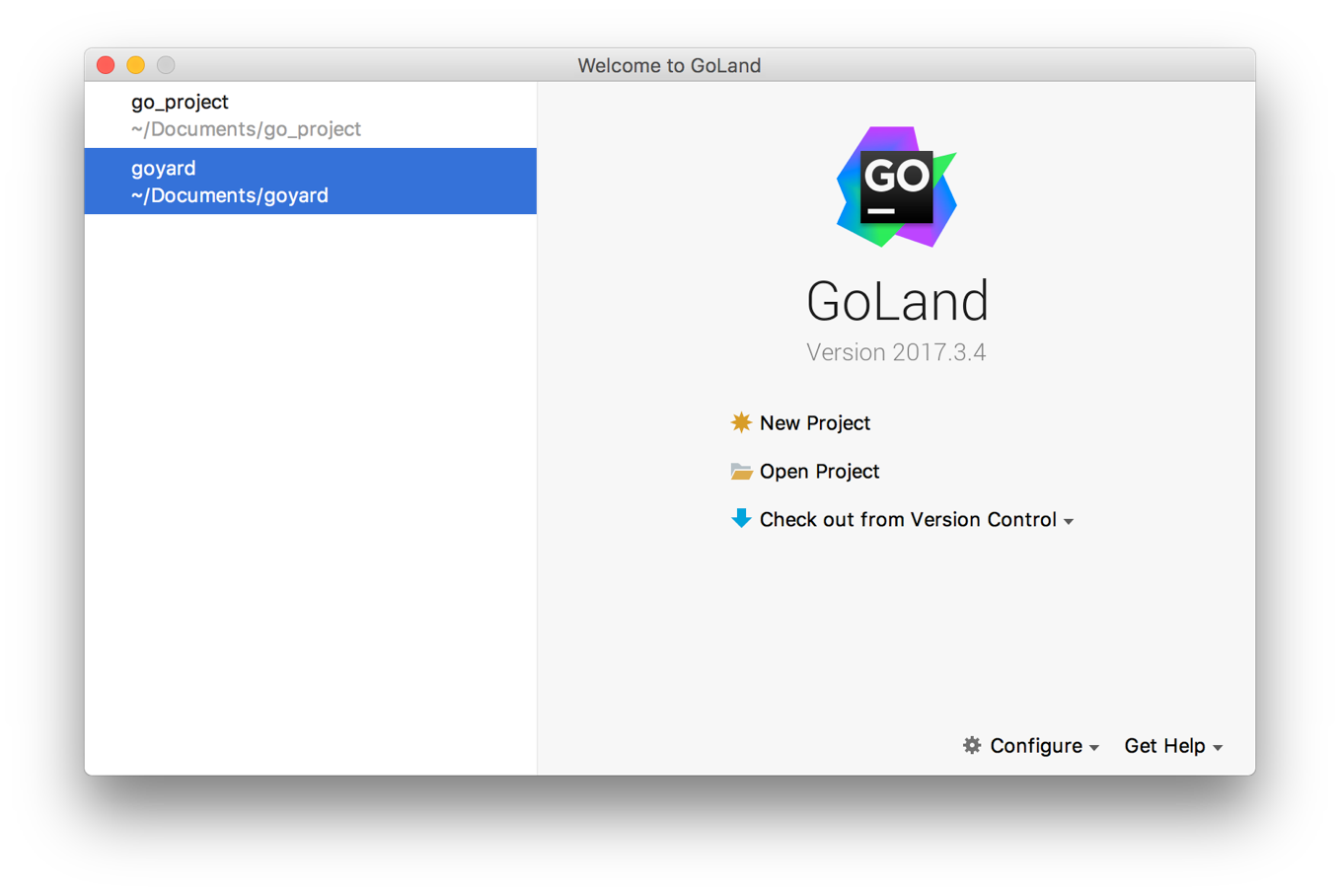


图 1.9

2.创建项目：

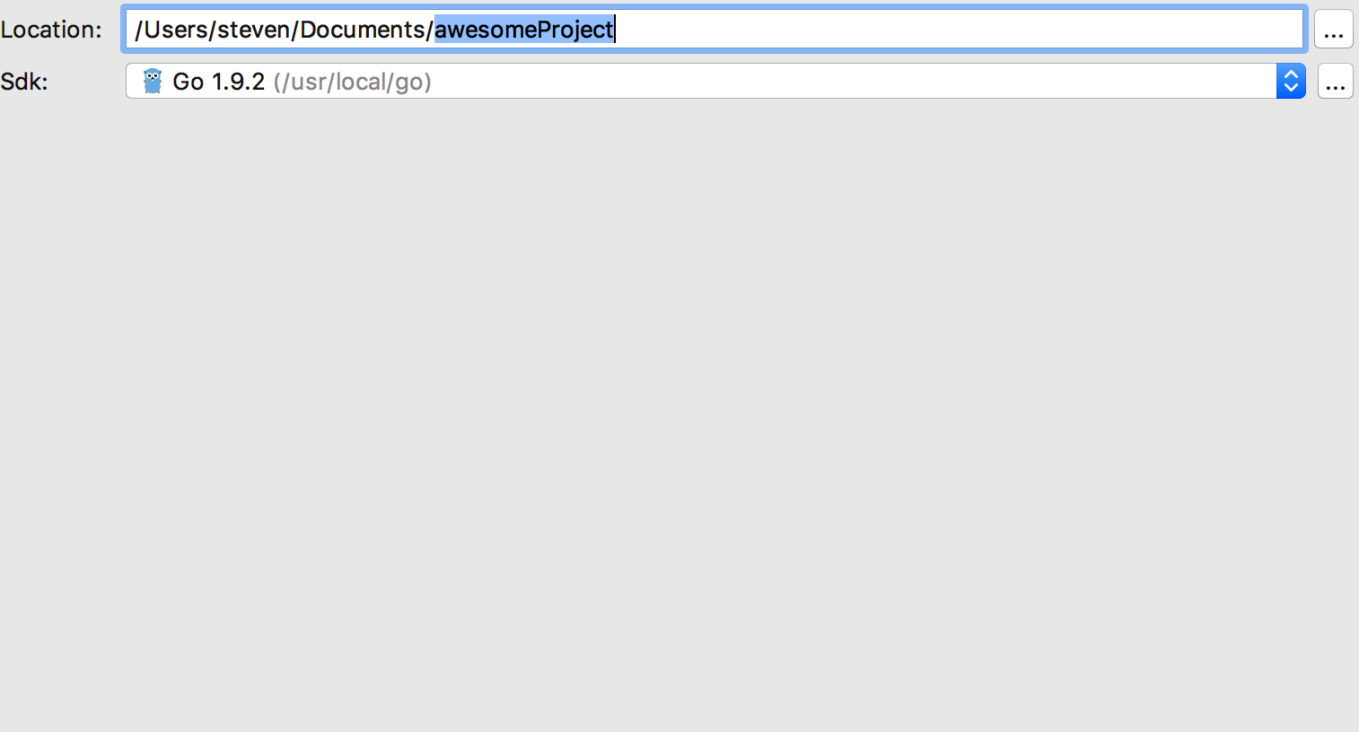


图 1.10

1.4.4 编写第一个程序HelloWorld

1.打开编辑器创建一个新的helloworld.go文件，并输入以下内容：

package main

import "fmt"

func main() {

/\* 输出 \*/

fmt.Println("Hello, World!")

}

2.执行go程序

执行go程序由几种方式

方式一：使用go run命令

step1：使用快捷键win+R，输入cmd打开命令行提示符

step2：进入helloworld.go所在的目录

step3：输入go run helloworld.go命令并观察运行结果。

方式二：使用go build命令

step1：使用快捷键win+R，输入cmd打开命令行提示符

step2：进入helloworld.go所在的目录

step3：输入go build helloworld.go命令进行编译，产生同名的helloworld.exe文件

step4：输入helloworld.exe，执行

1.4.5 Go Module基本使用说明

1.5 Go语言编码规范

1.5.1 HelloWorld程序解析

package main

import "fmt"

func main() {

/\* 这是第一个简单的程序 \*/

fmt.Println("Hello, World!")

}

程序解析

1. package main 定义了包名。必须在源文件中非注释的第一行指明这个文件属于哪个包，如：package main。package main表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 包。

2. import "fmt" 告诉 Go 编译器这个程序需要使用 fmt 包，fmt 包实现了格式化 IO（输入/输出）的函数。

3. func main() 是程序入口。main 函数是每一个可执行程序所必须包含的，一般来说都是在启动后第一个执行的函数，如果有 init() 函数则会先执行init()函数。

4. /\*...\*/ 是注释，在程序执行时将被忽略。单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 /\* 开头，并以 \*/ 结尾，且不可以嵌套使用，多行注释一般用于文档描述或代码片段。

5. fmt.Println(...) 可以将字符串输出到控制台，并在最后自动增加换行字符 \n。 使用 fmt.Print("hello, world\n") 可以得到相同的结果。

1.5.2 Go语言编码规范

1.注释

单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释

多行注释也叫块注释，均已以 /\* 开头，并以 \*/ 结尾，且不可以嵌套使用，多行注释一般用于文档描述或注释成块的代码片段

2.标识符

标识符用来命名变量、类型等程序实体。一个标识符实际上就是一个或是多个字母(A~Z和a~z)数字(0~9)、下划线\_组成的序列，但是第一个字符必须是字母或下划线而不能是数字。

Go不允许在标识符中使用@、$和%等标点符号。

Go是一种区分大小写的编程语言。因此，Manpower和manpower是两个不同的标识符。

以下是无效的标识符：

* 1xy（以数字开头）
* case（Go 语言的关键字）
* chan（Go 语言的关键字）
* x+y（运算符是不允许的）

3.Go 语言的空格

Go 语言中变量的声明必须使用空格隔开，如：var age int;语句中适当使用空格能让程序更易阅读。在变量与运算符间加入空格，程序看起来更加美观，如：a = x + y;

4.语句的结尾

在 Go 程序中，一行代表一个语句结束。Go语言中是不需要类似于Java需要分号结尾，因为这些工作都将由 Go 编译器自动完成；

如果打算将多个语句写在同一行，它们则必须使用分号“;”人为区分，但在实际开发中并不鼓励这种做法。

5.可见性规则

Go语言中，使用大小写来决定标识符（常量、变量、类型、接口、结构或函数）是否可以被外部包所调用。

以一个大写字母开头，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（使用时程序需要先导入这个包），如同面向对象语言中的 public。

如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的。

1.5.3 Go语言关键字及保留字

下面列举了 Go 代码中会使用到的 25 个关键字或保留字：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

表1.1 Go语言关键字或保留字

除了以上介绍的这些关键字，Go 语言还有 36 个预定义标识符：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | uintptr |

表1.2 Go语言预定义标识符

1.5.4 Go 程序结构组成

Go 文件的基本组成包括：包声明、引入包、函数、变量、语句 & 表达式、注释。

// 当前程序的包名

package main

// 导入其他包

import . "fmt"

// 常量定义

const PI = 3.14

// 全局变量的声明和赋值

var name = "gopher"

// 一般类型声明

type newType int

// 结构的声明

type gopher struct{}

// 接口的声明

type golang interface{}

// 由main函数作为程序入口点启动

func main() {

Println("Hello World!")

}

1.6 小结

本章带领大家初步认识Go语言。先给大家讲解Go语言的发展历史，设计Go语言的初衷以及Go语言的主要特征。接下来带领大家安装Go语言的运行环境，以及Go语言集成开发工具（IDE）——Goland。之后便是Go语言版本的HelloWorld程序，以及通过对HelloWorld的分析，掌握Go语言的构成及编码规范。为之后开始深入学习Go语言做好准备。

第二章Go语言基础

本章重点为大家介绍如下的内容：

* 变量声明、初始化及赋值
* 常量声明与初始化
* 数据类型
* 运算符
* 表达式

2.1 变量

2.1.1 变量的概念

变量是计算机语言中储存数据的抽象概念。变量的功能是存储数据。变量通过变量名访问；

变量的本质是计算机分配的一小块内存，专门用于存放指定数据，在程序运行过程中该数值可以发生改变；

变量的存储往往具有瞬时性，或者说是临时存储，当程序运行结束，存放该数据的内存就会释放，而该变量就会消失；

Go 语言的变量名由字母、数字、下划线组成，首个字符不能为数字;

Go语法规定，定义的局部变量若没有被调用则编译错误。

2.1.2 声明变量

标准变量声明格式

var <变量名> [变量类型]

例：

var str string

var关键字的另一种用法是将多个需要声明的变量放在一起，避免程序员重复写var关键字。

var (

<变量名> <变量类型>

<变量名> <变量类型>

<变量名> <变量类型>

)

例：

var (

a int

b string

c []float32

d func() bool

e struct {

x int

y string

}

)

与C语言不同，Go语言变量声明语句不需要使用分号作为结束符。

Go语言变量声明后默认值如下：

整形和浮点型变量默认值：0；

字符串默认值为空字符串；

布尔型默认值为false；

函数、指针变量初始值为nil。

2.1.3 变量初始化

初始化变量的标准格式：

var <变量名> [类型] = <表达式>

例：

var str string = "this is string"

初始化变量的编译器自动推断类型格式：

var <变量名> = <表达式>

例：

var str = "this is string"

初始化变量的简短声明格式（短变量声明格式）：

<变量名> := <表达式>

例：

s1 := "this is s1"

使用 := 赋值操作符，可以高效地创建一个新的变量，称之为初始化声明。使用赋值操作符声明语句省略了var 关键字，变量类型将由编译器根据表达式自动推断。赋值操作符是声明变量的首选形式，要注意它只能被用在函数体内，不可以应用于全局变量的声明与赋值。

使用赋值操作符赋值的变量必须是没有定义过的变量，如果定义过，将会发生编译错误。在多个短变量声明和赋值中，至少有一个新声明的变量出现在左侧中，那么即便有其它变量名可能是重复声明的，编译器也不会报错。

Go语法中，变量初始化和变量赋值是两个不同的概念。Go语言的变量赋值与其他语言一样，但是Go提供了其他程序员期待已久的多重赋值功能，可以实现变量交换。多重赋值让Go语言比其他语言更灵活、更简单。

曾经在不支持多重赋值的编程语言中，我们需要交换两个变量的值，使用如下方法：

t = i; i = j; j = t;

而在Go语言中，多重赋值帮我们解决了这个头痛的难题，我们只需要使用下面的代码就只可以实现变量互换。

i, j = j, i

多重赋值交换变量的原理是将两个被赋值的变量的值，都存储在临时变量里，然后再用临时变量去赋值。

2.1.4 匿名变量

Go语言的函数允许返回多个值，当我们只需要使用部分返回值的时候，可以使用匿名变量，用“\_”下划线替换即可。

匿名变量不占用命名空间，不会分配内存。

例

i , \_ := noname()

2.2 数据类型

Go 语言的数据类型分为基本数据类型与复合数据类型。

基本数据类型（原生数据类型）：整型、浮点型、布尔型、字符串、字符（byte、rune）。

复合数据类型（派生数据类型）：指针（pointer）、数组（array）、切片（slice）、映射（map）、函数（function）、结构体（struct）、通道（channel）。

2.2.1 整型

整型就最基础的数据类型，其按长度可分为：int8、int16、int32、int64、int；无符号的整型数分为：uint8、uint16、uint32、uint64、uint

其中uint8就是byte型，int16对应C语言的short型，int64对应C语言的long型。

|  |  |
| --- | --- |
| 序号 | 类型和描述 |
| 1 | uint8 无符号 8 位整型 (0 到 255) |
| 2 | uint16 无符号 16 位整型 (0 到 65535) |
| 3 | uint32 无符号 32 位整型 (0 到 4294967295) |
| 4 | uint64 无符号 64 位整型 (0 到 18446744073709551615) |
| 5 | int8 有符号 8 位整型 (-128 到 127) |
| 6 | int16 有符号 16 位整型 (-32768 到 32767) |
| 7 | int32 有符号 32 位整型 (-2147483648 到 2147483647) |
| 8 | int64 有符号 64 位整型 (-9223372036854775808 到 9223372036854775807) |

表2.1 Go语言中整型

还有其他数字类型

|  |  |
| --- | --- |
| 序号 | 类型和描述 |
| 1 | byte 类似 uint8 |
| 2 | rune 类似 int32 |
| 3 | uint 32或64位 |
| 4 | int 与 uint 一样大小 |
| 5 | uintptr 无符号整型，用于存放一个指针 |

表2.2 Go语言中其他数字整型

int8与int是完全不同的两种数据类型，编译器不能做自动的类型转换。

例：

func main() {

var v int8

v1 := 10 //系统自动推断数据类型为int

v = v1 //此处报错

fmt.Println(v)

}

报错内容如下：

cannot use v1 (type int) as type int8 in assignment

错误提示告诉我们“不能使用int类型作为int8类型的参数”，所以如果要使之成立就需要使用强制类型转换解决此问题。修改后的代码如下：

例：

func main() {

var v int8

v1 := 10

v = int8(v1)

fmt.Println(v)

}

同样int和uint也是完全不同的两种数据类型，编译器同样不能做自动的类型转换。

在表2.1看到的uint8这个数据类型也就是byte类型，如下例子的方式是可行的：

例：

func main() {

var v byte

var v1 uint8

v1 = 10

v = v1

fmt.Println(v)

}

既然二者为同一类型，为什么要给它们提供不同的名字呢？

一般情况下我们使用byte类型来说明此变量是一个原始的数据而不是一个小的整型数。

uintptr是一种无符号整数类型，没有指定大小，但是足够容纳指针。此类型在底层编程中使用较多，与C语言函数库或者操作系统接口交互的地方会使用，其它情况使用较少。

2.2.2 浮点型

Go语言支持4种浮点型数：float32、float64、complex64（32 位实数和虚数）、complex128（64 位实数和虚数）。float32的最大范围是3.4e38，用常量定义是：math.MaxFloat32；float64的最大范围是1.8e308，用常量定义是：math.MaxFloat64。

例：

func main() {

fmt.Println(math.MaxFloat32) //输出：3.4028234663852886e+38

fmt.Println(math.MaxFloat64) //输出：1.7976931348623157e+308

}

通常情况下我们建议使用float64，毕竟flaot64的精度相对较大，累计计算的误差相对较小。float32的有效bit位只有23个，其它的用于指数和符号；当整数大于23bit能表达的范围时，float32的表示将出现误差，例如：

例：

func main() {

var f float32 = 1 << 24 // 1 << 24

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

fmt.Println(f) // 1.6777216e+0.7

fmt.Println(f + 1) // 1.6777216e+0.7

}

上面的例子是显示指数形式的浮点数，我们平常在使用的时候使用的是类似3.1415926这样的浮点数，那我们需要如何输出呢？

例：

func main() {

var f float32 = 123453.1415926

fmt.Printf("%8.2f", f) // 123453.14

fmt.Printf("%3.1f", f) // 123453.1

fmt.Printf("%3.4f", f) // 123453.1406

}

注意，如果定义的时候不加小数点，会被推导为整型。

2.2.3 布尔型

声明方式：var flag bool

布尔型无法参与数值运算，也无法与其他类型进行转换。

例：

func main() {

var v1 bool

fmt.Println(v1) //false

v1 = true

fmt.Println(v1) //true

v1 = (1 == 2)

fmt.Println(v1) //false

v2 := (2 == 2)

fmt.Println(v2) //true

var a, b int

a = 2

b = 2

v3 := (a == b)

fmt.Println(v3) //true

}

由例子可见，定义了v1变量后，系统默认赋值为false，同时我们在v3变量可以看到，我们将一个逻辑判断表达式赋值给v3，v3会被自动推导为bool类型。

2.2.4 字符串

字符串在Go语言中是以基本数据类型出现的，使用字符串就像使用其他原生基本数据类型int、float32、float64、bool一样。

例：

func main() {

var s string

s1 := "this is String"

ch := s1[0]

fmt.Println(s) //这里什么也不会显示

fmt.Println(s1) //这里输出this is String

fmt.Println(reflect.TypeOf(ch)) //这里输出uint8，也就是byte类型

fmt.Println(ch) //这里输出ascii码116

fmt.Printf("%c", ch) //这里输出字母t

}

从上面的例子我们可以看到，只定义一个字符串，在不赋值的情况下，输出的是空内容；我们可以使用s1[N]的方式，来获取第N个字符（程序员的世界数字从0开始哦^\_^）；s1[N]方式取出来的值类型是uint8类型，也就是我们之前说过的byte类型，其内容为字符串的ascii码，我们使用printf的“%c”模式，将其转换成字符串，就可以得到我们获取到的字符串了。

如果我们可以使用s1[N]的方式得到字符串的第N个字符，那么我们是不是可以通过s1[N] = “a”的方式来赋值呢？我们看下面的例子：

例：

func main() {

var s = "this is String"

ch := s[1]

fmt.Printf("%c", ch)

s[1] = "k" //cannot assign to s[1]

}

这个例子会报一个编译错误，告诉我们不能给s[1]赋值。

cannot assign to s[1]

字符串中可以使用的转义字符如下：

* \r 回车
* \n 换行符
* \t 制表符TAB
* \' 单引号
* \" 双引号
* \\ 反斜杠

多行字符串需要使用“ ` ”反引号，多用于内嵌源码和内嵌数据；在反引号中的所有代码不会被编译器识别，而只是作为字符串的一部分。双引号书写字符串被称为字符串字面量（string literal），这种字面量不能跨行。

2.2.5 字符

字符串中的每一个元素叫做“字符”，定义字符时使用单引号。Go语言的字符有两种：

1.byte型：其实是uint8的别名。代表了一个ASCII码的一个字符。

2.rune型：其实就是int32。代表一个UTF-8字符。当需要处理中文等unicode字符集时需要用到rune类型。

var a byte = 'a'

var b rune = '一'

例：

func main() {

var s1 rune

var s2 rune

var s3 rune

s1 = 'a'

s2 = "a"//此处报错cannot use "a" (type string) as type rune is assignment

s3 = 'abc'//此处报错 invalid character literal(more than one character)

fmt.Println(s1) //97

fmt.Printf("%c", s1) //a

}

上例中，我们给s2赋值为”a”的时候报错为不能使用字符串为字符类型赋值，而我们给s3赋值为‘abc’的时候，编译错误为无效的字符，超过了一个字符。所以我们可以得出结论rune类型在使用及赋值的时候，必须使用单引号包起来，并且只能是一个字符，不允许赋值多个。

我们给s1赋值为’a’,实际我们刚才说过rune是int32的等价类型，所以直接输出我们得到的是字符的ascii码97。

2.2.6 数组

数组是一个由固定长度的特定类型元素组成的序列，一个数组可以由N（N >= 0）个元素组成。数组的长度必须是固定的不可变的，所以很少会直接使用数组；与数组对应的两种引用类型切片类型与字典类型是可以增长和收缩的动态序列，它们的使用也更灵活，所以这两种数据类型也是未来我们最常用的数据类型。但是要理解这两种数据类型，我们需要先从理解数组类型开始。

数组的每个元素都可以通过索引下标来访问，还记得我们之前说过，程序员的世界从0开始。所以索引下标也是从0开始的。

大家可以回忆一下在字符串中，我们是如何使用下标获取字符串中单个字符的。数组的基础操作，也和字符串中获取字符的方式类似。我们通过例子看一下如何构建数组，如何对数组进行基本操作。

例：

func main() {

var a [8]int //定义长度为8整型数组

//定义长度为5的数组并初始化

var b [5]int = [5]int{0, 1, 2, 3, 4}

fmt.Println(a[0]) //输出：0

fmt.Println(b[len(b)-1]) //输出:4

fmt.Println(b) //输出：[0 1 2 3 4]

b[0] = 100 //修改b数组第0个元素的值

fmt.Println(b[0]) //输出：100

}

数组遍历的方法有两种：

例：

func main() {

var b [5]int = [5]int{0, 1, 2, 3, 4} //定义长度为5的数组并初始化

//以下为第一种遍历方式

for i := 0; i < len(b); i++ {

fmt.Println(b[i])

//换行输出：

//0

//1

//2

//3

//4

}

//以下为第二种遍历方式

for key, value := range b {

fmt.Printf("下标是：%d 值是：%d\n", key, value)

//换行输出：

//下标是：0 值是：0

//下标是：1 值是：1

//下标是：2 值是：2

//下标是：3 值是：3

//下标是：4 值是：4

}

}

数组的长度必须是固定的，所以我们在给数组赋值的时候要注意a[8]int与b[5]int是完全不同的两种类型。

例：

func main() {

var a [8]int //定义长度为8整型数组

var b [5]int = [5]int{0, 1, 2, 3, 4} //定义长度为5的数组并初始化

a = b //编译错误：cannot use b(type [5]int) as type [8]int in assignment

}

当我们尝试将长度为5的数组赋值给长度为8的数组时，编译报错，告诉我们不允许这样的操作。

2.2.7 结构体

结构体是一种聚合的数据类型，由N（N >=0）个任意类型的值聚合成的实体，每个值被称为结构体的成员或属性。

结构体的基本定义与使用方法如下：

例：

type Object struct {

Hash string

length int

}

func main() {

var obj Object

obj.length = 10

obj.Hash = "45454afasf4asdf1a"

fmt.Println(obj.length) //输出：10

}

我们使用type关键字定义了一个Object的结构体（struct），然后将成员用“{”、“}”包含。任何数据类型都可以成为结构体的成员的类型，接下来的内容中，我们都将使用属性来替代成员这个词，大家只要看到属性，就一定要想到，这是和结构体有关。

当属性名的首字母大写时，我们就可以在任何位置上访问这个属性；如果属性首字母是小写，那这个属性就是私有的，只能在结构体声明的包下进行访问或赋值。上例中，结构体Object是在main名声明的，所以可以在main函数中直接访问操作。如果在别的包中，就不能访问length属性，只能访问Hash属性。

2.2.8 指针

Go语言中的指针相对于C语言中的指针简单了很多，对于开发者来说我们可以不必担心因为指针的错误使用而造成的各种各样奇奇怪怪的问题。

指针是一种指向变量内存地址的数据类型。一个指针变量可以指向任何一个变量的内存地址。

在Go语言中取地址的运算符为“&”，我将通过一个短例子查看地址与指针基本的样子。

例：

func main() {

//定义一个整型变量

var a int = 10

//定义一个指针变量

var p \*int

//指针p指向变量a的地址

p = &a

fmt.Println(a) //输出：10

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：10

\*p = 100

fmt.Println(a) //输出：100

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：100

fmt.Println(\*p == a) //输出：true

}

上面例子中我们使用指针p指向了变量a的地址（&a），通过输出我们发现指针p的值就是变量a的地址，而\*p的值是变量a的值；当我们改变了\*p的值的时候，变量a的值也随之改变，因此我们可以认为在这种情况下，\*p等价于变量a。

我们刚才说到，每个变量都有一个地址，指针本身也是一个变量，那么指针是不是也有地址呢？指针是不是也可以被另一个指针指向呢？我们看下面的例子。

例：

func main() {

//定义一个整型变量

var a int = 10

//定义一个指针变量

var p \*int

//指针p指向变量a的地址

p = &a

fmt.Println(a) //输出：10

fmt.Println(&a) //输出：0xc420014100

fmt.Println(p) //输出：0xc420014100

fmt.Println(\*p) //输出：10

fmt.Println(&p) //输出：0xc42000c028

//定义一个指向指针的指针

var ptr \*\*int

//将指针ptr指向指针p的地址

ptr = &p

fmt.Println(ptr) //输出：指针ptr的地址为0xc42000c028

fmt.Println(\*ptr) //输出：0xc420014100

fmt.Println(\*\*ptr) //输出：10

}

上例中我们定义了一个指向指针的指针ptr，并使其指向指针p的地址。

我们在输出的时候发现，ptr的值为指针p的地址；\*ptr的值为指针p的值，也就是变量a的地址；\*\*ptr的值为指针\*p的值，也就是变量a的值10。

2.2.9 切片

切片也叫数组切片，通过上面对数组的介绍，我们知道数组的长度一旦确定就不可以更改了。这种情况下是无法满足我们的真实需求的。

Go语言为我们提供了数组切片（slice）来补充数组的不足。

切片是一个可以随时动态扩充的存储空间，用于存放相同类型的数据；并且切片可以随意传递而不会导致所管理的元素被重复复制。

本书第5章将重点讲解切片的使用。

2.2.10 映射（map）

map是一种键/值（key/value）对形式的数据结构，它是一种未排序的集合，所有的key都是唯一的，通过key可以精准定位数据的位置，并对其进行相关操作。

在Go语言中，一个map就是一个哈希表的引用，所有的key的类型必须相同，value的类型也必须相同。要注意所有的的key都必须使用支持”==”比较运算符的类型。

需要注意的是浮点型最好不要使用在key中，一些误差会导致判断失败，影响程序的正常运行。value的类型则没有任何限制。

例：

func main() {

//创建一个map,key为string类型，value为string类型

m := make(map[string]string)

//给map增加值

m["username"] = "admin"

m["sex"] = "man"

m["age"] = "20"

fmt.Println(m) //输出：map[username:admin sex:man age:20]

//删除键值

delete(m, "age")

fmt.Println(m) //输出：map[username:admin sex:man]

//查询键值是否存在

value, ok := m["username"]

if ok {

fmt.Println(value) //输出：value的结果，此处为admin

} else {

fmt.Println("nil") //输出：如果ok返回false，则输出nil

}

}

通过上面的例子我们可以了解到如何创建map，如何给map增加数据、修改数据、删除数据以及查询数据是否存在 。这一个示例，已经基本上包含了map的大部分操作。

2.2.11 函数

在Go语言中，函数也作为一种数据类型。既然是数据类型那么函数也就可以当作一个值来传递和使用。

例：

func main() {

//定义一个匿名函数，赋值给fun变量

fun := func(a, b int) int {

if a > b {

return a

}

return b

}

//调用匿名函数并保存返回值

fmt.Println(fun(5, 10)) //输出：10

fmt.Println(reflect.TypeOf(fun)) //输出:func()int,int)int

}

在上面例子中，我们定义了一个匿名函数，将其赋值给变量fun，我们在程序结尾输出类型的时候，看到类型是func类型，也就是函数类型；函数类型的调用方式也与常规函数的方法一样，关于函数详细的使用方式，我们将在第4章函数中进行详细的解读。

2.2.12 通道

通道(channel)是Go语言中为我们提供的用于在进程间通信的一种方式，我们使用channel在多个不同的进程间传递消息。

channel本身有类型的限制，一个channel只能传递一种类型的数据，在声明的时候就必须指定好要传递的类型。本书第12章并发编程中将会对channel的使用进行详细的解释说明，需要提前了解的读者请移步第12章

2.3 打印格式化

Print、Println 、Printf 、Sprintf 、Fprintf都是fmt 包中的公共方法，在需要打印信息时需要用到这些函数，本节主要以Printf方法格式化输出为主。

|  |  |
| --- | --- |
| Print | 输出到控制台(不接受任何格式化，它等价于对每一个操作数都应用 %v) |
| Println | 输出到控制台并换行 |
| Printf | 只可以打印出格式化的字符串。只可以直接输出字符串类型的变量 |
| Sprintf | 格式化并返回一个字符串而不带任何输出。 |
| Fprintf | 格式化并输出到 io.Writers 而不是 os.Stdout。 |

表2.3 格式化打印介绍

1.通用

%v 值的默认格式表示；

%+v 类似%v，但输出结构体时会添加字段名；

%#v 值的Go语法表示；

%T 值的类型的Go语法表示。

2.布尔值

%t 单词true或false。

3.整数

%b 表示为二进制；

%c 该值对应的unicode码值；

%d 表示为十进制；

%8d 表示该整型长度是8，不足8则在数值前补空格。如果超出8，则以实际为准；

%08d 数字长度是8，不足8位的，在数字前补0。如果超出8，则以实际为准；

%o 表示为八进制；

%q 该值对应的单引号括起来的go语法字符字面值，必要时会采用安全的转义表示；

%x 表示为十六进制，使用a-f；

%X 表示为十六进制，使用A-F；

%U 表示为Unicode格式：U+1234，等价于"U+%04X"。

4.浮点数与复数的两个组分

%b 无小数部分、二进制指数的科学计数法，如-123456p-78；参见strconv.FormatFloat；

%e （=%.6e）有6位小数部分的科学计数法，如-1234.456e+78；

%E 科学计数法，如-1234.456E+78；

%f （=%.6f）有6位小数部分，如123.456123；

%F 等价于%f；

%g 根据实际情况采用%e或%f格式（以获得更简洁、准确的输出）；

%G 根据实际情况采用%E或%F格式（以获得更简洁、准确的输出）。

5.字符串和[]byte

%s 直接输出字符串或者[]byte；

%q 该值对应的双引号括起来的go语法字符串字面值，必要时会采用安全的转义表示；

%x 每个字节用两字符十六进制数表示（使用a-f）；

%X 每个字节用两字符十六进制数表示（使用A-F）。

6.指针

%p 表示为十六进制，并加上前导的0x。

没有%u。整数如果是无符号类型自然输出也是无符号的。类似的，也没有必要指定操作数的尺寸（int8，int64）。

宽度通过一个紧跟在百分号后面的十进制数指定，如果未指定宽度，则表示值时除必需之外不作填充。精度通过（可选的）宽度后跟点号后跟的十进制数指定。如果未指定精度，会使用默认精度；如果点号后没有跟数字，表示精度为0。举例如下：

%f: 默认宽度，默认精度；

%9f 宽度9，默认精度；

%.2f 默认宽度，精度2；

%9.2f 宽度9，精度2；

%9.f 宽度9，精度0。

7.其它flag

'+' 总是输出数值的正负号；对%q（%+q）会生成全部是ASCII字符的输出（通过转义）；

' ' 对数值，正数前加空格而负数前加负号；

'-' 在输出右边填充空白而不是默认的左边（即从默认的右对齐切换为左对齐）；

'#' 切换格式；

八进制数前加0（%#o），十六进制数前加0x（%#x）或0X（%#X），指针去掉前面的0x（%#p）；

对%q（%#q），如果strconv.CanBackquote返回真会输出反引号括起来的未转义字符串；

对%U（%#U），输出Unicode格式后，如字符可打印，还会输出空格和单引号括起来的go字面值；

对字符串采用%x或%X时（% x或% X）会给各打印的字节之间加空格；

'0' 使用0而不是空格填充，对于数值类型会把填充的0放在正负号后面。

2.4 数据类型转换

2.4.1 T(表达式)

采用数据类型前置加括号的方式进行类型转换。T表示要转换的类型；表达式包括变量、数值、函数返回值等。类型转换时，需要考虑两种类型之间的关系和范围，是否会发生数值截断。需要注意的是布尔型无法与其他类型进行转换。

例：

func main(){

var sum int = 17

var count int = 5

var mean float32

mean = float32(sum)/float32(count)

fmt.Printf("mean 的值为: %f\n",mean)//输出：mean 的值为: 3.400000

}

2.4.2 float与int之间转换

需要注意float转int时精度的损失，小数点后的数据会被舍弃。

例：

func main(){

var mean float32 = 3.6845822

fmt.Println(int(mean))//输出：3

}

2.4.3 int转string

int转string其实相当于是byte或rune转string。

该int数值是ASCII码的编号或Unicode字符集的编号。转成string就是将根据字符集，将对应编号的字符查找出来。

当该数值超出Unicode编号范围，则转成的字符串显示为乱码。

例如19968转string，就是“一”。

ASCII字符集中数字的10进制范围是[30 - 39]

ASCII字符集中大写字母的10进制范围是[65 - 90]

ASCII字符集中小写字母的10进制范围是[97 - 122]

Unicode字符集中汉字的范围是[4e00-9fa5]，10进制范围是[19968 - 40869]

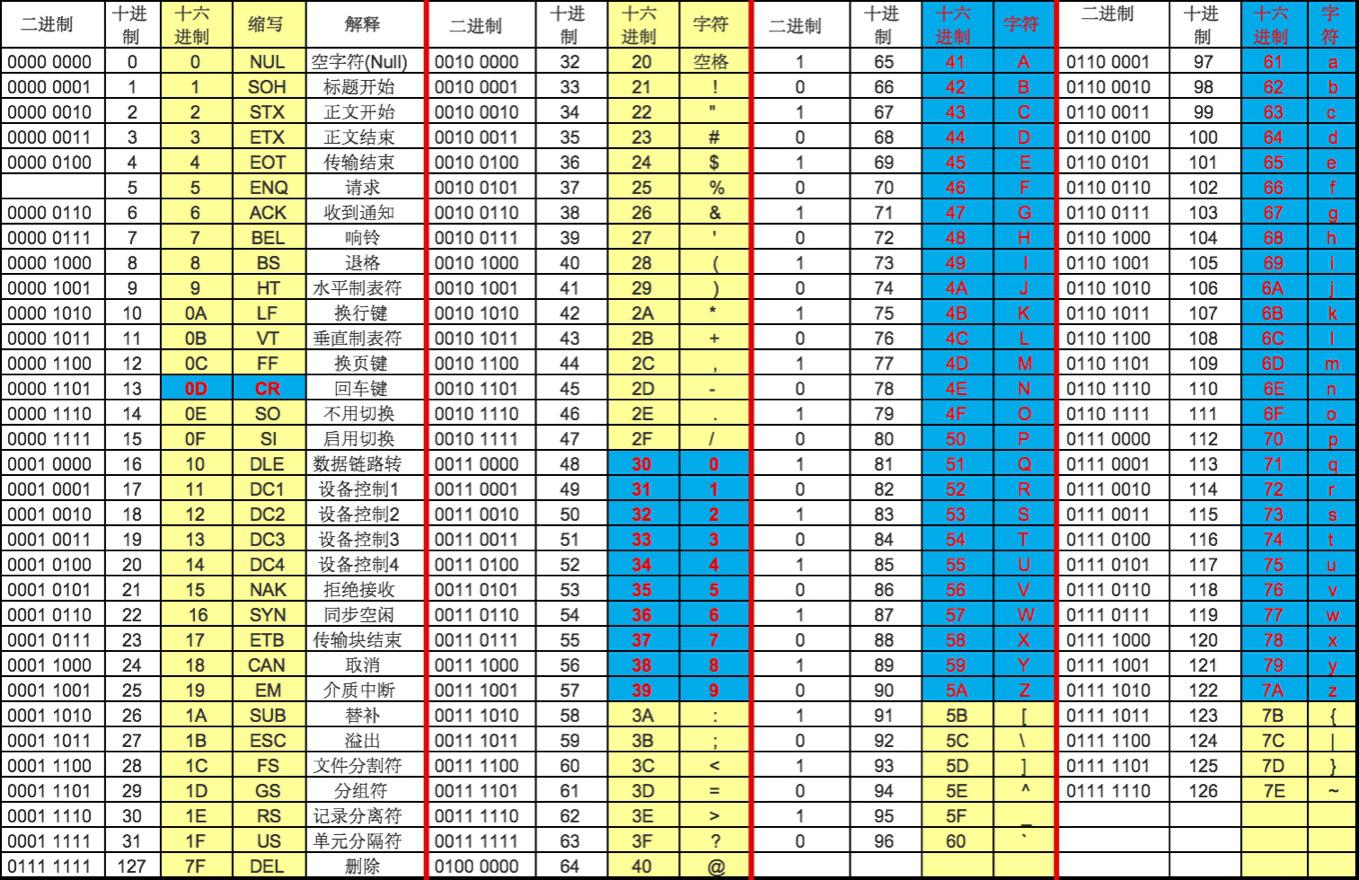


图 2.1

2.4.4 string转int

go语言中不允许string类型转换为int类型。

例：

func main(){

var mean string = "this is string"

fmt.Println(int(mean))

}

上面代码在编译过程中报错

cannot convert mean (type string) to type int

2.5常量

2.5.1 声明方式

相对于变量，常量是恒定不变的值，例如圆周率。常量是一个简单值的标识符，在程序运行时，不会被修改。常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

常量的定义格式：

const <标识符> [类型] = <值>

可以省略类型说明符 [type]，因为编译器可以根据变量的值来自动推断其类型。

显式类型定义：

const B string = "Steven"

隐式类型定义：

const C = "Steven"

多个相同类型的声明可以简写为：

const WIDTH , HEIGHT = value1, value2

常量定义后未被使用，不会在编译时出错。

2.5.2 常量用于枚举（常量组）

例如以下格式：

const (

Unknown = 0

Female = 1

Male = 2

)

数字 0、1 和 2 分别代表未知性别、女性和男性。

常量组中如果不指定类型和初始值，则与上一行非空常量的值相同。

const (

a = 10

b

c

)

打印a、b、c，输出：10 10 10。

2.5.3 iota

iota是特殊常量值，是一个系统定义的可以被编译器修改的常量值。iota只能用在常量赋值中。

在每一个const关键字出现时，被重置为0，然后每出现一个常量，iota所代表的数值会自动增加1。iota可以理解成常量组中常量的计数器，不论该常量的值是什么，只要有一个常量，那么iota就加1。

iota 可以被用作枚举值：

const (

a = iota

b = iota

c = iota

)

println(a, b, c)

打印输出：0 1 2

第一个 iota 等于 0，每当 iota 在新的一行被使用时，它的值都会自动加 1；所以 a=0, b=1, c=2

常量组中如果不指定类型和初始值，则与上一行非空常量的值相同。所以上述的枚举可以简写为如下形式：

const (

a = iota

b

c

)

println(a, b, c)

打印输出：0 1 2

例：

const (

i = 1<<iota

j = 3<<iota

k

l

)

func main() {

fmt.Println("i=",i)

fmt.Println("j=",j)

fmt.Println("k=",k)

fmt.Println("l=",l)

}

打印输出结果：

i= 1

j= 6

k= 12

l= 24

例：

const (

a1 = '一'

b1

c1 = iota

d1

)

func main() {

fmt.Println(a1, b1, c1, d1)

}

打印输出结果：

19968 19968 2 3

2.6类型别名（Type Alias）

2.6.1 概要

类型别名是Go1.9版本添加的新功能。主要用于代码升级、迁移中类型的兼容性问题。

在Go1.9版本前内建类型定义的代码是：

type byte uint8

type rune int32

而在Go1.9版本之后变更为：

type byte = uint8

type rune = int32

2.6.2 类型别名与类型定义

类型别名的语法格式：

type 类型别名 = 类型

定义类型的语法格式：

type 新的类型名 类型

例：

type NewString string

该语句是将NewString定义为string类型。通过type关键字，NewString会形成一种新的类型。NewString本身依然具备string的特性。

type StringAlias = string

该语句是将StringAlias定义为string的一个别名。使用StringAlias与string等效。别名类型只会在代码中存在，编译完成时，不会有别名类型。

2.7 Go 语言运算符

运算符用于在程序运行时执行数学或逻辑运算。

Go 语言内置的运算符有：

* 算术运算符
* 关系运算符
* 逻辑运算符
* 位运算符
* 赋值运算符
* 其他运算符

2.7.1 算术运算符 （Arithmetic operator）

下表列出了所有Go语言的算术运算符。假定 A 值为 10，B 值为 20。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| + | 相加 | A + B 输出结果 30 |
| - | 相减 | A - B 输出结果 -10 |
| \* | 相乘 | A \* B 输出结果 200 |
| / | 相除 | B / A 输出结果 2 |
| % | 求余 | B % A 输出结果 0 |
| ++ | 自增 | A++ 输出结果 11 |
| -- | 自减 | A-- 输出结果 9 |

表2.4 Go语言中算术运算符

以下实例演示了各个算术运算符的用法：

package main

import "fmt"

func main() {

var a int = 21

var b int = 10

var c int

c = a + b

fmt.Printf("第一行 - c 的值为 %d\n", c )

c = a - b

fmt.Printf("第二行 - c 的值为 %d\n", c )

c = a \* b

fmt.Printf("第三行 - c 的值为 %d\n", c )

c = a / b

fmt.Printf("第四行 - c 的值为 %d\n", c )

c = a % b

fmt.Printf("第五行 - c 的值为 %d\n", c )

a++

fmt.Printf("第六行 - a 的值为 %d\n", a )

a=21 // 为了方便测试，a 这里重新赋值为 21

a--

fmt.Printf("第七行 - a 的值为 %d\n", a )

}

以上实例运行结果：

第一行 - c 的值为 31

第二行 - c 的值为 11

第三行 - c 的值为 210

第四行 - c 的值为 2

第五行 - c 的值为 1

第六行 - a 的值为 22

第七行 - a 的值为 20

2.7.2 关系运算符（Relational operator）

下表列出了所有Go语言的关系运算符。假定 A 值为 10，B 值为 20。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| == | 检查两个值是否相等，如果相等返回 True 否则返回 False。 | (A == B) 为 False |
| != | 检查两个值是否不相等，如果不相等返回 True 否则返回 False。 | (A != B) 为 True |
| > | 检查左边值是否大于右边值，如果是返回 True 否则返回 False。 | (A > B) 为 False |
| < | 检查左边值是否小于右边值，如果是返回 True 否则返回 False。 | (A < B) 为 True |
| >= | 检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。 | (A >= B) 为 False |
| <= | 检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。 | (A <= B) 为 True |

表2.5 Go语言中的关系运算符

以下实例演示了关系运算符的用法：

package main

import "fmt"

func main() {

var a int = 21

var b int = 10

if( a == b ) {

fmt.Printf("第一行 - a 等于 b\n" )

} else {

fmt.Printf("第一行 - a 不等于 b\n" )

}

if ( a < b ) {

fmt.Printf("第二行 - a 小于 b\n" )

} else {

fmt.Printf("第二行 - a 不小于 b\n" )

}

if ( a > b ) {

fmt.Printf("第三行 - a 大于 b\n" )

} else {

fmt.Printf("第三行 - a 不大于 b\n" )

}

/\* Lets change value of a and b \*/

a = 5

b = 20

if ( a <= b ) {

fmt.Printf("第四行 - a 小于等于 b\n" )

}

if ( b >= a ) {

fmt.Printf("第五行 - b 大于等于 a\n" )

}

}

以上实例运行结果：

第一行 - a 不等于 b

第二行 - a 不小于 b

第三行 - a 大于 b

第四行 - a 小于等于 b

第五行 - b 大于等于 a

2.7.3 逻辑运算符（Logical operator）

下表列出了所有Go语言的逻辑运算符。假定 A 值为 True，B 值为 False。

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| && | 逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。 | (A && B) 为 False |
| || | 逻辑 OR 运算符。 如果两边的操作数有一个 True，则条件 True，否则为 False。 | (A || B) 为 True |
| ! | 逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。 | !(A && B) 为 True |

表2.6 Go语言中的逻辑运算符

以下实例演示了逻辑运算符的用法：

package main

import "fmt"

func main() {

var a bool = true

var b bool = false

if ( a && b ) {

fmt.Printf("第一行 - 条件为 true\n" )

}

if ( a || b ) {

fmt.Printf("第二行 - 条件为 true\n" )

}

/\* 修改 a 和 b 的值 \*/

a = false

b = true

if ( a && b ) {

fmt.Printf("第三行 - 条件为 true\n" )

} else {

fmt.Printf("第三行 - 条件为 false\n" )

}

if ( !(a && b) ) {

fmt.Printf("第四行 - 条件为 true\n" )

}

}

以上实例运行结果：

第二行 - 条件为 true

第三行 - 条件为 false

第四行 - 条件为 true

2.7.4 位运算符（Bitwise operator）

位运算符对整数在内存中的二进制位进行操作。

位运算符比一般的算术运算符速度要快，而且可以实现一些算术运算符不能实现的功能。如果要开发高效率程序，位运算符是必不可少的。位运算符用来对二进制位进行操作，包括：按位与（&）、按位或（|）、按位异或（^）、按位左移（<<）、按位右移（>>）。

假定 A = 60; B = 13; 其二进制数转换为：

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

Go 语言支持的位运算符如下表所示。假定 A 为60，B 为13：

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| & | 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。 | (A & B) 结果为 12, 二进制为 0000 1100 |
| | | 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 | (A | B) 结果为 61, 二进制为 0011 1101 |
| ^ | 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，当两对应的二进位相异时，结果为1。 | (A ^ B) 结果为 49, 二进制为 0011 0001 |
| << | 左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。 | A << 2 结果为 240 ，二进制为 1111 0000 |
| >> | 右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。 | A >> 2 结果为 15 ，二进制为 0000 1111 |

表2.7 Go语言中的位运算符

按位与（&）：对两个数进行操作，然后返回一个新的数，这个数的每个位都需要两个输入数的同一位都为1 时才为1。简单说就是：同一位同时为1则为1。

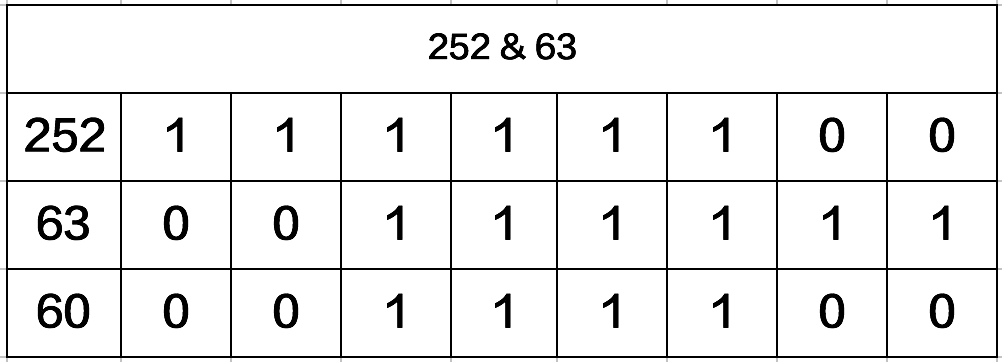


图 2.2

按位或（|）：比较两个数，然后返回一个新的数，这个数的每一位设置1的条件是任意一个数的同一位为1 则为1。简单说就是：同一位其中一个为1则为1。

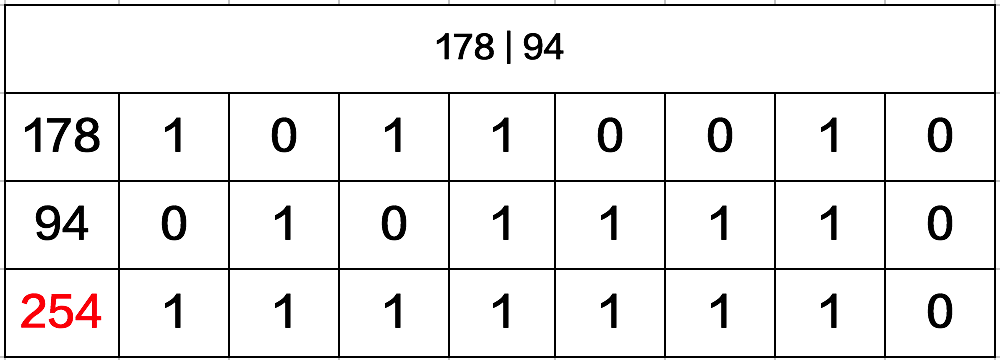


图 2.3

按位异或（^）：比较两个数，然后返回一个数，这个数的每个位设为1 的条件是两个输入数的同一位不同则为1，如果相同就设为0。简单说就是同一位不相同则为1。

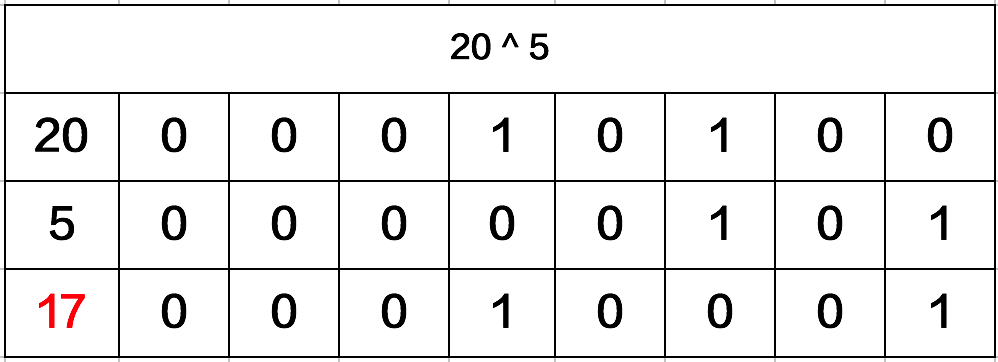


图 2.4

左移运算符（<<）是按二进制形式把所有的数字向左移动对应的位数，高位移出(舍弃)，低位的空位补零。

语法格式:

需要移位的数字 << 移位的次数

例如： 3 << 4，则是将数字3左移4位

计算过程：

3 << 4

首先把3转换为二进制数字0000 0000 0000 0000 0000 0000 0000 0011，然后把该数字高位(左侧)的两个零移出，其他的数字都朝左平移4位，最后在低位(右侧)的两个空位补零。则得到的最终结果是0000 0000 0000 0000 0000 0000 0011 0000，则转换为十进制是48。

在数字没有溢出的前提下，对于正数和负数，左移一位都相当于乘以2的1次方，左移n位就相当于乘以2的n次方。

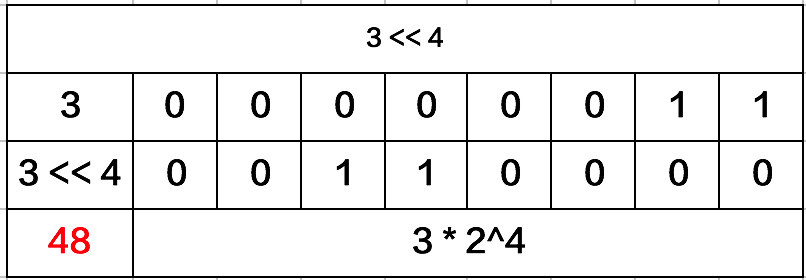


图 2.5

右移运算符（>>）是按二进制形式把所有的数字向右移动对应位移位数，低位移出(舍弃)，高位的空位补符号位，即正数补零，负数补1。

语法格式：

需要移位的数字 >> 移位的次数

例如11 >> 2，则是将数字11右移2位

计算过程：

11的二进制形式为：0000 0000 0000 0000 0000 0000 0000 1011，然后把低位的最后两个数字移出，因为该数字是正数，所以在高位补零。则得到的最终结果是0000 0000 0000 0000 0000 0000 0000 0010。转换为十进制是2。

右移一位相当于除2，右移n位相当于除以2的n次方。



图 2.6

以下实例演示了位运算符的用法：

package main

import "fmt"

func main() {

var a uint = 60 /\* 60 = 0011 1100 \*/

var b uint = 13 /\* 13 = 0000 1101 \*/

var c uint = 0

c = a & b /\* 12 = 0000 1100 \*/

fmt.Printf("第一行 - c 的值为 %d\n", c )

c = a | b /\* 61 = 0011 1101 \*/

fmt.Printf("第二行 - c 的值为 %d\n", c )

c = a ^ b /\* 49 = 0011 0001 \*/

fmt.Printf("第三行 - c 的值为 %d\n", c )

c = a << 2 /\* 240 = 1111 0000 \*/

fmt.Printf("第四行 - c 的值为 %d\n", c )

c = a >> 2 /\* 15 = 0000 1111 \*/

fmt.Printf("第五行 - c 的值为 %d\n", c )

}

以上实例运行结果：

第一行 - c 的值为 12

第二行 - c 的值为 61

第三行 - c 的值为 49

第四行 - c 的值为 240

第五行 - c 的值为 15

2.7.5 赋值运算符（Assignment operator）

下表列出了所有Go语言的赋值运算符。

|  |  |  |
| --- | --- | --- |
| 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 |
| <<= | 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 |

表2.8 Go语言中的赋值运算符

以下实例演示了赋值运算符的用法：

package main

import "fmt"

func main() {

var a int = 21

var c int

c = a

fmt.Printf("第 1 行 - = 运算符实例，c 值为 = %d\n", c )

c += a

fmt.Printf("第 2 行 - += 运算符实例，c 值为 = %d\n", c )

c -= a

fmt.Printf("第 3 行 - -= 运算符实例，c 值为 = %d\n", c )

c \*= a

fmt.Printf("第 4 行 - \*= 运算符实例，c 值为 = %d\n", c )

c /= a

fmt.Printf("第 5 行 - /= 运算符实例，c 值为 = %d\n", c )

c = 200;

c <<= 2

fmt.Printf("第 6行 - <<= 运算符实例，c 值为 = %d\n", c )

c >>= 2

fmt.Printf("第 7 行 - >>= 运算符实例，c 值为 = %d\n", c )

c &= 2

fmt.Printf("第 8 行 - &= 运算符实例，c 值为 = %d\n", c )

c ^= 2

fmt.Printf("第 9 行 - ^= 运算符实例，c 值为 = %d\n", c )

c |= 2

fmt.Printf("第 10 行 - |= 运算符实例，c 值为 = %d\n", c )

}

以上实例运行结果：

第 1 行 - = 运算符实例，c 值为 = 21

第 2 行 - += 运算符实例，c 值为 = 42

第 3 行 - -= 运算符实例，c 值为 = 21

第 4 行 - \*= 运算符实例，c 值为 = 441

第 5 行 - /= 运算符实例，c 值为 = 21

第 6行 - <<= 运算符实例，c 值为 = 800

第 7 行 - >>= 运算符实例，c 值为 = 200

第 8 行 - &= 运算符实例，c 值为 = 0

第 9 行 - ^= 运算符实例，c 值为 = 2

第 10 行 - |= 运算符实例，c 值为 = 2

2.7.6 其他运算符

下表列出了Go语言的其他运算符。

|  |  |  |
| --- | --- | --- |
| 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 |

表2.9 Go语言中的其他运算符

以下实例演示了其他运算符的用法：

package main

import "fmt"

func main() {

var a int = 4

var b int32

var c float32

var ptr \*int

/\* 运算符实例 \*/

fmt.Printf("第 1 行 - a 变量类型为 = %T\n", a );

fmt.Printf("第 2 行 - b 变量类型为 = %T\n", b );

fmt.Printf("第 3 行 - c 变量类型为 = %T\n", c );

/\* & 和 \* 运算符实例 \*/

ptr = &a /\* 'ptr' 包含了 'a' 变量的地址 \*/

fmt.Printf("a 的值为 %d\n", a);

fmt.Printf("\*ptr 为 %d\n", \*ptr);

}

以上实例运行结果：

第 1 行 - a 变量类型为 = int

第 2 行 - b 变量类型为 = int32

第 3 行 - c 变量类型为 = float32

a 的值为 4

\*ptr 为 4

2.8 运算符优先级

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

|  |  |
| --- | --- |
| 优先级 | 运算符 |
| 7 | ^ ! |
| 6 | \* / % << >> & &^ |
| 5 | + - | ^ |
| 4 | == != < <= >= > |
| 3 | <- |
| 2 | && |
| 1 | || |

表2.10 Go语言中的运算符优先级

当然，你可以通过使用括号来临时提升某个表达式的整体运算优先级。

以上实例运行结果：

package main

import "fmt"

func main() {

var a int = 20

var b int = 10

var c int = 15

var d int = 5

var e int;

e = (a + b) \* c / d; // ( 30 \* 15 ) / 5

fmt.Printf("(a + b) \* c / d 的值为 : %d\n", e );

e = ((a + b) \* c) / d; // (30 \* 15 ) / 5

fmt.Printf("((a + b) \* c) / d 的值为 : %d\n" , e );

e = (a + b) \* (c / d); // (30) \* (15/5)

fmt.Printf("(a + b) \* (c / d) 的值为 : %d\n", e );

e = a + (b \* c) / d; // 20 + (150/5)

fmt.Printf("a + (b \* c) / d 的值为 : %d\n" , e );

}

以上实例运行结果：

(a + b) \* c / d 的值为 : 90

((a + b) \* c) / d 的值为 : 90

(a + b) \* (c / d) 的值为 : 90

a + (b \* c) / d 的值为 : 50

2.8 小结

本章给大家讲解Go语言的基本语法，包括变量、常量、数据类型及运算符。学习任何一门编程语言，其基本语法无外乎这几部分。但是值得注意的是，本章包含了其他编程语言所没有的内容。首先是变量的多重赋值，其次是匿名变量，第三点是常量中iota的用法。掌握了基本语法有助于更好的理解代码世界。

第三章 Go语言流程控制

本章重点为大家介绍如下的内容：

* Go语言流程控制概述
* if条件判断语句
* switch分支语句
* Go语言中循环语句概述
* for循环语句及其多种语法形式
* for嵌套循环语句
* 循环控制语句（break、continue、goto）

3.1 Go 语言流程控制

流程控制是每种编程语言控制逻辑走向和执行次序的重要部分，流程控制是一门语言的经脉；流程控制语句包含条件判断语句、条件分支语句及循环语句。

Go语言的基本流程控制语句包括：

* if条件判断语句
* switch分支语句
* for循环语句
* goto跳转语句及break和continue循环控制语句

Go 语言提供了以下几种条件判断、分支语句

|  |  |
| --- | --- |
| 语句 | 描述 |
| if 语句 | if 语句由一个布尔表达式后紧跟一个或多个语句组成。 |
| if...else 语句 | if 语句后可以使用可选的else 语句, else 语句中的表达式在布尔表达式为 false 时执行。 |
| if 嵌套语句 | 你可以在if或else if语句中嵌入一个或多个if或else if语句。 |
| switch 语句 | switch语句用于基于不同条件执行不同动作。 |
| select 语句 | select语句类似于switch语句，但是select会随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。 |

表 3.1

Go 语言提供了以下几种循环语句

|  |  |
| --- | --- |
| 循环类型 | 描述 |
| for循环 | 重复执行语句块 |
| 循环嵌套 | 在 for 循环中嵌套一个或多个 for 循环 |

表 3.2

GO 语言支持以下几种循环控制语句

|  |  |
| --- | --- |
| 控制语句 | 描述 |
| break语句 | 经常用于中断当前 for 循环或跳出 switch 语句 |
| continue语句 | 跳过当前循环的剩余语句，然后继续进行下一轮循环。 |
| goto语句 | 将控制转移到被标记的语句。 |

表 3.3

3.2 if条件判断语句

Go 编程语言中 if 语句的语法如下：

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

}

if 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则不执行。

Go 编程语言中 if...else 语句的语法如下：

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

} else {

/\* 在布尔表达式为 false 时执行 \*/

}

if 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则执行 else 语句块。

Go 编程语言中 if...else if ... else 语句的语法如下：

if 布尔表达式1 {

/\* 在布尔表达式1为 true 时执行 \*/

} else if 布尔表达式2 {

/\* 在布尔表达式2为 true 时执行 \*/

} else {

/\* 在不满足以上条件时执行 \*/

}

if语句中的注意细节如下：

* 不需使用括号将条件包含起来;
* 大括号{}必须存在，即使只有一行语句;
* 左括号必须在if或else的同一行;
* 在if之后，条件语句之前，可以添加变量初始化语句，使用"；"进行分隔。

例：

1.用 if 语句判断数据奇数偶数

package main

import "fmt"

func main() {

num := 20

if num%2 == 0 {

fmt.Println(num, "偶数")

} else {

fmt.Println(num, "奇数")

}

}

2.判断学生平均成绩。有优、良、中等、及格、不及格等五档。

package main

import "fmt"

func main() {

score := 88

if score >= 90 {

fmt.Println("优秀")

} else if score >= 80 {

fmt.Println("良好")

} else if score >= 70 {

fmt.Println("中等")

} else if score >= 60 {

fmt.Println("及格")

} else if score < 60 {

fmt.Println("不及格")

}

}

if语句还有一个变体。它的语法是：

if statement; condition {

//代码块

}

例：

判断一个数是奇数还是偶数？

package main

import "fmt"

func main() {

if num := 10; num%2 == 0 {

fmt.Println(num, "偶数")

} else {

fmt.Println(num, "奇数")

}

}

需要注意的是，num的定义在if里，那么只能够在该if..else语句块中使用，否则编译器会报错的。

3.3 if嵌套语句

可以在 if 或 else if 语句中嵌入一个或多个 if 或 else if 语句。

Go 编程语言中 if...else 语句的语法如下：

if 布尔表达式 1 {

/\* 在布尔表达式 1 为 true 时执行 \*/

if 布尔表达式 2 {

/\* 在布尔表达式 2 为 true 时执行 \*/

}

}

可以以同样的方式在 if 语句中嵌套 else if...else 语句

例：

判断学生平均成绩。有优、良、中等、及格、不及格等五档。

package main

import "fmt"

func main() {

if score := 98; score >= 60 {

if score >= 70 {

if score >= 80 {

if score >= 90 {

fmt.Println("优")

} else {

fmt.Println("良")

}

} else {

fmt.Println("中等")

}

} else {

fmt.Println("及格")

}

} else {

fmt.Println("不及格")

}

}

3.4 switch分支语句

Go 编程语言中 switch 语句的语法如下：

switch var1 {

case val1:

...

case val2:

...

default:

...

}

switch 语句执行的过程自上而下，直到找到case匹配项，匹配项中无需使用break，因为Go语言中的switch默认给每个case自带break，因此匹配成功后不会向下执行其他的case分支，而是跳出整个switch。

变量 var1 可以是任何类型，而 val1 和 val2 则可以是同类型的任意值。类型不被局限于常量或整数，但必须是相同类型或最终结果为相同类型的表达式。

case后的值不能重复。

可以同时测试多个符合条件的值，也就是说case后可以有多个值，这些值之间使用逗号分割，例如：case val1, val2, val3。

Go语言中switch后的表达式可以省略，那么默认是switch true。

Go语言中的switch case因为自带break，所以匹配某个case后不会自动向下执行其他case，如需贯通后续的case，可以添加fallthrough（中文含义是：贯穿），强制执行后面的case分支。

fallthrough必须放在case分支的最后一行。如果它出现在中间的某个地方，编译器就会抛出错误（fallthrough statement out of place，含义是fallthrough不在合适的位置）。

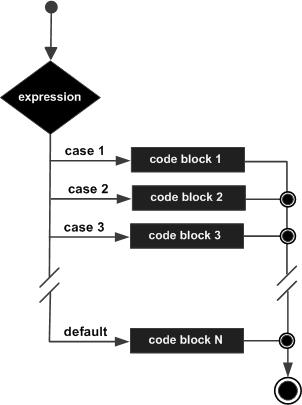


图 3.1

例：

1.判断学生平均成绩。有优、良、中等、及格、不及格等五档。

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

grade := ""

score := 78.5

//思考：以下代码逻辑错误在哪里？

//switch {

//case score < 60:

// grade = "E"

//case score >= 60:

// grade = "D"

//case score >= 70:

// grade = "C"

//case score >= 80:

// grade = "B"

//case score >= 90:

// grade = "A"

//}

switch { //switch后面省略不写，默认相当于：switch true

case score >= 90:

grade = "A"

case score >= 80:

grade = "B"

case score >= 70:

grade = "C"

case score >= 60:

grade = "D"

default:

grade = "E"

}

fmt.Printf("你的等级是: %s\n", grade)

fmt.Print("最终评价是: ")

switch grade {

case "A":

fmt.Printf("优秀!\n")

case "B":

fmt.Printf("良好\n")

case "C":

fmt.Printf("中等\n")

case "D":

fmt.Printf("及格\n")

default:

fmt.Printf("差\n")

}

}

2.判断某年某月的天数

package main

import "fmt"

func main() {

/\* 定义局部变量:年、月、日 \*/

year := 2008

month := 2

days := 0

switch month {

case 1, 3, 5, 7, 8, 10, 12:

days = 31

case 4, 6, 9, 11:

days = 30

case 2:

if (year%4 == 0 && year%100 != 0) || year%400 == 0 {

days = 29

} else {

days = 28

}

default:

days = -1

}

fmt.Printf("%d 年 %d 月的天数为：%d\n", year, month, days)

}

switch 语句还可以被用于 type-switch 来判断某个 interface 变量中实际存储的变量类型。

语法结构如下：

switch x.(type){

case type:

statement(s);

case type:

statement(s);

/\* 你可以定义任意个数的case \*/

default: /\* 可选 \*/

statement(s);

}

例：

package main

import "fmt"

func main() {

var x interface{}

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")

}

}

结果

x 的类型 :<nil>

3.5 Go循环语句

在不少实际问题中有许多具有规律性的重复操作，因此在程序中就需要重复执行某些语句。循环语句包括循环处理语句及循环控制语句。

循环处理语句有：

|  |  |
| --- | --- |
| 循环类型 | 描述 |
| for 循环 | 重复执行语句块 |
| 循环嵌套 | 在 for 循环中嵌套一个或多个 for 循环 |

表 3.4

循环控制语句可以控制循环体内语句的执行过程。循环控制语句有：

|  |  |
| --- | --- |
| 控制语句 | 描述 |
| break语句 | 经常用于中断当前 for 循环或跳出 switch 语句 |
| continue语句 | 跳过当前循环的剩余语句，然后继续进行下一轮循环。 |
| goto语句 | 将控制转移到被标记的语句。 |

表 3.5

循环语句表示条件满足，可以反复的执行某段代码。for是Go语言中唯一的循环语句，Go没有while、do...while循环。按语法结构来分，Go语言的for循环有4种形式，只有其中第一种使用分号。for循环中for关键字后不能加小括号。

3.5.1 for循环语法结构

1.基本for循环语法结构：

for 初始语句init; 条件表达式condition; 结束语句post {

//循环体代码

}

三个组成部分，即初始化、条件表达式和post都是可选的。

因此这种基本的for循环语法结构又能演化出四种略有不同的写法。

例：

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

fmt.Printf("%d ", i)

}

语法解释

（1）初始语句init：

初始语句是在第一次循环前执行的语句，一般为赋值表达式，给控制变量赋初始值。如果控制变量在此处被声明，其作用域将被局限在这个for的范围内；在for循环中声明的变量仅在循环范围内可用。初始语句可以省略不写，但是初始语句之后的分号必须要写。

i := 0

for ; i <= 10; i++ {

fmt.Printf("%d ", i)

}

（2）条件表达式condition：

条件表达式是控制循环与否的开关；如果表达式为true，则循环继续，否则结束循环；条件表达式可以省略不写，之后的分号必须要写；省略条件表达式默认形成无限循环。

i := 0

for ; ; i++ {

if (i > 20) {

break

}

fmt.Printf("%d ", i)

}

（3）结束语句post：

一般为赋值表达式，给控制变量递增或者递减；post语句将在循环的每次成功迭代之后执行。

for语句执行过程如下：

先执行初始化语句，对控制变量赋初始值。初始化语句只执行一次。

其次根据控制变量判断条件表达式的返回值，若其值为true，满足循环条件，则执行循环体内语句，之后执行 post语句，开始下一次循环。

执行post语句之后，将重新计算条件表达式的返回值，如果是true，循环将继续执行，否则循环终止。然后执行循环体外语句。

2.for关键字后只有一个条件表达式：

for 循环条件condition { }

效果类似其它编程语言中的while循环

例：

var i int

for i <= 10 {

fmt.Println(i)

i++

}

3.for关键字后无表达式：

for { }

效果与其它编程语言的for(;;) {}一致，此时for执行无限循环

例：

var i int

for {

if (i > 10) {

break

}

fmt.Println(i)

i++

}

4.for ... range：

for 循环的 range 格式

主要用于对字符串、slice、数组、map等进行迭代循环

for key, value := range oldMap {

newMap[key] = value

}

例：

遍历字符串，获得字符

func traverseString() {

str := "123ABCabc一丁丂"

for i, value := range str {

fmt.Printf("第 %d 位的ASCII值=%d ，字符是%c \n", i, value ,value)

}

}

打印结果：

第 0 位的ASCII值=49 ，字符是1

第 1 位的ASCII值=50 ，字符是2

第 2 位的ASCII值=51 ，字符是3

第 3 位的ASCII值=65 ，字符是A

第 4 位的ASCII值=66 ，字符是B

第 5 位的ASCII值=67 ，字符是C

第 6 位的ASCII值=97 ，字符是a

第 7 位的ASCII值=98 ，字符是b

第 8 位的ASCII值=99 ，字符是c

第 9 位的ASCII值=19968 ，字符是一

第 12 位的ASCII值=19969 ，字符是丁

第 15 位的ASCII值=19970 ，字符是丂

例：

遍历切片中元素

func traverseSlice() {

arr := []int{100, 200, 300}

for i, value := range arr {

fmt.Println(i, ":", value)

}

}

3.5.2 for循环使用案例

1.求1-100 的和

func summation() {

sum := 0

for i := 1; i <= 100; i++ {

sum += i;

}

fmt.Println(sum)

}

2.求1-100之间3的倍数的和

func summation2() {

i := 1

sum := 0

for i <= 100 {

if i%3 == 0 {

sum += i

fmt.Print(i)

if i < 99 {

fmt.Print("+")

} else {

fmt.Printf(" = %d \n", sum)

}

}

i++

}

}

3.截竹竿。32米竹竿，每次截1.5米，最快截几次之后能小于4米？

func cutBamboo() {

count := 0;

for i := 32.0; i >= 4; i -= 1.5 {

count++;

}

fmt.Println(count)

}

3.6 for嵌套循环语句

for [condition | ( init; condition; increment ) | Range] {

for [condition | ( init; condition; increment ) | Range] {

statement(s);

}

statement(s);

}

例：

1.打印直角三角形

func printRightTriangle() {

// 定义行数

lines := 8

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

for n := 0; n < 2\*i+1; n++ {

fmt.Print("\* ");

}

fmt.Println()

}

}

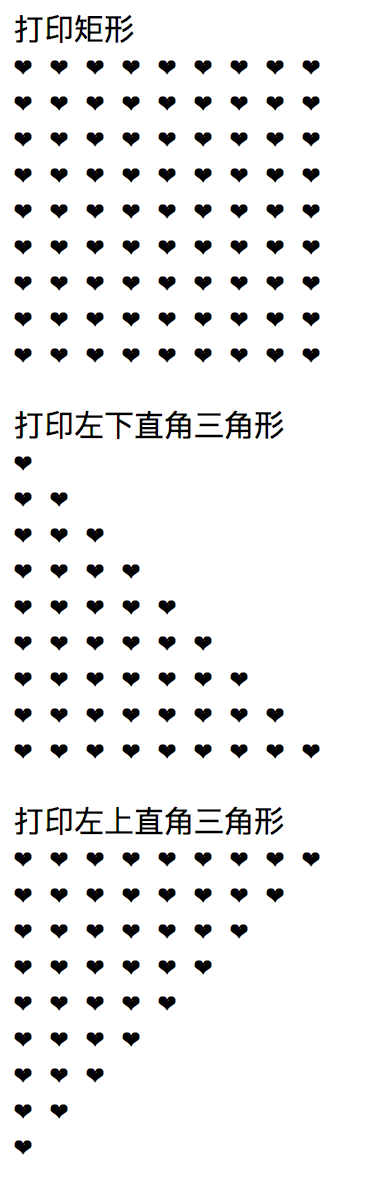


图 3.2

2.打印等腰三角形

func printTriangle() {

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

//控制每一行符号前的空格的数量

for m := 10; m > i; m-- {

fmt.Print(" ")

}

//控制每一行符号的数量

for j := 1; j <= 2\*i-1; j++ {

fmt.Print("❤ ")

}

fmt.Println()

}

}

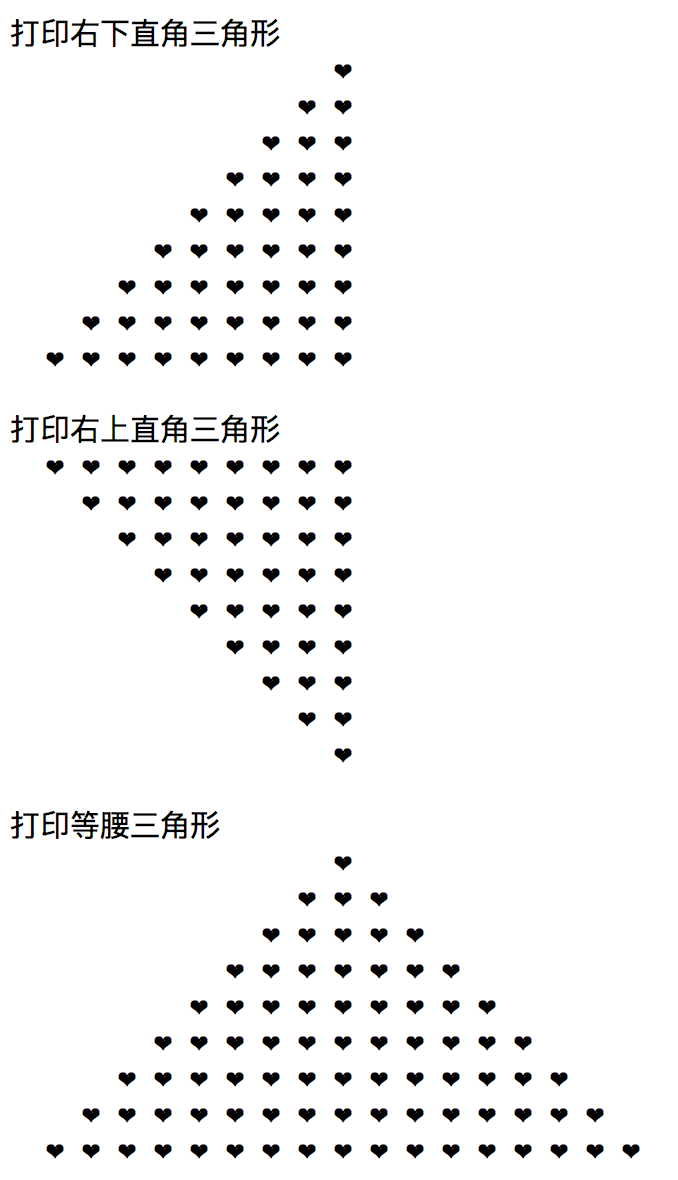


图 3.3

3.打印九九乘法表

func multiply99() {

for i := 1; i <= 9; i++ { // i 控制行数，是乘法中的第二个数。

for j := 1; j <= i; j++ { // j 控制每行的列数，是乘法中的第一个数。

fmt.Printf("%d\*%d=%d ", j, i, i\*j);

}

fmt.Println()

}

}

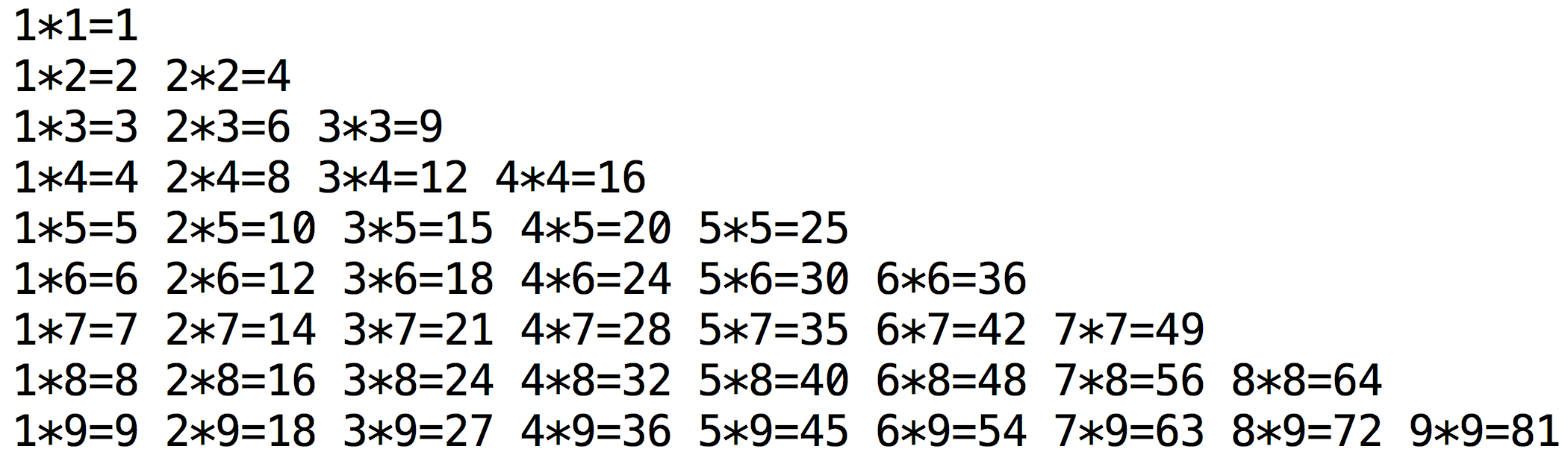


图 3.4

4.使用循环嵌套来输出 2 到 100 间的素数：

func printPrimeNumber() {

/\* 定义局部变量 \*/

fmt.Print("1-100的素数：")

var a, b int

for a = 2; a <= 100; a++ {

for b = 2; b <= (a / b); b++ {

if a%b == 0 {

break // 如果发现因子，则不是素数

}

}

if b > (a / b) {

fmt.Printf("%d\t", a)

}

}

}

3.7 循环控制语句

3.7.1 break语句

break：跳出循环体。break语句用于在结束其正常执行之前突然终止for循环，并开始执行循环之后的语句。

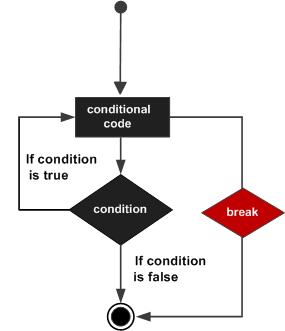


图 3.5

例：

func main() {

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

if i > 5 {

break // 如果i > 5，则循环终止（loop is terminated ）

}

fmt.Printf("%d ", i)

}

fmt.Printf("\nline after for loop")

}

3.7.2 continue语句

Go 语言的 continue 语句 有点像 break 语句。但是 continue 不是跳出循环，而是跳过当前循环执行下一次循环语句。for 循环中，执行 continue 语句会触发for增量语句的执行。换句话说，continue语句用于跳过for循环的当前迭代，循环将继续到下一个迭代。

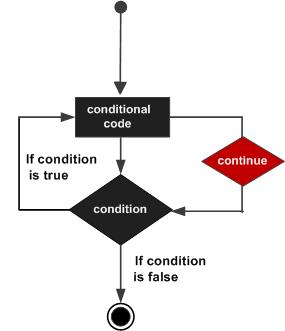


图 3.6

例：

func main() {

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

if i%2 == 0 {

continue

}

fmt.Printf("%d ", i)

}

}

备注：

break，continue的区别：

● break语句将无条件跳出并结束当前的循环， 然后执行循环体后的语句;

● continue语句是跳过当前的循环， 而开始执行下一次循环。

// break终止循环

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

if i == 5 {

break

}

fmt.Print(i)

}

//结果是：01234

//continue跳过某次循环

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

if i == 5 {

continue

}

fmt.Print(i);

}

//结果是：012346789

例：输出1-50之间所有不包含数字4的数（continue实现）

func eludeFour() {

fmt.Println("\n输出1-50之间所有不包含数字4的数")

//定义局部变量

num := 0

//循环开始

for num < 50 {

num++

/\* 跳过迭代 \*/

if num%10 == 4 || num/10%10 == 4 {

continue

}

fmt.Printf("%d\t", num)

}

}

3.7.3 goto语句

Go 语言的 goto 语句允许无条件地转移到程序指定的行。goto语句通常与条件语句配合使用。可用来实现条件转移，构成循环，跳出循环体等功能。但是，在结构化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。

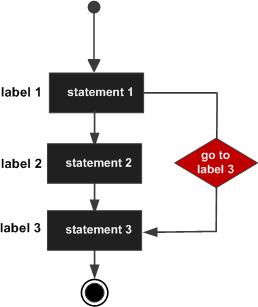


图 3.7

goto 语法格式如下：

LABEL: statement

goto LABEL

例：

输出1-50之间不包含4的数（goto实现）

func gotoTest() {

//定义局部变量

num := 0

/\* 跳过迭代 \*/

LOOP:

for num < 50 {

num++

if num%10 == 4 || num/10%10 == 4 {

goto LOOP

}

fmt.Printf("%d\t", num)

}

}

例：

求1-100的素数（借助goto跳转）

func printPrimeNumberGoto() {

var C, c int //声明变量

C = 1 /\*这里不写入FOR循环是因为For语句执行之初会将C的值变为1，当我们goto A时for语句会重新执行（不是重新一轮循环）\*/

LOOP:

for C < 100 {

C++ //C=1不能写入for这里就不能写入

for c = 2; c < C; c++ {

if C%c == 0 {

goto LOOP //若发现因子则不是素数

}

}

fmt.Printf("%d \t" , C)

}

}

3.8 小结

本章学习Go语言的流程控制语句及循环语句。流程控制包括if条件判断语句、switch分支语句。循环语句包括for循环、for嵌套循环以及与循环语句相关的语句——break、continue、goto。通过掌握流程控制，配合第二章学习的基本语法，我们已经可以写出一些简单的小程序。截止到现在我们看起来能做得还是很少，实际上，大家学会的东西将是未来从事软件开发过程中天天都在打交道的知识点，熟练的应用有助于理解程序的运行逻辑。

第四章 函数与指针

本章重点为大家介绍如下的内容：

* 什么是函数
* 声明函数
* 变量作用域
* 函数变量（函数作为值）
* 匿名函数
* 闭包
* 可变参数
* 递归函数
* 指针
* 函数参数传递（值传递与引用传递）

4.1 函数

函数是组织好的、可重复使用的执行特定任务的代码块。它可以提高应用程序的模块性和代码的重复利用率。

Go语言支持普通函数、匿名函数和闭包，从设计上对函数进行了优化和改进，让函数使用起来更加方便。Go语言的函数本身可以作为值进行传递、支持匿名函数和闭包（closure）、函数可以满足接口。

4.1.1 声明函数

普通函数需要先声明才能调用，一个函数的声明包括参数和函数名等。编译器通过声明才能了解函数应该怎样在调用代码和函数体之间传递参数和返回参数。

语法格式：

func 函数名（参数列表）（返回参数列表）{

//函数体

}

例：

func funcName (parametername type1, parametername type2...) (output1 type1, output2 type2...){

//逻辑代码

//返回多个值

return value1, value2...

}

使用func关键字定义一个函数。每一个函数由 func 开始声明。

funcName：函数名。函数名和参数列表一起构成了函数签名。函数名由字母、数字和下划线组成。函数名的第一个字母不能为数字。在同一个包内，函数名称不能重名。

parametername type：参数列表。参数就像一个占位符，定义函数时的参数叫做形式参数，形参变量是函数的局部变量；当函数被调用时，你可以将值传递给参数，这个值被称为实际参数。

参数列表指定的是参数类型、顺序、及参数个数。参数是可选的，也就是说函数也可以不包含参数。在参数列表中，如果有多个参数变量，则以逗号分隔；如果相邻变量是同类型，则可以将类型省略。

例：

func add （a , b int） {}

Go语言的函数支持可变参数。接受变参的函数是有着不定数量的参数的。

例：

func myfunc(arg ...int) {}

arg ...int告诉Go这个函数接受不定数量的参数。注意，这些参数的类型全部是int。在函数体中，变量arg是一个int的slice（切片）。

output1 type1, output2 type2：返回值列表。

返回值返回函数的执行结果，结束函数的执行。Go语言的函数可以返回多个值；返回值的类型可以是返回数据的数据类型，或者是变量名+变量类型的组合；函数声明时有返回值的，必须在函数体中使用return语句提供返回值列表；如果只有一个返回值且不声明返回值变量，那么可以省略包括返回值的括号；return后的数据，要保持和声明的返回值类型、数量、顺序一致。如果函数没有声明返回值，函数中也可以使用return关键字，用于强制结束函数。

4.1.2 变量作用域

作用域是指变量、常量、类型、函数的作用范围。

语言中变量可以在三个地方声明：

* 函数内定义的变量称为局部变量

在函数体内声明的变量称为局部变量，它们的作用域只在函数体内，参数和返回值变量也是局部变量。

* 函数外定义的变量称为全局变量

在函数体外声明的变量称之为全局变量，全局变量可以在整个包甚至外部包（被导出后）使用。全局变量可以在任何函数中使用。Go 语言程序中全局变量与局部变量名称可以相同，但是函数内的局部变量会被优先考虑。

* 函数中定义的参数称为形式参数

形式参数会作为函数的局部变量来使用。

例：

package main

import "fmt"

/\* 声明全局变量 \*/

var a1 int = 7

var b1 int = 9

func main() {

/\* main 函数中声明局部变量 \*/

a1, b1, c1 := 10, 20, 0

fmt.Printf("main()函数中 a1 = %d\n", a1) //10

fmt.Printf("main()函数中 b1 = %d\n", b1) //20

fmt.Printf("main()函数中 c1 = %d\n", c1) //0

c1 = sum(a1, b1)

fmt.Printf("main()函数中 c1 = %d\n", c1) //33

}

/\* 函数定义-两数相加 \*/

func sum(a1, b1 int) (c1 int) {

a1++

b1 += 2

c1 = a1 + b1

fmt.Printf("sum() 函数中 a1 = %d\n", a1) //11

fmt.Printf("sum() 函数中 b1 = %d\n", b1) //22

fmt.Printf("sum() 函数中 c1 = %d\n", c1) //33

return c1

}

输出结果：

main()函数中 a1 = 10

main()函数中 b1 = 20

main()函数中 c1 = 0

sum() 函数中 a1 = 11

sum() 函数中 b1 = 22

sum() 函数中 c1 = 33

main()函数中 c1 = 33

4.1.3 函数变量

在Go语言中，函数也是一种类型，可以和其它类型一样被保存在变量中。

可以通过type来定义一个自定义类型。函数的参数完全相同（包括：参数类型、个数、顺序），函数返回值相同。

例：

package main

import (

"fmt"

"strings"

)

func main() {

result := StringToLower("AbcdefGHijklMNOPqrstUVWxyz", processCase)

fmt.Println(result)

result = StringToLower2("AbcdefGHijklMNOPqrstUVWxyz", processCase)

fmt.Println(result)

}

//处理字符串，奇数偶数依次显示为大小写

func processCase(str string) string {

result := ""

for i, value := range str {

if i%2 == 0 {

result += strings.ToUpper(string(value))

} else {

result += strings.ToLower(string(value))

}

}

return result

}

func StringToLower(str string, f func(string) string) string {

fmt.Printf("%T \n", f)

return f(str)

}

type caseFunc func(string) string // 声明了一个函数类型。通过type关键字，caseFunc会形成一种新的类型。

func StringToLower2(str string, f caseFunc) string {

fmt.Printf("%T \n", f)

return f(str)

}

例：

package main

import "fmt"

type processFunc func(int) bool // 声明了一个函数类型

func main() {

slice := []int{1, 2, 3, 4, 5, 7}

fmt.Println("slice = ", slice)

odd := filter(slice, isOdd) // 函数当做值来传递

fmt.Println("奇数元素: ", odd)

even := filter(slice, isEven) // 函数当做值来传递

fmt.Println("偶数元素: ", even)

}

//判断元素是否是偶数

func isEven(integer int) bool {

if integer%2 == 0 {

return true

}

return false

}

//判断元素是否是奇数

func isOdd(integer int) bool {

if integer%2 == 0 {

return false

}

return true

}

//根据函数来处理切片，根据元素奇数偶数分组，返回新的切片

func filter(slice []int, f processFunc) []int {

var result []int

for \_, value := range slice {

if f(value) {

result = append(result, value)

}

}

return result

}

通过上面例子可以看到processFunc这个类型是一个函数类型，然后两个filter函数的参数和返回值与processFunc类型是一样的。用户可以实现很多种的逻辑，这样使得程序变得非常的灵活。

4.1.4 匿名函数

Go语言支持匿名函数，即在需要使用函数时，再定义函数，匿名函数没有函数名，只有函数体，函数可以被作为一种类型被赋值给变量，匿名函数也往往以变量方式被传递。匿名函数经常被用于实现回调函数、闭包等。

定义格式如下：

func(参数列表) （返回参数列表） {

//函数体

}

例：

1.在定义时调用匿名函数

package main

import "fmt"

func main() {

func(data int) {

fmt.Println("hello" , data)

}(100)

}

2.将匿名函数赋值给变量

package main

import "fmt"

func main() {

f:= func(data string) {

fmt.Println(data)

}

f("欢迎学习Go语言！")

}

3.匿名函数的用法——作回调函数

package main

import (

"fmt"

"math"

)

func main() {

//调用函数，对每个元素进行求平方根操作

arr := []float64{1, 9, 16, 25, 30}

visit(arr, func(v float64) {

v = math.Sqrt(v)

fmt.Printf("%.2f \n", v)

})

//调用函数，对每个元素进行求平方操作

visit(arr, func(v float64) {

v = math.Pow(v , 2)

fmt.Printf("%.0f \n", v)

})

}

//定义一个函数，遍历切片元素，对每个元素进行处理

func visit(list []float64, f func(float64)) {

for \_, value := range list {

f(value)

}

}

4.1.5 闭包

闭包并不是什么新奇的概念，它早在高级语言开始发展的年代就产生了。闭包（Closure）是词法闭包（Lexical Closure）的简称。对闭包的具体定义有很多种说法，大体可以分为两类：

* 一种说法认为闭包是符合一定条件的函数，比如这样定义闭包：闭包是在其词法上下文中引用了自由变量的函数。
* 另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。比如这样的定义：在实现深约束时，需要创建一个能显式表示引用环境的东西，并将它与相关的子程序捆绑在一起，这样捆绑起来的整体被称为闭包。函数 + 引用环境 = 闭包

上面的定义，一个认为闭包是函数，另一个认为闭包是函数和引用环境组成的整体。显然第二种说法更确切。闭包只是在形式和表现上像函数，但实际上不是函数。

函数是一些可执行的代码，这些代码在函数被定义后就确定了，不会在执行时发生变化，所以一个函数只有一个实例。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。闭包在某些编程语言中也被称为Lambda表达式。

函数本身不存储任何信息，只有与引用环境结合后形成的闭包才具有“记忆性”。函数是编译器静态的概念，而闭包是运行期动态的概念。

对象是附有行为的数据，而闭包是附有数据的行为。

闭包的价值在于：

* 加强模块化
* 抽象
* 简化代码

闭包有益于模块化编程，它能以简单的方式开发较小的模块，从而提高开发速度和程序的可复用性。和没有使用闭包的程序相比，使用闭包可将模块划分得更小。比如我们要计算一个数组中所有数字的和，这只需要循环遍历数组，把遍历到的数字加起来就行了。如果现在要计算所有元素的积呢？要打印所有的元素呢？解决这些问题都要对数组进行遍历，如果是在不支持闭包的语言中，我们不得不一次又一次重复地写循环语句。而这在支持闭包的语言中是不必要的。这种处理方法多少有点像回调函数，不过要比回调函数写法更简单，功能更强大。闭包是数据和行为的组合，这使得闭包具有较好抽象能力。

一个编程语言需要哪些特性来支持闭包呢？

函数是一阶值（First-class value，一等公民），即函数可以作为另一个函数的返回值或参数，还可以作为一个变量的值。函数可以嵌套定义，即在一个函数内部可以定义另一个函数。允许定义匿名函数。可以捕获引用环境，并把引用环境和函数代码组成一个可调用的实体；

例：

1.没有使用闭包进行计数的代码

package main

import "fmt"

func main() {

for i := 0; i < 5; i++ {

fmt.Printf("i=%d \t", i)

fmt.Println(add2(i))

}

}

func add2(x int) int {

sum := 0

sum += x

return sum

}

运行结果：

i=0 0

i=1 1

i=2 2

i=3 3

i=4 4

for循环每执行一次，sum都会清零，没有实现sum累加计数。

2.使用闭包函数实现计数器：

package main

import "fmt"

func main() {

pos := adder()

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

fmt.Printf("i=%d \t", i)

fmt.Println(pos(i))

}

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

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

fmt.Printf("i=%d \t", i)

fmt.Println(pos(i))

}

}

func adder() func(int) int {

sum := 0

return func(x int) int {

fmt.Printf("sum1=%d \t", sum)

sum += x

fmt.Printf("sum2=%d \t", sum)

return sum

}

}

运行结果为：

i=0 sum1=0 sum2=0 0

i=1 sum1=0 sum2=1 1

i=2 sum1=1 sum2=3 3

i=3 sum1=3 sum2=6 6

i=4 sum1=6 sum2=10 10

---------------------

i=0 sum1=10 sum2=10 10

i=1 sum1=10 sum2=11 11

i=2 sum1=11 sum2=13 13

i=3 sum1=13 sum2=16 16

i=4 sum1=16 sum2=20 20

例：

package main

import "fmt"

func main() {

myfunc := Counter()

//fmt.Printf("%T\n", myfunc)

fmt.Println("myfunc", myfunc)

/\* 调用 myfunc 函数，i 变量自增 1 并返回 \*/

fmt.Println(myfunc())

fmt.Println(myfunc())

fmt.Println(myfunc())

/\* 创建新的函数 nextNumber1，并查看结果 \*/

myfunc1 := Counter()

fmt.Println("myfunc1", myfunc1)

fmt.Println(myfunc1())

fmt.Println(myfunc1())

}

//计数器.闭包函数

func Counter() func() int {

i := 0

res := func() int {

i += 1

return i

}

//fmt.Printf("%T , %v \n" , res , res) //func() int , 0x1095af0

fmt.Println("Counter中的内部函数:", res) //0x1095af0

return res

}

4.1.6 可变参数

如果一个函数的参数类型一致，但个数不定，可以使用函数的可变参数。

语法格式:

func 函数名(参数名 ...类型) [(返回值列表)] {

//函数体

}

该语法格式定义了一个接受任何数目、任何类型参数的函数。这里特殊的语法是三个点“...”，在一个变量后面加上三个点后，表示从该处开始接受不定参数。

当要传递若干个值到不定参数函数中得时候，可以手动书写每个参数，也可以将一个slice传递给该函数，通过"..."可以将slice中的参数对应的传递给函数。

例：

计算学员考试总成绩及平均成绩

package main

import (

"fmt"

)

func main() {

//1传进n个参数

sum, avg, count := GetScore(90, 82.5, 73, 64.8)

fmt.Printf("学员共有%d门成绩，总成绩为：%.2f，平均成绩为：%.2f", count, sum, avg)

fmt.Println()

// 2传切片作为参数

scores := []float64{92, 72.5, 93, 74.5, 89, 87, 74}

sum, avg, count = GetScore(scores...)

fmt.Printf("学员共有%d门成绩，总成绩为：%.2f，平均成绩为：%.2f", count, sum, avg)

}

//累加求和，参数个数不定，参数个数从0-n

func GetScore(scores ...float64) (sum, avg float64, count int) {

for \_, value := range scores {

sum += value

count++

}

avg = sum / float64(count)

return

}

可变参数注意细节：

一个函数最多只能有一个可变参数

参数列表中还有其它类型参数，则可变参数写在所有参数的最后。

4.1.7 递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

递归函数必须满足以下两个条件：

1.在每一次调用自己时，必须是（在某种意义上）更接近于解；

2.必须有一个终止处理或计算的准则。

例：

求阶乘，计算阶乘n! = 1 x 2 x 3 x ... x n，用函数fact(n)表示，可以看出：fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n。所以，fact(n)可以表示为n x fact(n-1)，只有n=1时需要特殊处理。

package main

import "fmt"

func main() {

fmt.Println(factorial(5))

}

//通过递归实现阶乘

func factorial(n int) int {

if n == 0 {

return 1

}

return n \* factorial(n-1)

}

//通过循环实现阶乘

func getMultiple(num int) (result int) {

result = 1

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

result \*= i

}

return

}

递归的计算过程

===> factorial(5)

===> 5 \* factorial(4)

===> 5 \* (4 \* factorial(3))

===> 5 \* (4 \* (3 \* factorial(2)))

===> 5 \* (4 \* (3 \* (2 \* factorial(1))))

===> 5 \* (4 \* (3 \* (2 \* 1)))

===> 5 \* (4 \* (3 \* 2))

===> 5 \* (4 \* 6)

===> 5 \* 24

===> 120

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以用循环的方式实现，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈，每当函数返回，栈就会减一层。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

4.2 指针

指针是存储另一个变量的内存地址的变量。

变量是一种使用方便的占位符，变量都指向计算机的内存地址。

一个指针变量可以指向任何一个值的内存地址。

如下图：变量b的值为156，存储在内存地址0x1040a124。变量a持有b的地址，则a被认为指向b。



图 4.1

Go 语言的取地址符&，在一个变量前使用&，会返回该变量的内存地址。

例：

func main() {

a := 10

fmt.Printf("变量的地址: %x \n", &a)

}

运行结果：

变量的地址: c420014050

Go语言指针的最大特点是：指针不能运算（不同于C语言）。

在Go语言中如果对指针进行运算会报错：nvalid operation: p++ (non-numeric type \*int)

4.2.1 声明指针

声明指针，\*T是指针变量的类型，它指向T类型的值。

var 指针变量名 \*指针类型

\* 号用于指定变量是一个指针。

var ip \*int //指向整型的指针

var fp \*float32 //指向浮点型的指针

指针使用流程如下：

* 定义指针变量。
* 为指针变量赋值。
* 访问指针变量中指向地址的值。

获取指针的值时，在指针类型的变量前加上 \* 号（前缀）来获取指针所指向的内容。

获取一个指针意味着访问指针指向的变量的值。语法是：\*a

例：

func main() {

//声明实际变量

var a int = 120

//声明指针变量

var ip \*int

//给指针变量赋值，将变量a的地址赋值给ip

ip = &a

//打印a的类型和值

fmt.Printf("a 的类型是%T，值是%v \n", a, a)

//打印&a的类型和值

fmt.Printf("&a 的类型是%T，值是%v \n", &a, &a)

//打印ip的类型和值

fmt.Printf("ip 的类型是%T，值是%v \n", ip, ip)

//打印变量\*ip的类型和值

fmt.Printf("\*ip 变量的类型是%T，值是%v \n", \*ip, \*ip)

//打印变量\*&a的类型和值

fmt.Printf("\*&a 变量的类型是%T，值是%v \n", \*&a, \*&a)

fmt.Println(a, &a, \*&a)

fmt.Println(ip, &ip, \*ip, \*(&ip), &(\*ip))

}

运行结果：

a 的类型是int，值是120

&a 的类型是\*int，值是0xc420014050

ip 的类型是\*int，值是0xc420014050

\*ip 变量的类型是int，值是120

\*&a 变量的类型是int，值是120

120 0xc420014050 120

0xc420014050 0xc42000c028 120 0xc420014050 0xc420014050

例：

package main

import "fmt"

type Student struct {

name string

age int

married bool

sex int8

}

func main() {

var s1 = Student{"Steven", 35, true, 1}

var s2 = Student{"Sunny", 20, false, 0}

var a \*Student = &s1 //将s1的内存地址赋值给Student指针变量a

var b \*Student = &s2 //将s2的内存地址赋值给Student指针变量b

fmt.Println("\n---------------------")

fmt.Printf("s1类型为%T，值为%v \n", s1, s1)

fmt.Printf("s2类型为%T，值为%v \n", s2, s2)

fmt.Println("\n---------------------")

fmt.Printf("a类型为%T，值为%v \n", a, a)

fmt.Printf("b类型为%T，值为%v \n", b, b)

fmt.Println("\n---------------------")

fmt.Printf("\*a类型为%T，值为%v \n", \*a, \*a)

fmt.Printf("\*b类型为%T，值为%v \n", \*b, \*b)

fmt.Println("\n---------------------")

fmt.Println(s1.name, s1.age, s1.married, s1.sex)

fmt.Println(a.name, a.age, a.married, a.sex)

fmt.Println("\n---------------------")

fmt.Println(s2.name, s2.age, s2.married, s2.sex)

fmt.Println(b.name, b.age, b.married, b.sex)

fmt.Println("\n---------------------")

fmt.Println((\*a).name, (\*a).age, (\*a).married, (\*a).sex)

fmt.Println((\*b).name, (\*b).age, (\*b).married, (\*b).sex)

fmt.Println("\n---------------------")

fmt.Printf("&a类型为%T，值为%v\n", &a, &a)

fmt.Printf("&b类型为%T，值为%v\n", &b, &b)

fmt.Println("\n---------------------")

fmt.Println(&a.name, &a.age, &a.married, &a.sex)

fmt.Println(&b.name, &b.age, &b.married, &b.sex)

}

运行结果：

---------------------

s1类型为main.Student，值为{Steven 35 true 1}

s2类型为main.Student，值为{Sunny 20 false 0}

---------------------

a类型为\*main.Student，值为&{Steven 35 true 1}

b类型为\*main.Student，值为&{Sunny 20 false 0}

---------------------

\*a类型为main.Student，值为{Steven 35 true 1}

\*b类型为main.Student，值为{Sunny 20 false 0}

---------------------

Steven 35 true 1

Steven 35 true 1

---------------------

Sunny 20 false 0

Sunny 20 false 0

---------------------

Steven 35 true 1

Sunny 20 false 0

---------------------

&a类型为\*\*main.Student，值为0xc42000c028

&b类型为\*\*main.Student，值为0xc42000c030

---------------------

0xc42000a060 0xc42000a070 0xc42000a078 0xc42000a079

0xc42000a080 0xc42000a090 0xc42000a098 0xc42000a099

4.2.2 空指针

Go 空指针是指一个指针被定义后没有分配到任何变量的指针，它的值为 nil。nil 指针也称为空指针。nil在概念上和其它语言的null、None、NULL一样，都指代零值或空值。一个指针变量通常缩写为 ptr。

空指针判断：

if(ptr != nil) // ptr 不是空指针

if(ptr == nil) // ptr 是空指针

4.2.3 操作指针改变变量的数值

package main

import (

"fmt"

)

func main() {

b := 3158

a := &b

fmt.Println("b 的地址：", a) //0xc420014050

fmt.Println("\*a 的值：", \*a) //3158

\*a++

fmt.Println("b 的新值：", b)//3159

}

运行结果

b 的地址： 0xc420014050

\*a 的值：3158

b 的新值： 3159

4.2.4 使用指针作为函数的参数

基本数据类型指针作为函数参数

例：

package main

import (

"fmt"

)

func main() {

a := 58

fmt.Println("函数调用之前a的值：", a)

fmt.Printf("%T \n", a)

fmt.Printf("%x \n", &a)

//b := &a

var b \*int = &a

change(b)

fmt.Println("函数调用之后的a的值：", a)

}

func change(val \*int) {

\*val = 15

}

运行结果

函数调用之前a的值： 58

int

c420014050

函数调用之后的a的值： 15

例：

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

a := 100

b := 200

//返回值的写法

a, b = swap0(a, b)

//指针作为参数的写法

swap(&a, &b)

fmt.Printf("交换后 a 的值 : %d\n", a)

fmt.Printf("交换后 b 的值 : %d\n", b)

}

//具有返回值的惯用写法

func swap0(x, y int) (int , int) {

return y, x

}

//指针作为参数的写法

func swap(x \*int, y \*int) {

\*x, \*y = \*y, \*x

}

例：

将一个指向切片的指针传递给函数。虽然将指针传递给一个切片作为函数的参数，可以实现对该切片中元素的修改，但这并不是实现这一目标的惯用方法。惯用做法是使用切片。

package main

import "fmt"

func main() {

a := [3]int{89, 90, 91}

modify(&a)

fmt.Println(a)

}

func modify(arr \*[3]int) {

(\*arr)[0] = 189

}

运行结果

[189 90 91]

建议做法是使用切片来实现更改元素数值，代码如下：

package main

import "fmt"

func main() {

a := []int{89, 90, 91}

modify(a[:])

fmt.Println(a)

}

func modify(sls []int) {

sls[0] = 190

}

4.2.5指针数组

指针数组是指元素为指针类型的数组。

定义一个指针数组，例如：var ptr [3]\*string ；

有一个元素个数相同的数组，将该数组中每个元素的地址赋值给该指针数组。也就是说该指针数组与某一个数组完全对应。

可以通过\*指针变量获取到该地址所对应的数值。

例：

package main

import "fmt"

const COUNT int = 4

func main() {

a := [COUNT]string{"abc", "ABC", "123", "一二三"}

i := 0

//定义指针数组

var ptr [COUNT]\*string

fmt.Printf("%T , %v \n", ptr, ptr)

for i = 0; i < COUNT; i++ {

//将数组中每个元素的地址赋值给指针数组

ptr[i] = &a[i]

}

fmt.Printf("%T , %v \n", ptr, ptr)

//获取指针数组中第一个值，其实就是一个地址

fmt.Println(ptr[0])

//根据数组元素的每个地址获取该地址所指向的元素的数值

for i = 0; i < COUNT; i++ {

fmt.Printf("a[%d] = %s \n", i, \*ptr[i])

}

}

4.2.6 指针的指针

如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。当定义一个指向指针的指针变量时，第一个指针存放第二个指针的地址，第二个指针存放变量的地址：



图 4.2

指向指针的指针变量声明格式如下：

var ptr \*\*int

指向指针的指针变量为整型，访问指向指针的指针变量值需要使用两个 \* 号。

例：

package main

import "fmt"

func main() {

var a int

var ptr \*int

var pptr \*\*int

a = 1234

/\* 指针 ptr 地址 \*/

ptr = &a

fmt.Println("ptr" , ptr)

/\* 指向指针 ptr 地址 \*/

pptr = &ptr

fmt.Println("pptr" , ptr)

/\* 获取 pptr 的值 \*/

fmt.Printf("变量 a = %d\n", a)

fmt.Printf("指针变量 \*ptr = %d\n", \*ptr)

fmt.Printf("指向指针的指针变量 \*\*pptr = %d\n", \*\*pptr)

}

运行结果：

ptr 0xc420014050

pptr 0xc420014050

变量 a = 1234

指针变量 \*ptr = 1234

指向指针的指针变量 \*\*pptr = 1234

4.3 函数的参数传递

函数如果使用参数，该参数变量称为函数的形参。形参就像定义在函数体内的局部变量。调用函数，可以通过两种方式来传递参数。即：值传递和引用传递，或者叫做传值和传引用。

4.3.1 值传递（传值）

值传递是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到原内容数据。

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到原内容数据。

每次调用函数，都将实参拷贝一份再传递到函数中。每次拷贝一份，性能是不是就下降了呢？其实Go语言中使用指针和值传递配合就避免了性能降低问题，也就是通过传指针参数来解决实参拷贝的问题。

4.3.2 引用传递（传引用）

引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到原内容数据。

严格来说Go语言只有值传递一种传参方式，Go语言是没有引用传递的。

Go语言中可以借助传指针来实现引用传递的效果。函数参数使用指针参数，传参时其实是在拷贝一份指针参数，也就是拷贝了一份变量地址。

函数的参数如果是指针，当函数调用时，虽然参数仍然是按拷贝传递的，但是此时仅仅只是拷贝一个指针，也就是一个内存地址，这样就不用担心实参拷贝造成的内存浪费、时间开销、性能降低的情况。

传指针使得多个函数能操作同一个对象。传指针更轻量级 (8bytes)，只需要传内存地址。如果参数是非指针参数，那么值传递的过程中，每次在拷贝上面就会花费相对较多的系统开销（内存和时间）。所以当要传递大的结构体的时候，用指针是一个明智的选择。

Go语言中slice、map、chan类型的实现机制都是类似指针，所以可以直接传递，而不必取地址后传递指针。

例：

1.函数传传int型参数

package main

import "fmt"

func main() {

a := 10

fmt.Printf("1.变量a的内存地址：%p ，值为：%v \n\n", &a, a) //10

fmt.Printf("========int型变量a的内存地址：%p \n\n", a) //？？？%!p

//传值

changeIntVal(a)

fmt.Printf("2.changeIntVal函数调用之后：变量a的内存地址：%p ，值为：%v \n\n", &a, a) //10

//传引用

changeIntPtr(&a)

fmt.Printf("3.changeIntPtr函数调用之后：变量a的内存地址：%p ，值为：%v \n\n", &a, a) //50

}

func changeIntVal(a int) {

fmt.Printf("--------changeIntVal函数内：值参数a的内存地址：%p ，值为：%v \n", &a, a) //10

a = 90

}

func changeIntPtr(a \*int) {

fmt.Printf("--------changeIntPtr函数内：指针参数a的内存地址：%p ，值为：%v \n", &a, a) //地址

\*a = 50

}

2.函数传值和传引用\_传slice型参数

package main

import "fmt"

func main() {

a := []int{1, 2, 3, 4}

fmt.Printf("1、变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[1,2,3,4]

fmt.Printf("切片型变量a内存地址是：%p \n\n", a)//可以获取到地址，类似：0xc420018080

//传值

changeSliceVal(a)

fmt.Printf("2、changeSliceVal函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[1,2,3,4]

//传引用

changeSlicePtr(&a)

fmt.Printf("3、changeSlicePtr函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[250,2,3,4]

}

func changeSliceVal(a []int) {

fmt.Printf("----------changeSliceVal函数内：值参数a的内存地址是：%p ，值为：%v \n", &a, a) //[1,2,3,4]

fmt.Printf("----------changeSlicePtr函数内：值参数a的内存地址是：%p \n", a)

a[0] = 99

}

func changeSlicePtr(a \*[]int) {

fmt.Printf("----------changeSlicePtr函数内：指针参数a的内存地址是：%p ，值为：%v \n", &a, a) //&[1,2,3,4]

(\*a)[1] = 250

}

3.函数传值和传引用\_传数组

package main

import "fmt"

func main() {

a := [4]int{1, 2, 3, 4}

fmt.Printf("1、变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[1,2,3,4]

fmt.Printf("数组型变量a内存地址是：%p \n\n", a)//可以获取到地址？

//传值

changeArrayVal(a)

fmt.Printf("2、changeArrayVal函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[99,2,3,4]

//传引用

changeArrayPtr(&a)

fmt.Printf("3、changeArrayPtr函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//[99,250,3,4]

}

func changeArrayVal(a [4]int) {

fmt.Printf("----------changeArrayVal函数内：值参数a的内存地址是：%p ，值为：%v \n", &a, a) //[1,2,3,4]

fmt.Printf("----------changeArrayPtr函数内：值参数a的内存地址是：%p \n", a) //获取不到地址

a[0] = 99

}

func changeArrayPtr(a \*[4]int) {

fmt.Printf("----------changeArrayPtr函数内：指针参数a的内存地址是：%p ，值为：%v \n", &a, a) //&[1,2,3,4]

(\*a)[1] = 250

}

4.函数传值和传引用\_传struct结构体

package main

import "fmt"

type Teacher struct {

name string

age int

married bool

sex int8

}

func main() {

a := Teacher{"Steven", 35, true, 1}

fmt.Printf("1、变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//{Steven 35 true 1}

fmt.Printf("struct型变量a内存地址是：%p \n\n", a)//可以获取到地址？

//传值

changeStructVal(a)

fmt.Printf("2、changeArrayVal函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//{Steven 35 true 1}

//传引用

changeStructPtr(&a)

fmt.Printf("3、changeArrayPtr函数调用后：变量a的内存地址是：%p ，值为：%v \n\n", &a, a)//

}

func changeStructVal(a Teacher) {

fmt.Printf("----------changeArrayVal函数内：值参数a的内存地址是：%p ，值为：%v \n", &a, a) //

fmt.Printf("----------changeArrayPtr函数内：值参数a的内存地址是：%p \n", a) //获取不到地址?

a.name = "Josh"

a.age = 29

a.married = false

}

func changeStructPtr(a \*Teacher) {

fmt.Printf("----------changeArrayPtr函数内：指针参数a的内存地址是：%p ，值为：%v \n", &a, a) //{Daniel 20 false 1}

(\*a).name = "Daniel"

(\*a).age = 20

(\*a).married = false

}

Go语言中所有的传参都是值传递（传值），都是一个副本，一个拷贝。

拷贝的内容有时候是值类型（int、string、bool、数组、struct属于值类型），这样就在函数中就无法修改原内容数据；有的是引用类型（指针、slice、map、chan属于引用类型），这样就可以修改原内容数据。

是否可以修改原内容数据，和传值、传引用没有必然的关系。在C++中，传引用肯定是可以修改原内容数据的，在Go语言里，虽然只有传值，但是我们也可以修改原内容数据，因为参数可以是引用类型。

传引用和引用类型是两个概念。虽然Go语言只有传值一种方式，但是可以通过传引用类型变量达到跟传引用一样的效果。

4.4 小结

本章学习Go语言的函数及指针。由于Go语言的特性决定，我们将一直以函数化的方式编写所有的代码，函数本身也将成为我们细分代码的最重要的工具，尽可能的遵守一个函数只做一件事情的原则，在函数功能设计时尽可能的将单一职责考虑在内。指针同样也是非常重要的概念，在某些场景，我们必须依赖指针，虽然指针理解难度较大，但是经过不断的练习，依旧可以很好的掌握。

第五章 Go语言内置容器

本章重点为大家介绍如下的内容：

* 数组的用法（数组声明、元素访问、值类型、多维数组）
* 切片的用法（创建切片、元素遍历、动态增减元素、内容复制）
* 冒泡排序
* map（概念、声明、初始化和赋值、遍历map、键值对是否存在、map元素删除、清空map）
* 列表list（概念、声明、初始化、遍历list、插入元素、从list中删除元素）

5.1 数组(array)

5.1.1 数组简介

Go 语言提供了数组类型的数据结构。数组是具有相同类型的一组长度固定的数据序列，这种类型可以是任意的基本数据类型或复合数据类型及自定义类型。

数组元素可以通过索引下标（位置）来读取或者修改元素数据。索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。数组的下标取值范围是从0开始，到长度减1。数组一旦定义后，大小不能更改。

5.1.2 一维数组

Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

var 变量名 [数组长度] 数据类型

数组长度必须是整数且大于 0，未初始化的数组不是nil，也就是说没有空数组（与切片不同）

例：

var lines [5]int

var titles [10]string

数组可以在声明的时候直接赋值，这种操作叫数组初始化。

例：

var lines = [5]int{2,4,6,8,10}

var titles = []string{"编号","部门","教研室","姓名"}

初始化数组中 {} 中的元素个数不能大于 [] 中的数字。如果忽略 [] 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小。

可以忽略声明中数组的长度并将其替换为…，编译器会自动计算长度。

例：

var nums = [...]int{1 , 2 , 3 , 4 ,5 }

该实例没有设置数组的大小，与上面的实例是一样的效果。

数组元素可以通过索引（位置）来读取。格式为数组名后加中括号，中括号中为索引的值。例如：

balance[3] = 4

以上实例为数组第4个元素赋值。数组元素可以通过索引（位置）来读取（或者修改），索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。

如果已知一个数组，希望得到数组的长度，需要使用len()函数获得数组长度。通过将数组作为参数传递给len()函数，得到数组的长度。忽略声明中数组的长度并将其替换为…，编译器可以找到长度。

例：

package main

import (

"fmt"

)

func main() {

var titles = []string{"编号","部门","教研室","姓名"}

var persons = [...]string{"小明","小强","小亮"}

fmt.Print("titles:")

fmt.Println(len(titles))

fmt.Print("persons:")

fmt.Println(len(persons))

}

输出结果为：

titles:4

persons:3

通过循环语句与len()函数，可以实现遍历数组所有数据；通过for range也可以实现同样的功能。

例：

package main

import "fmt"

func main() {

a := [4]float64{67.7, 89.8, 21, 78}

b := [...]int{2, 3, 5}

//遍历数组方式1

for i := 0; i < len(a); i++ {

fmt.Print(a[i], "\t")

}

fmt.Println()

// 遍历数组方式2

for \_, value := range b {

fmt.Print(value, "\t")

}

}

输出结果：

67.7 89.8 21 78

2 3 5

5.1.3 多维数组

Go 语言支持多维数组，以下为常用的多维数组声明方式：

var variable\_name [SIZE1][SIZE2]...[SIZEn] variable\_type

以下实例声明了三维的整型数组：

var threedim [5][10][4]int

二维数组是最简单的多维数组，二维数组本质上是由一维数组组成的。二维数组定义方式如下：

var arrayName [ x ][ y ] variable\_type

例：

a = [3][4]int{

{0, 1, 2, 3} , /\* 第一行索引为 0 \*/

{4, 5, 6, 7} , /\* 第二行索引为 1 \*/

{8, 9, 10, 11} /\* 第三行索引为 2 \*/

}

二维数组通过指定坐标来访问。如数组中的行索引与列索引。

例：

int val = a[2][3]

以上实例访问了二维数组 val 第三行的第四个元素。

二维数组可以使用循环嵌套的方式实现数组遍历。

例：

package main

import "fmt"

func main() {

/\* 数组 - 5 行 2 列\*/

var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}

fmt.Println(len(a))

fmt.Println(len(a[0]))

/\* 输出数组元素 \*/

for i := 0; i < len(a); i++ {

for j := 0; j < len(a[0]); j++ {

fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )

}

}

}

Go中的数组是值类型，而不是引用类型。这意味着当它们被分配给一个新变量时，将把原始数组的副本分配给新变量。如果对新变量进行了更改，则不会在原始数组中反映。当将数组传递给函数作为参数时，它们将通过值传递，而原始数组将保持不变。

例：

package main

import "fmt"

func main() {

a := [...]string{"USA", "China", "India", "Germany", "France"}

b := a // a copy of a is assigned to b

b[0] = "Singapore"

fmt.Println("a ：", a)

fmt.Println("b ： ", b)

}

运行结果：

a ： [USA China India Germany France]

b ： [Singapore China India Germany France]

5.2 切片(Slice)

5.2.1 切片简介

Go 语言切片是对数组的抽象。Go语言中数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型切片("动态数组")；

与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。切片本身没有任何数据，它们只是对现有数组的引用。切片与数组相比，不需要设定长度，在[]中不用设定值，相对来说比较自由。从概念上面来说slice像一个结构体，这个结构体包含了三个元素：

* 指针，指向数组中slice指定的开始位置；
* 长度，即slice的长度；
* 最大长度，也就是slice开始位置到数组的最后位置的长度；

5.2.2 切片的语法

1.切片在声明时不需要说明长度，使用一个未指定长度的数组来定义切片如下：

var identifier []type

采用这种声明方式，且未初始化的切片称为空切片。这种切片默认为 nil，长度为 0。

2.使用make()函数来创建切片如下：

var slice1 []type = make([]type, len)

可以简写为：slice1 := make([]type, len)

可以指定容量，其中capacity为可选参数：make([]T, length, capacity)

package main

import "fmt"

func main() {

var numbers = make([]int,3,5)

fmt.Printf("%T\n" , numbers)

fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)

}

切片可以在声明的时候直接赋值，这种操作叫切片初始化。

1.直接初始化切片

例：

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

2.通过对数组或切片截取来初始化切片

* s := arr[:]，切片中包含数组所有元素；
* s := arr[startIndex:endIndex]，将arr中从下标startIndex到endIndex-1 下的元素创建为一个新的切片（前闭后开），长度为endIndex-startIndex；
* s := arr[startIndex:]，缺省endIndex时将表示一直到arr的最后一个元素；
* s := arr[:endIndex]，缺省startIndex时将表示从arr的第一个元素开始。

例：

package main

import "fmt"

func main() {

arr := [5]int {1,2,3,4,5}

s := arr[:]//完全截取

fmt.Println(s)

s = arr[1:2]//截取从索引1到索引2

fmt.Println(s)

s = arr[3:]//从索引3开始截取到结束

fmt.Println(s)

s = arr[:2]//截取到索引2

fmt.Println(s)

}

输出结果：

[1 2 3 4 5]

[2]

[4 5]

[1 2]

3.通过切片截取来初始化切片：

可以通过设置下限及上限来设置截取切片 [lower-bound:upper-bound]

package main

import "fmt"

func main() {

/\* 创建切片 \*/

numbers := []int{0,1,2,3,4,5,6,7,8}

printSlice(numbers)

/\* 打印原始切片 \*/

fmt.Println("numbers ==", numbers)//[0 1 2 3 4 5 6 7 8]

/\* 打印子切片从索引1(包含) 到索引4(不包含)\*/

fmt.Println("numbers[1:4] ==", numbers[1:4])//[1 2 3 ]

/\* 默认下限为 0\*/

fmt.Println("numbers[:3] ==", numbers[:3])//[0 1 2]

/\* 默认上限为 len(s)\*/

fmt.Println("numbers[4:] ==", numbers[4:])//[4 5 6 7 8]

/\* 打印子切片从索引 0(包含) 到索引 2(不包含) \*/

number2 := numbers[:2]

printSlice(number2) //[0 1]

/\* 打印子切片从索引 2(包含) 到索引 5(不包含) \*/

number3 := numbers[2:5]

printSlice(number3) // [2 3 4]

}

func printSlice(x []int){

fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)

}

运行结果：

len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]

numbers == [0 1 2 3 4 5 6 7 8]

numbers[1:4] == [1 2 3]

numbers[:3] == [0 1 2]

numbers[4:] == [4 5 6 7 8]

len=2 cap=9 slice=[0 1]

len=3 cap=7 slice=[2 3 4]

5.2.3 len() 和 cap() 函数

切片的长度是切片中元素的数量。切片的容量是从创建切片的索引开始的底层数组中元素的数量。切片是可索引的，并且可以由 len() 方法获取长度。切片提供了计算容量的方法 cap()， 可以测量切片最长可以达到多少。切片实际的是获取数组的某一部分，len切片<=cap切片<=len数组。

package main

import "fmt"

func main() {

sliceCap()

}

func sliceCap() {

arr0 := [...]string{"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k"}

fmt.Println("cap(arr0)=", cap(arr0), arr0) //[a b c d e f g h i j k]

//截取数组，形成切片

s01 := arr0[2:8]

fmt.Printf("%T \n", s01) //[]string

fmt.Println("cap(s01)=", cap(s01), s01) //9，[c d e f g h]

s02 := arr0[4:7]

fmt.Println("cap(s02)=", cap(s02), s02) //7，[e f g]

//截取切片，形成切片

s03 := s01[3:9]

fmt.Println("截取s01[3:9]后形成s03：", s03) //[f g h i j k]

s04 := s02[4:7]

fmt.Println("截取s02[4:7]后形成s04：", s04) //[i j k]

//切片是引用类型

s04[0] = "x"

fmt.Print(arr0, s01, s02, s03, s04)

}

输出结果：

cap(arr0)= 11 [a b c d e f g h i j k]

[]string

cap(s01)= 9 [c d e f g h]

cap(s02)= 7 [e f g]

截取s01[3:9]后形成s03： [f g h i j k]

截取s02[4:7]后形成s04： [i j k]

[a b c d e f g h x j k] [c d e f g h] [e f g] [f g h x j k] [x j k]

slice没有自己的任何数据，它只是底层数组的一个引用。对slice所做的任何修改都将反映在底层数组中。数组是值类型，而切片是应用类型。

例：

package main

import "fmt"

func main() {

a := [4]float64{67.7, 89.8, 21, 78}

b := []int{2, 3, 5}

fmt.Printf("变量a —— 地址：%p ， 类型：%T，数值：%v，长度：%d \n", &a, a, a, len(a))

fmt.Printf("变量b —— 地址：%p ， 类型：%T，数值：%v，长度：%d \n", &b, b, b, len(b))

c := a

d := b

fmt.Printf("变量c —— 地址：%p ， 类型：%T，数值：%v，长度：%d \n", &c, c, c, len(c))

fmt.Printf("变量d —— 地址：%p ， 类型：%T，数值：%v，长度：%d \n", &d, d, d, len(d))

a[1] = 200

fmt.Println("a=", a, "c=", c)

d[0] = 100

fmt.Println("b=", b, "d=", d)

}

运行结果：

变量a —— 地址：0xc000010300 ， 类型：[4]float64，数值：[67.7 89.8 21 78]，长度：4

变量b —— 地址：0xc0000044a0 ， 类型：[]int，数值：[2 3 5]，长度：3

变量c —— 地址：0xc0000103a0 ， 类型：[4]float64，数值：[67.7 89.8 21 78]，长度：4

变量d —— 地址：0xc000004520 ， 类型：[]int，数值：[2 3 5]，长度：3

a= [67.7 200 21 78] c= [67.7 89.8 21 78]

b= [100 3 5] d= [100 3 5]

当多个片共享相同的底层数组时，每个元素所做的更改将在数组中反映出来。

例：

package main

import "fmt"

func main() {

//定义数组

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

//根据数组截取切片

nums1 := arr[:]

nums2 := arr[:]

fmt.Println("arr=", arr)//[1 2 3]

nums1[0] = 100

fmt.Println("arr=", arr)//[100 2 3]

nums2[1] = 200

fmt.Println("arr=", arr)//[100 200 3]

}

运行结果：

arr= [1 2 3]

arr= [100 2 3]

arr= [100 200 3]

5.2.4 append() 和 copy() 函数

1.函数append()

append()函数实现往切片尾部追加新元素，函数可以向slice里面追加一个或者多个元素，也可以追加一个切片。

append函数会改变slice所引用的数组的内容，从而影响到引用同一数组的其它slice。当使用append追加元素到切片时，如果容量不够（也就是(cap-len) == 0），Go就会创建一个新的内存地址来储存元素。

2.函数copy()

copy()函数实现复制切片元素。将源切片中的元素复制到目标切片中，返回复制的元素的个数

copy函数是不会建立源切片与目标切片之间的联系。也就是两个切片不存在联系，一个修改不影响另一个。

例：

package main

import "fmt"

func main() {

numbers := make([]int , 0 , 20)

numbers = append(numbers, 2, 3, 4, 5, 6, 7)

numbers1 := make([]int, len(numbers), (cap(numbers))\*2)

/\* 拷贝 numbers 的内容到 numbers1 \*/

count := copy(numbers1, numbers)

fmt.Println("拷贝个数：", count)

printSlices("numbers1:", numbers1)

numbers[len(numbers)-1] = 99

numbers1[0] = 100

/\*numbers1与numbers两者不存在联系，numbers发生变化时，

numbers1是不会随着变化的。也就是说copy方法是不会建立两个切片的联系的

\*/

printSlices("numbers1:", numbers1)

printSlices("numbers:", numbers)

}

func printSlices(name string, x []int) {

fmt.Print(name, "\t")

fmt.Printf("addr:%p \t len=%d \t cap=%d \t slice=%v\n", x , len(x), cap(x), x)

}

输出结果：

拷贝个数： 6

numbers1: addr:0xc000108000 len=6 cap=40 slice=[2 3 4 5 6 7]

numbers1: addr:0xc000108000 len=6 cap=40 slice=[100 3 4 5 6 7]

numbers: addr:0xc000106000 len=6 cap=20 slice=[2 3 4 5 6 99]

5.2.5 切片高级操作

1.删除切片中的元素。

Go语言中并没有为切片删除元素及提供相应的函数，如果需要实现删除切片中的某个元素，可以利用切片截取及append()函数实现删除元素

删除第一个元素使用numbers = numbers[1:]；

删除最后一个元素使用numbers = numbers[:len(numbers)-1]；

删除任意元素n使用numbers = append(numbers[:n] , numbers[n+1:]...)

例：

package main

import "fmt"

func main() {

fmt.Println("1、------------------")

//var numbers []int

numbers := make([]int , 0 , 20)

printSlices("numbers:", numbers)

numbers = append(numbers, 0) //[0]

printSlices("numbers:", numbers)

/\* 向切片添加一个元素 \*/

numbers = append(numbers, 1) //[0 1]

printSlices("numbers:", numbers)

/\* 同时添加多个元素 \*/

numbers = append(numbers, 2, 3, 4, 5, 6, 7) //[0 1 2 3 4 5 6 7]

printSlices("numbers:", numbers)

fmt.Println("2、------------------")

//追加一个切片

s1 := []int{100, 200, 300, 400, 500, 600, 700}

numbers = append(numbers, s1...)

printSlices("numbers:", numbers)

fmt.Println("3、------------------")

//切片删除元素

//删除第一个元素

numbers = numbers[1:]

printSlices("numbers:", numbers)

//删除最后一个元素

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

printSlices("numbers:", numbers)

//删除中间一个元素

a := int(len(numbers)/2)

fmt.Println("中间数：" , a)

numbers = append(numbers[:a] , numbers[a+1:]...)

printSlices("numbers:", numbers)

}

func printSlices(name string, x []int) {

fmt.Print(name, "\t")

fmt.Printf("addr:%p \t len=%d \t cap=%d \t slice=%v\n", x , len(x), cap(x), x)

}

运行结果：

1、------------------

numbers: addr:0xc000106000 len=0 cap=20 slice=[]

numbers: addr:0xc000106000 len=1 cap=20 slice=[0]

numbers: addr:0xc000106000 len=2 cap=20 slice=[0 1]

numbers: addr:0xc000106000 len=8 cap=20 slice=[0 1 2 3 4 5 6 7]

2、------------------

numbers: addr:0xc000106000 len=15 cap=20 slice=[0 1 2 3 4 5 6 7 100 200 300 400 500 600 700]

3、------------------

numbers: addr:0xc000106008 len=14 cap=19 slice=[1 2 3 4 5 6 7 100 200 300 400 500 600 700]

numbers: addr:0xc000106008 len=13 cap=19 slice=[1 2 3 4 5 6 7 100 200 300 400 500 600]

中间数： 6

numbers: addr:0xc000106008 len=12 cap=19 slice=[1 2 3 4 5 6 100 200 300 400 500 600]

5.3 冒泡排序

5.3.1 概念：

冒泡排序（Bubble Sort），是一种计算机科学领域的较简单的排序算法。

它重复地遍历要排序的数组元素，一次比较两个元素，如果他们的顺序错误就把他们交换过来。重复地进行直到没有再需要交换，也就是说该数组已经排序完成。

这个算法的名字由来是因为越大的元素会经由交换慢慢“浮”到数列的顶端，故名“冒泡排序”。

5.3.2 冒泡排序算法的原理如下：

1.比较相邻的元素。如果第一个比第二个大，就交换他们两个。

2.对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。

3.针对所有的元素重复以上的步骤，除了最后一个。

4.持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

5.3.3 冒泡排序分析：

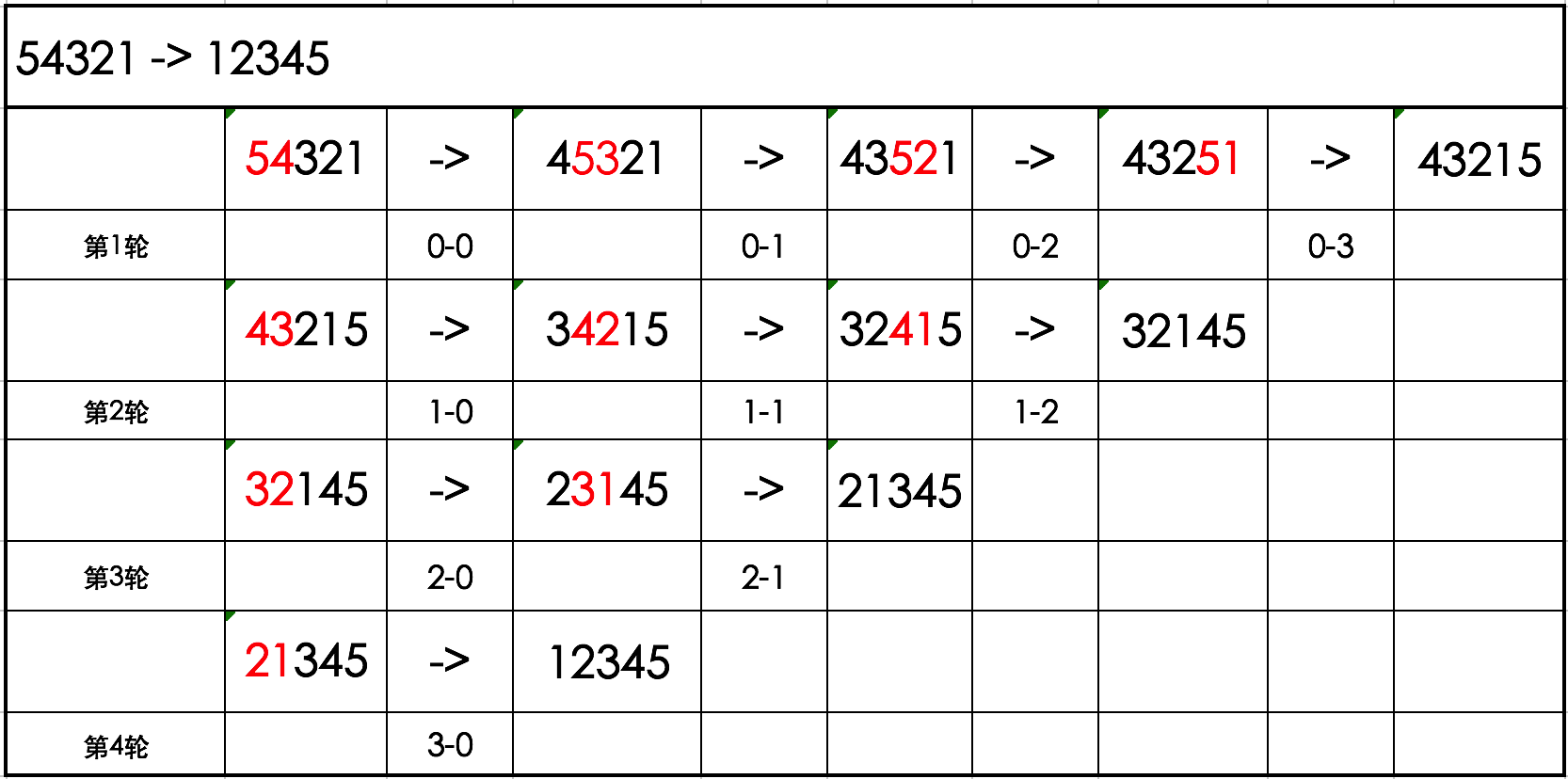


图5.1

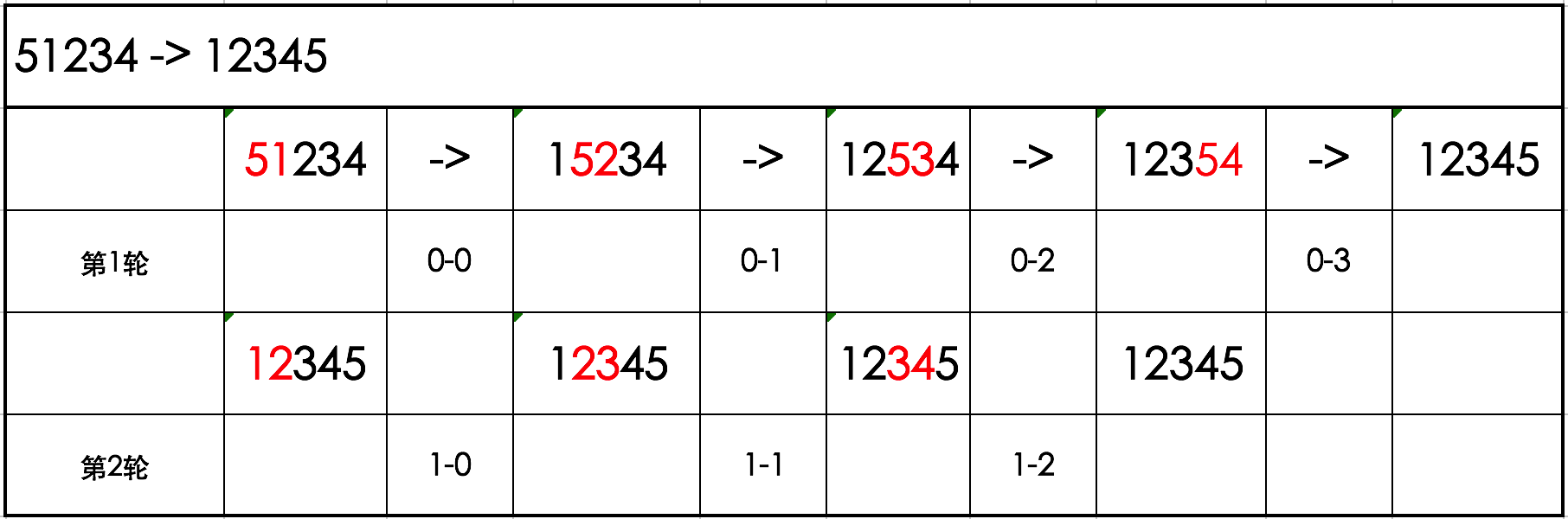


图5.2

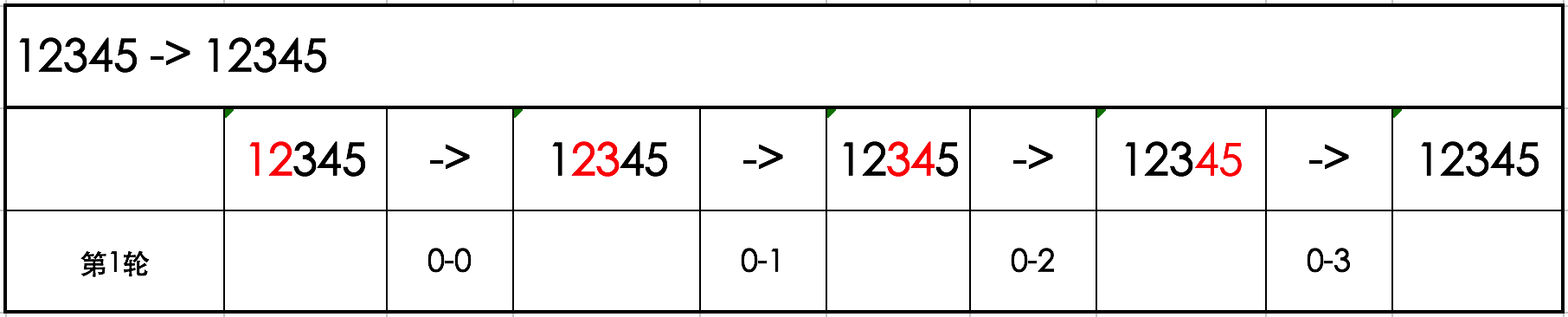


图5.3

5.3.4 冒泡排序核心代码

//从小到大排列

func BubbleSort(arr []int) {

iCount := 0 //记录内循环次数

jCount := 0 //记录外循环的次数

//双层for循环

for i := 0; i < len(arr)-1; i++ {

//定义一个标记，判断本轮是否有两两换位。如果没有换位，那说明排序结束，可以跳出循环。

//例如：12345，当执行第一轮循环后，所有相邻的两个数值都无需换位，那说明排序正常，无需排序。不用执行后续的循环。

flag := true

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

//判断相邻两个数据的大小

if arr[j] > arr[j+1] {

//如果前者比后者大，则执行数据交换

arr[j], arr[j+1] = arr[j+1], arr[j]

//如果出现换位，说明排序还没有结束，需要继续循环执行。

flag = false

}

iCount++

}

jCount++

//如果本轮没有换位，那说明排序结束，则跳出循环

if (flag) {

break

}

}

fmt.Println("i循环次数=", jCount)

fmt.Println("j循环次数=", iCount)

}

5.4 map

5.4.1 map简介

map是Go中的内置类型，它将一个值与一个键关联起来。可以使用相应的键检索值。有资料翻译成地图、映射或字典。但是大多数习惯上翻译成集合。map 是一种无序的键值对（key-value pair ）的集合。map通过 key 来快速检索数据，key 类似于索引，指向相应的value值。

map 是一种集合，所以可以像遍历数组或切片那样去遍历它，不过map 是无序的，所以无法决定它的展示顺序，这是因为 map 是使用 hash 表来实现的。

map是无序的，每次打印出来的map都会不一样，它不能通过index获取，而必须通过key获取；

map的长度是不固定的，和slice一样可以扩展。内置的len()函数同样适用于map，返回map拥有的键值对的数量。但是map不能通过cap()函数计算容量（或者说cap()函数的参数不可以是map）；

同一个map中key必须保证唯一。key的数据类型必须是可参与比较运算的类型，也就是支持==或!=操作的类型。如布尔型、整数型、浮点型、字符串型、数组。对于切片、函数等引用类型则不能作为键的类型；(Invalid map key type: must not be must not be a function , map or slice)。

map的value可以是任何数据类型。和slice一样，map也是一种引用类型；

5.4.2 map的语法

1.使用map关键字定义map

var 变量名 map[key类型]value类型

var声明变量，默认 map 是 nil 。nil map 不能用来存放键值对。var声明后，要么声明时初始化，要么再使用make()函数分配到内存空间，这样才能在其中存放键值对。

例：

声明时初始化数值

var country = map[string]string{

"China": "Beijing",

"France": "Paris",

"Italy": "Rome",

"Japan": "Tokyo",

//"Japan": "Tokyo2",//key名不能重复

}

map1 := map[string]string{

"element": "div",

"width": "100px",

"height": "200px",

"border": "solid",

"color": "red",

"background": "none",

}

rating := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }

2.使用 make 函数

变量名 := make(map[key类型]value类型)

该声明方式，如果不初始化 map，那么map也不等于nil。

例：

定义时初始化

rating := map[string]float32{"C": 5, "Go": 4.5, "Python": 4.5, "C++": 2}

fmt.Println(rating)

创建集合后再赋值

countryCapitalMap := make(map[string]string)

/\* map 插入 key-value 对，各个国家对应的首都 \*/

countryCapitalMap["China"] = "Beijing"

countryCapitalMap["France"] = "Paris"

countryCapitalMap["Italy"] = "Rome"

countryCapitalMap["Japan"] = "Tokyo"

countryCapitalMap["India"] = "New Delhi"

3.map数值遍历

由于map是无序的数据集合，遍历时只能使用迭代遍历的方式，使用for range的方式迭代map集合。

例：

package main

import "fmt"

func main() {

countryCapitalMap := make(map[string]string)

countryCapitalMap["China"] = "Beijing"

countryCapitalMap["France"] = "Paris"

countryCapitalMap["Italy"] = "Rome"

countryCapitalMap["Japan"] = "Tokyo"

countryCapitalMap["India"] = "New Delhi"

for k, v := range countryCapitalMap {

fmt.Println("国家是：", k, "首都：", v)

}

}

输出结果：

国家是： China 首都： Beijing

国家是： France 首都： Paris

国家是： Italy 首都： Rome

国家是： Japan 首都： Tokyo

国家是： India 首都： New Delhi

4.查看元素在集合中是否存在

我们可以通过key获取map中对应的value值。语法为：map[key]

但是当key如果不存在的时候，会得到该value值类型的默认值，比如string类型得到空字符串，int类型得到0。但是程序不会报错。

所以可以通过 value, ok := map[key] ，获取key/value是否存在。ok是bool型，如果 ok 是 true, 则该键值对存在，否则不存在。

例：

captial, ok := countryCapitalMap["United States"]

/\* 如果 ok 是 true, 则存在，否则不存在 \*/

if ok {

fmt.Println("首都是：", captial)

} else {

fmt.Println("该国家的首都没有列出！")

}

5.4.4 delete() 函数

delete(map, key) 函数用于删除集合的某个元素, 参数为 map 和其对应的 key。删除函数不返回任何值。

例：

package main

import "fmt"

func main() {

map1 := map[string]string{

"element": "div",

"width": "100px",

"height": "200px",

"border": "solid",

"color": "red",

"background": "none",

}

fmt.Println("删除前：", map1)

if \_, ok := map1["background"]; ok {

delete(map1, "background")

}

fmt.Println("删除后：", map1)

map1 = make(map[string]string)

fmt.Println("清空后：", map1 , len(map1))

}

输出结果：

删除前： map[background:none border:solid color:red element:div height:200px width:100px]

删除后： map[border:solid color:red element:div height:200px width:100px]

清空后： map[] 0

Go语言没有为map提供任何清空所有元素的函数；清空map的唯一办法是重新make一个新的map；不用担心垃圾回收的效率，Go语言的垃圾回收比写一个清空函数更高效。

与切片相似，map是引用类型。当将map分配给一个新变量时，它们都指向相同的内部数据结构。因此，一个的变化会反映另一个。

例：

package main

import "fmt"

func main() {

personSalary := map[string]int{

"Steven": 18000,

"Daniel": 5000,

"Josh": 20000,

}

fmt.Println("原始薪资：", personSalary)

newPersonSalary := personSalary

newPersonSalary["Daniel"] = 8000

fmt.Println("修改后newPersonSalary：", newPersonSalary)

fmt.Println("personSalary受影响情况：", personSalary)

}

运行结果：

原始薪资： map[Daniel:5000 Josh:20000 Steven:18000]

修改后newPersonSalary： map[Daniel:8000 Josh:20000 Steven:18000]

personSalary受影响情况： map[Daniel:8000 Josh:20000 Steven:18000]

5.5 list

5.5.1 概述

list是一种非连续存储的容器，由多个节点组成，节点通过一些变量记录彼此之间的关系。list有多种实现方法，如单向链表、双向链表等。

假设A、B、C三个都有电话号码，如果A把号码告诉B，B把号码告诉C。这个过程就建立了一个单向链表结构；

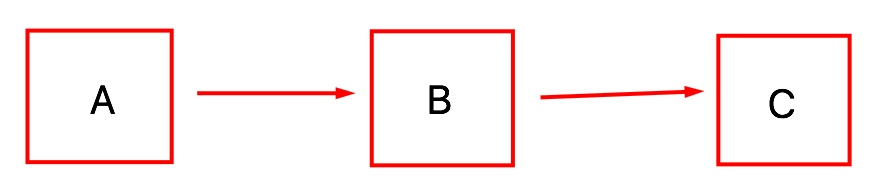


图 5.4

如果在单链表的基础上，再从C开始，将自己的号码给告诉自己号码的人，这样就形成了双向链表结构。



图 5.5

Go语言中list的实现原理是双向链表。list能高效地进行任意位置的元素插入和删除操作。

Golang的标准库提供了高级的数据结构List。具体在包container/list。

container/list包里主要有两个数据结构类型：“Element”、“List”；Element类型代表双向链表中的一个元素，相当于C++里面的"iterator"；List代表一个双向链表。List零值为一个空的、可用的链表。Element有Prev和Next方法用于得到前一个或者下一个Element，Element可以直接调用Value属性；

5.5.2 声明list

list的声明有两种方法：New和List声明。

1.通过container/list包的New方法声明list

变量名 := list.New()

2.通过var声明list

var 变量名 list.List

list与切片和map不同，没有具体元素类型的限制。list中的元素可以是任意类型。在CPP里面，list的成员必须是同一个数据类型，但是Go语言中却允许list中插入任意类型的成员。建议使用New()实现声明list。

5.5.3 element常用方法

1.func (e \*Element) Next() \*Element

Next返回链表的后一个元素或者nil。

2.func (e \*Element) Prev() \*Element

Prev返回链表的前一个元素或者nil。

5.5.4 list常用方法

1.func New() \*List

New创建一个链表。

2.func (l \*List) Init() \*List

Init清空链表。

3.func (l \*List) Len() int

Len返回链表中元素的个数，复杂度O(1)。

4.func (l \*List) Front() \*Element

Front返回链表第一个元素或nil。

5.func (l \*List) Back() \*Element

Back返回链表最后一个元素或nil。

6.func (l \*List) PushFront(v interface{}) \*Element

PushBack将一个值为v的新元素插入链表的第一个位置，返回生成的新元素。

7.func (l \*List) PushFrontList(other \*List)

添加另一个列表到开头。PushFrontList创建链表other的拷贝，并将拷贝的最后一个位置连接到链表l的第一个位置。

8.func (l \*List) PushBack(v interface{}) \*Element

PushBack将一个值为v的新元素插入链表的最后一个位置，返回生成的新元素。

9.func (l \*List) PushBackList(other \*List)

追加另一个列表到末尾。PushBack创建链表other的拷贝，并将链表l的最后一个位置连接到拷贝的第一个位置。

10.func (l \*List) InsertBefore(v interface{}, mark \*Element) \*Element

InsertBefore将一个值为v的新元素插入到mark前面，并返回生成的新元素。如果mark不是l的元素，l不会被修改。

11.func (l \*List) InsertAfter(v interface{}, mark \*Element) \*Element

InsertAfter将一个值为v的新元素插入到mark后面，并返回新生成的元素。如果mark不是l的元素，l不会被修改。

12.func (l \*List) MoveToFront(e \*Element)

MoveToFront将元素e移动到链表的第一个位置，如果e不是l的元素，l不会被修改。

13.func (l \*List) MoveToBack(e \*Element)

MoveToBack将元素e移动到链表的最后一个位置，如果e不是l的元素，l不会被修改。

14.func (l \*List) MoveBefore(e, mark \*Element)

MoveBefore将元素e移动到mark的前面。如果e或mark不是l的元素，或者e==mark，l不会被修改。

15.func (l \*List) MoveAfter(e, mark \*Element)

MoveAfter将元素e移动到mark的后面。如果e或mark不是l的元素，或者e==mark，l不会被修改。

16.func (l \*List) Remove(e \*Element) interface{}

Remove删除链表中的元素e，并返回e.Value。

5.5.5 遍历list

1.顺序遍历

for e := l.Front(); e != nil; e = e.Next() {

fmt.Print(e.Value, " ")

}

2.逆序遍历

for e := l.Back(); e != nil; e = e.Prev() {

fmt.Print(e.Value, " ")

}

5.5.6 list是值类型还是引用类型

1.list本质是什么？

type List struct {

root Element // sentinel list element, only &root, root.prev, and root.next are used

len int // current list length excluding (this) sentinel element

}

type Element struct {

next, prev \*Element

// The list to which this element belongs.

list \*List

// The value stored with this element.

Value interface{}

}

2.struct是值类型

例：

package main

import (

"container/list"

"fmt"

)

func main() {

copyList()

}

//list是值类型，不过采用New()方法声明的是一个指针。所以在拷贝操作和参数传递时具有引用类型的特征。

func copyList() {

//声明list1

list1 := list.New()

printListInfo2("刚声明的list1：", list1)

//给list1赋值

list1.PushBack("one")

list1.PushBack(2)

list1.PushBack("three")

list1.PushFront("first")

printListInfo2("赋值后的list1", list1)

iterateList2(list1)

//将list1拷贝给list2。其实拷贝的是地址

list2 := list1

printListInfo2("刚拷贝的list2", list2)

iterateList2(list2)

//list2修改后

list2.PushBack(250)

list2.PushBack(350)

list2.PushBack(450)

printListInfo2("修改后的list2", list2)

iterateList2(list2)

//list2的修改是否影响到list1？

printListInfo2("修改list2的list1", list1)

iterateList2(list1)

}

func printListInfo2(info string, l \*list.List) {

fmt.Println(info + "----------")

fmt.Printf("%T:%v \t ， 长度为：%d \n", l, l, l.Len())

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

}

func iterateList2(l \*list.List) {

i := 0

for e := l.Front(); e != nil; e = e.Next() {

i++

fmt.Printf("%d:%v \t", i, e.Value)

}

fmt.Println("\n----------")

}

输出结果：

刚声明的list1：----------

\*list.List:&{{0xc00005e330 0xc00005e330 <nil> <nil>} 0} ， 长度为：0

----------

赋值后的list1----------

\*list.List:&{{0xc00005e420 0xc00005e3f0 <nil> <nil>} 4} ， 长度为：4

----------

1:first 2:one 3:2 4:three

----------

刚拷贝的list2----------

\*list.List:&{{0xc00005e420 0xc00005e3f0 <nil> <nil>} 4} ， 长度为：4

----------

1:first 2:one 3:2 4:three

----------

修改后的list2----------

\*list.List:&{{0xc00005e420 0xc00005e510 <nil> <nil>} 7} ， 长度为：7

----------

1:first 2:one 3:2 4:three 5:250 6:350 7:450

----------

修改list2的list1----------

\*list.List:&{{0xc00005e420 0xc00005e510 <nil> <nil>} 7} ， 长度为：7

----------

1:first 2:one 3:2 4:three 5:250 6:350 7:450

----------

Process finished with exit code 0

4.结论：

list是值类型，不过采用list的New()方法声明的是一个指针变量。所以在拷贝操作和参数传递时具有引用类型的特征。

5.6 小结

本章学习Go语言的数组、切片、字典等内置容器，通过内置容器的学习，可以丰富我们的编程思路。同时在未来的开发过程中，只有能够灵活的运用内置容器，才能实现各种各样的功能，容器将是开发过程中必不可少的重要组成部分。尤其切片是重中之重。

第六章 strings包和time及math包

本章重点为大家介绍如下的内容：

* strings包常用函数
* strconv包常用函数比特币系统整体架构
* time包
* math包
* math/rand包——随机数包

6.1 字符串处理包概述

strings包实现了用于操作字符的简单函数。

strconv包实现了字符串与其它基本数据类型之间的类型转换。

大多数语言的字符串处理库提供的函数都大同小异，且越高级的语言提供的函数越多。

掌握基本的字符串处理函数后，更丰富的字符串处理函数都是通过封装基本的处理函数实现。所以熟悉go strings库后基本就能借此封装符合自己需求的，应用于特定场景的字符串处理函数了。

例：

package main

import (

"fmt"

"unicode/utf8"

)

func main() {

s := "我爱Go语言"

fmt.Println(s)

fmt.Println("字节长度：", len(s))

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

//获取字符串引用

for i, ch := range s {

fmt.Printf("%d:%c \n", i, ch)

}

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

//遍历所有字节

for i, ch := range []byte(s) {

fmt.Printf("%d:%X \n", i, ch)

}

fmt.Println("字节长度:", len(s))

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

//获取所有字符

//rune相当于go的char（字符型）

count := 0

for i, ch := range []rune(s) {

fmt.Printf("%d:%c \n", i, ch)

count++

}

fmt.Println("字符长度:", count)

fmt.Println("字符长度：", utf8.RuneCountInString(s))

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

}

6.2 strings包中的字符串处理函数

6.2.1 检索字符串

1.func Contains(s, substr string) bool

判断字符串s是否包含substr字符串

fmt.Println(strings.Contains("seafood", "foo"))//输出：true

fmt.Println(strings.Contains("seafood", "bar"))//输出：false

fmt.Println(strings.Contains("seafood", ""))//输出：true

fmt.Println(strings.Contains("", ""))//输出：true

2.func ContainsAny(s, chars string) bool

判断字符串s是否包含chars字符串中的任一字符

fmt.Println(strings.ContainsAny("team", "i"))//输出：false

fmt.Println(strings.ContainsAny("failure", "u & i"))//输出：true

fmt.Println(strings.ContainsAny("foo", ""))//输出：false

fmt.Println(strings.ContainsAny("", ""))//输出：false

3.func Count(s, sep string) int

返回字符串s包含字符串sep的个数

fmt.Println(strings.Count("cheese", "e"))//输出：3

fmt.Println(strings.Count("five", "")) //输出：5

4.func Index(s, sep string) int

返回字符串s中字符串sep首次出现的位置

fmt.Println(strings.Index("chicken", "ken"))//输出：4

fmt.Println(strings.Index("chicken", "dmr"))//输出：-1

5.func IndexAny(s, chars string) int

返回字符串chars中的任一unicode码值r在s中首次出现的位置

fmt.Println(strings.IndexAny("chicken", "aeiouy"))//输出：2

fmt.Println(strings.IndexAny("crwth", "aeiouy"))//输出：-1

6.func IndexFunc(s string, f func(rune) bool) int

返回字符串s中满足函数f(r)==true字符首次出现的位置

f := func(c rune) bool {

return unicode.Is(unicode.Han, c)

}

fmt.Println(strings.IndexFunc("Hello, 世界", f))//输出：7

fmt.Println(strings.IndexFunc("Hello, world", f))//输出：-1

7.func IndexRune(s string, r rune) int

返回unicode码值r在字符串中首次出现的位置

fmt.Println(strings.IndexRune("chicken", 'k'))//输出：4

fmt.Println(strings.IndexRune("chicken", 'd'))//输出：-1

8.func LastIndex(s, sep string) int

返回字符串s中字符串sep最后一次出现的位置

fmt.Println(strings.Index("go gopher", "go"))//输出：0

fmt.Println(strings.LastIndex("go gopher", "go"))//输出：3

fmt.Println(strings.LastIndex("go gopher", "rodent"))//输出：-1

6.2.2 分割字符串：

1.func Fields(s string) []string

返回将字符串按照空白（unicode.IsSpace确定，可以是一到多个连续的空白字符）分割的多个字符串。如果字符串全部是空白或者是空字符串的话，会返回空切片。

fmt.Printf("Fields are: %q", strings.Fields(" foo bar baz "))//输出：Fields are: ["foo" "bar" "baz"]

2.func FieldsFunc(s string, f func(rune) bool) []string

类似Fields，但使用函数f来确定分割符（满足f的unicode码值）。如果字符串全部是分隔符或者是空字符串的话，会返回空切片。

f := func(c rune) bool {

return !unicode.IsLetter(c) && !unicode.IsNumber(c)

}

fmt.Printf("Fields are: %q", strings.FieldsFunc(" foo1;bar2,baz3...", f))//输出：Fields are: ["foo1" "bar2" "baz3"]

3.func Split(s, sep string) []string

用去掉s中出现的sep的方式进行分割，会分割到结尾，并返回生成的所有片段组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个字符串。

fmt.Printf("%q\n", strings.Split("a,b,c", ","))//输出：["a" "b" "c"]

fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a "))//输出：["" "man " "plan " "canal panama"]

fmt.Printf("%q\n", strings.Split(" xyz ", ""))//输出：[" " "x" "y" "z" " "]

fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))//输出：[""]

4.func SplitAfter(s, sep string) []string

用从s中出现的sep后面切断的方式进行分割，会分割到结尾，并返回生成的所有片段组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个字符串。

fmt.Printf("%q\n", strings.SplitAfter("a,b,c", ","))//输出：["a," "b," "c"]

5.func SplitAfterN(s, sep string, n int) []string

用从s中出现的sep后面切断的方式进行分割，会分割到结尾，并返回生成的所有片段组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个字符串。参数n决定返回的切片的数目：

n > 0 : 返回的切片最多n个子字符串；最后一个子字符串包含未进行切割的部分。

n == 0: 返回nil

n < 0 : 返回所有的子字符串组成的切

fmt.Printf("%q\n", strings.SplitAfterN("a,b,c", ",", 2))//输出：["a," "b,c"]

6.func SplitN(s, sep string, n int) []string

用去掉s中出现的sep的方式进行分割，会分割到结尾，并返回生成的所有片段组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个字符串。参数n决定返回的切片的数目：

n > 0 : 返回的切片最多n个子字符串；最后一个子字符串包含未进行切割的部分。

n == 0: 返回nil

n < 0 : 返回所有的子字符串组成的切片

fmt.Printf("%q\n", strings.SplitN("a,b,c", ",", 2))//输出：["a" "b,c"]

z := strings.SplitN("a,b,c", ",", 0)

fmt.Printf("%q (nil = %v)\n", z, z == nil)//输出：[] (nil = true)

6.2.3 大小写转换：

1.func Title(s string) string

将字符串s每个单词首字母大写返回

fmt.Println(strings.Title("her royal highness"))//输出：Her Royal Highness

2.func ToLower(s string) string

将字符串s转换成小写返回

fmt.Println(strings.ToLower("Gopher"))//输出：gopher

3.func ToTitle(s string) string

将字符串s转换成大写返回

fmt.Println(strings.ToTitle("loud noises"))//输出：LOUD NOISES

fmt.Println(strings.ToTitle("хлеб"))//输出：ХЛЕБ

4.func ToUpper(s string) string

将字符串s转换成大写返回

fmt.Println(strings.ToUpper("Gopher"))//输出：GOPHER

6.2.4 trim修剪函数：

1.func Trim(s string, cutset string) string

将字符串s中首尾包含cutset中的任一字符去掉返回

fmt.Printf("[%q]", strings.Trim(" !!! Achtung! Achtung! !!! ", "! "))//输出：["Achtung! Achtung"]

2.func TrimFunc(s string, f func(rune) bool) string

将字符串s首尾满足函数f(r)==true的字符去掉返回

3.func TrimLeft(s string, cutset string) string

将字符串s左边包含cutset中的任一字符去掉返回

4.func TrimLeftFunc(s string, f func(rune) bool) string

将字符串s左边满足函数f(r)==true的字符去掉返回

5.func TrimPrefix(s, prefix string) string

将字符串s中前缀字符串prefix去掉返回

var s = "Goodbye,, world!"

s = strings.TrimPrefix(s, "Goodbye,")

s = strings.TrimPrefix(s, "Howdy,")

fmt.Print("Hello" + s)//输出：Hello, world!

6.func TrimRight(s string, cutset string) string

将字符串s右边包含cutset中的任一字符去掉返回

7.func TrimRightFunc(s string, f func(rune) bool) string

将字符串s右边满足函数f(r)==true的字符去掉返回

8.func TrimSpace(s string) string

将字符串s首尾空白去掉返回

fmt.Println(strings.TrimSpace(" \t\n a lone gopher \n\t\r\n"))//输出：a lone gopher

9.func TrimSuffix(s, suffix string) string

将字符串s中后缀字符串prefix去掉返回

var s = "Hello, goodbye, etc!"

s = strings.TrimSuffix(s, "goodbye, etc!")

s = strings.TrimSuffix(s, "planet")

fmt.Print(s, "world!")//输出：Hello, world!

6.2.5 比较字符串

1.func Compare(a, b string) int

按字典顺序比较a和b字符串大小

fmt.Println(strings.Compare("a", "b"))//输出：-1

fmt.Println(strings.Compare("a", "a"))//输出：0

fmt.Println(strings.Compare("b", "a"))//输出：1

2.func EqualFold(s, t string) bool

判断s和t两个utf8字符串是否相等，忽略大小写

fmt.Println(strings.EqualFold("Go", "go"))//输出：true

3.func Repeat(s string, count int) string

将字符串s重复count次返回

fmt.Println("ba" + strings.Repeat("na", 2))//输出：banana

4.func Replace(s, old, new string, n int) string

替换字符串s中old字符为new字符并返回，n<0是替换所有old字符串

fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))//输出：oinky oinky oink

fmt.Println(strings.Replace("oink oink oink", "oink", "moo", -1))//输出：moo moo moo

5.func Join(a []string, sep string) string

将a中的所有字符串连接成一个字符串，使用字符串sep作为分隔符

s := []string{"foo", "bar", "baz"}

fmt.Println(strings.Join(s, ", "))//输出：foo, bar, baz

6.3 strconv包中常用函数

6.3.1 Parse类函数（将字符串转其它类型）

1.func Atoi(s string) (int, error)

Atoi 返回 ParseInt(s, 10, 0) 转换为 int 类型的结果。【alphabet：字母】

v := "10"

if s, err := strconv.Atoi(v); err == nil {

fmt.Printf("%T, %v", s, s)//输出：int, 10

}

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

ParseInt 解释给定基础（2到36）中的字符串 s 并返回相应的值 i。如果 base == 0，则基数由字符串的前缀隐含：base 16代表“0x”，base 8代表“0”，否则以10为底数。

v32 := "-354634382"

if s, err := strconv.ParseInt(v32, 10, 32); err == nil {

fmt.Printf("%T, %v\n", s, s)//输出：int64, -354634382

}

if s, err := strconv.ParseInt(v32, 16, 32); err != nil {

//输出：strconv.ParseInt: parsing "-354634382": value out of range

fmt.Println(err.Error())

fmt.Printf("%T, %v\n", s, s)//输出：int64, -2147483648

}

v64 := "-3546343826724305832"

if s, err := strconv.ParseInt(v64, 10, 64); err == nil {

fmt.Printf("%T, %v\n", s, s)//输出：int64, -3546343826724305832

}

if s, err := strconv.ParseInt(v64, 16, 64); err != nil {

//输出：strconv.ParseInt: parsing "-3546343826724305832": value out of range

fmt.Println(err.Error())

fmt.Printf("%T, %v\n", s, s)//输出：int64, -9223372036854775808

}

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

ParseUint 就像 ParseInt，但是对于无符号数字。

v := "42"

if s, err := strconv.ParseUint(v, 10, 32); err == nil {

fmt.Printf("%T, %v\n", s, s)//输出：uint64, 42

}

if s, err := strconv.ParseUint(v, 10, 64); err == nil {

fmt.Printf("%T, %v\n", s, s)//输出：uint64, 42

}

4.func ParseFloat(s string, bitSize int) (float64, error)

ParseFloat 将字符串 s 转换为浮点数，精度由 bitSize：32指定，float32为64; float64为64。当 bitSize = 32时，结果仍然具有 float64 类型，但可以在不更改其值的情况下将其转换为 float32。

var v = "3.1415926535"

if s, err := strconv.ParseFloat(v, 32); err == nil {

fmt.Printf("%T, %v\n", s, s)//输出：float64, 3.1415927410125732

}

if s, err := strconv.ParseFloat(v, 64); err == nil {

fmt.Printf("%T, %v\n", s, s)//输出：float64, 3.1415926535

}

5.func ParseBool(str string) (bool, error)

ParseBool 返回字符串表示的布尔值。

它接受1，t，T，TRUE，true，True，

0，f，F，FALSE，false，False。

任何其他值都会返回错误。

v := "true"

if s, err := strconv.ParseBool(v); err == nil {

fmt.Printf("%T, %v\n", s, s)//输出：bool, true

}

6.3.2 Format类函数

1.func Itoa(i int) string

Itoa 是 FormatInt(int64(i), 10) 的缩写。

i := 10

s := strconv.Itoa(i)

fmt.Printf("%T, %v\n", s, s)//输出：string, 10

2.func FormatInt(i int64, base int) string

FormatInt 返回给定基数中的i的字符串表示，对于2 <= base <= 36.结果对于数字值> = 10使用小写字母 'a' 到 'z' 。

v := int64(-42)

s10 := strconv.FormatInt(v, 10)

fmt.Printf("%T, %v\n", s10, s10)//输出：string, -42

s16 := strconv.FormatInt(v, 16)

fmt.Printf("%T, %v\n", s16, s16)//输出：string, -2a

3.func FormatUint(i uint64, base int) string

FormatUint 返回给定基数中的 i 的字符串表示，对于2 <= base <= 36.结果对于数字值> = 10使用小写字母 'a' 到 'z' 。

v := uint64(42)

s10 := strconv.FormatUint(v, 10)

fmt.Printf("%T, %v\n", s10, s10)//输出：string, 42

s16 := strconv.FormatUint(v, 16)

fmt.Printf("%T, %v\n", s16, s16)//输出：string, 2a

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

FormatFloat 根据格式 fmt 和 precision prec 将浮点数f转换为字符串。它将结果进行四舍五入，假设原始数据是从 bitSize 位的浮点值获得的（float32为32，float64为64）。

格式 fmt 是 'b'，'e'，'E'，'f'，'g'或 'G'。

v := 3.1415926535

s32 := strconv.FormatFloat(v, 'E', -1, 32)

fmt.Printf("%T, %v\n", s32, s32)//输出：string, 3.1415927E+00

s64 := strconv.FormatFloat(v, 'E', -1, 64)

fmt.Printf("%T, %v\n", s64, s64)//输出：string, 3.1415926535E+00

5.func FormatBool(b bool) string

FormatBool 根据 b 的值返回“true”或“false”

v := true

s := strconv.FormatBool(v)

fmt.Printf("%T, %v\n", s, s)//输出：string, true

6.4 time包

需要先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。

6.4.1 time包中核心方法介绍

1.func Now() Time

Now返回当前本地时间。

2.func (t Time) Local() Time

Local将时间转成本地时区，但指向同一时间点的Time。

3.func (t Time) UTC() Time

UTC将时间转成UTC和零时区，但指向同一时间点的Time。

国际上通过英国伦敦格林尼治天文台原址的那条经线称为0°经线，也叫本初子午线。

4.func Date(year int, month Month, day, hour, min, sec, nsec int, loc \*Location) Time

Date可以根据指定数值，返回一个时间。时区为loc，时间格式为：year-month-day hour:min:sec + nsec nanoseconds的时间点。loc可以是time.Local.time.UTC。string转time

5.func Parse(layout, value string) (Time, error)

Parse能将一个格式化的时间字符串解析成它所代表的时间。就是string转time

layout定义了参考时间：Mon Jan 2 15:04:05 -0700 MST 2006

如果缺少表示时区的信息，Parse会将时区设置为UTC。layout简写格式：Mon Jan 2 15:04:05 2006

预定义的ANSIC.UnixDate.RFC3339和其他版式描述了参考时间的标准或便捷表示。

6.func (t Time) Format(layout string) string

Format根据layout指定的格式返回t代表的时间点的格式化文本表示。就是time转string

layout定义了参考时间：Mon Jan 2 15:04:05 -0700 MST 2006

7.func (t Time) String() string

String将时间格式化成字符串(time转string，相当于是固定格式的Format方法)，格式为："2006-01-02 15:04:05.999999999 -0700 MST"

8.func (t Time) Unix() int64

Unix将t表示为Unix时间（时间戳，int64），即从时间点January 1, 1970 UTC到时间点t所经过的时间（单位秒）。

9.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）。

10.func (t Time) Equal(u Time) bool

判断两个时间是否相同，会考虑时区的影响，因此不同时区标准的时间也可以正确比较。本方法和用t==u不同，这种方法还会比较地点和时区信息。

11.func (t Time) Before(u Time) bool

如果t代表的时间点在u之前，返回真；否则返回假。

12.func (t Time) After(u Time) bool

如果t代表的时间点在u之后，返回真；否则返回假。

13.func (t Time) Date() (year int, month Month, day int)

返回时间点t对应的年.月.日。

14.func (t Time) Year() int

返回时间点t对应的年份。

15.func (t Time) Month() Month

返回时间点t对应那一年的第几月。

16.func (t Time) Day() int

返回时间点t对应那一月的第几日。

17.func (t Time) Weekday() Weekday

返回时间点t对应的那一周的周几。

18.func (t Time) Clock() (hour, min, sec int)

返回t对应的那一天的时.分.秒。

19.func (t Time) Hour() int

返回t对应的那一天的第几小时，范围[0, 23]。

20.func (t Time) Minute() int

返回t对应的那一小时的第几分种，范围[0, 59]。

21.func (t Time) Second() int

返回t对应的那一分钟的第几秒，范围[0, 59]。

22.func (t Time) Nanosecond() int

返回t对应的那一秒内的纳秒偏移量，范围[0, 999999999]。

23.func (t Time) Sub(u Time) Duration

返回一个时间段t-u。如果结果超出了Duration可以表示的最大值/最小值，将返回最大值/最小值。要获取时间点t-d（d为Duration），可以使用t.Add(-d)。

24.func (d Duration) Hours() float64

Hours将时间段表示为float64类型的小时数。

25.func (d Duration) Minutes() float64

Minutes将时间段表示为float64类型的分钟数。

26.func (d Duration) Seconds() float64

Seconds将时间段表示为float64类型的秒数。

27.func (d Duration) Nanoseconds() int64

Nanoseconds将时间段表示为int64类型的纳秒数，等价于int64(d)。

28.func (d Duration) String() string

返回时间段采用"72h3m0.5s"格式的字符串表示。最前面可以有符号，数字+单位为一个单元，开始部分的0值单元会被省略；如果时间段<1s，会使用"ms"."us"."ns"来保证第一个单元的数字不是0；如果时间段为0，会返回"0"。

29.func ParseDuration(s string) (Duration, error)

ParseDuration解析一个时间段字符串。一个时间段字符串是一个序列，每个片段包含可选的正负号.十进制数.可选的小数部分和单位后缀，如"300ms"."-1.5h"."2h45m"。合法的单位有"ns"."us" /"µs"."ms"."s"."m"."h"。

30.func (t Time) Add(d Duration) Time

Add返回时间点t+d。

31.func (t Time) AddDate(years int, months int, days int) Time

AddDate返回增加了给出的年份.月份和天数的时间点Time。例如，时间点January 1, 2011调用AddDate(-1, 2, 3)会返回March 4, 2010。

AddDate会将结果规范化，类似Date函数的做法。因此，举个例子，给时间点October 31添加一个月，会生成时间点December 1。（从时间点November 31规范化而来）

6.5 math包

使用时需要import "math"，math包提供了基本的数学常数和数学函数。

6.5.1 math包中核心函数介绍

1.func IsNaN(f float64) (is bool)

报告f是否表示一个NaN（Not A Number）值。

2.func Ceil(x float64) float64

返回不小于x的最小整数（的浮点值）

3.func Floor(x float64) float64

返回不大于x的最小整数（的浮点值）

4.func Trunc(x float64) float64

返回x的整数部分（的浮点值）。

5.func Abs(x float64) float64

返回x的绝对值

6.func Max(x, y float64) float64

返回x和y中最大值

7.func Min(x, y float64) float64

返回x和y中最小值

8.func Dim(x, y float64) float64

函数返回x-y和0中的最大值

9.func Mod(x, y float64) float64

取余运算，可以理解为 x-Trunc(x/y)\*y，结果的正负号和x相同

10.func Sqrt(x float64) float64

返回x的二次方根

11.func Cbrt(x float64) float64

返回x的三次方根，特例如下：

12.func Hypot(p, q float64) float64

返回Sqrt(p\*p + q\*q)

13.func Pow(x, y float64) float64

返回x^y

14.func Sin(x float64) float64

求正弦。

15.func Cos(x float64) float64

求余弦。

16.func Tan(x float64) float64

求正切。

17.func Log(x float64) float64

求自然对数

18.func Log2(x float64) float64

求2为底的对数。

19.func Log10(x float64) float64

求10为底的对数。

6.6 随机数(math/rand包)

使用时需要import "math/rand"，rand包实现了伪随机数生成器。随机数从资源生成。包水平的函数都使用的默认的公共资源。该资源会在程序每次运行时都产生确定的序列。如果需要每次运行产生不同的序列，应使用Seed函数进行初始化。默认资源可以安全的用于多协程并发。

6.6.1 rand包中核心方法介绍

1.func NewSource(seed int64) Source

使用给定的种子创建一个伪随机资源。

2.func New(src Source) \*Rand

返回一个使用src生产的随机数来生成其他各种分布的随机数值的\*Rand。

3.func (r \*Rand) Seed(seed int64)

使用给定的seed来初始化生成器到一个确定的状态。

4.func (r \*Rand) Int() int

返回一个非负的伪随机int值。

5.func (r \*Rand) Intn(n int) int

返回一个取值范围在[0,n)的伪随机int值，如果n<=0会panic。

6.func (r \*Rand) Float64() float64

返回一个取值范围在[0.0, 1.0]的伪随机float64值。

获取随机数的几种方式：

1.通过默认随机数种子获取随机数

rand.Int()

rand.Float64()

rand.Intn(n) // 例如获取0-n之间随机数

总是生成固定的随机数。默认情况下，随机数种子都是1。seed是一个64位整数。

2.动态随机数种子生成随机资源，实例随机对象来获取随机数

s1 := rand.NewSource(time.Now().UnixNano())

r1 := rand.New(s1)

randnum := r1.Intn(n) // 例如获取0-n之间随机数

3.简写形式：动态变化随机数种子来获取随机数

（1）.获取整型随机数[0,10]

rand.Seed(time.Now().UnixNano()）

rand.Intn(10)

（2）. 获取浮点型0.0至1.0之间的随机数

rand.Seed(time.Now().UnixNano())

rand.Float64()

（3）. 获取两数之间随机数[m , n]

rand.Seed(time.Now().UnixNano())

随机数 = rand.Intn(n - m + 1) + m

例如：获取[5,11]之间随机数： rand.Intn(7) + 5

6.7 键盘输入

6.7.1 scanln

1.fmt.scanln()

例：

package main

import "fmt"

func main() {

username := ""

age := 0

fmt.Scanln(&username, &age)

fmt.Println("账号信息为：", username, age)

fmt.Printf("用户名是：%q ， 年龄是：%d \n", username, age)

fmt.Printf("用户名是：%s ， 年龄是：%d \n", username, age)

fmt.Println(&username)

}

6.7.2 随机数+键盘输入案例——猜数字游戏

package main

import (

"fmt"

"math/rand"

"strings"

"time"

)

func main() {

play()

}

func play() {

target := generateRandNum(10, 100)

//fmt.Println("产生随机数：", target)

fmt.Println("请输入随机数：")

fmt.Println(strings.Repeat("-" , target))

//记录猜测的次数

count := 0

for {

count++

yourNum := 0

fmt.Scanln(&yourNum)

//fmt.Scanf("%d", &yourNum)

if yourNum < target {

fmt.Println("小了")

} else if yourNum > target {

fmt.Println("大了")

} else {

fmt.Println("正确")

fmt.Printf("您一共猜测了 %d 次！\n", count)

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

play()

}

alertInfo(count, target)

}

}

//生成随机数

func generateRandNum(min int, max int) int {

rand.Seed(time.Now().UnixNano())

return rand.Intn(max-min+1) + min

}

//提示信息

func alertInfo(count int, target int) {

if count >= 6 {

fmt.Printf("您一共猜测了 %d 次都没有猜中，太笨了！😓\n", count)

fmt.Println("正确结果是：", target)

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

fmt.Println("")

play()

}

}

6.8 小结

本章学习Go语言的内置函数库的使用，由于篇幅所限，不能将所有的函数都详细讲解，读者如果需要了解其它函数的细节，请参阅官方文档，文档中有详细的函数使用说明。在软件开发过程中，我们也离不开系统提供的各种函数，充分了解函数库里提供了哪些函数，有助于减少自己做无用功，尽可能的使用系统提供的函数来完成实际的功能开发。

第七章 Go语言面向对象编程

本章重点为大家介绍如下的内容：

* 面向对象编程思想
* struct结构体
* 方法
* 什么是接口
* 接口的语法
* duck typing鸭子模型
* 接口实现多态
* 空接口
* 接口对象转型

7.1 面向对象思想的概述

7.1.1 OOP概述

面向对象程序设计以程序业务中涉及的对象为出发点，分析每个对象的属性及方法，通过抽象的方式，将程序本身进行分解整理，对抽象的结果进行封装，通过继承及多态的特性实现，简化程序开发的过程，同时保持程序本身的健壮性及可扩展性。

面向对象程序设计代码管理方便，易于模块化开发。类中包含方法和属性，属性用来记录数据，方法表示行为，类实例化后为对象；函数也表示行为，但是与函数配合的数据却散落摆放，缺乏类这样的结构来统一管理。对象既表示行为，又记录数据，行为和数据由对象来统一管理。写出来的代码方法与属性各归各类，代码逻辑清晰，阅读方便，方便也管理，利于扩展，易于模块化开发。

面向对象程序设计代码重用性高。面向对象的代码在使用时，通过调用各个对象中的方法和属性，不同的排列组合就能适应各种不同的业务场景。代码冗余量小，重用性高。

面向对象四大特性：

* 抽像
* 封装
* 继承
* 多态

7.1.2 Go语言面向对象

Go并不是一个纯面向对象的编程语言。在Go的面向对象中，以结构体struct为主体，方法method可以在结构体上添加，提供了捆绑数据和方法的行为。

Go语言设计的非常简洁优雅，Go没有沿袭传统面向对象编程中的诸多概念，比如继承、虚方法、构造方法和析构方法、this等。Go不支持继承，尽管匿名字段的内存和行为类似继承，但是它不是继承。Go语言没有继承和多态。但是通过别的方式可以实现类似的效果。继承：通过匿名字段实现；多态：通过接口实现。Go语言中学习面向对象，主要学习结构体struct、方法method、接口interface。

7.2 结构体

7.2.1 定义结构体

Go 语言中数组可以存储同一类型的数据，但在结构体中我们可以为不同项定义不同的数据类型。结构体是由一系列具有相同类型或不同类型的数据构成的数据集合。

结构体的定义格式

type 类型名 struct {

成员属性1 类型1

成员属性2 类型2

成员属性3 , 成员属性4 类型3

...

}

类型名用于标识结构体的名称，在同一个包内不能重复。结构体中属性，也叫字段必须唯一。同类型的成员属性可以写在一行。

例：

//定义一个结构体

type Teacher struct {

name string

age int8

sex byte

}

7.2.2 实例化结构体——为结构体分配内存并初始化

实例化就是根据结构体定义的格式创建一份与格式一致的内存区域。结构体实例之间的内存是完全独立的。结构体的定义只是一种内存布局的描述，只有当结构体实例化时，才会真正分配内存。因此必须在定义结构体并实例化后才能使用结构体；

例：

var声明方式实例化结构体，初始化方式为：对象.属性=值

var p1 Teacher

p1.name = "Steven"

p1.age = 35

p1.sex = 1

例：

变量简短声明格式实例化结构体，初始化方式为：对象.属性=值

p2 := Teacher{}

p2.name = "David"

p2.age = 33

p2.sex = 1

例：

变量简短声明格式实例化结构体，声明时初始化。初始化方式为：属性:值 。属性:值可以同行，也可以换行。（类似map的用法）

p3 := Teacher{

name: "Josh",

age: 28,

sex: 1,

}

或者：p3 = Teacher{name: "Josh2", age: 28, sex: 1}

例：

变量简短声明格式实例化结构体，声明时初始化，不写属性名，按属性顺序只写属性值

p4 := Teacher{"Ruby", 30, 0}

例：创建指针类型的结构体

使用内置函数new()对结构体进行实例化，结构体实例化后形成指针类型的结构体。new内置函数会分配内存。第一个参数是类型，而不是值，返回的值是指向该类型新分配的零值的指针。该函数用于创建某个类型的指针。

p5 := new(Teacher)

(\*p5).name = "Running"

(\*p5).age = 31

p5.sex = 0 //语法简写形式，语法糖

7.2.3 结构体中的语法糖

语法糖（Syntactic sugar），也译为糖衣语法，是由英国计算机科学家彼得·约翰·兰达（Peter J. Landin）发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。通常来说使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会。结构体和数组中都含有语法糖。

例：

package main

import (

"fmt"

)

//定义一个结构体

type Emp struct {

name string

age int8

sex byte

}

func main() {

//new内置函数声明结构体。

emp1 := new(Emp)

fmt.Printf("emp1: %T，%v , %p \n", emp1, emp1, emp1)

(\*emp1).name = "David"

(\*emp1).age = 30

(\*emp1).sex = 1

//语法简写形式，语法糖

emp1.name = "Steven"

emp1.age = 35

emp1.sex = 1

fmt.Println(emp1)

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

SyntacticSugar()

}

//数组中语法糖

func SyntacticSugar() {

arr := [4]int{1, 2, 3, 4}

arr2 := &arr

fmt.Println((\*arr2)[3])

fmt.Println(arr2[3])

//切片中有语法糖吗？

arr3 := []int{10, 20, 30, 40}

arr4 := &arr3

fmt.Println((\*arr4)[3])

//fmt.Println(arr4[3])

}

7.2.4 结构体是值类型

package main

import (

"fmt"

)

type Human struct {

name string

age int8

sex byte

}

func main() {

//初始化Human

h1 := Human{"Steven", 35, 1}

fmt.Printf("h1：%T , %v , %p \n", h1, h1, &h1)

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

//将结构体对象进行拷贝

h2 := h1

h2.name = "David"

h2.age = 30

fmt.Printf("h2修改后=%T , %v , %p \n", h2, h2, &h2)

fmt.Printf("h1：%T , %v , %p \n", h1, h1, &h1)

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

//将结构体对象作为参数传递

changeName(h1)

fmt.Printf("h1：%T , %v , %p \n", h1, h1, &h1)

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

//changeName2(&h1)

//fmt.Printf("h1：%T , %v , %p \n", h1, h1, &h1)

}

//传对象

func changeName(h Human) {

h.name = "Daniel"

h.age = 13

fmt.Printf("函数内h修改后=%T , %v , %p \n", h, h, &h)

}

7.2.5 结构体的深拷贝和浅拷贝

值类型是深拷贝，引用类型是浅拷贝。

例：

package main

import (

"fmt"

)

type Dog struct {

name string

color string

age int8

kind string //品种

}

func main() {

//实现结构体的深拷贝

//struct的数据类型：值类型，所以默认的复制就是深拷贝

d1 := Dog{"豆豆", "黑色", 2, "二哈"} //Dog

fmt.Printf("d1：%T , %v , %p \n", d1, d1, &d1)

d2 := d1 //深拷贝 dog

fmt.Printf("d2：%T , %v , %p \n", d2, d2, &d2)

//修改d2，d1是否也发生变化？

d2.name = "毛毛"

fmt.Println("d2修改后=", d2)

fmt.Println("d1=", d1)

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

//实现结构体的浅拷贝：直接拷贝指针地址实现浅拷贝

d3 := &d1

fmt.Printf("d3：%T , %v , %p \n", d3, d3, d3)

d3.kind = "萨摩耶"

d3.color = "白色"

d3.name = "球球"

fmt.Println("d3修改后=", d3)

fmt.Println("d1=", d1)

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

//实现结构体的浅拷贝

//拷贝通过new函数实例化的对象

d4 := new(Dog) //\*Dog

d4.name = "多多"

d4.color = "棕色"

d4.age = 1

d4.kind = "巴哥犬"

d5 := d4 //\*Dog

fmt.Printf("d4：%T , %v , %p \n", d4, d4, d4)

fmt.Printf("d5：%T , %v , %p \n", d5, d5, d5)

//修改d2，d1是否也发生变化？

d5.color = "金色"

d5.kind = "金毛"

fmt.Println("d5修改后=", d5)

fmt.Println("d4=", d4)

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

}

7.2.6 结构体对象或指针作为函数的参数及函数返回值

例：

package main

import (

"fmt"

)

type Flower struct {

name string

color string

}

func main() {

//测试结构体作为参数

f1 := Flower{"玫瑰", "红"}

fmt.Printf("f1：%T , %v , %p \n", f1, f1, &f1)

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

//将结构体对象作为参数传递

changeInfo1(f1)

fmt.Printf("f1：%T , %v , %p \n", f1, f1, &f1)

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

//将结构体指针作为参数传递

changeInfo2(&f1)

fmt.Printf("f1：%T , %v , %p \n", f1, f1, &f1)

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

//测试结构体作为返回值

//结构体对象作为返回值

f2 := getFlower1()

f3 := getFlower1()

fmt.Println(f2, f3)

f2.name = "杏花"

fmt.Println(f2, f3)

fmt.Printf("f2：%T , %v , %p \n", f2, f2, &f2)

fmt.Printf("f3：%T , %v , %p \n", f3, f3, &f3)

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

//结构体指针作为返回值

f4 := getFlower2()

f5 := getFlower2()

fmt.Println(f4, f5)

f4.name = "桃花"

fmt.Println(f4, f5)

fmt.Printf("f4：%T , %v , %p \n", f4, f4, f4)

fmt.Printf("f5：%T , %v , %p \n", f5, f5, f5)

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

}

//传结构体对象

func changeInfo1(f Flower) {

f.name = "月季"

f.color = "粉"

fmt.Printf("函数内f修改后=%T , %v , %p \n", f, f, &f)

}

//传对象指针

func changeInfo2(f \*Flower) {

f.name = "蔷薇"

f.color = "紫"

fmt.Printf("函数内f修改后=%T , %v , %p \n", f, f, f)

}

//返回结构体对象

func getFlower1() (f Flower) {

f = Flower{"牡丹", "白"}

return

}

//返回结构体指针

func getFlower2() (f \*Flower) {

f = &Flower{"芙蓉", "红"}

return

}

7.2.7 匿名结构体和匿名字段

匿名结构体是指没有名字的结构体。无需通过type关键字定义就可以直接使用。创建匿名结构体时，同时要创建对象。名结构体由结构体定义和键值对初始化两部分组成。

语法格式：

变量名 := struct {

//定义成员属性

} {//初始化成员属性}

例：

package main

import (

"fmt"

"math"

)

func main() {

//匿名函数

res := func(a, b float64) float64 {

return math.Pow(a, b)

}(2, 3)

fmt.Println(res)

// 匿名结构体

addr := struct {

province,city string

}{"陕西省" , "西安市"}

fmt.Println(addr)

cat := struct {

name,color string

age int8

} {

name:"绒毛",

color:"黑白",

age:1,

}

fmt.Println(cat)

}

结构体的匿名字段没有名字，只包含一个没有字段名的类型。这些字段被称为匿名字段。如果字段没有名字，那么默认使用类型作为字段名。注意：同一个类型只能写一个。结构体嵌套中采用匿名结构体字段可以模拟继承关系。

例：

type User struct {

//name string

//sex byte

//age int8

//height float64

string

byte

int8

float64

}

func main() {

//实例化结构体

user := User{"Steven", 'm',35, 177.5}

fmt.Println(user)

//如果想依次获得姓名、年龄、身高可以写成：

fmt.Printf("姓名：%s , 性别：%c , 身高：%.2f ， 年龄：%d \n", user.string, user.byte , user.float64, user.int8)

}

7.2.8 结构体嵌套

将一个结构体作为另一个结构体的属性（字段），这种结构就是结构体嵌套。结构体嵌套可以模拟面向对象中的两种关系：

聚合关系：一个类作为另一个类的属性。

继承关系：一个类作为另一个类的子类。子类和父类。

例：

//结构体嵌套模拟聚合关系

package main

import (

"fmt"

)

type Address struct {

province, city string

}

type Person struct {

name string

age int

address Address

}

func main() {

//模拟对象之间的聚合关系

p := Person{}

p.name = "Steven"

p.age = 35

//赋值方式1

addr := Address{}

addr.province = "北京市"

addr.city = "海淀区"

p.address = addr

fmt.Println(p)

fmt.Println("姓名:", p.name)

fmt.Println("年龄:", p.age)

fmt.Println("省:", p.address.province)

fmt.Println("市:", p.address.city)

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

//修改Person对象的数据，是否影响Address对象？

p.address.city = "昌平区"

fmt.Println("姓名:", p.name)

fmt.Println("年龄:", p.age)

fmt.Println("省:", p.address.province)

fmt.Println("市:", p.address.city)

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

fmt.Println("市:", addr.city) //没有影响

//修改Address对象的数据，是否影响Person对象？

addr.city = "大兴区"

fmt.Println("姓名:", p.name)

fmt.Println("年龄:", p.age)

fmt.Println("省:", p.address.province)

fmt.Println("市:", p.address.city)

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

//赋值方式2：

p.address = Address{

province: "陕西省",

city: "西安市",

}

fmt.Println(p)

fmt.Println("姓名:", p.name)

fmt.Println("年龄:", p.age)

fmt.Println("省:", p.address.province)

fmt.Println("市:", p.address.city)

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

}

结构体嵌套可以模拟继承关系。继承是传统面向对象编程中三大特征之一。用于描述两个类之间的关系。一个类（子类、派生类）继承于另一个类（父类、超类）。子类可以有自己的属性和方法，也可以重写父类已有的方法。子类可以直接访问父类所有的属性和方法。

在结构体中属于匿名结构体的字段称为提升字段，因为它们可以被访问，就好像它们属于拥有匿名结构字段的结构一样。换句话说，父类中的字段就是提升字段。

继承的意义：避免重复代码，扩展类的功能。采用匿名字段的形式就是模拟继承关系。而模拟聚合关系时一定要采用有名字的结构体作为字段。

例：

package main

import (

"fmt"

)

type Person struct {

Name string

Age int

Sex string

}

type Student struct {

Person //采用匿名结构体字段模拟继承关系

SchoolName string

}

func main() {

//初始化Person

p1 := Person{"Steven", 35, "男"}

fmt.Println(p1)

fmt.Printf("p1: %T , %+v \n", p1, p1)

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

//初始化Student

//写法1：

s1 := Student{p1, "北航软件学院"}

fmt.Println(s1)

fmt.Printf("s1: %T , %+v \n", s1, s1)

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

//写法2：

s2 := Student{Person{"Josh", 30, "男"}, "北外高翻学院"}

fmt.Println(s2)

fmt.Printf("s2: %T , %+v \n", s2, s2)

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

//写法3：

s3 := Student{Person: Person{

Name: "Penn",

Age: 19,

Sex: "男",

},

SchoolName: "北大元培学院",

}

fmt.Println(s3)

fmt.Printf("s3: %T , %+v \n", s3, s3)

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

//写法4：

s4 := Student{}

s4.Name = "Daniel"

s4.Sex = "男"

s4.Age = 12

s4.SchoolName = "十一龙樾"

fmt.Println(s4)

fmt.Printf("s4: %T , %+v \n", s4, s4)

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

}

结构体嵌套时，可能拥有相同的成员名，成员重名会发生什么？

例：

package main

import "fmt"

type A struct {

a, b int

}

type B struct {

a, d int

}

type C struct {

A

B

}

func main() {

c := C{}

c.A.a = 1

c.B.a = 2 //如果调用c.a = 2，则会提示“引起歧义的参数。”

c.b = 3

c.d = 4

fmt.Println(c)

}

当重名时，编译器会报错：Ambiguous reference。

7.3 方法

7.3.1 什么是方法？

Go语言同时有函数和方法，方法的本质是函数，但是方法和函数又具有不同点。

函数function是一段具有独立功能的代码，可以被反复多次调用，从而实现代码复用。而方法method是一个类的行为功能，只有该类的对象才能调用。

Go语言的方法method是一种作用于特定类型变量的函数。这种特定类型变量叫做Receiver（接受者、接收者、接收器）。接受者的概念类似于传统面向对象语言中的this或self关键字。Go语言的接受者强调了方法具有作用对象，而函数没有作用对象。一个方法就是一个包含了接受者的函数。Go语言中，接受者的类型可以是任何类型，不仅仅是结构体，也可以是struct类型外的其他任何类型。只要接受者不同，则方法名可以一样。

7.3.2 方法的语法格式：

func （接受者变量 接受者类型） 方法名(参数列表) （返回值列表） {

//方法体

}

接受者在func关键字和方法名之间编写，接受者可以是struct类型或非struct类型，可以是指针类型和非指针类型。接受者中的变量在命名时，官方建议使用接受者类型的第一个小写字母。

例：

package main

import "fmt"

type Employee struct {

name, currency string

salary int

}

func main() {

emp1 := Employee{

name: "Daniel Wang",

salary: 2000,

currency: "$",

}

//调用方法

emp1.displaySalary()

//调用函数

displaySalary(emp1)

}

//displaySalary() 方法，接受者类型为Employee

func (e Employee) displaySalary() {

fmt.Printf("员工姓名：%s ，薪资： %s%d \n", e.name, e.currency, e.salary)

}

//displaySalary() 函数，参数为Employee类型对象

func displaySalary(e Employee) {

fmt.Printf("员工姓名：%s ，薪资： %s%d \n", e.name, e.currency, e.salary)

}

7.3.3 方法和函数

既然可以用函数来写相同的程序，为什么还要使用方法？

Go不是一种纯粹面向对象的编程语言，它不支持类。因此其方法是一种实现类似于类的行为的方法。相同名称的方法可以在不同的类型上定义，而具有相同名称的函数是不允许的。假设我们有一个正方形和圆形的结构。可以在正方形和圆形上定义一个名为Area的求取面积的方法。

虽然method的名字一模一样，但是如果接受者不一样，那么method就不一样。method里面可以访问接受者的字段，调用method通过“.”访问，就像struct里面访问字段一样

例：

package main

import (

"fmt"

"math"

)

type Rectangle struct {

width, height float64

}

type Circle struct {

radius float64

}

func main() {

r1 := Rectangle{10, 4}

r2 := Rectangle{12, 5}

c1 := Circle{1}

c2 := Circle{10}

fmt.Println("r1的面积：", r1.area())

fmt.Println("r2的面积：", r2.area())

fmt.Println("c1的面积：", c1.area())

fmt.Println("c2的面积：", c2.area())

}

//该 method 属于 Rectangle类型的对象

func (r Rectangle) area() float64 {

return r.width \* r.height

}

//该 method 属于 Circle 类型的对象

func (c Circle) area() float64 {

return c.radius \* c.radius \* math.Pi

}

运行结果

r1的面积： 40

r2的面积： 60

c1的面积： 3.141592653589793

c2的面积： 314.1592653589793

7.3.4 指针作为接受者

若接受者不是指针，实际只是获取了一个copy，而不能真正改变接受者中原来的数据。

例：

package main

import (

"fmt"

)

type Rectangle struct {

width, height float64

}

func main() {

r1 := Rectangle{5, 8}

r2 := r1

//打印内存地址

fmt.Printf("r1的地址：%p \n", &r1)

fmt.Printf("r2的地址：%p \n", &r2)

r1.setVal()

fmt.Println("r1.height=", r1.height)

fmt.Println("r2.height=", r2.height)

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

r1.setVal2()

fmt.Println("r1.height=", r1.height)

fmt.Println("r2.height=", r2.height)

}

func (r Rectangle) setVal() {

fmt.Printf("setVal()方法中r的地址：%p \n", &r)

r.height = 10

}

func (r \*Rectangle) setVal2() {

fmt.Printf("setVal2()方法中r的地址：%p \n", r)

r.height = 20

}

运行结果：

r1的地址：0xc420014050

r2的地址：0xc420014060

setVal()方法中r的地址：0xc420014080

r1.height= 8

r2.height= 8

--------------------

setVal2()方法中r的地址：0xc420014050

r1.height= 20

r2.height= 8

7.3.5 method继承

method是可以继承的，如果匿名字段实现了一个method，那么包含这个匿名字段的struct也能调用该匿名结构体中的method

例：

package main

import "fmt"

type Human struct {

name, phone string

age int

}

type Student struct {

Human //匿名字段

school string

}

type Employee struct {

Human //匿名字段

company string

}

func main() {

s1 := Student{Human{"Daniel", "15012345678", 13}, "十一中学"}

e1 := Employee{Human{"Steven", "17812345678", 35}, "1000phone"}

s1.SayHi()

e1.SayHi()

}

func (h \*Human) SayHi() {

fmt.Printf("大家好!我是 %s ，我%d岁，我的联系方式是：%s\n", h.name, h.age , h.phone)

}

运行结果

大家好!我是 Daniel ，我13岁，我的联系方式是：15012345678

大家好!我是 Steven ，我35岁，我的联系方式是：17812345678

7.3.6 method重写

方法是可以继承和重写的。存在继承关系时，按照就近原则进行调用。

例：

package main

import "fmt"

type Human struct {

name, phone string

age int

}

type Student struct {

Human //匿名字段

school string

}

type Employee struct {

Human //匿名字段

company string

}

func main() {

s1 := Student{Human{"Daniel", "15012345678", 13}, "十一中学"}

e1 := Employee{Human{"Steven", "17812345678", 35}, "1000phone"}

s1.SayHi()

e1.SayHi()

}

func (h \*Human) SayHi() {

fmt.Printf("大家好! 我是 %s ，我%d岁，我的联系方式是：%s\n", h.name, h.age, h.phone)

}

//Student的method重写Human的method

func (s \*Student) SayHi() {

fmt.Printf("大家好! 我是 %s ，我%d岁，我在%s上学，我的联系方式是：%s\n", s.name, s.age, s.school, s.phone)

}

//Employee的method重写Human的method

func (e \*Employee) SayHi() {

fmt.Printf("大家好! 我是 %s ，我%d岁，我在%s工作，我的联系方式是：%s\n", e.name, e.age, e.company, e.phone)

}

运行结果

大家好! 我是 Daniel ，我13岁，我在十一中学上学，我的联系方式是：15012345678

大家好! 我是 Steven ，我35岁，我在1000phone工作，我的联系方式是：17812345678

7.4 什么是接口?

7.4.1 概念

面向对象语言中，接口用于定义对象的行为。接口只指定对象应该做什么，实现这种行为的方法(实现细节)是由对象来决定。

在Go语言中，接口是一组方法签名。接口只指定了类型应该具有的方法，类型决定了如何实现这些方法。当某个类型为接口中的所有方法提供了具体的实现细节时，这个类型就被称为实现了该接口。接口定义了一组方法，如果某个对象实现了该接口的所有方法，则此对象就实现了该接口。

Go语言的类型都是隐式实现接口的。任何定义了接口中所有方法的类型都被称为隐式地实现了该接口。

7.4.2 接口的定义语法及示例

定义接口

type 接口名字 interface {

方法1([参数列表]) [返回值]

方法2([参数列表]) [返回值]

...

方法n([参数列表]) [返回值]

}

定义结构体

type 结构体名 struct {

//属性

}

结构体实现接口方法

func (变量名 结构体类型) 方法1([参数列表]) [返回值] {

//方法体

}

func (变量名 结构体类型) 方法2([参数列表]) [返回值] {

//方法体

}

func (变量名 结构体类型) 方法n([参数列表]) [返回值] {

//方法体

}

例：

package main

import "fmt"

type Phone interface {

call()

}

type AndroidPhone struct {

}

type IPhone struct {

}

func (a AndroidPhone) call() {

fmt.Println("我是安卓手机，我可以打电话!")

}

func (i IPhone) call() {

fmt.Println("我是苹果手机，我可以打电话!")

}

func main() {

//定义接口类型的变量

var phone Phone

phone = new(AndroidPhone)

phone.call()

phone = new(IPhone)

phone.call()

}

运行结果：

我是安卓手机，我可以打电话!

我是苹果手机，我可以打电话!

并没有见到上述案例中出现AndroidPhone及iPhone实现接口Phone的语句。那么为什么 new(AndroidPhone)以及new(IPhone)可以直接赋值给接口变量phone呢？

Go没有 implements, extends 关键字。其实这种编程语言叫做duck typing编程语言。

7.4.3 duck typing

编程语言中的鸭子类型



大黄鸭是鸭子吗？

鸭子：脊索动物门、脊椎动物亚门、鸟纲雁形目；大黄鸭无生命：不是鸭子。

duck typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

"当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。"

扩展后，可以理解为：“看起来像鸭子，那么它就是鸭子”。描述事物的外部行为而非内部结构。在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的。

一般来讲，使用 duck typing 的编程语言往往被归类到“动态类型语言”或者“解释型语言”里，比如 Python, Javascript, Ruby 等等；而非duck typing语言往往被归到“静态类型语言“中，比如 C/C++/Java。

以 Java为例， 一个类必须显式地声明：“类实现了某个接口”， 然后才能用在这个接口可以使用的地方。如果你有一个第三方的 Java 库，这个库中的某个类没有声明它实现了某个接口，那么即使这个类中真的有那些接口中的方法，你也不能把这个类的对象用在那些要求用接口的地方。但如果在duck typing的语言中， 你就可以这样做，因为它不要求一个类显式地声明它实现了某个接口。

动态类型的好处很多，Python代码写起来很快。但是缺陷也是显而易见的：错误往往要在运行时才能被发现。相反，静态类型语言往往在编译时就是发现这类错误：如果某个变量的类型没有显式声明实现了某个接口，那么，这个变量就不能用在要求一个实现了这个接口的地方。

Go 类型系统采取了折中的办法：

第一，结构体类型T不需要显式地声明它实现了接口 I。只要类型 T 实现了接口 I 规定的所有方法，它就自动地实现了接口 I。 这样就像动态语言一样省了很多代码，少了许多限制。

第二，将结构体类型的变量显式或者隐式地转换为接口 I 类型的变量 i。这样就可以和其它静态类型语言一样，在编译时检查参数的合法性。

例：

package main

import "fmt"

type ISayHello interface {

SayHello()

}

type Person struct{}

type Duck struct{}

type Duck2 struct{}

func (person Person) SayHello() {

fmt.Printf("Hello!")

}

func (duck Duck) SayHello() {

fmt.Printf("ga ga ga!")

}

func greeting(i ISayHello) {

i.SayHello()

}

func main() {

//person := Person{}

//duck := Duck{}

person := new(Person)

duck := new(Duck)

//以下输出跟接口没有关系

fmt.Println("非接口调用形式")

person.SayHello()

duck.SayHello()

fmt.Println("\n---------------------")

//定义接口变量。

fmt.Println("接口调用形式")

var i ISayHello

i = person

greeting(i)

i = duck

greeting(i)

//可否将一个未实现接口方法的结构体对象赋值给接口呢？

//i = new(Duck2)

}

运行结果：

非接口调用形式

Hello!ga ga ga!

---------------------

接口调用形式

Hello!ga ga ga!

【备注：接口的用法】

用法一：一个函数如果接收接口类型作为参数，那么实际上可以传入该接口的任意实现类对象作为参数。

用法二：定义一个接口变量，那么实际上可以赋值任意实现了该接口的对象。

如果定义了一个接口类型的容器（数组或切片），实际上该容器中可以存储任意的实现类对象。

7.4.4 多态

Go中的多态性是在接口的帮助下实现的。定义接口类型，创建实现该接口的结构体对象。定义接口类型的对象，可以保存实现该接口的任何类型的值。Go语言接口变量的这个特性实现了Go语言中的多态性。接口类型的对象，不能访问其实现类中的属性字段。

例：

package main

import "fmt"

type Income interface {

calculate() float64

source() string

}

//固定账单项目

type FixedBilling struct {

projectName string

biddedAmount float64 //招标总额

}

//定时和材料项目（定时生产项目）

type TimeAndMaterial struct {

projectName string

workHours float64 //工作时长

hourlyRate float64 //每小时工资率

}

//固定收入项目

func (fb FixedBilling) calculate() float64 {

return fb.biddedAmount

}

func (fb FixedBilling) source() string {

return fb.projectName

}

//定时生产项目

func (tm TimeAndMaterial) calculate() float64 {

return tm.workHours \* tm.hourlyRate

}

func (tm TimeAndMaterial) source() string {

return tm.projectName

}

//假设该组织通过广告找到了新的收入来源。让我们看看如何简单地添加新的收入方式和计算总收入，而不用对calculateNetIncome函数做任何更改。由于多态性，这样是可行的。

//首先让我们定义Advertisement类型和calculate()和source()方法。

//广告类型有三个字段adName, costPerclick(每次点击的花费，cost per click)。

type Advertisement struct {

adName string

costPerclick float64

clickCount int

}

func (a Advertisement) calculate() float64 {

return a.costPerclick \* float64(a.clickCount)

}

func (a Advertisement) source() string {

return a.adName

}

//计算和打印总收入的calculateNetIncome函数

func calculateNetIncome(ic []Income) {

netincome := 0.0

for \_, income := range ic {

fmt.Printf("收入来源： %s = $%.2f \n", income.source(), income.calculate())

netincome += income.calculate()

}

fmt.Printf("公司净收入合计 = $%.2f ", netincome)

}

func main() {

project1 := FixedBilling{projectName: "项目1", biddedAmount: 5000}

project2 := FixedBilling{projectName: "项目2", biddedAmount: 10000}

project3 := TimeAndMaterial{projectName: "项目3", workHours: 100, hourlyRate: 40}

project4 := TimeAndMaterial{projectName: "项目4", workHours: 250, hourlyRate: 20}

project5 := Advertisement{adName: "广告5", costPerclick: 0.1, clickCount: 10000}

incomeStreams := []Income{project1, project2, project3, project4, project5}

calculateNetIncome(incomeStreams)

}

说明：

没有对calculateNetIncome函数做任何更改，尽管添加了新的收入方式。全靠多态性而起作用。

由于新的Advertisement类型也实现了Income接口，可以将它添加到incomeStreams切片中。

calculateNetIncome函数也在没有任何更改的情况下工作，因为它可以调用Advertisement类型的calculate()和source()方法。

7.4.5 空接口

空接口：该接口中没有任何的方法。任意类型都可以实现该接口。空interface这样定义：interface{}，也就是包含0个method的interface。用空接口表示任意数据类型。类似于java中的object。空接口常用于以下情形：

* println的参数就是空接口
* 定义一个map：key是string，value是任意数据类型
* 定义一个切片，其中存储任意类型的数据

例：

package main

import "fmt"

//定义空接口

type A interface {

}

type Cat struct {

name string

age int

}

type Person struct {

name string

sex string

}

func main() {

// 用空接口表示任意数据类型。类似于java中的object

var a1 A = Cat{name: "Mimi", age: 1}

var a2 A = Person{"Steven", "man"}

var a3 A = "Learn golang with me!"

var a4 A = 100

var a5 A = 3.14

fmt.Printf("%T，%v \n", a1, a1)

fmt.Printf("%T，%v \n", a2, a2)

fmt.Printf("%T，%v \n", a3, a3)

fmt.Printf("%T，%v \n", a4, a4)

fmt.Printf("%T，%v \n", a5, a5)

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

//println的参数就是空接口

fmt.Println("println的参数可以是任何数据类型，用空接口表示\n", 100, 3.14, Cat{"小天", 2})

//定义一个map：key是string，value是任意数据类型

map1 := make(map[string]interface{})

map1["name"] = "Daniel"

map1["age"] = 13

fmt.Println(map1)

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

//定义一个切片，其中存储任意类型的数据

slice1 := make([]interface{}, 0, 10)

slice1 = append(slice1, a1, a2, a3, a4, a5)

fmt.Println(slice1)

testInterface(slice1)

}

func testInterface(s []interface{}) {

for i := range s {

fmt.Println("第", i+1, "个数据：")

switch ins := s[i].(type) {

case Cat:

fmt.Println("\tcat对象：", ins.name, ins.age)

case Person:

fmt.Println("\tperson对象：", ins.name, ins.sex)

case int:

fmt.Println("\tint类型：", ins)

case string:

fmt.Println("\tstring类型：", ins)

case float64:

fmt.Println("\tfloat64类型：", ins)

}

}

}

7.4.6 接口对象转型

方式一：

instance，ok := 接口对象.(实际类型)

如果该接口对象是对应的实际类型，那么instance就是转型之后对象，ok的值为true。配合if ... else if...语句使用。

方式二：

接口对象.(type)

配合switch...case语句使用。

例：

package main

import (

"fmt"

"math"

)

//1.定义一个接口

type Shape interface {

perimeter() float64

area() float64

}

//2.矩形

type Rectangle struct {

a, b float64

}

//3.三角形

type Triangle struct {

a, b, c float64

}

//4.圆形

type Circle struct {

radius float64 //半径

}

//实现接口的方法

func (r Rectangle) perimeter() float64 {

return 2 \* (r.a + r.b)

}

func (r Rectangle) area() float64 {

return r.a \* r.b

}

func (t Triangle) perimeter() float64 {

return t.a + t.b + t.c

}

func (t Triangle) area() float64 {

p := t.perimeter() / 2 //半周长

//海伦公式

s := math.Sqrt(p \* (p - t.a) \* (p - t.b) \* (p - t.c))

return s

}

func (c Circle) perimeter() float64 {

return 2 \* math.Pi \* c.radius

}

func (c Circle) area() float64 {

return math.Pow(c.radius, 2) \* math.Pi

}

//测试函数

func testShape(s Shape) {

fmt.Printf("周长：%.2f ， 面积：%.2f \n", s.perimeter(), s.area())

}

func main() {

var s Shape

s = Rectangle{3, 4}

testShape(s)

s = Triangle{3, 4, 5}

testShape(s)

s = Circle{1}

testShape(s)

}

//接口对象转型——方式1

func getType(s Shape) {

if instance, ok := s.(Rectangle); ok {

fmt.Printf("矩形：长度为%.2f ， 宽为%.2f ，\t", instance.a, instance.b)

} else if instance, ok := s.(Triangle); ok {

fmt.Printf("三角形：三边分别为%.2f ，%.2f ， %.2f ，\t", instance.a, instance.b, instance.c)

} else if instance, ok := s.(Circle); ok {

fmt.Printf("圆形：半径为%.2f ，\t", instance.radius)

}

}

//接口对象转型——方式2

func getType2(s Shape) {

switch instance := s.(type) {

case Rectangle:

fmt.Printf("矩形：长度为%.2f ， 宽为%.2f ，\t", instance.a, instance.b)

case Triangle:

fmt.Printf("三角形：三边分别为%.2f ，%.2f ， %.2f ，\t", instance.a, instance.b, instance.c)

case Circle:

fmt.Printf("圆形：半径为%.2f ，\t", instance.radius)

}

}

7.5 小结

本章学习Go语言的面向对象相关的内容，Go语言的面向对象设计与其它语言不同，需要用心研究与理解。如果说函数是细分代码最重要的工具，那么面向对象特性可以理解为切分业务的最重要的工具，我们在设计对象时将按照单一职责的约束，保证每个对象只做一件事，每个方法只做一件事，这样才能让代码具备足够的可扩展性及健壮性。另外需要注意，面向对象思想是一种最接近人类思考方式的方法。面向对象编程是在软件开发的过程中，使用面向对象思想对软件进行分析与设计。面向对象编程语言是指仅支持面向对象语法的编程语言，例如Java。Go语言仅仅是支持面向对象的语法，而并不是完全的面向对象的编程语言。

第八章 Go语言异常处理

在本章中我们将学习以下内容：

* Go语言中的错误处理
* 创建error对象的几种方式
* 自定义错误
* Go语言中的异常处理
* defer延迟函数
* defer延迟方法
* defer延迟参数
* panic及recover

8.1 错误处理

8.1.1 错误是什么?

错误指程序中出现异常的情况，从而导致程序无法继续执行。大多编程语言中使用try...catch...finally语句捕获异常。

假设我们正在尝试打开一个文件，文件系统中不存在这个文件。这是一个异常情况，它表示为一个错误。

Go语言中没有try...catch。Go语言通过内置的错误类型提供了非常简单的错误处理机制。错误值可以存储在变量中，通过函数中返回。如果一个函数或方法返回一个错误，按照惯例它必须是函数返回的最后一个值。处理错误的惯用方式是将返回的错误与nil进行比较。nil值表示没有发生错误，而非nil值表示出现错误。如果不是nil，需打印输出错误。

error错误类型的本质是一个接口类型，其中包含一个Error()方法。

type error interface {

Error() string

}

任何实现这个接口的类型都可以作为一个错误使用。这个方法提供了对错误的描述。

8.1.2 创建error对象的方式

1.使用errors包下的New()函数返回error对象。errors.New()创建新的错误对象。

代码分析

//errors 包实现了操作errors类型的方法

package errors

//New方法返回一个错误对象

func New(text string) error {

return &errorString{text}

}

type errorString struct {

s string

}

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

return e.s

}

2.使用fmt包下的Errorf()函数返回error对象，其本质上还是调用errors.New()

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

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

}

8.1.3 自定义错误

定义一个结构体，用于表示自定义错误的类型；

自定义错误类型实现error接口的方法 ：Error() string

定义一个返回error的函数，根据实际需求实现。

例：

package main

import (

"fmt"

"time"

)

//定义一个结构体，表示自定义错误的类型

type MyError struct {

When time.Time

What string

}

//自定义错误类型实现error接口的方法 ：Error() string

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

return fmt.Sprintf("%v : %v", e.When, e.What)

}

//定义一个返回error的函数。求矩形的面积

func getArea(width, length float64) (float64, error) {

errorMsg := ""

if width < 0 && length < 0 {

errorMsg = fmt.Sprintf("长度:%v ，宽度:%v ，均为负数", length, width)

} else if length < 0 {

errorMsg = fmt.Sprintf("长度:%v ，出现负数", length)

} else if width < 0 {

errorMsg = fmt.Sprintf("宽度:%v ，出现负数", width)

}

if errorMsg != "" {

return 0, &MyError{time.Now(), errorMsg}

} else {

return width \* length, nil

}

}

func main() {

res1, err := getArea(-4, -6)

if err != nil {

fmt.Printf(err.Error())//输出：2020-03-28 13:57:24.8765849 +0800 CST m=+0.004970401 : 长度:-6 ，宽度:-4 ，均为负数

} else {

fmt.Println("面积是:", res1)

}

}

8.2 延迟是什么?

defer即延迟语句，在需要的情况下，Go允许使用defer、panic、recover这种异常处理形式。

defer可以延迟函数、延迟方法、延迟参数。

8.2.1 延迟函数

Go语言允许在函数中添加多个defer语句。当函数执行到最后时，这些defer语句会按照逆序执行，最后该函数返回。特别是在执行一些打开资源的操作时，遇到错误需要提前返回，在返回前你需要关闭相应的资源，不然很容易造成资源泄露等问题。如果有很多调用defer，那么defer是采用后进先出的模式。

在离开所在的方法时，defer也会执行（报错的时候也会执行）。

所在的goroutine发生panic时，defer也会执行。

例：

package main

import "fmt"

func main() {

s1 := []int{78, 109, 2, 563, 300}

largest(s1)

}

func finished() {

fmt.Println("结束！")

}

func largest(s []int) {

defer finished()

fmt.Println("开始寻找最大数...")

max := s[0]

for \_, v := range s {

if v > max {

max = v

}

}

fmt.Printf("%v中的最大数为：%v \n", s , max)

}

运行结果：

开始寻找最大数...

[78 109 2 563 300]中的最大数为：563

结束！

8.2.2 延迟方法

延迟并不仅仅局限于函数。延迟一个方法调用也是完全合法的。

例：

package main

import "fmt"

type person struct {

firstName string

lastName string

}

func (p person) fullName() {

fmt.Printf("%s %s", p.firstName, p.lastName)

}

func main() {

p := person{"Steven", "Wang"}

defer p.fullName()

fmt.Printf("Welcome ")

}

运行结果：

Welcome Steven Wang

8.2.3 延迟参数

延迟函数的参数在执行延迟语句时被执行，而不是在执行实际的函数调用时执行。

例：

package main

import "fmt"

func printAdd(a , b int) {

fmt.Printf("延迟函数中：参数a , b分别为%d,%d ，两数之和为：%d\n", a , b , a+b)

}

func main() {

a := 5

b := 6

//延迟函数的参数在执行延迟语句时被执行，而不是在执行实际的函数调用时执行。

defer printAdd(a , b)

a = 10

b = 7

fmt.Printf("延迟函数执行前：参数a , b分别为%d,%d ，两数之和为：%d\n", a , b , a+b)

}

运行结果：

延迟函数执行前：参数a , b分别为10,7 ，两数之和为：17

延迟函数中：参数a , b分别为5,6 ，两数之和为：11

8.2.4 堆栈的推迟

当一个函数有多个延迟调用时，它们被添加到一个堆栈中，并在Last In First Out（LIFO）后进先出的顺序中执行。

例：利用defer实现字符串倒序。

package main

import "fmt"

func main() {

name := "StevenWang欢迎学习区块链"

fmt.Printf("原始字符串: %s\n", name)

fmt.Println("翻转后字符串: ")

ReverseString(name)

}

func ReverseString(str string) {

for \_, v := range []rune(str) {

defer fmt.Printf("%c", v)

}

}

返回结果：

原始字符串: StevenWang欢迎学习区块链

翻转后字符串:

链块区习学迎欢gnaWnevetS

8.2.5 延迟的应用

到目前为止，我们所写的示例代码，并没有实际的应用。现在看一下关于延迟的应用。在不考虑代码流的情况下，延迟被执行。让我们以一个使用WaitGroup的程序示例来理解这个问题。我们将首先编写程序而不使用延迟，然后我们将修改它以使用延迟，并理解延迟是多么有用。

例：

package main

import (

"fmt"

"sync"

)

type rect struct {

length int

width int

}

func (r rect) area(wg \*sync.WaitGroup) {

if r.length < 0 {

fmt.Printf("rect %v's length should be greater than zero\n", r)

wg.Done()

return

}

if r.width < 0 {

fmt.Printf("rect %v's width should be greater than zero\n", r)

wg.Done()

return

}

area := r.length \* r.width

fmt.Printf("rect %v's area %d\n", r, area)

wg.Done()

}

func main() {

var wg sync.WaitGroup

r1 := rect{-67, 89}

r2 := rect{5, -67}

r3 := rect{8, 9}

rects := []rect{r1, r2, r3}

for \_, v := range rects {

wg.Add(1)

go v.area(&wg)

}

wg.Wait()

fmt.Println("All go routines finished executing")

}

修改以上代码：

package main

import (

"fmt"

"sync"

)

type rect struct {

length int

width int

}

func (r rect) area(wg \*sync.WaitGroup) {

defer wg.Done()

if r.length < 0 {

fmt.Printf("rect %v's length should be greater than zero\n", r)

return

}

if r.width < 0 {

fmt.Printf("rect %v's width should be greater than zero\n", r)

return

}

area := r.length \* r.width

fmt.Printf("rect %v's area %d\n", r, area)

}

func main() {

var wg sync.WaitGroup

r1 := rect{-67, 89}

r2 := rect{5, -67}

r3 := rect{8, 9}

rects := []rect{r1, r2, r3}

for \_, v := range rects {

wg.Add(1)

go v.area(&wg)

}

wg.Wait()

fmt.Println("All go routines finished executing")

}

程序运行结果：

rect {8 9}'s area 72

rect {-67 89}'s length should be greater than zero

rect {5 -67}'s width should be greater than zero

All go routines finished executing

8.3 panic和recover（宕机和宕机恢复）

8.3.1 panic和recover机制

panic：词义"恐慌"，recover："恢复"。Go语言追求简洁优雅，Go没有像Java那样的 try...catch...finally 异常处理机制。Go语言设计者认为，将异常与流程控制混在一起会让代码变得混乱。Go语言中，使用多值返回来返回错误。不用异常代替错误，更不用异常来控制流程。Go语言利用panic()，recover()，实现程序中的极特殊的异常处理。换句话说，在需要的情况下，Go才使用defer、panic、recover这种异常处理形式。

panic()，让当前的程序进入恐慌，中断程序的执行。或者说，panic 是一个内建函数，可以中断原有的控制流程，进入一个令人恐慌的流程中。当函数F调用panic，函数F的执行被中断，但是F中的延迟函数会正常执行，然后F返回到调用它的地方。在调用的地方，F的行为就像调用了panic。这一过程继续向上，直到发生panic的goroutine中所有调用的函数返回，此时程序退出。恐慌可以直接调用panic产生。也可以由运行时错误产生，例如访问越界的数组。

recover 是一个内建的函数，可以让进入令人恐慌的流程中的goroutine恢复过来。recover()，让程序恢复，必须在defer函数中执行。换句话说，recover仅在延迟函数中有效。

在正常 的执行过程中，调用recover会返回nil，并且没有其它任何效果。如果当前的goroutine陷入恐慌，调用 recover可以捕获到panic的输入值，并且恢复正常的执行。

一定要记住，应当把它作为最后的手段来使用，也就是说，我们的代码中应当没有，或者很少有panic这样的东西。

8.3.2 案例

package main

import "fmt"

func main() {

funA()

funB()

funC()

fmt.Println("main...over....")

}

func funA() {

fmt.Println("我是函数funA()...")

}

func funB() { //外围函数

defer func() {

if msg := recover(); msg != nil {

fmt.Println(msg, "恢复啦。。。")

}

}()

fmt.Println("我是函数funB()...")

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

fmt.Println("i:", i)

if i == 5 {

//让程序中断

panic("funB函数，恐慌啦。。。") //打断程序的执行。。

}

}

//当外围函数中的代码引发运行恐慌时，只有其中所有的延迟函数都执行完毕后，该运行时恐慌才会真正被扩展至调用函数。

}

func funC() {

defer func() {

fmt.Println("func的延迟函数。。。")

//if msg := recover(); msg != nil {

// fmt.Println(msg, "恢复啦。。。")

//}

fmt.Println("recover执行了" , recover())

}()

fmt.Println("我是函数funC()。。")

panic("funC恐慌啦。。")

}

8.4 小结

本章学习Go语言的异常处理的相关内容。在Go语言中我们习惯使用返回err对象的方式给出异常信息，在Go语言软件开发中，这也将是我们最常见的一种方法，通过对异常处理的研究，可以更好的处理软件开发过程中出现的各种异常，并能帮助我们快速的定位问题。是提高开发效率的利器。

第九章 Go语言的IO操作

本章重点为大家介绍如下的内容：

* 文件信息FileInfo
* 文件的常规操作（os包）
* 读取文件和写入文件（io及os包）
* ioutil包
* bufio包

9.1 文件信息

9.1.1 FileInfo接口

FileInfo 描述一个文件，并由 Stat 和 Lstat 返回。

type FileInfo interface {

Name() string // 文件名，无扩展名

Size() int64 // 文件大小

Mode() FileMode // 文件模式位

ModTime() time.Time // 修改时间

IsDir() bool // 是否是目录

Sys() interface{} // 底层数据来源(可以return nil)

}

例:

package main

import (

"fmt"

"os"

)

func main() {

/\*

文件路径：

绝对路径：absolute

/Users/steven/Documents/go\_project/files/dsa.png

相对路径：relative 都是相当于当前的工程

.当前目录

..上一层

\*/

//绝对路径形式

fileInfo, err := os.Stat("/Users/steven/Documents/go\_project/files")

fileInfo, err = os.Stat("/Users/steven/Documents/go\_project/files/dsa.png")

//相对路径

fileInfo, err = os.Stat("./files/yesterday.mp3")

fileInfo, err = os.Stat("../node\_test/open.js")

if err != nil {

fmt.Println("err:", err.Error())

} else {

fmt.Printf("%v, %T \n", fileInfo, fileInfo)

//文件名

fmt.Println(fileInfo.Name())

//是否是目录

fmt.Println(fileInfo.IsDir())

//尺寸大小

fmt.Println(fileInfo.Size())

//权限

fmt.Println(fileInfo.Mode())

//修改时间

fmt.Println(fileInfo.ModTime())

}

}

第一符号表示类型。如果是-表示文件，如果是d表示目录。

文件的权限一共有9个字符表示，分成三组，分别表示文件所属用户owner的权限，文件所属用户组group的权限，其他人others的权限；

r 表示读权限, w表示写权限， x表示执行权限。

并且此文件所属用户拥有读、写、执行三项权限，其余的用户组，其他用户不拥有任何权限（全部都是-）

例如：-rwxrwxrwx 表示这是一个文件，用户权限、用户组权限、其他人权限都是可读、可写、可操作

drwxr-xr-x 表示这是一个目录，用户权限是可读、可写、可操作，用户组权限是可读可操作，其他人权限是可读可操作。

还可以用8进制表示法：

r 4

w 2

x 1

- 0

例如：-rwxrwxrwx 权限用8进制表示为：0777

9.1.2 文件路径

路径操作相关的函数

filepath.IsAbs():判断是否是绝对路径

例：

fileName1 := "/Users/steven/Documents/go\_project/files/dsa.png"

ileName2 := "files/blockchain.txt"

fmt.Println(filepath.IsAbs(fileName1)) //true

fmt.Println(filepath.IsAbs(fileName2)) //false

filepath.Rel():获取相对路径

例：

fmt.Println(filepath.Rel("/Users/steven/Documents", fileName1))

filepath.Abs():获取绝对路径

例：

fmt.Println(filepath.Abs(fileName2))

path.Join():将任意数量的路径元素加入到单个路径中

例：

fmt.Println("获取父目录：", path.Join(fileName1, "."))

9.2 文件常规操作

9.2.1 创建目录

os.MKdir()：使用指定的名称和权限位创建一个新目录。如果有错误，它将是 \* PathError 类型。

os.MKdirAll()：创建一个名为 path 的目录以及任何必要的父项，并返回 nil ，否则返回错误。许可位 perm 用于 MkdirAll 创建的所有目录。如果 path 已经是一个目录，MkdirAll 什么也不做，并返回 nil 。

9.2.2 创建文件

os.Create() :使用模式0666（在 umask 之前）创建命名文件，如果它已经存在，则截断它。如果成功，返回文件上的方法可用于 I/O ； 关联的文件描述符具有模式 O\_RDWR 。如果有错误，它将是 \* PathError 类型。

9.2.3 打开文件

打开文件，让当前的程序和指定的文件建立了一个链接

os.Open(filename) 打开指定文件。如果成功，以 O\_RDONLY 模式打开文件并返回文件对象。如果有错误，返回 \* PathError 类型。

func OpenFile(name string, flag int, perm FileMode) (\*File, error)

FileMode代表文件的模式和权限位。这些字位在所有的操作系统都有相同的含义，因此文件的信息可以在不同的操作系统之间安全的移植。不是所有的位都能用于所有的系统，唯一共有的是用于表示目录的ModeDir位。

文件的打开方式:

O\_RDONLY：只读模式(read-only)

O\_WRONLY：只写模式(write-only)

O\_RDWR：读写模式(read-write)

O\_APPEND：追加模式(append)

O\_CREATE：文件不存在就创建(create a new file if none exists.)

OpenFile 是广义的公开称呼；大多数用户将使用“打开”或“创建”。它打开具有指定标志（O\_RDONLY等）和指定（0666等）的指定文件（如果适用）。如果成功，返回文件上的方法可用于 I/O 。如果有错误，它将是 \* PathError 类型

9.2.4 关闭文件

file.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。关闭文件使程序和文件之间的链接断开。

9.2.5 删除

os.Remove() 删除已命名的文件或目录，该目录必须是个空目录。

os.RemoveAll() 移除所有的路径和它包含的任何子节点。

9.3 读写文件及复制文件

9.3.1 读取文件

读取文件的步骤：

打开文件->读取文件->关闭文件

使用file.Read([]byte)从文件中开始读取数据，返回值n是实际读取的字节数。如果读取到文件末尾，n为0，err为EOF（end of file）

例：

//打开文件

fileName = "./files/blockchain.txt"

file, err := os.Open(fileName)

if err != nil {

fmt.Println("打开文件有误：", err.Error())

return

}

//读取文件内容

//从file对应的文件中读取最多len(bs)个数据，存入到bs切片中，n是实际读入的数量

sli := make([]byte, 1024, 1024)

n := -1

for {

n, err = file.Read(sli)

if n == 0 || err == io.EOF {

fmt.Println("读取到文件末尾了，结束读取操作。。")

break

}

fmt.Println(string(sli[:n]))

}

//关闭文件

file.Close()

9.3.2 写入文件

写入文件的步骤：

打开或创建文件->写入文件->关闭文件

例：

//打开文件

file, err := os.OpenFile("./test1/abc2.txt", os.O\_CREATE|os.O\_WRONLY, os.ModePerm)

if err != nil {

fmt.Println("打开文件有误：", err.Error())

}

fmt.Println(file)

//延迟执行关闭文件

defer file.Close()

//写入文件

n, err := file.Write([]byte("abcde123456"))

fmt.Println(err)

fmt.Println(n)

n, err = file.WriteString("中国人")

fmt.Println(err)

fmt.Println(n)

9.3.3 复制文件

/\*

该函数的功能：实现文件的拷贝，返回值是拷贝的总数量(字节),错误

\*/

func copyFile1(srcFile, destFile string) (int, error) {

file1, err := os.Open(srcFile)

if err != nil {

return 0, err

}

file2, err := os.OpenFile(destFile, os.O\_WRONLY|os.O\_CREATE, os.ModePerm)

if err != nil {

return 0, err

}

defer file1.Close()

defer file2.Close()

//拷贝数据

bs := make([]byte, 1024, 1024)

n := -1 //读取的数据量

total := 0

for {

n, err = file1.Read(bs)

if err == io.EOF || n == 0 {

fmt.Println("拷贝完毕。。")

break

}

if err != nil {

fmt.Println("err:" , err.Error())

return total, err

} else {

total += n

file2.Write(bs[:n])

}

}

return total, nil

}

func copyFile2(srcFile, destFile string) (int64, error) {

file1, err := os.Open(srcFile)

if err != nil {

return 0, err

}

file2, err := os.OpenFile(destFile, os.O\_WRONLY|os.O\_CREATE, os.ModePerm)

if err != nil {

return 0, err

}

defer file1.Close()

defer file2.Close()

return io.Copy(file2, file1)

}

9.4 ioutil包

9.4.1 ioutil包核心函数

ReadFile()

读取文件中的所有的数据，返回读取的字节数组

WriteFile()

向指定文件写入数据，如果文件不存在，则创建文件，写入数据之前清空文件

ReadDir()

读取一个目录下的子内容：子文件和子目录，但是仅有一层

TempDir(）

在当前目录下，创建一个以指定字符串为前缀的临时文件夹，并返回文件夹路径

TempFile()

在当前目录下，创建一个以指定字符串为前缀的文件，并以读写模式打开文件，并返回os.File指针对象

9.4.2 示例代码

//读取文件中的所有的数据

fileName1 := "./files/blockchain.txt"

data, err := ioutil.ReadFile(fileName1)

if err != nil {

fmt.Println("读取文件异常：", err.Error())

} else {

fmt.Println(string(data))

}

//写出数据

fileName2 := "./files/xyz.txt"

s1 := "helloworld面朝大海春暖花开"

err = ioutil.WriteFile(fileName2, []byte(s1), 0777)

if err != nil {

fmt.Println("写入文件异常：", err.Error())

} else {

fmt.Println("写入文件ok")

}

//ReadDir(),读取一个目录下的子内容：子文件和子目录，但是仅有一层

dirName := "./src/"

fileInfos, \_ := ioutil.ReadDir(dirName)

fmt.Println(len(fileInfos))

for i := 0; i < len(fileInfos); i++ {

//fmt.Printf("%T\n",fileInfos[i])

fmt.Println(i, fileInfos[i].Name(), fileInfos[i].IsDir())

}

9.5 bufio包

9.5.1 bufio的原理

bufio实现了带缓冲的 I/O 操作，达到高效io读写。

bufio 包封装一个 Reader 及 Writer结构体对象；bufio 包中的Reader 及 Writer结构体分别实现了io.Reader和io.Writer接口；bufio包对io包下的Reader、Write对象进行包装，通过对io模块的封装，提供了数据缓冲功能，能够一定程度减少大块数据读写带来的开销，所以bufio 要比io的读写更快速。

bufio 是通过缓冲来提高效率；把文件读取进缓冲区之后，再读取的时候就可以避免文件系统的io ，从而提高速度；在进行写操作时，先把文件写入缓冲区，然后由缓冲写入文件系统。

有人可能会表示困惑，直接把 内容->文件 和 内容->缓冲->文件相比， 缓冲区好像没有起到作用嘛。

其实缓冲区的设计是为了存储多次的写入，最后一口气把缓冲区内容写入文件。当发起一次读写操作时，会首先尝试从缓冲区获取数据；只有当缓冲区没有数据时，才会从数据源获取数据更新缓冲。

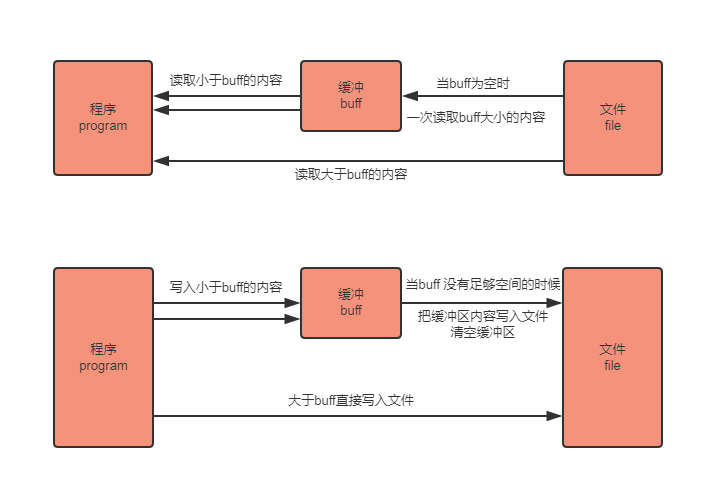


图9.1

bufio.Read(p []byte) 相当于读取大小len(p)的内容，思路如下：

当缓冲区有内容时，将缓冲区内容全部填入p并清空缓冲区

当缓冲区没有内容的时候且len(p)>len(buf)，即要读取的内容比缓冲区还要大，直接去文件读取即可

当缓冲区没有内容的时候且len(p)<len(buf)，即要读取的内容比缓冲区小，缓冲区从文件读取内容充满缓冲区，并将p填满（此时缓冲区有剩余内容）

以后再次读取时缓冲区有内容，将缓冲区内容全部填入p并清空缓存冲（此时和情况1一样）

bufio.Write(p []byte) 的思路如下：

判断buf中可用容量是否可以放下 p

如果能放下，直接把p拼接到buf后面，即把内容放到缓冲区

如果缓冲区的可用容量不足以放下，且此时缓冲区是空的，直接把p写入文件即可

如果缓冲区的可用容量不足以放下，且此时缓冲区有内容，则用p把缓冲区填满，把缓冲区所有内容写入文件，并清空缓冲区

判断p的剩余内容大小能否放到缓冲区，如果能放下（此时和步骤1情况一样）则把内容放到缓冲区

如果p的剩余内容依旧大于缓冲区，（注意此时缓冲区是空的，情况和步骤2一样）则把p的剩余内容直接写入文件

9.5.2 bufio.Reader结构体

1.bufio.Reader的所有方法

func NewReader(rd io.Reader) \*Reader

func NewReaderSize(rd io.Reader, size int) \*Reader

func (b \*Reader) Buffered() int

func (b \*Reader) Discard(n int) (discarded int, err error) //丢弃接下来n个byte数据

func (b \*Reader) Peek(n int) ([]byte, error) //获取当前缓冲区内接下来的n个byte，但是不移动指针

func (b \*Reader) Read(p []byte) (n int, err error) //读取n个byte数据

func (b \*Reader) ReadByte() (byte, error) //读取一个byte

func (b \*Reader) ReadBytes(delim byte) ([]byte, error) //读取byte列表

func (b \*Reader) ReadLine() (line []byte, isPrefix bool, err error) //读取一行数据，由’\n’分隔

func (b \*Reader) ReadRune() (r rune, size int, err error) //读取一个utf-8字符

func (b \*Reader) ReadSlice(delim byte) (line []byte, err error)

func (b \*Reader) ReadString(delim byte) (string, error) //读取一个字符串

func (b \*Reader) Reset(r io.Reader) //清空整个缓冲区

func (b \*Reader) UnreadByte() error

func (b \*Reader) UnreadRune() error

func (b \*Reader) WriteTo(w io.Writer) (n int64, err error)

2.NewReader()与NewReaderSize()

将 rd 封装成一个拥有 size 大小缓存的 bufio.Reader 对象

NewReader 相当于 NewReaderSize(rd, 4096)

3.ReadLine()

ReadLine 是一个低级的原始的行读取操作

大多数情况下，应该使用 ReadBytes('\n') 或 ReadString('\n')，或者使用一个 Scanner

ReadLine 通过调用 ReadSlice 方法实现，返回的也是缓存的切片

ReadLine 尝试返回一个单行数据，不包括行尾标记（\n 或 \r\n）

如果在缓存中找不到行尾标记，则设置 isPrefix 为 true，表示查找未完成

同时读出缓存中的数据并作为切片返回

只有在当前缓存中找到行尾标记，才将 isPrefix 设置为 false，表示查找完成

可以多次调用 ReadLine 来读出一行

返回的数据在下一次读取操作之前是有效的

如果 ReadLine 无法获取任何数据，则返回一个错误信息（通常是 io.EOF）

4. ReadBytes()

ReadBytes 在 b 中查找 delim 并读出 delim 及其之前的所有数据

如果ReadBytes 在找到 delim 之前遇到错误，则返回遇到错误之前的所有数据，同时返回遇到的错误（通常是 io.EOF）

只有当 ReadBytes 找不到 delim 时，err 才不为 nil

对于简单的用途，使用 Scanner 可能更方便

5.ReadString()

ReadString 功能同 ReadBytes，只不过返回的是一个字符串

例：

//测试Reader的ReadString()

func testReader() {

fileName := "./files/blockchain.txt"

file1, \_ := os.Open(fileName) //看作是io包下的Reader，Write的实现

reader1 := bufio.NewReader(file1) //构建带缓存的Reader对象：bufio.Reader

fmt.Printf("%T\n", reader1)

for {

s1, err := reader1.ReadString('\n')

//ReadBytes reads until the first occurrence of delim in the input,returning a slice containing the data up to and including the delimiter.

fmt.Print(s1)

if err == io.EOF {

fmt.Println("\n读取完毕！")

break

}

}

file1.Close()

}

9.5.3 bufio.Writer结构体

1.bufio.Writer的所有方法

func NewWriter(w io.Writer) \*Writer

func NewWriterSize(w io.Writer, size int) \*Writer

func (b \*Writer) Write(p []byte) (nn int, err error) // 写入n个 byte数据

func (b \*Writer) Reset(w io.Writer) // 重置当前缓冲区

func (b \*Writer) Flush() error // 清空当前缓冲区，将数据写入输出

func (b \*Writer) WriteByte(c byte) error // 写入一个字节

func (b \*Writer) WriteRune(r rune) (size int, err error） // 写入一个字符

func (b \*Writer) WriteString(s string) (int, error) // 写入字符串

2.NewWriter()与NewWriterSize()

func NewWriter(wr io.Writer) \*Writer

NewWriter 相当于 NewWriterSize(wr, 4096)

3.Write()和WriteString(）

例：

//通过bufio拷贝文件

func testWriter() {

fileName2 := "./files/music.mp3"

file2, \_ := os.Open(fileName2) //看作是io包下的Reader，Write的实现

reader2 := bufio.NewReader(file2)

fileName3 := "./files/abc.mp3"

file3, \_ := os.OpenFile(fileName3, os.O\_WRONLY|os.O\_CREATE, os.ModePerm)

writer1 := bufio.NewWriter(file3)

for {

bs, err := reader2.ReadBytes(' ')

writer1.Write(bs)

writer1.Flush()

if err == io.EOF {

fmt.Println("读取完毕。。")

break

}

}

file2.Close()

file3.Close()

}

9.5.4 Scanner

实际使用中，更推荐使用Scanner对数据进行读取，而非直接使用Reader类。Scanner可以通过splitFunc将输入数据拆分为多个token，然后依次进行读取。和Reader类似，Scanner需要绑定到某个io.Reader上，通过NewScannner进行创建，函数声明如下：

func NewScanner(r io.Reader) \*Scanner

常用方法

func (s \*Scanner) Scan() bool

func (s \*Scanner) Text() string

func (s \*Scanner) Text() []byte

bufio模块提供了几个默认splitFunc，能够满足大部分场景的需求，包括：

ScanBytes，按照byte进行拆分

ScanLines，按照行(“\n”)进行拆分

ScanRunes，按照utf-8字符进行拆分

ScanWords，按照单词(” “)进行拆分

通过Scanner的Split方法，可以为Scanner指定splitFunc。使用方法如下：

scanner.split(bufio.ScanWords）

例：

//测试scanner

func testScanner() {

//s := ""

//fmt.Scanln(&s)

//fmt.Print(s)

reader1 := bufio.NewReader(os.Stdin)

scanner := bufio.NewScanner(reader1)

//ScanBytes，按照byte进行拆分

//ScanLines，按照行(“\n”)进行拆分

//ScanRunes，按照utf-8字符进行拆分

//ScanWords，按照单词(” “)进行拆分

scanner.Split(bufio.ScanWords)

//等待输入

for scanner.Scan() {

fmt.Println(scanner.Text())

if scanner.Text() == "q!" {

break

}

}

}

9.6 小结

本章学习Go语言的IO流相关内容。IO操作也是在软件开发中占有重要地位的内容，其作用几乎覆盖了开发的每一个过程，包括且不限于日志存储、图片处理、文本处理等众多场景。虽然IO本身有Go提供的函数包，但是我们依旧需要了解最基本的IO操作，这也有利于我们理解后续的一些与IO相关的内容。

第十章 Go语言数据库编程

本章重点为大家介绍如下的内容：

* 什么是数据库
* 关系型数据库
* 数据库范式
* MySQL的特点
* Go程序中实现MySQL数据库操作

10.1 数据库介绍

数据库（Database）是按照数据结构来组织、存储和管理数据的仓库，每个数据库都有不同的API用于创建、存储、管理数据。

为了方便数据的存储和管理，将数据按照特定的规律存储在磁盘上。通过数据库管理系统，有效地组织和管理存储在数据库中的数据。

数据库（database）是按照数据结构来组织、存储和管理数据的仓库（通常是一个文件或一组文件）。

理解数据库的一种最简单的办法就是将其想象为一个文件柜，此文件柜是一个存放数据的物理位置。

数据其实可以存储在文本文件中，但是只适用于数据量很少的时候，数据量很少的情况下，在文件中保存数据、读写数据，操作简单便利且速度快；但是当数据量稍微大的情况，普通文本文件保存数据的方式就显得捉襟见肘，甚至束手无策。比如数据结构、数据类型无法设计，数据读写方式复杂且速度慢，对于复杂的数据查询更是无法做到。所以需要专业的数据库软件；

存储数据也可以使用微软的 Excel，但是，随着数据越来越多，在 Excel这样的电子表格里查找数据就变得很慢，而且不可靠。所以需要更高效的数据库管理软件。

而在当今的互联网中，最常见的数据库模型主要是两种，即关系型数据库和非关系型数据库。

关系型数据库包括：Oracle、MySQL、SQLServer、Access、DB2等

非关系型数据库包括：

键值存储型数据库：Redis

文档存储型数据库：MongoDB、CouchDB

列表存储型数据库：HBase

图数据库：FlockDB

10.2 MySQL数据库介绍

MySQL 是一个关系型数据库管理系统，目前属于 Oracle （甲骨文）旗下产品；

MySQL 是最流行的关系型数据库管理系统之一。在 WEB 应用方面，MySQL是最好的 RDBMS (Relational Database Management System，关系型数据库管理系统) 应用软件。关系型数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性；

MySQL所使用的 SQL 语言是用于访问数据库的最标准的结构化查询语言；

MySQL支持大型的数据库。可以处理拥有上千万条记录的大型数据库。支持5000万条记录的数据仓库，32位系统表文件最大可支持4GB，64位系统支持最大的表文件为8TB；

MySQL支持运行于多个系统上，且支持多种语言。包括C、C++、Python、Java、Perl、PHP、Ruby等；

与其他的大型数据库，例如 Oracle、DB2、SQL Server等相比，MySQL 自有它的不足之处，但是这丝毫也没有减少它受欢迎的程度。对于一般的个人使用者和中小型企业来说，MySQL提供的功能已经绰绰有余。而且MySQL 采用了双授权政策，分为社区版和商业版，由于其体积小、速度快，尤其是开源这一特点，不需要支付额外的费用，总体拥有成本低，一般中小型网站的开发都选择 MySQL 作为网站数据库。

10.3 Go程序操作MySQL数据库

10.3.1 安装mysql模块

go官方提供了database包，database包下有sql/driver。该包用来定义操作数据库的接口，这保证了无论使用哪种数据库，他们的操作方式都是相同的。但go官方并没有提供连接数据库的driver，如果要操作数据库，还需要第三方的driver 包。最常用的有：https://github.com/go-sql-driver/mysql ，支持database/sql，全部采用go编写。获取该第三方库的方法是在代码中直接引用该库的地址：

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

Go Module会自动帮我们查找模块并下载，获取到可用的库之后，我们就可以尝试着连接MySQL服务器。

例：

import (

"database/sql"

"fmt"

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

)

匿名导包：只导入包但是不使用包内的类型和数据。使用匿名导包的方式(在包路径前添加下划线 “\_” )导入MySQL驱动。匿名导入的包与其他方式导入包一样，会让导入包编译到可执行文件中。

Go 提供了database/sql包，用于对SQL数据库的访问。它提供了一系列接口方法，用于访问关系数据库。它并不会提供数据库特有的方法，那些特有的方法交给数据库驱动去实现。

通常来说,导入包后就能调用该包中的数据和方法。但是对于数据库操作来说，我们不应该直接使用导入的驱动包所提供的方法，而是应该使用 sql.DB对象所提供的统一的方法。因此在导入 mysql 驱动时，使用了匿名导入包的方式。当导入一个数据库驱动后，该驱动会自行初始化并注册到Golang的database/sql上下文中，这样就可以通过 database/sql 包所提供的方法来访问数据库了。

10.3.2 连接数据库

使用sql包中的Open()函数连接数据库，sql.Open()函数原型如下:

func Open(driverName, dataSourceName string) (\*DB, error)

driverName:使用的驱动名，这个名字其实就是数据库驱动注册到 database/sql 时所使用的名字。

dataSourceName: 数据库连接信息，这个连接包含了数据库的用户名,密码,数据库主机以及需要连接的数据库名等信息。

db, err := sql.Open("mysql", "用户名:密码@tcp(IP:端口)/数据库?charset=utf8")

例：

package main

import (

"database/sql"

"fmt"

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

)

func main() {

db,err := sql.Open("mysql", "root:123456@tcp(127.0.0.1:3306)/testdb?charset=utf8")

if err != nil {

fmt.Println(err)

return

}

db.Close()

}

sql.Open()函数返回sql.DB对象，作为操作数据库的入口对象sql.DB,主要为我们提供了两个重要的功能。

1.sql.DB 通过数据库驱动为我们提供管理底层数据库连接的打开和关闭操作。

2.sql.DB 为我们管理数据库连接池。正在使用的连接被标记为繁忙，用完后回到连接池等待下次使用。所以，如果你没有把连接释放回连接池，会导致过多连接使系统资源耗尽。

sql.Open()返回的sql.DB对象是协程并发安全的。sql.DB的设计就是用来作为长连接使用的。不要频繁Open, Close。比较好的做法是，为每个不同的datastore建一个DB对象，保持这些对象Open。如果需要短连接，那么把DB作为参数传入function，而不要在function中Open, Close。

10.3.3 增删改数据

通过直接调用db对象的Exec()方法执行增删改的SQL语句。

func (db \*DB) Exec(query string, args ...interface{}) (Result, error)

通过db.Exec()插入数据，通过返回的err可知插入失败的原因，通过返回的结果可以进一步查询本次插入数据影响的行数RowsAffected和最后插入的id(如果数据库支持查询最后插入id)。

例：

result, err := db.Exec("INSERT INTO userinfo (username, departname, created) VALUES (?, ?, ?)","Steven","区块链教学部","2017-10-1")

预编译语句(PreparedStatement)提供了诸多好处, 因此我们在开发中尽量使用它。PreparedStatement 可以实现自定义参数的查询，PreparedStatement 通常来说,比手动拼接字符串 SQL 语句高效，PreparedStatement 可以防止SQL注入攻击。一般用Prepared Statements和Exec()完成INSERT, UPDATE, DELETE操作。

使用db对象的Prepare()方法获得预编译对象stmt，然后调用Exec()方法。

func (db \*DB) Prepare(query string) (\*Stmt, error)

例：

stmt, err := db.Prepare("INSERT userinfo SET username=?,departname=?,created=?")

result, err := stmt.Exec("Jackson", "研发部", "2017-10-1")

获取影响数据库的行数，可以根据该数值判断是否操作（插入、删除或修改）成功。

count, err := result.RowsAffected()

获得刚刚添加数据的自增ID

id, err := result.LastInsertId()

10.3.4 查询数据

数据库查询的一般步骤如下：

1.调用 db.Query()方法执行 SQL 语句, 此方法返回一个 Rows 作为查询结果；

func (db \*DB) Query(query string, args ...interface{}) (\*Rows, error)

2.将rows.Next()方法的返回值作为for循环的条件，迭代查询数据；

func (rs \*Rows) Next() bool

3.在循环中，通过 rows.Scan()方法读取每一行数据；

func (rs \*Rows) Scan(dest ...interface{}) error

4.调用db.Close()关闭查询。

也可以通过QueryRow()方法查询单条数据

func (db \*DB) QueryRow(query string, args ...interface{}) \*Row

例：

var username, departname, created string

err := db.QueryRow("SELECT username,departname,created FROM user\_info WHERE uid=?", 3).Scan(&username, &departname, &created)

通过prepare批量数据查询如下：

例：

stmt, err := db.Prepare("SELECT \* FROM user\_info WHERE uid<?")

rows, err := stmt.Query(10)

user := new(UserTable)

for rows.Next() {

err := rows.Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

continue

}

fmt.Println(\*user)

}

查询数据时需要注意以下的细节，rows.Scan()方法的参数顺序很重要，必须和查询结果的column相对应（数量和顺序都需要一致），例如 “SELECT \* From user\_info where age >=20 AND age < 30” 查询的column 顺序是 “id, name, age” 和插入操作顺序相同, 因此 rows.Scan() 也需要按照此顺序 rows.Scan(&id, &name, &age), 不然会造成数据读取的错位。

因为Go是强类型语言，所以查询数据时先定义数据类型。查询数据库中的数据存在三种可能：存在值、存在零值、未赋值NULL 三种状态，因此可以将待查询的数据类型定义为sql.NullString 、 sql.NullInt64类型等。可以通过判断Valid值来判断查询到的值是否为赋值状态还是未赋值NULL状态。

每次db.Query操作后, 都建议调用rows.Close()。因为 db.Query() 会从数据库连接池中获取一个连接, 这个底层连接在结果集(rows)未关闭前会被标记为处于繁忙状态。当遍历读到最后一条记录时，会发生一个内部EOF错误，自动调用rows.Close()。但如果出现异常，提前退出循环，rows不会关闭，连接不会回到连接池中，连接也不会关闭, 则此连接会一直被占用。因此通常使用 defer rows.Close() 来确保数据库连接可以正确放回到连接池中。

阅读源码发现rows.Close()操作是幂等操作，而一个幂等操作的特点是：其任意多次执行所产生的影响与一次执行的影响相同。所以即便对已关闭的rows再执行close()也没关系。

10.3.5 示例代码

1.案例中的表结构

CREATE TABLE user\_info (

uid int(10) NOT NULL AUTO\_INCREMENT,

username varchar(64) DEFAULT NULL,

departname varchar(64) DEFAULT NULL,

created date DEFAULT NULL,

PRIMARY KEY (uid)

) ;

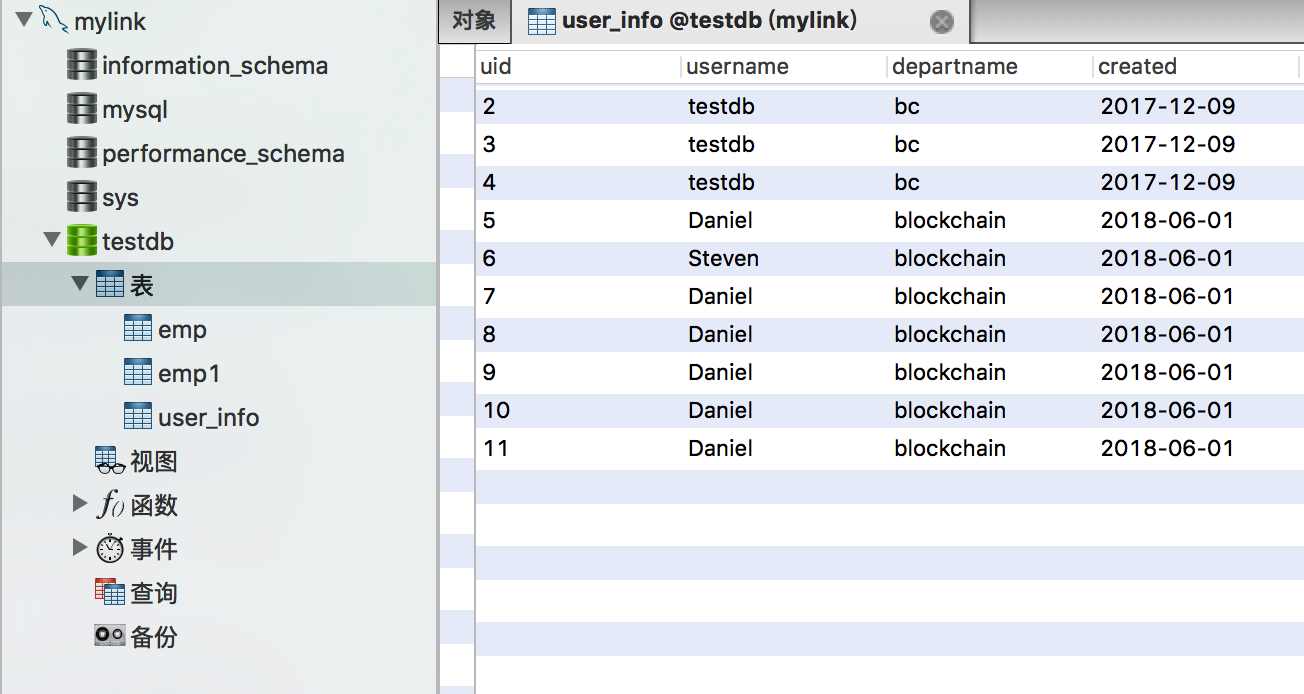


图 10.1

2.完整示例代码

package main

import (

"database/sql"

"fmt"

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

)

//定义数据库连接信息

type DbConn struct {

Dsn string //数据库驱动连接字符串

Db \*sql.DB

}

//user\_info表的映射对象

type UserTable struct {

Uid int

Username string

Department string

Created string

}

func main() {

var err error

dbConn := DbConn{

Dsn: "root:root@tcp(127.0.0.1:3306)/testdb?charset=utf8",

}

dbConn.Db, err = sql.Open("mysql", dbConn.Dsn)

if err != nil {

panic(err)

return

}

defer dbConn.Db.Close()

//1.测试封装的ExecData()方法

//dbConn.execData(&dbConn)

//2.测试封装的PreExecData()方法

//dbConn.preExecData(&dbConn)

//3.查询单行数据

//查询最后一条数据的信息

//result := dbConn.QueryRowData("select \* from user\_info where uid=(select max(uid) from user\_info)")

//fmt.Println(result)

//4.查询多行数据

//result := dbConn.QueryData("select \* from user\_info where uid<10")

//fmt.Println(len(result))

//遍历查询的结果集

//for k, v := range result {

// fmt.Println("uid：", k, v)

//}

//5.查询多行数据

result := dbConn.PreQueryData("select \* from user\_info where uid<? order by uid desc" , 10)

fmt.Println(len(result))

//遍历查询的结果集

for k, v := range result {

fmt.Println("uid：", k, v)

}

}

//测试封装的ExecData()函数

func execData(dbConn \*DbConn) {

count, id, err := dbConn.ExecData("INSERT user\_info(username , departname , created) VALUES ('Josh','business group','2018-07-3')")

//count , err := execData("UPDATE user\_info SET created='2018-06-30' WHERE uid=14")

//count , err := execData("DELETE FROM user\_info WHERE uid=10")

if err != nil {

fmt.Println(err.Error())

} else {

fmt.Println("受影响行数：", count)

fmt.Println("新添加数据的id：", id)

}

}

//测试封装的PreExecData()函数

func preExecData(dbConn \*DbConn) {

count, id, err := dbConn.PreExecData("INSERT user\_info(username , departname , created) VALUES (?,?,?)", "Jackson", "Education Department", "2017-10-8")

//count, id, err := PreExecData("Delete from user\_info WHERE uid<?", 4)

//count, id, err := PreExecData("UPDATE user\_info set departname=? WHERE departname = ?", "BC Group", "blockchain")

if err != nil {

fmt.Println(err.Error())

} else {

fmt.Println("受影响行数：", count)

fmt.Println("新添加数据的id：", id)

}

}

//1.封装增删改数据的函数，该函数直接使用DB的Exec()方法实现数据操作

func (dbConn \*DbConn) ExecData(sqlString string) (count, id int64, err error) {

result, err := dbConn.Db.Exec(sqlString)

if err != nil {

panic(err)

return

}

if id, err = result.LastInsertId(); err != nil {

panic(err)

return

}

if count, err = result.RowsAffected(); err != nil {

panic(err)

return

}

return count, id, nil

}

//2.封装增删改数据的函数，该函数使用预编译语句加Exec()方法实现增删改数据

func (dbConn \*DbConn) PreExecData(sqlString string, args ...interface{}) (count, id int64, err error) {

stmt, err := dbConn.Db.Prepare(sqlString)

defer stmt.Close()

if err != nil {

panic(err)

return

}

result, err := stmt.Exec(args ...)

if err != nil {

panic(err)

return

}

if id, err = result.LastInsertId(); err != nil {

panic(err)

return

}

if count, err = result.RowsAffected(); err != nil {

panic(err)

return

}

return count, id, nil

}

//3.查询当行数据

func (dbConn \*DbConn) QueryRowData(sqlString string) (data UserTable) {

user := new(UserTable)

err := dbConn.Db.QueryRow(sqlString).Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

return

}

return \*user

}

//4.未使用预编译，直接查询多行数据

func (dbConn \*DbConn) QueryData(sqlString string) (resultSet map[int]UserTable) {

rows, err := dbConn.Db.Query(sqlString)

defer rows.Close()

if err != nil {

panic(err)

return

}

resultSet = make(map[int]UserTable)

user := new(UserTable)

for rows.Next() {

err := rows.Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

continue

}

resultSet[user.Uid] = \*user

}

return resultSet

}

//5.使用预编译语句进行查询多行数据

func (dbConn \*DbConn) PreQueryData(sqlString string , args ...interface{}) (resultSet map[int]UserTable) {

stmt, err := dbConn.Db.Prepare(sqlString)

defer stmt.Close()

if err != nil {

panic(err)

return

}

rows, err := stmt.Query(args ...)

defer rows.Close()

if err != nil {

panic(err)

return

}

resultSet = make(map[int]UserTable)

user := new(UserTable)

for rows.Next() {

err := rows.Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

continue

}

resultSet[user.Uid] = \*user

}

return resultSet

}

//无返回值，只打印输出，用于测试

func (dbConn \*DbConn) PreQueryData2(sqlString string , args ...interface{}) {

stmt, err := dbConn.Db.Prepare(sqlString)

defer stmt.Close()

if err != nil {

panic(err)

return

}

rows, err := stmt.Query(args ...)

defer rows.Close()

if err != nil {

panic(err)

return

}

user := new(UserTable)

for rows.Next() {

err := rows.Scan(&user.Uid, &user.Username, &user.Department, &user.Created)

if err != nil {

panic(err)

continue

}

fmt.Println(\*user)

}

}

10.4 gorm库

gorm 是一个简单而强大的Go语言ORM库. 通过它可以使数据库操作非常简便。

gorm特性如下：

* 全功能ORM
* 关联（Has One, Has Many, Belongs To, Many To Many多态,）
* Callbacks（创建/保存/更新/删除/查找之前/之后）
* 预加载
* 事务
* 复合主键
* SQL 生成器
* 数据库自动迁移
* 自定义日志
* 可扩展性, 可基于 GORM 回调编写插件

10.4.1 获取gorm库

获取该gorm库的方法是在代码中直接引用该库的地址：

import github.com/jinzhu/gorm

Go Module会自动帮我们查找模块并下载，获取到可用的库之后，我们就可以尝试着使用gorm操作MySQL服务。

gorm库文档地址：https://gorm.io/zh\_CN/docs/index.html

gorm库手册地址：https://pkg.go.dev/github.com/jinzhu/gorm?tab=doc

10.4.2 gorm连接MySQL数据库

想要连接数据库，需要先导入对应数据库的驱动，GORM 已经包装了一些驱动，以便更容易的记住导入路径，可以这样导入 mysql 的驱动：

import \_ "github.com/jinzhu/gorm/dialects/mysql"

// import \_ "github.com/jinzhu/gorm/dialects/postgres"

// import \_ "github.com/jinzhu/gorm/dialects/sqlite"

// import \_ "github.com/jinzhu/gorm/dialects/mssql"

想要能正确的处理 time.Time，需要添加 parseTime 参数。

想要完全的支持 UTF-8 编码，需要修改charset=utf8 为 charset=utf8mb4。

如果想指定主机，需要使用 (). 例如:

user:password@(localhost)/dbname?charset=utf8&parseTime=True&loc=Local

例：

package main

import (

"fmt"

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

"github.com/jinzhu/gorm"

)

func main() {

db, err := gorm.Open("mysql", "root:@(localhost)/testdb?charset=utf8mb4&parseTime=True&loc=Local")

defer db.Close()

if err != nil {

fmt.Println(err)

}

// 启用Logger，显示详细日志

db.LogMode(true)

// 禁用日志记录器，不显示任何日志

// db.LogMode(false)

}

10.4.3 gorm模型

1.模型（Models）通常只是正常的 go structs、基本的 go 类型或它们的指针。 同时也支持sql.Scanner及driver.Valuer 接口（interfaces）。

模型（Model）示例:

type User struct {

gorm.Model

Name string

Age sql.NullInt64

Birthday \*time.Time

Email string `gorm:"type:varchar(100);unique\_index"`

Role string `gorm:"size:255"` // 设置字段大小为255

MemberNumber \*string `gorm:"unique;not null"` // 设置会员号（member number）唯一并且不为空

Num int `gorm:"AUTO\_INCREMENT"` // 设置 num 为自增类型

Address string `gorm:"index:addr"` // 给address字段创建名为addr的索引

IgnoreMe int `gorm:"-"` // 忽略本字段

}

结构体标记（tags）在声明模型时是可选项。gorm 支持以下标记:

|  |  |
| --- | --- |
| 结构体标记（Tag） | 描述 |
| Column | 指定列名 |
| Type | 指定列数据类型 |
| Size | 指定列大小, 默认值255 |
| PRIMARY\_KEY | 将列指定为主键 |
| UNIQUE | 将列指定为唯一 |
| DEFAULT | 指定列默认值 |
| PRECISION | 指定列精度 |
| NOT NULL | 将列指定为非 NULL |
| AUTO\_INCREMENT | 指定列是否为自增类型 |
| INDEX | 创建具有或不带名称的索引, 如果多个索引同名则创建复合索引 |
| UNIQUE\_INDEX | 和 INDEX 类似，只不过创建的是唯一索引 |
| EMBEDDED | 将结构设置为嵌入 |
| EMBEDDED\_PREFIX | 设置嵌入结构的前缀 |
| - | 忽略此字段 |
| MANY2MANY | 指定连接表 |
| FOREIGNKEY | 设置外键 |
| ASSOCIATION\_FOREIGNKEY | 设置关联外键 |
| POLYMORPHIC | 指定多态类型 |
| POLYMORPHIC\_VALUE | 指定多态值 |
| JOINTABLE\_FOREIGNKEY | 指定连接表的外键 |
| ASSOCIATION\_JOINTABLE\_FOREIGNKEY | 指定连接表的关联外键 |
| SAVE\_ASSOCIATIONS | 是否自动完成 save 的相关操作 |
| ASSOCIATION\_AUTOUPDATE | 是否自动完成 update 的相关操作 |
| ASSOCIATION\_AUTOCREATE | 是否自动完成 create 的相关操作 |
| ASSOCIATION\_SAVE\_REFERENCE | 是否自动完成引用的 save 的相关操作 |
| PRELOAD | 是否自动完成预加载的相关操作 |

gorm.Model 是一个包含了ID, CreatedAt, UpdatedAt, DeletedAt四个字段的GoLang结构体。可以将它嵌入到自定义的 Model 中。

例：

// gorm.Model 定义

type Model struct {

ID uint `gorm:"primary\_key"`

CreatedAt time.Time

UpdatedAt time.Time

DeletedAt \*time.Time

}

// 将 `ID`, `CreatedAt`, `UpdatedAt`, `DeletedAt`字段注入到`User`模型中

type User struct {

gorm.Model

Name string

}

// 不使用gorm.Model定义模型

type User struct {

ID int

Name string

}

GORM 默认会使用名为ID的字段作为表的主键。

例：

type User struct {

ID string // 名为`ID`的字段会默认作为表的主键

Name string

}

// 使用`AnimalID`作为主键

type Animal struct {

AnimalID int64 `gorm:"primary\_key"`

Name string

Age int64

}

2.表名默认就是结构体名称的复数。

例：

type User struct {} // 默认表名是 `users`

// 将 User 的表名设置为 `profiles`

func (User) TableName() string {

return "profiles"

}

func (u User) TableName() string {

if u.Role == "admin" {

return "admin\_users"

} else {

return "users"

}

}

// 禁用默认表名的复数形式，如果置为 true，则 `User` 的默认表名是 `user`

db.SingularTable(true)

// 使用User结构体创建名为`deleted\_users`的表

db.Table("deleted\_users").CreateTable(&User{})

var deleted\_users []User

db.Table("deleted\_users").Find(&deleted\_users)

// SELECT \* FROM deleted\_users;

db.Table("deleted\_users").Where("name = ?", "jinzhu").Delete()

// DELETE FROM deleted\_users WHERE name = 'jinzhu';

//可以通过定义DefaultTableNameHandler来设置默认表名的命名规则。

gorm.DefaultTableNameHandler = func (db \*gorm.DB, defaultTableName string) string {

return "prefix\_" + defaultTableName;

}

3.结构体属性的名称与数据表字段名称对应，多个单词构成的字段名称使用下划线连接，属性名称使用驼峰法与之对应。

type User struct {

ID uint // column name is `id`

Name string // column name is `name`

Birthday time.Time // column name is `birthday`

CreatedAt time.Time // column name is `created\_at`

}

// 覆盖字段名

type Animal struct {

AnimalId int64 `gorm:"column:beast\_id"` // 设置字段名为 `beast\_id`

Birthday time.Time `gorm:"column:day\_of\_the\_beast"` // 设置字段名为 `day\_of\_the\_beast`

Age int64 `gorm:"column:age\_of\_the\_beast"` // 设置字段名为 `age\_of\_the\_beast`

}

4.时间点（Timestamp）跟踪

如果模型有 CreatedAt字段，该字段的值将会是初次创建记录的时间。

db.Create(&user) // `CreatedAt`将会是当前时间

// 可以使用`Update`方法来改变`CreateAt`的值

db.Model(&user).Update("CreatedAt", time.Now())

如果模型有UpdatedAt字段，该字段的值将会是每次更新记录的时间。

db.Save(&user) // `UpdatedAt`将会是当前时间

db.Model(&user).Update("name", "jinzhu") // `UpdatedAt`将会是当前时间

如果模型有DeletedAt字段，调用Delete删除该记录时，将会设置DeletedAt字段为当前时间，而不是直接将记录从数据库中删除。

10.4.3 gorm读写数据

1.创建新数据

例：

package main

import (

"fmt"

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

"github.com/jinzhu/gorm"

)

type Product struct {

gorm.Model

Code string

Price uint

Style string

}

func main() {

db, err := gorm.Open("mysql", "root:@(localhost)/testdb?charset=utf8mb4&parseTime=True&loc=Local")

defer db.Close()

if err != nil {

fmt.Println(err)

}

db.AutoMigrate(&Product{})

db.LogMode(true)

addNew(db)

}

func addNew(db \*gorm.DB) {

product := Product{Code: "17736", Price: 1000}

fmt.Println(product) //{{0 0001-01-01 00:00:00 +0000 UTC 0001-01-01 00:00:00 +0000 UTC <nil>} 17736 1000 }

//检查主键是否为空，为空返回true,不为空返回false

if db.NewRecord(product) {

//创建数据，写入数据库

db.Create(&product)

if db.NewRecord(product) {

fmt.Println("添加失败，请查看日志")

return

} else {

fmt.Println(product) //{{6 2020-05-04 20:36:51.3467893 +0800 CST m=+0.027997901 2020-05-04 20:36:51.3467893 +0800 CST m=+0.027997901 <nil>} 17736 1000 }

fmt.Println(product.ID) //6

}

} else {

fmt.Println("主键不为空")

return

}

}

可以在gorm tag中定义默认值，然后插入SQL将忽略具有默认值的这些字段，并且其值为空，并且在将记录插入数据库后，gorm将从数据库加载这些字段的值。

type Product struct {

gorm.Model

Code string

Price uint

Style string `gorm:"default:'galeone'"`

}

product := Product{Code: "17736", Price: 1000}

db.Create(&product )

//INSERT INTO `products` (`created\_at`,`updated\_at`,`deleted\_at`,`code`,`price`) VALUES ('2020-05-05 00:14:35','2020-05-05 00:14:35',NULL,'17736',1000)

// SELECT `style` FROM `products` WHERE (id = 4); // 返回主键为 4

// product .Style => 'galeone'

2.查询

例：

product := Product{}

// 获取第一条记录，按主键排序

db.First(&product)

// SELECT \* FROM products ORDER BY id LIMIT 1;

// 获取最后一条记录，按主键排序

db.Last(&product)

// SELECT \* FROM products ORDER BY id DESC LIMIT 1;

// 获取所有记录

db.Find(&product)

// SELECT \* FROM products;

// 使用主键获取记录

db.First(&product, 10)

// SELECT \* FROM products WHERE id = 10;

type User struct {

gorm.Model

Name string

Age sql.NullInt64

Birthday \*time.Time

Email string `gorm:"type:varchar(100);unique\_index"`

Role string `gorm:"size:255"` // 设置字段大小为255

MemberNumber \*string `gorm:"unique;not null"` // 设置会员号（member number）唯一并且不为空

Num int `gorm:"AUTO\_INCREMENT"` // 设置 num 为自增类型

Address string `gorm:"index:addr"` // 给address字段创建名为addr的索引

IgnoreMe int `gorm:"-"` // 忽略本字段

}

users := User{}

// 获取第一个匹配记录

db.Where("name = ?", "jinzhu").First(&users)

// SELECT \* FROM users WHERE name = 'jinzhu' limit 1;

// 获取所有匹配记录

db.Where("name = ?", "jinzhu").Find(&users)

// SELECT \* FROM users WHERE name = 'jinzhu';

db.Where("name <> ?", "jinzhu").Find(&users)

// SELECT \* FROM users WHERE name <> 'jinzhu';

db.Where("name in (?)", []string{"jinzhu", "jinzhu 2"}).Find(&users)

//SELECT \* FROM `users` WHERE name in ('jinzhu','jinzhu 2')

db.Where("name LIKE ?", "%jin%").Find(&users)

//SELECT \* FROM `users` WHERE name LIKE '%jin%'

db.Where("name = ? AND age >= ?", "jinzhu", "22").Find(&users)

//SELECT \* FROM `users` WHEREname = 'jinzhu' AND age >= '22'

//注意：当使用struct查询时，GORM将只查询那些具有值的字段

db.Where(&User{Name: "jinzhu", Age: 20}).First(&user)

// SELECT \* FROM users WHERE name = "jinzhu" AND age = 20 LIMIT 1;

db.Where(map[string]interface{}{"name": "jinzhu", "age": 20}).Find(&users)

// SELECT \* FROM users WHERE name = "jinzhu" AND age = 20;

// 主键的Slice

db.Where([]int64{20, 21, 22}).Find(&users)

// SELECT \* FROM users WHERE id IN (20, 21, 22);

db.Not("name", "jinzhu").First(&user)

// SELECT \* FROM users WHERE name <> "jinzhu" LIMIT 1;

db.Not("name", []string{"jinzhu", "jinzhu 2"}).Find(&users)

// SELECT \* FROM users WHERE name NOT IN ("jinzhu", "jinzhu 2");

// 不在主键切片中

db.Not([]int64{1,2,3}).First(&user)

// SELECT \* FROM users WHERE id NOT IN (1,2,3);

db.Not([]int64{}).First(&user)

// SELECT \* FROM users;

// 普通 SQL

db.Not("name = ?", "jinzhu").First(&user)

// SELECT \* FROM users WHERE NOT(name = "jinzhu");

db.Not(User{Name: "jinzhu"}).First(&user)

// SELECT \* FROM users WHERE name <> "jinzhu";

db.Where("role = ?", "admin").Or("role = ?", "super\_admin").Find(&users)

// SELECT \* FROM users WHERE role = 'admin' OR role = 'super\_admin';

db.Where("name = 'jinzhu'").Or(User{Name: "jinzhu 2"}).Find(&users)

// SELECT \* FROM users WHERE name = 'jinzhu' OR name = 'jinzhu 2';

db.Where("name = 'jinzhu'").Or(map[string]interface{}{"name": "jinzhu 2"}).Find(&users)

//允许指定要从数据库检索的字段，默认情况下，将选择所有字段;

db.Select("name, age").Find(&users)

// SELECT name, age FROM users;

db.Select([]string{"name", "age"}).Find(&users)

// SELECT name, age FROM users;

db.Table("users").Select("COALESCE(age,?)", 42).Rows()

// SELECT COALESCE(age,'42') FROM users;

//在从数据库检索记录时指定顺序，将重排序设置为true以覆盖定义的条件

db.Order("age desc, name").Find(&users)

// SELECT \* FROM users ORDER BY age desc, name;

3.更新

例：

//Save将包括执行更新SQL时的所有字段，即使它没有更改

db.First(&user)

user.Name = "jinzhu 2"

user.Age = 100

db.Save(&user)

// UPDATE users SET name='jinzhu 2', age=100, birthday='2016-01-01', updated\_at = '2013-11-17 21:34:10' WHERE id=111;

//如果只想更新更改的字段，可以使用Update, Updates

// 更新单个属性（如果更改）

db.Model(&user).Update("name", "hello")

// UPDATE users SET name='hello', updated\_at='2013-11-17 21:34:10' WHERE id=111;

// 使用组合条件更新单个属性

db.Model(&user).Where("active = ?", true).Update("name", "hello")

// UPDATE users SET name='hello', updated\_at='2013-11-17 21:34:10' WHERE id=111 AND active=true;

// 使用`map`更新多个属性，只会更新这些更改的字段

db.Model(&user).Updates(map[string]interface{}{"name": "hello", "age": 18, "actived": false})

// UPDATE users SET name='hello', age=18, actived=false, updated\_at='2013-11-17 21:34:10' WHERE id=111;

// 使用`struct`更新多个属性，只会更新这些更改的和非空白字段

db.Model(&user).Updates(User{Name: "hello", Age: 18})

// UPDATE users SET name='hello', age=18, updated\_at = '2013-11-17 21:34:10' WHERE id = 111;

// 警告:当使用struct更新时，FORM将仅更新具有非空值的字段

// 对于下面的更新，什么都不会更新为""，0，false是其类型的空白值

db.Model(&user).Updates(User{Name: "", Age: 0, Actived: false})

//如果您只想在更新时更新或忽略某些字段，可以使用Select, Omit

db.Model(&user).Select("name").Updates(map[string]interface{}{"name": "hello", "age": 18, "actived": false})

// UPDATE users SET name='hello', updated\_at='2013-11-17 21:34:10' WHERE id=111;

db.Model(&user).Omit("name").Updates(map[string]interface{}{"name": "hello", "age": 18, "actived": false})

// UPDATE users SET age=18, actived=false, updated\_at='2013-11-17 21:34:10' WHERE id=111;

//更新更改字段但不进行Callbacks

以上更新操作将执行模型的BeforeUpdate, AfterUpdate方法，更新其UpdatedAt时间戳，在更新时保存它的Associations，如果不想调用它们，可以使用UpdateColumn, UpdateColumns

// 更新单个属性，类似于`Update`

db.Model(&user).UpdateColumn("name", "hello")

// UPDATE users SET name='hello' WHERE id = 111;

// 更新多个属性，与“更新”类似

db.Model(&user).UpdateColumns(User{Name: "hello", Age: 18})

// UPDATE users SET name='hello', age=18 WHERE id = 111;

//Callbacks在批量更新时不会运行

db.Table("users").Where("id IN (?)", []int{10, 11}).Updates(map[string]interface{}{"name": "hello", "age": 18})

// UPDATE users SET name='hello', age=18 WHERE id IN (10, 11);

// 使用struct更新仅适用于非零值，或使用map[string]interface{}

db.Model(User{}).Updates(User{Name: "hello", Age: 18})

// UPDATE users SET name='hello', age=18;

// 使用`RowsAffected`获取更新记录计数

db.Model(User{}).Updates(User{Name: "hello", Age: 18}).RowsAffected

4. 删除/软删除

例：

//注意！！！删除记录时，请确保主键字段有值，GORM 会通过主键去删除记录，如果主键为空，GORM 会删除该 model 的所有记录。

// 删除存在的记录

db.Delete(&email)

// DELETE from emails where id=10;

// 为Delete语句添加额外的SQL选项

db.Set("gorm:delete\_option", "OPTION (OPTIMIZE FOR UNKNOWN)").Delete(&email)

// DELETE from emails where id=10 OPTION (OPTIMIZE FOR UNKNOWN);

//删除所有匹配记录

db.Where("email LIKE ?", "%jinzhu%").Delete(Email{})

// DELETE from emails where email LIKE "%jinhu%";

db.Delete(Email{}, "email LIKE ?", "%jinzhu%")

// DELETE from emails where email LIKE "%jinhu%";

//如果模型有DeletedAt字段，它将自动获得软删除功能！ 那么在调用Delete时不会从数据库中永久删除，而是只将字段DeletedAt的值设置为当前时间。

db.Delete(&user)

// UPDATE users SET deleted\_at="2013-10-29 10:23" WHERE id = 111;

// 批量删除

db.Where("age = ?", 20).Delete(&User{})

// UPDATE users SET deleted\_at="2013-10-29 10:23" WHERE age = 20;

// 软删除的记录将在查询时被忽略

db.Where("age = 20").Find(&user)

// SELECT \* FROM users WHERE age = 20 AND deleted\_at IS NULL;

// 使用Unscoped查找软删除的记录

db.Unscoped().Where("age = 20").Find(&users)

// SELECT \* FROM users WHERE age = 20;

// 使用Unscoped永久删除记录

db.Unscoped().Delete(&order)

// DELETE FROM orders WHERE id=10;

10.5 小结

本章学习Go语言操作数据库方面的内容。总有人说CURD的程序员不算程序员，但是实际上，又有几个软件是与CURD无关的呢。可以说CURD是每个程序员必备的技能，也是使用频率最高的技能。本章中的gorm库也只是列举了部分的使用方法，更为丰富的功能及版本更新请关注作者的github，最好能star一下，方便能随时找到。当然如果有兴趣，也可以加入这个开源库的维护团队中，为开源贡献自己的一分力量。

第十一章 Go语言操作缓存

本章重点为大家介绍如下的内容：

* Go语言操作Memcached
* Go语言操作redis
* GroupCache简介

11.1 Go语言操作Memcached

Memcached是一个自由开源的，高性能，分布式内存对象缓存系统。

Memcached是以LiveJournal旗下Danga Interactive公司的Brad Fitzpatric为首开发的一款软件。现在已成为mixi、hatena、Facebook、Vox、LiveJournal等众多服务中提高Web应用扩展性的重要因素。

Memcached是一种基于内存的key-value存储，用来存储小块的任意数据（字符串、对象）。这些数据可以是数据库调用、API调用或者是页面渲染的结果。

Memcached简洁而强大。它的简洁设计便于快速开发，减轻开发难度，解决了大数据量缓存的很多问题。它的API兼容大部分流行的开发语言。

本质上，它是一个简洁的key-value存储系统。

一般的使用目的是，通过缓存数据库查询结果，减少数据库访问次数，以提高动态Web应用的速度、提高可扩展性。

memcached作为高速运行的分布式缓存服务器，具有以下的特点。

协议简单

基于libevent的事件处理

内置内存存储方式

memcached不互相通信的分布式

11.1.1 安装MemCached驱动

获取memcached第三方库的方法是在代码中直接引用该库的地址：

import github.com/bradfitz/gomemcache/memcache

Go Module会自动帮我们查找模块并下载，获取到可用的库之后，我们就可以尝试着连接memcached服务器。

例：

package main

import (

"fmt"

"github.com/bradfitz/gomemcache/memcache" //此行代码文中后面的例子不会给出，请注意

)

func main() {

mc := memcache.New("127.0.0.1:11211")

if mc == nil {

fmt.Println("memcache New failed")

return

}

fmt.Println("memcache New success")

}

通过以上代码，我们就实现了连接memcached服务器的功能，连接成功之后，我们就可以进行一系列的基本操作。通过测试我们发现，memcache.New方法的返回值是一个\*Clien结构体，所以不会存在返回值为nil的情况。我们也无法根据返回的内容进行判断连接是否成功，只有在具体操作的时候，才会返回连接超时的信息，所以此处的判断代码只有象征意义，并不能说明连接是否成功。

11.1.2 Go语言操作Memcached

Go语言Memcache库set操作

使用set操作命令格式如下：

func (c \*Client) Set(item \*Item) error

参数：item是一个item结构体

item结构体的结构如下：

type Item struct {

// Item的key(最大250字节).

Key [string](https://godoc.org/builtin" \l "string)

// 需要存储的值

Value [][byte](https://godoc.org/builtin" \l "byte)

//服务标识位，取决于应用的不同。.

Flags [uint32](https://godoc.org/builtin" \l "uint32)

// 缓存过期时间。（单位：秒）

//0代表永不过期

Expiration [int32](https://godoc.org/builtin" \l "int32)

// contains filtered or unexported fields

}

例：

func main() {

mc := memcache.New("127.0.0.1:11211")

err := mc.Set(&memcache.Item{Key: "foo", Value: []byte("my value")})

if err != nil {

fmt.Println(err)

return

}

err = mc.Set(&memcache.Item{Key: "foo", Value: []byte("my value1")})

fmt.Println("set is success")

}

上面的例子中我们需要注意两个细节。

1.value是使用[]byte存储的，所以在获取数据的时候，需要使用类型转换将[]byte转换成string才可以正常使用

2.set操作是强制覆盖的，重名情况下不会有任何提示，直接将内容重新写入。

11.1.3 Go语言Memcached库get操作

使用get操作命令格式如下：

func (c \*[Client](https://godoc.org/github.com/bradfitz/gomemcache/memcache" \l "Client)) Get(key [string](https://godoc.org/builtin" \l "string)) (item \*[Item](https://godoc.org/github.com/bradfitz/gomemcache/memcache" \l "Item), err [error](https://godoc.org/builtin" \l "error))

通过key获取item。如果获取的内容不存在，返回” memcache: cache miss”。key的长度不能大于250位。

例：

func main() {

mc := memcache.New("127.0.0.1:11211")

err := mc.Set(&memcache.Item{Key: "foo", Value: []byte("my value")})

if err != nil {

fmt.Println(err)

return

}

err = mc.Set(&memcache.Item{Key: "foo", Value: []byte("my value1")})

it, err := mc.Get("foo")

if err != nil {

fmt.Println(err)

return

}

fmt.Println(string(it.Value)) //my value1

\_, err = mc.Get("t")

fmt.Println(err) //memcache: cache miss

}

通过上例，验证了同键名情况下，值会被直接覆盖。Get取到的数据是[]byte类型，需要使用string()进行类型转换。

11.1.4 Go语言Memcached库add操作

使用add操作命令格式如下：

func (c \*[Client](https://godoc.org/github.com/bradfitz/gomemcache/memcache" \l "Client)) Add(item \*[Item](https://godoc.org/github.com/bradfitz/gomemcache/memcache" \l "Item)) [error](https://godoc.org/builtin" \l "error)

add用于将value存储在指定的key中，如果add的key已经存在，则不会更新数据（过期的key会更新），之前的值不变，并且获得响应NOT\_STORED

例：

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

if mc == nil {

fmt.Println("memcache New failed")

return

}

mc.Add(&memcache.Item{Key: "foo3", Value: []byte("bluegogo1")})

it, err := mc.Get("foo3")

if err != nil {

fmt.Println("Add failed")

} else {

if string(it.Key) == "foo3" {

fmt.Println("Add value is ", string(it.Value))// Add value is bluegogo1

} else {

fmt.Println("Get failed")

}

}

}

11.1.5 Go语言Memcached库 replace操作

使用replace操作命令格式如下：

func (c \*[Client](https://godoc.org/github.com/bradfitz/gomemcache/memcache" \l "Client)) Replace(item \*[Item](https://godoc.org/github.com/bradfitz/gomemcache/memcache" \l "Item)) [error](https://godoc.org/builtin" \l "error)

replace 命令用于替换已存在的 key(键) 的 value(数据值)。

如果 key 不存在，则替换失败，并且您将获得响应 NOT\_STORED。

例：

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

if mc == nil {

fmt.Println("memcache New failed")

return

}

mc.Set(&memcache.Item{Key: "foo2", Value: []byte("my value1")})

mc.Replace(&memcache.Item{Key: "foo2", Value: []byte("mobike")})

it, err := mc.Get("foo2")

if err != nil {

fmt.Println("foo2 failed")

} else {

if string(it.Key) == "foo2" {

fmt.Println("foo2 value is ", string(it.Value)) //foo2 value is mobike

} else {

fmt.Println("Get failed")

}

}

}

11.1.6 Go语言Memcached库 delete操作

使用delete操作命令格式如下：

func (c \*Client) Delete(key string) error

delete 命令用于删除已存在的 key(键)。

例：

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

if mc == nil {

fmt.Println("memcache New failed")

return

}

errd := mc.Delete("foo")

if errd != nil {

fmt.Println("Delete failed:", errd.Error())

}

\_, err := mc.Get("foo")

if err != nil {

fmt.Println("delete success")

}

}

11.1.7 Go语言Memcached库incrby操作

使用incrby操作命令格式如下：

func (c \*[Client](https://godoc.org/github.com/bradfitz/gomemcache/memcache" \l "Client)) Increment(key [string](https://godoc.org/builtin" \l "string), delta [uint64](https://godoc.org/builtin" \l "uint64)) (newValue [uint64](https://godoc.org/builtin" \l "uint64), err [error](https://godoc.org/builtin" \l "error))

incr命令用于对已存在的 key(键) 的数字值进行自增或自减操作。

incr 命令操作的数据必须是十进制的32位无符号整数。

如果 key 不存在返回 NOT\_FOUND，如果键的值不为数字，则返回 CLIENT\_ERROR，其他错误返回 ERROR。

例：

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

err := mc.Set(&memcache.Item{Key: "aaa", Value: []byte("1")})

if err != nil {

fmt.Println("Set failed :", err.Error())

}

it, err := mc.Get("foo")

if err != nil {

fmt.Println("Get failed ", err.Error())

} else {

fmt.Println("src value is:", it.Value)

}

value, err := mc.Increment("aaa", 7)

if err != nil {

fmt.Println("Increment failed")

} else {

fmt.Println("after increment the value is :", value)

}

}

11.1.8 Go语言Memcached库decrby操作

使用decrby操作命令格式如下：

func (c \*Client) Decrement(key string, delta uint64) (newValue uint64, err error)

decr 命令用于对已存在的 key(键) 的数字值进行自增或自减操作。

decr 命令操作的数据必须是十进制的32位无符号整数。

如果 key 不存在返回 NOT\_FOUND，如果键的值不为数字，则返回 CLIENT\_ERROR，其他错误返回 ERROR。

例：

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

value, err := mc.Decrement("aaa", 4)

if err != nil {

fmt.Println("Decrement failed", err.Error())

} else {

fmt.Println("after decrement the value is ", value)

}

}

11.2 GO语言操作Redis

Redis是一个开源的，在内存中存储数据的缓存系统。可以用作数据库的缓存和消息代理。它支持的数据结构包括：字符串、哈希、列表、set集合、有序set集合、位图等类型。Redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

Redis支持主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。这使得Redis可执行单层树复制。存盘可以有意无意的对数据进行写操作。由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时，可订阅一个频道并接收主服务器完整的消息发布记录。同步对读取操作的可扩展性和数据冗余很有帮助。

11.2.1 安装redigo

获取redigo第三方库的方法是在代码中直接引用该库的地址：

import github.com/gomodule/redigo/redis

Go Module会自动帮我们查找模块并下载，获取到可用的库之后，我们就可以尝试着连接redis服务器。

例：

package main

import (

"fmt"

"github.com/garyburd/redigo/redis" //此行代码文中后面的例子不会给出，请注意

)

func main() {

host := "127.0.0.1"

port := "6379"

protocol := "tcp"

redis, err := redis.Dial(protocol, host+":"+port)

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

fmt.Println("Connect to redis succeed")

defer redis.Close()

}

通过代码，我们就实现了连接redis服务器的功能，连接成功之后，我们就可以进行一系列的基本操作。接下来我们会根据redigo提供的各种功能，详细的为读者介绍，如何与redis进行交互。

11.2.2 Go语言Redis基本操作

使用Redigo操作数据的方式非常简单，命令格式如下：

Do(commandName string, args ...interface{}) (reply interface{}, err error)

参数：commandName是命令名称，包括“GET”，“SET“，“DEL”等

参数：args是一个字符串集，可以同时操作多个信息。

例：

func main() {

redigo, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

v, err := redigo.Do("SET", "color", "red")

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出ok

v, err = redis.String(redigo.Do("GET", "color"))

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出red

}

我们使用上面的例子了解到，对于字符串类型的数据，如何操作。我们使用do方法，使用“SET”命令，将数据保存到redis中，在下面使用”GET”命令，将存储的数据取出来

例：

func main() {

redigo, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

v, err := redigo.Do("SET", "color", "red")

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出ok

v, err = redigo.Do("DEL", "color")

if err != nil {

fmt.Println(err)

return

}

v, err = redis.String(redigo.Do("GET", "color"))

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出：redigo: nil returned

}

上面例子中我们使用了DEL命令将已经存入的数据删除后，再获取数据，得到的结果是nil returned，说明我们已经把刚才的值删除成功了。

例：

package main

import (

"fmt"

"github.com/garyburd/redigo/redis"

)

func main() {

redigo, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

redigo.Do("lpush", "redlist", "qqq")

redigo.Do("lpush", "redlist", "www")

redigo.Do("lpush", "redlist", "eee")

values, \_ := redis.Values(redigo.Do("lrange", "redlist", "0", "100"))

for \_, v := range values {

fmt.Println(string(v.([]byte)))

}

//输出：

// eee

// www

// qqq

}

列表的操作与普通字符串的操作一致，都是使用Do方法，由于使用的是“lpush”操作，所以我们在遍历的时候发现输出的顺序与我们写入的不一致。如果改成”rpush”，顺序就会一致了。读者可以自行尝试一下使用”rpush”操作查看一下效果。

例：

func main() {

redigo, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

redigo.Do("SADD", "tt", "1")

redigo.Do("SADD", "tt", "2")

redigo.Do("SADD", "tt", "3")

redigo.Do("SADD", "tt", "1")

values, \_ := redis.Values(redigo.Do("SMEMBERS", "tt"))

for \_, v := range values {

fmt.Println(string(v.([]byte)))

}

//输出：

// 1

// 2.

// 3

}

对于set类型的数据，我们使用”SADD”命令可以将数据写入。

通过上面几个例子。我们看到redigo操作redis的方式与我们用命令行的方式几乎完全一样，读者在熟悉基本redis操作的情况下，可以非常容易上手。

在大量数据需要一次性操作完成的时候，可以使用Redis为我们提供的管道操作。管道操作可以将一组操作合并成一次操作，可以减少时间消耗。

我们先看一下基本的管道操作的例子。

例：

func main() {

c, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

c.Send("SET", "foo", "bar")

c.Send("GET", "foo")

c.Flush()

s, err := c.Receive() // reply from SET

fmt.Println(s) //输出:OK

v, err := c.Receive() // reply from GET

fmt.Printf("%s", v) //输出：bar

}

上面的例子我们可以了解管道的基本用法，现在我们来比较下管理操作的速度差异。

我们先看一下不使用管道的写入操作。

例：

func main() {

c, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

start := time.Now().UnixNano() //获取开始时间戳

for i := 0; i < 100000; i++ {

c.Do("SET", "foo"+string(i), "bar"+string(i))

}

end := time.Now().UnixNano() //获取结束时间戳

fmt.Println(end - start) //输出4896866986纳秒=4.896867秒

}

我们使用循环向redis写入100000条数据，不使用管道的情况下使用接近5秒的时间。

例：

func main() {

c, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

start := time.Now().UnixNano() //获取开始时间戳

for i := 0; i < 100000; i++ {

c.Send("SET", "foo"+string(i), "bar"+string(i))

}

end := time.Now().UnixNano() //获取结束时间戳

fmt.Println(end - start) //输出141727169纳秒=0.1417272秒

}

我们使用管道方式写入100000条数据，耗时仅仅0.14秒多一点点。时间效率之间的差异，由此可见一斑。

11.2.3 Go语言操作Redis连接池管理

redigo提供了池来管理连接。为了避免每次操作redis时，建立连接，用完后再关闭，会造成大量的连接处于TIME\_WAIT状态，我们建议开发者在使用的时候，通过池来管理连接。

例：

var RedisClient \*redis.Pool

func init() {

// 建立连接池

RedisClient = &redis.Pool{

//最大空闲连接数

MaxIdle: 1,

//最大激活连接数

MaxActive: 10,

//最大的空闲连接等待时间，超过此时间后，空闲连接将被关闭

IdleTimeout: 180 \* time.Second,

Dial: func() (redis.Conn, error) {

c, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

return nil, err

}

return c, nil

},

}

}

func main() {

// 从池里获取连接

rc := RedisClient.Get()

v, err := rc.Do("SET", "color", "red")

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出ok

v, err = redis.String(rc.Do("GET", "color"))

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出red

// 用完后将连接放回连接池

defer rc.Close()

}

11.3 Go语言操groupcache

11.3.1 groupcache简介

groupcache 是使用 Go 语言编写的缓存及缓存过滤库，作为 memcached 许多场景下的替代版本。

groupcache 与 memcached 的相似之处：通过 key 分片，并且通过 key 来查询响应的 peer。

groupcache 与 memcached 的不同之处：

groupcache 不需要对服务器进行单独的设置，这将大幅度减少部署和配置的工作量。groupcache 既是客户端库也是服务器库，并连接到自己的 peer 上。

groupcache 具有缓存过滤机制。众所周知，在 memcached 出现“Sorry，cache miss（缓存丢失）”时，经常会因为不受控制用户数量的请求而导致数据库（或者其它组件）产生“惊群效应（thundering herd）”；groupcache 会协调缓存填充，只会将重复调用中的一个放于缓存，而处理结果将发送给所有相同的调用者。

groupcache 不支持多个版本的值。如果“foo”键对应的值是“bar”，那么键“foo”的值永远都是“bar”。这里既没有缓存的有效期，也没有明确的缓存回收机制，因此同样也没有 CAS 或者 Increment/Decrement。

基于上一点的改变，groupcache 就具备了自动备份“超热”项进行多重处理，这就避免了 memcached 中对某些键值过量访问而造成所在机器 CPU 或者 NIC 过载。

groupcache 当下只支持Go

11.3.2 groupcache运行机制

简而言之，groupcache查找一个Get（“foo”）的过程类似下面的情景（机器#5上，它是运行相同代码N台机器集中的一台）：

1. key“foo”的值是否会因为“过热”而储存在本地内存，如果是，就直接使用

2. key“foo”的值是否会因为peer #5是其拥有者而储存在本地内存，如果是，就直接使用

3. 首先确定key “fool”是否归属自己N个机器集合的peer中，如果是，就直接加载。如果有其它的调用者介入（通过相同的进程或者是peer的RPC请求，这些请求将会被阻塞，而处理结束后，他们将直接获得相同的结果）。如果不是，将key的所有者RPC到响应的peer。如果RPC失败，那么直接在本地加载（仍然通过备份来应对负载）。

目前groupcache已经在dl.Google.com、Blogger、Google Code、Google Fiber、Google生产监视系统等项目中投入使用。

11.3.3 使用groupcache构建服务端

例：groupcache

package main

import (

"fmt"

"io/ioutil"

"log"

"net/http"

"os"

"github.com/golang/groupcache"

)

var (

peers\_addrs = []string{"http://127.0.0.1:8001", "http://127.0.0.1:8002", "http://127.0.0.1:8003"}

)

func main() {

// 验证启动参数

if len(os.Args) <= 1 {

fmt.Println("invalid hosts")

os.Exit(1)

}

local\_addr := os.Args[1]

// 创建HTTP池

peers := groupcache.NewHTTPPool("http://" + local\_addr)

peers.Set(peers\_addrs...)

// 创建Group

var image\_cache = groupcache.NewGroup("cache", 8<<30, groupcache.GetterFunc(

// 无法命中缓存的处理方式

func(ctx groupcache.Context, key string, dest groupcache.Sink) error {

// 从文件中读取内容

result, err := ioutil.ReadFile(key)

if err != nil {

fmt.Printf("read file error %s.\n", err.Error())

message := "{'status':'0','msg':'" + err.Error() + "'}"

dest.SetBytes([]byte(message))

return nil

}

fmt.Printf("asking for %s from local file system\n", key)

// 设置缓存内容

dest.SetBytes([]byte(result))

return nil

}))

// 创建请求接口

http.HandleFunc("/cache", func(rw http.ResponseWriter, r \*http.Request) {

// 响应cache请求

var data []byte

// 获取参数

k := r.URL.Query().Get("id")

fmt.Printf("user get %s from groupcache\n", k)

// 根据key获取缓存信息

image\_cache.Get(nil, k, groupcache.AllocatingByteSliceSink(&data))

// 提供返回值

rw.Write([]byte(data))

})

log.Fatal(http.ListenAndServe(local\_addr, nil))

}

以上代码在命令行中执行以后，可以使用浏览器通过http://127.0.0.1:8001/cache访问

使用：http://127.0.0.1:8001/image?id=1.ini

第一次请求会看到控制台输出user get 1.ini from groupcache，同时会输出asking for 1.ini from local file system。由于是第一次访问，缓存中没有装载数据，所以会到本地去寻找要加载的文件，将文件内容装载到缓存中，显示到页面中。

第二次请求访问http://127.0.0.1:8001/image?id=1.ini1

控制台输出read file error open 1.ini1: no such file or directory.提示没有找到1.ini1这个文件。大家在测试代码的时候，需要提前创建文件，以防止无法找到。

对于1.ini由于已经存在了。在后续的多次访问中，都会显示从groupcache中获取。由于groupcache设计的时候就只允许添加，所以我们目前可以考虑将配置类信息，和持续不变的信息放到groupcache中。替代部分Memcached的功能。本书仅给出一个简单的示例，具体的使用还需要读者在实际工作中去寻找合适的场景。

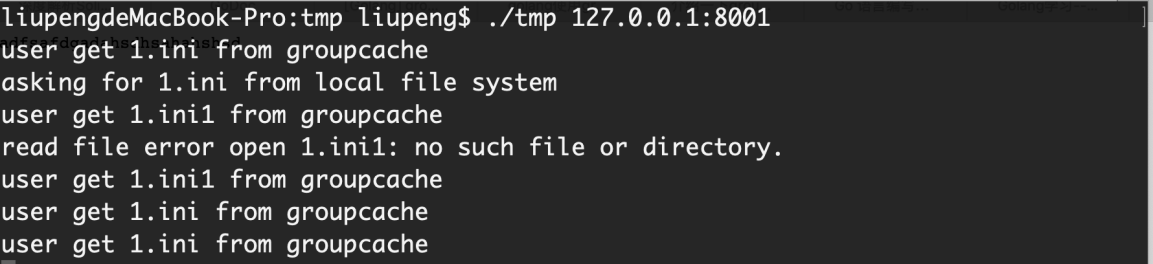


图11.1

11.4 小结

本章学习Go语言操作缓存方面的内容。软件开发过程中，把使用频率较高的数据或者需要在某段时间内长驻的数据放到缓存中，提高数据存取的效率，目前redis使用的频率相对较高，Memcached用得已经相对较少了，但是依旧需要了解，万一有时候用得上呢。

第十二章 Go语言并发编程

本章重点为大家介绍如下的内容：

* 什么是并发与并行
* 进程和线程
* 轻量级线程——coroutine
* go语言中的协程——goroutine
* 多个goroutine间通信的通道channel
* time包中跟通道相关的函数
* 缓冲通道和定向通道
* select分支语句
* sync包中的WaitGroup同步等待组
* 共享数据安全问题及sync包中的Mutex互斥锁
* sync包中的RWMutex读写互斥锁及Cond条件变量

12.1 并发与并行

12.1.1 Go是并发语言，而不是并行语言

在讨论如何在Go中进行并发处理之前，我们首先必须了解什么是并发，以及它与并行性有什么不同。Go是并发语言，而不是并行语言

并发性Concurrency是指把任务在不同的时间点交给处理器进行处理。在同一个时间点，任务不会同时运行。并发性是同时处理许多事情的能力。

举个例子，打电话和吃饭。吃饭时，电话来了，需要停止吃饭去接电话。电话接完后回来继续吃饭，这个过程是并发执行。

并行性parallelism是把一个任务分配给每一个处理器独立完成。在同一个时间点，任务一定是同时运行。并行就是同时做很多事情。这听起来可能与并发类似，但实际上是不同的。

用同样打电话和吃饭的例子去理解。吃饭时，电话来了，边吃饭边接电话，这个过程是并行执行。这就是所谓的并行性(parallelism)。

Go在Gomaxprocs数量与任务数量相等时，可以做到并行执行，但是一般情况下都是并发执行。

并发性和并行性——一种技术上的观点。

假设我们正在编写一个web浏览器。web浏览器有各种组件，其中两个是web页面呈现区域和从internet下载文件的下载器。当这个浏览器运行在单核处理器中时，处理器将在浏览器的两个组件之间进行上下文切换。它可能会下载一个文件一段时间，然后它可能会切换到呈现用户请求的网页的html，这就是所谓的并发性。并发进程从不同的时间点开始，它们的执行周期重叠。在这种情况下，下载和呈现从不同的时间点开始，它们的执行重叠。

假设同一浏览器运行在多核处理器上。在这种情况下，文件下载组件和HTML呈现组件可能同时在不同的内核中运行，这就是所谓的并行性。并行性Parallelism不会总是导致更快的执行时间。这是因为并行运行的组件可能需要相互通信。例如，在我们的浏览器中，当文件下载完成时，应该将其传递给用户，比如使用弹出窗口。这种通信发生在负责下载的组件和负责呈现用户界面的组件之间。当组件在多个内核中并行运行时，这种通信开销很高。因此，并行程序并不总是导致更快的执行时间!这种通信开销在并发concurrent 系统中很低。在单CPU上，是os代码强制把一个进程或者线程挂起，换成另外一个，所以实际上是并发的，只是“概念上的并行”。在现在的多核的cpu上，线程可能是“真正并行的”。

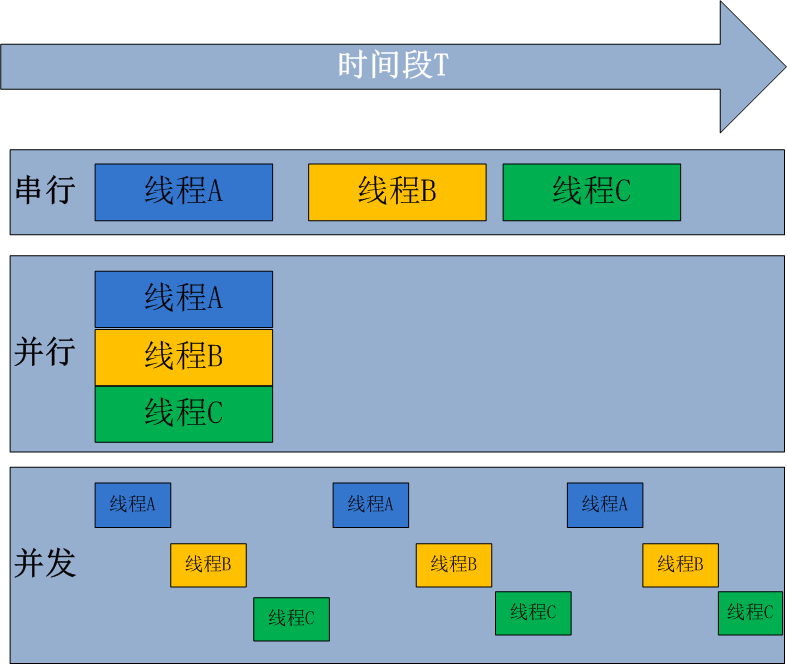


图 12.1

12.1.2 什么是并发？

并发指在同一时间内可以执行多个任务。并发编程含义广泛，可以是多线程编程，也可以是多进程编程，还可以是编写分布式程序。并发编程的思想来自于多任务操作系统，它允许同时运行多个程序。

早期单用户操作系统，任务是一个接一个运行，单个任务的执行完全是串行的。只有在一个任务运行完成之后，另一个任务才会被读取；多任务操作系统则允许终端用户同时运行多个程序。当一个程序暂时不需要使用CPU的时候，系统会把该程序挂起或中断，以使其他程序可以使用CPU。最早支持并发编程的计算机编程语言是汇编语言。

串行程序特指只能被顺序执行的指令列表，并发程序则是可以被并发执行的多个串行程序的综合体。因此并发程序内部会被划分为多个部分，每个部分都可以看做一个串行程序。串行程序中的所有代码的先后顺序都是固定的，而并发程序中只有部分代码是有序的。这一特性被称为不确定性。这导致并发程序每次运行的代码执行路径都是不同的，即便输入数据相同也会如此。

Go语言通过编译器运行时runtime，从语言上支持并发的特性。Go语言的并发通过goroutine特性完成。

12.2 进程和线程

12.2.1 基本概念

进程（process）和线程（thread）是操作系统级别的两个基本概念。计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。

进程就好比工厂的车间，它代表CPU所能处理的单个任务。进程是一个容器。线程就好比车间里的工人。一个进程可以包括多个线程。线程是容器中的工作单位。一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。就如同车间的空间是工人们共享的，许多房间是每个工人都可以进出的。车间的每个房间大小不同，有些房间最多只能容纳一人，比如厕所。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫"互斥锁"（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。还有些房间，可以同时容纳n个人，比如厨房。如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法用来保证多个线程不会互相冲突。

进程是程序的一次动态执行过程；进程需要经历从代码加载、代码执行到执行完毕的一个完整过程，这个过程也就是进程本身从产生、发展到消亡的过程；每个运行中的程序就是一个进程；进程是一个实体，每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）；进程在系统中独立存在，拥有自己独立的资源，多个进程可以在单个处理器上并发执行且互不影响；操作系统可以同时执行多个任务，每个任务就是进程。

进程有三个状态：就绪、运行和阻塞。就绪状态其实就是获取了除CPU之外的所有资源，只要处理器分配资源就可以马上执行。就绪状态有排队序列什么的，排队原则不再赘述；运行态就是获得了处理器分配的资源，程序开始执行；阻塞态，当程序条件不够时候，需要等待条件满足时候才能执行，如等待i/o操作时候，此刻的状态就叫阻塞态。

线程是CPU的基本调度单位；线程是在进程之后发展出来的概念，线程也叫轻量级进程；通常一个进程包含若干个线程，当然一个进程中至少有一个线程，不然没有存在的意义；线程可以利用进程所拥有的资源，在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位；由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统多个程序间并发执行的程度。

杀毒软件程序是一个进程，该程序在为计算机进行检查的同时，可以清理垃圾文件，并进行日志记录。检查计算机、清理垃圾文件、日志记录这就是线程。

启动word文档就是运行了一个进程，文字录入时，可以进行拼写错误检查，并使首字母大写。文字录入、拼写错误检查、首字母大写就属于word操作进程中的线程。

12.2.2 进程与线程的区别

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。

进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

一个程序至少有一个进程，一个进程至少有一个线程。线程的划分尺度小于进程，使得多线程程序的并发性高。进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

线程在执行过程中与进程还是有区别的。每个独立的进程有一个程序运行的入口、顺序执行程序。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP(多核处理机)机器上运行，而进程则可以跨机器迁移。

12.2.3 相关背景知识

进程和线程都是一个时间段的描述，是CPU工作时间段的描述，不过是颗粒大小不同。

CPU+RAM+各种资源（比如显卡、GPU）构成我们的电脑，但是电脑的运行，实际就是CPU和相关寄存器以及RAM之间的事情。

一个最基础的事实：CPU太快，太快，太快了，寄存器仅仅能够追的上他的脚步，RAM和别的挂在各总线上的设备完全是望其项背。那当多个任务要执行的时候怎么办呢？轮流着来?或者谁优先级高谁来？不管怎么样的策略，一句话就是在CPU就是轮流着来。

一个必须知道的事实：执行一段程序代码，实现一个功能的过程，当得到CPU的时候，相关的资源必须也已经就位，就是显卡啊，GPU什么的必须就位，然后CPU开始执行。这里除了CPU以外所有的就构成了这个程序的执行环境，也就是我们所定义的程序上下文。当这个程序执行完了，或者分配给他的CPU执行时间用完了，那它就要被切换出去，等待下一次CPU的临幸。在被切换出去的最后一步工作就是保存程序上下文，因为这个是下次他被CPU临幸的运行环境，必须保存。

串联起来的事实：前面讲过在CPU看来所有的任务都是一个一个的轮流执行的，具体的轮流方法就是：先加载程序A的上下文，然后开始执行A，保存程序A的上下文，调入下一个要执行的程序B的程序上下文，然后开始执行B,保存程序B的上下文……

进程就是包括上下文切换的程序执行时间总和 = CPU加载上下文 + CPU执行 + CPU保存上下文

线程是什么呢？进程的颗粒度太大，每次都要有上下的调入，保存，调出。如果我们把进程比喻为一个运行在电脑上的软件，那么一个软件的执行不可能是一条逻辑执行的，必定有多个分支和多个程序段，就好比要实现程序A 实际分成 a，b，c等多个块组合而成。那么这里具体的执行就可能变成：程序A得到CPU -> CPU加载上下文，开始执行程序A的a小段，然后执行A的b小段，然后再执行A的c小段，最后CPU保存A的上下文。

这里a，b，c的执行是共享了A的上下文，CPU在执行的时候没有进行上下文切换的。这里的a，b，c就是线程，也就是说线程是共享了进程的上下文环境的更为细小的CPU时间段。

操作系统的设计，可归结为三点：

（1）以多进程形式，允许多个任务同时运行；

（2）以多线程形式，允许单个任务分成不同的部分运行；

（3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

12.3 轻量级线程——协程coroutine

12.3.1 什么是协程？

协程（Coroutine），又称为微线程。是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。协程最初在1963年被提出。

协程不是进程或线程，其执行过程更类似于子程序，或者说不带返回值的函数调用。

最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。Coroutine是编译器级的，Process和Thread是操作系统级的。

协程与传统的系统级线程和进程相比，协程的最大优势在于其轻量级，可以轻松创建上百万个，而不会导致系统资源衰竭；线程和进程通常最多也不能超过一万个。这也是协程叫做轻量级线程的原因。

协程的执行效率极高。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销。和多线程比，线程数量越多，协程的性能优势就越明显。

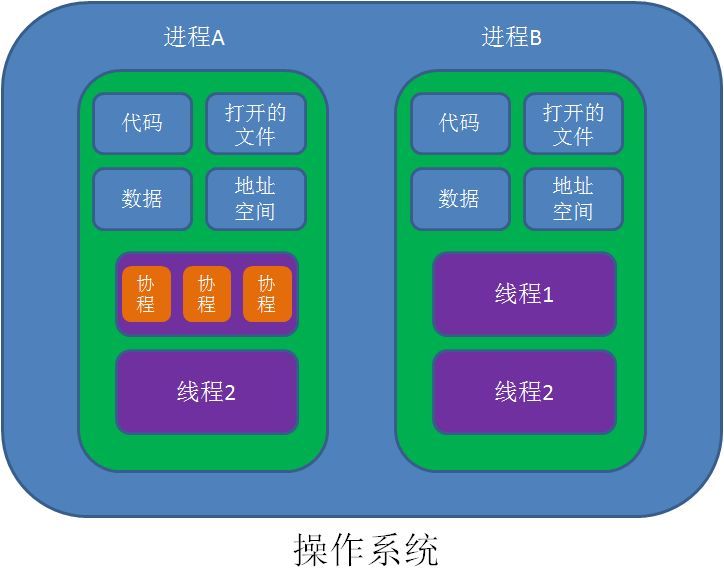


图 13.2

12.3.2 协程的应用

有哪些编程语言应用到了协程呢？

Lua语言：Lua从5.0版本开始使用协程，通过扩展库coroutine来实现。

Python语言：python可以通过 yield/send 的方式实现协程。在python 3.5以后，async/await 成为了更好的替代方案。

Go语言：Go语言对协程的实现非常强大而简洁，可以轻松创建成百上千个协程并发执行。

Java语言：Java语言并没有对协程的原生支持，但是某些开源框架模拟出了协程的功能，例如：Kilim框架。

12.4 Go语言中协程——goroutine

12.4.1 什么是Goroutine？

在执行多任务时，为了保证每个任务能及时被分配到CPU上运行处理，同时避免多个任务频繁地在线程间切换执行而损失效率，常使用线程池；但是如果面对随时随地可能发生的并发和线程处理需求，线程池不是非常直观和方便；

使用者分配足够多的任务，系统能自动帮助使用者把任务分配到CPU上，让这些任务尽量并发运行。这种机制在GO语言中被称为Goroutine。Goroutine类似于线程，Goroutine由GO程序运行时runtime调度和管理，GO程序会智能地将Goroutine中的任务合理地分配给每个CPU。

Go程序从main包的main()函数开始，在程序启动时，Go程序会为main()函数创建一个默认的Goroutine。

12.4.2 Goroutine在线程上的优势

Go中使用Goroutine来实现并发concurrently。Goroutine是与其他函数或方法同时运行的函数或方法。

与线程相比，Goroutine非常便宜。它们只是堆栈大小的几个kb，堆栈可以根据应用程序的需要增长和收缩，而在线程的情况下，堆栈大小必须指定并且是固定的。由于创建Goroutine的成本很小。因此，Go应用程序可以并发运行数千个Goroutine。

当使用Goroutine访问共享内存时，通过设计的通道可以防止竞态条件发生。通道可以被认为是Goroutine通信的管道。

Coroutine与Goroutine比较：

都可以将函数或者语句在独立的环境中运行，但是他们之间存在两点不同：

（1）goroutine可能发生并行执行；但是coroutine始终顺序执行；goroutine可能发生在多线程环境下，coroutine始终发生在单线程，coroutine程序需要主动交出控制权，宿主才能获得控制权并将控制权交给其他coroutine。coroutine的运行机制属于协作式任务处理，应用程序在不需要使用CPU时，需要主动交出CPU使用权。如果开发者无意间让应用程序长时间占用CPU，操作系统也无能为力，计算机很容易失去响应或者死机。

goroutine属于抢占式任务处理，已经和现有的多线程和多进程任务处理非常类似。应用程序对CPU的控制最终需要有操作系统来管理，操作系统如何发现一个应用程序常时间占用CPU，那么用户有权终止这个任务。

（2）goroutine间使用channel通信；coroutine使用yield和resume操作。

12.4.3 使用普通函数创建goroutine

在函数或方法调用前面加上关键字go，将会同时运行一个新的Goroutine。

使用go关键字创建Goroutine时，被调用的函数往往没有返回值，如果有返回值也会被忽略。

如果需要在Goroutine中返回数据，必须使用通道channel，通过通道把数据从Goroutine中作为返回值传出。

例：

package main

import (

"fmt"

)

func hello() {

fmt.Println("Hello world goroutine")

}

func main() {

go hello()

//time.Sleep(50 \* time.Microsecond)

fmt.Println("main function")

}

运行结果：

main function

Go程序的执行过程是：创建和启动主goroutine，初始化操作，执行main()函数，当main()函数结束，主goroutine随之结束，程序结束。

被启动的goroutine叫做子goroutine。

如果main的goroutine终止了，程序将被终止，而其他Goroutine将不再运行。换句话说，所有goroutine在main()函数结束时会一同结束。

修改代码

func hello() {

fmt.Println("Hello world goroutine")

}

func main() {

go hello()

time.Sleep(50 \* time.Microsecond)

fmt.Println("main function")

}

运行结果：

Hello world goroutine

main function

在上面的程序中，我们已经调用了时间包的Sleep方法，它会在执行过程中睡觉。在这种情况下，main的goroutine被用来睡觉50毫秒。现在调用go hello()有足够的时间在main Goroutine终止之前执行。这个程序首先打印Hello world goroutine，等待50毫秒，然后打印main函数。

例2：

func main() {

go running()

var input string

fmt.Scanln(&input)

}

func running() {

var times int

for {

times++

fmt.Println("tick", times)

time.Sleep(time.Second)

}

}

程序运行效果。控制台不断输出tick，同时还可以接收用户输入。两个环节同时运行。

该案例中，Go程序在启动时，运行时runtime默认为main()函数创建一个goroutine。在main()函数的goroutine执行到go running()语句时，归属于running()函数的goroutine被创建，running()函数开始在自己的goroutine中执行。

此时，main()继续执行，两个goroutine通过GO程序的调度机制同时运行。

go也可以使用匿名函数创建goroutine，go关键字后也可以是匿名函数或闭包。

例：

func main() {

go func() {

var times int

for {

times++

fmt.Println("tick" , times)

time.Sleep(time.Second)

}

}()

var input string

fmt.Scanln(&input)

}

go也允许启动多个Goroutines。

例：

func numbers() {

for i := 1; i <= 5; i++ {

time.Sleep(250 \* time.Millisecond)

fmt.Printf("%d ", i)

}

}

func alphabets() {

for i := 'a'; i <= 'e'; i++ {

time.Sleep(400 \* time.Millisecond)

fmt.Printf("%c ", i)

}

}

func main() {

go numbers()

go alphabets()

time.Sleep(3000 \* time.Millisecond)

fmt.Println("\n main terminated")

}

运行结果：

1 a 2 3 b 4 c 5 d e

main terminated

12.4.4 调整并发的运行性能Gomaxprocs

在Go程序运行时runtime实现了一个小型的任务调度器。这套调度器的工作原理类似于操作系统调度线程，Go程序调度器可以高效地将CPU资源分配给每一个任务。

传统逻辑中，开发者需要维护线程池中的线程与CPU核心数量的对应关系。在Go语言中可以通过runtime.GOMAXPROCS()函数做到。

语法为：runtime.GOMAXPROCS(逻辑CPU数量)

逻辑CPU数量有如下几种数值：

<1 ，不修改任何数值；

=1，单核执行；

>1，多核并发执行。

Go1.5版本之前，默认使用单核执行。Go1.5版本开始，默认执行：runtime.GOMAXPROCS(逻辑CPU数量)，让代码并发执行，最大效率地利用CPU。

12.5 多个goroutine间通信的通道channel

12.5.1 通道的概述

单纯地讲函数并发执行没有意义。函数与函数间需要交换数据才能体现出并发执行的意义。虽然可以使用共享内存进行数据交换，但是共享内存在不同goroutine中容易发生竞态问题，必须使用互斥对内存进行加锁，所以会造成性能问题。

Go语言中提倡使用通道channel的方式代替共享内存。也就是说，Go语言中主张，应该通过数据传递来实现共享内存，而不是通过共享内存来实现消息传递。

排队的目的是避免拥堵、插队造成的资源使用和交换过程低效问题。多个goroutine为了争抢数据，势必造成低效，使用队列的方式是最高效的，channel就是一种队列结构。

Go语言中的通道channel是一种特殊的类型。通道像一个传送带或者队列，总是遵循先入先出first in first out的规则，保证收发数据的顺序。通道可以被认为是Goroutine通信的管道。类似于管道中的水从一端到另一端的流动，数据可以从一端发送到另一端，通过通道接收。每个通道都有与其相关的类型。该类型是通道允许传输的数据类型。通道的零值为nil。nil通道没有任何用处，因此通道必须使用类似于map和slice的方法来定义。

12.5.2 声明通道类型

var 通道变量 chan 通道类型

chan类型的空值是nil，声明后需要配合make才能使用。

通道是引用类型，需要使用make进行创建

语法：通道示例 := make(chan 数据类型)

例如：

ch1 := make(chan int) //创建一个整数类型通道

ch2 := make(chan interface{}) //创建一个空接口类型的通道，可以存放任意数据

type Equip struct {/\* 属性 \*/}

ch3 := make(chan \*Equip) //创建一个Equip指针类型的通道，可以存放Equip指针

12.5.3 通道发送数据

通道发送使用特殊的操作符"<-"，将数据通过通道发送的语法为：

通道变量 <- 值

通道发送的值可以是变量、常量、表达式或函数返回值等。值的类型必须与ch通道的元素类型一致。

把数据往通道中发送，如果接收方一直没有接收，那么发送操作将持续阻塞。此时所有的goroutine，包括main的goroutine都处于等待状态。

运行会提示报错：fatal error: all goroutines are asleep - deadlock!

死锁deadlock

使用通道时要考虑的一个重要因素是死锁。

如果Goroutine在一个通道上发送数据，那么预计其他的Goroutine应该接收数据。如果这种情况不发生，那么程序将在运行时出现死锁。

类似地，如果Goroutine正在等待从通道接收数据，那么另一些Goroutine将会在该通道上写入数据，否则程序将会死锁。

例：

func main() {

//创建一个空指针型通道

ch := make(chan interface{})

//将0通过通道发送

ch <- 0

//发送字符串

ch <- "StevenWang"

}

12.5.4 阻塞

一个通道发送和接收数据，默认是阻塞的。当一个数据被发送到通道时，在发送语句中被阻塞，直到另一个Goroutine从该通道读取数据。类似地，当从通道读取数据时，读取被阻塞，直到一个Goroutine将数据写入该通道。

这些通道的特性是帮助Goroutines有效地进行通信，而无需像使用其他编程语言中非常常见的显式锁或条件变量。

例：

func main() {

var ch1 chan int

fmt.Println(ch1) //<nil>,

fmt.Printf("%T\n", ch1) //chan int

ch1 = make(chan int)

fmt.Println(ch1) //0xc4200200c0

ch2 := make(chan bool)

go func() {

fmt.Println("子goroutine。。。")

data, ok := <-ch1 //阻塞式，从通道中读取数据

time.Sleep(1 \* time.Second)

fmt.Println("子goroutine从通道中读取到main传来的数据是：", ok, data)

ch2 <- true //向通道中写入数据，表示结束

}()

ch1 <- 100 //阻塞式，main goroutine向通道中写入数据

<-ch2 //目的是防止main goroutine先执行完毕后退出。因为如果main的goroutine终止了，程序将被终止，而其他Goroutine将不再运行。

fmt.Println("main..over")

}

运行结果：

<nil>

chan int

0xc420076060

子goroutine。。。

子goroutine从通道中读取到main传来的数据是： true 100

main..over

12.5.5 通道接收数据

通道接收同样使用特殊的操作符"<-"。

通道变量 <- 值

通道收发操作在不同的两个goroutine间进行。接收操作将持续阻塞，直到发送方发送数据。每次接收一个元素。

阻塞接收数据的完整写法如下：

data , ok := <-ch

data ：表示接收到的数据。未接收到数据时，data为通道类型的零值。

ok：表示是否接收到数据。

执行该语句时将会阻塞，直到接收到数据并赋值给data变量。通过ok值可以判断当前通道是否被关闭。

<-ch表示接收任意数据，忽略接收的数据。执行该语句时将会阻塞。其目的不在于接收通道中数据，而是为了阻塞goroutine。

循环接收数据，需要配合使用关闭通道。借助普通for循环和for ... range语句循环接收多个元素，遍历通道，遍历的结果就是接收到的数据，数据类型就是通道的数据类型。

普通for循环接收通道数据，需要有break循环的条件；for range会自动判断出通道已关闭，而无需通过判断来终止循环。

例：

//循环接收数据的三种方式

func main() {

ch1 := make(chan string)

go sendData(ch1)

//循环接收数据方式1

for {

data := <-ch1

//如果通道关闭，通道中传输的数据则为各数据类型的默认值。chan int 默认值为0，chan string默认值为"" 等。

if data == "" {

break

}

fmt.Println("从通道中读取数据方式1：", data)

}

//循环接收数据方式2

for {

data, ok := <-ch1

//通过多个返回值的形式来判断通道是否关闭，如果通道关闭，则ok值为false。

if !ok {

break

}

fmt.Println("从通道中读取数据方式2：", data)

}

//循环接收数据方式3

//for range循环会自动判断通道是否关闭，自动break循环。

for value := range ch1 {

fmt.Println("从通道中读取数据方式3：", value)

}

}

func sendData(ch1 chan string) {

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

ch1 <- fmt.Sprintf("发送数据%d\n", i)

}

fmt.Println("发送数据完毕。。")

//显式调用close()实现关闭通道

close(ch1)

}

12.5.6 关闭通道

发送方如果数据写入完毕，需要关闭通道，用于通知接受方数据传递完毕。一般都是发送方关闭通道。如何判断一个channel是否已经关闭？可以在读取的时候使用多重返回值的方式。如果返回值是false，则表示通道已经被关闭。如果往关闭的通道中写入数据，会报错：panic: send on closed channel。但是可以从关闭后的通道中取数据，不过返回数据默认值和false。

例：

func main() {

//通道关闭后是否可以写入和读取呢？

ch1 := make(chan int)

go func() {

ch1 <- 100

ch1 <- 200

close(ch1)

//ch1 <- 10 //关闭的通道，无法写入数据

}()

data, ok := <-ch1

fmt.Println("main读取数据：", data, ok)

data, ok = <-ch1

fmt.Println("main读取数据：", data, ok)

data, ok = <-ch1

fmt.Println("main读取数据：", data, ok)

data, ok = <-ch1

fmt.Println("main读取数据：", data, ok)

data, ok = <-ch1

fmt.Println("main读取数据：", data, ok)

}

返回结果：

main读取数据： 100 true

panic: send on closed channel

main读取数据： 200 true

main读取数据： 0 false

main读取数据： 0 false

main读取数据： 0 false

goroutine 5 [running]:

main.main.func1(0xc42006e060)

/Users/steven/Documents/go\_project/src/ch10\_2/demo01\_test1.go:12 +0x79

created by main.main

/Users/steven/Documents/go\_project/src/ch10\_2/demo01\_test1.go:8 +0x70

12.6 缓冲通道和定向通道

12.6.1 缓冲通道

默认创建的通道都是非缓冲通道，读写都是即时阻塞；缓冲通道是自带一块缓冲区，可以暂时存储数据，如果缓冲区满了，那么才会阻塞。

例：

func main() {

//1.非缓冲通道

ch1 := make(chan int)

fmt.Println("非缓冲通道：", len(ch1), cap(ch1)) //0, 0

go func() {

data := <-ch1 //阻塞

fmt.Println("获取数据：", data)

}()

ch1 <- 100 //阻塞

//time.Sleep(1)

fmt.Println("写入数据ok\n-----------------------")

//2.缓冲通道,缓冲区满了才会阻塞

//ch2 := make(chan int, 5)

//fmt.Println("缓冲通道：", len(ch2), cap(ch2)) //0,5

//go func() {

// for data := range ch2 {

// //time.Sleep(1)

// fmt.Println("获取数据：", data)

// }

//}()

//

//ch2 <- 1

//fmt.Println(len(ch2), cap(ch2))

//ch2 <- 2

//fmt.Println(len(ch2), cap(ch2))

//ch2 <- 3

//fmt.Println(len(ch2), cap(ch2)) //3,5

//ch2 <- 4

//fmt.Println(len(ch2), cap(ch2))

//ch2 <- 5

//fmt.Println(len(ch2), cap(ch2)) //5, 5

//ch2 <- 6 //阻塞

//fmt.Println(len(ch2), cap(ch2))

//close(ch2)

//fmt.Println("main。。。over。。")

//3.缓冲通道

ch3 := make(chan string, 5)

fmt.Printf("%T\n", ch3)

go sendData(ch3)

for data := range ch3 {

//time.Sleep(1 \* time.Second)

fmt.Println("\t读取数据：", data)

}

fmt.Println("读取完毕。。")

}

func sendData(ch3 chan string) {

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

ch3 <- fmt.Sprintf("data%d", i) //1,2,3,4,5,6,7

fmt.Println("写入数据：", i) //1,2,3,4,5

fmt.Println(len(ch3), cap(ch3))

}

close(ch3)

}

缓冲通道模拟生产者和消费者

func main() {

ch1 := make(chan int , 5)

ch2 := make(chan bool)//判断结束

rand.Seed(time.Now().UnixNano())

//写入数据：生产者

go func() {

for i := 1; i <= 20; i++ {

ch1 <- i

fmt.Println("写入数据：", i)

time.Sleep(time.Duration(rand.Intn(1000))\*time.Millisecond)

}

close(ch1)

}()

//读取数据：消费者

go func() {

for data := range ch1 {

fmt.Println("\t1号消费者：", data)

time.Sleep(time.Duration(rand.Intn(1000))\*time.Millisecond)

}

ch2 <- true

}()

go func() {

for data := range ch1 {//1

fmt.Println("\t2号消费者：", data)

time.Sleep(time.Duration(rand.Intn(1000))\*time.Millisecond)

}

ch2 <- true

}()

<- ch2

fmt.Println("main...over...")

}

12.6.2 定向通道

通道默认都是双向通道。即可写入数据，又可读取数据。定向通道：也叫单向通道，只读或只写。

只读：make(<- chan Type)，只能读取数据，不能写入数据

<- chan

只写：make(chan <- Type)，只能写入数据，不能读取数据

chan <- data

将通道作为参数传递的时候使用单向通道。定义函数，只有写入数据功能，或定义函数，只有读取数据功能。定向通道的意义：在语法级别，保证通道的操作安全

例：

func main() {

//1.双向通道

ch1 := make(chan string)

go fun1(ch1)

data := <-ch1

fmt.Println("main，接受到数据：", data)

ch1 <- "Go语言好学么？"

ch1 <- "区块链好学么?"

go fun2(ch1)

go fun3(ch1)

time.Sleep(1 \* time.Second)

fmt.Println("main over!")

}

func fun1(ch1 chan string) {

ch1 <- "我是Steven老师"

data := <-ch1

data2 := <-ch1

fmt.Println("回应：", data, data2)

}

//功能：只有写入数据

func fun2(ch1 chan<- string) {

//只能写入

ch1 <- "How are you?"

//<- ch1 //invalid operation: <-ch1 (receive from send-only type chan<- string)

}

//功能：只有读取数据

func fun3(ch1 <-chan string) {

data := <-ch1

fmt.Println("只读：", data)

//ch1 <- "hello" //invalid operation: ch1 <- "hello" (send to receive-only type <-chan string)

}

12.7 time包中跟通道相关的函数

12.7.1 Timer结构体

计时器类型表示单个事件。当计时器过期时，当前时间将被发送到C上(C是一个只读通道<-chan time.Time，该通道中放入的是Time结构体)，除非计时器是AfterFunc创建的。计时器必须使用NewTimer或AfterFunc创建。

Timer结构体的源码

// The Timer type represents a single event.

// When the Timer expires, the current time will be sent on C,

// unless the Timer was created by AfterFunc.

// A Timer must be created with NewTimer or AfterFunc.

type Timer struct {

C <-chan Time

r runtimeTimer

}

12.7.2 NewTimer函数

NewTimer创建一个新的计时器，它会在至少持续时间d之后将当前时间发送到其通道上。

NewTimer()函数的源码

// NewTimer creates a new Timer that will send

// the current time on its channel after at least duration d.

func NewTimer(d Duration) \*Timer

例：

//.创建计时器

timer1 := time.NewTimer(5 \* time.Second)

fmt.Printf("%T\n", timer1) //\*time.Timer

fmt.Println(time.Now())

data := <-timer1.C //<-chan time.Time

fmt.Printf("%T\n",timer1.C) //<-chan time.Time

fmt.Printf("%T\n",data) //time.Time

fmt.Println(data)

12.7.3 After()函数

After()函数相当于NewTimer(d). C。

After()函数的源码

// After waits for the duration to elapse and then sends the current time

// on the returned channel.

// It is equivalent to NewTimer(d).C.

// The underlying Timer is not recovered by the garbage collector

// until the timer fires. If efficiency is a concern, use NewTimer

// instead and call Timer.Stop if the timer is no longer needed.

func After(d Duration) <-chan Time {

return NewTimer(d).C

}

例：

//使用After(),返回值<- chan Time,同Timer.C

ch1 := time.After(5 \* time.Second)

fmt.Println(time.Now())

data := <-ch1

fmt.Printf("%T\n",data) //time.Time

fmt.Println(data)

12.8 select分支语句

select 语句类似于 switch 语句，但是select会随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。每个case都必须是一个通信，所有channel表达式都会被求值，所有被发送的表达式都会被求值。

如果任意某个通信可以进行，它就执行；其他被忽略。如果有多个case都可以运行，select会随机公平地选出一个执行，其他不会执行；如果有default子句，则执行该语句；如果没有default字句，select将阻塞，直到某个通道可以运行；Go不会重新对channel或值进行求值。

例1：

func main() {

ch1 := make(chan int)

ch2 := make(chan int)

go func() {

time.Sleep(1 \* time.Second)

ch1 <- 100

}()

go func() {

time.Sleep(1 \* time.Second)

ch2 <- 200

}()

select {

case data := <-ch1:

fmt.Println("ch1中读取数据了:", data)

case data := <-ch2:

fmt.Println("ch2中读取数据了：", data)

default:

fmt.Println("执行了default。。。")

}

}

运行结果：

执行了default。。。

例2：

func main() {

ch1 := make(chan int)

ch2 := make(chan int)

go func() {

time.Sleep(1 \* time.Second)

data := <-ch1

fmt.Println("ch1：", data)

}()

go func() {

time.Sleep(2 \* time.Second)

data := <-ch2

fmt.Println("ch2：", data)

}()

select {

case ch1 <- 100: //阻塞

close(ch1)

fmt.Println("ch1中写入数据。。")

case ch2 <- 200: //阻塞

close(ch2)

fmt.Println("ch2中写入数据。。")

case <-time.After(2 \* time.Second): //阻塞

fmt.Println("执行延时通道")

//default:

// fmt.Println("default..")

}

time.Sleep(4 \* time.Second)

fmt.Printf("main over ")

}

运行结果：

ch1： 100

ch1中写入数据。。

main over

12.9 sync包

sync包提供了基本的同步基元，如互斥锁。除了Once和WaitGroup类型，大部分都是适用于低水平程序线程，高水平的同步使用channel通信更好一些。本包的类型的值不应被拷贝。

之前的案例中，我们都使用time.Sleep()函数，通过睡眠将主线程阻塞至所有线程结束。而更好的做法是使用WaitGroup来实现。

12.9.1 WaitGroup同步等待组

* 同步：sync是串行执行；
* 异步：async是同时执行；
* WaitGroup：同步等待组。

type WaitGroup struct {

noCopy noCopy

state1 [12]byte

sema uint32

}

WaitGroup用于等待一组线程的结束。父线程调用Add方法来设定应等待的线程的数量。每个被等待的线程在结束时应调用Done方法。同时，主线程里可以调用Wait方法阻塞至所有线程结束。

WaitGroup中的方法如下：

func (wg \*WaitGroup) Add(delta int)

Add方法向内部计数加上delta，delta可以是负数；如果内部计数器变为0，Wait方法阻塞等待的所有线程都会释放，如果计数器小于0，方法panic。注意Add加上正数的调用应在Wait之前，否则Wait可能只会等待很少的线程。一般来说本方法应在创建新的线程或者其他应等待的事件之前调用。

func (wg \*WaitGroup) Done()

Done方法减少WaitGroup计数器的值，应在线程的最后执行。

func (wg \*WaitGroup) Wait()

Wait方法阻塞直到WaitGroup计数器减为0。

例：

func main() {

var wg sync.WaitGroup

fmt.Printf("%T\n", wg) //sync.WaitGroup

fmt.Println(wg) //{{} [0 0 0 0 0 0 0 0 0 0 0 0] 0}

wg.Add(3)

rand.Seed(time.Now().UnixNano())

go printNum1(&wg)

go printNum2(&wg)

go printNum3(&wg)

wg.Wait() //main goroutine进入阻塞状态，当计数器为0后解除阻塞

fmt.Println("main解除阻塞，main over...")

}

func printNum1(wg \*sync.WaitGroup) {

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

fmt.Println("子goroutine1,i：", i)

time.Sleep(time.Duration(rand.Intn(1000)))

}

wg.Done() //计数器减1

}

func printNum2(wg \*sync.WaitGroup) {

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

fmt.Println("\t 子goroutine1,i：", i)

time.Sleep(time.Duration(rand.Intn(1000)))

}

wg.Done() //计数器减1

}

func printNum3(wg \*sync.WaitGroup) {

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

fmt.Println("\t\t 子goroutine1,i：", i)

time.Sleep(time.Duration(rand.Intn(1000)))

}

wg.Done() //计数器减1

}

12.9.2 互斥锁

Mutex是一个互斥锁，可以创建为其他结构体的字段；零值为解锁状态。Mutex类型的锁和线程无关，可以由不同的线程加锁和解锁。

type Mutex struct {

state int32

sema uint32

}

func (m \*Mutex) Lock()

Lock方法锁住m，如果m已经加锁，则阻塞直到m解锁。

func (m \*Mutex) Unlock()

Unlock方法解锁m，如果m未加锁会导致运行时错误。锁和线程无关，可以由不同的线程加锁和解锁。

例：

//互斥锁实现售票示例

var tickts = 100 //全局变量，仅一份

var wg sync.WaitGroup

var mutex sync.Mutex //互斥锁

func main() {

/\*

练习题：模拟火车站卖票

火车票100张，4个售票口出售(4个goroutine)。

\*/

var wg sync.WaitGroup

wg.Add(4)

go saleTickets("售票口1", &wg) //g1

go saleTickets("售票口2", &wg) //g2

go saleTickets("售票口3", &wg) //g3

go saleTickets("售票口4", &wg) //g4

wg.Wait()

fmt.Println("所有车票已售空。程序结束！")

}

func saleTickets(name string, wg \*sync.WaitGroup) {

for {

mutex.Lock()

if tickts > 0 {

time.Sleep(1 \* time.Second)

fmt.Println(name, "：", tickts) //1

tickts--

} else {

fmt.Println(name, ",结束卖票。。")

mutex.Unlock()

break

}

mutex.Unlock() //解锁

}

wg.Done()

}

12.9.3 读写互斥锁RWMutex

RWMutex是读写互斥锁。该锁可以被同时多个读取者持有或唯一个写入者持有。RWMutex可以创建为其他结构体的字段；零值为解锁状态。RWMutex类型的锁也和线程无关，可以由不同的线程加读取锁/写入和解读取锁/写入锁。

type RWMutex struct {

w Mutex // held if there are pending writers

writerSem uint32 // semaphore for writers to wait for completing readers

readerSem uint32 // semaphore for readers to wait for completing writers

readerCount int32 // number of pending readers

readerWait int32 // number of departing readers

}

锁定的规则：

读写锁的使用中：写操作都是互斥的、读和写是互斥的、读和读不互斥。

理解为：

可以多个goroutine同时读取数据，但是写只允许一个goroutine写数据。

Mutex中的方法如下：

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) Lock()

Lock方法将rw锁定为写入状态，禁止其他线程读取或者写入。

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) Unlock()

Unlock方法解除rw的写入锁状态，如果m未加写入锁会导致运行时错误。

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) RLock()

RLock方法将rw锁定为读取状态，禁止其他线程写入，但不禁止读取。

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) RUnlock()

Runlock方法解除rw的读取锁状态，如果m未加读取锁会导致运行时错误。

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) RLocker() [Locker](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Locker)

Rlocker方法返回一个互斥锁，通过调用rw.Rlock和rw.Runlock实现了Locker接口。

例：

func main() {

var rwm sync.RWMutex

for i := 1; i <= 3; i++ {

go func(i int) {

fmt.Printf("goroutine %d，尝试读锁定。。\n", i)

rwm.RLock()

fmt.Printf("goroutine %d，已经读锁定了。。\n", i)

time.Sleep(5 \* time.Second)

fmt.Printf("goroutine %d,读解锁。。\n", i)

rwm.RUnlock()

}(i)

}

time.Sleep(1\*time.Second)

fmt.Println("main..尝试写锁定。。")

rwm.Lock()

fmt.Println("main。。已经写锁定了。。")

rwm.Unlock()

fmt.Printf("main。。写解锁。。。")

}

12.9.4 条件变量Cond

Cond实现了一个条件变量，一个线程集合地，供线程等待或者宣布某事件的发生。

每个Cond实例都有一个相关的锁（一般是\*Mutex或\*RWMutex类型的值），它必须在改变条件时或者调用Wait方法时保持锁定。Cond可以创建为其他结构体的字段，Cond在开始使用后不能被拷贝。条件变量：sync.Cond,多个goroutine等待或接受通知的集合地

type Cond struct {

noCopy noCopy

// L is held while observing or changing the condition

L Locker

notify notifyList

checker copyChecker

}

Cond中的方法如下：

func NewCond(l [Locker](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Locker)) \*[Cond](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Cond)

使用锁l创建一个\*Cond。Cond条件变量，总是要和锁结合使用。

func (c \*[Cond](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Cond)) Broadcast()

Broadcast唤醒所有等待c的线程。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。

func (c \*[Cond](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Cond)) Signal()

Signal唤醒等待c的一个线程（如果存在）。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。发送通知给一个人。

func (c \*[Cond](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Cond)) Wait()

Wait自行解锁c.L并阻塞当前线程，在之后线程恢复执行时，Wait方法会在返回前锁定c.L。和其他系统不同，Wait除非被Broadcast或者Signal唤醒，不会主动返回。广播给所有人。

因为线程中Wait方法是第一个恢复执行的，而此时c.L未加锁。调用者不应假设Wait恢复时条件已满足，相反，调用者应在循环中等待：

例：

func main() {

var mutex sync.Mutex

cond := sync.Cond{L:&mutex}

condition := false

go func() {

time.Sleep(1\*time.Second)

cond.L.Lock()

fmt.Println("子goroutine已经锁定。。。")

fmt.Println("子goroutine更改条件数值，并发送通知。。")

condition = true//更改数值

cond.Signal() //发送通知：一个goroutine

fmt.Println("子gorutine。。。继续。。。")

time.Sleep(5\*time.Second)

fmt.Println("子groutine解锁。。")

cond.L.Unlock()

}()

cond.L.Lock()

fmt.Println("main..已经锁定。。。")

if !condition{

fmt.Println("main.。即将等待。。。")

//wait()

// 1.wait尝试解锁，

// 2.等待--->当前的groutine进入了阻塞状态，等待被唤醒：signal(),broadcast()

// 3.一旦被唤醒后，又会锁定

cond.Wait()

fmt.Println("main.被唤醒。。")

}

fmt.Println("main。。。继续")

fmt.Println("main..解锁。。。")

cond.L.Unlock()

}

12.10 小结

本章学习Go语言并发的内容。并发能更客观地表现问题模型；并发可以充分利用 CPU 核心的优势，提高程序的执行效率；并发能充分利用 CPU 与其他硬件设备固有的异步性。并发是软件开发过程中必备的部分，掌握了Go语言并发编程，才算是掌握了Go语言的精髓之一。

第十三章 Go语言加密算法

本章重点为大家介绍如下的内容：

* 哈希算法（MD5 RipeMD160 SHA256 ）
* DES、3DES对称加密
* RSA非对称加密算法
* 椭圆曲线数字签名算法ECDSA
* 编码解码 （base64 base58 base58check）

13.1 哈希算法

13.1.1 Hash的定义

hash （哈希或散列）算法是信息技术领域非常基础也非常重要的技术。它能任意长度的二进制值（明文）映射为较短的固定长度的二进制值（Hash 值），并且不同的明文很难映射为相同的 Hash 值。hash 值在应用中又被称为数字指纹（fingerprint）或数字摘要（digest）、消息摘要。

例如计算一段话“hello blockchain”的 MD5 hash 值为：78e6a8bcdef7a4a254c16054b082c783

这意味着我们只要对某文件进行 MD5 Hash 计算，得到结果为 78e6a8bcdef7a4a254c16054b082c783，这就说明文件内容极大概率上就是 “hello blockchain”。可见，Hash 的核心思想十分类似于基于内容的编址或命名。

一个优秀的 hash 算法，将能实现：

正向快速：给定明文和 hash 算法，在有限时间和有限资源内能计算出 hash 值。

逆向困难：给定（若干） hash 值，在有限时间内很难（基本不可能）逆推出明文。

输入敏感：原始输入信息修改一点信息，产生的 hash 值看起来应该都有很大不同。

抗冲突：对不同的关键字可能得到同一散列地址，或者说两段内容不同的明文，它们的 hash 值可能一致。这种现象称冲突或者碰撞。具有相同函数值的关键字对该散列函数来说称做同义词。

抗冲突又称为“抗碰撞性”或冲突避免。哈希函数抗冲突就是不同的输入不能产生相同的输出。

抗冲突并不是不会有冲突，只不过找到有冲突的两个输入的代价非常大。就好像暴力破解一个有效期为20年的密码，整个破解过程或许长达30年。这样虽然最后密码被破解了，但是也失去意义了。

13.1.2 流行的哈希算法

目前流行的 Hash 算法包括 MD5、SHA-1 和 SHA-2系列（SHA-224、SHA-256、SHA-384，和 SHA-512 ）。

1.MD4

MD4（RFC 1320）是 MIT (Massachusetts Institute of Technology , 麻州理工大学) 的 Ronald L. Rivest （荣获2002年图灵奖）在 1990 年设计的，MD 是 Message Digest 的缩写。其输出为 128 位。

2.MD5

MD5（RFC 1321）是 Rivest 于1991年对 MD4 的改进版本。其输出是 128 位。MD5 比 MD4 复杂，并且计算速度要慢一点，更安全一些。MD5 已被证明不具备“强抗碰撞性”。

MD5 是一个经典的 hash 算法，其和 SHA-1 算法都已被证明安全性不足应用于商业场景。



图 13.1

3.SHA

SHA （Secure Hash Algorithm， [sɪ'kjʊə(r)] ['ælɡərɪðəm] ，加密哈希算法）是一个 Hash 函数族，由 NIST（National Institute of Standards and Technology ， 美国国家标准技术研究所）于 1993 年发布第一个算法。SHA-1 在 1995 年面世，它的输出为长度 160 位的 hash 值，SHA-1 设计时基于和 MD4 相同原理，SHA-1 已被证明不具备“强抗碰撞性”。

为了提高安全性，NIST 还设计出了 SHA-224、SHA-256、SHA-384，和 SHA-512 算法（统称为 SHA-2），跟 SHA-1 算法原理类似。

目前，一般认为 MD5 和 SHA1 已经不够安全，推荐至少使用 SHA-256 算法。

4.RIPEMD-160(RACE Integrity Primitives Evaluation Message Digest 160 ， RACE完整的原始评估信息摘要 )是一个160位的加密哈希函数，旨在替代128位哈希函数MD4、MD5。

13.1.3 Hash与加密解密的区别

1.哈希（Hash）与加密（Encrypt）的区别

这两个概念不是很清晰，容易混淆两者。而正确区别两者是正确选择和使用哈希与加密的基础。

哈希（Hash）是将目标文本转换成具有相同长度的、不可逆的杂凑字符串（或叫做消息摘要），而加密（Encrypt）是将目标文本转换成具有不同长度的、可逆的密文。

哈希算法往往被设计成生成具有相同长度的文本，而加密算法生成的文本长度与明文本身的长度有关。

哈希的结果具有相同的长度，而加密的结果则长度不同。实际上，如果使用相同的哈希算法，不论你的输入有多么长，得到的结果长度是一个常数，而加密算法往往与明文的长度成正比。

哈希算法是不可逆的，而加密算法是可逆的。

哈希不是一一映射，是不可逆的。而加密算法是一一映射，是可逆的。

哈希是不可能可逆的，因为如果可逆，那么哈希就是世界上最强悍的压缩方式了——能将任意大小的文件压缩成固定大小。

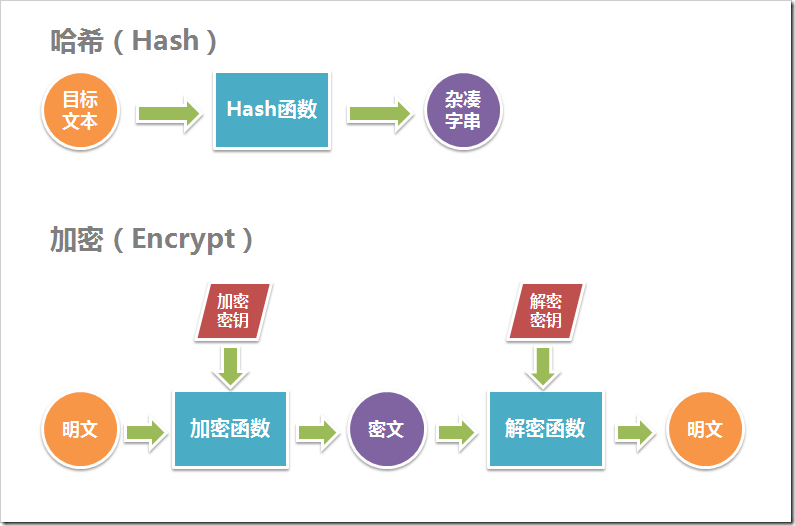


图 13.2

2.哈希（Hash）与加密（Encrypt）的选择

要实现数据保护，可以选择使用哈希或加密两种方式。那么在什么时候该选择哈希、什么时候该选择加密呢？

基本原则是：

如果被保护数据仅仅用作比较验证，在以后不需要还原成明文形式，则使用哈希；

如果被保护数据在以后需要被还原成明文，则需要使用加密。

例如，你正在做一个系统，你打算当用户忘记自己的登录口令时，重置此用户口令为一个随机口令，而后将此随机口令发给用户，让用户下次使用此口令登录，则适合使用哈希。实际上很多网站都是这么做的，想想你以前登录过的很多网站，是不是当你忘记口令的时候，网站并不是将你忘记的口令发送给你，而是发送给你一个新的、随机的口令，然后让你用这个新口令登录。这是因为你在注册时输入的口令被哈希后存储在数据库里，而哈希算法不可逆，所以即使是网站管理员也不可能通过哈希结果复原你的口令，而只能重置口令。

相反，如果你做的系统要求在用户忘记口令的时候必须将原口令发送给用户，而不是重置其口令，则必须选择加密而不是哈希。

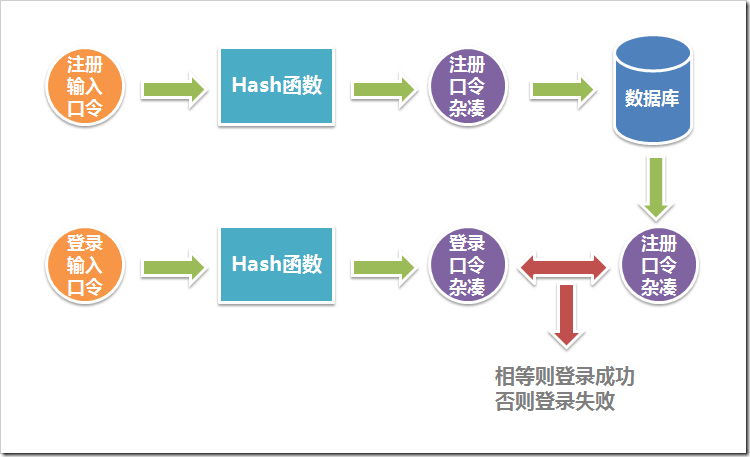


图 13.3

3.对简单哈希（Hash）的攻击

对于哈希的攻击，主要有寻找碰撞法和穷举法。

目前对于MD5和SHA1并不存在有效地寻找碰撞方法。我国杰出的数学家王小云教授曾经在国际密码学会议上发布了对于MD5和SHA1的碰撞寻找改进算法，但这种方法和“破解”相去甚远。该理论目前仅具有数学上的意义，她将破解MD5的预期步骤降低了好几个数量级，但对于实际应用来说仍然是一个天文数字。



图 13.4

穷举法（或暴力破解法），通俗来说，就是在一个范围内，如从000000到999999，将其中所有值一个一个用哈希算法哈希，然后将结果和杂凑串比较，如果相同，则这个值就一定是源字串或源字串的一个碰撞，于是就可以用这个值非法登录了。穷举法看似笨拙，但目前几乎所有的MD5破解机或MD5在线破解都是用这种穷举法。

纠其缘由，就是相当一部分口令是非常简单的，如“123456”或“000000”。穷举法是否能成功很大程度上取决于口令的复杂性。因为穷举法扫描的区间往往是单字符集、规则的区间，或者由字典数据进行组合，因此，如果使用复杂的口令，例如“!@#$%^&\*()”这种口令，穷举法就很难奏效了。

13.1.4 SHA256

SHA-256算法输入报文的最大长度不超过2^64 bit，输入按512-bit分组进行处理，产生的输出是一个256-bit的报文摘要。

SHA256算法包括以下几步：

step1:附加填充比特。

对报文进行填充使报文长度与448 模512 同余（长度=448 mod 512）,填充的比特数范围是1 到512，填充比特串的最高位为1,其余位为0。就是先在报文后面加一个 1，再加很多个0，直到长度满足mod 512=448.为什么是448，因为448+64=512. 第二步会加上一个 64bit的原始报文的 长度信息。

step2:附加长度值。

将用64-bit 表示的初始报文（填充前）的位长度附加在步骤1的结果 后（低位字节优先）。

step3:初始化缓存。

使用一个256-bit 的缓存来存放该散列函数的中间及最终结果。该缓存表示为A=0x6A09E667 , B=0xBB67AE85 , C=0x3C6EF372 , D=0xA54FF53A, E=0x510E527F , F=0x9B05688C , G=0x1F83D9AB , H=0x5BE0CD19 。

step4:处理512-bit（16 个字）报文分组序列。

该算法使用了六种基本逻辑函数，由64 步迭代运算组成。每步都以256-bit 缓存值ABCDEFGH 为输入，然后更新缓存内容。

每步使用一个32-bit 常数值Kt 和一个32-bit Wt。

13.1.5 核心代码

//1.基础工具函数

/\*\*

\* 将字节数组转成16进制字符串： []byte -> string

\*/

func BytesToHexString(arr []byte) string {

return hex.EncodeToString(arr)

}

/\*\*

\* 将16进制字符串转成字节数组： hex string -> []byte

\*/

func HexStringToBytes(s string) ([]byte, error) {

arr, err := hex.DecodeString(s)

return arr, err

}

/\*\*

\* 16进制字符串大端和小端颠倒

\*/

func ReverseHexString(hexStr string) string {

arr, \_ := hex.DecodeString(hexStr)

ReverseBytes(arr)

return hex.EncodeToString(arr)

}

/\*\*

\* 字节数组大端和小端颠倒

\*/

func ReverseBytes(data []byte) {

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

data[i], data[j] = data[j], data[i]

}

}

//2.哈希函数

func SHA256HexString(text string) string {

sha256Instance := sha256.New()

arr, \_ := hex.DecodeString(text)

sha256Instance.Write(arr)

ciphertext := sha256Instance.Sum(nil)

return fmt.Sprintf("%x", ciphertext)

}

func SHA256Double(text string, isHex bool) string {

sha256Instance := sha256.New()

if isHex {

arr, \_ := hex.DecodeString(text)

sha256Instance.Write(arr)

} else {

sha256Instance.Write([]byte(text))

}

hashBytes := sha256Instance.Sum(nil)

sha256Instance.Reset()

sha256Instance.Write(hashBytes)

hashBytes = sha256Instance.Sum(nil)

return fmt.Sprintf("%x", hashBytes)

}

func HASH(text string, hashType string, isHex bool) string {

var hashInstance hash.Hash

switch hashType {

case "md4":

hashInstance = md4.New()

case "md5":

hashInstance = md5.New()

case "sha1":

hashInstance = sha1.New()

case "sha256":

hashInstance = sha256.New()

case "sha512":

hashInstance = sha512.New()

case "ripemd160":

hashInstance = ripemd160.New()

}

if isHex {

arr, \_ := hex.DecodeString(text)

hashInstance.Write(arr)

} else {

hashInstance.Write([]byte(text))

}

ciphertext := hashInstance.Sum(nil)

return fmt.Sprintf("%x", ciphertext)

}

13.2 对称加密算法

13.2.1 对称加密的概述

在对称加密算法中，数据发信方将明文（原始数据）和加密密钥一起经过特殊加密算法处理后，使其变成复杂的加密密文发送出去。收信方收到密文后，若想解读原文，则需要使用加密用过的密钥及相同算法的逆算法对密文进行解密，才能使其恢复成可读明文。在对称加密算法中，使用的密钥只有一个，发收信双方都使用这个密钥对数据进行加密和解密，这就要求解密方事先必须知道加密密钥。

对称加密(也叫私钥加密算法)指加密和解密使用相同密钥的加密算法。有时又叫传统密码算法。在大多数的对称算法中，加密密钥和解密密钥是相同的，所以也称这种加密算法为单密钥加密算法。它要求发送方和接收方在安全通信之前，商定一个密钥。对称算法的安全性依赖于密钥，泄漏密钥就意味着任何人都可以对他们发送或接收的消息解密，所以密钥的保密性对通信的安全性至关重要。

优势：对称加密算法的特点是算法公开、计算量小、加密速度快、加密效率高。

不足之处是，交易双方都使用同样钥匙，安全性得不到保证。此外，每对用户每次使用对称加密算法时，都需要使用其他人不知道的惟一钥匙，这会使得发收信双方所拥有的钥匙数量呈几何级数增长，密钥管理成为用户的负担。对称加密算法在分布式网络系统上使用较为困难，主要是因为密钥管理困难，使用成本较高。而与公开密钥加密算法比起来，对称加密算法能够提供加密和认证却缺乏了签名功能，使得使用范围有所缩小。

基于“对称密钥”的加密算法主要有DES、3DES（TripleDES）、AES、RC2、RC4、RC5和Blowfish等。本章将介绍最常用的对称加密算法DES、3DES（TripleDES）和AES。

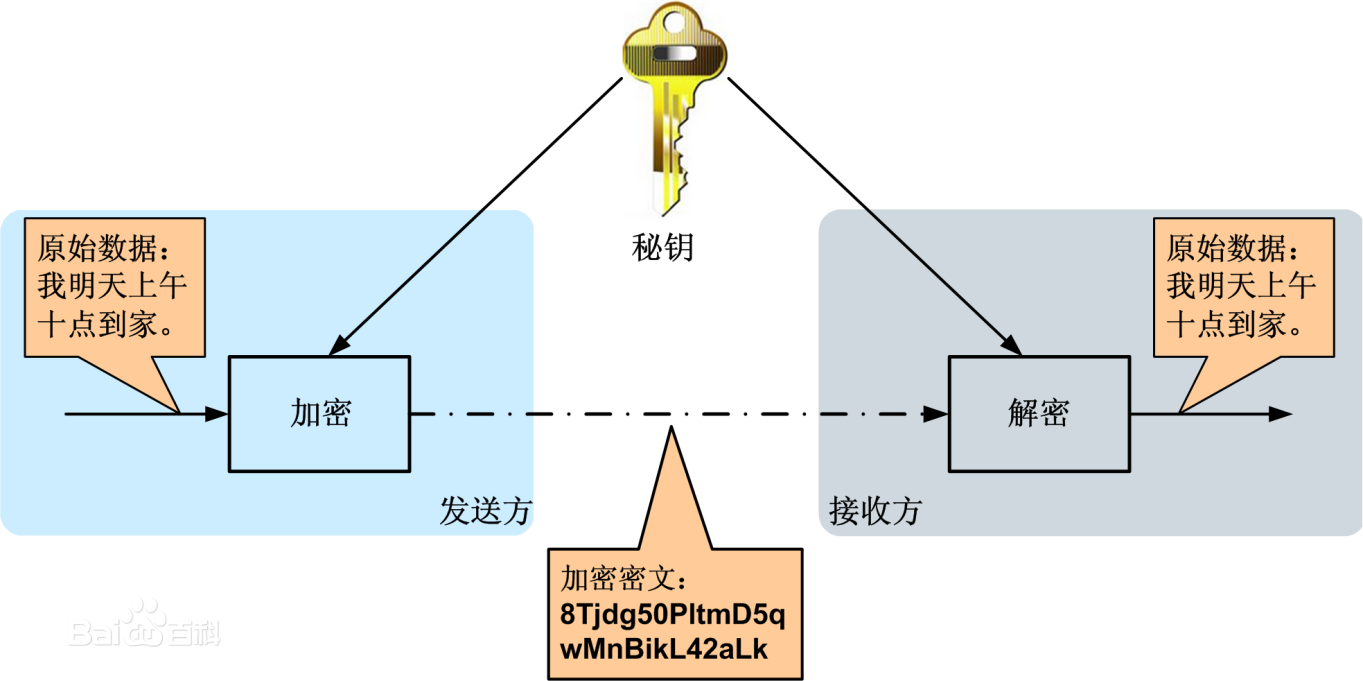


图 13.5

13.2.2 DES和TripleDES算法

DES算法全称为Data Encryption Standard，即数据加密标准算法。

DES是加密和解密使用相同密钥的加密算法，也叫做单密钥算法或私钥加密算法，传统密钥算法。它是IBM公司于1975年研究成功并公开发表的。

DES算法的入口参数有三个：Key、Data、Mode。

其中Key是DES算法的工作密钥，8个字节共64位；

Data是要被加密或被解密的数据；

Mode为DES的工作方式,有两种：加密或解密。

没有密钥的情况下，解密耗费时间非常长，基本上认为没有可能。

加密解密耗时和需要加密的文本大小成正比，这是P问题。

如果知道明文和对应的密文，求解所用的密钥，这是NP问题。

目前还没有NP的求解算法，但是很容易得到验证。想得到NP的解，只能暴力破解（穷举破解）

穷举验证成为对称加密仅有的求解方式，求解时间呈指数级增长。

DES算法把64位的明文输入块变为数据长度为64位的密文输出块，其中8位为奇偶校验位，另外56位作为密码的长度。

首先，DES把输入的64位数据块按位重新组合，并把输出分为L0、R0两部分，每部分各长32位，并进行前后置换，最终由L0输出左32位，R0输出右32位

根据这个法则经过16次迭代运算后，得到L16、R16，将此作为输入，进行与初始置换相反的逆置换，即得到密文输出。

DES算法具有极高的安全性，到目前为止，除了用穷举搜索法对DES算法进行攻击外，还没有发现更有效的办法。

56位长密钥的穷举空间为2^56，这意味着如果一台计算机的速度是每秒种检测100万个密钥，那么它搜索完全部密钥就需要将近2285年的时间，因此DES算法是一种很可靠的加密方法。

3DES密钥是24字节，即192位二进制。

13.2.3 AES加密

高级加密标准（英语：Advanced Encryption Standard，缩写：AES）,AES高级加密标准，是下一代的加密算法标准，速度快，安全级别高，目前 AES 标准的一个实现是Rijndael 算法；

Rijndael（读作rain-dahl）是由美国国家标准与技术协会（NIST）所选的高级加密标准（AES）的算法。

Rijndael由比利时计算机科学家Vincent Rijmen和Joan Daemen开发，近似读音：Rijn [rain] dael [del] (莱恩戴尔)。

这个标准用来替代原先的DES，已经被多方分析且广为全世界所使用。

AES高级加密标准由美国国家标准与技术研究院（NIST）于2001年11月26日发布，并在2002年5月26日成为有效的标准。2006年，高级加密标准已然成为对称密钥加密中最流行的算法之一。

该算法与其他对称密码算法相比更安全、效率更高等特点。

Rijndael是一个反复运算的加密算法，它允许可变动的数据区块及密钥的长度。数据区块与密钥长度的变动是各自独立的。

AES使用128位，192位或者256位的密钥长度（密钥分别是：16字节、24字节、32字节），使得它比密钥长度为56位的DES更健壮可靠。Rijndael被设计用来支持更多的密钥长度，然而除了上述3种密钥长度，其他密钥长度并没有被AES采用。

2^64 = 18446744073709551616

2^64这个数大于全球小麦1000年的产量。如果1微秒验证一个密码（1秒验证100万个），穷举需要费时58万年。

2^256 约= 10 ^ 77

10^80 是当前人类可见宇宙中所有物质原子数目的总和。

13.2.4 AES的加密模式

|  |  |
| --- | --- |
| 加密模式(英文名称及简写) | 中文名称 |
| Electronic Code Book(ECB) | 电子密码本模式 |
| Cipher Block Chaining(CBC) | 密码分组链接模式 |
| Cipher Feedback Mode(CFB) | 加密反馈模式 |
| Output Feedback Mode(OFB) | 输出反馈模式 |

ECB：最基本的加密模式，也就是通常理解的加密，相同的明文将永远加密成相同的密文，无初始向量，容易受到密码本重放攻击，一般情况下很少用。

CBC：明文被加密前要与前面的密文进行异或运算后再加密，因此只要选择不同的初始向量，相同的密文加密后会形成不同的密文，这是目前应用最广泛的模式。CBC加密后的密文是上下文相关的，但明文的错误不会传递到后续分组，但如果一个分组丢失，后面的分组将全部作废(同步错误)。

CFB：类似于自同步序列密码，分组加密后，按8位分组将密文和明文进行移位异或后得到输出同时反馈回移位寄存器，优点最小可以按字节进行加解密，也可以是n位的，CFB也是上下文相关的，CFB模式下，明文的一个错误会影响后面的密文(错误扩散)。

OFB：将分组密码作为同步序列密码运行，和CFB相似，不过OFB用的是前一个n位密文输出分组反馈回移位寄存器，OFB没有错误扩散问题。

13.2.5 核心代码

//1.DES加密解密

//DES加密字节数组，返回字节数组

func DesEncrypt(originalBytes, key []byte) ([]byte, error) {

block, err := des.NewCipher(key)

if err != nil {

return nil, err

}

originalBytes = PKCS5Padding(originalBytes, block.BlockSize())

blockMode := cipher.NewCBCEncrypter(block, key)

cipherArr := make([]byte, len(originalBytes))

blockMode.CryptBlocks(cipherArr, originalBytes)

return cipherArr, nil

}

//DES解密字节数组，返回字节数组

func DesDecrypt(cipherBytes, key []byte) ([]byte, error) {

block, err := des.NewCipher(key)

if err != nil {

return nil, err

}

blockMode := cipher.NewCBCDecrypter(block, key)

originalText := make([]byte, len(cipherBytes))

blockMode.CryptBlocks(originalText, cipherBytes)

originalText = PKCS5UnPadding(originalText)

return originalText, nil

}

//DES加密文本，返回加密后文本

func DesEncryptString(originalText string, key []byte) (string, error) {

cipherArr, err := DesEncrypt([]byte(originalText), key)

if err != nil {

return "", err

}

base64str := base64.StdEncoding.EncodeToString(cipherArr)

return base64str, nil

}

//对加密文本进行DES解密，返回解密后明文

func DesDecryptString(cipherText string, key []byte) (string, error) {

cipherArr, \_ := base64.StdEncoding.DecodeString(cipherText)

cipherArr, err := DesDecrypt(cipherArr, key)

if err != nil {

return "", err

}

return string(cipherArr), nil

}

//2.3DES加密解密

//----------------------------------

// 3DES加密字节数组，返回字节数组

func TripleDesEncrypt(originalBytes, key []byte) ([]byte, error) {

block, err := des.NewTripleDESCipher(key)

if err != nil {

return nil, err

}

originalBytes = PKCS5Padding(originalBytes, block.BlockSize())

// originalBytes = ZeroPadding(originalBytes, block.BlockSize())

blockMode := cipher.NewCBCEncrypter(block, key[:8])

cipherArr := make([]byte, len(originalBytes))

blockMode.CryptBlocks(cipherArr, originalBytes)

return cipherArr, nil

}

// 3DES解密字节数组，返回字节数组

func TripleDesDecrypt(cipherBytes, key []byte) ([]byte, error) {

block, err := des.NewTripleDESCipher(key)

if err != nil {

return nil, err

}

blockMode := cipher.NewCBCDecrypter(block, key[:8])

originalArr := make([]byte, len(cipherBytes))

blockMode.CryptBlocks(originalArr, cipherBytes)

originalArr = PKCS5UnPadding(originalArr)

// origData = ZeroUnPadding(origData)

return originalArr, nil

}

// 3DES加密字符串，返回base64处理后字符串

func TripleDesEncrypt2Str(originalText string, key []byte) (string, error) {

block, err := des.NewTripleDESCipher(key)

if err != nil {

return "", err

}

originalData := PKCS5Padding([]byte(originalText), block.BlockSize())

// originalData = ZeroPadding(originalData, block.BlockSize())

blockMode := cipher.NewCBCEncrypter(block, key[:8])

cipherArr := make([]byte, len(originalData))

blockMode.CryptBlocks(cipherArr, originalData)

cipherText := base64.StdEncoding.EncodeToString(cipherArr)

return cipherText, nil

}

// 3DES解密base64处理后的加密字符串，返回明文字符串

func TripleDesDecrypt2Str(cipherText string, key []byte) (string, error) {

cipherArr, \_ := base64.StdEncoding.DecodeString(cipherText)

block, err := des.NewTripleDESCipher(key)

if err != nil {

return "", err

}

blockMode := cipher.NewCBCDecrypter(block, key[:8])

originalArr := make([]byte, len(cipherArr))

blockMode.CryptBlocks(originalArr, cipherArr)

originalArr = PKCS5UnPadding(originalArr)

// origData = ZeroUnPadding(origData)

return string(originalArr), nil

}

//3.AES加密解密

//AES加密字节数组，返回字节数组

func AesEncrypt(originalBytes, key []byte) ([]byte, error) {

block, err := aes.NewCipher(key)

if err != nil {

return nil, err

}

blockSize := block.BlockSize()

originalBytes = PKCS5Padding(originalBytes, blockSize)

// originalBytes = ZeroPadding(originalBytes, block.BlockSize())

blockMode := cipher.NewCBCEncrypter(block, key[:blockSize])

cipherBytes := make([]byte, len(originalBytes))

// 根据CryptBlocks方法的说明，如下方式初始化crypted也可以

// crypted := originalBytes

blockMode.CryptBlocks(cipherBytes, originalBytes)

return cipherBytes, nil

}

//AES解密字节数组，返回字节数组

func AesDecrypt(cipherBytes, key []byte) ([]byte, error) {

block, err := aes.NewCipher(key)

if err != nil {

return nil, err

}

blockSize := block.BlockSize()

blockMode := cipher.NewCBCDecrypter(block, key[:blockSize])

originalBytes := make([]byte, len(cipherBytes))

// origData := cipherBytes

blockMode.CryptBlocks(originalBytes, cipherBytes)

originalBytes = PKCS5UnPadding(originalBytes)

// origData = ZeroUnPadding(origData)

return originalBytes, nil

}

//AES加密文本，返回对加密后字节数组进行base64处理后字符串

func AesEncryptString(originalText string, key []byte) (string, error) {

cipherBytes, err := AesEncrypt([]byte(originalText), key)

if err != nil {

return "", err

}

base64str := base64.StdEncoding.EncodeToString(cipherBytes)

return base64str, nil

}

//对Base64处理后的加密文本进行DES解密，返回解密后明文

func AesDecryptString(cipherText string, key []byte) (string, error) {

cipherBytes, \_ := base64.StdEncoding.DecodeString(cipherText)

cipherBytes, err := AesDecrypt(cipherBytes, key)

if err != nil {

return "", err

}

return string(cipherBytes), nil

}

//4.填充函数

//对称加密需要的填充函数

func PKCS5Padding(data []byte, blockSize int) []byte {

padding := blockSize - len(data)%blockSize

padtext := bytes.Repeat([]byte{byte(padding)}, padding)

return append(data, padtext...)

}

func PKCS5UnPadding(data []byte) []byte {

length := len(data)

// 去掉最后一个字节 unpadding 次

unpadding := int(data[length-1])

return data[:(length - unpadding)]

}

func ZeroPadding(data []byte, blockSize int) []byte {

padding := blockSize - len(data)%blockSize

padtext := bytes.Repeat([]byte{0}, padding)

return append(data, padtext...)

}

func ZeroUnPadding(data []byte) []byte {

return bytes.TrimRightFunc(data, func(r rune) bool {

return r == rune(0)

})

}

13.3 非对称加密算法

13.3.1 非对称加密算法的发展史

该思想最早由瑞夫·墨克（Ralph C. Merkle）在1974年提出，之后在1976年，惠特菲尔德·迪菲（Whitfield Diffie）与马丁·赫尔曼（Martin Hellman）两位斯坦福的学者以单向函数为基础，创建了DH密码交换算法。

RSA公钥加密算法是1977年美国麻省理工学院的Rivest（里维斯特）、Shamir和Adleman三位教授开发的。RSA取名来自开发他们三者的名字。RSA是目前最有影响力的公钥加密算法，它能够抵抗到目前为止已知的所有密码攻击，已被ISO推荐为公钥数据加密标准。

其他常见的公钥加密算法有：ElGamal、背包算法、Rabin（RSA的特例）、椭圆曲线加密算法（Elliptic Curve Cryptography, ECC）。

13.3.2 非对称加密（Asymmetric Cryptography）

非对称加密又叫做公开密钥加密（Public key cryptography）或公钥加密。

指加密和解密使用不同密钥的加密算法，也称为公私钥加密。

假设两个用户要加密交换数据，双方交换公钥，使用时一方用对方的公钥加密，另一方即可用自己的私钥解密。如果企业中有n个用户，企业需要生成n对密钥，并分发n个公钥。由于公钥是可以公开的，用户只要保管好自己的私钥即可，因此加密密钥的分发将变得 十分简单。同时，由于每个用户的私钥是唯一的，其他用户除了可以可以通过信息发送者的公钥来验证信息的来源是否真实，还可以确保发送者无法否认曾发送过该信息。

非对称加密的缺点是加解密速度要远远慢于对称加密，在某些极端情况下，甚至能比非对称加密慢上1000倍。

公钥加密，需要两个密钥，一个是公开密钥，另一个是私有密钥；一个用作加密的时候，另一个则用作解密。

使用其中一个密钥把明文加密后所得的密文，只能用相对应的另一个密钥才能解密得到原本的明文；甚至连最初用来加密的密钥也不能用作解密。由于加密和解密需要两个不同的密钥，故被称为非对称加密。

不同于加密和解密都使用同一个密钥的对称加密。虽然两个密钥在数学上相关，但如果知道了其中一个，并不能凭此计算出另外一个。因此其中一个可以公开，称为公钥，任意向外发布；不公开的密钥为私钥，必须由用户自行严格秘密保管，绝不通过任何途径向任何人提供。

公钥用来加密消息、验证签名；私钥用来解密信息和进行签名。加密消息的密钥是不能解密消息的。

13.3.3 对称加密与非对称加密的区别

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法类型 | 特点 | 优势 | 缺陷 | 代表算法 |
| 对称加密 | 加解密密钥相同或可推算 | 计算效率高，加密强度高 | 需提前共享密钥；易泄露 | DES、3DES、AES、IDEA |
| 非对称加密 | 加解密密钥不相关 | 无需提前共享密钥 | 计算效率低，中间人攻击可能性低 | RSA、ElGamal、椭圆曲线系列算法ECC |

13.3.4 核心代码

//1.生成RSA密钥文件

func main() {

var bits int

flag.IntVar(&bits, "b", 2048, "密钥长度，默认为1024位")

if err := GenRsaKey(bits); err != nil {

log.Fatal("密钥文件生成失败！")

}

log.Println("密钥文件生成成功！")

}

func GenRsaKey(bits int) error {

// 生成私钥文件

privateKey, err := rsa.GenerateKey(rand.Reader, bits)

if err != nil {

return err

}

derStream := x509.MarshalPKCS1PrivateKey(privateKey)

block := &pem.Block{

Type: "私钥",

Bytes: derStream,

}

file, err := os.Create("private.pem")

if err != nil {

return err

}

err = pem.Encode(file, block)

if err != nil {

return err

}

// 生成公钥文件

publicKey := &privateKey.PublicKey

derPkix, err := x509.MarshalPKIXPublicKey(publicKey)

if err != nil {

return err

}

block = &pem.Block{

Type: "公钥",

Bytes: derPkix,

}

file, err = os.Create("public.pem")

if err != nil {

return err

}

err = pem.Encode(file, block)

if err != nil {

return err

}

return nil

}

//2.RSA加密与解密

var decrypted string

var privateKey, publicKey []byte

func init() {

var err error

flag.StringVar(&decrypted, "d", "", "加密过的数据")

flag.Parse()

publicKey, err = ioutil.ReadFile("public.pem")

if err != nil {

os.Exit(-1)

}

privateKey, err = ioutil.ReadFile("private.pem")

if err != nil {

os.Exit(-1)

}

}

// 加密字节数组，返回字节数组

func RsaEncrypt(origData []byte) ([]byte, error) {

block, \_ := pem.Decode(publicKey)

if block == nil {

return nil, errors.New("public key error")

}

pubInterface, err := x509.ParsePKIXPublicKey(block.Bytes)

if err != nil {

return nil, err

}

pub := pubInterface.(\*rsa.PublicKey)

return rsa.EncryptPKCS1v15(rand.Reader, pub, origData)

}

// 解密字节数组，返回字节数组

func RsaDecrypt(ciphertext []byte) ([]byte, error) {

block, \_ := pem.Decode(privateKey)

if block == nil {

return nil, errors.New("private key error!")

}

priv, err := x509.ParsePKCS1PrivateKey(block.Bytes)

if err != nil {

return nil, err

}

return rsa.DecryptPKCS1v15(rand.Reader, priv, ciphertext)

}

// 加密字符串，返回base64处理的字符串

func RsaEncryptString(origData string) (string, error) {

block, \_ := pem.Decode(publicKey)

if block == nil {

return "", errors.New("public key error")

}

pubInterface, err := x509.ParsePKIXPublicKey(block.Bytes)

if err != nil {

return "", err

}

pub := pubInterface.(\*rsa.PublicKey)

cipherArr, err := rsa.EncryptPKCS1v15(rand.Reader, pub, []byte(origData))

if err != nil {

return "", err

} else {

return base64.StdEncoding.EncodeToString(cipherArr), nil

}

}

// 解密经过base64处理的加密字符串，返回加密前的明文

func RsaDecryptString(cipherText string) (string, error) {

block, \_ := pem.Decode(privateKey)

if block == nil {

return "", errors.New("private key error!")

}

priv, err := x509.ParsePKCS1PrivateKey(block.Bytes)

if err != nil {

return "", err

}

cipherArr, \_ := base64.StdEncoding.DecodeString(cipherText)

originalArr, err := rsa.DecryptPKCS1v15(rand.Reader, priv, cipherArr)

if err != nil {

return "", err

} else {

return string(originalArr), nil

}

}

13.4 椭圆曲线加密算法ECC和椭圆曲线数字签名算法ECDSA

13.4.1 椭圆曲线密码学概述

椭圆曲线密码学（Elliptic curve cryptography，缩写为 ECC），是基于椭圆曲线数学理论实现的一种非对称加密算法。椭圆曲线在密码学中的使用是在1985年由Neal Koblitz和Victor Miller分别独立提出的。

比特币使用椭圆曲线算法生成公钥和私钥，选择的是secp256k1曲线。与RSA（Ron Rivest，Adi Shamir，Len Adleman三位天才的名字）一样，ECC（Elliptic Curves Cryptography，椭圆曲线加密）也属于公开密钥算法。

ECC与RSA算法的优势对比

椭圆曲线公钥系统是代替RSA的强有力的竞争者。

与经典的RSA、DSA等公钥密码体制相比，椭圆密码体制有以下优点：

（1）安全性能更高（ECC可以使用更短的密钥）：

160位ECC加密算法的安全强度相当于1024位RSA加密；

210位ECC加密算法的安全强度相当于2048位RSA加密。

（2）处理速度快：计算量小，处理速度快 在私钥的处理速度上（解密和签名），ECC远 比RSA、DSA快得多。

（3）存储空间占用小： ECC的密钥尺寸和系统参数与RSA、DSA相比要小得多， 所以占用的存储空间小得多。

（4）带宽要求低使得ECC具有广泛的应用前景。

ECC的这些特点使它必将取代RSA，成为通用的公钥加密算法。

13.4.2 数字签名

所谓数字签名(Digital Signature)（又称公开密钥数字签名、电子签章）,是一种类似写在纸上的普通的物理签名，但是使用了公钥加密领域的技术实现，用于鉴别数字信息的方法。

一套数字签名通常定义两种互补的运算，一个用于签名，另一个用于验证。数字签名可以验证数据的来源，可以验证数据传输过程中是否被修改。

数字签名由两部分组成：第一部分是使用私钥（签名密钥）从消息（交易）创建签名的算法； 第二部分是允许任何人验证签名的算法。

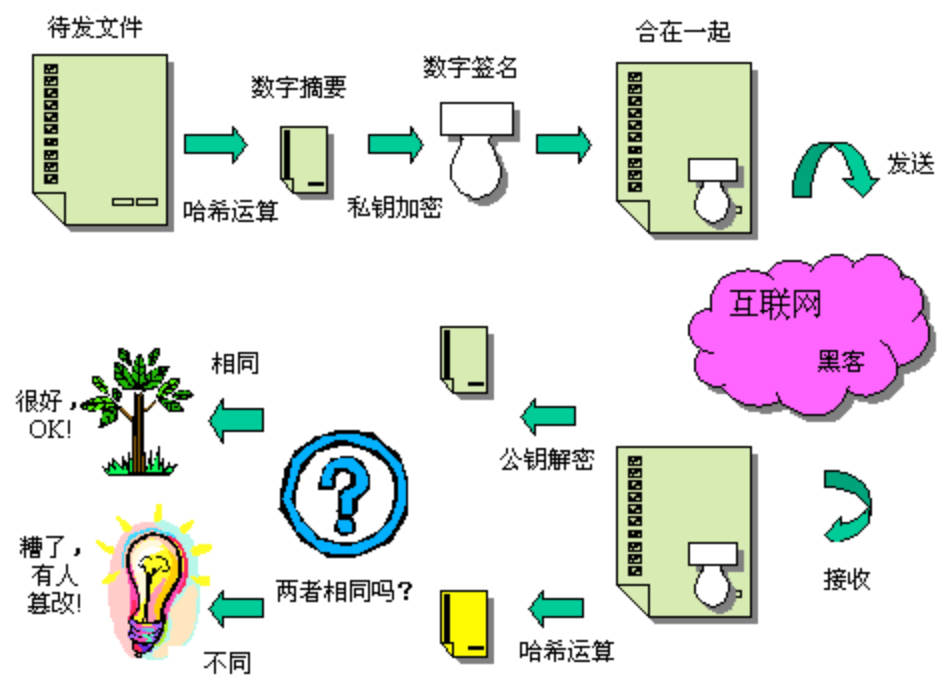


图 13.6

数字签名应该满足如下要求：

* 签名不可伪造性；
* 签名不可抵赖的；
* 签名可信性，签名的识别和应用相对容易，任何人都可以验证签名的有效性；
* 签名是不可复制的，签名与原文是不可分割的整体；
* 签名消息不可篡改，因为任意比特数据被篡改，其签名便被随之改变，那么任何人可以验证而拒绝接受此签名。

数字签名和验证过程

1.只有转账人才能生成的一段防伪造的字符串。通过验证该字符串，一方面证明该交易是转出方本人发起的，另一方面证明交易信息在传输过程中没有被更改。

2.数字签名由：数字摘要和非对称加密技术组成。数字摘要把交易信息hash成固定长度的字符串；再用私钥对hash后的交易信息进行加密形成数字签名。

3.交易中，需要将完整的交易信息和数字签名一起广播给矿工。矿工节点用转账人公钥对签名验证，验证成功说明该交易确实是转账人发起的；矿工节点将交易信息进行hash后与签名中的交易信息摘要进行比对，如果一致，则说明交易信息在传输过程中没有被篡改。

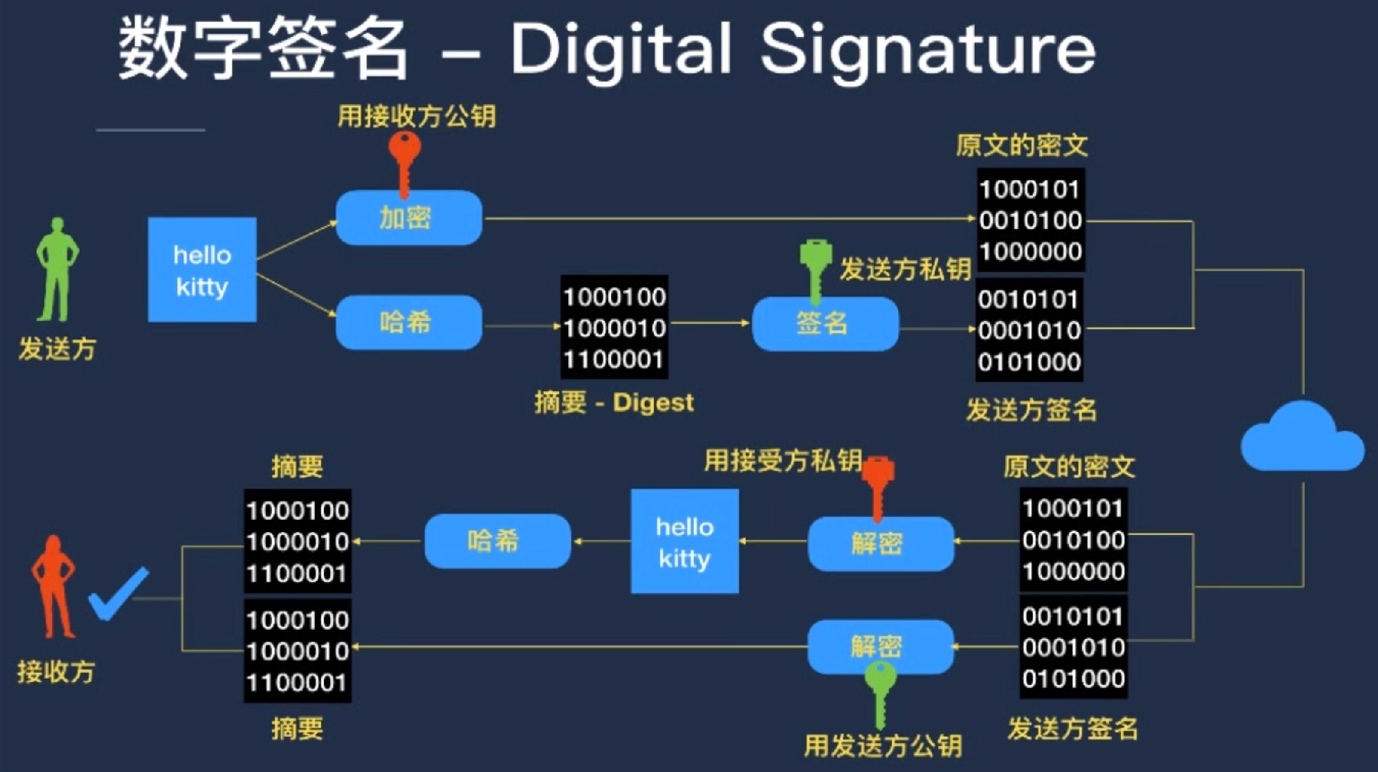


图 13.7

13.4.3 ECC数字签名(ECDSA)核心代码

//生成私钥和公钥，生成的私钥为结构体ecdsa.PrivateKey的指针

func NewKeyPair() (ecdsa.PrivateKey, []byte) {

//生成secp256椭圆曲线

curve := elliptic.P256()

//产生的是一个结构体指针，结构体类型为ecdsa.PrivateKey

private, err := ecdsa.GenerateKey(curve, rand.Reader)

if err != nil {

log.Panic(err)

}

fmt.Printf("私钥：%x\n", private)

fmt.Printf("私钥X：%x\n", private.X.Bytes())

fmt.Printf("私钥Y：%x\n", private.Y.Bytes())

fmt.Printf("私钥D：%x\n", private.D.Bytes())

//x坐标与y坐标拼接在一起，生成公钥

pubKey := append(private.X.Bytes(), private.Y.Bytes()...)

//打印公钥，公钥用16进制打印出来长度为128，包含了x轴坐标与y轴坐标。

fmt.Printf("公钥：%x \n", pubKey)

return \*private, pubKey

}

//生成签名的DER格式

func MakeSignatureDerString(r, s string) string {

// 获取R和S的长度

lenSigR := len(r) / 2

lenSigS := len(s) / 2

// 计算DER序列的总长度

lenSequence := lenSigR + lenSigS + 4

// 将10进制长度转16进制字符串

strLenSigR := DecimalToHex(int64(lenSigR))

strLenSigS := DecimalToHex(int64(lenSigS))

strLenSequence := DecimalToHex(int64(lenSequence))

// 拼凑DER编码

derString := "30" + strLenSequence

derString = derString + "02" + strLenSigR + r

derString = derString + "02" + strLenSigS + s

derString = derString + "01"

return derString

}

//验证签名1

func VerifySig(pubKey, message []byte, r, s \*big.Int) bool {

curve := elliptic.P256()

//公钥的长度

keyLen := len(pubKey)

//前一半为x轴坐标，后一半为y轴坐标

x := big.Int{}

y := big.Int{}

x.SetBytes(pubKey[:(keyLen / 2)])

y.SetBytes(pubKey[(keyLen / 2):])

rawPubKey := ecdsa.PublicKey{curve, &x, &y}

//根据交易哈希、公钥、数字签名验证成功。ecdsa.Verify func Verify(pub \*PublicKey, hash []byte, r \*big.Int, s \*big.Int) bool

res := ecdsa.Verify(&rawPubKey, message, r, s)

return res

}

//验证签名2

func VerifySignature(pubKey, message []byte, r, s string) bool {

curve := elliptic.P256()

//公钥的长度

keyLen := len(pubKey)

//前一半为x轴坐标，后一半为y轴坐标

x := big.Int{}

y := big.Int{}

x.SetBytes(pubKey[:(keyLen / 2)])

y.SetBytes(pubKey[(keyLen / 2):])

rawPubKey := ecdsa.PublicKey{curve, &x, &y}

//根据交易哈希、公钥、数字签名验证成功。ecdsa.Verify func Verify(pub \*PublicKey, hash []byte, r \*big.Int, s \*big.Int) bool

rint := big.Int{}

sint := big.Int{}

rByte, \_ := hex.DecodeString(r)

sByte, \_ := hex.DecodeString(s)

rint.SetBytes(rByte)

sint.SetBytes(sByte)

//fmt.Println("------", rint.SetBytes(rByte))

//fmt.Println("------", sint.SetBytes(sByte))

res := ecdsa.Verify(&rawPubKey, message, &rint, &sint)

return res

}

//验证过程

func main() {

//1、生成签名

fmt.Println("1、生成签名-------------------------------")

//调用函数生成私钥与公钥

privKey, pubKey := cryptotool.NewKeyPair()

//信息的哈希

msg := sha256.Sum256([]byte("hello world"))

//根据私钥和信息的哈希进行数字签名，产生r和s

r, s, \_ := ecdsa.Sign(rand.Reader, &privKey, msg[:])

//生成r、s字符串

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

strSigR := fmt.Sprintf("%x", r)

strSigS := fmt.Sprintf("%x", s)

fmt.Println("r、s的10进制分别为：", r, s)

fmt.Println("r、s的16进制分别为：", strSigR, strSigS)

//r和s拼接在一起，形成数字签名的der格式

signatureDer := cryptotool.MakeSignatureDerString(strSigR, strSigS)

//打印数字签名的16进制显示

fmt.Println("数字签名DER格式为：", signatureDer)

fmt.Println()

//2、签名验证过程

fmt.Println("2、签名验证过程-------------------------------")

res := cryptotool.VerifySig(pubKey, msg[:], r, s)

fmt.Println("签名验证结果：" , res)

res = cryptotool.VerifySignature(pubKey, msg[:], strSigR, strSigS)

fmt.Println("签名验证结果：" , res)

}

13.5 字符编码/解码

13.5.1 Base64

Base64就是一种基于64个可打印字符来表示二进制数据的方法

Base64使用了26个小写字母、26个大写字母、10个数字以及两个符号（例如“+”和“/”），用于在电子邮件这样的基于文本的媒介中传输二进制数据。

Base64通常用于编码邮件中的附件。

Base64字符集：ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

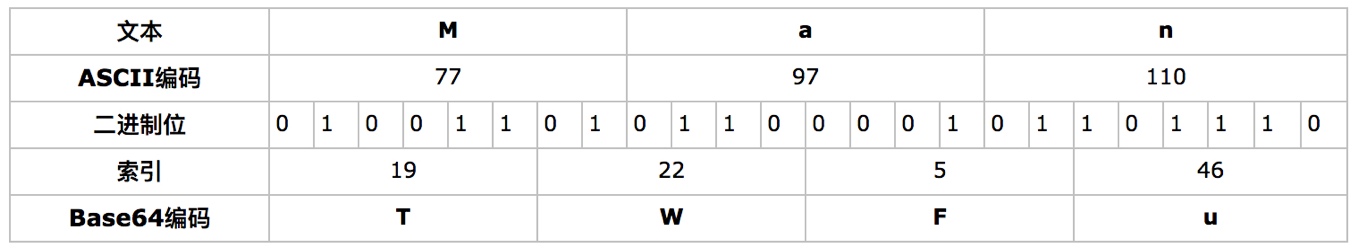


图 13.8

Base64的步骤：

1. 将每个字符转成ASCII编码（10进制）

2. 将10进制编码转成2进制编码

3. 将2进制编码按照6位一组进行平分

4. 将6位一组的2进制数高位补零，然后转成10进制数

5. 将10进制数作为索引，从Base64编码表中查找字符

6. 每3个字符的文本将编码为4个字符长度（3\*8=4\*6）

a. 若文本为3个字符，则正好编码为4个字符长度；

b. 若文本为2个字符，则编码为3个字符，由于不足4个字符，则在尾部用一个“=”补齐；

c. 若文本为1个字符，则编码为2个字符，由于不足4个字符，则在尾部用两个“=”补齐。

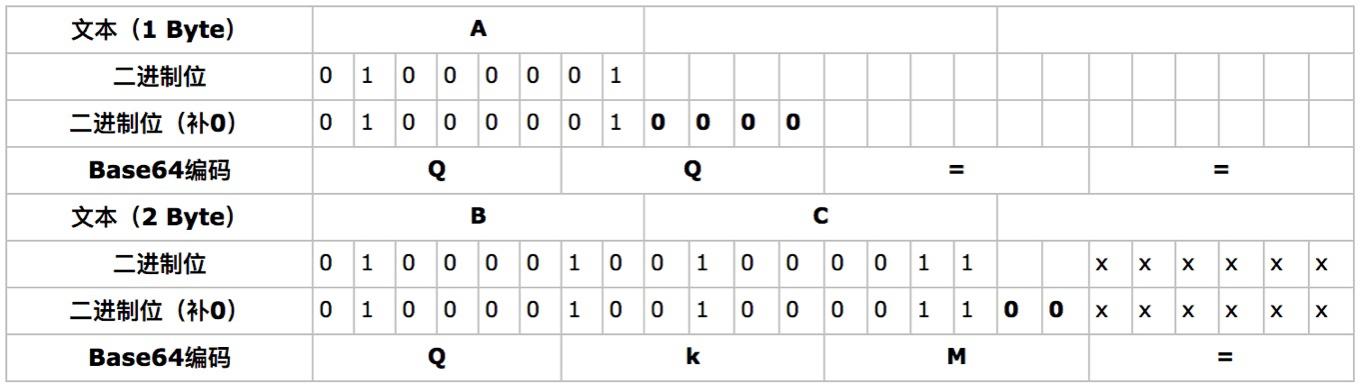


图 13.9

Base64相关的函数：

func Base64EncodeString(str string) string {

return base64.StdEncoding.EncodeToString([]byte(str))

}

func Base64DecodeString(str string) string {

result, \_ := base64.StdEncoding.DecodeString(str)

return string(result)

}

13.5.2 Base58

Base58是一种基于文本的二进制编码格式，用在比特币和其它的加密货币中。这种编码格式不仅实现了数据压缩，保持了易读性，还具有错误诊断功能。

Base58是Base64编码格式的子集，同样使用大小写字母和10个数字，但舍弃了一些容易错读和在特定字体中容易混淆的字符。

Base58不含Base64中的0（数字0）、O（大写字母o）、l（小写字母L）、I（大写字母i），以及“+”和“/”两个字符。目的就是去除容易混淆的字符。

简而言之，Base58就是由不包括（0，O，l，I）的大小写字母和数字组成。

比特币的Base58字母表：123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

13.5.3 Base58Check

Base58Check是一种常用在比特币中的Base58编码格式，增加了错误校验码来检查数据在转录中出现的错误。

校验码长4个字节，添加到需要编码的数据之后。

校验码是从需要编码的数据的哈希值中得到的，所以可以用来检测并避免转录和输入中产生的错误。

使用Base58check编码格式时，编码软件会计算原始数据的校验码并和结果数据中自带的校验码进行对比。二者不匹配则表明有错误产生，那么这个Base58Check格式的数据就是无效的。例如，一个错误比特币地址就不会被钱包认为是有效的地址，否则这种错误会造成资金的丢失。

为了使用Base58Check编码格式对数据（数字）进行编码，首先我们要对数据添加一个称作“版本字节”的前缀，这个前缀用来明确需要编码的数据的类型。

例如，比特币地址的前缀是0（十六进制是0x00），而对私钥编码时前缀是128（十六进制是0x80）。

附加：

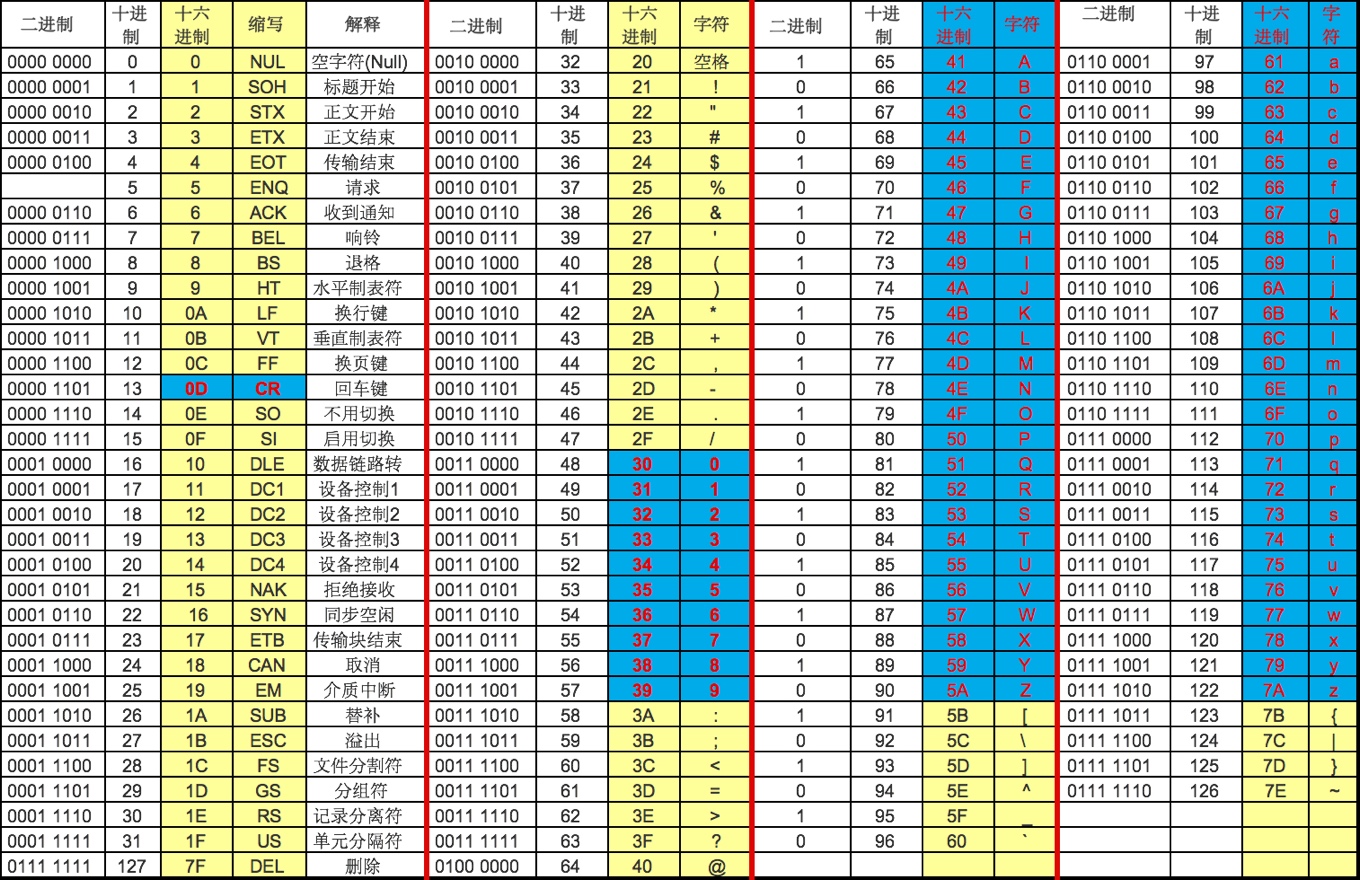


图 13.10

13.5.4 核心代码

var base58Alphabets = []byte("123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz")

// Base58Encode encodes a byte array to Base58

func Base58Encode(input []byte) []byte {

var result []byte

x := big.NewInt(0).SetBytes(input)

base := big.NewInt(int64(len(base58Alphabets)))

zero := big.NewInt(0)

mod := &big.Int{}

for x.Cmp(zero) != 0 {

x.DivMod(x, base, mod)

result = append(result, base58Alphabets[mod.Int64()])

}

// https://en.bitcoin.it/wiki/Base58Check\_encoding#Version\_bytes

if input[0] == 0x00 {

result = append(result, base58Alphabets[0])

}

ReverseBytes(result)

return result

}

// Base58Decode decodes Base58-encoded data

func Base58Decode(input []byte) []byte {

result := big.NewInt(0)

for \_, b := range input {

charIndex := bytes.IndexByte(base58Alphabets, b)

result.Mul(result, big.NewInt(58))

result.Add(result, big.NewInt(int64(charIndex)))

}

decoded := result.Bytes()

if input[0] == base58Alphabets[0] {

decoded = append([]byte{0x00}, decoded...)

}

return decoded

}

13.6 小结

本章学习Go语言加密处理相关的内容。此部分的内容在软件开发过程中使用的频率并没有其他的那么高，但依旧是非常重要的内容，软件的安全问题越来越被重视，用户信息的保密以及数据在网络传递过程中的保密犹为重要，这些场景都离不开各种不同的加密方式，而区块链技术对于加密解密部分尤其依赖，因为虽然使用频率不高，但依旧是重中之重。

第十四章Socket编程

网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个socket。建立网络通信连接至少要一对端口号。比如甲要给乙打电话，通话双方甲乙当于相互通信的2个进程x，电话号码相当于网络地址。任何用户在通话之前，首先要占有一部手机，相当于申请一个Socket；同时要知道对方的号码，相当于对方有一个Socket。然后向对方拨号呼叫，相当于发出连接请求。假如对方空闲（相当于通信的另一主机开机且可以接受连接请求），接听电话，双方就可以正式通话，相当于连接成功。双方通话的过程，是一方向手机发出信号和对方从手机接收信号的过程，相当于向Socket发送数据和从socket接收数据。通话结束后，一方挂断电话相当于关闭Socket，撤消连接。一般用户只能感受到本地和对方号码的存在，建立通话的过程，话音传输的过程以及整个电话系统的技术细节对他都是透明的，这也与Socket机制非常相似。Socket利用网间网通信设施实现进程通信，但它对通信设施的细节毫不关心，只要通信设施能提供足够的通信能力，它就满足了。

本章重点为大家介绍如下的内容：

* 什么是Socket通信
* Dail（）函数
* 网络七层协议
* TCP/IP五层模型
* TCP Socket通信
* 聊天室案例

14.1 Dial()函数

Go语言标准库里提供的net包，支持基于IP层、TCP/UDP层及更高层面（如HTTP、FTP、SMTP）的网络操作，其中用于IP层的称为Raw Socket。net包对于网络I/O提供了便携式接口，包括TCP/IP,UDP，域名解析以及Unix Socket。尽管net包提供了大量访问底层的接口，但是大多数情况下，客户端仅仅只需要最基本的接口，例如Dial，LIsten，Accepte以及分配的conn连接和listener接口。 crypto/tls包使用相同的接口以及类似的Dial和Listen函数。下面对net包进行具体分析。

首先介绍其中常量：

const ( IPv4len = 4 IPv6len = 16 )

很容易看出这表示ip地址的长度（bytes），其中ipv4长度是4，ipv6地址长度是16

接下来讲一下IPv4和IPv6相关知识，了解的请直接跳过。

IP地址（Internet Protocol Address，称为互联网协议地址，简写为IP Address），是一种给主机在Internet上进行编址的方式。常见的IP地址，分为IPv4和IPv6地址两类。

IPv4地址：目前的全球因特网所采用的协议族是TCP/IP协议。IP是TCP/IP协议中网络层的协议，是TCP/IP协议族的核心协议。目前主要采用的IP协议的版本号是4(简称为IPv4)，发展至今已经使用了30多年。

IPv4的地址位数为32位，也就是最多有2的32次方的网络设备可以联到Internet上。近十年来由于互联网的蓬勃发展，IP位址的需求量愈来愈大，使得IP位址的发放愈趋紧张，前一段时间，据报道IPV4的地址已经发放完毕，我们公司目前很多服务器的IP都是一个宝贵的资源。

地址格式类似这样：127.0.0.1 172.122.121.111。IPv4是由四位0~255的十进制数字组成，即从0.0.0.0~255.255.255.255，每组XXX代表小于或等于255的10进制数，IPv4地址总数为232，不过，一些地址是为特殊用途所保留的，如专用网络（约18百万个地址）和多播地址（约270百万个地址）。

此外还有几个特殊IP地址：

0.0.0.0只能做源地址

255.255.255.255是广播地址

127.x.x.x为环回地址，本机使用

专用地址：

10/8 地址范围：10.0.0.0——10.255.255.255，

172.16/12 地址范围：172.16.0.0——172.31.255.255，

192.168/16地址范围：192.168.0.0——192.168.255.255。

IPv6是下一版本的互联网协议，也可以说是下一代互联网的协议，它是为了解决IPv4在实施过程中遇到的各种问题而被提出的，IPv6采用128位地址长度，几乎可以不受限制地提供地址。按保守方法估算IPv6实际可分配的地址，整个地球的每平方米面积上仍可分配1000多个地址。在IPv6的设计过程中除了一劳永逸地解决了地址短缺问题以外，还考虑了在IPv4中解决不好的其它问题，主要有端到端IP连接、服务质量（QoS）、安全性、多播、移动性、即插即用等。

地址格式类似这样：ac:ed:32:aa:12:bf

net包的Dial()函数用于创建网络连接，函数原型如下：

func Dial(net, addr string) (Conn, error)

其中net参数是网络协议的名字，addr参数是IP地址或域名；如连接成功，返回连接对象，否则返回error。

目前，Dial()函数支持如下几种网络协议："tcp"、"udp"、"ip"、"ip6"等，例如：

conn, err := net.Dial("tcp", "192.168.0.123:1234") 　　 // TCP连接

conn, err := net.Dial("udp", "192.168.0.132:4321") 　　// UDP连接

conn, err := net.Dial("ip4:icmp", "www.baidu.com") 　// ICMP连接

在成功连接连接后，可以使用conn的Write()和Read()方法读写数据。

实际上，Dial()函数是对DialTCP()、DialUDP()、DialIP()、DialUnix()函数的封装：

func DialTCP(net string, laddr, raddr \*TCPAddr) (c \*TCPConn, err error)

func DialUDP(net string, laddr, raddr \*UDPAddr) (c \*UDPConn, err error)

func DialIP(netProto string, laddr, raddr \*IPAddr) (c \*IPConn, err error)

func DialUnix(net string, laddr, raddr \*UnixAddr) (c \*UnixConn, err error)

下面是一个TCP示例程序。

例：

func checkError(err error) {

if err != nil {

fmt.Fprintf(os.Stderr, "致命错误: %s", err.Error())

os.Exit(1)

}

}

func readFully(conn net.Conn) ([]byte, error) {

defer conn.Close()

result := bytes.NewBuffer(nil)

var buf [512]byte

for {

n, err := conn.Read(buf[0:])

result.Write(buf[0:n])

if err != nil {

if err == io.EOF {

Break

}

return nil, err

}

}

return result.Bytes(), nil

}

func main() {

addr := "127.0.0.1:80"

conn, err := net.Dial("tcp", addr)

checkError(err)

\_, err = conn.Write([]byte("GET /api/v3/get HTTP/1.1\r\n\r\n"))

checkError(err)

result, err := readFully(conn)

checkError(err)

fmt.Println(string(result))

os.Exit(0)

}

14.2 TCP Socket 通信

网络七层协议由下往上分别为物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。其中物理层、数据链路层和网络层通常被称作媒体层，是网络工程师所研究的对象；传输层、会话层、表示层和应用层则被称作主机层，是用户所面向和关心的内容。

HTTP协议对应于应用层，TCP协议对应于传输层，IP协议对应于网络层，HTTP协议是基于TCP连接的,三者本质上没有可比性。 TCP/IP是传输层协议，主要解决数据如何在网络中传输；而HTTP是应用层协议，主要解决如何包装数据。Socket是应用层与TCP/IP协议族通信的中间软件抽象层，是它的一组接口。

TCP/IP五层模型的协议分为：应用层、传输层、网络层、数据链路层和物理层。中继器、集线器、还有我们通常说的双绞线也工作在物理层；网桥（现已很少使用）、以太网交换机（二层交换机）、网卡（其实网卡是一半工作在物理层、一半工作在数据链路层）在数据链路层；路由器、三层交换机在网络层；传输层主要是四层交换机、也有工作在四层的路由器。

TCP/IP协议中的应用层处理七层模型中的第五层、第六层和第七层的功能。TCP/IP协议中的传输层并不能总是保证在传输层可靠地传输数据包，而七层模型可以做到。TCP/IP协议还提供一项名为UDP（用户数据报协议）的选择。UDP不能保证可靠的数据包传输。

以前使用Socket编程时，一般是如下步骤：

* 建立连接
* 绑定socket
* 监听
* 接受连接
* 接受/发送

服务器端代码：

package main

import (

"net"

"fmt"

)

func main() {

//第一步：提供本机的地址

tcpAddr, \_ := net.ResolveTCPAddr("tcp4", "10.0.154.238:9527")

//第二步：提供监听对象

listener, \_ := net.ListenTCP("tcp", tcpAddr)

//第三步：监听端口，等待客户端链接

fmt.Println("服务器端已经准备就绪，等待客户端链接......")

for{

conn, \_ := listener.Accept()

fmt.Println("已有客户端%s接入......", conn.RemoteAddr())

//第四步：读写数据

//服务器读取客户端

go handleData(conn)

}

}

func handleData(conn net.Conn){

for {

bs := make([] byte, 512)

n, \_ := conn.Read(bs) //阻塞的

content:=string(bs[:n])

if content=="over" ||n == 0{

fmt.Println("客户端已经断开连接......")

break

}

fmt.Println("客户端说：", content, n)

//写给客户端

conn.Write([] byte(content))

}

//第五步：关闭

conn.Close()

}

客户端代码：

package main

import (

"net"

"fmt"

)

func main() {

//第一步：提供服务器地址

tcpAddr, \_ := net.ResolveTCPAddr("tcp4", "10.0.154.238:9527")

//第二步：连接服务器

tcpConn, \_ := net.DialTCP("tcp", nil, tcpAddr)

//\*TCPConn

//数据交互

handleData2(tcpConn)

}

func handleData2(tcpConn \*net.TCPConn){

//第三步：读写

for {

//1）读取键盘输入

line := ""

fmt.Scanln(&line) //键盘输入，阻塞的

fmt.Println("--",line,"--")

//2）发送给服务器

n,\_:=tcpConn.Write([] byte(line))

fmt.Println("客户端写出数据：",n)

if line == "over"{

fmt.Println("客户端即将结束......")

break

}

//读取服务器

bs := make([] byte, 512)

n,\_= tcpConn.Read(bs) //阻塞式

fmt.Println("服务器说：", string(bs[:n]))

}

//第四步：关闭

tcpConn.Close()

}

14.3 聊天室案例

场景一：一对一聊天

服务器端代码：

package main

import (

"fmt"

"net"

"os"

)

func main() {

//第一步：定义服务端的端口：

ip := ":1234"

//第二步：获取服务器端的地址

tcpAddr, err := net.ResolveTCPAddr("tcp4", ip)

checkErr(err)

fmt.Printf("%T\n", tcpAddr) //\*net.TCPAddr

fmt.Println(tcpAddr) // :1234

// 第三步：服务端需要监听该端口

listener, err := net.ListenTCP("tcp4", tcpAddr)

fmt.Printf("%T\n", listener) //\*net.TCPListener

fmt.Println(listener)

// 第四步：等待客户端的链接

fmt.Println("等待客户端的链接......")

map1 := make(map[string]net.Conn)

done := make(chan bool)

for i := 0; i < 2; i++ {

conn, \_ := listener.Accept()

// 获取远程客户端的信息

remoAddr := conn.RemoteAddr()

fmt.Println("已有客户端接入：", i, remoAddr)

map1[remoAddr.String()] = conn

fmt.Println(map1)

go sendData3(conn, map1, done)

}

}

func sendData3(conn net.Conn, map1 map[string]net.Conn, done chan bool) {

bs := make([]byte, 512)

sendName := conn.RemoteAddr().String()

for {

count, \_ := conn.Read(bs)

content := string(bs[:count])

fmt.Println(sendName, "发来消息：", content)

if content == "over" {

//conn.Close()

fmt.Println(sendName, "已经断开于服务器连接!")

delete(map1, sendName)

fmt.Println(map1)

break

}

for key, v := range map1 {

if key == sendName {

continue

}

msg := sendName + "说：" + content

v.Write([]byte(msg))

}

}

if len(map1) == 0 {

done <- true

}

}

func checkErr(err error) {

if err != nil {

fmt.Println("连接失败......")

os.Exit(1) //程序退出，非0表示不正常退出

}

}

客户端代码：

package main

import (

"fmt"

"net"

"os"

)

func main() {

//1.确定要访问的服务端的ip和端口

service := "127.0.0.1:1234"

tcpAddr, err := net.ResolveTCPAddr("tcp4", service)

fmt.Printf("%T\n", tcpAddr) //\*net.TCPAddr

fmt.Println(tcpAddr) //192.168.1.128:1234

checkErr(err)

// 2.客户端申请链接服务器

tcpConn, err := net.DialTCP("tcp4", nil, tcpAddr)

fmt.Printf("%T\n", tcpConn) //\*net.TCPConn

fmt.Println(tcpConn) //\*net.TCPConn

// 3.向服务器发送消息

go sendData(tcpConn)

line := ""

for {

fmt.Scanln(&line)

tcpConn.Write([]byte(line))

fmt.Println("客户端数据写完!")

if line == "" || line == "over" {

fmt.Println("客户端已经断开于服务器连接!")

break

}

}

tcpConn.Close()

}

func sendData(tcpConn \*net.TCPConn) {

bs := make([]byte, 512)

for {

count, \_ := tcpConn.Read(bs)

if count == 0 {

fmt.Println("退出！")

break

}

fmt.Println(string(bs[:count]), count)

}

}

func checkErr(err error) {

if err != nil {

fmt.Println("连接服务器失败！", err)

os.Exit(0)

}

}

场景二：一对多聊天

服务器端代码：

package main

import (

"fmt"

"net"

"os"

)

func main() {

//1.定义服务端的端口：

ip := ":1234"

//2：获取服务器端的地址

tcpAddr, err := net.ResolveTCPAddr("tcp4", ip)

checkErr(err)

fmt.Printf("%T\n", tcpAddr) //\*net.TCPAddr

fmt.Println(tcpAddr) // :1234

//3：服务端需要监听该端口

listener, err := net.ListenTCP("tcp4", tcpAddr)

fmt.Printf("%T\n", listener) //\*net.TCPListener

//4：等待客户端的链接

fmt.Println("服务器准备就绪，等待客户端连接......")

map1 := make(map[string]net.Conn)

for {

conn, \_ := listener.Accept()

// 获取远程客户端的信息

remoAddr := conn.RemoteAddr()

fmt.Println("已有客户端接入：", remoAddr)

map1[remoAddr.String()] = conn

fmt.Println(map1)

go sendData3(conn, map1)

}

}

func sendData3(conn net.Conn, map1 map[string]net.Conn) {

bs := make([]byte, 512)

sendName := conn.RemoteAddr().String()

for {

count, \_ := conn.Read(bs)

content := string(bs[:count])

fmt.Println(sendName, "发来消息：", content)

if content == "over" {

fmt.Println(sendName, "已经断开于服务器连接")

delete(map1, sendName)

fmt.Println(map1)

break

}

for key, v := range map1 {

if key == sendName {

continue

}

msg := sendName + "说：" + content

v.Write([]byte(msg))

}

}

}

func checkErr(err error) {

if err != nil {

fmt.Println("服务器发生错误！")

os.Exit(1) //程序退出，非0表示不正常退出

}

}

客户端代码：

package main

import (

"fmt"

"net"

"os"

)

func main() {

//1.确定要访问的服务端的ip和端口

service := "127.0.0.1:1234"

tcpAddr, err := net.ResolveTCPAddr("tcp4", service)

fmt.Printf("%T\n", tcpAddr) //\*net.TCPAddr

fmt.Println(tcpAddr)

checkErr(err)

// 2.客户端申请链接服务器

tcpConn, err := net.DialTCP("tcp4", nil, tcpAddr)

fmt.Printf("%T\n", tcpConn) //\*net.TCPConn

fmt.Println(tcpConn) //\*net.TCPConn

// 3.向服务器发送消息

go sendData(tcpConn)

line := ""

for {

fmt.Scanln(&line)

tcpConn.Write([]byte(line))

fmt.Println("客户端数据写完!")

if line == "" || line == "over" {

fmt.Println("客户端退出程序!")

break

}

}

//关闭

tcpConn.Close()

}

func sendData(tcpConn \*net.TCPConn) {

bs := make([]byte, 512)

for {

count, \_ := tcpConn.Read(bs)

if count == 0 {

fmt.Println("退出！")

break

}

fmt.Println(string(bs[:count]), count)

}

}

func checkErr(err error) {

if err != nil {

fmt.Println("错误了！", err)

os.Exit(0)

}

}

14.4 小结

本章学习Go语言Socket编程的内容，本章相对于其它章节来说，很少会用得到，但其依旧是网络编程中非常重要的内容。建议读者在了解的基础之上，能够再深入的动手实践一些。

第十五章 HTTP编程

本章重点为大家介绍如下的内容：

* 什么是HTTP协议
* HTTP状态码有哪些
* HTTP客户端请求方法
* GET方法和POST方法的区别
* HTTP服务器端实现
* HTTP客户端实现
* 服务器端如何使用ServeMux分发任务
* 服务器端如何获取客户端数据
* 常见案例

15.1 HTTP协议

15.1.1 HTTP协议介绍

HTTP协议（HyperText Transfer Protocol，超文本传输协议）是用于从WWW服务器传输超文本到本地浏览器的传送协议，定义了客户端和服务端之间请求和响应的传输标准。它可以使浏览器更加高效，使网络传输减少。它不仅保证计算机正确快速地传输超文本文档，还确定传输文档中的哪一部分，以及哪部分内容首先显示(如文本先于图形)等。HTTP是一个应用层协议，由请求和响应构成，是一个标准的客户端服务器模型。HTTP是一个无状态的协议。

HTTP协议通常承载于TCP协议之上，有时也承载于TLS或SSL协议层之上，这个时候，就成了我们常说的HTTPS。HTTPS协议是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比HTTP协议安全。

HTTPS和HTTP的区别主要如下：

HTTPS协议需要到CA申请证书，一般免费证书较少，因而需要一定费用。

HTTP是超文本传输协议，信息是明文传输，HTTPS则是具有安全性的ssl加密传输协议。

HTTP和HTTPS使用的是完全不同的连接方式，用的端口也不同。

HTTP的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比HTTP协议安全。

HTTP默认的端口号为80，HTTPS的端口号为443。

HTTP协议是一个无状态的协议，同一个客户端的这次请求和上次请求是没有对应关系。

15.1.2HTTP状态码

HTTP状态码（HTTP Status Code）是用以表示网页服务器HTTP响应状态的数字代码。当我们从客户端向服务器发送请求时，服务器向我们返回状态码，状态码就是告诉我们服务器响应的状态 ，通过它，我们就可以知道当前请求是成功了还是出现了什么问题。

状态码是由3位数字和原因短语组成的（比如最常见的：200 OK） ，其中第一位数字表示响应类别，响应类别从1到5分为五种 。

|  |  |  |
| --- | --- | --- |
| 状态码 | 响应类别 | 原因短语 |
| 1XX | 信息性状态码（Informational） | 服务器正在处理请求 |
| 2XX | 成功状态码（Success） | 请求已正常处理完毕 |
| 3XX | 重定向状态码（Redirection） | 需要进行额外操作以完成请求 |
| 4XX | 客户端错误状态码（Client Error） | 客户端原因导致服务器无法处理请求 |
| 5XX | 服务器错误状态码（Server Error） | 服务器原因导致处理请求出错 |

状态码响应类别的的定义是必须遵守的，后面的两个数字服务器自己DIY也没什么问题

下面列举了一些状态码，作为程序员应该了解。

2开头 （请求成功）表示成功处理了请求的状态代码。

|  |  |  |
| --- | --- | --- |
| 状态码 | 响应类别 | 原因短语 |
| 200 | 成功 | 服务器已成功处理了请求。 |
| 201 | 已创建 | 请求成功并且服务器创建了新的资源。 |
| 202 | 已接受 | 服务器已接受请求，但尚未处理。 |
| 203 | 非授权信息 | 服务器已成功处理了请求，但返回的信息可能来自另一来源。 |
| 204 | 无内容 | 服务器成功处理了请求，但没有返回任何内容。 |
| 205 | 重置内容 | 服务器成功处理了请求，但没有返回任何内容。 |
| 206 | 部分内容 | 服务器成功处理了部分GET请求。 |

3开头 （请求被重定向）表示要完成请求，需要进一步操作。 通常，这些状态代码用来重定向。

|  |  |  |
| --- | --- | --- |
| 状态码 | 响应类别 | 原因短语 |
| 300 | 多种选择 | 针对请求，服务器可执行多种操作。服务器可根据请求者(user agent)选择一项操作，或提供操作列表供请求者选择。 |
| 301 | 永久移动 | 请求的网页已永久移动到新位置。服务器返回此响应（对GET或HEAD请求的响应）时，会自动将请求者转到新位置。 |
| 302 | 临时移动 | 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。 |
| 303 | 查看其他位置 | 请求者应当对不同的位置使用单独的GET请求来检索响应时，服务器返回此代码。 |
| 304 | 未修改 | 自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容。 |
| 305 | 使用代理 | 请求者只能使用代理访问请求的网页。如果服务器返回此响应，还表示请求者应使用代理。 |
| 307 | 临时重定向 | 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。 |

4开头 （请求错误）这些状态代码表示请求可能出错，妨碍了服务器的处理。

|  |  |  |  |
| --- | --- | --- | --- |
| 状态码 | 响应类别 | | 原因短语 |
| 400 | | 错误请求 | 服务器不理解请求的语法。 |
| 401 | | 未授权 | 请求要求身份验证。对于需要登录的网页，服务器可能返回此响应。 |
| 403 | | 禁止 | 服务器拒绝请求。 |
| 404 | | 未找到 | 服务器找不到请求的网页。 |
| 405 | | 方法禁用 | 禁用请求中指定的方法。 |
| 406 | | 不接受 | 无法使用请求的内容特性响应请求的网页。 |
| 407 | | 需要代理授权 | 此状态代码与401（未授权）类似，但指定请求者应当授权使用代理。 |
| 408 | | 请求超时 | 服务器等候请求时发生超时。 |
| 409 | | 冲突 | 服务器在完成请求时发生冲突。服务器必须在响应中包含有关冲突的信息。 |
| 410 | | 已删除 | 如果请求的资源已永久删除，服务器就会返回此响应。 |
| 411 | | 需要有效长度 | 服务器不接受不含有效内容长度标头字段的请求。 |
| 412 | | 未满足前提条件 | 服务器未满足请求者在请求中设置的其中一个前提条件。 |
| 413 | | 请求实体过大 | 服务器无法处理请求，因为请求实体过大，超出服务器的处理能力。 |
| 414 | | 请求的URI过长 | 请求的URI（通常为网址）过长，服务器无法处理。 |
| 415 | | 不支持的媒体类型 | 请求的格式不受请求页面的支持。 |
| 416 | | 请求范围不符合要求 | 如果页面无法提供请求的范围，则服务器会返回此状态代码。 |
| 417 | | 未满足期望值 | 服务器未满足"期望"请求标头字段的要求。 |

5开头（服务器错误）这些状态代码表示服务器在尝试处理请求时发生内部错误。 这些错误可能是服务器本身的错误，而不是请求出错。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 状态码 | 响应类别 | | 原因短语 | |
| 500 | | 服务器内部错误 | | 服务器遇到错误，无法完成请求。 |
| 501 | | 尚未实施 | | 服务器不具备完成请求的功能。 |
| 502 | | 错误网关 | | 服务器作为网关或代理，从上游服务器收到无效响应。 |
| 503 | | 服务不可用 | | 服务器目前无法使用（由于超载或停机维护）。通常，这只是暂时状态。 |
| 504 | | 网关超时 | | 服务器作为网关或代理，但是没有及时从上游服务器收到请求。 |
| 505 | | HTTP版本不受支持 | | 服务器不支持请求中所用的HTTP协议版本。 |

15.2 HTTP客户端

Go语言中提供了net/HTTP库来实现网络请求，我们在import中需要导入我们用到的库。

HTTP请求方法:

HTTP请求方法指定了客户端想对指定的资源/服务器作何种操作，本书重点讲解其中两种GET方法和POST方法。

GET方法用来请求已被URI识别的资源。指定的资源经服务器端解析后返回响应内容（也就是说，如果请求的资源是文本，那就保持原样返回；如果是CGI[通用网关接口]那样的程序，则返回经过执行后的输出结果）。最常用于向服务器查询某些信息。必要时，可以将查询字符串参数追加到URL末尾，以便将信息发送给服务器。golang要请求远程网页，可以使用net/HTTP包中的client提供的方法实现。

15.2.1 GET方法示例

func httpGet() {

resp, err := http.Get("http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName?theCityName=北京")

iferr != nil{

// handle error

}

deferresp.Body.Close()

body, err := ioutil.ReadAll(resp.Body)

iferr != nil{

// handle error

}

fmt.Println(string(body))

}

15.2.2 http.NewRequest示例

package \_client

import (

"fmt"

"io/ioutil"

"log"

"net/http"

)

func main() {

urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName?theCityName=北京"

client := http.Client{}

request, err := http.NewRequest("GET", urlStr, nil)

if err != nil {

log.Fatal(err)

}

response, err := client.Do(request)

if err != nil {

log.Fatal(err)

}

defer response.Body.Close()

if response.StatusCode == 200 {

body, err := ioutil.ReadAll(response.Body)

if err != nil {

log.Fatal(err)

}

fmt.Println(string(body))

}

fmt.Printf("response:%+v\n", response)

fmt.Printf("response.Header:%+v\n", response.Header)

fmt.Printf("response.Cookies:%+v\n", response.Cookies())

}

15.2.3 http.clinet.Get示例

func main() {

urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName?theCityName=北京"

clinet := http.Client{}

response, err := clinet.Get(urlStr)

if err != nil {

log.Fatal(err)

}

defer response.Body.Close()

if response.StatusCode == 200 {

body, err := ioutil.ReadAll(response.Body)

if err != nil {

log.Fatal(err)

}

fmt.Println(string(body))

}

fmt.Printf("%+v", response)

}

15.2.4 POST方法：

POST方法用来传输实体的主体。虽然用GET方法也可以传输实体的主体，但一般不用GET方法进行传输，而是用POST方法；虽然GET方法和POST方法很相似，但是POST的主要目的并不是获取响应的主体内容。POST请求的主体可以包含非常多的数据，而且格式不限。

1、http.Post示例1：

func httpPost() {

resp, err := http.Post("http://localhost/test/post",

"application/x-www-form-urlencoded",

strings.NewReader("id=1"))

if err != nil {

fmt.Println(err)

}

deferresp.Body.Close()

body, err := ioutil.ReadAll(resp.Body)

if err != nil {

// handle error

}

fmt.Println(string(body))

}

注意：使用这个方法的话，第二个参数要设置成”application/x-www-form-urlencoded”，否则post参数无法传递。

HTTP.Post示例2：

package main

import (

"bytes"

"fmt"

"io/ioutil"

"log"

"net/http"

"net/url"

)

func main() {

urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName"

param := url.Values{

"theCityName": {"北京"},

}

requestBody := bytes.NewBufferString(param.Encode())

response, err := http.Post(urlStr, "application/x-www-form-urlencoded", requestBody)

if err != nil {

log.Fatal(err)

}

defer response.Body.Close()

if response.StatusCode == 200 {

body, err := ioutil.ReadAll(response.Body)

if err != nil {

log.Fatal(err)

}

fmt.Println(string(body))

}

fmt.Printf("%+v\n", response)

fmt.Printf("%+v\n", response.Body)

fmt.Printf("%+v\n", response.Header)

fmt.Printf("%+v\n", response.StatusCode)

fmt.Printf("%+v\n", response.Status)

fmt.Printf("%+v\n", response.Request)

}

15.2.5 http.NewRequest示例：

package main

import (

"bytes"

"fmt"

"io/ioutil"

"log"

"net/http"

"net/url"

)

func main() {

urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName"

client := http.Client{}

param := url.Values{

"theCityName": {"北京"},

}

requestBody := bytes.NewBufferString(param.Encode())

request, err := http.NewRequest("POST", urlStr, requestBody)

if err != nil {

log.Fatal(err)

}

cookId := &http.Cookie{Name: "id", Value: "1", Domain: "baidu.com"}

cookName := &http.Cookie{Name: "name", Value: "go"}

request.AddCookie(cookId)

request.AddCookie(cookName)

//使用text/html就会出错

request.Header.Set("Content-Type", "application/x-www-form-urlencoded")

response, err := client.Do(request)

if err != nil {

log.Fatal(err)

}

defer response.Body.Close()

if response.StatusCode == 200 {

body, err := ioutil.ReadAll(response.Body)

if err != nil {

log.Fatal(err)

}

fmt.Println(string(body))

}

fmt.Printf("response:%+v\n", response)

fmt.Printf("response.Header:%+v\n", response.Header)

fmt.Printf("response.Cookies:%+v\n", response.Cookies())

fmt.Printf("request.Header:%+v\n", request.Header)

fmt.Printf("request.Cookies:%+v\n", request.Cookies())

}

15.2.6 http.client示例：

package main

import (

"bytes"

"fmt"

"io/ioutil"

"log"

"net/http"

"net/url"

)

func main() {

urlStr := "http://www.webxml.com.cn/WebServices/WeatherWebService.asmx/getWeatherbyCityName"

clinet := http.Client{}

param := &url.Values{

"theCityName": {"北京"},

}

requestBody := bytes.NewBufferString(param.Encode())

response, err := clinet.Post(urlStr, "application/x-www-form-urlencoded", requestBody)

if err != nil {

log.Fatal(err)

}

defer response.Body.Close()

if response.StatusCode == 200 {

body, err := ioutil.ReadAll(response.Body)

if err != nil {

log.Fatal(err)

}

fmt.Println(string(body))

}

fmt.Printf("%+v", response)

}

15.2.7 http.PostForm方法示例

func httpPostForm() {

resp, err := http.PostForm("http://www.baidu.com/test/post.html",

url.Values{"key": {"Value"}, "id": {"1"}})

if err != nil {

// handle error

}

deferresp.Body.Close()

body, err := ioutil.ReadAll(resp.Body)

if err != nil {

// handle error

}

fmt.Println(string(body))

}

15.2.8 复杂的请求

有时需要在请求的时候设置头参数、cookie之类的数据，就可以使用http.Do方法。

func httpDo() {

client := &http.Client{}

req, err := http.NewRequest("POST", "http://www.baidu.com", strings.NewReader("id=1"))

iferr != nil{

// handle error

}

req.Header.Set("Content-Type", "application/x-www-form-urlencoded")

req.Header.Set("Cookie", "name=go")

resp, err := client.Do(req)

deferresp.Body.Close()

body, err := ioutil.ReadAll(resp.Body)

iferr != nil{

// handle error

}

fmt.Println(string(body))

}

同上面的post请求，必须要设定Content-Type为application/x-www-form-urlencoded，post参数才可正常传递。

15.2.9 GET方法和POST方法区别

下面的表格比较了两种 HTTP 方法：GET 和 POST

|  |  |  |
| --- | --- | --- |
|  | GET | POST |
| 后退按钮/刷新 | 无害 | 数据会被重新提交（浏览器应该告知用户数据会被重新提交） |
| 书签 | 可收藏为书签 | 不可收藏为书签 |
| 缓存 | 能被缓存 | 不能缓存 |
| 编码类型 | application/x-www-form-rulencoded | application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码 |
| 历史 | 参数保留在浏览器历史中 | 参数不会保留在浏览器历史中 |
| 对数据长度的限制 | 是的。当发送数据时，GET方法向URL添加数据；URL的长度是受限制的（URL的最大长度是2048个字符） | 无限制 |
| 对数据类型的限制 | 只允许ASCII字符 | 没有限制。也允许二进制数据 |
| 安全性 | 与POST相比，GET安全性较差，因为发送的数据是URL的一部分。在发送密码或其他敏感信息时绝不要使用GET | POST比GET更安全，因为参数不会被保存在浏览器历史或WEB服务器日志中 |

15.3 HTTP服务端

Go语言标准库内建提供了net/http包，涵盖了HTTP客户端和服务端的具体实现。使用net/http包，我们可以很方便地编写HTTP客户端或服务端的程序。

下面看以下几个服务器端的例子：

例1：

package main

import (

"fmt"

"net/http"

)

//服务端编写的业务逻辑处理程序

//hander函数： 具有func(w http.ResponseWriter, r \*http.Requests)签名的函数

func myHandler(w http.ResponseWriter, r \*http.Request) {

fmt.Println(r.RemoteAddr, "连接成功") //r.RemoteAddr远程网络地址

fmt.Println("method = ", r.Method) //请求方法

fmt.Println("url = ", r.URL.Path)

fmt.Println("header = ", r.Header)

fmt.Println("body = ", r.Body)

w.Write([]byte("hello world")) //给客户端回复数据

}

func main() {

http.HandleFunc("/go", myHandler)

//该方法用于在指定的 TCP 网络地址 addr 进行监听，然后调用服务端处理程序来处理传入的连接请求。

//该方法有两个参数：第一个参数 addr 即监听地址；第二个参数表示服务端处理程序，通常为空

//第二个参数为空意味着服务端调用 http.DefaultServeMux 进行处理

http.ListenAndServe("127.0.0.1:1234", nil)

}

客户端测试1:

在浏览器中输入：http://127.0.0.1:1234/go

结果： hello world

客户端测试2:

package main

import (

"fmt"

"io/ioutil"

"net/http"

)

func main() {

response, \_ := http.Get("http://127.0.0.1:1234/go")

defer response.Body.Close()

body, \_ := ioutil.ReadAll(response.Body)

fmt.Println(string(body))

}

结果： hello world

例2：

package main

import (

"fmt"

"log"

"net/http"

"strings"

)

func sayhelloName(w http.ResponseWriter, r \*http.Request) {

r.ParseForm() //解析参数，默认是不会解析的

fmt.Println(r.Form) //这些信息是输出到服务器端的打印信息

fmt.Println("path", r.URL.Path)

fmt.Println("scheme", r.URL.Scheme)

fmt.Println(r.Form["url\_long"])

for k, v := range r.Form {

fmt.Println("key:", k)

fmt.Println("val:", strings.Join(v, ""))

}

fmt.Fprintf(w, "Hello Wrold!") //这个写入到w的是输出到客户端的

}

func main() {

http.HandleFunc("/", sayhelloName) //设置访问的路由

err := http.ListenAndServe(":9090", nil) //设置监听的端口

if err != nil {

log.Fatal("ListenAndServe: ", err)

}

}

在浏览器中输入了http://localhost:9090

结果： 浏览器页面中输入出 Hello World!

测试在浏览器地址后面加一些参数试试：http://localhost:9090?url\_long=111&url\_long=222

服务器端使用ServeMux分发任务

ServeMux大致作用是，有一张map表，map里的key记录的是r.URL.String()，而value记录的是一个方法，这个方法和ServeHTTP是一样的，这样ServeMux是实现Handler接口的。这个方法有一个别名，叫HandlerFunc。

例3：

package main

import (

"io"

"net/http"

)

func main() {

mux := http.NewServeMux()

mux.HandleFunc("/h", func(w http.ResponseWriter, r \*http.Request) {

io.WriteString(w, "hello")

})

mux.HandleFunc("/bye", func(w http.ResponseWriter, r \*http.Request) {

io.WriteString(w, "byebye")

})

mux.HandleFunc("/baidu", func(w http.ResponseWriter, r \*http.Request) {

http.Redirect(w, r, "http://www.baidu.com", http.StatusTemporaryRedirect)

})

mux.HandleFunc("/hello", sayhello)

http.ListenAndServe(":1234", mux)

}

func sayhello(w http.ResponseWriter, r \*http.Request) {

io.WriteString(w, "hello world")

}

当http.ListenAndServe(":1234", nil)的第2个参数是nil时，HTTP内部会自己建立一个叫DefaultServeMux的ServeMux，因为这个ServeMux是HTTP自己维护的，如果要向这个ServeMux注册的话，就要用http.HandleFunc这个方法。

func FileServer

func FileServer(root FileSystem) Handler

FileServer返回一个使用FileSystem接口root提供文件访问服务的HTTP处理器。要使用操作系统的FileSystem接口实现，可使用http.Dir：

log.Fatal(http.ListenAndServe(":8080", http.FileServer(http.Dir("/usr/share/doc"))))

服务端如何获取客户端请求数据

1. 通过GET参数获取

比较常见的是如下方式获取：

r.ParseForm()

if len(r.Form["id"]) > 0 {

fmt.Fprintln(w, r.Form["id"][0])

}

其中r表示\*http.Request类型，w表示http.ResponseWriter类型。

r.Form是url.Values字典类型，r.Form["id"]取到的是一个数组类型。因为http.request在解析参数的时候会将同名的参数都放进同一个数组里，所以这里要用[0]获取到第一个。

2. 通过POST参数获取

这里要分两种情况：

普通的post表单请求，Content-Type=application/x-www-form-urlencoded

有文件上传的表单，Content-Type=multipart/form-data

第一种情况比较简单，直接用PostFormValue就可以取到了。

fmt.Fprintln(w, r.PostFormValue("id"))

3. 通过COOKIE参数获取

cookie, err := r.Cookie("id")

if err == nil {

fmt.Fprintln(w, "Domain:", cookie.Domain)

fmt.Fprintln(w, "Expires:", cookie.Expires)

fmt.Fprintln(w, "Name:", cookie.Name)

fmt.Fprintln(w, "Value:", cookie.Value)

}

r.Cookie返回\*http.Cookie类型，可以获取到domain、过期时间、值等数据。

application/x-www-form-urlencoded与 text/html 的区别

若需使用r.PostForm、r.Form等方法，必须先调用r.ParseForm()

客户端使用post提交数据到body时，若服务端需使用r.PostForm、r.Form等方法，客户端的请求体必须使用"Content-Type":"application/x-www-form-urlencoded") 类型，并且r.ParseForm()须放在ioutil.ReadAll(r.Body)解析body之前。否则数据将在body中。

HTTP包的运行机制

首先了解服务端的几个概念：

Request：用户请求的信息，用来解析用户的请求信息，包括post，get，Cookie，url等信息。

Response:服务器需要反馈给客户端的信息。

Conn：用户的每次请求链接。

Handle:处理请求和生成返回信息的处理逻辑。

Go实现web服务的流程

第一步：创建Listen Socket，监听指定的端口，等待客户端请求到来。

第二步：Listen Socket接受客户端的请求，得到Client Socket，接下来通过Client Socket与客户端通信。

第三步：处理客户端请求，首先从Client Socket读取HTTP请求的协议头，如果是POST方法，还可能要读取客户端提交的数据，然后交给相应的handler处理请求，handler处理完，将数据通过Client Socket返回给客户端。

15.4 案例

案例1. 使用Get方式实现登录功能

要求：服务端根据客户端的请求判断是否能成功登陆

其中：用户名 username长度为4-10，密码 password长度为6-16

若满足条件返回"登录成功"，否则返回"登录失败"。

服务器端代码：

package main

import (

"fmt"

"net/http"

)

var userDatas = make(map[string]string)

/\*

给客户端响应数据之后，最好return，可以保证不会重复发送数据到客户端

\*/

func login(responseWriter http.ResponseWriter, request \*http.Request) {

//解析提交的form表单数据（若要使用FormValue、form等方法，必须要调用ParseForm方法）

request.ParseForm()

fmt.Println(request.Header)

//判断客户端请求的方式（包括GET、POST、HEAD、PUT、DELETE）

if request.Method == "GET" {

//获取客户端提交的username这个字段的值

username := request.FormValue("username")

password := request.FormValue("password")

fmt.Println(username, password)

//服务端的业务处理

if len(username) < 4 || len(username) > 10 {

responseWriter.Write([]byte("用户名不符合要求"))

return

}

if len(password) < 6 || len(password) > 16 {

responseWriter.Write([]byte("密码不符合要求"))

return

}

//判断客户端发送的username存不存在

//passw是存在服务端系统里面的数据

//password是客户端登录时发送过来的数据

passw := userDatas[username]

if len(passw) <= 0 {

responseWriter.Write([]byte("用户名不存在"))

return

}

if passw != password {

responseWriter.Write([]byte("密码不对"))

return

}

responseWriter.Write([]byte("登录成功"))

return

} else {

responseWriter.Write([]byte("请求方式不对"))

return

}

responseWriter.Write([]byte("登录失败"))

}

func register(responseWriter http.ResponseWriter, request \*http.Request) {

request.ParseForm()

if request.Method == "POST" {

//获取客户端提交的username这个字段的值

username := request.FormValue("username")

password := request.FormValue("password")

fmt.Println(username, password)

//服务端的业务处理

if len(username) < 4 || len(username) > 10 {

responseWriter.Write([]byte("用户名不符合要求"))

return

}

if len(password) < 6 || len(password) > 16 {

responseWriter.Write([]byte("密码不符合要求"))

return

}

//判断系统里面有么有该用户名

if len(userDatas[username]) > 0 {

responseWriter.Write([]byte("用户名已经存在"))

return

}

responseWriter.Write([]byte("注册成功"))

userDatas[username] = password

return

} else {

responseWriter.Write([]byte("请求方式不对"))

return

}

responseWriter.Write([]byte("注册失败"))

}

func other(w http.ResponseWriter, r \*http.Request) {

w.Write([]byte("找不到方法"))

}

func main() {

mux := http.NewServeMux()

mux.HandleFunc("/login", login)

mux.HandleFunc("/register", register)

mux.HandleFunc("/", other)

http.ListenAndServe(":2000", mux)

}

客户端代码：

package main

import (

"bytes"

"fmt"

"io/ioutil"

"net/http"

"net/url"

)

func main() {

//构建参数

param := url.Values{

"username": {"xiaoming"},

"password": {"123456"},

}

//参数转化成body

buffer := bytes.NewBufferString(param.Encode())

//发起post请求

response, err := http.Post("http://192.168.149.138:2000/register",

"application/x-www-form-urlencoded", buffer)

if err != nil {

fmt.Println(err)

}

//获取到服务端的响应数据

if response.StatusCode == 200 {

//操作响应数据

defer response.Body.Close()

//读取body里面的数据

responseBody, err := ioutil.ReadAll(response.Body)

if err != nil {

fmt.Println(err)

}

//打印响应体里面的数据

fmt.Println(string(responseBody))

} else {

fmt.Println("请求失败", response.Status)

}

}

案例2. 实现用户注册和登录功能

要求：

1）实现注册功能

服务端根据客户端的请求判断是否能成功注册，注册成功要求如下：

用户名 username长度为4-10

密码 password长度为6-16

若满足条件返回"注册成功"，否则返回"注册失败"

2） 若请求其它url则返回"找不到方法"

3） 服务端将注册后的数据保存在全局变量中，登录时需要判断usernam是否存在，并且判断密码是否正确。

服务器端代码：

package main

import (

"fmt"

"net/http"

)

var userDatas = make(map[string]string)

/\*

给客户端响应数据之后，最好return，可以保证不会重复发送数据到客户端

\*/

func login(responseWriter http.ResponseWriter, request \*http.Request) {

//解析提交的form表单数据（若要使用FormValue、form等方法，必须要调用ParseForm方法）

request.ParseForm()

fmt.Println(request.Header)

//判断客户端请求的方式

if request.Method == "GET" {

//获取客户端提交的username这个字段的值

username := request.FormValue("username")

password := request.FormValue("password")

fmt.Println(username, password)

//服务端的业务处理

if len(username) < 4 || len(username) > 10 {

responseWriter.Write([]byte("用户名不符合要求"))

return

}

if len(password) < 6 || len(password) > 16 {

responseWriter.Write([]byte("密码不符合要求"))

return

}

//判断客户端发送的username存不存在

//passw是存在服务端系统里面的数据

//password是客户端登录时发送过来的数据

passw := userDatas[username]

if len(passw) <= 0 {

responseWriter.Write([]byte("用户名不存在"))

return

}

if passw != password {

responseWriter.Write([]byte("密码不对"))

return

}

responseWriter.Write([]byte("登录成功"))

return

} else {

responseWriter.Write([]byte("请求方式不对"))

return

}

responseWriter.Write([]byte("登录失败"))

}

func register(responseWriter http.ResponseWriter, request \*http.Request) {

request.ParseForm()

if request.Method == "POST" {

//获取客户端提交的username这个字段的值

username := request.FormValue("username")

password := request.FormValue("password")

fmt.Println(username, password)

//服务端的业务处理

if len(username) < 4 || len(username) > 10 {

responseWriter.Write([]byte("用户名不符合要求"))

return

}

if len(password) < 6 || len(password) > 16 {

responseWriter.Write([]byte("密码不符合要求"))

return

}

//判断系统里面有么有该用户名

if len(userDatas[username]) > 0 {

responseWriter.Write([]byte("用户名已经存在"))

return

}

responseWriter.Write([]byte("注册成功"))

userDatas[username] = password

return

} else {

responseWriter.Write([]byte("请求方式不对"))

return

}

responseWriter.Write([]byte("注册失败"))

}

func other(w http.ResponseWriter, r \*http.Request) {

w.Write([]byte("找不到方法"))

}

func main() {

mux := http.NewServeMux()

mux.HandleFunc("/login", login)

mux.HandleFunc("/register", register)

mux.HandleFunc("/", other)

http.ListenAndServe(":1234", mux)

}

客户端代码：

package main

import (

"bytes"

"fmt"

"io/ioutil"

"net/http"

"net/url"

)

func main() {

//构建参数

param := url.Values{

"username": {"admin"},

"password": {"123456"},

}

//参数转化成body

buffer := bytes.NewBufferString(param.Encode())

//发起post请求

response, err := http.Post("http://localhost:1234/register",

"application/x-www-form-urlencoded", buffer)

if err != nil {

fmt.Println(err)

}

//获取到服务端的响应数据

if response.StatusCode == 200 {

//操作响应数据

defer response.Body.Close()

//读取body里面的数据

responseBody, err := ioutil.ReadAll(response.Body)

if err != nil {

fmt.Println(err)

}

//打印响应体里面的数据

fmt.Println(string(responseBody))

} else {

fmt.Println("请求失败", response.Status)

}

}

15.5 小结

本章学习Go语言HTTP编程的相关内容，很多开源框架已经将此部分的操作都封装好了。在软件开发过程中，通过使用开源框架，此部分的内容需要自己动手的部分已经极少了。但是熟练的使用基础的网络库功能，对于未来阅读开源代码会有很大的帮助，所以此部分建议大家多花些时间，多动手练习。

第十六章 JSON处理

本章重点为大家介绍如下的内容：

* 什么是JSON
* JSON语法规则
* JSON的优点和缺点
* 对象转换为JSON
* JSON转换为对象
* JSON转换为map
* JSON流解析

16.1 JSON简介

JSON是一种轻量级的数据交换格式(JavaScript Object Notation)，因其简单、可读性强广泛使用。

JSON语法规则

对象是一个无序的“‘名称/值’对”集合。一个对象以“{”（左括号）开始，“}”（右括号）结束。每个“名称”后跟一个“:”（冒号）；“‘名称/值’ 对”之间使用“,”（逗号）分隔。

数组是值（value）的有序集合。一个数组以“[”（左中括号）开始，“]”（右中括号）结束。值之间使用“,”（逗号）分隔。

值（value）可以是双引号括起来的字符串（string）、数值(number)、true、false、 null、对象（object）或者数组（array）。这些结构可以嵌套。

字符串（string）是由双引号包围的任意数量Unicode字符的集合，使用反斜线转义。一个字符（character）即一个单独的字符串（character string）。

简单的json：

{

name: "张三",

age: 23

}

复杂的json：

{

"employee": {

"firstName": "张",

"lastName": "三",

"employeeNumber": 101,

"age": 23

}

}

JSON的优点:

数据格式比较简单, 易于读写, 格式都是压缩的, 占用带宽小

因为JSON格式能够直接为服务器端代码使用, 大大简化了服务器端和客户端的代码开发量, 但是完成的任务不变, 且易于维护

16.2 对象转换为JSON

Go的标准包encoding/json对JSON的编解码提供了完整的支持。

将Go数据类型转换为JSON。用到的函数：

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

该函数递归遍历v的结构，生成对应的JSON。

字典转JSON

package main

import (

"encoding/json"

"fmt"

)

func main() {

m := map[string][]string{

"level": {"debug"},

"message": {"File not found", "Stack overflow"},

}

if data, err := json.Marshal(m); err == nil {

fmt.Printf("%s\n", data)

}

}

输出结果：

{"level":["debug"],"message":["File not found","Stack overflow"]}

大家可以看到Marshal函数返回的JSON字符串是没有空白字符和缩进的，这种紧凑的表示形式是最常用的传输形式，但是不好阅读。如果需要为前端生成便于阅读的格式，可以调json.MarshaIndent，该函数有两个参数表示每一行的前缀和缩进方式：

if data, err := json.MarshalIndent(m, "", " "); err == nil {

fmt.Printf("%s\n", data)

}

输出结果：

{

"level": [

"debug"

],

"message": [

"File not found",

"Stack overflow"

]

}

Go数据类型转换为JSON规则

在编码过程中，json包会将Go的类型转换为JSON类型，转换规则如下：

bool -> JSON boolean

浮点数, 整数, Number -> JSON number

string -> JSON string

数组、切片 -> JSON数组

[]byte -> base64 string

struct、map -> JSON object

结构体转JSON

经常会使用结构体来转换成JSON。json包是通过反射机制来实现编解码的，因此结构体必须导出所转换的字段，不导出的字段不会被json包解析：

package main

import (

"encoding/json"

"fmt"

)

type DebugInfo struct {

Level string

Msg string

author string // 未导出字段不会被json解析(首字母小写)

}

func main() {

dbgInfs := []DebugInfo{

DebugInfo{"debug", `File: "test.txt" Not Found`, "Cynhard"},

DebugInfo{"", "Logic error", "Gopher"},

}

if data, err := json.Marshal(dbgInfs); err == nil {

fmt.Printf("%s\n", data)

}

}输出结果如下：

[{"Level":"debug","Msg":"File: \"test.txt\" Not Found"},{"Level":"","Msg":"Logic error"}]

结构体字段标签：

json包在解析结构体时，如果遇到key为“json”的字段标签，则会按照一定规则解析该标签：第一个出现的是字段在JSON串中使用的名字，之后为其他选项，例如“omitempty”指定空值字段不出现在JSON中。如果整个value为'"-"'，则不解析该字段。例如将上例中的结构体改为如下：

type DebugInfo struct {

Level string `json:"level,omitempty"` // Level解析为level,忽略空值

Msg string `json:"message" ` // Msg解析为message

Author string `json:"-"` // 忽略Author

}

则输出为：

[{"level":"debug","message":"File: \"test.txt\" Not Found"},{"message":"Logic error"}]

结构体中匿名字段：

json包在解析匿名字段时，会将匿名字段的字段当成该结构体的字段处理：

package main

import (

"encoding/json"

"fmt"

)

type Point struct {

X, Y int

}

type Circle struct {

Point

Radius int

}

func main() {

if data, err := json.Marshal(Circle{Point{50, 50}, 25}); err == nil {

fmt.Printf("%s\n", data)

}

}

输出结果：

{"X":50,"Y":50,"Radius":25}

注意：

Marshal函数只有在转换成功的时候才会返回数据，在转换的过程中我们需要注意几点：

JSON对象只支持string作为key，所以要编码一个map，那么必须是map[string]T这种类型(T是Go语言中任意的类型)

Channel, complex和function是不能被编码成JSON的

指针在编码的时候会输出指针指向的内容，而空指针会输出null

16.3 JSON转换为对象

将JSON转换为Go数据类型。用到的函数：

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

JSON转数组

package main

import (

"encoding/json"

"fmt"

)

func main() {

data := `[{"Level":"debug","Msg":"File: \"test.txt\" Not Found"},{"Level":"","Msg":"Logic error"}]`

var dbgInfos []map[string]string

json.Unmarshal([]byte(data), &dbgInfos)

fmt.Println(dbgInfos)

}

输出为：

[map[Level:debug Msg:File: "test.txt" Not Found] map[Level: Msg:Logic error]]

在解码过程中，json包会将JSON类型转换为Go类型，转换规则如下：

JSON boolean -> bool

JSON number -> float64

JSON string -> string

JSON数组 -> []interface{}

JSON object -> map

null -> nil

JSON转结构体

JSON可以转换成结构体。同编码一样，json包是通过反射机制来实现解码的，因此结构体必须导出所转换的字段，不导出的字段不会被json包解析，另外解析时不区分大小写：

package main

import (

"encoding/json"

"fmt"

)

type DebugInfo struct {

Level string

Msg string

author string // 未导出字段不会被json解析

}

func (dbgInfo DebugInfo) String() string {

return fmt.Sprintf("{Level: %s, Msg: %s}", dbgInfo.Level, dbgInfo.Msg)

}

func main() {

data := `[{"level":"debug","msg":"File Not Found","author":"Cynhard"},{"level":"","msg":"Logic error","author":"Gopher"}]`

var dbgInfos []DebugInfo

json.Unmarshal([]byte(data), &dbgInfos)

fmt.Println(dbgInfos)

}

输出结果如下：

[{Level: debug, Msg: File Not Found} {Level: , Msg: Logic error}]

结构体字段标签

解码时依然支持结构体字段标签，规则和编码时一样：

package main

import (

"encoding/json"

"fmt"

)

type DebugInfo struct {

Level string `json:"level"` // level 解码为 Level

Msg string `json:"message"` // message 解码为 Msg

Author string `json:"-"` // 忽略Author

}

func (dbgInfo DebugInfo) String() string {

return fmt.Sprintf("{Level: %s, Msg: %s}", dbgInfo.Level, dbgInfo.Msg)

}

func main() {

data := `[{"level":"debug","message":"File Not Found","author":"Cynhard"},{"level":"","message":"Logic error","author":"Gopher"}]`

var dbgInfos []DebugInfo

json.Unmarshal([]byte(data), &dbgInfos)

fmt.Println(dbgInfos)

}

则结果为：

[{Level: debug, Msg: File Not Found} {Level: , Msg: Logic error}]

匿名字段

编码时，和解码类似，在解码JSON时，如果找不到字段，则查找字段的字段：

package main

import (

"encoding/json"

"fmt"

)

type Point struct{ X, Y int }

type Circle struct {

Point

Radius int

}

func main() {

data := `{"X":80,"Y":80,"Radius":40}`

var c Circle

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

fmt.Println(c)

}

输出结果：

{{80 80} 40}

16.4 JSON转换为map

package main

mport (

"encoding/json"

"fmt"

)

func main() {

map2json2map()

}

func map2json2map() {

map1 := make(map[string]interface{})

map1["1"] = "hello"

map1["2"] = "world"

//return []byte

str, err := json.Marshal(map1)

if err != nil {

fmt.Println(err)

}

fmt.Println("map to json", string(str))

//json([]byte) to map

map2 := make(map[string]interface{})

err = json.Unmarshal(str, &map2)

if err != nil {

fmt.Println(err)

}

fmt.Println("json to map ", map2)

fmt.Println("The value of key1 is", map2["1"])

}

运行结果：

map to json {"1":"hello","2":"world"}

json to map map[1:hello 2:world]

The value of key1 is hello

16.5 JSON流解析

json 包提供了Decoder和Encoder用来支持JSON 数据流的读写。函数NewDecoder 和 NewEncoder封装了io.Reader和io.Writer 接口类型。

func NewDecoder(r io.Reader) \*Decoder

func NewEncoder(w io.Writer) \*Encoder

由于读写操作的普遍性，类型Encoder和Decoder可以用于多种场合，例如读写HTTP 链接，WebSockets或者文件。

func main() {

data := `{"name":"go语言学习", "Count":34, "student":["张三", "里斯"], "address":{"floor":2, "classroom":"201"}}`

//流式编码

dec := json.NewDecoder(bytes.NewBufferString(data))

var v map[string]interface{}

if err := dec.Decode(&v); err != nil {

log.Println(err)

return

}

for k := range v {

if k == "name" {

delete(v, k)

}

}

fmt.Printf("%T %v\n", v, v)

//流式解码

buffer := bytes.Buffer{}

enc := json.NewEncoder(&buffer)

if err := enc.Encode(&v); err != nil {

log.Println(err)

}

fmt.Printf("%v", buffer.String())

}

map[string]interface {} map[Count:34 address:map[classroom:201 floor:2] student:[张三 里斯]]

{"Count":34,"address":{"classroom":"201","floor":2},"student":["张三","里斯"]}

16.6 小结

本章学习Go语言JSON操作相关的内容，在目前的软件开发中JSON的作用与地位已无可替代，此章的内容用得多了就熟悉了，不必刻意去深入学习，因为随着用得越来越熟练，不深入都是不可能的。