

零壹快学程序设计系列丛书

书网合一

小白实战手册

零壹
快学

零基础 Go 语言 从入门到精通

立体化教学模式 零基础快速入门

零壹快学 编著

真人教学，手把手教你学 Go 语言

- 丰富示例，贴近场景——丰富的代码示例，编程场景化
- 视频教学，动手操作——扫码即可学习配套视频，通俗易懂
- 线上问答，及时解惑——在线论坛，解答学习中遇到的疑问
- 海量题库，随时学习——大量题目练习，加快理解



扫一扫
获取视频激活码

SPM 南方出版传媒·广东人民出版社

◇—— 零壹快学简介 ——◇

零基础，一起学！

零壹快学是一个集教育、培训、交流于一体的计算机学习融媒体平台，覆盖Web、APP和微信小程序等场景，为广大IT学习者提供计算机各学科的教学资源、培训课程、测评系统等。通过人工智能手段，模拟真实课堂环境，力求打造成为一个全面、权威、高效的一流计算机在线教育平台。目前，平台已开发了包括计算机编程、云计算、区块链等在内的多门课程，提供计算机等级考试在线题库系统，将陆续开发计算机技术与软件专业技术资格（水平）考试、CCIE等在线题库系统，同时拥有一支由一线互联网公司工程师、知名高校教师组成的专家团队，精心为广大IT学习者提供优质的学习资源、课程和服务。

零基础 GO 从入门到精通

零壹快学 GO 团队出品

广东人民出版社

图书在版编目 (CIP) 数据

零基础Go语言从入门到精通 / 零壹快学编著. —广州：广东人民出版社，2020.4
ISBN 978-7-218-13975-3

I . ①零… II . ①零… III . ①程序语言—程序设计 IV . ①TP312

中国版本图书馆CIP数据核字 (2019) 第247966号

Ling Jichu Go Yuyan Cong Rumen Dao Jingtong
零 基 础 Go 语 言 从 入 门 到 精 通
零壹快学 编著

 版权所有 翻印必究

出版人：肖风华

统筹策划：李婧玮
责任编辑：陈泽洪
封面设计：画画鸭工作室
内文设计：奔流文化
责任技编：吴彦斌

出版发行：广东人民出版社
地 址：广州市海珠区新港西路204号2号楼（邮政编码：510300）
电 话：(020) 85716809（总编室）
传 真：(020) 85716872
网 址：<http://www.gdpph.com>
印 刷：广东信源彩色印务有限公司
开 本：787毫米×1092毫米 1/16
印 张：26 字 数：500千
版 次：2020年4月第1版
印 次：2020年4月第1次印刷
定 价：79.00元

如发现印装质量问题，影响阅读，请与出版社（020-32449105）联系调换。
售书热线：020-32449123

前 言

历经七十多年的发展，无论是对于国内数以十万计的学习者而言，还是在有着多年培训经验的编者们看来，学习编程语言，仍存在不小的难度，甚至有不少学习者因编程语言的复杂多变、难度太大而选择了中途放弃。实际上，只要掌握了其变化规律，即使再晦涩难懂的计算机专业词汇也无法阻挡学习者们的脚步。对于初学者来说，若有一本能看得懂，甚至可以用于自学的编程入门书是十分难得的。为初学者提供这样一本书，正是我们编写本套丛书的初衷。

零壹快学以“零基础，一起学”为主旨，针对零基础编程学习者的需求和学习特点，由专业团队量身打造了本套计算机编程入门教程。本套丛书的作者都从事编程教育和培训工作多年，拥有丰富的一线教学经验，对于学习者常遇到的问题十分熟悉，在编写过程中针对这些问题花费了大量的时间和精力来加以阐释，对书中的每个示例反复推敲，加以取舍，按照学习者的接受程度雕琢示例涉及的技术点，力求成就一套真正适合初学者的编程书籍。

本套丛书涵盖了Java、PHP、Python、JavaScript、HTML、CSS、Linux、iOS、Go语言、C++、C#等计算机语言，同时借助大数据和云计算等技术，为广大编程学习者提供计算机各学科的视频课程、在线题库、测评系统、互动社区等学习资源。

◆ 课程全面，聚焦实战

本套丛书涵盖多门计算机语言，内容全面、示例丰富、图文并茂，通过通俗易懂的语言讲解相关计算机语言的特性，以点带面，突出开发技能的培养，既方便学习者了解基础知识点，也能帮助他们快速掌握开发技能，为编程开发设计积累实战经验。

◆ 专业团队，紧贴前沿

本套丛书作者由一线互联网公司高级工程师、知名高校教师和研究所技术人员等组成，线上线下同步进行专业讲解及点评分析，为学习者扫除学习障碍。与此同时，团队

在内容研发方向上紧跟当前技术领域热点，及时更新，直击痛点和难点。

◆ 全网覆盖，应用面广

本套丛书已全网覆盖Web、APP和微信小程序等客户端，为广大学习者提供包括计算机编程、人工智能、大数据、云计算、区块链、计算机等级考试等在内的多门视频课程，配有相关测评系统和技术交流社区，互动即时性强，可实现在线教育随时随地轻松学。

随着大数据和云计算时代的到来，近些年来Go语言逐渐成为了程序开发人员最想了解和学习的编程语言之一。Go语言具有原生支持并发、代码风格清晰和编译速度快等特性，目前主要应用于微服务、高并发处理和网络编程。

本书基于Go 1.12编写而成，先从Go语言的基础知识入门，再到对Go开发的详细讲解，最终带领读者进入Go语言应用开发领域。对于零基础的读者而言，本书可以作为Go开发的快速入门教材。

• 本书内容

- ◆ 简介和概况：第1~2章，主要介绍了Go语言的概况和Go开发工具GoLand的使用方式。
- ◆ 基础知识：第3~10章，主要介绍Go语言的基础知识，包括基本变量与类型、常量与运算符、流程控制、内置容器、函数、包和结构体等。
- ◆ 进阶知识：第11~15章，主要介绍Go语言的进阶知识，包括文件操作、接口、并发与通道、反射等。
- ◆ 应用实战：第16~20章，主要介绍如何通过Go语言来实现网络编程和数据库操作，并深入理解之前章节所学的知识点。

• 本书特点

- ◆ 由浅入深，循序渐进。本书先介绍Go语言基础，再介绍网络编程和数据库操作的实战应用，讲解过程详尽，通俗易懂。
- ◆ 示例丰富，贴近场景。本书提供了丰富的代码示例，每个知识点均有对应示例代码进行演示，便于读者清晰理解。这些示例大部分来自于工作场景，有利于读者理解其中的使用逻辑，快速掌握。
- ◆ 视频教学，动手操作。本书每一章都配有教学视频，直观展示了代码的运行效

果，并配有通俗易懂的解释。

◆ **知识拓展，难度提升。**本书的大部分章节结尾设有“知识拓展”，在讲解基础知识的同时提供了一些有一定难度的知识点，方便有能力的读者深入思考，强化学习，加深对Go开发的理解。

◆ **线上问答，及时解惑。**本书为确保广大读者的学习能够顺利进行，提供了在线答疑服务，希望通过这种方式及时解决读者在学习Go开发的过程中所遇到的困难和疑惑。

• 本书配套资源（可扫下方二维码获取）

- ◆ **大量的代码示例。**通过运行这些代码，读者可以进一步巩固所学的知识。
- ◆ **零壹快学官方视频教程。**力求让读者学以致用，知行并进，加强实战能力。
- ◆ **在线答疑。**为读者解惑，帮助读者解决学习中的困难，快速掌握要点难点。

• 本书适用对象

- ◆ 编程的初学者、爱好者与自学者
- ◆ 高等院校和培训学校的师生
- ◆ 职场新人
- ◆ 准备进入互联网行业的再就业人群
- ◆ “菜鸟”程序员
- ◆ 初、中级程序开发人员



零壹快学微信公众号

《零基础Go语言从入门到精通》由零壹快学汪心桐、廖旭编写。本书从初学者角度出发，详细讲述了Go语言应用开发所需的基础知识和开发实战中的必备技能。全书内容通俗易懂，示例丰富，步骤清晰，图文并茂，可以使读者轻松掌握Go语言应用开发的精髓，活学活用，是Go开发实战中必备的参考书。

编 者

2020年3月

目 录

CONTENTS

第1章 走进 Go	1	第3章 基本变量与类型	22
1.1 Go 编程语言概述	1	3.1 Go 语言变量	22
1.1.1 Go 的历史	1	3.1.1 声明变量	22
1.1.2 Go 语言特性及应用场景	2	3.1.2 初始化变量	23
1.1.3 使用 Go 语言的项目	3	3.1.3 变量值交换	25
1.2 Go 语言官方文档	4	3.1.4 匿名变量	26
1.3 学好 Go 的建议	5	3.2 Go 语言数据类型	27
1.4 常用软件及网站	6	3.2.1 整型	27
1.5 小结	7	3.2.2 浮点型	28
第2章 安装和运行 Go	8	3.2.3 字符串	29
2.1 GOROOT 和 GOPATH	8	3.2.4 字符和转义字符	30
2.2 在 Windows 下安装 Go	9	3.2.5 布尔型	31
2.3 在 Linux 下安装 Go	11	3.2.6 数据类型判断	31
2.4 在 Mac OS 下安装 Go	12	3.2.7 数据类型转换	32
2.5 安装开发工具	13	3.3 指针	34
2.5.1 GoLand 下载	13	3.3.1 声明指针	34
2.5.2 GoLand 安装	13	3.3.2 取变量地址	34
2.5.3 GoLand 使用	16	3.3.3 获取指针所指向的内容	35
2.6 编写第一个 Go 程序	19	3.3.4 使用指针修改值	36
2.6.1 GoLand 编写	19	3.4 其他数据类型	37
2.6.2 文本编辑器编写	20	3.5 小结	38
2.7 小结	21	3.6 知识拓展	38

3.6.1 字符串的应用	38
3.6.2 Go 语言注释	43
第 4 章 常量与运算符	44
4.1 常量	44
4.1.1 常量的定义	44
4.1.2 常量组的定义	45
4.1.3 常量枚举	46
4.2 运算符	47
4.2.1 算术运算符	47
4.2.2 比较运算符	49
4.2.3 赋值运算符	50
4.2.4 位运算符	51
4.2.5 逻辑运算符	52
4.2.6 其他运算符	53
4.3 小结	54
4.4 知识拓展	54
第 5 章 流程控制.....	56
5.1 if 判断	56
5.2 循环控制	58
5.2.1 for 循环.....	58
5.2.2 break 跳出循环	59
5.2.3 continue 继续循环	61
5.3 switch 分支	61
5.4 goto 跳转	63
5.5 小结	64
5.6 知识拓展	64
5.6.1 斐波那契数列	64
5.6.2 循环嵌套	67

第 6 章 Go 语言内置容器	69
6.1 数组	69
6.1.1 声明数组	70
6.1.2 初始化数组	70
6.1.3 range 关键字	71
6.1.4 遍历数组	72
6.2 切片	72
6.2.1 从数组生成一个新的切片	73
6.2.2 从切片生成一个新的切片	74
6.2.3 直接生成一个新的切片	75
6.2.4 为切片添加元素	77
6.2.5 从切片删除元素	79
6.2.6 遍历切片	80
6.3 映射	81
6.3.1 声明映射	81
6.3.2 初始化映射	82
6.3.3 遍历映射	83
6.3.4 从映射中删除键值对	84
6.4 小结	85
6.5 知识拓展	86
第 7 章 函数.....	90
7.1 声明函数	90
7.1.1 函数参数的简写	91
7.1.2 函数返回值的简写	91
7.1.3 带有变量名的返回值	92
7.2 使用函数	92
7.3 函数变量	93
7.4 可变参数	95
7.4.1 可变参数函数的使用	95
7.4.2 可变参数与内置函数	96

7.4.3 可变参数的传递	96	8.10 知识拓展	124
7.5 匿名函数和闭包	98	第 9 章 结构体	126
7.5.1 定义和使用匿名函数	98	9.1 理解结构体	126
7.5.2 闭包的定义	99	9.2 定义结构体	127
7.5.3 闭包的“记忆力”	99	9.3 实例化结构体	128
7.6 延迟执行语句	101	9.3.1 标准实例化	128
7.7 小结	103	9.3.2 new 函数实例化	129
7.8 知识拓展	103	9.3.3 取地址实例化	129
7.8.1 函数参数传递的本质	103	9.3.4 访问成员	130
7.8.2 Go 内置函数	106	9.4 初始化结构体	131
第 8 章 包管理	109	9.4.1 键值对格式初始化	131
8.1 工作区	109	9.4.2 列表格式初始化	132
8.1.1 工作区结构	109	9.5 结构体方法	133
8.1.2 GOPATH	110	9.5.1 指针类型接收者	134
8.1.3 GOROOT	112	9.5.2 值类型接收者	135
8.1.4 GOBIN	112	9.6 结构体内嵌	136
8.2 包的声明	112	9.6.1 初始化结构体内嵌	138
8.3 包的导入	113	9.6.2 内嵌匿名结构体	139
8.3.1 导入声明	113	9.7 匿名结构体	140
8.3.2 远程导入	114	9.7.1 匿名结构体定义与初始化	140
8.3.3 别名导入	115	9.7.2 匿名结构体的应用	142
8.3.4 匿名导入	116	9.8 小结	143
8.4 main 包	117	9.9 知识拓展	143
8.5 包的初始化	118	第 10 章 错误处理	147
8.6 包的命名	119	10.1 错误处理的方式	147
8.7 依赖包管理	120	10.2 自定义错误	148
8.8 Go 语言命名规范	121	10.2.1 错误类型	148
8.8.1 驼峰式命名法	121	10.2.2 创建错误	149
8.8.2 导出标识符	122	10.2.3 自定义错误格式	149
8.9 小结	123		

10.3 Go 语言宕机	150	12.4 空接口	199
10.4 宕机恢复	151	12.4.1 将值保存到空接口	199
10.4.1 recover 捕获宕机	151	12.4.2 从空接口取值	200
10.4.2 recover 应用	153	12.4.3 空接口的常见使用	201
10.5 Go 语言错误的应用	154	12.5 类型断言	203
10.6 小结	155	12.5.1 ok-pattern	206
10.7 知识拓展	156	12.5.2 switch-type	207
第 11 章 文件操作	158	12.6 小结	209
11.1 目录基本操作	158	12.7 知识拓展	209
11.1.1 列目录	159		
11.1.2 创建目录	163		
11.1.3 删除目录	166		
11.2 文件基本操作	168		
11.2.1 文件创建与打开	169		
11.2.2 文件读取	171		
11.2.3 文件写入	174		
11.2.4 删除文件	177		
11.3 处理 JSON 文件	178		
11.3.1 编码 JSON	179		
11.3.2 解码 JSON	182		
11.4 小结	186		
第 12 章 接口与类型	187		
12.1 接口定义	187		
12.1.1 接口的定义	187		
12.1.2 鸭子类型	188		
12.2 接口的创建与实现	189		
12.2.1 接口创建	189		
12.2.2 接口赋值	193		
12.3 接口嵌入	196		
12.4 空接口	199		
12.4.1 将值保存到空接口	199		
12.4.2 从空接口取值	200		
12.4.3 空接口的常见使用	201		
12.5 类型断言	203		
12.5.1 ok-pattern	206		
12.5.2 switch-type	207		
12.6 小结	209		
12.7 知识拓展	209		
第 13 章 并发与通道	213		
13.1 概述	213		
13.1.1 并行与并发	214		
13.1.2 Go 并发优势	214		
13.2 goroutine	215		
13.2.1 goroutine 定义	215		
13.2.2 创建 goroutine	216		
13.2.3 main 函数	218		
13.2.4 使用匿名函数创建 goroutine	220		
13.2.5 runtime 包	220		
13.3 channel	225		
13.3.1 channel 类型	225		
13.3.2 缓冲机制	227		
13.3.3 close 和 range	229		
13.3.4 单向 channel	232		
13.3.5 定时器	233		
13.4 select	234		
13.4.1 select 作用	234		
13.4.2 超时	236		
13.4.3 死锁	237		
13.5 小结	238		

第 14 章 反射	239	
14.1 反射定义	239	
14.1.1 反射的定义	239	
14.1.2 与其他语言的区别	239	
14.2 基本用法	241	
14.2.1 获取类型信息	243	
14.2.2 获取类型的值	245	
14.2.3 使用反射调用函数	246	
14.3 对结构体的反射操作	248	
14.3.1 获取结构体成员类型	248	
14.3.2 获取结构体成员字段的值	250	
14.3.3 反射执行结构体方法	251	
14.4 反射三定律	252	
14.4.1 接口到反射类型的转换	252	
14.4.2 反射到接口类型的转换	253	
14.4.3 修改反射类型对象	254	
14.5 反射的性能	255	
14.6 小结	257	
第 15 章 Go 命令行工具	258	
15.1 编译相关指令	259	
15.1.1 build	259	
15.1.2 run	262	
15.1.3 install	264	
15.1.4 交叉编译	265	
15.2 代码获取 (get)	267	
15.3 格式化代码 (fmt)	268	
15.4 注释文档 (doc)	269	
15.5 代码测试 (test)	271	
15.5.1 单元测试	271	
15.5.2 基准测试	274	
15.5.3 覆盖率测试	277	
15.6 性能分析	278	
15.6.1 安装 Graphviz	278	
15.6.2 通过文件方式	279	
15.6.3 通过 HTTP 方式	281	
15.7 小结	284	
第 16 章 正则表达式	285	
16.1 正则表达式介绍	285	
16.2 正则表达式语法	286	
16.2.1 普通字符	286	
16.2.2 字符转义	288	
16.2.3 元字符	288	
16.2.4 限定符	290	
16.2.5 定位符	290	
16.2.6 分组构造	291	
16.2.7 匹配模式	291	
16.3 regexp 包	292	
16.3.1 MatchString 函数	292	
16.3.2 FindStringIndex 函数	294	
16.3.3 ReplaceAllString 函数	295	
16.4 小结	296	
16.5 知识拓展	296	
第 17 章 HTTP 编程 (上)	298	
17.1 HTTP 简介	298	
17.1.1 HTTP 协议	298	
17.1.2 URL 地址	300	
17.1.3 Web 工作方式	300	
17.2 HTTP 客户端	301	
17.2.1 REST	301	

17.2.2 Client 和 Request	302	19.1.3 IPv4 和 IPv6	353
17.2.3 发起 GET 请求	307	19.1.4 子网掩码	354
17.2.4 发起 POST 请求	312	19.2 Socket 基础	355
17.3 爬虫框架 gocolly	315	19.2.1 Socket 简介	355
17.3.1 gocolly 简介	316	19.2.2 netcat 工具简介	356
17.3.2 实战爬取西刺代理 IP	320	19.3 TCP 编程	357
17.4 小结	324	19.3.1 TCP 简介	357
17.5 知识拓展	324	19.3.2 TCP 客户端	359
第 18 章 HTTP 编程（下）	327	19.3.3 TCP 服务端	364
18.1 错误处理的方式	327	19.4 UDP 编程	369
18.1.1 最常见的 Web 服务端	328	19.4.1 UDP 简介	369
18.1.2 请求方法	328	19.4.2 UDP 客户端	369
18.1.3 响应状态码	329	19.4.3 UDP 服务端	372
18.2 HTTP 服务端	329	19.5 小结	376
18.2.1 Server 和 Response	330	19.6 知识拓展	377
18.2.2 启动 HTTP 服务器	332		
18.2.3 启动 HTTPS 服务器	333		
18.2.4 静态文件	336		
18.3 路由	338		
18.4 中间件	341		
18.5 Web 框架 gin	343		
18.5.1 启动 HTTP 服务器	344		
18.5.2 静态文件	344		
18.5.3 路由	345		
18.6 小结	347		
18.7 知识拓展	347		
第 19 章 Socket 编程	350		
19.1 计算机网络简介	350		
19.1.1 OSI 七层网络模型	350		
19.1.2 TCP/IP 协议	352		
19.2 Socket 基础	355		
19.2.1 Socket 简介	355		
19.2.2 netcat 工具简介	356		
19.3 TCP 编程	357		
19.3.1 TCP 简介	357		
19.3.2 TCP 客户端	359		
19.3.3 TCP 服务端	364		
19.4 UDP 编程	369		
19.4.1 UDP 简介	369		
19.4.2 UDP 客户端	369		
19.4.3 UDP 服务端	372		
19.5 小结	376		
19.6 知识拓展	377		
第 20 章 数据库编程	381		
20.1 MySQL 简介	381		
20.1.1 安装 MySQL	382		
20.1.2 MySQL 常见命令	382		
20.2 database/sql	383		
20.3 数据库基本操作	384		
20.3.1 连接数据库	384		
20.3.2 创建数据表	386		
20.3.3 插入数据	389		
20.3.4 查询数据	392		
20.3.5 更改数据	395		
20.3.6 删除数据	397		
20.3.7 MySQL 事务	398		
20.4 小结	401		
20.5 知识拓展	401		

>> 第 1 章

走进 Go

<<

1.1

Go编程语言概述

Go语言也叫Golang，是由谷歌（Google）公司在2007年推出的一款静态编译型语言。Go语言高效、简洁、容易上手，加上它的高并发和自带垃圾回收等特性，一出现就备受大家的喜爱。目前，越来越多的公司开始推广并使用Go语言，主要将其用于服务端开发、并发编程和网络编程等。

本章将带你走进Go语言，体会Go语言的独特魅力。

1.1.1 Go的历史

Go语言起源于2007年，于2009年正式对外发布。Go语言项目主要由三位IT届著名人士参与开发：肯·汤普逊（Ken Thompson）、罗布·派克（Rob Pike）和罗伯特·格里泽默（Robert Griesemer）。此处简单介绍一下他们的贡献。

肯·汤普逊：图灵奖得主，Unix发明人，B语言（C语言前身）作者。1983年，他与丹尼斯·里奇（Dennis Ritchie）共同获得了图灵奖。2006年，他进入谷歌公司工作，与其他人共同设计了Go语言。

罗布·派克： Unix团队和Plan 9操作系统计划的成员，曾经参与过贝尔实验室九号计划、操作系统Inferno与编程语言Limbo的开发。

罗伯特·格里泽默：曾协助Java的HotSpot编译器和Chrome浏览器的JavaScript引擎V8的制作。

Go语言主要大事件如下：

2007年，Go语言雏形设计。

2009年，首次公开发布。

2010年，当选TIOBE2009年度最佳编程语言，谷歌正式将其投入使用。

2011年，Google App Engine支持Go语言。

2017年，当选TIOBE2016年度最佳编程语言。

截至目前，Go语言的最新版本为Go 1.13。

另外，Go语言还有个十分可爱的吉祥物——金花鼠（图1.1.1），由罗布·派克的妻子芮妮·弗伦奇（Renée French）绘制。



图1.1.1 Go语言吉祥物

1.1.2 Go语言特性及应用场景

随着人工智能、大数据和云计算时代的到来，Python、Java和PHP等编程语言风靡盛行，C/C++等编程语言也有自己的特性和应用场景，那么作为后起之秀的Go语言究竟有哪些特性和优势？这些特性又适用于哪些应用场景呢？

1. 容易上手

Go语言上手非常容易，许多零基础的初学者在学习大约一周的时间后就可以使用Go语言完成一些既定的任务。而C/C++等语言，则需要经过几个月的时间学习才开始上手，并且学习中的大量时间都用于避免语法错误和调试代码bug。

2. 编译速度快

在Go编程语言的1.7版本更新中，通过减少编译器所使用的内存量、减少链接时间来减少总的编译时间。根据测试，这项改进和Go1.5.3相比减少了20%~30%的总编译时间。

另外，Go语言工程结构简单，没有头文件和不允许包的交叉编译等规则，这也在很大程度上减少了编译所需的时间。

3. 原生支持并发

Go语言最主要的特性就是从语言层面原生支持并发，无须任何第三方库。Go的并发基于goroutine，可以理解为一种微线程。与Python的多线程性能限制于GIL全局锁不同，Go语言的并发可以充分利用CPU的资源，将goroutine合理地分配到每个CPU中，最大限度地使用CPU的性能。goroutine之间的通信可以使用Go语言的sync包和channel机制来实现。

4. 垃圾回收

一直以来，内存管理是程序员开发中的一大难题。传统的编程语言C/C++中，程序员必须对内存进行细致的管理操作，控制内存的申请及释放，否则就可能产生内存泄漏问题。

为了解决这个问题，Go语言实现了自己的垃圾回收器，在1.3版本前采用传统的Mark and Sweep算法。Go语言在1.5的版本更新中进行了较大改进，使用了三色标记算法，进一步改善了垃圾回收的机制，使程序员更加专注于业务本身，不用把过多的精力投入到垃圾回收中。

5. 代码风格清晰

Go语言语法在C语言的基础上进行优化，去掉了表达式和for循环中冗余的括号，强制统一了代码风格，如Go语言中，左括号必须紧接着语句不进行换行。有些开发者，特别是有其他编程语言开发经验的人，刚开始时会对这个特性感到不习惯，但随着对Go语言的不断深入熟悉，就会发现这个特性能让代码阅读者将更多精力投放到代码本身，而不是代码风格上去。

由于以上特性，Go语言主要应用于以下三个方面：（1）服务端开发；（2）高并发处理；（3）网络编程。

基于以上三个方面，Go语言可以具体用于服务端网络请求接收和解析、分布式系统开发、后端API接口开发和虚拟机处理等。

1.1.3 使用Go语言的项目

Go语言在2009年首次公开发布后受到越来越多的开发者的关注，Go语言社区不断壮大，开源项目不断涌现。下面列举使用Go语言进行开发的部分项目。

1. Docker项目

网址：<https://github.com/docker/docker-ce>

Docker是一个开源的应用容器引擎，基于Go语言开发并遵从Apache 2.0协议开源。Docker可以让开发者打包他们的应用和依赖包到一个轻量级、可移植的容器中，也可以实现虚拟化，更重要的是容器性能开销极低。

2. Golang项目

网址：<https://github.com/golang/go>

Go语言自1.5版本后，可以完全使用Go语言自身来实现。对Go语言的源码研究有助于深入理解Go的底层实现，想对Go语言有持续深入了解的读者可阅读此项目。

3. Kubernetes项目

网址：<https://github.com/kubernetes/kubernetes>

Kubernetes是自动化容器操作的开源平台，使用Kubernetes可以进行自动化容器的部署和复制、随时扩展或收缩容器规模和容器间的负载均衡等。

4. Beego项目

网址：<https://github.com/astaxie/beego>

Beego是一个使用Go编写的极轻量级、高可伸缩性和高性能的Web应用框架。

5. Codis项目

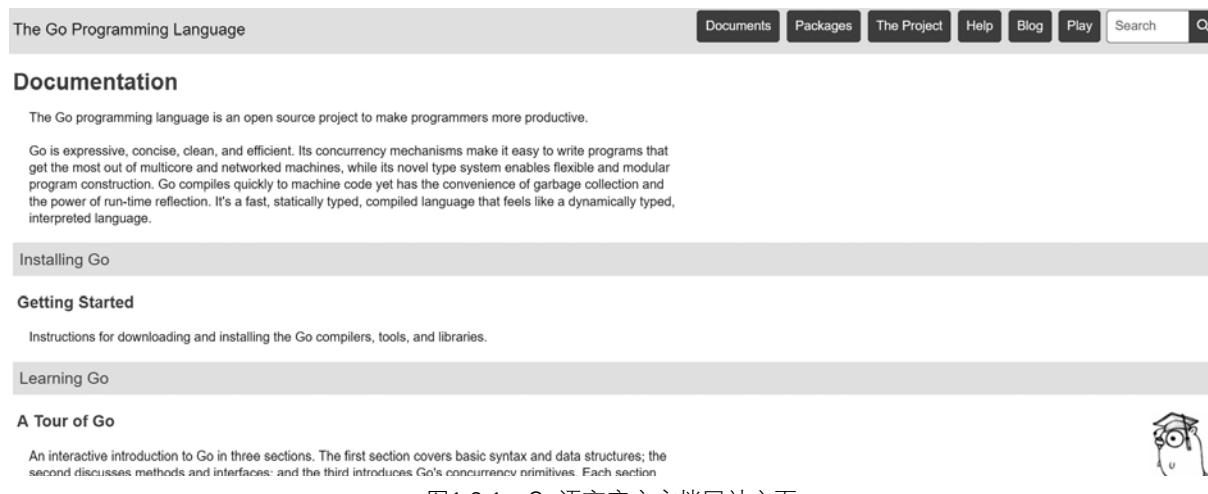
网址：<https://github.com/CodisLabs/codis>

Codis是一个国产开源的分布式redis解决方案，对于上层的应用透明，就像连接到原生的redis服务器那样。

1.2 Go语言官方文档

Go语言官方文档网址：<https://golang.org/doc/>。官方文档中有关于如何写Go代码的建议、编辑器推荐和Go语言的发布历史等内容。

Go语言官方文档网站主页如图1.2.1所示。



The Go programming language is an open source project to make programmers more productive.

Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.

Installing Go

Getting Started

Instructions for downloading and installing the Go compilers, tools, and libraries.

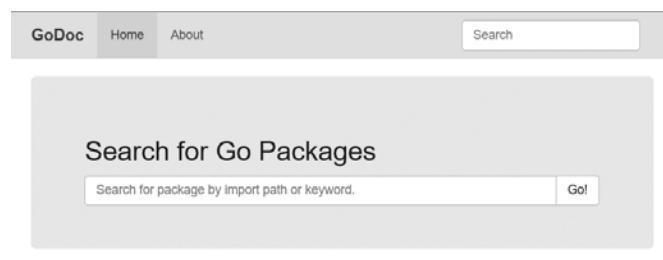
Learning Go

A Tour of Go



图1.2.1 Go语言官方文档网站主页

如果要查看Go语言标准库或其他库函数的用法，可参见官方的包索引与搜索引擎网站：<https://godoc.org/>，里面有详细的函数用法和样例，主页如图1.2.2所示。



GoDoc hosts documentation for Go packages on Bitbucket, GitHub, Google Project Hosting and Launchpad.
Read the About Page for information about adding packages to GoDoc and more.

Popular Packages

github.com/Shopify/sarama
github.com/aws/aws-sdk-go
github.com/dgrijalva/jwt-go
github.com/gln/gonic/gln
github.com/go-redis/redis
github.com/go-sql-driver/mysql
github.com/gomodule/redigo/redis
github.com/gorilla/websocket
github.com/hyperledger/fabric/core/chaincode/shim
github.com/jinzhu/gorm
github.com/libpq
github.com/pkg/errors
github.com/prometheus/client_golang/prometheus
github.com/robfig/cron
github.com/sirupsen/logrus
github.com/stretchr/testify/assert

More Packages

[Go Standard Packages](#)
[Go Sub-repository Packages](#)
[Projects @ go-wiki](#)
Most stars, most forks, recently updated on GitHub

图1.2.2 Go语言官方包索引与搜索引擎网站主页

1.3 学好Go的建议

在没有编程基础的情况下，初学者开始学习任何一门编程语言都会比较困难，经常会不知道从哪里开始学起，或即使看懂了，自己也写不出代码，甚至会中途放弃，希望本节内容可以帮助大家更好地去学习Go。

1. 了解语言特性及自身需求

首先，每种编程语言都有自己的特性及应用场景，正所谓好钢使在刀刃上，我们不可能用纯C语言去写一个类似于WordPress（使用PHP语言开发）的内容管理系统，因为没有必要而且极度浪费时间。我们也不可能用PHP语言去开发软件，进行一些相对底层的操作，如写一个驱动安装程序。

其次，你需要厘清自身需求，比如是业余兴趣研究还是需要用来解决工作中的实际业务问题。如果是后者且已经有一定编程基础，在学习时你可以有所侧重，选择性地针对用到的相关包和相关Go语言特性进行学习。

2. 动手写代码

学习编程语言，一开始最重要的就是学习这门语言的语法，语法就如同这门编程语言的词汇表。不断地学习Go语法，查阅相关代码，自己动手写每一个简短的例子。本书每一个知识点之后都有“动手写”的代码示例，每一个例子都是可以真实运行的，动手运行这些例子有助于初学者更好地理解Go。

经过不断地重复练习，对于很多语法有了一定认识后，你就要开始举一反三。比如在学习“文件操作”这一章时，书中会讲到使用Go写文件，这里就可以举一反三：Go有几种写文件的方式？这几种写文件方式的执行结果是什么？如果两个Go程序同时向一个文件写入内容会怎么样？学新知识的时候，你要不断地提出这类问题，通过编写代码进行测试并找到答案，这样，知识的覆盖面也会更广。

3. 参与项目

在自己能动手写一些简单代码后，你可以尝试参与到实际的Go项目中去。不管这些项目是工作所需还是兴趣驱动，在项目中你会遇到很多实际的问题，学会去思考并解决它们，在这过程中和项目组的其他小伙伴们互相交流学习并协作解决问题。所以，参与到实际项目中，不仅能得到成长，还能结识许多志同道合的小伙伴，何乐而不为呢？

4. 阅读英文文档

还有一点就是需要学好英语。目前一些Go语言文档有中文版本，但还有许多文档都只有英文版本。学会阅读英文文档是一项非常基础的能力，毕竟Go语言和其他大多数编程语言一样，都是由外国人编写的，而网上的中文文档翻译质量参差不齐，所以提高自身英语阅读能力才是硬道理。

1.4 常用软件及网站

1. 开发工具（IDE或编辑器）

Go开发工具有许多，其中IDE（Integrated Development Environment，集成开发环境）功能比较强大，会提供代码提示、文件和目录管理、代码搜索和替换、查找函数等功能，因此工程师一般会通过IDE进行代码开发。文本编辑器功能比较简单，但是有的编辑器例如Visual Studio Code和Sublime等也可以通过安装插件来达到IDE所提供的大部分功能。

Visual Studio Code，是一个由微软开发的，同时支持Windows、Linux和Mac OS操作系统并且开放源代码的文本编辑器。它支持调试，并内置了Git版本控制功能，同时也具有开发环境功能，例如代码补全、代码片段、代码重构等。该编辑器支持用户自定义配置，例如改变主题颜色、键盘快捷方式、编辑器属性和其他参数，还支持扩展程序并在编辑器中内置了扩展程序管理的功能。

GoLand是JetBrains推出的一款新型商用IDE，旨在为Go开发提供符合人体工程学的环境。新的IDE扩展了IntelliJ平台，提供Go语言特有的编码辅助和工具集成，强大的静态代码分析和人体工程学设计使开发更高效。

2. 代码管理工具

一个网站通常由多个开发人员共同完成，而代码管理工具可以记录一个项目从开始到结束的整个过程，追踪项目中所有内容的变化情况，如增加了什么内容、删除了什么内容、修改了什么内容等。它还可以管理网站的版本，让开发人员清楚地知道每个版本之间的异同点，如版本2.0与版本1.0相比多了什么内容和功能等。开发人员通过代码管理工具进行权限控制，能防止代码混乱，提高安全性，防止一些不必要的损失和麻烦。

SVN（Subversion）是一个开源的集中式版本控制系统，管理随时间改变的数据，所有数据集中存放在中央仓库（repository）。repository就好比一个普通的文件服务器，不过它会记住每一次文件的变动，这样你就可以把Java文件恢复成旧的版本，或是浏览Java文件的变动历史。

GIT，是一个开源的分布式版本控制系统，和SVN功能类似，但GIT的每台电脑都相当于一个服务器，代码是最新的，比较灵活，可以有效、高速地处理项目版本管理。全球最大的代码托管平台GitHub网站，采用的也是GIT技术。

3. 其他工具

Jira，是Atlassian公司出品的项目与事务跟踪工具，可以进行网站bug管理、缺陷跟踪、任务跟踪和敏捷管理等。

Redmine，是由Ruby编程语言开发的一套跨平台项目管理系统，通过“项目（Project）”的形式把成员、任务（问题）、文档、讨论以及各种形式的资源组织在一起，让大家参与更新任务、文档等内容来推动项目的进度，同时利用时间线索和各种动态的报表形式来自动给成员汇报项目进度，并提供Wiki、新闻台等，还可以集成其他版本管理系统和bug跟踪系统。

XMind，是一款实用的思维导图软件，可以用来绘制产品架构图、项目流程图、功能分解图等。XMind简单易用、界面美观、功能强大，拥有高效的可视化思维模式，具备可扩展、跨平台、稳定性强等性能，真正帮助用户提高生产率，促进有效沟通及协作。

TeamCola，是由国内团队开发的时间管理工具，能较好地解决时间问题，而其管理颗粒度为半小时，也不会过多增加管理成本。

4. 网站资源

在遇到代码bug或者包使用问题时，要学会用互联网资源解决问题，如官方文档、第三方IT类网站、技术论坛和个人博客等。

此处列举了一些常用的网站资源供参考：

◇ CSDN：创立于1999年，是中国最大的IT社区和服务平台。

网址：<https://www.csdn.net/>

◇ Go官方文档

网址：<https://golang.org/doc/>

◇ Go语言官方包索引与搜索引擎

网址：<https://godoc.org/>

◇ Stack Overflow：一个与程序相关的IT技术问答网站。用户可以在网站免费提交问题，浏览问题，索引相关内容。一般情况下，一些在CSDN找不到答案的问题都可以在Stack Overflow上找到。

网址：<https://stackoverflow.com/>

◇ GitHub：一个面向开源及私有软件项目的托管平台，目前已被微软收购。可在GitHub上搜索到开源的Go源代码和第三方库，提交issue或修改有问题的代码。

网址：<https://github.com>

1.5 小结

- ◇ Go语言也叫Golang，是由谷歌公司在2007年推出的一款静态编译型语言。Go语言高效、简洁且容易上手，加上它的高并发和自带垃圾回收等特性，一出现就备受大家的喜爱。
- ◇ 目前，越来越多的公司开始推广并使用Go语言，主要将其用于服务端开发、并发编程和网络编程等。
- ◇ Go语言起源于2007年，于2009年正式对外发布。Go语言项目主要由肯·汤普逊、罗布·派克和罗伯特·格里泽默参与开发。
- ◇ 学好Go语言的建议：了解语言特性及自身需求、动手写代码、参与项目和阅读英文文档。

>> 第 2 章

安装和运行 Go

<<

Go的安装主要分为两种方式：安装包安装和源代码安装。

安装包安装：即已编译好的可直接运行的程序，下载后可直接运行完成安装工作。

源代码安装：需自行编译源代码来完成安装工作。

一般情况下，我们推荐在Go官网下载最新的Go语言安装包来进行安装。如果操作系统满足以下要求（表2.1.1），那么你就可以使用安装包来安装Go，否则只能通过源代码来安装。

表2.1.1 Go安装包安装需满足的要求

操作系统	架构	备注
FreeBSD 10.3 或更高版本	amd64、386	不支持 Debian GNU/kFreeBSD
Linux 2.6.23 或更高版本带 glibc	amd64、386、arm、arm64	不支持 CentOS/RHEL 5.x
	s390x、ppc64le	从其他 libc 的源安装
Mac OS 10.10 或更高版本	amd64	使用 clang 或 gcc 自带对 cgo 支持的 Xcode
Windows 7, Server 2008R2 或更高版本	amd64、386	使用 MinGW gcc，不需要 Cygwin 或 MSYS

本章将演示在不同的操作系统上安装Go以及运行第一个Go程序。

2.1 GOROOT和GOPATH

在进行安装工作之前，我们首先需要了解GOROOT和GOPATH这两个Go语言中与环境变量配置相关的基本概念。

GOROOT环境变量很好理解，顾名思义即Go安装的根目录，一般在安装完成后，会自动完成对GOROOT环境变量的设置。

GOPATH是Go的主要工作目录，在使用go install和go get等go工具时会用到GOPATH环境变量，如果GOPATH环境变量没有正确设置，那么将无法进行Go的开发工作。

GOPATH目录下主要包含三个重要目录：

1. bin： Go编译可执行文件的存放路径。
2. pkg： Go编译包时生成的.a中间文件存放路径。
3. src： Go标准库源码路径。

2.2 在Windows下安装Go

在本节中，我们将学习如何在Windows下安装Go。Go可以在所有主流Windows下运行，包括且不限于Windows 7、Windows 8、Windows 8.1和Windows 10。

用任意浏览器访问Go语言官方下载地址<https://golang.google.cn/dl/>，即可看到Go安装包的下载列表（图2.2.1）。

以Win7 32位系统为例，选择“go1.10.3.windows-386.msi”安装包，鼠标点击进行安装包的下载。

完成下载后，双击安装程序进行安装。请接受大部分默认设置。

文件名	类型	操作系统(OS)	架构(Arch)	大小	SHA256 Checksum
go1.10.3.src.tar.gz	Source			17MB	567b1e66e9704d1e019e60be946f22ed11e61ba0244210bf74e670b155f2
go1.10.3.darwin-amd64.tar.gz	Archive	macOS	x86-64	124MB	131f8430950a134492e+75e+e456+ef4443+492d527d951e7e04a5458d
go1.10.3.darwin-amd64.pkg	Installer	macOS	x86-64	124MB	6+e28d0dd1f67e+e4e6fffe990162479020fd817b9af4a7b470fbdd1600+e
go1.10.3.freebsd-386.tar.gz	Archive	FreeBSD	x86	111MB	92a2fccc0bca173295490f7c1f10f3u+e40953le096315+e57d5027204
go1.10.3.freebsd-amd64.tar.gz	Archive	FreeBSD	x86-64	122MB	c31d9e6f315ac+e1193e+d1839e+e9f1e51570f5+e9u+e6e5b457a7f7f22995a2b
go1.10.3.linux-386.tar.gz	Archive	Linux	x86	115MB	3d8fe1932e904u1urb13d4e07ab5954ff+e381+e9e6e9490e9240eb4e+e
go1.10.3.linux-amd64.tar.gz	Archive	Linux	x86-64	126MB	e1b0e45d0647e2522515e1204ab049cd+e1f177e9f2101e43004e901035
go1.10.3.linux-arm64.tar.gz	Archive	Linux	ARMv8	110MB	955129e051456+e68782143001u10e0431510e3897640f71530b+92024+d2
go1.10.3.linux-armv6l.tar.gz	Archive	Linux	ARMv6	110MB	424f1u2d153e01041e24f31u02f0e21e1f92e15552f3e021u+220f0456
go1.10.3.linux-ppc64le.tar.gz	Archive	Linux	ppc64le	109MB	e3640b2e0990a9617e937775669a+18f10a2e+424e5e87ab+e794a407b8347
go1.10.3.linux-s390x.tar.gz	Archive	Linux	s390x	108MB	34395fe4051fb2fb11d433de410e2ab4e237bde8f7f3e43015e908e3e04
go1.10.3.windows-386.zip	Archive	Windows	x86	121MB	0904e2534d000f4u0061216e21024u+e6b2e750e541424e+e7f2+e177e+e146
go1.10.3.windows-386.msi	Installer	Windows	x86	104MB	2a39f-e126e04682d0b0942d7f6226e339940d7401b574a6084444d3f270
go1.10.3.windows-amd64.zip	Archive	Windows	x86-64	133MB	a3f194f+e04b48836b349603e347e64e31ab830e+e2fe+e39083644418+e
go1.10.3.windows-amd64.msi	Installer	Windows	x86-64	114MB	e36975165340e+e9+e12120e09bb10f053e76660a12705e0fd1fb5e64f1660e

图2.2.1 下载Go的Windows版本安装包

安装程序显示安装完成后（图2.2.2），打开命令提示符并执行go version，如果你看到屏幕打印出Go的版本（图2.2.3），就证明你正确地完成了Go的安装。

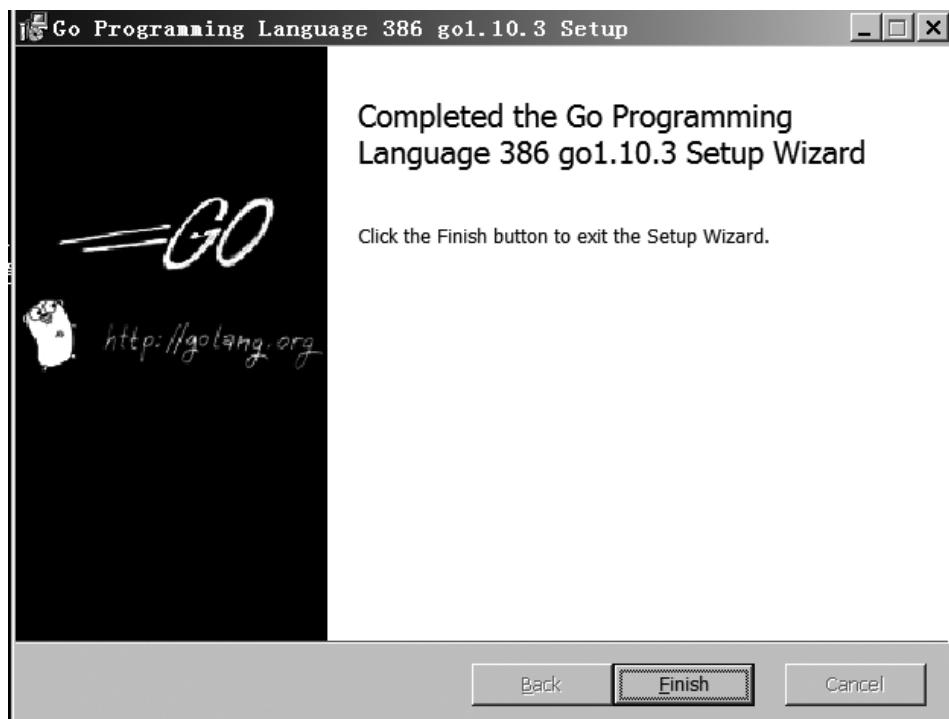
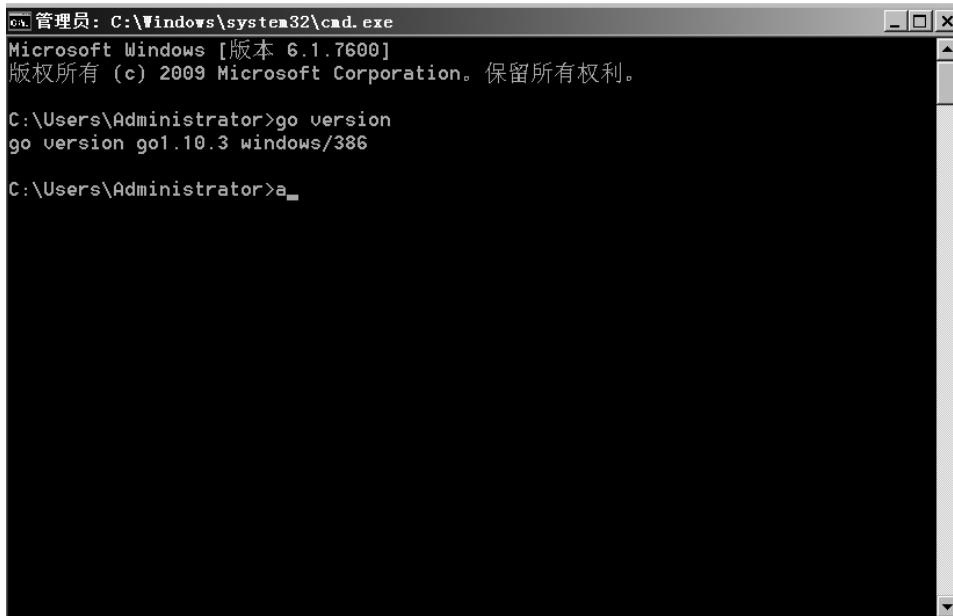


图2.2.2 成功在Windows下安装Go



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7600]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

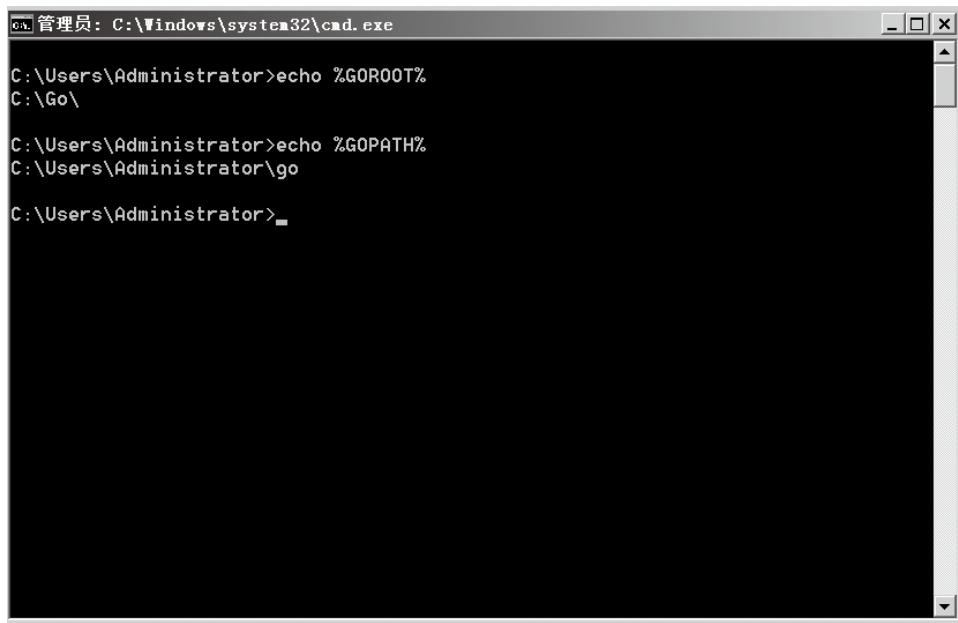
C:\Users\Administrator>go version
go version go1.10.3 windows/386

C:\Users\Administrator>a_
```

图2.2.3 输出Go安装的版本

在Windows系统中按照默认设置完成Go的安装后，系统会自动设置GOROOT和GOPATH环境变量的值。

本例中，GOROOT环境变量值为：C:\Go\，GOPATH值为：C:\Users\Administrator\go。（图2.2.4）



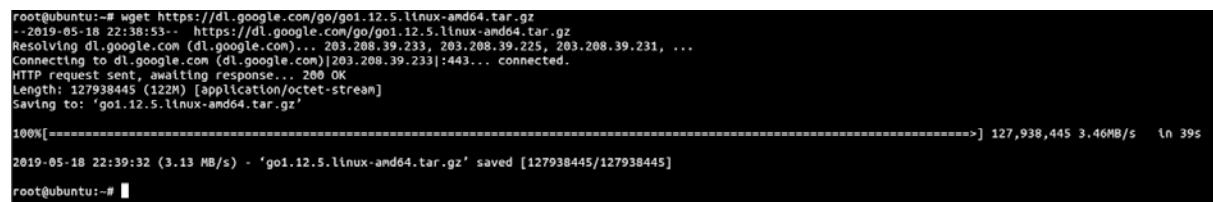
```
管理员: C:\Windows\system32\cmd.exe
C:\Users\Administrator>echo %GOROOT%
C:\Go\
C:\Users\Administrator>echo %GOPATH%
C:\Users\Administrator\go
C:\Users\Administrator>_
```

图2.2.4 GOROOT和GOPATH环境变量值

2.3 在Linux下安装Go

本节主要介绍在Linux下安装Go的方法。由于Linux有众多发行版本，本节会以Ubuntu 14.04作为示例进行安装介绍，其他发行版本请参考相关发行版本手册。

首先，我们以root用户登录系统，通过wget命令下载Linux版本的安装包（图2.3.1），安装包下载地址为：<https://dl.google.com/go/go1.12.5.linux-amd64.tar.gz>。



```
root@ubuntu:~# wget https://dl.google.com/go/go1.12.5.linux-amd64.tar.gz
--2019-05-18 22:38:53-- https://dl.google.com/go/go1.12.5.linux-amd64.tar.gz
Resolving dl.google.com (dl.google.com)... 203.208.39.233, 203.208.39.225, 203.208.39.231, ...
Connecting to dl.google.com (dl.google.com)|203.208.39.233|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 127938445 (122M) [application/octet-stream]
Saving to: 'go1.12.5.linux-amd64.tar.gz'

100%[=====] 127,938,445 3.46MB/s   in 39s
2019-05-18 22:39:32 (3.13 MB/s) - 'go1.12.5.linux-amd64.tar.gz' saved [127938445/127938445]
root@ubuntu:~#
```

图2.3.1 下载Linux版本安装包

使用tar命令，将安装包解压到指定目录下，此处我们解压到/usr/local目录下：tar -C /usr/local-xzf go1.10.3.linux-amd64.tar.gz。

解压完成后，/usr/local/go目录下会新增如下文件。（图2.3.2）

```
root@ubuntu:/usr/local/go# ls -al /usr/local/go/
total 216
drwxr-xr-x 10 root root 4096 May  6 14:14 .
drwxr-xr-x 14 root root 4096 May 18 22:44 ..
drwxr-xr-x  2 root root 4096 May  6 14:14 api
-rw-r--r--  1 root root 55358 May  6 14:14 AUTHORS
drwxr-xr-x  2 root root 4096 May  6 14:17 bin
-rw-r--r--  1 root root 1339 May  6 14:14 CONTRIBUTING.md
-rw-r--r--  1 root root 78132 May  6 14:14 CONTRIBUTORS
drwxr-xr-x  8 root root 4096 May  6 14:14 doc
-rw-r--r--  1 root root 5686 May  6 14:14 favicon.ico
drwxr-xr-x  3 root root 4096 May  6 14:14 lib
-rw-r--r--  1 root root 1479 May  6 14:14 LICENSE
drwxr-xr-x 13 root root 4096 May  6 14:14 misc
-rw-r--r--  1 root root 1303 May  6 14:14 PATENTS
drwxr-xr-x  6 root root 4096 May  6 14:17 pkg
-rw-r--r--  1 root root 1607 May  6 14:14 README.md
-rw-r--r--  1 root root  26 May  6 14:14 robots.txt
drwxr-xr-x 46 root root 4096 May  6 14:14 src
drwxr-xr-x 21 root root 12288 May  6 14:14 test
-rw-r--r--  1 root root    8 May  6 14:14 VERSION
root@ubuntu:/usr/local/go#
```

图2.3.2 Go安装目录

接下来，我们编辑/etc/profile文件，将“export PATH=\$PATH:/usr/local/go/bin”写入文件末尾并保存，即将/usr/local/go/bin目录添加到系统的PATH环境变量中。

最后，我们可以使用go env来检验Go是否成功安装以及各个环境变量的值。（图2.3.3）

```
root@ubuntu:~# go env
GOARCH="amd64"
GOBIN=""
GOCACHE="/root/.cache/go-build"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/root/go"
GOPROXY=""
GORACE=""
GOROOT="/usr/local/go"
GOTMPDIR=""
GOTOOLDIR="/usr/local/go/pkg/tool/linux_amd64"
GCCGO="gccgo"
CC="gcc"
CXX="g++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/tmp/go-build983965328=/tmp/go-build -fno-record-gcc-switches"
root@ubuntu:~#
```

图2.3.3 成功在Linux下安装Go

2.4 在Mac OS下安装Go

在Mac OS下安装Go与在Windows下安装类似，用任意浏览器访问Go语言官方下载地址<https://golang.google.cn/dl/>，选择“Apple macOS”一栏中的“go1.13.darwin-amd64.pkg”，鼠标点击进行下载。

完成下载后，双击安装程序进行安装。请接受大部分默认设置。

2.5 安装开发工具

互联网有很多可用的开发工具（IDE），对于Go开发者来说，选一款最好用的工具，可以更高效地编码和构建项目。GoLand是一款由JetBrains公司（一家技术领先的软件开发公司，专门从事增效型智能软件的开发）开发推出的商业集成开发工具。GoLand具备安装即用的可视化调试器，可靠的IDE功能自动执行常规任务，包含多种前沿技术，支持远程部署等工具，随时随地轻松调试，非常适合开发Go项目，本书推荐使用GoLand。

2.5.1 GoLand下载

下载地址：<http://www.jetbrains.com/go/>

该软件是商业软件，需要付费使用，有30天的试用期。

2.5.2 GoLand安装

下载GoLand安装程序后，双击GoLand安装包，进入欢迎安装界面，然后点击“Next”按钮继续安装，如图2.5.1所示。



图2.5.1 GoLand欢迎界面

进入“选择安装目录”界面，选择安装的目录，然后点击“Next”按钮继续安装，如图2.5.2所示。



图2.5.2 GoLand选择目录

如图2.5.3所示，进入“安装选项”界面，建议全选，这些后缀文件都将会默认使用GoLand打开，然后点击“Next”按钮继续安装。

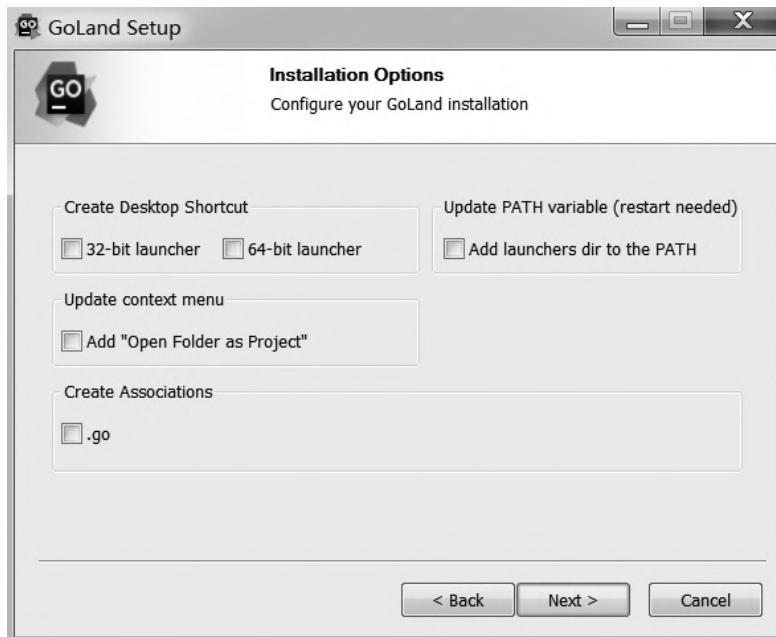


图2.5.3 GoLand安装选项

接下来进入“选择菜单文件夹”，默认即可，然后点击“Install”按钮进行安装，如图2.5.4所示。



图2.5.4 GoLand菜单文件夹

如图2.5.5所示，GoLand正在安装，等到进度条刷新完成，会进入到软件安装完成界面，最后点击“Finish”按钮即可。

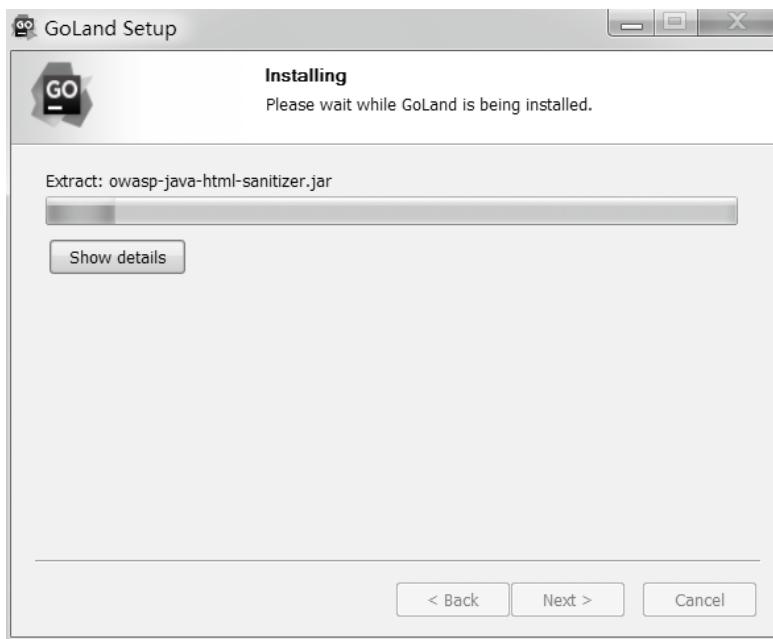


图2.5.5 GoLand安装过程

2.5.3 GoLand使用

第一次打开GoLand时，会提示选择新建项目还是打开已有项目，如图2.5.6所示。

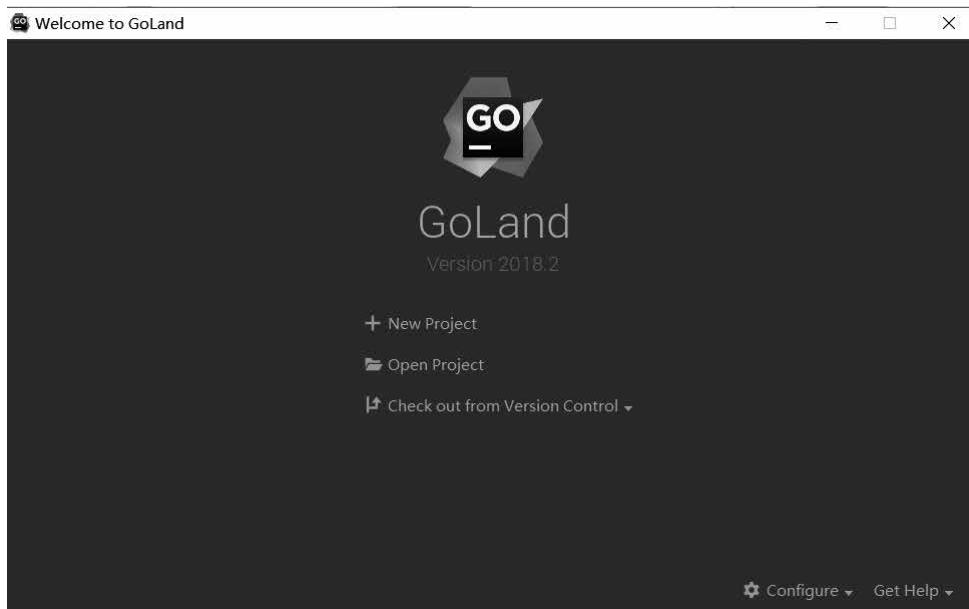


图2.5.6 第一次打开GoLand

此处我们选择新建项目，项目名为“01快学”，项目位置为：F:\goland\mygo\src\01快学，点击“Create”创建新项目。（图2.5.7）

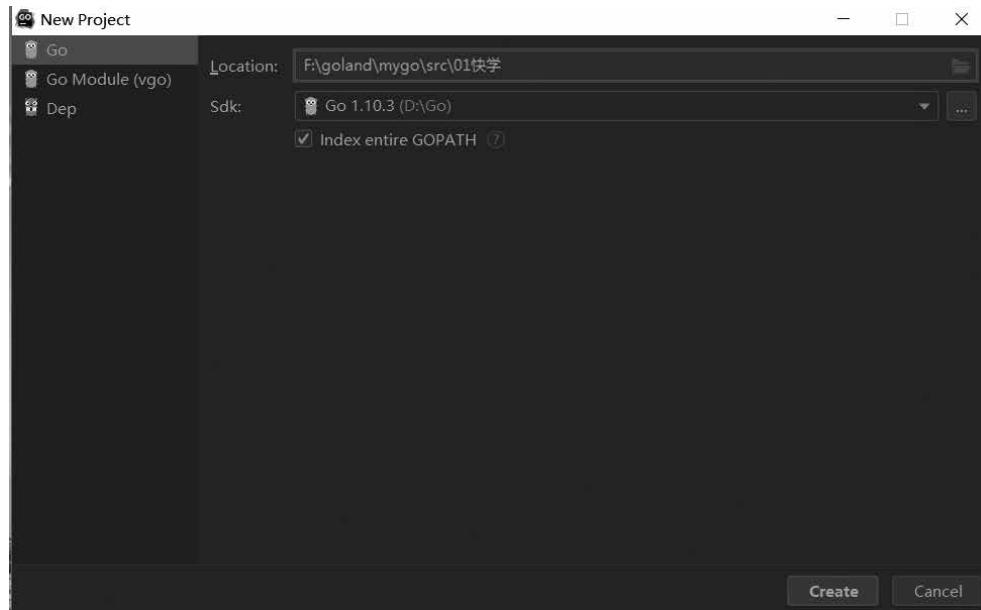


图2.5.7 创建“01快学”项目

项目成功创建后，GoLand会自动打开“01快学”项目。（图2.5.8）

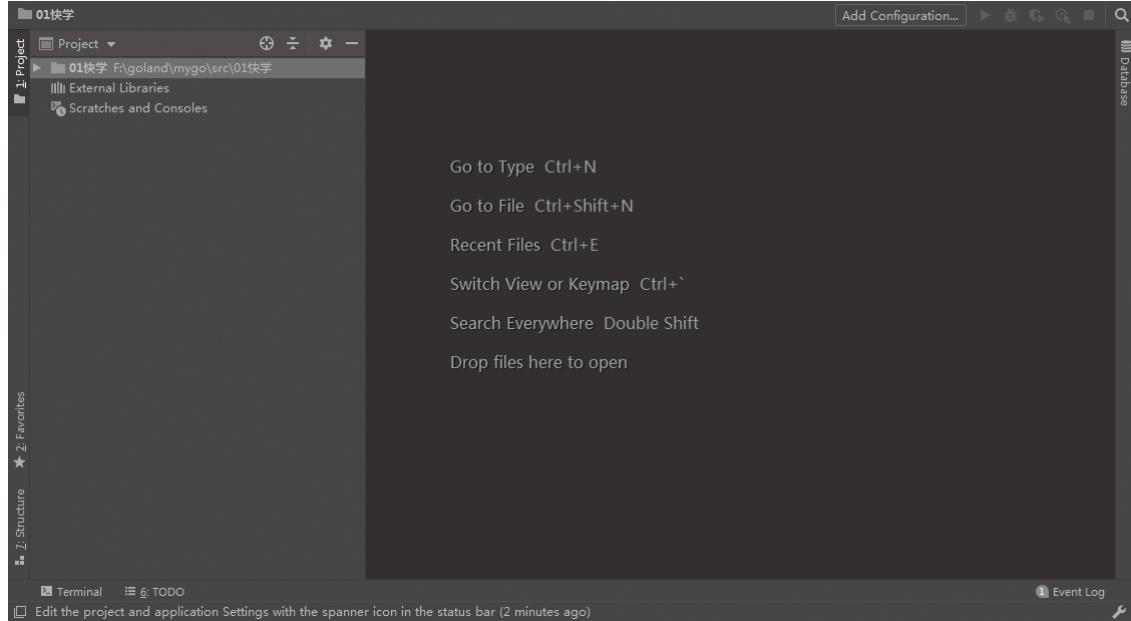


图2.5.8 “01快学”项目

左上方项目名处，右键选择“New”“Go File”（图2.5.9），File Name处输入“test”（图2.5.10），创建一个新的Go源文件。

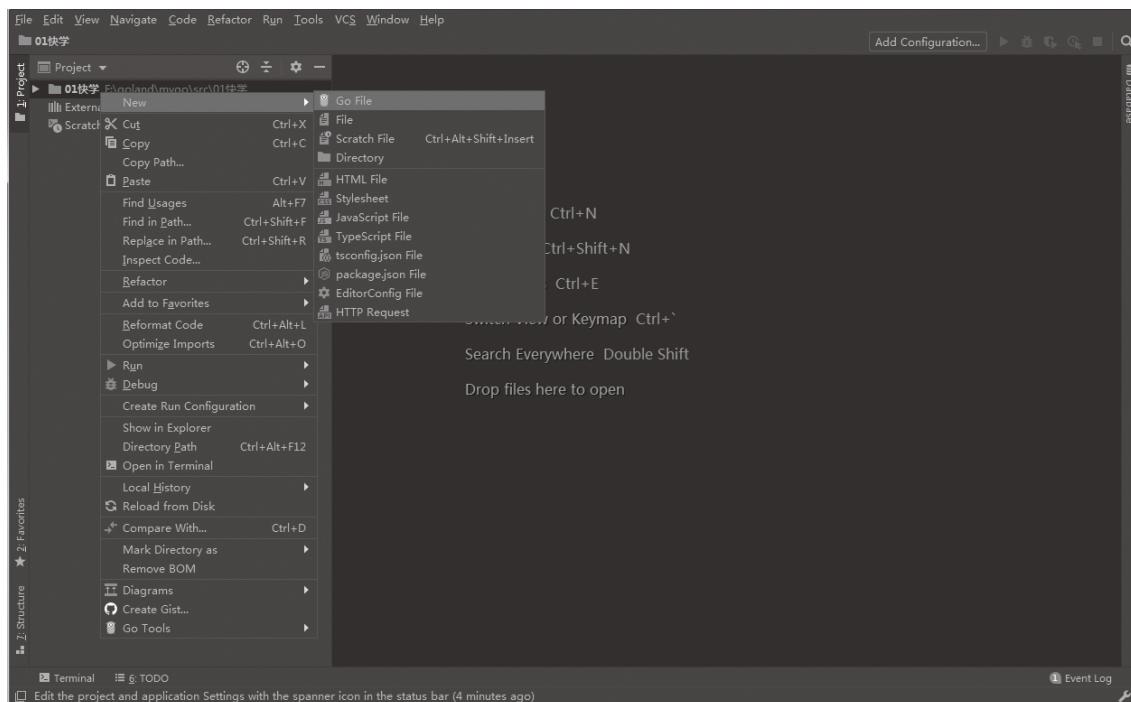


图2.5.9 创建新文件

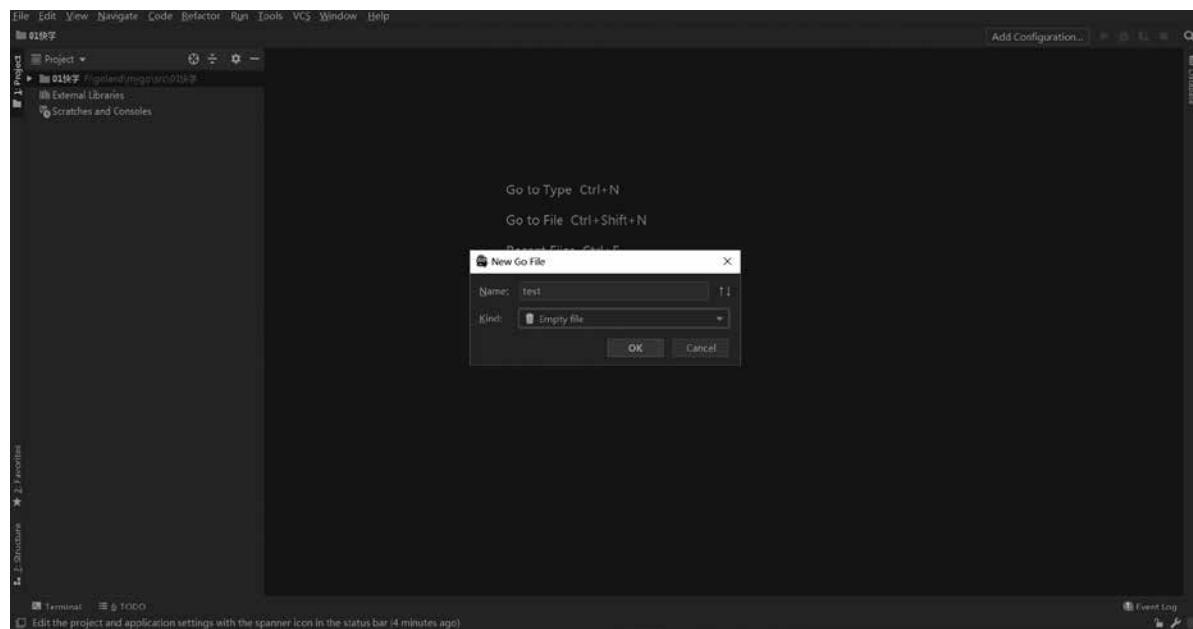


图2.5.10 文件名为“test”

如图2.5.11所示，可以看到我们创建的test.go文件，双击文件查看内容，现在我们可以使用GoLand来编写我们的第一个Go程序了。

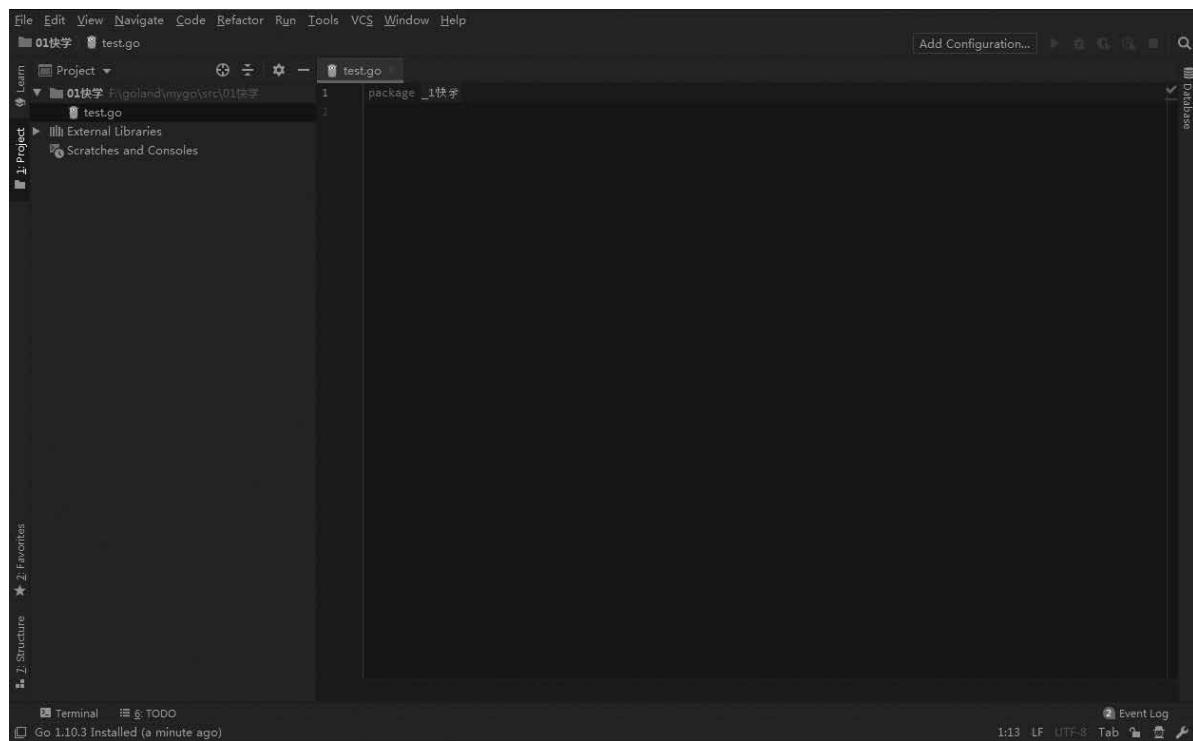


图2.5.11 test.go文件内容

2.6 编写第一个Go程序

Go和GoLand安装完成后，我们可以使用Go语言来编写我们的第一个程序——Hello world。

一般推荐使用GoLand编写Go程序，当然普通的文本编辑器也可进行编写。

2.6.1 GoLand编写

在图2.5.11中，我们输入如下代码，这段代码的作用就是在屏幕上打印“Hello world”。

动手写2.6.1

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     fmt.Println("Hello world")
07 }
```

如图2.6.1所示，在代码输入区域点击右键，选择“Run'go build test.go'”，编译并运行test.go文件。

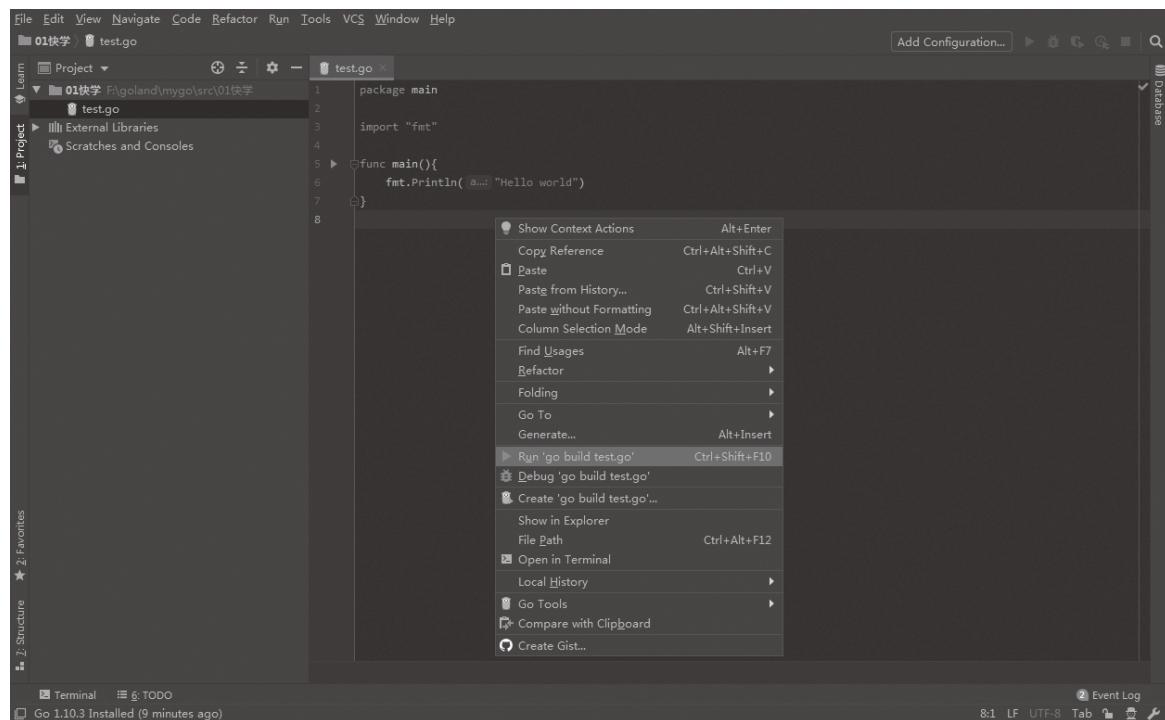


图2.6.1 编译并运行test.go

GoLand输出结果如图2.6.2所示。

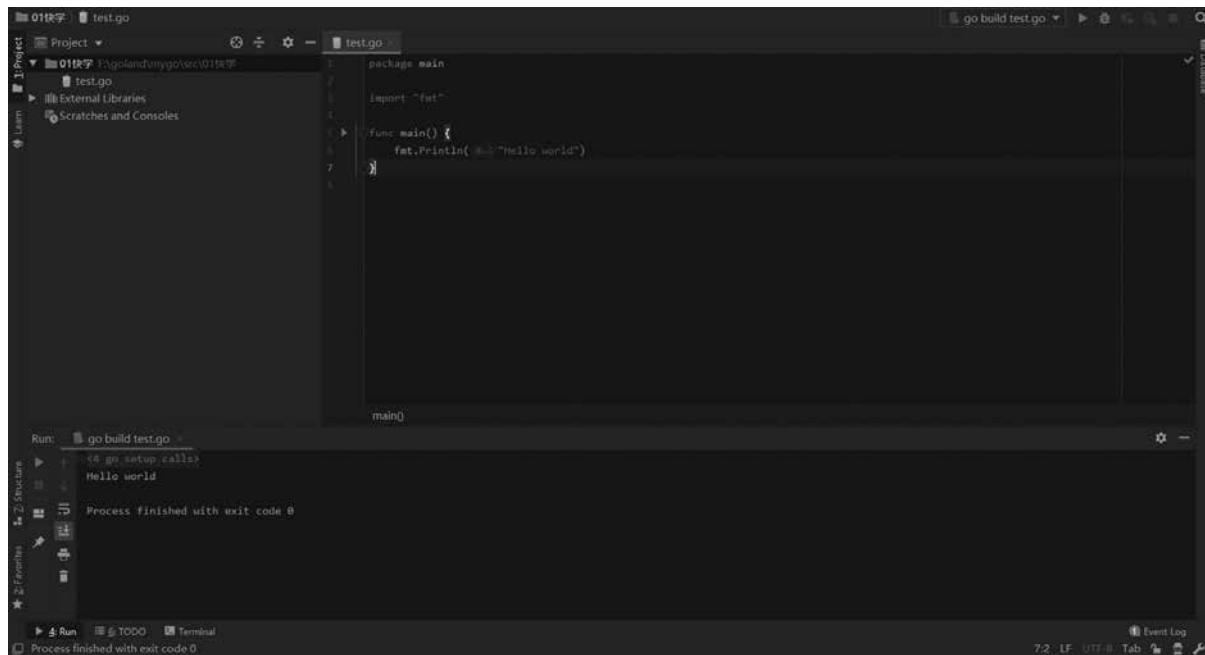


图2.6.2 GoLand输出结果

2.6.2 文本编辑器编写

我们新建一个文件，名字为“main.go”，输入代码如下：

动手写2.6.2

```
01 package main  
02  
03 import "fmt"  
04  
05 func main() {  
06     fmt.Println("Hello world")  
07 }
```

调用命令行，进入到main.go所在目录下，使用go run main.go命令来编译并运行我们的第一个程序，执行结果如图2.6.3所示。

A screenshot of a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The window shows the following text:

```
Microsoft Windows [版本 10.0.17134.706]
(c) 2018 Microsoft Corporation。保留所有权利。

F:\goland\mygo\src\test>go run main.go
Hello world

F:\goland\mygo\src\test>
```

The window has standard Windows UI elements like minimize, maximize, and close buttons.

图2.6.3 运行Hello world程序

2.7 小结

- ◇ 推荐通过在Go官网下载最新的Go语言安装包来进行安装工作。
- ◇ Go语言官方下载地址为<https://golang.google.cn/dl/>。
- ◇ GoLand具备安装即用的可视化调试器，可靠的IDE功能自动执行常规任务，包含多种前沿技术，支持远程部署等工具，可随时随地轻松调试，非常适合开发Go项目，本书推荐使用GoLand。

>> 第 3 章

本变量与类型

<<

学习任何一门编程语言，都要从语言的基础语法学起。变量与数据类型是编程中最常见且最基础的概念。本章将介绍Go语言中基本的变量和数据类型。

如果读者有其他编程语言基础，特别是C语言，那么对本章内容的学习将会非常轻松。如果读者没有编程语言基础也没有关系，因为本章的内容易于理解。

3.1 Go语言变量

变量是用来存储用户数据的。在编程过程中，会有许多不同的对象类型，那么对于不同的对象类型，我们就需要不同的变量类型来存储不同类型的用户数据。

编程语言中常见的数据类型有整型、浮点型、布尔型和结构体等。Go语言的每个变量都有自己的类型，在使用它们之前，必须先经过声明。

3.1.1 声明变量

1. 标准格式

变量的声明以关键字var开头，后接变量类型，行尾没有其他标点符号。

Go语言变量的标准声明格式如下：

```
var 变量名 变量类型
```

例如，声明变量num的类型为int：

```
var num int
```

2. 批量格式

在实际编程过程中，我们往往需要同时声明多个变量，这种情况下我们可以使用批量格式。

使用关键字var和小括号，可以同时声明多个变量。

Go语言变量的批量声明格式如下：

```
var (
    a int
    b string
    c bool
)
```

3.1.2 初始化变量

Go语言在声明变量时，会自动对每个变量对应的内存区域进行初始化，即每个变量会有其类型对应的默认值：

- ◇ 整型和浮点型变量的默认值为0。
- ◇ 字符串变量的默认值为空字符串。
- ◇ 布尔型变量默认为false。
- ◇ 切片、映射、函数和指针变量默认为nil。

注意：nil相当于其他编程语言中的null、None和NULL等，指代零值，在Go语言中只能赋值给切片、映射、函数、接口、指针或通道类型。

1. 标准格式

Go语言变量的标准初始化格式如下：

```
var 变量名 变量类型 = 表达式
```

例如，声明变量num类型为int并赋值为1：

```
var num int =1
```

2. 编译器推导类型格式

在标准格式基础上，把变量类型省略后，编译器会根据等号右边的表达式推导变量的类型。

例如，使用编译器推导类型格式初始化变量age值为20：

```
var age =20
```

3. 短变量声明并初始化

变量声明并初始化有一种更为简洁的写法，例如：

```
age :=30
```

使用短变量声明并初始化时，编译器同样会根据等号右边的表达式推导变量的类型。

相对于前两种变量的初始化方法，短变量声明并初始化方法在开发中使用最为普遍。

使用这种方法，我们可以同时给多个变量赋值，例如：

动手写3.1.1

```
01 package main
02 import "fmt"
03 func main() {
04     name, age := "Tom", 18
05     fmt.Println("name:", name)
06     fmt.Println("age:", age)
07 }
```

执行结果如下：

```
name: Tom
age: 18
```

但是，使用短变量声明并初始化时，需注意变量重复声明的情况：

动手写3.1.2

```
01 package main
02 import "fmt"
03 func main() {
04     var name string
05     name := "Tom"
06     fmt.Println("name:", name)
07 }
```

执行结果如下：

```
# command-line-arguments
.\test.go:5:7: no new variables on left side of :=
Compilation finished with exit code 2
```

很显然，编译器报错：在“:=”的左边没有新变量出现，因为name变量已经在代码的第四行中被声明了。

动手写3.1.3

```

01 package main
02 import "fmt"
03 func main() {
04     var name string
05     name, age := "Tom", 18
06     fmt.Println("name:", name)
07     fmt.Println("age:", age)
08 }
```

执行结果如下：

```

name: Tom
age: 18
```

在多个短变量声明并初始化中，应至少有一个新声明的变量出现在左值中，这样即使其他变量名存在重复声明的情况，编译器也不会报错。

3.1.3 变量值交换

实际编程中，我们经常会用到变量值的交换，比如需要交换变量a和变量b的值，可通过中间变量c进行交换，代码如下：

动手写3.1.4

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     var c int
07     a:= 1
08     b:= 2
09     c=a      //变量c临时存储变量a的值
10     a=b
11     b=c
12     fmt.Println(a)
13     fmt.Println(b)
14 }
```

执行结果如下：

```
2
1
```

在Go语言中，变量值的交换可以不再使用这种复杂的写法，而是通过多重赋值的特性，完成变量值的交换工作。

动手写3.1.5

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     a := 1
07     b := 2
08     a, b = b, a
09     fmt.Println(a)
10     fmt.Println(b)
11 }
```

执行结果如下：

```
2
1
```

多重赋值时，变量的左值和右值按照从左到右的顺序依次赋值。

3.1.4 匿名变量

在赋值给多个变量时，如果存在不需要接收值的变量，可以使用匿名变量来代替。

动手写3.1.6

```
01 package main
02 import "fmt"
03 func ReturnData() (int, int) {
04     return 10, 20
05 }
06
```

```

07 func main() {
08     a,_ := ReturnData()
09     _,b := ReturnData()
10     fmt.Println(a, b)
11 }
```

执行结果如下：

```
10 20
```

ReturnData()是我们自定义的一个函数，对于它的每次调用，会返回10和20两个整型数值。

调用ReturnData()函数，使用变量a接收第一个返回值，第二个返回值由匿名变量接收。

调用ReturnData()函数，使用变量b接收第二个返回值，第一个返回值由匿名变量接收。

由于匿名变量不占用命名空间，也不会分配系统内存，匿名变量与匿名变量之间不会因为多次声明而无法使用。

3.2 Go语言数据类型

Go语言的数据类型十分丰富，常见的包括整型、浮点型、字符串和布尔型等。Go语言特有的数据类型包括接口和通道等。本节将会对Go语言中常用的数据类型进行讲解。

3.2.1 整型

整型主要分为有符号整型和无符号整型两大类：

1. 有符号整型：int8、int16、int32、int64。
2. 无符号整型：uint8、uint16、uint32、uint64。

有符号整型其二进制最高位储存符号，因此两者的区别就是无符号整型可以存放的正数范围比有符号整型中的正数范围大一倍。

例如：int16的范围为-32768（即 -2^{15} ）到32767（即 $2^{15}-1$ ），uint16的范围为0到65535（即 $2^{16}-1$ ）。

动手写3.2.1

```

01 package main
02 import "fmt"
03
04 func main() {
```

```
05     a := 3
06     b := 2
07     fmt.Println(a/b)
08 }
```

执行结果如下：

```
1
```

注意：在Go语言中，对于两个整型变量的除法运算，小数部分将会直接被截取，只取整数部分，不会存在四舍五入的情况。

3.2.2 浮点型

编程语言中的浮点数就是我们常说的小数。

Go语言支持两种浮点数：float32和float64。float32浮点数的最大范围约为3.4e38，float64浮点数最大范围约为1.8e308。

那么对于动手写3.2.1，我们可以用以下方式获取除法计算的精确值。

动手写3.2.2

```
01 package main
02 import (
03     "fmt"
04     "reflect"
05 )
06
07 func main() {
08     a := 3.0
09     b := 2.0
10     fmt.Println(a/b)
11     fmt.Println("变量a的类型为：" , reflect.TypeOf(a))
12     fmt.Println("变量b的类型为：" , reflect.TypeOf(b))
13 }
```

执行结果如下：

```
1.5
```

变量a的类型为：float64

变量b的类型为：float64

此处，我们调用reflect.TypeOf()函数来打印变量a和b的类型。

在3.1.2小节中，我们提到过短变量声明并初始化的方法。由于代码所在计算机为64位系统，我们通过赋值时带上小数点来告诉编译器该变量类型为float64，这样除法得到的结果也是float64类型，即可以显示出精确结果。

3.2.3 字符串

Go语言中，字符串的值为双引号中的内容，而且可以直接输入中文。

动手写3.2.3

```
01 package main
02 import (
03     "fmt"
04 )
05
06 func main() {
07     str := "你好：Hello"
08     fmt.Println(str)
09 }
```

执行结果如下：

```
你好：Hello
```

实际编程中，我们会遇到输入多行字符串的情况，此时需要使用“`”字符，即反引号。

动手写3.2.4

```
01 package main
02 import (
03     "fmt"
04 )
05
06 func main() {
07     str := `这是第一行
08 这是第二行
09 这是第三行
10 `
11     fmt.Println(str)
12 }
```

执行结果如下：

```
这是第一行
这是第二行
这是第三行
```

3.2.4 字符和转义字符

1. 字符

字符串中的每个元素就是字符。

Go语言中，字符的值为单引号中的内容，而且可以直接输入中文。

Go语言的字符有以下两种类型：

- ◇ `uint8`类型：代表了ASCII码的一个字符。
- ◇ `rune`类型：代表了UTF格式的一个字符（如中文、日文或其他复合字符），本质是`int32`类型。

动手写3.2.5

```
01 package main
02 import (
03     "fmt"
04 )
05
06 func main() {
07     english := 'a'
08     chinese := '我'
09     fmt.Println(english)
10     fmt.Println(chinese)
11 }
```

执行结果如下：

```
97
25105
```

第一行打印了字符“a”的ASCII码值97，第二行打印了中文“我”的`int32`类型值25105。

2. 转义字符

通常我们使用反斜线“\”来对字符进行转义，转义字符具有特定的含义，不同于字符原有的

意义，所以称为转义字符。

常见的转义字符如表3.2.1所示。

表3.2.1 Go语言常见转义符

转义符	含义
\n	匹配换行符
\r	匹配回车符
\t	匹配制表符
'	匹配单引号
"	匹配双引号
\\"	匹配反斜杠

3.2.5 布尔型

布尔型是最简单的数据类型，只有两个值：false（假）和true（真）。

动手写3.2.6

```

01 package main
02 import "fmt"
03
04 func main() {
05     var a bool
06     fmt.Println(a)
07     a= true
08     fmt.Println(a)
09 }
```

执行结果如下：

```

false
true
```

3.2.6 数据类型判断

如果我们需要判断变量的类型，可以使用Go语言标准库中的reflect包，通过将变量传入此包的TypeOf()方法，得到变量的数据类型。

动手写3.2.7

```
01 package main
02
03 import (
04     "fmt"
05     "reflect"
06 )
07
08 func main() {
09     a := 1
10     b := "test"
11     c := true
12     fmt.Println(reflect.TypeOf(a))
13     fmt.Println(reflect.TypeOf(b))
14     fmt.Println(reflect.TypeOf(c))
15 }
```

执行结果如下：

```
int
string
bool
```

TypeOf()方法直接返回传入变量的类型，通过Println()方法打印到控制台。

3.2.7 数据类型转换

Go语言常见的数据类型之间能够互相进行类型转换，通过使用类型前置加小括号的方式进行。

动手写3.2.8

```
01 package main
02 import (
03     "fmt"
04     "reflect"
05 )
06
```

```

07 func main() {
08     var a int16 = 97
09     fmt.Println("变量a值为: ", a, ", 变量类型为: ", reflect.TypeOf(a))
10     b := int32(a)
11     fmt.Println("变量b值为: ", b, ", 变量类型为: ", reflect.TypeOf(b))
12     fmt.Println("转换变量b类型为string: ", string(b))
13 }
```

执行结果如下：

```

变量a值为: 97 , 变量类型为: int16
变量b值为: 97 , 变量类型为: int32
转换变量b类型为string: a
```

在转换变量类型时，我们需要注意变量原本的值是否会发生改变。

动手写3.2.9

```

01 package main
02 import (
03     "fmt"
04     "reflect"
05 )
06
07 func main() {
08     var a int32 = 1234567891
09     fmt.Println("变量a值为: ", a, ", 变量类型为: ", reflect.TypeOf(a))
10     fmt.Println("转换变量a类型为int16, 变量a值变为: ", int16(a), ", 变量a
11     类型变为: ", reflect.TypeOf(int16(a)))
11 }
```

执行结果如下：

```

变量a值为: 1234567891, 变量类型为: int32
转换变量a类型为int16, 变量a值变为: 723, 变量a类型变为: int16
```

由于16位有符号整型的范围为-32768 ~ 32767，而变量a的值1234567891不在这个范围内，导致变量a原本的值发生改变。

1234567891对应的十六进制为0x499602d3，转变为16位变量后，长度缩短一半，丢失了前

(高) 4位十六进制，即变为：0x02d3，其对应的十进制值为723。

3.3 指针

谈及指针，很多人可能马上会联想到C/C++中的指针，指针的存在是C/C++强大的根本所在，但同时也带来很多安全问题，相比之下Go语言的指针则更加高效和安全。

3.3.1 声明指针

指针是一种地址值，这个地址值代表着计算机内存空间中的某个位置。指针变量就是存放地址值的变量，指针变量的声明格式如下：

```
var 变量名 *int
```

一般情况下，我们将指针变量的类型声明为*int，变量名为“p”开头（指代“point”）的单词，如“p”或“ptr”。

动手写3.3.1

```
01 package main
02
03 import "fmt"
04 func main() {
05     var p *int
06     fmt.Println(p)
07 }
```

执行结果如下：

```
<nil>
```

由于指针变量未指向任何地址，所以打印值为nil。

3.3.2 取变量地址

Go语言中，使用操作符“&”取变量地址，取得的地址值可以赋给指针变量。

动手写3.3.2

```
01 package main
02
03 import "fmt"
```

```

04 func main() {
05     num :=1
06     var p *int
07     p = &num
08     fmt.Println("num变量的地址为:", p)
09     fmt.Println("指针变量p的地址为:", &p)
10 }
```

执行结果如下：

```

num变量的地址为: 0xc042066080
指针变量p的地址为: 0xc042086018
```

由于指针变量本身也是变量，因此指针变量在计算机内存中也有自己的地址。需注意的是，读者运行上述程序时得到的实际结果可能与以上结果不符，甚至多次运行该程序得到的结果可能都不一致，这是由于变量在内存中的位置都是随机分配的。

3.3.3 获取指针所指向的内容

指针变量存储的值为地址值，通过在指针变量前面加上“*”符号可以获取指针所指向地址值的内容。

动手写3.3.3

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     num :=1
07     var p *int
08     p = &num
09     fmt.Println("num变量的地址为:", p)
10     fmt.Println("指针变量p的地址为:", &p)
11     fmt.Println("指针变量p所指向的内容:", *p)
12 }
```

执行结果如下：

```

num变量的地址为: 0xc042066080
```

注意：p指针声明后其值为nil，这时如果获取指针p指向的地址内容，则会出错，如动手写3.3.4所示。

动手写3.3.4

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var p *int
07     fmt.Println("指针变量p指向的地址为:", p)
08     fmt.Println("指针变量p所指向的内容:", *p)
09 }
```

执行结果如下：

```
指针变量p指向的地址为: <nil>
panic: runtime error: invalid memory address or nil pointer dereference
```

3.3.4 使用指针修改值

在指针变量有实际指向的地址值后，可以通过如下格式直接修改指针所指向内存地址的内容：

```
*变量名 =修改值
```

动手写3.3.5

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     num :=1
07     var p *int
08     p = &num
09     fmt.Println("指针变量p所指向的内容:", *p)
10     *p = 10
11     fmt.Println("指针变量p所指向的内容:", *p)
12 }
```

执行结果如下：

```
指针变量p所指向的内容： 1
指针变量p所指向的内容： 10
```

对于动手写3.3.5中的问题，在使用指针修改值时也需注意，可使用new()函数来给指针分配地址并初始化地址对应的值。

动手写3.3.6

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var p *int
07     p = new(int)
08     fmt.Println("指针变量p所指向的内容:", *p)
09     *p = 10
10     fmt.Println("指针变量p所指向的内容:", *p)
11 }
```

执行结果如下：

```
指针变量p所指向的内容： 0
指针变量p所指向的内容： 10
```

3.4 其他数据类型

Go语言中有丰富的数据类型，本章主要介绍基本的数据类型，如整型、浮点型、字符串、字符和布尔型。除了以上类型，Go语言还有切片、通道、map（映射）和函数等类型。

数组、切片和map类型是开发中常用的数据类型，将在第6章进行讲解。

函数是Go语言常见的数据类型，将在第7章进行讲解。

结构体作为Go语言中最为复杂难懂的数据类型，将在第9章进行讲解。

接口作为Go语言的高级特性，将在第12章进行讲解。

通道和Go语言并发操作有关，将在第13章进行讲解。

3.5 小结

- ◇ 变量是用来存储用户数据的，Go语言的每个变量都有自己的类型，在使用它们之前，必须先经过声明。
- ◇ 短变量声明并初始化方法在开发中使用最为普遍。
- ◇ 在赋值给多个变量时，如果存在不需要接收值的变量，可以使用匿名变量来代替。
- ◇ 整型主要分为有符号整型和无符号整型两大类。
- ◇ Go语言常见的数据类型之间能够互相进行类型转换，通过使用类型前置加小括号的方式进行。

3.6 知识拓展

3.6.1 字符串的应用

Go语言中，经常会对字符串进行各类操作，如两个字符串的拼接、字符串截取和修改字符串内容等。对于这些操作，目前都有现成的解决方案。

1. 字符串拼接

和其他编程语言类似，Go语言的字符串拼接操作可以通过“+”操作符来完成，代码如下：

动手写3.6.1

```
01 package main
02
03 import "fmt"
04
05 func main(){
06     a := "012345"
07     b := "6789"
08     c := a+b
09     fmt.Println(c)
10 }
```

执行结果如下：

```
0123456789
```

然而，当需要拼接的字符串较长时，使用“+”操作符进行字符串的拼接并不高效，因此推荐使用字节缓冲的方式来进行。

动手写3.6.2

```

01 package main
02
03 import (
04     "bytes"
05     "fmt"
06     "reflect"
07 )
08
09 func main() {
10     a := "012345"
11     b := "6789"
12     var c bytes.Buffer           //声明变量c，类型为字节缓冲
13     c.WriteString(a)            //写入字符串变量a内容
14     c.WriteString(b)            //写入字符串变量b内容
15     fmt.Println(c.String())
16     fmt.Println(reflect.TypeOf(c))
17 }
```

执行结果如下：

```
0123456789
bytes.Buffer
```

其中bytes包中的Buffer数据结构如下，其本质是字节数组，字符串本质上也是一种字节数组，可以通过WriteString()方法来写入。

```

type Buffer struct {
    buf        []byte
    off        int
    bootstrap [64]byte
    lastRead   readOp
}
```

2. 字符串截取

字符串的截取是开发中经常遇到的场景，开发者需要截取目标字符串的某一段内容。Go语言中的strings包的Index()方法会从头对字符串进行搜索，获得搜索内容所属下标，完成对字符串的截取。

动手写3.6.3

```
01 package main
02
03 import (
04     "strings"
05     "fmt"
06 )
07
08 func main() {
09     str := "Go语言"
10     index := strings.Index(str, "语")
11     fmt.Println(index)
12     fmt.Println(str[index:])
13 }
```

执行结果如下：

```
2
语言
```

另外，strings包中还提供了LastIndex()方法来反向搜索字符串。例如，我们只想截取字符串“Go语言,Python语言”中最后的“语言”，就可以使用LastIndex()方法。

动手写3.6.4

```
01 package main
02
03 import (
04     "strings"
05     "fmt"
06 )
07
08 func main() {
```

```

09     str := "Go语言,Python语言"
10     index := strings.LastIndex(str, "语")
11     fmt.Println(index)
12     fmt.Println(str[index:])
13 }
```

执行结果如下：

```

15
语言
```

3. 修改字符串

Go语言无法对字符串直接进行修改，只能将字符串转换为字节数组后再进行操作。

动手写3.6.5

```

01 package main
02
03 import (
04     "fmt"
05 )
06
07 func main() {
08     str := "Go语言"
09     bytes := []byte(str)
10     for i := 0; i < 2; i++ {
11         bytes[i] = ' '
12     }
13     fmt.Println(string(bytes))
14 }
```

执行结果如下：

```
语言
```

该程序代码涉及[]byte和for循环语句，此处仅作为演示，相关知识点将在后续章节详细讲解。

4. 字符串格式化输出

对字符串的格式化使用fmt包的Sprintf()方法，Sprintf()方法的调用格式如下：

```
Sprintf(format,arg1,arg2...)
```

第一个参数为需要格式化的字符串，其中包含格式化动词，格式化动词以“%”开头。调用样例如下：

动手写3.6.6

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var day =1
07     var hour = 24
08     str := fmt.Sprintf("%d天包含%d个小时",day,hour)
09     fmt.Println(str)
10 }
```

执行结果如下：

```
1天包含24个小时
```

程序中需要格式化的字符串为“%d天包含%d个小时”，其中两个“%d”代表整型以十进制方式显示，传入的实参day和hour从左至右与需要格式化字符串的两个动词相对应。

对字符串的格式化常用动词及其功能的对照如表3.6.1所示。

表3.6.1 常用格式化动词及其功能对照

格式化动词	动词功能
%v	按值的本来值输出
%+v	在 %v 基础上，对结构体字段名和值进行展开
%#v	输出 Go 语言语法格式的值
%T	输出 Go 语言语法格式的类型和值
%%	输出 % 本体
%b	整型以二进制方式显示
%o	整型以八进制方式显示
%d	整型以十进制方式显示
%x	整型以十六进制方式显示
%X	整型以十六进制、字母大写方式显示
%U	Unicode 字符
%f	浮点数
%p	指针，以十六进制方式显示

3.6.2 Go语言注释

当程序变得越来越大，代码变得越来越复杂时，阅读也会变得越来越困难。程序的各个代码段之间紧密衔接，使了解整个程序的功能很困难。在日常开发中，开发人员经常会碰到一时半会儿难以弄清楚一段代码在做什么、为什么这么做的情况。因此，在程序中加入自然语言来介绍或者描述代码段是相当重要的。这就类似于给代码做笔记，这种笔记称为“注释（comments）”。Go语言中的注释必须以“//”开头。

注释可以单独一行，也可以放在句尾：

```
01 package main
02
03 import (
04     "bytes"
05     "fmt"
06     "reflect"
07 )
08
09 func main() {
10     a := "012345"
11     b := "6789"
12     var c bytes.Buffer           //声明变量c，类型为字节缓冲
13     c.WriteString(a)            //写入字符串变量a内容
14     c.WriteString(b)            //写入字符串变量b内容
15     fmt.Println(c.String())
16     fmt.Println(reflect.TypeOf(c))
17 }
```

一行代码从“//”符号开始到行末都会被编译器忽略，这部分不会对程序产生任何影响。注释的目的是为了让自己和别人更快地了解程序的功能。注释一般放在代码的上方或者尾部，对一段代码或一个方法进行文字说明，用来帮助我们理解这段代码的含义。注释不要放在代码块的下方，因为这不利于阅读者的理解。

常量与运算符



各类编程语言中，常量与运算符是基本概念，对于Go语言，可以使用它们进行赋值、计算和比较。Go语言支持的运算包括算术运算、比较运算和逻辑运算等。

4.1 常量

所谓常量，就是值不能变的量，比如常用的数学常数“π”就是一个常量。大多数的编程语言会使用全大写的变量名表示常量，所以约定俗成，如果常量的名字是全大写的变量，一般不做修改。

4.1.1 常量的定义

1. 显式定义

常量的声明以关键字const开头，后接变量类型并进行赋值，行尾没有其他标点符号。

常量的显式声明格式如下：

```
const 常量名 常量类型 = value
```

注意：一个常量被声明后可以不使用，但是变量一旦声明则必须使用。

动手写4.1.1

```
01 package main
02
03 import "fmt"
04
05 func main() {
```

```

06     const a int = 3.1415
07     const b char = "Hello world"          //定义常量b但是不使用
08     fmt.Println(a)
09 }
```

执行结果如下：

```
3.1415
```

2. 隐式定义

由于Go是编译型语言，定义常量时可以省略常量类型，因为编译器可以根据变量的值来推断其类型。

常量的隐式声明格式如下：

```
const 常量名 = value
```

动手写4.1.2

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     const a = 3.1415
07     const b = "Hello world"
08     fmt.Println(a)
09 }
```

执行结果如下：

```
3.1415
```

4.1.2 常量组的定义

如果程序需要用到大量值相同的常量，我们可以直接定义常量组。

在定义常量组时，若不提供初始值，则表示将使用上行的表达式。

动手写4.1.3

```

01 package main
02
03 import "fmt"
04
```

```
05 const (
06     a = 3.14
07     b
08     c
09     d = 100
10 )
11
12 func main() {
13     fmt.Println("a:", a)
14     fmt.Println("b:", b)
15     fmt.Println("c:", c)
16     fmt.Println("d:", d)
17 }
```

执行结果如下：

```
a: 3.14
b: 3.14
c: 3.14
d: 100
```

4.1.3 常量枚举

枚举通常针对一个有穷序列集而言，一个集的枚举就是列出有穷序列集的所有成员。

Go语言现阶段还没有枚举，但是我们可以使用一种特殊常量“iota”来模拟枚举。

iota在const关键字出现时被重置为0，const中每新增一行常量声明，将使iota进行一次计数，我们可以简单地将iota理解为const语句块中的行索引。

动手写4.1.4

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     const (
07         a = iota //0
```

```

08         b          //沿用上一行iota, 此时iota=1, b=1
09         c = "Hello world" //iota += 1, iota=2
10         d          //沿用上一行的"Hello world", d="Hello
11             world", iota += 1, iota=3
12     )
13     fmt.Println("a:", a)
14     fmt.Println("b:", b)
15     fmt.Println("c:", c)
16     fmt.Println("d:", d)
17     fmt.Println("e:", e)
18 }

```

执行结果如下：

```

a: 0
b: 1
c: Hello world
d: Hello world
e: 4

```

4.2 运算符

Go数据是通过使用操作运算符来进行操作的，与数学运算符类似。操作运算符接受一个或多个参数，并生成一个新的值，本节将详细介绍这些操作运算符。

4.2.1 算术运算符

算术运算符用于Go语言的数学表达式中，它们的作用和在数学中的作用一样。表4.2.1列出了Go语言中所有的算术运算符。

表4.2.1 Go的算术运算符

运算符	说明	示例
+	加法：相加运算符两侧的值	a + b
-	减法：左操作数减去右操作数	a - b
*	乘法：操作符两侧的值相乘	a * b

(续上表)

运算符	说明	示例
/	除法：左操作数除以右操作数	a / b
%	取模：左操作数除以右操作数的余数	a % b
++	自增：操作数加 1	a++
--	自减：操作数减 1	a--

动手写4.2.1

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     a := 1
07     b := 2
08     fmt.Println("a+b=", a+b)
09     fmt.Println("a-b=", a-b)
10     fmt.Println("a*b=", a*b)
11     fmt.Println("a/b=", a/b)
12     fmt.Println("a%b=", a%b)
13     a++
14     fmt.Println("a++后a值为:", a)
15     b--
16     fmt.Println("b--后b值为:", b)
17 }
```

执行结果如下：

```
a+b= 3
a-b= -1
a*b= 2
a/b= 0
a%b= 1
a++后a值为: 2
b--后b值为: 1
```

4.2.2 比较运算符

比较运算符，对符号两边的变量进行比较，包括大小、相等等。如果比较结果是正确，返回真（true），否则返回假（false）。

表4.2.2 Go的比较运算符

运算符	说明	示例
<code>==</code>	等于：比较对象是否相等	<code>a == b</code>
<code>!=</code>	不等于：比较两个对象是否不相等	<code>a != b</code>
<code>></code>	大于：返回 a 是否大于 b	<code>a > b</code>
<code><</code>	小于：返回 a 是否小于 b	<code>a < b</code>
<code>>=</code>	大于等于：返回 x 是否大于等于 y	<code>x >= y</code>
<code><=</code>	小于等于：返回 x 是否小于等于 y	<code>x <= y</code>

动手写4.2.2

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     a := 1
07     b := 2
08     fmt.Println("a==b:", a==b)
09     fmt.Println("a!=b:", a!=b)
10     fmt.Println("a>b:", a>b)
11     fmt.Println("a<b:", a<b)
12     fmt.Println("a>=b:", a>=b)
13     fmt.Println("a<=b:", a<=b)
14 }
```

执行结果如下：

```

a==b: false
a!=b: true
a>b: false
a<b: true
a>=b: false
a<=b: true
```

4.2.3 赋值运算符

最常用的赋值运算符是等号“=”，表示把右边的结果值赋值给左边的变量。其他的赋值运算符大多都是算术运算符和赋值运算符的简写。

表4.2.3 Go的赋值运算符

运算符	说明	示例	展开形式
=	将右边值赋值给左边	a=100	a=100
+=	将左边值加右边值	a+=10	a=a+10
-=	将左边值减右边值	a-=10	a=a-10
=	将左边值乘以右边值	a=10	a=a*10
/=	将左边值除以右边值	a/=10	a=a/10
%=	将左边值对右边值做取模	a%-=10	a=a%10

动手写4.2.3

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var a = 10
07     fmt.Println("a=", a)
08     a += 2
09     fmt.Println("a += 2, a=", a)
10     a -= 2
11     fmt.Println("a -= 2, a=", a)
12     a *= 2
13     fmt.Println("a *= 2, a=", a)
14     a /= 2
15     fmt.Println("a /= 2, a=", a)
16     a %= 2
17     fmt.Println("a %= 2, a=", a)
18 }
```

执行结果如下：

```
a= 10
a += 2,a= 12
a -= 2,a= 10
a *= 2,a= 20
a /= 2,a= 10
a %= 2,a= 0
```

4.2.4 位运算符

Go定义了位运算符，应用在两个数的运算上，会对数字的二进制所有位数从低到高进行运算。

表4.2.4 Go的位运算符

运算符	说明	示例
&	按位与，如果相对应位都是1，则结果为1，否则为0	a & b
	按位或，如果相对应位都是0，则结果为0，否则为1	a b
^	按位异或，如果相对应位值相同，则结果为0，否则为1	a ^ b
<<	按位左移运算符，左操作数按位左移右操作数指定的位数	a << b
>>	按位右移运算符，左操作数按位右移右操作数指定的位数	a >> b

动手写4.2.4

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     a := 9
07     b := 13
08     fmt.Println("a&b = ", a&b)
09     fmt.Println("a|b = ", a|b)
10     fmt.Println("a^b = ", a^b)
11     fmt.Println("a<<2 = ", a<<2)
12     fmt.Println("b>>2 = ", b>>2)
13 }
```

执行结果如下：

```
a&b = 9
a|b = 13
a^b = 4
a<<2 = 36
b>>2 = 3
```

以上各个运算说明如下：

- 按位与：将参与运算的两个数按二进制位展开后进行与运算，9按二进制位展开为1001，13按二进制位展开为1101，1001和1101的与运算结果为1001，即十进制9。
- 按位或：将参与运算的两个数按二进制位展开后进行或运算，9按二进制位展开为1001，13按二进制位展开为1101，1001和1101的或运算结果为1101，即十进制13。
- 按位异或：将参与运算的两个数按二进制位展开后进行异或运算，9按二进制位展开为1001，13按二进制位展开为1101，1001和1101的异或运算结果为0100，即十进制4。
- 按位左移：将参与运算的数按二进制位展开后全部左移指定位数，9按二进制位展开为1001，全部左移2位后变为100100，即十进制36。
- 按位右移：将参与运算的数按二进制位展开后全部右移指定位数，13按二进制位展开为1101，全部右移2位后变为11，即十进制3。

4.2.5 逻辑运算符

逻辑运算符在Go语言中非常重要，广泛应用在逻辑判断的地方。

表4.2.5 Go的逻辑运算符

运算符	说明	示例
&&	逻辑与，当且仅当两个操作数都为真，条件才为真	a && b
	逻辑或，两个操作数中任意一个为真，条件为真	a b
!	逻辑非，用来反转操作数的逻辑状态。如果条件为 true，则逻辑非运算符将得到 false	! a

动手写4.2.5

```
01 package main
02
03 import "fmt"
04
```

```

05 func main() {
06     a := true
07     b := false
08     fmt.Println("a&&b = ", a&&b)
09     fmt.Println("a||b = ", a||b)
10     fmt.Println("!a = ", !a)
11 }
```

执行结果如下：

```

a&&b =  false
a||b =  true
!a =  false
```

4.2.6 其他运算符

Go语言中有指针这一概念，与C/C++中的指针类似。由于这一部分的理解对于新手而言有一定难度，关于指针的相关知识和具体实例可查看结构体与指针一章（第9章）。

另外，需要注意的是按位取反运算符“`~`”，它和按位异或运算符不同，涉及的操作数只有一个，因此也可归类为单目运算符。

表4.2.6 Go的其他运算符

运算符	说明	示例
<code>&</code>	返回变量存储地址	<code>&a</code>
<code>*</code>	指针变量	<code>*a</code>
<code>~</code>	按位取反（注意与按位异或区分）	<code>~a</code>

动手写4.2.6

```

01 package main
02
03 import (
04     "fmt"
05 )
06
07 func main() {
08     a := 1
```

```
09     var p *int          //声明指针p，指针类型为int类型
10     p = &a              //指针p取变量a的地址
11     fmt.Println("^a = ", ^a)
12     fmt.Println("变量a的地址为:", p)
13 }
```

执行结果如下：

```
^a = -2
变量a的地址为：0xc042066080
```

正数1按二进制位展开为001，其中最左位0表示正数，取反操作对所有二进制位取反，结果为110，其中最左位1表示负数，即十进制-2。

4.3 小结

- ◇ 所谓常量，就是值不能变的量。常量的定义可通过显式和隐式两种方法来实现。
- ◇ 可通过常量组来定义多个常量，并通过特殊常量*iota*来模拟对常量的枚举。
- ◇ Go语言运算符主要包括算术运算符、比较运算符、赋值运算符、位运算符和逻辑运算符等，运算符之间有优先级。

4.4 知识拓展

运算符优先级

运算符有不同的优先级，对于二元运算符，运算方向均是从左到右。表4.4.1列出了所有运算符以及它们的优先级，表由上至下代表优先级由高到低。（表中第一行运算符“`^`”代表按位取反运算符，第八行运算符“`~`”代表按位异或运算符。）

表4.4.1 运算符优先级

运算符	关联性
<code>!、^、++、--</code>	从右到左
<code>*（乘）、/、%</code>	从左到右
<code>+（加）、-（减）</code>	从左到右

(续上表)

运算符	关联性
<<、 >>	从左到右
<、 <=、 >、 >=	从左到右
==、 !=	从左到右
& (按位与)	从左到右
^	从左到右
	从左到右
&&	从左到右
	从左到右
=、 +=、 -=、 *=、 /=、 %=、 >>=、 <<=、 &=、 ^=、 =	从右到左

>> 第 5 章

流程控制 <<

与其他编程语言一样，使用Go语言的编程者需要通过流程控制语句来控制程序的逻辑走向和执行次序。

Go语言基本上继承了C/C++语言所有流程控制语句，如果读者有C/C++的编程经验，那么理解本章内容将较为轻松。

流程控制语句主要包括：条件判断语句（if和switch）、循环控制语句（for、break和continue）和跳转语句（goto）。

5.1 if判断

Go语言中，通过if关键字构成的条件判断语句进行条件判断，格式如下：

```
if 表达式1 {  
    分支1  
} else if 表达式2 {  
    分支2  
} else {  
    分支3  
}
```

当表达式1的执行结果为true时，执行分支1，否则对表达式2的执行结果进行判断；若表达式2的结果为true，执行分支2；如果都不满足，则执行分支3。

当然，可以只存在表达式1，else if和else语句都是可选的。

注意：表达式后跟的左括号必须与表达式放在同一行中，否则程序在编译时将会触发错误，导致程序编译无法通过。

同理，else if和else语句后跟的左括号也必须和对应表达式处在同一行。另外，if、else if和else分支中对应的右括号可以另外换行，也可以与对应的左括号处在同一行。

动手写5.1.1

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     a := 101
07     if a > 100 {
08         fmt.Println(a, " > 100")
09     } else if a == 100 {
10         fmt.Println(a, " = 100")
11     } else {
12         fmt.Println(a, " < 100")
13     }
14 }
```

执行结果如下：

```
101 > 100
```

if还有一种较为常见的写法，就是在if的表达式前添加一个语句，使用变量接收语句返回的值，通过对该变量的判断再选择执行的分支。

动手写5.1.2

```

01 package main
02
03 import (
04     "runtime"
05     "fmt"
06 )
07
08 func main() {
```

```
09     if num := runtime.NumCPU(); num >= 1 {
10         fmt.Println("程序使用的CPU核数为: ", num)
11     }
12 }
```

以上程序引入了runtime包，使用NumCPU()函数获取了程序使用的CPU核数并传递给num变量。

执行结果如下：

```
程序使用的CPU核数为: 8
```

注意：由于是在if表达式前定义num变量，num变量的作用域仅限于该分支中，程序的执行结果与宿主机的配置有关。关于函数的详细内容可参照函数一章（第7章）。

5.2 循环控制

5.2.1 for循环

Go语言中的循环逻辑通过for关键字实现。不同于其他编程语言，Go语言没有while关键字，不存在while循环。

for循环格式如下：

```
for 初始语句; 条件表达式; 赋值表达式 {
    循环体
}
```

循环体中代码会不断地被执行，直到条件表达式的结果为false，程序才会继续执行for循环之后的程序代码。

其中，初始语句、条件表达式和赋值表达式都是可选的。

动手写5.2.1

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     for i := 1; i <= 5; i++ {
07         fmt.Println(i)
08     }
09 }
```

执行结果如下：

```
1  
2  
3  
4  
5
```

5.2.2 break跳出循环

break语句可以用来结束for循环，而且可以在语句后面添加标签，表示退出标签对应的代码块逻辑。

注意：break语句如果不带标签，则默认跳出最内层的for循环。

我们可以对动手写5.2.1中的程序进行更改：

动手写5.2.2

```
01 package main  
02  
03 import "fmt"  
04  
05 func main() {  
06     i := 1  
07     for {  
08         for {  
09             if i > 5 {  
10                 fmt.Println("跳出内层for循环")  
11                 break  
12             }  
13             fmt.Println(i)  
14             i++  
15         }  
16         fmt.Println("跳出外层for循环")  
17         break  
18     }  
19 }
```

执行结果如下：

```
1
2
3
4
5
```

跳出内层for循环

跳出外层for循环

我们也可以使用带标签的break语句，直接跳出最外层的for循环：

动手写5.2.3

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     i := 1
07     OuterLoop:
08     for {
09         for {
10             if i > 5 {
11                 break OuterLoop //跳出OuterLoop标签对应的循环
12             }
13             fmt.Println(i)
14             i++
15         }
16     }
17 }
```

执行结果如下：

```
1
2
3
4
5
```

5.2.3 continue继续循环

continue语句可以立即结束当前循环体中的逻辑，开始下一次循环。和break语句类似，continue语句后也可跟标签，表示开始标签所对应的循环。

动手写5.2.4

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     OuterLoop:
07     for i := 0; i < 2; i++ {
08         for j := 0; j < 3; j++ {
09             if j == 1 {
10                 fmt.Println(i, j)
11                 continue OuterLoop
12             }
13         }
14     }
15 }
```

执行结果如下：

```

0 1
1 1
```

5.3 switch分支

switch语句常用于基于大量不同条件来执行不同动作，每一个条件对应一个case分支。

switch语句的执行过程从上至下，直到找到匹配项，匹配项后面也不需要再加break。

每一个switch语句只能包含一个可选的default分支，若没有找到匹配项，会默认执行default分支中的代码块。

Go语言中的switch语法如下：

```
switch var1 {  
    case value1:  
        代码块1  
    case value2:  
        代码块2  
    default:  
        代码块3  
}
```

变量var1可以是任何类型，但value1和value2必须是相同的类型或最终结果为相同类型的表达式。每个case分支后可跟多个可能符合条件的值，使用逗号分隔它们，例如：case value1,value2,value3。

动手写5.3.1

```
01 package main  
02  
03 import "fmt"  
04  
05 func main() {  
06     switch 1+1 {  
07         case 1:  
08             fmt.Println("1+1=1")  
09         case 2:  
10             fmt.Println("1+1=2")  
11         case 3:  
12             fmt.Println("1+1=3")  
13     default:  
14         fmt.Println("1+1不等于1或2或3")  
15     }  
16 }
```

执行结果如下：

```
1+1=2
```

默认情况下，switch匹配成功后就不会执行后续其他case，如果我们需要无条件强制执行后面的 case，可以使用fallthrough关键字。

动手写5.3.2

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     switch {
07         //false, 肯定不会执行
08         case false:
09             fmt.Println("case 1为false")
10             fallthrough
11         //true, 肯定执行
12         case true:
13             fmt.Println("case 2为 true")
14             fallthrough
15         //由于上一个case中有fallthrough, 即使是false, 也强制执行
16         case false:
17             fmt.Println("case 3为 false")
18             fallthrough
19         default:
20             fmt.Println("默认 case")
21     }
22 }
```

执行结果如下：

```

case 2为 true
case 3为 false
默认 case
```

上述示例中，switch语句省略了条件表达式，表达式由下面的case给出。

5.4 goto跳转

goto语句用于代码间的无条件跳转，格式如下：

```
goto 标签
```

动手写5.4.1

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     fmt.Println("Hello")
07     goto sign
08     fmt.Println("无效代码")
09     sign:
10     fmt.Println("world")
11 }
```

执行结果如下：

```
Hello
world
```

一般情况下，在程序中不建议使用goto语句，过多的goto语句会破坏程序结构，使程序的可读性变差。

5.5 小结

- ◇ Go语言的流程控制语句主要包括：条件判断语句（if和switch）、循环控制语句（for、break和continue）和跳转语句（goto）。
- ◇ Go语言中的循环逻辑通过for关键字实现，不存在while循环。
- ◇ switch语句常用于基于大量不同条件来执行不同动作，每一个条件对应一个case分支。
- ◇ 一般情况下，在程序中不建议使用goto语句，过多的goto语句会破坏程序结构，使程序的可读性变差。

5.6 知识拓展

5.6.1 斐波那契数列

斐波那契数列，又称黄金分割数列，因数学家列昂纳多·斐波那契（Leonardoda Fibonacci）

以兔子繁殖为例子而引入，故又称为兔子数列，指的是这样一个数列：1、1、2、3、5、8、13、21、34……这个数列从第三项开始，每一项都等于前两项之和。斐波那契数列在现代物理、准晶体结构、化学等领域都有直接的应用。

那么，我们是否可以编写一个斐波那契数列的程序，输入第n项，程序输出对应第n项的值？答案是肯定的，最常见的实现方式就是循环和递归。

在Go语言中，使用for循环实现斐波那契数列的程序如下：

动手写5.6.1

```

01 package main
02
03 import "fmt"
04
05 func fibonacci(n int) (res int) {
06     a:=1
07     b:=1
08     for i:=2;i<n;i++{
09         c := b
10         b = a+b
11         a = c
12     }
13     return b
14 }
15
16 func main(){
17     n:=9      //求斐波那契数列第9项的值
18     fmt.Printf("斐波那契数列第%d项值为%d",n,fibonacci(n))
19 }
```

执行结果如下：

斐波那契数列第9项值为34

对于斐波那契数列，我们也可以使用递归来实现：

动手写5.6.2

```

01 package main
02
03 import "fmt"
04
```

```
05 func fibonacci(n int) (res int) {
06     if n==1 || n==2{
07         return 1
08     }else{
09         return fibonacci(n-2)+fibonacci(n-1)
10     }
11 }
12
13 func main(){
14     n:=5      //求斐波那契数列第5项的值
15     fmt.Printf("斐波那契数列第%d项值为%d",n,fibonacci(n))
16 }
```

执行结果如下：

```
斐波那契数列第5项值为5
```

在使用递归方法来实现斐波那契数列时，我们也可以用switch分支来进行代替：

动手写5.6.3

```
01 package main
02
03 import "fmt"
04
05 func fibonacci(n int) (res int) {
06     switch n {
07     case 1:
08         return 1
09     case 2:
10         return 1
11     default:
12         return fibonacci(n-2) + fibonacci(n-1)
13     }
14 }
15
16 func main(){
17     n:=6      //求斐波那契数列第6项的值
18     fmt.Printf("斐波那契数列第%d项值为%d",n,fibonacci(n))
19 }
```

执行结果如下：

```
斐波那契数列第6项值为8
```

5.6.2 循环嵌套

在5.2.1小节中，我们学习了简单的for循环。当然，一个循环结构内可以含有另一个循环，这被称为循环嵌套，又称多重循环。常用的循环嵌套是二重循环，外层循环称为外循环，内层循环称为内循环。

双重循环的结构：

```
for (初始语句; 条件表达式; 赋值表达式)
{
    循环体
    for (初始语句; 条件表达式; 赋值表达式)
    {
        循环体
    }
}
```

上面的结构共由两个for循环组成，它们之间的层次关系是一个嵌套住另一个，我们把这种关系叫作嵌套关系。这种层次关系是唯一且不可改变的。

下面我们通过数字金字塔这一实例来进行循环嵌套的实践。

动手写5.6.4

```
01 package main
02
03 import "fmt"
04
05 func pyramid(n int) {
06     //n表示总层数
07     for i := 1; i <= n; i++ {
08         //打印*前先打印空格，规律为总层数-当前层数
09         for j := 1; j <= n-i; j++ {
10             fmt.Print(" ")
11         }
12         //k表示每层打印多少*，规律为 2 * i - 1
13         for k := 1; k <= 2 * i - 1; k++ {
```

```
14             fmt.Println("*")
15         }
16     fmt.Println()
17 }
18 }
19
20 func main() {
21     n := 9
22     pyramid(n)
23 }
```

执行结果如下：

```
*  
***  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

变量和常量虽能存储数据，但是在编写一些逻辑稍复杂的程序中，往往需要存储更多、更复杂且不同类型的数据，这些数据一般存储在Go语言的内置容器中。

Go语言的内置容器主要有数组、切片和映射。

本章将详细介绍以上三种内置容器的特点和使用方法，从而让读者们在编程中能使用恰当的容器存储数据并对其进行增加、删除和修改等操作。

6.1 数组

数组是具有相同类型且长度固定的一组数据项序列，这组数据项序列对应存放在内存中的一块连续区域中。

数组中存放的元素类型可以是整型、字符串或其他自定义类型。数组在使用前需先声明，声明时必须指定数组的大小且数组大小之后不可再变。

数组元素可以通过数组下标来读取或修改，数组下标从0开始，第一个元素的数组下标为0，第二个元素的数组下标为1，以此类推。

如图6.1.1所示， a_1 代表数组下标为0的元素， a_2 代表数组下标为1的元素……

a_1	a_2	a_3	a_4	a_5	a_n
0	1	2	3	4		$n-1$

图6.1.1 数组元素与数组下标的对应关系

6.1.1 声明数组

数组声明格式如下：

```
var 数组变量名 [数组长度]元素类型
```

例如，声明数组student，长度为3，元素类型为string：

```
var student [3]string
```

动手写6.1.1

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student [3]string
07     fmt.Println(student)          //直接打印整个数组
08 }
```

执行结果如下：

```
[ ]
```

由于以上代码仅声明了数组，没有对数组进行赋值，因此打印出来的数组为空数组。

6.1.2 初始化数组

数组可在声明时进行赋值，样例如下：

```
var student = [3]string{"Tom", "Ben", "Peter"}
```

使用这种方式初始化数组，需要保证大括号里面的元素数量和数组大小一致。

如果忽略中括号内的数字，不设置数组大小，Go语言编译器在编译时也可根据元素的个数来设置数组的大小，通过用“...”代替数组大小来实现。样例如下：

```
var student = [...]string{"Tom", "Ben", "Peter"}
```

动手写6.1.2

```
01 package main
02
03 import "fmt"
```

```

04
05 func main() {
06     var student = [...]string{"Tom", "Ben", "Peter"}
07     fmt.Println(student)
08 }
```

执行结果如下：

```
[Tom Ben Peter]
```

6.1.3 range关键字

range是Go语言中非常常用的一个关键字，其主要作用就是配合for关键字对数组以及之后会介绍到的切片和映射等数据结构进行迭代。

动手写6.1.3

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     var num = [...]int{1, 2, 3, 4}
07     for k, v := range num {
08         fmt.Println("变量k: ", k, " ", "变量v: ", v)
09     }
10 }
```

执行结果如下：

```
变量k: 0      变量v: 1
变量k: 1      变量v: 2
变量k: 2      变量v: 3
变量k: 3      变量v: 4
```

range后接的表达式称为range表达式，本例的range表达式为数组。在迭代时，关键字range会返回两个值，分别由变量k和v接收。其中k是当前循环迭代到的索引位置，v是该位置对应元素值的一份副本。

其他range表达式及其对应的返回值如表6.1.1所示。

表6.1.1 range表达式及对应返回值

range 表达式	第一返回值	第二返回值
数组	元素下标	元素值
切片	元素下标	元素值
映射	键	值
通道	元素	N/A

6.1.4 遍历数组

数组元素可以通过数组下标来读取或修改，数组下标从0开始，第一个元素的数组下标为0，第二个元素的数组下标为1，以此类推。

我们现在可以通过遍历数组的方式（for循环）来对其进行打印。

动手写6.1.4

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student = [...]string{"Tom", "Ben", "Peter"}
07     for k, v := range student{
08         fmt.Println("数组下标: ", k, ", 对应元素", v)
09     }
10 }
```

执行结果如下：

```
数组下标: 0 , 对应元素 Tom
数组下标: 1 , 对应元素 Ben
数组下标: 2 , 对应元素 Peter
```

从打印结果可以看到，Go语言可以通过循环十分方便地获取数组的下标和对应元素。

6.2 切片

相对于数组，切片（slice）是一种更方便和强大的数据结构，它同样表示多个同类型元素的连续集合，但是切片本身并不存储任何元素，而只是对现有数组的引用。

切片结构包括：地址、长度和容量。

◇ 地址：切片的地址一般指切片中第一个元素所指向的内存地址，用十六进制表示。

◇ 长度：切片中实际存在元素的个数。

◇ 容量：从切片的起始元素开始到其底层数组中的最后一个元素的个数。

切片的长度和容量都是不固定的，可以通过追加元素使切片的长度和容量增大。

切片主要有三种生成方式：

1. 从数组生成一个新的切片；

2. 从切片生成一个新的切片；

3. 直接生成一个新的切片。

6.2.1 从数组生成一个新的切片

从数组或切片生成新的切片语法格式如下：

```
slice [开始位置:结束位置]
```

对于动手写6.1.2中长度为3的student数组，我们可以生成一个新的切片student1，使用len()函数可获得当前切片长度，cap()函数可获得当前切片容量。

动手写6.2.1

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student = [...]string{"Tom", "Ben", "Peter"}
07     var student1 = student[1:2]
08     fmt.Println("student数组: ", student)
09     fmt.Println("student1切片: ", student1)
10     fmt.Println("student数组地址为", &student[1]) //取student[1]元素的地址
11     fmt.Println("student1切片地址为", &student1[0]) //取student1[0]元素的地址
12     fmt.Println("student1切片长度为: ", len(student1))
13     fmt.Println("student1切片容量为: ", cap(student1))
14 }
```

执行结果如下：

```
student数组: [Tom Ben Peter]
student1切片: [Ben]
student数组地址为 0xc04207a1f0
student1切片地址为 0xc04207a1f0
student1切片长度为: 1
student1切片容量为: 2
```

根据运行结果，我们可以归纳出从数组或切片生成新的切片有如下特性：

- ◇ 新生成的切片长度：结束位置-开始位置。
- ◇ 新生成的切片取出的元素不包括结束位置对应的元素。
- ◇ 新生成的切片是对现有数组或切片的引用，其地址与截取的数组或切片开始位置对应的元素地址相同。
- ◇ 新生成的切片容量指从切片的起始元素开始到其底层数组中的最后一个元素的个数。

6.2.2 从切片生成一个新的切片

我们重新从student数组生成student1切片，再从student1切片生成student2切片。

动手写6.2.2

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student = [...]string{"Tom", "Ben", "Peter"}
07     var student1 = student[1:3]
08     var student2 = student1[0:1]
09     fmt.Println("student数组: ", student[:])
10     fmt.Println("student1切片: ", student1[:])
11     fmt.Println("student2切片: ", student2[:])
12     fmt.Println("student数组地址为", &student[1])
13     fmt.Println("student1切片地址为", &student1[0])
14     fmt.Println("student2切片地址为", &student2[0])
15     fmt.Println("student1切片长度为: ", len(student1))
16     fmt.Println("student1切片容量为: ", cap(student1))
```

```

17     fmt.Println("student2切片长度为: ", len(student2))
18     fmt.Println("student2切片容量为: ", cap(student2))
19
20 }

```

执行结果如下：

```

student数组: [Tom Ben Peter]
student1切片: [Ben Peter]
student2切片: [Ben]
student数组地址为 0xc04207a1f0
student1切片地址为 0xc04207a1f0
student2切片地址为 0xc04207a1f0
student1切片长度为: 2
student1切片容量为: 2
student2切片长度为: 1
student2切片容量为: 2

```

为了将student的最后一个元素也取到，student1切片的结束位置设为了3。

根据结果，我们可以发现student2切片仍然是对底层数组student的引用。

另外，也可以通过slice[:]来表示切片本身。

6.2.3 直接生成一个新的切片

1. 声明切片

切片的声明格式如下：

```
var 切片变量名 []元素类型
```

动手写6.2.3

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     var student []int
07     fmt.Println("student切片: ", student)

```

```
08     fmt.Println("student切片长度: ", len(student))
09     fmt.Println("student切片容量: ", cap(student))
10     fmt.Println("判定student切片是否为空: ", student==nil)
11 }
```

执行结果如下：

```
student切片: []
student切片长度: 0
student切片容量: 0
判定student切片是否为空: true
```

从运行结果可以知道，切片声明后其内容为空，长度和容量均为0。

2. 初始化切片

(1) 在声明的同时初始化

我们可以在声明切片的同时进行初始化赋值，示例如下：

动手写6.2.4

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student = []string{"Tom", "Ben", "Peter"}
07     fmt.Println("student切片: ", student)
08     fmt.Println("student切片长度: ", len(student))
09     fmt.Println("student切片容量: ", cap(student))
10     fmt.Println("判定student切片是否为空: ", student==nil)
11 }
```

执行结果如下：

```
student切片: [Tom Ben Peter]
student切片长度: 3
student切片容量: 3
判定student切片是否为空: false
```

(2) 使用make()函数初始化

声明完切片后，可以通过内建函数make()来初始化切片，格式如下：

```
make([]元素类型, 切片长度, 切片容量)
```

注意：切片的容量值必须大于等于切片长度值，否则程序会报错。对于切片的容量应该有个大概的估值，若容量值过小，对切片的多次扩充会造成性能损耗。

动手写6.2.5

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student []int
07     student = make([]int, 2, 10)
08     fmt.Println("student切片: ", student)
09     fmt.Println("student切片长度: ", len(student))
10     fmt.Println("student切片容量: ", cap(student))
11     fmt.Println("判定student切片是否为空: ", student==nil)
12 }
```

执行结果如下：

```
student切片: [0 0]
student切片长度: 2
student切片容量: 10
判定student切片是否为空: false
```

参照动手写6.2.3，我们可以发现student切片在初始化后，自动填充了0值且不再为空。

6.2.4 为切片添加元素

Go语言中，我们可以使用append()函数来对切片进行元素的添加。当切片不能再容纳其他元素时（即当前切片长度值等于容量值），下一次使用append()函数对切片进行元素添加，容量会按2倍数进行扩充。

动手写6.2.6

```
01 package main
02
03 import "fmt"
```

```
04
05 func main() {
06     student := make([]int, 1, 1)
07     for i:=0;i<8;i++{
08         student = append(student,i)
09         fmt.Println("当前切片长度: ",len(student),"当前切片容
10        量: ",cap(student))
11 }
```

执行结果如下：

```
当前切片长度: 2 当前切片容量: 2
当前切片长度: 3 当前切片容量: 4
当前切片长度: 4 当前切片容量: 4
当前切片长度: 5 当前切片容量: 8
当前切片长度: 6 当前切片容量: 8
当前切片长度: 7 当前切片容量: 8
当前切片长度: 8 当前切片容量: 8
当前切片长度: 9 当前切片容量: 16
```

现在我们参照动手写6.2.1，对程序稍加改动。

动手写6.2.7

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student = [...]string{"Tom", "Ben", "Peter"}
07     var student1 = student[0:1] //从student数组生成切片student1
08     fmt.Println("student数组: ",student)
09     fmt.Println("student1切片: ",student1)
10     student1 = append(student1,"Danny") //对student1切片的元素添加，会
11     覆盖引用数组对应的元素
12     fmt.Println("扩充Danny后的student1切片: ",student1,",切片长度
13     为: ",len(student1),",切片容量为: ",cap(student1))
14     fmt.Println("扩充Danny后的student数组: ",student)
```

执行结果如下：

```
student数组: [Tom Ben Peter]
student1切片: [Tom]
扩充Danny后的student1切片: [Tom Danny] ,切片长度为: 2 ,切片容量为: 3
扩充Danny后的student数组: [Tom Danny Peter]
```

由于student1切片是从student数组生成（即对student数组的引用），为student1添加元素会覆盖student数组中对应的元素。

所以，如果切片是从其他数组或切片生成，新切片的元素添加需要考虑对原有数组或切片中数据的影响。

6.2.5 从切片删除元素

由于Go语言没有为删除切片元素提供方法，所以需要我们手动将删除点前后的元素连接起来，从而实现对切片中元素的删除。

动手写6.2.8

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student = []string{"Tom", "Ben", "Peter", "Danny"}
07     student = append(student[0:1], student[2:]...)
08     fmt.Println("student切片: ", student)
09     fmt.Println("student切片长度: ", len(student))
10     fmt.Println("student切片容量: ", cap(student))
11 }
```

执行结果如下：

```
student切片: [Tom Peter Danny]
student切片长度: 3
student切片容量: 4
```

其中append()函数中传入的省略号代表按student切片展开，该行代码等价于：

```
student = append(student[0:1], student[2], student[3])
```

如果需要清空切片中的所有元素，可以把切片的开始下标和结束下标都设为0来实现：

动手写6.2.9

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student = []string{"Tom", "Ben", "Peter", "Danny"}
07     student = student[0:0]
08     fmt.Println("student切片: ", student)
09     fmt.Println("student切片长度: ", len(student))
10     fmt.Println("student切片容量: ", cap(student))
11 }
```

执行结果如下：

```
student切片: []
student切片长度: 0
student切片容量: 4
```

6.2.6 遍历切片

切片的遍历和数组类似，可以通过切片下标来进行遍历。切片下标同样从0开始，第一个元素的数组下标为0，第二个元素的数组下标为1，以此类推。

动手写6.2.10

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var student = []string{"Tom", "Ben", "Peter", "Danny"}
07     for k, v := range student{
08         fmt.Println("切片下标: ", k, ", 对应元素", v)
09     }
10 }
```

执行结果如下：

```
切片下标: 0 , 对应元素 Tom
切片下标: 1 , 对应元素 Ben
切片下标: 2 , 对应元素 Peter
切片下标: 3 , 对应元素 Danny
```

6.3 映射

映射（map）是一种无序的键值对的集合，map的键类似于索引，指向数据的值。当我们的程序中需要存放有关联关系的数据时，往往就会用到map。

例如，我们可以将国家中文名和其英文名关联起来，生成如下映射：

```
country := map[string]string{
    "中国": "China",
    "美国": "America",
    "日本": "Japan",
}
```

6.3.1 声明映射

map的声明格式如下：

```
var map [键类型]值类型
```

如果我们需要建立一个学生和其成绩的对应关系，样例如下：

动手写6.3.1

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var studentScoreMap map[string]int
07     fmt.Println(studentScoreMap)
08 }
```

执行结果如下：

```
map []
```

6.3.2 初始化映射

1. 在声明的同时初始化

我们可以在声明map的同时对其进行初始化：

动手写6.3.2

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var studentScoreMap = map[string]int{
07         "Tom":80,
08         "Ben":85,
09         "Peter":90,
10     }
11     fmt.Println(studentScoreMap)
12 }
```

执行结果如下：

```
map [Tom:80 Ben:85 Peter:90]
```

2. 使用make()函数初始化

与切片的初始化类似，map也可以使用make()函数来进行初始化，格式如下：

```
make (map [键类型] 值类型, map容量)
```

注意：使用make()函数初始化map时可以不指定map容量，但是对于map的多次扩充会造成性能损耗。

cap()函数只能用于获取切片的容量，无法获得map的容量，因此可以通过len()函数获取map的当前长度。

动手写6.3.3

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var studentScoreMap map[string]int
07     studentScoreMap = make(map[string]int)
08     studentScoreMap["Tom"] = 80
09     studentScoreMap["Ben"] = 85
10     studentScoreMap["Peter"] = 90
11     //fmt.Println(cap(studentScoreMap))
12     fmt.Println("map长度为:", len(studentScoreMap))
13     fmt.Println(studentScoreMap)
14 }
```

执行结果如下：

```
map长度为: 3
map [Tom:80 Ben:85 Peter:90]
```

6.3.3 遍历映射

map的遍历主要通过for循环来完成，遍历时可同时获得map的键和值。

动手写6.3.4

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var studentScoreMap map[string]int
07     studentScoreMap = make(map[string]int)
08     studentScoreMap["Tom"] = 80
09     studentScoreMap["Ben"] = 85
10     studentScoreMap["Peter"] = 90
```

```
11     for k,v := range studentScoreMap{  
12         fmt.Println(k,v)  
13     }  
14 }
```

执行结果如下：

```
Tom 80  
Ben 85  
Peter 90
```

若只想遍历键，for循环可改写成如下形式：

```
for k := range studentScoreMap{  
    fmt.Println(k)  
}
```

执行结果如下：

```
Tom  
Ben  
Peter
```

若只想遍历值，for循环可改写成如下形式：

```
for _,v := range studentScoreMap{  
    fmt.Println(v)  
}
```

执行结果如下：

```
80  
85  
90
```

6.3.4 从映射中删除键值对

Go语言通过delete()函数来对map中的指定键值对进行删除操作，delete()函数格式如下：

```
delete(map, 键)
```

其中map为要删除的map实例，键为map键值对中的键。

动手写6.3.5

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     var studentScoreMap map[string]int
07     studentScoreMap = make(map[string]int)
08     studentScoreMap["Tom"] = 80
09     studentScoreMap["Ben"] = 85
10     studentScoreMap["Peter"] = 90
11     delete(studentScoreMap, "Tom")
12     fmt.Println(studentScoreMap)
13 }
```

执行结果如下：

```
map[Ben:85 Peter:90]
```

注意：`delete()`函数会直接删除指定的键值对，而不是仅仅删除键或值。

另外，Go语言没有为map提供清空所有元素的方法，想要清空map的唯一方法就是重新定义一个新的map。

6.4 小结

- ◇ 数组是具有相同类型且长度固定的一组数据项序列，这组数据项序列对应存放在内存中的一块连续区域中，数组大小之后不可再变。
- ◇ 切片表示多个同类型元素的连续集合，但是切片本身并不存储任何元素，而只是对现有数组的引用。
- ◇ 如果切片是从其他数组或切片生成，新切片的元素添加需要考虑对原有数组或切片中数据的影响。
- ◇ Go语言没有为删除切片元素提供方法，所以需要我们手动将删除点前后的元素连接起来，从而实现对切片中元素的删除。

- ◇ 映射是一种无序的键值对的集合，当程序中需要存放有关联关系的数据时，往往就会用到 map。
- ◇ Go语言没有为map提供清空所有元素的方法，想要清空map的唯一方法就是重新定义一个新的map。

6.5 知识拓展

并发操作map

对于map，我们可以定义一个键和值，然后从map中获取、变更和删除这个值。6.3小节中，对map的操作都是在单协程的情况下完成的，这种情况下一般不会出现错误。如果是多个协程并发访问一个map，就有可能会导致程序异常退出，具体示例程序如下：

动手写6.5.1

```
01 package main
02
03 func main() {
04     GoMap := make(map[int]int)
05     for i := 0; i < 10000; i++ {
06         go writeMap(GoMap, i, i)
07         go readMap(GoMap, i)
08     }
09 }
10
11 func readMap(Gomap map[int]int, key int) int {
12     return Gomap[key]
13 }
14
15 func writeMap(Gomap map[int]int, key int, value int) {
16     Gomap[key] = value
17 }
```

执行结果如下：

```

fatal error: concurrent map writes

goroutine 21 [running]:
runtime.throw(0x468b61, 0x15)
    D:/Go/src/runtime/panic.go:616 +0x88 fp=0xc04205ff50 sp=0xc04205
ff30 pc=0x423a78
runtime.mapassign_fast64(0x459100, 0xc04205a000, 0x2, 0x0)
    D:/Go/src/runtime/hashmap_fast.go:531 +0x2fd fp=0xc04205ff98 sp=
0xc04205ff50 pc=0x40835d
main.writeMap(0xc04205a000, 0x2, 0x2)
    F:/goland/mygo/src/awesomeProject/a.go:16 +0x48 fp=0xc04205ffc8
sp=0xc04205ff98 pc=0x44c008
runtime.goexit()
    D:/Go/src/runtime/asm_amd64.s:2361 +0x1 fp=0xc04205ffd0
sp=0xc04205ffc8 pc=0x446b11
created by main.main

```

从运行结果可以发现，程序异常终止，原因是出现了严重错误：多个协程在尝试对map进行同时写入。

由于map不是协程安全的，同一时刻只能有一个协程对map进行操作。最常见的解决方案就是使用sync包对map加锁或直接使用Go在1.9版本中提供的线程安全map。

加锁的本质其实就是当前协程在对map操作前需先加上锁，加锁后其他任何协程无法对map进行任何操作，直至当前协程解锁。样例如下：

动手写6.5.2

```

01 package main
02
03 import (
04     "sync"
05     "fmt"
06 )
07
08 var lock sync.RWMutex
09 func main() {
10     GoMap := make(map[int]int)
11     for i := 0; i < 100000; i++ {

```

```
12         go writeMap(GoMap, i, i)
13         go readMap(GoMap, i)
14     }
15     fmt.Println("Done")
16 }
17
18 func readMap(Gomap map[int]int, key int) int {
19     lock.Lock()          //读map操作前先加锁
20     m := Gomap[key]
21     lock.Unlock()        //读完map后解锁
22     return m
23 }
24
25 func writeMap(Gomap map[int]int, key int, value int) {
26     lock.Lock()          //写map操作前先加锁
27     Gomap[key] = value
28     lock.Unlock()        //写完map后解锁
29 }
```

执行结果如下：

```
Done
```

由于加锁对程序性能会有一定影响，因此，如果需要在多协程情况下对map进行操作，我们推荐使用Go在1.9版本中提供的一种效率较高的并发安全的map——sync.Map。

sync.Map有以下特点：

- ◇ 内部通过冗余的数据结构降低加锁对性能的影响。
- ◇ 使用前无须初始化，直接声明即可。
- ◇ sync.Map不使用map中的方式进行读取和赋值等操作。

对于动手写6.5.2，我们使用sync.Map进行替换：

动手写6.5.3

```
01 package main
02
03 import (
04     "sync"
```

```

05      "fmt"
06      )
07
08 func main() {
09     var GoMap sync.Map
10     for i := 0; i < 100000; i++ {
11         go writeMap(GoMap, i, i)
12         go readMap(GoMap, i)
13     }
14     fmt.Println("Done")
15 }
16
17 func readMap(Gomap sync.Map, key int) int {
18     res, ok := Gomap.Load(key)          //线程安全读取
19     if ok == true{
20         return res.(int)
21     }else { return 0 }
22 }
23
24 func writeMap(Gomap sync.Map, key int, value int) {
25     Gomap.Store(key,value)           //线程安全设置
26 }
```

执行结果如下：

Done

注意：

- ◇ sync.Map无须使用make创建。
- ◇ Load()方法的第一个返回值是接口类型，需要将其转换为map值的类型。
- ◇ 目前sync.Map没有提供获取map数量的方法，解决方案是通过循环遍历map。
- ◇ 与较普通的map相比，sync.Map为了保证并发安全，会有性能上的损失，因此在非并发情况下，推荐使用map。

>> 第 7 章

函数 <<

在编程中经常会调用相同或者类似的操作，这些相同或者类似的操作由同一段代码完成，函数的出现，可以避免重复编写这些代码。函数的作用就是把相对独立的某个功能抽象出来，使之成为一个独立的实体。

例如，开发一个支持人与人之间进行对话的社交网站，对话这个功能比较复杂，可以将它封装为一个函数，每次调用该函数就可以发起对话；大型网站都有日志功能，对所有重要操作都会记录日志，而日志处理需要由多行Go文件操作相关代码组成，将这些代码组装为函数，则每次写日志时调用此函数即可。

Go语言函数支持的特性包括：

- ◇ 参数数量不固定（可变参数）。
- ◇ 匿名函数及其闭包。
- ◇ 函数本身作为值传递。
- ◇ 函数的延迟执行。
- ◇ 把函数作为接口调用。

其中，最后一个特性由于涉及接口相关的概念，会放在第12章中介绍，其他特性会在本章中一一讲解。

7.1 声明函数

Go语言中，函数的声明以关键字func为标识，具体格式如下：

```
func 函数名(参数列表) (返回参数列表) {  
    函数体  
}
```

◇ 函数名：函数名由字母、数字和下划线构成，但是函数名不能以数字开头；在同一个包内，函数名不可重复。

注意：可简单地将一个包理解为一个文件夹，在第8章中将会详细讲解Go语言包相关的概念。

◇ 参数列表：参数列表中的每个参数都由参数名称和参数类型两部分组成，参数变量为函数的局部变量。如果函数的参数数量不固定，Go语言函数还支持可变参数。

◇ 返回参数列表：返回参数列表中的每个参数由返回的参数名称和参数类型组成，也可简写为返回值类型列表。

◇ 函数体：函数体指函数的主体代码逻辑，若函数有返回参数列表，则函数体中必须有return语句返回值列表。

我们使用标准格式定义一个名为add的函数，其功能是进行两个整型数字的加法，并返回结果。

动手写7.1.1

```
01 func add(x int, y int) (sum int) {
02     sum = x + y
03     return sum
04 }
```

7.1.1 函数参数的简写

在参数列表中，如果相邻的变量为同类型，则不必重复写出类型。

对于动手写7.1.1，可将第一个参数x的类型省略：

```
01 func add(x, y int) (sum int) {
02     sum = x + y
03     return sum
04 }
```

7.1.2 函数返回值的简写

如果函数的返回值都是同一类型，在返回值列表中可将返回参数省略。

```
func returnValue() (int, int) {
    return 0, 1
}
```

一般情况下不推荐这种写法，因为会降低代码的可读性，无法区分每个返回值的实际意义。

7.1.3 带有变量名的返回值

使用带有变量名的返回值时，返回默认值为类型的默认值，函数结束处直接调用return即可。

```
func defaultValue() (a int,b string,c bool) {  
    return  
}
```

以上函数在调用后会返回：0、空字符串和false。

当然，如果return后跟返回值列表也是允许的：

```
func defaultValue() (a int,b string,c bool) {  
    return 1,"",false  
}
```

7.2 使用函数

在定义函数后，可通过对函数的调用使用函数，函数体内的代码逻辑执行完毕后，程序将继续执行被调用函数后的代码。

动手写7.2.1

```
01 package main  
02  
03 import "fmt"  
04  
05 func addSub(x int,y int) (sum int,sub int) {  
06     sum = x+y  
07     sub = x-y  
08     return sum,sub  
09 }  
10  
11 func main() {  
12     a :=1  
13     b :=2  
14     sum ,sub := addSub(a,b) //调用addSub函数  
15     fmt.Println(a,"+",b,"=",sum)
```

```

16     fmt.Println(a,"-",b,"=",sub)
17 }
```

执行结果如下：

```

1 + 2 = 3
1 - 2 = -1
```

以上程序中，我们一般将传入函数的变量a和b称为实参，将函数中的x和y称为形参。变量a和b通过值传递的方式将值赋给形参x和y。

注意：

- ◇ addSub函数中的形参x和y作用域仅限于函数体内。
- ◇ main函数中定义的变量sum和sub与addSub函数中定义的局部变量sum和sub完全无关，函数体内定义的变量作用域仅限于函数体内。
- ◇ 若不想接收函数的某个返回值，可用匿名变量“_”，但是不能所有返回值都用匿名变量代替。

7.3 函数变量

Go语言中，函数也是一种类型，我们可以将其保存在变量中。

函数变量的声明格式如下：

```
var 变量名称 func()
```

动手写7.3.1

```

01 package main
02
03 import "fmt"
04
05 func addSub(x int,y int) (sum int,sub int) {
06     sum = x+y
07     sub = x-y
08     return sum,sub
09 }
```

```
10 func main() {
11     a :=1
12     b :=2
13     var f1 func(x int,y int) (sum int,sub int)
14     f1 = addSub
15     sum ,sub := f1(a,b)
16     fmt.Println(a,"+",b,"=",sum)
17     fmt.Println(a,"-",b,"=",sub)
18 }
```

执行结果如下：

```
1 + 2 = 3
1 - 2 = -1
```

函数变量f1声明后其值初始化为nil，在将addSub函数赋值给f1后，所有对f1的调用即为对addSub函数的调用。

函数变量也可用短变量格式进行声明和初始化：

动手写7.3.2

```
01 package main
02
03 import "fmt"
04
05 func addSub(x int,y int) (sum int,sub int) {
06     sum = x+y
07     sub = x-y
08     return sum,sub
09 }
10
11 func main() {
12     a :=1
13     b :=2
14     f1 := addSub
15     sum ,sub := f1(a,b)
16     fmt.Println(a,"+",b,"=",sum)
17     fmt.Println(a,"-",b,"=",sub)
18 }
```

执行结果如下：

```
1 + 2 = 3
1 - 2 = -1
```

函数变量的声明和初始化推荐使用这种方式，可使代码更为简洁美观。

7.4 可变参数

Go语言支持可变参数的特性，即函数声明时可以没有固定数量的参数。

可变参数的函数格式如下：

```
func 函数名 (固定参数列表, v ...T) (返回参数列表) {
    函数体
}
```

- ◇ 可变参数一般放在函数参数列表的末尾，也可不存在固定参数列表。
- ◇ “v ...T” 代表的其实就是变量v为T类型的切片，v和T之间为三个“.”。

7.4.1 可变参数函数的使用

对于动手写7.1.1，add函数只能对两个int类型的数进行加法计算，现在我们可以使用可变参数实现任意个int类型数的加法运算，最后返回加法结果。

动手写7.4.1

```
01 package main
02
03 import "fmt"
04
05 func add(slice ... int) (int) {
06     sum := 0
07     for _, value := range slice{
08         sum = sum + value
09     }
10     return sum
11 }
12
13 func main() {
14     fmt.Println("1+2+...+9+10=", add(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
15 }
```

执行结果如下：

```
1+2+...+9+10= 55
```

在使用可变参数的函数体中，我们常常会使用for循环来对切片中的项进行操作。

7.4.2 可变参数与内置函数

Go语言中许多内置函数的参数都用了可变参数，比如最常用的fmt包中的Println函数和Printf函数。

fmt包中的Println函数源码如下（在GoLand编辑器中，可以将鼠标移到目标函数上，按下ctrl+鼠标左键来跳转到函数定义的代码块），Println函数所有的参数都为可变参数：

```
func Println(a ...interface{}) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}
```

Printf函数源码如下，第一个参数指定了需要打印的格式：

```
func Printf(format string, a ...interface{}) (n int, err error) {
    return Fprintf(os.Stdout, format, a...)
}
```

7.4.3 可变参数的传递

可变参数本质上是一个切片，如果要在多个函数中传递可变参数，可在传递时添加“...”。

我们对动手写7.4.1的例子稍加改动来进行可变参数传递：

动手写7.4.2

```
01 package main
02
03 import "fmt"
04
05 func addall(slice ... int) {
06     sum := 0
07     for _, value := range slice{
08         sum = sum + value
09     }
10     fmt.Println(sum)
```

```
11  }
12
13 func add(num ... int) {
14     addall(num...)
15 }
16
17 func main() {
18     add(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
19 }
```

执行结果如下：

55

如果想传递可变参数本身，可将addall函数的可变参数改为切片：

动手写7.4.3

```
01 package main
02
03 import "fmt"
04
05 func addall(slice []int) {
06     sum := 0
07     for _, value := range slice{
08         sum = sum + value
09     }
10     fmt.Println(sum)
11 }
12
13 func add(num ... int) {
14     addall(num)
15 }
16
17 func main() {
18     add(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
19 }
```

执行结果如下：

55

7.5

匿名函数和闭包

匿名函数即在需要函数时定义函数，匿名函数能以变量方式传递，它常常被用于实现闭包。

首先，我们来了解什么是匿名函数以及它的调用方式。

7.5.1 定义和使用匿名函数

匿名函数的格式如下：

```
func (参数列表) (返回参数列表) {  
    函数体  
}
```

匿名函数的调用有两种方式：

- ◇ 定义并同时调用匿名函数。
- ◇ 将匿名函数赋值给变量。

1. 定义并同时调用匿名函数

可以在匿名函数后添加“0”直接传入实参：

动手写7.5.1

```
01 package main  
02  
03 import "fmt"  
04  
05 func main() {  
06     func(data string) {  
07         fmt.Println("Hello "+data)  
08     } ("world!")  
09 }
```

执行结果如下：

Hello world!

2. 将匿名函数赋值给变量

将匿名函数赋值给一个变量，之后再进行调用：

动手写7.5.2

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     f1 := func(data string) {
07         fmt.Println("Hello "+data)
08     }
09     f1("world!")
10 }
```

执行结果如下：

```
Hello world!
```

7.5.2 闭包的定义

闭包就是包含了自由变量的匿名函数，其中的自由变量即使已经脱离了原有的自由变量环境也不会被删除，在闭包的作用域内可继续使用这个自由变量，同一个匿名函数和不同的引用环境组成了不同的闭包。

闭包就如同有“记忆力”一般，可对作用域内的变量的引用进行修改。

对于闭包有许多应用，在第10章中将会结合Go语言中的错误和异常进一步阐述闭包的作用。

7.5.3 闭包的“记忆力”

闭包可对作用域内变量的引用进行修改，在闭包内成功修改变量值后会对变量实际的值产生修改。

我们首先来看一个简单的例子：

动手写7.5.3

```
01 package main
02
03 import "fmt"
04
```

```
05 func main() {
06     num := 1
07     fmt.Printf("%p\n", &num)
08     func() {
09         num++
10         fmt.Println(num)
11         fmt.Printf("%p\n", &num)
12     }()
13     func() {
14         num++
15         fmt.Println(num)
16         fmt.Printf("%p\n", &num)
17     }()
18 }
```

执行结果如下：

```
0xc042066080
2
0xc042066080
3
0xc042066080
```

以上程序中的匿名函数由于在函数体内部引用了外部的自由变量num而形成了闭包。闭包每次对num变量的加1操作都是对变量num引用的修改。

动手写7.5.4

```
01 package main
02
03 import "fmt"
04
05 func addOne(i int) func() int {
06     return func() int {
07         i++
08         return i
09     }
}
```

```

10 }
11
12 func main() {
13     a1 := addOne(0)
14     fmt.Println(a1())           // 0+1=1
15     fmt.Println(a1())           // 1+1=2
16     a2 := addOne(10)
17     fmt.Println(a2())
18     fmt.Print("a1闭包的地址为: ")
19     fmt.Printf("%p\n", &a1)
20     fmt.Print("a2闭包的地址为: ")
21     fmt.Printf("%p\n", &a2)
22 }
```

执行结果如下：

```

1
2
11
a1闭包的地址为: 0xc042086018
a2闭包的地址为: 0xc042086028
```

addOne函数返回了一个闭包函数，通过定义a1和a2变量，创建了两个闭包的实例（引用环境不同导致）。

每次调用闭包实例，i的值都会在原有的基础上加1。从打印的结果可以看到，两个闭包实例的地址完全不同，两个闭包的调用结果互不影响。

7.6 延迟执行语句

Go语言中存在一种延迟执行的语句，由defer关键字标识，格式如下：

```
defer 任意语句
```

defer后的语句不会被马上执行，在defer所属的函数即将返回时，函数体中的所有defer语句将会按出现的顺序被逆序执行，即函数体中的最后一个defer语句最先被执行。

动手写7.6.1

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     fmt.Println("start now")
07     defer fmt.Println("这是第一句defer语句")
08     defer fmt.Println("这是第二句defer语句")
09     defer fmt.Println("这是第三句defer语句")
10     fmt.Println("end")
11 }
```

执行结果如下：

```
start now
end
这是第三句defer语句
这是第二句defer语句
这是第一句defer语句
```

由于defer语句是在当前函数即将返回时被调用，所以defer常常被用来释放资源。

动手写7.6.2

```
01 package main
02
03 import (
04     "fmt"
05     "net"
06 )
07
08 func tcpSend() {
09     conn, err := net.Dial("tcp", "www.baidu.com:80")
10     if err == nil {
11         defer conn.Close()
12         fmt.Println("remote address:", conn.RemoteAddr())
13     }
}
```

```

14     fmt.Println("error:", err)
15 }
16 func main() {
17     tcpSend()
18 }
```

执行结果如下：

```
remote address: 180.101.49.12:80
```

由于www.baidu.com对应的远端IP的80端口处于开启状态，在本机与该地址建立连接后和函数退出前，需要对打开的文件描述符conn进行关闭。

7.7 小结

- ◇ Go语言中，函数的声明以关键字func为标识。
- ◇ Go语言支持可变参数的特性，即函数声明时可以没有固定数量的参数。
- ◇ 匿名函数即在需要函数时定义函数，匿名函数能以变量方式传递，它常常被用于实现闭包。
- ◇ 闭包就是包含了自由变量的匿名函数，其中的自由变量即使已经脱离了原有的自由变量环境也不会被删除，在闭包的作用域内可继续使用这个自由变量。
- ◇ defer语句是在当前函数即将返回时被调用，所以defer常常被用来释放资源。

7.8 知识拓展

7.8.1 函数参数传递的本质

在讲述参数传递前，我们首先要了解两个基本概念：值传递和引用传递。

值传递：将变量的一个副本传递给函数，函数中不管如何操作该变量副本，都不会改变原变量的值。

引用传递：将变量的内存地址传递给函数，函数中操作变量时会找到保存在该地址的变量，对其进行操作，会改变原变量的值。

Go语言函数传入参数时使用的始终是值传递，对于值传递，Go语言主要分为以下两种情况：

1. 对于int、string和bool等值类型变量，传递的是原变量的副本，对副本的操作不会影响原

变量。

2. 对于指针、切片、map和channel（通道）引用类型变量，传递的是原变量指针的一份副本，该副本指向了原变量地址，因此对该副本的操作会影响原变量，从而达到了其他编程语言中类似于引用传递的效果。

下面我们通过实例来进行讲解。

动手写7.8.1

```
01 package main
02
03 import "fmt"
04
05 func passByValue(numPara int) { //值传递函数
06     fmt.Printf("passByValue函数中变量numPara地址为: %p\n", &numPara)
07     numPara = 100
08 }
09
10 func passByReference(numPara *int) { //引用传递函数
11     fmt.Printf("passByReference函数中指针变量numPara地址为: %p\n",
12                 &numPara)
13     fmt.Printf("passByReference函数中指针变量numPara指向的地址为: %p\
14             \n", numPara)
15     *numPara = 100
16 }
17
18 func main() {
19     num := 1
20     fmt.Printf("main函数中变量num地址为: %p\n", &num)
21     passByValue(num)
22     fmt.Printf("num变量值为: %d\n", num)
23     passByReference(&num)
24     fmt.Printf("num变量值为: %d\n", num)
25 }
```

执行结果如下：

```
main函数中变量num地址为: 0xc042066080
passByValue函数中变量numPara地址为: 0xc042066088
num变量值为: 1
passByReference函数中指针变量numPara地址为: 0xc042086020
passByReference函数中指针变量numPara指向的地址为: 0xc042066080
num变量值为:100
```

从程序执行结果可以发现，num变量在传递到passByReference函数后，其原本的值发生了变化。原因是其传递的是num变量的地址，该地址经过拷贝后传递给passByReference函数，但是其指向的值为num变量的地址（0xc042066080），因此在passByReference函数中的“*numPara = 100”会影响原变量的值。

类似地，使用map变量传递的是原变量的指针：

动手写7.8.2

```
01 package main
02
03 import "fmt"
04
05 func passByReference(mapNumReference map[int]int) {
06     fmt.Printf("passByReference函数中变量mapNumReference地址为: %p\n",
07                mapNumReference)
08     fmt.Printf("passByReference函数中变量mapNumReference所属指针地址
09                 为: %p\n", &mapNumReference)
10     mapNumReference[1] =100
11 }
12
13 func main() {
14     mapNum := map[int]int{1:10}
15     fmt.Printf("main函数中变量mapNum地址为: %p\n", mapNum)
16     fmt.Printf("main函数中变量mapNum所属指针的地址为: %p\n", &mapNum)
17     fmt.Printf("mapNum变量值为: %d\n", mapNum)
18     passByReference(mapNum)
19     fmt.Printf("mapNum变量值为: %d\n", mapNum)
20 }
```

执行结果如下：

```
main函数中变量mapNum地址为: 0xc04206e1e0
main函数中变量mapNum所属指针的地址为: 0xc042004028
mapNum变量值为: map [1:10]
passByReference函数中变量mapNumReference地址为: 0xc04206e1e0
passByReference函数中变量mapNumReference所属指针地址为: 0xc042004038
mapNum变量值为: map [1:100]
```

通过map类型变量参数传递的例子我们可以发现，函数进行值传递后拷贝的是map类型变量指针，但拷贝后的指针地址指向的还是map的地址，从而导致在函数passByReference中对map的操作会影响原变量。

7.8.2 Go内置函数

Go语言中自带了很多功能强大的内置函数，这些内置函数的定义都位于builtin.go文件中，使用时直接调用即可。本小节简要列出了Go的所有内置函数及其作用，如表7.8.1所示。

表7.8.1 Go内置函数

内置函数	功能
append	将数据追加到 slice 的末尾，返回一个 slice
copy	将源 slice 的数据复制到目标 slice 中，当目标 slice 的空间不够的时候会舍弃超出的部分，返回的是复制成功的元素个数
delete	删除 map 中对应键值对
len	计算数组、slice、map 和 channel 等类型的长度
cap	计算数组、slice 和 channel 类型的容量
make	为 slice、map 或 channel 类型分配内存并初始化对象
new	为 slice、map 和 channel 以外的类型分配内存并初始化对象，返回类型指针
complex	构建复数
real	返回复数的实部
imag	返回复数的虚部
close	用于关闭 channel
panic	触发宕机
recover	捕获 panic 内置函数抛出的异常信息
print	打印信息到标准输出，结尾没有换行符
println	打印信息到标准输出，结尾自动换行

这里以对切片和复数的操作作为应用场景，介绍几个Go内置函数及其作用。

我们创建sourceSlice切片，将元素4、5、6追加到该切片中，之后将其复制给targetSlice切片，输出切片的长度和容量。

动手写7.8.3

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     //将4、5、6追加到sourceSlice切片中
07     sourceSlice := []int{1, 2, 3}
08     sourceSlice = append(sourceSlice, 4, 5, 6)
09     fmt.Println("sourceSlice:", sourceSlice)
10
11     //将sourceSlice切片中的元素复制到targetSlice切片中
12     targetSlice := make([]int, 3)
13     num := copy(targetSlice, sourceSlice)
14     fmt.Println("复制成功的元素个数为: ", num)
15     fmt.Println("targetSlice:", targetSlice)
16     fmt.Println("targetSlice切片长度为: ", len(targetSlice))
17     fmt.Println("targetSlice切片容量为: ", cap(targetSlice))
18 }
```

执行结果如下：

```

sourceSlice: [1 2 3 4 5 6]
复制成功的元素个数为: 3
targetSlice: [1 2 3]
targetSlice切片长度为: 3
targetSlice切片容量为: 3
```

使用Go语言内置的complex函数可创建复数，并使用real函数和imag函数提取复数的实部与虚部。

动手写7.8.4

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     c1 := complex(1, 2)
07     fmt.Println(c1, "实部为: ", real(c1))
08     fmt.Println(c1, "虚部为: ", imag(c1))
09 }
```

执行结果如下：

```
(1+2i) 实部为: 1
(1+2i) 虚部为: 2
```

可以看到，在许多情况下，使用Go语言的内置函数可解决开发中的许多实际问题，避免自己使用代码重复实现相关逻辑。

对于大部分编程语言来说，代码包都是最有效的代码管理方式，Go语言也是使用包来管理代码的。如同其他语言一样，Go语言包的主要作用是把功能相似或相关的代码组织在同一个包中，以方便查找和使用。本章会详细介绍Go语言工程结构和包的使用。熟练掌握包管理是Go语言编码的基础。

8.1 工作区

Go语言中没有工程文件的概念，而是通过目录结构来体现工程的结构关系。Go代码必须放在工作区中。

8.1.1 工作区结构

工作区其实就是一个对应于特定工程的目录，它应包含三个子目录：src目录、pkg目录和bin目录。如下所示：

```
GO工作区：  
├── bin  
|   ...  
├── pkg  
|   └── windows_amd64  
|       ├── github.com  
|       ...  
└── src  
    ├── github.com  
    ...
```

其中各目录的作用如下：

- ◇ src目录：用于以代码包的形式组织并保存Go源码文件（如.go、.c、.h、.s等），同时也是Go编译时查找代码的地方。
- ◇ pkg目录：用于存放经由go get/install命令构建安装后的代码包的“.a”归档文件，也是编译生成的lib文件存储的地方。
- ◇ bin目录：与pkg目录类似，在通过go get/install命令完成安装后，保存由Go命令源码文件生成的可执行文件。

目录src用于包含所有的源代码，是Go命令行工具一个强制的规则，而pkg和bin则无须手动创建，必要时Go命令行工具在构建过程中会自动创建这些目录。

提示

归档文件也就是分类，在Linux中就是把一些文件放在一个包中归类，一般扩展名为“.a”，意思是archive。在C程序中也指程序编译后生成的静态库文件。

8.1.2 GOPATH

GOPATH可以理解为工作目录或工作区，也是平时接触最多的一个环境变量。它可以是一个目录，也可以是多个目录路径，每个目录代表一个工作区。

Go的所有操作（编码、依赖管理、构建、测试、安装等）基本上都是围绕GOPATH来进行的。GOPATH必须要设置，但并不是固定不变的，GOPATH的目的是为了告知Go需要代码的时候去哪里查找。需要注意的是，这里的代码包括本项目和引用外部项目的代码。GOPATH可以随着项目的不同而重新设置。

在实际开发环境中，工作目录往往有多个。这些工作目录的目录路径都需要添加至GOPATH。为了能够构建这个工程，需要先把所需工程的根目录加入到环境变量GOPATH中。否则，即使处于同一工作目录（工作区），代码之间也无法通过绝对代码包路径完成调用。

这里重点介绍一下如何使用命令行添加GOPATH。Windows下的GOPATH默认为：%USERPROFILE%\go；Linux下默认为：\$HOME/go。在Windows下，使用echo命令查看当前的GOPATH，需要在GOPATH前后添加“%”，表示为一个系统变量：

```
C:\Users\admin>echo %GOPATH%
C:\Users\admin\go
```

临时添加C:\NewProject目录到GOPATH可使用set命令，Windows下使用分号作为分隔符：

```
C:\Users\admin>set GOPATH=%GOPATH%;C:\NewProject

C:\Users\admin>echo %GOPATH%
C:\Users\admin\go;C:\NewProject
```

使用set添加GOPATH，添加的是临时变量，当cmd窗口结束时，该变量就会结束。如果想要永久添加一个GOPATH路径，可以使用setx，只需要将如上命令的“set”更换成“setx”即可。

在Linux下，查看GOPATH使用echo，须在GOPATH前添加“\$”符号表示为系统变量：

```
[root@localhost ~]# echo $GOPATH
/root/go
```

临时添加一个/home/NewProject目录到GOPATH可使用如下命令，Linux下使用冒号作为分隔符：

```
GOPATH=$GOPATH: /home/NewProject
```

如果要永久添加该路径到GOPATH中，需要修改配置文件/etc/profile，向该文件末尾添加如下信息：

```
export GOPATH=$GOPATH:/home/NewProject

[root@localhost ~]# echo $GOPATH
/root/go:/home/NewProject
```

之后再使用source命令重新执行刚修改的初始化文件，使之立即生效：

```
source /etc/profile
```

如果GOPATH设置了多个工作区，那么查找依赖包是以怎样的顺序进行呢？例如包a依赖包b，包b依赖包c，那么会先查找c包。那在工作区是如何查找这个依赖包c的呢？

首先，在查找依赖包的时候，总是会先查找GOROOT目录，也就是Go语言的安装目录，如果没有找到依赖的包，才到工作区去找相应的包。在工作区中是按照设置的先后顺序来查找的，也就是会从第一个开始依次查找，如果找到就不再继续，如果没有找到就报错。

如果多个工作区中存在导入路径相同的代码包会产生冲突吗？不冲突，因为按顺序找到所需要的包就不往后继续找了。

提示

go get会下载代码包到src目录，但是只会下载到第一个工作区目录。

8.1.3 GOROOT

GOROOT是Go语言的程序安装目录，并非必须要设置的。Linux默认Go会安装在/usr/local/go下，但也允许自定义安装位置。GOROOT的目的就是告知Go当前的安装位置，编译的时候从GOROOT去找SDK的system library。

Windows下查看GOROOT：

```
C:\Users\admin>echo %GOROOT%
D:\Go\
```

8.1.4 GOBIN

GOBIN目录一般为GOPATH的可执行文件放置目录，一般指bin。默认安装的Go语言是没有设置GOBIN目录的。

需要注意的是，只有当环境变量GOPATH中只包含一个工作区的目录路径时，go install命令才会把命令源码安装到当前工作区的bin目录下。若环境变量GOPATH中包含多个工作区的目录路径，像这样执行go install命令就会失效，此时必须设置环境变量GOBIN。

Windows下查看GOBIN：

```
C:\Users\admin>echo %GOBIN%
C:\Users\admin\go\bin
```

8.2 包的声明

包是结构化代码的一种方式：每个程序都由包（通常简称为pkg）的概念组成，可以使用自身的包或者从其他包中导入内容。我们先来了解一下Go语言源码的组织方式：

- ◇ Go语言的源代码以代码包为基本的组织单位。
- ◇ 在文件系统中，代码包是与文件目录一一对应的，文件目录的子目录也就是代码包的子包。
- ◇ 在工作区中，一个代码包的导入路径实际上就是从src子目录到该包的实际存储位置的相对路径。

每一个Go源文件的第一行都需要声明包的名称，声明一个包使用关键字package。如声明一个main包：

```
package main
```

如同其他一些编程语言中的类库或命名空间的概念，每个Go文件都属于且仅属于一个包。一个包可以由许多以“.go”为扩展名的源文件组成，因此文件名和包名一般来说都是不相同的。

代码包的名称一般会与源码文件所在的目录同名，如果不同名，在构建和安装的过程中会以代码包的名称为准。

在Go语言中，代码包中的源码文件名可以是任意的。但是，这些任意名称的源码文件都必须以包声明语句作为文件中的第一行，每个包都对应一个独立的名字空间。

包中成员以名称首字母大小写决定访问权限：

- ◇ `public`: 首字母大写，可被包外访问。
- ◇ `private`: 首字母小写，仅包内成员可以访问。

注意：同一个目录下不能定义不同的`package`。

8.3 包的导入

在Go语言程序中，每个包都有一个全局唯一的导入路径。导入语句中类似“`"fmt"`”的字符串对应包的导入路径。

Go语言的规范并没有定义这些字符串的具体含义或包来自哪里，它们是由构建工具来解释的。一个导入路径代表一个目录中的一个或多个Go源文件。

8.3.1 导入声明

导入包需要使用关键字`import`，它会告诉编译器你想引用该位置的包中的代码，包的路径可以是相对路径，也可以是绝对路径。

```
// 声明方式一
import "fmt"
import "time"

// 声明方式二
import (
    "fmt"
    "time"
)
```

如上所示的标准包会在Go语言的安装位置下的`src`目录中找到，而其他包都需要在`GOPATH`环境变量中的目录下寻找，`GOPATH`指定的这些目录就是开发者的个人工作空间。

如果在GOPATH中仍然没有找到所需的包，编译器在编译或运行时就会报错。

8.3.2 远程导入

我们经常使用的包都是标准库中的包，也就是本地包。Go语言不仅支持我们调用本地包，还支持调用远程服务器的包，例如托管在GitHub上的一个非常热门的Web框架gin，我们就可以通过远程导入的方式将其导入使用。

```
import "github.com/gin-gonic/gin"
```

在导入之前还需要使用go get命令下载远程包：

```
go get github.com/gin-gonic/gin
```

以下是gin官方提供的一个框架入门的例子：

动手写8.3.1

```
01 package main
02
03 import "github.com/gin-gonic/gin"
04
05 func main() {
06     r := gin.Default()
07     r.GET("/ping", func(c *gin.Context) {
08         c.JSON(200, gin.H{
09             "message": "pong",
10         })
11     })
12     r.Run() // listen and serve on 0.0.0.0:8080
13 }
```

运行后，就启动了一个HTTP服务器，并且可以看到输出了如下提示信息：

```
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and
Recovery middleware already attached.

[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release"
mode in production.

- using env: export GIN_MODE=release
- using code: gin.SetMode(gin.ReleaseMode)
```

```
[GIN-debug] GET      /ping          --> main.main.func1 (3 handlers)
[GIN-debug] Environment variable PORT is undefined. Using port: 8080
by default
[GIN-debug] Listening and serving HTTP on :8080
```

浏览器访问`http://localhost:8080/ping`后可以看到服务器返回如下消息（关于gin的这个Web框架，在后续章节会重点介绍，这里只作简要说明）：

```
{"message": "pong"}
```

此时控制台会输出如下信息，表示有人请求了我们的http服务：

```
[GIN] 2019/05/11 - 09:15:50 | 200 | 999.1µs | ::1 | GET /ping
```

8.3.3 别名导入

在导入包时，可以指定包成员的访问方式，例如对包进行重命名，也就是使用别名的方式，避免导入两个相同包名的不同包而发生冲突。

定义包的别名只需要在导入的包名前指定别名即可。例如将fmt包取一个别名叫print：

```
import print "fmt"
```

动手写8.3.2

```
01 package main
02
03 import (
04     print "fmt"
05     date "time"
06 )
07
08 func main() {
09     print.Println("当前日期为: ", date.Now().Day())
10 }
```

动手写8.3.2定义了两个别名，fmt包的别名print和time包的别名date。需要注意的是，定义了别名后，就不能再使用该包原本的名字来调用它，只能使用它的别名来调用，运行结果如下：

```
当前日期为: 11
```

8.3.4 匿名导入

在Go语言中，如果导入了一个包而不使用，在编译时会产生“unused import”的编译错误。有时我们可能需要导入一个包，但不会引用到这个包的标识符，在这种情况下可以使用包的匿名导入的方式，就是使用下划线“_”来重命名该包。

```
import _ "fmt"
```

使用匿名导入来引入一个包，而不使用包里的函数，主要是为了调用该包中的init函数。

例如当我们想分析一个Web应用程序的性能时，导入net/http/pprof包即可，该包的init函数主要做了路由匹配，具体如下：

```
func init() {
    http.HandleFunc("/debug/pprof/", Index)
    http.HandleFunc("/debug/pprof/cmdline", Cmdline)
    http.HandleFunc("/debug/pprof/profile", Profile)
    http.HandleFunc("/debug/pprof/symbol", Symbol)
    http.HandleFunc("/debug/pprof/trace", Trace)
}
```

动手写8.3.3

```
01 package main
02
03 import (
04     "net/http"
05     _ "net/http/pprof"
06 )
07
08 func ping(writer http.ResponseWriter, request *http.Request) {
09     writer.Write([]byte(`{"message": "pong"}`))
10 }
11
12 func main() {
13     http.HandleFunc("/ping", ping)
14     http.ListenAndServe(":8080", nil)
15 }
```

动手写8.3.3实现了和动手写8.3.1一样的功能，但匿名导入了性能分析包pprof后就可以对该Web程序进行性能分析；之后浏览器访问`http://localhost:8080/debug/pprof`就可以查看相关的性能参数，还可以使用`go tool pprof`命令进行辅助分析。关于程序性能如何分析会在后续章节中介绍。

Go语言中的数据库操作也使用了相同的方法，让用户导入程序所必需的驱动，例如MySQL的驱动包`github.com/go-sql-driver/mysql`，使用“`go get github.com/go-sql-driver/mysql`”命令安装该包，该包的`init`函数所做的事就是注册MySQL驱动。

```
func init() {
    sql.Register("mysql", &MySQLDriver{})
}
```

当我们想要使用该驱动时就可以用匿名导入的方式导入驱动。

动手写8.3.4

```
01 package main
02
03 import (
04     "database/sql"
05     _ "github.com/go-sql-driver/mysql" // 导入MySQL驱动
06 )
07
08 func main() {
09     dbname := "db连接字符串"
10     db, err := sql.Open("mysql", dbname) // OK
11     // 数据库操作
12     _, _ =db, err
13 }
```

动手写8.3.4导入了MySQL驱动，使得程序支持了操作MySQL的能力。

8.4 main包

在Go语言中，命名为`main`的包具有特殊的含义。Go语言的编译程序会试图把这种名字的包编译为二进制可执行文件。所有用Go语言编译的可执行程序都必须有一个名叫`main`的包，`main`包有且仅有一个。

当编译器发现某个包的名字为main时，它一定也会发现名为main()的函数，否则不会创建可执行文件。main()函数是程序的入口，所以，如果没有这个函数，程序就没有办法开始执行。程序编译时，会使用声明main包的代码所在的目录名称作为可执行文件的文件名。

一个应用程序可以包含不同的包，而且即使你只使用main包也不必把所有的代码都写在一个巨大的文件里，你可以用一些较小的文件，并且在每个文件非注释的第一行都使用package main来指明这些文件都属于main包。如果你打算编译包名不为main的源文件，如mypack，编译后产生的对象文件将会是mypack.a而不是可执行程序。

另外要注意的是，所有的包名都应该使用小写字母。

8.5 包的初始化

在Go语言里面有两个保留的函数：init函数（能够应用于所有的package）和main函数（只能应用于package main）。这两个函数在定义时不能有任何的参数和返回值。虽然一个package里面可以写任意多个init函数，但无论是对于可读性还是以后的可维护性来说，我们都强烈建议用户在一个package中，每个文件只写一个init函数。

我们导入了一个包就可以使用该包中的常量和变量，当然这些常量和变量必须是可导出的。

动手写8.5.1

```
01 package main
02
03 import (
04     "fmt"
05     "time"
06 )
07
08 func main() {
09     fmt.Printf("time包中的常量Second: %v \n", time.Second)
10     fmt.Printf("time包中的UTC变量: %v \n", time.UTC)
11     time.UTC, _ = time.LoadLocation("Local")
12     fmt.Printf("time包中的UTC变量更改后的值: %v \n", time.UTC)
13 }
```

动手写8.5.1导入了time包，使用了time包的Second常量和UTC变量，运行结果如下：

```
time包中的常量Second: 1s
time包中的UTC变量: UTC
time包中的UTC变量更改后的值: Local
```

Go程序会自动调用init()和main()，所以你不需要在任何地方调用这两个函数。每个package中的init函数都是可选的，但package main就必须包含一个main函数。

每个包可以包含任意多个init()，这些函数都会在程序执行开始的时候被调用。所有被编译器发现的init()都会安排在main()之前执行。init()用于设置包、初始化变量或者其他要在程序运行前优先完成的引导工作。

程序的初始化和执行都起始于main包。如果main包还导入了其他的包，那么就会在编译时将它们依次导入。

有时一个包会被多个包同时导入，但它只会被导入一次（例如很多包可能都会用到fmt包，但它只会被导入一次，因为没有必要导入多次）。

当一个包被导入时，如果该包还导入了其他的包，那么会先将其他的包导入进来，然后再对该包的包级常量和变量进行初始化，最后执行init函数（如果存在）。包的导入如图8.5.1所示，是一个递归的过程，等所有被导入的包加载完毕，就会对main包中的包级常量和变量进行初始化，然后执行main包中的init函数（如果存在），最后执行main函数。

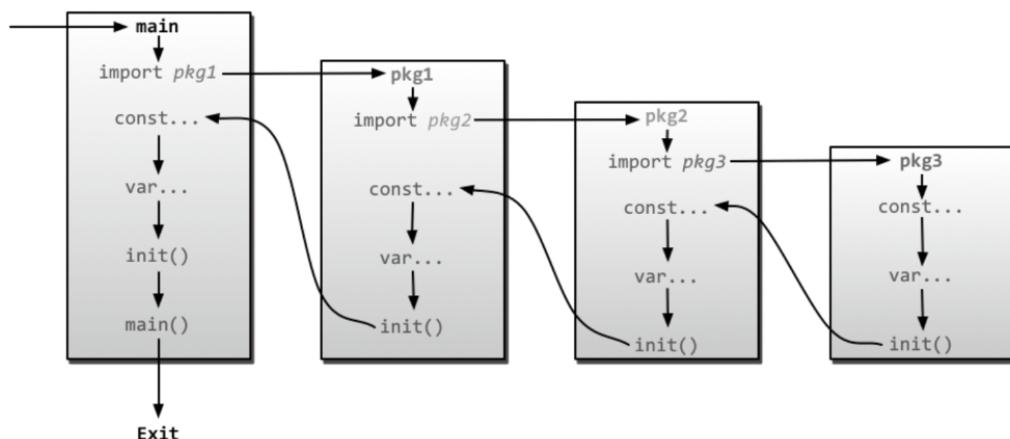


图8.5.1 包的导入

8.6 包的命名

每个包都有一个包名，包名一般是短小的名字，在包的声明处指定。通常来说，默认的包名就是包导入路径名的最后一段，因此即使两个包的导入路径不同，它们依然可能有一个相同的包

名。包的命名有如下规则：

第一条规则，同目录下的源码文件的代码包声明语句要一致。也就是说，它们要同属于一个代码包。这对于所有源码文件都是适用的。

第二条规则，源码文件声明的代码包的名称可以与其所在的目录的名称不同（import后面的最后一个元素是路径，即目录，并非包名）。如当前目录名为project1，但包名可以为main，或者为mypkg。

```
package main

package mypkg
```

8.7 依赖包管理

在Go语言开发过程中，我们经常会使用很多第三方的依赖包，如GitHub的第三方包。当第三方包做了不兼容升级，仅通过GOPATH来管理依赖包会产生严重的问题，如果我们升级了依赖包，则会严重影响我们的项目。因此在1.5版本以前，为了规避这个问题，通常会将当前使用的依赖包拷贝出来。

为了解决这个问题，Go在1.5版本引入了vendor属性（默认关闭，需要设置Go环境变量GO15VENDOREXPERIMENT=1），并在1.6版本中默认开启了vendor属性。

简单来说，vendor属性就是让Go在编译时，优先从项目源码树根目录下的vendor目录查找代码（可以理解为GOPATH），如果vendor中有，则不再去GOPATH中查找。

然而，有时候依赖包多了，逐个拷贝是非常费时费力的，因此在Go 1.11之后，官方发布了支持的版本管理工具mod。使用go mod命令即可运行mod工具。mod支持如下命令：

download	下载依赖的module到本地cache
edit	编辑go.mod文件
graph	打印模块依赖图
init	在当前文件夹下初始化一个新的module，创建go.mod文件
tidy	增加丢失的module，去掉未用的module
vendor	将依赖复制到vendor下
verify	校验依赖
why	解释为什么需要依赖

当需要将所有外部依赖拷贝到vendor目录下时，只需要运行一条命令即可：

```
go mod vendor
```

对于低于Go 1.11的版本，可以使用第三方工具实现，这里推荐使用govendor进行依赖项管理。使用go get命令安装govendor：

```
go get -u github.com/kardianos/govendor
```

govendor主要提供了如下命令：

init	初始化vendor目录
list	列出所有的依赖包
add	加包到vendor目录，如govendor add +external添加所有外部包
add PKG_PATH	添加指定的依赖包到vendor目录
update	从\$GOPATH更新依赖包到vendor目录
remove	从vendor管理中删除依赖
status	列出所有缺失、过期和修改过的包
fetch	添加或更新包到本地vendor目录
sync	本地存在vendor.json时拉取依赖包，匹配所记录的版本
get	类似go get目录，拉取依赖包到vendor目录

govendor生成vendor目录的时候需要两条命令：

govendor init	生成vendor/vendor.json，此时文件中只有本项目的信息
govendor add +external	更新vendor/vendor.json，并拷贝GOPATH下的代码到 vendor目录中

其中命令“govendor add +external”也可以简写为“govendor add +e”。

8.8 Go语言命名规范

对于Go语言命名规范，每一家公司根据自己的实际情况可能都有不同。一般而言，为了增强代码的可读性，使不同开发者编写的代码更易于理解，对于命名的规范会达成某种共识，如驼峰式命名法和package命名规范。另外，我们也可以多阅读Go语言标准库源码，因为标准库的代码可以说是我们写代码参考的标杆。

8.8.1 驼峰式命名法

Go语言中，我们推荐使用驼峰式命名法来对各类标识符（变量、常量和结构体等）和文件进行命名。驼峰式命名法有两种格式：小驼峰命名法和大驼峰命名法。

1. 小驼峰命名法

第一个单词以小写字母开始，其他单词的首字母则需要大写：

```
var sumNum int
var firstName string
var isValid bool
var printValue func()
```

2. 大驼峰命名法

每一个单词的首字母都采用大写：

```
var SumNum int
var FirstName string
var IsValid bool
var PrintValue func()
```

需要注意的是，在进行测试用例的编写时，文件的命名不遵循驼峰式命名法，而是以xx_test.go的格式进行命名。测试用例的函数以大驼峰命名法进行命名且以“Test”开头。

如Go语言fmt包中的某个测试文件scan_test.go中的TestScan函数：

```
func TestScan(t *testing.T) {
    for _, r := range readers {
        t.Run(r.name, func(t *testing.T) {
            testScan(t, r.f, Fscan)
        })
    }
}
```

8.8.2 导出标识符

Go语言中标识符根据首字母的大小写来确定可以访问的权限。如果首字母大写，则表示可以被其他包中的代码访问（可被导出）；如果首字母小写，则表示该标识符只能在本包中使用（不可导出）。如果你学习过其他编程语言，那么你可以这样理解：首字母大写的标识符是公有的，而首字母小写的标识符是私有的。

例如，下方的目录结构中，main文件夹下有main.go文件，pkg文件夹下有pkg.go文件：

```
├── main
│   └── main.go
└── pkg
    └── pkg.go
```

pkg.go属于pkg包，代码如下，其中只有SumNum变量是可导出的：

动手写8.8.1

```
01 package pkg
02
03 import "fmt"
04
05 var SumNum = 1
06 var firstName = "Peter"
07 var isValid = false
08 func printValue () {
09     fmt.Println("01快学")
10 }
```

main.go属于main包，代码如下，调用并打印了pkg包下的SumNum变量：

```
01 package main
02
03 import (
04     "fmt"
05     "book/pkg"
06 )
07
08 func main () {
09     fmt.Println("pkg包下的SumNum变量值为：", pkg.SumNum)
10 }
```

运行结果如下：

```
pkg包下的SumNum变量值为：1
```

对于pkg包中的firstName、isValid和printValue函数，由于是不可导出的，所以在其他包中无法访问。

8.9 小结

- ◇ GOPATH必须要设置，但并不是固定不变的。GOPATH的目的是为了告知Go需要代码的时候去哪里查找。

- ◇ package是Go语言最基本的分发单位和工程管理中依赖关系的体现。
- ◇ 每个Go语言源代码文件开头都拥有一个package声明，表示该Go语言代码所属的package。
- ◇ 要生成Go语言可执行程序，必须建立一个名为main的package，并且在该package中必须包含一个名为main()的函数。
- ◇ import关键字导入的是package路径，而在源文件中使用package时，才需要package名。经常可见的import的目录名和源文件中使用的package名可以不一致。
- ◇ 不得导入源代码文件中没有用到的package，否则Go语言编译器会报编译错误。
- ◇ 解决包依赖问题可以使用go mod命令或者第三方工具govendor。

8.10 知识拓展

标准包简介

1. unsafe

unsafe包含了一些打破Go语言“类型安全”的命令，一般的程序不会使用它，可用于C/C++程序的调用中。

2. os-os/exec-syscall

os：提供给我们一个平台无关性的操作系统功能接口，采用类Unix设计，隐藏了不同操作系统间的差异，让不同的文件系统和操作系统对象表现一致。

os/exec：提供给我们运行外部操作系统命令和程序的方式。

syscall：底层的外部包，提供了操作系统底层调用的基本接口。

3. archive/tar和zip-compress

压缩（解压缩）文件功能。

4. fmt-io-bufio-path/filepath-flag

fmt：提供了格式化输入输出功能。

io：提供了基本输入输出功能，大多数是围绕系统功能的封装。

bufio：缓冲输入输出功能的封装。

path/filepath：用来操作在当前系统中的目标文件名路径。

flag：对命令行参数的操作。

5. strings-strconv-unicode-regexp-bytes

strings：提供对字符串的操作。

strconv：提供将字符串转换为基础类型的功能。

unicode：为unicode型的字符串提供特殊的功能。

regexp：正则表达式功能。

bytes：提供对字符型分片的操作。

6. math–math/cmath–math/rand–sort–math/big

math：基本的数学函数。

math/cmath：对复数的操作。

math/rand：伪随机数生成。

sort：为数组排序和自定义集合。

math/big：大数的实现和计算。

7. container–list–ring–heap

实现对集合的操作。

list：双链表。

ring：环形链表。

8. time–log

time：日期和时间的基本操作。

log：记录程序运行时产生的日志。

9. encoding/Json–encoding/xml–text/template

encoding/Json：读取并解码和写入并编码JSON数据。

encoding/xml：简单的XML 1.0解析器。

text/template：生成像HTML一样的数据与文本混合的数据驱动模板。

10. net–net/http–html

net：网络数据的基本操作。

http：提供了一个可扩展的HTTP服务器和客户端，解析HTTP请求和回复。

html：HTML5解析器。

11. runtime

Go程序运行时的交互操作，例如垃圾回收和协程创建。

12. reflect

实现通过程序运行时反射，让程序操作任意类型的变量。

>> 第 9 章

结构体 <<

在Go语言官网的常见问题解答一栏中，有这样一个问题：“Is Go an object-oriented language? (Go语言是否是一种面向对象语言？)” 比较有意思的是，Go官方给出的回答是：“Yes and no. (既是也不是。)”

不同于Python和C++等编程语言，虽然Go有着面向对象语言的编程风格，但Go并不是典型的面向对象的语言。Go语言中不支持类，但它提供了结构体，Go语言中的结构体和接口实现了面向对象编程的特性。此外，Go中的方法比C++或Java等编程语言中的方法更通用，可以为任何类型的数据定义这些方法。

9.1 理解结构体

通过使用结构体，我们可以自定义一系列由相同类型或不同类型的数据构成的数据集合，用来实现较复杂的数据结构。

Go语言的结构体是一种对现实生活中实体的抽象，结构体由一系列成员变量构成，这些成员变量对应着实体不同的属性。

例如，对于某个图书馆中的所有书籍，我们可以抽象出书这个结构体，经过总结归纳，该图书馆中的书都拥有以下属性：书名、作者、数量、ID号。

那么，我们可以定义以下结构体：

结构体名：book

◇ 成员1：书名

◇ 成员2：作者

- ◇ 成员3：数量
- ◇ 成员4：ID号

9.2 定义结构体

GO语言中通过关键字type定义自定义类型，结构体定义需要使用type和struct关键字，具体定义格式如下：

```
type 结构体名 struct {
    成员变量1 类型1
    成员变量2 类型2
    成员变量3 类型3
    ...
}
```

注意：

- ◇ 结构体名：同一个包内结构体名不能重复。
- ◇ 成员名：同一个结构体内，成员名不能重复。
- ◇ 类型1、类型2……：表示结构体成员变量的类型。
- ◇ 同类型的成员名可以写在同一行。
- ◇ 当结构体、方法名或变量名的首字母为大写时（可被导出），就可以在当前包外进行访问。

对于book结构体，其定义如下：

```
type Book struct {
    title string
    author string
    num int
    id int
}
```

对于同变量类型的结构体成员，可以将它们写在同一行：

```
type Book struct {
    title,author string
    num,id int
}
```

9.3 实例化结构体

一个结构体在定义完成后才能进行使用。结构体实例化时，会真正地分配内存。因此，必须在定义结构体并实例化后才能使用结构体的字段。Go语言实例化结构体主要有以下三种方式：

- ◇ 标准实例化。
- ◇ new函数实例化。
- ◇ 取地址实例化。

9.3.1 标准实例化

结构体本身是一种自定义数据类型，和其他基本数据类型的声明格式类似，可以通过var关键字进行实例化。

标准实例化格式如下：

```
var    结构体实例    结构体类型
```

下方的程序实例化了book1，对book1的各个成员进行了初始化赋值。

动手写9.3.1

```
01 package main
02
03 import "fmt"
04
05 type Book struct {
06     title string
07     author string
08     num int
09     id int
10 }
11 func main () {
12     var book1 Book
13     fmt.Println(book1)
14 }
```

执行结果如下：

```
{ 0 0 }
```

由于string和int类型变量的初始值为""和0，因此程序执行结果如上。

9.3.2 new函数实例化

new函数可对结构体进行实例化，实例化完成后会返回结构体的指针类型。

动手写9.3.2

```

01 package main
02
03 import "fmt"
04
05 type Book struct {
06     title string
07     author string
08     num int
09     id int
10 }
11 func main (){
12     book1 := new(Book)
13     fmt.Println(book1)
14 }
```

执行结果如下：

```
&{ 0 0 }
```

9.3.3 取地址实例化

取地址实例化与使用new函数进行实例化类似，返回的是结构体指针类型。取地址实例化格式如下：

```
结构体实例 := &结构体类型 { }
```

动手写9.3.3

```

01 package main
02
03 import "fmt"
04
05 type Book struct {
06     title string
07     author string
```

```
08     num int
09     id int
10 }
11 func main () {
12     book1 := &Book{ }
13     fmt.Println(book1)
14 }
```

执行结果如下：

```
&{ 0 0 }
```

9.3.4 访问成员

结构体实例化完成后，可通过“.”来访问结构体的成员变量，对成员变量进行赋值或修改。

动手写9.3.4

```
01 package main
02
03 import "fmt"
04
05 type Book struct {
06     title string
07     author string
08     num int
09     id int
10 }
11 func main () {
12     book1 := &Book{ }
13     book1.title= "Go语言"
14     book1.author= "Tom"
15     book1.num= 20
16     book1.id= 152368
17     fmt.Println("title:",book1.title)
18     fmt.Println("author:",book1.author)
19     fmt.Println("num:",book1.num)
20     fmt.Println("id:",book1.id)
21 }
```

执行结果如下：

```
title: Go语言
author: Tom
num: 20
id: 152368
```

不论使用哪种方式对结构体进行实例化，都可通过“.”的方式来对结构体的成员变量进行访问。

9.4 初始化结构体

上一小节我们讲解了在结构体实例化后，再使用“.”的方式对成员变量进行赋值。另外，我们还可以在声明的同时对结构体的成员变量进行赋值。

Go语言中可以通过键值对格式和列表格式对结构体进行初始化。

9.4.1 键值对格式初始化

键值对初始化的格式如下：

```
结构体实例 := 结构体类型{
    成员变量1:值1,
    成员变量2:值2,
    成员变量3:值3,
}
```

这种类型的初始化类似于对map数据类型的初始化操作，键和值之间以冒号分隔，键值对之间以逗号分隔。

动手写9.4.1

```
01 package main
02
03 import "fmt"
04
05 type Book struct {
06     title string
07     author string
}
```

```
08     num int
09     id int
10 }
11 func main () {
12     book1 := &Book{
13         title:"Go语言",
14         author:"Tom",
15         num:20,
16         id:152368,
17     }
18     fmt.Println("title:",book1.title)
19     fmt.Println("author:",book1.author)
20     fmt.Println("num:",book1.num)
21     fmt.Println("id:",book1.id)
22 }
```

执行结果如下：

```
title: Go语言
author: Tom
num: 20
id: 152368
```

9.4.2 列表格式初始化

如果觉得使用键值对初始化结构体的方式比较麻烦，那么我们可以使用一种类似列表的方式对结构体进行初始化，格式如下：

```
结构体实例 := 结构体类型 {
    值1,
    值2,
    值3,
}
```

注意：

- ◇ 使用这种方式初始化结构体必须初始化所有的成员变量。
- ◇ 值的填充顺序必须和结构体成员变量声明顺序保持一致。
- ◇ 该方式与键值对的初始化方式不能混用。

动手写9.4.2

```
01 package main
02
03 import "fmt"
04
05 type Book struct {
06     title string
07     author string
08     num int
09     id int
10 }
11 func main (){
12     book1 := &Book{
13         "Go语言",
14         "Tom",
15         20,
16         152368,
17     }
18     fmt.Println("title:",book1.title)
19     fmt.Println("author:",book1.author)
20     fmt.Println("num:",book1.num)
21     fmt.Println("id:",book1.id)
22 }
```

执行结果如下：

```
title: Go语言
author: Tom
num: 20
id: 152368
```

9.5

结构体方法

Go语言中，一个方法就是一个包含了接收者的函数。对于结构体方法，接收者可以是结构体类型的值或是指针。

9.5.1 指针类型接收者

当接收者类型为指针时，可以通过该方法改变该接收者的成员变量值，即使你使用了非指针类型实例调用该函数，也可以改变实例对应的成员变量值。

在以下示例中，我们为User结构体创建了ChangeName方法，在该方法内部将接收者的Name值修改为Tom，接收者的类型为指针类型：

动手写9.5.1

```
01 package main
02
03 import "fmt"
04
05 type User struct {
06     Name string
07     Email string
08 }
09
10 func (u *User) ChangeName() { //指针类型接收者
11     u.Name = "Tom"
12 }
13
14 func main() {
15     u := &User{"Peter", "go@go.com"} //创建指针类型结构体实例
16     fmt.Println("Name:", u.Name, " Email:", u.Email)
17     u.ChangeName()
18     fmt.Println("Name:", u.Name, " Email:", u.Email)
19 }
```

执行结果如下：

```
Name: Peter Email: go@go.com
Name: Tom Email: go@go.com
```

只要结构体方法的接收者为指针类型，即使实例不是指针类型，修改也能生效：

动手写9.5.2

```
01 package main
02
03 import "fmt"
04
```

```

05 type User struct {
06     Name string
07     Email string
08 }
09
10 func (u *User) ChangeName() { //指针类型接收者
11     u.Name = "Tom"
12 }
13
14 func main() {
15     u := User{"Peter", "go@go.com"} //创建非指针类型结构体实例
16     fmt.Println("Name:", u.Name, " Email:", u.Email)
17     u.ChangeName()
18     fmt.Println("Name:", u.Name, " Email:", u.Email)
19 }
```

执行结果如下：

```

Name: Peter Email: go@go.com
Name: Tom Email: go@go.com
```

9.5.2 值类型接收者

当接收者不是一个指针时，该方法操作对应接收者值的副本，否则即使你使用了指针调用函数，也无法改变成员变量值。

动手写9.5.3

```

01 package main
02
03 import "fmt"
04
05 type User struct {
06     Name string
07     Email string
08 }
09
10 func (u User) ChangeName() { //值类型接收者
```

```
11     u.Name = "Tom"
12 }
13
14 func main() {
15     u := &User{"Peter", "go@google.com"} //创建指针类型结构体实例
16     fmt.Println("Name:", u.Name, " Email:", u.Email)
17     u.ChangeName()
18     fmt.Println("Name:", u.Name, " Email:", u.Email)
19 }
```

执行结果如下：

```
Name: Peter Email: go@google.com
Name: Peter Email: go@google.com
```

如果方法需要修改接收者，接收者必须是指针类型。一般约定的接收者命名是类型的一个或两个字母的缩写（如u或者us之于User）。

9.6 结构体内嵌

Go语言的结构体内嵌是一种组合特性，使用结构体内嵌可构建一种面向对象编程思想中的继承关系。结构体实例化后，可直接访问内嵌结构体的所有成员变量和方法。结构体内嵌格式如下：

```
type 结构体名1 struct {
    成员变量1 类型1
    成员变量2 类型2
}
type 结构体名2 struct {
    结构体名1
    成员变量3 类型3
}
```

对于图书馆中的书，我们可大致分为两类：可外借的书和不可外借的书。我们现在将图书馆中的书抽象为结构体Book，将可借阅的书抽象为结构体BookBorrow，将不可借阅的书抽象为结构体BookNotBorrow。

可借阅和不可借阅的书作为图书馆中的书，都继承了书的基本属性和方法，具体代码如下：

动手写9.6.1

```
01 package main
02
03 import "fmt"
04
05 type Book struct {
06     title string
07     author string
08     num int
09     id int
10 }
11
12 type BookBorrow struct
13 {
14     Book
15     borrowTime string
16 }
17
18 type BookNotBorrow struct
19 {
20     Book
21     readTime string
22 }
23
24 func main () {
25     bookBorrow := &BookBorrow{}
26     bookNotBorrow := &BookNotBorrow{}
27     fmt.Println(bookBorrow)
28     fmt.Println(bookNotBorrow)
29 }
```

执行结果如下：

```
&{ { 0 0 } }
&{ { 0 0 } }
```

9.6.1 初始化结构体内嵌

结构体内嵌的初始化和结构体初始化类似，可以使用键值对或“.”的方式来进行初始化。

例如，已借阅书籍和未借阅书籍的初始化：

动手写9.6.2

```
01 package main
02
03 import "fmt"
04
05 type Book struct {
06     title string
07     author string
08     num int
09     id int
10 }
11
12 type BookBorrow struct {
13     Book
14     borrowTime string
15 }
16
17
18 type BookNotBorrow struct {
19     Book
20     readTime string
21 }
22
23
24 func main () {
25     bookBorrow := &BookBorrow{
26         Book:Book{ "Go语言",
27             "Tom",
28             20,
29             152368,
30         },
31 }
```

```

31         borrowTime:"30",
32     }
33     fmt.Println(bookBorrow)
34     bookNotBorrow := &BookNotBorrow{
35     }
36     bookNotBorrow.title="Python语言"
37     bookNotBorrow.author="Peter"
38     bookNotBorrow.num=10
39     bookNotBorrow.id=152369
40     bookNotBorrow.readTime="50"
41     fmt.Println(bookNotBorrow)
42 }
```

执行结果如下：

```

&{{Go语言 Tom 20 152368} 30}
&{{Python语言 Peter 10 152369} 50}
```

9.6.2 内嵌匿名结构体

在部分场景下，我们会使用匿名结构体来代替普通的结构体，以使代码的编写更加便利。

在定义匿名结构体时，无须type关键字，但是在初始化被嵌入的匿名结构体时，需要再次声明结构体才能赋予数据。

动手写9.6.3

```

01 package main
02
03 import "fmt"
04
05 type BookBorrow struct
06 {
07     Book struct {           //内嵌匿名结构体
08         title string
09         author string
10         num int
11         id int
12     }
```

```
13     borrowTime string
14 }
15
16 func main () {
17     bookBorrow := &BookBorrow{
18         Book:struct{      //声明类型
19             title string
20             author string
21             num int
22             id int
23         }{    "Go语言",
24             "Tom",
25             20,
26             152368,
27         },
28         borrowTime:"30",
29     }
30     fmt.Println(bookBorrow)
31 }
```

执行结果如下：

```
&{ {Go语言 Tom 20 152368} 30 }
```

9.7 匿名结构体

9.7.1 匿名结构体定义与初始化

匿名结构体，顾名思义，即没有名字的结构体，与匿名函数类似。匿名结构体无须type关键字就可以直接使用，匿名结构体在创建的同时也要创建对象。

匿名结构体的初始化和使用更加简单，无须通过type关键字定义，且不用写出类型名称。

匿名结构体在初始化时需进行匿名结构体定义和成员变量初始化（可选），格式如下：

```
title: Go语言
结构体实例 := struct{
```

```
//匿名结构体定义
成员变量1 类型1
成员变量2 类型2
成员变量3 类型3
...
} {
    //成员变量初始化（可选）
    成员变量1:值1,
    成员变量2:值2,
    成员变量3:值3,
    ...
}
```

动手写9.7.1

```
01 package main
02
03 import "fmt"
04
05 type Book struct {
06     title string
07     author string
08     num int
09     id int
10 }
11 func main (){
12     book1 := struct{
13         title string
14         author string
15         num int
16         id int
17     }{
18         title:"Go语言",
19         author:"Tom",
20         num:20,
21         id:152368,
```

```
22      }
23      fmt.Println("title:",book1.title)
24      fmt.Println("author:",book1.author)
25      fmt.Println("num:",book1.num)
26      fmt.Println("id:",book1.id)
27 }
```

执行结果如下：

```
title: Go语言
author: Tom
num: 20
id: 152368
```

如果需要初始化多个不同的实例（书），那么会存在大量重复的代码。所以，实际开发过程中，不建议通过这种方式来使用结构体。

9.7.2 匿名结构体的应用

匿名结构体一般可用于组织全局变量、构建数据模板和解析JSON等。

例如，我们可以通过自定义匿名结构体将同一类的全局变量组织在一起。

```
var config struct {
    IP      string
    Port    int
}

config.IP = "192.168.1.1"
config.Port = 8080
```

另外，我们经常会使用匿名结构体来临时存储经过解析后的JSON数据：

动手写9.7.2

```
01 package main
02
03 import (
04     "encoding/json"
05     "fmt"
06 )
```

```

07
08 func main() {
09     data := &struct {
10         Code int
11         Msg  string
12     }{}
13     jsonData := `{"code":200,"msg":""}`
14     if err := json.Unmarshal([]byte(jsonData), data); err != nil {
15         fmt.Println(err)
16     }
17     fmt.Println("code:", data.Code)
18     fmt.Println("msg:", data.Msg)
19 }
```

执行结果如下：

```

code: 200
msg:
```

在以上例子中，由于json包的Unmarshal函数的第二个参数为JSON解析后传递的对象，因此必须传入对象引用而非值传递。

9.8 小结

- ◇ Go语言的结构体是一种对现实生活中实体的抽象，结构体由一系列成员变量构成，这些成员变量对应着实体不同的属性。
- ◇ 定义结构体后，需要对结构体进行实例化才能使用结构体的字段。Go语言实例化结构体主要有标准实例化、new函数实例化、取地址实例化三种方式。
- ◇ Go语言中可以通过键值对格式和列表格式对结构体进行初始化。

9.9 知识拓展

使用结构体解析XML文件

XML即可扩展标记语言，标准通用标记语言的子集，是一种用于标记电子文件使其具有结构性的标记语言。

在计算机中，标记指计算机所能理解的信息符号，通过此类标记，计算机之间可以处理包含各种信息的文章等。它可以用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。它非常适合万维网传输，提供统一的方法来描述和交换独立于应用程序或供应商的结构化数据。它是网络环境中跨平台的、依赖于内容的技术，也是当今处理分布式结构信息的有效工具。早在1998年，W3C就发布了XML 1.0规范，用来简化网络间的文档信息传输。

XML本质上是一种树形的数据格式，我们可以定义与之匹配的Go语言的struct类型，然后通过xml包对XML文件中的数据进行解析，存储到自定义的结构体中。

我们先创建test.xml文件，文件内容如下，用于进行解析：

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <Persons>
03   <Person>
04     <Name>小张</Name>
05     <Age>25</Age>
06     <Interests>
07       <Interest>看书</Interest>
08       <Interest>游泳</Interest>
09     </Interests>
10   </Person>
11   <Person>
12     <Name>小李</Name>
13     <Age>32</Age>
14     <Interests>
15       <Interest>听音乐</Interest>
16       <Interest>看电影</Interest>
17     </Interests>
18   </Person>
19 </Persons>
```

根据XML文件中的内容，自定义结构体如下。必须注意的是，解析的时候，tag、字段名、XML元素都是对大小写敏感的，所以必须一一对应字段：

```
type Result struct {
    Person []Person
}
```

```
type Person struct {
    Name      string
    Age       string
    Interests Interests
}

type Interests struct {
    Interest []string
}
```

使用ioutil包中的ReadFile方法，将源代码所在目录下的test.xml文件内容读入content变量，类型为字节数组类型：

```
content, err := ioutil.ReadFile("test.xml")
```

之后使用xml包中的Unmarshal方法解析XML文件内容：

```
err = xml.Unmarshal(content, res)
```

程序源代码如下，最后打印所解析的XML文件内容：

动手写9.9.1

```
01 package main
02
03 import (
04     "encoding/xml"
05     "fmt"
06     "io/ioutil"
07 )
08
09 type Result struct {
10     Person []Person
11 }
12
13 type Person struct {
14     Name      string
15     Age       string
}
```

```
16     Interests Interests
17 }
18
19 type Interests struct {
20     Interest []string
21 }
22
23 func main() {
24     var res Result
25     content, err := ioutil.ReadFile("test.xml")
26     if err != nil{
27         fmt.Println(err)
28         return
29     }
30     err = xml.Unmarshal(content, &res)
31     if err != nil{
32         fmt.Println(err)
33     }
34     fmt.Println("XML文件解析后内容为: ")
35     fmt.Println(res)
36 }
```

执行结果如下：

```
XML文件解析后内容为:
{{{"小张": 25, {"看书": "游泳"}}, {"小李": 32, {"听音乐": "看电影"}}}}
```

Go语言中是利用反射机制将来自XML文件中的数据反射成对应的struct对象的，其中缺少的元素或空属性值将被解析为零值。

错误处理

我们在编写程序时，为了加强程序的健壮性，往往会影响到对程序中可能出现的错误和异常进行处理。

Go语言设计者认为类似try–catch–finally的传统异常处理机制很容易造成开发者对异常机制的滥用，从而使代码结构变得混乱。因此，在Go语言中会使用多值返回来返回错误。这种检查错误的方式给程序员提供了很大的控制权。

10.1 错误处理的方式

我们首先来观察下ioutil包中的ReadFile方法，代码如下：

```
func ReadFile(filename string) ([]byte, error) {
    f, err := os.Open(filename)
    if err != nil {
        return nil, err
    }
    defer f.Close()
    var n int64 = bytes.MinRead
    if fi, err := f.Stat(); err == nil {
        if size := fi.Size() + bytes.MinRead; size > n {
            n = size
        }
    }
    return readAll(f, n)
}
```

我们通常使用该函数来打开指定路径下的文件。ReadFile方法包含两个返回值，类型分别为[]byte和error。我们通过error类型的返回值来判断方法调用是否产生错误，若值为nil，则调用未产生错误。

打开文件的代码如下：

动手写10.1.1

```
01 package main
02
03 import (
04     "io/ioutil"
05     "fmt"
06 )
07
08 func main() {
09     f, err := ioutil.ReadFile("test.txt")
10     if err != nil{
11         fmt.Println(err)
12     }else {
13         fmt.Println(f)
14     }
15 }
```

执行结果如下：

```
open test.txt: The system cannot find the file specified.
```

由于源代码所在文件夹下没有test.txt文件，系统未找到指定的文件，因此使用ReadFile函数读取文件失败。

这也是Go语言编程中最常见的错误处理方式之一。

10.2 自定义错误

10.2.1 错误类型

Go语言的error类型实际上是一个接口，定义如下：

```
type error interface {
    Error() string
}
```

error接口包含Error方法，用来返回一个字符串。换言之，所有符合Error(string)格式的方法，都能实现错误接口。

10.2.2 创建错误

Go语言标准库中的errors包可以用来创建错误，自定义错误内容，格式如下：

```
err := errors.New("this is an error")
```

我们使用errors包的New函数来创建自定义错误，代码如下：

动手写10.2.1

```
01 package main
02
03 import (
04     "fmt"
05     "errors"
06 )
07
08 func main() {
09     err := errors.New("this is an error")
10     var err2 error
11     fmt.Println(err.Error())
12     fmt.Println(err2)
13 }
```

执行结果如下：

```
this is an error
<nil>
```

从执行结果可知：error的类型变量在声明后其默认值为nil。

10.2.3 自定义错误格式

标准库中的fmt包提供了Errorf方法，用来设置返回错误内容的格式，其源码如下：

```
func Errorf(format string, a ...interface{}) error {
    return errors.New(Sprintf(format, a...))
}
```

我们可以发现Errorf方法本质上还是调用了errors包的New方法来创建了一个错误，并使用Sprintf方法来对错误进行格式化。

动手写10.2.2

```
01 package main
02
03 import (
04     "fmt"
05     "runtime"
06 )
07
08 func main() {
09     if _, _, line, ok := runtime.Caller(0); ok == true { // 错误所在位置
10         err := fmt.Errorf("****Line %d error****", line)
11         fmt.Println(err.Error())
12     }
13 }
```

执行结果如下：

```
****Line 9 error****
```

在程序中，我们通过调用runtime包的Caller方法获取当前代码所在行数，用来模拟错误的产生。

10.3 Go语言宕机

一般而言，只有当程序发生不可逆的错误时，才会使用panic方法来触发宕机。panic方法是Go语言的一个内置函数，使用panic方法后，程序的执行将直接中断。

panic方法的源代码如下，由于其参数为空接口类型，因此我们可以传入任意类型的值作为宕机内容：

```
func panic(v interface{})
```

动手写10.3.1

```
01 package main
02
03 import "fmt"
```

```

04
05 func main() {
06     panic("Serious bug")
07     fmt.Println("Invalid code") //程序退出，无法执行该行代码
08 }

```

执行结果如下：

```

panic: Serious bug

goroutine 1 [running]:
main.main()
    F:/goland/mygo/src/test/test.go:6 +0x40

Process finished with exit code 2

```

调用panic方法后，之后的代码都是无效代码，因为程序会直接退出。

如果遇到以下情形，可以调用panic方法来退出程序：

- ◇ 程序处于失控状态且无法恢复，继续执行将会影响其他正常程序，引发操作系统异常甚至是死机。
- ◇ 发生不可预知的错误。

10.4

宕机恢复

10.4.1 recover捕获宕机

Go语言通过内置函数recover来捕获宕机，类似于其他编程语言中的try-catch机制。

在使用panic方法触发宕机后，且在退出当前函数前，会调用延迟执行语句defer，代码示例如下：

动手写10.4.1

```

01 package main
02
03 import "fmt"
04
05 func protect() {
06     defer func() { //protect函数退出前执行defer语句

```

```
07         fmt.Println("func protect exit")
08     }()
09     panic("Serious bug")           //触发宕机
10 }
11
12 func main(){
13     defer func() {               //protect函数退出前执行defer语句
14         fmt.Println("func main exit")
15     }()
16     protect()
17     fmt.Println("Invalid code")
18 }
```

执行结果如下：

```
func protect exit
func main exit
panic: Serious bug

goroutine 1 [running]:
main.protect()
    F:/goland/mygo/src/test/test.go:9 +0x5c
main.main()
    F:/goland/mygo/src/test/test.go:16 +0x49

Process finished with exit code 2
```

以上程序流程如下：

1. protect函数内的panic方法触发宕机。
2. 由于protect函数内的匿名函数通过defer语句延迟执行，在panic方法触发宕机后，且在退出protect函数前，会执行protect函数中的匿名函数，打印“func protect exit”。
3. 由于main函数内的匿名函数通过defer语句延迟执行，在main函数退出前会执行main函数中的匿名函数，打印“func main exit”。
4. 程序退出。

由于defer语句延迟执行的特性，我们可以通过“defer语句+匿名函数+recover方法”来完成对宕机的捕获。

动手写10.4.2

```

01 package main
02
03 import "fmt"
04
05 func protect() {
06     defer func() {
07         if err := recover(); err != nil { //recover()获取panic()传
08             fmt.Println(err)
09         }
10     }()
11     panic("Serious bug")
12 }
13
14 func main() {
15     protect()
16     fmt.Println("valid code")
17 }
```

执行结果如下：

```

 Serious bug
 valid code
```

以上程序通过recover方法获取到panic()传入的参数，进行打印展示，程序从宕机点退出当前函数后继续执行。

10.4.2 recover应用

在实际编程中，我们会专门封装一个函数，以一种安全模式来运行所有传入的方法，核心思想与动手写10.4.2类似，代码如下：

动手写10.4.3

```

01 package main
02
03 import "fmt"
04
05 func protect(f func()) { //以安全模式来运行所有传入的方法
```

```
06     defer func() {
07         if err := recover(); err != nil{
08             fmt.Println(err)
09         }
10     }()
11     f()
12 }
13
14 func main(){
15     protect(func() {           //模拟函数1
16         fmt.Println("This is function 1")
17         panic("Serious bug from function 1")
18     })
19     protect(func() {           //模拟函数2
20         fmt.Println("This is function 2")
21         panic("Serious bug function 2")
22     })
23     fmt.Println("valid code")
24 }
```

执行结果如下：

```
This is function 1
Serious bug from function 1
This is function 2
Serious bug function 2
valid code
```

protect函数用于以安全模式运行所有传入的匿名函数或闭包后的执行函数，即使发生panic，主程序仍然能继续正常运行。

10.5 Go语言错误的应用

10.4.2小节中的panic()和recover()虽然能模拟其他语言的异常机制，但是并不建议在Go语言编程中使用类似的方法。正如本章开头所提到的，推荐使用多值返回来返回错误。

动手写 10.5.1

```

01 package main
02
03 import (
04     "fmt"
05     "errors"
06 )
07
08 func div(dividend int, divisor int) (int, error) {
09     if divisor == 0{    //除数为0则返回错误
10         return 0,errors.New("divisor is zero")
11     }
12     return dividend/divisor,nil
13 }
14
15 func main(){
16     res1,err:= div(4,2);if err != nil {
17         fmt.Println(err.Error())
18     }else { fmt.Println("4/2 =",res1) }
19     res2,err := div(1,0);if err != nil{
20         fmt.Println(err.Error())
21     }else { fmt.Println("1/0 =",res2) }
22 }
```

执行结果如下：

```

4/2 = 2
divisor is zero
```

10.6 小结

- ◇ 在 Go 语言中使用多值返回来返回错误。
- ◇ Go 语言的 error 类型实际上是一个接口，error 接口包含 Error 方法，用来返回一个字符串。
- ◇ Go 语言标准库中的 errors 包可以用来创建错误，自定义错误内容。

- ◇ 标准库中的fmt包提供了Errorf方法，用来设置返回错误内容的格式。
- ◇ 一般而言，只有当程序发生不可逆的错误时，才会使用panic方法来触发宕机。
- ◇ Go语言通过内置函数recover来捕获宕机，类似于其他编程语言中的try-catch机制。

10.7 知识拓展

error接口的应用

在“10.2自定义错误”一节中，我们通过Go语言标准库中errors包的New方法来创建自定义错误。那么，除了通过这种方法，是否还可通过其他方法来创建自定义错误呢？答案是肯定的。

Go语言的error类型实际上是一个接口，定义如下：

```
type error interface {
    Error() string
}
```

那么，只要我们自定义一个类型，且该类型实现了Error方法，它就实现了error接口。

以计算一个数的算术平方根为例，被开方数必须是非负数，自定义类型如下：

```
type ErrNegSqrt float64
```

需实现Error函数，返回自定义错误信息：

```
func (e ErrNegSqrt) Error() string{
    return fmt.Sprintf("cannot Sqrt negative number: %v", float64(e))
}
```

计算算术平方根函数如下，对于被开方数为负数的情况抛出错误：

```
func sqrt(x float64) (float64, error) {
    if x<0.0{
        return 0, ErrNegSqrt(x)
    }else {
        return math.Sqrt(x), nil
    }
}
```

程序源代码如下：

动手写 10.7.1

```
01 package main
02
03 import (
04     "fmt"
05     "math"
06 )
07
08 type ErrNegSqrt float64
09
10 func (e ErrNegSqrt) Error() string{
11     return fmt.Sprintf("cannot Sqrt negative number: %v",float64(e))
12 }
13
14 func sqrt(x float64) (float64, error) {
15     if x<0.0{
16         return 0, ErrNegSqrt(x)
17     }else {
18         return math.Sqrt(x),nil
19     }
20 }
21
22 func main() {
23     res ,err := sqrt(-2)
24     if err !=nil{
25         fmt.Println(err)
26     }else {
27         fmt.Println(res)
28     }
29 }
```

执行结果如下：

```
cannot Sqrt negative number: -2
```

>> 第 11 章

文件操作 <<

计算机文件是以硬盘为载体的信息存储集合，文件可以是文本、图片、程序等。在编写程序时，我们经常会和文件打交道，比如从文件读取信息，保存程序结果、程序状态等。实际上，在前面的章节中，我们已经多次使用了文件，如源代码文件、编译后的可执行文件。从现在开始，我们将要学习如何使用程序去操作处理文件。

文件通常被分为两类：文本文件和二进制文件。所有你能用记事本打开并正常显示的文件都可以叫作文本文件，而像图片、可执行程序、压缩包等文件叫作二进制文件。

11.1

目录基本操作

计算机系统为了更好地管理文件，便使用了目录。目录是存放文件的地方，为树形层次结构。目录在不同系统中存在部分差异，操作系统主要分为Windows和Unix两类，我们日常使用的便是Windows系统。

Windows文件系统的特点是分盘，每个盘符有自己的根目录，形成多个树形并排结构，也就是我们常说的C盘、D盘等。Unix类文件系统只有一个根目录，也就是“/”。需要注意的是，Windows的目录不区分大小写，也就是说，“C:\windows”和“C:\WINDOWS”表示的是同一个目录，而Unix系统是区分大小写的。

提示

Unix系统是1969年AT&T公司在美国新泽西州开发的，大多数的操作系统都受到了Unix的启发，如苹果操作系统Mac OS和开源的Linux系统都是基于Unix发展起来的。

11.1.1 列目录

计算机的文件处理主要会用到Go语言提供的I/O操作库，为了方便开发者使用，Go语言将I/O操作封装在了如下几个包中：

- ◇ io——为IO原语（I/O primitives）提供基本的接口。
- ◇ io/ioutil——封装一些实用的I/O函数。
- ◇ fmt——实现格式化I/O，类似于C语言中的printf和scanf。
- ◇ bufio——实现带缓冲I/O。

我们最常用的I/O操作库是“fmt”，即控制台的输入输出，基本每个程序都离不开这个库，相信大家通过前面章节的学习已经能熟练使用它了。

本章重点介绍“io/ioutil”这个库，虽然io包提供了不少函数方法，但有时候使用起来并不是那么方便。为此，标准库中提供了一些更加常用、方便的I/O操作函数，将其封装在“io/ioutil”这个库中。

提示

I/O（Input/Output），即输入、输出操作。我们编写的程序除了自身会定义一些数据信息外，经常还会引用外界的数据，或是将自身的数据发送到外界。比如我们编写的程序想读取一个文本文件，又或者我们想将程序中的某些数据写入到一个文件中，这时我们就要使用输入与输出。

想要读取一个目录的内容，可以使用io/ioutil库中的ReadDir方法，此方法返回一个有序列表，列表内容是dirname指定的目录的目录信息。

```
func ReadDir(dirname string) ([]os.FileInfo, error)
```

我们尝试使用这个方法获取“C:\Users”（Windows系统使用“\”作为路径分隔符，“\\”表示字符串中的转义）目录下的所有文件信息，并打印出文件名（包括文件夹名）。

动手写11.1.1

```
01 package main
02
03 import (
04     "io/ioutil"
05     "fmt"
06 )
07
08 /* *
```

```
09  * ReadDir
10  * @author 零壹快学
11  */
12
13 func main() {
14     dir, err := ioutil.ReadDir("C:\\\\Users")
15     if err != nil {
16         fmt.Println(err)
17     }
18     for _, file := range dir {
19         fmt.Println(file.Name())
20     }
21 }
```

运行结果如下：

```
All Users
Default
Default User
Public
desktop.ini
admin
```

动手写11.1.1获取了“C:\\\\Users”目录下的所有文件名，但此时我们并不知道哪些是文件，哪些是目录。我们看到函数ReadDir的返回信息是“[]os.FileInfo”，也就是os.FileInfo的一个数组。而os.FileInfo的定义可以在源码中找到：

```
type FileInfo interface {
    Name() string           // 文件名称
    Size() int64             // 文件的长度（单位字节）
    Mode() FileMode          // 文件打开模式
    ModTime() time.Time      // 文件的修改时间
    IsDir() bool              // 是否是文件夹
    Sys() interface{}        // 基础数据源
}
```

通过os.FileInfo的IsDir属性就可以判断出该文件是否是一个文件夹。我们可以将列目录这个操作封装成一个函数。

动手写 11.1.2

```
01 package main
02
03 import (
04     "io/ioutil"
05     "fmt"
06 )
07
08 /**
09 * ListDir
10 * @author 零壹快学
11 */
12
13 // 获取指定目录下的所有文件，不进入下一级目录搜索
14 func ListDir(dirPth string) error {
15     dir, err := ioutil.ReadDir(dirPth)
16     if err != nil {
17         return err
18     }
19     for _, fi := range dir {
20         if fi.IsDir() { // 忽略目录
21             fmt.Println("目录: " + fi.Name())
22         } else {
23             fmt.Println("文件: " + fi.Name())
24         }
25     }
26     return nil
27 }
28
29 func main() {
30     ListDir("C:\\\\Users")
31 }
```

运行结果如下：

```
文件: All Users
目录: Default
文件: Default User
目录: Public
文件: desktop.ini
目录: admin
```

有时候可能需要获取一个目录下的所有文件名（包含子文件夹内的所有文件），也就是递归遍历一个文件夹内的所有文件。Go标准库提供了一个更加方便的操作函数，这个函数位于path/filepath库中。

```
func Walk(root string, walkFn WalkFunc) error
```

Walk函数会遍历root指定的目录下的文件树，对每一个该文件树中的目录和文件都会调用walkFn，包括root自身。所有访问文件/目录时遇到的错误都会传递给walkFn过滤。文件是按词法顺序遍历的，这让输出显得更漂亮，但也导致处理非常大的目录时效率会降低。Walk函数不会遍历文件树中的符号链接（快捷方式）文件包含的路径。

动手写11.1.3

```
01 package main
02
03 import (
04     "path/filepath"
05     "os"
06     "fmt"
07 )
08
09 /**
10 * WalkDir
11 * @author 零壹快学
12 */
13
14 //获取指定目录及所有子目录下的所有文件
15 func WalkDir(path string) {
16     err := filepath.Walk(path, func(path string, f os.FileInfo,
17         err error) error {
```

```

17         if f == nil {
18             return err
19         }
20         if f.IsDir() {
21             return nil
22         }
23         println(path)
24         return nil
25     })
26     if err != nil {
27         fmt.Printf("filepath.Walk() returned %v\n", err)
28     }
29 }
30
31 func main() {
32     WalkDir("C:\\Windows\\Temp")
33 }
```

运行结果如下：

```
C:\\Windows\\Temp\\log.txt
C:\\Windows\\Temp\\vmware-SYSTEM\\vmauthd.log
C:\\Windows\\Temp\\vmware-SYSTEM\\vmware-usbarb-3564.log
```

11.1.2 创建目录

Go标准库中的os库提供了平台无关性的操作系统功能接口。创建目录时可以使用os库的如下接口：

```
func Mkdir(name string, perm FileMode) error
```

Mkdir接口函数使用指定的权限和名称创建一个目录。如果出错，会返回*PathError底层类型（即语言自带的类型）的错误。

动手写11.1.4

```

01 package main
02
03 import (
```

```
04     "os"
05     "fmt"
06 )
07
08 /**
09 * 创建目录
10 * @author 零壹快学
11 */
12
13 func createDir(path string, dirName string) {
14     dirPath := path + "\\\" + dirName
15     //0777也可以os.ModePerm
16     err := os.Mkdir(dirPath, 0777)
17     if err != nil {
18         fmt.Println(err)
19     }
20     os.Chmod(dirPath, 0777)
21     fmt.Println("Create Dir => " + path + dirName)
22 }
23
24 func main() {
25     createDir("C:\\Users", "test")
26     createDir("C:\\Windows\\Temp", "test")
27 }
```

运行结果如下：

```
Create Dir => C:\\Users\\test
mkdir C:\\Users\\test: Access is denied.
Create Dir => C:\\Windows\\Temp\\test
Create Success!
```

可以发现，当没有足够的权限去创建一个文件夹时，会报出“Access is denied”的错误，提示我们权限不足，导致文件夹创建失败。我们再次运行这段代码后，运行结果如下：

```
Create Dir => C:\\Users\\test
mkdir C:\\Users\\test: Access is denied.
```

```
Create Dir => C:\Windows\Temp\test
mkdir C:\Windows\Temp\test: Cannot create a file when that file already
exists.
```

当“C:\Windows\Temp\test”文件夹存在时，再次调用Mkdir接口函数会抛出“文件夹已存在”的错误。Mkdir一次只能创建一级目录，如果使用此函数创建多级目录则会报“The system cannot find the path specified.”的错误。这时只能使用另一个创建目录的接口函数：

```
func MkdirAll(path string, perm FileMode) error
```

MkdirAll使用指定的权限和名称创建一个目录，包括任何必要的上级目录，并返回nil，否则返回错误。例如在目录“C:\Windows\Temp\”创建目录“C:\Windows\Temp\dir1\dir2\dir3”：

动手写11.1.5

```
01 package main
02
03 import (
04     "os"
05     "fmt"
06 )
07
08 /**
09 * 递归创建目录
10 * @author 零壹快学
11 */
12
13 func createDirAll(path string, dirName string)  {
14     dirPath := path + "\\\" + dirName
15     fmt.Println("Create Dir => " + dirPath)
16     err := os.MkdirAll(dirPath, 0777)
17     if err != nil {
18         fmt.Println(err)
19     } else {
20         fmt.Println("Create Success!")
21     }
22     os.Chmod(dirPath, 0777)
23 }
```

```
24 func main() {
25     createDirAll("C:\\Windows\\Temp", "dir1\\dir2\\dir3")
26 }
```

运行结果如下：

```
Create Dir => C:\\Windows\\Temp\\dir1\\dir2\\dir3
Create Success!
```

再次运行动手写11.1.5，运行结果如下：

```
Create Dir => C:\\Windows\\Temp\\dir1\\dir2\\dir3
Create Success!
```

使用MkdirAll创建一个目录path，当path已经存在时，不会报错，直接返回nil。这一点与Mkdir有一定的区别，需要注意。

11.1.3 删除目录

删除目录可以使用如下接口：

```
func Remove(name string) error
```

Remove删除name指定的文件或目录。此接口只能删除空文件夹，如果文件夹非空，则会删除失败，返回“文件夹非空”错误。

动手写11.1.6

```
01 package main
02
03 import (
04     "os"
05     "fmt"
06 )
07
08 /**
09 * 删除空目录
10 * @author 零壹快学
11 */
12
13 func deleteEmptyDir(dirPath string) {
```

```

14     fmt.Println("Delete Dir => " + dirPath)
15     err := os.Remove(dirPath)
16     if err != nil {
17         fmt.Println(err)
18     } else {
19         fmt.Println("Delete Success!")
20     }
21 }
22
23 func main() {
24     deleteEmptyDir("C:\\Windows\\Temp\\test")
25 }
```

运行结果如下：

```

Delete Dir => C:\\Windows\\Temp\\test
Delete Success!
```

如果文件夹非空，运行结果如下：

```

Delete Dir => C:\\Windows\\Temp\\test
remove C:\\Windows\\Temp\\test: The directory is not empty.
```

对于非空文件夹，可以使用以下接口删除：

```
func RemoveAll(path string) error
```

动手写 11.1.7

```

01 package main
02
03 import (
04     "os"
05     "fmt"
06 )
07
08 /**
09 * 删除非空目录
10 * @author 零壹快学
11 */
```

```
12 func deleteNotEmptyDir(dirPath string) {
13     fmt.Println("Delete Dir => " + dirPath)
14     err := os.RemoveAll(dirPath)
15     if err != nil {
16         fmt.Println(err)
17     } else {
18         fmt.Println("Delete Success!")
19     }
20 }
21
22 func main() {
23     deleteNotEmptyDir("C:\\Windows\\Temp\\dir1")
24 }
```

运行结果如下：

```
Delete Dir => C:\\Windows\\Temp\\dir1
Delete Success!
```

11.2 文件基本操作

在学习文件操作之前，我们先来了解一下Linux下的文件权限。文件有三种权限，分别为读取、写入和执行，对应字母为r、w、x。

Linux下权限的粒度有拥有者、所属组、其他组三种。每个文件都可以针对三种粒度，设置不同的rwx（读、写、执行）权限。通常情况下，一个文件只能归属于一个用户和组，如果其他的用户想拥有这个文件的权限，则可以将该用户加入具备权限的群组，一个用户可以同时归属于多个组。

表11.2.1 文件权限

权限	权限数值	具体作用
r	4	read, 读取。当前用户可以读取文件内容；当前用户可以浏览目录
w	2	write, 写入。当前用户可以新增或修改文件内容；当前用户可以删除、移动目录或目录内文件
x	1	execute, 执行。当前用户可以执行文件；当前用户可以进入目录

在创建一个文件或文件夹时，我们通常会赋予这个文件一定的权限。例如777权限就代表给这个文件的拥有者、所属组、其他用户赋予的读写执行权限。

11.2.1 文件创建与打开

对于文件的创建与打开，使用的是标准库os中的OpenFile。

```
func OpenFile(name string, flag int, perm FileMode) (file *File, err error)
```

OpenFile是一个更底层的文件打开函数，大多数调用者都应用Open或Create代替本函数。它会使用指定的选项（如O_RDONLY等）、指定的模式（如0666等）打开指定名称的文件。如果操作成功，返回的文件对象可用于I/O。如果出错，错误底层类型是*PathError。

位掩码参数flag用于指定文件的访问模式，可用的值在os中定义为常量（以下值并非所有操作系统都可用）：

```
const (
    O_RDONLY int = syscall.O_RDONLY      // 只读模式打开文件
    O_WRONLY int = syscall.O_WRONLY      // 只写模式打开文件
    O_RDWR    int = syscall.O_RDWR       // 读写模式打开文件
    O_APPEND  int = syscall.O_APPEND     // 写操作时将数据附加到文件尾部
    O_CREATE  int = syscall.O_CREAT      // 如果不存在将创建一个新文件
    O_EXCL    int = syscall.O_EXCL       // 和O_CREATE配合使用，文件必须不存在
    O_SYNC    int = syscall.O_SYNC       // 打开文件用于同步I/O
    O_TRUNC   int = syscall.O_TRUNC      // 如果可能，打开时清空文件
)
```

其中，O_RDONLY、O_WRONLY、O_RDWR应该只指定一个，剩下的通过“|”操作符来指定。该函数内部会给flags加上syscall.O_CLOEXEC，在fork子进程时会关闭通过OpenFile打开的文件，即子进程不会重用该文件描述符。

提示

由于历史原因，O_RDONLY | O_WRONLY并非等于O_RDWR，它们的值一般是0、1、2。

位掩码参数perm指定了文件的模式和权限位，类型是os.FileMode，文件模式位常量定义在os中：

```
const (
    // 单字符是被 String 方法用于格式化的属性缩写
    ModeDir      FileMode = 1 << (32 - 1 - iota) // d: 目录
    ModeAppend   FileMode = iota + 1           // a: 只能写入, 且只能写入到末尾
    ModeExclusive FileMode = iota + 2           // l: 用于执行
    ModeTemporary FileMode = iota + 3           // t: 临时文件(非备份文件)
    ModeSymlink   FileMode = iota + 4           // L: 符号链接(不是快捷方式文件)
    ModeDevice    FileMode = iota + 5           // D: 设备
    ModeNamedPipe FileMode = iota + 6           // p: 命名管道(FIFO)
    ModeSocket    FileMode = iota + 7           // S: Unix域socket
    ModeSetuid    FileMode = iota + 8           // u: 表示文件具有其创建者用户id权限
    ModeSetgid    FileMode = iota + 9           // g: 表示文件具有其创建者组id的权限
    ModeCharDevice FileMode = iota + 10          // c: 字符设备, 需已设置ModeDevice
    ModeSticky    FileMode = iota + 11          // t: 只有root/创建者能删除/移动文件

    // 覆盖所有类型位(用于通过&获取类型位), 对普通文件, 所有这些位都不应被设置
    ModeType = ModeDir | ModeSymlink | ModeNamedPipe | ModeSocket |
    ModeDevice
    ModePerm FileMode = 0777 // 覆盖所有Unix权限位(用于通过&获取类型位)
)
```

以上常量在所有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地移植。

动手写11.2.1

```
01 package main
02
03 import (
04     "os"
05     "fmt"
06 )
07
08 /**
09  * 打开文件
10  * @author 零壹快学
11 */
12
```

```

13 func main() {
14     //以读写方式打开文件，如果不存在，则创建
15     file, err := os.OpenFile("C:\\\\Windows\\\\Temp\\\\1.txt", os.O_
16         RDWR|os.O_CREATE, 0766)
17     if err != nil {
18         fmt.Println(err)
19     }
20     fmt.Println(file)
21     file.Close()
22 }
```

运行结果如下：

```
&{0xc042074780}
```

动手写11.2.1中，“os.O_RDWR|os.O_CREATE”表示以读写方式打开文件，如果文件不存在，则创建这个文件。大多数情况下，我们会使用如下两个函数来创建或打开文件。

```
func Open(name string) (file *File, err error)
```

Open打开一个文件用于读取。如果操作成功，返回的文件对象的方法可用于读取数据；对应的文件描述符具有O_RDONLY模式。如果出错，错误底层类型是*PathError。

```
func Create(name string) (file *File, err error)
```

Create采用模式0666（任何人都可读写，不可执行）创建一个名为name的文件，如果文件已存在就会截断它（为空文件）。如果成功，返回的文件对象可用于I/O；对应的文件描述符具有O_RDWR模式。如果出错，错误底层类型是*PathError。

Open和Create的函数定义如下：

```

func Open(name string) (*File, error) {
    return OpenFile(name, O_RDONLY, 0)
}

func Create(name string) (*File, error) {
    return OpenFile(name, O_RDWR|O_CREATE|O_TRUNC, 0666)
}
```

11.2.2 文件读取

想要读取文件可以使用os库中的Read接口：

```
func (f *File) Read(b []byte) (n int, err error)
```

Read方法从文件中读取最多len(b)字节数据并写入byte数组b，它返回读取的字节数和可能遇到的任何错误。文件终止标志是读取0个字节且返回值err为io.EOF。

动手写11.2.2

```
01 package main
02
03 import (
04     "os"
05     "fmt"
06 )
07
08 /**
09  * 读文件
10 * @author 零壹快学
11 */
12
13 func ReadFile(path string) {
14     //读取文件内容
15     file, err := os.Open(path)
16     if err != nil {
17         fmt.Println(err)
18     }
19     //创建byte的slice用于接收文件读取数据
20     buf := make([]byte, 1024)
21     fmt.Println("以下是文件内容：")
22     //循环读取
23     for {
24         //Read函数会改变文件当前偏移量
25         len, _ := file.Read(buf)
26         //读取字节数为0时跳出循环
27         if len == 0 {
28             break
29         }
}
```

```

30         fmt.Println(string(buf))
31     }
32     file.Close()
33 }
34
35 func main() {
36     ReadFile("C:\\Windows\\Temp\\1.txt")
37 }
```

运行结果如下：

以下是文件内容：

```
test1
test2
```

当遇到特别大的文件，并且只需要读取文件最后部分的内容时，Read接口就不能满足我们的需要了，这时可以使用另外一个文件读取接口ReadAt，这个接口可以指定从文件的什么位置开始读取。

```
func (f *File) ReadAt(b []byte, off int64) (n int, err error)
```

ReadAt从指定的位置（相对于文件开始位置）读取len(b)字节数据并写入byte数组b。它返回读取的字节数和可能遇到的任何错误。当n<len(b)时，本方法总是会返回错误；如果是因为到达文件结尾，返回值err会是io.EOF。

动手写11.2.3

```

01 package main
02
03 import (
04     "fmt"
05     "os"
06 )
07
08 /**
09  * 读文件
10  * @author 零壹快学
11 */
```

```
12
13 func ReadFile2(path string) {
14     //读取文件内容
15     file, err := os.Open(path)
16     if err != nil {
17         fmt.Println(err)
18     }
19     //创建byte的slice用于接收文件读取数据
20     buf := make([]byte, 1024)
21     fmt.Println("以下是文件内容: ")
22     // 读取偏移9字节的数据
23     _, _ = file.ReadAt(buf, 9)
24     fmt.Println(string(buf))
25     _ = file.Close()
26 }
27
28 func main() {
29     ReadFile2("C:\\Windows\\Temp\\1.txt")
30 }
```

动手写11.2.3指定从文件开始位置，即ix=0处开始读取，运行结果如下：

以下是文件内容：

第2行

第3行

第4行

 提示

Read和ReadAt的区别：前者从文件当前偏移量处开始读取，且会改变文件当前的偏移量；而后者从off指定的位置开始读取，且不会改变文件当前偏移量。

11.2.3 文件写入

与之前的文件读取相比，向文件写入内容也有两个接口，分别为Write和WriteAt。

```
func (f *File) Write(b []byte) (n int, err error)
```

Write向文件中写入len(b)字节数据。它返回写入的字节数和可能遇到的任何错误。如果返回值为n!=len(b)，本方法会返回一个非nil的错误。

动手写11.2.4

```

01 package main
02
03 import (
04     "os"
05     "fmt"
06 )
07
08 /**
09 * 写文件
10 * @author 零壹快学
11 */
12
13 func main() {
14     file, err := os.Create("C:\\Windows\\Temp\\1.txt")
15     if err != nil {
16         fmt.Println(err)
17     }
18     data := "我是数据\r\n"
19     for i := 0; i < 3; i++ {
20         //写入byte的slice数据
21         file.Write([]byte(data))
22     }
23     file.Close()
24 }
```

运行后，文件内容如下：

```

我是数据
我是数据
我是数据
```

使用WriteAt可以指定从文件的什么位置开始写：

```
func (f *File) WriteAt(b []byte, off int64) (n int, err error)
```

WriteAt在指定的位置（相对于文件开始位置）写入len(b)字节数据。它返回写入的字节数和可能遇到的任何错误。如果返回值为n!=len(b)，本方法会返回一个非nil的错误。

动手写11.2.5

```
01 package main
02
03 import (
04     "os"
05     "fmt"
06     "strconv"
07 )
08
09 /**
10 * 写文件
11 * @author 零壹快学
12 */
13
14 func main() {
15     file, err := os.Create("C:\\\\Windows\\\\Temp\\\\1.txt")
16     if err != nil {
17         fmt.Println(err)
18     }
19     for i := 0; i < 3; i++ {
20         //按指定偏移量写入数据
21         ix := i * 64
22         file.WriteAt([]byte("我是数据"+strconv.Itoa(i)+"\\r\\n"),
23                     int64(ix))
24     }
25 }
```

运行后，文件内容如下：

```
我是数据0
                    我是数据1
                    我是数据2
```

使用Write和WriteAt接口函数对文件进行写入数据时，会将原文件覆盖，并从文件开始位置处写入内容。

11.2.4 删 除文件

删除文件使用os库中的Remove和RemoveAll接口，与删除目录的接口一致。

动手写11.2.6

```
01 package main
02
03 import (
04     "os"
05     "fmt"
06 )
07
08 /**
09 * 删除文件
10 * @author 零壹快学
11 */
12
13 func main() {
14     //删除文件
15     err := os.Remove("C:\\Windows\\Temp\\1.txt")
16     if err != nil {
17         fmt.Println(err)
18     } else {
19         fmt.Println("删除成功! ")
20     }
21
22     //删除指定path下的所有文件
23     err = os.RemoveAll("C:\\Windows\\Temp\\test\\")
24     if err != nil {
25         fmt.Println(err)
26     } else {
27         fmt.Println("删除成功! ")
28     }
29 }
```

运行结果如下：

删除成功！

删除成功！

11.3 处理JSON文件

JSON (JavaScript Object Notation, JS对象简谱) 是一种轻量级的数据交换格式。JSON最初是属于JavaScript的一部分，后来由于其良好的可读性和便于快速编写的特性，现在已独立于语言，基本上所有的语言都支持JSON数据的编码和解码。特别是对于网络编程而言，JSON的重要性不言而喻。

JSON中的键都是字符串形式，值可以取任意类型。它有以下三种结构：

值为字符串或数组类型：{"name": "John", "age": 20}

JSON数组：[{"name": "John", "age": 20}, {"name": "Tom", "age": 21}]

值为对象类型：{"name": "John", "birthday": {"month": 8, "day": 26}}，类似于对象嵌套对象

其中，大括号 “{}” 用来描述一组“不同类型的无序键值对集合”，方括号 “[]” 用来描述一组“相同类型的有序数据集合”。

关于JSON的更多信息，可以访问JSON官方网址<http://json.org/>进行查阅。我们先来看一个JSON样例，后面的程序都是基于这个样例编写的。

```
{  
    "name": "小王",  
    "age": 24,  
    "sex": true,  
    "birthday": "1995-01-01",  
    "company": "零壹快学",  
    "language": [  
        "Go",  
        "PHP",  
        "Python"  
    ]  
}
```

11.3.1 编码JSON

标准库提供了encoding/json库来处理JSON。编码JSON，即从其他的数据类型编码成JSON字符串，这个过程我们会使用如下的接口：

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

Marshal函数返回interface{}类型的JSON编码，通常interface{}类型会使用map或者结构体。为了让输出的JSON字符串更加直观，可以使用另一个JSON编码接口，对输出的JSON进行格式化操作。

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

MarshalIndent类似于Marshal，但会使用缩进将输出格式化。我们可以使用map来创建最简单的JSON。

动手写11.3.1

```
01 package main
02
03 import (
04     "encoding/json"
05     "fmt"
06 )
07
08 /**
09  * 编码JSON-map
10  * @author 零壹快学
11 */
12
13 func main() {
14     //创建一个map
15     m := make(map[string]interface{}, 6)
16     m["name"] = "小王"
17     m["age"] = 24
18     m["sex"] = true
19     m["birthday"] = "1995-01-01"
20     m["company"] = "零壹快学"
21     m["language"] = []string{"Go", "PHP", "Python"}
22 }
```

```
23     //编码成JSON
24     result, _ := json.Marshal(m)
25     resultFormat, _ := json.MarshalIndent(m, "", "    ")
26
27     fmt.Println("result = ", string(result))
28     fmt.Println("resultFormat = ", string(resultFormat))
29 }
```

运行结果如下：

```
result = {"age":24,"birthday":"1995-01-01","company":"零壹快学",
"language":["Go","PHP","Python"], "name":"小王","sex":true}
resultFormat = {
    "age": 24,
    "birthday": "1995-01-01",
    "company": "零壹快学",
    "language": [
        "Go",
        "PHP",
        "Python"
    ],
    "name": "小王",
    "sex": true
}
```

大多数情况下，我们会使用struct结构体来进行快速的JSON编码、解码，特别是在JSON解码时，使用struct会相当方便。

动手写11.3.2

```
01 package main
02
03 import (
04     "encoding/json"
05     "fmt"
06 )
07
08 /**
```

```
09  * 编码JSON-struct
10  * @author 零壹快学
11  */
12
13 type Person struct {
14     Name          string      `json:"name"`
15     Age           int         `json:"age"`
16     Sex           bool        `json:"sex"`
17     Birthday      string      `json:"birthday"`
18     Company       string      `json:"company, omitempty"`
19     Language      []string    `json:"language"`
20 }
21
22 func main() {
23     //定义一个结构体变量
24     person := Person{"小王", 24, true, "1995-01-01", "零壹快学", []string{"Go", "PHP", "Python"}}
25
26     //result, err := json.Marshal(person)
27     result, err := json.MarshalIndent(person, "", "    ")
28     if err != nil {
29         fmt.Println(err)
30     }
31     fmt.Println("result = ", string(result))
32 }
```

运行结果如下：

```
result = {
    "name": "小王",
    "age": 24,
    "sex": true,
    "birthday": "1995-01-01",
    "company": "零壹快学",
```

```
"language": [
    "Go",
    "PHP",
    "Python"
]
}
```

在定义struct字段的时候，可以在字段后面添加标签来控制编码/解码的过程：是否要编码或解码某个字段，JSON中的字段名称是什么。可以选择的控制字段有三种：

1. `-`: 不要解析这个字段。
2. `omitempty`: 当字段为空（默认值）时，不要解析这个字段。比如`false`、`0`、`nil`、长度为0的`array`、`map`、`slice`、`string`。
3. `FieldName`: 当解析JSON的时候，使用这个名字。

在动手写11.3.2中，“`json:"name"`”就是定义的第三类标签，表示将`Name`属性的key值解析为`name`。

```
Name     string     `json:"name"`
```

11.3.2 解码JSON

解码JSON会使用到`Unmarshal`接口，也就是`Marshal`的反操作。

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

`Unmarshal`函数解析JSON编码的数据并将结果存入`v`指向的值。要将JSON数据解码写入一个接口类型值，函数会将数据解码为如下类型写入接口：

<code>Bool</code>	对应JSON布尔类型
<code>float64</code>	对应JSON数字类型
<code>string</code>	对应JSON字符串类型
<code>[]interface{}</code>	对应JSON数组
<code>map[string]interface{}</code>	对应JSON对象
<code>nil</code>	对应JSON的null

如果一个JSON值不匹配给出的目标类型，或者一个JSON数字写入目标类型时溢出，`Unmarshal`函数会跳过该字段并尽量完成其余的解码操作。我们首先来看使用`map`解码JSON。

动手写 11.3.3

```
01 package main
02
03 import (
04     "encoding/json"
05     "fmt"
06 )
07
08 /**
09  * 解码JSON-map
10  * @author 零壹快学
11 */
12
13 func main() {
14     jsonStr := `{
15     "name": "小王",
16     "age": 24,
17     "sex": true,
18     "birthday": "1995-01-01",
19     "company": "零壹快学",
20     "language": [
21         "Go",
22         "PHP",
23         "Python"
24     ]
25 }
26 }` // 创建一个map
27 m := make(map[string]interface{}, 6)
28 err := json.Unmarshal([]byte(jsonStr), &m)
29 if err != nil {
30     fmt.Println(err)
31 }
32
33 fmt.Println("m = ", m)
```

```
34
35     //类型断言
36     for key, value := range m {
37         switch data := value.(type) {
38             case string:
39                 fmt.Printf("map[%s]的值类型为string, value = %s\n",
40                             key, data)
41             case float64:
42                 fmt.Printf("map[%s]的值类型为int, value = %f\n",
43                             key, data)
44             case bool:
45                 fmt.Printf("map[%s]的值类型为bool, value = %t\n",
46                             key, data)
47             case []string:
48                 fmt.Printf("map[%s]的值类型为[]string, value = %v\n",
49                             key, data)
50             case []interface{}:
51                 fmt.Printf("map[%s]的值类型为[]interface{}, value =
52                             %v\n", key, data)
53         }
54     }
55 }
```

运行结果如下：

```
m = map[name:小王 age:24 sex:true birthday:1995-01-01 company:零壹快学
language:[Go PHP Python]]
map[language]的值类型为[]interface{}, value = [Go PHP Python]
map[name]的值类型为string, value = 小王
map[age]的值类型为int, value = 24.000000
map[sex]的值类型为bool, value = true
map[birthday]的值类型为string, value = 1995-01-01
map[company]的值类型为string, value = 零壹快学
```

使用map来解码JSON时，需要使用Go的类型断言（关于类型断言详见第12章“接口与类型”）。可见，使用map解码JSON需要对其类型进行判断，非常烦琐。我们再来看看使用struct来解码JSON，JSON库会自动对结构体的类型进行解析，无须类型判断。

动手写 11.3.4

```
01 package main
02
03 import (
04     "encoding/json"
05     "fmt"
06 )
07
08 /**
09  * 解码JSON-struct
10  * @author 零壹快学
11 */
12
13 type Person struct {
14     Name          string      `json:"name"`
15     Age           int         `json:"age"`
16     Sex           bool        `json:"sex"`
17     Birthday      string      `json:"birthday"`
18     Company       string      `json:"company"`
19     Language      []string    `json:"language"`
20 }
21
22 func main() {
23     jsonStr := `{
24     "name": "小王",
25     "age": 24,
26     "sex": true,
27     "birthday": "1995-01-01",
28     "company": "零壹快学",
29     "language": [
30         "Go",
31         "PHP",
32         "Python"
```

```
34      ]
35  }、
36  var person Person
37  err := json.Unmarshal([]byte(jsonStr), &person)
38  if err != nil {
39      fmt.Println(err)
40  }
41  fmt.Printf("person = %v", person)
42 }
```

运行结果如下：

```
person = {Name:小王 Age:24 Sex:true Birthday:1995-01-01 Company:零壹快
学 Language:[Go PHP Python] }
```

 提示

当我们需要对大量的JSON进行快速解析处理时，标准库提供的JSON处理库“encoding/json”可能就不能满足我们的需求，GitHub开源了一个比标准库解析速度快10倍的fastjson库（<https://github.com/vablyala/fastjson>），有兴趣的读者可以尝试操作。

11.4 小结

- ◇ 掌握对目录的新建、查看和删除。
- ◇ 掌握对文件的新建、查看和删除。
- ◇ 掌握如何进行JSON编码与解码。

在前面的章节中，我们学习了如何使用函数方法，而接口是描述了一系列方法的集合。本章我们将深入学习什么是接口，以及如何使用接口。在使用接口时，我们经常会遇到这样一个问题：如何判断一个接口的类型？这时，我们就需要学习使用Go语言的类型断言来解决问题了。

接口的使用是为了抽象，方便代码的解耦。本章内容是Go语言学习的重点，也是一个难点，掌握好了就能写出更加优质、简洁、可复用的代码。

12.1

接口定义

12.1.1 接口的定义

在Go语言中，接口（interface）是一个自定义类型。接口类型是一种抽象的类型，它不会暴露出它所代表的内部属性的结构，而只会展示出它自己的方法，因此，不能将接口类型实例化。

根据Go语言规范，单个函数的接口命名为函数名加上“er”作为后缀，例如Reader、Writer、Formatter等。接口命名规范如下：

- ◇ 单个函数的接口名以“er”作为后缀，接口的实现则去掉“er”。
- ◇ 两个函数的接口名综合两个函数名，以“er”作为后缀，接口的实现则去掉“er”。
- ◇ 三个以上函数的接口，抽象这个接口的功能，类似于结构体命名。

Go语言的接口是非常灵活的，它通过一种方式来声明对象的行为，谁实现了这些行为，就相当于实现了这个接口里面声明各种方法的集合，但接口本身不去实现这些方法所需要的一些操作，因为这些方法没有被实现，所以它们是抽象的方法。这就非常像其他语言中实现面向对象的抽象方法，只不过其他语言中需要先继承再实现相对应的方法，而Go不需要继承，只要在结构中

声明和实现了这些方法，也就相当于实现了这个抽象方法接口声明格式。

和定义结构体相似，定义一个接口还是使用type，这说明了接口也是一种类型，下面举一个接口的例子。

动手写12.1.1

```
01 type Animal interface {
02     Name() string
03     Speak() string
04 }
05
06 type Cat struct {
07 }
08
09 func (cat Cat)Name() string {
10     return "Cat"
11 }
12
13 func (cat Cat)Speak() string {
14     return "喵喵喵"
15 }
```

动手写12.1.1定义了一个叫Animal的接口，结构体Cat实现了Animal接口的两个方法，因此我们就可以认为，Cat实现了Animal接口。

提示

需要注意的是，与其他语言不同，Go语言的接口没有数据字段，只有定义的方法。

12.1.2 鸭子类型

在程序设计中，鸭子类型（duck typing）是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口来决定，而是由当前方法和属性的集合决定。

这个概念的名字来源于詹姆斯·惠特科姆·莱利（James Whitcomb Riley）提出的鸭子测试。

“鸭子测试”可以这样表述：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。

Go语言通过接口实现了鸭子类型，对于Go来说，我们不关心对象是什么类型，或者到底是不是鸭子，我们只关心行为，关心它是如何使用的。

提示

一般来说，静态类型语言在编译时便已确定了变量的类型，但是Go语言的实现是在编译时推断变量的类型。

12.2

接口的创建与实现

Go语言接口是方法的集合，使用接口是实现模块化的重要方式。本节将重点介绍如何创建和实现一个Go语言接口。

12.2.1 接口创建

接口是用来定义行为类型的。这些被定义的行为不由接口直接实现，而是由用户自定义的类型实现，一个实现了这些方法的具体类型就叫作这个接口类型的实例。

上一节也简单介绍了接口的定义，使用type和interface关键字即可创建一个接口。接口内部只需要声明存在什么方法，不需要实现这些方法。

```
type InterfaceName interface{
    Method()
}
```

比如，我们创建一个数据库操作的接口：

动手写12.2.1

```
01 type IDatabaser interface {
02     Connect() error
03     Disconnect() error
04 }
```

动手写12.2.1创建了一个叫IDatabaser的接口，这个接口声明了两个方法——Connect()和Disconnect()，只要哪个类型实现了这两个方法，就可以说该类型是IDatabaser接口的实例。

动手写12.2.2

```
01 // MySQL数据库操作
02 type Mysql struct {
03     DBName string
04     isConnect bool
05 }
06
07 func (mysql *Mysql) Connect() error {
08     fmt.Println("Mysql Connect DB => "+ mysql.DBName)
09
10    //do Connect
11    mysql.isConnect = true
12
13    if mysql.isConnect {
14        fmt.Println("Mysql Connect Success!")
15        return nil
16    } else {
17        return errors.New("Connect failure! ")
18    }
19 }
20
21 func (mysql *Mysql) Disconnect() error {
22    //do Disconnect
23    fmt.Println("Mysql Disconnect Success!")
24    return nil
25 }
```

动手写12.2.2创建了一个对MySQL数据库操作的类型，这个类型实现了IDatabaser接口中的两个方法，因此Mysql类型就是IDatabaser的一个实例。我们再来创建一个对Redis数据库操作的类型：

动手写12.2.3

```
01 // Redis数据库操作
02 type Redis struct {
03     DBName string
04 }
```

```

05 func (redis *Redis) Connect() error {
06     fmt.Println("Redis Connect DB => "+ redis.DBName)
07     //do Connect
08     fmt.Println("Redis Connect Success!")
09     return nil
10 }
11
12 func (redis *Redis) Disconnect() error {
13     //do Disconnect
14     fmt.Println("Redis Disconnect Success!")
15     return nil
16 }

```

这时候，我们就可以直接使用Mysql和Redis这两个类型来完成数据库操作：

动手写12.2.4

```

01 func main() {
02     var mysql = Mysql{DBName:"student"}
03     fmt.Println("开始连接")
04     mysql.Connect()
05     //do something
06     fmt.Println("断开连接")
07     mysql.Disconnect()
08
09     var redis = Redis{DBName:"teacher"}
10     fmt.Println("开始连接")
11     redis.Connect()
12     //do something
13     fmt.Println("断开连接")
14     redis.Disconnect()
15 }

```

运行结果如下：

```

开始连接
Mysql Connect DB => student
Mysql Connect Success!

```

断开连接

Mysql Disconnect Success!

开始连接

Redis Connect DB => teacher

Redis Connect Success!

断开连接

Redis Disconnect Success!

可以看到，我们虽然定义了接口，但是没有面向接口编程，所以动手写12.2.4会有很多重复的代码。数据库类型越多，重复的代码也会越多，当我们面向接口编程时，就可以抽象出更简洁、更易于管理的代码，例如动手写12.2.5。

动手写12.2.5

```
01 func HandleDB(db IDatabaser) {
02     fmt.Println("开始连接")
03     db.Connect()
04     //do something
05     fmt.Println("断开连接")
06     db.Disconnect()
07 }
08
09 func main() {
10     var mysql = Mysql{DBName:"student"}
11     HandleDB(&mysql)
12
13     var redis = Redis{DBName:"teacher"}
14     HandleDB(&redis)
15 }
```

运行结果如下：

开始连接

Mysql Connect DB => student

Mysql Connect Success!

断开连接

Mysql Disconnect Success!

开始连接

```
Redis Connect DB => teacher
```

```
Redis Connect Success!
```

断开连接

```
Redis Disconnect Success!
```

动手写12.2.5中，HandleDB()函数只有一个，却能实现处理多个不同类型的数据，这也称为Go的多态。

提示

Go语言中，一个类型可以实现多个接口，反之，多个类型也可以实现同一个接口，任何一个类型都必然实现了空接口。

12.2.2 接口赋值

如果用户自定义的类型实现了某个接口类型所声明的一组方法，那么这个用户定义的类型的值就可以赋值给这个接口。这个赋值会把用户定义的类型的值存入接口类型的值。

这里涉及接口变量，接口变量存储了两部分信息，一是分配给接口变量的具体值（接口实现者的值），二是值的类型的描述器（接口实现者的类型）。

简单来说，接口赋值存在以下两种情况：

1. 将对象实例赋值给接口；
2. 将一个接口赋值给另一个接口。

当一个对象的类型是一个接口的实例时，这个对象就可以赋值给这个接口。需要注意的是，只能将对象的指针赋值给接口变量，不能将对象值直接赋值给接口变量，否则就会发生错误。

动手写12.2.6

```
01 package main
02
03 import (
04     "fmt"
05 )
06
07 type IDatabase interface {
08     Connect() error
09     Disconnect() error
10 }
```

```
10 }
11
12 type Redis struct {
13     DBName string
14 }
15
16 func (redis *Redis) Connect() error {
17     fmt.Println("Redis Connect DB => "+ redis.DBName)
18     //do Connect
19     fmt.Println("Redis Connect Success!")
20     return nil
21 }
22
23
24 func (redis *Redis) Disconnect() error {
25     //do Disconnect
26     fmt.Println("Redis Disconnect Success!")
27     return nil
28 }
29
30 func main() {
31     var redis = Redis{DBName:"teacher"}
32     var idb IDatabaser = &redis
33     idb.Connect()
34     idb.Disconnect()
35 }
```

运行结果如下：

```
Redis Connect DB => teacher
Redis Connect Success!
Redis Disconnect Success!
```

对于第二种情况（通过一个接口给另一个接口赋值），在Go语言中只要两个接口拥有同样的方法集（次序不同不要紧），那么它们就是相同的，可以相互赋值。如果两个接口不是相同的，接口A的方法集是接口B方法集的子集，那么接口B可以赋值给接口A，反之则不成立。举个例子：

动手写12.2.7

```
01 package main
02
03 import (
04     "fmt"
05 )
06
07 type IDatabaser interface {
08     Connect() error
09     Disconnect() error
10 }
11
12 type IRediser interface {
13     Connect() error
14 }
15
16 type Redis struct {
17     DBName string
18 }
19
20 func (redis *Redis) Connect() error {
21     fmt.Println("Redis Connect DB => "+ redis.DBName)
22     //do Connect
23     fmt.Println("Redis Connect Success!")
24     return nil
25 }
26
27 func (redis *Redis) Disconnect() error {
28     //do Disconnect
29     fmt.Println("Redis Disconnect Success!")
30     return nil
31 }
32
33 func main() {
```

```
34     var idb IDatabaser = &Redis{DBName:"teacher"}  
35  
36     var iredis IRediser  
37     iredis = idb  
38     iredis.Connect()  
39 }
```

运行结果如下：

```
Redis Connect DB => teacher  
Redis Connect Success!
```

动手写12.2.7中，IRediser接口的方法集是IDatabaser接口方法集的子集，IDatabaser接口变量就可以直接赋值给IRediser接口变量。

12.3 接口嵌入

接口嵌入，也叫接口组合，在其他语言中，这种接口的组合叫作继承；Go语言舍弃了繁杂的继承体系，但继承这个特性还是通过接口嵌入得以实现了。接口嵌入也就是指如果一个接口interface1作为interface2的一个嵌入字段，那么interface2隐式包含了interface1里面所有的方法，例如动手写12.3.1。

动手写12.3.1

```
01 type interface1 interface {  
02     Method()  
03 }  
04 type interface2 interface {  
05     interface1  
06 }
```

接口嵌入同样体现了非侵入式的风格，关于非侵入式，可查看后面的章节。关于接口嵌套需要注意：首先，一个接口类型只接受其他的接口类型的嵌入，嵌入结构体或其他类型则会报错；其次，一个接口类型不能嵌入自身，这包括直接嵌入和间接嵌入。

直接嵌入如下：

动手写12.3.2

```

01 type interface1 interface {
02     interface1
03 }
```

间接嵌入如下：

动手写12.3.3

```

01 type interface1 interface {
02     interface2
03 }
04
05 type interface2 interface {
06     interface3
07 }
08
09 type interface3 interface {
10     interface1
11 }
```

动手写12.3.2和动手写12.3.3都会报“invalid recursive type”的错误。

我们来看一个完整的例子：

动手写12.3.4

```

01 package main
02
03 import "fmt"
04
05 type IPerson interface{
06     Speak()
07 }
08
09 type IStudent interface {
10     IPerson
11     Study()
12 }
13
```

```
14 type ITeacher interface {
15     IPerson
16     Teach()
17 }
18
19 type Student struct {
20     Name string
21 }
22
23 func (s *Student) Speak() {
24     fmt.Println("My name is", s.Name)
25 }
26
27 func (s *Student) Study() {
28     fmt.Println(s.Name, "is studying")
29 }
30
31 type Teacher struct {
32     Name string
33 }
34
35 func (t *Teacher) Speak() {
36     fmt.Println("My name is", t.Name)
37 }
38
39 func (t *Teacher) Teach() {
40     fmt.Println(t.Name, "is teaching")
41 }
42
43 func main() {
44     var stu Student = Student{"Xiao Ming"}
45     var teacher Teacher = Teacher{"Mr. Li"}
46
47     stu.Speak()
48     stu.Study()
49 }
```

```

50     teacher.Speak()
51     teacher.Teach()
52 }
```

动手写12.3.4定义了三个接口：IPerson、IStudent、ITeacher。其中IStudent和ITeacher都嵌入了IPerson接口，可以理解为IStudent和ITeacher都继承了IPerson接口，都拥有了IPerson接口中的Speak()方法；而结构体Teacher要实现ITeacher接口，则同时需要实现IPerson接口才行。动手写12.3.4的运行结果如下：

```

My name is Xiao Ming
Xiao Ming is studying
My name is Mr. Li
Mr. Li is teaching
```

12.4 空接口

12.4.1 将值保存到空接口

空接口（interface{}）是Go语言中最特殊的接口。在Java语言中，所有的类都继承自一个基类Object，而Go中的interface{}接口就相当于Java语言里的Object。

在Go语言中，空接口不包含任何方法，也正因如此，所有的类型都实现了空接口，因此空接口可以存储任意类型的数值。举个例子：

动手写12.4.1

```

01 package main
02
03 import "fmt"
04
05 func Log(name string, i interface{}) {
06     fmt.Printf("Name = %s, Type = %T, value = %v\n", name, i, i)
07 }
08
09 func main() {
10     var v1 interface{} = 1
11     var v2 interface{} = "abc"
```

```
12     var v3 interface{} = true
13     var v4 interface{} = &v1
14     var v5 interface{} = struct {
15         Name string
16     {"Xiao Ming"}
17     var v6 interface{} = &v5
18
19     Log("v1",v1)
20     Log("v2",v2)
21     Log("v3",v3)
22     Log("v4",v4)
23     Log("v5",v5)
24     Log("v6",v6)
25 }
```

运行结果如下：

```
Name = v1, Type = int, value = 1
Name = v2, Type = string, value = abc
Name = v3, Type = bool, value = true
Name = v4, Type = *interface {}, value = 0xc0420461c0
Name = v5, Type = struct { Name string }, value = {Xiao Ming}
Name = v6, Type = *interface {}, value = 0xc0420461d0
```

动手写12.4.1展示了空接口可以存储数字、字符串、结构体、指针等任意类型的数值，其中v4和v6指针在不同的计算机中，数值是随机的一个地址，所以会不同。

12.4.2 从空接口取值

我们接下来看如何从空接口获取值：

动手写12.4.2

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     var a string = "abc"
```

```

07     var i interface{} = a
08     var b string = i
09     fmt.Println(b)
10 }
```

动手写12.4.2运行后报错，报错结果如下：

```
cannot use i (type interface {}) as type string in assignment: need
type assertion
```

这个错误的意思是不能将空接口赋值到其他类型，如果需要的话必须使用类型断言。将动手写12.4.2改为如动手写12.4.3所示，就能成功从空接口中取值。

动手写12.4.3

```

01 package main
02
03 import "fmt"
04
05 func main() {
06     var a string = "abc"
07     var i interface{} = a
08     var b string = i.(string)
09     fmt.Println(b)
10 }
```

关于类型断言，在后面的章节会详细介绍。

12.4.3 空接口的常见使用

我们经常使用的打印函数`fmt.Println`就使用了空接口，`Println`的定义如下：

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

其中`interface{}`前面的三个点代表的是可变长参数。可变长参数函数即其参数数量是可变的——0个或多个。声明可变参数函数的方式是在其参数类型前带上省略符（三个点）前缀，如果除了可变长参数外还有其他参数，则可变长参数必须放置在参数列表的末尾，如`fmt.Printf`：

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

打印函数支持传入任意个值，并且值的类型也是任意的。我们可以写一个这样的例子：

动手写12.4.4

```
01 package main
02
03 import "fmt"
04
05 func Log(args ...interface{}) {
06     for num, arg := range args {
07         fmt.Printf("Index => %d, Value => %v\n", num, arg)
08     }
09 }
10
11 func main() {
12     s := make([]interface{}, 3)
13     s[0] = 1
14     s[1] = "abc"
15     s[2] = struct {
16         Num int
17     }{1}
18
19     //可变长参数
20     fmt.Println("====将切片拆散====")
21     Log(s...)
22     fmt.Println("====直接传入切片====")
23     Log(s)
24 }
```

运行结果如下：

```
====将切片拆散====
Index => 0, Value => 1
Index => 1, Value => abc
Index => 2, Value => {1}
====直接传入切片====
Index => 0, Value => [1 abc {1}]
```

这里可以看见，动手写12.4.4中存在这样的用法——“s...”，这个用法的意思是将切片拆散传入到参数列表中。在本例中，`Log(s...)`可以等同于`Log(s[0], s[1], s[2])`。

12.5

类型断言

类型断言是使用在接口变量上的操作。简单来说，接口类型向普通类型的转换就是类型断言。

类型断言的语法是：

```
t, ok := X.(T)
```

这里X表示一个接口变量，T表示一个类型（也可为接口类型），这句代码的含义是判断X的类型是否是T。如果断言成功，则ok为true，t的值为接口变量X的动态值；如果断言失败，则t的值为类型T的初始值，t的类型始终为T。举个例子：

动手写12.5.1

```

01 package main
02
03 import "fmt"
04
05 func checkTpye(t interface{}, ok bool) {
06     if ok {
07         fmt.Println("断言成功!")
08     } else {
09         fmt.Println("断言失败!")
10     }
11     fmt.Printf("变量t的类型 = %T , 值 = %v \n", t, t)
12 }
13
14 func main() {
15     var X interface{} = 1
16
17     fmt.Println("第一次断言：")
18     t0, ok := X.(string)
19     checkTpye(t0, ok)
20
21     fmt.Println("第二次断言：")
22     t1, ok := X.(float64)
23     checkTpye(t1, ok)
24 }
```

```
25     fmt.Println("第三次断言: ")
26     t2, ok := X.(int)
27     checkTpye(t2, ok)
28 }
```

运行结果如下：

```
第一次断言:
断言失败!
变量t的类型 = string , 值 =
第二次断言:
断言失败!
变量t的类型 = float64 , 值 = 0
第三次断言:
断言成功!
变量t的类型 = int , 值 = 1
```

动手写12.5.1中，我们定义了一个接口变量X，接口变量的动态类型为int，动态值为1。第一次我们断言X的动态类型为string，断言失败，t0的类型为string，值为空字符串，所以这里没有打印出任何东西；断言类型为float64时，t1的值为0，也就是float64的初始值。当我们断言为int时，断言成功，t2的类型变为int，值为接口变量X的动态值1。

类型断言分两种情况：

◇ 第一种，如果断言的类型T是一个具体类型，类型断言X.(T)就检查X的动态类型是否和T的类型相同。

如果这个检查成功了，类型断言的结果是一个类型为T的对象，该对象的值为接口变量X的动态值。换句话说，具体类型的类型断言从它的操作对象中获得具体的值。

如果检查失败，接下来这个操作会抛出panic，除非用两个变量来接收检查结果，如：f, ok := w.(*os.File)。

◇ 第二种，如果断言的类型T是一个接口类型，类型断言X.(T)检查X的动态类型是否满足T接口。

如果这个检查成功，则检查结果的接口值的动态类型和动态值不变，但是该接口值的类型被转换为接口类型T。换句话说，对一个接口类型的类型断言改变了类型的表述方式，改变了可以获取的方法集合（通常更大），但是它保护了接口值内部的动态类型和值的部分。

如果检查失败，接下来这个操作会抛出panic，除非用两个变量来接收检查结果，如：f, ok :=

w.(io.ReadWriter)。

我们再来看第二种情况。

动手写 12.5.2

```
01 package main
02
03 import "fmt"
04
05 type Person interface {
06     Speak()
07 }
08
09 type Student struct {
10     Name string
11 }
12
13 func (s Student) Speak() {
14     fmt.Println("My name is", s.Name)
15 }
16
17 func checkTpye(t interface{}, ok bool) {
18     if ok {
19         fmt.Println("断言成功!")
20     } else {
21         fmt.Println("断言失败!")
22     }
23     fmt.Printf("变量t的类型 = %T , 值 = %v \n", t, t)
24 }
25
26 func main() {
27     var student interface{} = Student{"小明"}
28
29     fmt.Println("第一次断言: ")
30     t0, ok := student.(string)
31     checkTpye(t0, ok)
32 }
```

```
33     fmt.Println("第二次断言: ")
34     t1, ok := student.(Person)
35     checkTpye(t1, ok)
36 }
```

运行结果如下：

```
第一次断言:
断言失败!
变量t的类型 = string , 值 =
第二次断言:
断言成功!
变量t的类型 = main.Student , 值 = {小明}
```

动手写12.5.2创建了一个Person接口和实现该接口的Student结构体。使用Student结构体实例化了一个student对象，第一次断言student为string类型，断言失败，第二次断言为Person接口，断言成功，并打印输出了student的值。

12.5.1 ok-pattern

接口类型断言有两种方式，一种是ok-pattern，一种是switch-type。一般来说，当要断言的接口类型种类较少时，可以使用ok-pattern这种方式，结构如下：

```
if value, ok := 接口变量.(类型); ok == true {
    //接口变量是该类型时的处理
}
```

举个例子：

动手写12.5.3

```
01 package main
02
03 import "fmt"
04
05 type Person struct {
06     Name string
07     Age int
08 }
09
```

```

10 func main() {
11     s := make([]interface{}, 3)
12     s[0] = 1
13     s[1] = "str"
14     s[2] = Person{"张三", 20}
15
16     for index, data := range s {
17         if value, ok := data.(int); ok == true {
18             fmt.Printf("s[%d] Type = int, Value = %d\n",
19                         index, value)
20         }
21         if value, ok := data.(string); ok == true {
22             fmt.Printf("s[%d] Type = string, Value = %s\n",
23                         index, value)
24         }
25         if value, ok := data.(Person); ok == true {
26             fmt.Printf("s[%d] Type = Person, Person.Name = %v,
27                         Person.Age = %d\n", index, value.Name, value.Age)
28         }
29     }
30 }
```

动手写12.5.3创建了一个长度为3的接口数组，分别赋值不同的变量类型，使用ok-pattern模式进行判断，运行结果如下：

```

s[0] Type = int, Value = 1
s[1] Type = string, Value = str
s[2] Type = Person, Person.Name = 张三, Person.Age = 20
```

12.5.2 switch-type

当要断言的接口类型种类较多时，使用ok-pattern方式就显得很累赘，代码非常冗余，这时我们可以使用switch-type来进行批量判断，switch-type的结构如下：

```

switch value := 接口变量.(type) {
    case 类型1:
        // 接口变量是类型1时的处理
```

```
case类型2:  
    // 接口变量是类型2时的处理  
case类型3:  
    // 接口变量是类型3时的处理  
...  
default:  
    // 接口变量不是所有case中所列举的类型时的处理  
}
```

同样拿上面的数据来举个例子：

动手写12.5.4

```
01 package main  
02  
03 import "fmt"  
04  
05 type Person struct {  
06     Name string  
07     Age int  
08 }  
09  
10 func main() {  
11     s := make([]interface{}, 3)  
12     s[0] = 1  
13     s[1] = "str"  
14     s[2] = Person{"张三", 20}  
15  
16     for index, data := range s {  
17         switch value := data.(type) {  
18             case int:  
19                 fmt.Printf("s[%d] Type = int, Value = %d\n",  
20                         index, value)  
21             case string:  
22                 fmt.Printf("s[%d] Type = string, Value = %s\n",  
23                         index, value)  
24             case Person:
```

```

23             fmt.Printf("s[%d] Type = Person, Person.Name = %v,
24                                         Person.Age = %d\n", index, value.Name, value.Age)
25         }
26     }

```

动手写12.5.4使用switch-type模式进行类型断言，代码结构更加清晰，利于维护，运行结果如下：

```

s[0] Type = int, Value = 1
s[1] Type = string, Value = str
s[2] Type = Person, Person.Name = 张三, Person.Age = 20

```

12.6 小结

- ◇ 了解什么是鸭子类型。
- ◇ 掌握接口的定义与实现。
- ◇ 了解什么是接口嵌入。
- ◇ 掌握空接口的常见用法，如空接口的赋值操作。
- ◇ 掌握类型断言的两种方式：ok-pattern和switch-type。
- ◇ Go语言的接口是非常易用、简洁的，需要重点掌握的是Go语言的接口实现和类型断言，能够判断接口值是什么类型，接口值的内容是什么。

12.7 知识拓展

非侵入式接口

Go语言的接口是非侵入式的，我们先来看侵入式接口与非侵入式接口的区别：

- ◇ 侵入式接口：需要显式地创建一个类去实现一个接口。
- ◇ 非侵入式接口：不需要显式地创建一个类去实现一个接口。

大部分语言的接口都是侵入式接口，需要显式地去创建声明实现一个接口。例如Java的接口定义如下，要实现一个接口，必须使用implements显式声明。

动手写12.7.1

```
01 interface IFoo {
02     void Bar();
03 }
04 class Foo implements IFoo {
05     void Bar() {}
06 }
```

非侵入式接口有三个好处：

- ◇ 去掉了繁杂的继承体系，Go语言的标准库再也不需要绘制类库的继承树图。在Go中，类的继承树并无意义，你只需要知道这个类实现了哪些方法、每个方法是何含义就足够了。
- ◇ 实现类的时候，只需要关心自己应该提供哪些方法，不用再纠结接口需要拆得多细才合理。接口由使用方按需定义，而不用事前规划。
- ◇ 不用为了实现一个接口而导入一个包，因为多引用一个外部的包，就意味着更多的耦合。接口由使用方按自身需求来定义，使用方无须关心是否有其他模块定义过类似的接口。

总的来说，非侵入式的接口设计更简洁、灵活，更注重实用性。下面我们从代码层面来更深入地理解。

动手写12.7.2

```
01 type IPhoner interface {
02     Call() error
03     Video() error
04     Game() error
05 }
06
07 type Apple interface {
08     Call() error
09     Video() error
10 }
11
12 type Huawei interface {
13     Call() error
14     Game() error
15 }
```

动手写12.7.2创建了三个接口，分别是IPhoner、Apple、Huawei，其中Apple、Huawei接口的方

法在IPhoner接口中全部包含。我们再来创建一个实现IPhoner接口的结构体：

动手写12.7.3

```

01 type Phone struct {
02     Name string
03 }
04
05 func (p *Phone) Call() error {
06     fmt.Println(p.Name, "Start Call")
07     return nil
08 }
09
10 func (p *Phone) Video() error {
11     fmt.Println(p.Name, "Start Video")
12     return nil
13 }
14
15 func (p *Phone) Game() error {
16     fmt.Println(p.Name, "Start Game")
17     return nil
18 }
```

动手写12.7.3创建了一个叫Phone的结构体，并有实现IPhoner的三个方法Call()、Video()、Game()，也就是说，Phone是IPhoner这个接口的一个实现。这时，如果我们想要获得一个Apple的对象，在传统的侵入式接口语言里，至少要写一个appleClass类去实现Apple接口，而为了得到一个Huawei的对象又不得不写一个huaweiClass类去实现Huawei接口。同理，要得到一个IPhoner的对象也得这么做，但显然这样的操作，代码的复用率并不高。

而在Go语言中，并不需要这样，只需要写一个通用的IPhoner接口实现就可以完成以上需求，也就是只需要Phone这个实现就可以了。

动手写12.7.4

```

01 func main() {
02     var apple Apple = &Phone{"apple"}
03     var huawei Huawei = &Phone{"huawei"}
04 }
```

```
05     apple.Call()  
06     huawei.Game()  
07 }
```

运行结果如下：

```
apple Start Call  
huawei Start Game
```

动手写12.7.4仅使用Phone这个接口实现就完成了Apple和Huawei这两个接口的实例。动手写12.7.4也可以使用new来生成Apple的实例：

动手写12.7.5

```
01 func main() {  
02     var apple Apple = new(Phone)  
03     var huawei Huawei = new(Phone)  
04  
05     apple.Call()  
06     huawei.Game()  
07 }
```

运行结果如下：

```
Start Call  
Start Game
```

并发与通道

<<

并发是指在同一段时间内，程序可以执行多个任务。随着社会需求的发展，光靠硬件的提升是无法满足高并发的需求的，因此从程序的角度去解决并发问题就显得尤为重要。最早支持并发编程的语言是汇编语言，但那时没有任何理论来支持这种编程模式，因此一个细微的编程错误就可能导致程序变得非常不稳定。到现在为止，基本所有的语言都支持并发编程，而Go语言最大的特点就是从语言层面支持并发。本章会详细讲解Go语言如何利用协程（coroutine）和通道（channel）来解决并发问题。

13.1 概述

在程序中往往有很多很耗时的工作，比如上传文件、下载文件、网络聊天。这时候，一个线程是服务不了多个用户的，会产生因为资源独占导致的等待问题，这时候就需要使用并发的手段来解决。并发编程的含义比较广泛，包括多线程编程、多进程编程以及分布式程序。本章所讲述的并发叫作协程（coroutine），属于多线程编程。

在计算机刚出现的那个年代并没有“并发”这个概念，因为以前的命令式程序是以串行为基础，程序会依次执行每一条指令，整个程序只有一个上下文和一个调用栈。并发则意味着程序可以有多个上下文、多个调用栈。

提示

调用栈是计算机科学中存储有关正在运行的子程序的消息的栈，经常被用于存放子程序的返回地址。在调用任何子程序时，主程序都必须暂存子程序运行完毕后应该返回到的地址。因此，

如果被调用的子程序还要调用其他的子程序，其自身的返回地址就必须存入调用栈，在其自身运行完毕后再取回。

13.1.1 并行与并发

很多时候我们会把“并行”和“并发”理解成同一个概念，但实际上，它们有着清晰的区别。我们先来看它们的定义。

- ◇ 并行 (parallelism)：指在同一时刻，有多条指令在多个处理器上同时执行。
- ◇ 并发 (concurrency)：指在同一时刻只能有一条指令执行，但多个进程指令被快速地轮换执行，得到在宏观上有多个进程同时执行的效果，但在微观上并不是同时执行，只是把时间片分成了若干段，使得多个进程快速交替执行。

并行是真正意义上的同时执行，而并发只是从宏观的角度来看具有同时执行的效果。如图13.1.1所示，对于任务A和B，在任何时刻，任务A、B都在执行，这就是并行。如图13.1.2所示为并发，在任何时刻，有且仅有一个任务在执行，但由于计算机处理速度非常快，对于任务A和B，从宏观的角度来看就感觉它们是在同时执行。

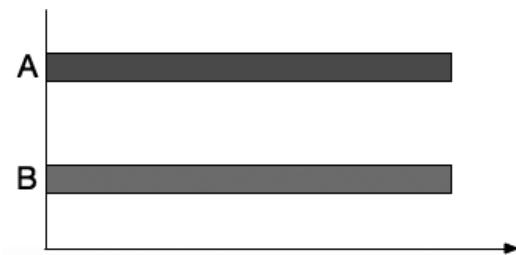


图13.1.1 并行

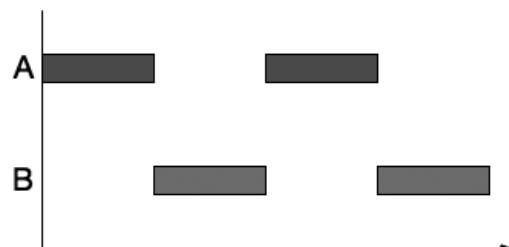


图13.1.2 并发

13.1.2 Go并发优势

Go语言的优势在于从语言层面上支持了并发，并发编程的内存管理有时是非常复杂的，而开发者不用担心并发的底层逻辑、内存管理，因为这些在语言层面上已经解决了，开发者只需要编写好自己的业务逻辑即可。Go语言也提供了十分强大的自动垃圾回收机制，开发者不用担心创建的量如何销毁。

在Go语言里，想要编写一个并发程序是非常容易的事，它不需要额外引用其他的第三方库，只需要使用“go”关键字就可以实现。在其他语言中，编写并发程序往往需要使用其他的并发库才能实现。

在Go语言里，只需要使用“go”加上函数名称就可以让这个函数变为并发函数，如下所示：

动手写13.1.1

```

01 package main
02
03 func run(arg string) {
04     // 此线程的任务
05 }
06
07 func main() {
08     go run("this is new thread")
09 }
```

Go语言的并发基于CSP（Communication Sequential Process，通信顺序进程）模型，CSP模型是在20世纪70年代提出的用于描述两个独立的并发实体通过共享的通信管道（channel）进行通信的并发模型。CSP中channel是一类对象，它不关注发送消息的实体，而关注发送消息时使用的通信管道。简单来说，CSP模型提倡通过通信来共享内存，而非通过共享内存来通信。

基于CSP模型，也就意味着显式锁都是可以避免的，比如资源竞争——多个进程同时获取文件资源并需要进行修改时，首先拿到资源的进程会加上锁，等修改完之后再把锁去掉，然后提供给下一个进程来进行修改，只有这样才不会出现数据不一致。但是Go语言不是通过锁的方式，而是通过通信的方式，通过安全的通道发送和接收数据以实现同步，这就大大简化了并发编程的编写。

13.2 goroutine

goroutine是Go并发设计的核心，也叫协程，它比线程更加轻量，因此可以同时运行成千上万个并发任务。不仅如此，Go语言内部已经实现了goroutine之间的内存共享，它比线程更加易用、高效和轻便。

13.2.1 goroutine定义

在Go语言中，每一个并发的执行单元叫作一个goroutine。想要编写一个并发任务，只需要在调用的函数前面添加go关键字，就能使这个函数以协程的方式运行。

go 函数名(函数参数)

如果函数有返回值，返回值会被忽略。因此，一旦使用go关键字，就不能使用函数返回值来

与主进程进行数据交换，而只能使用channel，关于channel将会在后面的章节中进行介绍。

对用户来说，协程与线程几乎没什么区别，但是实际上二者是有一定区别的。线程有固定的栈，基本都是2 MB，都是固定分配的；这个栈用于保存局部变量，在函数切换时使用。但是对于goroutine这种轻量级的协程来说，一个大小固定的栈可能会导致资源浪费，所以Go采用了动态扩张收缩的策略，初始化为2 KB，最大可扩张到1 GB。

提示

操作系统对线程进行调度也需要成本。线程挂起前会保存线程上下文到栈空间，再切换到可执行线程。线程数很多时，线程上下文切换会导致CPU开销变大。

每个线程都有一个id，这个在线程创建时就会返回，所以可以很方便地通过id操作某个线程。但是goroutine内没有这个概念，这是在Go语言设计之初出于防止被滥用的考虑而定的，所以你不能在一个协程中“杀死”另外一个协程，编码时需要考虑到协程什么时候创建以及什么时候释放。

协程和线程最重要的区别在于：首先，线程切换需要陷入内核，然后进行上下文切换，而协程在用户态由协程调度器完成，不需要陷入内核，这样代价就小了。其次，协程的切换时间点是由调度器决定，而不是由系统内核决定的，尽管它们的切换点都是时间片超过一定阈值，或者是进入I/O或睡眠等状态时。最后，基于垃圾回收的考虑，Go实现了垃圾回收，但垃圾回收的必要条件是内存位于一致状态，因此就需要暂停所有的线程。如果交给系统去做，那么会暂停所有的线程使其一致；但如果在Go里，调度器知道什么时候内存位于一致状态，也就没有必要暂停所有运行的线程。

13.2.2 创建goroutine

只需要在函数调用语句前面添加go关键字，就可以创建并发执行单元。开发人员无须了解任何执行细节，调度器会自动将其安排到合适的系统线程上去执行。

在并发编程里，我们通常要将一个过程分割成几块，然后让不同的goroutine各自负责其中的一块工作。

动手写13.2.1

```
01 package main
02
03 import (
04     "fmt"
05     "time"
```

```
06  )
07
08 func Task1() {
09     for {
10         fmt.Println(time.Now().Format("15:04:05"), "正在处理Task1
11             的任务!")
12     }
13 }
14
15 func Task2() {
16     for {
17         fmt.Println(time.Now().Format("15:04:05"), "正在处理Task2
18             的任务!")
19     }
20 }
21
22 func main() {
23     go Task1()
24     go Task2()
25     for {
26         fmt.Println(time.Now().Format("15:04:05"), "正在处理主进程
27             的任务!")
28     }
29 }
```

动手写13.2.1中，有三个任务模块，分别是main、Task1、Task2，这里使用了两个go关键字将Task1和Task2改变成了两个并发模块，运行结果如下：

```
16:29:47 正在处理Task1的任务!
16:29:47 正在处理Task2的任务!
16:29:47 正在处理主进程的任务!
16:29:48 正在处理Task2的任务!
16:29:49 正在处理主进程的任务!
```

```
16:29:49 正在处理Task2的任务!
```

```
16:29:50 正在处理Task1的任务!
```

```
.....
```

如果将main中的两个关键字删除，由于Task1是死循环，就会一直执行循环里的内容，导致执行不到其他Task，因此程序运行的结果如下：

```
16:38:07 正在处理Task1的任务!
```

```
16:38:10 正在处理Task1的任务!
```

```
16:38:13 正在处理Task1的任务!
```

```
16:38:16 正在处理Task1的任务!
```

```
.....
```

13.2.3 main函数

当一个程序启动时，其主函数即在一个单独的goroutine中运行，我们称之为main goroutine。新的goroutine会使用go来创建。先来看一个例子：

动手写13.2.2

```
01 package main
02
03 import (
04     "fmt"
05     "time"
06 )
07
08 func Task1() {
09     for {
10         fmt.Println(time.Now().Format("15:04:05"), "正在处理Task1
11             的任务!")
12         time.Sleep(time.Second * 3)
13     }
14
15 func main() {
16     go Task1()
17 }
```

动手写13.2.2运行后将不会有任何输出，这是因为运行go Task1()，程序会立刻返回到main函数，而main函数后面没有任何代码逻辑，程序就会判断为执行完毕，终止所有协程。因此，想要让Task1函数执行，可以在main函数添加一些等待逻辑，例如Sleep()。

动手写13.2.3

```

01 package main
02
03 import (
04     "fmt"
05     "time"
06 )
07
08 func Task1() {
09     for {
10         fmt.Println(time.Now().Format("15:04:05"), "正在处理Task1
11             的任务!")
12         time.Sleep(time.Second * 3)
13     }
14
15 func main() {
16     go Task1()
17     time.Sleep(time.Second * 100)
18 }
```

动手写13.2.3的运行结果如下：

```

16:52:51 正在处理Task1的任务!
16:52:54 正在处理Task1的任务!
16:52:57 正在处理Task1的任务!
.....
```

提示

所有goroutine在main函数结束时会一并结束。goroutine虽然类似于线程概念，但调度性能上不如线程细致，而细致程度取决于goroutine调度器的实现和运行环境。终止goroutine最好的方法是直接在函数中自然返回。

13.2.4 使用匿名函数创建goroutine

go关键字后面也可以为匿名函数，格式如下：

```
go func(参数列表) {  
    函数体  
} (调用参数列表)
```

因此动手写13.2.3可以写成如下形式，这样就不用为Task1命名了。

动手写13.2.4

```
01 package main  
02  
03 import (  
04     "time"  
05     "fmt"  
06 )  
07  
08 func main() {  
09     go func() {  
10         for {  
11             fmt.Println(time.Now().Format("15:04:05"), "正在处理  
12                 Task1的任务!")  
13             time.Sleep(time.Second * 3)  
14         }()  
15         time.Sleep(time.Second * 100)  
16     }  
}
```

运行结果如下：

```
17:00:08 正在处理Task1的任务!  
17:00:11 正在处理Task1的任务!  
17:00:14 正在处理Task1的任务!  
.....
```

13.2.5 runtime包

Go语言中runtime（运行时）包实现了一个小型的任务调度器。这个调度器的工作原理和系统

对线程的调度类似，Go语言调度器可以高效地将CPU资源分配给每一个任务。本小节重点介绍三个函数：Gosched()、Goexit()、GOMAXPROCS()。

1. Gosched()

```
func Gosched()
```

Gosched()使当前Go协程放弃处理器，以让其他Go协程运行。它不会挂起当前Go协程，因此当前Go协程未来会恢复执行。

Go语言的协程是抢占式调度的，当遇到长时间执行或者进行系统调用时，会主动把当前goroutine的CPU (P)转让出去，让其他goroutine能被调度并执行。一般出现如下几种情况，goroutine就会发生调度：

- ◇ syscall。
- ◇ C函数调用（本质上和syscall一样）。
- ◇ 主动调用runtime.Gosched。
- ◇ 某个goroutine的调用时间超过100 ms，并且这个goroutine调用了非内联的函数。

提示

内联函数是指当编译器发现某段代码在调用一个内联函数时，它不是去调用该函数，而是将该函数的代码整段插入到当前位置。这样做的好处是省去了调用的过程，加快程序运行速度。

动手写13.2.5

```
01 package main
02
03 import (
04     "fmt"
05 )
06
07 func main() {
08     go func() {
09         for i := 0; i < 3; i++ {
10             fmt.Println("go")
11         }
12     }()
13
14     for i := 0; i < 2; i++ {
```

```
15         fmt.Println("main")
16     }
17 }
```

动手写13.2.5中，由于main函数中的第二个for循环一直抢占了CPU控制权，直到运行结束也没有发生goroutine调度，因此，第一个Go协程匿名函数无法获取CPU控制权，导致自始至终都无法执行，运行结果如下：

```
main
main
```

如果想要第一个for循环执行，可以在第二个for循环中使用runtime.Gosched()来阻止其获取控制权，如下所示。

动手写13.2.6

```
01 package main
02
03 import (
04     "fmt"
05     "runtime"
06 )
07
08 func main() {
09     go func() {
10         for i := 0; i < 3; i++ {
11             fmt.Println("go")
12         }
13     }()
14
15     for i := 0; i < 2; i++ {
16         runtime.Gosched()
17         fmt.Println("main")
18     }
19 }
```

运行结果如下：

```
go
go
go
main
main
```

2. Goexit()

```
func Goexit()
```

Goexit()终止调用它的Go协程，但其他Go协程不会受影响。Goexit()会在终止该Go协程前执行所有defer的函数。

在程序的main go协程调用本函数时，会终结该Go协程，而不会让main返回。这是因为main函数没有返回，程序会继续执行其他的Go协程。如果所有其他Go协程都退出了，程序就会崩溃。

动手写13.2.7

```
01 package main
02
03 import (
04     "fmt"
05     "time"
06     "runtime"
07 )
08
09 func Task1() {
10     defer fmt.Println("task1 stop")
11     fmt.Println("task1 start")
12     fmt.Println("task1 work")
13 }
14
15 func Task2() {
16     defer fmt.Println("task2 stop")
17     fmt.Println("task2 start")
18     runtime.Goexit() // 效果和return一样
19     fmt.Println("task2 work")
```

```
20 }
21
22 func main() {
23     go Task1()
24     go Task2()
25     time.Sleep(time.Second * 5)
26 }
```

动手写13.2.7中，由于Task2在打印“work”之前就执行了runtime.Goexit()，可以理解为直接返回，因此后面一句打印是不会运行的，最后的运行结果如下：

```
task1 start
task1 work
task1 stop
task2 start
task2 stop
```

3. GOMAXPROCS()

GOMAXPROCS(n int)函数可以设置程序在运行中所使用的CPU数，在以后的编程中是用得最多的。Go语言程序默认会使用最大CPU数进行计算。

```
func GOMAXPROCS(n int) int
```

GOMAXPROCS()设置可同时执行的最大CPU数，并返回先前的设置。若n < 1，它就不会更改当前设置。本地机器的逻辑CPU数可通过NumCPU查询。本函数在调度程序优化后会去掉。

动手写13.2.8

```
01 package main
02
03 import (
04     "runtime"
05     "fmt"
06     "time"
07 )
08
09 func main() {
10     n := runtime.GOMAXPROCS(1)
```

```

11     fmt.Println("先前的CPU核数设置为: ", n)
12
13     last := time.Now()
14     for i := 0; i < 100000; i++ {
15         go func() {
16             // 耗时任务
17             a := 999999 ^ 9999999
18             a = a + 1
19         }()
20     }
21
22     now := time.Now()
23     fmt.Println(now.Sub(last))
24 }
```

动手写13.2.8中，使用一个CPU进行计算时，运行结果如下：

```

先前的CPU核数设置为: 4
637.3143ms
```

当使用4核进行计算时，可以看见，效率得到了显著的提升，运行结果如下：

```

先前的CPU核数设置为: 4
27.4389ms
```

13.3

channel

goroutine运行在相同的地址空间，因此访问共享内存必须做好同步。引用类型channel是CSP模式的具体体现，用于多个goroutine之间的通信。其内部实现了同步，确保并发安全。

13.3.1 channel类型

channel是一种特殊的类型，和map类似，channel也是一个对应make创建的底层数据结构的引用。声明一个channel的方式如下：

```
var 通道变量 chan 通道类型
```

通道变量是保存通道的引用变量；通道类型是指该通道可传输的数据类型。当我们复制一个channel或用于函数参数传递时，我们只是拷贝了一个channel引用，因此调用者与被调用者都将引用同一个对象。和其他引用类型一样，channel的零值也是nil。

定义一个channel时，也需要定义发送到channel的值的类型。channel可以使用内置的make()函数来创建：

```
make(chan Type) // 等价于make(chan Type, 0)  
make(chan Type, capacity)
```

当capacity为0时，channel是无缓冲阻塞读写的；当capacity大于0时，channel是有缓冲、非阻塞的，直到写满capacity个元素才阻塞写入。

channel通过操作符“<-”来接收和发送数据，接收和发送数据的语法如下：

```
channel <- value           // 发送value到channel  
<-channel                // 接收并将其丢弃  
x := <-channel             // 从channel中接收数据，并赋值给x  
x, ok := <-channel         // 同上，并检查通道是否关闭，将此状态赋值给ok
```

默认情况下，channel接收和发送数据都是阻塞的，除非另一端已准备好接收，这样就使得goroutine的同步更加简单，而不需要显式锁。举个例子：

动手写13.3.1

```
01 package main  
02  
03 import (  
04     "time"  
05     "fmt"  
06 )  
07  
08 func main() {  
09     ch := make(chan string)  
10  
11     go func() {  
12         fmt.Println(<-ch)  
13     }()
}
```

```

14     ch <- "test"
15
16     time.Sleep(time.Second)
17 }
```

动手写13.3.1中定义并创建了一个可以传输string类型的ch通道变量，在匿名协程函数中，从ch通道中接收数据并打印，运行结果如下：

```
test
```

13.3.2 缓冲机制

channel按是否支持缓冲区可分为无缓冲的通道（unbuffered channel）和有缓冲的通道（buffered channel）。

无缓冲的通道是指在接收前没有能力保存任何值的通道。

这种类型的通道要求发送goroutine和接收goroutine同时准备好，才能完成发送和接收操作。如果两个goroutine没有同时准备好，会导致先执行发送或接收操作的goroutine阻塞等待。

这种对通道进行发送和接收的交互行为本身就是同步的，其中任意一个操作都无法离开另一个操作单独存在。

无缓冲的channel创建格式：

```
make(chan Type) //等价于make(chan Type, 0)
```

如果没有指定缓冲区容量，那么该通道就是同步的。

动手写13.3.2

```

01 package main
02
03 import (
04     "time"
05     "fmt"
06 )
07
08 func main() {
09     ch := make(chan int, 0)
10     go func() {
```

```
11         for i := 0; i < 3; i++ {
12             fmt.Printf("len(ch)=%v, cap(ch)=%v\n", len(ch),
13                         cap(ch))
14             ch <- i
15         }()
16     for i := 0; i < 3; i++ {
17         time.Sleep(time.Second)
18         fmt.Println(<-ch)
19     }
20 }
```

动手写13.3.2创建了一个无缓冲通道ch，由于该通道是无缓冲的，因此只有当接收者收到了数据，发送者才能继续发送数据，可以看到运行结果如下：

```
len(ch)=0, cap(ch)=0
0
len(ch)=0, cap(ch)=0
1
len(ch)=0, cap(ch)=0
2
```

有缓冲通道是一种在被接收前能存储一个或多个值的通道。创建一个有缓冲通道的方式如下：

```
make(chan Type, capacity)
```

这种类型的通道并不强制要求goroutine之间必须同时完成接收和发送。通道阻塞发送和接收的条件也会不同。只有在通道中没有要接收的值时，接收动作才会阻塞。只有在通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

这导致有缓冲的通道和无缓冲的通道之间有一个很大的不同：无缓冲的通道保证进行发送和接收的goroutine会在同一时间进行数据交换，有缓冲的通道没有这种保证。

动手写13.3.3

```
01 package main
02
03 import (
```

```

04      "time"
05      "fmt"
06  )
07
08 func main() {
09     ch := make(chan int, 3)
10     go func() {
11         for i := 0; i < 3; i++ {
12             fmt.Printf("len(ch)=%v, cap(ch)=%v\n", len(ch),
13                     cap(ch))
14             ch <- i
15         }
16         for i := 0; i < 3; i++ {
17             time.Sleep(time.Second)
18             fmt.Println(<-ch)
19         }
20     }

```

动手写13.3.3创建了一个容量为3的缓冲channel，由于存在缓冲区，在缓冲区未填满的情况下，程序就不会被阻塞执行。运行结果如下：

```

len=0, cap=3
len=1, cap=3
len=2, cap=3
0
1
2

```

13.3.3 close和range

当发送者知道没有更多的值需要发送到channel时，让接收者也能及时知道没有更多的值需要接收是很有必要的，因为这样就可以让接收者停止不必要的等待。这可以通过内置的close函数和range关键字来实现。

1. close

使用close关闭channel时需要注意：

- ◇ channel不像文件一样需要经常去关闭，只有当你确实没有任何需要发送的数据时，或者想要显式地结束range循环之类的，才会去关闭channel。
- ◇ 关闭channel后，无法向channel再次发送数据，再次发送将会引发panic错误。
- ◇ 关闭channel后，可以继续从channel接收数据。
- ◇ 对于nil channel，无论接收还是发送都会被阻塞。

动手写13.3.4

```
01 package main
02
03 import (
04     "fmt"
05 )
06
07 func main() {
08     ch := make(chan int, 3)
09     go func() {
10         for i := 0; i < 3; i++ {
11             fmt.Printf("len=%v, cap=%v\n", len(ch), cap(ch))
12             ch <- i
13         }
14         close(ch)
15     }()
16     for {
17         if val, ok := <-ch; ok == true {
18             fmt.Println(val)
19         } else {
20             return
21         }
22     }
23 }
```

动手写13.3.4中，发送者在发送完数据后，使用了close关闭channel。channel被关闭后，ok的值就会变为false，最终程序结束运行。运行结果如下：

```
len=0, cap=3
len=0, cap=3
```

```
len=1, cap=3
0
1
2
```

2. range

除了使用上面的方式来遍历channel，还可以使用更加简洁的range关键字，格式如下：

```
for data := range ch {
}
```

动手写13.3.5

```
01 package main
02
03 import (
04     "fmt"
05 )
06
07 func main() {
08     ch := make(chan int, 3)
09     go func() {
10         for i := 0; i < 3; i++ {
11             fmt.Printf("len=%v, cap=%v\n", len(ch), cap(ch))
12             ch <- i
13         }
14         close(ch)
15     }()
16     for data := range ch {
17         fmt.Println(data)
18     }
19 }
```

动手写13.3.5使用range遍历了channel，当channel关闭后，range也能自动结束本次遍历。运行结果如下：

```
len=0, cap=3
len=0, cap=3
len=1, cap=3
0
1
2
```

13.3.4 单向channel

默认情况下，通道是双向的，就是既可以向通道发送数据，也可以从通道中接收数据。但是，我们建一个通道作为参数进行传递，经常希望只单向使用，即要么只发送数据，要么只接收数据，这时候就可以指定通道的方向，也就是使用单向通道。

单向channel变量的声明非常简单，如下：

```
var ch1 chan int           // ch1为一个双向通道
var ch2 chan<- int         // ch2为一个只能接收的单向通道
var <-chan int              // ch3为一个只能发送的单向通道
```

“chan<” 表示数据进入通道，只要把数据写入通道，对于调用者而言就是输出。“<-chan” 表示数据从通道中出来，对于调用者就是得到通道的数据，也就是输入。

动手写13.3.6

```
01 package main
02
03 import "fmt"
04
05 func producer(out chan<- int) {
06     for i := 0; i < 10; i++ {
07         out <- i
08     }
09     close(out)
10 }
11
12 func consumer(in <-chan int) {
13     for val := range in {
14         fmt.Println(val)
```

```

15      }
16  }
17
18 func main() {
19     ch := make(chan int)
20     go producer(ch)
21     consumer(ch)
22 }
```

动手写13.3.6展示了生产者—消费者模型的实现，producer是数据生产者，consumer是数据消费者，数据流向是固定的，也就是从生产者流到消费者，这种情况就适合使用单向通道。运行结果如下：

```

0
1
2
3
4
5
6
7
8
9
```

提示

可以将普通的双向channel隐式转换为单向channel，不能将单向channel转换为双向channel。

13.3.5 定时器

在Go语言标准库的time包中，定时器的实现使用的就是单向channel。我们来看看如何使用定时器实现每隔一段时间打印一串字符：

动手写13.3.7

```

01 package main
02
```

```
03 import (
04     "time"
05     "fmt"
06 )
07
08 func main() {
09     ticker := time.NewTicker(time.Second)
10
11     for {
12         <-ticker.C
13         fmt.Println("loop")
14     }
15 }
```

运行结果如下：

```
loop
loop
loop
.....
```

13.4 select

Go语言中，通过关键字select可以监听channel上的数据流动。select的用法和switch非常相似，由select开始一个新的选择块，每个选择条件由case语句来描述。

13.4.1 select作用

与switch语句可以选择任何可使用相等比较的条件相比，select有较多的限制，其中最大的限制就是每个case语句里面必须是一个I/O操作，大致结构如下：

```
select {
case <-chan1:
    // 如果chan1成功读到数据，则执行该case语句
case chan2 <- 1:
    // 如果成功向chan2写入数据，则执行该case语句
```

```

default:
    //如果上面的case都没有执行成功，则执行该default语句
}

```

在一个select语句中，Go语言会按顺序从头至尾评估每一个发送和接收语句，如果其中的任意一个语句可以继续执行（没有被阻塞），那么就从那些可以执行的语句中随机选择一条来使用。

如果没有任何一条语句可以执行（即所有通道都被阻塞），就会默认执行default语句，同时程序的执行会从select语句后的语句中恢复。如果没有default语句，则select语句将被阻塞，直到有一个channel可以进行下去。

动手写13.4.1

```

01 package main
02
03 import (
04     "fmt"
05     "time"
06 )
07
08 func main() {
09     ch := make(chan int)
10     go func() {
11         for i := 0; i < 3; i++ {
12             ch<- i
13         }
14     }()
15
16     for {
17         select {
18             case msg := <-ch:
19                 fmt.Println(msg)
20             default:
21                 time.Sleep(time.Second)
22         }
23     }
24 }

```

运行结果如下：

```
0
1
2
```

13.4.2 超时

有时候会出现goroutine阻塞的情况，为了避免程序长时间进入阻塞，我们可以使用select来实现阻塞超时机制，通过如下方法实现：

动手写13.4.2

```
01 package main
02
03 import (
04     "fmt"
05     "time"
06 )
07
08 func main() {
09     ch := make(chan int)
10     done := make(chan bool)
11     go func() {
12         for {
13             select {
14                 case val := <-ch:
15                     fmt.Println(val)
16                 case <-time.After(time.Second * 3):
17                     fmt.Println("已超时3秒")
18                     done <- true
19             }
20         }
21     }()
22
23     for i := 0; i < 10; i++ {
```

```

24         ch <- i
25     }
26
27     <-done
28     fmt.Println("程序终止")
29 }
```

运行结果如下：

```

0
1
2
3
4
5
6
7
8
9
已超时3秒
程序终止
```

13.4.3 死锁

在编写并发程序时可能会碰到死锁。什么是死锁？就是所有的线程或进程都在等待资源的释放，举个例子：

动手写13.4.3

```

01 package main
02
03 func main() {
04     ch := make(chan int)
05     <- ch // 阻塞main goroutine, 信道ch被锁
06 }
```

动手写13.4.3中只有一个goroutine，我们向里面加数据或者存数据的话，都会锁死信道，并且

阻塞当前goroutine，也就是所有的goroutine（其实只有main线程一个）都在等待信道的开放（没人拿走数据的话，信道是不会开放的），这就产生了死锁。

在非缓冲信道上如果发生了流入无流出，或者流出无流入，都会导致死锁。同样地，使用select关键字，其中不加任何代码也会产生死锁。

动手写13.4.4

```
01 package main
02
03 func main() {
04     select {
05     }
06 }
```

运行结果如下：

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [select (no cases)]:  
main.main()
```

13.5 小结

- ◇ 了解并行和并发的区别。
- ◇ 掌握goroutine的创建和使用。
- ◇ 掌握channel的创建和使用，以及多个goroutine之间的通信。
- ◇ 掌握如何使用runtime包对goroutine进行调度。
- ◇ 掌握channel的缓冲机制。
- ◇ 掌握Go关键字select的常见用法。

>> 第 14 章

反射

<<

在学习反射之前，我们先来思考一个问题。我们常用的一个打印函数fmt.Println()可以打印任何类型的数据，但是它本身是怎么实现的呢？解读源码可以看到，fmt包使用了一个叫reflect的反射标准库，也就是基于Go语言反射的特性写出来的，具体源码读者可以自行研究。

反射的强大之处就在于它非常灵活，通常用于做一些通用框架代码，而不需要理解业务，因此具有快速处理不同业务的功能。但它强大的同时也带来了很多弊端，比如代码可读性和可维护性变差，性能也大打折扣。因此，是否使用反射需要进行利弊权衡后决定，并不是所有的程序都适合使用反射。本章将会讲述如何使用反射及反射的特性。

14.1 反射定义

14.1.1 反射的定义

Go语言提供了一种机制：在运行时更新变量和检查它们的值、调用它们的方法和它们支持的内在操作，但是在编译时并不知道这些变量的具体类型。这种机制被称为反射。Go语言在reflect包中实现了反射，通过reflect.TypeOf()，reflect.ValueOf()分别从类型、值的角度来描述一个Go对象。

反射并不是一个新概念，Go语言反射与其他语言有一定的差异，我们先来了解一下其他语言中的反射。

14.1.2 与其他语言的区别

Java的反射机制是其标志性的特征之一，正是这种语言本身支持的强大的机制，使很多流行的框架有了用武之地。

在运行状态中，Java对于任何的类，都能够确认这个类的所有方法和属性；对于任何一个对

象，都能调用它的任意方法和属性。这种动态获取或者调用的方式就是Java的反射机制。

在Java中，通过反射机制在运行时能够做到：

- ◇ 确认对象的类。
- ◇ 确认类的所有成员变量和方法。
- ◇ 动态调用任意一个对象的方法。

下面举一个例子，看在Java中如何通过反射获取一个Class对象。

动手写14.1.1

```
01 package com.test;
02
03 public class Ref {
04     public static void main(String[] args) {
05         //1.通过对象调用getClass()方法来获取，通常应用在比如你传过来一个
06         //Object类型的对象，而我不知道你具体是什么类的情况
07         Student stu1 = new Student();
08         Class stuClass = stu1.getClass();
09
10         //2.直接通过“类名.class”的方式得到，该方法最为安全可靠，程序性能
11         //更高
12         //这说明任何一个类都有一个隐含的静态成员变量class
13         Class stuClass2 = Student.class;
14
15         //3.通过Class对象的forName()静态方法来获取，用得最多，但可能抛出
16         //ClassNotFoundException异常
17         try {
18             //注意此字符串必须是真实路径，就是带包名的类路径“包名.类名”
19             Class stuClass3 = Class.forName("com.test.Student");
20         } catch (ClassNotFoundException e) {
21             e.printStackTrace();
22         }
23     }
24 }
```

Java可以通过三种方式反射获得一个Class对象，分别是通过对象调用getClass()方法、通过类名.class以及通过Class对象的forName()静态方法来获取。

在Go语言中，只能使用如上所示的第一种方法来反射获取运行状态的对象的值或方法，最重

要的是，Go语言不支持通过对字符串解析，从而反射出对应的类型结构，这点与Java反射有着很大的区别。

我们再来看一个Go反射的例子：

动手写14.1.2

```

01 package main
02
03 import (
04     "reflect"
05     "fmt"
06 )
07
08 func main() {
09     var a interface{} = "我是字符串"
10     typeOfa := reflect.TypeOf(a)
11     fmt.Println("变量a的类型为：" + typeOfa.Name())
12     valueOfa := reflect.ValueOf(a)
13     if typeOfa.Kind() == reflect.String {
14         fmt.Println("变量a的值为：" + valueOfa.String())
15     }
16 }
```

动手写14.1.2中，使用reflect.TypeOf()获取了变量a的值类型，使用reflect.ValueOf()获取了变量a的原始值，这两个函数非常重要，我们在后面会详细介绍它们的使用，运行结果如下：

```

变量a的类型为: string
变量a的值为: 我是字符串
```

提示

Go语言不支持解析string然后执行。Go语言的反射机制只能作用于已经存在的对象上。

14.2 基本用法

在使用反射时，我们会经常使用到反射的种类（Kind），reflect.Kind在Go语言reflect包中的定

义如下：

```
type Kind uint

const (
    Invalid Kind = iota           // 非法类型
    Bool                  // 布尔型
    Int                   // 有符号整型
    Int8                 // 有符号8位整型
    Int16                // 有符号16位整型
    Int32                // 有符号32位整型
    Int64                // 有符号64位整型
    Uint                  // 无符号整型
    Uint8                // 无符号8位整型
    Uint16               // 无符号16位整型
    Uint32               // 无符号32位整型
    Uint64               // 无符号64位整型
    Uintptr              // 指针
    Float32              // 单精度浮点数
    Float64              // 双精度浮点数
    Complex64            // 64位复数类型
    Complex128           // 128位复数类型
    Array                // 数组
    Chan                 // 通道
    Func                 // 函数
    Interface            // 接口
    Map                  // 映射
    Ptr                  // 指针
    Slice                // 切片
    String               // 字符串
    Struct               // 结构体
    UnsafePointer         // 底层指针
)
```

其中Map、Slice、Chan属于引用类型，使用起来类似于指针，但是在种类常量定义中仍然属于独立的种类，不属于Ptr。type A struct{}定义的结构体属于Struct种类，*A属于Ptr。

当我们使用反射获取到变量的类型时，就是用reflect.Kind来判断所属系统类型，举个例子：

动手写14.2.1

```

01 package main
02
03 import (
04     "reflect"
05     "fmt"
06 )
07
08 func main() {
09     var a int
10     typeOfA := reflect.TypeOf(a)
11     if typeOfA.Kind() == reflect.Int {
12         fmt.Printf("变量a的Kind是int")
13     } else{
14         fmt.Printf("变量a的Kind不是int")
15     }
16 }
```

动手写14.2.1定义了一个变量a，使用反射获取a的类型Kind与reflect.Int比较，运行结果如下：

```
变量a的Kind是int
```

14.2.1 获取类型信息

Go语言中使用reflect.TypeOf()来获取变量对象的类型信息，返回值的反射类型对象（reflect.Type），使用方法如下：

```
var typeOfNum reflect.Type = reflect.TypeOf(num)
```

获取类型名称可以使用reflect.Type中的Name()方法；获取类型的种类Kind，可以使用对应的Kind()方法；对于指针类型变量，可以使用reflect.Type中的Elem()来反射获取指针指向的元素类型。

动手写14.2.2

```

01 package main
02
03 import (
04     "reflect"
05     "fmt"
```

```
06 )
07
08 type Number int
09
10 type Person struct {
11 }
12
13 func checkType(t reflect.Type) {
14     if t.Kind() == reflect.Ptr {
15         fmt.Printf("变量的类型名称%v, 指向的变量为:", t.Kind())
16         t = t.Elem()
17     }
18     fmt.Printf("变量的类型名称 => %v, 类型种类 => %v \n", t.Name(), t.Kind())
19 }
20
21 func main() {
22     var num Number = 10
23     typeOfNum := reflect.TypeOf(num)
24     fmt.Println("typeOfNum")
25     checkType(typeOfNum)
26
27     var person Person
28     typeOfPerson := reflect.TypeOf(person)
29     fmt.Println("typeOfPerson")
30     checkType(typeOfPerson)
31
32     typeOfPersonPtr := reflect.TypeOf(&person)
33     fmt.Println("typeOfPersonPtr")
34     checkType(typeOfPersonPtr)
35 }
```

动手写14.2.2中，定义了int种类的Number类型和一个Person结构体类型。checkType()函数对传入的reflect.Type变量进行判断，如果变量的种类为指针类型，就输出类型种类，并获取其指向的元素，输出类型名和类型Kind。运行结果如下：

```

typeOfNum
变量的类型名称 => Number, 类型种类 => int

typeOfPerson
变量的类型名称 => Person, 类型种类 => struct

typeOfPersonPtr
变量的类型名称ptr, 指向的变量为: 变量的类型名称 => Person, 类型种类 =>struct

```

14.2.2 获取类型的值

Go语言使用reflect.TypeOf来获取类型信息，使用reflect.ValueOf来获取变量值的信息。reflect.ValueOf()函数可以获取值的反射值对象（reflect.Value），使用方法如下：

```
var valueOfNum reflect.Value = reflect.ValueOf(num)
```

reflect.Value是反射中非常重要的类型，获取变量的值和反射调用函数都需要使用该类型。reflect.Value有一系列的获取变量值的方法，例如Int()方法用来获取int类型变量的值，如果用此函数获取其他Kind（非Int、Int8、Int16、Int32、Int64类型变量）的值将会引发panic。

动手写14.2.3

```

01 package main
02
03 import (
04     "reflect"
05     "fmt"
06 )
07
08 func checkValue(v reflect.Value) {
09     if v.Kind() == reflect.Ptr {
10
11         v = v.Elem()
12     }
13     if v.Kind() == reflect.Int {
14         // 方法一
15         var v1 int = int(v.Int())
16         // 方法二
17         var v2 int = v.Interface().(int)

```

```
18         fmt.Println(v1, v2)
19     }
20 }
21
22 func main() {
23     var num int = 10
24     valueOfNum := reflect.ValueOf(num)
25     fmt.Println("valueOfNum")
26     checkValue(valueOfNum)
27
28     valueOfNumPtr := reflect.ValueOf(&num)
29     fmt.Println("valueOfNumPtr")
30     checkValue(valueOfNumPtr)
31 }
```

动手写14.2.3使用int类型变量为例子，函数checkValue()依旧会对指针类型变量做特殊处理，使用两个方法获取变量num的值，一个是使用上面介绍的Int()函数获取，再从reflect.Value类型强制转换成int类型数据，另一个是使用Interface()方法转换成Interface{}类型，再从Interface{}类型转换成int类型数据。运行结果如下：

```
valueOfNum
10 10
valueOfNumPtr
10 10
```

14.2.3 使用反射调用函数

使用反射调用函数需要用到reflect.ValueOf()方法传入想要反射的函数名，获取到reflect.Value对象，再通过reflect.Value对象的Call方法调用该函数，Call方法的声明如下：

```
func (v Value) Call(in []Value) []Value
```

Call方法使用输入的参数in调用v持有的函数。例如，如果len(in) == 3，v.Call(in)代表调用v(in[0], in[1], in[2])（其中Value值表示其持有值）。如果v的Kind不是Func会引发panic。它返回函数所有输出结果的Value封装的切片。和Go代码一样，每一个输入实参的持有值都必须可以直接赋值给函数对应输入参数的类型。如果v持有值是可变参数函数，Call方法会自行创建一个代表可变参数的切片，将对应可变参数的值都拷贝到里面。

动手写14.2.4

```
01 package main
02
03 import (
04     "reflect"
05     "fmt"
06 )
07
08 func Equal(a, b int) bool {
09     if a == b {
10         return true
11     }
12     return false
13 }
14
15 func main() {
16     // 反射调用函数需使用ValueOf
17     valueOfFunc := reflect.ValueOf(Equal)
18
19     // 构造函数参数
20     args := []reflect.Value{reflect.ValueOf(1), reflect.ValueOf(2)}
21
22     // 通过反射调用函数计算
23     result := valueOfFunc.Call(args)
24
25     fmt.Println("函数运行结果: ", result[0].Bool())
26 }
```

动手写14.2.4定义了一个Equal函数，Equal函数传入参数a、b判断两个值是否相等，返回对应结果。主函数使用reflect.ValueOf获取reflect.Value对象，并构造了参数args传入Call函数中，运行结果如下：

```
函数运行结果:  false
```

14.3 对结构体的反射操作

反射不仅可以获取普通类型变量的值，还可以获取结构体成员的类型、成员变量的值以及调用结构体方法。

14.3.1 获取结构体成员类型

结构体通过reflect.TypeOf获取反射类型变量后，可以调用reflect.Type对象的NumField()方法获取结构体成员的数量，调用Field()则可以根据索引返回对应结构体字段的详细信息，具体信息如下：

```
type StructField struct {
    Name string           // 字段名
    PkgPath string         // 字段路径
    Type     Type          // 字段反射类型对象
    Tag      StructTag      // 字段结构体标签
    Offset   uintptr        // 字段在结构体中的偏移
    Index    []int          // 字段的索引值
    Anonymous bool          // 是否为匿名字段
}
```

reflect.Type中还可以使用FieldByName(name string)通过查找字段名来获取字段信息，以及使用FieldByIndex(index []int)通过索引值来获取字段信息。

动手写14.3.1

```
01 package main
02
03 import (
04     "reflect"
05     "fmt"
06 )
07
08 type Person struct {
09     Name string
10     Age int `json:"age"`
11     string
12 }
13
```

```

14 func main() {
15     person := Person{"小张", 10, "备注"}
16     typeOfPerson := reflect.TypeOf(person)
17
18     // 遍历所有结构体成员获取字段信息
19     fmt.Println("遍历结构体")
20     for i := 0; i < typeOfPerson.NumField(); i++ {
21         field := typeOfPerson.Field(i)
22         fmt.Printf("字段名: %v 字段标签: %v 是否是匿名字段: %v \n",
23             field.Name, field.Tag, field.Anymous)
24     }
25
26     // 通过字段名获取字段信息
27     if field, ok := typeOfPerson.FieldByName("Age"); ok {
28         fmt.Println("通过字段名")
29         fmt.Printf("字段名: %v, 字段标签中json: %v \n", field.Name,
30             field.Tag.Get("json"))
31     }
32
33     // 通过下标获取字段信息
34     field := typeOfPerson.FieldByIndex([]int{1})
35 }
```

动手写14.3.1中创建了一个Person结构体，在main函数中使用Field(i)遍历获取所有字段信息，其中包括字段名、字段标签以及是否为匿名字段。使用FieldByName()及FieldByIndex()函数获取Age字段的信息，运行结果如下：

遍历结构体

```

字段名: Name 字段标签: 是否是匿名字段: false
字段名: Age 字段标签: json:"age" 是否是匿名字段: false
字段名: string 字段标签: 是否是匿名字段: true
```

通过字段名

```
字段名: Age, 字段标签中json: age
```

通过下标

```
字段名: Age, 字段标签: json:"age"
```

提示

当我们创建结构体时，字段可以只有类型，而没有字段名，这样的字段称为匿名字段（Anonymous Field）。

14.3.2 获取结构体成员字段的值

获取结构体成员的值，我们仍需要使用reflect.ValueOf()方法获取到reflect.Value对象，使用对象的NumField()方法可以获取结构体成员的数量，使用Field()则可以根据索引返回对应结构体字段的reflect.Value反射值类型。

动手写14.3.2

```
01 package main
02
03 import (
04     "reflect"
05     "fmt"
06 )
07
08 type Person struct {
09     Name string
10     Age int `json:"age"`
11     string
12 }
13
14 func main() {
15     person := Person{"小张", 10, "备注"}
16     valueOfPerson := reflect.ValueOf(person)
17
18     fmt.Printf("person的字段数量: %v\n", valueOfPerson.NumField())
19
20     // 通过下标访问获取字段值
21     fmt.Println("Field")
22     field := valueOfPerson.Field(1)
23     fmt.Printf("字段值: %v \n", field.Int())
24 }
```

```

25     // 通过字段名获取字段值
26     field = valueOfPerson.FieldByName("Age")
27     fmt.Println("FieldByName")
28     fmt.Printf("字段值: %v \n", field.Interface())
29
30     // 通过下标索引获取字段值
31     field = valueOfPerson.FieldByIndex([]int{0})
32     fmt.Println("FieldByIndex")
33     fmt.Printf("字段值: %v \n", field.Interface())
34
35 }
```

动手写14.3.2对Person结构体变量分别使用Field()、FieldByName()、FieldByIndex()三个方法来获取结构体成员的值类型对象，并打印输出变量的值，运行结果如下：

```

person的字段数量: 3
Field
字段值: 10
FieldByName
字段值: 10
FieldByIndex
字段值: 小张
```

14.3.3 反射执行结构体方法

对于结构体方法，需要使用reflect.ValueOf()获取reflect.Value对象，然后调用该对象的MethodByName(Name)函数，找到想要反射调用的方法，再通过Call函数进行反射调用。

动手写14.3.3

```

01 package main
02
03 import (
04     "reflect"
05     "fmt"
06 )
07
08 type Person struct {
```

```
09     Name string
10     Age int `json:"age"`
11     string
12 }
13
14 func (p Person) GetName() {
15     fmt.Println(p.Name)
16 }
17
18 func main() {
19     person := Person{"小张", 10, "备注"}
20     valueOfPerson := reflect.ValueOf(person)
21
22     // 根据名字获取方法
23     f := valueOfPerson.MethodByName("GetName")
24
25     // 执行结构体方法
26     f.Call([]reflect.Value{})
27
28 }
```

运行结果如下：

```
小张
```

14.4 反射三定律

到现在为止，读者应该已经对反射的使用了解得比较清楚了。接下来我们将要介绍反射三定律，在使用反射时，牢记这三条定律会让你对反射有更加清晰的认识。

14.4.1 接口到反射类型的转换

这里的反射类型指reflect.Type和reflect.Value。反射的第一定律为：反射可以将接口类型变量转换为反射类型变量。我们来看reflect.TypeOf的定义，通过该函数，将传入的interface{}类型的变量进行解析后返回reflect.Type类型：

```
func TypeOf(i interface{}) Type
```

`reflect.ValueOf`则是将传入的`interface{}`类型的变量进行解析后返回`reflect.Value`类型：

```
func ValueOf(i interface{}) Value
```

动手写14.4.1

```
01 package main
02
03 import (
04     "fmt"
05     "reflect"
06 )
07
08 func main() {
09     var a int = 5
10
11     fmt.Printf("type:%T \n", reflect.TypeOf(a))
12     fmt.Printf("value:%T \n", reflect.ValueOf(a))
13 }
```

动手写14.4.1打印输出了`reflect.TypeOf(a)`和`reflect.ValueOf(a)`的类型，其中“%T”表示输出变量的类型，运行结果如下：

```
type:*reflect.rtype
value:reflect.Value
```

14.4.2 反射到接口类型的转换

反射的第二定律为：反射可以将反射类型变量转换为接口类型。这主要使用了`reflect.Value`对象的`Interface()`方法，可以将一个反射类型变量转换为接口变量。举个例子：

动手写14.4.2

```
01 package main
02
03 import (
04     "fmt"
05     "reflect"
06 )
07
```

```
08 func main() {
09     var a int = 5
10     valueOfA := reflect.ValueOf(a)
11
12     fmt.Println(valueOfA.Interface())
13 }
```

运行结果如下：

```
5
```

14.4.3 修改反射类型对象

反射的第三定律为：想要使用反射来修改变量的值，其值必须是可写的（CanSet）。这个值必须满足两个条件：一是变量可以被寻址（CanAddr），二是变量是可导出的（结构体字段名首字母需大写）。

对于可以被寻址，需要使用reflect.ValueOf()方法对反射对象的地址获取其reflect.Value对象，再使用Elem()获取实例地址的元素，通过此方式获取的reflect.Value是可以被寻址的，例如：

```
valueOfPerson := reflect.ValueOf(&person)
```

对于结构体而言，结构体字段必须是可导出的，不能为匿名字段，且字段名的首字母必须要大写。

动手写14.4.3

```
01 package main
02
03 import (
04     "reflect"
05     "fmt"
06 )
07
08 type Person struct {
09     Name string
10     age int `json:"age"`
11     string
12 }
```

```

13
14 func main() {
15     person := Person{"小张", 10, "备注"}
16     valueOfPerson := reflect.ValueOf(&person)
17     typeOfPerson := reflect.TypeOf(&person)
18
19     for i := 0; i < valueOfPerson.Elem().NumField(); i++ {
20         fieldValue := valueOfPerson.Elem().Field(i)
21         fieldType := typeOfPerson.Elem().Field(i)
22         fmt.Printf("类型名: %v 可以寻址: %v 可以设置: %v \n",
23                 fieldType.Name, fieldValue.CanAddr(), fieldValue.
24                 CanSet())
25     }
26     fmt.Println("修改前: ", person)
27     // 必须满足可寻址和可导出两个条件才能修改变量值
28     valueOfPerson.Elem().Field(0).SetString("xiao zhang")
29
30     fmt.Println("修改后: ", person)
31 }
```

动手写14.4.3中的结构体person中只有Name字段的值可以修改，age字段和匿名字段由于不可导出的原因导致其值不能修改，运行结果如下：

```

类型名: Name 可以寻址: true 可以设置: true
类型名: age 可以寻址: true 可以设置: false
类型名: string 可以寻址: true 可以设置: false
修改前: {小张 10 备注}
修改后: {xiao zhang 10 备注}
```

14.5 反射的性能

我们在学习反射之前就提到了反射的性能极差，在考虑性能的地方能不用反射就不用。我们接下来写一个测试代码来针对变量进行赋值和运行函数，分别对比使用反射和不使用反射的时间

效率。关于Go语言性能测试的具体使用方法会在后面的章节中学习，本节不做详细讨论，仅关注程序运行的结果。

动手写14.5.1

```
01 package main
02
03 import (
04     "reflect"
05     "testing"
06 )
07
08 type data struct {
09     N int
10 }
11
12 const AssignTimes = 100000000
13
14 func BenchmarkNativeAssign(t *testing.B) {
15     v := data{N: 1}
16     for i := 0; i < AssignTimes; i++ {
17         v.N = 3
18     }
19 }
20 func BenchmarkReflectAssign(t *testing.B) {
21     v := data{N: 1}
22     vv := reflect.ValueOf(&v).Elem()
23     for i := 0; i < AssignTimes; i++ {
24         vv.FieldByName("N").SetInt(3)
25     }
26 }
27 func foo(v int) {
28 }
29
30 const CallTimes = 100000000
31
32 func BenchmarkNativeCall(t *testing.B) {
33     for i := 0; i < CallTimes; i++ {
34         foo(i)
```

```

35      }
36  }
37 func BenchmarkReflectCall(t *testing.B) {
38     v := reflect.ValueOf(foo)
39     for i := 0; i < CallTimes; i++ {
40         v.Call([]reflect.Value{reflect.ValueOf(2)})
41     }
42 }
```

将以上代码保存，文件名以“_test.go”结尾，运行如下命令：

```
go test -bench=.
```

BenchmarkNativeAssign表示不使用反射进行赋值操作，BenchmarkReflectAssign表示使用反射进行赋值操作，BenchmarkNativeCall表示不使用反射进行函数调用，BenchmarkReflectCall表示使用常规方式调用函数。运行结果如下：

```

goos: windows
goarch: amd64
BenchmarkNativeAssign-4        2000000000    0.02 ns/op
BenchmarkReflectAssign-4       1            13194950100 ns/op
BenchmarkNativeCall-4         2000000000    0.02 ns/op
BenchmarkReflectCall-4        1            22787966400 ns/op
PASS
```

重点关注最后一列，它表示运行该函数所需的时间，可见使用反射相差了十二个数量级，反射的效率特别低。

14.6 小结

- ◊ 了解Go语言中反射的使用及反射的特性。
- ◊ 掌握反射中的关键函数（reflect.TypeOf、reflect.ValueOf）及关键类型（reflect.Type、reflect.Value）的使用。
- ◊ 掌握如何使用反射获取结构体的成员信息。
- ◊ 掌握反射三定律的使用。
- ◊ 反射大大提高了程序执行的灵活性，同时也带来了不可避免的弊端——代码可读性变差、程序执行效率变低，因此我们要合理地使用反射。

>> 第 15 章

Go 命令行工具

<<

Go语言的工具箱提供了很多常用工具。它可以被看作是特殊包管理器，用于包的查询、计算包的依赖关系、从远程版本控制系统下载包等任务。它也是一个构建系统，计算文件的依赖关系，然后调用编译器、汇编器和连接器构建程序。它还是一个单元测试、基准测试和性能测试的驱动程序，可让我们非常方便地进行各类代码测试，快速查找各类问题。

本章将详细介绍Go语言工具箱的具体功能，包括获取源码、编译、注释文档、测试、性能分析、源码格式化等。

你可以运行go或go help命令查看内置的帮助文档，为了方便查看，我们列出了最常用的命令：

```
C:\Users\admin>go help
```

```
...
```

```
The commands are:
```

build	编译打包依赖
clean	清理链接和缓存文件
doc	展示包的文档
env	打印Go环境变量信息
bug	启动Go语言调试
fix	更新文档语法
fmt	格式化文档结构
generate	通过扫描Go源码中的特殊注释来识别要运行的常规命令
get	下载安装包和依赖项
install	编译安装包和依赖项
list	查询包

run	编译运行Go程序
test	测试包
tool	运行Go的其他工具
version	打印版本号
vet	静态错误检查工具
...	

15.1

编译相关指令

在之前的学习中，我们最熟悉的Go命令行工具就是go build和go run，现在我们来系统性地学习这两个工具的使用。相较于Java和C++的编译速度，Go语言的快速编译是一个主要的效率优势。

15.1.1 build

go build指令用于编译我们的源码文件、代码包以及它们的依赖包。

我们创建相互引用的两个文件lib.go和main.go，分别对应动手写15.1.1和动手写15.1.2，两个文件同属于main包，将其放在GOPATH下的main目录，如C:\Users\admin\go\src\main。

动手写15.1.1

```
01 package main
02
03 import "fmt"
04
05 func LibFunc() {
06     fmt.Println("This file is 15.1.1.go")
07 }
```

动手写15.1.2

```
01 package main
02
03 import "fmt"
04
05 func main() {
06     LibFunc()
07     fmt.Println("This file is 15.1.2.go")
08 }
```

接下来，我们将试图编译这两个文件。go build支持三种编译方式。

1. go build（无参编译）

如果我们在执行go build命令时不后跟任何代码包，那么命令将试图编译当前目录所对应的代码包。

```
C:\Users\admin\go\src\main>go build
C:\Users\admin\go\src\main>dir
驱动器C中的卷是系统
卷的序列号是 6123-B186

C:\Users\admin\go\src\main 的目录

2019/05/16  21:00      <DIR>          .
2019/05/16  21:00      <DIR>          ..
2019/05/16  20:50            86  lib.go
2019/05/16  21:00        2,048,000 main.exe
2019/05/16  20:50            94  main.go
                           3 个文件       2,048,180 字节
                           2 个目录     19,189,739,520 可用字节
C:\Users\admin\go\src\main>main.exe
This file is 15.1.1.go
This file is 15.1.2.go
```

2. go build+文件列表

```
C:\Users\admin\go\src\main>go build -o main.exe lib.go main.go
```

使用这种方式编译时，作为参数的多个Go源码文件必须在同一个目录中。

3. go build+包

```
C:\Users\admin\go\src\main>go build main
```

使用“go build+包”的方式编译需要将编译的包放到GOPATH下，否则编译器会找不到你想要编译的包。

Go语言有个不足的地方是，简单的几行代码使用go build命令编译出来大小会达到2~3 MB。如何能减小可执行文件的体积呢？

go build命令后面加上编译参数-lldflags "-s -w"就可以减小体积。其中-s的作用是去掉符号信息，在程序panic时的stack trace就没有任何文件名/行号信息了；-w是去掉DWARF调试信息，而后

得到的程序就不能用gdb调试了。

```
C:\Users\admin\go\src\main>go build -ldflags "-s -w"

C:\Users\admin\go\src\main>dir
驱动器C中的卷是系统
卷的序列号是 6123-B186

C:\Users\admin\go\src\main 的目录

2019/05/17 20:06    <DIR>          .
2019/05/17 20:06    <DIR>          ..
2019/05/16 20:50            86 lib.go
2019/05/17 20:06        1,257,984 main.exe
2019/05/16 20:50            94 main.go
                           3 个文件      1,258,164 字节
                           2 个目录 18,966,319,104 可用字节
```

使用这个参数后，可以看见程序体积减小了40%左右。当然，如果你觉得体积仍然较大，可以使用第三方工具对Go程序进行压缩处理，如UPX压缩。UPX程序下载网址：<https://github.com/upx/upx/releases>。

```
C:\Users\admin\go\src\main>upx -9 main.exe
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2018
UPX 3.95w      Markus Oberhumer, Laszlo Molnar & John Reiser
Aug 26th 2018
```

File size	Ratio	Format	Name
1257984 -> 427520	33.98%	win64/pe	main.exe

Packed 1 file.

```
C:\Users\admin\go\src\main>dir
驱动器C中的卷是系统
卷的序列号是 6123-B186
```

C:\Users\admin\go\src\main 的目录

```
2019/05/17 20:12 <DIR> .
2019/05/17 20:12 <DIR> ..
2019/05/16 20:50 86 lib.go
2019/05/17 20:06 427,520 main.exe
2019/05/16 20:50 94 main.go
2018/08/26 17:55 406,528 upx.exe
    4 个文件      834,228 字节
    2 个目录 18,965,323,776 可用字节
```

upx -9参数表使用压缩效果最好的算法进行压缩，可以看到现在的程序体积已经是最开始的20%左右，只有400 KB左右。

go build还支持如下的附加参数。

-v	编译时显示包名
-a	强制重新构建
-n	打印输出编译时所用到的命令，但不真正执行
-x	打印输出编译时所用到的命令
-race	开启竞态检测

提示

Go语言的源码文件分为三大类：命令源码文件、库源码文件和测试源码文件。命令源码文件总是作为可执行的程序的入口，库源码文件一般用于集中放置各种待使用的程序实体，而测试源码文件主要用于对前两种源码文件中的程序实体的功能和性能进行测试。

15.1.2 run

我们在调试代码时通常会使用go run命令。该命令会编译执行Go语言源码，不会在当前目录生成可执行文件，而是生成在临时目录下。

go run的对象只能是单个或多个.go文件（必须同属于main包），且不能为测试文件，例如：

```
// 编译运行file1.go文件
go run file1.go
// 编译运行file1.go和file2.go文件
```

```
go run file1.go file2.go
// 编译运行当前目录下的所有文件
go run *.go
```

无法针对包运行 go run 指令，只能使用 go build 编译整个包，再运行编译后的可执行文件。如果 go run 指定源文件中引用了其他文件的方法或定义的变量，编译器会抛出如下 undefined 的错误。

```
# command-line-arguments
15.6.1.go:19:18: undefined: Num
```

动手写 15.1.3

```
01 package main
02
03 import (
04     "fmt"
05     "os"
06 )
07
08 func main() {
09     fmt.Println("os.Args:", os.Args)
10 }
```

动手写 15.1.3 运行后，会将可执行文件编译生成到 C:\Users\admin\AppData\Local\Temp 目录中，Linux 则是生成到临时目录 “/tmp” 里。os.Args 为命令行的参数，第一个参数为可执行文件的文件名，包括全路径，剩余参数为程序执行前传入的参数，也就是 go run 15.1.1.go 后面的参数。

```
$ go run 15.1.1.go test
os.Args: [C:\Users\admin\AppData\Local\Temp\go-build565313162\b001\
exe\15.1.1.exe test]
```

我们在使用 go run 命令时，还可以使用 -work 参数来显示当前的编译目录。

```
$go run -work 15.1.1.go
WORK=C:\Users\admin\AppData\Local\Temp\go-build161637094
os.Args: [C:\Users\admin\AppData\Local\Temp\go-build161637094\b001\
exe\15.1.1.exe]
```

go run 这个命令具体干了些什么事情呢？我们可以使用 -n 参数输出编译过程时所用到的命令。

```
$go run -n 15.1.1.go
mkdir -p $WORK\b001\
cat >$WORK\b001\importcfg.link << 'EOF' # internal
packagefile
command-line-arguments=C:\Users\admin\AppData\Local\go-build\89\8967f
6f0c99aa2f31a2f9af1afeb25d8423d2337565134e5dbc2932a1e4816c1-d
packagefile fmt=D:\Go\pkg\windows_amd64\fmt.a
.....
packagefile
internal/syscall/windows/registry=D:\Go\pkg\windows_amd64\internal\
syscall\windows\registry.a
EOF
mkdir -p $WORK\b001\exe\
cd .
"D:\\\\Go\\\\pkg\\\\tool\\\\windows_amd64\\\\link.exe" -o "$WORK\\\\b001\\\\
exe\\\\15.1.1.exe" -importcfg "$WORK\\\\b001\\\\importcfg.link" -s -w
-buildmode=exe -build
id=Ufv-aG-WEkHpp7uJxV7v/6ds1imf9o9HRVOLm78ib/mIpUYfUHqu2CIV6v4vIV/
Ufv-aG-WEkHpp7uJxV7v -extld=gcc "C:\\\\Users\\\\admin\\\\AppData\\\\Local\\\\
go-build\\\\89\\\\89
6f0c99aa2f31a2f9af1afeb25d8423d2337565134e5dbc2932a1e4816c1-d"
$WORK\b001\exe\15.1.1.exe
```

这里可以看到创建了两个临时文件夹：b001和exe。编译器先是执行了compile命令，然后是link，生成了归档文件.a和最终可执行文件，最终的可执行文件放在exe文件夹里面。命令的最后一步就是执行了可执行文件。

15.1.3 install

go install命令的作用是编译后安装，该命令依赖于GOPATH，因此不能在独立的目录中使用该命令。与go build不同的是，go install是将编译的中间件放到pkg目录下，将编译出来的可执行文件放到bin目录下。

使用动手写15.1.1和动手写15.1.2来运行go install命令。

```
C:\\\\Users\\\\admin\\\\go\\\\src\\\\main>go install
```

运行后，在目录C:\\\\Users\\\\admin\\\\go\\\\bin中就会生成可执行文件main.exe，由于这个包没有产生中

间文件，因此在目录C:\Users\admin\go\pkg下就没有该包的pkg文件。

go install 可以使用大部分 go build 的附加参数，由于 go install 的可执行文件的输出目录始终是 GOPATH 下的 bin 目录，因此不能使用 -o 参数指定输出的可执行文件名。go install 依旧可以使用 -n 参数来查看运行过程中该命令到底做了哪些事情。

```
C:\Users\admin\go\src\main>go install -n
mkdir -p $WORK\b001\
cat >$WORK\b001\importcfg.link << 'EOF' # internal
packagefile
main=C:\Users\admin\AppData\Local\go-build\1f\1ff53fb27e1485cf3189d3
4d5707415b6b4171f30ad0c527609693e601a44a1-d
.....
packagefile
internal/syscall/windows/sysdll=G:\Go\pkg\windows_amd64\internal\
syscall\windows\sysdll.a
packagefile
internal/syscall/windows/registry=G:\Go\pkg\windows_amd64\internal\
syscall\windows\registry.a
EOF
mkdir -p $WORK\b001\exe\
cd .
"G:\Go\pkg\tool\windows_amd64\link.exe" -o "$WORK\b001\exe\aa.out.exe" -importcfg "$WORK\b001\importcfg.link" -buildmode=exe -builddid=Jv2HMVpSq6xMgFZ1suB3/2KAapJGgp5g0pFVPGM5I/K-UNETzu0cuwVj6a2CmK/Jv2HMVpSq6xMgFZ1suB3 -extld=gcc "C:\Users\admin\AppData\Local\go-build\1f\1ff53fb27e1485cf3189d34d5707415b6b4171f30ad0c527609693e601a44a1-d"
"G:\Go\pkg\tool\windows_amd64\builddid.exe" -w "$WORK\b001\exe\aa.out.exe" # internal
mkdir -p C:\Users\admin\go\bin\
mv $WORK\b001\exe\aa.out.exe C:\Users\admin\go\bin\main.exe
```

可见，go install 命令只是多了将可执行文件移动到 bin 目录下的动作。

15.1.4 交叉编译

交叉编译就是在同一个平台上生成另一个平台上的可执行代码。同一个体系结构可以运行不同的操作系统。

举个例子，你使用Windows系统编写了一个Go语言程序，你的朋友想要在他的电脑上运行，可是他的电脑是Mac系统，又没有安装Go语言的编译器，这时你就可以在你的电脑上通过交叉编译的方式编译一个Mac系统可以直接运行的程序。

Go语言的交叉编译非常方便，只需在编译前设置一下Go的环境变量CGO_ENABLED、GOOS和GOARCH即可。在Windows上编译Mac程序如下：

```
01 # Windows下编译Mac
02 SET CGO_ENABLED=0
03 SET GOOS=darwin
04 SET GOARCH=amd64
05 go build main.go
```

其中GOOS为目标操作系统，GOARCH是目标操作系统的架构，CGO_ENABLED表示是否使用C语言版本的GO编译器，参数配置为0的时候就关闭C语言版本的编译器了。

注：自从Golang 1.5以后，Go语言进行了自举，也就是Go语言编译器使用Go语言编写，而不再使用C语言。

1. Windows下编译Linux

```
01 # Windows下编译Linux
02 SET CGO_ENABLED=0
03 SET GOOS=linux
04 SET GOARCH=amd64
05 go build main.go
```

2. Mac下编译Linux、Windows

```
01 # Mac下编译Linux
02 CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build main.go
03
04 # Mac下编译Windows
05 CGO_ENABLED=0 GOOS=windows GOARCH=amd64 go build main.go
```

3. Linux下编译Mac、Windows

```
01 # Linux下编译Mac
02 CGO_ENABLED=0 GOOS=darwin GOARCH=amd64 go build main.go
03
04 # Linux下编译Windows
05 CGO_ENABLED=0 GOOS=windows GOARCH=amd64 go build main.go
```

表15.1.1 各系统版本与对应的GOOS和GOARCH(系统架构)参数列表

系统	GOOS	GOARCH
Windows 32位	windows	386
Windows 64位	windows	amd64
OS X 32位	darwin	386
OS X 64位	darwin	amd64
Linux 32位	linux	386
Linux 64位	linux	amd64

15.2 代码获取 (get)

go get命令用于从远程仓库中下载安装远程代码包，这是我们常用且非常重要的指令。我们在开发程序时往往需要引用大量的第三方包，这时就需要用go get命令将其下载下来使用。例如：

```
go get github.com/gin-gonic/gin
```

大量的Go语言代码包都被托管到了github.com网站上，该网站使用git进行代码管理，其他类似的网站还有golang.org、google.golang.org、gopkg.in等。其中包的路径都是非常规范的——首先是网站路径（如github.com），其次是包的创建者，最后是包名。

go get命令会把当前的代码包下载到GOPATH中的第一个工作区的src目录并安装。

go install支持在下载过程中加入-d参数，那么程序就只会下载源码而不安装它。还有一个比较重要的参数-u，通过它就可以强制使用网络去更新包和它的依赖包。如果不使用这个参数，对存在的包运行go get命令后会什么都不执行；加了这个参数，go get就会使用git pull命令拉取最新版的代码并安装。

对于go get命令，我们仍然可以使用-n参数来查看运行过程所使用到的命令，可以发现go get其实就是使用git clone下载源码，之后再对源码进行编译安装。

我们在使用go get下载安装第三方包时，有时会因为网络问题导致下载失败，这时可以给git添加代理，以加速下载。

http代理：

```
git config --global https.proxy 'http://127.0.0.1:1080'  
git config --global http.proxy 'https://127.0.0.1:1080'
```

socks5代理：

```
git config --global http.proxy 'socks5://127.0.0.1:1080'
```

```
git config --global https.proxy 'socks5://127.0.0.1:1080'
```

取消代理：

```
git config --global --unset http.proxy
```

```
git config --global --unset https.proxy
```

其中，127.0.0.1:1080是你自己的代理IP和端口。关于代理这里就不详细介绍了。

15.3 格式化代码 (fmt)

Go语言官方为了统一代码风格，推出了gofmt工具来帮助开发者使用相同的规范格式化他们的代码。gofmt是一个cli程序，而go fmt命令是对gofmt命令进行简单的封装，它们能够格式化所有Go语言源码文件，即包括命令源码文件、库源码文件和测试源码文件。

gofmt会优先读取标准输入，如果传入了文件路径，会格式化这个文件；如果传入一个目录，会格式化目录中的所有.go文件；如果不传参数，会格式化当前目录下的所有.go文件。如下是gofmt所支持的命令参数：

```
C:\Users\admin>gofmt -h
usage: gofmt [flags] [path ...]

-cpuprofile string
    write cpu profile to this file
-d      display diffs instead of rewriting files
-e      report all errors (not just the first 10 on different lines)
-l      list files whose formatting differs from gofmt's
-r string
    rewrite rule (e.g., 'a[b:len(a)] -> a[b:]')
-s      simplify code
-w      write result to (source) file instead of stdout
```

gofmt命令需要使用参数-w，否则格式化结果不会写入文件。gofmt的对象可以是目录，如gofmt -w src/可以格式化整个项目。gofmt默认不对代码进行简化，使用-s参数可以开启简化代码功能。

go fmt命令是对gofmt命令进行简单的封装，在调用gofmt时添加了-l -w参数，相当于执行了gofmt -l -w。go fmt支持的参数只有-n和-x，如果需要更多的功能，如清除注释、简化代码、自定义缩进等，则需要使用gofmt命令。

一般我们很少使用 go fmt 命令，因为我们的开发工具在保存时有自动格式化代码的功能，底层也是调用了 go fmt 命令格式化代码。

15.4

注释文档 (doc)

Go 语言文档工具 go doc 和 go fmt 一样，也是对 godoc 的简单封装。我们通常使用 go doc 查看指定包的文档。

```
C:\Users\admin>go doc -h
Usage of [go] doc:
    go doc
    go doc <pkg>
    go doc <sym>[.<method>]
    go doc [<pkg>].<sym>[.<method>]
    go doc <pkg> <sym>[.<method>]

For more information run
    go help doc

Flags:
    -c      symbol matching honors case (paths not affected)
    -cmd   show symbols with package docs even if package is a command
    -u      show unexported symbols as well as exported
```

例如我们查看函数 fmt.Println 的文档说明：

```
C:\Users\admin>go doc fmt.Println
func Println(a ...interface{}) (n int, err error)
    Println formats using the default formats for its operands and
    writes to standard output. Spaces are always added between
    operands and a newline is appended. It returns the number of
    bytes written and any write error encountered.
```

Go 语言文档是如何写的？我们可以查看 fmt.Println 的源码文件：

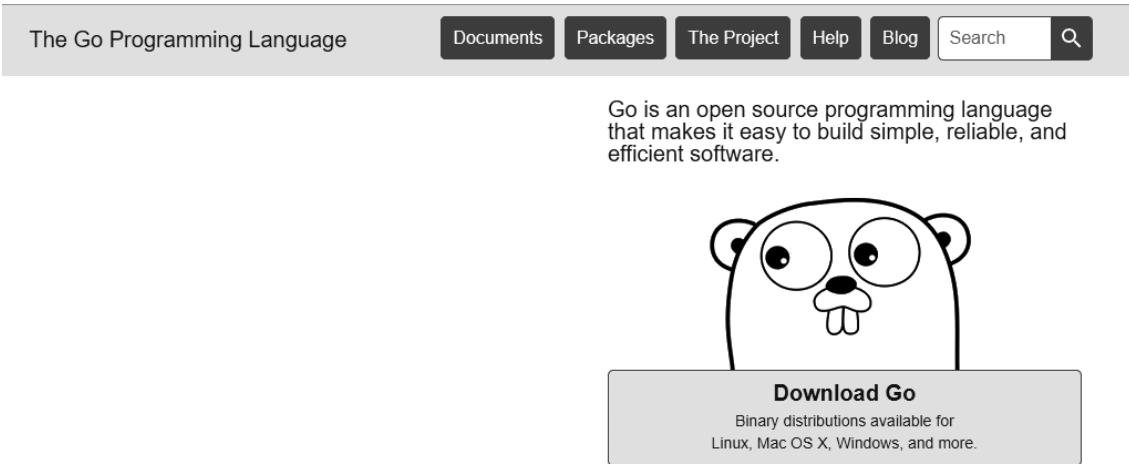
```
// Println formats using the default formats for its operands and
writes to standard output.
// Spaces are always added between operands and a newline is appended.
// It returns the number of bytes written and any write error
encountered.
func Println(a ...interface{}) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}
```

由此可以看出，Go语言的文档只需要在每个函数上方用注释的方式介绍该函数的作用及使用方法，gofmt命令就会自动将这些注释转化为文档展示出来。

gofmt命令有一个非常重要的参数–http，这个参数的作用是开启Web服务，提供交互式的文档查看页面。

```
godoc -http :8080
```

运行该命令后，浏览器访问`http://127.0.0.1:8080/`即可通过网页的方式查看Go语言文档，如图15.4.1所示。



The Go Programming Language

Documents Packages The Project Help Blog Search

Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.

Download Go
Binary distributions available for Linux, Mac OS X, Windows, and more.

Featured video

Featured articles

Read more

图15.4.1 通过网页方式查看Go语言文档

15.5 代码测试（test）

`go test`命令用于对Go语言编写的代码包进行测试。可以指定要测试的文件，也可以直接对整个包进行测试，但需要包中含有测试文件，测试文件都是以“`_test.go`”结尾。

代码测试的目的是确保代码按照你的设想执行，编写测试代码可以尽早地发现程序所存在的问题。

15.5.1 单元测试

在Go语言中可以使用测试框架`test`编写单元测试，使用命令行即可对代码进行测试及输出测试报告。

单元测试的目的是发现产品代码是否存在功能性问题，代码是否存在缺陷、是否按预期运行，因此，一个测试函数的结果要么是通过（PASS），要么是失败（FAIL）。和其他语言相比，Go语言的测试是非常高效的，开发者不用花费太多的时间就能写出较为全面的测试代码。

要编写测试代码就需要导入测试框架`testing`。单元测试的函数名使用“`Test`”为前缀，后面为要测试的函数名，并且需要传入`*testing.T`类型的参数。例如要编写函数`Add`的测试代码，需要使用如下方式：

```
func TestAdd(t *testing.T) {
    // 测试代码
}
```

动手写15.5.1

```
01 package main
02
03 const PI float64 = 3.14
04
05 // 两数相加
06 func Add(a, b int) int {
07     return a + b
08 }
09
10 // 求圆的面积
11 func Area(r float64) float64 {
12     return PI * r * r
13 }
```

动手写15.5.1定义了两个函数Add和Area，我们现在来尝试编写关于这两个函数的单元测试代码。例如对Add函数的测试代码如下，其中t.Error表示在测试不通过时，打印输出错误信息。

```
func TestAdd(t *testing.T) {
    if Add(1, 3) != 4 {
        t.Error("Add函数存在问题!")
    }
}
```

如果测试不通过，就会打印如下信息：

```
> go test
--- FAIL: TestAdd (0.00s)
    15.5.2_test.go:7: Add函数存在问题!
FAIL
FAIL    command-line-arguments 0.651s
```

TestAdd测试函数使用了一个测试数据，然而我们在测试函数时，往往需要测试更多不同维度的数据，如果每组数据都是使用以上方式，代码就会显得冗余且不美观，这时我们可以使用如下方式，给多组数据进行测试。

```
func TestAdd(t *testing.T) {
    inputs := [][]int{{1, 2}, {2, 3}, {4, 7}}
    expected := []int{3, 5, 11}
    for i := 0; i < len(inputs); i++ {
        ret := Add(inputs[i][0], inputs[i][1])
        if ret != expected[i] {
            t.Errorf("input is %d, the expected is %d, the
actual %d",
                    inputs[i], expected[i], ret)
        }
    }
}
```

inputs为函数的参数数组，expected为使用该函数期望得到的返回值，以上代码测试了三组数据。当然，如果想要测试更多数据，直接在inputs和expected添加就可以了，完整的测试代码如动手写15.5.2所示。

动手写 15.5.2

```

01 package main
02
03 import "testing"
04
05 func TestAdd(t *testing.T) {
06     inputs := [][]int{{1, 2}, {2, 3}, {4, 7}}
07     expected := []int{3, 5, 11}
08     for i := 0; i < len(inputs); i++ {
09         ret := Add(inputs[i][0], inputs[i][1])
10         if ret != expected[i] {
11             t.Errorf("input is %d, the expected is %d, the
12                             actual %d",
13                             inputs[i], expected[i], ret)
14         }
15     }
16
17 func TestArea(t *testing.T) {
18     inputs := []float64{1, 2, 4}
19     expected := []float64{3.14, 12.56, 50.24}
20     for i := 0; i < len(inputs); i++ {
21         ret := Area(inputs[i])
22         if ret != expected[i] {
23             t.Errorf("input is %f, the expected is %f, the
24                             actual %f",
25                             inputs[i], expected[i], ret)
26         }
27     }

```

测试代码编写完后，可以使用 go test 命令对指定文件进行单元测试，-v 参数表示输出详细信息，可以看见所有函数都通过了测试，结果为 PASS。

```

> go test 15.5.1.go 15.5.2_test.go -v
==== RUN    TestAdd

```

```
--- PASS: TestAdd (0.00s)
==== RUN TestArea
--- PASS: TestArea (0.00s)
PASS
ok      command-line-arguments 0.613s
```

15.5.2 基准测试

基准测试提供可自定义的计时器和一套基准测试算法，能方便快速地分析一段代码可能存在的CPU性能和内存分配问题。类似于单元测试，使用者无须准备高精度的计时器和各种分析工具，基准测试本身即可打印出非常标准的测试报告。

基准测试的使用也有一套书写规范，使用“Benchmark”为前缀，后面加上需要测试的函数名，并使用*testing.B作为函数参数，如下所示。

```
func BenchmarkFunc(b *testing.B) {
    b.ResetTimer()
    for idx := 0; idx < b.N; idx++ {
        // 要测试的代码
    }
    b.StopTimer()
}
```

有些测试代码需要一定的初始化时间，如果从Benchmark()函数开始计时，很大程度上会影响测试结果的精准性。testing.B提供了可以方便地控制计时器的一系列方法，从而让计时器只在需要的区间进行测试。ResetTimer()表示重置计时器，StopTimer()表示停止计时，需要精准测试的代码只需要放到这两个函数中间即可。

其中b.N表示必须循环b.N次，b.N这个数字会在运行中调整，以便最终达到合适的时间消耗，方便计算出合理的数据。

通过基准测试，我们能够知道使用哪种编码方式会让程序的运行效率最高。举个例子，针对字符串拼接这个场景，有非常多的实现方式，比如使用标准库中的fmt.Sprintf、使用字符串连接符“+”直接相加、使用BytesBuf方式等。想要知道哪种方式才是性能最优的选择，可以用基准测试来判断。

动手写15.5.3

```
01 package main
02
```

```
03 import (
04     "bytes"
05     "fmt"
06     "strconv"
07     "strings"
08     "testing"
09 )
10
11 const Num = 10000
12
13 func Benchmark.Sprintf(b *testing.B) {
14     b.ResetTimer()
15     for idx := 0; idx < b.N; idx++ {
16         var str string
17         for i := 0; i < Num; i++ {
18             str = fmt.Sprintf("%s%d", str, i)
19         }
20     }
21     b.StopTimer()
22 }
23 func Benchmark.StringBuilder(b *testing.B) {
24     b.ResetTimer()
25     for idx := 0; idx < b.N; idx++ {
26         var builder strings.Builder
27         for i := 0; i < Num; i++ {
28             builder.WriteString(strconv.Itoa(i))
29         }
30         _ = builder.String()
31     }
32     b.StopTimer()
33 }
34
35 func Benchmark.BytesBuf(b *testing.B) {
36     b.ResetTimer()
```

```
37     for idx := 0; idx < b.N; idx++ {
38         var buf bytes.Buffer
39         for i := 0; i < Num; i++ {
40             buf.WriteString(strconv.Itoa(i))
41         }
42         _ = buf.String()
43     }
44     b.StopTimer()
45 }
46
47 func BenchmarkStringAdd(b *testing.B) {
48     b.ResetTimer()
49     for idx := 0; idx < b.N; idx++ {
50         var str string
51         for i := 0; i < Num; i++ {
52             str += strconv.Itoa(i)
53         }
54     }
55     b.StopTimer()
56 }
```

动手写15.5.3分别针对几种不同的字符串连接方式进行基准测试，使用for循环连接10000个数字。运行go test -bench=.就可以获取结果。

```
> go test -bench=.
goos: windows
goarch: amd64
BenchmarkPrintf-4           50000          24345 ns/op
BenchmarkStringBuilder-4    1000000        2093 ns/op
BenchmarkBytesBuf-4          1000000        2022 ns/op
BenchmarkStringAdd-4         200000        8384 ns/op
PASS
ok   _/C_/Users/icx/Desktop/go语言/代码示例/16/15.5  8.078s
```

可以发现，使用StringBuilder和BytesBuf这两种字符串连接的效率最优，而使用Printf连接的效

率最差。

15.5.3 覆盖率测试

覆盖率测试用于统计通过运行程序包的测试有多少代码得到了执行。如果执行测试代码有70%的语句得到了运行，则测试覆盖率为70%。使用`-cover`参数就可以对源码文件进行覆盖率测试。

```
go test -cover xxxx.go
```

例如，我们有如下`Level`函数需要进行测试，`Level`函数根据不同的分数进行不同的评级。

动手写15.5.4

```
01 package main
02
03 func Level(point int) string {
04     switch {
05     case point < 60:
06         return "不及格"
07     case point < 70:
08         return "及格"
09     case point < 80:
10         return "良"
11     case point < 100:
12         return "优秀"
13     default:
14         return "数据出错"
15     }
16 }
```

针对动手写15.5.4，我们编写如下测试代码，分别测试分数为20、65、75时，程序是否运行正确。

动手写15.5.5

```
01 package main
02
03 import "testing"
04
05 func TestLevel(t *testing.T) {
```

```
06     inputs := []int{20, 65, 75}
07     expected := []string{"不及格", "及格", "良"}
08     for i := 0; i < len(inputs); i++ {
09         ret := Level(inputs[i])
10         if ret != expected[i] {
11             t.Errorf("input is %d, the expected is %s, the
12                             actual %s",
13                         inputs[i], expected[i], ret)
14         }
15     }
```

动手写15.5.5中，由于只测试了“不及格”“及格”“良”这三种情况的数据，而“优秀”和“数据出错”这两种情况未测试到，因此覆盖率约为66%。运行结果如下：

```
> go test -cover 15.5.4.go 15.5.5_test.go -v
== RUN TestLevel
--- PASS: TestLevel (0.00s)
PASS
coverage: 66.7% of statements
ok command-line-arguments 0.594s coverage: 66.7% of statements
```

15.6 性能分析

性能分析和调优是一种强大的技术，用于验证是否满足实际性能要求。Go语言支持使用go tool pprof工具进行性能查看及调优。性能分析工具可以将程序的CPU耗用、内存分配、竞态等问题以图形化方式展现出来。在使用这个工具前，我们需要安装其依赖的图形绘制工具Graphviz。

15.6.1 安装Graphviz

Graphviz是AT&T实验室开发的图形绘制工具，它可以很方便地用来绘制结构化的图形网络，支持多种格式输出，生成图片的质量和速度都不错。Graphviz本身是开源的产品，支持Mac、Windows、Linux平台，我们可以在这个网址中下载：<http://www.graphviz.org/download/>。

下载安装后需要配置Graphviz的安装目录如下：

```
C:\Program Files (x86)\Graphviz2.38\bin
```

添加到环境变量PATH中，以便系统能找到Graphviz的cli程序dot。

```
C:\Users\admin>dot -V
dot - graphviz version 2.38.0 (20140413.2041)
```

15.6.2 通过文件方式

为了能分析Go程序的性能，我们需要在程序中导入runtime/pprof来生成性能分析所需要的profile文件。

pprof包提供了StartCPUProfile(w io.Writer)接口来输出CPU性能参数到文件中。如下是针对CPU的性能参数获取代码：

```
f, _ := os.Create("cpu_file.prof")
// 开始cpu profile, 结果写到文件f中
pprof.StartCPUProfile(f)
// 结束profile
defer pprof.StopCPUProfile()
```

对于内存堆栈可以使用pprof包的WriteHeapProfile(w io.Writer)接口，用法和StartCPUProfile类似，这里就不举例了。

动手写15.6.1

```
01 package main
02
03 import (
04     "fmt"
05     "os"
06     "runtime/pprof"
07 )
08
09 const Num int = 10000
10
11 func main() {
12     f, _ := os.Create("cpu_file.prof")
```

```
13     // 开始cpu profile, 结果写到文件f中
14     pprof.StartCPUProfile(f)
15     // 结束profile
16     defer pprof.StopCPUProfile()
17
18     var str string
19     for i := 0; i < Num; i++ {
20         str = fmt.Sprintf("%s%d", str, i)
21     }
22 }
```

运行动手写15.6.1后会在当前路径下生成cpu_file.prof文件，这时我们就可使用go tool pprof + cpu.prof工具对整个文件进行性能分析。

```
> go tool pprof cpu_file.prof
Main binary filename not available.
Type: cpu
Time: May 19, 2019 at 2:17pm (CST)
Duration: 212.11ms, Total samples = 60ms (28.29%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 60ms, 100% of 60ms total
Showing top 10 nodes out of 29
      flat      flat%      sum%      cum      cum%
  20ms  33.33%  33.33%  20ms  33.33% runtime.memclrNoHeapPointers
  10ms  15.67%  50.00%  10ms  15.67% fmt.(*pp).free
  10ms  15.67%  66.67%  10ms  15.67% runtime.(*bucket).mp
  10ms  15.67%  83.33%  10ms  15.67% runtime.heapBits.next
  10ms  15.67%   100%  10ms  15.67% runtime.osyield
    0      0%   100%  20ms  33.33% fmt.Sprintf
    0      0%   100%  20ms  33.33% main.main
    0      0%   100%  20ms  33.33%
                                         runtime.(*mcache).nextFree.func1
    0      0%   100%  20ms  33.33% runtime.(*mcache).refill
    0      0%   100%  20ms  33.33% runtime.(*mcentral).cacheSpan
```

top命令的作用是输出最耗时的十处代码，可以发现runtime.memclrNoHeapPointers操作最耗时，由于这个函数是标准库runtime中的代码，我们无法对其进行代码优化，这时我们可以对main.main进行分析，使用list命令就可以查看费时的代码在哪一行。

```
(pprof) list main.main
Total: 60ms
ROUTINE ===== main.main in 15.6.1.go
  0      20ms (flat, cum) 33.33% of Total
  .
  .      15:      // 结束profile
  .
  .      16:      defer pprof.StopCPUProfile()
  .
  .      17:
  .
  .      18:      var str string
  .
  .      19:      for i := 0; i < Num; i++ {
  .
  .      20:          str = fmt.Sprintf("%s%d", str, i)
  .
  .      21:      }
  .
  .      22:}
```

分析得知第20行代码fmt.Sprintf耗时最多，花费了20 ms，我们就可以针对这里的逻辑进行优化，使用其他方式的字符串连接。不仅如此，我们还可以使用web命令来查看图形界面的程序执行流程，如图15.6.1所示（见第282页）。

```
(pprof) web
```

15.6.3 通过HTTP方式

通过文件的方式我们可以对大部分cli程序进行性能分析，但如果是Web程序，上述方法就有点捉襟见肘了。Go语言给我们提供了另一个针对Web程序的实时性能分析方式，只需要使用匿名导入的方式引入如下包并启动HTTP服务器即可。

```
import _ "net/http/pprof"
```

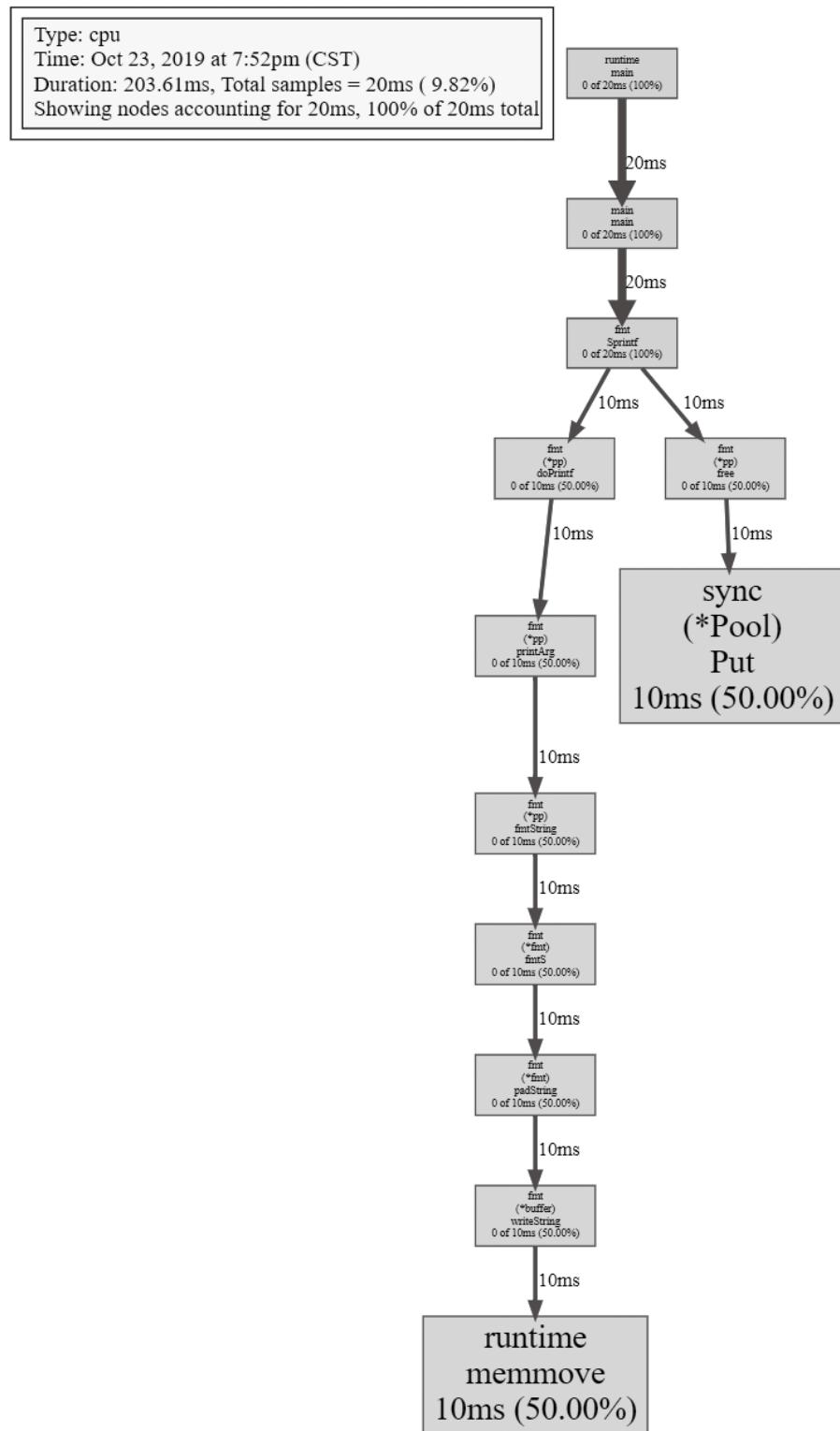


图15.6.1 图形界面的程序执行流程

动手写15.6.2

```

01 package main
02
03 import (
04     "net/http"
05     "fmt"
06     "log"
07     _ "net/http/pprof"
08 )
09
10 const Num int = 10000
11
12 func concat_str(writer http.ResponseWriter, request *http.
Request) {
13     var str string
14     for i := 0; i < Num; i++ {
15         str = fmt.Sprintf("%s%d", str, i)
16     }
17 }
18
19 func main() {
20     http.HandleFunc("/str", concat_str)
21     log.Fatal(http.ListenAndServe(":8080", nil))
22 }
```

动手写15.6.2在本地127.0.0.1的8080端口开启了一个HTTP服务器，并注册了一个/str路由，浏览器只要访问http:// 127.0.0.1:8080/str，就会执行concat_str函数。

由于我们导入了net/http/pprof包，在浏览器访问如下地址，就可以看到相关的性能参数。

```
http://127.0.0.1:8080/debug/pprof/
```

使用如下命令对CPU性能进行分析：

```
go tool pprof http://127.0.0.1:8080/debug/pprof/profile?seconds=5
```

pprof工具会等待5秒，持续获取这5秒内的CPU运行状况，默认seconds参数为30秒，运行后就需要在这5秒内持续访问http:// 127.0.0.1:8080/str来获取concat_str的CPU性能参数。

```
> go tool pprof http://127.0.0.1:8080/debug/pprof/profile?seconds=5
Fetching profile over HTTP from http://127.0.0.1:8080/debug/pprof/
profile?seconds=5
Saved profile in C:\Users\admin\pprof\pprof.samples.cpu.005.pb.gz
Type: cpu
Time: May 19, 2019 at 2:44pm (CST)
Duration: 5.01s, Total samples = 560ms (11.18%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 370ms, 66.07% of 560ms total
Showing top 10 nodes out of 88
      flat      flat%     sum%     cum      cum%
100ms  17.86%  17.86%  100ms  17.86%  runtime.memmove
  80ms  14.29%  32.14%  130ms  23.21%  runtime.scanobject
  70ms  12.50%  44.64%   70ms  12.50%  runtime.memclrNoHeapPointers
  30ms   5.36%  50.00%   30ms   5.36%  runtime.scanblock
  20ms   3.57%  53.57%   40ms   7.14%  runtime.(*mspan).sweep
  20ms   3.57%  57.14%   20ms   3.57%  runtime.heapBitsForObject
  20ms   3.57%  60.71%   20ms   3.57%  runtime.osyield
  10ms   1.79%  62.50%   60ms   0.71%  fmt.(*pp).fmtString
  10ms   1.79%  64.29%   10ms   1.79%  fmt.(*pp).free
  10ms   1.79%  66.07%   10ms   1.79%  runtime.(*fixalloc).alloc
```

至此，分析性能的方式和通过文件的方式一致。

15.7 小结

- ◇ go run/build可以方便我们构建系统，计算文件的依赖关系，然后调用编译器、汇编器和连接器构建程序。
- ◇ go get/install可以获取安装远程代码，方便包管理。
- ◇ go fmt能以Go语言编码标准来格式化我们写的代码。
- ◇ go doc可以从注释中生成文档并展示出来。
- ◇ go test可用来进行代码测试、覆盖率测试。
- ◇ go tool pprof可用来对程序进行性能测试。
- ◇ 熟练掌握Go语言工具能很大程度地提高我们的编码效率。

正则表达式

<<

正则表达式（Regular Expression，在代码中常简写为regex、regexp或RE），又称正规表示式、正规表示法、正规表达式、规则表达式、常规表示法，是计算机科学的一个概念。目前，很多编程语言都支持正则表达式。

16.1 正则表达式介绍

正则表达式，即符合一定规则的表达式，是用于匹配字符串中字符组合的模式。正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。在很多文本编辑器里，正则表达式通常被用来检索、替换那些匹配某个模式的文本（字符串）。

字符串是在编程中多有涉及的一种数据结构，所以对字符串的操作也是各式各样，而且形式多变，那么如何方便快捷地处理字符串就是我们要学习的重中之重了。

例如，我们要判断用户输入的E-mail地址是否合法，如果我们不使用正则表达式来判断，就得通过自定义一个函数提取“@”关键字，再分割其前后的字符串，然后分别判断是否合法等步骤来实现。又例如在各大网站注册用户时常看到的对用户名的要求（如6~18个字符，可使用字母、数字、下划线，需以字母开头等），如果不是用正则表达式来判断，我们就要写一堆麻烦的代码来判断用户输入的用户名是否合法。这样的代码不但冗长、不能一目了然，而且还难以重复利用，若要应对多变的需求就更不方便维护了。

但是，有了正则表达式便简单多了，正则表达式正是为这种匹配判断文本类型的工作而诞生的。

正则表达式的设计思想就是使用一些描述性的符号和文字为字符串定义一个规则。凡是符合这个规则的，程序就认为文本是“匹配”的，否则就认为文本是“不匹配”的。通俗地讲，正则

表达式的匹配过程就是逐字匹配表达式的描述规则，如果每个字符都匹配，那么程序就认为匹配成功；只要有一个匹配不成功，那么匹配就失败。

16.2 正则表达式语法

16.2.1 普通字符

普通字符是正则表达式中最基本的结构之一，要理解正则表达式自然也要从普通字符开始。本小节主要介绍普通字符。

普通字符包括没有显式指定为元字符的所有可打印和不可打印字符。这包括所有大写和小写字母、所有数字、所有标点符号和一些其他符号。

我们先以数字作为例子。假设我们想判断一个长度为1的字符串是否是数字（即这个位置上的字符只能是“0” “1” “2” “3” “4” “5” “6” “7” “8” “9” 这十个数字），如果我们使用程序去判断，那么一个可能的思路是用十个条件分支去判断这个字符串是否等于这十个字符。伪代码如下：

```
num == "0" or num == "1" or num == "2" ..... or num == "9"
```

这不失为一种有效的方法，但是太过于烦琐。如果我们判断的是英文字母或者长度非常长并且可能是各种字符混合的字符串时，代码就几乎无法阅读了。但是我们使用普通字符就可以非常简单地解决此类问题。

```
[0123456789]
```

这就是判断一个长度为1的字符串是否是数字的正则表达式。方括号 “[]” 表示这是一个字符组，代表一位字符。方括号中的数字 “0123456789” 表示只要待匹配的字符串与其中任何一个字符相同，那么程序就会认为匹配成功，反之则认为匹配失败。

当然还有更简单的写法：

```
[0-9]
```

如果符合规则的字符范围是连续的，我们可以用 “-” 省略中间的字符，相当于汉字中的“到”，直接可以读成“零到九”。

为什么是“0-9”而不是“9-0”呢？要理解这个问题，必须要了解字符的本质。在正则中，所有的字符类型都有对应的编码。图16.2.1是一张ASCII编码表，“0”对应的是十进制的48，“9”对应的是十进制的57。码值小的在前，码值大的在后，所以判断数字须写成“[0-9]”。

Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char
0	0000 0000	00	[NUL]	32	0010 0000	20	space	64	0100 0000	40	€	96	0110 0000	60	`
1	0000 0001	01	[SOH]	33	0010 0001	21	!	65	0100 0001	41	A	97	0110 0001	61	a
2	0000 0010	02	[STX]	34	0010 0010	22	"	66	0100 0010	42	B	98	0110 0010	62	b
3	0000 0011	03	[ETX]	35	0010 0011	23	#	67	0100 0011	43	C	99	0110 0011	63	c
4	0000 0100	04	[EOT]	36	0010 0100	24	\$	68	0100 0100	44	D	100	0110 0100	64	d
5	0000 0101	05	[ENQ]	37	0010 0101	25	%	69	0100 0101	45	E	101	0110 0101	65	e
6	0000 0110	06	[ACK]	38	0010 0110	26	&	70	0100 0110	46	F	102	0110 0110	66	f
7	0000 0111	07	[BEL]	39	0010 0111	27	'	71	0100 0111	47	G	103	0110 0111	67	g
8	0000 1000	08	[BS]	40	0010 1000	28	(72	0100 1000	48	H	104	0110 1000	68	h
9	0000 1001	09	[TAB]	41	0010 1001	29)	73	0100 1001	49	I	105	0110 1001	69	i
10	0000 1010	0A	[LF]	42	0010 1010	2A	*	74	0100 1010	4A	J	106	0110 1010	6A	j
11	0000 1011	0B	[VT]	43	0010 1011	2B	+	75	0100 1011	4B	K	107	0110 1011	6B	k
12	0000 1100	0C	[FF]	44	0010 1100	2C	,	76	0100 1100	4C	L	108	0110 1100	6C	l
13	0000 1101	0D	[CR]	45	0010 1101	2D	-	77	0100 1101	4D	M	109	0110 1101	6D	m
14	0000 1110	0E	[SO]	46	0010 1110	2E	.	78	0100 1110	4E	N	110	0110 1110	6E	n
15	0000 1111	0F	[SI]	47	0010 1111	2F	/	79	0100 1111	4F	O	111	0110 1111	6F	o
16	0001 0000	10	[DLE]	48	0011 0000	30	0	80	0101 0000	50	P	112	0111 0000	70	p
17	0001 0001	11	[DC1]	49	0011 0001	31	1	81	0101 0001	51	Q	113	0111 0001	71	q
18	0001 0010	12	[DC2]	50	0011 0010	32	2	82	0101 0010	52	R	114	0111 0010	72	r
19	0001 0011	13	[DC3]	51	0011 0011	33	3	83	0101 0011	53	S	115	0111 0011	73	s
20	0001 0100	14	[DC4]	52	0011 0100	34	4	84	0101 0100	54	T	116	0111 0100	74	t
21	0001 0101	15	[NAK]	53	0011 0101	35	5	85	0101 0101	55	U	117	0111 0101	75	u
22	0001 0110	16	[SYN]	54	0011 0110	36	6	86	0101 0110	56	V	118	0111 0110	76	v
23	0001 0111	17	[ETB]	55	0011 0111	37	7	87	0101 0111	57	W	119	0111 0111	77	w
24	0001 1000	18	[CAN]	56	0011 1000	38	8	88	0101 1000	58	X	120	0111 1000	78	x
25	0001 1001	19	[EM]	57	0011 1001	39	9	89	0101 1001	59	Y	121	0111 1001	79	y
26	0001 1010	1A	[SUB]	58	0011 1010	3A	:	90	0101 1010	5A	Z	122	0111 1010	7A	z
27	0001 1011	1B	[ESC]	59	0011 1011	3B	;	91	0101 1011	5B	[123	0111 1011	7B	{
28	0001 1100	1C	[FS]	60	0011 1100	3C	<	92	0101 1100	5C	\	124	0111 1100	7C	
29	0001 1101	1D	[GS]	61	0011 1101	3D	=	93	0101 1101	5D]	125	0111 1101	7D	}
30	0001 1110	1E	[RS]	62	0011 1110	3E	>	94	0101 1110	5E	^	126	0111 1110	7E	~
31	0001 1111	1F	[US]	63	0011 1111	3F	?	95	0101 1111	5F	_	127	0111 1111	7F	[DEL]

图16.2.1 ASCII编码表

同理，如果想判断一个长度为1的字符串是不是英文小写字母，我们可以写成：

[a-z]

注意：虽然ASCII编码表中大写字母在小写字母之前，但是我们并不应该用“[A-z]”来包括所有大小写英文字母，因为在这个范围中也包含了其他特殊字符，严谨的方法应该是 “[A-Za-z]” 或者 “[a-zA-Z]”。

那么如何判断一个长度为2的字符串是否是数字呢？

[0-9] [0-9]

没错，只要写两遍就行了（下一节将会介绍更简捷的办法）。假设想判断用户输入的是 “Y” 还是 “y”，正则只需写成：

[YY]

当允许的字符范围只有一个时可以省略“[]”。例如，判断用户输入的是“Yes”还是“yes”：

```
[Yy]es
```

16.2.2 字符转义

在上一小节中，我们看到代表数字范围“0”到“9”时使用的是“[0-9]”，其中“-”表示范围，并不代表字符“-”本身，此类字符我们称之为元字符。不只是“-”，例子中“[”“]”都是元字符，这些字符在匹配中都有着特殊的意义。那么如果我们想匹配“-”字符本身的话，就需要做特殊处理了。

其实在正则中，这类字符转义都有通用的方法，就是在字符前加上“\”。例如，匹配“[”本身，正则可以写成：

```
[\[]
```

如果想匹配“0”“-”和“9”这三个字符，则可以写成：

```
[0\^-9]
```

这样就只会匹配“0”“-”和“9”三个字符，而不是“0”到“9”十个字符。

16.2.3 元字符

元字符就如上文所说，是在正则表达式中有特殊意义的字符。表16.2.1列出了正则中常见的元字符。

表16.2.1 常见元字符

元字符	说明
\	将下一个字符标记为特殊字符或字面值。例如，n 匹配字符 n，而\n 匹配换行符；序列\\ 匹配\，而\() 匹配(
^	匹配输入的开始部分
\$	匹配输入的结束部分
*	零次或更多次匹配前面的字符。例如，zo* 匹配 z 或 zoo
+	一次或更多次匹配前面的字符。例如，zo+ 匹配 zoo，但是不匹配 z
?	零次或一次匹配前面的字符。例如，a?ve? 匹配 never 中的 ve
.	匹配任何单个字符，但换行符除外
(pattern)	匹配模式并记住匹配项。通过使用以下代码，匹配的子串可以检索自生成的匹配项集合：Item [0]...[n]。要匹配圆括号字符 ()，请使用\()或\\()

(续上表)

元字符	说明
x y	匹配 x 或 y。例如 z wood 匹配 z 或 wood, (z w)oo 匹配 zoo 或 woo
{n}	n 是一个非负整数, 精确匹配 n 次。例如, o{2} 不匹配 Bob 中的 o, 但是匹配 fooood 中的前两个 o
{n,}	在此表达式中, n 是一个非负整数, 至少 n 次匹配前面的字符。例如, o{2,} 不匹配 Bob 中的 o, 但是匹配 fooood 中的所有 o。o{1,} 表达式等效于 o+, o{0,} 等效于 o*
{n,m}	m 和 n 变量是非负整数。至少 n 次且至多 m 次匹配前面的字符。例如, o{1,3} 匹配 fooooood 中的前三个 o。o{0,1} 表达式等效于 o?
[xyz]	一个字符集, 匹配任意一个包含的字符。例如, [abc] 匹配 plain 中的 a
[^xyz]	一个否定字符集, 匹配任意未包含的字符。例如, [^abc] 匹配 plain 中的 p
[a-z]	字符范围, 匹配指定范围中的任意字符。例如, [a-z] 匹配英语字母中的任意小写字母字符
[^m-z]	一个否定字符范围, 匹配未在指定范围中的任意字符。例如, [^m-z] 匹配未在范围 m 到 z 之间的任意字符
\A	仅匹配字符串的开头
\b	匹配某个单词边界, 即某个单词和空格之间的位置。例如, er\b 匹配 never 中的 er, 但是不匹配 verb 中的 er
\B	匹配非单词边界。ea*\B 表达式匹配 never early 中的 ear
\d	匹配数字字符
\D	匹配非数字字符
\f	匹配换页字符
\n	匹配换行符
\r	匹配回车字符
\s	匹配任意空白字符, 包括空格、制表符、换页字符等
\S	匹配任意非空白字符
\t	匹配跳进字符
\v	匹配垂直跳进字符
\w	匹配任意单词字符, 包括下划线。此表达式等效于 [A-Za-z0-9_]
\W	匹配任意非单词字符。此表达式等效于 [^A-Za-z0-9_]
\z	仅匹配字符串的结尾
\Z	仅匹配字符串的结尾, 或者结尾的换行符之前

元字符数量较多, 本章后面几节将会对常见元字符展开介绍。

16.2.4 限定符

限定符指定输入中必须存在字符、组或字符类的多少个实例才能找到匹配项。上一小节中的“*” “+” “?” “{n}” “{n,}” 和 “{n, m}” 都是限定符。

我们先来介绍 “{n}”。“{n}” 限定符表示匹配上一元素n次，其中n是任意非负整数。例如，“y{5}” 只能匹配 “yyyyy”；“3{2}” 则只能匹配 “33”；“\w{3}” 可以匹配任意三位英文字母，“yes” “Yes” “abc” 和 “ESC” 都是可以匹配的，但是 “No” “123” “No1” 都不能被匹配。

“{n,}” 限定符表示至少匹配上一元素n次，其中n是任意非负整数。例如，“y{3,}” 可以匹配 “yyy” 也可以匹配 “yyyyyy”。同理，“[0-9]{3,}” 可以匹配任意三位数及数位为三以上的数字。

“{n, m}” 限定符表示至少匹配上一元素n次，但不超过m次，其中n和m是非负整数。例如，“y{2,4}” 可以匹配 “yy” “yyy” 和 “yyyy”。同理，“[0-9]{8,11}” 表示可以匹配任意八位至十一位的数字。

“*” 限定符表示与前面的元素匹配零次或多次，它相当于 “{0,}” 限定符。例如，“91*9*” 可以匹配 “919” “9119” “9199999” 等，但是不能匹配 “9129” “929” 等。

“+” 限定符表示匹配上一元素一次或多次，它相当于 “{1,}” 限定符。例如，“an\w+” 可以匹配 “antrum” 等以 “an” 开头的字母数量为三及以上的单词，但是不能匹配 “an”。

“?” 限定符表示匹配上一元素零次或一次，它相当于 “{0,1}” 。例如，“an?” 可以匹配 “a” 和 “an”，但是不能匹配 “antrum”。

16.2.5 定位符

定位符能够将正则表达式固定到行首或行尾。它们还能够创建这样的正则表达式：正则表达式将出现在一个单词内、一个单词的开头或者一个单词的结尾。

定位符用来描述字符串或单词的边界，“^” 和 “\$” 分别指字符串的开始与结束，“\b” 描述单词的前或后边界，“\B” 表示非单词边界。正则表达式的定位符如表16.2.2所示。

表16.2.2 正则表达式中的定位符

字符	说明
^	匹配输入字符串的开始位置，如果设置了 Multiline 属性，^ 也匹配 \n 或 \r 之后的位置。除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配字符 ^ 本身，请使用 \^
\$	匹配输入字符串的结尾位置。如果设置了 Multiline 属性，\$ 还会与 \n 或 \r 之前的位置匹配
\b	匹配一个单词边界，即单词与空格间的位置
\B	非单词边界匹配

“^” 定位符指定以下模式必须从字符串的第一个字符位置开始。例如，“\w+” 可以匹配 “123abc” 中的 “abc”，“^w+” 则不能匹配 “123abc”，但是可以匹配 “abc123” 中的

“abc”，因为整个字符串必须是以字母开头。

“\$”定位符指定前面的模式必须出现在输入字符串的末尾，或出现在输入字符串末尾的“\n”之前。例如，“\w+”可以匹配“abc123”中的“abc”，“\w+\$”则不能匹配“abc123”，但是可以匹配“123abc”，因为整个字符串必须是以字母结尾。

“\b”定位符指定匹配必须出现在单词字符（“\w”语言元素）和非单词字符（“\W”语言元素）之间的边界上。单词字符包括字母、数字、字符和下划线；非单词字符包括不为字母、数字、字符或下划线的任意字符。匹配也可以出现在字符串开头或结尾处的单词边界上。“\b”定位符经常用于确保子表达式与整个单词（而不仅与单词的开头或结尾）匹配。例如，字符串“area bare arena mare”使用正则表达式“\bare\w*\b”去匹配，“area”“arena”是满足此正则表达式的。

“\B”定位符指定匹配不得出现在单词边界上，它与“\b”定位符截然相反。例如，字符串“equity queen equip acquaint quiet”使用正则表达式“\Bqu\w+”去匹配，“quity”“quip”和“quaint”是满足此正则表达式的。

16.2.6 分组构造

分组构造描述了正则表达式的子表达式，用于捕获输入字符串的子字符串。以下是分组构造捕获匹配的子表达式：

(子表达式)

其中“子表达式”为任何有效正则表达式模式。使用括号的捕获按正则表达式中左括号的顺序从一开始就由左到右自动编号。

例如，字符串“He said that that was the correct answer.”我们使用“(\w+)\s(\w+)\W”来匹配，结果则是：

- “He said”一组，其中“He”和“said”各为一个子组；
- “that that”一组，其中“that”和“that”各为一个子组；
- “was the”一组，其中“was”和“the”各为一个子组；
- “correct answer.”一组，其中“correct”和“answer.”各为一个子组。

16.2.7 匹配模式

匹配模式指的是匹配的时候使用的规则。使用不同的匹配模式可能会改变正则表达式的识别，也可能会改变正则表达式中字符的匹配规定。本小节将会介绍几种常见的匹配规则。

不区分大小写模式指的就是在匹配单词时，正则表达式将不会区分字符串中的大小写。例如期望用户输入“yes”，但是用户也有可能输入“Yes”或者“yES”等，如果区分大小写，那么正

则表达式就要写成 “[yY][eE][sS]” , 这样做确实可以匹配到想要的结果，但是写起来就很麻烦。如果启用了不区分大小写模式匹配字符串，我们只需使用正则 “yes” 就可以匹配用户输入的各种大小写混杂的 “Yes” “yes” “yEs” 了。

单行模式（也叫点号通配）会改变元字符 “.” 的匹配方式。元字符 “.” 几乎可以匹配任何字符，但是默认情况下 “.” 不会匹配 “\n” 换行符。然而，有时候确实想要匹配 “任何字符” , 此时单行模式就可以让 “.” 匹配任何字符（当然也可以使用如 “[s\S]” “[w\W]” 等技巧来匹配所有字符）。

多行模式改变的是 “^” 和 “\$” 的匹配方式。默认模式下，“^” 和 “\$” 匹配的是整个字符串的起始位置和结束位置；但是在多行模式下，它们将会匹配字符串内部某一行文本的起始位置和结束位置。

16.3 regexp包

Go在处理正则表达式时主要使用regexp包，包中实现了正则表达式的查找、替换和模式匹配功能。

16.3.1 MatchString函数

MatchString函数接收一个要查找的正则表达式和目标字符串，并根据匹配结果返回true或false，函数定义如下：

```
func MatchString(pattern string, s string) (matched bool, err error)
```

对于目标字符串 “hello world” , 我们通过MatchString函数匹配其中的 “hello” 字符串，并返回匹配结果：

动手写16.3.1

```
01 package main
02
03 import (
04     "regexp"
05     "fmt"
06 )
07
08 func main() {
```

```
09     targetString := "hello world"
10    matchString := "hello"
11    match, err := regexp.MatchString(matchString, targetString)
12    if err != nil{
13        fmt.Println(err)
14    }
15    fmt.Println(match)
16 }
```

执行结果如下：

```
true
```

程序执行结果表明匹配成功。那么，如果我们把目标字符串改为“Hello world”，匹配是否会成功呢？

动手写 16.3.2

```
01 package main
02
03 import (
04     "regexp"
05     "fmt"
06 )
07
08 func main() {
09     targetString := "Hello world"
10     matchString := "hello"
11     match, err := regexp.MatchString(matchString, targetString)
12     if err != nil{
13         fmt.Println(err)
14     }
15     fmt.Println(match)
16 }
```

执行结果如下：

```
false
```

匹配结果显示失败了，由于正则表达式区分大小写，如果要以不区分大小写的方式查找，必须修改正则表达式，使用一种特殊的语法：

```
matchString := "(?i)hello"
```

字符串开头的语法让正则表达式匹配时不区分大小写，运行修改后的示例，结果为true。

动手写16.3.3

```
01 package main
02
03 import (
04     "regexp"
05     "fmt"
06 )
07
08 func main() {
09     targetString := "Hello world"
10     matchString := "(?i)hello"
11     match, err := regexp.MatchString(matchString, targetString)
12     if err != nil{
13         fmt.Println(err)
14     }
15     fmt.Println(match)
16 }
```

执行结果如下：

```
true
```

16.3.2 FindStringIndex函数

FindStringIndex函数接收一个目标字符串，并返回第一个匹配的起始位置和结束位置，函数定义如下：

```
func (re *Regexp) FindStringIndex(s string) (loc []int)
```

由于FindStringIndex函数是Regexp结构体的成员函数，需要对正则表达式进行编译，编译成功后方能使用。通常使用Compile或MustCompile函数进行编译。

1. Compile函数：若正则表达式未通过编译，则返回错误。

2. MustCompile函数：若正则表达式未通过编译，则引发panic。

对于目标字符串“hello world”，我们通过FindStringIndex函数匹配其中的“hello”字符串对应的起始和结束位置：

动手写16.3.4

```
01 package main
02
03 import (
04     "regexp"
05     "fmt"
06 )
07
08 func main() {
09     targetString := "hello world"
10     re := regexp.MustCompile(`(\w)+`)
11     res := re.FindStringIndex(targetString)
12     fmt.Println(res)
13 }
```

执行结果如下：

```
[0 5]
```

16.3.3 ReplaceAllString函数

ReplaceAllString函数返回第一个参数的拷贝，将第一个参数中所有re的匹配结果都替换为repl，函数定义如下：

```
func (re *Regexp) ReplaceAllString(src, repl string) string
```

同样对于目标字符串“hello world”，我们通过ReplaceAllString函数匹配其中的“o”字符并将替换为“O”：

动手写16.3.5

```
01 package main
02
03 import (
04     "regexp"
```

```
05      "fmt"
06  )
07
08 func main() {
09     targetString := "hello world"
10     re := regexp.MustCompile(`o`)
11     res := re.ReplaceAllString(targetString, "O")
12     fmt.Println(res)
13 }
```

执行结果如下：

```
hello wOrld
```

16.4 小结

- ◇ 正则表达式是符合一定规则的表达式，用于匹配字符串中字符组合的模式。
- ◇ 正则表达式的设计思想就是使用一些描述性的符号和文字为字符串定义一个规则。凡是符合这个规则的，程序就认为文本是“匹配”的，否则就认为文本是“不匹配”的。
- ◇ Go在处理正则表达式时主要使用regexp包，包中实现了正则表达式的查找、替换和模式匹配功能。

16.5 知识拓展

常用正则表达式参考

1. E-mail地址

```
^\w+([-.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$
```

2. URL地址

```
^(https?:\/\/)?([\da-z.-]+)\.([a-z.]{2,6})([\/\w\.-])\/?$/
```

3. 匹配首尾空白字符的正则表达式

```
^\s|\s$
```

4. 手机号码

```
^(13[0-9]|14[0-9]|15[0-9]|166|17[0-9]|18[0-9]|19[8|9])\d{8}$
```

5. 电话号码

```
^(\\d{3,4}-)?\\d{7,8}$
```

电话号码类型较多，包括XXX-XXXXXXX、XXXX-XXXXXXXX、XXXX-XXXXXXX、XXX-XXXXXXX、XXXXXXX和XXXXXXX等。例如0511-1234567、021-12345678。

6. 18位身份证号码（数字、字母x结尾）

```
^((\\d{18})|([0-9x]{18})|([0-9X]{18}))$
```

7. 账号是否合法（5~16字节，允许字母、数字、下划线，以字母开头）

```
^[a-zA-Z][a-zA-Z0-9_]{4,15}$
```

8. 一年的12个月（01~09和1~12）

```
^(0?[1-9]|1[0-2])$
```

9. 日期格式（2018-01-01只做粗略匹配，格式不限制，二月有30天等）

```
^\\d{4}-\\d{1,2}-\\d{1,2}$
```

10. 一个月的31天（01~09和1~31）

```
^((0?[1-9])|((1|2)[0-9])|30|31)$
```

11. IP地址

```
^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\\.){3}(([0-9]|[1-9][0-9]{2}|2[0-4][0-9]|25[0-5]))$
```

在第15章中，我们介绍了Go语言中的常用命令行工具，它可以帮助我们创建、使用和测试工程代码。从本章开始，我们将会学习如何使用Go语言进行网络编程。Go语言的最大优势在于并发和性能，其性能可以媲美C和C++，并在网络编程中更是至关重要。

本章将会学习如何使用Go语言创建HTTP服务器和客户端，使用Go语言开发Web服务，让开发者不需要进行各种繁杂的性能优化就可以很轻松地开发出一个高性能的Web服务。

17.1 HTTP简介

17.1.1 HTTP协议

HTTP，即超文本传输协议（HyperText Transfer Protocol），是互联网上应用最为广泛的一种网络协议。它详细规定了浏览器和万维网服务器之间的相互通信规则，所有的WWW文件都必须遵守这个标准。设计HTTP最初的目的就是为了提供一种发布和接收HTML页面的方法。

提示

HTML叫超文本标记语言（HyperText Markup Language），是标准通用标记语言下的一个应用。HTML不是一种编程语言，而是一种标记语言（markup language），是网页制作所必备的。“超文本”就是指页面内可以包含图片、链接，甚至音乐、程序等非文字元素。超文本标记语言（或超文本标签语言）的结构包括“头”部分和“主体”部分，其中“头”部提供关于网页的信息，“主体”部分提供网页的具体内容。

HTTP是一个客户端和服务器端请求和应答的标准。客户端是终端用户，服务器端是网站。通过使用Web浏览器、网络爬虫或者其他工具，客户端发起一个到服务器上指定端口（默认端口为80，HTTPS网站默认端口为443）的HTTP请求。

提示

HTTP协议是基于TCP协议之上建立的，有时也会承载于TLS或SSL协议之上，这时候HTTP协议就变成了加密的HTTPS协议。

通常，由HTTP客户端发起一个请求，建立一个到服务器指定端口的TCP连接，HTTP服务器则在那个端口监听客户端发送过来的请求。一旦收到请求，服务器（向客户端）发回一个状态行，比如“HTTP/1.1 200 OK”，以及响应的消息，消息的消息体可能是请求的文件、错误消息或者其他一些信息。

HTTP协议包含请求体和响应体，我们来看最简单的HTTP协议的请求体：

```
GET / HTTP/1.1
Host: www.baidu.com
User-Agent: curl/7.55.1
Accept: */*
```

这段请求的目标地址是www.baidu.com，User-Agent为curl，表示这个请求是由curl这个工具发出的（关于curl工具可参考“17.5 知识拓展”部分），请求的地址为目标地址的根目录“/”。服务端接收到后，会做出响应的回应，响应体主要包含响应头和响应内容，如下所示，“<!DOCTYPE html>……”为响应的内容，之上的内容为响应头。

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform
Connection: Keep-Alive
Content-Length: 2381
Content-Type: text/html
Date: Sun, 25 Aug 2019 12:13:07 GMT
Etag: "588604dc-94d"
Last-Modified: Mon, 23 Jan 2017 13:27:56 GMT
Pragma: no-cache
```

```
Server: bfe/1.0.8.18
Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/
<!DOCTYPE html>
<!--STATUS OK--><html>
....
```

17.1.2 URL地址

URL (Uniform Resource Locator) 是统一资源定位符，是对可以从互联网上得到的资源的位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

基本URL包含模式（或称协议）、服务器名称（或IP地址）、路径和文件名。

协议://主机地址/路径?查询

<https://www.baidu.com/s?wd=Go语言>

完整的、带有授权部分的普通统一资源标志符语法如下：

协议://用户名:密码@子域名.域名.顶级域名:端口号/目录/文件名.文件后缀?参数=值#标志

<http://admin:password@ceshi.baidu.com:8080/dir/test.php?id=1#flag=1>

URL的长度有限制，不同的服务器限制值不同：

- ◇ IE (Browser)：对URL的最大限制为2083个字符。
- ◇ Firefox (Browser)：URL的长度限制为65536个字符。
- ◇ Safari (Browser)：URL最大长度限制为80000个字符。
- ◇ Opera (Browser)：URL最大长度限制为190000个字符。
- ◇ Chrome (Browser)：URL最大长度限制为8182个字符。
- ◇ Apache (Server)：能接受最大URL长度为8192个字符。
- ◇ Microsoft Internet Information Server (IIS)：能接受最大URL长度为16384个字符。

17.1.3 Web工作方式

我们在浏览网页时，从打开浏览器、输入网址到按下回车键后网页上出现内容，这过程中浏览器到底做了哪些操作呢？

浏览器本身是一个客户端（手机端客户端内部也是浏览器实现的），当你输入URL的时候，首先浏览器会去请求DNS服务器，通过DNS获取相应的域名对应的IP，如：

```
www.example.com => DNS服务器 => 获得域名地址为1.1.1.1
```

然后，浏览器通过IP地址找到对应的服务器，要求建立TCP连接，等浏览器发送完HTTP请求包，服务器接收到请求后开始处理请求包，服务器调用自身服务，返回HTTP响应包，客户端收到来自服务器的响应后开始渲染这个响应包里的主体body，等收到全部的内容后断开与该服务器之间的TCP连接。

17.2 HTTP客户端

在学习如何使用Go语言模拟客户端前，我们需要明白HTTP请求的整个过程：用户通过浏览器或其他客户端向服务器发起请求传递request对象，服务器接收request对象，并给出响应，回复response对象。在HTTP客户端发起请求时，经常需要请求不同的资源，这时就需要使用服务端的路由机制，通过不同的请求地址进行相应的路由，如通过GET /user/{id}就能定位到该使用哪个接口来处理这次的请求，user就是路由，而id为路由使用到的方法所需要的参数。

```
GET /user/1541
PUT /user/1212
```

在Web开发过程中，我们最常见到的路由风格为RESTful，接下来我们将学习什么是RESTful。

提示

HTTP路由也称路径映射机制，负责将HTTP请求交给对应的方法函数进行处理，可以理解为通过使用不同的请求地址来获取不同的资源。

17.2.1 REST

REST (Representational State Transfer) 是一个标准、一种规范，遵循REST风格可以使开发的接口通用，便于调用者理解接口的作用，使用这种风格可以清晰地了解一个请求的目的是什么。

RESTful架构风格规定，数据的元操作，即CRUD (create, read, update和delete，即数据的增查改删) 操作，分别对应四种HTTP操作：POST用来新建资源（也可以用于更新资源），GET用来获取资源，PUT用来更新资源，DELETE用来删除资源。这样就统一了数据操作的接口，仅通过HTTP方法，就可以完成对数据的所有增查改删工作。即：

- ◇ GET (READ) : 从服务器取出资源（一项或多项）。
- ◇ POST (CREATE) : 在服务器新建一个资源。
- ◇ PUT (UPDATE) : 在服务器更新资源（客户端提供完整资源数据）。
- ◇ PATCH (UPDATE) : 在服务器更新资源（客户端提供需要修改的资源数据）。
- ◇ DELETE (DELETE) : 从服务器删除资源。

REST风格架构有如下约束：

1. 使用客户/服务器模型。客户和服务器之间通过一个统一的接口来互相通信。
2. 层次化的系统。在一个REST系统中，客户端并不会固定地与一个服务器打交道。
3. 无状态。在一个REST系统中，服务端并不会保存有关客户的任何状态。也就是说，客户端自身负责用户状态的维持，并在每次发送请求时都需要提供足够的信息。
4. 可缓存。REST系统需要能够恰当地缓存请求，以尽量减少服务端和客户端之间的信息传输，提高性能。
5. 统一的接口。一个REST系统需要使用一个统一的接口来完成子系统之间以及服务与用户之间的交互，这使得REST系统中的各个子系统可以独自完成演化。

REST是面向资源的，这个概念非常重要，而资源是通过URI进行暴露的。URI的设计只要负责把资源通过合理方式暴露出来就可以了，对资源的操作与它无关，操作是通过HTTP动词来体现的，所以REST通过URI暴露资源时，会强调不要在URI中出现动词。举个例子：

GET /rest/api/user	获取一个用户信息
POST /rest/api/user	添加一个用户信息
PUT /rest/api/user/: user_id	更新一个用户信息
DELETE /rest/api/user/:user_id	删除一个用户信息

如上述例子中，所有资源都是名词，所有的动作都由HTTP的方法决定（想要获取资源就使用GET方法，删除一个资源就使用DELETE方法），就能非常直观地了解一个请求的目的是什么，请求后能够获取什么。

路由不是固定的，而是多变的，所以我们在服务器端做路由适配，通过不同的路由去匹配不同的处理器，处理不同的请求，这就又引出了两个新的对象，即路由转接器ServeMux和处理器Handler。http包提供了HTTP客户端和服务端的实现，关于http包的内容我们将在后面的小节中进行学习。

17.2.2 Client和Request

学习HTTP客户端，我们需要理解和掌握两个非常重要的类型，一个是Client，另一个是Request。首先，我们来看一下Client的结构体类型。

```
type Client struct {
    Transport RoundTripper
    CheckRedirect func(req *Request, via []*Request) error
    Jar CookieJar
    Timeout time.Duration
}
```

Client 的结构相对较为简单，只有四个成员，分别是 Transport、CheckRedirect、Jar 和 Timeout。Transport 指定执行独立、单次 HTTP 请求的机制。如果 Transport 为 nil，则使用 DefaultTransport。CheckRedirect 指定处理重定向的策略。如果 CheckRedirect 不为 nil，客户端会在执行重定向之前调用本函数字段。参数 req 和 via 是将要执行的请求和已经执行的请求（切片，越新的请求越靠后）。如果 CheckRedirect 返回一个错误，本类型的 GET 方法不会发送请求 req，而是返回之前得到的最后一个回复和该错误（包装进 url.Error 类型里）。如果 CheckRedirect 为 nil，会采用默认策略：连续十次请求后停止。

Jar 指定 Cookie 管理器。如果 Jar 为 nil，请求中不会发送 Cookie，回复中的 Cookie 会被忽略。

Timeout 指定本类型的值执行请求的时间限制。该超时限制包括连接时间、重定向和读取回复主体的时间。计时器会在 Head、Get、Post 或 Do 方法返回后继续运作，并在超时后中断回复主体的读取。Timeout 为零值表示不设置超时。

Client 类型主要充当浏览器的角色，它拥有如下方法：

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

上述方法中最重要的是 Do 方法，其他的方法 GET、POST 等都是基于 Do 方法的封装，这些方法的具体使用我们将会在后面的章节中进行介绍。

除了 Client 类型，我们还需要了解 Request 类型。Request 主要是对请求体的封装，任何形式的 HTTP 请求都可以使用 Request 来构造，构造完后使用 Client 端发送请求。Request 的结构体如下：

```
type Request struct {
    Method string           // 请求方法
    URL    *url.URL         // 请求地址
```

```
Proto      string          // 协议版本, "HTTP/1.0"
ProtoMajor int             // 协议主版本号, "1"
ProtoMinor int             // 协议主副版本号, "0"
Header     Header          // 请求头
Body       io.ReadCloser   // 请求的Body
ContentLength int64        // ContentLength记录相关内容的长度
TransferEncoding []string  // TransferEncoding按从最外到最里的顺序列出传输编码
Close      bool             // Close在服务端指定是否在回复请求后关闭连接，在客户端指定是否在发送请求后关闭连接
Host      string           // Host指定URL会在其上寻找资源的主机
Form      url.Values       // Form是解析好的表单数据，包括URL字段的query参数和POST或PUT的表单数据
PostForm  url.Values       // PostForm是解析好的POST或PUT的表单数据
MultipartForm *multipart.Form // MultipartForm是解析好的多部件表单，包括上传的文件
Trailer    Header          // Trailer指定了会在请求主体之后发送的额外的头域
RemoteAddr string          // RemoteAddr允许HTTP服务器和其他软件记录该请求的来源地址，一般用于日志
RequestURI string          // RequestURI是被客户端发送到服务端的请求中未修改的URI
TLS       *tls.ConnectionState // TLS字段允许HTTP服务器和其他软件记录接收到该请求的TLS连接的信息
}
```

对于Request，我们需要掌握的成员字段有Method设置请求方法，如果是需要使用GET方法就设置为“GET”。URL字段为请求的地址，如：

```
http://www.baidu.com
```

对于模拟客户端发起请求，我们经常需要设置请求头Header，HTTP规定头的键名（头名）是大小写敏感的，请求的解析器通过规范化请求头的键名来实现这一点。如：

```
Accept-Encoding: gzip, deflate
Accept-Language: en-us
Connection: keep-alive
```

还有一个重要的成员为Form，Form是解析好的表单数据，包括URL字段的query参数和POST或PUT的表单数据，该字段只有在调用ParseForm后才有效。如果需要模拟上传文件，则需要使用MultipartFile成员。

在客户端进行请求数据时，使用最多的请求方法为GET和POST，我们会针对这两个方法进行深入的学习。

- ◇ GET——从指定的资源请求数据。
- ◇ POST——向指定的资源提交要被处理的数据。

GET和POST是表单提交数据的两种基本方式。GET请求数据通过域名后缀url传送，用户可见，不安全；POST请求数据在请求报文正文里传输，相对比较安全。

GET用于请求指定的页面信息，并返回实体主体。它有如下特点：

- ◇ GET请求可被缓存。
- ◇ GET请求保留在浏览器历史记录中。
- ◇ GET请求可被收藏为书签。
- ◇ GET请求不应在处理敏感数据时使用。
- ◇ GET请求有长度限制。
- ◇ GET请求只应当用于取回数据。

一个最简单的GET请求示例如下，通过GET请求数据，并提交了两个参数key1和key2，多个参数之间使用符号“&”连接。需要注意的是，在请求头结束后，也就是(Accept: */*)下面，有一空行，这才是一个完整的HTTP请求头。

```
GET /index?key1=value1&key2=value2 HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.55.1
Accept: */*
```

POST用于向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和（或）已有资源的修改。POST提交数据有如下特点：

- ◇ POST请求不会被缓存。

- ◇ POST请求不会保留在浏览器历史记录中。
- ◇ POST不能被收藏为书签。
- ◇ POST请求对数据长度没有要求。

一个最简单的POST请求示例如下，与GET最大的不同之处在于，POST方法将请求的参数放到请求的body中，而不是URL中。

```
POST /index HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.55.1
Accept: */*

key1=value1&key2=value2
```

提示

对于GET方式的请求，浏览器会把HTTP header和data一并发送出去，服务器响应200并返回数据。而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器再响应200并返回数据。并不是所有浏览器都会在POST中发送两次包，如Firefox就只发送一次。

Go语言中Client类型的GET和POST方法，在数据传输过程中分别对应了HTTP协议中的GET和POST方法。二者主要区别如下：

1. GET是用来从服务器上获得数据，而POST是用来向服务器传递数据。
2. GET将表单中的数据按照variable=value的形式，添加到action所指向的URL后面，并且两者使用“?”连接，而各个变量之间使用“&”连接；POST是将表单中的数据放在form的数据体中，按照变量和值相对应的方式，传递到action所指向的URL。
3. GET是不安全的，因为在传输过程中，数据被放在请求的URL中，而如今现有的很多服务器、代理服务器或者用户代理都会将请求URL记录到日志文件中，然后放在某个地方，这样就可能会有一些隐私的信息被第三方看到。另外，用户也可以在浏览器上直接看到提交的数据，一些系统内部消息将会一同显示在用户面前。而POST的所有操作对用户来说都是不可见的。
4. GET传输的数据量小，这主要是因为受URL长度限制，而POST可以传输大量的数据，所以在上传文件时只能使用POST（当然还有一个原因，将在后面提到）。
5. GET限制Form表单的数据集的值必须为ASCII字符，而POST支持整个ISO 10646字符集。
6. 使用POST传输的数据，可以通过设置编码的方式正确转化中文，而GET传输的数据却没有

变化。在以后的程序中，我们一定要注意这一点。

7. POST会有浏览器提示重新提交表单的问题，GET则没有。

17.2.3 发起GET请求

从本小节开始我们将会学习如何使用Go语言模拟浏览器发起HTTP请求。发起请求前需要创建一个请求对象，使用NewRequest创建。

```
func NewRequest(method, urlStr string, body io.Reader) (*Request, error)
```

NewRequest使用指定的方法、网址和可选的body创建并返回一个新的*Request。我们先来看如何发起一个GET请求：

动手写17.2.1

```
01 package main
02
03 import (
04     "net/http"
05     "fmt"
06 )
07
08 func main() {
09     client := &http.Client{}
10     request, err := http.NewRequest("GET", "http://www.baidu.
com", nil)
11     if err != nil {
12         fmt.Println(err)
13     }
14     response, err := client.Do(request)
15     fmt.Println(response.StatusCode)
16 }
```

动手写17.2.1对百度网站发起了一次GET请求，程序的主要流程为创建了一个客户端对象client和一个请求对象request，分别对其进行初始化，request初始化需要提供请求方法和请求地址。这两个对象初始化后，client客户端调用Do方法发起一次请求并获得服务端的响应，最后程序打印出响应的状态码，执行结果如下：

200

如果想要打印响应的主要内容body，可以使用标准包ioutil中的ReadAll方法来读取响应的流数据。

动手写17.2.2

```
01 package main
02
03 import (
04     "net/http"
05     "fmt"
06     "io/ioutil"
07 )
08
09 func main() {
10     client := &http.Client{}
11     request, err := http.NewRequest("GET", "http://www.baidu.
com", nil)
12     if err != nil {
13         fmt.Println(err)
14     }
15     response, err := client.Do(request)
16     res, err := ioutil.ReadAll(response.Body)
17     if err != nil {
18         fmt.Println(err)
19     }
20     // 打印body
21     fmt.Println(string(res))
22 }
```

动手写17.2.2发起请求并打印输出了响应内容body，由于Body的类型为io.ReadCloser，不能直接读取其中的内容，需要通过ioutil.ReadAll读取，返回字节数组，最后转化成字符串进行输出。输出的内容如下：

```
<!DOCTYPE html><html lang="zh-cmn-Hans"><head><meta charset="UTF-
8"><title>百度一下，你就知道</title>.....
```

当一个网站有一些资源需要登录后才能访问时，我们希望能够通过Go语言实现批量自动化爬取这些内容，那如何让服务器知道我们用Go编写的爬虫客户端是已登录状态？这就需要设置使用Cookie，Cookie通常用来标识客户端的登录状态。Request实例可以使用AddCookie方法给请求添加Cookie。

动手写17.2.3

```

01 package main
02
03 import (
04     "net/http"
05     "fmt"
06     "strconv"
07 )
08
09 func main() {
10     client := &http.Client{}
11     request, err := http.NewRequest("GET", "http://www.baidu.
com", nil)
12     if err != nil {
13         fmt.Println(err)
14     }
15     // 使用http.Cookie结构体初始化一个cookie键值对
16     cookie := &http.Cookie{Name: "userId", Value: strconv.
Itoa(12345)}
17     // 使用前面构建的request方法AddCookie往请求中添加Cookie
18     request.AddCookie(cookie)
19     request.AddCookie(&http.Cookie{Name: "session", Value:
"YWRtaW4="})
20     response, err := client.Do(request)
21     fmt.Println(response.Request.Cookies())
22 }
```

动手写17.2.3使用AddCookie添加了userId和session两个Cookie，通常服务端使用session来判断客户端是否为登录用户，运行结果如下：

```
[userId=12345 session=YWRtaW4=]
```

做了反爬策略的网站，一般都会根据Header头中的User-Agent的值解析判断是浏览器还是爬虫，这时我们需要设置成浏览器的UA来绕过这类反爬策略。Request可以直接使用request.Header.Set(key,value)来设置Header。

动手写17.2.4

```
01 package main
02
03 import (
04     "net/http"
05     "fmt"
06 )
07
08 func main() {
09     client := &http.Client{}
10     request, err := http.NewRequest("GET", "http://www.baidu.
com", nil)
11     if err != nil {
12         fmt.Println(err)
13     }
14     // 设置request的Header, 具体可参考HTTP协议
15     request.Header.Set("Accept", "text/html, application/xhtml+
xml, application/xml;q=0.9, */*;q=0.8")
16     request.Header.Set("Accept-Charset", "GBK, utf-8;q=0.7, *;
q=0.3")
17     request.Header.Set("Accept-Encoding", "gzip, deflate, sdch")
18     request.Header.Set("Accept-Language", "zh-CN, zh;q=0.8")
19     request.Header.Set("Cache-Control", "max-age=0")
20     request.Header.Set("Connection", "keep-alive")
21     request.Header.Set("User-Agent", "Mozilla/5.0 (Windows NT
10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
72.0.3626.121 Safari/537.36")
22     response, err := client.Do(request)
23     fmt.Printf("%#v", response.Request.Header)
24 }
```

动手写17.2.4使用了自定义的Header来发起请求，运行结果如下：

```
http.Header{"Accept-Language":[]string{"zh-CN, zh;q=0.8"}, "Referer":[]string{"http://www.baidu.com"}, "Cache-Control":[]string{"max-age=0"}, "Connection":[]string{"keep-alive"}, "User-Agent":[]string{"Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"}, "Accept":[]string{"text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8"}, "Accept-Charset":[]string{"GBK, utf-8;q=0.7, *;q=0.3"}, "Accept-Encoding":[]string{"gzip, deflate, sdch"}}
```

在上一小节中，我们提到了Client类型含有GET方法，而GET方法是对Do方法的封装，使用GET方法可以快速发起一个GET请求。http包中也有一个叫GET的方法，底层就是调用Client的GET方法。

动手写17.2.5

```
01 package main
02
03 import (
04     "fmt"
05     "io/ioutil"
06     "net/http"
07 )
08
09 func main() {
10     // http.Get实际上是DefaultClient.Get(url)，Get函数是高度封装的，只
11     // 有一个参数url
12     // 对于一般的http Request是可以使用，但是不能定制Request
13     response, err := http.Get("http://www.baidu.com")
14     if err != nil {
15         fmt.Println(err)
16     }
17     defer response.Body.Close()
18     body, _ := ioutil.ReadAll(response.Body)
19     fmt.Println(string(body))
20 }
```

运行结果如下：

```
<!DOCTYPE html>
<!--STATUS OK-->
.....
```

17.2.4 发起POST请求

在上一小节我们使用了http.NewRequest来创建一个请求，http.NewRequest有三个参数，第一个是请求方法Method，第二个是请求地址URL，最后一个参数是body，其类型为io.Reader。GET方法不需要使用body参数，只需传入nil即可，如果是POST请求，则需要使用该参数。

如果body参数实现了io.Closer接口，Request返回值的Body字段会被设置为body，并会被Client类型的Do、Post和PostForm方法以及Transport.RoundTrip方法关闭。举个例子：

动手写17.2.6

```
01 package main
02
03 import (
04     "net/http"
05     "fmt"
06     "strings"
07     "io/ioutil"
08 )
09
10 func main() {
11     client := &http.Client{}
12     // 使用NopCloser创建一个实现io.Closer接口的body
13     body := ioutil.NopCloser(strings.NewReader("user=admin&pass=
admin"))
14     req, err := http.NewRequest("POST", "http://www.baidu.com",
body)
15     if err != nil {
16         fmt.Println(err)
17     }
18     // 打印输出body
19     req_body, err := ioutil.ReadAll(req.Body)
```

```

20     fmt.Println(string(req_body))
21
22     // POST提交表单需要添加此Header头
23     req.Header.Set("Content-Type", "application/x-www-form-
24         urlencoded")
25     resp, err := client.Do(req)
26     if err != nil {
27         fmt.Println(err)
28     }
29
30     // 在client使用Do方法后，再次打印Body发现Body已经被关闭，使用ReadAll
31     // 读取会导致错误
32     req_body, err = ioutil.ReadAll(resp.Request.Body)
33     if err != nil {
34         fmt.Println(err)
35     }
36     fmt.Println(string(req_body))
37 }
```

动手写17.2.6使用了ioutil.NopCloser创建了一个实现io.Closer接口的body对象，将其传入http.NewRequest的第三个参数，在发起请求前（client.Do()前），打印输出了req.Body。在请求后再次以同样的方式打印输出请求Body就会发现程序报错，原因是请求的Body在Client类型的Do方法运行后就会被关闭，程序输出空指针异常。运行结果如下：

```

user=admin&pass=admin
panic: runtime error: invalid memory address or nil pointer
dereference [recovered]
    panic: runtime error: invalid memory address or nil pointer
    dereference
[signal 0xc0000005 code=0x0 addr=0x18 pc=0x496993]
.....
```

我们同样可以使用http包封装后的POST方法来实现POST请求。

```

func Post(url string, bodyType string, body io.Reader) (*resp
*Response, err error)
```

Post向指定的URL发出一个POST请求，bodyType为POST数据的类型，body为POST数据，作为请求的主体。如果参数body实现了io.Closer接口，它会在发送请求后被关闭，调用者有责任在读取完返回值resp的主体后关闭它。该方法是对Client的POST方法的包装。

动手写17.2.7

```
01 package main
02
03 import (
04     "net/http"
05     "strings"
06     "fmt"
07     "io/ioutil"
08 )
09
10 func main() {
11     resp, err := http.Post("http://www.baidu.com",
12                         "application/x-www-form-urlencoded",
13                         strings.NewReader("user=admin&pass=admin"))
14     if err != nil {
15         fmt.Println(err)
16     }
17
18     defer resp.Body.Close()
19     body, err := ioutil.ReadAll(resp.Body)
20     if err != nil {
21         fmt.Println(err)
22     }
23     fmt.Println(string(body))
24 }
```

除了POST方法，http包还封装了一个叫PostForm的方法。

```
func PostForm(url string, data url.Values) (resp *Response, err error)
```

PostForm向指定的URL发出一个POST请求，url.Values类型的data会被编码为请求的主体。如果返回值err为nil，resp.Body总是非nil的，调用者应该在读取完resp.Body后关闭它。该方法是对Client

的 PostForm 方法的包装。

我们使用 PostForm 来改写动手写 17.2.7：

动手写 17.2.8

```

01 package main
02
03 import (
04     "net/http"
05     "net/url"
06     "io/ioutil"
07     "fmt"
08 )
09
10 func main() {
11     resp, err := http.PostForm("http://www.baidu.com",
12         url.Values{"user": {"admin"}, "pass": {"admin"}})
13     if err != nil {
14         fmt.Println(err)
15     }
16     defer resp.Body.Close()
17     body, err := ioutil.ReadAll(resp.Body)
18     if err != nil {
19         fmt.Println(err)
20     }
21     fmt.Println(string(body))
22 }
```

17.3 爬虫框架 gocolly

我们在之前的章节中学习了如何使用标准库实现 HTTP 爬虫，在实际项目中，我们往往会展开一些第三方的爬虫框架来编写爬虫程序，这是因为框架是基于标准库的封装，本身会提供非常丰富且功能强大的 API 调用，我们只需编写简短的代码就能实现一个高性能爬虫，这里推荐使用 colly 爬虫框架。

colly 是用 Go 语言实现的网络爬虫框架。colly 快速优雅，在单核上每秒可以发起 1K 以上请求；以回调函数的形式提供了一组接口，可以实现任意类型的爬虫。

17.3.1 gocolly简介

colly的代码托管在GitHub上，官方网站为<http://go-colly.org/>。colly使用起来简洁高效，有着如下特性：

- ◇ 非常清晰的API。
- ◇ 快速（单核上>1K请求/秒）。
- ◇ 管理请求延迟和每个域的最大并发数。
- ◇ 自动Cookie和会话处理。
- ◇ 同步/异步/并行抓取。
- ◇ 高速缓存。
- ◇ 自动处理非Unicode的编码。
- ◇ Robots.txt支持。
- ◇ 分布式抓取。
- ◇ 通过环境变量配置。
- ◇ 可扩展。

下载安装colly框架只需使用如下命令：

```
go get -u github.com/gocolly/colly
```

colly的主要实体是一个Collector对象。Collector在收集器作业运行时管理网络通信并负责执行附加的回调。要使用colly，必须先初始化Collector。

```
c := colly.NewCollector()
```

一旦得到一个colly对象，就可以向colly附加各种不同类型的回调函数（回调函数在colly中广泛使用）来控制收集作业或获取信息，回调函数如下：

```
// 请求发送前所运行的函数
c.OnRequest(func(r *colly.Request) {
    fmt.Println("Visiting", r.URL)
})

// 遇到错误时所运行的函数
c.OnError(func(_ *colly.Response, err error) {
    log.Println("Something went wrong:", err)
})
```

```
// 接收到响应后所运行的函数
c.OnResponse(func(r *colly.Response) {
    fmt.Println("Visited", r.Request.URL)
})

// 接收到HTML网页后所运行的函数
c.OnHTML("a[href]", func(e *colly.HTMLElement) {
    e.Request.Visit(e.Attr("href"))
})

c.OnHTML("tr td:nth-of-type(1)", func(e *colly.HTMLElement) {
    fmt.Println("First column of a table row:", e.Text)
})

// 接收到XML格式的响应内容后所运行的函数
c.OnXML("//h1", func(e *colly.XMLElement) {
    fmt.Println(e.Text)
})

// 请求完成后所运行的函数
c.OnScraped(func(r *colly.Response) {
    fmt.Println("Finished", r.Request.URL)
})
```

colly 的回调函数非常清晰，比如 OnRequest 表示在请求之前所需要做的事情，OnResponse 表示在获取到响应后所做的事情。我们来看一个最简单的例子：

动手写 17.3.1

```
01 package main
02
03 import (
04     "fmt"
05     "github.com/gocolly/colly"
06 )
07
08 func main() {
```

```
09     c := colly.NewCollector()
10
11     // 查找和访问所有的链接
12     c.OnHTML("a[href]", func(e *colly.HTMLElement) {
13         e.Request.Visit(e.Attr("href"))
14     })
15
16     c.OnRequest(func(r *colly.Request) {
17         fmt.Println("Visiting", r.URL)
18     })
19
20     c.Visit("http://go-colly.org/")
21 }
```

动手写17.3.1使用NewCollector创建了一个Collector对象，每当colly获取到HTML网页时，就会查找网页中的所有链接，并添加到访问队列中。OnRequest在发起请求前就会打印输出访问的URL。c.Visit表示初始访问链接为http://go-colly.org/，这段程序的作用是访问全站链接。程序运行的结果如下：

```
Visiting http://go-colly.org/
```

NewCollector在初始化时，可以传入colly.UserAgent来设置请求的UA，绕过网站的反爬策略。OnRequest一般会在请求前设置一些模拟真实浏览器的Header头，举个例子：

动手写17.3.2

```
01 package main
02
03 import (
04     "fmt"
05     "github.com/gocolly/colly"
06 )
07
08 func main() {
09     // NewCollector(options ...func(*Collector)) *Collector
10     // 声明初始化NewCollector对象时可以指定Agent，连接递归深度，URL过滤以及domain限制等
```

```
11     c := colly.NewCollector(  
12         colly.UserAgent("Mozilla/5.0 (Windows NT 10.0; WOW64)  
13             AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121  
14             Safari/537.36"))  
15  
16     // 发出请求时附的回调  
17     c.OnRequest(func(r *colly.Request) {  
18         // Request头部设定  
19         r.Headers.Set("Connection", "keep-alive")  
20         r.Headers.Set("Accept", "*/*")  
21         r.Headers.Set("Origin", "")  
22         r.Headers.Set("Accept-Encoding", "gzip, deflate")  
23         r.Headers.Set("Accept-Language", "zh-CN, zh;q=0.9")  
24         //r.Headers.Set("User-Agent", "Mozilla/5.0 (Windows  
25             NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)  
26             Chrome/72.0.3626.121 Safari/537.36")  
27         fmt.Println("Visiting", r.URL)  
28     })  
29  
30     // 打印响应状态码  
31     c.OnResponse(func(r *colly.Response) {  
32         fmt.Println("response received", r.StatusCode)  
33     })  
34     // 对响应的HTML元素处理  
35     c.OnHTML("title", func(e *colly.HTMLElement) {  
36         //e.Request.Visit(e.Attr("href"))  
37         fmt.Println("title:", e.Text)  
38     })  
39  
40     c.Visit("https://www.baidu.com")  
41 }
```

动手写17.3.2设置了大量请求头信息，User-Agent也可以在请求头中设置，运行结果如下：

```
Visiting https://www.baidu.com
response received 200
title: 百度一下，你就知道
```

17.3.2 实战爬取西刺代理IP

我们在写爬虫时，往往需要使用很多代理IP来防止IP被封。网上有很多免费的代理IP，虽然质量较差，但是作为学习使用还是可以满足我们的需求。

这里我们针对西刺代理进行爬取，网站地址为：<https://www.xicidaili.com/>。现在我们针对网站内的高匿代理进行爬取，网站截图如图17.3.1所示。



The screenshot shows the homepage of XiciDaili.com. At the top, there's a logo of a stylized leaf or flower, followed by the text "国内高匿代理IP" and "XiciDaili.com". Below the header, there are several navigation links: 首页, API提取, 国内高匿代理, 国内普通代理, 国内HTTPS代理, 国内HTTP代理, and 代理小百科. A breadcrumb trail indicates the current location: 当前位置: 首页 > 国内高匿代理. Below this, there's a promotional banner with bullet points: 每天15万不重复IP, 24小时自助提取, 更快的访问速度, 支持客户端提取, 专属客户服务, 支持API调用. It also features two price options: 9元包天 不限量 and 98元包月 不限量, and a "购买专业版" button. A note below the banner states: "公告: 本站所有代理IP地址均收集整理自国内公开互联网, 本站不维护运营任何代理服务器, 请自行筛选。" The main content is a table listing proxy IP addresses. The columns are: 国家, IP地址, 端口, 服务器地址, 是否匿名, 类型, 速度, 连接时间, 存活时间, and 验证时间. The table contains 10 rows of data, each representing a different proxy entry.

国家	IP地址	端口	服务器地址	是否匿名	类型	速度	连接时间	存活时间	验证时间
■	1.198.73.66	9999	河南	高匿	HTTPS	<div style="width: 100px;"></div>	■	6天	19-05-25 15:21
■	1.198.73.70	9999	河南	高匿	HTTPS	<div style="width: 100px;"></div>	■	6天	19-05-25 15:21
■	1.197.203.129	9999	河南济源	高匿	HTTPS	<div style="width: 100px;"></div>	■	1分钟	19-05-25 15:00
■	58.253.153.85	9999	广东揭阳	高匿	HTTPS	<div style="width: 100px;"></div>	■	1分钟	19-05-25 15:00
■	112.85.128.47	9999	江苏南通	高匿	HTTPS	<div style="width: 100px;"></div>	■	2天	19-05-25 15:00
■	112.85.167.167	9999	江苏南通	高匿	HTTPS	<div style="width: 100px;"></div>	■	28天	19-05-25 14:21
■	119.254.94.93	46323	北京	高匿	HTTPS	<div style="width: 100px;"></div>	■	227天	19-05-25 14:20
■	163.204.246.6	9999	广东	高匿	HTTP	<div style="width: 100px;"></div>	■	1分钟	19-05-25 14:20
■	115.199.197.185	8118	浙江杭州	高匿	HTTPS	<div style="width: 100px;"></div>	■	1分钟	19-05-25 14:00
■	202.120.38.131	80	上海	高匿	HTTP	<div style="width: 100px;"></div>	■	18天	19-05-25 14:00

图17.3.1 西刺的国内高匿代理IP截图

我们已经学会了如何使用colly爬取网页，现在我们需要了解的问题是如何解析网页的内容。我们所需的内容在IP地址那一列，解析网页需要使用OnHTML回调函数。

```
func (c *Collector) OnHTML(goquerySelector string, f HTMLCallback)
```

colly解析网页是基于goquery这个库，地址如下：

```
https://github.com/PuerkitoBio/goquery
```

goquery将jQuery的语法和特性引入到了Go语言中，我们需要学习jQuery查找元素的语法，这部分将留给读者自行学习，这里直接给出获取IP的代码，如下所示：

```

01 c.OnHTML("#ip_list", func(e *colly.HTMLElement) {
02     e.ForEach("tr > td:nth-of-type(2)", func(i int, element *colly.
03             HTMLElement) {
04                 fmt.Println(element.Text)
05             })
06 })

```

#ip_list 表示查找 dom 节点中 id 为 ip_list 的节点，查找到后在该 dom 节点中继续查找满足 tr > td:nth-of-type(2) 条件的 dom 节点。nth-of-type(2) 表示获取 tr 元素中的第二个 td 元素 dom。完整代码如下所示：

动手写 17.3.3

```

01 package main
02
03 import (
04     "fmt"
05     "github.com/gocolly/colly"
06     "strconv"
07 )
08
09 func main() {
10     // 初始化
11     c := colly.NewCollector(
12         colly.UserAgent("Mozilla/5.0 (Windows NT 10.0; WOW64)
13         AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121
14         Safari/537.36"))
15
16     // 发出请求时附的回调
17     c.OnRequest(func(r *colly.Request) {
18         // Request 头部设定
19         r.Headers.Set("Connection", "keep-alive")
20         r.Headers.Set("Accept", "*/*")
21         r.Headers.Set("Origin", "")
22         r.Headers.Set("Accept-Encoding", "gzip, deflate")
23         r.Headers.Set("Accept-Language", "zh-CN, zh;q=0.9")
24     })
25 }

```

```
22         fmt.Println("Visiting", r.URL)
23
24     })
25
26     // 打印响应状态码
27     c.OnResponse(func(r *colly.Response) {
28         fmt.Println("response received", r.StatusCode)
29     })
30
31     // 对响应的HTML元素处理
32     c.OnHTML("title", func(e *colly.HTMLElement) {
33         //e.Request.Visit(e.Attr("href"))
34         fmt.Println("title:", e.Text)
35     })
36
37     // 在OnResponse之后被调用
38     c.OnHTML("#ip_list", func(e *colly.HTMLElement) {
39         e.ForEach("tr > td:nth-of-type(2)", func(i int, element
40             *colly.HTMLElement) {
41             fmt.Println(element.Text)
42         })
43     })
44
45     // 请求出错时，打印错误原因
46     c.OnError(func(r *colly.Response, e error) {
47         fmt.Println(e.Error())
48     })
49
50     // 设置visit的线程数
51     c.Limit(&colly.LimitRule{
52         Parallelism: 2,
53     })
54     // 爬取完成后，在OnHTML之后被调用
```

```
55     c.OnScraped(func(r *colly.Response) {
56         fmt.Println("Finished", r.Request.URL)
57     })
58
59     // 获取前三页的IP地址
60     for i := 1; i <= 3; i++ {
61         c.Visit("https://www.xicidaili.com/nn/" + strconv.
62             Itoa(i))
63     }
```

动手写 17.3.3 运行后，会爬取西刺高匿代理的前三页 IP 并打印输出，运行结果如下：

```
Visiting https://www.xicidaili.com/nn/1
response received 200
title: 国内高匿免费HTTP代理IP__第1页国内高匿
1.198.73.66
.....
36.250.156.30
Finished https://www.xicidaili.com/nn/1
Visiting https://www.xicidaili.com/nn/2
response received 200
title: 国内高匿免费HTTP代理IP__第2页国内高匿
1.193.244.88
.....
1.192.242.99
Finished https://www.xicidaili.com/nn/2
Visiting https://www.xicidaili.com/nn/3
response received 200
title: 国内高匿免费HTTP代理IP__第3页国内高匿
60.13.42.114
.....
122.193.245.220
Finished https://www.xicidaili.com/nn/3
```

17.4 小结

- ◇ 理解HTTP协议基础。
- ◇ 理解什么是RESTful API模式。
- ◇ 掌握如何使用Go语言标准库发起一次HTTP请求。
- ◇ 学会并熟练使用爬虫框架gocolly会使你编写爬虫的效率大大提升。

17.5 知识拓展

curl工具详解

1. curl简介

curl是一个利用URL语法在命令行下工作的文件传输工具，于1997年首次发行。它支持文件上传和下载，所以是综合传输工具，但按传统，习惯称curl为下载工具。curl还包含了用于程序开发的libcurl。

curl支持的通信协议有FTP、FTPS、HTTP、HTTPS、TFTP、SFTP、Gopher、SCP、TELNET、DICT、FILE、LDAP、LDAPS、IMAP、POP3、SMTP和RTSP。

该工具的官网地址为：<https://curl.haxx.se>。

2. curl安装

对于Linux系统而言，系统已默认安装此工具，无须再次安装。对于Windows系统，大部分发行版本都默认没有安装curl工具，部分系统如Win10已默认安装。对于没有安装curl工具的系统，可自行下载安装包安装，下载地址为：<https://curl.haxx.se/windows/>。

3. curl常见命令

(1) 查看网页源码

```
curl https://curl.haxx.se
```

这是最简单的使用方法，用这个命令获得了<https://curl.haxx.se>指向的页面。同样，如果这里的URL指向的是一个文件或者一幅图，都可以直接下载到本地。

如果要把这个网页保存下来，可以使用-o参数：

```
curl -o https://curl.haxx.se
```

(2) 显示头信息

-i参数可以显示HTTP的Response的头信息，连同网页代码一起。-I参数则只显示HTTP的Response的头信息。

```
curl -i https://curl.haxx.se
```

响应结果如下：

```
HTTP/1.1 301 Moved Permanently
Server: Varnish
Retry-After: 0
Location: https://curl.haxx.se/
Content-Length: 0
Accept-Ranges: bytes
Date: Sun, 25 Aug 2019 12:44:49 GMT
Via: 1.1 varnish
Connection: close
X-Served-By: cache-hnd18732-HND
X-Cache: HIT
X-Cache-Hits: 0
X-Timer: S1566737089.235760,VS0,VE0
```

(3) 显示通信过程

-v参数可以显示一次http通信的整个过程，包括端口连接和HTTP的Request头信息。

```
curl -v https://curl.haxx.se
```

响应结果如下：

```
* Rebuilt URL to: https://curl.haxx.se/
*   Trying 151.101.110.49...
* TCP_NODELAY set
* Connected to curl.haxx.se (151.101.110.49) port 80 (#0)
> GET / HTTP/1.1
> Host: curl.haxx.se
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 301 Moved Permanently
< Server: Varnish
< Retry-After: 0
< Location: https://curl.haxx.se/
< Content-Length: 0
```

```
< Accept-Ranges: bytes
< Date: Sun, 25 Aug 2019 12:46:09 GMT
< Via: 1.1 varnish
< Connection: close
< X-Served-By: cache-tyo19944-TYO
< X-Cache: HIT
< X-Cache-Hits: 0
< X-Timer: S1566737170.884661,VS0,VE0
<
* Closing connection 0
```

(4) 发送表单信息

发送表单信息有GET和POST两种方法。GET方法相对简单，只要把数据附在网址后面即可。

```
curl example.com/form.cgi?data=xxx
```

POST方法必须把数据和网址分开，curl就要用到--data或者-d参数。

```
curl -X POST --data "data=xxx" example.com/form.cgi
```

(5) 选择使用的请求方法

默认的HTTP请求方法是GET，使用-X参数可以支持其他方法。

```
01 curl -X POST https://curl.haxx.se
02 curl -X PUT https://curl.haxx.se
```

(6) 设置UA

--user-agent参数用来表示客户端的设备信息，服务器有时会根据这个字段，针对不同设备返回不同格式的网页，比如手机版和桌面版。

```
curl --user-agent "[User Agent]" https://curl.haxx.se
```

(7) 设置Cookie

使用--cookie参数，可以让curl发送Cookie。

```
curl --cookie "name=xxx" https://curl.haxx.se
```

(8) 增加头信息

有时需要在HTTP的Request之中自行增加一个头信息，--header参数就可以起到这个作用。

```
curl --header "Content-Type:application/json" http://example.com
```

>> 第 18 章

HTTP 编程（下）<<

在上一章中，我们详细介绍了如何使用Go语言编写HTTP客户端程序，本章将会延续上一章的话题（HTTP），讲解如何使用Go语言来实现HTTP和HTTPS服务端程序。

HTTP是典型的C/S架构，客户端向服务端发送请求（request），服务端做出应答（response），本章将会详细介绍如何创建一个HTTP服务端。服务端与客户端关注点有些不同，我们会学习如何进行路由注册、开启监听、处理连接以及编写路由处理函数。

18.1

错误处理的方式

Web服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的程序，可以向浏览器等Web客户端提供文档，也可以放置网站文件，让全世界浏览，还可以放置数据文件，让全世界下载。

HTTP协议定义服务器端和客户端之间文件传输的沟通方式。目前HTTP协议的版本是HTTP/1.1。RFC 2616描述了HTTP协议的具体信息。这个协议已经成为浏览器和Web站点之间的标准。

客户端与服务端传输文件的最基本的过程是：

1. 客户端连接一个主机；
2. 服务器接收连接；
3. 客户端请求一个文件；
4. 服务器发送一个应答。

18.1.1 最常见的Web服务端

目前最主流的几个Web服务器是Apache、Tomcat、IIS、Nginx。下面就来了解一下这些服务器的主要作用。

1. Apache

Apache是一个纯粹的Web服务器，经常与Tomcat配对使用。它对HTML页面具有强大的解释能力，但是不能解释嵌入页面内的服务器端脚本代码（JSP/Servlet）。

2. Tomcat

早期的Tomcat是一个嵌入Apache内的JSP/Servlet解释引擎，Apache+Tomcat就相当于IIS+ASP（动态服务器页面）。后来的Tomcat已不再嵌入Apache内，Tomcat进程独立于Apache进程运行，且Tomcat已经是一个独立的Servlet和JSP容器，业务逻辑层代码和界面交互层代码可以分离了。因此，有人把Tomcat叫作轻量级应用服务器。

3. IIS

微软早期的IIS，就是一个纯粹的Web服务器。后来它嵌入了ASP引擎，可以解释VBScript和JScript服务器端代码，这时，它就可以兼作应用服务器。从原理上说，它勉强可以称为应用服务器，但确切地说，它是兼有一点应用服务器功能的Web服务器。

4. Nginx

Nginx是一款轻量级的Web服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器，在BSD-like协议下发行。其特点是占用内存少，并发能力强，事实上Nginx的并发能力确实在同类型的网页服务器中表现较好。

总之，Apache及Nginx是纯粹的Web服务器，而Tomcat和IIS因为具有了解释执行服务器端代码的能力，可以称作轻量级应用服务器或带有服务器功能的Web服务器。WebLogic、WebSphere因为能提供强大的J2EE功能，毫无疑问是绝对的应用服务器。对于处于中间位置的Tomcat，它可以配合纯Web服务器Apache一起使用，也可以作为应用服务器的辅助与应用服务器一起部署。Go语言也可以通过代码直接创建一个功能齐全的Web服务器。

18.1.2 请求方法

根据HTTP标准，HTTP请求可以使用多种请求方法：

1. HTTP 1.0定义了三种请求方法：GET、POST和HEAD方法。
2. HTTP 1.1新增了五种请求方法：OPTIONS、PUT、DELETE、TRACE和CONNECT方法。

请求方法指定了客户端想对指定的资源/服务器做何种操作，所有方法全为大写，各个方法的解释如下：

- ◇ GET：请求获取Request-URI所标识的资源。

- ◇ POST：在 Request-URI 所标识的资源后附加新的数据。
- ◇ HEAD：请求获取由 Request-URI 所标识的资源的响应消息报头。
- ◇ OPTIONS：请求查询服务器的性能，或者查询与资源相关的选项和需求。
- ◇ PUT：请求服务器存储一个资源，并用 Request-URI 作为其标识。
- ◇ DELETE：请求服务器删除 Request-URI 所标识的资源。
- ◇ TRACE：请求服务器回送收到的请求信息，主要用于测试或诊断。
- ◇ CONNECT：HTTP/1.1 协议中预留给能够将连接改为通道方式的代理服务器。

18.1.3 响应状态码

HTTP 状态码（HTTP Status Code）是用以表示网页服务器超文本传输协议响应状态的 3 位数字代码。它由 RFC 2616 规范定义，所有状态码的第一个数字代表了响应的五种状态之一。所示的消息短语是典型的，但是可以提供任何可读取的替代方案。除非另有说明，状态码是 HTTP/1.1 标准的一部分。

状态码为客户端提供了一种理解事务处理结果的便捷方式，HTTP 状态码被分为了五大类：

- ◇ 1xx：信息性状态码，表示请求已经接受，正在继续处理。
- ◇ 2xx：成功状态码，表示请求已经被成功接收、理解、接受。
- ◇ 3xx：重定向状态码，表示要完成请求必须进行更进一步的操作。重定向状态码用于重定向资源，如果资源已被移动，可发送一个重定向状态码和一个可选的 Location 首部来告知客户端资源已被移走，以及在哪里可以找到它，这样，浏览器就可以在不打扰使用者的情况下，透明地转入新的位置了。
- ◇ 4xx：客户端错误状态码，表示请求有语法错误或请求无法实现。客户端错误状态码反馈客户端请求导致的错误，一般由浏览器来处理，只有少量错误（如 404）会反馈到用户面前。
- ◇ 5xx：服务器错误状态码，表示服务器未能实现合法的请求。当服务器自身出错后，回送报文中将包含此告警码。

具体状态码详解参见“18.7 知识拓展”部分。

18.2

HTTP 服务端

Go 语言的标准库 net/http 提供了 http 编程有关的接口，封装了内部 TCP 连接和报文解析的复杂琐碎的细节，使用者只需要与 http.request 和 http.ResponseWriter 这两个对象交互就行。也就是说，我们只要写一个 handler，请求会通过参数传递进来，而它要做的就是根据请求的数据做处理，把结果写到 Response 中。

18.2.1 Server和Response

Go语言封装好了一个针对HTTP服务器的Server结构体，只需要使用它就可以快速创建一个HTTP服务器。先来看这个结构体的成员字段。

```
type Server struct {
    Addr                 string
    Handler             Handler
    ReadTimeout         time.Duration
    WriteTimeout        time.Duration
    MaxHeaderBytes      int
    TLSConfig           *tls.Config
    TLSNextProto        map[string]func(*Server, *tls.Conn, Handler)
    ConnState           func(net.Conn, ConnState)
    ErrorLog            *log.Logger
}
```

Server类型定义了运行HTTP服务端的参数，共有九个成员字段，字段含义分别为：

- ◇ **Addr**: 监听的TCP地址，如果为空字符串会使用":http"。
- ◇ **Handler**: 调用的处理器，如为nil会调用http.DefaultServeMux。
- ◇ **ReadTimeout**: 请求的读取操作在超时前的最大持续时间。
- ◇ **WriteTimeout**: 回复的写入操作在超时前的最大持续时间。
- ◇ **MaxHeaderBytes**: 请求的头域最大长度，如为0则用DefaultMaxHeaderBytes。
- ◇ **TLSConfig**: 可选的TLS配置，用于ListenAndServeTLS方法。
- ◇ **TLSNextProto**: TLSNextProto（可选）指定一个函数在一个NPN型协议升级出现时接管TLS连接的所有权。映射的键为商谈的协议名；映射的值为函数，该函数的Handler参数应处理HTTP请求，并且初始化Handler.ServeHTTP的*Request参数的TLS和RemoteAddr字段（如果未设置）。连接在函数返回时会自动关闭。
- ◇ **ConnState**: ConnState字段指定一个可选的回调函数，该函数会在一个与客户端的连接改变状态时被调用。
- ◇ **ErrorLog**: ErrorLog指定一个可选的日志记录器，用于记录接收连接时的错误和处理器不正常的行为。如果本字段为nil，日志会通过log包的标准日志记录器写入os.Stderr。

Server类型主要有如下四个方法，我们最常使用的是ListenAndServe，该方法可以指定监听地址并启动一个HTTP服务器。

```
func (s *Server) SetKeepAlivesEnabled(v bool)
func (srv *Server) Serve(l net.Listener) error
func (srv *Server) ListenAndServe() error
func (srv *Server) ListenAndServeTLS(certFile, keyFile string) error
```

其中每个方法的作用如下所示：

```
func (s *Server) SetKeepAlivesEnabled(v bool)
```

`SetKeepAlivesEnabled`控制是否允许HTTP闲置连接重用（keep-alive）功能。默认该功能总是被启用的，只有资源非常紧张的环境或者服务端在关闭进程中时，才应该关闭该功能。

```
func (srv *Server) Serve(l net.Listener) error
```

`Serve`会接收监听器`l`收到的每一个连接，并为每一个连接创建一个新的服务go程。该go程会读取请求，然后调用`srv.Handler`回复请求。

```
func (srv *Server) ListenAndServe() error
```

`ListenAndServe`监听`srv.Addr`指定的TCP地址，并且会调用`Serve`方法处理接收到的连接。如果`srv.Addr`为空字符串，会使用":http"。

```
func (srv *Server) ListenAndServeTLS(certFile, keyFile string) error
```

`ListenAndServeTLS`监听`srv.Addr`确定的TCP地址，并且会调用`Serve`方法处理接收到的连接。必须提供证书文件和对应的私钥文件。如果证书是由权威机构签发的，`certFile`参数必须是顺序串联的服务端证书和CA证书。如果`srv.Addr`为空字符串，会使用":https"。

了解到`Server`是如何创建之后，我们再来学习如何构造一个响应。在平时的编程中，我们一般不会去直接构造响应，而是通过路由处理器`Handler`给出的已构造好的响应，对响应设置一些属性和内容。响应类型`Response`和请求类型`Request`非常相似，我们只需要大致了解响应包含了哪些字段信息即可。

```
type Response struct {
    Status      string           // 例如"200 OK"
    StatusCode int              // 例如200
    Proto       string           // 例如"HTTP/1.0"
    ProtoMajor  int              // 例如1
    ProtoMinor int              // 例如0
```

```
Header Header          // 响应Header头域
Body io.ReadCloser    // 响应Body
ContentLength int64   // 响应的长度
TransferEncoding []string
Close bool            // 是否关闭
Trailer Header       // Trailer字段保存和头域相同格式的trailer键值对
Request *Request     // 响应对应的请求
TLS *tls.ConnectionState // TLS包含接收到该回复的TLS连接的信息
}
```

我们最常使用的属性有StatusCode、Header、Body和Request。

- ◇ **StatusCode:** StatusCode表示响应的状态码，通过对请求的解析处理进行相应的设置。关于状态码可查看“18.7 知识拓展”部分。
- ◇ **Header:** Header保管头域的键值对。如果回复中有多个头的键相同，Header中保存为该键对应用逗号分隔串联起来的这些头的值。被本结构体中的其他字段复制保管的头（如ContentLength）会从Header中删掉。Header中的键都是规范化的，参见CanonicalHeaderKey函数。
- ◇ **Body:** Body代表回复的主体。Client类型和Transport类型会保证Body字段总是非nil的，即使回复没有主体或主体长度为0。关闭主体是调用者的责任。如果服务端采用“chunked”传输编码发送的回复，Body字段会自动进行解码。
- ◇ **Request:** Request是用来获取此回复的请求。Request的Body字段是nil（因为已经被用掉了），这个字段是被Client类型发出请求并获得回复后填充的。

18.2.2 启动HTTP服务器

Go语言http包提供了http.ListenAndServe方法，ListenAndServe监听TCP地址addr，并且会使用handler参数调用Serve函数处理接收到的连接。handler参数一般会设为nil，此时会使用DefaultServeMux。ListenAndServe函数如下所示：

```
func ListenAndServe(addr string, handler Handler) error {
    server := &Server{Addr: addr, Handler: handler}
    return server.ListenAndServe()
}
```

基于http.ListenAndServe方法我们可以快速地搭建起一个HTTP服务端程序。

动手写18.2.1

```

01 package main
02
03 import (
04     "fmt"
05     "log"
06     "net/http"
07 )
08
09 func myfunc(w http.ResponseWriter, r *http.Request) {
10     fmt.Fprintf(w, "hello world!")
11 }
12
13 func main() {
14     http.HandleFunc("/hello", myfunc)
15     log.Fatal(http.ListenAndServe(":8080", nil))
16 }
```

动手写18.2.1中启动了一个监听本机地址、端口为8080的服务器，我们可以使用curl命令访问这个服务器。

```
> curl http://127.0.0.1:8080/hello
hello world!
```

其中，`log.Fatal(http.ListenAndServe(":8080", nil))`这句代码表示，如果服务器监听启动或在运行时出错，就会打印出错信息，等同于以下代码：

```

err := http.ListenAndServe(":8080", nil)
if err != nil {
    log.Fatal("ListenAndServe: ", err)
}
```

18.2.3 启动HTTPS服务器

http包还提供了`http.ListenAndServeTLS`函数来启动一个HTTPS服务端，`ListenAndServeTLS`函数和`ListenAndServe`函数的行为基本一致，除了它期望的是HTTPS连接之外。此外，必须提供证书文

件和对应的私钥文件。如果证书是由权威机构签发的，certFile参数必须是顺序串联的服务端证书和CA证书。如果srv.Addr为空字符串，会使用":https"。ListenAndServeTLS函数构造如下所示：

```
func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error {
    server := &Server{Addr: addr, Handler: handler}
    return server.ListenAndServeTLS(certFile, keyFile)
}
```

我们也可以自己构造CA证书，但它们是不被信任的，如图18.2.1所示。



您的连接不是私密连接

攻击者可能会试图从 **127.0.0.1** 窃取您的信息（例如：密码、通讯内容或信用卡信息）。[了解详情](#)

NET::ERR_CERT_AUTHORITY_INVALID

您可以选择向 Google 发送一些系统信息和网页内容，以帮助我们改进安全浏览功能。[隐私权政策](#)

高级

返回安全连接

图18.2.1 自己构造的证书不被信任

提示

认证机构（CA，Certificate Authority）是采用公开密钥基础技术，专门提供网络身份认证服务，负责签发和管理数字证书，且具有权威性和公正性的第三方信任机构。CA的作用类似于我们现实生活中颁发证件的机构，如身份证办理机构等。

CA证书，顾名思义，就是CA颁发的证书。人人都可以找工具制作证书，但是自己制作出来的证书是没有用处的，因为自制证书不具权威性。

在Linux系统下我们可以使用openssl创建证书，如果是CentOS系统可以使用yum命令安装openssl。

```
yum install openssl
```

安装完后，运行如下指令创建一个证书，证书创建过程中，需要填写证书的相关信息，如证书所属的国家、地区、公司、组织等。

```
openssl genrsa -des3 -out server.key 2048      #生成server.key
openssl rsa -in server.key -out server.key      #生成server.key去掉密码,
用来生成server.crt
openssl req -new -x509 -key server.key -out server-ca.crt -days 3650
#生成server-ca.crt, 用来生成server.crt, 需要填写使用者信息
openssl req -new -key server.key -out server.csr    #生成server.csr, 用
来生成server.crt
openssl x509 -req -days 3650 -in server.csr -CA server-ca.crt -CAkey
server.key -CAcreateserial -out server.crt
```

运行完上述命令后就会在当前目录下生成多个文件，我们需要的只是server.crt和server.key文件作为证书和证书密钥，并通过构造的证书创建一个HTTPS服务端。

动手写18.2.2

```
01 package main
02
03 import (
04     "fmt"
05     "net/http"
06     "log"
07 )
08
09 func myfunc(w http.ResponseWriter, r *http.Request) {
10     fmt.Fprintf(w, "hello world!")
11 }
12
13 func main() {
14     http.HandleFunc("/hello", myfunc)
15     log.Fatal(http.ListenAndServeTLS(":8443", "server.crt",
16                                     "server.key", nil))
17 }
```

动手写18.2.2创建了一个HTTPS服务端，端口为8443，直接使用curl命令访问这个服务端会提示证书不可信的错误。

```
> curl https://127.0.0.1:8443/hello
curl: (77) schannel: next InitializeSecurityContext failed: SEC_E_
UNTRUSTED_ROOT (0x80090325) - 证书链是由不受信任的颁发机构颁发的。
```

这时需要添加“`--insecure`”忽略证书错误，运行后结果如下：

```
> curl --insecure https://127.0.0.1:8443/hello
hello world!
```

18.2.4 静态文件

`http`包提供了`http.FileServer`方法来创建一个静态文件服务器。

```
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`。

```
http.Handle("/", http.FileServer(http.Dir("/tmp")))
```

我们来实现一个最简单的静态文件服务端。

动手写18.2.3

```
01 package main
02
03 import (
04     "log"
05     "net/http"
06 )
07
08 func main() {
09     h := http.FileServer(http.Dir("./static/"))
10     log.Fatal(http.ListenAndServe(":8080", h))
11 }
```

动手写18.2.3创建了一个基于当前目录的`static`目录的文件服务器，浏览器访问该服务器可以看到`static`目录下的静态资源，如图18.2.2所示。

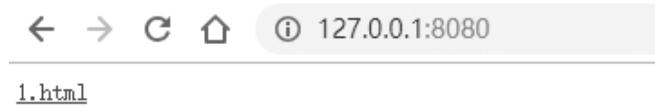


图18.2.2 访问static目录下的静态资源

一般情况下，我们需要限制静态资源文件的访问路由，如访问“`http://127.0.0.1:8080/static/`”目录才会显示当前static目录的静态资源，这时可以使用`http.Handle`来实现。

动手写18.2.4

```

01 package main
02
03 import (
04     "fmt"
05     "log"
06     "net/http"
07 )
08
09 func index(w http.ResponseWriter, r *http.Request) {
10     fmt.Fprintf(w, "Hello!")
11 }
12
13 func main() {
14     http.HandleFunc("/", index)
15
16     // 设置静态目录
17     fsh := http.FileServer(http.Dir("./static"))
18     http.Handle("/static/", http.StripPrefix("/static/", fsh))
19
20     log.Fatal(http.ListenAndServe(":8080", nil))
21 }
```

动手写18.2.4创建了一个功能更强大的资源服务器，能同时提供动态和静态资源，访问根目录可以获得Hello响应，如图18.2.3所示。



图18.2.3 访问根目录

访问 “<http://127.0.0.1:8080/static/>” 目录，可以看到所有静态资源，如图18.2.4所示。



图18.2.4 访问static目录

18.3 路由

服务端在接受request的过程中最重要的一个环节就是路由（router），路由将URL匹配信息与处理器函数（handler）连接起来，当用户访问的URL与路由中的URL匹配时，会调用对应的处理器（handler）。Go中内置了一个默认路由DefaultServeMux。使用路由前我们需要先明白什么是路由处理器Handler，它的结构如下：

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

只要是实现了Handler接口的类型，都可以作为路由处理器来实现对请求的处理及响应。我们创建一个helloHandler类型来实现Handler接口，并创建一个HTTP服务器。

动手写18.3.1

```
01 package main
02
03 import (
04     "net/http"
05 )
06
07 type helloHandler struct{}
```

```

09 func (h *helloHandler) ServeHTTP(w http.ResponseWriter, r *http.
    Request) {
10     w.Write([]byte("Hello world!"))
11 }
12
13 func main() {
14     http.Handle("/", &helloHandler{})
15     http.ListenAndServe(":8080", nil)
16 }
```

动手写18.3.1创建了一个helloHandler类型，并实现了Handler接口的ServeHTTP方法，可以使用http.Handle("/", &helloHandler{})来处理访问“/”目录的所有请求。

```
> curl http://127.0.0.1:8080/
Hello world!
```

在上一节中，我们并没有通过http.Handle来创建一个路由处理器处理请求，因为使用它会有一个不方便的地方：每次写Handler的时候，都要定义一个类型，然后编写对应的ServeHTTP方法，这个步骤对于所有Handler都是一样的。重复的工作总是可以抽象出来，http包也正是这么做了，它提供了http.HandleFunc方法，允许直接把特定类型的函数作为handler。

其实，HandleFunc只是一个适配器。

```

type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

它自动给f函数添加了HandlerFunc这个壳，最终调用的还是ServerHTTP，只不过会直接使用f(w, r)。这样封装的好处是：使用者可以专注于业务逻辑的编写，省去了很多重复的代码处理逻辑。

在实际开发中，HTTP接口会有许多的URL和对应的Handler。这里就要讲http包的另外一个重要的概念——路由转接器ServeMux。Mux是multiplexor的缩写，就是多路传输的意思（请求传过来，根据某种判断，分流到后端多个不同的地方）。ServeMux可以注册多个URL和handler的对应关系，并自动把请求转发到对应的handler进行处理。

ServeMux有如下所示的方法：

```
type ServeMux struct {
    // 内含隐藏或非导出字段
}

func NewServeMux() *ServeMux
func (mux *ServeMux) Handle(pattern string, handler Handler)
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request))
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string)
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
```

使用ServeMux就可以较为方便地创建URL和handler的对应关系。

动手写18.3.2

```
01 package main
02
03 import (
04     "fmt"
05     "log"
06     "net/http"
07 )
08
09 func index(w http.ResponseWriter, r *http.Request) {
10     fmt.Fprintf(w, "This is index!")
11 }
12
13 func hello(w http.ResponseWriter, r *http.Request) {
14     fmt.Fprintf(w, "hello world!")
15 }
16
17 func main() {
18     mux := http.NewServeMux()
19     mux.HandleFunc("/hello", hello)
20     mux.HandleFunc("/", index)
```

```

21
22     log.Fatal(http.ListenAndServe(":8080", mux))
23 }

```

动手写 18.3.2 创建了一个路由转发器 mux，注册了两个路由关系 “/” 和 “/hello”，使用 curl 命令访问结果如下：

```

> curl http://127.0.0.1:8080/
This is index!

> curl http://127.0.0.1:8080/hello
hello world!

```

提示

默认的 http 包中的 mux 不支持带参数的路由，根据经验，只要你的路由带有参数，并且这个项目的 API 数目超过了 10，就尽量不要使用 net/http 中默认的路由。在 Go 开源界应用最广泛的 router 是 httpRouter，很多开源的 router 框架都是基于 httpRouter 进行一定程度的改造的成果，如 gin。

18.4 中间件

中间件技术的作用是将业务和非业务代码功能进行解耦，对于非业务的通用代码，我们通过编写中间件的方式进行统一维护。

比如现在有一个需求，我们想要统计之前写的 hello 服务的处理耗时，需求很简单，我们对上面的程序进行少量修改：

```

func hello(wr http.ResponseWriter, r *http.Request) {
    start := time.Now()
    log.Printf("Started %s %s", r.Method, r.URL.Path)
    // 原程序的逻辑
    fmt.Fprintf(w, "hello world!")
    // 打印输出程序处理时间
    log.Printf("Completed %s in %v", r.URL.Path, time.Since(start))
}

```

但是当服务越来越多时，每一个服务都要添加一个这样的代码，并且当需求有变更时，代码更改就会显得非常复杂且容易出错。这时使用中间件技术就可以完美解决这个问题。中间件其实就是对http.Handler的封装，与之前所学的HandlerFunc相似。创建一个中间件使用如下方式：

```
func middlewareHandler(next http.Handler) http.Handler{
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.
Request) {
        // 执行handler之前的逻辑
        next.ServeHTTP(w, r)
        // 执行完毕handler后的逻辑
    })
}
```

我们来编写一下完整的代码。

动手写18.4.1

```
01 package main
02
03 import (
04     "net/http"
05     "fmt"
06     "time"
07     "log"
08 )
09
10 func index(w http.ResponseWriter, r *http.Request) {
11     fmt.Fprintf(w, "This is index!")
12 }
13
14 func hello(w http.ResponseWriter, r *http.Request) {
15     fmt.Fprintf(w, "hello world!")
16 }
17
18 func timeHandler(next http.Handler) http.Handler {
19     return http.HandlerFunc(func(w http.ResponseWriter, r *http.
Request) {
```

```

20         start := time.Now()
21         log.Printf("Started %s %s", r.Method, r.URL.Path)
22         next.ServeHTTP(w, r)
23         log.Printf("Completed %s in %v", r.URL.Path, time.
24             Since(start))
25     })
26
27 func main() {
28     // 创建一个路由转发器
29     mux := http.NewServeMux()
30     // 给hello函数注册timeHandler中间件
31     mux.HandleFunc("/hello", timeHandler (http.HandlerFunc(hello)))
32     mux.HandleFunc("/", timeHandler(http.HandlerFunc(index)))
33     http.ListenAndServe(":8080", mux)
34 }

```

动手写18.4.1创建了一个timeHandler，在后面的编程中，要使用这个中间件的函数时，只需要使用timeHandler(http.HandlerFunc(func))的方式注册一下就可以了。程序运行结果如下：

```

2019/05/26 15:56:14 Started GET /hello
2019/05/26 15:56:14 Completed /hello in 72.3665ms
2019/05/26 15:56:18 Started GET /
2019/05/26 15:56:18 Completed / in 0s

```

18.5 Web 框架gin

gin是一种路由框架，是对httpRouter进行简单的封装，然后提供定制的中间件和一些简单的小工具集成，主打轻量、易学、高性能。它运行速度快，具有分组的路由器、良好的崩溃捕获和错误处理，能非常好地支持中间件和JSON。

下载安装gin包：

```
go get -u github.com/gin-gonic/gin
```

18.5.1 启动HTTP服务器

gin创建一个HTTP服务器页的过程非常简单，使用gin.Default()初始化一个gin.Engine对象，后面的所有操作都是基于这个对象，如添加路由解析。

动手写18.5.1

```
01 package main
02
03 import "github.com/gin-gonic/gin"
04
05 func main() {
06     r := gin.Default()
07     r.GET("/ping", func(c *gin.Context) {
08         c.JSON(200, gin.H{
09             "message": "pong",
10         })
11     })
12     r.Run() // listen and serve on 0.0.0.0:8080
13 }
```

动手写18.5.1运行后使用curl命令访问，结果如下：

```
>curl http://127.0.0.1:8080/ping
{"message":"pong"}
```

18.5.2 静态文件

gin对静态文件的支持有多种方式，这里只介绍针对文件夹和单个文件的路由使用。针对文件夹，可以使用StaticFS(relativePath, fs)的方式，传入URL的相对路径和系统文件夹的路径（使用http.Dir获取）。单个文件和文件夹的路由方式较为相似，使用StaticFile(relativePath, filepath)，传入URL的相对路径和系统文件的相对路径（不用http.Dir获取，直接传入相对路径的字符串）。

动手写18.5.2

```
01 package main
02
03 import (
04     "net/http"
```

```

05      "github.com/gin-gonic/gin"
06  )
07
08 func main() {
09     router := gin.Default()
10
11     // 静态资源加载, 本例为css,js以及资源图片
12     router.StaticFS("/static", http.Dir("static"))
13     router.StaticFile("/favicon.ico", "./static/favicon.ico")
14
15     // Listen and serve on 0.0.0.0:80
16     router.Run(":8080")
17 }

```

动手写18.5.2运行后使用curl命令，请求结果如下：

```

> curl http://127.0.0.1:8080/static
1.html
favicon.ico

```

18.5.3 路由

使用gin框架的最大好处就是它的路由非常方便简洁，并且支持较为复杂的路由。这里针对它可以在URL地址中携带参数的路由进行学习。

URL路径携带参数有两种情况：一种是:arg，主要传递参数值如name；另一种是*arg，主要传递路径。

```
/user/:name/*action
```

两种方式都可以使用c.Param方法取出数据。

```

c.Param("name")           // 如 “zhangsan”
c.Param("action ")        // 如 “/send”

```

动手写18.5.3

```

01 package main
02
03 import (

```

```
04     "github.com/gin-gonic/gin"
05     "net/http"
06 )
07
08 func main() {
09     router := gin.Default()
10
11     router.GET("/user/:name", func(c *gin.Context) {
12         name := c.Param("name")
13         c.String(http.StatusOK, "Hello %s", name)
14     })
15
16     router.GET("/user/:name/*action", func(c *gin.Context) {
17         name := c.Param("name")
18         action := c.Param("action")
19         message := c.Request.Method + " => " + name + " is " +
20             action
21         c.String(http.StatusOK, message)
22     })
23
24     router.POST("/user/:name/*action", func(c *gin.Context) {
25         name := c.Param("name")
26         action := c.Param("action")
27         message := c.Request.Method + " => " + name + " is " +
28             action
29         c.String(http.StatusOK, message)
30     })
31 }
```

动手写18.5.3创建了三个路由，并且都是地址中带参数的路由。分别对这三个路由地址进行访问，结果如下：

```
> curl http://127.0.0.1:8080/user/john
```

```
Hello john

> curl http://127.0.0.1:8080/user/john/send
GET => john is /send

> curl -X POST http://127.0.0.1:8080/user/john/send
POST => john is /send
```

18.6 小结

- ◇ 了解常见的HTTP响应码。
- ◇ 掌握如何创建HTTP服务端。
- ◇ 学会使用路由转发器和处理器。
- ◇ 了解什么是Web中间件。
- ◇ 掌握Web框架gin的使用。

18.7 知识拓展

响应状态码详解

表18.7.1 信息性状态码

消息	说明
100 Continue	说明收到了请求的初始部分,请客户端继续,发送了这个状态码之后,服务端在收到请求之后必须进行响应。客户端需要发送一个携带了值为100 Continue的Expect请求首部,服务端收到后,将回送一个状态码为100的响应,并期待客户端发送请求实体到服务端处理
101 Switching Protocols	说明服务器正在根据客户端的指定,将协议切换成Update首部所列的协议

表18.7.2 成功状态码

消息	说明
200 OK	请求没问题,实体的主体部分包含了所请求的资源
201 Created	用于创建服务器对象的请求,如PUT,服务器必须在发送这个状态码之前创建好对象

(续上表)

消息	说明
202 Accepted	请求已被接受，但服务器还未对其执行任何动作，不能保证服务器会完成这个请求
203 Non-Authoritative Information	实体首部包含的信息不是来自于源端服务器，而是来自资源的一份副本。如果中间节点上有一份资源副本，但无法或者没有对它所发送的与资源有关的元信息进行验证，就会出现这种情况
204 No Content	响应报文中包含若干首部和一个状态行，但没有实体的主体部分。主要用于在浏览器不转为现实新文档的情况下，对其进行更新
205 Reset Content	告知浏览器清除当前页面中的所有 HTML 表单元素
206 Partial Content	客户端可以通过一些特殊的首部来获取部分或某个范围内的文档，这个状态码说明范围请求成功了

表18.7.3 重定向状态码

消息	说明
300 Multiple Choices	客户端请求一个实际指向多个资源的 URL 时会返回这个状态码
301 Moved Permanently	在请求的 URL 已被移除时使用。响应的 Location 首部中应该包含资源现在所处的 URL
302 Found	与 301 类似，但客户端应该使用 Location 首部给出的 URL 来临时定位资源，将来的请求仍应使用老的 URL
303 See Other	告知客户端应该用另一个 URL 来获取资源，新的 URL 位于响应报文的 Location 首部，其主要目的是允许 POST 请求的响应将客户端定位到某个资源上去
304 Not Modified	客户端可以通过所包含的请求首部，使其请求变成有条件的。客户端发起一个条件 GET 请求，而最近资源未被修改的话，就可以用这个状态码说明资源未被修改，带有这个状态码的响应不包含实体的主体部分
305 Use Proxy	用来说明必须通过一个代理来访问资源，代理的位置由 Location 首部给出
307 Temporary Redirect	是 302 状态码的细化，主要用在对非 GET、HEAD 方法的响应上；对于 POST 请求，表示请求还没有被处理，客户端应该向 Location 里的 URI 重新发起 POST 请求

表18.7.4 客户端错误状态码

消息	说明
400 Bad Request	告知客户端它发送了一个错误的请求

(续上表)

消息	说明
401 Unauthorized	与适当的首部一同返回，在这些首部中请求客户端在获取对资源的访问权之前对自己进行认证
402 Payment Required	目前还未使用，保留
403 Forbidden	用于说明请求被服务器拒绝了
404 Not Found	服务器无法找到所请求的 URL
405 Method Not Allowed	发起的请求中带有所请求的 URL 不支持的方法时，使用此状态码。会在响应中包含 Allow 首部，以告知客户端对所请求的资源可以使用哪些方法
406 Not Acceptable	客户端可以指定参数来说明它们愿意接收什么类型的实体，服务器没有与客户端可接受的 URL 相匹配的资源时，使用此代码
407 Proxy Authentication Required	与 401 类似，但要求对资源进行认证的代理服务器
408 Request Timeout	超时后，服务器可以回送此状态码，并关闭连接
409 Conflict	用于说明请求可能在资源上引发的一些冲突
410 Gone	与 404 类似，只是服务器曾经拥有过此资源，但此资源已经被移除
411 Length Required	服务器要求在请求报文中包含 Content-Length 首部时使用
412 Precondition Failed	在客户端发起了条件请求，且其中一个条件失败了的时候使用。客户端包含了 Expect 首部时发起的就是条件请求
413 Request Entity Too Large	客户端发送的实体主体部分比服务器能够或者希望处理的要大时，使用此状态码
414 Request URI Too Long	客户端所发请求中的请求 URL 比服务器能够或者希望处理的要长时，使用此状态码
415 Unsupported Media Type	服务器无法理解或无法支持客户端所发实体的内容类型时，使用此状态码
416 Requested Range Not Satisfiable	请求报文所请求的是指定资源的某个范围，而此范围无效或无法满足时，使用此状态码
417 Expectation Failed	请求的 Expect 请求首部包含了一个期望，但服务器无法满足此期望时，使用此状态码

表18.7.5 服务器错误状态码

消息	说明
500 Internal Server Error	服务器出错
501 Not Implemented	客户端发起的请求超出服务器的能力范围
502 Bad Gateway	作为代理或网关使用的服务器从请求响应链的下一条链路上收到了一条伪响应
503 Service Unavailable	用来说明服务器暂时无法为请求提供服务，可以包含一个 Retry-After 首部，告诉客户端资源什么时候可用
504 Gateway Timeout	响应来自网关或代理，在等待另一个服务器对其请求进行响应时超时了
505 HTTP Version Not Supported	服务器收到的请求使用了它无法或不愿支持的协议版本

在之前的章节中，我们学习了如何实现HTTP编程，而HTTP是基于Socket协议的，在学习Socket之前，我们需要了解计算机网络的基本架构、什么是TCP/IP以及如何使用TCP/IP中的Socket连接实现网络通信。Socket是我们在使用Go语言的过程中会使用到的最底层的网络协议，大部分的网络通信协议都是基于TCP/IP的Socket协议。

19.1

计算机网络简介

计算机网络是指将地理位置不同的具有独立功能的多台计算机及其外部设备通过通信线路连接起来，在网络操作系统、网络管理软件及网络通信协议的管理和协调下，实现资源共享和信息传递的计算机系统。

19.1.1 OSI七层网络模型

OSI (Open System Interconnection, 开放式系统互联) 是一个开放性的通信系统互联参考模型。OSI模型有七层结构，每层都可以有几个子层。OSI的七层从上到下分别是：7-应用层、6-表示层、5-会话层、4-传输层、3-网络层、2-数据链路层、1-物理层。其中高层（即7、6、5、4层）定义了应用程序的功能，下面三层（即3、2、1层）主要面向通过网络的端到端的数据流。如图19.1.1所示。

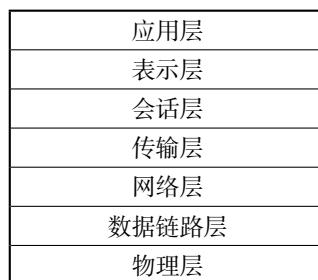


图19.1.1 OSI七层模型

网络中的计算机与终端间要想正确地传送信息和数据，必须在数据传输的顺序、数据的格式及内容等方面有一个约定或规则，这种约定或规则称作协议（protocol）。OSI的每一层都是为了完成各自功能，为了实现这些功能，就需要大家都遵守共同的协议。各层的功能与相关的协议如下。

1. 应用层

这一层是最靠近用户的OSI层，为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。

常见的协议有：HTTP、FTP、TFTP、SMTP、SNMP、DNS、TELNET、HTTPS、POP3、DHCP。

2. 表示层

表示层可确保一个系统的应用层所发送的信息能够被另一个系统的应用层读取。例如，PC程序与另一台计算机进行通信，其中一台计算机使用扩展二进制编码的十进制交换码（EBCDIC），而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通用格式来实现多种数据格式之间的转换。

常见的协议有：JPEG、ASCII、DECOIC。

3. 会话层

会话层通过传输层（端口号：传输端口与接收端口）建立数据传输的通路。它主要在系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是IP，也可以是MAC或者是主机名）。

常见的协议有：RPC、SCP、SSH、ZIP。

4. 传输层

传输层定义了一些传输数据的协议和端口号（WWW端口80等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高、数据量大的数据），UDP（用户数据报协议，与TCP特性恰恰相反，用于传输可靠性要求不高、数据量小的数据，如QQ聊天数据就是通过这种方式传输的）。传输层主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。我们常常把这一层数据叫作段。

常见的协议有：TCP、UDP。

5. 网络层

网络层为位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。互联网的发展使从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。

常见的协议有：ICMP、IGMP、IP、RARP。

6. 数据链路层

数据链路层定义了如何格式化数据以进行传输，以及如何控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。

常见的协议有：PPP、IEEE 802.3/802.2。

7. 物理层

物理层主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由1、0转化为电流强弱来进行传输，到达目的地后再转化为1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫作比特。

常见的协议有：Ethernet 802.3。

大多数的计算机网络都采用层次式结构，即将一个计算机网络分为若干层次，处在高层次的系统仅是利用较低层次的系统提供的接口和功能，无须了解低层实现该功能所采用的算法和协议；较低层次系统也仅是使用从高层系统传送来的参数，这就是层次间的无关性。因为有了这种无关性，层次间的每个模块可以用一个新的模块取代，只要新的模块与旧的模块具有相同的功能和接口，即使它们使用的算法和协议都不一样也无妨。

将网络分层有如下好处：

- ◇ 人们可以很容易地讨论和学习协议的规范细节。
- ◇ 层间的标准接口方便了工程模块化。
- ◇ 创建了一个更好的互联环境。
- ◇ 降低了复杂度，使程序更容易被修改，产品开发的速度更快。
- ◇ 每层利用紧邻的下层服务，更容易记住各层的功能。

19.1.2 TCP/IP协议

互联网协议族（Internet Protocol Suite）是一个网络通信模型，包括整个网络传输协议家族，是互联网的基础通信架构。它常被称为TCP/IP协议族（TCP/IP Protocol Suite，或TCP/IP Protocols），简称为TCP/IP，这是因为该协议家族的两个核心协议TCP（Transmission Control Protocol，传输控制协议）和IP（Internet Protocol，网际互连协议）是该家族中最早通过的标准。

TCP/IP提供点对点的链接机制，将数据应该如何封装、定址、传输、路由以及在目的地如何接收，都加以标准化。它将软件通信过程抽象为四个抽象层，采取协议堆栈的方式，分别实现不同的通信协议。协议族下的各种协议，依其功能不同，被分别归属到这四个层次结构之中，常被视为简化的OSI七层模型，如表19.1.1所示。

表19.1.1 OSI模型与TCP/IP模型对应关系

OSI七层模型	TCP/IP模型
应用层	
表示层	应用层
会话层	

(续上表)

OSI七层模型	TCP/IP模型
传输层	传输层
网络层	网络层
数据链路层	
物理层	网络访问层

TCP/IP参考模型是首先由ARPANET所使用的网络体系结构。这个体系结构在它的两个主要协议出现以后被称为TCP/IP参考模型（TCP/IP Reference Model）。这一网络协议共分为四层：网络访问层、网络层、传输层和应用层。

1. 网络访问层（Network Access Layer）

在TCP/IP参考模型中并没有详细描述，只是指出主机必须使用某种协议与网络相连。

2. 网络层（Internet Layer）

这是整个体系结构的关键部分，其功能是使主机可以把分组发往任何网络，并使分组独立地传向目标。这些分组可能经由不同的网络，到达的顺序和发送的顺序也可能不同。高层如果需要顺序收发，那么就必须自行处理对分组的排序。TCP/IP参考模型的网络层使用网际互连协议（IP）。在功能上它与OSI参考模型的网络层非常相似。

3. 传输层（Transport Layer）

传输层使源端和目的端机器上的对等实体可以进行会话。这一层定义了两个端到端的协议：传输控制协议（TCP）和用户数据报协议（UDP）。TCP是面向连接的协议，它提供可靠的报文传输和对上层应用的连接服务。因此，除了基本的数据传输外，它还有可靠性保证、流量控制、多路复用、优先权和安全性控制等功能。UDP是面向无连接的不可靠传输的协议，主要用于不需要TCP的排序和流量控制等功能的应用程序。

4. 应用层（Application Layer）

应用层包含所有的高层协议，包括远程终端协议（TELNET）、文件传输协议（FTP）、电子邮件传输协议（SMTP）、域名服务（DNS）、网络新闻传输协议（NNTP）和超文本传输协议（HTTP）等。TELNET允许一台机器上的用户登录到另一台远程机器上并进行工作；FTP提供有效地将文件从一台机器上移到另一台机器上的方法；SMTP用于电子邮件的收发；DNS用于把主机名映射到网络地址；NNTP用于新闻的发布、检索和获取；HTTP用于在WWW上获取主页。

19.1.3 IPv4和IPv6

IPv4是Internet Protocol Version 4的缩写，即网际互连协议的第四版，也是第一个被广泛使用、构成现今互联网技术的基石的协议。1981年，约翰·伯斯泰尔（Jon Postel）在RFC 791中定

义了IP，IPv4可以运行在各种各样的底层网络上，比如端对端的串行数据链路（PPP协议和SLIP协议）、卫星链路等。局域网中最常用的是以太网。

传统的TCP/IP协议基于IPv4，属于第二代互联网技术，核心技术属于美国。它的最大问题是网络地址资源有限。从理论上讲，编址1600万个网络、40亿台主机，但采用A、B、C三类编址方式后，可用的网络地址和主机地址的数目大打折扣，以致IP地址已经枯竭。其中，北美占有 $\frac{3}{4}$ ，约30亿个，而人口最多的亚洲只有不到4亿个。截至2010年6月，中国的IPv4地址数量达到2.5亿，落后于4.2亿网民的需求。虽然用动态IP及NAT地址转换等技术实现了一些缓冲，但IPv4地址枯竭已经成为不争的事实。在此，专家提出IPv6的互联网技术，也正在推行，但从IPv4到IPv6的使用需要一段很长的过渡期。目前大部分网络以IPv4为主。

IPv6是Internet Protocol Version 6的缩写，它是IETF（The Internet Engineering Task Force，互联网工程任务组）设计的用于替代现行版本IP协议（IPv4）的下一代IP协议。与IPv4相比，IPv6具有以下几个优势：

- ◇ IPv6具有更大的地址空间。IPv4中规定IP地址长度为32，即有 $2^{32}-1$ 个地址；而IPv6中IP地址的长度为128，即有 $2^{128}-1$ 个地址。
- ◇ IPv6使用更小的路由表。IPv6的地址分配一开始就遵循聚类（Aggregation）的原则，这使路由器能在路由表中用一条记录（Entry）表示一片子网，大大减小了路由器中路由表的长度，提高了路由器转发数据包的速度。
- ◇ IPv6增加了增强的组播（Multicast）支持以及对流的控制（Flow Control），这使网络上的多媒体应用有了长足发展的机会，为服务质量（QoS，Quality of Service）控制提供了良好的网络平台。
- ◇ IPv6加入了对自动配置（Auto Configuration）的支持。这是对DHCP协议的改进和扩展，使网络（尤其是局域网）的管理更加方便和快捷。
- ◇ IPv6具有更高的安全性。在使用IPv6网络时用户可以对网络层的数据进行加密并对IP报文进行校验，极大增强了网络的安全性。

19.1.4 子网掩码

子网掩码（subnet mask）又叫网络掩码、地址掩码、子网络遮罩，它是一种用来指明一个IP地址的哪些位标识的是主机所在的子网，以及哪些位标识的是主机的位掩码。子网掩码不能单独存在，它必须结合IP地址一起使用。子网掩码只有一个作用，就是将某个IP地址划分成网络地址和主机地址两部分。

子网掩码——屏蔽一个IP地址的网络部分的“全1”比特模式。对于A类地址来说，默认的子网掩码是255.0.0.0；对于B类地址来说，默认的子网掩码是255.255.0.0；对于C类地址来说，默认的

子网掩码是255.255.255.0。

利用子网掩码可以把大的网络划分成子网，即VLSM（可变长子网掩码），也可以把小的网络归并成大的网络，即超网。

例如：

IP地址为：192.168.0.1

子网掩码为：255.255.255.0

转化为二进制进行运算：

IP地址为：11000000.10101000.00000000.00000001

子网掩码为：11111111.11111111.11111111.00000000

AND运算：11000000.10101000.00000000.00000000

转化为十进制后为：192.168.0.0

这里的AND运算法则：1与1为1，1与0为0，0与1为0，0与0为0，即当对应位均为1时结果为1，其余为0，如表19.1.2所示。

表19.1.2 AND运算

第1位	第2位	结果
0	0	0
0	1	0
1	0	0
1	1	1

19.2 Socket基础

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（Socket）。Socket的本质是编程接口（API），对TCP/IP的封装，TCP/IP也要提供可供程序员做网络开发所用的接口，这就是Socket编程接口。如果说HTTP是轿车，提供了封装或者显示数据的具体形式，那么Socket就是发动机，提供了网络通信的能力。

19.2.1 Socket简介

Socket的英文原义是“孔”或“插座”，作为BSD UNIX的进程通信机制，取后一种意思。Socket通常也称作“套接字”，用于描述IP地址和端口，是一个通信链的句柄，可以用来实现不同虚拟机或不同计算机之间的通信。在网络上的主机一般运行了多个服务软件，同时提供几种服

务。每种服务都打开一个Socket，并绑定到一个端口上，不同的端口对应于不同的服务。Socket正如其英文原义那样，像一个多孔插座。一台主机犹如布满各种插座的房间，每个插座都有一个编号，有的插座提供220伏交流电，有的提供110伏交流电，有的则提供有线电视节目。客户软件将插头插到不同编号的插座中，就可以得到不同的服务。

Socket起源于Unix，而Unix的基本哲学之一就是“一切皆文件”，都可以使用如下模式来操作。

```
打开 -> 读写write/read -> 关闭close
```

Socket就是该模式的一个实现，网络的Socket数据传输是一种特殊的I/O，Socket也是一种文件描述符。Socket的类型有两种：流式Socket和数据报式Socket。流式是一种面向连接的Socket，针对于面向连接的TCP服务应用；数据报式Socket是一种无连接的Socket，对应于无连接的UDP服务应用。

19.2.2 netcat工具简介

命令行工具nc，全称netcat，在测试TCP/IP的客户端和服务端时非常便利。本小节将介绍它的一些常见用法。

nc可以作为TCP服务的客户端使用，例如作为客户端连接监听在本机127.0.0.1上端口为1234的服务，命令如下：

```
nc 127.0.0.1 1234
```

命令行工具nc默认使用TCP协议。如果使用UDP协议，可以在执行nc命令时携带-u参数。

```
nc -u 127.0.0.1 1234
```

如果要使用nc模拟服务器，可以携带-l参数，nc将监听指定端口号的连接，例如在本机的1234端口建立一个TCP服务端。

```
nc -l -p 1234
```

同样，建立UDP服务器只需加-u参数。

```
nc -l -u -p 1234
```

```
nc -lup 1234 (简写)
```

如果希望netcat(1)生成详细的输出，可以使用-v和-vv参数，这些输出为排查网络连接故障提供了很大的便利。

```
nc -v 127.0.0.1 1234
```

nc工具可以在SourceForge网站上获取，地址如下。

```
http://netcat.sourceforge.net/
```

19.3

TCP编程

19.3.1 TCP简介

TCP是Transmission Control Protocol的缩写，中文名是传输控制协议。它是一种面向连接的、可靠的、基于字节流的传输层通信协议，由IETF的RFC 793定义。在简化的计算机网络OSI模型中，它完成第四层传输层所指定的功能，用户数据报协议（UDP）是同一层内另一个重要的传输协议。在互联网协议族中，TCP层是位于IP层之上、应用层之下的中间层。不同主机的应用层之间经常需要可靠的、像通道一样的连接，但是IP层不提供这样的流机制，而是提供不可靠的包交换。

TCP是因特网中的传输层协议，使用三次握手协议建立连接。当主动方发出SYN连接请求后，等待对方回答SYN+ACK，并最终对对方的SYN执行ACK确认。这种建立连接的方法可以防止产生错误的连接。TCP使用的流量控制协议是可变大小的滑动窗口协议。TCP三次握手的过程（图19.3.1）如下：

1. 客户端发送SYN（seq=J）报文给服务器端，进入SYN_SENT状态；
2. 服务器端收到SYN报文，回应一个SYN（seq=K）+ACK(ack=J+1) 报文，进入SYN_RCVD状态；
3. 客户端收到服务器端的SYN报文，回应一个ACK(ack=K+1) 报文，进入ESTABLISHED状态。

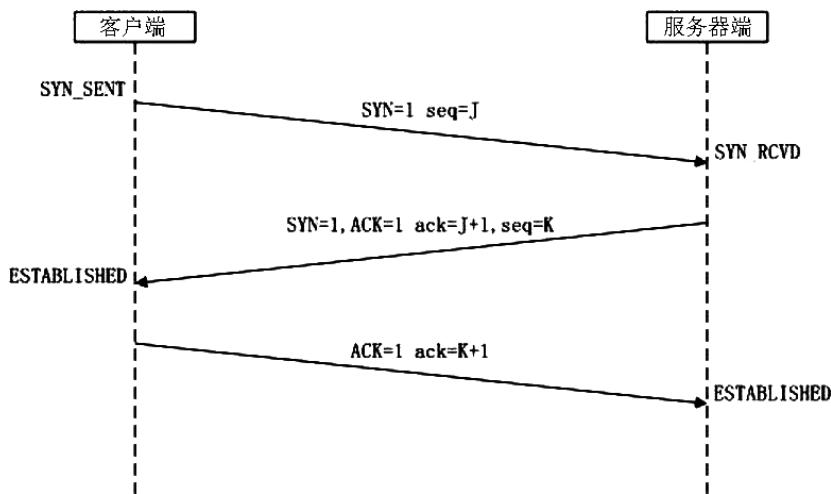


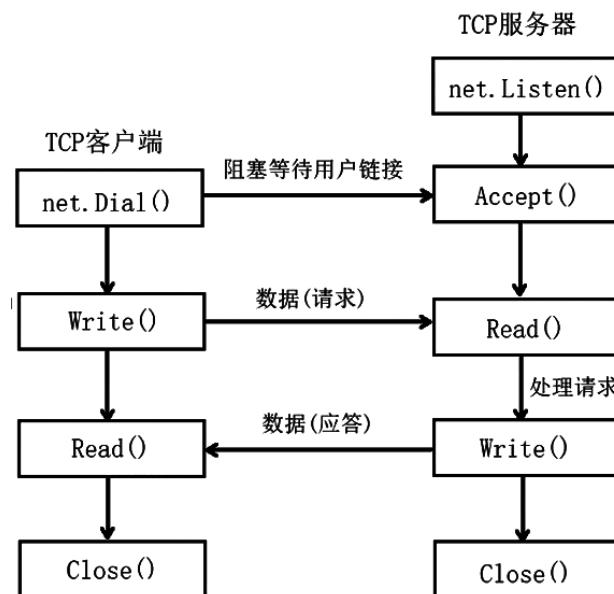
图19.3.1 TCP三次握手的过程

三次握手完成，TCP客户端和服务器端成功地建立连接，便可以开始传输数据了。

TCP通过下列方式来提供可靠性：

1. 应用数据被分割成TCP认为最适合发送的数据块。这和UDP完全不同，应用程序产生的数据长度将保持不变。由TCP传递给IP的信息单位称为报文段或段（segment）。
2. 当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。当TCP收到发自TCP连接另一端的数据时，它将发送一个确认。TCP有延迟确认的功能，若此功能没有打开，则是立即确认；功能打开，则由定时器触发确认时间点。
3. TCP将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP将丢弃这个报文段以及不确认收到此报文段（希望发送端超时并重发）。
4. 既然TCP报文段作为IP数据报来传输，而IP数据报的到达可能会失序，因此TCP报文段的到达也可能会失序。如有必要，TCP将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。
5. 既然IP数据报会发生重复，TCP的接收端必须丢弃重复的数据。
6. TCP还能提供流量控制。TCP连接的每一方都有固定大小的缓冲空间。TCP的接收端只允许另一端发送接收端缓冲区所能接纳的数据，这将防止较快主机致使较慢主机的缓冲区溢出。

在Go语言中使用net包来处理TCP网络编程，服务端需要使用net.Listen()来监听一个端口启动TCP服务，客户端需要使用net.Dial()来与TCP服务端进行连接请求，详细的流程如图19.3.2所示。



19.3.2 TCP客户端

Go语言提供了net包来实现Socket编程，大部分使用者只需要Dial、Listen和Accept函数提供的基本接口，以及相关的Conn和Listener接口。创建一个TCP客户端只需使用Dial函数即可。

```
func Dial(network, address string) (Conn, error)
```

在网络network上连接地址address，并返回一个Conn接口。可用的网络类型有：tcp、tcp4、tcp6、udp、udp4、udp6、ip、ip4、ip6、unix、unixgram、unixpacket。

对于TCP和UDP网络，地址格式是“host:port”或 “[host]:port”，例如：

```
Dial("tcp", "12.34.56.78:80")
Dial("tcp", "google.com:http")
Dial("tcp", "[2001:db8::1]:http")
Dial("tcp", "[fe80::1%lo0]:80")
```

对于IP网络，network必须是ip、ip4、ip6后跟冒号和协议号或者协议名，地址必须是IP地址字面值。

```
Dial("ip4:1", "127.0.0.1")
Dial("ip6:ospf", "::1")
```

我们使用Dial函数来尝试连接百度服务器。

动手写19.3.1

```
01 package main
02
03 import (
04     "net"
05     "log"
06 )
07
08 func main() {
09     // 尝试连接百度服务器
10     conn, err := net.Dial("tcp", "www.baidu.com:80")
11     if err != nil {
12         log.Fatal("连接失败!", err)
13     }
14     defer conn.Close()
15     log.Println("连接成功!")
16 }
```

动手写19.3.1开启了一个对百度服务器的80端口的TCP连接，如果没有报错，就表示连接成功，程序运行后输出如下：

```
2019/06/01 20:36:33 连接成功!
```

对于网络编程而言，我们推荐使用log包代替fmt包进行打印信息，log包打印时，会附加打印出时间，方便我们调试程序。log.Fatal表示当遇到严重错误时打印错误信息，并停止程序的运行。

Dial函数在连接时，如果端口未开放，尝试连接就会立刻返回服务器拒绝连接的错误。

动手写19.3.2

```
01 package main
02
03 import (
04     "net"
05     "log"
06 )
07
08 func main() {
09     // 尝试连接本地1234端口
10     conn, err := net.Dial("tcp", ":1234")
11     if err != nil {
12         log.Fatal("连接失败! ", err)
13     }
14     defer conn.Close()
15     log.Println("连接成功! ")
16 }
```

动手写19.3.2尝试连接本地（127.0.0.1）的1234端口，由于该端口未开放任何TCP服务，程序就会抛出连接失败的信息，如下所示：

```
2019/06/01 20:37:42 连接失败! dial tcp :1234: connectex: No connection
could be made because the target machine actively refused it.
```

有时我们会遇到这种情况：需要连接的TCP服务开放着，但由于网络或者防火墙的原因，导致始终无法连接成功。这时我们需要设置超时时间来避免程序一直阻塞运行，设置超时可以使用DialTimeout函数。

```
func DialTimeout(network, address string, timeout time.Duration)
(Conn, error)
```

DialTimeout类似于Dial，但其采用了超时。timeout参数如果必要可包含名称解析。

动手写19.3.3

```

01 package main
02
03 import (
04     "net"
05     "time"
06     "log"
07 )
08
09 func main() {
10     // 设置超时
11     conn, err := net.DialTimeout("tcp", "www.baidu.com:81", time.Second * 3)
12     if err != nil {
13         log.Fatal("连接失败!", err)
14     }
15     defer conn.Close()
16     log.Println("连接成功!")
17 }
```

动手写19.3.3对百度服务器的81端口进行连接，设置了3秒的超时，3秒内没有连接成功就会抛出超时异常，如下所示：

```
2019/06/01 20:38:03 连接失败! dial tcp 182.61.200.6:81: i/o timeout
```

建立TCP连接后，就会进行数据的传输。数据传输时使用Read和Write函数分别从连接中读取（接收）和写入（发送）数据。

```
Read(b []byte) (n int, err error)
```

Read从连接中读取数据，Read方法可能会在超过某个固定时间限制后超时返回错误，该错误的Timeout()方法返回真。

```
Write(b []byte) (n int, err error)
```

Write从连接中写入数据，Write方法可能会在超过某个固定时间限制后超时返回错误，该错误

的Timeout()方法返回真。

动手写19.3.4

```
01 package main
02
03 import (
04     "net"
05     "log"
06 )
07
08 func main() {
09     // ncat -l kp 1234
10     conn, err := net.Dial("tcp", ":1234")
11     if err != nil {
12         log.Fatal("连接失败! ", err)
13     }
14     defer conn.Close()
15     log.Println("连接成功! ")
16     // 发送数据
17     conn.Write([]byte("test\n"))
18     // 接收数据
19     var buf = make([]byte, 10)
20     conn.Read(buf)
21     log.Println(buf)
22 }
```

动手写19.3.4连接了本地的1234端口进行收发数据，在运行该程序前，我们需要使用nc工具创建一个TCP服务端，创建后再运行动手写19.3.4，就可以看到ncat收到了程序发送的信息。

```
> ncat -l kp 1234
test
```

前面也提到了HTTP协议是基于TCP的Socket协议实现的，因此我们可以使用TCP客户端来请求百度的HTTP服务。

动手写19.3.5

```
01 package main
02
```

```
03 import (
04     "net"
05     "log"
06 )
07
08 func main() {
09     // 尝试连接百度服务器
10     conn, err := net.Dial("tcp", "www.baidu.com:80")
11     if err != nil {
12         log.Fatal("连接失败! ", err)
13     }
14     defer conn.Close()
15     log.Println("连接成功!")
16     // 发送HTTP形式的内容
17     conn.Write([]byte("GET / HTTP/1.1\r\nHost: www.baidu.com\r\
18             nUser-Agent: curl/7.55.1\r\nAccept: */*\r\n\r\n"))
19
20     var buf = make([]byte, 1024)
21     conn.Read(buf)
22     log.Println(string(buf))
```

动手写 19.3.5 连接百度服务器的 80 端口，并向 80 端口发送了 HTTP 请求包，模拟了一次 HTTP 请求，百度服务器接收到并成功解析该请求后，就会做出响应，程序运行结果如下：

```
2019/06/01 20:39:12 连接成功!
2019/06/01 20:39:12 HTTP/1.1 200 OK
Accept-Ranges: bytes
.....
Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/
<!DOCTYPE html>
<!--STATUS OK--><html>
.....
```

19.3.3 TCP服务端

创建一个TCP服务端只需使用Listen函数即可。

```
func Listen(net, laddr string) (Listener, error)
```

Listen函数返回在一个本地网络地址laddr上监听的Listener。网络类型参数net必须是面向流的网络：tcp、tcp4、tcp6、unix或unixpacket。

可见，Listen是无法创建一个UDP服务器的，因为UDP是面向数据报式的网络（无连接）。

动手写19.3.6

```
01 package main
02
03 import (
04     "net"
05     "log"
06 )
07
08 func main() {
09     // 监听8080端口
10     l, err := net.Listen("tcp", ":8080")
11     if err != nil {
12         log.Fatal("服务启动失败!", err)
13     }
14     defer l.Close()
15     log.Println("服务启动成功!")
16 }
```

动手写19.3.6在本地8080端口创建了一个TCP服务，程序运行后输出如下：

```
2019/06/01 20:40:40 服务启动成功!
```

在创建了一个TCP服务器后，就需要使用Accept函数获取和处理来自客户端的连接。

```
func (l *TCPListener) Accept() (Conn, error)
```

Accept用于实现Listener接口的Accept方法，它会等待下一个呼叫，并返回一个该呼叫的Conn接口。

```
func (c *TCPConn) SetDeadline(t time.Time) error
```

SetDeadline设置读写操作期限，设定该连接的读写deadline，等价于同时调用SetReadDeadline和SetWriteDeadline。deadline是一个绝对时间，超过该时间后，I/O操作就会直接因超时失败返回而不会阻塞。deadline对之后的所有I/O操作都起效，而不仅仅是下一次的读或写操作。参数t为零值时表示不设置期限。

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline设置读操作期限，设定该连接的读操作deadline，参数t为零值时表示不设置期限。

```
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline设置写操作期限，设定该连接的写操作deadline，参数t为零值时表示不设置期限。即使写入超时，返回值n也可能>0，说明成功写入了部分数据。

动手写19.3.7

```

01 package main
02
03 import (
04     "net"
05     "time"
06     "log"
07 )
08
09 func main() {
10     // 监听8080端口
11     l, err := net.Listen("tcp", ":8080")
12     if err != nil {
13         log.Fatal("服务启动失败! ", err)
14     }
15     defer l.Close()
16     log.Println("服务启动成功!")
17     // 阻塞等待用户连接
18     c, err := l.Accept()
19     // 设置连接超时时间
20     c.SetDeadline(time.Now().Add(time.Second))
21     // 设置读取超时时间

```

```
22     c.SetReadDeadline(time.Now().Add(time.Second))  
23     // 设置写入超时时间  
24     c.SetWriteDeadline(time.Now().Add(time.Second))  
25 }
```

运行结果如下：

```
2019/06/01 20:41:00 服务启动成功!
```

对于服务端而言，接收和发送数据也是使用Read和Write函数。

动手写19.3.8

```
01 package main  
02  
03 import (  
04     "net"  
05     "log"  
06 )  
07  
08 func checkErr(err error){  
09     if err != nil {  
10         log.Fatal(err)  
11     }  
12 }  
13  
14 func main() {  
15     // 监听8080端口  
16     l, err := net.Listen("tcp", ":8080")  
17     checkErr(err)  
18     defer l.Close()  
19     log.Println("服务启动成功! ")  
20     // 阻塞等待用户连接  
21     c, err := l.Accept()  
22     checkErr(err)  
23  
24     defer c.Close()  
25     // 读取打印接收的信息  
26     var buf = make([]byte, 10)
```

```

27     log.Println("start to read from conn")
28     n, err := c.Read(buf)
29     checkErr(err)
30     log.Println("接收字节数: " ,n , "接收字内容为: " ,string(buf))
31 }
```

动手写19.3.8在本地8080端口创建了一个TCP服务器，运行后，可以使用nc工具作为TCP客户端进行连接，连接后可以发送数据等待服务端接收。

```
> nc 127.0.0.1 8080
Hello (发送)
```

服务端接收数据后，输出如下：

```

2019/06/01 20:41:38 服务启动成功!
2019/06/01 20:42:03 start to read from conn
2019/06/01 20:42:09 接收字节数: 6 接收字内容为: Hello
```

动手写19.3.8还不能为实际应用提供服务，它在运行一次后就会终止，这时我们需要使用for循环持续不断地接收客户端的请求，并且可以给每一个请求使用新的协程来处理，提高并发能力。

动手写19.3.9

```

01 package main
02
03 import (
04     "net"
05     "log"
06 )
07
08 func checkErr(err error) {
09     if err != nil {
10         log.Fatal(err)
11     }
12 }
13
14 func handleConn(c net.Conn) {
15     defer c.Close()
```

```
16     for {
17         var buf = make([]byte, 10)
18         log.Println("start to read from conn")
19         n, err := c.Read(buf)
20         checkErr(err)
21         log.Printf("read %d bytes, content is %s\n", n,
22                     string(buf[:n]))
22     }
23 }
24
25 func main() {
26     // 监听8080端口
27     l, err := net.Listen("tcp", ":8080")
28     checkErr(err)
29     defer l.Close()
30     log.Println("服务启动成功!")
31
32     for {
33         c, err := l.Accept()
34         checkErr(err)
35         // 开启新协程处理连接
36         go handleConn(c)
37     }
38 }
```

动手写19.3.9展示了一个较为全面的TCP服务端的实现，这个服务端能以较高的性能持续处理每一个连接请求。处理连接的函数为handleConn，它实现了对每一个请求都接收并打印出客户端传来的数据，可以使用nc模拟客户端发送请求。

```
> nc 127.0.0.1 8080
Hello
test
```

发送后，服务端接收并打印出发送的信息：

```
2019/06/01 20:42:37 服务启动成功!
2019/06/01 20:42:45 start to read from conn
2019/06/01 20:42:51 read 6 bytes, content is Hello

2019/06/01 20:42:51 start to read from conn
2019/06/01 20:43:06 read 5 bytes, content is test
```

19.4

UDP编程

19.4.1 UDP简介

UDP是User Datagram Protocol的缩写，中文名是用户数据报协议。它是OSI参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

UDP协议在网络中与TCP协议一样用于处理数据包，是一种无连接的协议。在OSI模型中位于第四层——传输层，处于IP协议的上一层。UDP有不提供数据包分组、组装和不能对数据包进行排序的缺点，也就是说，当报文发送之后，是无法得知其是否安全完整到达的。UDP用来支持那些需要在计算机之间传输数据的网络应用，包括网络视频会议系统在内的众多的客户/服务器模式的网络应用。UDP协议从问世至今已经被使用了很多年，虽然其最初的光彩已经被一些类似协议所掩盖，但是即使是在今天，UDP仍然不失为一项非常实用和可行的网络传输层协议。

19.4.2 UDP客户端

与TCP客户端类似，创建一个UDP客户端同样使用Dial函数，只需在参数中声明发起的请求协议为UDP即可。

```
net.Dial("udp", ":1234")
```

动手写19.4.1

```
01 package main
02
03 import (
04     "net"
05     "log"
06 )
07
```

```
08 func main() {
09     // 尝试连接本地1234端口
10     conn, err := net.Dial("udp", ":1234")
11     if err != nil {
12         log.Fatal("连接失败! ", err)
13     }
14     defer conn.Close()
15     log.Println("连接成功!")
16 }
```

动手写19.4.1尝试连接本地UDP的1234端口，该端口是关闭的，打印输出的结果如下：

```
2019/06/01 20:43:46 连接成功!
```

由于UDP是无连接的协议，只关心信息是否成功发送，不关心对方是否成功接收，只要消息报文发送成功，就不会报错，因此会输出连接成功的信息。

创建一个UDP客户端还可以使用DialUDP函数，该函数会用到ResolveUDPAddr函数来获取一个UDP的地址。

```
func ResolveUDPAddr(net, addr string) (*UDPAddr, error)
```

ResolveUDPAddr将addr作为UDP地址解析并返回。参数addr格式为“host:port”或“[ipv6-host%zone]:port”，解析得到网络名和端口名；net必须是“udp”“udp4”或“udp6”。IPv6地址字面值/名称必须用方括号包起来，如“[:1]:80” “[ipv6-host]:http”或 “[ipv6-host%zone]:80”。

```
func DialUDP(net string, laddr, raddr *UDPAddr) (*UDPConn, error)
```

DialUDP在网络协议net上连接本地地址laddr和远端地址raddr。net必须是“udp”“udp4”“udp6”；如果laddr不是nil，将使用它作为本地地址，否则自动选择一个本地地址。

动手写19.4.2

```
01 package main
02
03 import (
04     "net"
05     "log"
06 )
07
08 func main() {
```

```
09     // 创建一个UDP地址
10    udpaddr, err := net.ResolveUDPAddr("udp4", "127.0.0.1:8080")
11    // 建立连接
12    conn, err := net.DialUDP("udp", nil, udpaddr)
13    if err != nil {
14        log.Fatal("连接失败! ", err)
15    }
16    defer conn.Close()
17    log.Println("连接成功!")
18 }
```

动手写19.4.2使用了DialUDP的方式创建了一个UDP客户端，运行结果如下：

```
2019/06/01 20:44:04 连接成功!
```

UDP的接收和发送操作与TCP相似，使用Read函数进行接收，使用Write函数进行发送。

动手写19.4.3

```
01 package main
02
03 import (
04     "net"
05     "log"
06 )
07
08 func checkErr(err error) {
09     if err != nil {
10         log.Fatal(err)
11     }
12 }
13
14 func main() {
15     // 创建一个UDP地址
16     udpaddr, err := net.ResolveUDPAddr("udp4", "127.0.0.1:1234")
17     checkErr(err)
18     // 建立连接
```

```
19     conn, err := net.DialUDP("udp", nil, udpaddr)
20     checkErr(err)
21     defer conn.Close()
22     log.Println("连接成功! ")
23     // 发送数据
24     conn.Write([]byte("Hello\r\n"))
25     // 接收数据
26     var buf = make([]byte, 1024)
27     conn.Read(buf)
28 }
```

动手写19.4.3创建了一个UDP的客户端，创建UDP连接后就会发送“Hello”，可以使用nc模拟UDP服务端，接收到“Hello”后，nc可以给服务端发送“Hi”，运行结果如下：

```
> nc -lup 1234
Hello
Hi
```

服务端打印日志如下：

```
2019/06/01 20:46:58 连接成功!
2019/06/01 20:47:01 接收: [72 105 10 0 0 0 0 0 0 ...]
```

19.4.3 UDP服务端

与TCP服务端不同，创建一个UDP服务端无法使用有连接的Listen函数，而要使用无连接的ListenUDP函数。

```
func ListenUDP(net string, laddr *UDPAddr) (*UDPConn, error)
```

ListenUDP创建一个接收目的地是本地地址laddr的UDP数据包的网络连接。net必须是“udp”“udp4”“udp6”；如果laddr端口为0，函数将选择一个当前可用的端口，可以用Listener的Addr方法获得该端口。返回的*UDPConn的ReadFrom和WriteTo方法可以用来发送和接收UDP数据包（每个包都可获得来源地址或设置目标地址）。

动手写19.4.4

```
01 package main
02
```

```

03 import (
04     "net"
05     "log"
06 )
07
08 func checkErr(err error) {
09     if err != nil {
10         log.Fatal(err)
11     }
12 }
13 func main() {
14     // 创建一个UDP地址
15     udpaddr, err := net.ResolveUDPAddr("udp4", "127.0.0.1:1234")
16     checkErr(err)
17     // 创建UDP服务
18     conn, err := net.ListenUDP("udp", udpaddr)
19     checkErr(err)
20     defer conn.Close()
21     log.Println("UDP服务创建成功!")
22 }
23 }
```

动手写19.4.4使用ListenUDP创建了一个UDP服务器，运行结果如下：

2019/06/01 20:52:13 UDP服务创建成功!

对于UDP服务端的接收操作还是使用Read函数，发送操作则是使用WriteToUDP函数，其中addr表示目标客户端的地址，可以使用ReadFromUDP函数获取。

```
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int, error)
```

WriteToUDP通过c向地址addr发送一个数据包，b为包的有效负载，返回写入的字节。WriteToUDP方法会在超过一个固定的时间点之后超时，并返回一个错误。在面向数据包的连接上，写入超时是十分罕见的。

```
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err error)
```

ReadFromUDP从c读取一个UDP数据包，将有效负载拷贝到b，返回拷贝字节数和数据包来源地址。ReadFromUDP方法会在超过一个固定的时间点之后超时，并返回一个错误。

动手写19.4.5

```
01 package main
02
03 import (
04     "net"
05     "log"
06 )
07
08 func checkErr(err error) {
09     if err != nil {
10         log.Fatal(err)
11     }
12 }
13
14 func main() {
15     // 创建一个UDP地址
16     udpaddr, err := net.ResolveUDPAddr("udp4", "127.0.0.1:1234")
17     checkErr(err)
18     // 创建UDP服务
19     conn, err := net.ListenUDP("udp", udpaddr)
20     checkErr(err)
21     defer conn.Close()
22     log.Println("UDP服务创建成功! ")
23
24     var buf = make([]byte, 1024)
25     conn.Read(buf)
26     log.Println(string(buf))
27
28     _, raddr, err := conn.ReadFromUDP(buf)
29
30     conn.Write([]byte("Hello Write\r\n"))
31     conn.WriteToUDP([]byte("Hello WriteToUDP\r\n"), raddr)
32 }
```

动手写19.4.5创建了一个UDP服务端，运行后，使用nc工具连接，结果如下：

```
>ncat -u 127.0.0.1 1234  
hi (发送)
```

```
Hello WriteToUDP (接收)
```

服务端日志输出如下：

```
2019/06/02 15:27:30 UDP服务创建成功!  
2019/06/02 15:27:34 hi
```

基于UDP的协议有很多，如DNS域名解析服务、NTP网络时间协议等。我们来模拟一个最简单的NTP服务器，每当接收到任意字节的信息，就将当前的时间发送给客户端。

动手写19.4.6

```
01 package main  
02  
03 import (  
04     "net"  
05     "log"  
06     "time"  
07 )  
08  
09 func checkErr(err error){  
10     if err != nil {  
11         log.Fatal(err)  
12     }  
13 }  
14  
15 func handleClient(conn *net.UDPConn) {  
16     data := make([]byte, 1024)  
17     n, remoteAddr, err := conn.ReadFromUDP(data)  
18     checkErr(err)  
19     log.Println(n, remoteAddr)  
20     b := make([]byte, 1024)  
21     b = []byte(string(time.Now().String()))  
22     conn.WriteToUDP(b, remoteAddr)
```

```
23 }
24
25 func main() {
26     // 创建一个UDP地址
27     udpaddr, err := net.ResolveUDPAddr("udp4", "127.0.0.1:1234")
28     checkErr(err)
29     // 创建UDP服务
30     conn, err := net.ListenUDP("udp", udpaddr)
31     checkErr(err)
32     defer conn.Close()
33     log.Println("UDP服务创建成功! ")
34     for {
35         handleClient(conn)
36     }
37 }
```

动手写19.4.6模拟了一个最简单的NTP服务，每当接收到客户端发来的请求，不管内容是什么都会往回发送当前的时间信息。这个例子也是一个较为完整的UDP服务端实现。运行后使用nc模拟客户端，直接按下回车键就能接收到当前的时间信息。

```
>ncat -u 127.0.0.1 1234
2019-06-02 15:57:49.709676 +0800 CST m=+5.219016501
2019-06-02 15:57:50.844026 +0800 CST m=+6.353366501
```

服务端输出日志如下：

```
2019/06/02 15:57:44 UDP服务创建成功!
2019/06/02 15:57:49 1 127.0.0.1:57093
2019/06/02 15:57:50 1 127.0.0.1:57093
```

19.5 小结

- ◇ 了解常见的网络架构。
- ◇ 了解OSI七层网络模型与TCP/IP模型的差异。
- ◇ 掌握Socket的原理。

- ◇ 能够熟练使用TCP编写客户端和服务端。
- ◇ 能够熟练使用UDP编写客户端和服务端。

19.6 知识拓展

前面的章节详细介绍了如何使用TCP和UDP进行Socket编程。本节我们将会使用之前所学的内容来实现一个端口扫描器，主要针对TCP的端口扫描，通过命令行的方式指定端口，扫描获取端口是否开放。

动手写19.6.1

```
01 package main
02
03 import (
04     "flag"
05     "net"
06     "time"
07     "log"
08 )
09
10 func scan(ip string, port string) {
11     conn, err := net.DialTimeout("tcp", ip+":"+port, time.Second*2)
12     if err != nil {
13         log.Fatal("端口未开放！")
14     }
15     defer conn.Close()
16     log.Println("端口开放！")
17 }
18
19 func main() {
20     ip := flag.String("h", "127.0.0.1", "指定主机IP")
21     port := flag.String("p", "8080", "指定扫描的端口")
22     flag.Parse()
23 }
```

```
24     log.Println("扫描的端口为", *ip, *port)
25
26     // 扫描单个端口
27     scan(*ip, *port)
28 }
```

动手写19.6.1实现了一个最简单的端口扫描器，指定IP和端口就可以进行端口开放性检测。使用flag包解析命令行参数，-h可以打印出参数含义。

```
> go run 19.5.1.go -h
flag needs an argument: -h
Usage of 19.5.1.exe:
-h string
指定主机IP (default "127.0.0.1")
-p string
指定扫描的端口 (default "8080")
```

尝试使用动手写19.6.1检测本机的445端口，检测结果如下：

```
> go run 19.5.1.go -h 127.0.0.1 -p 445
2019/06/01 20:57:18 扫描的端口为 127.0.0.1 445
2019/06/01 20:57:18 端口开放!
```

动手写19.6.1实现了简单的端口开放性探测，但它存在一个问题，就是不能批量端口探测。我们再来实现一个可以指定如“80,443,1000–2000”这种格式的批量端口扫描器。

动手写19.6.2

```
01 package main
02
03 import (
04     "flag"
05     "log"
06     "strings"
07     "strconv"
08     "sync"
09     "net"
10     "time"
11 )
```

```
12
13 func processPortItem(port string) []string {
14     var ports []string
15     arr := strings.Split(port, ",")
16     for _, p := range arr {
17         if strings.Contains(p, "-") {
18             ports = append(ports, rangeToArr(p)...)
19         } else {
20             ports = append(ports, p)
21         }
22     }
23     return ports
24 }
25
26 //convert "1-3" to ["1", "2", "3"]
27 func rangeToArr(s string) []string {
28     if strings.Contains(s, "-") {
29         var arr []string
30         from,_ := strconv.Atoi(strings.Split(s, "-")[0])
31         to,_ := strconv.Atoi(strings.Split(s, "-")[1])
32         if from == 0 { from = 1 }
33         if to == 0 { to = 65535 }
34         for i := from; i <= to; i++ {
35             arr = append(arr, strconv.Itoa(i))
36         }
37         return arr
38     } else {
39         return []string{s}
40     }
41 }
42
43 func scan(ip string, port string, wg *sync.WaitGroup) {
44     conn, err := net.DialTimeout("tcp", ip + ":" + port, time.Second)
45     if err != nil {
```

```
46             wg.Done()
47         return
48     }
49     wg.Done()
50     defer conn.Close()
51     log.Println(ip, port, "端口开放!")
52 }
53
54 func main() {
55     ip := flag.String("h", "127.0.0.1", "指定主机IP")
56     port := flag.String("p", "1-1000", "指定扫描的端口")
57     flag.Parse()
58     log.Println("扫描的端口为", *ip, *port)
59
60     // 线程同步
61     wg := &sync.WaitGroup{}
62
63     for _, p := range processPortItem(*port) {
64         wg.Add(1)
65         go scan(*ip, p, wg)
66     }
67     wg.Wait()
68 }
```

动手写19.6.2使用了WaitGroup，WaitGroup在Go语言中用于线程同步，单从字面意思上理解，“wait”是“等待”的意思，“group”是“组、团队”的意思，“WaitGroup”就是指等待一组或一个系列执行完成后才会继续向下执行。简单来说就是避免Go语言的主程序的执行完毕导致其他还未执行完毕的协程终止。

动手写19.6.2使用processPortItem方法来解析传入的端口参数，如将“1-2,3”端口解析为数组“["1","2","3"]”，这样就可以实现批量化的端口扫描，运行示例如下：

```
> go run 19.5.2.go -h 127.0.0.1 -p 1-1000,8080-9000
2019/06/01 20:59:29 扫描的端口为 127.0.0.1 1-1000,8080-9000
2019/06/01 20:59:30 127.0.0.1 445 端口开放!
2019/06/01 20:59:30 127.0.0.1 135 端口开放!
```

>> 第 20 章

数据库编程

<<

在前面的章节中我们介绍了如何使用网络编程来传输数据，而如何存储和管理这些传输的数据便是本章所要介绍的内容。

数据库（database）是按照数据结构来组织、存储和管理数据的仓库。相对于其他存储方式，存储只是数据库的其中一个功能，数据的组织和管理才是数据库的核心。

简单地说，数据库就是一个存放数据的仓库，这个仓库是按照一定的数据结构来组织和存储的，我们可以通过数据库提供的多种方法来管理数据库里的数据。相较于数据库，文件保存数据有以下几个缺点：

- ◇ 文件有安全性问题。
- ◇ 文件不利于对数据的查询和管理。
- ◇ 文件不便于存放海量数据。
- ◇ 文件在程序中不便于控制。

使用数据库可以带来许多好处，如减少数据的冗余度，从而大大节省了数据的存储空间；实现数据资源的充分共享；等等。此外，数据库技术还为用户提供了非常简便的使用手段，使用户易于编写有关数据库应用程序。我们可以通过结构化查询语言，即SQL语句，去操纵更改数据。

数据库的使用水平是衡量一个程序员能力的重要指标。

20.1 MySQL简介

MySQL是一个关系型数据库管理系统，由瑞典MySQL AB公司开发，目前属于甲骨文公司（Oracle）旗下产品。MySQL是最流行的关系型数据库管理系统之一，而在Web应用方面，MySQL

是最好的RDBMS (Relational Database Management System, 关系数据库管理系统) 应用软件之一。

MySQL所使用的SQL语言是用于访问数据库的最常用标准化语言。MySQL软件采用了双授权政策，分为社区版和商业版，由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，一般中小型网站的开发都选择MySQL作为网站数据库。

我们可以使用MariaDB代替MySQL，MariaDB数据库管理系统是MySQL的一个分支，主要由开源社区在维护，采用GPL授权许可。开发这个分支的原因之一是Oracle收购了MySQL后，有将MySQL闭源的潜在风险，因此社区采用分支的方式来避开这个风险。

MariaDB的目的是完全兼容MySQL，包括API和命令行，使之能轻松成为MySQL的替代品。MariaDB由MySQL的创始人迈克尔·维德纽斯 (Michael Widenius) 主导开发，他早前曾以10亿美元的价格，将自己创建的公司MySQL AB卖给了SUN。此后，随着SUN被Oracle收购，MySQL的所有权也落入Oracle的手中。MariaDB的名称来源于迈克尔·维德纽斯的女儿玛丽亚 (Maria) 的名字。

20.1.1 安装MySQL

MySQL有非常多的分支版本，官方版本（目前由Oracle维护）的下载地址为：<https://www.mysql.com/downloads/>。

在CentOS上可以使用如下命令安装：

```
wget http://repo.mysql.com/mysql-community-release-el7-5.noarch.rpm
rpm -ivh mysql-community-release-el7-5.noarch.rpm
yum update
yum install mysql-server
```

MariaDB的下载地址为：<https://mariadb.org/download/>。

在CentOS上可以使用如下命令安装：

```
yum install mariadb-server mariadb
```

20.1.2 MySQL常见命令

安装完MySQL后，可以使用如下格式连接MySQL服务器，需要注意的是-p与密码之间不能有空格。

```
> mysql -h主机地址 -u用户名 -p用户密码
```

1. 连接到本机上的MySQL。

首先打开cmd窗口，输入mysql -uroot -p，然后输入密码，进入MySQL控制台，MySQL的提示符是：mysql>。

```
> mysql -uroot -proot
```

2. 连接到远程主机上的MySQL。

假设远程主机的IP为：192.168.1.1，用户名为root，密码为root。

```
> mysql -h192.168.1.1 -uroot -proot
```

3. 退出MySQL命令：exit或quit（回车）。

```
mysql> exit/quit;
```

登录进MySQL服务器后，我们来创建一个数据库：

```
01 mysql> create database 01kuaixue;
02 Query OK, 1 row affected (0.00 sec)
03
04 mysql> show databases;
05 +-----+
06 Database           |
07 +-----+
08 | information_schema   |
09 | 01kuaixue          |
10 | mysql               |
11 | performance_schema |
12 | test                |
13 +-----+
14 5 rows in set (0.00 sec)
```

20.2 database/sql

Go操作数据库，是通过database/sql包以及第三方的实现了database/sql/driver接口的数据库驱动包来共同完成的。

其中database/sql/driver中的接口Conn和Stmt，官方交给第三方实现驱动，并且是协程不安全的。官方实现的database/sql包中的DB和Stmt是协程安全的，因为内部实现是连接池。

Go语言和其他语言不同的地方是，Go官方没有提供数据库驱动，而是编写了开发数据库驱动的标准接口，开发者可以根据定义的接口来开发相应的数据库驱动。这样做的好处在于，只要

是按照标准接口开发的代码，以后迁移数据库时，不需要做任何修改，极大方便了后期的架构调整。

在每一个由第三方开发者编写的数据库驱动中，都会实现一个init函数，在init函数内会调用一个叫Register的方法来完成数据库驱动的注册。

```
func Register(name string, driver driver.Driver)
```

Register注册并命名一个数据库，可以在Open函数中使用该命名启用该驱动。如果Register注册同一名称两次，或者driver参数为nil，会导致panic。

接着，我们来看MySQL的init函数是怎么写的。在init函数中，就只有一个调用sql.Register注册MySQL驱动的语句。

```
func init() {
    sql.Register("mysql", &MySQLDriver{})
}
```

在之前的章节就介绍过，包在引入的过程中会自动调用包中的init函数，因此，我们在注册数据库驱动时只需要使用匿名导入的方式引用该包即可，这样我们的代码就可以直接使用这个数据库驱动了。

```
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

20.3 数据库基本操作

数据库的最基本操作便是增（create）、查（read）、改（update）、删（delete），简称CRUD。数据库中有八类对象，分别是数据库、数据表、记录、字段、索引、查询、过滤器、视图，每个对象的含义参见“20.5知识拓展”部分，我们可以对这八类对象进行CRUD操作。本节会详细介绍如何使用Go语言对记录进行增查改删操作。

20.3.1 连接数据库

Go语言中，sql包提供了一个Open方法来创建一个数据库连接。

```
func Open(driverName, dataSourceName string) (*DB, error)
```

Open打开一个dirverName指定的数据库，dataSourceName指定数据源，一般至少包括数据库文件名和（可能的）连接信息。

大多数用户会通过数据库特定的连接帮助函数打开数据库，返回一个*DB。返回的DB可以安全地被多个go程同时使用，并且会维护自身的闲置连接池。这样一来，Open函数只需调用一次。很少需要关闭DB。

Open函数只是验证其参数，而不创建与数据库的连接。如果要检查数据源的名称是否合法，应调用返回值的Ping方法。

```
func (db *DB) Ping() error
```

Ping检查与数据库的连接是否仍有效，如果需要会创建连接。

动手写20.3.1

```
01 package main
02
03 import (
04     "database/sql"
05     _ "github.com/go-sql-driver/mysql"
06     "log"
07 )
08
09
10 func main() {
11     db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)/
12         01kuaixue")
13     if err != nil {
14         log.Fatal(err)
15     }
16     defer db.Close()
17
18     // 验证连接的可用性
19     err = db.Ping()
20     if err != nil {
21         log.Fatal("数据库连接失败: ", err)
22     }
23     log.Println("数据库连接成功! ")
```

动手写20.3.1创建了一个MySQL的数据库连接，数据库的连接字符串表示MySQL服务开启本地127.0.0.1的3306端口，连接的用户名和密码都是root，连接的数据库为01kuaixue。之后使用Ping方法检查数据库连接是否正常，如果MySQL服务关闭，就会连接失败。

```
2019/06/05 19:28:16 数据库连接失败: dial tcp 127.0.0.1:3306: connectex:  
No connection could be made because the target machine actively  
refused it.
```

连接成功就会提示：

```
2019/06/05 19:30:39 数据库连接成功!
```

20.3.2 创建数据表

创建MySQL数据表需要定义表名、表字段名、字段类型及约束。创建数据表语法结构为：

```
CREATE TABLE 表名 (  
    字段名1 数据类型 [列级别约束条件] [默认值],  
    字段名2 数据类型 [列级别约束条件] [默认值],  
    字段名3 数据类型 [列级别约束条件] [默认值],  
    ... [表级别约束条件]  
) ;
```

我们来创建一张数据表：

```
01 CREATE TABLE `01kuaixue`.`user` (  
02     `uid` INT(10) NOT NULL AUTO_INCREMENT,  
03     `username` VARCHAR(64) NULL DEFAULT 1,  
04     `gender` TINYINT(1) NULL DEFAULT NULL,  
05     `password` VARCHAR(64) NULL DEFAULT NULL,  
06     `created` DATE NULL DEFAULT NULL,  
07     PRIMARY KEY (`uid`)  
08 );
```

如果你不想字段为NULL，可以设置字段的属性为NOT NULL，这样在操作数据库时，如果输入该字段的数据为NULL就会报错。

AUTO_INCREMENT定义列为自增的属性，一般用于主键，数值会自动加1。

PRIMARY KEY关键字用于定义列为主键。你可以使用多列来定义主键，列间以逗号分隔。

DEFAULT为设置默认值关键字，设置后每插入一条数据，如果插入数据的该字段没有赋值，

则将其设置为默认值。

我们在命令行中运行以上SQL语句，就会创建一张user表，如下所示：

```
mysql> use 01kuaixue;
Database changed
mysql> CREATE TABLE `user` (
    ->   `uid` INT(10) NOT NULL AUTO_INCREMENT,
    ->   `username` VARCHAR(64) NULL DEFAULT 1,
    ->   `gender` TINYINT(1) NULL DEFAULT NULL,
    ->   `password` VARCHAR(64) NULL DEFAULT NULL,
    ->   `created` DATE NULL DEFAULT NULL,
    ->   PRIMARY KEY (`uid`)
    -> );
Query OK, 0 rows affected (0.01 sec)
```

提示

使用箭头标记“->”不是SQL语句的一部分，它仅仅表示一个新行。如果一条SQL语句太长，我们可以通过回车键创建一个新行来编写SQL语句。SQL语句的命令结束符为分号“;”。

使用Go语言创建数据表需要使用Exec函数。

```
func (db *DB) Exec(query string, args ...interface{}) (Result, error)
```

Exec执行一次命令（包括查询、删除、更新、插入等），不返回任何执行结果。参数args表示query中的占位参数。

Exec的返回值为Result接口，Result的定义如下：

```
type Result interface {
    LastInsertId() (int64, error)
    RowsAffected() (int64, error)
}
```

Result主要有两个方法。LastInsertId返回一个数据库生成的回应命令的整数，当插入新行时，返回由数据库执行插入操作得到的自增ID号。RowsAffected返回被update、insert或delete命令影响的行数。

动手写20.3.2

```
01 package main
02
03 import (
04     "database/sql"
05     _ "github.com/go-sql-driver/mysql"
06     "log"
07 )
08
09 func main() {
10     db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306) /
11     01kuaixue")
12     if err != nil {
13         log.Fatal(err)
14     }
15     defer db.Close()
16     // 验证连接的可用性
17     err = db.Ping()
18     if err != nil {
19         log.Fatal("数据库连接失败: ", err)
20     }
21     log.Println("数据库连接成功! ")
22     // 创建一张数据表
23     createTable := "CREATE TABLE `01kuaixue`.`user` (" +
24         "`uid` INT(10) NOT NULL AUTO_INCREMENT," +
25         "`username` VARCHAR(64) NULL DEFAULT 1," +
26         "`gender` TINYINT(1) NULL DEFAULT NULL," +
27         "`password` VARCHAR(64) NULL DEFAULT NULL," +
28         "`created` DATE NULL DEFAULT NULL," +
29         "PRIMARY KEY (`uid`)" +
30         ");"
31     _, err = db.Exec(createTable)
32     if err != nil {
33         log.Fatal("数据表创建失败! ", err)
34     }
35 }
```

动手写20.3.2如果创建表成功，会打印如下信息：

```
2019/06/05 21:27:39 数据库连接成功!
```

```
2019/06/05 21:27:40 数据表创建成功!
```

20.3.3 插入数据

MySQL中使用INSERT INTO语句来插入数据，插入的语法结构为：

```
INSERT INTO table_name ( field1, field2,...fieldN )
VALUES ( value1, value2,...valueN );
```

如果需要同时插入多条数据，可以使用如下方式：

```
INSERT INTO table_name (field1, field2,...fieldN)
VALUES (valueA1,valueA2,...valueAN), (valueB1,valueB2,...value
BN) .....
```

SQL语句中如果数据是字符型，必须使用单引号或者双引号，如："value"。

```
INSERT INTO `user` (username,gender,password,created) VALUES ("Ail
sa",0,"123456","1992-01-01")
```

添加数据的时候可以指定列进行添加。如果所有的列都要添加数据，可以不指定列进行添加：

```
INSERT INTO `user` VALUES (0,"john",1,"123456","1990-01-01")
```

如果添加过主键自增（PRIMARY KEY AUTO_INCREMENT），第一列在增加数据的时候可以写为“0”或者“null”，这样添加数据可以自增，从而可以添加全部数据，而不用特意指定哪几列添加数据。

动手写20.3.3

```
01 package main
02
03 import (
04     "database/sql"
05     _ "github.com/go-sql-driver/mysql"
06     "log"
07     "time"
08 )
```

```
09
10 func checkErr(err error) {
11     if err != nil {
12         log.Fatal(err)
13     }
14 }
15
16 func main() {
17     db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306) /
18     01kuaixue")
19     checkErr(err)
20     defer db.Close()
21
22     // 验证连接的可用性
23     err = db.Ping()
24     checkErr(err)
25     log.Println("数据库连接成功!")
26
27     rs, err := db.Exec("INSERT INTO `user` (username,gender,password,
28     created) VALUES (?,?,?,?)","john",1,"123456",time.Now())
29     checkErr(err)
30
31     rowCount, err := rs.RowsAffected()
32     checkErr(err)
33     log.Printf("插入了 %d 行", rowCount)
34 }
```

动手写20.3.3使用Exec执行了插入语句，其中“?”表示参数占位符，使用这种方式传入参数可以有效地防止出现SQL注入这类漏洞，运行结果如下：

```
2019/06/05 21:44:45 数据库连接成功!
2019/06/05 21:44:45 插入了 1 行
```

提示

SQL注入（SQLi）是一种注入攻击，可以执行恶意SQL语句。它通过将任意SQL代码插入数据库

查询，使攻击者能够完全控制Web应用程序后面的数据库服务器。攻击者可以使用SQL注入漏洞绕过应用程序安全措施；可以绕过网页或Web应用程序的身份验证和授权，并检索整个SQL数据库的内容；还可以使用SQL注入来添加、修改和删除数据库中的记录。

sql包还提供一种预编译的方式来执行SQL语句，通常在处理批量SQL语句时会用到这种方式，这种方式比手动拼接字符串SQL语句高效，还可以防止SQL注入攻击。

```
func (db *DB) Prepare(query string) (*Stmt, error)
```

Prepare创建一个准备好的状态用于之后的查询和命令。返回值可以同时执行多个查询和命令。

动手写20.3.4

```
01 package main
02
03 import (
04     "database/sql"
05     _ "github.com/go-sql-driver/mysql"
06     "log"
07     "time"
08 )
09
10 func checkErr(err error){
11     if err != nil {
12         log.Fatal(err)
13     }
14 }
15
16 func main() {
17     db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306) /
18     01kuaixue")
19     checkErr(err)
20     defer db.Close()
21
22     // 验证连接的可用性
23     err = db.Ping()
24     checkErr(err)
```

```
24     log.Println("数据库连接成功! ")  
25  
26     // 使用预编译方式，等效于  
27     // rs, err := db.Exec("INSERT INTO `user` (username,gender,  
28     // password,created) VALUES (?, ?, ?, ?)", "Ailsa", 0, "111111", time.  
29     Now())  
30     stmt, err := db.Prepare("INSERT INTO `user` (username,gender,  
31     // password,created) VALUES (?, ?, ?, ?)")  
32     defer stmt.Close()  
33     rs, err := stmt.Exec("Ailsa", 0, "111111", time.Now())  
34     checkErr(err)  
35     rowCount, err := rs.RowsAffected()  
36     checkErr(err)  
37  
38     log.Printf("插入了 %d 行", rowCount)  
39 }
```

动手写20.3.4使用预编译的方式对动手写20.3.3进行了改写，需要注意的是返回的stmt要及时关闭。若是批量操作的话建议使用Prepare的方式，否则直接使用Exec更方便一些，省去了创建stmt的时间。

20.3.4 查询数据

MySQL数据库使用SELECT语句来查询数据。以下为在MySQL数据库中查询数据通用的SELECT语法：

```
SELECT column_name,column_name  
      FROM <表 1>, <表 2>...  
      JOIN<表3>on...  
      [WHERE <表达式>  
      [GROUP BY <group by definition>  
      [HAVING <expression> [{<operator> <expression>}...]]  
      [ORDER BY <order by definition>]  
      [LIMIT [<offset>,] <row count>]
```

语法解释如下：

- ◇ SELECT之后是逗号分隔列或星号 (*) 的列表，表示要返回所有列。
- ◇ FROM指定要查询数据的表或视图。

- ◇ JOIN根据某些连接条件从其他表中获取数据。
- ◇ WHERE过滤结果集中的行。
- ◇ GROUP BY将一组行组合成小分组，并对每个小分组应用聚合函数。
- ◇ HAVING过滤器是基于GROUP BY子句定义的小分组。
- ◇ ORDER BY指定用于排序的列的列表。
- ◇ LIMIT限制返回行的数量。

你可以在查询语句中使用一个或者多个表，表之间使用逗号（,）分隔，并使用WHERE语句来设定查询条件。SELECT命令可以读取一条或者多条记录。你可以使用星号（*）来代替其他字段，SELECT语句会返回表的所有字段数据。

```
mysql> use 01kuaixue;
Database changed
mysql> select * from user where gender=1 limit 0,1;
+----+-----+-----+-----+
| uid | username | gender | password | created |
+----+-----+-----+-----+
| 5   | john     | 1      | 111111  | 2019-06-05 |
+----+-----+-----+-----+
1 row in set (0.00 sec)
```

在平常的编程中，不推荐使用星号（*）来查询数据，它存在如下问题：

1. 使用星号（*）可能会返回不使用的列的数据。它在MySQL数据库服务器和应用程序之间产生不必要的I/O磁盘和网络流量。
2. 如果明确指定列，则结果集更可预测并且更易于管理。而当你使用星号（*）并且有人通过添加更多列来更改表格数据时，将会得到一个与预期不同的结果集。
3. 使用星号（*）可能会将敏感信息暴露给未经授权的用户。

在Go语言中，我们可以使用Query函数来查询数据：

```
func (db *DB) Query(query string, args ...interface{}) (*Rows, error)
```

Query执行一次查询，返回多行结果（即Rows），一般用于执行SELECT命令。参数args表示Query中的占位参数，Rows是查询的结果。它的游标指向结果集的第0行，使用Next方法来遍历各行结果。

查询到数据后使用rows.Next获取一行结果，并使用Scan将查询到的结果赋值到目标变量中。

```
func (r *Row) Scan(dest ...interface{}) error
```

Scan将该行查询结果的各列分别保存进dest参数指定的值中。如果该查询匹配多行，Scan会使用第一行结果并丢弃其余各行。如果没有匹配查询的行，Scan会返回ErrNoRows。

动手写20.3.5

```
01 package main
02
03 import (
04     "database/sql"
05     _ "github.com/go-sql-driver/mysql"
06     "log"
07 )
08
09 type User struct {
10     Uid int
11     Username string
12     Gender bool
13     Password string
14     Created string
15 }
16
17 func checkErr(err error) {
18     if err != nil {
19         log.Fatal(err)
20     }
21 }
22
23 func main() {
24     db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)/01kuaixue")
25     checkErr(err)
26     defer db.Close()
27     // 验证连接的可用性
28     err = db.Ping()
29     checkErr(err)
30     log.Println("数据库连接成功! ")
31
32     rows, err := db.Query("select * from `user` where username=?", "john")
33     defer rows.Close()
```

```

34     for rows.Next() {
35         user := User{}
36         err := rows.Scan(&user.Uid, &user.Username, &user.
37             Gender, &user.Password, &user.Created)
38         checkErr(err)
39         log.Println(user)
40     }
41 }
```

运行结果如下：

```

2019/06/07 13:25:35 数据库连接成功!
2019/06/07 13:25:35 {1 john false 111111 2019-06-02}
```

20.3.5 更改数据

如果需要修改或更新MySQL中的数据，我们可以使用UPDATE命令来操作。

使用UPDATE语句修改单个表，语法格式为：

```

UPDATE <表名>
    SET 字段 1=值 1 [,字段 2=值 2... ]
    [WHERE 子句]
    [ORDER BY 子句]
    [LIMIT 子句]
```

语法解释如下：

- ◇ <表名>：用于指定要更新的表名称。
- ◇ SET：用于指定表中要修改的列名及其列值。其中，每个指定的列值可以是表达式，也可以是该列对应的默认值。如果指定的是默认值，可用关键字DEFAULT表示列值。
- ◇ WHERE：可选。用于限定表中要修改的行。若不指定，则修改表中所有的行。
- ◇ ORDER BY：可选。用于限定表中的行被修改的次序。
- ◇ LIMIT：可选。用于限定被修改的行数。

注意：修改一行数据的多个列值时，SET子句的每个值用逗号分开即可。

动手写20.3.6

```

01 package main
02
```

```
03 import (
04     "database/sql"
05     _ "github.com/go-sql-driver/mysql"
06     "log"
07 )
08
09 func checkErr(err error) {
10     if err != nil {
11         log.Fatal(err)
12     }
13 }
14
15 func main() {
16     db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)
17 /01kuaixue")
18     checkErr(err)
19     defer db.Close()
20
21     // 验证连接的可用性
22     err = db.Ping()
23     checkErr(err)
24     log.Println("数据库连接成功! ")
25
26     rs, err := db.Exec("update `user` set password=? where sname=
27 ?, "123123", "john")
28     checkErr(err)
29
30
31     if rowCount > 0 {
32         log.Println("更新成功! ")
33     }
34
35 }
```

运行结果如下：

```
2019/06/07 13:27:34 数据库连接成功!
```

```
2019/06/07 13:27:34 更新成功!
```

20.3.6 删 除 数据

MySQL 使用 DELETE 语句从单个表中删除数据，语句格式为：

```
DELETE FROM <表名>
[WHERE 子句]
[ORDER BY 子句]
[LIMIT 子句]
```

语法解释如下：

◇ <表名>：指定要删除数据的表名。

◇ WHERE 子句：可选项。表示为删除操作限定删除条件，若省略该子句，则代表删除该表中的所有行。

◇ ORDER BY 子句：可选项。表示删除时，表中各行将按照子句中指定的顺序进行删除。

◇ LIMIT 子句：可选项。用于告知服务器在控制命令被返回到客户端前被删除行的最大值。

注意：在不使用 WHERE 条件的时候，将删除所有数据；数据库一旦删除数据，数据就会永远消失。因此，在执行 DELETE 语句之前，应该先备份数据库，以防需要找回被删除的数据。

动手写 20.3.7

```
01 package main
02
03 import (
04     "database/sql"
05     _ "github.com/go-sql-driver/mysql"
06     "log"
07 )
08
09 func checkErr(err error) {
10     if err != nil {
11         log.Fatal(err)
12     }
13 }
14
```

```
15 func main() {
16     db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)
17 /01kuaixue")
18     checkErr(err)
19     defer db.Close()
20
21     // 验证连接的可用性
22     err = db.Ping()
23     checkErr(err)
24     log.Println("数据库连接成功!")
25
26     rs, err := db.Exec("delete from `user` where uid=?", 3)
27     checkErr(err)
28
29     rowCount, err := rs.RowsAffected()
30     checkErr(err)
31
32     if rowCount > 0 {
33         log.Println("删除成功!")
34     }
35 }
```

动手写20.3.7使用DELETE语句删除了user表中uid为3的数据记录，运行结果如下：

```
2019/06/07 13:27:49 数据库连接成功!
```

```
2019/06/07 13:27:49 删除成功!
```

20.3.7 MySQL事务

MySQL数据库中的事务是用户一系列的数据库操作序列，这些操作要么全做，要么全不做，是一个不可分割的工作单位。

事务具有四个特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。这四个特性简称为ACID原则。

1. 原子性

事务必须是原子工作单元，事务中的操作要么全部执行，要么全都不执行，不能只完成部分

操作。原子性在数据库系统中由恢复机制来实现。

2. 一致性

事务开始之前，数据库处于一致性的状态；事务结束后，数据库必须仍处于一致性状态。数据库一致性的定义是由用户负责的。例如，在银行转账中，用户可以定义转账前后两个账户金额之和保持不变。

3. 隔离性

系统必须保证事务不受其他并发执行事务的影响，即当多个事务同时运行时，各事务之间相互隔离，不可互相干扰。事务查看数据时数据所处的状态，要么是另一个并发事务修改它之前的状态，要么是另一个并发事务修改它之后的状态，事务不会查看中间状态的数据。隔离性通过系统的并发控制机制实现。

4. 持久性

一个已完成的事务对数据所做的任何变动在系统中是永久有效的，即使该事务产生的修改不正确，错误也将一直保持。持久性通过恢复机制实现，发生故障时，可以通过日志等手段恢复数据库信息。

事务的ACID原则保证了一个事务或者成功提交，或者失败回滚，二者必居其一。因此，它对事务的修改具有可恢复性，即当事务失败时，它对数据的修改都会恢复到该事务执行前的状态。

简单来说，事务处理就两个过程，要么成功提交，要么失败回滚，在Go语言中使用Tx结构体来表示事务。

```
type Tx interface{
    Commit() error
    Rollback() error
}
```

Tx代表一个进行中的数据库事务。一次事务必须以对Commit或Rollback的调用结束。调用Commit或Rollback后，所有对事务的操作都会失败并返回错误值ErrTxDone。

动手写20.3.8

```
01 package main
02
03 import (
04     "database/sql"
05     _ "github.com/go-sql-driver/mysql"
06     "log"
```

```
07  )
08
09 func checkErr(err error) {
10     if err != nil {
11         log.Fatal(err)
12     }
13 }
14
15 func checkErrWithTx(err error, tx *sql.Tx) {
16     if err != nil {
17         tx.Rollback()
18         log.Fatal(err)
19     }
20 }
21
22 func main() {
23     db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)
24     /01kuaixue")
25     checkErr(err)
26     defer db.Close()
27
28     // 验证连接的可用性
29     err = db.Ping()
30     checkErr(err)
31     log.Println("数据库连接成功!")
32     var password string
33
34     tx, err := db.Begin()
35     checkErr(err)
36
37     // 查找john的密码，如果密码为123123就将密码改为111111，否则不执行任何
38     // 操作
39     err = tx.QueryRow("select password from `user` where
40     username=?", "john").Scan(&password)
```

```

39     checkErrWithTx(err, tx)
40     if password == "123123" {
41         rs, err := tx.Exec("update `user` set password=? where
42         username=?", "111111", "john")
43         checkErrWithTx(err, tx)
44         rowCount, err := rs.RowsAffected()
45         checkErrWithTx(err, tx)
46         if rowCount > 0 {
47             log.Println("密码更新完成!")
48         }
49     }
50     tx.Commit()
51     log.Println("事务处理完成!")
52 }
```

动手写20.3.8创建了一个事务对象tx，使用tx查询john的密码，如果密码为123123，就将密码改为111111，否则不执行任何操作。例子中还定义了checkErrWithTx函数来对出现错误时进行回滚，运行结果如下：

```

2019/06/07 13:34:14 数据库连接成功!
2019/06/07 13:34:14 密码更新完成!
2019/06/07 13:34:14 事务处理完成!
```

20.4 小结

- ◇ 了解MySQL与MariaDB的关系。
- ◇ 了解Go语言的数据库驱动与其他语言的不同之处。
- ◇ 掌握如何使用Go语言进行数据库的基本操作（增查改删）。
- ◇ 掌握数据库事务的基本原理及使用场景。

20.5 知识拓展

数据库的结构

数据库中存在如下八种对象结构。

1. 数据库 (Database)

所谓“数据库”，是以一定方式存储在一起、能与多个用户共享、具有尽可能小的冗余度、与应用程序彼此独立的数据集合。MySQL中使用的数据库是关系型数据库（Relational Database）。一个数据库由一个或一组数据表组成。每个数据库都以文件的形式存放在磁盘上，即对应于一个物理文件。不同的数据库与物理文件对应的方式也不一样。

2. 数据表 (Table)

数据表简称表，由一组数据记录组成，数据库中的数据是以表为单位进行组织的。一个表是一组相关的按行排列的数据，每个表中都含有相同类型的信息。表实际上是一个二维表格，例如一个班所有学生的考试成绩可以存放在一个表中，表中的每一行对应一个学生，这一行包括学生的学号、姓名及各门课程成绩。

3. 记录 (Record)

表中的每一行称为一个记录，它由若干个字段组成。

4. 字段 (Field)

字段也称域。表中的每一列称为一个字段。每个字段都有相应的描述信息，如数据类型、数据宽度等。

5. 索引 (Index)

为了提高访问数据库的效率，可以对数据库使用索引。在较大的数据库里查找指定的记录时，使用索引和不使用索引的效率有很大差别。索引实际上是一种特殊类型的表，其中含有关键字段的值（由用户定义）和指向实际记录位置的指针，这些值和指针按照特定的顺序（也由用户定义）存储，从而能以较快的速度查找到所需要的数据记录。

6. 查询 (Query)

查询是一条SQL（结构化查询语言）命令，用来从一个或多个表中获取一组指定的记录，或者对某个表执行指定的操作。当从数据库中读取数据时，我们往往希望读出的数据符合某些条件，并且能按某个字段排序。使用SQL可以使这一操作容易实现而且更加有效。SQL是非过程化语言（有人称其为第四代语言），在用它查找指定的记录时，只需指出做什么，不必说明如何做。每个语句可以看作是一个查询，根据这个查询，可以得到需要的查询结果。

7. 过滤器 (Filter)

过滤器是数据库的一个组成部分，它把索引和排序结合起来，用来设置条件，然后根据给定的条件输出所需要的数据。

8. 视图 (View)

数据的视图指的是查找到（或处理）的记录数和显示（或处理）这些记录的顺序。在一般情况下，视图由过滤器和索引控制。

统筹策划：李婧玮

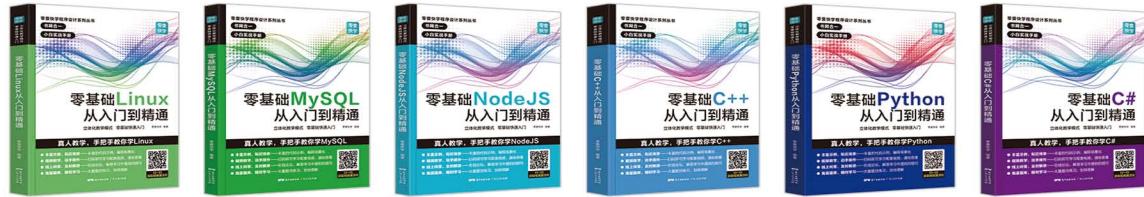
责任编辑：陈泽洪

封面设计：画画鸭工作室

责任技编：吴彦斌

◇— 丛书简介 —◇

零壹快学程序设计系列丛书是一套针对零基础编程学习者的入门教程，涵盖 Java、PHP、Python、JavaScript、HTML、CSS、Linux、iOS、Go语言、C++、C#等计算机语言。该丛书内容全面、讲解清晰、示例丰富，配套同步教学视频、实战源码、在线答疑等大量学习资源与支持，以立体化、场景化、全方位的教学模式，帮助初学者轻松实现从入门到精通的快速进阶。



千门课程，名师指导，万千学员，零基础，一起学！

零壹快学官方网站：<https://www.01kuaixue.com/>



零壹快学微信小程序
扫一扫，免费获取随书视频教程



扫码下载零壹快学APP
免费领取更多视频课程



零壹快学
微信公众号

