**Go Language**

## Go语言简介

### 1 Go 语言简介

**（1）Go 语言（或 Golang）**

Google 在 2007 年开发的一种开源编程语言，于 2009 年 11 月 10 日向全球公布。Go 是非常年轻的一门语言，它的主要目标是“兼具 Python 等动态语言的开发速度和 C/C++等编译型语言的性能与安全性”。Go 语言是编程语言设计的又一次尝试，是对类C语言的重大改进，它不但能让你访问底层操作系统，还提供了强大的网络编程和并发编程支持。Go 语言的用途众多，可以进行网络编程、系统编程、并发编程、分布式编程。Go 语言的推出，旨在不损失应用程序性能的情况下降低代码的复杂性，具有“部署简单、并发性好、语言设计良好、执行性能好”等优势，目前国内诸多 IT 公司均已采用 Go 语言开发项目。此外，很多重要的开源项目都是使用 Go 语言开发的，其中包括 **Docker、Go-Ethereum、Thrraform 和 Kubernetes。**

**（2）Go语言简史**

对语言进行评估时，明白设计者的动机以及语言要解决的问题很重要。Go 语言出自 Ken Thompson 和 Rob Pike、Robert Griesemer 之手，他们都是计算机科学领域的重量级人物。在 20 世纪 70 年代，Ken Thompson 设计并实现了最初的 UNIX 操作系统，仅从这一点说，他对计算机科学的贡献怎么强调都不过分。他还与 Rob Pike 合作设计了 UTF-8 编码方案。除帮助设计 UTF-8 外，Rob Pike 还帮助开发了分布式多用户操作系统 Plan 9，并与人合著了《The Unix Programming Environment》，对 UNIX 的设计理念做了正统的阐述。

Robert Griesemer 就职于 Google，对语言设计有深入的认识，并负责 Chrome 浏览器和 Node.js 使用的 Google V8 JavaScript 引擎的代码生成部分。

这些计算机科学领城的重量级人物设计 Go 语言的初衷是满足 Google 的需求。设计此语言花费了两年的时间，融入了整个团队多年的经验及对编程语言设计的深入认识。设计团队借鉴了 Pascal、Oberon 和C语言的设计智慧，同时让 Go 语言具备动态语言的便利性。因此，Go 语言体现了经验丰富的计算机科学家的语言设计理念，是为全球最大的互联网公司之一设计的。

Go 语言的所有设计者都说，设计 Go 语言是因为 C++ 给他们带来了挫败感。在 Google I/O 2012 的 Go 设计小组见面会上，Rob Pike 是这样说的：

我们做了大量的 C++ 开发，厌烦了等待编译完成，尽管这是玩笑，但在很大程度上来说也是事实。

您无须知道 Go 语言的设计历史就能使用它。您只需知道，Go 语言的设计和实现体现了多位计算机专家多年的经验以及对其他编程语言优缺点的深入认识。因 C++ 的不良体验而出现的 Go 语言是一门现代编程语言，可用来创建性能卓越的 Web 服务器和系统程序。

**（3）Go 是编译型语言**

Go 使用编译器来编译代码。编译器将源代码编译成二进制（或字节码）格式；在编译代码时，编译器检查错误、优化性能并输出可在不同平台上运行的二进制文件。要创建并运行 Go 程序，程序员必须执行如下步骤。

使用文本编辑器创建 Go 程序；

保存文件；

编译程序；

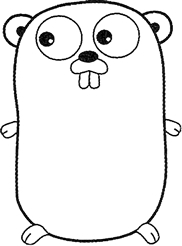
运行编译得到的可执行文件。

这不同于 Python、Ruby 和 JavaScript 等语言，它们不包含编译步骤。Go 自带了编译器，因此无须单独安装编译器。

**（4）为什么要学习 Go 语言**

如果你要创建系统程序，或者基于网络的程序，Go 语言是很不错的选择。作为一种相对较新的语言，它是由经验丰富且受人尊敬的计算机科学家设计的，旨在应对创建大型并发网络程序面临的挑战。如果你觉得 Java 或 C/C++ 的语法导致编程困难，那么 Go 语言将可能提供更佳的体验。对于具备诸如 Ruby、Python、JavaScript 等动态语言使用经验的程序员来说，Go 语言提供了类型安全，同时又不像传统语言那么死板。

**（5）Go 语言吉祥物**

Go 语言有一个吉祥物，在会议、文档页面和博文中，大多会包含下图所示的 Go Gopher，这是才华横溢的插画家 Renee French 设计的，她也是 Go 设计者之一 Rob Pike 的妻子。

### 2 Go语言为并发而生

**（1）多核 CPU**

已经成为服务器的标配，但是对多核的运算能力挖掘一直由程序员人工设计算法及框架来完成，这个过程需要开发人员具有一定的并发设计及框架设计能力。

虽然一些编程语言的框架在不断地提高多核资源使用效率，例如 Java 的 Netty 等，但仍然需要开发人员花费大量的时间和精力搞懂这些框架的运行原理后才能熟练掌握。Go 语言在多核并发上拥有原生的设计优势。Go 语言从 2009 年 11 月开源，2012 年发布 Go 1.0 稳定版本以来，已经拥有活跃的社区和全球众多开发者，并且与苹果公司的 Swift 一样，成为当前非常流行的开发语言之一。很多公司，特别是中国的互联网公司，即将或者已经完成了使用 Go 语言改造旧系统的过程。经过 Go 语言重构的系统能使用更少的硬件资源而有更高的并发和 I/O 吞吐表现。

**（2）Go语言从底层原生支持并发**

无须第三方库、开发者的编程技巧和开发经验就可以轻松地在 Go 语言运行时来帮助开发者决定怎么使用 CPU 资源。

Go语言的并发是基于 goroutine 的，goroutine 类似于线程，但并非线程。可以将 goroutine 理解为一种虚拟线程。Go 语言运行时会参与调度 goroutine，并将 goroutine 合理地分配到每个 CPU 中，最大限度地使用CPU性能。

多个 goroutine 中，Go 语言使用通道（channel）进行通信，程序可以将需要并发的环节设计为生产者模式和消费者的模式，将数据放入通道。通道的另外一端的代码将这些数据进行并发计算并返回结果，如下图所示。

**（3）生产者实例**

下面代码中的生产者每秒生成一个字符串，并通过通道传给消费者，生产者使用两个 goroutine 并发运行，消费者在 main() 函数的 goroutine 中进行处理。

package main

import (

"fmt"

"math/rand"

"time"

)

// 数据生产者

func producer(header string, channel chan<- string) {

// 无限循环, 不停地生产数据

for {

// 将随机数和字符串格式化为字符串发送给通道

channel <- fmt.Sprintf("%s: %v", header, rand.Int31())

// 等待1秒

time.Sleep(time.Second)

}

}

// 数据消费者

func customer(channel <-chan string) {

// 不停地获取数据

for {

// 从通道中取出数据, 此处会阻塞直到信道中返回数据

message := <-channel

// 打印数据

fmt.Println(message)

}

}

func main() {

// 创建一个字符串类型的通道

channel := make(chan string)

// 创建producer()函数的并发goroutine

go producer("cat", channel)

go producer("dog", channel)

// 数据消费函数

customer(channel)

}

运行结果：

dog: 2019727887

cat: 1298498081

dog: 939984059

cat: 1427131847

cat: 911902081

dog: 1474941318

dog: 140954425

cat: 336122540

cat: 208240456

dog: 646203300

**（4）对代码的分析：**

第03行，导入格式化（fmt）、随机数（math/rand）、时间（time）包参与编译。

第10行，生产数据的函数，传入一个标记类型的字符串及一个只能写入的通道。

第13行，for{}构成一个无限循环。

第15行，使用rand.Int31()生成一个随机数，使用fmt.Sprintf()函数将header和随机数格式化为字符串。

第18行，使用time.Sleep()函数暂停1秒再执行这个函数。如果在goroutine中执行时，暂停不会影响其他goroutine的执行。

第23行，消费数据的函数，传入一个只能写入的通道。

第26行，构造一个不断消费消息的循环。

第28行，从通道中取出数据。

第31行，将取出的数据进行打印。

第35行，程序的入口函数，总是在程序开始时执行。

第37行，实例化一个字符串类型的通道。

第39行和第40行，并发执行一个生产者函数，两行分别创建了这个函数搭配不同参数的两个goroutine。

第42行，执行消费者函数通过通道进行数据消费。

整段代码中，没有线程创建，没有线程池也没有加锁，仅仅通过关键字 go 实现 goroutine，和通道实现数据交换。

### 3 哪些项目使用Go语言开发

Go 语言从发布 1.0 版本以来备受众多开发者关注并得到广泛使用，Go 语言的简单、高效、并发特性吸引了众多传统语言开发者的加入，而且人数越来越多。使用 Go 语言开发的开源项目非常多。早期的 Go 语言开源项目只是通过 Go 语言与传统项目进行C语言库绑定实现，例如 Qt、Sqlite 等；后期的很多项目都使用 Go 语言进行重新原生实现，这个过程相对于其他语言要简单一些，这也促成了大量使用 Go 语言原生开发项目的出现。下面列举的是原生使用 Go 语言进行开发的部分项目。

**1) Docker**

Docker 是一种操作系统层面的虚拟化技术，可以在操作系统和应用程序之间进行隔离，也可以称之为容器。Docker 可以在一台物理服务器上快速运行一个或多个实例。例如，启动一个 CentOS 操作系统，并在其内部命令行执行指令后结束，整个过程就像自己在操作系统一样高效。

项目链接：<https://github.com/docker/docker>

**2) go语言**

Go 语言自己的早期源码使用C语言和汇编语言写成。从 Go 1.5 版本后，完全使用 Go 语言自身进行编写。Go 语言的源码对了解 Go 语言的底层调度有极大的参考意义，建议希望对 Go 语言有深入了解的读者读一读。

项目链接：<https://github.com/golang/go>

**3) Kubernetes**

Google 公司开发的构建于 Docker 之上的容器调度服务，用户可以通过 Kubernetes 集群进行云端容器集群管理。

项目链接：<https://github.com/kubernetes/kubernetes>

**4) etcd**

一款分布式、可靠的 KV 存储系统，可以快速进行云配置。

项目链接：<https://github.com/coreos/etcd>

**5) beego**

beego 是一个类似 Python 的 Tornado 框架，采用了 RESTFul 的设计思路，使用 Go 语言编写的一个极轻量级、高可伸缩性和高性能的 Web 应用框架。

项目链接：<https://github.com/astaxie/beego>

**6) martini**

一款快速构建模块化的 Web 应用的 Web 框架。

项目链接：<https://github.com/go-martini/martini>

**7) codis**

国产的优秀分布式 Redis 解决方案。

项目链接：<https://github.com/CodisLabs/codis>

**8) delve**

Go语言强大的调试器，被很多集成环境和编辑器整合。

### 4 Go语言的性能如何？

这里以国外的一个编程语言性能测试网站 http://benchmarksgame.alioth.debian.org/ 为测试基准和数据源。这个网站可以对常见的编程语言进行性能比较，网站使用都是最新的语言版本和常见的一些算法。进行测试的编程语言包括：C(gcc)、[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank)、[Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank)、JavaScript 和 [Go语言](http://c.biancheng.net/golang/" \t "http://c.biancheng.net/view/_blank)。性能比较如下表所示，表中数据的单位为秒，数值越小表明运行性能越好。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 常见编程语言的运行性能比较 | | | | | | | | | | |
| **编程语言↓ / 测试用例→** | **reverse-complement** | **pidigits** | **fannkuch-redux** | **fasta** | **spectral-norm** | **n-body** | **k-nucleotide** | **mandelbrot** | **binary-trees** | **regex-redux** |
| [C语言](http://c.biancheng.net/c/" \t "http://c.biancheng.net/view/_blank) | 0.42 | 1.73 | 8.97 | 1.33 | 1.99 | 9.96 | 5.38 | 1.65 | 2.38 | 1.45 |
| C++ | 0.6 | 1.89 | 10.35 | 1.48 | 1.99 | 9.31 | 7.18 | 1.73 | 2.36 | 17.14 |
| Go | 0.49 | 2.02 | 14.49 | 2.17 | 3.96 | 21.47 | 14.79 | 5.46 | 35.18 | 29.29 |
| Java | 1.13 | 3.12 | 15.09 | 2.32 | 4.25 | 22.56 | 8.38 | 6.08 | 8.58 | 10.38 |
| JavaScript | 4.3 | N/A | 81.49 | 9.79 | 16.17 | 28.74 | 66.07 | 19.04 | 53.64 | 4.44 |

可以看出，[Go 语言](http://c.biancheng.net/golang/" \t "http://c.biancheng.net/view/_blank)在性能上更接近于 Java 语言，虽然在某些测试用例上不如经过多年优化的 Java 语言，但毕竟 Java 语言已经经历了多年的积累和优化。Go 语言在未来的版本中会通过不断的版本优化提高单核运行性能。

### 5 Go语言标准库强大

**[Go语言](http://c.biancheng.net/golang/" \t "http://c.biancheng.net/view/_blank)的标准库覆盖网络、系统、加密、编码、图形等各个方面，可以直接使用标准库的 http 包进行 HTTP 协议的收发处理；**网络库基于高性能的操作系统通信模型（[Linux](http://c.biancheng.net/linux_tutorial/" \t "http://c.biancheng.net/view/_blank) 的 epoll、Windows 的 IOCP）；所有的加密、编码都内建支持，不需要再从第三方开发者处获取。[Go 语言](http://c.biancheng.net/golang/" \t "http://c.biancheng.net/view/_blank)的编译器也是标准库的一部分，通过词法器扫描源码，使用语法树获得源码逻辑分支等。Go 语言的周边工具也是建立在这些标准库上。在标准库上可以完成几乎大部分的需求。Go 语言的标准库以包的方式提供支持，下表是 Go 语言标准库中常见的包及其功能。

|  |  |
| --- | --- |
|  | |
| **Go语言标准库包名** | **功  能** |
| bufio | 带缓冲的 I/O 操作 |
| bytes | 实现字节操作 |
| container | 封装堆、列表和环形列表等容器 |
| crypto | 加密算法 |
| database | 数据库驱动和接口 |
| debug | 各种调试文件格式访问及调试功能 |
| encoding | 常见算法如 JSON、XML、Base64 等 |
| flag | 命令行解析 |
| fmt | 格式化操作 |
| go | Go 语言的词法、语法树、类型等。可通过这个包进行代码信息提取和修改 |
| html | HTML 转义及模板系统 |
| image | 常见图形格式的访问及生成 |
| io | 实现 I/O 原始访问接口及访问封装 |
| math | 数学库 |
| net | 网络库，支持 Socket、HTTP、邮件、RPC、SMTP 等 |
| os | 操作系统平台不依赖平台操作封装 |
| path | 兼容各操作系统的路径操作实用函数 |
| plugin | Go 1.7 加入的插件系统。支持将代码编译为插件，按需加载 |
| reflect | 语言反射支持。可以动态获得代码中的类型信息，获取和修改变量的值 |
| regexp | 正则表达式封装 |
| runtime | 运行时接口 |
| sort | 排序接口 |
| strings | 字符串转换、解析及实用函数 |
| time | 时间接口 |
| text | 文本模板及 Token 词法器 |

### 6 Go语言上手简单

**（1） Go语言标准库强大Go语言代码风格清晰、简单**

Go 语言简单易学，学习曲线平缓，不需要像 C/C++ 语言动辄需要两到三年的学习期。Go 语言被称为“互联网时代的C语言”。互联网的短、频、快特性在 Go 语言中体现得淋漓尽致。一个熟练的开发者只需要短短的一周时间就可以从学习阶段转到开发阶段，并完成一个高并发的服务器开发。Go 语言是 Google 公司开发的一种静态型、编译型并自带垃圾回收和并发的编程语言。

Go 语言的风格类似于C语言。其语法在C语言的基础上进行了大幅的简化，去掉了不需要的表达式括号，循环也只有 for 一种表示方法，就可以实现数值、键值等各种遍历。因此，Go 语言上手非常容易。很多读者表示自己是在看了介绍后才开始了解这门语言的，他们一般也会使用两到三门编程语言。Go 语言对于他们来说，也就是一到两天的熟悉过程，之后就可以开始使用 Go 语言解决具体问题了，大约一周左右已经可以使用 Go 语言完成既定的任务了。

Go 语言这种从零开始使用到解决问题的速度，在其他语言中是完全不可想象的。学过 C++ 的朋友都知道，一到两年大强度的理论学习和实战操练也只能学到这门语言的皮毛，以及知道一些基本的避免错误的方法。

**（2）那么，Go 语言到底有多么简单？下面从实现一个 HTTP 服务器开始了解。**

HTTP 文件服务器是常见的 Web 服务之一。开发阶段为了测试，需要自行安装 Apache 或 Nginx 服务器，下载安装配置需要大量的时间。使用 Go 语言实现一个简单的 HTTP 服务器只需要几行代码，如下所示。

package main

import (

"net/http"

)

func main() {

http.Handle("/", http.FileServer(http.Dir(".")))

http.ListenAndServe(":8080", nil)

}

下面是代码说明：

第 1 行，标记当前文件为 main 包，main 包也是 Go 程序的入口包。

第 3~5 行，导入 net/http 包，这个包的作用是 HTTP 的基础封装和访问。

第 7 行，程序执行的入口函数 main()。

第 8 行，使用 http.FileServer 文件服务器将当前目录作为根目录（/目录）的处理器，访问根目录，就会进入当前目录。

第 9 行，默认的 HTTP 服务侦听在本机 8080 端口。

把这个源码保存为 main.go（Go 语言的源文件后缀就是.go），安装 Go 语言的开发包（后续我们会讲解如何安装），在命令行输入如下命令：

$ go run main.go

在浏览器里输入http://127.0.0.1:8080即可浏览文件，这些文件正是当前目录在HTTP服务器上的映射目录。

**（3）Go 语言工程结构简单**

Go 语言的源码无须头文件，编译的文件都来自于后缀名为.go的源码文件。Go 语言无须解决方案、工程文件和 Make File，只要将工程文件按照 GOPATH 的规则进行填充，即可使用 go build/go install 进行编译，编译安装的二进制可执行文件统一放在 bin 文件夹下。

后面的章节会介绍 GOPATH 及 go build/go install 的详细使用方法。

**（4）Go 语言编译速度快**

Go 语言可以利用自己的特性实现并发编译，并发编译的最小元素是包。从 Go 1.9 版本开始，最小并发编译元素缩小到函数，整体编译速度提高了 20%。另外，Go 语言语法简单，具有严谨的工程结构设计、没有头文件、不允许包的交叉依赖等规则，在很大程度上加速了编译的过程。

### 7 Go语言代码风格清晰、简单

Go 语言写起来类似于C语言，因此熟悉C语言及其派生语言（C++、C#、Objective-C 等）的人都会迅速熟悉这门语言。

C语言的有些语法会让代码可读性降低甚至发生歧义。Go 语言在C语言的基础上取其精华，弃其糟粕，将C语言中较为容易发生错误的写法进行调整，做出相应的编译提示。

**1) 去掉循环冗余括号**

Go 语言在众多大师的丰富实战经验的基础上诞生，去除了C语言语法中一些冗余、烦琐的部分。下面的代码是C语言的数值循环：

// C语言的for数值循环

for(int a = 0;a<10;a++){

// 循环代码

}

在Go语言中，这样的循环变为：

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

// 循环代码

}

for 两边的括号被去掉，int 声明被简化为:=，直接通过编译器右值推导获得 a 的变量类型并声明。

**2) 去掉表达式冗余括号**

同样的简化也可以在判断语句中体现出来，以下是C语言的判断语句：

if (表达式){

// 表达式成立

}

在 Go 语言中，无须添加表达式括号，代码如下：

if 表达式{

// 表达式成立

}

**3) 强制的代码风格**

Go 语言中，左括号必须紧接着语句不换行。其他样式的括号将被视为代码编译错误。这个特性刚开始会使开发者有一些不习惯，但随着对 Go 语言的不断熟悉，开发者就会发现风格统一让大家在阅读代码时把注意力集中到了解决问题上，而不是代码风格上。

同时 Go 语言也提供了一套格式化工具。一些 Go 语言的开发环境或者编辑器在保存时，都会使用格式化工具进行修改代码的格式化，让代码提交时已经是统一格式的代码。

**4) 不再纠结于 i++ 和 ++i**

C语言非常经典的考试题为：

int a, b;

a = i++;

b = ++i;

这种题目对于初学者简直摸不着头脑。为什么一个简单的自增表达式需要有两种写法？

在 Go 语言中，自增操作符不再是一个操作符，而是一个语句。因此，在 Go 语言中自增只有一种写法：

i++

如果写成前置自增++i，或者赋值后自增a=i++都将导致编译错误。

### 8 安装Go语言开发包

|  |  |
| --- | --- |
| **（1）Go 安装包命名及对应的平台** | |
| 文件名 | 说明 |
| go1.10.3.src.tar.gz | 源码包，供源码研究，对于日常开发不建议下载此包 |
| go1.10.3.darwin-amd64.pkg | Mac OS 平台安装包 |
| go1.10.3.linux-amd64.tar.gz | Linux 平台安装包 |
| go1.10.3.windows-amd64.msi | Windows 平台安装包 |

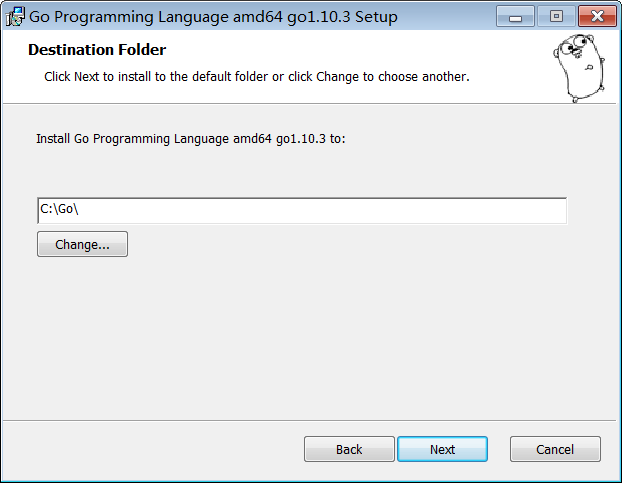
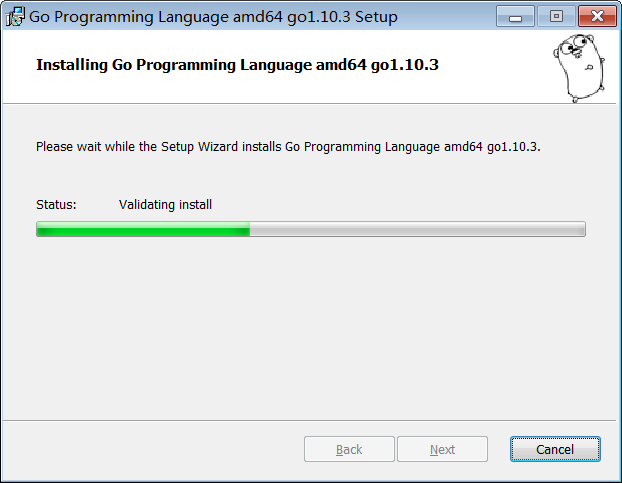
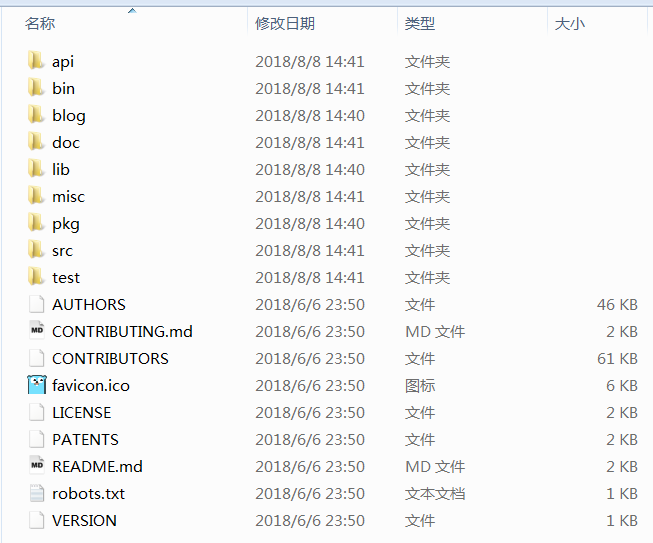
**（2）Windows 版安装**

Go 语言的 Windows 版安装包一般格式为 MSI 格式，可以直接安装到系统，Go 语言的 Windows 安装包一般命名如下：

go1.10.3.windows-amd64.msi

1.10.3 表示 Go 安装包的版本；

Windows 表示这是一个 Windows 安装包；

amd64 表示匹配的 CPU 版本，这里匹配的是 64 位 CPU。  
Windows 下 Go 开发包的默认安装路径是 C 盘的 Go 目录下，推荐在这个目录下安装 Go 开发包，使用起来较为方便。Go 开发包安装完毕后占用磁盘空间大概是 300MB 左右。当然，在安装过程中你也可以选择安装路径，如下图所示：  
  
  
Go 开发包的安装没有其他选项，接下来是安装程序的文件复制操作，如下图所示：  
  
  
安装完成后，安装目录下将生成一些目录和文件，如下图所示：  
  
  
这个目录的结构遵守 GOPATH 规则，后面的章节会提到这个概念。GOPATH 及相关的目录命名是 Go 语言编译的核心规则。

|  |  |
| --- | --- |
| Go 开发包的安装目录的功能及说明 | |
| 目录名 | 说明 |
| api | 每个版本的 api 变更差异 |
| bin | go 源码包编译出的编译器（go）、文档工具（godoc）、格式化工具（gofmt） |
| blog | Go 博客的模板，使用 Go 的网页模板，有一定的学习意义 |
| doc | 英文版的 Go 文档 |
| lib | 引用的一些库文件 |
| misc | 杂项用途的文件，例如 [Android](http://c.biancheng.net/android/" \t "http://c.biancheng.net/view/_blank) 平台的编译、git 的提交钩子等 |
| pkg | Windows 平台编译好的中间文件 |
| src | 标准库的源码 |
| test | 测试用例 |

开发时，无须关注这些目录。当读者希望深度了解底层原理时，可以通过上面的介绍继续探索。

**（3）Linux 版安装**

Linux 版的 Go 语言压缩包格式如下：

go1.10.3.linux-amd64.tar.gz

需要将这个包解压到/usr/local/go下，可以用下列命令来完成：

tar -C /usr/local -xzf go1.10.3.linux-amd64.tar.gz

请根据下载的Go语言压缩包的版本进行安装。  
接下来，需要将/usr/local/go/bin目录添加到PATH环境变量中，可以使用以下命令来完成：

export PATH=$PATH:/usr/local/go/bin

使用go env指令，可以查看 Go 压缩包是否安装成功：

$ go env  
GOARCH="amd64"  
GOBIN=""  
GOEXE=""  
GOHOSTARCH="amd64"  
GOHOSTOS="linux"  
GOOS="linux"  
GOPATH="/root/go"  
GORACE=""  
GOROOT="/usr/local/go"  
GOTOOLDIR="/usr/local/go/pkg/tool/linux\_amd64"  
[GCC](http://c.biancheng.net/gcc/" \t "http://c.biancheng.net/view/_blank)GO="gccgo"  
CC="gcc"  
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/  
tmp/go-build305492722=/tmp/go-build -gno-record-gcc-switches"  
CXX="g++"  
CGO\_ENABLED="1"  
CGO\_CFLAGS="-g -O2"  
CGO\_CPPFLAGS=""  
CGO\_CXXFLAGS="-g -O2"  
CGO\_FFLAGS="-g -O2"  
CGO\_LDFLAGS="-g -O2"  
PKG\_CONFIG="pkg-config"

## [Go语言基本语法](http://c.biancheng.net/golang/syntax/)

### 1 Go语言变量的声明（使用var关键字）

（1）变量（Variable）的功能是存储用户的数据。不同的逻辑有不同的对象类型，也就有不同的变量类型。经过半个多世纪的发展，编程语言已经形成一套固定的类型，这些类型在不同的编程语言中基本是相通的。常见变量的数据类型有：整型、浮点型、布尔型、结构体等。Go 语言作为C语言家族的新派代表，在C语言的定义方法和类型上做了优化和调整，更加灵活易学。Go 语言的每一个变量都拥有自己的类型，必须经过声明才能开始用。

var a int

var b string

var c []float32

var d func() bool

var e struct{

x int

}

代码说明如下：

第1行，声明一个整型类型的变量，可以保存整数数值。

第2行，声明一个字符串类型的变量。

第3行，声明一个 32 位浮点切片类型的变量，浮点切片表示由多个浮点类型组成的数据结构。

第4行，声明一个返回值为布尔类型的函数变量，这种形式一般用于回调函数，即将函数以变量的形式保存下来，在需要的时候重新调用这个函数。

第5行，声明一个结构体类型的变量，这个结构体拥有一个整型的 x 字段。

上面代码的共性是，以 var 关键字开头，要声明的变量名放在中间，而将其类型放在后面

变量的声明有几种形式，通过下面几节进行整理归纳。

**（2）标准格式**

Go 语言的变量声明格式为：

var 变量名 变量类型

变量声明以关键字 var 开头，后置变量类型，行尾无须分号。

**（3）批量格式**

觉得每行都用 var 声明变量比较烦琐？没关系，还有一种为懒人提供的定义变量的方法：

纯文本复制

var (

a int

b string

c []float32

d func() bool

e struct {

x int

}

)

使用关键字var和括号，可以将一组变量定义放在一起。

### 2 Go语言变量的初始化

**（1）Go 语言在声明变量时，自动对变量对应的内存区域进行初始化操作。每个变量会初始化其类型的默认值**

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 bool。

切片、函数、指针变量的默认为 nil。

当然，依然可以在变量声明时赋予变量一个初始值。

回顾C语言

在C语言中，变量在声明时，并不会对变量对应内存区域进行清理操作。此时，变量值可能是完全不可预期的结果。开发者需要习惯在使用C语言进行声明时要初始化操作，稍有不慎，就会造成不可预知的后果。在网络上只有程序员才能看懂的“烫烫烫”和“屯屯屯”的梗，就来源于C/C++中变量默认不初始化。微软的 VC 编译器会将未初始化的栈空间以 16 进制的 0xCC 填充，未初始化的堆空间使用 0xCD 填充，而 0xCCCC 和 0xCDCD 在中文的 GB2312 编码中刚好对应“烫”和“屯”字。因此，如果一个字符串没有结束符\0，直接输出的内存数据转换为字符串就刚好对应“烫烫烫”和“屯屯屯”。

**（2）变量初始化的标准格式**

var 变量名 类型 = 表达式

例如，游戏中，玩家的血量初始值为100。可以这样写：

var hp int = 100

这句代码中，hp 为变量名，类型为 int，hp 的初始值为 100。

上面代码中，100 和 int 同为 int 类型，int 可以认为是冗余信息，因此可以进一步简化初始化的写法。

**（3）编译器推导类型的格式**

在标准格式的基础上，将 int 省略后，编译器会尝试根据等号右边的表达式推导 hp 变量的类型。

var hp = 100

等号右边的部分在编译原理里被称做右值（rvalue）。

下面是编译器根据右值推导变量类型完成初始化的例子。

var attack = 40

var defence = 20

var damageRate float32 = 0.17

var damage = float32(attack-defence) \* damageRate

fmt.Println(damage)

代码说明如下：

第 1 和 2 行，右值为整型，attack 和 defence 变量的类型为 int。

第 3 行，表达式的右值中使用了 0.17。Go 语言和C语言一样，编译器会尽量提高精确度，以避免计算中的精度损失。

默认情况下，如果不指定 damageRate 变量的类型，Go 语言编译器会将 damageRate 类型推导为 float64。由于这个例子中不需要 float64 的精度，所以强制指定类型为 float32。

第 4 行，将 attack 和 defence 相减后的数值结果依然为整型，使用 float32() 将结果转换为 float32 类型，再与 float32 类型的 damageRate 相乘后，damage 类型也是 float32 类型。

提示：damage 变量的右值是一个复杂的表达式，整个过程既有 attack 和 defence 的运算还有强制类型转换。强制类型转换会在后面的章节中介绍。

第 5 行，输出 damage 的值。

以上代码输出结果为：

3.4

**（4）短变量声明并初始化**

var 的变量声明还有一种更为精简的写法，例如：

hp := 100

这是 Go 语言的推导声明写法，编译器会自动根据右值类型推断出左值的对应类型。

注意：由于使用了:=，而不是赋值的=，因此推导声明写法的左值变量必须是没有定义过的变量。若定义过，将会发生编译错误。

如果 hp 已经被声明过，但依然使用:=时编译器会报错，代码如下：

// 声明 hp 变量

var hp int

// 再次声明并赋值

hp := 10

编译报错如下：

no new variables on left side of :=

意思是，在“:=”的左边没有新变量出现，意思就是“:=”的左边变量已经被声明了。

短变量声明的形式在开发中的例子较多，比如：

conn, err := net.Dial("tcp","127.0.0.1:8080")

net.Dial 提供按指定协议和地址发起网络连接，这个函数有两个返回值，一个是连接对象，一个是 err 对象。如果是标准格式将会变成：

var conn net.Conn

var err error

conn, err = net.Dial("tcp", "127.0.0.1:8080")

因此，短变量声明并初始化的格式在开发中使用比较普遍。

注意：在多个短变量声明和赋值中，至少有一个新声明的变量出现在左值中，即便其他变量名可能是重复声明的，编译器也不会报错，代码如下：

conn, err := net.Dial("tcp", "127.0.0.1:8080")

conn2, err := net.Dial("tcp", "127.0.0.1:8080")

上面的代码片段，编译器不会报err重复定义。

### 3 Go语言多个变量同时赋值

**（1）编程最简单的算法之一，莫过于变量交换。交换变量的常见算法需要一个中间变量进行变量的临时保存**。用传统方法编写变量交换代码如下：

var a int = 100

var b int = 200

var t int

t = a

a = b

b = t

fmt.Println(a, b)

**（2）在计算机刚发明时，内存非常“精贵”。**这种变量交换往往是非常奢侈的。于是计算机“大牛”发明了一些算法来避免使用中间变量：

var a int = 100

var b int = 200

a = a ^ b

b = b ^ a

a = a ^ b

fmt.Println(a, b)

这样的算法很多，但是都有一定的数值范围和类型要求。

**（3）到了 Go 语言时，内存不再是紧缺资源，而且写法可以更简单。**使用 Go 的“多重赋值”特性，可以轻松完成变量交换的任务：

var a int = 100

var b int = 200

b, a = a, b

fmt.Println(a, b)

多重赋值时，变量的左值和右值按从左到右的顺序赋值。

多重赋值在 Go 语言的错误处理和函数返回值中会大量地使用。

例如，使用 Go 语言进行排序时就需要使用交换，代码如下：

type IntSlice []int

func (p IntSlice) Len() int { return len(p) }

func (p IntSlice) Less(i, j int) bool { return p[i] < p[j] }

func (p IntSlice) Swap(i, j int) { p[i], p[j] = p[j], p[i] }

代码说明如下：

第 1 行，将 []int 声明为 IntSlice 类型。

第 3 行，为这个类型编写一个 Len 方法，提供切片的长度。

第 4 行，根据提供的 i、j 元素索引，获取元素后进行比较，返回比较结果。

第 5 行，根据提供的 i、j 元素索引，交换两个元素的值。

### 4 Go语言匿名变量（没有名字的变量）

在使用多重赋值时，如果不需要在左值中接收变量，可以使用匿名变量（anonymous variable）。

匿名变量的表现是一个下画线\_，使用匿名变量时，只需要在变量声明的地方使用下画线替换即可。例如：

func GetData() (int, int) {

return 100, 200

}

a, \_ := GetData()

\_, b := GetData()

fmt.Println(a, b)

代码运行结果：

100 200

GetData() 是一个函数，拥有两个整型返回值。每次调用将会返回 100 和 200 两个数值。

代码说明如下：

第 5 行只需要获取第一个返回值，所以将第二个返回值的变量设为下画线。

第 6 行将第一个返回值的变量设为匿名。

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

提示：在 Lua 等编程语言里，匿名变量也被叫做哑元变量。

### 5 Go语言整型（整数类型）

**（1）Go 语言中有丰富的数据类型，除了基本的整型、浮点型、布尔型、字符串外，还有切片、结构体、函数、map、通道（channel）等。**Go 语言的基本类型和其他语言大同小异，切片类型有着指针的便利性，但比指针更为安全，很多高级语言都配有切片进行安全和高效率的内存操作。

**（2）整型分为以下两个大类：**

按长度分为：int8、int16、int32、int64

还有对应的无符号整型：uint8、uint16、uint32、uint64

其中，uint8 就是我们熟知的 byte 型，int16 对应C语言中的 short 型，int64 对应C语言中的 long 型。

自动匹配平台的 int 和 uint

**（3）Go 语言也有自动匹配特定平台整型长度的类型—— int 和 uint。**

可以跨平台的编程语言可以运行在多种平台上。平台的字节长度是有差异的。64 位平台现今已经较为普及，但 8 位、16 位、32 位的操作系统依旧存在。16 位平台上依然可以使用 64 位的变量，但运行性能和内存性能上较差。同理，在 64 位平台上大量使用 8 位、16 位等与平台位数不等长的变量时，编译器也是尽量将内存对齐以获得最好的性能。不能正确匹配平台字节长度的程序就类似于用轿车运一头牛和用一辆卡车运送一头牛的情形一样。在使用 int 和 uint 类型时，不能假定它是 32 位或 64 位的整型，而是考虑 int 和 uint 可能在不同平台上的差异。

**（4）哪些情况下使用 int 和 uint**

逻辑对整型范围没有特殊需求。例如，对象的长度使用内建 len() 函数返回，这个长度可以根据不同平台的字节长度进行变化。实际使用中，切片或 map 的元素数量等都可以用 int 来表示。反之，在二进制传输、读写文件的结构描述时，为了保持文件的结构不会受到不同编译目标平台字节长度的影响，不要使用 int 和 uint。

### 6 Go语言浮点类型（小数类型）

Go语言支持两种浮点型数：float32 和 float64。这两种浮点型数据格式遵循 IEEE 754 标准：

float32 的浮点数的最大范围约为 3.4e38，可以使用常量定义：math.MaxFloat32。

float64 的浮点数的最大范围约为 1.8e308，可以使用一个常量定义：math.MaxFloat64。

打印浮点数时，可以使用 fmt 包配合动词%f，代码如下：

package main

import (

"fmt"

"math"

)

func main() {

fmt.Printf("%f\n", math.Pi)

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

}

代码说明如下：

第 9 行，按默认宽度和精度输出整型。

第 10 行，按默认宽度，2 位精度输出（小数点后的位数）。

代码运行结果如下：

3.141593

3.14

### 7 Go语言输出正弦函数（Sin）图像

**（1）在 Go 语言中，正弦函数由 math 包提供，函数入口为 math.Sin。**正弦函数的参数为 float64，返回值也是 float64。在使用正弦函数时，根据实际精度可以进行转换。

Go 语言的标准库支持对图片像素进行访问，并且支持输出各种图片格式，如 JPEG、PNG、GIF 等。

首先给出本节完整的代码：

package main

import (

"image"

"image/color"

"image/png"

"log"

"math"

"os"

)

func main() {

// 图片大小

const size = 300

// 根据给定大小创建灰度图

pic := image.NewGray(image.Rect(0, 0, size, size))

// 遍历每个像素

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

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

// 填充为白色

pic.SetGray(x, y, color.Gray{255})

}

}

// 从0到最大像素生成x坐标

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

// 让sin的值的范围在0~2Pi之间

s := float64(x) \* 2 \* math.Pi / size

// sin的幅度为一半的像素。向下偏移一半像素并翻转

y := size/2 - math.Sin(s)\*size/2

// 用黑色绘制sin轨迹

pic.SetGray(x, int(y), color.Gray{0})

}

// 创建文件

file, err := os.Create("sin.png")

if err != nil {

log.Fatal(err)

}

// 使用png格式将数据写入文件

png.Encode(file, pic) //将image信息写入文件中

// 关闭文件

file.Close()

}

**（2）设置图片背景色**

以下是设置图片背景的代码：

// 图片大小

const size = 300

// 根据给定大小创建灰度图

pic := image.NewGray(image.Rect(0, 0, size, size))

// 遍历每个像素

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

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

// 填充为白色

pic.SetGray(x, y, color.Gray{255})

}

}

代码说明如下：

第 2 行，声明一个 size 常量，值为 300。

第 5 行，使用 image 包的 NewGray() 函数创建一个图片对象，使用区域由 image.Rect 结构提供。image.Rect 描述一个方形的两个定位点 (x1,y1) 和 (x2,y2)。image.Rect(0,0,size,size) 表示使用完整灰度图像素，尺寸为宽 300，长 300。

第 8 行和第 9 行，遍历灰度图的所有像素。

第 11 行，将每一个像素的灰度设为 255，也就是白色。

灰度图是一种常见的图片格式，一般情况下颜色由 8 位组成，灰度范围为 0～255，0 表示黑色，255 表示白色。

初始化好的灰度图对象内存区域默认值都是 0，对应全是黑色，考虑到显示效果和习惯，将所有像素设置为 255，也就是白色。

**（3）绘制正弦函数轨迹**

正弦函数是一个周期函数。定义域是实数集，值域范围是 [-1, 1]。用编程的通俗易懂的话来说就是：math.Sin 函数的参数支持任意浮点数范围，函数返回值的范围总是在 -1～1之间（两端包含）。要将正弦函数放在图片上需要考虑以下一些因素：

math.Sin 的返回值在 -1～1 之间。需要考虑将正弦的输出幅度变大，可以将 math.Sin 的返回值乘以一个常量进行放大。片的坐标系原点在左上角，而 math.Sin 基于笛卡尔坐标系原点在左下角。需要对图像进行上下翻转和平移。

将这些处理逻辑汇总为代码如下：

// 从0到最大像素生成x坐标

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

// 让sin的值的范围在0~2Pi之间

s := float64(x) \* 2 \* math.Pi / size

// sin的幅度为一半的像素。向下偏移一半像素并翻转

y := size/2 - math.Sin(s)\*size/2

// 用黑色绘制sin轨迹

pic.SetGray(x, int(y), color.Gray{0})

}

代码说明如下：

1) 第 2 行，生成 0 到 size（300）的 x 坐标轴。

2) 第 5 行，计算 math.Sin 的定义域，这段代码等效为：

rate := x / size

s := rate \* 2 \* math.Pi

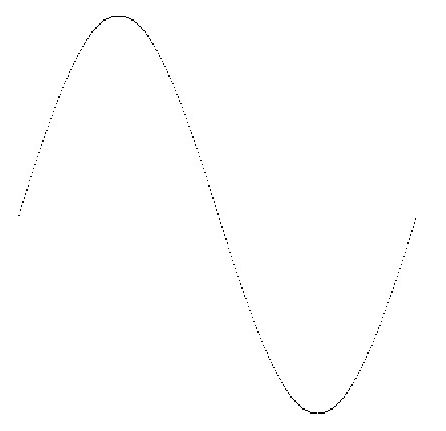
x 的范围是 0 到 size，因此除以 size 后，rate 的范围是 0～1 之间，再乘以 2π 后，s 的范围刚好是 0～2π 之间。

float64(x) 表示将整型的 x 变量转换为 float64 类型，之后运算的所有表达式将以 float64 类型进行。

3) 第 8 行中，math.Sin(s)\*size/2 表示将正弦函数的返回值幅度从 1 扩大到二分之一的 size。负号表示将正弦函数图形以图形中心上下翻转。叠加 size/2 表示将图形在 y 轴上向下偏移二分之一的 size（图片坐标系的 y 向下）。

4) 第 11 行将计算好的 x 轴和 y 轴数据，以灰度为 0（黑色）使用 SetGray() 方法填充到像素中。

写入图片的正弦函数图像如下图所示：



**（4）写入图片文件**

内存中的正弦函数图形是不可见的，我们选用 PNG 格式将图形输出为文件。Go 语言提供了文件创建函数和 PNG 格式写入函数，代码如下：

// 创建文件

file, err := os.Create("sin.png")

if err != nil {

log.Fatal(err)

}

// 使用PNG格式将数据写入文件

png.Encode(file, pic) //将image信息写入文件中

// 关闭文件

file.Close()

代码说明如下：

第 2 行，创建 sin.png 的文件。

第 4 行，如果创建文件失败，返回错误，打印错误并终止。

第 8 行，使用 PNG 包，将图形对象写入文件中。

第 11 行，关闭文件。

### 8 Go语言bool类型（布尔类型）

布尔型数据在 Go 语言中以 bool 类型进行声明，布尔型数据只有 true（真）和 false（假）两个值。

Go 语言中不允许将整型强制转换为布尔型，代码如下：

var n bool

fmt.Println(int(n) \* 2)

编译错误，输出如下：

cannot convert n (type bool) to type int

布尔型无法参与数值运算，也无法与其他类型进行转换。

### 9 Go语言字符串

**（1）字符串在 Go 语言中以原生数据类型出现，使用字符串就像使用其他原生数据类型（int、bool、float32、float64 等）一样。**

提示：在 C++、C# 语言中，字符串以类的方式进行封装。

C# 语言中在使用泛型匹配约束类型时，字符串是以 Class 的方式存在，而不是 String，因为并没有“字符串”这种原生数据类型。

在 C++ 语言中使用模板匹配类型时，为了使字符串与其他原生数据类型一样支持赋值操作，需要对字符串类进行操作符重载。

字符串的值为双引号中的内容，可以在 Go 语言的源码中直接添加非 ASCII 码字符，代码如下：

str := "hello world"

ch := "中文"

**（2）字符串转义符**

Go 语言的字符串常见转义符包含回车、换行、单双引号、制表符等，如下表所示。

Go语言的常见转义符

转移符 含 义

\r 回车符（返回行首）

\n 换行符（直接跳到下一行的同列位置）

\t 制表符

\' 单引号

\" 双引号

\\ 反斜杠

在 Go 语言源码中使用转义符代码如下：

package main

import (

"fmt"

)

func main() {

fmt.Println("str := \"c:\\Go\\bin\\go.exe\"")

}

代码运行结果：

str := "c:\Go\bin\go.exe"

这段代码中将双引号和反斜杠“\”进行转义。

**（3）字符串实现基于 UTF-8 编码**

Go 语言里的字符串的内部实现使用 UTF-8 编码。通过 rune 类型，可以方便地对每个 UTF-8 字符进行访问。当然，Go 语言也支持按传统的 ASCII 码方式进行逐字符访问。

提示：Python 语言的 2.0 版本不是基于 UTF-8 编码设计，到了 3.0 版才改为 UTF-8 编码设计。因此，使用 2.0 版本时，在编码上会出现很多混乱情况。同样，C/C++语言的 std::string 在使用 UTF-8 时，经常因为没有方便的 UTF-8 配套封装让编写极为困难。关于字符串的 UTF-8 字符访问的详细方法，后面的章节将会详细介绍。

**（4）定义多行字符串**

在源码中，将字符串的值以双引号书写的方式是字符串的常见表达方式，被称为字符串字面量（string literal）。这种双引号字面量不能跨行。如果需要在源码中嵌入一个多行字符串时，就必须使用`字符，代码如下：

const str = ` 第一行

第二行

第三行

\r\n

`

fmt.Println(str)

代码运行结果：

第一行

第二行

第三行

\r\n

`叫反引号，就是键盘上 1 键左边的键，两个反引号间的字符串将被原样赋值到 str 变量中。

在这种方式下，反引号间换行将被作为字符串中的换行，但是所有的转义字符均无效，文本将会原样输出。

多行字符串一般用于内嵌源码和内嵌数据等，代码如下：

const codeTemplate = `// Generated by github.com/davyxu/cellnet/

protoc-gen-msg

// DO NOT EDIT!{{range .Protos}}

// Source: {{.Name}}{{end}}

package {{.PackageName}}

{{if gt .TotalMessages 0}}

import (

"github.com/davyxu/cellnet"

"reflect"

\_ "github.com/davyxu/cellnet/codec/pb"

)

{{end}}

func init() {

{{range .Protos}}

// {{.Name}}{{range .Messages}}

cellnet.RegisterMessageMeta("pb","{{.FullName}}", reflect.TypeOf((\*{{.Name}})(nil)).Elem(), {{.MsgID}}) {{end}}

{{end}}

}

`

这段代码只定义了一个常量 codeTemplate，类型为字符串，使用`定义。字符串的内容为一段代码生成中使用到的 Go 源码格式。在`间的所有代码均不会被编译器识别，而只是作为字符串的一部分。

### 10 Go语言字符类型（byte和rune）

**（1）字符串中的每一个元素叫做“字符”，在遍历或者单个获取字符串元素时可以获得字符。**

Go 语言的字符有以下两种：

一种是 uint8 类型，或者叫 byte 型，代表了 ASCII 码的一个字符。

另一种是 rune 类型，代表一个 UTF-8 字符。当需要处理中文、日文或者其他复合字符时，则需要用到 rune 类型。rune 类型实际是一个 int32。

使用 fmt.Printf 中的%T动词可以输出变量的实际类型，使用这个方法可以查看 byte 和 rune 的本来类型，代码如下：

var a byte = 'a'

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

var b rune = '你'

fmt.Printf("%d %T\n", b, b)

例子输出结果：

97 uint8

20320 int32

可以发现，byte 类型的 a 变量，实际类型是 uint8，其值为 'a'，对应的 ASCII 编码为 97。

rune 类型的 b 变量的实际类型是 int32，对应的 Unicode 码就是 20320。

Go 使用了特殊的 rune 类型来处理 Unicode，让基于 Unicode 的文本处理更为方便，也可以使用 byte 型进行默认字符串处理，性能和扩展性都有照顾。

**（2）UTF-8 和 Unicode 有何区别？**

Unicode 是字符集，ASCII 也是一种字符集。

字符集为每个字符分配一个唯一的 ID，我们使用到的所有字符在 Unicode 字符集中都有唯一的一个 ID 对应，例如上面例子中的 a 在 Unicode 与 ASCII 中的编码都是 97。“你”在 Unicode 中的编码为 20320，但是在不同国家的字符集中，“你”的 ID 会不同。而无论任何情况下，Unicode 中的字符的 ID 都是不会变化的。UTF-8 是编码规则，将 Unicode 中字符的 ID 以某种方式进行编码。UTF-8 的是一种变长编码规则，从 1 到 4 个字节不等。编码规则如下：

0xxxxxx 表示文字符号 0～127，兼容 ASCII 字符集。

从 128 到 0x10ffff 表示其他字符。

根据这个规则，拉丁文语系的字符编码一般情况下，每个字符依然占用一个字节，而中文每个字符占用 3 个字节。

广义的 Unicode 指一个标准，定义字符集及编码规则，即 Unicode 字符集和 UTF-8、UTF-16 编码等。

### 11 Go语言数据类型转换

**（1）Go 语言使用类型前置加括号的方式进行类型转换，一般格式如下：**

T(表达式)

其中，T 代表要转换的类型。表达式包括变量、复杂算子和函数返回值等。

**（2）类型转换时，需要考虑两种类型的关系和范围，是否会发生数值截断等，参见下面代码：**

package main

import (

"fmt"

"math"

)

func main() {

// 输出各数值范围

fmt.Println("int8 range:", math.MinInt8, math.MaxInt8)

fmt.Println("int16 range:", math.MinInt16, math.MaxInt16)

fmt.Println("int32 range:", math.MinInt32, math.MaxInt32)

fmt.Println("int64 range:", math.MinInt64, math.MaxInt64)

// 初始化一个32位整型值

var a int32 = 1047483647

// 输出变量的十六进制形式和十进制值

fmt.Printf("int32: 0x%x %d\n", a, a)

// 将a变量数值转换为十六进制, 发生数值截断

b := int16(a)

// 输出变量的十六进制形式和十进制值

fmt.Printf("int16: 0x%x %d\n", b, b)

// 将常量保存为float32类型

var c float32 = math.Pi

// 转换为int类型, 浮点发生精度丢失

fmt.Println(int(c))

}

**（3）代码说明如下：**

第 11～14 行，输出常见整型类型的数值范围。

第 17 行，声明 int32 类型的 a 变量并初始化。

第 19 行，使用 fmt.Printf 的%x动词将数值以十六进制格式输出。这一行输出 a 在转换前的 32 位的值。

第 22 行，将 a 的值转换为 int16 类型，也就是从 32 位有符号整型转换为 16 位有符号整型。由于 16 位变量没有 32 位变量的数值范围大，因此数值会进行截断。

第 24 行，输出转换后的 a 变量值，也就是 b 的值。同样以十六进制和十进制两种方式进行打印。

第 27 行，math.Pi 是 math 包的常量，默认没有类型，会在引用到的地方自动根据实际类型进行推导。这里 math.Pi 被存到 c 中，类型为 float32。

第 29 行，将 float32 转换为 int 类型并输出。

代码输出如下：

int8 range: -128 127

int16 range: -32768 32767

int32 range: -2147483648 2147483647

int64 range: -9223372036854775808 9223372036854775807

int32: 0x3e6f54ff 1047483647

int16: 0x54ff 21759

3

根据输出结果，16 位有符号整型的范围是 -32768～32767，而 a 变量的 1047483647 不在这个范围内。1047483647 对应的十六进制为 0x3e6f54ff，转为 16 位变量后，长度缩短一半，也就是在十六进制上砍掉一半，变成 0x54ff，对应的十进制值为 21759。

浮点数在转换为整型时，会将小数部分去掉，只保留整数部分。

整型截断在类型转换中发生的较为隐性，有些即为难追查的问题，很小一部分是由整型截断造成。

### 12 Go语言指针

**（1）指针（pointer）概念在 Go 语言中被拆分为两个核心概念：**

类型指针，允许对这个指针类型的数据进行修改。传递数据使用指针，而无须拷贝数据。类型指针不能进行偏移和运算。

切片，由指向起始元素的原始指针、元素数量和容量组成。

受益于这样的约束和拆分，Go 语言的指针类型变量拥有指针的高效访问，但又不会发生指针偏移，从而避免非法修改关键性数据问题。同时，垃圾回收也比较容易对不会发生偏移的指针进行检索和回收。

切片比原始指针具备更强大的特性，更为安全。切片发生越界时，运行时会报出宕机，并打出堆栈，而原始指针只会崩溃。

C/C++中的指针

说到 C/C++ 中的指针，会让许多人“谈虎色变”，尤其对指针偏移、运算、转换都非常恐惧。

其实，指针是使 C/C++ 语言有极高性能的根本，在操作大块数据和做偏移时方便又便捷。因此，操作系统依然使用C语言及指针特性进行编写。

C/C++ 中指针饱受诟病的根本原因是指针运算和内存释放。

C/C++ 语言中的裸指针可以自由偏移，甚至可以在某些情况下偏移进入操作系统核心区域。我们的计算机操作系统经常需要更新、修复漏洞的本质，是为解决指针越界访问所导致的“缓冲区溢出”。

要明白指针，需要知道几个概念：指针地址、指针类型和指针取值，下面将展开细说。

**（2）认识指针地址和指针类型**

每个变量在运行时都拥有一个地址，这个地址代表变量在内存中的位置。Go 语言中使用&作符放在变量前面对变量进行“取地址”操作。

格式如下：

ptr := &v // v的类型为T

其中 v 代表被取地址的变量，被取地址的 v 使用 ptr 变量进行接收，ptr 的类型就为\*T，称做 T 的指针类型。\*代表指针。

指针实际用法，通过下面的例子了解：

package main

import (

"fmt"

)

func main() {

var cat int = 1

var str string = "banana"

fmt.Printf("%p %p", &cat, &str)

}

运行结果：

0xc042052088 0xc0420461b0

代码说明如下：

第 8 行，声明整型 cat 变量。

第 9 行，声明字符串 str 变量。

第 10 行，使用 fmt.Printf 的动词%p输出 cat 和 str 变量取地址后的指针值，指针值带有0x的十六进制前缀。

输出值在每次运行是不同的，代表 cat 和 str 两个变量在运行时的地址。

在 32 位平台上，将是 32 位地址；64 位平台上是 64 位地址。

提示：变量、指针和地址三者的关系是：每个变量都拥有地址，指针的值就是地址。

**（3）从指针获取指针指向的值**

在对普通变量使用&操作符取地址获得这个变量的指针后，可以对指针使用\*操作，也就是指针取值，代码如下。

package main

import (

"fmt"

)

func main() {

// 准备一个字符串类型

var house = "Malibu Point 10880, 90265"

// 对字符串取地址, ptr类型为\*string

ptr := &house

// 打印ptr的类型

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

// 打印ptr的指针地址

fmt.Printf("address: %p\n", ptr)

// 对指针进行取值操作

value := \*ptr

// 取值后的类型

fmt.Printf("value type: %T\n", value)

// 指针取值后就是指向变量的值

fmt.Printf("value: %s\n", value)

}

运行结果：

ptr type: \*string

address: 0xc0420401b0

value type: string

value: Malibu Point 10880, 90265

代码说明如下：

第 10 行，准备一个字符串并赋值。

第 13 行，对字符串取地址，将指针保存到 ptr 中。

第 16 行，打印 ptr 变量的类型，类型为 \*string。

第 19 行，打印 ptr 的指针地址，每次运行都会发生变化。

第 22 行，对 ptr 指针变量进行取值操作，value 变量类型为 string。

第 25 行，打印取值后 value 的类型。

第 28 行，打印 value 的值。

取地址操作符&和取值操作符\*是一对互补操作符，&取出地址，\*根据地址取出地址指向的值。变量、指针地址、指针变量、取地址、取值的相互关系和特性如下：

对变量进行取地址（&）操作，可以获得这个变量的指针变量。指针变量的值是指针地址。对指针变量进行取值（\*）操作，可以获得指针变量指向的原变量的值。

**（4）使用指针修改值**

通过指针不仅可以取值，也可以修改值。

前面已经使用多重赋值的方法进行数值交换，使用指针同样可以进行数值交换，代码如下：

package main

import "fmt"

// 交换函数

func swap(a, b \*int) {

// 取a指针的值, 赋给临时变量t

t := \*a

// 取b指针的值, 赋给a指针指向的变量

\*a = \*b

// 将a指针的值赋给b指针指向的变量

\*b = t

}

func main() {

// 准备两个变量, 赋值1和2

x, y := 1, 2

// 交换变量值

swap(&x, &y)

// 输出变量值

fmt.Println(x, y)

}

运行结果：

2 1

代码说明如下：

第 6 行，定义一个交换函数，参数为 a、b，类型都为 \*int，都是指针类型。

第 9 行，将 a 指针取值，把值（int类型）赋给 t 变量，t 此时也是 int 类型。

第 12 行，取 b 指针值，赋给 a 变量指向的变量。注意，此时\*a的意思不是取 a 指针的值，而是“a指向的变量”。

第 15 行，将 t 的值赋给 b 指向的变量。

第 21 行，准备 x、y 两个变量，赋值 1 和 2，类型为 int。

第 24 行，取出 x 和 y 的地址作为参数传给 swap() 函数进行调用。

第 27 行，交换完毕时，输出 x 和 y 的值。

\*操作符作为右值时，意义是取指针的值；作为左值时，也就是放在赋值操作符的左边时，表示 a 指向的变量。其实归纳起来，\*操作符的根本意义就是操作指针指向的变量。当操作在右值时，就是取指向变量的值；当操作在左值时，就是将值设置给指向的变量。

如果在 swap() 函数中交换操作的是指针值，会发生什么情况？可以参考下面代码：

package main

import "fmt"

func swap(a, b \*int) {

b, a = a, b

}

func main() {

x, y := 1, 2

swap(&x, &y)

fmt.Println(x, y)

}

运行结果：

1 2

结果表明，交换是不成功的。上面代码中的 swap() 函数交换的是 a 和 b 的地址，在交换完毕后，a 和 b 的变量值确实被交换。但和 a、b 关联的两个变量并没有实际关联。这就像写有两座房子的卡片放在桌上一字摊开，交换两座房子的卡片后并不会对两座房子有任何影响。

**（5）示例：使用指针变量获取命令行的输入信息**

Go 语言的 flag 包中，定义的指令以指针类型返回。通过学习 flag 包，可以深入了解指针变量在设计上的方便之处。

下面的代码通过提前定义一些命令行指令和对应变量，在运行时，输入对应参数的命令行参数后，经过 flag 包的解析后即可通过定义的变量获取命令行的数据。

获取命令行输入：

package main

// 导入系统包

import (

"flag"

"fmt"

)

// 定义命令行参数

var mode = flag.String("mode", "", "process mode")

func main() {

// 解析命令行参数

flag.Parse()

// 输出命令行参数

fmt.Println(\*mode)

}

将这段代码命名为main.go，然后使用如下命令行运行：

$ go run flagparse.go --mode=fast

命令行输出结果如下：

fast

代码说明如下：

第 10 行，通过 flag.String，定义一个 mode 变量，这个变量的类型是 \*string。后面 3 个参数分别如下：

参数名称：在给应用输入参数时，使用这个名称。

参数值的默认值：与 flag 所使用的函数创建变量类型对应，String 对应字符串、Int 对应整型、Bool 对应布尔型等。

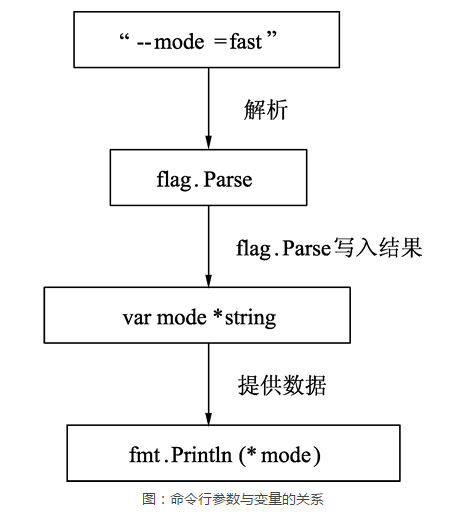
参数说明：使用 -help 时，会出现在说明中。

第 15 行，解析命令行参数，并将结果写入创建的指令变量中，这个例子中就是 mode 变量。

第 18 行，打印 mode 指针所指向的变量。

由于之前使用 flag.String 已经注册了一个 mode 的命令行参数，flag 底层知道怎么解析命令行，并且将值赋给 mode\*string 指针。在 Parse 调用完毕后，无须从 flag 获取值，而是通过自己注册的 mode 这个指针，获取到最终的值。代码运行流程如下图所示。

图：命令行参数与变量的关系



**（6）创建指针的另一种方法——new() 函数**

Go 语言还提供了另外一种方法来创建指针变量，格式如下：

new(类型)

一般这样写：

str := new(string)

\*str = "ninja"

fmt.Println(\*str)

new() 函数可以创建一个对应类型的指针，创建过程会分配内存。被创建的指针指向的值为默认值。

### 13 Go语言变量生命期，Go语言变量逃逸分析

讨论变量生命期之前，先来了解下计算机组成里两个非常重要的概念：堆和栈。

**（1）什么是栈**

栈（Stack）是一种拥有特殊规则的线性表数据结构。

**1) 概念**

栈只允许往线性表的一端放入数据，之后在这一端取出数据，按照后进先出（LIFO，Last InFirst Out）的顺序，如下图所示。往栈中放入元素的过程叫做入栈。入栈会增加栈的元素数量，最后放入的元素总是位于栈的顶部，最先放入的元素总是位于栈的底部。

从栈中取出元素时，只能从栈顶部取出。取出元素后，栈的数量会变少。最先放入的元素总是最后被取出，最后放入的元素总是最先被取出。不允许从栈底获取数据，也不允许对栈成员（除栈顶外的成员）进行任何查看和修改操作。

栈的原理类似于将书籍一本一本地堆起来。书按顺序一本一本从顶部放入，要取书时只能从顶部一本一本取出。

**2) 变量和栈有什么关系**

栈可用于内存分配，的分配和回收速度非常快。下面代码展示栈在内存分配上的作用，代码如下：

func calc(a, b int) int {

var c int

c = a \* b

var x int

x = c \* 10

return x

}

代码说明如下：

第 1 行，传入 a、b 两个整型参数。

第 2 行，声明 c 整型变量，运行时，c 会分配一段内存用以存储 c 的数值。

第 3 行，将 a 和 b 相乘后赋予 c。

第 5 行，声明 x 整型变量，x 也会被分配一段内存。

第 6 行，让 c 乘以 10 后存储到 x 变量中。

第 8 行，返回 x 的值。

上面的代码在没有任何优化情况下，会进行 c 和 x 变量的分配过程。Go 语言默认情况下会将 c 和 x 分配在栈上，这两个变量在 calc() 函数退出时就不再使用，函数结束时，保存 c 和 x 的栈内存再出栈释放内存，整个分配内存的过程通过栈的分配和回收都会非常迅速。

**（2）什么是堆**

堆在内存分配中类似于往一个房间里摆放各种家具，家具的尺寸有大有小。分配内存时，需要找一块足够装下家具的空间再摆放家具。经过反复摆放和腾空家具后，房间里的空间会变得乱七八糟，此时再往空间里摆放家具会存在虽然有足够的空间，但各空间分布在不同的区域，无法有一段连续的空间来摆放家具的问题。此时，内存分配器就需要对这些空间进行调整优化。堆分配内存和栈分配内存相比，堆适合不可预知大小的内存分配。但是为此付出的代价是分配速度较慢，而且会形成内存碎片。

**（3）变量逃逸（Escape Analysis）——自动决定变量分配方式，提高运行效率**

堆和栈各有优缺点，该怎么在编程中处理这个问题呢？在 C/C++ 语言中，需要开发者自己学习如何进行内存分配，选用怎样的内存分配方式来适应不同的算法需求。比如，函数局部变量尽量使用栈；全局变量、结构体成员使用堆分配等。程序员不得不花费很多年的时间在不同的项目中学习、记忆这些概念并加以实践和使用。

Go 语言将这个过程整合到编译器中，命名为“变量逃逸分析”。这个技术由编译器分析代码的特征和代码生命期，决定应该如何堆还是栈进行内存分配，即使程序员使用 Go 语言完成了整个工程后也不会感受到这个过程。

1) 逃逸分析

使用下面的代码来展现 Go 语言如何通过命令行分析变量逃逸，代码如下：

package main

import "fmt"

// 本函数测试入口参数和返回值情况

func dummy(b int) int {

// 声明一个c赋值进入参数并返回

var c int

c = b

return c

}

// 空函数, 什么也不做

func void() {

}

func main() {

// 声明a变量并打印

var a int

// 调用void()函数

void()

// 打印a变量的值和dummy()函数返回

fmt.Println(a, dummy(0))

}

代码说明如下：

第 6 行，dummy() 函数拥有一个参数，返回一个整型值，测试函数参数和返回值分析情况。

第 9 行，声明 c 变量，这里演示函数临时变量通过函数返回值返回后的情况。

第 16 行，这是一个空函数，测试没有任何参数函数的分析情况。

第 23 行，在 main() 中声明 a 变量，测试 main() 中变量的分析情况。

第 26 行，调用 void() 函数，没有返回值，测试 void() 调用后的分析情况。

第 29 行，打印 a 和 dummy(0) 的返回值，测试函数返回值没有变量接收时的分析情况。

接着使用如下命令行运行上面的代码：

$ go run -gcflags "-m -l" main.go

使用 go run 运行程序时，-gcflags 参数是编译参数。其中 -m 表示进行内存分配分析，-l 表示避免程序内联，也就是避免进行程序优化。

运行结果如下：

# command-line-arguments

./main.go:29:13: a escapes to heap

./main.go:29:22: dummy(0) escapes to heap

./main.go:29:13: main ... argument does not escape

0 0

程序运行结果分析如下：

输出第 2 行告知“main 的第 29 行的变量 a 逃逸到堆”。

第 3 行告知“dummy(0)调用逃逸到堆”。由于 dummy() 函数会返回一个整型值，这个值被 fmt.Println 使用后还是会在其声明后继续在 main() 函数中存在。

第 4 行，这句提示是默认的，可以忽略。

上面例子中变量 c 是整型，其值通过 dummy() 的返回值“逃出”了 dummy() 函数。c 变量值被复制并作为 dummy() 函数返回值返回，即使 c 变量在 dummy() 函数中分配的内存被释放，也不会影响 main() 中使用 dummy() 返回的值。c 变量使用栈分配不会影响结果。

**2) 取地址发生逃逸**

下面的例子使用结构体做数据，了解在堆上分配的情况，代码如下：

package main

import "fmt"

// 声明空结构体测试结构体逃逸情况

type Data struct {

}

func dummy() \*Data {

// 实例化c为Data类型

var c Data

//返回函数局部变量地址

return &c

}

func main() {

fmt.Println(dummy())

}

代码说明如下：

第 6 行，声明一个空的结构体做结构体逃逸分析。

第 9 行，将 dummy() 函数的返回值修改为 \*Data 指针类型。

第 12 行，将 c 变量声明为 Data 类型，此时 c 的结构体为值类型。

第 15 行，取函数局部变量 c 的地址并返回。Go 语言的特性允许这样做。

第 20 行，打印 dummy() 函数的返回值。

执行逃逸分析：

$ go run -gcflags "-m -l" main.go

# command-line-arguments

./main.go:15:9: &c escapes to heap

./main.go:12:6: moved to heap: c

./main.go:20:19: dummy() escapes to heap

./main.go:20:13: main ... argument does not escape

&{}

注意第 4 行出现了新的提示：将 c 移到堆中。这句话表示，Go 编译器已经确认如果将 c 变量分配在栈上是无法保证程序最终结果的。如果坚持这样做，dummy() 的返回值将是 Data 结构的一个不可预知的内存地址。这种情况一般是 C/C++ 语言中容易犯错的地方：引用了一个函数局部变量的地址。

Go 语言最终选择将 c 的 Data 结构分配在堆上。然后由垃圾回收器去回收 c 的内存。

**3) 原则**

在使用 Go 语言进行编程时，Go 语言的设计者不希望开发者将精力放在内存应该分配在栈还是堆上的问题。编译器会自动帮助开发者完成这个纠结的选择。但变量逃逸分析也是需要了解的一个编译器技术，这个技术不仅用于 Go 语言，在 Java 等语言的编译器优化上也使用了类似的技术。

编译器觉得变量应该分配在堆和栈上的原则是：

变量是否被取地址。

变量是否发生逃逸。

### 14 Go语言常量和const关键字

相对于变量，常量是恒定不变的值，例如圆周率。可以在编译时，对常量表达式进行计算求值，并在运行期使用该计算结果，计算结果无法被修改。常量表示起来非常简单，如下面的代码：

const pi = 3.141592

const e = 2.718281

常量的声明和变量声明非常类似，只是把 var 换成了 const。

多个变量可以一起声明，类似的，常量也是可以多个一起声明的，如下面的代码：

const (

pi = 3.141592

e = 2.718281

)

常量因为在编译期确定，所以可以用于数组声明，如下面的代码：

纯文本复制

const size = 4

var arr [size]int

### 15 Go语言type关键字（类型别名）

1. **注意：本节内容涉及 Go 语言新版本的功能。**

内容上会涉及后续章节讲解的类型定义及结构体嵌入等特性。另外，本节内容适用于对 Go 语言很熟悉且正在关注工程升级、代码重构等问题的读者阅读。类型别名是 Go 1.9 版本添加的新功能。主要用于代码升级、迁移中类型的兼容性问题。在 C/C++ 语言中，代码重构升级可以使用宏快速定义新的一段代码。Go 语言中没有选择加入宏，而是将解决重构中最麻烦的类型名变更问题。在 Go 1.9 版本之前的内建类型定义的代码是这样写的：

type byte uint8

type rune int32

而在 Go 1.9 版本之后变为：

type byte = uint8

type rune = int32

这个修改就是配合类型别名而进行的修改。

**（2）区分类型别名与类型定义**

类型别名的写法为：

type TypeAlias = Type

类型别名规定：TypeAlias 只是 Type 的别名，本质上 TypeAlias 与 Type 是同一个类型。就像一个孩子小时候有小名、乳名，上学后用学名，英语老师又会给他起英文名，但这些名字都指的是他本人。

类型别名与类型定义表面上看只有一个等号的差异，那么它们之间实际的区别有哪些呢？下面通过一段代码来理解。

package main

import (

"fmt"

)

// 将NewInt定义为int类型

type NewInt int

// 将int取一个别名叫IntAlias

type IntAlias = int

func main() {

// 将a声明为NewInt类型

var a NewInt

// 查看a的类型名

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

// 将a2声明为IntAlias类型

var a2 IntAlias

// 查看a2的类型名

fmt.Printf("a2 type: %T\n", a2)

}

代码运行结果：

a type: main.NewInt

a2 type: int

代码说明如下：

第 8 行，将 NewInt 定义为 int 类型，这是常见定义类型的方法，通过 type 关键字的定义，NewInt 会形成一种新的类型。NewInt 本身依然具备int的特性。

第 11 行，将 IntAlias 设置为 int 的一个别名，使用 IntAlias 与 int 等效。

第 16 行，将 a 声明为 NewInt 类型，此时若打印，则 a 的值为 0。

第 18 行，使用%T格式化参数，显示 a 变量本身的类型。

第 21 行，将 a2 声明为 IntAlias 类型，此时打印 a2 的值为 0。

第 23 行，显示 a2 变量的类型。

结果显示a的类型是 main.NewInt，表示 main 包下定义的 NewInt 类型。a2 类型是 int。IntAlias 类型只会在代码中存在，编译完成时，不会有 IntAlias 类型。

**（3）非本地类型不能定义方法**

能够随意地为各种类型起名字，是否意味着可以在自己包里为这些类型任意添加方法？参见下面的代码演示：

package main

import (

"time"

)

// 定义time.Duration的别名为MyDuration

type MyDuration = time.Duration

// 为MyDuration添加一个函数

func (m MyDuration) EasySet(a string) {

}

func main() {

}

代码说明如下：

第 8 行，使用类型别名为 time.Duration 设定一个别名叫 MyDuration。

第 11 行，为这个别名添加一个方法。

编译上面代码报错，信息如下：

cannot define new methods on non-local type time.Duration

编译器提示：不能在一个非本地的类型 time.Duration 上定义新方法。非本地方法指的就是使用 time.Duration 的代码所在的包，也就是 main 包。因为 time.Duration 是在 time 包中定义的，在 main 包中使用。time.Duration 包与 main 包不在同一个包中，因此不能为不在一个包中的类型定义方法。

**解决这个问题有下面两种方法：将第 8 行修改为 type MyDuration time.Duration，也就是将 MyDuration 从别名改为类型。将 MyDuration 的别名定义放在 time 包中。**

**（4）在结构体成员嵌入时使用别名**

当类型别名作为结构体嵌入的成员时会发生什么情况？请参考下面的代码。

package main

import (

"fmt"

"reflect"

)

// 定义商标结构

type Brand struct {

}

// 为商标结构添加Show()方法

func (t Brand) Show() {

}

// 为Brand定义一个别名FakeBrand

type FakeBrand = Brand

// 定义车辆结构

type Vehicle struct {

// 嵌入两个结构

FakeBrand

Brand

}

func main() {

// 声明变量a为车辆类型

var a Vehicle

// 指定调用FakeBrand的Show

a.FakeBrand.Show()

// 取a的类型反射对象

ta := reflect.TypeOf(a)

// 遍历a的所有成员

for i := 0; i < ta.NumField(); i++ {

// a的成员信息

f := ta.Field(i)

// 打印成员的字段名和类型

fmt.Printf("FieldName: %v, FieldType: %v\n", f.Name, f.Type.

Name())

}

}

代码输出如下：

FieldName: FakeBrand, FieldType: Brand

FieldName: Brand, FieldType: Brand

代码说明如下：

第 9 行，定义商标结构。

第 13 行，为商标结构添加 Show() 方法。

第 17 行，为 Brand 定义一个别名 FakeBrand。

第 20～25 行，定义车辆结构 Vehicle，嵌入 FakeBrand 和 Brand 结构。

第 30 行，将 Vechicle 实例化为 a。

第 33 行，显式调用 Vehicle 中 FakeBrand 的 Show() 方法。

第 36 行，使用反射取变量 a 的反射类型对象，以查看其成员类型。

第 39～42 行，遍历 a 的结构体成员。

第 45 行，打印 Vehicle 类型所有成员的信息。

这个例子中，FakeBrand 是 Brand 的一个别名。在 Vehicle 中嵌入 FakeBrand 和 Brand 并不意味着嵌入两个 Brand。FakeBrand 的类型会以名字的方式保留在 Vehicle 的成员中。

如果尝试将第 33 行改为：

a.Show()

编译器将发生报错：

ambiguous selector a.Show

在调用 Show() 方法时，因为两个类型都有 Show() 方法，会发生歧义，证明 FakeBrand 的本质确实是 Brand 类型。

## [Go语言容器](http://c.biancheng.net/golang/container/)

### 1 Go语言数组详解

**（1）数组（Array）是一段固定长度的连续内存区域。**

在 Go 语言中，数组从声明时就确定，使用时可以修改数组成员，但是数组大小不可变化。

**（2）C/C++ 中的数组**

C语言和 Go 语言中的数组概念完全一致。C语言的数组也是一段固定长度的内存区域，数组的大小在声明时固定下来。下面演示一段C语言的数组：

int a[10]={ 0,1,2,3,4,5,6,7,8,9 };

int b[4];

此时，a 和 b 类型都是 int\*，也就是整型指针。而C语言中，也可以使用 malloc() 函数动态地分配一段内存区域。C++ 语言中可以使用 new() 函数。例如：

int\* a = (int\*)malloc(10);

int\* b = new int(4);

此时，a 和 b 的类型也是 int\*。a 和 b 此时分配内存的方式类似于 Go 语言的切片。

Go 的数组和切片都是从C语言延续过来的设计。

**（3）Go 语言数组的声明**

数组的写法如下：

var 数组变量名 [元素数量]T

其中：

数组变量名：数组声明及使用时的变量名。

元素数量：数组的元素数量。可以是一个表达式，但最终通过编译期计算的结果必须是整型数值。也就是说，元素数量不能含有到运行时才能确认大小的数值。

T 可以是任意基本类型，包括 T 为数组本身。但类型为数组本身时，可以实现多维数组。

下面是一段数组的演示例子：

var team [3]string

team[0] = "hammer"

team[1] = "soldier"

team[2] = "mum"

fmt.Println(team)

输出结果：

[hammer soldier mum]

代码说明如下：

第 1 行，将 team 声明为包含 3 个元素的字符串数组。

第 2～4 行，为 team 的元素赋值。

**（4）Go语言数组的初始化**

数组可以在声明时使用初始化列表进行元素设置，参考下面的代码：

var team = [3]string{"hammer", "soldier", "mum"}

这种方式编写时，需要保证大括号后面的元素数量与数组的大小一致。但一般情况下，这个过程可以交给编译器，让编译器在编译时，根据元素个数确定数组大小。

var team = [...]string{"hammer", "soldier", "mum"}

...表示让编译器确定数组大小。上面例子中，编译器会自动为这个数组设置元素个数为 3。

（5）遍历数组——访问每一个数组元素

遍历数组也和遍历切片类似，看下面代码：

var team [3]string

team[0] = "hammer"

team[1] = "soldier"

team[2] = "mum"

for k, v := range team {

fmt.Println(k, v)

}

代码输出结果：

hammer

soldier

mum

代码说明如下：

第 6 行，使用 for 循环，遍历 team 数组，遍历出的键 k 为数组的索引，值 v 为数组的每个元素值。

第 7 行，将每个键值打印出来。

### 2 Go语言切片详解

**（1）切片（Slice）是一个拥有相同类型元素的可变长度的序列。**Go 语言切片的内部结构包含地址、大小和容量。切片一般用于快速地操作一块数据集合。如果将数据集合比作切糕的话，切片就是你要的“那一块”。切的过程包含从哪里开始（这个就是切片的地址）及切多大（这个就是切片的大小）。容量可以理解为装切片的口袋大小，如下图所示。

图：切片结构和内存分配

**（2）从数组或切片生成新的切片**

切片默认指向一段连续内存区域，可以是数组，也可以是切片本身。

从连续内存区域生成切片是常见的操作，格式如下：

slice [开始位置:结束位置]

slice 表示目标切片对象。

开始位置对应目标切片对象的索引。

结束位置对应目标切片的结束索引。

从数组生成切片，代码如下：

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

fmt.Println(a, a[1:2])

a 是一个拥有 3 个整型元素的数组，被初始化数值 1 到 3。使用 a[1:2] 可以生成一个新的切片。代码运行结果如下：

[1 2 3] [2]

[2] 就是 a[1:2] 切片操作的结果。

从数组或切片生成新的切片拥有如下特性：

取出的元素数量为：结束位置-开始位置。

取出元素不包含结束位置对应的索引，切片最后一个元素使用 slice[len(slice)] 获取。

当缺省开始位置时，表示从连续区域开头到结束位置。

当缺省结束位置时，表示从开始位置到整个连续区域末尾。

两者同时缺省时，与切片本身等效。

两者同时为0时，等效于空切片，一般用于切片复位。

根据索引位置取切片 slice 元素值时，取值范围是（0～len(slice)-1），超界会报运行时错误。生成切片时，结束位置可以填写 len(slice) 但不会报错。

下面在具体的例子中熟悉切片的特性。

**1) 从指定范围中生成切片**

切片和数组密不可分。如果将数组理解为一栋办公楼，那么切片就是把不同的连续楼层出租给使用者。出租的过程需要选择开始楼层和结束楼层，这个过程就会生成切片。示例代码如下：

var highRiseBuilding [30]int

for i := 0; i < 30; i++ {

highRiseBuilding[i] = i + 1

}

// 区间

fmt.Println(highRiseBuilding[10:15])

// 中间到尾部的所有元素

fmt.Println(highRiseBuilding[20:])

// 开头到中间的所有元素

fmt.Println(highRiseBuilding[:2])

代码输出如下：

[11 12 13 14 15]

[21 22 23 24 25 26 27 28 29 30]

[1 2]

代码中构建了一个 30 层的高层建筑。数组的元素值从 1 到 30，分别代表不同的独立楼层。输出的结果是不同租售方案。

代码说明如下：

第 8 行，尝试出租一个区间楼层。

第 11 行，出租 20 层以上。

第 14 行，出租 2 层以下，一般是商用铺面。

切片有点像C语言里的指针。指针可以做运算，但代价是内存操作越界。切片在指针的基础上增加了大小，约束了切片对应的内存区域，切片使用中无法对切片内部的地址和大小进行手动调整，因此切片比指针更安全、强大。

**2) 表示原有的切片**

生成切片的格式中，当开始和结束都范围都被忽略，则生成的切片将表示和原切片一致的切片，并且生成的切片与原切片在数据内容上是一致的，代码如下：

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

fmt.Println(a[:])

a 是一个拥有 3 个元素的切片。将 a 切片使用 a[:] 进行操作后，得到的切片与 a 切片一致，代码输出如下：

[1 2 3]

**3) 重置切片，清空拥有的元素**

把切片的开始和结束位置都设为 0 时，生成的切片将变空，代码如下：

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

fmt.Println(a[0:0])

代码输出如下：

[]

**（3）直接声明新的切片**

除了可以从原有的数组或者切片中生成切片，你也可以声明一个新的切片。每一种类型都可以拥有其切片类型，表示多个类型元素的连续集合。因此切片类型也可以被声明。切片类型声明格式如下：

var name []T

name 表示切片类型的变量名。

T 表示切片类型对应的元素类型。

下面代码展示了切片声明的使用过程：

// 声明字符串切片

var strList []string

// 声明整型切片

var numList []int

// 声明一个空切片

var numListEmpty = []int{}

// 输出3个切片

fmt.Println(strList, numList, numListEmpty)

// 输出3个切片大小

fmt.Println(len(strList), len(numList), len(numListEmpty))

// 切片判定空的结果

fmt.Println(strList == nil)

fmt.Println(numList == nil)

fmt.Println(numListEmpty == nil)

代码输出结果：

[] [] []

0 0 0

true

true

false

代码说明如下：

第 2 行，声明一个字符串切片，切片中拥有多个字符串。

第 5 行，声明一个整型切片，切片中拥有多个整型数值。

第 8 行，将 numListEmpty 声明为一个整型切片。本来会在{}中填充切片的初始化元素，这里没有填充，所以切片是空的。但此时 numListEmpty 已经被分配了内存，但没有元素。

第 11 行，切片均没有任何元素，3 个切片输出元素内容均为空。

第 14 行，没有对切片进行任何操作，strList 和 numList 没有指向任何数组或者其他切片。

第 17 行和第 18 行，声明但未使用的切片的默认值是 nil。strList 和 numList 也是 nil，所以和 nil 比较的结果是 true。

第 19 行，numListEmpty 已经被分配到了内存，但没有元素，因此和 nil 比较时是 false。

切片是动态结构，只能与nil判定相等，不能互相判等时。

声明新的切片后，可以使用 append() 函数来添加元素。

**（4）使用 make() 函数构造切片**

如果需要动态地创建一个切片，可以使用 make() 内建函数，格式如下：

make( []T, size, cap )

T：切片的元素类型。

size：就是为这个类型分配多少个元素。

cap：预分配的元素数量，这个值设定后不影响 size，只是能提前分配空间，降低多次分配空间造成的性能问题。

示例如下：

a := make([]int, 2)

b := make([]int, 2, 10)

fmt.Println(a, b)

fmt.Println(len(a), len(b))

代码输出如下：

[0 0] [0 0]

2 2

a 和 b 均是预分配 2 个元素的切片，只是 b 的内部存储空间已经分配了 10 个，但实际使用了 2 个元素。

容量不会影响当前的元素个数，因此 a 和 b 取 len 都是 2。

温馨提示

使用 make() 函数生成的切片一定发生了内存分配操作。但给定开始与结束位置（包括切片复位）的切片只是将新的切片结构指向已经分配好的内存区域，设定开始与结束位置，不会发生内存分配操作。

切片不一定必须经过 make() 函数才能使用。生成切片、声明后使用 append() 函数均可以正常使用切片。

### 3 Go语言append()为切片添加元素

**（1）Go 语言的内建函数 append() 可以为切片动态添加元素。**每个切片会指向一片内存空间，这片空间能容纳一定数量的元素。当空间不能容纳足够多的元素时，切片就会进行“扩容”。“扩容”操作往往发生在 append() 函数调用时。

切片在扩容时，容量的扩展规律按容量的 2 倍数扩充，例如 1、2、4、8、16……，代码如下：

var numbers []int

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

numbers = append(numbers, i)

fmt.Printf("len: %d cap: %d pointer: %p\n", len(numbers), cap(numbers), numbers)

}

代码输出如下：

len: 1 cap: 1 pointer: 0xc0420080e8

len: 2 cap: 2 pointer: 0xc042008150

len: 3 cap: 4 pointer: 0xc04200e320

len: 4 cap: 4 pointer: 0xc04200e320

len: 5 cap: 8 pointer: 0xc04200c200

len: 6 cap: 8 pointer: 0xc04200c200

len: 7 cap: 8 pointer: 0xc04200c200

len: 8 cap: 8 pointer: 0xc04200c200

len: 9 cap: 16 pointer: 0xc042074000

len: 10 cap: 16 pointer: 0xc042074000

**（2）代码说明如下：**

第 1 行，声明一个整型切片。

第 4 行，循环向 numbers 切片添加10个数。

第 5 行中，打印输出切片的长度、容量和指针变化。使用 len() 函数查看切片拥有的元素个数，使用 cap() 函数查看切片的容量情况。

通过查看代码输出，有一个有意思的规律：len() 函数并不等于 cap。

往一个切片中不断添加元素的过程，类似于公司搬家。公司发展初期，资金紧张，人员很少，所以只需要很小的房间即可容纳所有的员工。随着业务的拓展和收入的增加就需要扩充工位，但是办公地的大小是固定的，无法改变。因此公司选择搬家，每次搬家就需要将所有的人员转移到新的办公点。

员工和工位就是切片中的元素。

办公地就是分配好的内存。

搬家就是重新分配内存。

无论搬多少次家，公司名称始终不会变，代表外部使用切片的变量名不会修改。

因为搬家后地址发生变化，因此内存“地址”也会有修改。

**（3）append() 函数除了添加一个元素外，也可以一次性添加很多元素。**

var car []string

// 添加1个元素

car = append(car, "OldDriver")

// 添加多个元素

car = append(car, "Ice", "Sniper", "Monk")

// 添加切片

team := []string{"Pig", "Flyingcake", "Chicken"}

car = append(car, team...)

fmt.Println(car)

代码输出如下：

[OldDriver Ice Sniper Monk Pig Flyingcake Chicken]

代码说明如下：

第 1 行，声明一个字符串切片。

第 4 行，往切片中添加一个元素。

第 7 行，使用 append() 函数向切片中添加多个元素。

第 10 行，声明另外一个字符串切片

第 11 行，在team后面加上了...，表示将 team 整个添加到 car 的后面。

### 4 Go语言copy()：切片复制（切片拷贝）

使用 Go 语言内建的 copy() 函数，可以迅速地将一个切片的数据复制到另外一个切片空间中，copy() 函数的使用格式如下：

copy( destSlice, srcSlice []T) int

srcSlice 为数据来源切片。

destSlice 为复制的目标。目标切片必须分配过空间且足够承载复制的元素个数。来源和目标的类型一致，copy 的返回值表示实际发生复制的元素个数。

下面的代码将演示对切片的引用和复制操作后对切片元素的影响。

package main

import "fmt"

func main() {

// 设置元素数量为1000

const elementCount = 1000

// 预分配足够多的元素切片

srcData := make([]int, elementCount)

// 将切片赋值

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

srcData[i] = i

}

// 引用切片数据

refData := srcData

// 预分配足够多的元素切片

copyData := make([]int, elementCount)

// 将数据复制到新的切片空间中

copy(copyData, srcData)

// 修改原始数据的第一个元素

srcData[0] = 999

// 打印引用切片的第一个元素

fmt.Println(refData[0])

// 打印复制切片的第一个和最后一个元素

fmt.Println(copyData[0], copyData[elementCount-1])

// 复制原始数据从4到6(不包含)

copy(copyData, srcData[4:6])

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

fmt.Printf("%d ", copyData[i])

}

}

代码说明如下：

第 8 行，定义元素总量为 1000。

第 11 行，预分配拥有 1000 个元素的整型切片，这个切片将作为原始数据。

第 14～16 行，将 srcData 填充 0～999 的整型值。

第 19 行，将 refData 引用 srcData，切片不会因为等号操作进行元素的复制。

第 22 行，预分配与 srcData 等大（大小相等）、同类型的切片 copyData。

第 24 行，使用 copy() 函数将原始数据复制到 copyData 切片空间中。

第 27 行，修改原始数据的第一个元素为 999。

第 30 行，引用数据的第一个元素将会发生变化。

第 33 行，打印复制数据的首位数据，由于数据是复制的，因此不会发生变化。

第 36 行，将 srcData 的局部数据复制到 copyData 中。

第 38～40 行，打印复制局部数据后的 copyData 元素。

### 5 Go语言从切片中删除元素

Go 语言并没有对删除切片元素提供专用的语法或者接口，需要使用切片本身的特性来删除元素。示例代码如下：

seq := []string{"a", "b", "c", "d", "e"}

// 指定删除位置

index := 2

// 查看删除位置之前的元素和之后的元素

fmt.Println(seq[:index], seq[index+1:])

// 将删除点前后的元素连接起来

seq = append(seq[:index], seq[index+1:]...)

fmt.Println(seq)

代码输出结果：

[a b] [d e]

[a b d e]

第 1 行，声明一个整型切片，保存含有从 a 到 e 的字符串。

第 4 行，为了演示和讲解方便，使用 index 变量保存需要删除的元素位置。

第 7 行中：seq[:index] 表示的就是被删除元素的前半部分，值为：

[1 2]

seq[index+1:] 表示的是被删除元素的后半部分，值为：

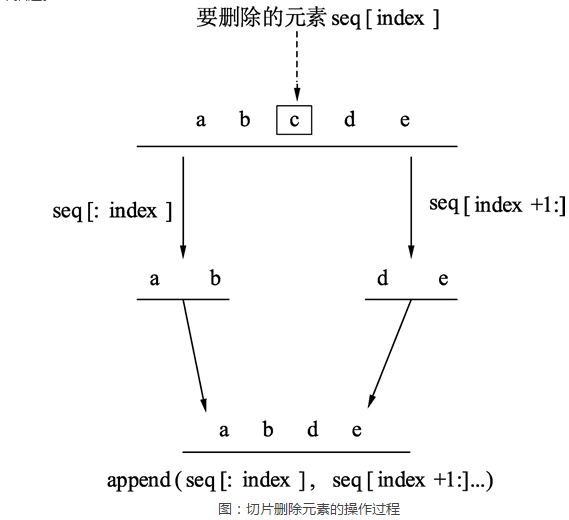
[4 5]

第 10 行使用 append() 函数将两个切片连接起来。

第 12 行，输出连接好的新切片。此时，索引为 2 的元素已经被删除。

代码的删除过程可以使用下图来描述。

图：切片删除元素的操作过程



Go 语言中切片删除元素的本质是：以被删除元素为分界点，将前后两个部分的内存重新连接起来。

提示

Go 语言中切片元素的删除过程并没有提供任何的语法糖或者方法封装，无论是初学者学习，还是实际使用都是极为麻烦的。

连续容器的元素删除无论是在任何语言中，都要将删除点前后的元素移动到新的位置。随着元素的增加，这个过程将会变得极为耗时。因此，当业务需要大量、频繁地从一个切片中删除元素时，如果对性能要求较高，就需要反思是否需要更换其他的容器（如双链表等能快速从删除点删除元素）。

### 6 Go语言map（Go语言映射）

**（1）在业务和算法中需要使用任意类型的关联关系时，就需要使用到映射，如学号和学生的对应、名字与档案的对应等。**

Go 语言提供的映射关系容器为 map，map使用散列表（hash）实现。

提示

大多数语言中映射关系容器使用两种算法：散列表和平衡树。

散列表可以简单描述为一个数组（俗称“桶”），数组的每个元素是一个列表。根据散列函数获得每个元素的特征值，将特征值作为映射的键。如果特征值重复，表示元素发生碰撞。碰撞的元素将被放在同一个特征值的列表中进行保存。散列表查找复杂度为 O(1)，和数组一致。最坏的情况为 O(n)，n 为元素总数。散列需要尽量避免元素碰撞以提高查找效率，这样就需要对“桶”进行扩容，每次扩容，元素需要重新放入桶中，较为耗时。

平衡树类似于有父子关系的一棵数据树，每个元素在放入树时，都要与一些节点进行比较。平衡树的查找复杂度始终为 O(log n)。

**（2）添加关联到 map 并访问关联和数据**

Go 语言中 map 的定义是这样的：

map[KeyType]ValueType

KeyType为键类型。

ValueType是键对应的值类型。

一个 map 里，符合 KeyType 和 ValueType 的映射总是成对出现。

下面代码展示了 map 的基本使用环境。

scene := make(map[string]int)

scene["route"] = 66

fmt.Println(scene["route"])

v := scene["route2"]

fmt.Println(v)

代码输出如下：

66

0

**（3）代码说明如下：**

第 1 行 map 是一个内部实现的类型，使用时，需要手动使用 make 创建。如果不创建使用 map 类型，会触发宕机错误。

第 3 行向 map 中加入映射关系。写法与使用数组一样，key 可以使用除函数以外的任意类型。

第 5 行查找 map 中的值。

第 7 行中，尝试查找一个不存在的键，那么返回的将是 ValueType 的默认值。

某些情况下，需要明确知道查询中某个键是否在 map 中存在，可以使用一种特殊的写法来实现，看下面的代码：

v, ok := scene["route"]

在默认获取键值的基础上，多取了一个变量 ok，可以判断键 route 是否存在于 map 中。

map 还有一种在声明时填充内容的方式，代码如下：

纯文本复制

m := map[string]string{

"W": "forward",

"A": "left",

"D": "right",

"S": "backward",

}

例子中并没有使用 make，而是使用大括号进行内容定义，就像 JSON 格式一样，冒号的左边是 key，右边是值，键值对之间使用逗号分隔。

### 7 Go语言遍历map（访问map中的每一个键值对）

**（1）map 的遍历过程使用 for range 循环完成，代码如下：**

scene := make(map[string]int)

scene["route"] = 66

scene["brazil"] = 4

scene["china"] = 960

for k, v := range scene {

fmt.Println(k, v)

}

遍历对于 Go 语言的很多对象来说都是差不多的，直接使用 for range 语法。遍历时，可以同时获得键和值。如只遍历值，可以使用下面的形式：

for \_, v := range scene {

将不需要的键改为匿名变量形式。

只遍历键时，使用下面的形式：

for k := range scene {

无须将值改为匿名变量形式，忽略值即可。

注意：遍历输出元素的顺序与填充顺序无关。不能期望 map 在遍历时返回某种期望顺序的结果。

**（2）如果需要特定顺序的遍历结果，正确的做法是排序，代码如下：**

scene := make(map[string]int)

// 准备map数据

scene["route"] = 66

scene["brazil"] = 4

scene["china"] = 960

// 声明一个切片保存map数据

var sceneList []string

// 将map数据遍历复制到切片中

for k := range scene {

sceneList = append(sceneList, k)

}

// 对切片进行排序

sort.Strings(sceneList)

// 输出

fmt.Println(sceneList)

代码输出如下：

[brazil china route]

代码说明如下：

第 1 行，创建一个 map 实例，键为字符串，值为整型。

第 4～6 行，将 3 个键值对写入 map 中。

第 9 行，声明 sceneList 为字符串切片，以缓冲和排序 map 中的所有元素。

第 12 行，将 map 中元素的键遍历出来，并放入切片中。

第 17 行，对 sceneList 字符串切片进行排序。排序时，sceneList 会被修改。

第 20 行，输出排好序的 map 的键。

sort.Strings 的作用是对传入的字符串切片进行字符串字符的升序排列。排序接口的使用将在后面的章节中介绍。

### 8 Go语言map元素的删除和清空

使用 delete() 函数从 map 中删除键值对

**（1）使用 delete() 内建函数从 map 中删除一组键值对，delete() 函数的格式如下：**

delete(map, 键)

map 为要删除的 map 实例。

键为要删除的 map 键值对中的键。

从 map 中删除一组键值对可以通过下面的代码来完成：

scene := make(map[string]int)

// 准备map数据

scene["route"] = 66

scene["brazil"] = 4

scene["china"] = 960

delete(scene, "brazil")

for k, v := range scene {

fmt.Println(k, v)

}

代码输出如下：

route 66

china 960

这个例子中使用 delete() 函数将 brazil 从 scene 这个 map 中删除了。

**（2）清空 map 中的所有元素**

有意思的是，Go 语言中并没有为 map 提供任何清空所有元素的函数、方法。清空 map 的唯一办法就是重新 make 一个新的 map。不用担心垃圾回收的效率，Go 语言中的并行垃圾回收效率比写一个清空函数高效多了。

### 9 Go语言sync.Map（在并发环境中使用的map）

**（1）Go 语言中的 map 在并发情况下，只读是线程安全的，同时读写线程不安全。**

下面来看下并发情况下读写 map 时会出现的问题，代码如下：

// 创建一个int到int的映射

m := make(map[int]int)

// 开启一段并发代码

go func() {

// 不停地对map进行写入

for {

m[1] = 1

}

}()

// 开启一段并发代码

go func() {

// 不停地对map进行读取

for {

\_ = m[1]

}

}()

// 无限循环, 让并发程序在后台执行

for {

}

运行代码会报错，输出如下：

fatal error: concurrent map read and map write

**（2）运行时输出提示：并发的 map 读写。**也就是说使用了两个并发函数不断地对 map 进行读和写而发生了竞态问题。map 内部会对这种并发操作进行检查并提前发现。

需要并发读写时，一般的做法是加锁，但这样性能并不高。Go 语言在 1.9 版本中提供了一种效率较高的并发安全的 sync.Map。sync.Map 和 map 不同，不是以语言原生形态提供，而是在 sync 包下的特殊结构。

sync.Map有以下特性：

无须初始化，直接声明即可。

sync.Map 不能使用 map 的方式进行取值和设置等操作，而是使用 sync.Map 的方法进行调用。Store 表示存储，Load 表示获取，Delete 表示删除。

使用 Range 配合一个回调函数进行遍历操作，通过回调函数返回内部遍历出来的值。Range 参数中的回调函数的返回值功能是：需要继续迭代遍历时，返回 true；终止迭代遍历时，返回 false。

并发安全的 sync.Map 演示代码如下：

package main

import (

"fmt"

"sync"

)

func main() {

var scene sync.Map

// 将键值对保存到sync.Map

scene.Store("greece", 97)

scene.Store("london", 100)

scene.Store("egypt", 200)

// 从sync.Map中根据键取值

fmt.Println(scene.Load("london"))

// 根据键删除对应的键值对

scene.Delete("london")

// 遍历所有sync.Map中的键值对

scene.Range(func(k, v interface{}) bool {

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

return true

})

}

代码输出如下：

100 true

iterate: egypt 200

iterate: greece 97

代码说明如下：

第 10 行，声明 scene，类型为 sync.Map。注意，sync.Map 不能使用 make 创建。

第 13～15 行，将一系列键值对保存到 sync.Map 中，sync.Map 将键和值以 interface{} 类型进行保存。

第 18 行，提供一个 sync.Map 的键给 scene.Load() 方法后将查询到键对应的值返回。

第 21 行，sync.Map 的 Delete 可以使用指定的键将对应的键值对删除。

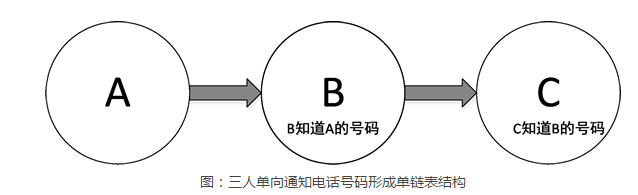
第 24 行，Range() 方法可以遍历 sync.Map，遍历需要提供一个匿名函数，参数为 k、v，类型为 interface{}，每次 Range() 在遍历一个元素时，都会调用这个匿名函数把结果返回。

sync.Map 没有提供获取 map 数量的方法，替代方法是获取时遍历自行计算数量。sync.Map 为了保证并发安全有一些性能损失，因此在非并发情况下，使用 map 相比使用 sync.Map 会有更好的性能。

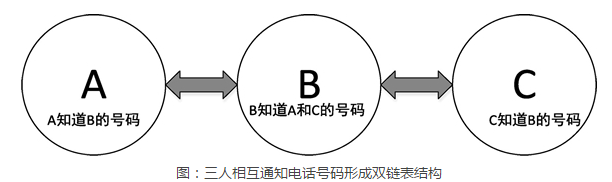
### 10 Go语言list（列表）

**（1）列表是一种非连续存储的容器，由多个节点组成，节点通过一些变量记录彼此之间的关系。列表有多种实现方法，如单链表、双链表等**。

列表的原理可以这样理解：假设 A、B、C 三个人都有电话号码，如果 A 把号码告诉给 B，B 把号码告诉给 C，这个过程就建立了一个单链表结构，如下图所示。

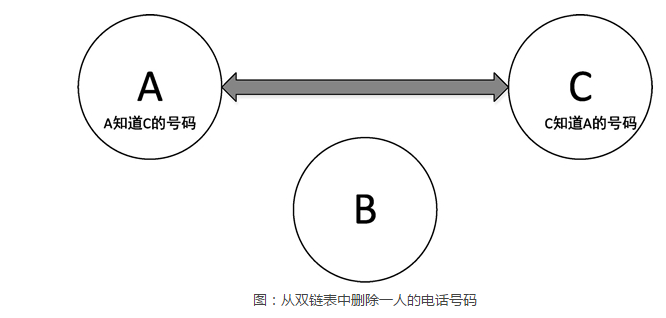


如果在这个基础上，再从 C 开始将自己的号码给自己知道号码的人，这样就形成了双链表结构，如下图所示。



那么如果需要获得所有人的号码，只需要从 A 或者 C 开始，要求他们将自己的号码发出来，然后再通知下一个人如此循环。这个过程就是列表遍历。

如果 B 换号码了，他需要通知 A 和 C，将自己的号码移除。这个过程就是列表元素的删除操作，如下图所示。

在 Go 语言中，将列表使用 container/list 包来实现，内部的实现原理是双链表。列表能够高效地进行任意位置的元素插入和删除操作。

**（2）初始化列表**

list 的初始化有两种方法：New 和声明。两种方法的初始化效果都是一致的。

**1) 通过 container/list 包的 New 方法初始化 list**

变量名 := list.New()

**2) 通过声明初始化list**

var 变量名 list.List

列表与切片和 map 不同的是，列表并没有具体元素类型的限制。因此，列表的元素可以是任意类型。这既带来遍历，也会引来一些问题。给一个列表放入了非期望类型的值，在取出值后，将 interface{} 转换为期望类型时将会发生宕机。

**（3）在列表中插入元素**

双链表支持从队列前方或后方插入元素，分别对应的方法是 PushFront 和 PushBack。

这两个方法都会返回一个 \*list.Element 结构。如果在以后的使用中需要删除插入的元素，则只能通过 \*list.Element 配合 Remove() 方法进行删除，这种方法可以让删除更加效率化，也是双链表特性之一。

下面代码展示如何给list添加元素：

l := list.New()

l.PushBack("fist")

l.PushFront(67)

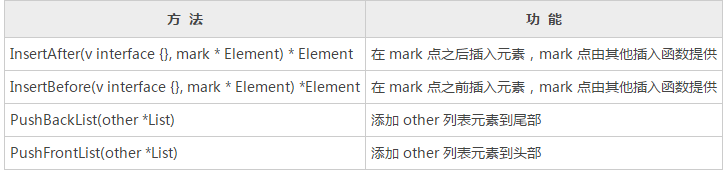
代码说明如下：

第 1 行，创建一个列表实例。

第 3 行，将 fist 字符串插入到列表的尾部，此时列表是空的，插入后只有一个元素。

第 4 行，将数值 67 放入列表。此时，列表中已经存在 fist 元素，67 这个元素将被放在 fist 的前面。

列表插入元素的方法如下表所示。



**（4）从列表中删除元素**

列表的插入函数的返回值会提供一个 \*list.Element 结构，这个结构记录着列表元素的值及和其他节点之间的关系等信息。从列表中删除元素时，需要用到这个结构进行快速删除。

列表操作元素：

package main

import "container/list"

func main() {

l := list.New()

// 尾部添加

l.PushBack("canon")

// 头部添加

l.PushFront(67)

// 尾部添加后保存元素句柄

element := l.PushBack("fist")

// 在fist之后添加high

l.InsertAfter("high", element)

// 在fist之前添加noon

l.InsertBefore("noon", element)

// 使用

l.Remove(element)

}

代码说明如下：

第 6 行，创建列表实例。

第 9 行，将 canon 字符串插入到列表的尾部。

第 12 行，将 67 数值添加到列表的头部。

第 15 行，将 fist 字符串插入到列表的尾部，并将这个元素的内部结构保存到 element 变量中。

第 18 行，使用 element 变量，在 element 的位置后面插入 high 字符串。

第 21 行，使用 element 变量，在 element 的位置前面插入 noon 字符串。

第 24 行，移除 element 变量对应的元素。

下表中展示了每次操作后列表的实际元素情况。

列表元素操作的过程



**（5）遍历列表——访问列表的每一个元素**

遍历双链表需要配合 Front() 函数获取头元素，遍历时只要元素不为空就可以继续进行。每一次遍历调用元素的 Next，如代码中第 9 行所示。

l := list.New()

// 尾部添加

l.PushBack("canon")

// 头部添加

l.PushFront(67)

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

fmt.Println(i.Value)

}

代码输出如下：

67

canon

代码说明如下：

第 1 行，创建一个列表实例。

第 4 行，将 canon 放入列表尾部。

第 7 行，在队列头部放入 67。

第 9 行，使用 for 语句进行遍历，其中 i:=l.Front() 表示初始赋值，只会在一开始执行一次；每次循环会进行一次 i!=nil 语句判断，如果返回 false，表示退出循环，反之则会执行 i=i.Next()。

第 10 行，使用遍历返回的 \*list.Element 的 Value 成员取得放入列表时的原值。

## 四、Go[流程控制](http://c.biancheng.net/golang/flow_control/)

### 1 Go语言if else（分支结构）

**（1）在 Go 语言中，可以通过 if 关键字进行条件判断，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 的结果为 true 时，执行分支 1，否则判断表达式 2，如果满足则执行分支 2，都不满足时，则执行分支 3。表达式 2、分支 2 和分支 3 都是可选的，可以根据实际需要进行选择。

Go 语言规定与 if 匹配的左括号{必须与 if 和表达式放在同一行，如果尝试将{放在其他位置，将会触发编译错误。

同理，与 else 匹配的{也必须与 else 在同一行，else 也必须与上一个 if 或 else if 的右边的大括号在一行。

**（2）举例**

通过下面的例子来了解 if 的写法：

var ten int = 11

if ten > 10 {

fmt.Println(">10")

} else {

fmt.Println("<=10")

}

代码输出如下：

>10

代码说明如下：

第 1 行，声明整型变量并赋值 11。

第 2 行，判断当 ten 的值大于 10 时执行第 3 行，否则执行第 4 行。

第 3 和第 5 行，分别打印大于 10 和小于等于 10 时的输出。

**（3）特殊写法**

if 还有一种特殊的写法，可以在 if 表达式之前添加一个执行语句，再根据变量值进行判断，代码如下：

if err := Connect(); err != nil {

fmt.Println(err)

return

}

Connect 是一个带有返回值的函数，err:=Connect() 是一个语句，执行 Connect 后，将错误保存到 err 变量中。

err！=nil 才是 if 的判断表达式，当 err 不为空时，打印错误并返回。

这种写法可以将返回值与判断放在一行进行处理，而且返回值的作用范围被限制在 if、else 语句组合中。在编程中，变量在其实现了变量的功能后，作用范围越小，所造成的问题可能性越小，每一个变量代表一个状态，有状态的地方，状态就会被修改，函数的局部变量只会影响一个函数的执行，但全局变量可能会影响所有代码的执行状态，因此限制变量的作用范围对代码的稳定性有很大的帮助。

### 2 Go语言for（循环结构）

**（1）Go 语言中的所有循环类型均可以使用 for 关键字来完成。**

基于语句和表达式的基本 for 循环格式如下：

for 初始语句;条件表达式;结束语句{

循环体代码

}

循环体不停地进行循环，直到条件表达式返回 false 时自动退出循环，执行 for 的}之后的语句。

for 循环可以通过 break、goto、return、panic 语句强制退出循环。for 的初始语句、条件表达式、结束语句的详细介绍如下。

**（2）for 中的初始语句——开始循环时执行的语句**

初始语句是在第一次循环前执行的语句，一般使用初始语句执行变量初始化，如果变量在此处被声明，其作用域将被局限在这个 for 的范畴内。

初始语句可以被忽略，但是初始语句之后的分号必须要写，代码如下：

step := 2

for ; step > 0; step-- {

fmt.Println(step)

}

这段代码将 step 放在 for 的前面进行初始化，for 中没有初始语句，此时 step 的作用域就比在初始语句中声明 step 要大。

**（3）for 中的条件表达式——控制是否循环的开关**

对每次循环开始前计算的表达式，如果表达式为 true，则循环继续，否则结束循环。条件表达式可以被忽略，被忽略条件的表达式默认形成无限循环。

**1) 结束循环时带可执行语句的无限循环**

下面代码忽略条件表达式，但是保留结束语句，代码如下：

var i int

for ; ; i++ {

if i > 10 {

break

}

}

代码说明如下：

第 3 行，无须设置 i 的初始值，因此忽略 for 的初始语句。两个分号之间是条件表达式，也被忽略，此时循环会一直持续下去；for 的结束语句为 i++，每次结束循环前都会调用。

第 5 行，判断 i 大于 10 时，通过 break 语句跳出 for 循环到第 9 行。

**2) 无限循环**

上面的代码还可以改写为更美观的写法，代码如下：

var i int

for {

if i > 10 {

break

}

i++

}

代码说明如下：

第 3 行，忽略 for 的所有语句，此时 for 执行无限循环。

第 9 行，将 i++ 从 for 的结束语句放置到函数体的末尾是等效的，这样编写的代码更具有可读性。

无限循环在收发处理中较为常见，但需要无限循环有可控的退出方式来结束循环。

**3) 只有一个循环条件的循环**

在上面代码的基础上进一步简化代码，将 if 判断整合到 for 中，变为下面的代码：

var i int

for i <= 10 {

i++

}

在代码第 3 行中，将之前使用 if i>10{} 判断的表达式进行取反，变为判断 i 小于等于 10 时持续进行循环。

上面这段代码其实类似于其他编程语言中的 while，在 while 后添加一个条件表达式，满足条件表达式时持续循环，否则结束循环。

**（4）for 中的结束语句——每次循环结束时执行的语句**

在结束每次循环前执行的语句，如果循环被 break、goto、return、panic 等语句强制退出，结束语句不会被执行。

### 3 Go语言输出九九乘法表

熟悉了 Go 语言的基本循环格式后，让我们用一个例子来温习一遍吧。

输出九九乘法表：

package main

import "fmt"

func main() {

// 遍历, 决定处理第几行

for y := 1; y <= 9; y++ {

// 遍历, 决定这一行有多少列

for x := 1; x <= y; x++ {

fmt.Printf("%d\*%d=%d ", x, y, x\*y)

}

// 手动生成回车

fmt.Println()

}

}

结果输出如下：

1\*1=1

1\*2=2 2\*2=4

1\*3=3 2\*3=6 3\*3=9

1\*4=4 2\*4=8 3\*4=12 4\*4=16

1\*5=5 2\*5=10 3\*5=15 4\*5=20 5\*5=25

1\*6=6 2\*6=12 3\*6=18 4\*6=24 5\*6=30 6\*6=36

1\*7=7 2\*7=14 3\*7=21 4\*7=28 5\*7=35 6\*7=42 7\*7=49

1\*8=8 2\*8=16 3\*8=24 4\*8=32 5\*8=40 6\*8=48 7\*8=56 8\*8=64

1\*9=9 2\*9=18 3\*9=27 4\*9=36 5\*9=45 6\*9=54 7\*9=63 8\*9=72 9\*9=81

代码说明如下：

第 8 行，生成 1～9 的数字，对应乘法表的每一行，也就是被乘数。

第 11 行，乘法表每一行中的列数随着行数的增加而增加，这一行的 x 表示该行有多少列。

第 12 行，打印一个空行，实际作用就是换行。

这段程序按行优先打印，打印完一行，换行（第12行），接着执行下一行乘法表直到整个数值循环完毕。

### 4 Go语言for range（键值循环）

Go 语言可以使用 for range 遍历数组、切片、字符串、map 及通道（channel）。通过 for range 遍历的返回值有一定的规律：

数组、切片、字符串返回索引和值。map 返回键和值。通道（channel）只返回通道内的值。

**（1）遍历数组、切片——获得索引和元素**

在遍历代码中，key 和 value 分别代表切片的下标及下标对应的值。下面的代码展示如何遍历切片，数组也是类似的遍历方法：

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

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

}

代码输出如下：

key:0 value:1

key:1 value:2

key:2 value:3

key:3 value:4

**（2）遍历字符串——获得字符**

Go 语言和其他语言类似，可以通过 for range 的组合，对字符串进行遍历，遍历时，key 和 value 分别代表字符串的索引（base0）和字符串中的每一个字符。

下面这段代码展示了如何遍历字符串：

var str = "hello 你好"

for key, value := range str {

fmt.Printf("key:%d value:0x%x\n", key, value)

}

代码输出如下：

key:0 value:0x68

key:1 value:0x65

key:2 value:0x6c

key:3 value:0x6c

key:4 value:0x6f

key:5 value:0x20

key:6 value:0x4f60

key:9 value:0x597d

代码中的 v 变量，实际类型是 rune，实际上就是 int32，以十六进制打印出来就是字符的编码。

**（3）历map——获得map的键和值**

对于 map 类型来说，for range 遍历时，key 和 value 分别代表 map 的索引键 key 和索引对应的值，一般被称为 map 的键值对，因为它们总是一对一对的出现。下面的代码演示了如何遍历 map。

m := map[string]int{

"hello": 100,

"world": 200,

}

for key, value := range m {

fmt.Println(key, value)

}

代码输出如下：

hello 100

world 200

注意

对 map 遍历时，遍历输出的键值是无序的，如果需要有序的键值对输出，需要对结果进行排序。

**（4）遍历通道（channel）——接收通道数据**

for range 可以遍历通道（channel），但是通道在遍历时，只输出一个值，即管道内的类型对应的数据。

下面代码为我们展示了通道的遍历：

c := make(chan int)

go func() {

c <- 1

c <- 2

c <- 3

close(c)

}()

for v := range c {

fmt.Println(v)

}

代码说明如下：

第 1 行创建了一个整型类型的通道。

第 3 行启动了一个 goroutine，其逻辑的实现体现在第 5～8 行，实现功能是往通道中推送数据 1、2、3，然后结束并关闭通道。

这段 goroutine 在声明结束后，在第 9 行马上被并行执行。

从第 11 行开始，使用 for range 对通道 c 进行遍历，其实就是不断地从通道中取数据，直到通道被关闭。

**（5）在遍历中选择希望获得的变量**

在使用 for range 循环遍历某个对象时，一般不会同时需要 key 或者 value，这个时候可以采用一些技巧，让代码变得更简单。下面将前面的例子修改一下，参考下面的代码示例：

m := map[string]int{

"hello": 100,

"world": 200,

}

for \_, value := range m {

fmt.Println(value)

}

代码输出如下：

100

200

在例子中将 key 变成了下画线，那么这里的下画线就是匿名变量。什么是匿名变量？

可以理解为一种占位符。

本身这种变量不会进行空间分配，也不会占用一个变量的名字。

在 for range 可以对 key 使用匿名变量，也可以对 value 使用匿名变量。

再看一个匿名变量的例子：

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

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

}

代码输出如下：

key:0

key:1

key:2

key:3

在该例子中，value 被设置为匿名变量，只使用 key，而 key 本身就是切片的索引，所以例子输出索引。

我们总结一下for的功能：

Go 语言的 for 包含初始化语句、条件表达式、结束语句，这 3 个部分均可缺省。

for range 支持对数组、切片、字符串、map、通道进行遍历操作。

在需要时，可以使用匿名变量对 for range 的变量进行选取。

### 5 Go语言switch case语句

分支选择可以理解为一种批量的if语句，使用 switch 语句可方便地对大量的值进行判断。

在 Go 语言中的 switch，不仅可以基于常量进行判断，还可以基于表达式进行判断

C/C++ 语言中的 switch 语句只能支持数值常量，不能对字符串、表达式等复杂情况进行处理，这么设计的主要原因是性能。C/C++ 的 switch 可以根据 case 的值作为偏移量直接跳转代码，在性能敏感代码处，这样做显然是有好处的。

到了 Go 语言的时代，语言的运行效率并不能直接决定最终的效率，I/O 效率现在是最主要的问题。因此，Go 语言中的 switch 语法设计尽量以使用方便为主。

**（1）基本写法**

Go 语言改进了 switch 的语法设计，避免人为造成失误。Go 语言的 switch 中的每一个 case 与 case 间是独立的代码块，不需要通过 break 语句跳出当前 case 代码块以避免执行到下一行。示例代码如下：

var a = "hello"

switch a {

case "hello":

fmt.Println(1)

case "world":

fmt.Println(2)

default:

fmt.Println(0)

}

代码输出如下：

1

上面例子中，每一个 case 均是字符串格式，且使用了 default 分支，Go 语言规定每个 switch 只能有一个 default 分支。

1) 一分支多值

当出现多个 case 要放在一起的时候，可以像下面代码这样写：

var a = "mum"

switch a {

case "mum", "daddy":

fmt.Println("family")

}

不同的 case 表达式使用逗号分隔。

2) 分支表达式

case 后不仅仅只是常量，还可以和 if 一样添加表达式，代码如下：

var r int = 11

switch {

case r > 10 && r < 20:

fmt.Println(r)

}

注意，这种情况的 switch 后面不再跟判断变量，连判断的目标都没有了。

**（2）跨越case的fallthrough——兼容C语言的case设计**

在 Go 语言中 case 是一个独立的代码块，执行完毕后不会像C语言那样紧接着下一个 case 执行。但是为了兼容一些移植代码，依然加入了 fallthrough 关键字来实现这一功能，代码如下：

var s = "hello"

switch {

case s == "hello":

fmt.Println("hello")

fallthrough

case s != "world":

fmt.Println("world")

}

代码输出如下：

hello

world

### 6 Go语言goto语句——跳转到指定的标签

goto 语句通过标签进行代码间的无条件跳转。goto 语句可以在快速跳出循环、避免重复退出上有一定的帮助。Go 语言中使用 goto 语句能简化一些代码的实现过程。

**（1）使用goto退出多层循环**

下面这段代码在满足条件时，需要连续退出两层循环，使用传统的编码方式如下：

package main

import "fmt"

func main() {

var breakAgain bool

// 外循环

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

// 内循环

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

// 满足某个条件时, 退出循环

if y == 2 {

// 设置退出标记

breakAgain = true

// 退出本次循环

break

}

}

// 根据标记, 还需要退出一次循环

if breakAgain {

break

}

}

fmt.Println("done")

}

代码说明如下：

第 10 行，构建外循环。

第 13 行，构建内循环。

第 16 行，当 y==2 时需要退出所有的 for 循环。

第 19 行，默认情况下循环只能一层一层退出，为此就需要设置一个状态变量 isbreak，需要退出时，设置这个变量为 true。

第 22 行，使用 break 退出当前循环，执行后，代码调转到第 28 行。

第 28 行，退出一层循环后，根据 isbreak 变量判断是否需要再次退出外层循环。

第 34 行，退出所有循环后，打印 done。

**（2）使用 goto 集中处理错误**

将上面的代码使用 Go 语言的 goto 语句进行优化。

goto 跳出循环：

package main

import "fmt"

func main() {

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

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

if y == 2 {

// 跳转到标签

goto breakHere

}

}

}

// 手动返回, 避免执行进入标签

return

// 标签

breakHere:

fmt.Println("done")

}

代码说明如下：

第 13 行，使用 goto 语句跳转到指明的标签处，标签在第 23 行定义。

第 20 行，标签只能被 goto 使用，但不影响代码执行流程，此处如果不手动返回，在不满足条件时，也会执行第 24 行代码。

第 23 行，定义 breakHere 标签。

使用 goto 语句后，无须额外的变量就可以快速退出所有的循环。

**（3）统一错误处理**

多处错误处理存在代码重复时是非常棘手的，例如：

err := firstCheckError()

if err != nil {

fmt.Println(err)

exitProcess()

return

}

err = secondCheckError()

if err != nil {

fmt.Println(err)

exitProcess()

return

}

fmt.Println("done")

代码说明如下：

第 1 行，执行某逻辑，返回错误。

第 2～6 行，如果发生错误，打印错误退出进程。

第 8 行，执行某逻辑，返回错误。

第 10～14 行，发生错误后退出流程。

第 16 行，没有任何错误，打印完成。

在上面代码中，加粗部分都是重复的错误处理代码。后期陆续在这些代码中如果添加更多的判断，就需要在每一块雷同代码中依次修改，极易造成疏忽和错误。

如果使用 goto 语句来实现同样的逻辑：

err := firstCheckError()

if err != nil {

goto onExit

}

err = secondCheckError()

if err != nil {

goto onExit

}

fmt.Println("done")

return

onExit:

fmt.Println(err)

exitProcess()

代码说明如下：

第 3 行和第 9 行，发生错误时，跳转错误标签 onExit。

第 17 行和第 18 行，汇总所有流程进行错误打印并退出进程。

### 7 Go语言break（跳出循环）

break 语句可以结束 for、switch 和 select 的代码块。break 语句还可以在语句后面添加标签，表示退出某个标签对应的代码块，标签要求必须定义在对应的 for、switch 和 select 的代码块上。

跳出指定循环：

package main

import "fmt"

func main() {

OuterLoop:

for i := 0; i < 2; i++ {

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

switch j {

case 2:

fmt.Println(i, j)

break OuterLoop

case 3:

fmt.Println(i, j)

break OuterLoop

}

}

}

}

代码输出如下：

0 2

代码说明如下：

第 7 行，外层循环的标签。

第 8 行和第 9 行，双层循环。

第 10 行，使用 switch 进行数值分支判断。

第 13 和第 16 行，退出 OuterLoop 对应的循环之外，也就是跳转到第 20 行。

### 8 Go语言continue（继续下一次循环）

continue 语句可以结束当前循环，开始下一次的循环迭代过程，仅限在 for 循环内使用。在 continue 语句后添加标签时，表示开始标签对应的循环。例如：

package main

import "fmt"

func main() {

OuterLoop:

for i := 0; i < 2; i++ {

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

switch j {

case 2:

fmt.Println(i, j)

continue OuterLoop

}

}

}

}

代码输出结果如下：

0 2

1 2

代码说明：第 14 行将结束当前循环，开启下一次的外层循环，而不是第 10 行的循环。

## Go[语言函数](http://c.biancheng.net/golang/func/)

### 1 Go语言函数声明（函数定义）

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

**（1）普通函数的声明形式**

Go 语言的函数声明以 func 标识，后面紧接着函数名、参数列表、返回参数列表及函数体，具体形式如下：

func 函数名(参数列表)(返回参数列表){

函数体

}

下面是对各个部分的说明：

函数名：由字母、数字、下画线组成。其中，函数名的第一个字母不能为数字。在同一个包内，函数名称不能重名。

包（package）是 Go 源码的一种组织方式，一个包可以认为是一个文件夹，在后续章节中将会详细讲解包的概念。

参数列表：一个参数由参数变量和参数类型组成，例如：

func foo( a int, b string )

其中，参数列表中的变量作为函数的局部变量而存在。

返回参数列表：可以是返回值类型列表，也可以是类似参数列表中变量名和类型名的组合。函数在声明有返回值时，必须在函数体中使用 return 语句提供返回值列表。

函数体：能够被重复调用的代码片段。

**（2）参数类型的简写**

在参数列表中，如有多个参数变量，则以逗号分隔；如果相邻变量是同类型，则可以将类型省略。例如：

func add(a, b int) int {

return a + b

}

以上代码中，a 和 b 的参数均为 int 类型，因此可以省略 a 的类型，在 b 后面有类型说明，这个类型也是 a 的类型。

**（3）函数的返回值**

Go 语言支持多返回值，多返回值能方便地获得函数执行后的多个返回参数，Go 语言经常使用多返回值中的最后一个返回参数返回函数执行中可能发生的错误。示例代码如下：

conn, err := connectToNetwork()

在这段代码中，connectToNetwork 返回两个参数，conn 表示连接对象，err 返回错误。

***1) 同一种类型返回值***

如果返回值是同一种类型，则用括号将多个返回值类型括起来，用逗号分隔每个返回值的类型。

使用 return 语句返回时，值列表的顺序需要与函数声明的返回值类型一致。示例代码如下：

func typedTwoValues() (int, int) {

return 1, 2

}

a, b := typedTwoValues()

fmt.Println(a, b)

代码输出结果：

1 2

纯类型的返回值对于代码可读性不是很友好，特别是在同类型的返回值出现时，无法区分每个返回参数的意义。

***2) 带有变量名的返回值***

Go 语言支持对返回值进行命名，这样返回值就和参数一样拥有参数变量名和类型。

命名的返回值变量的默认值为类型的默认值，即数值为 0，字符串为空字符串，布尔为 false、指针为 nil 等。

下面代码中的函数拥有两个整型返回值，函数声明时将返回值命名为 a 和 b，因此可以在函数体中直接对函数返回值进行赋值。在命名的返回值方式的函数体中，在函数结束前需要显式地使用 return 语句进行返回，代码如下：

func namedRetValues() (a, b int) {

a = 1

b = 2

return

}

代码说明如下：

第 1 行，对两个整型返回值进行命名，分别为 a 和 b。

第 3 行和第 4 行，命名返回值的变量与这个函数的布局变量的效果一致，可以对返回值进行赋值和值获取。

第 6 行，当函数使用命名返回值时，可以在 return 中不填写返回值列表，如果填写也是可行的。下面代码的执行效果和上面代码的效果一样。

func namedRetValues() (a, b int) {

a = 1

return a, 2

}

同一种类型返回值和命名返回值两种形式只能二选一，混用时将会发生编译错误，例如下面的代码：

func namedRetValues() (a, b int, int)

编译报错提示：

mixed named and unnamed function parameters

意思是：在函数参数中混合使用了命名和非命名参数。

**（3）调用函数**

函数在定义后，可以通过调用的方式，让当前代码跳转到被调用的函数中进行执行。调用前的函数局部变量都会被保存起来不会丢失；被调用的函数结束后，恢复到被调用函数的下一行继续执行代码，之前的局部变量也能继续访问。

函数内的局部变量只能在函数体中使用，函数调用结束后，这些局部变量都会被释放并且失效。

Go语言的函数调用格式如下：

返回值变量列表 = 函数名(参数列表)

下面是对各个部分的说明：

函数名：需要调用的函数名。

参数列表：参数变量以逗号分隔，尾部无须以分号结尾。

返回值变量列表：多个返回值使用逗号分隔。

例如，加法函数调用样式如下：

result := add(1,1)

### 2 Go语言函数变量——把函数作为值保存到变量中

在 Go 语言中，函数也是一种类型，可以和其他类型一样被保存在变量中。下面的代码定义了一个函数变量 f，并将一个函数名 fire() 赋给函数变量 f，这样调用函数变量 f 时，实际调用的就是 fire() 函数，代码如下：

package main

import (

"fmt"

)

func fire() {

fmt.Println("fire")

}

func main() {

var f func()

f = fire

f()

}

代码输出结果：

fire

代码说明：

第 7 行，定义了一个 fire() 函数。

第 13 行，将变量f声明为 func() 类型，此时 f 就被俗称为“回调函数”。此时 f 的值为 nil。

第 15 行，将 fire() 函数名作为值，赋给 f 变量，此时 f 的值为 fire() 函数。

第 17 行，使用 f 变量进行函数调用，实际调用的是 fire() 函数。

### 3 Go语言字符串的链式处理——操作与数据分离的设计技巧

使用 SQL 语言从数据库中获取数据时，可以对原始数据进行排序（sort by）、分组（group by）和去重（distinct）等操作。SQL 将数据的操作与遍历过程作为两个部分进行隔离，这样操作和遍历过程就可以各自独立地进行设计，这就是常见的数据与操作分离的设计。

对数据的操作进行多步骤的处理被称为链式处理。本例中使用多个字符串作为数据集合，然后对每个字符串进行一系列的处理，用户可以通过系统函数或者自定义函数对链式处理中的每个环节进行自定义。

首先给出本节完整代码：

package main

import (

"fmt"

"strings"

)

// 字符串处理函数，传入字符串切片和处理链

func StringProccess(list []string, chain []func(string) string) {

// 遍历每一个字符串

for index, str := range list {

// 第一个需要处理的字符串

result := str

// 遍历每一个处理链

for \_, proc := range chain {

// 输入一个字符串进行处理，返回数据作为下一个处理链的输入。

result = proc(result)

}

// 将结果放回切片

list[index] = result

}

}

// 自定义的移除前缀的处理函数

func removePrefix(str string) string {

return strings.TrimPrefix(str, "go")

}

func main() {

// 待处理的字符串列表

list := []string{

"go scanner",

"go parser",

"go compiler",

"go printer",

"go formater",

}

// 处理函数链

chain := []func(string) string{

removePrefix,

strings.TrimSpace,

strings.ToUpper,

}

// 处理字符串

StringProccess(list, chain)

// 输出处理好的字符串

for \_, str := range list {

fmt.Println(str)

}

}

**（1）字符串处理函数**

字符串处理函数（StringProccess）需要外部提供数据源：一个字符串切片（list[]string），另外还要提供一个链式处理函数的切片（chain[]func(string)string），链式处理切片中的一个处理函数的定义如下：

func(string)string

这种处理函数能够接受一个字符串输入，处理后输出。

strings 包中将字符串变为小写就是一种处理函数的形式，strings.ToLower() 函数能够将传入的字符串的每一个字符变为小写，strings.ToLower 定义如下：

func ToLower(s string) string

字符串处理函数（StringProccess）内部遍历每一个数据源提供的字符串，每个字符串都需要经过一系列链式处理函数处理后被重新放回切片，参见下面代码。

字符串的链式处理：

// 字符串处理函数, 传入字符串切片和处理链

func StringProccess(list []string, chain []func(string) string) {

// 遍历每一个字符串

for index, str := range list {

// 第一个需要处理的字符串

result := str

// 遍历每一个处理链

for \_, proc := range chain {

// 输入一个字符串进行处理, 返回数据作为下一个处理链的输入

result = proc(result)

}

// 将结果放回切片

list[index] = result

}

}

代码说明如下：

第 2 行，传入字符串切片 list 作为数据源，一系列的处理函数作为 chain 处理链。

第 5 行，遍历字符串切片的每个字符串，依次对每个字符串进行处理。

第 8 行，将当前字符串保存到 result 变量中，作为第一个处理函数的参数。

第 11 行，遍历每一个处理函数，将字符串按顺序经过这些处理函数处理。

第 14 行，result 变量即是每个处理函数的输入变量，处理后的变量又会重新保存到 result 变量中。

第 18 行，将处理完的字符串保存回切片中。

**（2）自定义的处理函数**

处理函数可以是系统提供的处理函数，如将字符串变大写或小写，也可以使用自定义函数。本例中的字符串处理的逻辑是使用一个自定义的函数实现移除指定go前缀的过程，参见下面代码：

// 自定义的移除前缀的处理函数

func removePrefix(str string) string {

return strings.TrimPrefix(str, "go")

}

此函数使用了 strings.TrimPrefix() 函数实现移除字符串的指定前缀。处理后，移除前缀的字符串结果将通过 removePrefix() 函数的返回值返回。

**（3）字符串处理主流程**

字符串处理的主流程包含以下几个步骤：

准备要处理的字符串列表。

准备字符串处理链。

处理字符串列表。

打印输出后的字符串列表。

详细流程参考下面的代码：

func main() {

// 待处理的字符串列表

list := []string{

"go scanner",

"go parser",

"go compiler",

"go printer",

"go formater",

}

// 处理函数链

chain := []func(string) string{

removePrefix,

strings.TrimSpace,

strings.ToUpper,

}

// 处理字符串

StringProccess(list, chain)

// 输出处理好的字符串

for \_, str := range list {

fmt.Println(str)

}

}

代码说明如下：

第 4 行，定义字符串切片，字符串包含 go 前缀及空格。

第 13 行，准备处理每个字符串的处理链，处理的顺序与函数在切片中的位置一致。removePrefix() 为自定义的函数，功能是移除 go 前缀；移除前缀的字符串左边有一个空格，使用 strings.TrimSpace 移除，这个函数的定义刚好符合处理函数的格式：func(string)string；strings.ToUpper 用于将字符串转为大写。

第 20 行，传入字符串切片和字符串处理链，通过 StringProcess() 函数对字符串进行处理。

第 23 行，遍历字符串切片的每一个字符串，打印处理好的字符串结果。

### 4 Go语言函数类型实现接口——把函数作为接口来调用

函数和其他类型一样都属于“一等公民”，其他类型能够实现接口，函数也可以，本节将分别对比结构体与函数实现接口的过程。

**（1）首先给出本节完整的代码：**

package main

import (

"fmt"

)

// 调用器接口

type Invoker interface {

// 需要实现一个Call方法

Call(interface{})

}

// 结构体类型

type Struct struct {

}

// 实现Invoker的Call

func (s \*Struct) Call(p interface{}) {

fmt.Println("from struct", p)

}

// 函数定义为类型

type FuncCaller func(interface{})

// 实现Invoker的Call

func (f FuncCaller) Call(p interface{}) {

// 调用f函数本体

f(p)

}

func main() {

// 声明接口变量

var invoker Invoker

// 实例化结构体

s := new(Struct)

// 将实例化的结构体赋值到接口

invoker = s

// 使用接口调用实例化结构体的方法Struct.Call

invoker.Call("hello")

// 将匿名函数转为FuncCaller类型，再赋值给接口

invoker = FuncCaller(func(v interface{}) {

fmt.Println("from function", v)

})

// 使用接口调用FuncCaller.Call，内部会调用函数本体

invoker.Call("hello")

}

有如下一个接口：

// 调用器接口

type Invoker interface {

// 需要实现一个Call()方法

Call(interface{})

}

这个接口需要实现 Call() 方法，调用时会传入一个 interface{} 类型的变量，这种类型的变量表示任意类型的值。

**（2）结构体实现接口**

结构体实现 Invoker 接口的代码如下：

// 结构体类型

type Struct struct {

}

// 实现Invoker的Call

func (s \*Struct) Call(p interface{}) {

fmt.Println("from struct", p)

}

代码说明如下：

第 2 行，定义结构体，该例子中的结构体无须任何成员，主要展示实现 Invoker 的方法。

第 6 行，Call() 为结构体的方法，该方法的功能是打印 from struct 和传入的 interface{} 类型的值。

将定义的 Struct 类型实例化，并传入接口中进行调用，代码如下：

// 声明接口变量

var invoker Invoker

// 实例化结构体

s := new(Struct)

// 将实例化的结构体赋值到接口

invoker = s

// 使用接口调用实例化结构体的方法Struct.Call

invoker.Call("hello")

代码说明如下：

第 2 行，声明 Invoker 类型的变量。

第 5 行，使用 new 将结构体实例化，此行也可以写为 s:=&Struct。

第 8 行，s 类型为 \*Struct，已经实现了 Invoker 接口类型，因此赋值给 invoker 时是成功的。

第 11 行，通过接口的 Call() 方法，传入 hello，此时将调用 Struct 结构体的 Call() 方法。

接下来，对比下函数实现结构体的差异。

代码输出如下：

from struct hello

**（3）函数体实现接口**

函数的声明不能直接实现接口，需要将函数定义为类型后，使用类型实现结构体。当类型方法被调用时，还需要调用函数本体。

// 函数定义为类型

type FuncCaller func(interface{})

// 实现Invoker的Call

func (f FuncCaller) Call(p interface{}) {

// 调用f()函数本体

f(p)

}

代码说明如下：

第 2 行，将 func(interface{}) 定义为 FuncCaller 类型。

第 5 行，FuncCaller 的 Call() 方法将实现 Invoker 的 Call() 方法。

第 8 行，FuncCaller 的 Call() 方法被调用与 func(interface{}) 无关，还需要手动调用函数本体。

上面代码只是定义了函数类型，需要函数本身进行逻辑处理。FuncCaller 无须被实例化，只需要将函数转换为 FuncCaller 类型即可，函数来源可以是命名函数、匿名函数或闭包，参见下面代码：

// 声明接口变量

var invoker Invoker

// 将匿名函数转为FuncCaller类型, 再赋值给接口

invoker = FuncCaller(func(v interface{}) {

fmt.Println("from function", v)

})

// 使用接口调用FuncCaller.Call, 内部会调用函数本体

invoker.Call("hello")

代码说明如下：

第 2 行，声明接口变量。

第 5 行，将 func(v interface{}){} 匿名函数转换为 FuncCaller 类型（函数签名才能转换），此时 FuncCaller 类型实现了 Invoker 的 Call() 方法，赋值给 invoker 接口是成功的。

第 10 行，使用接口方法调用。

代码输出如下：

from function hello

**（4）HTTP包中的例子**

HTTP 包中包含有 Handler 接口定义，代码如下：

type Handler interface {

ServeHTTP(ResponseWriter, \*Request)

}

Handler 用于定义每个 HTTP 的请求和响应的处理过程。

同时，也可以使用处理函数实现接口，定义如下：

type HandlerFunc func(ResponseWriter, \*Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r \*Request) {

f(w, r)

}

要使用闭包实现默认的 HTTP 请求处理，可以使用 http.HandleFunc() 函数，函数定义如下：

func HandleFunc(pattern string, handler func(ResponseWriter, \*Request)) {

DefaultServeMux.HandleFunc(pattern, handler)

}

而 DefaultServeMux 是 ServeMux 结构，拥有 HandleFunc() 方法，定义如下：

func (mux \*ServeMux) HandleFunc(pattern string, handler func

(ResponseWriter, \*Request)) {

mux.Handle(pattern, HandlerFunc(handler))

}

上面代码将外部传入的函数 handler() 转为 HandlerFunc 类型，HandlerFunc 类型实现了 Handler 的 ServeHTTP 方法，底层可以同时使用各种类型来实现 Handler 接口进行处理。

### 5 Go语言可变参数（变参函数）

所谓可变参数，是指参数数量不固定的函数形式。Go 语言支持可变参数特性，函数声明和调用时没有固定数量的参数，同时也提供了一套方法进行可变参数的多级传递。

Go 语言的可变参数格式如下：

func 函数名(固定参数列表, v … T)(返回参数列表){

函数体

}

以下是对可变参数的函数的说明：

可变参数一般被放置在函数列表的末尾，前面是固定参数列表，当没有固定参数时，所有变量就将是可变参数。

v 为可变参数变量，类型为 []T，也就是拥有多个 T 元素的 T 类型切片，v 和 T 之间由...即3个点组成。

T 为可变参数的类型，当 T 为 interface{} 时，传入的可以是任意类型。

fmt包中的例子

可变参数有两种形式：所有参数都是可变参数的形式，如 fmt.Println，以及部分是可变参数的形式，如 fmt.Printf，可变参数只能出现在参数的后半部分，因此不可变的参数只能放在参数的前半部分。

**1) 所有参数都是可变参数：fmt.Println**

fmt.Println的函数声明如下：

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

return Fprintln(os.Stdout, a...)

}

fmt.Println 在使用时，传入的值类型不受限制，例如：

fmt.Println(5, "hello", &struct{ a int }{1}, true)

**2) 部分参数是可变参数：fmt.Printf**

fmt.Printf 的第一个参数为参数列表，后面的参数是可变参数，fmt.Printf 函数的格式如下：

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

return Fprintf(os.Stdout, format, a...)

}

fmt.Printf() 函数在调用时，第一个函数始终必须传入字符串，对应参数是 format，后面的参数数量可以变化，使用时，代码如下：

纯文本复制

fmt.Printf("pure string\n")

fmt.Printf("value: %v %f\n", true, math.Pi)

### 6 Go语言defer（延迟执行语句）

Go 语言的 defer 语句会将其后面跟随的语句进行延迟处理。在 defer 归属的函数即将返回时，将延迟处理的语句按 defer 的逆序进行执行，也就是说，先被 defer 的语句最后被执行，最后被 defer 的语句，最先被执行。

多个延迟执行语句的处理顺序

下面的代码是将一系列的数值打印语句按顺序延迟处理，参见演示代码：

package main

import (

"fmt"

)

func main() {

fmt.Println("defer begin")

// 将defer放入延迟调用栈

defer fmt.Println(1)

defer fmt.Println(2)

// 最后一个放入, 位于栈顶, 最先调用

defer fmt.Println(3)

fmt.Println("defer end")

}

代码输出如下：

defer begin

defer end

3

2

1

结果分析如下：

代码的延迟顺序与最终的执行顺序是反向的。

延迟调用是在 defer 所在函数结束时进行，函数结束可以是正常返回时，也可以是发生宕机时。

### 7 Go语言处理运行时错误

Go 语言的错误处理思想及设计包含以下特征：

一个可能造成错误的函数，需要返回值中返回一个错误接口（error）。如果调用是成功的，错误接口将返回 nil，否则返回错误。在函数调用后需要检查错误，如果发生错误，进行必要的错误处理。Go 语言没有类似 Java 或 .NET 中的异常处理机制，虽然可以使用 defer、panic、recover 模拟，但官方并不主张这样做。Go 语言的设计者认为其他语言的异常机制已被过度使用，上层逻辑需要为函数发生的异常付出太多的资源。同时，如果函数使用者觉得错误处理很麻烦而忽略错误，那么程序将在不可预知的时刻崩溃。

Go 语言希望开发者将错误处理视为正常开发必须实现的环节，正确地处理每一个可能发生错误的函数。同时，Go 语言使用返回值返回错误的机制，也能大幅降低编译器、运行时处理错误的复杂度，让开发者真正地掌握错误的处理。

**（1）net 包中的例子**

net.Dial() 是 Go 语言系统包 net 即中的一个函数，一般用于创建一个 Socket 连接。

net.Dial 拥有两个返回值，即 Conn 和 error。这个函数是阻塞的，因此在 Socket 操作后，会返回 Conn 连接对象和 error；如果发生错误，error 会告知错误的类型，Conn 会返回空。

根据 Go 语言的错误处理机制，Conn 是其重要的返回值。因此，为这个函数增加一个错误返回，类似为 error。参见下面的代码：

func Dial(network, address string) (Conn, error) {

var d Dialer

return d.Dial(network, address)

}

在 io 包中的 Writer 接口也拥有错误返回，代码如下：

type Writer interface {

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

}

io 包中还有 Closer 接口，只有一个错误返回，代码如下：

type Closer interface {

Close() error

}

**（2）错误接口的定义格式**

error 是 Go 系统声明的接口类型，代码如下：

type error interface {

Error() string

}

所有符合 Error()string 格式的方法，都能实现错误接口。

Error() 方法返回错误的具体描述，使用者可以通过这个字符串知道发生了什么错误。

**（3）自定义一个错误**

返回错误前，需要定义会产生哪些可能的错误。在 Go 语言中，使用 errors 包进行错误的定义，格式如下：

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

错误字符串由于相对固定，一般在包作用域声明，应尽量减少在使用时直接使用 errors.New 返回。

**1) errors 包**

Go 语言的 errors 中对 New 的定义非常简单，代码如下：

// 创建错误对象

func New(text string) error {

return &errorString{text}

}

// 错误字符串

type errorString struct {

s string

}

// 返回发生何种错误

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

return e.s

}

代码说明如下：

第 2 行，将 errorString 结构体实例化，并赋值错误描述的成员。

第 7 行，声明 errorString 结构体，拥有一个成员，描述错误内容。

第 12 行，实现 error 接口的 Error() 方法，该方法返回成员中的错误描述。

**2) 在代码中使用错误定义**

下面的代码会定义一个除法函数，当除数为 0 时，返回一个预定义的除数为 0 的错误。

package main

import (

"errors"

"fmt"

)

// 定义除数为0的错误

var errDivisionByZero = errors.New("division by zero")

func div(dividend, divisor int) (int, error) {

// 判断除数为0的情况并返回

if divisor == 0 {

return 0, errDivisionByZero

}

// 正常计算，返回空错误

return dividend / divisor, nil

}

func main() {

fmt.Println(div(1, 0))

}

代码输出如下：

0 division by zero

代码说明：

第 9 行，预定义除数为 0 的错误。

第 11 行，声明除法函数，输入被除数和除数，返回商和错误。

第 14 行，在除法计算中，如果除数为 0，计算结果为无穷大。为了避免这种情况，对除数进行判断，并返回商为 0 和除数为 0 的错误对象。

第 19 行，进行正常的除法计算，没有发生错误时，错误对象返回 nil。

**（4）示例：在解析中使用自定义错误**

使用 errors.New 定义的错误字符串的错误类型是无法提供丰富的错误信息的。那么，如果需要携带错误信息返回，就需要借助自定义结构体实现错误接口。

下面代码将实现一个解析错误（ParseError），这种错误包含两个内容：文件名和行号。解析错误的结构还实现了 error 接口的 Error() 方法，返回错误描述时，就需要将文件名和行号返回。

package main

import (

"fmt"

)

// 声明一个解析错误

type ParseError struct {

Filename string // 文件名

Line int // 行号

}

// 实现error接口，返回错误描述

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

return fmt.Sprintf("%s:%d", e.Filename, e.Line)

}

// 创建一些解析错误

func newParseError(filename string, line int) error {

return &ParseError{filename, line}

}

func main() {

var e error

// 创建一个错误实例，包含文件名和行号

e = newParseError("main.go", 1)

// 通过error接口查看错误描述

fmt.Println(e.Error())

// 根据错误接口具体的类型，获取详细错误信息

switch detail := e.(type) {

case \*ParseError: // 这是一个解析错误

fmt.Printf("Filename: %s Line: %d\n", detail.Filename, detail.Line)

default: // 其他类型的错误

fmt.Println("other error")

}

}

代码输出如下：

main.go:1

Filename: main.go Line: 1

代码说明如下：

第 8 行，声明了一个解析错误的结构体，解析错误包含有 2 个成员：文件名和行号。

第 14 行，实现了错误接口，将成员的文件名和行号格式化为字符串返回。

第 19 行，根据给定的文件名和行号创建一个错误实例。

第 25 行，声明一个错误接口类型。

第 27 行，创建一个实例，这个错误接口内部是 \*ParserError 类型，携带有文件名 main.go 和行号 1。

第 30 行，调用 Error() 方法，通过第 15 行返回错误的详细信息。

第 33 行，通过错误断言，取出发生错误的详细类型。

第 34 行，通过分析这个错误的类型，得知错误类型为 \*ParserError，此时可以获取到详细的错误信息。

第 36 行，如果不是我们能够处理的错误类型，会打印出其他错误做出其他的处理。

错误对象都要实现 error 接口的 Error() 方法，这样，所有的错误都可以获得字符串的描述。如果想进一步知道错误的详细信息，可以通过类型断言，将错误对象转为具体的错误类型进行错误详细信息的获取。

### 8 Go语言宕机（panic）——程序终止运行

宕机不是一件很好的事情，可能造成体验停止、服务中断，就像没有人希望在取钱时遇到 ATM 机蓝屏一样。但是，如果在损失发生时，程序没有因为宕机而停止，那么用户将会付出更大的代价，这种代价可以是金钱、时间甚至生命。因此，宕机有时是一种合理的止损方法。

手动触发宕机

Go 语言可以在程序中手动触发宕机，让程序崩溃，这样开发者可以及时地发现错误，同时减少可能的损失。

Go 语言程序在宕机时，会将堆栈和 goroutine 信息输出到控制台，所以宕机也可以方便地知晓发生错误的位置。如果在编译时加入的调试信息甚至连崩溃现场的变量值、运行状态都可以获取，那么如何触发宕机呢？例如下面的代码：

package main

func main() {

panic("crash")

}

代码运行崩溃并输出如下：

panic: crash

goroutine 1 [running]:

main.main()

F:/src/tester/main.go:5 +0x6b

以上代码中只用了一个内建的函数 panic() 就可以造成崩溃，panic() 的声明如下：

func panic(v interface{})

panic() 的参数可以是任意类型，后文将提到的 recover 参数会接收从 panic() 中发出的内容。

### 9 Go语言宕机恢复（recover）——防止程序崩溃

无论是代码运行错误由 Runtime 层抛出的 panic 崩溃，还是主动触发的 panic 崩溃，都可以配合 defer 和 recover 实现错误捕捉和恢复，让代码在发生崩溃后允许继续运行。

在其他语言里，宕机往往以异常的形式存在。底层抛出异常，上层逻辑通过 try/catch 机制捕获异常，没有被捕获的严重异常会导致宕机，捕获的异常可以被忽略，让代码继续运行。

Go 没有异常系统，其使用 panic 触发宕机类似于其他语言的抛出异常，那么 recover 的宕机恢复机制就对应 try/catch 机制。

**（1）让程序在崩溃时继续执行**

下面的代码实现了 ProtectRun() 函数，该函数传入一个匿名函数或闭包后的执行函数，当传入函数以任何形式发生 panic 崩溃后，可以将崩溃发生的错误打印出来，同时允许后面的代码继续运行，不会造成整个进程的崩溃。

保护运行函数：

package main

import (

"fmt"

"runtime"

)

// 崩溃时需要传递的上下文信息

type panicContext struct {

function string // 所在函数

}

// 保护方式允许一个函数

func ProtectRun(entry func()) {

// 延迟处理的函数

defer func() {

// 发生宕机时，获取panic传递的上下文并打印

err := recover()

switch err.(type) {

case runtime.Error: // 运行时错误

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

default: // 非运行时错误

fmt.Println("error:", err)

}

}()

entry()

}

func main() {

fmt.Println("运行前")

// 允许一段手动触发的错误

ProtectRun(func() {

fmt.Println("手动宕机前")

// 使用panic传递上下文

panic(&panicContext{

"手动触发panic",

})

fmt.Println("手动宕机后")

})

// 故意造成空指针访问错误

ProtectRun(func() {

fmt.Println("赋值宕机前")

var a \*int

\*a = 1

fmt.Println("赋值宕机后")

})

fmt.Println("运行后")

}

代码输出结果：

运行前

手动宕机前

error: &{手动触发panic}

赋值宕机前

runtime error: runtime error: invalid memory address or nil pointer

dereference

运行后

对代码的说明：

第 9 行声明描述错误的结构体，成员保存错误的执行函数。

第 17 行使用 defer 将闭包延迟执行，当 panic 触发崩溃时，ProtectRun() 函数将结束运行，此时 defer 后的闭包将会发生调用。

第 20 行，recover() 获取到 panic 传入的参数。

第 22 行，使用 switch 对 err 变量进行类型断言。

第 23 行，如果错误是有 Runtime 层抛出的运行时错误，如空指针访问、除数为 0 等情况，打印运行时错误。

第 25 行，其他错误，打印传递过来的错误数据。

第 44 行，使用 panic 手动触发一个错误，并将一个结构体附带信息传递过去，此时，recover 就会获取到这个结构体信息，并打印出来。

第 57 行，模拟代码中空指针赋值造成的错误，此时会由 Runtime 层抛出错误，被 ProtectRun() 函数的 recover() 函数捕获到。

**（2）panic和recover的关系**

panic 和 defer 的组合有如下特性：

有 panic 没 recover，程序宕机。

有 panic 也有 recover 捕获，程序不会宕机。执行完对应的 defer 后，从宕机点退出当前函数后继续执行。

虽然 panic/recover 能模拟其他语言的异常机制，但并不建议代表编写普通函数也经常性使用这种特性。在 panic 触发的 defer 函数内，可以继续调用 panic，进一步将错误外抛直到程序整体崩溃。如果想在捕获错误时设置当前函数的返回值，可以对返回值使用命名返回值方式直接进行设置。

## 六、Go语言[结构体](http://c.biancheng.net/golang/struct/)

### 1 Go语言结构体定义

Go语言的关键字 type 可以将各种基本类型定义为自定义类型，基本类型包括整型、字符串、布尔等。结构体是一种复合的基本类型，通过 type 定义为自定义类型后，使结构体更便于使用。

结构体的定义格式如下：

type 类型名 struct {

字段1 字段1类型

字段2 字段2类型

…

}

对各个部分的说明：

类型名：标识自定义结构体的名称，在同一个包内不能重复。

struct{}：表示结构体类型，type 类型名 struct{} 可以理解为将 struct{} 结构体定义为类型名的类型。

字段1、字段2……：表示结构体字段名。结构体中的字段名必须唯一。

字段1类型、字段2类型……：表示结构体字段的类型。

使用结构体可以表示一个包含 X 和 Y 整型分量的点结构，代码如下：

type Point struct {

X int

Y int

}

同类型的变量也可以写在一行。颜色的红、绿、蓝 3 个分量可以使用 byte 类型表示，定义的颜色结构体如下：

type Color struct {

R, G, B byte

}

结构体的定义只是一种内存布局的描述，只有当结构体实例化时，才会真正地分配内存，我们将在下节详细讲解。

### 2 Go语言实例化结构体——为结构体分配内存并初始化

结构体的定义只是一种内存布局的描述，只有当结构体实例化时，才会真正地分配内存。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。实例化就是根据结构体定义的格式创建一份与格式一致的内存区域，结构体实例与实例间的内存是完全独立的。Go 语言可以通过多种方式实例化结构体，根据实际需要可以选用不同的写法。

**（1）基本的实例化形式**

结构体本身是一种类型，可以像整型、字符串等类型一样，以 var 的方式声明结构体即可完成实例化。

基本实例化格式如下：

var ins T

其中，T 为结构体类型，ins 为结构体的实例。

用结构体表示的点结构（Point）的实例化过程请参见下面的代码：

type Point struct {

X int

Y int

}

var p Point

p.X = 10

p.Y = 20

在例子中，使用.来访问结构体的成员变量，如 p.X 和 p.Y 等。结构体成员变量的赋值方法与普通变量一致。

**（2）创建指针类型的结构体**

Go 语言中，还可以使用 new 关键字对类型（包括结构体、整型、浮点数、字符串等）进行实例化，结构体在实例化后会形成指针类型的结构体。

使用 new 的格式如下：

ins := new(T)

其中：

T 为类型，可以是结构体、整型、字符串等。

ins：T 类型被实例化后保存到 ins 变量中，ins 的类型为 \*T，属于指针。

Go 语言让我们可以像访问普通结构体一样使用.访问结构体指针的成员。

下面的例子定义了一个玩家（Player）的结构，玩家拥有名字、生命值和魔法值，实例化玩家（Player）结构体后，可对成员进行赋值，代码如下：

type Player struct{

Name string

HealthPoint int

MagicPoint int

}

tank := new(Player)

tank.Name = "Canon"

tank.HealthPoint = 300

经过 new 实例化的结构体实例在成员赋值上与基本实例化的写法一致。

Go语言和C/C++

在 C/C++ 语言中，使用 new 实例化类型后，访问其成员变量时必须使用->操作符。

在 Go 语言中，访问结构体指针的成员变量时可以继续使用.。这是因为 Go 语言为了方便开发者访问结构体指针的成员变量，使用了语法糖（Syntactic sugar）技术，将 ins.Name 形式转换为 (\*ins).Name。

**（3）取结构体的地址实例化**

在 Go 语言中，对结构体进行&取地址操作时，视为对该类型进行一次 new 的实例化操作。取地址格式如下：

ins := &T{}

其中：

T 表示结构体类型。

ins 为结构体的实例，类型为 \*T，是指针类型。

下面使用结构体定义一个命令行指令（Command），指令中包含名称、变量关联和注释等。对 Command 进行指针地址的实例化，并完成赋值过程，代码如下：

type Command struct {

Name string // 指令名称

Var \*int // 指令绑定的变量

Comment string // 指令的注释

}

var version int = 1

cmd := &Command{}

cmd.Name = "version"

cmd.Var = &version

cmd.Comment = "show version"

代码说明如下：

第 1 行，定义 Command 结构体，表示命令行指令

第 3 行，命令绑定的变量，使用整型指针绑定一个指针。指令的值可以与绑定的值随时保持同步。

第 7 行，命令绑定的目标整型变量：版本号。

第 9 行，对结构体取地址实例化。

第 10～12 行，初始化成员字段。

取地址实例化是最广泛的一种结构体实例化方式。可以使用函数封装上面的初始化过程，代码如下：

纯文本复制

func newCommand(name string, varref \*int, comment string) \*Command {

return &Command{

Name: name,

Var: varref,

Comment: comment,

}

}

cmd = newCommand(

"version",

&version,

"show version",

)

### 3 Go语言初始化结构体的成员变量

结构体在实例化时可以直接对成员变量进行初始化。初始化有两种形式：一种是字段“键值对”形式及多个值的列表形式。键值对形式的初始化适合选择性填充字段较多的结构体；多个值的列表形式适合填充字段较少的结构体。

使用“键值对”初始化结构体

结构体可以使用“键值对”（Key value pair）初始化字段，每个“键”（Key）对应结构体中的一个字段。键的“值”（Value）对应字段需要初始化的值。

键值对的充是可选的，不需要初始化的字段可以不填入初始化列表中。

结构体实例化后字段的默认值是字段类型的默认值，例如：数值为 0，字符串为空字符串，布尔为 false，指针为 nil 等。

**（1）键值对初始化结构体的书写格式**

键值对初始化的格式如下：

ins := 结构体类型名{

字段1: 字段1的值,

字段2: 字段2的值,

…

}

下面是对各个部分的说明：

结构体类型：定义结构体时的类型名称。

字段1、字段2：结构体的成员字段名。结构体类型名的字段初始化列表中，字段名只能出现一次。

字段1的值、字段2的值：结构体成员字段的初始值。

键值之间以:分隔；键值对之间以,分隔。

**（2） 使用键值对填充结构体的例子**

下面例子中描述了家里的人物关联。正如儿歌里唱的：“爸爸的爸爸是爷爷”，人物之间可以使用多级的 child 来描述和建立关联。使用键值对形式填充结构体的代码如下：

type People struct {

name string

child \*People

}

relation := &People{

name: "爷爷",

child: &People{

name: "爸爸",

child: &People{

name: "我",

},

},

}

代码说明如下：

第 1 行，定义 People 结构体。

第 2 行，结构体的字符串字段。

第 3 行，结构体的结构体指针字段，类型是 \*People。

第 6 行，relation 由 People 类型取地址后，形成类型为 \*People 的实例。

第 8 行，child 在初始化时，需要 \*People 类型的值。使用取地址初始化一个 People。

结构体成员中只能包含结构体的指针类型，包含非指针类型会引起编译错误。

### 4 Go语言构造函数

Go 语言的类型或结构体没有构造函数的功能。结构体的初始化过程可以使用函数封装实现。其他编程语言构造函数的一些常见功能及特性如下：

每个类可以添加构造函数，多个构造函数使用函数重载实现。

构造函数一般与类名同名，且没有返回值。

构造函数有一个静态构造函数，一般用这个特性来调用父类的构造函数。

对于 C++ 来说，还有默认构造函数、拷贝构造函数等。

**（1）多种方式创建和初始化结构体——模拟构造函数重载**

如果使用结构体描述猫的特性，那么根据猫的颜色和名字可以有不同种类的猫。那么不同的颜色和名字就是结构体的字段，同时可以使用颜色和名字构造不同种类的猫的实例，这个过程可以参考下面的代码：

type Cat struct {

Color string

Name string

}

func NewCatByName(name string) \*Cat {

return &Cat{

Name: name,

}

}

func NewCatByColor(color string) \*Cat {

return &Cat{

Color: color,

}

}

代码说明如下：

第 1 行定义 Cat 结构，包含颜色和名字字段。

第 6 行定义用名字构造猫结构的函数，返回 Cat 指针。

第 7 行取地址实例化猫的结构体。

第 8 行初始化猫的名字字段，忽略颜色字段。

第 12 行定义用颜色构造猫结构的函数，返回 Cat 指针。

在这个例子中，颜色和名字两个属性的类型都是字符串。由于 Go 语言中没有函数重载，为了避免函数名字冲突，使用 NewCatByName() 和 NewCatByColor() 两个不同的函数名表示不同的 Cat 构造过程。

**（2）带有父子关系的结构体的构造和初始化——模拟父级构造调用**

黑猫是一种猫，猫是黑猫的一种泛称。同时描述这两种概念时，就是派生，黑猫派生自猫的种类。使用结构体描述猫和黑猫的关系时，将猫（Cat）的结构体嵌入到黑猫（BlackCat）中，表示黑猫拥有猫的特性，然后再使用两个不同的构造函数分别构造出黑猫和猫两个结构体实例，参考下面的代码：

type Cat struct {

Color string

Name string

}

type BlackCat struct {

Cat // 嵌入Cat, 类似于派生

}

// “构造基类”

func NewCat(name string) \*Cat {

return &Cat{

Name: name,

}

}

// “构造子类”

func NewBlackCat(color string) \*BlackCat {

cat := &BlackCat{}

cat.Color = color

return cat

}

代码说明如下：

第 6 行，定义 BlackCat 结构，并嵌入了 Cat 结构体。BlackCat 拥有 Cat 的所有成员，实例化后可以自由访问 Cat 的所有成员。

第 11 行，NewCat() 函数定义了 Cat 的构造过程，使用名字作为参数，填充 Cat 结构体。

第 18 行，NewBlackCat() 使用 color 作为参数，构造返回 BlackCat 指针。

第 19 行，实例化 BlackCat 结构，此时 Cat 也同时被实例化。

第 20 行，填充 BlackCat 中嵌入的 Cat 颜色属性。BlackCat 没有任何成员，所有的成员都来自于 Cat。

这个例子中，Cat 结构体类似于面向对象中的“基类”。BlackCat 嵌入 Cat 结构体，类似于面向对象中的“派生”。实例化时，BlackCat 中的 Cat 也会一并被实例化。

### 5 Go语言方法和接收器

Go 语言中的方法（Method）是一种作用于特定类型变量的函数。这种特定类型变量叫做接收器（Receiver）。如果将特定类型理解为结构体或“类”时，接收器的概念就类似于其他语言中的 this 或者 self。在 Go 语言中，接收器的类型可以是任何类型，不仅仅是结构体，任何类型都可以拥有方法。在面向对象的语言中，类拥有的方法一般被理解为类可以做的事情。在 Go 语言中“方法”的概念与其他语言一致，只是 Go 语言建立的“接收器”强调方法的作用对象是接收器，也就是类实例，而函数没有作用对象。

**（1）为结构体添加方法**

本节中，将会使用背包作为“对象”，将物品放入背包的过程作为“方法”，通过面向过程的方式和 Go 语言中结构体的方式来理解“方法”的概念。

1) 面向过程实现方法

面向过程中没有“方法”概念，只能通过结构体和函数，由使用者使用函数参数和调用关系来形成接近“方法”的概念，代码如下：

type Bag struct {

items []int

}

// 将一个物品放入背包的过程

func Insert(b \*Bag, itemid int) {

b.items = append(b.items, itemid)

}

func main() {

bag := new(Bag)

Insert(bag, 1001)

}

代码说明如下：

第 1 行，声明 Bag 结构，这个结构体包含一个整型切片类型的 items 的成员。

第 6 行，定义了 Insert() 函数，这个函数拥有两个参数，第一个是背包指针（\*Bag），第二个是物品ID（itemid）。

第 7 行，用 append() 将 itemid 添加到 Bag 的 items 成员中，模拟往背包添加物品的过程。

第 12 行，创建背包实例 bag。

第 14 行，调用 Insert() 函数，第一个参数放入背包，第二个参数放入物品 ID。

Insert() 函数将 \*Bag 参数放在第一位，强调 Insert 会操作 \*Bag 结构体。但实际使用中，并不是每个人都会习惯将操作对象放在首位。一定程度上让代码失去一些范式和描述性。同时，Insert() 函数也与 Bag 没有任何归属概念。随着类似 Insert() 的函数越来越多，面向过程的代码描述对象方法概念会越来越麻烦和难以理解。

**（2）Go语言的结构体方法**

将背包及放入背包的物品中使用 Go 语言的结构体和方法方式编写：为 \*Bag 创建一个方法，代码如下：

type Bag struct {

items []int

}

func (b \*Bag) Insert(itemid int) {

b.items = append(b.items, itemid)

}

func main() {

b := new(Bag)

b.Insert(1001)

}

第 5 行中，Insert(itemid int) 的写法与函数一致。(b\*Bag) 表示接收器，即 Insert 作用的对象实例。

## 七、Go语言[接口](http://c.biancheng.net/golang/interface/)

### 1 Go语言结构声明（定义）

接口是双方约定的一种合作协议。接口实现者不需要关心接口会被怎样使用，调用者也不需要关心接口的实现细节。接口是一种类型，也是一种抽象结构，不会暴露所含数据的格式、类型及结构。

**（1）接口声明的格式**

每个接口类型由数个方法组成。接口的形式代码如下：

type 接口类型名 interface{

方法名1( 参数列表1 ) 返回值列表1

方法名2( 参数列表2 ) 返回值列表2

…

}

对各个部分的说明：

接口类型名：使用 type 将接口定义为自定义的类型名。Go 语言的接口在命名时，一般会在单词后面添加 er，如有写操作的接口叫 Writer，有字符串功能的接口叫 Stringer，有关闭功能的接口叫 Closer 等。

方法名：当方法名首字母是大写时，且这个接口类型名首字母也是大写时，这个方法可以被接口所在的包（package）之外的代码访问。

参数列表、返回值列表：参数列表和返回值列表中的参数变量名可以被忽略，例如：

type writer interface{

Write([]byte) error

}

**（2）开发中常见的接口及写法**

Go 语言提供的很多包中都有接口，例如 io 包中提供的 Writer 接口：

type Writer interface {

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

}

这个接口可以调用 Write() 方法写入一个字节数组（[]byte），返回值告知写入字节数（n int）和可能发生的错误（err error）。

类似的，还有将一个对象以字符串形式展现的接口，只要实现了这个接口的类型，在调用 String() 方法时，都可以获得对象对应的字符串。在 fmt 包中定义如下：

type Stringer interface {

String() string

}

Stringer 接口在 Go 语言中的使用频率非常高，功能类似于 Java 或者 C# 语言里的 ToString 的操作。

Go 语言的每个接口中的方法数量不会很多。Go 语言希望通过一个接口精准描述它自己的功能，而通过多个接口的嵌入和组合的方式将简单的接口扩展为复杂的接口。本章后面的小节中会介绍如何使用组合来扩充接口。

### 2 Go语言实现接口的条件

接口定义后，需要实现接口，调用方才能正确编译通过并使用接口。接口的实现需要遵循两条规则才能让接口可用。

**（1）接口被实现的条件一：接口的方法与实现接口的类型方法格式一致**

在类型中添加与接口签名一致的方法就可以实现该方法。签名包括方法中的名称、参数列表、返回参数列表。也就是说，只要实现接口类型中的方法的名称、参数列表、返回参数列表中的任意一项与接口要实现的方法不一致，那么接口的这个方法就不会被实现。

为了抽象数据写入的过程，定义 DataWriter 接口来描述数据写入需要实现的方法，接口中的 WriteData() 方法表示将数据写入，写入方无须关心写入到哪里。实现接口的类型实现 WriteData 方法时，会具体编写将数据写入到什么结构中。这里使用file结构体实现 DataWriter 接口的 WriteData 方法，方法内部只是打印一个日志，表示有数据写入，详细实现过程请参考下面的代码。

数据写入器的抽象：

package main

import (

"fmt"

)

// 定义一个数据写入器

type DataWriter interface {

WriteData(data interface{}) error

}

// 定义文件结构，用于实现DataWriter

type file struct {

}

// 实现DataWriter接口的WriteData方法

func (d \*file) WriteData(data interface{}) error {

// 模拟写入数据

fmt.Println("WriteData:", data)

return nil

}

func main() {

// 实例化file

f := new(file)

// 声明一个DataWriter的接口

var writer DataWriter

// 将接口赋值f，也就是\*file类型

writer = f

// 使用DataWriter接口进行数据写入

writer.WriteData("data")

}

代码说明如下：

第 8 行，定义 DataWriter 接口。这个接口只有一个方法，即 WriteData()，输入一个 interface{} 类型的 data，返回一个 error 结构表示可能发生的错误。

第 17 行，file 的 WriteData() 方法使用指针接收器。输入一个 interface{} 类型的 data，返回 error。

第 27 行，实例化 file 赋值给 f，f 的类型为 \*file。

第 30 行，声明 DataWriter 类型的 writer 接口变量。

第 33 行，将 \*file 类型的 f 赋值给 DataWriter 接口的 writer，虽然两个变量类型不一致。但是 writer 是一个接口，且 f 已经完全实现了 DataWriter() 的所有方法，因此赋值是成功的。

第 36 行，DataWriter 接口类型的 writer 使用 WriteData() 方法写入一个字符串。

运行代码，输出如下：

WriteData: data

当类型无法实现接口时，编译器会报错，下面列出常见的几种接口无法实现的错误。

**1) 函数名不一致导致的报错**

在以上代码的基础上尝试修改部分代码，造成编译错误，通过编译器的报错理解如何实现接口的方法。首先，修改 file 结构的 WriteData() 方法名，将这个方法签名（第17行）修改为：

func (d \*file) WriteDataX(data interface{}) error {

编译代码，报错：

cannot use f (type \*file) as type DataWriter in assignment:

\*file does not implement DataWriter (missing WriteData method)

报错的位置在第 33 行。报错含义是：不能将 f 变量（类型\*file）视为 DataWriter 进行赋值。原因：\*file 类型未实现 DataWriter 接口（丢失 WriteData 方法）。

WriteDataX 方法的签名本身是合法的。但编译器扫描到第 33 行代码时，发现尝试将 \*file 类型赋值给 DataWriter 时，需要检查 \*file 类型是否完全实现了 DataWriter 接口。显然，编译器因为没有找到 DataWriter 需要的 WriteData() 方法而报错。

**2) 实现接口的方法签名不一致导致的报错**

将修改的代码恢复后，再尝试修改 WriteData() 方法，把 data 参数的类型从 interface{} 修改为 int 类型，代码如下：

func (d \*file) WriteData(data int) error {

编译代码，报错：

cannot use f (type \*file) as type DataWriter in assignment:

\*file does not implement DataWriter (wrong type for WriteData method)

have WriteData(int) error

want WriteData(interface {}) error

这次未实现 DataWriter 的理由变为（错误的 WriteData() 方法类型）发现 WriteData(int)error，期望 WriteData(interface{})error。

这种方式的报错就是由实现者的方法签名与接口的方法签名不一致导致的。

**（2）接口被实现的条件二：接口中所有方法均被实现**

当一个接口中有多个方法时，只有这些方法都被实现了，接口才能被正确编译并使用。

在本节开头的代码中，为 DataWriter中 添加一个方法，代码如下：

// 定义一个数据写入器

type DataWriter interface {

WriteData(data interface{}) error

// 能否写入

CanWrite() bool

}

新增 CanWrite() 方法，返回 bool。此时再次编译代码，报错：

cannot use f (type \*file) as type DataWriter in assignment:

\*file does not implement DataWriter (missing CanWrite method)

需要在 file 中实现 CanWrite() 方法才能正常使用 DataWriter()。

Go 语言的接口实现是隐式的，无须让实现接口的类型写出实现了哪些接口。这个设计被称为非侵入式设计。

实现者在编写方法时，无法预测未来哪些方法会变为接口。一旦某个接口创建出来，要求旧的代码来实现这个接口时，就需要修改旧的代码的派生部分，这一般会造成雪崩式的重新编译。

传统的派生式接口及类关系构建的模式，让类型间拥有强耦合的父子关系。这种关系一般会以“类派生图”的方式进行。经常可以看到大型软件极为复杂的派生树。随着系统的功能不断增加，这棵“派生树”会变得越来越复杂。

对于 Go 语言来说，非侵入式设计让实现者的所有类型均是平行的、组合的。如何组合则留到使用者编译时再确认。因此，使用 GO 语言时，不需要同时也不可能有“类派生图”，开发者唯一需要关注的就是“我需要什么？”，以及“我能实现什么？”。

### 3 Go语言类型与接口的关系

类型和接口之间有一对多和多对一的关系，下面将列举出这些常见的概念，以方便读者理解接口与类型在复杂环境下的实现关系。

**（1）一个类型可以实现多个接口**

一个类型可以同时实现多个接口，而接口间彼此独立，不知道对方的实现。

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，连接的一端称为一个 Socket。Socket 能够同时读取和写入数据，这个特性与文件类似。因此，开发中把文件和 Socket 都具备的读写特性抽象为独立的读写器概念。

Socket 和文件一样，在使用完毕后，也需要对资源进行释放。

把 Socket 能够写入数据和需要关闭的特性使用接口来描述，请参考下面的代码：

type Socket struct {

}

func (s \*Socket) Write(p []byte) (n int, err error) {

return 0, nil

}

func (s \*Socket) Close() error {

return nil

}

Socket 结构的 Write() 方法实现了 io.Writer 接口：

type Writer interface {

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

}

同时，Socket 结构也实现了 io.Closer 接口：

type Closer interface {

Close() error

}

使用 Socket 实现的 Writer 接口的代码，无须了解 Writer 接口的实现者是否具备 Closer 接口的特性。同样，使用 Closer 接口的代码也并不知道 Socket 已经实现了 Writer 接口，如下图所示。

在代码中使用Socket结构实现的Writer接口和Closer接口代码如下：

// 使用io.Writer的代码, 并不知道Socket和io.Closer的存在

func usingWriter( writer io.Writer){

writer.Write( nil )

}

// 使用io.Closer, 并不知道Socket和io.Writer的存在

func usingCloser( closer io.Closer) {

closer.Close()

}

func main() {

// 实例化Socket

s := new(Socket)

usingWriter(s)

usingCloser(s)

}

usingWriter() 和 usingCloser() 完全独立，互相不知道对方的存在，也不知道自己使用的接口是 Socket 实现的。

**（2）多个类型可以实现相同的接口**

一个接口的方法，不一定需要由一个类型完全实现，接口的方法可以通过在类型中嵌入其他类型或者结构体来实现。也就是说，使用者并不关心某个接口的方法是通过一个类型完全实现的，还是通过多个结构嵌入到一个结构体中拼凑起来共同实现的。

Service 接口定义了两个方法：一个是开启服务的方法（Start()），一个是输出日志的方法（Log()）。使用 GameService 结构体来实现 Service，GameService 自己的结构只能实现 Start() 方法，而 Service 接口中的 Log() 方法已经被一个能输出日志的日志器（Logger）实现了，无须再进行 GameService 封装，或者重新实现一遍。所以，选择将 Logger 嵌入到 GameService 能最大程度地避免代码冗余，简化代码结构。详细实现过程如下：

// 一个服务需要满足能够开启和写日志的功能

type Service interface {

Start() // 开启服务

Log(string) // 日志输出

}

// 日志器

type Logger struct {

}

// 实现Service的Log()方法

func (g \*Logger) Log(l string) {

}

// 游戏服务

type GameService struct {

Logger // 嵌入日志器

}

// 实现Service的Start()方法

func (g \*GameService) Start() {

}

代码说明如下：

第 2 行，定义服务接口，一个服务需要实现 Start() 方法和日志方法。

第 8 行，定义能输出日志的日志器结构。

第 12 行，为 Logger 添加 Log() 方法，同时实现 Service 的 Log() 方法。

第 17 行，定义 GameService 结构。

第 18 行，在 GameService 中嵌入 Logger 日志器，以实现日志功能。

第 22 行，GameService 的 Start() 方法实现了 Service 的 Start() 方法。

此时，实例化 GameService，并将实例赋给 Service，代码如下：

var s Service = new(GameService)

s.Start()

s.Log(“hello”)

s 就可以使用 Start() 方法和 Log() 方法，其中，Start() 由 GameService 实现，Log() 方法由 Logger 实现。

### 4 Go语言实现日志系统（支持多种输出方式）

日志可以用于查看和分析应用程序的运行状态。日志一般可以支持输出多种形式，如命令行、文件、网络等。

**（1）日志对外接口**

本例中定义一个日志写入器接口（LogWriter），要求写入设备必须遵守这个接口协议才能被日志器（Logger）注册。日志器有一个写入器的注册方法（Logger 的 RegisterWriter() 方法）。

日志器还有一个 Log() 方法，进行日志的输出，这个函数会将日志写入到所有已经注册的日志写入器（LogWriter）中，详细代码实现请参考下面的代码。

package main

// 声明日志写入器接口

type LogWriter interface {

Write(data interface{}) error

}

// 日志器

type Logger struct {

// 这个日志器用到的日志写入器

writerList []LogWriter

}

// 注册一个日志写入器

func (l \*Logger) RegisterWriter(writer LogWriter) {

l.writerList = append(l.writerList, writer)

}

// 将一个data类型的数据写入日志

func (l \*Logger) Log(data interface{}) {

// 遍历所有注册的写入器

for \_, writer := range l.writerList {

// 将日志输出到每一个写入器中

writer.Write(data)

}

}

// 创建日志器的实例

func NewLogger() \*Logger {

return &Logger{}

}

代码说明如下：

第 4 行，声明日志写入器接口。这个接口可以被外部使用。日志的输出可以有多种设备，这个写入器就是用来实现一个日志的输出设备。

第 9 行，声明日志器结构。日志器使用 writeList 记录输出到哪些设备上。

第 15 行，使用日志器方法 RegisterWriter() 将一个日志写入器（LogWriter）注册到日志器（Logger）中。注册的意思就是将日志写入器的接口添加到 writeList 中。

第 20 行，日志器的 Log() 方法可以将 interface{} 类型的 data 写入到注册过的日志写入器中。

第 23 行，遍历日志器拥有的所有日志写入器。

第 26 行，将本次日志的内容写入日志写入器。

第 31 行，创建日志器的实例。

这个例子中，为了最大程度地展示接口的用法，仅仅只是将数据直接写入日志写入器中。复杂一些的日志器还可以将日期、级别等信息合并到数据中一并写入日志。

**（2）文件写入器**

文件写入器（fileWriter）是众多日志写入器（LogWriter）中的一种。文件写入器的功能是根据一个文件名创建日志文件（fileWriter 的 SetFile 方法）。在有日志写入时，将日志写入文件中。

文件写入器代码：

package main

import (

"errors"

"fmt"

"os"

)

// 声明文件写入器

type fileWriter struct {

file \*os.File

}

// 设置文件写入器写入的文件名

func (f \*fileWriter) SetFile(filename string) (err error) {

// 如果文件已经打开, 关闭前一个文件

if f.file != nil {

f.file.Close()

}

// 创建一个文件并保存文件句柄

f.file, err = os.Create(filename)

// 如果创建的过程出现错误, 则返回错误

return err

}

// 实现LogWriter的Write()方法

func (f \*fileWriter) Write(data interface{}) error {

// 日志文件可能没有创建成功

if f.file == nil {

// 日志文件没有准备好

return errors.New("file not created")

}

// 将数据序列化为字符串

str := fmt.Sprintf("%v\n", data)

// 将数据以字节数组写入文件中

\_, err := f.file.Write([]byte(str))

return err

}

// 创建文件写入器实例

func newFileWriter() \*fileWriter {

return &fileWriter{}

}

代码说明如下：

第 10 行，声明文件写入器，在结构体中保存一个文件句柄，以方便每次写入时操作。

第 15 行，文件写入器通过文件名创建文件，这里通过 SetFile 的参数提供一个文件名，并创建文件。

第 18 行，考虑到 SetFile() 方法可以被多次调用（函数可重入性），假设之前已经调用过 SetFile() 后再次调用，此时的 f.file 不为空，就需要关闭之前的文件，重新创建新的文件。

第 23 行，根据文件名创建文件，如果发生错误，通过 SetFile 的返回值返回。

第 30 行，fileWriter 的 Write() 方法实现了 LogWriter 接口的 Write() 方法。

第 33 行，如果文件没有准备好，文件句柄为 nil，此时使用 errors 包的 New() 函数返回一个错误对象，包含一个字符串“file not created”。

第 40 行，通过 Write() 方法传入的 data 参数是 interface{} 类型，而 f.file 的 Write() 方法需要的是 []byte 类型。使用 fmt.Sprintf 将 data 转换为字符串，这里使用的格式化参数是%v，意思是将 data 按其本来的值转换为字符串。

第 43 行，通过 f.file 的 Write() 方法，将 str 字符串转换为 []byte 字节数组，再写入到文件中。如果发生错误，则返回。

在操作文件时，会出现文件无法创建、无法写入等错误。开发中尽量不要忽略这些底层报出的错误，应该处理可能发生的所有错误。

文件使用完后，要注意使用 os.File 的 Close() 方法进行及时关闭，否则文件再次访问时会因为其属性出现无法读取、无法写入等错误。

一个完备的文件写入器会提供多种写入文件的模式，例子中使用的模式是将日志添加到日志文件的尾部。随着文件越来越大，文件的访问效率和查看便利性也会大大降低。此时，就需要另外一种写入模式：滚动写入文件。

滚动写入文件模式也是将日志添加到文件的尾部，但当文件达到设定的期望大小时，会自动开启一个新的文件继续写入文件，最终将获得多个日志文件。

日志文件名不仅可以按照文件大小进行分割，还可以按照日期范围进行分割。在到达设定的日期范围，如每天、每小时的周期范围时，日志器会自动创建新的日志文件。这种日志文件创建方法也能方便开发者按日志查看日志。

**（3）命令行写入器**

在 UNIX 的思想中，一切皆文件。文件包括内存、磁盘、网络和命令行等。这种抽象方法方便我们访问这些看不见摸不着的虚拟资源。命令行在 Go 中也是一种文件，os.Stdout 对应标准输出，一般表示屏幕，也就是命令行，也可以被重定向为打印机或者磁盘文件；os.Stderr 对应标准错误输出，一般将错误输出到日志中，不过大多数情况，os.Stdout 会与 os.Stderr 合并输出；os.Stdin 对应标准输入，一般表示键盘。os.Stdout、os.Stderr、os.Stdin 都是 \*os.File 类型，和文件一样实现了 io.Writer 接口的 Write() 方法。

下面的代码展示如何将命令行抽象为日志写入器：

package main

import (

"fmt"

"os"

)

// 命令行写入器

type consoleWriter struct {

}

// 实现LogWriter的Write()方法

func (f \*consoleWriter) Write(data interface{}) error {

// 将数据序列化为字符串

str := fmt.Sprintf("%v\n", data)

// 将数据以字节数组写入命令行中

\_, err := os.Stdout.Write([]byte(str))

return err

}

// 创建命令行写入器实例

func newConsoleWriter() \*consoleWriter {

return &consoleWriter{}

}

代码说明如下：

第 9 行，声明 consoleWriter 结构，以实现命令行写入器。

第 13 行，consoleWriter 的 Write() 方法实现了日志写入接口（LogWriter）的 Write() 方法。

第 16 行，与 fileWriter 类似，这里也将 data 通过 fmt.Sprintf 序列化为字符串。

第 19 行，与 fileWriter 类似，这里也将 str 字符串转换为字节数组并写入标准输出 os.Stdout。写入后的内容就会显示在命令行中。

第 25 行，创建命令行写入器的实例。

除了命令行写入器（consoleWriter）和文件写入器（fileWriter），读者还可以自行使用 net 包中的 Socket 封装实现网络写入器 socketWriter，让日志可以写入远程的服务器中或者可以跨进程进行日志保存和分析。

**（4）使用日志**

在程序中使用日志器一般会先通过代码创建日志器（Logger），为日志器添加输出设备（fileWriter、consoleWriter等）。这些设备中有一部分需要一些参数设定，如文件日志写入器需要提供文件名（fileWriter 的 SetFile() 方法）。

下面代码中展示了使用日志器的过程：

package main

import "fmt"

// 创建日志器

func createLogger() \*Logger {

// 创建日志器

l := NewLogger()

// 创建命令行写入器

cw := newConsoleWriter()

// 注册命令行写入器到日志器中

l.RegisterWriter(cw)

// 创建文件写入器

fw := newFileWriter()

// 设置文件名

if err := fw.SetFile("log.log"); err != nil {

fmt.Println(err)

}

// 注册文件写入器到日志器中

l.RegisterWriter(fw)

return l

}

func main() {

// 准备日志器

l := createLogger()

// 写一个日志

l.Log("hello")

}

代码说明如下：

第 6 行，一个创建日志的过程。这个过程一般隐藏在系统初始化中。程序启动时初始化一次。

第 9 行，创建一个日志器的实例，后面的代码会使用到它。

第 12 行，创建一个命令行写入器。如果全局有很多日志器，命令行写入器可以被共享，全局只会有一份。

第 18 行，创建一个文件写入器。一个程序的日志一般只有一个，因此不同的日志器也应该共享一个文件写入器。

第 21 行，创建好的文件写入器需要初始化写入的文件，通过文件名确定写入的文件。设置的过程可能会发生错误，发生错误时会输出错误信息。

第 26 行，将文件写入器注册到日志器中。

第 34 行，在程序一开始创建日志器。

第 37 行，往创建好的日志器中写入日志。

编译整个代码并运行，输出如下：

hello

同时，当前目录的 log.log 文件中也会出现 hello 字符。

Go 语言的 log 包实现了一个小型的日志系统。这个日志系统可以在创建日志器时选择输出设备、日志前缀及 flag，函数定义如下：

func New(out io.Writer, prefix string, flag int) \*Logger {

return &Logger{out: out, prefix: prefix, flag: flag}

}

在 flag 中，还可以定制日志中是否输出日期、日期精度和详细文件名等。

这个日志器在编写时，也最大程度地保证了输出的效率，如果读者对日志器的编写比较感兴趣，可以在 log 包的基础上进行扩展，形成方便自己使用的日志库。

### 5 Go语言排序（借助sort.Interface接口）

排序是常见的算法之一，也是常见的面试题之一，程序员对各种排序算法也是津津乐道。实际使用中，语言的类库会为我们提供健壮、高性能的排序算法库，开发者在了解排序算法基本原理的基础上，应该避免“造轮子”，直接使用已有的排序算法库，以缩短开发周期，提高开发效率。

Go语言中在排序时，需要使用者通过 sort.Interface 接口提供数据的一些特性和操作方法。接口定义代码如下：

type Interface interface {

// 获取元素数量

Len() int

// 小于比较

Less(i, j int) bool

// 交换元素

Swap(i, j int)

}

代码说明如下：

第 3 行，排序算法需要实现者提供需要排序的数据元素数量。

第 6 行，排序需要通过比较元素之间的关系才能做出具体的操作。Less() 方法需要提供两个给定索引（i 和 j）对应元素的小于比较（数值的 < 操作）的结果。参数的 i、j 传入的是元素的索引。将传入的 i、j 索引对应的元素按小于关系进行比较，完成后把结果通过 Less() 方法的返回值返回。

第 9 行，排序的过程就是不停地交换元素。Swap() 方法需要实现者通过传入 i、j 索引找到元素，并交换元素的值。

这个接口需要实现者实现的方法就是排序的经典操作：数量（Len）、比较（Less）、交换（Swap）。

**（1）使用sort.Interface接口进行排序**

对一系列字符串进行排序时，使用字符串切片（[]string）承载多个字符串。使用 type 关键字，将字符串切片（[]string）定义为自定义类型 MyStringList。为了让 sort 包能识别 MyStringList，能够对 MyStringList 进行排序，就必须让 MyStringList 实现 sort.Interface 接口。

下面是对字符串排序的详细代码（代码1）：

package main

import (

"fmt"

"sort"

)

// 将[]string定义为MyStringList类型

type MyStringList []string

// 实现sort.Interface接口的获取元素数量方法

func (m MyStringList) Len() int {

return len(m)

}

// 实现sort.Interface接口的比较元素方法

func (m MyStringList) Less(i, j int) bool {

return m[i] < m[j]

}

// 实现sort.Interface接口的交换元素方法

func (m MyStringList) Swap(i, j int) {

m[i], m[j] = m[j], m[i]

}

func main() {

// 准备一个内容被打乱顺序的字符串切片

names := MyStringList{

"3. Triple Kill",

"5. Penta Kill",

"2. Double Kill",

"4. Quadra Kill",

"1. First Blood",

}

// 使用sort包进行排序

sort.Sort(names)

// 遍历打印结果

for \_, v := range names {

fmt.Printf("%s\n", v)

}

}

代码输出结果：

1. First Blood

2. Double Kill

3. Triple Kill

4. Quadra Kill

5. Penta Kill

代码说明如下：

第 9 行，接口实现不受限于结构体，任何类型都可以实现接口。要排序的字符串切片 []string 是系统定制好的类型，无法让这个类型去实现 sort.Interface 排序接口。因此，需要将 []string 定义为自定义的类型。

第 12 行，实现获取元素数量的 Len() 方法，返回字符串切片的元素数量。

第 17 行，实现比较元素的 Less() 方法，直接取 m 切片的 i 和 j 元素值进行小于比较，并返回比较结果。

第 22 行，实现交换元素的 Swap() 方法，这里使用 Go 语言的多变量赋值特性实现元素交换。

第 29 行，由于将 []string 定义成 MyStringList 类型，字符串切片初始化的过程等效于下面的写法：

names := []string {

"3. Triple Kill",

"5. Penta Kill",

"2. Double Kill",

"4. Quadra Kill",

"1. First Blood",

}

第 38 行，使用 sort 包的 Sort() 函数，将 names（MyStringList类型）进行排序。排序时，sort 包会通过 MyStringList 实现的 Len()、Less()、Swap() 这 3 个方法进行数据获取和修改。

第 41 行，遍历排序好的字符串切片，并打印结果。

**（2）常见类型的便捷排序**

通过实现 sort.Interface 接口的排序过程具有很强的可定制性，可以根据被排序对象比较复杂的特性进行定制。例如，需要多种排序逻辑的需求就适合使用 sort.Interface 接口进行排序。但大部分情况中，只需要对字符串、整型等进行快速排序。Go 语言中提供了一些固定模式的封装以方便开发者迅速对内容进行排序。

**1) 字符串切片的便捷排序**

sort 包中有一个 StringSlice 类型，定义如下：

type StringSlice []string

func (p StringSlice) Len() int { return len(p) }

func (p StringSlice) Less(i, j int) bool { return p[i] < p[j] }

func (p StringSlice) Swap(i, j int) { p[i], p[j] = p[j], p[i] }

// Sort is a convenience method.

func (p StringSlice) Sort() { Sort(p) }

sort 包中的 StringSlice 的代码与 MyStringList 的实现代码几乎一样。因此，只需要使用 sort 包的 StringSlice 就可以更简单快速地进行字符串排序。将代码1中的排序代码简化后如下所示：

names := sort.StringSlice{

"3. Triple Kill",

"5. Penta Kill",

"2. Double Kill",

"4. Quadra Kill",

"1. First Blood",

}

sort.Sort(names)

简化后，只要两句代码就实现了字符串排序的功能。

**2) 对整型切片进行排序**

除了字符串可以使用 sort 包进行便捷排序外，还可以使用 sort.IntSlice 进行整型切片的排序。sort.IntSlice 的定义如下：

type IntSlice []int

func (p IntSlice) Len() int { return len(p) }

func (p IntSlice) Less(i, j int) bool { return p[i] < p[j] }

func (p IntSlice) Swap(i, j int) { p[i], p[j] = p[j], p[i] }

// Sort is a convenience method.

func (p IntSlice) Sort() { Sort(p) }

sort 包在 sort.Interface 对各类型的封装上还有更进一步的简化，下面使用 sort.Strings 继续对代码1进行简化，代码如下：

names := []string{

"3. Triple Kill",

"5. Penta Kill",

"2. Double Kill",

"4. Quadra Kill",

"1. First Blood",

}

sort.Strings(names)

// 遍历打印结果

for \_, v := range names {

fmt.Printf("%s\n", v)

}

代码说明如下：

第 1 行，需要排序的字符串切片。

第 9 行，使用 sort.Strings 直接对字符串切片进行排序。

**3) sort包内建的类型排序接口一览**

Go 语言中的 sort 包中定义了一些常见类型的排序方法，如下表所示。

sort 包中内建的类型排序接口

类 型 实现 sort.lnterface 的类型 直接排序方法 说 明

字符串（String） StringSlice sort.Strings(a [] string) 字符 ASCII 值升序

整型（int） IntSlice sort.Ints(a []int) 数值升序

双精度浮点（float64） Float64Slice sort.Float64s(a []float64) 数值升序

编程中经常用到的 int32、int64、float32、bool 类型并没有由 sort 包实现，使用时依然需要开发者自己编写。

**（3）对结构体数据进行排序**

除了基本类型的排序，也可以对结构体进行排序。结构体比基本类型更为复杂，排序时不能像数值和字符串一样拥有一些固定的单一原则。结构体的多个字段在排序中可能会存在多种排序的规则，例如，结构体中的名字按字母升序排列，数值按从小到大的顺序排序。一般在多种规则同时存在时，需要确定规则的优先度，如先按名字排序，再按年龄排序等。

**1) 完整实现sort.Interface进行结构体排序**

将一批英雄名单使用结构体定义，英雄名单的结构体中定义了英雄的名字和分类。排序时要求按照英雄的分类进行排序，相同分类的情况下按名字进行排序，详细代码实现过程如下。

结构体排序代码（代码2）：

package main

import (

"fmt"

"sort"

)

// 声明英雄的分类

type HeroKind int

// 定义HeroKind常量, 类似于枚举

const (

None HeroKind = iota

Tank

Assassin

Mage

)

// 定义英雄名单的结构

type Hero struct {

Name string // 英雄的名字

Kind HeroKind // 英雄的种类

}

// 将英雄指针的切片定义为Heros类型

type Heros []\*Hero

// 实现sort.Interface接口取元素数量方法

func (s Heros) Len() int {

return len(s)

}

// 实现sort.Interface接口比较元素方法

func (s Heros) Less(i, j int) bool {

// 如果英雄的分类不一致时, 优先对分类进行排序

if s[i].Kind != s[j].Kind {

return s[i].Kind < s[j].Kind

}

// 默认按英雄名字字符升序排列

return s[i].Name < s[j].Name

}

// 实现sort.Interface接口交换元素方法

func (s Heros) Swap(i, j int) {

s[i], s[j] = s[j], s[i]

}

func main() {

// 准备英雄列表

heros := Heros{

&Hero{"吕布", Tank},

&Hero{"李白", Assassin},

&Hero{"妲己", Mage},

&Hero{"貂蝉", Assassin},

&Hero{"关羽", Tank},

&Hero{"诸葛亮", Mage},

}

// 使用sort包进行排序

sort.Sort(heros)

// 遍历英雄列表打印排序结果

for \_, v := range heros {

fmt.Printf("%+v\n", v)

}

}

代码输出如下：

&{Name:关羽 Kind:1}

&{Name:吕布 Kind:1}

&{Name:李白 Kind:2}

&{Name:貂蝉 Kind:2}

&{Name:妲己 Kind:3}

&{Name:诸葛亮 Kind:3}

代码说明如下：

第 9 行，将 int 声明为 HeroKind 英雄类型，后面会将这个类型当做枚举来使用。

第 13 行，定义一些英雄类型常量，可以理解为枚举的值。

第 26 行，为了方便实现 sort.Interface 接口，将 []\*Hero 定义为 Heros 类型。

第 29 行，Heros 类型实现了 sort.Interface 的 Len() 方法，返回英雄的数量。

第 34 行，Heros 类型实现了 sort.Interface 的 Less() 方法，根据英雄字段的比较结果决定如何排序。

第 37 行，当英雄的分类不一致时，优先按分类的枚举数值从小到大排序。

第 42 行，英雄分类相等的情况下，默认根据英雄的名字字符升序排序。

第 46 行，Heros 类型实现了 sort.Interface 的 Swap() 方法，交换英雄元素的位置。

第 53～60 行，准备一系列英雄数据。

第 63 行，使用 sort 包进行排序。

第 66 行，遍历所有排序完成的英雄数据。

**2) 使用sort.Slice进行切片元素排序**

从 Go 1.8 开始，Go 语言在 sort 包中提供了 sort.Slice() 函数进行更为简便的排序方法。sort.Slice() 函数只要求传入需要排序的数据，以及一个排序时对元素的回调函数，类型为 func(i,j int)bool，sort.Slice() 函数的定义如下：

func Slice(slice interface{}, less func(i, j int) bool)

使用 sort.Slice() 函数，对代码2重新优化的完整代码如下：

package main

import (

"fmt"

"sort"

)

type HeroKind int

const (

None = iota

Tank

Assassin

Mage

)

type Hero struct {

Name string

Kind HeroKind

}

func main() {

heros := []\*Hero{

{"吕布", Tank},

{"李白", Assassin},

{"妲己", Mage},

{"貂蝉", Assassin},

{"关羽", Tank},

{"诸葛亮", Mage},

}

sort.Slice(heros, func(i, j int) bool {

if heros[i].Kind != heros[j].Kind {

return heros[i].Kind < heros[j].Kind

}

return heros[i].Name < heros[j].Name

})

for \_, v := range heros {

fmt.Printf("%+v\n", v)

}

}

第 33 行到第 39 行加粗部分是新添加的 sort.Slice() 及回调函数部分。对比前面的代码，这里去掉了 Heros 及接口实现部分的代码。

使用 sort.Slice() 不仅可以完成结构体切片排序，还可以对各种切片类型进行自定义排序。

### 6 Go语言接口的嵌套组合

在 Go 语言中，不仅结构体与结构体之间可以嵌套，接口与接口间也可以通过嵌套创造出新的接口。

接口与接口嵌套组合而成了新接口，只要接口的所有方法被实现，则这个接口中的所有嵌套接口的方法均可以被调用。

**（1）系统包中的接口嵌套组合**

Go 语言的 io 包中定义了写入器（Writer）、关闭器（Closer）和写入关闭器（WriteCloser）3个接口，代码如下：

type Writer interface {

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

}

type Closer interface {

Close() error

}

type WriteCloser interface {

Writer

Closer

}

代码说明如下：

第 1 行定义了写入器（Writer），如这个接口较为常用，常用于 I/O 设备的数据写入。

第 5 行定义了关闭器（Closer），如有非托管内存资源的对象，需要用关闭的方法来实现资源释放。

第 9 行定义了写入关闭器（WriteCloser），这个接口由 Writer 和 Closer 两个接口嵌入。也就是说，WriteCloser 同时拥有了 Writer 和 Closer 的特性。

**（2）在代码中使用接口嵌套组合**

在代码中使用 io.Writer、io.Closer 和 io.WriteCloser 这 3 个接口时，只需要按照接口实现的规则实现 io.Writer 接口和 io.Closer 接口即可。而 io.WriteCloser 接口在使用时，编译器会根据接口的实现者确认它们是否同时实现了 io.Writer 和 io.Closer 接口，详细实现代码如下：

package main

import (

"io"

)

// 声明一个设备结构

type device struct {

}

// 实现io.Writer的Write()方法

func (d \*device) Write(p []byte) (n int, err error) {

return 0, nil

}

// 实现io.Closer的Close()方法

func (d \*device) Close() error {

return nil

}

func main() {

// 声明写入关闭器, 并赋予device的实例

var wc io.WriteCloser = new(device)

// 写入数据

wc.Write(nil)

// 关闭设备

wc.Close()

// 声明写入器, 并赋予device的新实例

var writeOnly io.Writer = new(device)

// 写入数据

writeOnly.Write(nil)

}

代码说明如下：

第 8 行定义了 device 结构体，用来模拟一个虚拟设备，这个结构会实现前面提到的 3 种接口。

第 12 行，实现了 io.Writer 的 Write() 方法。

第 17 行，实现了 io.Closer 的 Close() 方法。

第 24 行，对 device 实例化，由于 device 实现了 io.WriteCloser 的所有嵌入接口，因此 device 指针就会被隐式转换为 io.WriteCloser 接口。

第 27 行，调用了 wc（io.WriteCloser接口）的 Write() 方法，由于 wc 被赋值 \*device，因此最终会调用 device 的 Write() 方法。

第 30 行，与 27 行类似，最终调用 device 的 Close() 方法。

第 33 行，再次创建一个 device 的实例，writeOnly 是一个 io.Writer 接口，这个接口只有 Write() 方法。

第 36 行，writeOnly 只能调用 Write() 方法，没有 Close() 方法。

### 7 Go语言类型分支（switch判断空接口中变量的类型）

Go 语言的 switch 不仅可以像其他语言一样实现数值、字符串的判断，还有一种特殊的用途——判断一个接口内保存或实现的类型。

**（1）类型断言的书写格式**

switch 实现类型分支时的写法格式如下：

switch 接口变量.(type) {

case 类型1:

// 变量是类型1时的处理

case 类型2:

// 变量是类型2时的处理

…

default:

// 变量不是所有case中列举的类型时的处理

}

对各个部分的说明：

接口变量：表示需要判断的接口类型的变量。

类型1、类型2……：表示接口变量可能具有的类型列表，满足时，会指定 case 对应的分支进行处理。

**（2）使用类型分支判断基本类型**

下面的例子将一个 interface{} 类型的参数传给 printType() 函数，通过 switch 判断 v 的类型，然后打印对应类型的提示，代码如下：

package main

import (

"fmt"

)

func printType(v interface{}) {

switch v.(type) {

case int:

fmt.Println(v, "is int")

case string:

fmt.Println(v, "is string")

case bool:

fmt.Println(v, "is bool")

}

}

func main() {

printType(1024)

printType("pig")

printType(true)

}

代码输出如下：

1024 is int

pig is string

true is bool

代码第 9 行中，v.(type) 就是类型分支的典型写法。通过这个写法，在 switch 的每个 case 中写的将是各种类型分支。

代码经过 switch 时，会判断 v 这个 interface{} 的具体类型从而进行类型分支跳转。

switch 的 default 也是可以使用的，功能和其他的 switch 一致。

### 8 Go语言接口和类型之间的转换

Go 语言中使用接口断言（type assertions）将接口转换成另外一个接口，也可以将接口转换为另外的类型。接口的转换在开发中非常常见，使用也非常频繁。

**（1）类型断言的格式**

类型断言的基本格式如下：

t := i.(T)

其中，i 代表接口变量，T 代表转换的目标类型，t 代表转换后的变量。

如果 i 没有完全实现 T 接口的方法，这个语句将会触发宕机。触发宕机不是很友好，因此上面的语句还有一种写法：

t,ok := i.(T)

这种写法下，如果发生接口未实现时，将会把 ok 置为 false，t 置为 T 类型的 0 值。正常实现时，ok 为 true。这里 ok 可以被认为是：i 接口是否实现 T 类型的结果。

**（2）将接口转换为其他接口**

实现某个接口的类型同时实现了另外一个接口，此时可以在两个接口间转换。

鸟和猪具有不同的特性，鸟可以飞，猪不能飞，但两种动物都可以行走。如果使用结构体实现鸟和猪，让它们具备自己特性的 Fly() 和 Walk() 方法就让鸟和猪各自实现了飞行动物接口（Flyer）和行走动物接口（Walker）。

将鸟和猪的实例创建后，被保存到 interface{} 类型的 map 中。interface{} 类型表示空接口，意思就是这种接口可以保存为任意类型。对保存有鸟或猪的实例的 interface{} 变量进行断言操作，如果断言对象是断言指定的类型，则返回转换为断言对象类型的接口；如果不是指定的断言类型时，断言的第二个参数将返回 false。例如下面的代码：

var obj interface = new(bird)

f, isFlyer := obj.(Flyer)

代码中，new(bird) 产生 \*bird 类型的 bird 实例，这个实例被保存在 interface{} 类型的 obj 变量中。使用 obj.(Flyer) 类型断言，将 obj 转换为 Flyer 接口。f 为转换成功时的 Flyer 接口类型，isFlyer 表示是否转换成功，类型就是 bool。

下面是详细的代码（代码1）：

package main

import "fmt"

// 定义飞行动物接口

type Flyer interface {

Fly()

}

// 定义行走动物接口

type Walker interface {

Walk()

}

// 定义鸟类

type bird struct {

}

// 实现飞行动物接口

func (b \*bird) Fly() {

fmt.Println("bird: fly")

}

// 为鸟添加Walk()方法, 实现行走动物接口

func (b \*bird) Walk() {

fmt.Println("bird: walk")

}

// 定义猪

type pig struct {

}

// 为猪添加Walk()方法, 实现行走动物接口

func (p \*pig) Walk() {

fmt.Println("pig: walk")

}

func main() {

// 创建动物的名字到实例的映射

animals := map[string]interface{}{

"bird": new(bird),

"pig": new(pig),

}

// 遍历映射

for name, obj := range animals {

// 判断对象是否为飞行动物

f, isFlyer := obj.(Flyer)

// 判断对象是否为行走动物

w, isWalker := obj.(Walker)

fmt.Printf("name: %s isFlyer: %v isWalker: %v\n", name, isFlyer, isWalker)

// 如果是飞行动物则调用飞行动物接口

if isFlyer {

f.Fly()

}

// 如果是行走动物则调用行走动物接口

if isWalker {

w.Walk()

}

}

}

代码说明如下：

第 6 行定义了飞行动物的接口。

第 11 行定义了行走动物的接口。

第 16 和 30 行分别定义了鸟和猪两个对象，并分别实现了飞行动物和行走动物接口。

第 41 行是一个 map，映射对象名字和对象实例，实例是鸟和猪。

第 47 行开始遍历 map，obj 为 interface{} 接口类型。

第 50 行中，使用类型断言获得 f，类型为 Flyer 及 isFlyer 的断言成功的判定。

第 52 行中，使用类型断言获得 w，类型为 Walker 及 isWalker 的断言成功的判定。

第 57 和 62 行，根据飞行动物和行走动物两者是否断言成功，调用其接口。

代码输出如下：

name: pig isFlyer: false isWalker: true

pig: walk

name: bird isFlyer: true isWalker: true

bird: fly

bird: walk

**（3）将接口转换为其他类型**

在代码 1 中，可以实现将接口转换为普通的指针类型。例如将 Walker 接口转换为 \*pig 类型，请参考下面的代码：

p1 := new(pig)

var a Walker = p1

p2 := a.(\*pig)

fmt.Printf("p1=%p p2=%p", p1, p2)

对代码的说明如下：

第 3 行，由于 pig 实现了 Walker 接口，因此可以被隐式转换为 Walker 接口类型保存于 a 中。

第 4 行，由于 a 中保存的本来就是 \*pig 本体，因此可以转换为 \*pig 类型。

第 6 行，对比发现，p1 和 p2 指针是相同的。

如果尝试将上面这段代码中的 Walker 类型的 a 转换为 \*bird 类型，将会发出运行时错误，请参考下面的代码：

p1 := new(pig)

var a Walker = p1

p2 := a.(\*bird)

运行时报错：

panic: interface conversion: main.Walker is \*main.pig, not \*main.bird

报错意思是：接口转换时，main.Walker 接口的内部保存的是 \*main.pig，而不是 \*main.bird。

因此，接口在转换为其他类型时，接口内保存的实例对应的类型指针，必须是要转换的对应的类型指针。

**（4）总结**

接口和其他类型的转换可以在 Go 语言中自由进行，前提是已经完全实现。

接口断言类似于流程控制中的 if。但大量类型断言出现时，应使用更为高效的类型分支 switch 特性。

## 八、Go语言[包（package）](http://c.biancheng.net/golang/package/)

### 1 Go语言GOPATH详解（Go语言工作目录）

GOPATH 是 Go 语言中使用的一个环境变量，它使用绝对路径提供项目的工作目录。

工作目录是一个工程开发的相对参考目录，好比当你要在公司编写一套服务器代码，你的工位所包含的桌面、计算机及椅子就是你的工作区。工作区的概念与工作目录的概念也是类似的。如果不使用工作目录的概念，在多人开发时，每个人有一套自己的目录结构，读取配置文件的位置不统一，输出的二进制运行文件也不统一，这样会导致开发的标准不统一，影响开发效率。

GOPATH 适合处理大量 Go 语言源码、多个包组合而成的复杂工程。

提示

C、C++、Java、C# 及其他语言发展到后期，都拥有自己的 IDE（集成开发环境），并且工程（Project）、解决方案（Solution）和工作区（Workspace）等概念将源码和资源组织了起来，方便编译和输出。

使用命令行查看GOPATH信息

在《安装Go语言开发包》一节中我们已经介绍过 Go 语言的安装方法。在安装过 Go 开发包的操作系统中，可以使用命令行查看 Go 开发包的环境变量配置信息，这些配置信息里可以查看到当前的 GOPATH 路径设置情况。在命令行中运行go env后，命令行将提示以下信息：

$ go env

GOARCH="amd64"

GOBIN=""

GOEXE=""

GOHOSTARCH="amd64"

GOHOSTOS="linux"

GOOS="linux"

GOPATH="/home/davy/go"

GORACE=""

GOROOT="/usr/local/go"

GOTOOLDIR="/usr/local/go/pkg/tool/linux\_amd64"

GCCGO="gccgo"

CC="gcc"

GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0"

CXX="g++"

CGO\_ENABLED="1"

CGO\_CFLAGS="-g -O2"

CGO\_CPPFLAGS=""

CGO\_CXXFLAGS="-g -O2"

CGO\_FFLAGS="-g -O2"

CGO\_LDFLAGS="-g -O2"

PKG\_CONFIG="pkg-config"

命令行说明如下：

第 1 行，执行 go env 指令，将输出当前 Go 开发包的环境变量状态。

第 2 行，GOARCH 表示目标处理器架构。

第 3 行，GOBIN 表示编译器和链接器的安装位置。

第 7 行，GOOS 表示目标操作系统。

第 8 行，GOPATH 表示当前工作目录。

第 10 行，GOROOT 表示 Go 开发包的安装目录。

从命令行输出中，可以看到 GOPATH 设定的路径为：/home/davy/go（davy 为笔者的用户名）。

在 Go 1.8 版本之前，GOPATH 环境变量默认是空的。从 Go 1.8 版本开始，Go 开发包在安装完成后，将 GOPATH 赋予了一个默认的目录，参见下表。

GOPATH 在不同平台上的安装路径

平 台 GOPATH 默认值 举 例

Windows 平台 %USERPROFILE%/go C:\Users\用户名\go

Unix 平台 $HOME/go /home/用户名/go

使用GOPATH的工程结构

在 GOPATH 指定的工作目录下，代码总是会保存在 $GOPATH/src 目录下。在工程经过 go build、go install 或 go get 等指令后，会将产生的二进制可执行文件放在 $GOPATH/bin 目录下，生成的中间缓存文件会被保存在 $GOPATH/pkg 下。

如果需要将整个源码添加到版本管理工具（Version Control System，VCS）中时，只需要添加 $GOPATH/src 目录的源码即可。bin 和 pkg 目录的内容都可以由 src 目录生成。

设置和使用GOPATH

本节以 Linux 为演示平台，为大家演示使用 GOPATH 的方法。

**1) 设置当前目录为GOPATH**

选择一个目录，在目录中的命令行中执行下面的指令：

export GOPATH=`pwd`

该指令中的 pwd 将输出当前的目录，使用反引号`将 pwd 指令括起来表示命令行替换，也就是说，使用`pwd`将获得 pwd 返回的当前目录的值。例如，假设你的当前目录是“/home/davy/go”，那么使用`pwd`将获得返回值“/home/davy/go”。

使用 export 指令可以将当前目录的值设置到环境变量 GOPATH中。

**2) 建立GOPATH中的源码目录**

使用下面的指令创建 GOPATH 中的 src 目录，在 src 目录下还有一个 hello 目录，该目录用于保存源码。

mkdir -p src/hello

mkdir 指令的 -p 可以连续创建一个路径。

**3) 添加main.go源码文件**

使用 Linux 编辑器将下面的源码保存为 main.go 并保存到 $GOPATH/src/hello 目录下。

package main

import "fmt"

func main(){

fmt.Println("hello")

}

**4) 编译源码并运行**

此时我们已经设定了 GOPATH，因此在 Go 语言中可以通过 GOPATH 找到工程的位置。

在命令行中执行如下指令编译源码：

go install hello

编译完成的可执行文件会保存在 $GOPATH/bin 目录下。

在 bin 目录中执行 ./hello，命令行输出如下：

hello world

在多项目工程中使用GOPATH

在很多与 Go 语言相关的书籍、文章中描述的 GOPATH 都是通过修改系统全局的环境变量来实现的。然而，根据笔者多年的 Go 语言使用和实践经验及周边朋友、同事的反馈，这种设置全局 GOPATH 的方法可能会导致当前项目错误引用了其他目录的 Go 源码文件从而造成编译输出错误的版本或编译报出一些无法理解的错误提示。

比如说，将某项目代码保存在 /home/davy/projectA 目录下，将该目录设置为 GOPATH。随着开发进行，需要再次获取一份工程项目的源码，此时源码保存在 /home/davy/projectB 目录下，如果此时需要编译 projectB 目录的项目，但开发者忘记设置 GOPATH 而直接使用命令行编译，则当前的 GOPATH 指向的是 /home/davy/projectA 目录，而不是开发者编译时期望的 projectB 目录。编译完成后，开发者就会将错误的工程版本发布到外网。

因此，建议大家无论是使用命令行或者使用集成开发环境编译 Go 源码时，GOPATH 跟随项目设定。在 Jetbrains 公司的 GoLand 集成开发环境（IDE）中的 GOPATH 设置分为全局 GOPATH 和项目 GOPATH，如下图所示。

图：全局和项目GOPATH

图中的 Global GOPATH 代表全局 GOPATH，一般来源于系统环境变量中的 GOPATH；Project GOPATH 代表项目所使用的 GOPATH，该设置会被保存在工作目录的 .idea 目录下，不会被设置到环境变量的 GOPATH 中，但会在编译时使用到这个目录。建议在开发时只填写项目 GOPATH，每一个项目尽量只设置一个 GOPATH，不使用多个 GOPATH 和全局的 GOPATH。

提示

Visual Studio 早期在设计时，允许 C++ 语言在全局拥有一个包含路径。当一个工程多个版本的编译，或者两个项目混杂有不同的共享全局包含时，会发生难以察觉的错误。在新版本 Visual Studio 中已经废除了这种全局包含的路径设计，并建议开发者将包含目录与项目关联。

Go 语言中的 GOPATH 也是一种类似全局包含的设计，因此鉴于 Visual Studio 在设计上的失误，建议开发者不要设置全局的 GOPATH，而是随项目设置 GOPATH。

### 2 Go语言package（创建包）

包（package）是多个 Go 源码的集合，是一种高级的代码复用方案，Go 语言默认为我们提供了很多包，如 fmt、os、io 包等，开发者可以根据自己的需要创建自己的包。

包要求在同一个目录下的所有文件的第一行添加如下代码，以标记该文件归属的包：

package 包名

包的特性如下：

一个目录下的同级文件归属一个包。

包名可以与其目录不同名。

包名为 main 的包为应用程序的入口包，编译源码没有 main 包时，将无法编译输出可执行的文件。

### 3 Go语言导出包中的标识符——让外部访问包的类型和值

在 Go 语言中，如果想在一个包里引用另外一个包里的标识符（如类型、变量、常量等）时，必须首先将被引用的标识符导出，将要导出的标识符的首字母大写就可以让引用者可以访问这些标识符了。

**（1）导出包内标识符**

下面代码中包含一系列未导出标识符，它们的首字母都为小写，这些标识符可以在包内自由使用，但是包外无法访问它们，代码如下：

package mypkg

var myVar = 100

const myConst = "hello"

type myStruct struct {

}

将 myStruct 和 myConst 首字母大写，导出这些标识符，修改后代码如下：

package mypkg

var myVar = 100

const MyConst = "hello"

type MyStruct struct {

}

此时，MyConst 和 MyStruct 可以被外部访问，而 myVar 由于首字母是小写，因此只能在 mypkg 包内使用，不能被外部包引用。

**（2）导出结构体及接口成员**

在被导出的结构体或接口中，如果它们的字段或方法首字母是大写，外部可以访问这些字段和方法，代码如下：

type MyStruct struct {

// 包外可以访问的字段

ExportedField int

// 仅限包内访问的字段

privateField int

}

type MyInterface interface {

// 包外可以访问的方法

ExportedMethod()

// 仅限包内访问的方法

privateMethod()

}

在代码中，MyStruct 的 ExportedField 和 MyInterface 的 ExportedMethod() 可以被包外访问。

### 4 Go语言import导入包——在代码中使用其他的代码

要引用其他包的标识符，可以使用 import 关键字，导入的包名使用双引号包围，包名是从 GOPATH 开始计算的路径，使用/进行路径分隔。

**（1）默认导入的写法**

导入有两种基本格式，即单行导入和多行导入，两种导入方法的导入代码效果是一致的。

**1) 单行导入**

单行导入格式如下：

import "包1"

import "包2"

**2) 多行导入**

当多行导入时，包名在 import 中的顺序不影响导入效果，格式如下：

import(

"包1"

"包2"

…

)

参考代码 8-1 的例子来理解 import 的机制。

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

代码 8-1 的目录层次如下：

.

└── src

└── chapter08

└── importadd

├── main.go

└── mylib

└── add.go

代码8-1　加函数（具体文件：…/chapter08/importadd/mylib/add.go）

package mylib

func Add(a, b int) int {

return a + b

}

第 3 行中的 Add() 函数以大写 A 开头，表示将 Add() 函数导出供包外使用。当首字母小写时，为包内使用，包外无法引用到。

add.go 在 mylib 文件夹下，习惯上将文件夹的命名与包名一致，命名为 mylib 包。

代码8-2　导入包（具体文件：…/chapter08/importadd/main.go）

package main

import (

"chapter08/importadd/mylib"

"fmt"

)

func main() {

fmt.Println(mylib.Add(1, 2))

}

代码说明如下：

第 4 行，导入 chapter08/importadd/mylib 包。

第 9 行，使用 mylib 作为包名，并引用 Add() 函数调用。

在命令行中运行下面代码：

export GOPATH=/home/davy/golangbook/code

go install chapter08/importadd

$GOPATH/bin/importadd

命令说明如下：

第 1 行，根据你的 GOPATH 不同，设置 GOPATH。

第 2 行，使用 go install 指令编译并安装 chapter08/code8-1 到 GOPATH 的 bin 目录下。

第 3 行，执行 GOPATH 的 bin 目录下的可执行文件 code8-1。

运行代码，输出结果如下：

3

**（2）导入包后自定义引用的包名**

在默认导入包的基础上，在导入包路径前添加标识符即可形成自定义引用包，格式如下：

customName "path/to/package"

其中，path/to/package 为要导入的包路径，customName 为自定义的包名。

在 code8-1 的基础上，在 mylib 导入的包名前添加一个标识符，代码如下：

package main

import (

renameLib "chapter08/importadd/mylib"

"fmt"

)

func main() {

fmt.Println(renameLib.Add(1, 2))

}

代码说明如下：

第 4 行，将 chapter08/importadd/mylib 包导入，并且使用 renameLib 进行引用。

第 9 行，使用 renameLib 调用 chapter08/importadd/mylib 包中的 Add() 函数。

**（3）匿名导入包——只导入包但不使用包内类型和数值**

如果只希望导入包，而不使用任何包内的结构和类型，也不调用包内的任何函数时，可以使用匿名导入包，格式如下：

import (

\_ "path/to/package"

)

其中，path/to/package 表示要导入的包名，下画线\_表示匿名导入包。

匿名导入的包与其他方式导入包一样会让导入包编译到可执行文件中，同时，导入包也会触发 init() 函数调用。

**（4）包在程序启动前的初始化入口：init**

在某些需求的设计上需要在程序启动时统一调用程序引用到的所有包的初始化函数，如果需要通过开发者手动调用这些初始化函数，那么这个过程可能会发生错误或者遗漏。我们希望在被引用的包内部，由包的编写者获得代码启动的通知，在程序启动时做一些自己包内代码的初始化工作。

例如，为了提高数学库计算三角函数的执行效率，可以在程序启动时，将三角函数的值提前在内存中建成索引表，外部程序通过查表的方式迅速获得三角函数的值。但是三角函数索引表的初始化函数的调用不希望由每一个外部使用三角函数的开发者调用，如果在三角函数的包内有一个机制可以告诉三角函数包程序何时启动，那么就可以解决初始化的问题。

Go 语言为以上问题提供了一个非常方便的特性：init() 函数。

init() 函数的特性如下

每个源码可以使用 1 个 init() 函数。

init() 函数会在程序执行前（main() 函数执行前）被自动调用。

调用顺序为 main() 中引用的包，以深度优先顺序初始化。

例如，假设有这样的包引用关系：main→A→B→C，那么这些包的 init() 函数调用顺序为：

C.init→B.init→A.init→main

说明：

同一个包中的多个 init() 函数的调用顺序不可预期。

init() 函数不能被其他函数调用。

**（5）理解包导入后的init()函数初始化顺序**

Go 语言包会从 main 包开始检查其引用的所有包，每个包也可能包含其他的包。Go 编译器由此构建出一个树状的包引用关系，再根据引用顺序决定编译顺序，依次编译这些包的代码。

在运行时，被最后导入的包会最先初始化并调用 init() 函数。

通过下面的代码理解包的初始化顺序。

代码8-3　包导入初始化顺序入口（…/chapter08/pkginit/main.go）

package main

import "chapter08/code8-2/pkg1"

func main() {

pkg1.ExecPkg1()

}

代码说明如下：

第 3 行，导入 pkg1 包。

第 7 行，调用 pkg1 包的 ExecPkg1() 函数。

代码8-4　包导入初始化顺序pkg1（…/chapter08/pkginit/pkg1/pkg1.go）

package pkg1

import (

"chapter08/code8-2/pkg2"

"fmt"

)

func ExecPkg1() {

fmt.Println("ExecPkg1")

pkg2.ExecPkg2()

}

func init() {

fmt.Println("pkg1 init")

}

代码说明如下：

第 4 行，导入 pkg2 包。

第 8 行，声明 ExecPkg1() 函数。

第 12 行，调用 pkg2 包的 ExecPkg2() 函数。

第 15 行，在 pkg1 包初始化时，打印 pkg1 init。

代码8-5　包导入初始化顺序pkg2（…/chapter08/pkginit/pkg2/pkg2.go）

package pkg2

import "fmt"

func ExecPkg2() {

fmt.Println("ExecPkg2")

}

func init() {

fmt.Println("pkg2 init")

}

代码说明如下：

第 5 行，声明 ExecPkg2() 函数。

第 10 行，在 pkg2 包初始化时，打印 pkg2 init。

执行代码，输出如下：

pkg2 init

pkg1 init

ExecPkg1

ExecPkg2

### 5 Go语言工厂模式自动注册——管理多个包的结构体

本例利用包的 init 特性，将 cls1 和 cls2 两个包注册到工厂，使用字符串创建这两个注册好的结构实例。

完整代码的结构如下：

.

└── src

└── chapter08

└── clsfactory

├── main.go

└── base

└── factory.go

└── cls1

└── reg.go

└── cls2

└── reg.go

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

类工厂（具体文件：…/chapter08/clsfactory/base/factory.go）

package base

// 类接口

type Class interface {

Do()

}

var (

// 保存注册好的工厂信息

factoryByName = make(map[string]func() Class)

)

// 注册一个类生成工厂

func Register(name string, factory func() Class) {

factoryByName[name] = factory

}

// 根据名称创建对应的类

func Create(name string) Class {

if f, ok := factoryByName[name]; ok {

return f()

} else {

panic("name not found")

}

}

这个包叫base，负责处理注册和使用工厂的基础代码，该包不会引用任何外部的包。

以下是对代码的说明：

第 4 行定义了“产品”：类。

第 10 行使用了一个 map 保存注册的工厂信息。

第 14 行提供给工厂方注册使用，所谓的“工厂”，就是一个定义为func() Class的普通函数，调用此函数，创建一个类实例，实现的工厂内部结构体会实现 Class 接口。

第 19 行定义通过名字创建类实例的函数，该函数会在注册好后调用。

第 20 行在已经注册的信息中查找名字对应的工厂函数，找到后，在第 21 行调用并返回接口。

第 23 行是如果创建的名字没有找到时，报错。

类1及注册代码（具体文件：…/chapter08/clsfactory/cls1/reg.go）

package cls1

import (

"chapter08/clsfactory/base"

"fmt"

)

// 定义类1

type Class1 struct {

}

// 实现Class接口

func (c \*Class1) Do() {

fmt.Println("Class1")

}

func init() {

// 在启动时注册类1工厂

base.Register("Class1", func() base.Class {

return new(Class1)

})

}

上面的代码展示了Class1的工厂及产品定义过程。

第 9～15 行定义 Class1 结构，该结构实现了 base 中的 Class 接口。

第 20 行，Class1 结构的实例化过程叫 Class1 的工厂，使用 base.Register() 函数在 init() 函数被调用时与一个字符串关联，这样，方便以后通过名字重新调用该函数并创建实例。

类2及注册代码（具体文件：…/chapter08/clsfactory/cls2/reg.go）

package cls2

import (

"chapter08/clsfactory/base"

"fmt"

)

// 定义类2

type Class2 struct {

}

// 实现Class接口

func (c \*Class2) Do() {

fmt.Println("Class2")

}

func init() {

// 在启动时注册类2工厂

base.Register("Class2", func() base.Class {

return new(Class2)

})

}

Class2 的注册与 Class1 的定义和注册过程类似。

类工程主流程（具体文件：…/chapter08/clsfactory/main.go）

package main

import (

"chapter08/clsfactory/base"

\_ "chapter08/clsfactory/cls1" // 匿名引用cls1包, 自动注册

\_ "chapter08/clsfactory/cls2" // 匿名引用cls2包, 自动注册

)

func main() {

// 根据字符串动态创建一个Class1实例

c1 := base.Create("Class1")

c1.Do()

// 根据字符串动态创建一个Class2实例

c2 := base.Create("Class2")

c2.Do()

}

下面是对代码的说明：

第 5 和第 6 行使用匿名引用方法导入了 cls1 和 cls2 两个包。在 main() 函数调用前，这两个包的 init() 函数会被自动调用，从而自动注册 Class1 和 Class2。

第 12 和第 16 行，通过 base.Create() 方法查找字符串对应的类注册信息，调用工厂方法进行实例创建。

第 13 和第 17 行，调用类的方法。

执行下面的指令进行编译：

export GOPATH=/home/davy/golangbook/code

go install chapter08/clsfactory

$GOPATH/bin/clsfactory

代码输出如下：

Class1

Class2

## 九、Go语言[并发](http://c.biancheng.net/golang/concurrent/)

### 1 goroutine简介

**（1）并发与并行简介**

并行(parallel)：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的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。

并行在多处理器系统中存在，而并发可以在单处理器和多处理器系统中都存在，并发能够在单处理器系统中存在是因为并发是并行的假象，并行要求程序能够同时执行多个操作，而并发只是要求程序假装同时执行多个操作（每个小时间片执行一个操作，多个操作快速切换执行）。

**（2）Coroutine简介**

Coroutine（协程）是一种用户态的轻量级线程，特点如下：

A、轻量级线程

B、非抢占式多任务处理，由协程主动交出控制权。

C、编译器/解释器/虚拟机层面的任务

D、多个协程可能在一个或多个线程上运行。

E、子程序是协程的一个特例。

不同语言对协程的支持：

A、C++通过Boost.Coroutine实现对协程的支持

B、Java不支持

C、Python通过yield关键字实现协程，Python3.5开始使用async def对原生协程的支持

**（3）goroutine简介**

在Go语言中，只需要在函数调用前加上关键字go即可创建一个并发任务单元，新建的任务会被放入队列中，等待调度器安排。

进程在启动的时候，会创建一个主线程，主线程结束时，程序进程将终止，因此，进程至少有一个线程。main函数里，必须让主线程等待，确保进程不会被终止。

go语言中并发指的是让某个函数独立于其它函数运行的能力，一个goroutine是一个独立的工作单元，Go的runtime（运行时）会在逻辑处理器上调度goroutine来运行，一个逻辑处理器绑定一个操作系统线程，因此goroutine不是线程，是一个协程。

进程：一个程序对应一个独立程序空间

线程：一个执行空间，一个进程可以有多个线程

逻辑处理器：执行创建的goroutine，绑定一个线程

调度器：Go运行时中的，分配goroutine给不同的逻辑处理器

全局运行队列：所有刚创建的goroutine队列

本地运行队列：逻辑处理器的goroutine队列

当创建一个goroutine后，会先存放在全局运行队列中，等待Go运行时的调度器进行调度，把goroutine分配给其中的一个逻辑处理器，并放到逻辑处理器对应的本地运行队列中，最终等着被逻辑处理器执行即可。

Go的并发是管理、调度、执行goroutine的方式。

默认情况下，Go默认会给每个可用的物理处理器都分配一个逻辑处理器。

可以在程序开头使用runtime.GOMAXPROCS(n)设置逻辑处理器的数量。

如果需要设置逻辑处理器的数量，一般采用如下代码设置：

runtime.GOMAXPROCS(runtime.NumCPU())

对于并发，Go语言本身自己实现的调度，对于并行，与物理处理器的核数有关，多核就可以并行并发，单核只能并发。

**（4）goroutinue使用示例**

在Go语言中，只需要在函数调用前加上关键字go即可创建一个并发任务单元，新建的任务会被放入队列中，等待调度器安排。

package main

import (

"fmt"

"sync"

)

func main(){

var wg sync.WaitGroup

wg.Add(2)

go func() {

defer wg.Done()

for i := 0; i < 10000; i++ {

fmt.Printf("Hello,Go.This is %d\n", i)

}

}()

go func() {

defer wg.Done()

for i := 0; i < 10000; i++ {

fmt.Printf("Hello,World.This is %d\n", i)

}

}()

wg.Wait()

}

sync.WaitGroup是一个计数的信号量，使main函数所在主线程等待两个goroutine执行完成后再结束，否则两个goroutine还在运行时，主线程已经结束。

sync.WaitGroup使用非常简单，使用Add方法设设置计数器为2，每一个goroutine的函数执行完后，调用Done方法减1。Wait方法表示如果计数器大于0，就会阻塞，main函数会一直等待2个goroutine完成再结束。

**（5）goroutine的本质**

goroutine是轻量级的线程，占用的资源非常小(Go将每个goroutine stack的size默认设置为2k)线程的切换由操作系统控制，而goroutine的切换则由用户控制。

goroutinue本质上是协程。

goroutinue可以实现并行，即多个goroutinue可以在多个处理器同时运行，而协程同一时刻只能在一个处理器上运行。

goroutine之间的通信是通过channel，而协程的通信是通过yield和resume()操作。

### 2 goroutine调度机制

**（1）线程调度模型**

高级语言对内核线程的封装实现通常有三种线程调度模型：

A、N:1模型。N个用户空间线程在1个内核空间线程上运行，优势是上下文切换非常快但无法利用多核系统的优点。

B、1:1模型。1个内核空间线程运行一个用户空间线程，充分利用了多核系统的优势但上下文切换非常慢，因为每一次调度都会在用户态和内核态之间切换。

C、M:N模型。每个用户线程对应多个内核空间线程，同时也可以一个内核空间线程对应多个用户空间线程，使用任意个内核模型管理任意个goroutine，但缺点是调度的复杂性。

**（2）Go调度器简介**

Go的最小调度单元为goroutine，但操作系统最小的调度单元依然是线程，所以go调度器（go scheduler）要做的工作是如何将众多的goroutine放在有限的线程上进行高效而公平的调度。

操作系统的调度不失为高效和公平，比如CFS调度算法。go引入goroutine的核心原因是goroutine轻量级，无论是从进程到线程，还是从线程到goroutine，其核心都是为了使调度单元更加轻量级，可以轻易创建几万几十万的goroutine而不用担心内存耗尽等问题。go引入goroutine试图在语言内核层做到足够高性能得同时（充分利用多核优势、使用epoll高效处理网络／IO、实现垃圾回收等机制）尽量简化编程。

**（3）Go调度器实现原理**

Go 1.1开始，Go scheduler实现了M:N的G-P-M线程调度模型，即任意数量的用户态goroutine可以运行在任意数量的内核空间线程线程上，不仅可以使上线文切换更加轻量级，又可以充分利用多核优势。

Go语言开发（九）、Go语言并发编程

为了实现M：N线程调度机制，Go引入了3个结构体：

M：操作系统的内核空间线程

G：goroutine对象，G结构体包含调度一个goroutine所需要的堆栈和instruction pointer（IP指令指针），以及其它一些重要的调度信息。每次go调用的时候，都会创建一个G对象。

P：Processor，调度的上下文，实现M：N调度模型的关键，M必须拿到P才能对G进行调度，P限定了go调度goroutine的最大并发度。每一个运行的M都必须绑定一个P。

P的个数是GOMAXPROCS（最大256），启动时固定，一般不修改； M的个数和P的个数不一定相同（会有休眠的M或者不需要太多的M）；每一个P保存着本地G任务队列，也能使用全局G任务队列。

Go语言开发（九）、Go语言并发编程

全局G任务队列会和各个本地G任务队列按照一定的策略互相交换。

P是用一个全局数组（255）来保存的，并且维护着一个全局的P空闲链表。

每次调用go的时候，都会：

A、创建一个G对象，加入到本地队列或者全局队列

B、如果有空闲的P，则创建一个M

C、M会启动一个底层线程，循环执行能找到的G任务

D、G任务的执行顺序是先从本地队列找，本地没有则从全局队列找（一次性转移(全局G个数/P个数）个，再去其它P中找（一次性转移一半）。

E、G任务执行是按照队列顺序（即调用go的顺序）执行的。

创建一个M过程如下：

A、先找到一个空闲的P，如果没有则直接返回。

B、调用系统API创建线程，不同的操作系统调用方法不一样。

C、 在创建的线程里循环执行G任务

如果一个系统调用或者G任务执行太长，会一直占用内核空间线程，由于本地队列的G任务是顺序执行的，其它G任务就会阻塞。因此，Go程序启动的时候，会专门创建一个线程sysmon，用来监控和管理，sysmon内部是一个循环：

A、记录所有P的G任务计数schedtick，schedtick会在每执行一个G任务后递增。

B、如果检查到 schedtick一直没有递增，说明P一直在执行同一个G任务，如果超过一定的时间（10ms），在G任务的栈信息里面加一个标记。

C、G任务在执行的时候，如果遇到非内联函数调用，就会检查一次标记，然后中断自己，把自己加到队列末尾，执行下一个G。

D、如果没有遇到非内联函数（有时候正常的小函数会被优化成内联函数）调用，会一直执行G任务，直到goroutine自己结束；如果goroutine是死循环，并且GOMAXPROCS=1，阻塞。

**（4）抢占式调度**

Go没有时间片的概念。如果某个G没有进行system call调用、没有进行I/O操作、没有阻塞在一个channel操作上，M通过抢占式调度让长任务G停下来并调度下一个G。

除非极端的无限循环或死循环，否则只要G调用函数，Go runtime就有抢占G的机会。Go程序启动时，Go runtime会启动一个名为sysmon的M(一般称为监控线程)，sysmon无需绑定P即可运行。sysmon是GO程序启动时创建的一个用于监控管理的线程。

sysmon每20us~10ms启动一次，sysmon主要完成如下工作：

A、释放闲置超过5分钟的span物理内存；

B、如果超过2分钟没有垃圾回收，强制执行；

C、将长时间未处理的netpoll结果添加到任务队列；

D、向长时间运行的G任务发出抢占调度；

E、收回因syscall长时间阻塞的P；

如果一个G任务运行10ms，sysmon就会认为其运行时间太久而发出抢占式调度的请求。一旦G的抢占标志位被设为true，那么待G下一次调用函数或方法时，runtime便可以将G抢占，并移出运行状态，放入P的local runq中，等待下一次被调度。

### 3 runtime包

**（1）Gosched**

runtime.Gosched()用于让出CPU时间片，让出当前goroutine的执行权限，调度器安排其它等待的任务运行，并在下次某个时候从该位置恢复执行。

**（2）Goexit**

调用runtime.Goexit()将立即终止当前goroutine执⾏，调度器确保所有已注册defer延迟调用被执行。

**（3）GOMAXPROCS**

调用runtime.GOMAXPROCS()用来设置可以并行计算的CPU核数的最大值，并返回设置前的值。

### 4 Channel通道

**（1）Channel简介**

Channel是goroutine之间通信的通道，用于goroutine之间发消息和接收消息。Channel是一种引用类型的数据，可以作为参数，也可以作为返回值。

**（2）Channel的创建**

channel声明使用chan关键字，channel的创建需要指定通道中发送和接收数据的类型。

使用make来建立一个信道:

var channel chan int = make(chan int)

// 或channel := make(chan int)

make有第二个参数，用于指定通道的大小。

**（3）Channel的操作**

//发送数据：写

channel<- data

//接收数据：读

data := <- channel

关闭通道：发送方关闭通道，用于通知接收方已经没有数据

关闭通道后，其它goroutine访问通道获取数据时，得到零值和false

有条件结束死循环：

for{

v ,ok := <- chan

if ok== false{

//通道已经关闭。。

break

}

}

//循环从通道中获取数据，直到通道关闭。

for v := range channel{

//从通道读取数据

}

Channel使用示例如下：

package main

import (

"fmt"

"time"

)

type Person struct {

name string

age uint8

address Address

}

type Address struct {

city string

district string

}

func SendMessage(person \*Person, channel chan Person){

go func(person \*Person, channel chan Person) {

fmt.Printf("%s send a message.\n", person.name)

channel<-\*person

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

channel<- \*person

}

close(channel)

fmt.Println("channel is closed.")

}(person, channel)

}

func main() {

channel := make(chan Person,1)

harry := Person{

"Harry",

30,

Address{"London","Oxford"},

}

go SendMessage(&harry, channel)

data := <-channel

fmt.Printf("main goroutine receive a message from %s.\n", data.name)

for {

i, ok := <-channel

time.Sleep(time.Second)

if !ok {

fmt.Println("channel is empty.")

break

}else{

fmt.Printf("receive %s\n",i.name)

}

}

}

结果如下：

Harry send a message.

main goroutine receive a message from Harry.

receive Harry

receive Harry

receive Harry

channel is closed.

receive Harry

receive Harry

channel is empty.

Go运行时系统并没有在通道channel被关闭后立即把false作为相应接收操作的第二个结果，而是等到接收端把已在通道中的所有元素值都接收到后才这样做，确保在发送端关闭通道的安全性。

被关闭的通道会禁止数据流入, 是只读的，仍然可以从关闭的通道中取出数据，但不能再写入数据。

给一个nil的channel发送数据，造成永远阻塞 ；从一个nil的channel接收数据，造成永远阻塞。给一个已经关闭的channel发送数据，引起panic ；

从一个已经关闭的channel接收数据，返回带缓存channel中缓存的值，如果通道中无缓存，返回0。

**（4）无缓冲通道**

make创建通道时，默认没有第二个参数，通道的大小为0，称为无缓冲通道。

无缓冲的通道是指通道的大小为0，即通道在接收前没有能力保存任何值，无缓冲通道发送goroutine和接收gouroutine必须是同步的，如果没有同时准备好，先执行的操作就会阻塞等待，直到另一个相对应的操作准备好为止。无缓冲通道也称为同步通道。

无缓冲的信道永远不会存储数据，只负责数据的流通。从无缓冲信道取数据，必须要有数据流进来才可以，否则当前goroutine会阻塞；数据流入无缓冲信道, 如果没有其它goroutine来拿取走数据，那么当前goroutine会阻塞。

package main

import (

"fmt"

)

func main() {

ch := make(chan int)

go func() {

var sum int = 0

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

sum += i

}

//发送数据到通道

ch <- sum

}()

//从通道接收数据

fmt.Println(<-ch)

}

在计算sum和的goroutine没有执行完，将值赋发送到ch通道前，fmt.Println(<-ch)会一直阻塞等待，main函数所在的主goroutine就不会终止，只有当计算和的goroutine完成后，并且发送到ch通道的操作准备好后，main函数的<-ch会接收计算好的值，然后打印出来。

无缓存通道的发送数据和读取数据的操作不能放在同一个协程中，防止发生死锁。通常，先创建一个goroutine对通道进行操作，此时新创建goroutine会阻塞，然后再在主goroutine中进行通道的反向操作，实现goroutine解锁，即必须goroutine在前，解锁goroutine在后。

**（5）有缓冲通道**

make创建通道时，指定通道的大小时，称为有缓冲通道。

对于带缓存通道，只要通道中缓存不满，可以一直向通道中发送数据，直到缓存已满；同理只要通道中缓存不为０，可以一直从通道中读取数据，直到通道的缓存变为０才会阻塞。

相对于不带缓存通道，带缓存通道不易造成死锁，可以同时在一个goroutine中放心使用。

带缓存通道不仅可以流通数据，还可以缓存数据，当带缓存通道达到满的状态的时候才会阻塞，此时带缓存通道不能再承载更多的数据。

带缓存通道是先进先出的。

**（6）单向通道**

对于某些特殊的场景，需要限制一个通道只可以接收，不能发送；限制一个通道只能发送，不能接收。只能单向接收或发送的通道称为单向通道。

定义单向通道只需要在定义的时候，带上<-即可。

var send chan<- int //只能发送

var receive <-chan int //只能接收

<-操作符的位置在后面只能发送，对应发送操作；<-操作符的位置在前面只能接收，对应接收操作。

单向通道通常用于函数或者方法的参数。

### 5 channel应用

**（1）广播功能实现**

当一个通道关闭时, 所有对此通道的读取的goroutine都会退出阻塞。

package main

import (

"fmt"

"time"

)

func notify(id int, channel chan int){

<-channel//接收到数据或通道关闭时退出阻塞

fmt.Printf("%d receive a message.\n", id)

}

func broadcast(channel chan int){

fmt.Printf("Broadcast:\n")

close(channel)//关闭通道

}

func main(){

channel := make(chan int,1)

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

go notify(i,channel)

}

go broadcast(channel)

time.Sleep(time.Second)

}

**（2）select使用**

select用于在多个channel上同时进行侦听并收发消息，当任何一个case满足条件时即执行，如果没有可执行的case则会执行default的case，如果没有指定default case，则会阻塞程序。select的语法如下：

select {

case communication clause :

statement(s);

case communication clause :

statement(s);

/\*可以定义任意数量的 case \*/

default : /\*可选 \*/

statement(s);

}

Select多路复用中：

A、每个case都必须是一次通信

B、所有channel表达式都会被求值

C、所有被发送的表达式都会被求值

D、如果任意某个通信可以进行，它就执行；其它被忽略。

E、如果有多个case都可以运行，Select会随机公平地选出一个执行。其它不会执行。

F、否则，如果有default子句，则执行default语句。如果没有default子句，select将阻塞，直到某个通信可以运行；Go不会重新对channel或值进行求值。

package main

import (

"fmt"

"time"

)

func doWork(channels \*[10]chan int){

for {

select {

case x1 := <-channels[0]:

fmt.Println("receive x1: ",x1)

case x2 := <-channels[1]:

fmt.Println("receive x2: ",x2)

case x3 := <-channels[2]:

fmt.Println("receive x3: ",x3)

case x4 := <-channels[3]:

fmt.Println("receive x4: ",x4)

case x5 := <-channels[4]:

fmt.Println("receive x5: ",x5)

case x6 := <-channels[5]:

fmt.Println("receive x6: ",x6)

case x7 := <-channels[6]:

fmt.Println("receive x7: ",x7)

case x8 := <-channels[7]:

fmt.Println("receive x8: ",x8)

case x9 := <-channels[8]:

fmt.Println("receive x9: ",x9)

case x10 := <-channels[9]:

fmt.Println("receive x10: ",x10)

}

}

}

func main(){

var channels [10]chan int

go doWork(&channels)

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

channels[i] = make(chan int,1)

channels[i]<- i

}

time.Sleep(time.Second\*5)

}

结果如下：

receive x4: 3

receive x10: 9

receive x9: 8

receive x5: 4

receive x2: 1

receive x7: 6

receive x8: 7

receive x1: 0

receive x3: 2

receive x6: 5

### 6 死锁

Go程序中死锁是指所有的goroutine在等待资源的释放。

通常，死锁的报错信息如下：

fatal error: all goroutines are asleep - deadlock!

Goroutine死锁产生的原因如下：

A、只在单一的goroutine里操作无缓冲信道，一定死锁

B、非缓冲信道上如果发生流入无流出，或者流出无流入，会导致死锁

因此，解决死锁的方法有：

A、取走无缓冲通道的数据或是发送数据到无缓冲通道

B、使用缓冲通道

### 7 等待组

除了可以使用通道（channel）和互斥锁进行两个并发程序间的同步外，还可以使用等待组进行多个任务的同步，等待组可以保证在并发环境中完成指定数量的任务  
等待组有下面几个方法可用，如下表所示。

|  |  |
| --- | --- |
| 等待组的方法 | |
| 方法名 | 功能 |
| (wg \* WaitGroup) Add(delta int) | 等待组的计数器 +1 |
| (wg \*WaitGroup) Done() | 等待组的计数器 -1 |
| (wg \*WaitGroup) Wait() | 当等待组计数器不等于 0 时阻塞直到变 0。 |

等待组内部拥有一个计数器，计数器的值可以通过方法调用实现计数器的增加和减少。当我们添加了 N 个并发任务进行工作时，就将等待组的计数器值增加 N。每个任务完成时，这个值减 1。同时，在另外一个 goroutine 中等待这个等待组的计数器值为 0 时，表示所有任务已经完成。  
  
下面的代码演示了这一过程：

package main

import (

"fmt"

"net/http"

"sync"

)

func main() {

// 声明一个等待组

var wg sync.WaitGroup

// 准备一系列的网站地址

var urls = []string{

"http://www.github.com/",

"https://www.qiniu.com/",

"https://www.golangtc.com/",

}

// 遍历这些地址

for \_, url := range urls {

// 每一个任务开始时, 将等待组增加1

wg.Add(1)

// 开启一个并发

go func(url string) {

// 使用defer, 表示函数完成时将等待组值减1

defer wg.Done()

// 使用http访问提供的地址

\_, err := http.Get(url)

// 访问完成后, 打印地址和可能发生的错误

fmt.Println(url, err)

// 通过参数传递url地址

}(url)

}

// 等待所有的任务完成

wg.Wait()

fmt.Println("over")

}

代码说明如下：

第 12 行，声明一个等待组，对一组等待任务只需要一个等待组，而不需要每一个任务都使用一个等待组。

第 15 行，准备一系列可访问的网站地址的字符串切片。

第 22 行，遍历这些字符串切片。

第 25 行，将等待组的计数器加1，也就是每一个任务加 1。

第 28 行，将一个匿名函数开启并发。

第 31 行，在匿名函数结束时会执行这一句以表示任务完成。wg.Done() 方法等效于执行 wg.Add(-1)。

第 34 行，使用 http 包提供的 Get() 函数对 url 进行访问，Get() 函数会一直阻塞直到网站响应或者超时。

第 37 行，在网站响应和超时后，打印这个网站的地址和可能发生的错误。

第 40 行，这里将 url 通过 goroutine 的参数进行传递，是为了避免 url 变量通过闭包放入匿名函数后又被修改的问题。

第 44 行，等待所有的网站都响应或者超时后，任务完成，Wait 就会停止阻塞。

### 8 Go语言互斥锁（sync.Mutex）和读写互斥锁（sync.RWMutex）

互斥锁是一种常用的控制共享资源访问的方法，它能够保证同时只有一个 goroutine 可以访问共享资源。在 Go 程序中的使用非常简单，参见下面的代码：

package main

import (

"fmt"

"sync"

)

var (

// 逻辑中使用的某个变量

count int

// 与变量对应的使用互斥锁

countGuard sync.Mutex

)

func GetCount() int {

// 锁定

countGuard.Lock()

// 在函数退出时解除锁定

defer countGuard.Unlock()

return count

}

func SetCount(c int) {

countGuard.Lock()

count = c

countGuard.Unlock()

}

func main() {

// 可以进行并发安全的设置

SetCount(1)

// 可以进行并发安全的获取

fmt.Println(GetCount())

}

代码说明如下：

第 10 行是某个逻辑步骤中使用到的变量，无论是包级的变量还是结构体成员字段，都可以。

第 13 行，一般情况下，建议将互斥锁的粒度设置得越小越好，降低因为共享访问时等待的时间。这里笔者习惯性地将互斥锁的变量命名为以下格式：

变量名+Guard

以表示这个互斥锁用于保护这个变量。

第 16 行是一个获取 count 值的函数封装，通过这个函数可以并发安全的访问变量 count。

第 19 行，尝试对 countGuard 互斥量进行加锁。一旦 countGuard 发生加锁，如果另外一个 goroutine 尝试继续加锁时将会发生阻塞，直到这个 countGuard 被解锁。

第 22 行使用 defer 将 countGuard 的解锁进行延迟调用，解锁操作将会发生在 GetCount() 函数返回时。

第 27 行在设置 count 值时，同样使用 countGuard 进行加锁、解锁操作，保证修改 count 值的过程是一个原子过程，不会发生并发访问冲突。

在读多写少的环境中，可以优先使用读写互斥锁（sync.RWMutex），它比互斥锁更加高效。sync 包中的 RWMutex 提供了读写互斥锁的封装。

我们将互斥锁例子中的一部分代码修改为读写互斥锁，参见下面代码：

var (

// 逻辑中使用的某个变量

count int

// 与变量对应的使用互斥锁

countGuard sync.RWMutex

)

func GetCount() int {

// 锁定

countGuard.RLock()

// 在函数退出时解除锁定

defer countGuard.RUnlock()

return count

}

代码说明如下：

第 6 行，在声明 countGuard 时，从 sync.Mutex 互斥锁改为 sync.RWMutex 读写互斥锁。

第 12 行，获取 count 的过程是一个读取 count 数据的过程，适用于读写互斥锁。在这一行，把 countGuard.Lock() 换做 countGuard.RLock()，将读写互斥锁标记为读状态。如果此时另外一个 goroutine 并发访问了 countGuard，同时也调用了 countGuard.RLock() 时，并不会发生阻塞。

第 15 行，与读模式加锁对应的，使用读模式解锁。

## 十、Go语言[反射](http://c.biancheng.net/golang/reflect/)

### 1 Go语言reflect.TypeOf()和reflect.Type（通过反射获取类型信息）

在 Go 程序中，使用 reflect.TypeOf() 函数可以获得任意值的类型对象（reflect.Type），程序通过类型对象可以访问任意值的类型信息。下面通过例子来理解获取类型对象的过程：

纯文本复制

package main

import (

"fmt"

"reflect"

)

func main() {

var a int

typeOfA := reflect.TypeOf(a)

fmt.Println(typeOfA.Name(), typeOfA.Kind())

}

代码输出如下：

int int

代码说明如下：

第 10 行，定义一个 int 类型的变量。

第 12 行，通过 reflect.TypeOf() 取得变量 a 的类型对象 typeOfA，类型为 reflect.Type()。

第 14 行中，通过 typeOfA 类型对象的成员函数，可以分别获取到 typeOfA 变量的类型名为 int，种类（Kind）为 int。

**（1）理解反射的类型（Type）与种类（Kind）**

在使用反射时，需要首先理解类型（Type）和种类（Kind）的区别。编程中，使用最多的是类型，但在反射中，当需要区分一个大品种的类型时，就会用到种类（Kind）。例如，需要统一判断类型中的指针时，使用种类（Kind）信息就较为方便。

1) 反射种类（Kind）的定义

Go 程序中的类型（Type）指的是系统原生数据类型，如 int、string、bool、float32 等类型，以及使用 type 关键字定义的类型，这些类型的名称就是其类型本身的名称。例如使用 type A struct{} 定义结构体时，A 就是 struct{} 的类型。

种类（Kind）指的是对象归属的品种，在 reflect 包中有如下定义：

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

)

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。

**2) 从类型对象中获取类型名称和种类的例子**

Go 语言中的类型名称对应的反射获取方法是 reflect.Type 中的 Name() 方法，返回表示类型名称的字符串。

类型归属的种类（Kind）使用的是 reflect.Type 中的 Kind() 方法，返回 reflect.Kind 类型的常量。

下面的代码中会对常量和结构体进行类型信息获取。

package main

import (

"fmt"

"reflect"

)

// 定义一个Enum类型

type Enum int

const (

Zero Enum = 0

)

func main() {

// 声明一个空结构体

type cat struct {

}

// 获取结构体实例的反射类型对象

typeOfCat := reflect.TypeOf(cat{})

// 显示反射类型对象的名称和种类

fmt.Println(typeOfCat.Name(), typeOfCat.Kind())

// 获取Zero常量的反射类型对象

typeOfA := reflect.TypeOf(Zero)

// 显示反射类型对象的名称和种类

fmt.Println(typeOfA.Name(), typeOfA.Kind())

}

代码输出如下：

cat struct

Enum int

代码说明如下：

第 18 行，声明结构体类型 cat。

第 22 行，将 cat 实例化，并且使用 reflect.TypeOf() 获取被实例化后的 cat 的反射类型对象。

第 25 行，输出cat的类型名称和种类，类型名称就是 cat，而 cat 属于一种结构体种类，因此种类为 struct。

第 28 行，Zero 是一个 Enum 类型的常量。这个 Enum 类型在第 9 行声明，第 12 行声明了常量。如没有常量也不能创建实例，通过 reflect.TypeOf() 直接获取反射类型对象。

第 31 行，输出 Zero 对应的类型对象的类型名和种类。

### 2 Go语言reflect.Elem()——通过反射获取指针指向的元素类型

Go 程序中对指针获取反射对象时，可以通过 reflect.Elem() 方法获取这个指针指向的元素类型。这个获取过程被称为取元素，等效于对指针类型变量做了一个\*操作，代码如下：

package main

import (

"fmt"

"reflect"

)

func main() {

// 声明一个空结构体

type cat struct {

}

// 创建cat的实例

ins := &cat{}

// 获取结构体实例的反射类型对象

typeOfCat := reflect.TypeOf(ins)

// 显示反射类型对象的名称和种类

fmt.Printf("name:'%v' kind:'%v'\n",typeOfCat.Name(), typeOfCat.Kind())

// 取类型的元素

typeOfCat = typeOfCat.Elem()

// 显示反射类型对象的名称和种类

fmt.Printf("element name: '%v', element kind: '%v'\n", typeOfCat.Name(), typeOfCat.Kind())

}

代码输出如下：

name: '' kind: 'ptr'

element name: 'cat', element kind: 'struct'

代码说明如下：

第 15 行，创建了cat结构体的实例，ins 是一个 \*cat 类型的指针变量。

第 18 行，对指针变量获取反射类型信息。

第 21 行，输出指针变量的类型名称和种类。Go 语言的反射中对所有指针变量的种类都是 Ptr，但注意，指针变量的类型名称是空，不是 \*cat。

第 24 行，取指针类型的元素类型，也就是 cat 类型。这个操作不可逆，不可以通过一个非指针类型获取它的指针类型。

第 27 行，输出指针变量指向元素的类型名称和种类，得到了 cat 的类型名称（cat）和种类（struct）。

### 3 Go语言通过反射获取结构体的成员类型

任意值通过 reflect.TypeOf() 获得反射对象信息后，如果它的类型是结构体，可以通过反射值对象（reflect.Type）的 NumField() 和 Field() 方法获得结构体成员的详细信息。与成员获取相关的 reflect.Type 的方法如下表所示。

**（1）结构体成员访问的方法列表**

方法 说明

Field(i int) StructField 根据索引，返回索引对应的结构体字段的信息。当值不是结构体或索引超界时发生宕机

NumField() int 返回结构体成员字段数量。当类型不是结构体或索引超界时发生宕机

FieldByName(name string) (StructField, bool) 根据给定字符串返回字符串对应的结构体字段的信息。没有找到时 bool 返回 false，当类型不是结构体或索引超界时发生宕机

FieldByIndex(index []int) StructField 多层成员访问时，根据 []int 提供的每个结构体的字段索引，返回字段的信息。没有找到时返回零值。当类型不是结构体或索引超界时 发生宕机

FieldByNameFunc( match func(string) bool) (StructField,bool) 根据匹配函数匹配需要的字段。当值不是结构体或索引超界时发生宕机

**（2）结构体字段类型**

reflect.Type 的 Field() 方法返回 StructField 结构，这个结构描述结构体的成员信息，通过这个信息可以获取成员与结构体的关系，如偏移、索引、是否为匿名字段、结构体标签（Struct Tag）等，而且还可以通过 StructField 的 Type 字段进一步获取结构体成员的类型信息。StructField 的结构如下：

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 // 字段的结构体标签

Offset uintptr // 字段在结构体中的相对偏移

Index []int // Type.FieldByIndex中的返回的索引值

Anonymous bool // 是否为匿名字段

}

字段说明如下。

Name：为字段名称。

PkgPath：字段在结构体中的路径。

Type：字段本身的反射类型对象，类型为 reflect.Type，可以进一步获取字段的类型信息。

Tag：结构体标签，为结构体字段标签的额外信息，可以单独提取。

Index：FieldByIndex 中的索引顺序。

Anonymous：表示该字段是否为匿名字段。

**（3）获取成员反射信息**

下面代码中，实例化一个结构体并遍历其结构体成员，再通过 reflect.Type 的 FieldByName() 方法查找结构体中指定名称的字段，直接获取其类型信息。

反射访问结构体成员类型及信息：

package main

import (

"fmt"

"reflect"

)

func main() {

// 声明一个空结构体

type cat struct {

Name string

// 带有结构体tag的字段

Type int `json:"type" id:"100"`

}

// 创建cat的实例

ins := cat{Name: "mimi", Type: 1}

// 获取结构体实例的反射类型对象

typeOfCat := reflect.TypeOf(ins)

// 遍历结构体所有成员

for i := 0; i < typeOfCat.NumField(); i++ {

// 获取每个成员的结构体字段类型

fieldType := typeOfCat.Field(i)

// 输出成员名和tag

fmt.Printf("name: %v tag: '%v'\n", fieldType.Name, fieldType.Tag)

}

// 通过字段名, 找到字段类型信息

if catType, ok := typeOfCat.FieldByName("Type"); ok {

// 从tag中取出需要的tag

fmt.Println(catType.Tag.Get("json"), catType.Tag.Get("id"))

}

}

代码输出如下：

name: Name tag: ''

name: Type tag: 'json:"type" id:"100"'

type 100

代码说明如下：

第 11 行，声明了带有两个成员的 cat 结构体。

第 15 行，Type 是 cat 的一个成员，这个成员类型后面带有一个以`开始和结尾的字符串。这个字符串在 Go 语言中被称为 Tag（标签）。一般用于给字段添加自定义信息，方便其他模块根据信息进行不同功能的处理。

第 19 行，创建 cat 实例，并对两个字段赋值。结构体标签属于类型信息，无须且不能赋值。

第 22 行，获取实例的反射类型对象。

第 25 行，使用 reflect.Type 类型的 NumField() 方法获得一个结构体类型共有多少个字段。如果类型不是结构体，将会触发宕机错误。

第 28 行，reflect.Type 中的 Field() 方法和 NumField 一般都是配对使用，用来实现结构体成员的遍历操作。

第 31 行，使用 reflect.Type 的 Field() 方法返回的结构不再是 reflect.Type 而是StructField 结构体。

第 35 行，使用 reflect.Type 的 FieldByName() 根据字段名查找结构体字段信息，cat Type 表示返回的结构体字段信息，类型为 StructField，ok 表示是否找到结构体字段的信息。

第 38 行中，使用 StructField 中 Tag 的 Get() 方法，根据 Tag 中的名字进行信息获取。

### 4 Go语言结构体标签（Struct Tag）

通过 reflect.Type 获取结构体成员信息 reflect.StructField 结构中的 Tag 被称为结构体标签（Struct Tag）。结构体标签是对结构体字段的额外信息标签。

JSON、BSON 等格式进行序列化及对象关系映射（Object Relational Mapping，简称 ORM）系统都会用到结构体标签，这些系统使用标签设定字段在处理时应该具备的特殊属性和可能发生的行为。这些信息都是静态的，无须实例化结构体，可以通过反射获取到。

结构体标签（Struct Tag）类似于 C# 中的特性（Attribute）。C# 允许在类、字段、方法等前面添加 Attribute，然后在反射系统中可以获取到这个属性系统。例如：

[Conditional("DEBUG")]

public static void Message(string msg)

{

Console.WriteLine(msg)；

}

**（1）结构体标签的格式**

Tag 在结构体字段后方书写的格式如下：

`key1:"value1" key2:"value2"`

结构体标签由一个或多个键值对组成。键与值使用冒号分隔，值用双引号括起来。键值对之间使用一个空格分隔。

**（2）从结构体标签中获取值**

StructTag 拥有一些方法，可以进行 Tag 信息的解析和提取，如下所示：

func(tag StructTag)Get(key string)string

根据 Tag 中的键获取对应的值，例如 `key1:"value1"key2:"value2"` 的 Tag 中，可以传入“key1”获得“value1”。

func(tag StructTag)Lookup(key string)(value string,ok bool)

根据 Tag 中的键，查询值是否存在。

**（3）结构体标签格式错误导致的问题**

编写 Tag 时，必须严格遵守键值对的规则。结构体标签的解析代码的容错能力很差，一旦格式写错，编译和运行时都不会提示任何错误，参见下面这个例子：

package main

import (

"fmt"

"reflect"

)

func main() {

type cat struct {

Name string

Type int `json: "type" id:"100"`

}

typeOfCat := reflect.TypeOf(cat{})

if catType, ok := typeOfCat.FieldByName("Type"); ok {

fmt.Println(catType.Tag.Get("json"))

}

}

代码输出空字符串，并不会输出期望的 type。

第 12 行中，在json:和"type"之间增加了一个空格。这种写法没有遵守结构体标签的规则，因此无法通过 Tag.Get 获取到正确的 json 对应的值。

这个错误在开发中非常容易被疏忽，造成难以察觉的错误。

### 5 Go语言reflect.ValueOf()和reflect.Value（通过反射获取值信息）

反射不仅可以获取值的类型信息，还可以动态地获取或者设置变量的值。Go 语言中使用 reflect.Value 获取和设置变量的值。

**（1）使用反射值对象包装任意值**

Go 语言中，使用 reflect.ValueOf() 函数获得值的反射值对象（reflect.Value）。书写格式如下：

value := reflect.ValueOf(rawValue)

reflect.ValueOf 返回 reflect.Value 类型，包含有 rawValue 的值信息。reflect.Value 与原值间可以通过值包装和值获取互相转化。reflect.Value 是一些反射操作的重要类型，如反射调用函数。

**（2）从反射值对象获取被包装的值**

Go 语言中可以通过 reflect.Value 重新获得原始值。

**1) 从反射值对象（reflect.Value）中获取值的方法**

可以通过下面几种方法从反射值对象 reflect.Value 中获取原值，如下表所示。

反射值获取原始值的方法

方法名 说 明

Interface() interface {} 将值以 interface{} 类型返回，可以通过类型断言转换为指定类型

Int() int64 将值以 int 类型返回，所有有符号整型均可以此方式返回

Uint() uint64 将值以 uint 类型返回，所有无符号整型均可以此方式返回

Float() float64 将值以双精度（float64）类型返回，所有浮点数（float32、float64）均可以此方式返回

Bool() bool 将值以 bool 类型返回

Bytes() []bytes 将值以字节数组 []bytes 类型返回

String() string 将值以字符串类型返回

**2) 从反射值对象（reflect.Value）中获取值的例子**

下面代码中，将整型变量中的值使用 reflect.Value 获取反射值对象（reflect.Value）。再通过 reflect.Value 的 Interface() 方法获得 interface{} 类型的原值，通过 int 类型对应的 reflect.Value 的 Int() 方法获得整型值。

package main

import (

"fmt"

"reflect"

)

func main() {

// 声明整型变量a并赋初值

var a int = 1024

// 获取变量a的反射值对象

valueOfA := reflect.ValueOf(a)

// 获取interface{}类型的值, 通过类型断言转换

var getA int = valueOfA.Interface().(int)

// 获取64位的值, 强制类型转换为int类型

var getA2 int = int(valueOfA.Int())

fmt.Println(getA, getA2)

}

代码输出如下：

1024 1024

代码说明如下：

第 11 行，声明一个变量，类型为 int，设置初值为 1024。

第 14 行，获取变量 a 的反射值对象，类型为 reflect.Value，这个过程和 reflect.TypeOf() 类似。

第 17 行，将 valueOfA 反射值对象以 interface{} 类型取出，通过类型断言转换为 int 类型并赋值给 getA。

第 20 行，将 valueOfA 反射值对象通过 Int 方法，以 int64 类型取出，通过强制类型转换，转换为原本的 int 类型。

### 6 Go语言通过反射访问结构体成员的值

反射值对象（reflect.Value）提供对结构体访问的方法，通过这些方法可以完成对结构体任意值的访问，如下表所示。

**（1）反射值对象的成员访问方法**

方 法 备 注

Field(i int) Value 根据索引，返回索引对应的结构体成员字段的反射值对象。当值不是结构体或索引超界时发生宕机

NumField() int 返回结构体成员字段数量。当值不是结构体或索引超界时发生宕机

FieldByName(name string) Value 根据给定字符串返回字符串对应的结构体字段。没有找到时返回零值，当值不是结构体或索引超界时发生宕机

FieldByIndex(index []int) Value 多层成员访问时，根据 []int 提供的每个结构体的字段索引，返回字段的值。 没有找到时返回零值，当值不是结构体或索引超界时发生宕机

FieldByNameFunc(match func(string) bool) Value 根据匹配函数匹配需要的字段。找到时返回零值，当值不是结构体或索引超界时发生宕机

下面代码构造一个结构体包含不同类型的成员。通过 reflect.Value 提供的成员访问函数，可以获得结构体值的各种数据。

**（2）反射访问结构体成员值：**

package main

import (

"fmt"

"reflect"

)

// 定义结构体

type dummy struct {

a int

b string

// 嵌入字段

float32

bool

next \*dummy

}

func main() {

// 值包装结构体

d := reflect.ValueOf(dummy{

next: &dummy{},

})

// 获取字段数量

fmt.Println("NumField", d.NumField())

// 获取索引为2的字段(float32字段)

floatField := d.Field(2)

// 输出字段类型

fmt.Println("Field", floatField.Type())

// 根据名字查找字段

fmt.Println("FieldByName(\"b\").Type", d.FieldByName("b").Type())

// 根据索引查找值中, next字段的int字段的值

fmt.Println("FieldByIndex([]int{4, 0}).Type()", d.FieldByIndex([]int{4, 0}).Type())

}

代码说明如下：

第 9 行，定义结构体，结构体的每个字段的类型都不一样。

第 24 行，实例化结构体并包装为 reflect.Value 类型，成员中包含一个 \*dummy 的实例。

第 29 行，获取结构体的字段数量。

第 32 和 35 行，获取索引为2的字段值（float32 字段），并且打印类型。

第 38 行，根据b字符串，查找到 b 字段的类型。

第 41 行，[]int{4,0} 中的 4 表示，在 dummy 结构中索引值为 4 的成员，也就是 next。next 的类型为 dummy，也是一个结构体，因此使用 []int{4,0} 中的 0 继续在 next 值的基础上索引，结构为 dummy 中索引值为 0 的 a 字段，类型为 int。

代码输出如下：

NumField 5

Field float32

FieldByName("b").Type string

FieldByIndex([]int{4, 0}).Type() int

### 7 Go语言IsNil()和IsValid()——判断反射值的空和有效性

反射值对象（reflect.Value）提供一系列方法进行零值和空判定，如下表所示。

反射值对象的零值和有效性判断方法

方 法 说 明

IsNil() bool 返回值是否为 nil。如果值类型不是通道（channel）、函数、接口、map、指针或 切片时发生 panic，类似于语言层的v== nil操作

IsValid() bool 判断值是否有效。 当值本身非法时，返回 false，例如 reflect Value不包含任何值，值为 nil 等。

下面的例子将会对各种方式的空指针进行 IsNil() 和 IsValid() 的返回值判定检测。同时对结构体成员及方法查找 map 键值对的返回值进行 IsValid() 判定，参考下面的代码。

反射值对象的零值和有效性判断：

package main

import (

"fmt"

"reflect"

)

func main() {

// \*int的空指针

var a \*int

fmt.Println("var a \*int:", reflect.ValueOf(a).IsNil())

// nil值

fmt.Println("nil:", reflect.ValueOf(nil).IsValid())

// \*int类型的空指针

fmt.Println("(\*int)(nil):", reflect.ValueOf((\*int)(nil)).Elem().IsValid())

// 实例化一个结构体

s := struct{}{}

// 尝试从结构体中查找一个不存在的字段

fmt.Println("不存在的结构体成员:", reflect.ValueOf(s).FieldByName("").IsValid())

// 尝试从结构体中查找一个不存在的方法

fmt.Println("不存在的结构体方法:", reflect.ValueOf(s).MethodByName("").IsValid())

// 实例化一个map

m := map[int]int{}

// 尝试从map中查找一个不存在的键

fmt.Println("不存在的键：", reflect.ValueOf(m).MapIndex(reflect.ValueOf(3)).IsValid())

}

代码输出如下：

var a \*int: true

nil: false

(\*int)(nil): false

不存在的结构体成员: false

不存在的结构体方法: false

不存在的键： false

代码说明如下：

第 11 行，声明一个 \*int 类型的指针，初始值为 nil。

第 12 行，将变量 a 包装为 reflect.Value 并且判断是否为空，此时变量 a 为空指针，因此返回 true。

第 15 行，对 nil 进行 IsValid() 判定（有效性判定），返回 false。

第 18 行，(\*int)(nil) 的含义是将 nil 转换为 \*int，也就是\*int 类型的空指针。此行将 nil 转换为 \*int 类型，并取指针指向元素。由于 nil 不指向任何元素，\*int 类型的 nil 也不能指向任何元素，值不是有效的。因此这个反射值使用 Isvalid() 判断时返回 false。

第 21 行，实例化一个结构体。

第 24 行，通过 FieldByName 查找 s 结构体中一个空字符串的成员，如成员不存在，IsValid() 返回 false。

第 27 行，通过 MethodByName 查找 s 结构体中一个空字符串的方法，如方法不存在，IsValid() 返回 false。

第 30 行，实例化一个 map，这种写法与 make 方式创建的 map 等效。

第 33 行，MapIndex() 方法能根据给定的 reflect.Value 类型的值查找 map，并且返回查找到的结果。

IsNil() 常被用于判断指针是否为空；IsValid() 常被用于判定返回值是否有效。

### 8 Go语言通过反射修改变量的值

使用 reflect.Value 对包装的值进行修改时，需要遵循一些规则。如果没有按照规则进行代码设计和编写，轻则无法修改对象值，重则程序在运行时会发生宕机。

**（1）判定及获取元素的相关方法**

使用 reflect.Value 取元素、取地址及修改值的属性方法请参考下表。

反射值对象的判定及获取元素的方法

方法名 备 注

Elem() Value 取值指向的元素值，类似于语言层\*操作。当值类型不是指针或接口时发生宕 机，空指针时返回 nil 的 Value

Addr() Value 对可寻址的值返回其地址，类似于语言层&操作。当值不可寻址时发生宕机

CanAddr() bool 表示值是否可寻址

CanSet() bool 返回值能否被修改。要求值可寻址且是导出的字段

**（2）值修改相关方法**

使用 reflect.Value 修改值的相关方法如下表所示。

反射值对象修改值的方法

Set(x Value) 将值设置为传入的反射值对象的值

Setlnt(x int64) 使用 int64 设置值。当值的类型不是 int、int8、int16、 int32、int64 时会发生宕机

SetUint(x uint64) 使用 uint64 设置值。当值的类型不是 uint、uint8、uint16、uint32、uint64 时会发生宕机

SetFloat(x float64) 使用 float64 设置值。当值的类型不是 float32、float64 时会发生宕机

SetBool(x bool) 使用 bool 设置值。当值的类型不是 bod 时会发生宕机

SetBytes(x []byte) 设置字节数组 []bytes值。当值的类型不是 []byte 时会发生宕机

SetString(x string) 设置字符串值。当值的类型不是 string 时会发生宕机

以上方法，在 reflect.Value 的 CanSet 返回 false 仍然修改值时会发生宕机。

在已知值的类型时，应尽量使用值对应类型的反射设置值。

**（3）值可修改条件之一：可被寻址**

通过反射修改变量值的前提条件之一：这个值必须可以被寻址。简单地说就是这个变量必须能被修改。示例代码如下：

package main

import (

"reflect"

)

func main() {

// 声明整型变量a并赋初值

var a int = 1024

// 获取变量a的反射值对象

valueOfA := reflect.ValueOf(a)

// 尝试将a修改为1(此处会发生崩溃)

valueOfA.SetInt(1)

}

程序运行崩溃，打印错误：

panic: reflect: reflect.Value.SetInt using unaddressable value

报错意思是：SetInt 正在使用一个不能被寻址的值。从 reflect.ValueOf 传入的是 a 的值，而不是 a 的地址，这个 reflect.Value 当然是不能被寻址的。将代码修改一下，重新运行：

package main

import (

"fmt"

"reflect"

)

func main() {

// 声明整型变量a并赋初值

var a int = 1024

// 获取变量a的反射值对象(a的地址)

valueOfA := reflect.ValueOf(&a)

// 取出a地址的元素(a的值)

valueOfA = valueOfA.Elem()

// 修改a的值为1

valueOfA.SetInt(1)

// 打印a的值

fmt.Println(valueOfA.Int())

}

代码输出如下：

1

下面是对代码的分析：

第 14 行中，将变量 a 取值后传给 reflect.ValueOf()。此时 reflect.ValueOf() 返回的 valueOfA 持有变量 a 的地址。

第 17 行中，使用 reflect.Value 类型的 Elem() 方法获取 a 地址的元素，也就是 a 的值。reflect.Value 的 Elem() 方法返回的值类型也是 reflect.Value。

第 20 行，此时 valueOfA 表示的是 a 的值且可以寻址。使用 SetInt() 方法设置值时不再发生崩溃。

第 23 行，正确打印修改的值。

提示

当 reflect.Value 不可寻址时，使用 Addr() 方法也是无法取到值的地址的，同时会发生宕机。虽然说 reflect.Value 的 Addr() 方法类似于语言层的&操作；Elem() 方法类似于语言层的\*操作，但并不代表这些方法与语言层操作等效。

**（4）值可修改条件之一：被导出**

结构体成员中，如果字段没有被导出，即便不使用反射也可以被访问，但不能通过反射修改，代码如下：

package main

import (

"reflect"

)

func main() {

type dog struct {

legCount int

}

// 获取dog实例的反射值对象

valueOfDog := reflect.ValueOf(dog{})

// 获取legCount字段的值

vLegCount := valueOfDog.FieldByName("legCount")

// 尝试设置legCount的值(这里会发生崩溃)

vLegCount.SetInt(4)

}

程序发生崩溃，报错：

panic: reflect: reflect.Value.SetInt using value obtained using unexported field

报错的意思是：SetInt() 使用的值来自于一个未导出的字段。

为了能修改这个值，需要将该字段导出。将 dog 中的 legCount 的成员首字母大写，导出 LegCount 让反射可以访问，修改后的代码如下：

type dog struct {

LegCount int

}

然后根据字段名获取字段的值时，将字符串的字段首字母大写，修改后的代码如下：

vLegCount := valueOfDog.FieldByName("LegCount")

再次运行程序，发现仍然报错：

panic: reflect: reflect.Value.SetInt using unaddressable value

这个错误表示第 13 行构造的 valueOfDog 这个结构体实例不能被寻址，因此其字段也不能被修改。修改代码，取结构体的指针，再通过 reflect.Value 的 Elem() 方法取到值的反射值对象。修改后的完整代码如下：

package main

import (

"reflect"

"fmt"

)

func main() {

type dog struct {

LegCount int

}

// 获取dog实例地址的反射值对象

valueOfDog := reflect.ValueOf(&dog{})

// 取出dog实例地址的元素

valueOfDog = valueOfDog.Elem()

// 获取legCount字段的值

vLegCount := valueOfDog.FieldByName("LegCount")

// 尝试设置legCount的值(这里会发生崩溃)

vLegCount.SetInt(4)

fmt.Println(vLegCount.Int())

}

代码输出如下：

4

代码说明如下：

第 11 行，将 LegCount 首字母大写导出该字段。

第 14 行，获取 dog 实例指针的反射值对象。

第 17 行，取 dog 实例的指针元素，也就是 dog 的实例。

第 20 行，取 dog 结构体中 LegCount 字段的成员值。

第 23 行，修改该成员值。

第 25 行，打印该成员值。

值的修改从表面意义上叫可寻址，换一种说法就是值必须“可被设置”。那么，想修改变量值，一般的步骤是：

取这个变量的地址或者这个变量所在的结构体已经是指针类型。

使用 reflect.ValueOf 进行值包装。

通过 Value.Elem() 获得指针值指向的元素值对象（Value），因为值对象（Value）内部对象为指针时，使用 set 设置时会报出宕机错误。

使用 Value.Set 设置值。

### 9 Go语言通过类型信息创建实例

当已知 reflect.Type 时，可以动态地创建这个类型的实例，实例的类型为指针。例如 reflect.Type 的类型为 int 时，创建 int 的指针，即\*int，代码如下：

package main

import (

"fmt"

"reflect"

)

func main() {

var a int

// 取变量a的反射类型对象

typeOfA := reflect.TypeOf(a)

// 根据反射类型对象创建类型实例

aIns := reflect.New(typeOfA)

// 输出Value的类型和种类

fmt.Println(aIns.Type(), aIns.Kind())

}

代码输出如下：

\*int ptr

代码说明如下：

第 13 行，获取变量 a 的反射类型对象。

第 16 行，使用 reflect.New() 函数传入变量 a 的反射类型对象，创建这个类型的实例值，值以 reflect.Value 类型返回。这步操作等效于：new(int)，因此返回的是 \*int 类型的实例。

第 19 行，打印 aIns 的类型为 \*int，种类为指针。

### 10 Go语言通过反射调用函数

如果反射值对象（reflect.Value）中值的类型为函数时，可以通过 reflect.Value 调用该函数。使用反射调用函数时，需要将参数使用反射值对象的切片 []reflect.Value 构造后传入 Call() 方法中，调用完成时，函数的返回值通过 []reflect.Value 返回。

下面的代码声明一个加法函数，传入两个整型值，返回两个整型值的和。将函数保存到反射值对象（reflect.Value）中，然后将两个整型值构造为反射值对象的切片（[]reflect.Value），使用 Call() 方法进行调用。

反射调用函数：

package main

import (

"fmt"

"reflect"

)

// 普通函数

func add(a, b int) int {

return a + b

}

func main() {

// 将函数包装为反射值对象

funcValue := reflect.ValueOf(add)

// 构造函数参数, 传入两个整型值

paramList := []reflect.Value{reflect.ValueOf(10), reflect.ValueOf(20)}

// 反射调用函数

retList := funcValue.Call(paramList)

// 获取第一个返回值, 取整数值

fmt.Println(retList[0].Int())

}

代码说明如下：

第 9～12 行，定义一个普通的加法函数。

第 17 行，将 add 函数包装为反射值对象。

第 20 行，将 10 和 20 两个整型值使用 reflect.ValueOf 包装为 reflect.Value，再将反射值对象的切片 []reflect.Value 作为函数的参数。

第 23 行，使用 funcValue 函数值对象的 Call() 方法，传入参数列表 paramList 调用 add() 函数。

第 26 行，调用成功后，通过 retList[0] 取返回值的第一个参数，使用 Int 取返回值的整数值。

反射调用函数的过程需要构造大量的 reflect.Value 和中间变量，对函数参数值进行逐一检查，还需要将调用参数复制到调用函数的参数内存中。调用完毕后，还需要将返回值转换为 reflect.Value，用户还需要从中取出调用值。因此，反射调用函数的性能问题尤为突出，不建议大量使用反射函数调用。

## Go语言[编译与工具](http://c.biancheng.net/golang/build/)

### 1 go build命令（go语言编译命令）完全攻略

Go 语言的编译速度非常快。Go 1.9 版本后默认利用 Go 语言的并发特性进行函数粒度的并发编译。

Go 语言的程序编写基本以源码方式，无论是自己的代码还是第三方代码，并且以 GOPATH 作为工作目录和一套完整的工程目录规则。因此 Go 语言中日常编译时无须像 C++ 一样配置各种包含路径、链接库地址等。

Go 语言中使用 go build 命令将源码编译为可执行文件。go build 有很多种编译方法，如无参数编译、文件列表编译、指定包编译等，使用这些方法都可以输出可执行文件。

go build 无参数编译

本小节需要用到的代码具体位置是./src/chapter11/gobuild。

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

代码相对于 GOPATH 的目录关系如下：

.

└── src

└── chapter11

└── gobuild

├── lib.go

└── main.go

main.go 代码如下：

package main

import (

"fmt"

)

func main() {

// 同包的函数

pkgFunc()

fmt.Println("hello world")

}

lib.go 代码如下：

package main

import "fmt"

func pkgFunc() {

fmt.Println("call pkgFunc")

}

如果源码中没有依赖 GOPATH 的包引用，那么这些源码可以使用无参数 go build。格式如下：

go build

在代码所在目录（./src/chapter11/gobuild）下使用go build 命令，如下所示：

$ cd src/chapter11/gobuild/

$ go build

$ ls

gobuild lib.go main.go

$ ./gobuild

call pkgFunc

hello world

命令行指令和输出说明如下：

第 1 行，转到本例源码目录下。

第 2 行，go build 在编译开始时，会搜索当前目录的 go 源码。这个例子中，go build 会找到 lib.go 和 main.go 两个文件。编译这两个文件后，生成当前目录名的可执行文件并放置于当前目录下，这里的可执行文件是 gobuild。

第 3 行和第 4 行，列出当前目录的文件，编译成功，输出 gobuild 可执行文件。

第 5 行，运行当前目录的可执行文件 gobuild。

第 6 行和第 7 行，执行 gobuild 后的输出内容。

go build+文件列表

编译同目录的多个源码文件时，可以在 go build 的后面提供多个文件名，go build 会编译这些源码，输出可执行文件，“go build+文件列表”的格式如下：

go build file1.go file2.go……

在代码代码所在目录（./src/chapter11/gobuild）中使用 go build，在 go build 后添加要编译的源码文件名，代码如下：

$ go build main.go lib.go

$ ls

lib.go main main.go

$ ./main

call pkgFunc

hello world

$ go build lib.go main.go

$ ls

lib lib.go main main.go

命令行指令和输出说明如下：

第 1 行在 go build 后添加文件列表，选中需要编译的 Go 源码。

第 2 行和第 3 行列出完成编译后的当前目录的文件。这次的可执行文件名变成了 main。

第 4～6 行，执行 main 文件，得到期望输出。

第 7 行，尝试调整文件列表的顺序，将 lib.go 放在列表的首位。

第 8 行和第 9 行，编译结果中出现了 lib 可执行文件。

提示

使用“go build+文件列表”方式编译时，可执行文件默认选择文件列表中第一个源码文件作为可执行文件名输出。

如果需要指定输出可执行文件名，可以使用-o参数，参见下面的例子：

$ go build -o myexec main.go lib.go

$ ls

lib.go main.go myexec

$ ./myexec

call pkgFunc

hello world

上面代码中，在 go build 和文件列表之间插入了-o myexec参数，表示指定输出文件名为 myexec。

注意

使用“go build+文件列表”编译方式编译时，文件列表中的每个文件必须是同一个包的 Go 源码。也就是说，不能像 C++ 语言一样，将所有工程的 Go 源码使用文件列表方式进行编译。编译复杂工程时需要用“指定包编译”的方式。

“go build+文件列表”方式更适合使用 Go 语言编写的只有少量文件的工具。

go build+包

“go build+包”在设置 GOPATH 后，可以直接根据包名进行编译，即便包内文件被增（加）删（除）也不影响编译指令。

**1) 代码位置及源码**

本小节需要用到的代码具体位置是./src/chapter11/goinstall。

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

相对于GOPATH的目录关系如下：

.

└── src

└── chapter11

└──goinstall

├── main.go

└── mypkg

└── mypkg.go

main.go代码如下：

package main

import (

"chapter11/goinstall/mypkg"

"fmt"

)

func main() {

mypkg.CustomPkgFunc()

fmt.Println("hello world")

}

mypkg.go代码如下：

package mypkg

import "fmt"

func CustomPkgFunc() {

fmt.Println("call CustomPkgFunc")

}

**2) 按包编译命令**

执行以下命令将按包方式编译 goinstall 代码：

$ export GOPATH=/home/davy/golangbook/code

$ go build -o main chapter11/goinstall

$ ./goinstall

call CustomPkgFunc

hello world

代码说明如下：

第 1 行，设置环境变量 GOPATH，这里的路径是笔者的目录，可以根据实际目录来设置 GOPATH。

第 2 行，-o执行指定输出文件为 main，后面接要编译的包名。包名是相对于 GOPATH 下的 src 目录开始的。

第 3～5 行，编译成功，执行 main 后获得期望的输出。

读者在参考这个例子编译代码时，需要将 GOPATH 更换为自己的目录。注意 GOPATH 下的目录结构，源码必须放在 GOPATH 下的 src 目录下。所有目录中不要包含中文。

go build编译时的附加参数

go build 还有一些附加参数，可以显示更多的编译信息和更多的操作，详见下表所示。

go build编译时的附加参数

附加参数 备 注

-v 编译时显示包名

-p n 开启并发编译，默认情况下该值为 CPU 逻辑核数

-a 强制重新构建

-n 打印编译时会用到的所有命令，但不真正执行

-x 打印编译时会用到的所有命令

-race 开启竞态检测

表中的附加参数按使用频率排列，读者可以根据需要选择使用。

### 2 go run命令——编译并运行

Python 或者 Lua 语言可以在不输出二进制的情况下，将代码使用虚拟机直接执行。Go 语言虽然不使用虚拟机，但可使用 go run 指令达到同样的效果。

go run 命令会编译源码，并且直接执行源码的 main() 函数，不会在当前目录留下可执行文件。

下面我们准备一个 main.go 的文件来观察 go run 的运行结果，源码如下：

package main

import (

"fmt"

"os"

)

func main() {

fmt.Println("args:", os.Args)

}

这段代码的功能是将输入的参数打印出来。使用 go run 运行这个源码文件，命令如下：

$ go run main.go --filename xxx.go

args: [/tmp/go-build006874658/command-line-arguments/\_obj/exe/main--filename xxx.go]

go run 不会在运行目录下生成任何文件，可执行文件被放在临时文件中被执行，工作目录被设置为当前目录。在 go run 的后部可以添加参数，这部分参数会作为代码可以接受的命令行输入提供给程序。

go run 不能使用“go run+包”的方式进行编译，如需快速编译运行包，需要使用如下步骤来代替：

使用 go build 生成可执行文件。

运行可执行文件。

### 3 go install命令——编译并安装

go install 的功能和 go build 类似，附加参数绝大多数都可以与 go build 通用。go install 只是将编译的中间文件放在 GOPATH 的 pkg 目录下，以及固定地将编译结果放在 GOPATH 的 bin 目录下。

本小节需要用到的代码位置是./src/chapter11/goinstall。

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

使用 go install 来执行代码，参考下面的 shell：

$ export GOPATH=/home/davy/golangbook/code

$ go install chapter11/goinstall

编译完成后的目录结构如下：

.

├── bin

│ └── goinstall

├── pkg

│ └── linux\_amd64

│ └── chapter11

│ └── goinstall

│ └── mypkg.a

└── src

└── chapter11

├── gobuild

│ ├── lib.go

│ └── main.go

└── goinstall

├── main.go

└── mypkg

└── mypkg.go

go install 的编译过程有如下规律：

go install 是建立在 GOPATH 上的，无法在独立的目录里使用 go install。

GOPATH 下的 bin 目录放置的是使用 go install 生成的可执行文件，可执行文件的名称来自于编译时的包名。

go install 输出目录始终为 GOPATH 下的 bin 目录，无法使用-o附加参数进行自定义。

GOPATH 下的 pkg 目录放置的是编译期间的中间文件。

### 4 go get命令——一键获取代码、编译并安装

go get 可以借助代码管理工具通过远程拉取或更新代码包及其依赖包，并自动完成编译和安装。整个过程就像安装一个 App 一样简单。

使用 go get 前，需要安装与远程包匹配的代码管理工具，如 Git、SVN、HG 等，参数中需要提供一个包名。

远程包的路径格式

Go 语言的代码被托管于 Github.com 网站，该网站是基于 Git 代码管理工具的，很多有名的项目都在该网站托管代码。其他类似的托管网站还有 code.google.com、bitbucket.org 等。

这些网站的项目包路径都有一个共同的标准，参见下图所示。

图：远程包路径格式

图中的远程包路径是 Go 语言的源码，这个路径共由 3 个部分组成：

网站域名：表示代码托管的网站，类似于电子邮件 @ 后面的服务器地址。

作者或机构：表明这个项目的归属，一般为网站的用户名，如果需要找到这个作者下的所有项目，可以直接在网站上通过搜索“域名/作者”进行查看。这部分类似于电子邮件 @ 前面的部分。

项目名：每个网站下的作者或机构可能会同时拥有很多的项目，图中标示的部分表示项目名称。

go get+远程包

默认情况下，go get 可以直接使用。例如，想获取 go 的源码并编译，使用下面的命令行即可：

$ go get github.com/davyxu/cellnet

获取前，请确保 GOPATH 已经设置。Go 1.8 版本之后，GOPATH 默认在用户目录的 go 文件夹下。

cellnet 只是一个网络库，并没有可执行文件，因此在 go get 操作成功后 GOPATH 下的 bin 目录下不会有任何编译好的二进制文件。

需要测试获取并编译二进制的，可以尝试下面的这个命令。当获取完成后，就会自动在 GOPATH 的 bin 目录下生成编译好的二进制文件。

$ go get github.com/davyxu/tabtoy

go get使用时的附加参数

使用 go get 时可以配合附加参数显示更多的信息及实现特殊的下载和安装操作，详见下表所示。

go get 使用时的附加参数

附加参数 备 注

-v 显示操作流程的日志及信息，方便检查错误

-u 下载丢失的包，但不会更新已经存在的包

-d 只下载，不安装

-insecure 允许使用不安全的 HTTP 方式进行下载操作

### 5 go test命令（Go语言测试命令）完全攻略

Go 语言拥有一套单元测试和性能测试系统，仅需要添加很少的代码就可以快速测试一段需求代码。

性能测试系统可以给出代码的性能数据，帮助测试者分析性能问题。

提示

单元测试（unit testing），是指对软件中的最小可测试单元进行检查和验证。对于单元测试中单元的含义，一般要根据实际情况去判定其具体含义，如C语言中单元指一个函数，Java 里单元指一个类，图形化的软件中可以指一个窗口或一个菜单等。总的来说，单元就是人为规定的最小的被测功能模块。

单元测试是在软件开发过程中要进行的最低级别的测试活动，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。

单元测试——测试和验证代码的框架

要开始一个单元测试，需要准备一个 go 源码文件，在命名文件时需要让文件必须以\_test结尾。

单元测试源码文件可以由多个测试用例组成，每个测试用例函数需要以Test为前缀，例如：

func TestXXX( t \*testing.T )

测试用例文件不会参与正常源码编译，不会被包含到可执行文件中。

测试用例文件使用 go test 指令来执行，没有也不需要 main() 作为函数入口。所有在以\_test结尾的源码内以Test开头的函数会自动被执行。

测试用例可以不传入 \*testing.T 参数。

helloworld 的测试代码（具体位置是./src/chapter11/gotest/helloworld\_test.go）：

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

package code11\_3

import "testing"

func TestHelloWorld(t \*testing.T) {

t.Log("hello world")

}

代码说明如下：

第 5 行，单元测试文件 (\*\_test.go) 里的测试入口必须以 Test 开始，参数为 \*testing.T 的函数。一个单元测试文件可以有多个测试入口。

第 6 行，使用 testing 包的 T 结构提供的 Log() 方法打印字符串。

**1) 单元测试命令行**

单元测试使用 go test 命令启动，例如：

$ go test helloworld\_test.go

ok command-line-arguments 0.003s

$ go test -v helloworld\_test.go

=== RUN TestHelloWorld

--- PASS: TestHelloWorld (0.00s)

helloworld\_test.go:8: hello world

PASS

ok command-line-arguments 0.004s

代码说明如下：

第 1 行，在 go test 后跟 helloworld\_test.go 文件，表示测试这个文件里的所有测试用例。

第 2 行，显示测试结果，ok 表示测试通过，command-line-arguments 是测试用例需要用到的一个包名，0.003s 表示测试花费的时间。

第 3 行，显示在附加参数中添加了-v，可以让测试时显示详细的流程。

第 4 行，表示开始运行名叫 TestHelloWorld 的测试用例。

第 5 行，表示已经运行完 TestHelloWorld 的测试用例，PASS 表示测试成功。

第 6 行打印字符串 hello world。

**2) 运行指定单元测试用例**

go test 指定文件时默认执行文件内的所有测试用例。可以使用-run参数选择需要的测试用例单独执行，参考下面的代码。

一个文件包含多个测试用例（具体位置是./src/chapter11/gotest/select\_test.go）

package code11\_3

import "testing"

func TestA(t \*testing.T) {

t.Log("A")

}

func TestAK(t \*testing.T) {

t.Log("AK")

}

func TestB(t \*testing.T) {

t.Log("B")

}

func TestC(t \*testing.T) {

t.Log("C")

}

这里指定 TestA 进行测试：

$ go test -v -run TestA select\_test.go

=== RUN TestA

--- PASS: TestA (0.00s)

select\_test.go:6: A

=== RUN TestAK

--- PASS: TestAK (0.00s)

select\_test.go:10: AK

PASS

ok command-line-arguments 0.003s

TestA 和 TestAK 的测试用例都被执行，原因是-run跟随的测试用例的名称支持正则表达式，使用-run TestA$即可只执行 TestA 测试用例。

**3) 标记单元测试结果**

当需要终止当前测试用例时，可以使用 FailNow，参考下面的代码。

测试结果标记（具体位置是./src/chapter11/gotest/fail\_test.go）

func TestFailNow(t \*testing.T) {

t.FailNow()

}

还有一种只标记错误不终止测试的方法，代码如下：

func TestFail(t \*testing.T) {

fmt.Println("before fail")

t.Fail()

fmt.Println("after fail")

}

测试结果如下：

=== RUN TestFail

before fail

after fail

--- FAIL: TestFail (0.00s)

FAIL

exit status 1

FAIL command-line-arguments 0.002s

从日志中看出，第 5 行调用 Fail() 后测试结果标记为失败，但是第 7 行依然被程序执行了。

**4) 单元测试日志**

每个测试用例可能并发执行，使用 testing.T 提供的日志输出可以保证日志跟随这个测试上下文一起打印输出。testing.T 提供了几种日志输出方法，详见下表所示。

单元测试框架提供的日志方法

方 法 备 注

Log 打印日志，同时结束测试

Logf 格式化打印日志，同时结束测试

Error 打印错误日志，同时结束测试

Errorf 格式化打印错误日志，同时结束测试

Fatal 打印致命日志，同时结束测试

Fatalf 格式化打印致命日志，同时结束测试

### 6 go pprof命令（Go语言性能分析命令）完全攻略

Go 语言工具链中的 go pprof 可以帮助开发者快速分析及定位各种性能问题，如 CPU 消耗、内存分配及阻塞分析。

性能分析首先需要使用 runtime.pprof 包嵌入到待分析程序的入口和结束处。runtime.pprof 包在运行时对程序进行每秒 100 次的采样，最少采样 1 秒。然后将生成的数据输出，让开发者写入文件或者其他媒介上进行分析。

go pprof 工具链配合 Graphviz 图形化工具可以将 runtime.pprof 包生成的数据转换为 PDF 格式，以图片的方式展示程序的性能分析结果。

安装第三方图形化显式分析数据工具（Graphviz）

Graphviz 是一套通过文本描述的方法生成图形的工具包。描述文本的语言叫做 DOT。

在 www.graphviz.org（http://www.graphviz.org）网站可以获取到最新的 Graphviz 各平台的安装包。

CentOS 下，可以使用 yum 指令直接安装：

$ yum install graphiviz

安装第三方性能分析来分析代码包

runtime.pprof 提供基础的运行时分析的驱动，但是这套接口使用起来还不是太方便，例如：

输出数据使用 io.Writer 接口，虽然扩展性很强，但是对于实际使用不够方便，不支持写入文件。

默认配置项较为复杂。

很多第三方的包在系统包 runtime.pprof 的技术上进行便利性封装，让整个测试过程更为方便。这里使用 github.com/pkg/profile 包进行例子展示，使用下面代码安装这个包：

$ go get github.com/pkg/profile

性能分析代码

下面代码故意制造了一个性能问题，同时使用 github.com/pkg/profile 包进行性能分析。

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

基准测试代码如下（具体文件：./src/chapter11/profile/cpu.go）：

package main

import (

"github.com/pkg/profile"

"time"

)

func joinSlice() []string {

var arr []string

for i := 0; i < 100000; i++ {

// 故意造成多次的切片添加(append)操作, 由于每次操作可能会有内存重新分配和移动, 性能较低

arr = append(arr, "arr")

}

return arr

}

func main() {

// 开始性能分析, 返回一个停止接口

stopper := profile.Start(profile.CPUProfile, profile.ProfilePath("."))

// 在main()结束时停止性能分析

defer stopper.Stop()

// 分析的核心逻辑

joinSlice()

// 让程序至少运行1秒

time.Sleep(time.Second)

}

代码说明如下：

第 4 行，引用 github.com/pkg/profile 第三方包封装。

第 14 行，为了进行性能分析，这里在已知元素大小的情况下，还是使用 append() 函数不断地添加切片。性能较低，在实际中应该避免，这里为了性能分析，故意这样写。

第 22 行，使用 profile.Start 调用 github.com/pkg/profile 包的开启性能分析接口。这个 Start 函数的参数都是可选项，这里需要指定的分析项目是 profile.CPUProfile，也就是 CPU 耗用。profile.ProfilePath(".") 指定输出的分析文件路径，这里指定为当前文件夹。profile.Start() 函数会返回一个 Stop 接口，方便在程序结束时结束性能分析。

第 25 行，使用 defer，将性能分析在 main() 函数结束时停止。

第 28 行，开始执行分析的核心。

第 31 行，为了保证性能分析数据的合理性，分析的最短时间是 1 秒，使用 time.Sleep() 在程序结束前等待 1 秒。如果你的程序默认可以运行 1 秒以上，这个等待可以去掉。

性能分析需要可执行配合才能生成分析结果，因此使用命令行对程序进行编译，代码如下：

$ go build -o cpu cpu.go

$ ./cpu

$ go tool pprof --pdf cpu cpu.pprof > cpu.pdf

代码说明如下：

第 1 行将 cpu.go 编译为可执行文件 cpu。

第 2 行运行可执行文件，在当前目录输出 cpu.pprof 文件。

第 3 行，使用 go tool 工具链输入 cpu.pprof 和 cpu 可执行文件，生成 PDF 格式的输出文件，将输出文件重定向为 cpu.pdf 文件。这个过程中会调用 Graphviz 工具，Windows 下需将 Graphviz 的可执行目录添加到环境变量 PATH 中。

最终生成 cpu.pdf 文件，使用 PDF 查看器打开文件，观察后发现下图所示的某个地方可能存在瓶颈。

图中的每一个框为一个函数调用的路径，第 3 个方框中 joinSlice 函数耗费了 50% 的 CPU 时间，存在性能瓶颈。重新优化代码，在已知切片元素数量的情况下直接分配内存，代码如下：

func joinSlice() []string {

const count = 100000

var arr []string = make([]string, count)

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

arr[i] = "arr"

}

return arr

}

代码说明如下：

第 5 行，将切片预分配 count 个数量，避免之前使用 append() 函数的多次分配。

第 8 行，预分配后，直接对每个元素进行直接赋值。

重新运行上面的代码进行性能分析，最终得到的 cpu.pdf 中将不会再有耗时部分。

## **Go语言“避坑”与技巧**

### 1 goroutine（Go语言并发）如何使用才更加高效？

Go 语言原生支持并发是被众人津津乐道的特性。goroutine 早期是 Inferno 操作系统的一个试验性特性，而现在这个特性与操作系统一起，将开发变得越来越简单。

很多刚开始使用 Go 语言开发的人都很喜欢使用并发特性，而没有考虑并发是否真正能解决他们的问题。

**（1）了解goroutine的生命期时再创建goroutine**

在 Go 语言中，开发者习惯将并发内容与 goroutine 一一对应地创建 goroutine。开发者很少会考虑 goroutine 在什么时候能退出和控制 goroutine 生命期，这就会造成 goroutine 失控的情况。下面来看一段代码。

失控的 goroutine：

package main

import (

"fmt"

"runtime"

)

// 一段耗时的计算函数

func consumer(ch chan int) {

// 无限获取数据的循环

for {

// 从通道获取数据

data := <-ch

// 打印数据

fmt.Println(data)

}

}

func main() {

// 创建一个传递数据用的通道

ch := make(chan int)

for {

// 空变量, 什么也不做

var dummy string

// 获取输入, 模拟进程持续运行

fmt.Scan(&dummy)

// 启动并发执行consumer()函数

go consumer(ch)

// 输出现在的goroutine数量

fmt.Println("goroutines:", runtime.NumGoroutine())

}

}

代码说明如下：

第 9 行，consumer() 函数模拟平时业务中放到 goroutine 中执行的耗时操作。该函数从其他 goroutine 中获取和接收数据或者指令，处理后返回结果。

第 12 行，需要通过无限循环不停地获取数据。

第 15 行，每次从通道中获取数据。

第 18 行，模拟处理完数据后的返回数据。

第 26 行，创建一个整型通道。

第 34 行，使用 fmt.Scan() 函数接收数据时，需要提供变量地址。如果输入匹配的变量类型，将会成功赋值给变量。

第 37 行，启动并发执行 consumer() 函数，并传入 ch 通道。

第 40 行，每启动一个 goroutine，使用 runtime.NumGoroutine 检查进程创建的 goroutine 数量总数。

运行程序，每输入一个字符串+回车，将会创建一个 goroutine，结果如下：

a

goroutines: 2

b

goroutines: 3

c

goroutines: 4

注意，结果中 a、b、c 为通过键盘输入的字符，其他为打印字符。

这个程序实际在模拟一个进程根据需要创建 goroutine 的情况。运行后，问题已经被暴露出来：随着输入的字符串越来越多，goroutine 将会无限制地被创建，但并不会结束。这种情况如果发生在生产环境中，将会造成内存大量分配，最终使进程崩溃。现实的情况也许比这段代码更加隐蔽：也许你设置了一个退出的条件，但是条件永远不会被满足或者触发。

为了避免这种情况，在这个例子中，需要为 consumer() 函数添加合理的退出条件，修改代码后如下：

package main

import (

"fmt"

"runtime"

)

// 一段耗时的计算函数

func consumer(ch chan int) {

// 无限获取数据的循环

for {

// 从通道获取数据

data := <-ch

if data == 0 {

break

}

// 打印数据

fmt.Println(data)

}

fmt.Println("goroutine exit")

}

func main() {

// 传递数据用的通道

ch := make(chan int)

for {

// 空变量, 什么也不做

var dummy string

// 获取输入, 模拟进程持续运行

fmt.Scan(&dummy)

if dummy == "quit" {

for i := 0; i < runtime.NumGoroutine()-1; i++ {

ch <- 0

}

continue

}

// 启动并发执行consumer()函数

go consumer(ch)

// 输出现在的goroutine数量

fmt.Println("goroutines:", runtime.NumGoroutine())

}

}

代码中加粗部分是新添加的代码，具体说明如下：

第 17 行，为无限循环设置退出条件，这里设置 0 为退出。

第 41 行，当命令行输入 quit 时，进入退出处理的流程。

第 43 行，runtime.NumGoroutine 返回一个进程的所有 goroutine 数，main() 的 goroutine 也被算在里面。因此需要扣除 main() 的 goroutine 数。剩下的 goroutine 为实际创建的 goroutine 数，对这些 goroutine 进行遍历。

第 44 行，并发开启的 goroutine 都在竞争获取通道中的数据，因此只要知道有多少个 goroutine 需要退出，就给通道里发多少个 0。

修改程序并运行，结果如下：

a

goroutines: 2

b

goroutines: 3

quit

goroutine exit

goroutine exit

c

goroutines: 2

**（2）避免在不必要的地方使用通道**

通道（channel）和 map、切片一样，也是由 Go 源码编写而成。为了保证两个 goroutine 并发访问的安全性，通道也需要做一些锁操作，因此通道其实并不比锁高效。

下面的例子展示套接字的接收和并发管理。对于 TCP 来说，一般是接收过程创建 goroutine 并发处理。当套接字结束时，就要正常退出这些 goroutine。

本例完整代码请参考./src/chapter12/exitnotify/exitnotify.go。

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

下面是对各个部分的详细分析。

**1) 套接字接收部分**

套接字在连接后，就需要不停地接收数据，代码如下：

// 套接字接收过程

func socketRecv(conn net.Conn, exitChan chan string) {

// 创建一个接收的缓冲

buff := make([]byte, 1024)

// 不停地接收数据

for {

// 从套接字中读取数据

\_, err := conn.Read(buff)

// 需要结束接收, 退出循环

if err != nil {

break

}

}

// 函数已经结束, 发送通知

exitChan <- "recv exit"

}

代码说明如下：

第 2 行传入的 net.Conn 是套接字的接口，exitChan 为退出发送同步通道。

第 5 行为套接字的接收数据创建一个缓冲。

第 8 行构建一个接收的循环，不停地接收数据。

第 11 行，从套接字中取出数据。这个例子中，不关注具体接收到的数据，只是关注错误，这里将接收到的字节数做匿名处理。

第 14 行，当套接字调用了 Close 方法时，会触发错误，这时需要结束接收循环。

第 21 行，结束函数时，与函数绑定的 goroutine 会同时结束，此时需要通知 main() 的 goroutine。

**2) 连接、关闭、同步 goroutine 主流程部分**

下面代码中尝试使用套接字的 TCP 协议连接一个网址，连接上后，进行数据接收，等待一段时间后主动关闭套接字，等待套接字所在的 goroutine 自然结束，代码如下：

func main() {

// 连接一个地址

conn, err := net.Dial("tcp", "www.163.com:80")

// 发生错误时打印错误退出

if err != nil {

fmt.Println(err)

return

}

// 创建退出通道

exit := make(chan string)

// 并发执行套接字接收

go socketRecv(conn, exit)

// 在接收时, 等待1秒

time.Sleep(time.Second)

// 主动关闭套接字

conn.Close()

// 等待goroutine退出完毕

fmt.Println(<-exit)

}

代码说明如下：

第 4 行，使用 net.Dial 发起 TCP 协议的连接，调用函数就会发送阻塞直到连接超时或者连接完成。

第 7 行，如果连接发生错误，将会打印错误并退出。

第 13 行，创建一个通道用于退出信号同步，这个通道会在接收用的 goroutine 中使用。

第 16 行，并发执行接收函数，传入套接字和用于退出通知的通道。

第 19 行，接收需要一个过程，使用 time.Sleep() 等待一段时间。

第 22 行，主动关闭套接字，此时会触发套接字接收错误。

第 25 行，从 exit 通道接收退出数据，也就是等待接收 goroutine 结束。

在这个例子中，goroutine 退出使用通道来通知，这种做法可以解决问题，但是实际上通道中的数据并没有完全使用

**3) 优化：使用等待组替代通道简化同步**

通道的内部实现代码在 Go 语言开发包的 src/runtime/chan.go 中，经过分析后大概了解到通道也是用常见的互斥量等进行同步。因此通道虽然是一个语言级特性，但也不是被神化的特性，通道的运行和使用都要比传统互斥量、等待组（sync.WaitGroup）有一定的消耗。

所以在这个例子中，更建议使用等待来实现同步，调整后的代码如下：

package main

import (

"fmt"

"net"

"sync"

"time"

)

// 套接字接收过程

func socketRecv(conn net.Conn, wg \*sync.WaitGroup) {

// 创建一个接收的缓冲

buff := make([]byte, 1024)

// 不停地接收数据

for {

// 从套接字中读取数据

\_, err := conn.Read(buff)

// 需要结束接收, 退出循环

if err != nil {

break

}

}

// 函数已经结束, 发送通知

wg.Done()

}

func main() {

// 连接一个地址

conn, err := net.Dial("tcp", "www.163.com:80")

// 发生错误时打印错误退出

if err != nil {

fmt.Println(err)

return

}

// 退出通道

var wg sync.WaitGroup

// 添加一个任务

wg.Add(1)

// 并发执行接收套接字

go socketRecv(conn, &wg)

// 在接收时, 等待1秒

time.Sleep(time.Second)

// 主动关闭套接字

conn.Close()

// 等待goroutine退出完毕

wg.Wait()

fmt.Println("recv done")

}

调整后的代码说明如下：

第 45 行，声明退出同步用的等待组。

第 48 行，为等待组的计数器加 1，表示需要完成一个任务。

第 51 行，将等待组的指针传入接收函数。

第 60 行，等待等待组的完成，完成后打印提示。

第 30 行，接收完成后，使用 wg.Done() 方法将等待组计数器减一。

### 2 Go语言反射——性能和灵活性的双刃剑

现在的一些流行设计思想需要建立在反射基础上，如控制反转（Inversion Of Control，IOC）和依赖注入（Dependency Injection，DI）。Go 语言中非常有名的 Web 框架 martini（https://github.com/go-martini/martini）就是通过依赖注入技术进行中间件的实现，例如使用 martini 框架搭建的 http 的服务器如下：

package main

import "github.com/go-martini/martini"

func main() {

m := martini.Classic()

m.Get("/", func() string {

return "Hello world!"

})

m.Run()

第 7 行，响应路径/的代码使用一个闭包实现。如果希望获得 Go 语言中提供的请求和响应接口，可以直接修改为：

m.Get("/", func(res http.ResponseWriter, req \*http.Request) string {

// 响应处理代码……

})

martini 的底层会自动通过识别 Get 获得的闭包参数情况，通过动态反射调用这个函数并传入需要的参数。martini 的设计广受好评，但同时也有人指出，其运行效率较低。其中最主要的因素是大量使用了反射。

虽然一般情况下，I/O 的延迟远远大于反射代码所造成的延迟。但是，更低的响应速度和更低的 CPU 占用依然是 Web 服务器追求的目标。因此，反射在带来灵活性的同时，也带上了性能低下的桎梏。

要用好反射这把双刃剑，就需要详细了解反射的性能。下面的一些基准测试从多方面对比了原生调用和反射调用的区别。

**1) 结构体成员赋值对比**

反射经常被使用在结构体上，因此结构体的成员访问性能就成为了关注的重点。下面例子中使用一个被实例化的结构体，访问它的成员，然后使用 Go 语言的基准化测试可以迅速测试出结果。

反射性能测试的完整代码位于./src/chapter12/reflecttest/reflect\_test.go，下面是对各个部分的详细说明。

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

原生结构体的赋值过程：

// 声明一个结构体, 拥有一个字段

type data struct {

Hp int

}

func BenchmarkNativeAssign(b \*testing.B) {

// 实例化结构体

v := data{Hp: 2}

// 停止基准测试的计时器

b.StopTimer()

// 重置基准测试计时器数据

b.ResetTimer()

// 重新启动基准测试计时器

b.StartTimer()

// 根据基准测试数据进行循环测试

for i := 0; i < b.N; i++ {

// 结构体成员赋值测试

v.Hp = 3

}

}

代码说明如下：

第 2 行，声明一个普通结构体，拥有一个成员变量。

第 6 行，使用基准化测试的入口。

第 9 行，实例化 data 结构体，并给 Hp 成员赋值。

第 12～17 行，由于测试的重点必须放在赋值上，因此需要极大程度地降低其他代码的干扰，于是在赋值完成后，将基准测试的计时器复位并重新开始。

第 20 行，将基准测试提供的测试数量用于循环中。

第 23 行，测试的核心代码：结构体赋值。

接下来的代码分析使用反射访问结构体成员并赋值的过程。

func BenchmarkReflectAssign(b \*testing.B) {

v := data{Hp: 2}

// 取出结构体指针的反射值对象并取其元素

vv := reflect.ValueOf(&v).Elem()

// 根据名字取结构体成员

f := vv.FieldByName("Hp")

b.StopTimer()

b.ResetTimer()

b.StartTimer()

for i := 0; i < b.N; i++ {

// 反射测试设置成员值性能

f.SetInt(3)

}

}

代码说明如下：

第 6 行，取v的地址并转为反射值对象。此时值对象里的类型为 \*data，使用值的 Elem() 方法取元素，获得 data 的反射值对象。

第 9 行，使用 FieldByName() 根据名字取出成员的反射值对象。

第 11～13 行，重置基准测试计时器。

第 18 行，使用反射值对象的 SetInt() 方法，给 data 结构的Hp字段设置数值 3。

这段代码中使用了反射值对象的 SetInt() 方法，这个方法的源码如下：

func (v Value) SetInt(x int64) {

v.mustBeAssignable()

switch k := v.kind(); k {

default:

panic(&ValueError{"reflect.Value.SetInt", v.kind()})

case Int:

\*(\*int)(v.ptr) = int(x)

case Int8:

\*(\*int8)(v.ptr) = int8(x)

case Int16:

\*(\*int16)(v.ptr) = int16(x)

case Int32:

\*(\*int32)(v.ptr) = int32(x)

case Int64:

\*(\*int64)(v.ptr) = x

}

}

可以发现，整个设置过程都是指针转换及赋值，没有遍历及内存操作等相对耗时的算法。

**（2）结构体成员搜索并赋值对比**

func BenchmarkReflectFindFieldAndAssign(b \*testing.B) {

v := data{Hp: 2}

vv := reflect.ValueOf(&v).Elem()

b.StopTimer()

b.ResetTimer()

b.StartTimer()

for i := 0; i < b.N; i++ {

// 测试结构体成员的查找和设置成员的性能

vv.FieldByName("Hp").SetInt(3)

}

}

这段代码将反射值对象的 FieldByName() 方法与 SetInt() 方法放在循环里进行检测，主要对比测试 FieldByName() 方法对性能的影响。FieldByName() 方法源码如下：

func (v Value) FieldByName(name string) Value {

v.mustBe(Struct)

if f, ok := v.typ.FieldByName(name); ok {

return v.FieldByIndex(f.Index)

}

return Value{}

}

底层代码说明如下：

第 3 行，通过名字查询类型对象，这里有一次遍历过程。

第 4 行，找到类型对象后，使用 FieldByIndex() 继续在值中查找，这里又是一次遍历。

经过底层代码分析得出，随着结构体字段数量和相对位置的变化，FieldByName() 方法比较严重的低效率问题。

**（3）调用函数对比**

反射的函数调用，也是使用反射中容易忽视的性能点，下面展示对普通函数的调用过程。

// 一个普通函数

func foo(v int) {

}

func BenchmarkNativeCall(b \*testing.B) {

for i := 0; i < b.N; i++ {

// 原生函数调用

foo(0)

}

}

func BenchmarkReflectCall(b \*testing.B) {

// 取函数的反射值对象

v := reflect.ValueOf(foo)

b.StopTimer()

b.ResetTimer()

b.StartTimer()

for i := 0; i < b.N; i++ {

// 反射调用函数

v.Call([]reflect.Value{reflect.ValueOf(2)})

}

}

代码说明如下：

第 2 行，一个普通的只有一个参数的函数。

第 10 行，对原生函数调用的性能测试。

第 17 行，根据函数名取出反射值对象。

第 25 行，使用 reflect.ValueOf(2) 将 2 构造为反射值对象，因为反射函数调用的参数必须全是反射值对象，再使用 []reflect.Value 构造多个参数列表传给反射值对象的 Call() 方法进行调用。

反射函数调用的参数构造过程非常复杂，构建很多对象会造成很大的内存回收负担。Call() 方法内部就更为复杂，需要将参数列表的每个值从 reflect.Value 类型转换为内存。调用完毕后，还要将函数返回值重新转换为 reflect.Value 类型返回。因此，反射调用函数的性能堪忧。

**（4）基准测试结果对比**

测试结果如下：

$ go test -v -bench=.

goos: linux

goarch: amd64

BenchmarkNativeAssign-4 2000000000 0.32 ns/op

BenchmarkReflectAssign-4 300000000 4.42 ns/op

BenchmarkReflectFindFieldAndAssign-4 20000000 91.6 ns/op

BenchmarkNativeCall-4 2000000000 0.33 ns/op

BenchmarkReflectCall-4 10000000 163 ns/op

PASS

结果分析如下：

第 4 行，原生的结构体成员赋值，每一步操作耗时 0.32 纳秒，这是参考基准。

第 5 行，使用反射的结构体成员赋值，操作耗时 4.42 纳秒，比原生赋值多消耗 13 倍的性能。

第 6 行，反射查找结构体成员且反射赋值，操作耗时 91.6 纳秒，扣除反射结构体成员赋值的 4.42 纳秒还富余，性能大概是原生的 272 倍。这个测试结果与代码分析结果很接近。SetInt 的性能可以接受，但 FieldByName() 的性能就非常低。

第 7 行，原生函数调用，性能与原生访问结构体成员接近。

第 8 行，反射函数调用，性能差到“爆棚”，花费了 163 纳秒，操作耗时比原生多消耗 494 倍。

经过基准测试结果的数值分析及对比，最终得出以下结论：

能使用原生代码时，尽量避免反射操作。

提前缓冲反射值对象，对性能有很大的帮助。

避免反射函数调用，实在需要调用时，先提前缓冲函数参数列表，并且尽量少地使用返回值。

### 3 Go语言接口的nil判断

nil 在 Go 语言中只能被赋值给指针和接口。接口在底层的实现有两个部分：type 和 data。在源码中，显式地将 nil 赋值给接口时，接口的 type 和 data 都将为 nil。此时，接口与 nil 值判断是相等的。但如果将一个带有类型的 nil 赋值给接口时，只有 data 为 nil，而 type 为 nil，此时，接口与 nil 判断将不相等。

**（1）接口与nil不相等**

下面代码使用 MyImplement() 实现 fmt 包中的 Stringer 接口，这个接口的定义如下：

type Stringer interface {

String() string

}

在 GetStringer() 函数中将返回这个接口。通过 \*MyImplement 指针变量置为 nil 提供 GetStringer 的返回值。在 main() 中，判断 GetStringer 与 nil 是否相等，代码如下：

package main

import "fmt"

// 定义一个结构体

type MyImplement struct{}

// 实现fmt.Stringer的String方法

func (m \*MyImplement) String() string {

return "hi"

}

// 在函数中返回fmt.Stringer接口

func GetStringer() fmt.Stringer {

// 赋nil

var s \*MyImplement = nil

// 返回变量

return s

}

func main() {

// 判断返回值是否为nil

if GetStringer() == nil {

fmt.Println("GetStringer() == nil")

} else {

fmt.Println("GetStringer() != nil")

}

}

代码说明如下：

第 9 行，实现 fmt.Stringer 的 String() 方法。

第 21 行，s 变量此时被 fmt.Stringer 接口包装后，实际类型为 \*MyImplement，值为 nil 的接口。

第 27 行，使用 GetStringer() 的返回值与 nil 判断时，虽然接口里的 value 为 nil，但 type 带有 \*MyImplement 信息，使用 == 判断相等时，依然不为 nil。

**（2）发现nil类型值返回时直接返回nil**

为了避免这类误判的问题，可以在函数返回时，发现带有 nil 的指针时直接返回 nil，代码如下：

func GetStringer() fmt.Stringer {

var s \*MyImplement = nil

if s == nil {

return nil

}

return s

}

### 4 Go语言map的多键索引——多个数值条件可以同时查询

**（1）在大多数的编程语言中，映射容器的键必须以单一值存在。**这种映射方法经常被用在诸如信息检索上，如根据通讯簿的名字进行检索。但随着查询条件越来越复杂，检索也会变得越发困难。下面例子中涉及通讯簿的结构，结构如下：

// 人员档案

type Profile struct {

Name string // 名字

Age int // 年龄

Married bool // 已婚

}

并且准备好了一堆原始数据，需要算法实现构建索引和查询的过程，代码如下：

func main() {

list := []\*Profile{

{Name: "张三", Age: 30, Married: true},

{Name: "李四", Age: 21},

{Name: "王麻子", Age: 21},

}

buildIndex(list)

queryData("张三", 30)

}

需要用算法实现 buildIndex() 构建索引函数及 queryData() 查询数据函数，查询到结果后将数据打印出来。

下面，分别基于传统的基于哈希值的多键索引和利用 map 特性的多键索引进行查询。

**（2）基于哈希值的多键索引及查询**

传统的数据索引过程是将输入的数据做特征值。这种特征值有几种常见做法：

将特征使用某种算法转为整数，即哈希值，使用整型值做索引。

将特征转为字符串，使用字符串做索引。

数据都基于特征值构建好索引后，就可以进行查询。查询时，重复这个过程，将查询条件转为特征值，使用特征值进行查询得到结果。

基于哈希的传统多键索引和查询的完整代码位于./src/chapter12/classic/classic.go，下面是对各个部分的分析。

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

**1) 字符串转哈希值**

本例中，查询键（classicQueryKey）的特征值需要将查询键中每一个字段转换为整型，字符串也需要转换为整型值，这里使用一种简单算法将字符串转换为需要的哈希值，代码如下：

func simpleHash(str string) (ret int) {

// 遍历字符串中的每一个ASCII字符

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

// 取出字符

c := str[i]

// 将字符的ASCII码相加

ret += int(c)

}

return

}

代码说明如下：

第 1 行传入需要计算哈希值的字符串。

第 4 行，根据字符串的长度，遍历这个字符串的每一个字符，以 ASCII 码为单位。

第 9 行，c变量的类型为 uint8，将其转为 int 类型并累加。

哈希算法有很多，这里只是选用一种大家便于理解的算法。哈希算法的选用的标准是尽量减少重复键的发生，俗称“哈希冲撞”，即同样两个字符串的哈希值重复率降到最低。

**2) 查询键**

有了哈希算法函数后，将哈希函数用在查询键结构中。查询键结构如下：

// 查询键

type classicQueryKey struct {

Name string // 要查询的名字

Age int // 要查询的年龄

}

// 计算查询键的哈希值

func (c \*classicQueryKey) hash() int {

// 将名字的Hash和年龄哈希合并

return simpleHash(c.Name) + c.Age\*1000000

}

代码说明如下：

第 2 行，声明查询键的结构，查询键包含需要索引和查询的字段。

第 8 行，查询键的成员方法哈希，通过调用这个方法获得整个查询键的哈希值。

第 10 行，查询键哈希的计算方法：使用 simpleHash() 函数根据给定的名字字符串获得其哈希值。同时将年龄乘以 1000000 与名字哈希值相加。

哈希值构建过程如下图所示

**3) 构建索引**

本例需要快速查询，因此需要提前对已有的数据构建索引。前面已经准备好了数据查询键，使用查询键获得哈希即可对数据进行快速索引，参考下面的代码：

// 创建哈希值到数据的索引关系

var mapper = make(map[int][]\*Profile)

// 构建数据索引

func buildIndex(list []\*Profile) {

// 遍历所有的数据

for \_, profile := range list {

// 构建数据的查询索引

key := classicQueryKey{profile.Name, profile.Age}

// 计算数据的哈希值, 取出已经存在的记录

existValue := mapper[key.hash()]

// 将当前数据添加到已经存在的记录切片中

existValue = append(existValue, profile)

// 将切片重新设置到映射中

mapper[key.hash()] = existValue

}

}

代码说明如下：

第 2 行，实例化一个 map，键类型为整型，保存哈希值；值类型为 \*Profile，为通讯簿的数据格式。

第 5 行，构建索引函数入口，传入数据切片。

第 8 行，遍历数据切片的所有数据元素。

第 11 行，使用查询键（classicQueryKey）来辅助计算哈希值，查询键需要填充两个字段，将数据中的名字和年龄赋值到查询键中进行保存。

第 14 行，使用查询键的哈希方法计算查询键的哈希值。通过这个值在 mapper 索引中查找相同哈希值的数据切片集合。因为哈希函数不能保证不同数据的哈希值一定完全不同，因此要考虑在发生哈希值重复时的处理办法。

第 17 行，将当前数据添加到可能存在的切片中。

第 20 行，将新添加好的数据切片重新赋值到相同哈希的 mapper 中。

具体哈希结构如下图所示

图：哈希结构

这种多键的算法就是哈希算法。map 的多个元素对应哈希的“桶”。哈希函数的选择决定桶的映射好坏，如果哈希冲撞很厉害，那么就需要将发生冲撞的相同哈希值的元素使用切片保存起来。

**4) 查询逻辑**

从已经构建好索引的数据中查询需要的数据流程如下：

给定查询条件（名字、年龄）。

根据查询条件构建查询键。

查询键生成哈希值。

根据哈希值在索引中查找数据集合。

遍历数据集合逐个与条件比对。

获得结果。

func queryData(name string, age int) {

// 根据给定查询条件构建查询键

keyToQuery := classicQueryKey{name, age}

// 计算查询键的哈希值并查询, 获得相同哈希值的所有结果集合

resultList := mapper[keyToQuery.hash()]

// 遍历结果集合

for \_, result := range resultList {

// 与查询结果比对, 确认找到打印结果

if result.Name == name && result.Age == age {

fmt.Println(result)

return

}

}

// 没有查询到时, 打印结果

fmt.Println("no found")

}

代码说明如下：

第 1 行，查询条件（名字、年龄）。

第 4 行，根据查询条件构建查询键。

第 7 行，使用查询键计算哈希值，使用哈希值查询相同哈希值的所有数据集合。

第 10 行，遍历所有相同哈希值的数据集合。

第 13 行，将每个数据与查询条件进行比对，如果一致，表示已经找到结果，打印并返回。

第 20 行，没有找到记录时，打印 no found。

**（2）利用map特性的多键索引及查询**

使用结构体进行多键索引和查询比传统的写法更为简单，最主要的区别是无须准备哈希函数及相应的字段无须做哈希合并。看下面的实现流程。

利用map特性的多键索引和查询的代码位于./src/chapter12/multikey/multikey.go，下面是对各个部分的分析。

本套教程所有源码下载地址：https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ 提取密码：hfyf

**1) 构建索引**

代码如下：

// 查询键

type queryKey struct {

Name string

Age int

}

// 创建查询键到数据的映射

var mapper = make(map[queryKey]\*Profile)

// 构建查询索引

func buildIndex(list []\*Profile) {

// 遍历所有数据

for \_, profile := range list {

// 构建查询键

key := queryKey{

Name: profile.Name,

Age: profile.Age,

}

// 保存查询键

mapper[key] = profile

}

}

代码说明如下：

第 2 行，与基于哈希值的查询键的结构相同。

第 8 行，在 map 的键类型上，直接使用了查询键结构体。注意，这里不使用查询键的指针。同时，结果只有 \*Profile 类型，而不是 \*Profile 切片，表示查到的结果唯一。

第 17 行，类似的，使用遍历到的数据的名字和年龄构建查询键。

第 23 行，更简单的，直接将查询键保存对应的数据。

**2) 查询逻辑**

// 根据条件查询数据

func queryData(name string, age int) {

// 根据查询条件构建查询键

key := queryKey{name, age}

// 根据键值查询数据

result, ok := mapper[key]

// 找到数据打印出来

if ok {

fmt.Println(result)

} else {

fmt.Println("no found")

}

}

代码说明如下：

第 5 行，根据查询条件（名字、年龄）构建查询键。

第 8 行，直接使用查询键在 map 中查询结果。

第 12 行，找到结果直接打印。

第 14 行，没有找到结果打印 no found。

**（3）总结**

基于哈希值的多键索引查询和利用map特性的多键索引查询的代码复杂程度显而易见。聪明的程序员都会利用Go语言的特性进行快速的多键索引查询。

其实，利用 map 特性的例子中的 map 类型即便修改为下面的格式，也一样可以获得同样的结果：

var mapper = make(map[interface{}]\*Profile)

代码量大大减少的关键是：Go 语言的底层会为 map 的键自动构建哈希值。能够构建哈希值的类型必须是非动态类型、非指针、函数、闭包。

非动态类型：可用数组，不能用切片。

非指针：每个指针数值都不同，失去哈希意义。

函数、闭包不能作为 map 的键。

## Go项目实战

### 1 Go语言使用事件系统实现事件的响应和处理

Go 语言可以将类型的方法与普通函数视为一个概念，从而简化方法和函数混合作为回调类型时的复杂性。这个特性和 C# 中的代理（delegate）类似，调用者无须关心谁来支持调用，系统会自动处理是否调用普通函数或类型的方法

本节中，首先将用简单的例子了解 Go 语言是如何将方法与函数视为一个概念，接着会实现一个事件系统，事件系统能有效地将事件触发与响应两端代码解耦。

**（1）方法和函数的统一调用**

本节的例子将让一个结构体的方法（class.Do）的参数和一个普通函数（funcDo）的参数完全一致，也就是方法与函数的签名一致。然后使用与它们签名一致的函数变量（delegate）分别赋值方法与函数，接着调用它们，观察实际效果。

详细实现请参考下面的代码。

package main

import "fmt"

// 声明一个结构体

type class struct {

}

// 给结构体添加Do方法

func (c \*class) Do(v int) {

fmt.Println("call method do:", v)

}

// 普通函数的Do

func funcDo(v int) {

fmt.Println("call function do:", v)

}

func main() {

// 声明一个函数回调

var delegate func(int)

// 创建结构体实例

c := new(class)

// 将回调设为c的Do方法

delegate = c.Do

// 调用

delegate(100)

// 将回调设为普通函数

delegate = funcDo

// 调用

delegate(100)

}

代码说明如下：

第 10 行，为结构体添加一个 Do() 方法，参数为整型。这个方法的功能是打印提示和输入的参数值。

第 16 行，声明一个普通函数，参数也是整型，功能是打印提示和输入的参数值。

第 24 行，声明一个 delegate 的变量，类型为 func(int)，与 funcDo 和 class 的 Do() 方法的参数一致。

第 30 行，将 c.Do 作为值赋给 delegate 变量。

第 33 行，调用 delegate() 函数，传入 100 的参数。此时会调用 c 实例的 Do() 方法。

第 36 行，将 funcDo 赋值给 delegate。

第 39 行，调用 delegate()，传入 100 的参数。此时会调用 funcDo() 方法。

运行代码，输出如下：

call method do: 100

call function do: 100

这段代码能运行的基础在于：无论是普通函数还是结构体的方法，只要它们的签名一致，与它们签名一致的函数变量就可以保存普通函数或是结构体方法。

了解了 Go 语言的这一特性后，我们就可以将这个特性用在事件中。

**（2）事件系统基本原理**

事件系统可以将事件派发者与事件处理者解耦。例如，网络底层可以生成各种事件，在网络连接上后，网络底层只需将事件派发出去，而不需要关心到底哪些代码来响应连接上的逻辑。或者再比如，你注册、关注或者订阅某“大V”的社交消息后，“大V”发生的任何事件都会通知你，但他并不用了解粉丝们是如何为她喝彩或者疯狂的。如下图所示为事件系统基本原理图。

图：事系统基本原理

一个事件系统拥有如下特性：

能够实现事件的一方，可以根据事件ID或名字注册对应的事件。

事件发起者，会根据注册信息通知这些注册者。

一个事件可以有多个实现方响应。

通过下面的步骤详细了解事件系统的构成及使用。

**（3）事件注册**

事件系统需要为外部提供一个注册入口。这个注册入口传入注册的事件名称和对应事件名称的响应函数，事件注册的过程就是将事件名称和响应函数关联并保存起来，详细实现请参考下面代码的 RegisterEvent() 函数。

package main

// 实例化一个通过字符串映射函数切片的map

var eventByName = make(map[string][]func(interface{}))

// 注册事件，提供事件名和回调函数

func RegisterEvent(name string, callback func(interface{})) {

// 通过名字查找事件列表

list := eventByName[name]

// 在列表切片中添加函数

list = append(list, callback)

// 将修改的事件列表切片保存回去

eventByName[name] = list

}

// 调用事件

func CallEvent(name string, param interface{}) {

// 通过名字找到事件列表

list := eventByName[name]

// 遍历这个事件的所有回调

for \_, callback := range list {

// 传入参数调用回调

callback(param)

}

}

代码说明如下：

第 4 行，创建一个 map 实例，这个 map 通过事件名（string）关联回调列表（[]func(interface{}），同一个事件名称可能存在多个事件回调，因此使用回调列表保存。回调的函数声明为 func(interface{})。

第 7 行，提供给外部的通过事件名注册响应函数的入口。

第 10 行，eventByName 通过事件名（name）进行查询，返回回调列表（[]func(interface{}）。

第 13 行，为同一个事件名称在已经注册的事件回调的列表中再添加一个回调函数。

第 16 行，将修改后的函数列表设置到 map 的对应事件名中。

拥有事件名和事件回调函数列表的关联关系后，就需要开始准备事件调用的入口了。

**（4）事件调用**

事件调用方和注册方是事件处理中完全不同的两个角色。事件调用方是事发现场，负责将事件和事件发生的参数通过事件系统派发出去，而不关心事件到底由谁处理；事件注册方通过事件系统注册应该响应哪些事件及如何使用回调函数处理这些事件。事件调用的详细实现请参考上面代码的 CallEvent() 函数。

代码说明如下：

第 20 行，调用事件的入口，提供事件名称 name 和参数 param。事件的参数表示描述事件具体的细节，例如门打开的事件触发时，参数可以传入谁进来了。

第 23 行，通过注册事件回调的 eventByName 和事件名字查询处理函数列表 list。

第 26 行，遍历这个事件列表，如果没有找到对应的事件，list 将是一个空切片。

第 29 行，将每个函数回调传入事件参数并调用，就会触发事件实现方的逻辑处理。

**（5）使用事件系统**

例子中，在 main() 函数中调用事件系统的 CallEvent 生成 OnSkill 事件，这个事件有两个处理函数，一个是角色的 OnEvent() 方法，还有一个是函数 GlobalEvent()，详细代码实现过程请参考下面的代码。

package main

import "fmt"

// 声明角色的结构体

type Actor struct {

}

// 为角色添加一个事件处理函数

func (a \*Actor) OnEvent(param interface{}) {

fmt.Println("actor event:", param)

}

// 全局事件

func GlobalEvent(param interface{}) {

fmt.Println("global event:", param)

}

func main() {

// 实例化一个角色

a := new(Actor)

// 注册名为OnSkill的回调

RegisterEvent("OnSkill", a.OnEvent)

// 再次在OnSkill上注册全局事件

RegisterEvent("OnSkill", GlobalEvent)

// 调用事件，所有注册的同名函数都会被调用

CallEvent("OnSkill", 100)

}

代码说明如下：

第 6 行，声明一个角色的结构体。在游戏中，角色是常见的对象，本例中，角色也是 OnSkill 事件的响应处理方。

第 10 行，为角色结构添加一个 OnEvent() 方法，这个方法拥有 param 参数，类型为 interface{}，与事件系统的函数（func(interface{})）签名一致。

第 16 行为全局事件响应函数。有时需要全局进行侦听或者处理一些事件，这里使用普通函数实现全局事件的处理。

第 27 行，注册一个 OnSkill 事件，实现代码由 a 的 OnEvent 进行处理。也就是 Actor的OnEvent() 方法。

第 30 行，注册一个 OnSkill 事件，实现代码由 GlobalEvent 进行处理，虽然注册的是同一个名字的事件，但前面注册的事件不会被覆盖，而是被添加到事件系统中，关联 OnSkill 事件的函数列表中。

第 33 行，模拟处理事件，通过 CallEvent() 函数传入两个参数，第一个为事件名，第二个为处理函数的参数。

整个例子运行结果如下：

actor event: 100

global event: 100

结果演示，角色和全局的事件会按注册顺序顺序地触发。

一般来说，事件系统不保证同一个事件实现方多个函数列表中的调用顺序，事件系统认为所有实现函数都是平等的。也就是说，无论例子中的 a.OnEvent 先注册，还是 GlobalEvent() 函数先注册，最终谁先被调用，都是无所谓的，开发者不应该去关注和要求保证调用的顺序。

一个完善的事件系统还会提供移除单个和所有事件的方法。

### 2 Prometheus深入

# **PromQL**

### 3

### 4

### 5