第 1 章

初识Go语言

1

在真正地讲解Go语言之前，先让我们对它有个宏观的认识。本章将介绍Go语言是什么、有哪些特性、有哪些优势和不足，以及应该怎样学习它。现在就让我们开始吧。

1.1 Go 语言特性一瞥

我们先来看看Go语言的主要特性。

- 开放源代码的通用计算机编程语言。开放源代码的软件（以下简称开源软件）更容易被修正和改进。因为，几乎所有的互联网用户都可以看到这些源代码，并可以提供自己的意见和建议，甚至还可以参与到实际的开发工作中去。与一些不公开源代码的软件（也可称为闭源软件）相比，开源软件的开发迭代周期要短很多，并且迭代速度要快很多。这正印证了“众人拾柴火焰高”这句俗语。
- 虽为静态类型、编译型的语言，但Go语言的语法却趋于脚本化，非常简洁。这方面的内容，我们会在第3章集中论述。我们可以直接使用各种字面量来表示各种数据类型以及它们的值，并且还可以使用非常简约的方式操作它们。
- 卓越的跨平台支持，无需移植代码。这里的跨平台主要是指跨计算架构和操作系统。Go语言目前已经支持绝大部分主流的计算架构和操作系统了，并且这个范围还在不断地扩大。我们只要下载与之对应的Go语言安装包，并且经过基本一致的安装和配置步骤，就可以使Go语言就绪了。除此之外，在编写Go语言程序的过程中，我们几乎感觉不到不同平台的差异。
- 全自动的垃圾回收机制，无需开发者干预。Go语言程序在运行过程中的垃圾回收工作由Go语言运行时系统负责。不过，Go语言也允许我们进行人工干预。在第三部分，我们会适时地介绍这方面的内容。
- 原生的先进并发编程模型和机制。Go语言拥有自己独特的并发编程模型，其组成部分有Goroutine（也可称为Go程）和Channel（也可称为通道）等。实际上，这部分内容也是本书最重要的主题。在第三部分，我们会一起领略它们的风采。
- 拥有函数式编程范式的特性，函数为一等代码块。Go语言支持多种编程风格，包括面向对象编程和函数式编程。而对函数式编程的最有力的支撑就是Go语言将函数类型视为了

第一等类型。我们会在第3章中说明这一点。

- 无继承层次的轻量级面向对象编程范式。Go语言中的接口与实现之间完全是非侵入式的。这种接口实现方式总是被人们津津乐道。不但如此，在Go语言中只有类型嵌入而没有类型继承。这规避了很多与继承有关的复杂问题，也使类型层次更加简单化了。这部分内容同样会在第3章中展示。
- 内含完善、全面的软件工程工具。Go语言自带的命令和工具相当地强大。通过它们，我们可以很轻松地完成Go语言程序的获取、编译、测试、安装、运行、运行分析等一系列工作。可以看到，这几乎涉及了开发和维护一个软件的所有环节。这可以算是一站式的编程体验了。在本章，我们会简要地浏览一下这些工具。
- 代码风格强制统一。Go语言的安装包中有自己的代码格式化工具。我们可以利用它来统一程序的编码风格。
- 程序编译和运行速度都非常快。由于Go语言的简洁语法，我们可以快速地编写出相应的程序。加之Go语言强大的运行时系统和先进的并发编程模型，Go语言程序可以充分地利用计算环境并运行飞快。在这本书的内容中，我们会通过对Go语言的各个方面的深入介绍使你真正地了解到Go语言如此优秀的原因。
- 标准库丰富，极适合开发服务端程序和Web程序。Go语言是通用的编程语言，但是它也有尤为擅长的几个方面，例如系统级程序和Web程序等，这主要得益于它丰富的标准库。我们会在本书中介绍很多Go语言标准库及其使用方法。但是由于篇幅原因，这些只会是冰山一角。

不知道在看到Go语言如此多的先进特性之后，你是否已经心动了。反正我已经为此折服并感到激动了。这也是我迫不及待地深入研究它并通过编写一本专著把我知道的所有细节与大家分享的原因。

1.2 Go语言的优劣

在软件行业做过一段时间的人都知道，没有万能的编程语言，没有万能开发框架，也没有万能的解决方案。任何新技术的产生都应该归功于一部分人对老旧技术的强烈不满。Go语言也不例外。比如，C的依赖管理、C++的垃圾回收、Java笨重的类型系统和厚重的Java EE规范，以及脚本语言（如PHP、Python和Ruby）的性能，这些都是很多开发者社区经常争论和抱怨的问题。

Go语言是集多编程范式之大成者，体现了优秀的软件工程思想和原则，其特性可以使开发者快速地开发、测试和部署程序，大大提高了生产效率。下面我们来看看与其他主流语言相比，Go语言具有的优势。

- 相对于C/C++来讲，Go语言拥有清晰的依赖管理和全自动的垃圾回收机制，因此其代码量大大降低，开发效率大大提高。
- 相对于Java来讲，Go语言拥有简明的类型系统、函数式编程范式和先进的并发编程模型。因此其代码块更小更简洁、可重用性更高，并可在多核计算环境下更快地运行。

- 对于PHP来讲，Go语言更具通用性和规范性。这使得其更适合构建大型的软件，并能够更好地将各个模块组织在一起。在性能方面，PHP不可与Go同日而语。
- 对于Python/Ruby来讲，Go的优势在于其简洁的语法、非侵入式和扁平化的类型系统和浑然天成的多范式编程模型。与PHP一样，Python的Ruby是动态类型的解释型语言，这就意味着它们的运行速度会比静态类型的编译型语言慢很多。

总而言之，Go语言对于当前大多数主流语言来讲，最大的优势在于具有较高的生产效率、先进的依赖管理和类型系统，以及原生的并发计算支持。因此，Go语言自发布以来就受到了各个领域开发者的关注和青睐。现在，我们来客观地看一下目前Go语言需要加强或改进的地方（虽然有些Gopher并不这么认为）。

- 从分布式计算的角度来看，Go语言的成熟度不及Erlang（现在已经出现了一些这方面的Go语言代码包，我们已经可以看到光明的未来了）。
- 从程序运行速度的角度来看，Go语言虽然已与Java不相上下，但还不及C（差距正在不断地缩小）。
- 从第三方库的角度来看，Go语言的库数量还远远不及其他几门主流语言（比如Java、Python、Ruby等）。不过与Go语言的年纪相比，用它实现的第三方库已经相当多了，并且它们的数量在持续地飞速增长中。

另外，在更深的层面，Go语言标准库中也有些不尽如人意的的地方。具体如下。

- 从语言语法角度来看，Go语言语法里的语法糖并不多，这让许多Python、Ruby爱好者们对它不屑一顾。另外，变量赋值方式多得有点儿累赘了。最让人遗憾的也是我比较在意的一个地方是，Go语言不支持自定义的泛型类型。后面我们会具体讲到这一点。不过，我们还是可以通过一些编程手段弥补这一缺陷的。
- 从并发编程角度来看，Go语言提供的并发模型很强大，但也有一些编写规则需要了解。否则，很容易踩进“坑”里。我其实不提倡把这叫作“坑”。因为这些所谓的“坑”，大都是我们由于对原理不熟悉而自己挖出来的。
- 从垃圾回收角度看，Go语言的垃圾回收采用的是并发的标记清除算法（Concurrent Mark and Sweep，CMS）。虽然是并发的操作，时间比串型操作短很多，但是还是会在垃圾回收期间停止所有用户程序的操作。这一点多少会影响到对实时性要求比较高的应用。不过，在最新的Go语言1.3版本中，这方面的问题已经得到了极大的改善。

虽然Go语言还有一些瑕疵，但从整体来看，它已经是一门非常优秀的通用编程语言了。并且，Go语言在今后的发展上会关注性能、可靠性、可移植性和一些功能增强，所以上述缺憾会随着版本的推进而逐渐减弱和消失。

1.3　怎样学习 Go 语言

作为作者，我当然希望大家能够通过阅读本书来系统地学习Go语言。☺不过，客观地讲，我们还是可以通过很多渠道来学习它的。比如，可以通过浏览Go语言官方网站（http://golang.org）

来了解所有规范和细节，还可以在代码包文档网站（http://godoc.org）中查询到标准库和几乎所有流行的第三方库中的代码包的文档和使用示例。

当然，我们能够利用的资源远不止这些。本书的附录会罗列出更多的学习资源，尤其是中文资源。不过，比查阅相关学习资源更重要的是——动手编码。毕竟我们学习的是软件开发技术，用代码说话是学习这类技术的有效途径。在编码过程中，我们可以感受到这门编程语言的鲜活性。只有积累了足够的代码量，我们才可以感悟到很多更高层次的思想（比如Go语言背后的哲学）。

总之，理论+实践并且交叉积累它们无疑是最好的学习方式。正因为如此，本书也会尽量以此种方式为大家呈现Go语言。我会讲解语言、规范，也会探究原理。与此同时，我会辅以大量代码作为示例，并专门展示和讲解很多规模更大的案例。希望这样的叙述方式能够帮助你更好、更快地掌握Go语言，并愿意使用它来编写程序。如果你觉得有更好的方式可以融入其中，那么请告诉我。我会在后续的版本中加以改进。

1.4 本章小结

这一章相当简单。相信读者已经对Go语言有一个宏观的了解了。虽说任何编程语言都会有不尽如人意的方面，但是我认为Go语言已经做得很好了。并且，它在今后还会持续、快速地改进，变得更加优秀。不论你把学习它当作一种技术投资，还是真正将它作为你的主力编程语言，都是一个非常不错的选择。

第 4 章

流程控制方法

4

在上一章中，我们介绍了Go语言的词法、数据类型以及数据的使用方法，它们都是我们编写程序的根基。在本章，我们将讲述怎样编写成段的甚至是小有规模的代码。在这样的代码中，各种流程控制语句会是我们经常用到的。它们可以制造出各种条件判断、各种循环和各种流程跳转。除此之外，我们还会详细介绍Go语言中与众不同的特殊流程控制方式。

Go语言在流程控制结构方面有些像C语言，但是在很多重要方面都与C不同。Go语言在这方面的特点如下。

- 在Go语言中没有do和while循环，只有一个更加广义的for语句。
- Go语言中的switch语句更加灵活多变。Go语言的switch语句还可以被用于进行类型判断。
- 与for语句类似，Go语言中的if语句和switch语句都可以接受一个可选的初始化子语句。
- Go语言支持在break语句和continue语句之后跟一个可选的标记（Label）语句，以标识需要终止或继续的代码块。
- Go语言中还有一个类似于多路转接器的select语句。
- Go语言中的go语句可以被用于灵活地启用Goroutine。
- Go语言中的defer语句可以使我们更加方便地执行异常捕获和资源回收任务。

另外，在这些语句的构成和语法方面，Go语言也有一些独到之处。我们会在后面一一揭晓。

与前面的章节不同，我们把本章的后面几节设置为了本章的实战环节。我们将会利用之前讲到的知识和方法去解决几个实际问题，以初步达到学有所用的目的。此外，通过对这几个实际问题的剖析和解决，我们还会得到几个在当前Go语言及其标准库中并未提供的高级数据类型（或者说高级数据结构），这会为我们今后的编程提供一定的便利。

4.1 基本流程控制

本节主要介绍大多数现代编程语言都会囊括的流程控制语句。当然，Go语言在流程控制和各种语句的编写方面有它自己的规则和特点。下面，我们就逐一对它们进行介绍。

4.1.1 代码块和作用域

在介绍各种流程控制语句之前，我们先来了解一下什么是代码块。我们在前面的章节中多次

提到过代码块，那么到底什么是代码块呢?

代码块就是一个由花括号“{”和“}”括起来的若干表达式和语句的序列。当然，代码块中也可以不包含任何内容，即为空代码块。

在Go语言的源代码中，除了显式的代码块之外，还有一些隐式的代码块，说明如下。

- 所有Go语言源代码形成了一个最大的代码块。这个最大的代码块也被称为全域代码块。
- 每一个代码包都是一个代码块，即代码包代码块。它们分别包含了当前代码包内的所有Go语言源代码。
- 每一个源码文件都是一个代码块，即源码文件代码块。它们分别包含了当前文件内的所有Go语言源码。
- 每一个if语句、for语句、switch语句和select语句都是一个代码块。
- 每一个在switch或select语句中的子句都是一个代码块。

我们之前说过，每一个标识符都有它的作用域。在Go语言中，使用代码块表示词法上的作用域范围，具体规则如下。

- 一个预定义标识符的作用域是全域代码块。
- 代表了一个常量、类型、变量或函数（不包括方法）的、被声明在顶层的（即在任何函数之外被声明的）标识符的作用域是代码包代码块。
- 一个被导入的代码包的名称的作用域是包含该代码包导入语句的源码文件代码块。
- 一个代表了方法接收者、方法参数或方法结果的标识符的作用域是方法代码块。
- 对于一个代表了常量或变量的标识符，如果它被声明在函数内部，那么它的作用域总是包含它的声明的那个最内层的代码块。
- 对于一个代表了类型的标识符，如果它被声明在函数内部，那么它的作用域就是包含它的声明的那个最内层的代码块。

此外，我们可以在某个代码块中对一个已经在包含它的外层代码块中声明过的标识符进行重声明。并且，当我们在内层代码块中使用这个标识符的时候，它代表的总是它在内层代码块中被重声明时与它绑定在一起的那个程序实体。也就是说，在这种情况下，在外层代码块中声明的那个同名标识符被屏蔽了。例如，有这样一个命令源码文件：

```
package main

import (
    "fmt"
)

var v string = "1, 2, 3"

func main() {
    v := []int{1, 2, 3}
    if v != nil {
        var v int = 123
        fmt.Printf("%v\n", v)
    }
}
```

当我们运行这个命令源码文件后，标准输出上会打印出什么内容呢？又或者，对它的编译是否会成功呢？读者可以根据上面的规则先自己思考一会儿。

答案揭晓：打印的内容是123。在这个命令文件中，我们首先在顶层代码块中声明了一个变量v，然后在main函数的代码块中的第一个行也声明了一个名为v的变量。此时，在main函数内部的变量v屏蔽了顶层的变量v。

我们在3.3.3节讲过，基本数据类型的值都无法与空值nil进行进行判等。之所以main函数中的第二行代码没有造成编译错误，就是因为这里的v代表的是一个切片类型值而不是一个string类型值。

我们再来看if代码块中的代码。其中的第一行代码用于声明一个名为v的变量（又是一个）。这个变量v屏蔽了在main函数中的第一个行声明那个变量v。现在，在if代码块内部，v代表的已经是一个int类型值了，而不是一个切片类型值，也不是一个string类型值。因此，if代码块中的第二行代码会向标准输出上打印的内容是123。

我们现在了解了代码块的含义和分类，以及标识符的作用域的推导方法。这对于我们编写稍具规模的Go语言程序非常重要。这些知识和规则会指导我们编写正确的代码。

4.1.2 if语句

Go语言中的if语句会根据一个布尔类型的表达式的结果来执行两个分支中的一个。如果那个表达式的结果值是true，那么if分支会被执行，否则else分支会被执行。

1. 组成和编写方法

Go语言的if语句总是以关键字if开始。在这之后，可以后跟一条简单语句（当然也可以没有），然后是一个作为条件判断的布尔类型的表达式以及一个用花括号“{”和“}”括起来的代码块。

常用的简单语句包括短变量声明、赋值语句和表达式语句。除了特殊的内建函数和代码包unsafe中的函数，针对其他函数和方法的调用表达式和针对通道类型值的接收表达式都可以出现在语句上下文中。换句话说，它们都可以称为表达式语句。在必要时，我们还可以使用圆括号“(”和“)”将它们括起来。其他的简单语句还包括发送语句、自增/自减语句和空语句。我们在后面的章节中会陆续介绍它们。

回归正题。根据上面描述的if语句的组成结构，我们可以很轻松地写出最简单的if语句，例如：

```
if 100 < number {
    number++
}
```

当然，if语句也可以有else分支，它由else关键字和一个用花括号“{”和“}”括起来的代码块。例如：

```
if 100 < number {
    number++
```

```
} else {
    number--
}
```

可能读者已经注意到了，其中的条件表达式100 < number并没有被圆括号括起来。实际上，这也是Go语言的流程控制语句的特点之一。另外，跟在条件表达式和else关键之后的两个代码块必须由花括号“{”和“}”括起来。这一点是强制的，不论代码块包含几条语句以及是否包含语句都是如此。

Go语言建议我们不要把括有代码块的花括号写在同一个代码行上。也就是说，左花括号“{”、其中的语句和右花括号“}”应该存在于不同的代码行上，即使我们编写的是最简单的那种if语句。这是一种良好的编码风格，我们理应这样去做，特别是当代码块中包含像return语句这样的终止语句的时候。

因为if语句可以接受一条初始化子语句，所以我们常常会使用它来初始化一个变量：

```
if diff := 100 - number; 100 < diff {
    number++
} else {
    number--
}
```

可以看到，初始化子句和条件表达式之间是需要用分号“;”分隔的。

与其他高级编程语句相同，Go语言的if语句也支持串联。例如：

```
if diff := 100 - number; 100 < diff {
    number++
} else if 200 < diff {
    number--
} else {
    number -= 2
}
```

正如上面的示例所展示的那样，我们需要把第二条if语句追加到第一条if语句中的else关键字的后面。同样地，在它们后面还可以追加一个else分支。如果我们要再串联第三条if语句，方法也是如此。原则上，我们可以串联任意多个if语句。不过要注意，我们把被串联在一起的if语句看作一个整体。所有的if语句中的条件表达式会共同实现条件筛选的功能。例如，在上面的示例中，当变量diff的值小于100时，第一个分支会被执行。而当变量diff的值不小于100但小于200时，第二个分支会被执行。若以上条件都不满足，则第三个分支会被执行。

在上面的示例中，我们看到了两个特殊符号：++和--。它们分别代表了自增语句和自减语句。注意，它们并不是操作符。++的作用是把它左边的标识符代表的值与无类型常量1相加并将结果再赋给左边的标识符，而--的作用则是把它左边的标识符代表的值与无类型常量1相减并将再结果赋给左边的标识符。也就是说，自增语句number++与赋值语句number = number + 1具有相同的语义，而自减语句number--则与赋值语句number = number - 1具有相同的语义。另外，在++和--左边的并不仅限于标识符，还可以是任何可被赋值的表达式，比如应用在切片类型值或字典类型值之上的索引表达式。

2. 更多惯用法

由于在Go语言中一个函数可以返回多个结果，因此我们常常会把在函数执行期间出现的常规错误也作为结果之一。这已经成为了编写Go语言程序的一个惯例。

例如，标准库代码包os中的函数Open就是这样的一个函数。它的声明如下：

```
func Open(name string) (file *File, err error)
```

函数os.Open返回的第一个结果是与已经被“打开”的文件相对应的*File类型的值，而第二个结果则是代表了常规错误的error类型的值。我们在之前说过，error是一个预定义的接口类型。所有实现它的类型都应该被用于描述一个常规错误。

在导入代码包os之后，我们可以像这样调用其中的Open函数：

```
f, err := os.Open(name)
```

在通常情况下，我们应该先去检查变量err的值是否为nil。如果变量err的值不为nil，那么就说明os.Open函数在被执行过程中发生了错误。这时的f变量的值肯定是不可用的。这已经是一个约定俗成的规则了。因此，调用os.Open函数的前4行代码一般都会是这样的：

```
f, err := os.Open(name)
if err != nil {
    return err
}
```

总之，if语句常被用来检查常规错误。

另外，if语句常被作为卫述语句。卫述语句是指被用来检查关键的先决条件的合法性并在检查未通过的情况下立即终止当前代码块的执行的语句。其实，在上一个示例中的if语句就是卫述语句中的一种。它在有错误发生的时候立即终止了当前代码块的执行并将错误返回给外层代码块。另一个例子是这样的：

```
func update(id int, deptment string) bool {
    if id <= 0 {
        return false
    }
    // 省略若干条语句
    return true
}
```

在函数update开始处的那条if语句就属于卫述语句。我们还可以对这个函数稍加改造一下，像这样：

```
func update(id int, deptment string) error {
    if id <= 0 {
        return errors.New("The id is INVALID!")
    }
    // 省略若干条语句
    return nil
}
```

如此一来，update函数返回的结果不但可以表示在函数执行期间是否发生了错误，而且还可以体现出错误的具体描述。不过，这需要我们事先导入标准库的代码包errors。

我们在介绍if语句的典型应用场景的同时，还透露了一部分常规错误生成和处理的方法。了解与程序异常处理有关的更多细节，请参见4.3节。

4.1.3　switch语句

与if语句类似，switch语句也提供了一种多分支执行的方法。它会用一个表达式或一个类型说明符与每一个case进行比较并决定执行哪一个分支。

1. 组成和编写方法

语句switch可以使用表达式或者类型说明符作为case判定方法。因此，switch语句也就可以被分成两类：表达式switch语句和类型switch语句。在表达式switch语句中，每一个case携带的表达式都会与switch语句要判定的那个表达式（也被称为switch表达式）相比较。而在类型switch语句中，每一个case所携带的不是表达式而是类型字面量，并且switch语句要判定的目标也变成了一个特殊的表达式。这个特殊表达式的结果是一个类型而不是一个类型值。下面我们分别对这两种switch语句进行说明。

2. 表达式switch语句

在表达式switch语句中，switch表达式和case携带的表达式（也被称为case表达式）都会被求值。对这些表达式的求值是自左向右、自上而下进行的。第一个与switch表达式的求值结果相等的case表达式所关联的那个分支会被执行，而其他分支会被忽略。如果没有找到匹配的case表达式并且存在default case，那么default case所关联的那个分支会被执行。default case最多只能有一个，并且它并不是必须作为switch语句的最后一个case出现。此外，如果在switch语句中没有显式的switch表达式，那么true将会被作为switch表达式。

我们先来看switch语句的一个简单形式：

```
switch content {
default:
    fmt.Println("Unknown language")
case "Python":
    fmt.Println("A interpreted Language")
case "Go":
    fmt.Println("A compiled language")
}
```

一般情况下，switch关键字之后会紧跟一个switch表达式。这种情况下，switch表达式中涉及的标识符都必须是已经被声明过的。作为更复杂一点的形式，我们还可以在这两者之间插入一条简单语句。像这样：

```
switch content := getContent(); content {
default:
    fmt.Println("Unknown language")
case "Python":
    fmt.Println("A interpreted Language")
case "Go":
    fmt.Println("A compiled language")
}
```

在这个示例中，我们在switch语句中先调用了getContent函数，并且把它的结果值赋给了新声明的变量content，后面紧接着的就是对content的值的判定过程。注意，简单语句content := getContent()会在switch表达式content被求值之前被执行。

现在来看case语句。一条case语句由一个case表达式和一个语句列表组成，并且这两者之间需要用冒号“:”分隔。在上例的switch语句中，一共有3个case语句。注意，default case是一种特殊的case语句。

一个case表达式由一个case关键字和一个表达式列表组成。注意，这里说的是一个表达式列表，而不是一个表达式。这意味着，一个case表达式中可以包含多个表达式。现在，我们利用这一特性来改造一下上面的switch语句，改造结果如下：

```
switch content := getContent(); content {
default:
    fmt.Println("Unknown language")
case "Ruby", "Python":
    fmt.Println("A interpreted Language")
case "C", "Java", "Go":
    fmt.Println("A compiled language")
}
```

大家知道，解释型编程语言和编译型编程语言都不止一个。所以，我们把几个解释型编程语言的名称放在同一个case表达式中，而把几个编译型编程语言都放到另一个case表达式中。每一个代表了编程语言名称的string类型值都作为了一个独立的表达式。在同一条case表达式中，多个表达式之间需要用逗号“,”分隔。当然，我们也可以把每一个string类型值都单独放在一个case表达式中。

在一条case语句中的语句列表的最后一条语句可以是fallthrough语句。在一条表达式switch语句中，一条fallthrough语句会将流程控制权转移到下一条case语句上。fallthrough语句极其直接和简单，仅由英文单词fallthrough组成。请看下面的示例：

```
switch content := getContent(); content {
default:
    fmt.Println("Unknown language")
case "Ruby":
    fallthrough
case "Python":
    fmt.Println("A interpreted Language")
case "C", "Java", "Go":
    fmt.Println("A compiled language")
}
```

这个示例是上一个示例的重构版本。其中的代码的功能与上一个示例中代码的功能是完全一致的。原来的表达式列表"Ruby", "Python"已经被拆分到了两个case语句当中。并且，包含了表达式"Ruby"的case语句的语句列表只包含了一条fallthrough语句，而在包含表达式"Python"的case语句的语句列表中包含了原先与case表达式case "Ruby", "Python"对应的那条语句。虽然有了这些更改，但是当变量content的值与"Ruby"相等的时候，在标准输出上打印出的内容依然会

是A interpreted Language。也就是说，虽然content的值与"Ruby"相等，但是与case "Python"对应的语句列表也会被执行。这是因为在case "Ruby"的语句列表的最后，fallthrough语句使流程控制权得以流转到了case "Python"上。不过，要注意的是，这种控制权流转并不存在传递性。在上面的示例中，当content的值为"Ruby"时，case "C", "Java", "Go"的语句列表一定不会被执行。另外，fallthrough语句只能够作为case语句中的语句列表的最后一条语句。更重要的是，fallthrough语句不能出现在最后一条case语句的语句列表中。

此外，break语句也可以出现在case语句中的语句列表中。一条break语句由一个break关键字和一个可选的标记组成。如果这两者都存在，那么它们之间应该有空格“ ”分隔。例如：

```
switch content := getContent(); content {
default:
    fmt.Println("Unknown language")
case "Ruby":
    break
case "Python":
    fmt.Println("A interpreted Language")
case "C", "Java", "Go":
    fmt.Println("A compiled language")
}
```

在break语句被执行后，包含它的switch语句、for语句或select语句的执行会被立即终止。流程控制权将会被转移到这些语句后面的语句上。请读者修改一下上面示例中的代码，使当content的值为"Ruby"或"Python"的时候，不输出任何内容而直接结束当前的switch语句的执行。这要求使用break语句来做，当然也可以用上fallthrough语句。

包含标记的break语句是与标记（Label）语句一起配合使用的。这个我们后面再讲。

3. 类型switch语句

类型switch语句将对类型进行判定，而不是值。在其他方面，它都与表达式switch语句如出一辙。类型switch语句中的switch表达式很特殊。这个switch表达式的表现形式与类型断言表达式有几分相似。但是与类型断言表达式不同的是，它使用关键字type来充当欲判定的类型，而不是使用一个具体的类型字面量。下面是一个简单的例子：

```
switch v.(type) {
case string:
    fmt.Printf("The string is '%s'.\n", v.(string))
case int, uint, int8, uint8, int16, uint16, int32, uint32, int64, uint64:
    fmt.Printf("The integer is %d.\n", v)
default:
    fmt.Printf("Unsupported value. (type=%T)\n", v)
}
```

在阅读这段示例之前，如果读者不知道或者忘记了类型断言表达式是怎么一回事，请先去3.1.6节寻找和学习一下相关知识。因为，类型断言对于正确理解这段示例代码很重要。

我们先从形式上介绍一下类型switch语句的特点，请结合上面的示例来理解下面的内容。首先，它的switch表达式会包含一个特殊的类型断言表达式，例如v.(type)。这个我们刚刚已经说

过。其次，每个case表达式中包含的都是类型字面量而不是表达式，处于同一个case表达式中的多个类型字面量之间同样也需要用逗号“,”分隔。请看上面示例中的前两个case表达式。

现在我们来具体分析这段示例代码。这条类型switch语句共包含了3条case语句。第一条case语句中的case表达式包含了类型字面量string。这就意味着，如果v的类型是string类型，那么该分支就会被执行。在这个分支中，我们使用类型断言表达式v.(string)把v的值转换成了string类型的值，并以特定格式打印出来。第二条case语句中的类型字面量有多个，包括了所有的整数类型。这就意味着只要v的类型属于整数类型，该分支就会被执行。注意，byte类型是uint8类型的别名类型，而rune类型则是uint32类型的别名类型。因此，如果v是byte类型或rune类型的，第二个分支也会被执行。由于任何整数类型都不能表示Go所支持的全部范围的整数（例如，int64类型和uint64类型的数值范围都很大且双方有很大重叠，但是其中一方的数值范围依然不能完全覆盖全部的数值范围），因此在这个分支中，我们并没有使用类型断言表达式把v的值转换成任何一个整数类型的值，而是利用fmt.Printf函数直接打印出了v所表示的整数值（注意格式化字符串中的%d）。如果v的类型既不是string类型也不是整数类型，那么default case的分支将会被执行。此分支中的那行语句会在标准输出上打印出提示内容并附上v的动态类型（注意格式化字符串中的%T）。顺便提一下，我们在这个示例中展现了fmt.Printf函数的一部分使用方法。关于传递给它的第一个参数中的%s、%d和%T的含义以及关于它的使用说明，请读者参看Go语言官方文档网站中的代码包fmt的文档页面。在那里，读者还可以看到该代码包中的其他打印函数。我们在后面的章节中会陆续用到这些打印函数。

我们在编写类型switch语句的时候，需要遵守两个特殊规则。首先，变量v的类型必须是某个接口类型。这也是理所当然的。如果v的具体类型已经确定了，那么我们也就没必要用类型switch语句来判定它了。其次，case表达式中的类型字面量必须是v的类型的实现类型。一个通用的方案是，把变量v的类型设置为interface{}（空接口）类型。由于任何Go语言数据类型都是interface{}的实现类，因此这样就等于支持最广义的类型判定了。尤其是当我们判定v的类型是否为某个或某些基础数据类型的时候，应该也必须这样做。

与表达式switch语句相同，我们在类型switch语句中的switch关键字和switch表达式之间也可以插入一条简单语句。另外，在类型switch语句中，case表达式中的类型字面量可以是nil。在前面的那个示例中，如果v的值是nil，那么表达式v.(type)的结果值也会是nil。因此，当这种情况发生时，如果存在包含了nil的case表达式，那么与它相对应的那个分支就会被执行。

与表达式switch语句不同的是，fallthrough语句不允许出现在类型switch语句中的任何case语句的语句列表中。这一点需要特别注意。

最后，值得特别提出的是，类型switch语句的switch表达式还有一种变形写法。我们使用这个变形写法对前面示例中的类型switch语句进行了重构，如下：

```
switch i := v.(type) {
case string:
    fmt.Printf("The string is '%s'.\n", i)
case int, uint, int8, uint8, int16, uint16, int32, uint32, int64, uint64:
    fmt.Printf("The integer is %d.\n", i)
```

```
default:
    fmt.Printf("Unsupported value. (type=%T)\n", i)
}
```

我们看到，现在处在switch表达式的位置上的是i := v.(type)。这实际上是一个短变量声明。当存在这种形式的switch表达式的时候，就相当于这个变量（这里是i）被声明在了每个case语句的语句列表的开始处。在每个case语句中，变量i的类型都是不同的。它的类型会和与它处于同一个case语句的case表达式包含的类型字面量所代表的那个类型相等。例如，在上面的示例中，第一个case语句相当于：

```
case string:
    i := v.(string)
    fmt.Printf("The string is '%s'.\n", i)
```

如果相应的case表达式包含多个类型字面量，那么它的类型会与表达式v.(type)的求值结果所代表的类型一致。例如，如果v的动态类型是uint16类型，那么第二个case语句相当于：

```
case int, uint, int8, uint8, int16, uint16, int32, uint32, int64, uint64:
    i := v.(uint16)
    fmt.Printf("The integer is %d.\n", i)
```

综上所述，这种形式的switch表达式为我们提供了便利。我们不再需要在每个case语句中分别对那个欲判定类型的值进行显式地类型转换了。

4. 更多惯用法

除了前面讲到的一些常规用法之外，我们还可以把switch语句作为串联的if语句的一种替代品。这需要使用switch语句的另一种变形来实现。这种变形去掉了switch语句在通常情况下会包含的switch表达式。在switch表达式缺失的情况下，该switch语句的判定目标会被视为布尔类型值true。也就是说，其中的所有case表达式的结果值都应该是布尔类型的。并且，自上而下，第一个结果值为true的case表达式所对应的分支会被执行。例如：

```
switch {
case number < 100:
    number++
case number < 200:
    number--
default:
    number -= 2
}
```

假设number是整数类型的。当number小于100时第一个分支会被执行，而当number不小于100但小于200时第二个分支会被执行，否则第三个分支会被执行。请读者仔细阅读这条switch语句，它的功能与我们在上一小节讲串联if语句时给出的那个if语句的功能完全一致。由此可知，这种switch语句的变形完全可以替代串联if语句，并且还能够提供更好的代码可读性。

作为switch语句的一种变形，在它的switch关键字和switch表达式之间也可以有一条简单语句。虽然这种switch语句并没有switch表达式， 但是为了不让Go语言和代码阅读者把这条简单语句误认为switch表达式，我们还是需要在该条简单语句的后面加上分号“;”，尽管这样看起来

会有些奇怪。例如：

```
switch number := 123; {
case number < 100:
    number++
case number < 200:
    number--
default:
    number -= 2
}
```

我们在前面介绍的各种switch语句都有各不相同的应用场景。只要我们真正地理解了它们所代表的含义，就可以根据实际需要正确地选用它们。

4.1.4 for语句

一条for语句会根据既定的条件重复执行一个代码块。这种重复执行一个代码块的行为也称为循环或迭代。迭代的开始和结束是受到既定条件的控制的。这个条件或由for子句直接给出，或从range子句中获得。

1. 组成和编写方法

一个最简单的for语句形式是，for语句一直重复执行一个代码块直到作为条件的表达式的求值结果为false。这个条件会在每次执行该代码块之前被求值。如果没有显式地指定该条件，那么true将会被作为缺省的条件。在这种情况下，如果在被重复执行的代码块中不存在break语句或者break语句总是没有被执行的机会，那么就产生了一个无限循环（或称死循环）。请看如下示例：

```
// number是一个int类型的变量

for number < 200 { // 当number大于等于200时for循环会退出。
    number += 2
}

for { // 很不幸，这是一个死循环，该代码块永远会被重复的执行下去……
    number++
}
```

2. for子句

一条for语句可以携带一个for子句，并可以使用这个for子句提供的条件来对迭代进行控制。除了条件，for子句还可以包含一条用来初始化上下文的简单语句（以下简称初始化子句）和一条用来为代码块的执行做后置处理的简单语句（以下简称后置子句）。for子句的这3个部分是有固定排列顺序的，即初始化子句在左、条件在中、后置子句在右。并且，它们之间需要用分号“;”来分隔。我们可以在编写for子句的时候省略掉其中的任何部分。但是，为了避免歧义，即使其中的一个部分被省略掉了，与它相邻的分隔符“;”也必须被保留。

在一般情况下，初始化子句为赋值语句或短变量声明，而后置子句则为自增语句或自减语句。当然，它们也可以是别的简单语句。但是要注意，后置语句一定不能是短变量声明。另外，初始

化子句总会在充当条件的表达式被第一次求值之前执行，且只会执行一次，而后置子句的执行总会在每次代码块执行完成之后紧接着进行。

下面我们来看一组示例：

```
for i := 0; i < 100; i++ {
    number++
}

var j uint = 1
for ; j%5 != 0; j *= 3 { // 省略初始化子句
    number++
}

for k := 1; k%5 != 0; { // 省略后置子句
    k *= 3
    number++
}
```

在for子句的初始化子句和后置子句同时被省略或者其中的所有部分都被省略的情况下，分隔符“;”可以被省略。这时，for语句的形式就和我们在本小节开始处描述的相同了。

3. range子句

一条for语句可以携带一个range子句，从而可以迭代出一个数组或切片值中的每个元素、一个字符串值中的每个字符或者一个字典值中的每个键值对。甚至，它还可以被用于持续接收一个通道类型值中的元素。随着迭代的进行，每一次被获取出的迭代值（元素、字符或键值对）都会被赋给相应的迭代变量，然后这些迭代变量将会被带入马上要被执行的for语句的代码块中。例如：

```
ints := []int{1, 2, 3, 4, 5}
for i, d := range ints {
    fmt.Printf("%d: %d\n", i, d)
}
```

又例如：

```
var i, d int
for i, d = range ints {
    fmt.Printf("%d: %d\n", i, d)
}
```

可以看到，range子句由3部分组成。其中，range关键字总是会处于中间的位置上。在range关键字右边的应该是一个表达式。这个表达式常被称为range表达式。其结果值可以是一个数组值、一个指向数组值的指针值、一个切片值、一个字符串值或者一个字典值，也可以是一个允许接收操作的通道类型值。注意，一般情况下，range表达式只会在迭代开始前被求值一次。当然，例外情况是存在的，不过我们一会儿再对此进行说明。

在range关键字左边的是相应的表达式列表或是标识符列表。如果是表达式列表或不包含未被声明过的标识符的标识符列表，那么在该列表与range关键字之间就必须由赋值操作符=分隔。如果是包含了未被声明过的标识符的标识符列表，那么在该列表与range关键字之间就必须由赋

值操作符:=分隔。显然，前者代表普通赋值，后者代表声明并赋值。不论怎样，左边列表中的每一个元素都代表了一个迭代变量。它们会在每一次迭代的时候被重用（重新赋值或重新声明并赋值）。对于未被声明过的标识符，它所代表的变量的类型会与相应的迭代值的类型相等，并且它的作用域是包含其声明的for语句。对于已被声明过的标识符和表达式，它们的类型与相应的迭代值之间必须满足赋值规则。并且，在for语句中对它们的更改不会因for语句的执行结束而失效。例如，在下面的for语句中，在range关键字和赋值操作符左边的就是一个表达式列表：

```
ints := []int{1, 2, 3, 4, 5}
length := len(ints)
indexesMirror := make([]int, length)
elementsMirror := make([]int, length)
var i int
for indexesMirror[length-i-1], elementsMirror[length-i-1] = range ints {
    i++
}
```

与range表达式不同，在range关键字左边的表达式列表中的表达式在每一次迭代的时候都会被求值一次。也就是说，它们被求值的次数与for语句的代码块被执行的次数相同。并且，对它们的求值总是会在代码块被执行之前进行。此外，由于每一次迭代的产出值（也就是迭代值）都与迭代变量共同组成了赋值语句，所以它们也具备赋值语句所拥有的一切特性，比如赋值的执行阶段和赋值顺序。

到这里，读者可能会有一个疑问，为什么我们在刚刚的示例中每次可以从切片值中迭代出两个值？

实际上，对于切片值来说，携带range子句的for语句每次迭代出的那两个值并不都是该切片值中的元素。并且，随着range表达式的结果值的不同，range子句会有不同的表现，具体如下。

- 对于一个数组值、一个指向数组值的指针值或一个切片值a来说，range循环的迭代产出值可以是一个也可以是两个。并且，迭代的顺序是与索引值（也就是第一个迭代值）的递增顺序一致的。如果只产出一个迭代值，那么range循环产生的迭代值会是从0到len(a)-1的多个int类型的索引值，并且不会发生根据索引值定位元素值的动作。另外，如果切片值为nil，那么迭代次数将会是0。
- 对于一个字符串值，range子句将会遍历其中的所有Unicode代码点。我们知道，在底层，字符串值其实是由其中的每个字符的UTF-8编码值组成并存储的。一个UTF-8编码值既可以由一个rune类型值代表，也可以由一个[]byte类型值代表。因此，我们可以把一个字符串值看成一个[]rune类型值或一个[]byte类型值。图4-1展示了这三者之间的对应关系。

字符串类型值："Golang爱好者"

[]byte类型值	0x47	0x6f	0x6c	0x61	0x6e	0x67	0xe7	0x88	0xb1	0xe5	0xa5	0xbd	0xe8	0x80	0x85
索引值	0	1	2	3	4	5	6			9			12		
[]rune类型值	'G'	'o'	'l'	'a'	'n'	'g'	'爱'			'好'			'者'		
索引值	0	1	2	3	4	5	6			7			8		

图4-1 range迭代与字符串

由图4-1可知，对于一个字符串值来说，在一个连续的迭代之上产出的索引值（第一个迭代值）即是其中某一个Unicode代码点（与rune类型值一一对应）的UTF-8编码值中的第一个字节在与其所属的[]byte类型值上的索引值。对照图4-1，当range表达式的结果值是字符串值"Golang爱好者"时，range子句的第一次迭代的第一个迭代值为int类型值0，第二个迭代值（若需要）为rune类型值'G'。而它的第八次迭代的第一个迭代值为int类型值9，第二个迭代值（若需要）为rune类型值'好'。注意，当迭代遭遇非法的UTF-8编码值时，第二个迭代值就会是'�'（对应的Unicode代码点为U+FFFD），且下一次迭代将会从在该非法UTF-8编码值之后的第一个字节开始。

- 对于一个字典值来说，它的迭代顺序是不固定的。如果字典值中的键值对在还没有被迭代到的时候就被删除了，那么相应的迭代值将不会被产出。另一方面，如果我们在字典值被迭代过程中向其添加了新的键值对，那么相应的迭代值是否会被产出是不确定的。对字典值迭代的产出值的数量可以是一个也可以是两个。如果字典值为nil，那么迭代次数将会是0。
- 对于通道类型值，这种迭代的效果类似于连续不断的从该通道中接收元素值，直到该通道被关闭。并且，对通道类型值的迭代每次都只会产出一个值。注意，如果通道类型值为nil，那么range表达式将会被永远地阻塞！

为了方便快速查询，我们在对上面的描述进行了简化并绘制了表4-1。

表4-1　range子句的迭代产出

range表达式的类型	第一个产出值	第二个产出值（若显式获取）	备　注
a：[n]E、*[n]E或[]E	i：int类型的元素索引值	与索引对应的元素的值a[i]，类型为E	a为range表达式的结果值。n为数组类型的长度。E为数组类型或切片类型的元素类型
s：string类型	i：int类型的元素索引值	与索引对应的元素的值s[i]，类型为rune	s为range表达式的结果值
m/：map[K]V	k：键值对中的键的值，类型为K	与键对应的元素值m[k]，类型为V	m为range表达式的结果值。K为字典类型的键的类型。V为字典类型的元素类型
c：chan E或	e：元素的值，类型为E		c为range表达式的结果值。E为通道类型的元素的类型

综上所述，如果range表达式的求值结果是一个通道类型值，那么仅会产出一个迭代值。也就是说，这时在range关键字和赋值操作符左边的表达式或标识符就只能有一个。否则，产出的迭代值可以是一个也可以是两个，这取决于在range关键字和赋值操作符左边的表达式或标识符的数量。例如，下面的这个for语句与我们在讲range子句时的第一个示例中的那个for语句在语义上是等价的：

```
ints := []int{1, 2, 3, 4, 5}
for i := range ints {
    d := ints[i]
    fmt.Printf("%d: %d\n", i, d)
}
```

如果range表达式的结果类型是某个数组类型或某个指向数组值的指针类型，同时它只被要求产出第一个迭代值，那么这个range表达式就只会被部分求值。这是什么意思呢？我们都知道数组值的长度是其类型的一部分。因此，对于上面这类情况，我们只需要求得到range表达式的结果的类型就可为后续迭代提供足够的支持了。当这个长度是常量的时候，该range表达式将不会被求值。此处的“长度是常量”的意思是，可以推断在该range表达式的求值结果上应用内建函数len所得到的结果一定是一个常量。这个推断的方法我们在上一章讲内建函数len的时候已经介绍过，这里就不再赘述了。

4. 更多惯用法

我们在前面说过，对于所有可迭代的数据类型的值来说，我们都可以要求每次迭代只产出第一个迭代值。例如：

```
m := map[uint]string{1: "A", 6: "C", 7: "B"}
var maxKey uint
for k := range m {
    if k > maxKey {
        maxKey = k
    }
}
```

但是，我们并没有介绍怎样忽略掉第一个迭代值而只使用第二个迭代值的方法。有些遗憾，与for语句相关的语法中并没有针对此问题的解决方法。并且，将第一个迭代值赋给迭代变量但不使用它也不是一个可行的办法。这会造成一个编译错误。因为Go语言编译器不允许程序中有未被使用的变量出现。不过，如果我们稍稍转变一下思考角度的话，这个问题就相当好解决了，也许读者早已经想到了这个方法。既然说迭代值和迭代变量之间是赋值和被赋值的关系，那么我们当然可以把迭代值赋给一个空标识符。这一点在我们先前讲的赋值规则的时候已有说明。这样也可以避免编译错误的发生。我们同样以前一个示例中的变量m为例，如下：

```
var values []string
for _, v := range m {
    values = append(values, v)
}
```

这种做法在for语句的编写过程中是很常用的。

在for语句中，我们还可以使用break语句来终止for语句的执行。若有一个变量namesCount，它的声明如下：

```
var namesCount map[string]int
```

这个变量的值包含了某个网站的所有用户昵称及其重复次数（用户昵称可以重复）。也就是说，这个字典值的键表示用户昵称，而值则代表了使用该昵称的用户的数量。现在我们想从中查找到所有的只包含中文的用户昵称的计数信息。这一需求的简单实现如下：

```
targetsCount := make(map[string]int)
for k, v := range namesCount {
    matched := true
    for _, r := range k {
```

```
            if r < '\u4e00' || r > '\u9fbf' {
                matched = false
                break
            }
        }
        if matched {
            targetsCount[k] = v
        }
    }
```

在上面这段代码中，我们使用了嵌套的for语句。外层的for语句对变量namesCount的值进行迭代，也就是说它会遍历其中的每一个键值对。而内层的for语句则对每个用户昵称中的每个字符进行遍历。如果用户昵称中包含了非中文字符，那么我们会设置一个标志（由变量matched代表）并且终止内层的for循环。只有在标志的值为true时，我们才会把相应的键值对添加到变量targetsCount中。break语句只会终止直接包含它的那条for语句的执行。因此，当碰到一个非全中文的用户昵称时，我们虽然使用break语句终止了内层for语句的执行，但是当外层for语句的代码块被执行完毕后，它的下一次迭代仍然会进行。

现在我们稍微改动一下上面的需求，加上一个限制条件：发现第一个非全中文的用户昵称的时候就停止查找。刚才提到，break语句只能终止直接包含它的那条for语句的执行。那么我们怎样在发现第一个非全中文的用户昵称之后就直接终止外层for语句的执行呢？最简单的解决方法是使用一个作为辅助标志的变量和两个break语句，代码如下：

```
targetsCount := make(map[string]int)
for k, v := range namesCount {
    matched := true
    for _, r := range k {
        if r < '\u4e00' || r > '\u9fbf' {
            matched = false
            break
        }
    }
    if !matched {
        break
    } else {
        targetsCount[k] = v
    }
}
```

这段代码与上一段代码非常类似，我们只不过对外层for语句的代码块中的最后几行代码做了一些修改。当作为辅助标志的变量的值为false的时候直接退出外层循环。这种做法很简单也很直观。不过，我们一定要使用那个辅助标志吗？

我们之前说过，break语句可以与标记（Label）语句一起配合使用。在我们改进上面的代码之前，先来介绍一下标记语句。

一条标记语句可以成为goto语句、break语句或continue语句的目标。标记语句中的标记只是一个标识符，它可以被放置在任何语句的左边以作为这个语句的标签。标记和被标记的语句之间需要用冒号“:”来分隔。一个标记、一个冒号“:”和那个被标记的语句就组成了一条标记语句，

就像这样：

```
L:
    for k, v := range namesCount {
        // 省略若干条语句
    }
```

需要注意的是，既然标记也是一个标识符，那么当它在未被使用的时候也同样会造成一个编译错误。那么怎样使用标记呢？其中一种方法就是让它成为break的目标：

```
L:
    for k, v := range namesCount {
        if v > 100 {
            fmt.Printf("The matched name: %v\n", k)
            break L
        }
    }
```

如上所示，我们在break语句的后面追加了一个空格“ ”和一个标记。这就意味着，终止执行的对象就是标记代表的那条语句。因此，执行break L语句就会终止L标记的那条for语句的执行，从而退出那个for循环转而执行其后面的语句（如果有的话）。

好了，我们现在来看怎样使用break语句和标记语句来完成我们刚刚提出的第二个需求。代码如下：

```
    targetsCount := make(map[string]int)
L:
    for k, v := range namesCount {
        for _, r := range k {
            if r < '\u4e00' || r > '\u9fbf' {
                break L
            }
        }
        targetsCount[k] = v
    }
```

可以看到，与之前的那个简单的解决方法相比，for语句中的matched变量被删除掉了，同时还省略掉了一条if语句。取而代之的是标记L和携带它的break语句。这确实减少了一些代码量，不是吗？这样的语句组合可以让我们非常方便地跳出嵌套的for语句，而且比使用辅助标志更加清晰。

现在，让我们回到原始需求上来。还记得吗？只实现了第一个需求的代码中依然用到了作为辅助标志的matched变量。这里的辅助标志可以去掉吗？答案是肯定的，使用continue语句可以达到这一目的。

实际上，Go语言中的continue语句只能在for语句中被使用。continue语句会使直接包含它的那个for循环直接进入下一次迭代。也就是说，当次迭代不会执行在该continue语句后面那些语句（它们被跳过了）而直接结束。例如，实现原始需求那段代码可以被修改成这样：

```
targetsCount := make(map[string]int)
for k, v := range namesCount {
    matched := true
```

```
        for _, r := range k {
            if r < '\u4e00' || r > '\u9fbf' {
                matched = false
                break
            }
        }
        if !matched {
            continue
        }
        targetsCount[k] = v
    }
```

在外层for语句的代码块中的最后那几行代码被修改了。其逻辑由如果matched的值是true就把当前键值对添加到targetsCount的值中改为了如果matched的值是false就不把当前键值对添加到targetsCount的值中。没错，这种修改实在没什么意义。

与break语句相同，continue语句也可以与标记语句组合起来使用。这样一来，continue语句的功能就会得到放大。下面我们就来看看真正的改进版本的代码：

```
    targetsCount := make(map[string]int)
L:
    for k, v := range namesCount {
        for _, r := range k {
            if r < '\u4e00' || r > '\u9fbf' {
                continue L
            }
        }
        targetsCount[k] = v
    }
```

在这段代码中，我们已经不需要辅助标志了。如果continue语句携带了标记，那么它就会使该标记代表的那个for循环直接进入下一次迭代。在该示例中，语句continue L使得外层的for循环直接进入到了下一次迭代。也就是说，当它被执行的时候，外层的for语句的当次迭代出的那个键值对不会被添加到targetsCount的值中。这使得这段代码与实现原始需求的前两个版本的代码拥有相同的语义。通过continue语句和标记语句的组合使用，我们用了更少的代码且更加清晰地实现了那个原始需求。不过，需要特别注意的是，在continue语句右边的标记必须代表一条闭合的for语句。也就是说，在这里的标记既不能代表在for语句之外的其他语句，也不能代表在for语句的代码块中的某条语句。

最后一个与for语句有关的编写技巧是关于for子句的。读者可以先想一想怎样使用Go语言的for语句写出反转一个切片类型值中的所有元素值的代码。一个附加的限制条件是，不允许使用在for语句之外声明的任何变量作为辅助。请读者思考一分钟。

好了，其实现代码如下：

```
// numbers 是一个[]int类型的变量，且其中已包含了若干元素

for i, j := 0, len(numbers)-1; i < j; i, j = i+1, j-1 {
    numbers[i], numbers[j] = numbers[j], numbers[i]
}
```

我们已经知道，在for子句中可以有初始化子句和后置子句。绝大多数的简单语句都可以充当初始化子句和后置子句。不过要注意，充当初始化子句和后置子句的只能是单一语句而不能是多个语句。但是，我们能够使用平行赋值的语句来丰富这两个子句的语义，就像上面展示的那样。想象一下，如果在初始化子句和后置子句中不允许出现平行赋值语句，那么我们又能怎样写出满足上述要求的实现代码呢？

至此，我们用了相当的篇幅介绍了Go语言的for语句的编写方法和技巧。for语句是Go语言中编写方法最多、最灵活的语句。它其中包含了很多个部分，也可以和很多其他语句组合使用。读者应该在阅读本小节中的示例的同时尝试使用for语句去解决各种各样的问题，并体会它的不同编写方法和组合用法之间的异同，这样才能真正地理解它。

4.1.5 goto语句

一条goto语句会把流程控制权无条件地转移到它右边的标记所代表的语句上。

1. 组成和编写方法

实际上，goto语句只能与标记语句连用，并且在它的右边必须要出现一个标记。

在我们理解了标记语句之后再来看goto语句，就会发现理解和使用它是非常简单的。但是，在goto的使用过程中有两个需要注意的地方。

第一，不允许因使用goto语句而使任何本不在当前作用域中的变量进入该作用域。这句话可能不太好理解。我们用下面的示例来说明。

```
    goto L
    v := "B"
L:
    fmt.Printf("V: %v\n", v)
```

在这个示例中，变量v实际上并不能够在标记L所指代的那条打印语句中被使用。因为语句goto L恰恰使变量v的声明语句被跳过了。因此，这段代码会造成一个编译错误。不过我们只需要稍加修改就可以使上面这段代码顺利通过编译。修改后的代码如下：

```
    v := "B"
    goto L
L:
    fmt.Printf("V: %v\n", v)
```

可以看到，我们只是将原本在语句goto L下面的那条语句移动到了goto L语句的上面。其根本思想是，让变量v的声明语句和使用它的代码处在相同的作用域中。当然，把变量v的声明语句移动到包含当前作用域的外层作用域中也是可以的。总之，当goto语句的执行致使某个或某些声明语句被跳过的时候，我们就要小心了。

第二，我们把某条goto语句的直属代码块叫作代码块A，而把该条goto语句右边的标记所指代的那条标记语句的直属代码块叫作代码块B。那么，只要代码块B不是代码块A的外层代码块，这条goto语句就是不合法的。示例如下：

```
// n是一个int类型的变量

if n%3 != 0 {
    goto L1
}
switch {
case n%7 == 0:
    fmt.Printf("%v is a common multiple of 7 and 3.\n", n)
default:
L1:
    fmt.Printf("%v isn't a multiple of 3.\n", n)
}
```

如上所示，标记L1所指代的标记语句的直属代码块是由switch语句代表的，而goto L1语句的直属代码块是由if语句代表的，并且前者并不是后者的直属代码块。因此，goto L1是非法的。我们编译这段代码的时候会得到一个编译错误。

要修正这个错误也并不难。代码如下：

```
    if n%3 != 0 {
        goto L1
    }
    switch {
    case n%7 == 0:
        n = 200
        fmt.Printf("%v is a common multiple of 7 and 3.\n", n)
    default:
    }
L1:
    fmt.Printf("%v isn't a multiple of 3.\n", n)
```

可以看到，我们只是把标记L1和它指代的那条语句移动到了switch语句的外边而已。但是，这样的一段代码是可以顺利通过编译的。原因就在于，这时的代码块B已经是代码块A的外层代码块了。

2. 更多惯用法

我们最常见到的一个使用场景是，利用goto语句跳出嵌套的流程控制语句的执行。这不仅限于我们在上一节涉及的嵌套for语句。因为goto语句几乎可以出现在任何Go语言代码块中，在这方面它与break语句和continue语句有很大不同。例如：

```
// 查找name中的第一个非法字符并返回。
// 如果返回的是空字符串就说明name中不包含任何非法字符。
func findEvildoer(name string) string {
    var evildoer string
    for _, r := range name {
        switch {
        case r >= '\u0041' && r <= '\u005a': // a-z
        case r >= '\u0061' && r <= '\u007a': // A-z
        case r >= '\u4e00' && r <= '\u9fbf': // 中文字符
        default:
            evildoer = string(r)
            goto L2
```

```
        }
    }
    goto L3
L2:
    fmt.Printf("The first evildoer of name '%s' is '%s'!\n", name, evildoer)
L3:
    return evildoer
}
```

如上所示，我们只允许变量name的值中出现大写或小写字母以及中文字符。如果碰到不符合要求的字符就立即停止对name的遍历，并在返回findEvildoer函数的结果值之前先打印出一条警告信息。当然，我们也可以换一种写法，使用break语句和if语句替换掉那两条goto语句，再调整一下标记的对象。修改后的代码如下：

```
func findEvildoer(name string) string {
    var evildoer string
L2:
    for _, r := range name {
        switch {
        case r >= '\u0041' && r <= '\u005a': // a-z
        case r >= '\u0061' && r <= '\u007a': // A-z
        case r >= '\u4e00' && r <= '\u9fbf': // 中文字符
        default:
            evildoer = string(r)
            break L2
        }
    }
    if evildoer != "" {
        fmt.Printf("The first evildoer of name '%s' is '%s'!\n", name, evildoer)
    }
    return evildoer
}
```

需要注意的是，上面示例中的break语句必须携带标记，否则它只会终止直接包含它的switch语句的执行，而外层的for语句的迭代依然会继续。这两个版本的findEvildoer函数所实现的语义是完全相同的。至于哪种方法更好就是仁者见仁智者见智了。

另一个比较适合使用goto语句的场景是集中式的错误处理，示例如下：

```
func checkValidity(name string) error {
    var errDetail string
    for i, r := range name {
        switch {
        case r >= '\u0041' && r <= '\u005a': // a-z
        case r >= '\u0061' && r <= '\u007a': // A-z
        case r >= '\u0030' && r <= '\u0039': // 0-9
        case r == '_' || r == '-' || r == '.': // 其他允许的符号
        default:
            errDetail = "The name contains some illegal characters。"
            goto L3
        }
        if i == 0 {
            switch r {
```

```
                case '_':
                    errDetail = "The name can not begin with a '_'."
                    goto L3
                case '-':
                    errDetail = "The name can not begin with a '-'."
                    goto L3
                case '.':
                    errDetail = "The name can not begin with a '.'."
                    goto L3
                }
            }
        }
        return nil
    L3:
        return errors.New("Validity check failure: " + errDetail)
    }
```

我们可以看到，只要发现了问题，流程控制权就会被跳转到checkValidity函数的最后一条语句上，无论检查出问题的代码处在for语句中的哪一行上。在这里，goto语句的优势同样在于可以非常方便地从错综复杂的流程控制语句中干脆地跳出。但是，它也存在一个劣势。这一劣势与标记语句有关。当存在多个相邻的标记语句时，除非使用额外的goto语句，否则我们就不能阻止这些标记语句被顺序地执行。请看下面的代码：

```
        fmt.Println("It always happens.")
    Error1:
        fmt.Println("Error1 occurred!")
    Error2:
        fmt.Println("Error2 occurred!")
```

在我们通过goto语句把流程控制权跳转到标记Error1所指代的语句之后，由于在默认情况下语句列表是会被顺序地执行的，所以标记Error2所指代的语句也会被执行。甚至，在不发生任何流程控制权跳转的情况下，标记Error1和Error2所指代的语句也会在第一条语句被执行后被相继地执行。除非我们加入一些额外的goto语句和标记作为辅助，像这样：

```
        fmt.Println("It always happens.")
        goto Post
    Error1:
        fmt.Println("Error1 occurred!")
        goto Post
    Error2:
        fmt.Println("Error2 occurred!")
    Post:
```

这显然严重影响了代码的清晰度，既增加了代码量，又对原有的代码造成了污染。

总之，虽然goto语句在某些场景下会为我们提供更多的便利，但是它却不像其他流程控制语句那样灵活。并且，充斥着goto语句的代码块的可读性会大大下降。所以，在很多时候，我们需要在便捷和简洁之间进行权衡，而后者往往会更占上风。我们需要有节制地使用goto语句，这样才能够在提高开发效率的同时降低开发维护的成本。

4.2 defer 语句

Go语言拥有一些特有的流程控制语句。其中最常用的就是defer语句。defer语句被用于预定对一个函数的调用。我们把这类被defer语句调用的函数称为延迟函数。注意，defer语句只能出现在函数或方法的内部。

一条defer语句总是以关键字defer开始。在defer的右边还必会有一条表达式语句，且它们之间要以空格“ ”分隔，就像这样：

```
defer fmt.Println("The finishing touches.")
```

这里的表达式语句必须代表一个函数或方法的调用。注意，既然是表达式语句，那么一些调用表达式就是不被允许出现在这里的。比如，针对各种内建函数的那些调用表达式。因为它们不能被称为表达式语句。另外，在这个位置上出现的表达式语句是不能被圆括号括起来的。

有意思的是，defer语句的执行时机总是在直接包含它的那个函数（以下简称外围函数）把流程控制权交还给它的调用方的前一刻，无论defer语句出现在外围函数的函数体中的哪一个位置上。具体分为下面几种情况。

- 当外围函数的函数体中的相应语句全部被正常执行完毕的时候，只有在该函数中的所有defer语句都被执行完毕之后该函数才会真正地结束执行。
- 当外围函数的函数体中的return语句被执行的时候，只有在该函数中的所有defer语句都被执行完毕之后该函数才会真正地返回。
- 当在外围函数中有运行时恐慌发生的时候，只有在该函数中的所有defer语句都被执行完毕之后该运行时恐慌才会真正地被扩散至该函数的调用方。

总之，外围函数的执行的结束会由于其中的defer语句的执行而被推迟。例如：

```
func isPositiveEvenNumber(number int) (result bool) {
    defer fmt.Println("done.")
    if number < 0 {
        panic(errors.New("The number is a negative number!"))
    }
    if number%2 == 0 {
        return true
    }
    return
}
```

在这个示例中，无论参数number是怎样的值，以及该函数的执行会以怎样的方式结束，在该函数的调用方重获流程控制权之前标准输出上都一定会出现done。

正因为defer语句有着这样的特性，所以它成为了执行释放资源或异常处理等收尾任务的首选。使用defer语句的优势有两个：一、收尾任务总会被执行，我们不会再因粗心大意而造成资源的浪费；二、我们可以把它们放到外围函数的函数体中的任何地方（一般是函数体开始处或紧跟在申请资源的语句的后面），而不是只能放在函数体的最后。这使得代码逻辑变得更加清晰，并且收尾任务是否被合理的指定也变得一目了然。

在defer语句中，我们调用的函数不但可以是已声明的命名函数，还可以是临时编写的匿名函数，就像这样：

```
defer func() {
        fmt.Println("The finishing touches.")
}()
```

注意，一个针对匿名函数的调用表达式是由一个函数字面量和一个代表了调用操作的一对圆括号组成的。一些刚刚学会编写defer语句的编程者常常会忘记添加后面的那对圆括号。

我们在这里选择匿名函数的好处是可以使该函数的收尾任务的内容更加直观。不过，我们也可以把比较通用的收尾任务单独放在一个命名函数中，然后再将其添加到需要它的defer语句中。无论在defer关键字右边的是命名函数还是匿名函数，我们都可以称之为延迟函数。因为它总是会被延迟到外围函数执行结束前一刻才被真正地调用。

每当defer语句被执行的时候，传递给延迟函数的参数都会以通常的方式被求值。请看下面的示例：

```
func begin(funcName string) string {
    fmt.Printf("Enter function %s.\n", funcName)
    return funcName
}

func end(funcName string) string {
    fmt.Printf("Exit function %s.\n", funcName)
    return funcName
}

func record() {
    defer end(begin("record"))
    fmt.Println("In function record.")
}
```

在我们对函数record进行调用之后，标准输出上会打印出如下内容：

```
Enter function record.
In function record.
Exit function record.
```

在这个示例中，调用表达式begin("record")是作为record函数的参数出现的。它会在defer语句被执行的时候被求值。也就是说，在record函数的函数体被执行之初，begin函数就被调用了。然而，end函数却是在外围函数record执行结束的前一刻被调用的。

这样做除了可以避免参数值在延迟函数被真正调用之前再次发生改变而给该函数的执行造成影响之外，还是出于同一条defer语句可能会被多次执行的考虑。请看下面的示例代码：

```
func printNumbers() {
    for i := 0; i < 5; i++ {
        defer fmt.Printf("%d ", i)
    }
}
```

在函数printNumbers真正执行结束之前，标准输出上会打印出这样的内容：4 3 2 1 0。这里

有两个细节需要特别说明。

第一个细节，在for语句的每次迭代的过程中都会执行一次其中的defer语句。在第一次迭代中，针对延迟函数的调用表达式最终会是fmt.Printf("%d ", 0)。这是由于在defer语句被执行的时候，参数i先被求值为了0，随后这个值被代入到了原来的调用表达式中，并形成了最终的延迟函数调用表达式。显然，这时的调用表达式已经与原来的表达式有所不同了。所以，Go语言会把代入参数值之后的调用表达式另行存储。以此类推，后面几次迭代所产生的延迟函数调用表达式依次为：

```
fmt.Printf("%d ", 1)
fmt.Printf("%d ", 2)
fmt.Printf("%d ", 3)
fmt.Printf("%d ", 4)
```

第二个细节是，对延迟函数调用表达式的求值顺序是与它们所在的defer语句被执行的顺序完全相反的。每当Go语言把已代入参数值的延迟函数调用表达式另行存储之后，还会把它追加到一个专门为当前外围函数存储延迟函数调用表达式的列表当中。而这个列表总是LIFO（Last In First Out，即后进先出）的。因此，这些延迟函数调用表达式的求值顺序会是：

```
fmt.Printf("%d ", 4)
fmt.Printf("%d ", 3)
fmt.Printf("%d ", 2)
fmt.Printf("%d ", 1)
fmt.Printf("%d ", 0)
```

依次对它们进行求值的结果即是我们在先前展示的结果。我们再来看一个例子：

```
func appendNumbers(ints []int) (result []int) {
    result = append(ints, 1)
    defer func() {
        result = append(result, 2)
    }()
    result = append(result, 3)
    defer func() {
        result = append(result, 4)
    }()
    result = append(result, 5)
    defer func() {
        result = append(result, 6)
    }()
    return result
}
```

如果我们对appendNumbers函数进行调用并以[]int{0}作为参数值，那么它的结果值总会是[]int{0, 1, 3, 5, 6, 4, 2}。这再次说明了多个延迟函数之间的执行顺序。读者可以试着按照我们刚刚讲到的两个细节对这个函数的执行过程进行分析，并验证上述结果值。

现在我们再来虑一个问题，如果我们把printNumbers函数的声明修改为：

```
func printNumbers() {
    for i := 0; i < 5; i++ {
        defer func() {
```

```
            fmt.Printf("%d ", i)
        }()
    }
}
```

那么，执行它又会使标准输出上出现什么样的内容呢？答案是：5 5 5 5 5。为什么会是这样呢？

我们说过，在defer语句被执行的时候传递给延迟函数的参数都会被求值，但是延迟函数调用表达式并不会在那时被求值。当我们把

```
defer fmt.Printf("%d ", i)
```

改为

```
defer func() {
    fmt.Printf("%d ", i)
}()
```

之后，虽然变量i依然是有效的，但是它所代表的值却已经完全不同了。让我们来简要地分析一下。在for语句的迭代过程中，其中defer语句被执行了5次。但是，由于我们并没有给延迟函数传递任何参数，所以Go语言运行时系统也就不需要对任何作为延迟函数的参数值的表达式进行求值（因为它们根本不存在）。在for语句被执行完毕的时候，共有5个延迟函数调用表达式被存储到了它们的专属列表中。注意，被存储在专属列表中的是5个相同的调用表达式：

```
func() {
    fmt.Printf("%d ", i)
}()
```

在printNumbers函数的执行即将结束的时候，那个专属列表中的延迟函数调用表达式就会被逆序地取出并被逐个地求值。然而，这时的变量i已经被修改为了5（请查看printNumbers函数中的那条for子句）。因此，对5个相同的调用表达式的求值都会使标准输出上打印出5。这也就得出了我们先前展示出的那个答案。

那么我们怎么才能修正这个问题呢？很简单，我们可以把printNumbers函数中的defer语句修改为

```
defer func(i int) {
    fmt.Printf("%d ", i)
}(i)
```

可以看到，我们虽然还是以匿名函数作为延迟函数，但是却为这个匿名函数添加了一个参数声明，并在代表调用操作的圆括号中加入了作为参数的变量i。这样，在defer语句被执行的时候，传递给延迟函数的这个参数i就会被求值。最终的延迟函数调用表达式也会类似于：

```
func(i int) {
    fmt.Printf("%d ", i)
}(0)
```

又因为延迟函数声明中的参数i屏蔽了在for语句中声明的变量i，所以在延迟函数被执行的时候，其中那条打印语句中所使用的i的值即为传递给延迟函数的那个参数值。

综上所述，最后这个版本的printNumbers函数的执行效果与第一个版本的printNumbers函数的执行效果是相同的。请读者对最后一个版本的printNumbers函数进行分析，并以求值顺序列出相应的延迟函数调用表达式。

最后，我们再来说说与延迟函数有关的另外一些小技巧。首先，如果延迟函数是一个匿名函数，并且在外围函数的声明中存在命名的结果声明，那么在延迟函数中的代码是可以对命名结果的值进行访问和修改的。请看下面的代码：

```
func modify(n int) (number int) {
    defer func() {
        number += n
    }()
    number++
    return
}
```

如果我们调用modify函数并传递给它的参数值为2，那么它的结果值总会是3。因为语句number++和number += n被先后地执行了。

其次，虽然在延迟函数的声明中可以包含结果声明，但是其返回的结果值会在它被执行完毕时被丢弃。因此，作为惯例，我们在编写延迟函数的声明的时候不会为其添加结果声明。另一方面，推荐以传参的方式提供延迟函数所需的外部值。作为总结，请看下面的示例：

```
func modify(n int) (number int) {
    defer func(plus int) (result int) {
        result = n + plus
        number += result
        return
    }(3)
    number++
    return
}
```

如果我们在调用modify函数的时候同样以2作为参数值，那么它的结果值总会是6。我们可以把想要传递给延迟函数的参数值依照规则放入到那个代表调用操作的圆括号中，就像调用普通函数那样。另一方面，虽然我们在延迟函数的函数体中返回了结果值，但是却不会产生任何效果。

好了，我们在本小节讲述的每一个知识点对于正确编写defer语句来说都是至关重要的。读者需要通过一定的练习才能够真正掌握defer语句的使用方法。由于本小节中的示例有限，所以请读者亲自动手去编写一些defer语句并进行相应的实例分析，这样才能真正记住和理解本小节所讲的内容。

4.3 异常处理

我们在本书前面的内容中已经涉及了一些Go语言的异常处理方面的内容，比如接口类型error、内建函数panic和标准库代码包errors。在本节，我们会对Go语言的各种异常处理方法进行系统的讲解，并试图一窥这些方法背后的内涵和哲学。

4.3.1 error

在编写Go语言代码的时候，我们应该习惯使用error类型值来表明非正常的状态。作为惯用法，在Go语言标准库代码包中的很多函数和方法也会以返回error类型值来表明错误状态及其详细信息。

我们之前说过，error是一个预定义标识符，它代表了一个Go语言内建的接口类型。这个接口类型的声明如下：

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

它非常地简单。其中的Error方法声明的意义就在于为方法调用方提供当前错误状态的详细信息。任何数据类型只要实现了这个可以返回string类型值的Error方法就可以成为一个error接口类型的实现。不过在通常情况下，我们并不需要自己去编写一个error的实现类型。Go语言的标准库代码包errors为我们提供了一个用于创建error类型值的函数New。该方法的声明如下：

```
func New(text string) error {
    return &errorString{text}
}
```

可以看到，errors.New函数接受一个string类型的参数值并可以返回一个error类型值。这个error类型值的动态类型就是errors.errorString类型。New函数的唯一参数被用于初始化那个errors.errorString类型的值。从代表这个实现类型的名称上可以看出，该类型是一个包级私有的类型。它只是errors包的内部实现的一部分，而非公开的API。errors.errorString类型及其方法的声明如下：

```
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

把errors.New函数、errors.errorString及其方法的声明联系起来看，我们就可以知道：传递给errors.New函数的参数值就是当我们调用它的Error方法的时候返回的那个结果值。

我们可以使用代码包fmt中的打印函数打印出error类型值所代表的错误的详细信息，就像这样：

```
var err error = errors.New("A normal error.")
fmt.Println(err) // 也可以是 fmt.Printf("%s\n", err) 等等。
```

这些打印函数在发现欲打印的内容是一个error类型值的时候都会调用该值的Error方法并将结果值作为该值的字符串表示形式。因此，我们传递给errors.New的参数值即是其返回的error类型值的字符串表示形式。

另一个可以生成error类型值的方法是调用fmt包中的Errorf函数。调用它的代码类似于：

```
err2 := fmt.Errorf("%s\n", "A normal error.")
```

与fmt.Printf函数相同，fmt.Errorf函数可以根据格式说明符和后续参数生成一个字符串类型值。但与fmt.Printf函数不同的是，fmt.Errorf函数并不会在标准输出上打印这个生成的字符串类型值，而是用它来初始化一个error类型值并作为该函数的结果值返回给调用方。在fmt.Errorf函数的内部，创建和初始化error类型值的操作正是通过调用errors.New函数来完成的。

在大多数情况下，errors.New函数和fmt.Errorf函数足以满足我们创建error类型值的要求。但是，接口类型error使得我们拥有了很大的扩展空间。我们可以根据需要定义自己的error类型。例如，我们可以使用额外的字段和方法让程序使用方能够获取更多的错误信息。例如，结构体类型os.PathError是一个error接口类型的实现类型。它的声明中包含了3个字段，这使得我们能够从它的Error方法的结果值当中获取到更多的信息。os.PathError类型及其方法的声明如下：

```
// PathError records an error and the operation and
// file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Err error    // Returned by the system call.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

从os.PathError类型的声明上我们可以获知，它的这3个字段都是公开的。因此，在任何位置上我们都可以直接通过选择符访问到它们。但是，在通常情况下，函数或方法中的相关结果声明的类型应该是error类型，而不应该是某一个error类型的实现类型。这也是为了遵循面向接口编程的原则。在这种情况下，我们常常需要先判定获取到的error类型值的动态类型，再依此来进行必要的类型转换和后续操作。例如：

```
file, err3 := os.Open("/etc/profile")
if err3 != nil {
    if pe, ok := err3.(*os.PathError); ok {
        fmt.Printf("Path Error: %s (op=%s, path=%s)\n", pe.Err, pe.Op, pe.Path)
    } else {
        fmt.Printf("Uknown Error: %s\n", err3)
    }
}
```

在这个示例中，我们通过类型断言表达式和if语句来对os.Open函数返回的error类型值进行处理。这与把error类型值作为结果值（之一）来表达函数执行的错误状态的做法一样，也属于Go语言中的异常处理的惯用法之一。

如果上面示例中的os.Open函数在执行过程中没有发生任何错误，那么我们就可以对变量file所代表的文件的内容进行读取了。相关代码如下：

```
r := bufio.NewReader(file)
var buf bytes.Buffer
```

```
for {
    byteArray, _, err4 := r.ReadLine()
    if err4 != nil {
        if err4 == io.EOF {
            break
        } else {
            fmt.Printf("Read Error: %s\n", err4)
            break
        }
    } else {
        buf.Write(byteArray)
    }
}
```

在这段代码中，我们使用到了几个之前没有遇到过的标准库代码包，它们是bufio、bytes和io。我们利用bufio.NewReader函数来创建一个可以读取文件内容的读取器，并利用bytes.Buffer类型的值来缓存从文件读取出来的内容。请读者注意示例中使用的error类型的变量io.EOF。在标准库代码包io中，它的声明如下：

```
var EOF = errors.New("EOF")
```

可以看到，io.EOF变量正是由errors.New函数的结果值来初始化的。EOF是文件结束符（End Of File）的缩写。对于文件读取操作来说，它意味着读取器已经读到了文件的末尾。因此，严格来说，EOF并不应该算作一个真正的错误，而仅仅属于一种“错误信号”。

变量r代表了一个读取器。它的ReadLine方法返回3个结果值。第三个结果值的类型就是error类型的。当读取器读到file所代表的文件的末尾时，ReadLine方法会直接将变量io.EOF的值作为它的第三个结果值返回。因此，我们可以很方便地通过比较操作符==来判断第三个结果值是否就是io.EOF变量的值。如果判断的结果为true，那么我们就可以直接终止那个被用于连续读取文件内容的for语句的执行。否则，我们就应该意识到在读取文件内容的过程中有真正的错误发生了，并采取相应的措施。

注意，只有当两个error类型的变量的值确实为同一个值的时候，使用比较操作符==进行判断时才会得到true。从另一个角度看，我们可以预先声明一些error类型的变量，并把它们作为特殊的“错误信号”来使用。任何需要返回同一类“错误信号”的函数或方法都可以直接把这类预先声明的变量的值拿来使用。这样我们就可以很便捷地使用==来识别这些“错误信号”并进行相应的操作了。

不过，需要注意的是，这类变量的值必须都是不可变的。也就是说，它们的实际类型的声明中不应该包含任何公开的字段，并且附属于这些类型的方法也不应该包含对其字段进行赋值的语句。例如，我们前面提到的os.PathError类型就不适合作为这类变量的值的动态类型，否则很可能会造成不可预知的后果。

这种通过预先声明error类型的变量为程序使用方提供便利的做法在Go语言标准库代码包中非常常见。除了我们刚刚讲的io.EOF，在诸如compress/gzip、crypto/dsa、bufio、bytes、database/sql、encoding/binary、fmt/scan等代码包中都包括这样的变量。

关于实现error接口类型的另一个技巧是，我们还可以通过把error接口类型嵌入到新的接口

类型中来对它进行扩展。例如，标准库代码包net中的Error接口类型，其声明如下：

```
// An Error represents a network error.
type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

一些在net包中声明的函数会返回动态类型为net.Error的error类型值。在使用方，对这种error类型值的动态类型的判定方法与前面提及的基本一致。

如果变量err的动态类型是net.Error，那么就我们可以根据它的Temporary方法的结果值来判断当前的错误状态是否临时的：

```
if netErr, ok := err.(net.Error); ok && netErr.Temporary() {
    // 省略若干条语句
}
```

如果是临时的，那么就可以间隔一段时间之后再进行对之前的操作进行重试，否则就记录错误状态的信息并退出。假如我们没有对这个error类型值进行类型断言，也就无法获取到当前错误状态的那个额外属性，更无法决定是否应该进行重试操作了。这种对error类型的无缝扩展方式所带来的益处是显而易见的。

在Go语言中，对错误的正确处理是非常重要的。语言本身的设计和标准库代码中展示的惯用法鼓励我们对发生的错误进行显式地检查。虽然这会使Go语言代码看起来稍显冗长，但是我们可以使用一些技巧来简化它们。这些技巧大都与通用的编程最佳实践大同小异，或者已经或将要包含在我们所讲的内容（自定义错误类型、使用卫述语句、单一职责函数等）中，所以这并不是问题。况且，这一点点代价比传统的try-catch方式带来的弊端要小得多。

4.3.2 panic和recover

在通常情况下，向程序使用方报告错误状态的方式可以是返回一个额外的error类型值。但是，当遇到不可恢复的错误状态的时候，很可能会导致程序无法继续运行。这时，上述错误处理方式显然就不适合了。反过来讲，在一般情况下，我们不应通过调用panic函数来报告普通的错误，而应该只把它作为报告致命错误的一种方式。

1. panic

为了使编程人员能够在自己的程序中报告运行期间的、不可恢复的错误状态，Go语言内建了一个专用函数——panic。我们在讲内建函数的时候已经提到过它。panic函数被用于停止当前的控制流程的执行并报告一个运行时恐慌。它可以接受一个任意类型的参数值。然而这个参数常常是一个string类型值或者error类型值，因为这样可以更容易地描述运行时恐慌的详细信息。请看下面的例子：

```
func main() {
    outerFunc()
}
```

```
func outerFunc() {
    innerFunc()
}

func innerFunc() {
    panic(errors.New("A intended fatal error!"))
}
```

当在函数innerFunc中调用了panic函数之后，函数innerFunc的执行会被停止。然后，流程控制权会被交回给函数innerFunc的调用方outerFunc函数。然而，outerFunc函数的执行也将被停止，就像在其中调用innerFunc函数的位置上调用了panic函数一样。运行时恐慌就这样沿着调用栈反方向进行传达，直至到达当前Goroutine（也被称为Go程，可以看作是一个能够独占一个系统线程并在其中运行程序的独立环境）调用栈的最顶层。这时，当前Goroutine的调用栈中的所有函数的执行都已经被停止了。这也意味着程序已经崩溃。

当然，运行时恐慌并不都是通过调用panic函数的方式引发的。它也可以由Go语言的运行时系统来引发。例如：

```
myIndex := 4
ia := [3]int{1, 2, 3}
_ = ia[myIndex]
```

这个示例中的第三行代码会引发一个运行时恐慌，因为它造成了一个数组访问越界的运行时错误。这个运行时恐慌就是由运行时系统报告的。它相当于我们显式地调用panic函数并传入一个runtime.Error类型的参数值。runtime.Error类型的声明如下：

```
type Error interface {
    error

    // RuntimeError is a no-op function but
    // serves to distinguish types that are runtime
    // errors from ordinary errors: a type is a
    // runtime error if it has a RuntimeError method.
    RuntimeError()
}
```

可以看到，接口类型runtime.Error将error接口类型嵌入其中，并添加了RuntimeError方法的声明。显然，runtime.Error类型是error接口类型的一个扩展，而RuntimeError方法声明则只是作为runtime.Error类型一个标志存在的。我们可以通过在上一小节讲到的惯用法来判定运行时恐慌中携带的error类型值的动态类型是否是RuntimeError类型。不过，这里有一个问题：我们怎样“拦截”一个运行时恐慌并取出其中携带的值呢？

2. recover

运行时恐慌一旦被引发就会向调用方传递直至程序崩溃。这当然不是我们愿意看到的，因为谁也不能保证程序不会发生任何运行时错误。不过，不要担心，Go语言为我们提供了专用于“拦截”运行时恐慌的内建函数——recover。它可以使当前的程序从运行时恐慌的状态中恢复并重新获得流程控制权。recover函数有一个interface{}类型的结果值。如果当前的程序正处于运行

时恐慌的状态下，那么调用recover函数将会让我们得到一个非nil的interface{}类型值。如果当时的运行时恐慌是由Go语言的运行时程序引发的，那么我们就会获得一个runtime.Error类型的值。

不过，单靠这个内建函数并不足以“拦截”运行时恐慌。因为运行时恐慌让当前程序失去了流程控制权，我们无法让一段代码在运行时恐慌被引发之后执行。但是这有一个例外，defer语句中的延迟函数总会被执行，不论它的外围函数是以怎样的方式被终止执行的。所以，我们还需要将recover函数与defer语句配合起来使用。更确切地说，只有在defer语句的延迟函数中调用recover函数才能够真正起到“拦截”运行时恐慌的作用。按照惯例，我们在函数或方法中使用它们的方式应该形如：

```
defer func() {
    if r := recover(); r != nil {
        fmt.Printf("Recovered panic: %s\n", r)
    }
}()
```

我们现在编写一个更复杂一些的示例，以使大家能够更加深刻地理解与panic函数、recover函数和defer语句有关的运行机制。我们把这些示例代码组织成了一个命令源码文件。它的完整代码如下：

```
package main

// 省略导入语句

func main()
    fetchDemo()
    fmt.Println("The main function is executed.")
}

func fetchDemo() {
    defer func() {
        if v := recover(); v != nil {
            fmt.Printf("Recovered a panic. [index=%d]\n", v)
        }
    }()
    ss := []string{"A", "B", "C"}
    fmt.Printf("Fetch the elements in %v one by one...\n", ss)
    fetchElement(ss, 0)
    fmt.Println("The elements fetching is done.")
}

func fetchElement(ss []string, index int) (element string) {
    if index >= len(ss) {
        fmt.Printf("Occur a panic! [index=%d]\n", index)
        panic(index)
    }
    fmt.Printf("Fetching the element... [index=%d]\n", index)
    element = ss[index]
    defer fmt.Printf("The element is \"%s\". [index=%d]\n", element, index)
```

```
    fetchElement(ss, index+1)
    return
}
```

在这个示例中，我们通过向标准输出打印不同内容的方式来体现程序在运行过程中的执行流程。在运行这个命令源码文件之后，标准输出上会出现如下内容：

```
1:  Fetch the elements in [A B C] one by one...
2:  Fetching the element... [index=0]
3:  Fetching the element... [index=1]
4:  Fetching the element... [index=2]
5:  Occur a panic! [index=3]
6:  The element is "C". [index=2]
7:  The element is "B". [index=1]
8:  The element is "A". [index=0]
9:  Recovered a panic. [index=3]
10: The main function is executed.
```

为了查看方便，我为每行打印内容都加入了行号。表示行号的数字均在每行的最左边，且与真正的打印内容之间用冒号“:”和若干空格分隔。现在，我们来解释一下上面的输出内容。main函数中的代码调用了fetchDemo函数。在fetchDemo函数中的代码调用fetchElement函数之前，第1行内容被打印出来了。由于在fetchElement函数中存在递归调用（fetchElement函数在其代码块的最后调用了自身），所以接下来的第2、3、4行的内容都是由于函数调用语句

```
fmt.Printf("Fetching the element... [index=%d]\n", index)
```

的执行而被打印出来的。

函数fetchElement中的递归调用使得延迟函数一直没有被执行的机会。还记得吗？defer语句中的延迟函数仅会在其外围函数的执行将要结束的时候才会被执行。这种情况直到在fetchElement函数被第四次调用的时候才有所转变。在fetchElement函数被第四次调用的时候，传递给它的第二个参数值大于了第一个参数的最大索引值，这时我们通过调用panic函数并传递给它当前的索引值引发了一个运行时恐慌。这时，调用语句

```
fmt.Printf("Occur a panic! [index=%d]\n", index)
```

已经使第5行的内容被打印到了标准输出上。在运行时恐慌发生后，它被沿着调用栈逐一地向顶层传达。这使fetchElement函数中的延迟函数调用语句得以执行，以至于第6、7、8行内容被陆续打印出来。当运行时恐慌已经被传递到fetchDemo函数中且正要向它的调用方继续传递的时候，被fetchDemo函数中的那个延迟函数中的代码“拦截”了。我们再来看一下这个延迟函数的代码：

```
defer func() {
    if v := recover(); v != nil {
        fmt.Printf("Recovered a panic. [index=%d]\n", v)
    }
}()
```

显然，运行时恐慌能够被“拦截”的原因是在该延迟函数中的那个针对recover函数的调用表达式。如果调用recover函数后得到的结果值为nil（参见3.3.5节中对此种情况的说明）就什么

都不做。但是在这里，这个结果值就是在fetchElement函数中调用panic函数时传入的那个参数值，即触发运行时恐慌的那个越界的索引值3。因此，也就是有了第9行打印内容。注意，在fetchDemo函数中的最后那条打印语句

```
fmt.Println("The elements fetching is done.")
```

永远没有机会被执行，因为在它上一行的针对fetchElement函数的调用语句在被执行的过程中总是会发生运行时恐慌。

最后，由于运行时恐慌在将要被继续传递给fetchDemo函数的调用方的时候被“拦截”（或者说被“平息”）了，因此fetchDemo函数的调用方（也就是main函数）得以重获流程控制权。所以，在main函数中的调用fetchDemo函数的语句下面的打印语句

```
fmt.Println("The main function is executed.")
```

是会被执行的。这也就是会有第10行打印内容的原因。

好了，通过上面的这个较大的示例和对它的详细讲解，相信读者已经对运行时恐慌的报告和处理机制有了更进一步的认识。

值得一提的是，在Go语言标准库中可以经常看到的一类惯用法值得我们在编写程序时参考，那就是，即使在我们使用的某个程序实体的内部发生了运行时恐慌，这个运行时恐慌也会在被传递给我们编写的程序使用方之前被“平息”并以error类型值的形式返回给使用方。

另外，在这些标准库代码包中，往往都会有自己的error接口类型的实现。只有当调用recover函数得到的结果值的类型是它们自定义的error类型的实现类型的时候，才会去处理这个运行时恐慌。否则就会重新引发（官方使用的词汇是re-panic）一个运行时恐慌并携带相同的值。

例如，在标准库代码包fmt中的Token函数就是这样处理运行时恐慌的。它的声明如下：

```
func (s *ss) Token(skipSpace bool, f func(rune) bool) (tok []byte, err error) {
    defer func() {
        if e := recover(); e != nil {
            if se, ok := e.(scanError); ok {
                err = se.err
            } else {
                panic(e)
            }
        }
    }()
    // 省略若干条语句
}
```

在Token函数包含的延迟函数中，当运行时恐慌携带的值的类型是fmt.scanError类型的时候，这个值就会被赋值给代表结果值的变量err，否则运行时恐慌就会被重新引发。如果这个被重新引发的运行时恐慌被传递到了调用栈的最顶层，那么标准输出上就会打印出类似这样的内容：

```
panic: <运行时恐慌被首次引发时携带的值的字符串形式> [recovered]
    panic: <运行时恐慌被重新引发时携带的值的字符串形式>

goroutine 1 [running]:
main.func·001()
```

```
<调用栈信息>

goroutine 2 [runnable]:
exit status 2
```

由于篇幅有限，我们省略了绝大部分调用栈信息。此外，我们还使用被尖括号“<”和“>”括起来的辅助描述来说明一些会根据实际情况变化的内容。我们可以看到，在上面这段打印内容的第一行的最右边包含了内容“[recovered]”。这意味着在运行时恐慌被首次引发之后又被“平息”了。但是，第二行内容表示此运行时恐慌又被重新引发了。因此，在下面的调用栈信息中，不但会包含与该运行时恐慌被首次引发时的调用轨迹和引发位置，还会包含该运行时恐慌被重新引发时的具体位置。

无论我们对一个运行时恐慌重新引发几次，它所有的引发信息都依然会被提供在最终的程序崩溃报告中，就像前面描述的那样。更明确地讲，该运行时恐慌被引发的根本原因永远不会丢失。所以，我们在重新引发一个运行时恐慌的时候使用最简单的方式就足够了，像这样：

```
panic(e)
```

也就是说，我们一般并不需要再为调用panic函数而另外创建一个新的参数值。

我们在编写自己的程序的时候可以使用上面介绍的这些惯用法。但是，我们应该在使用这种方案之前明确和统一可以被立即处理和需要被重新引发的运行时恐慌的种类。一般情况下，如果携带的值是动态类型为runtime.Error的error类型值的话，这个运行时恐慌就应该被重新引发。另外，从运行时恐慌的分类和处理决策角度看，我们在必要时自行定义一些error类型的实现类型是很有好处的。

综上所述，对于运行时恐慌的引发，我们应该持谨慎态度。更确切地说，我们应该仅在遇到致命的、不可恢复的错误状态时才去引发一个运行时恐慌。否则，我们完全可以利用函数或方法的结果值来向程序使用方传达错误状态。另一方面，我们应该仅在处于程序模块的边界位置上的函数或方法中对运行时恐慌进行“拦截”和“平息”。在运行时恐慌的处理方式上，我们可以大致遵循这样的流程：“拦截”、判定运行时恐慌的种类、根据相关决策处理（“平息”、记录日志或重新引发，等等）。

总之，对运行时恐慌的合理运用是优秀代码和程序的必备条件之一。这涉及panic函数、defer语句和recover函数。希望本小节的内容能够让读者更好地使用它们。

4.4　实战演练——Set

从本节开始，我们就要运用之前了解到的Go语言基础知识来实际开发一些高级数据结构。这些数据结构都是Go语言本身及其标准库中没有涉及的。

在很多编程语言中，集合（Set）的底层都是由哈希表（Hash Table）来实现的。比如，C++语言的代码库STL中的数据结构hash_set、Java语言的标准库中的java.util.HashSet类，以及Python语句的标准数据结构set，等等。

在Go语言的标准数据类型中并没有集合这种数据类型。但是，它却拥有作为Hash Table实现

的字典（Map）类型。我们在对Set和Map进行比较之后会发现它们在一些主要特性上是极其相似的，如下所示。

- 它们中的元素都是不可重复的。
- 它们都只能用迭代的方式取出其中的所有元素。
- 对它们中的元素进行迭代的顺序都是与元素插入顺序无关的，同时也不保证任何有序性。

但是，它们之间也有一些区别。

- Set的元素是一个单一的值，而Map的元素则是一个键值对。
- Set的元素不可重复指的是不能存在任意两个单一值相等的情况。Map的元素不可重复指的是任意两个键值对中的键的值不能相等。

仔细看过上面罗列的这些异同点之后，我们会发现Set更像是Map的一种简化版本。我们可不可以利用Map来编写一个Set的实现呢？答案当然是肯定的。实际上，在Java语言中，java.util.HashSet类就是用java.util.HashMap类作为底层支持的。java.util.HashSet相当于是java.util.HashMap类的一个代理类。

1. 基本定义

首先，我们创建一个名为hash_set.go的源码文件，并把它放在goc2p项目的代码包basic/set中。我们需要首先在这个源码文件的第一行上写入这样一行代码：

```
package set
```

这是为了声明源码文件hash_set.go是代码包basic/set中的一员。我们刚才说过，可以把集合类型作为字典类型的一个简化版本。那么我们就声明一个其中包含了一个字典类型的字段的结构体类型。它的声明如下：

```
type HashSet struct {
    m map[interface{}]bool
}
```

这个类型声明中的唯一的字段的类型是map[interface{}]bool。之所以选择这样的一个字典类型是有原因的。因为我们希望HashSet类型的元素可以是任何类型的，所以我们将字典m的键类型设置为了interface{}。又由于我们只需要用到m的值中的键来存储HashSet类型的元素值，所以就应该选用值占用空间最小的类型来作为m的值的元素类型。这里使用bool类型有3个好处。

- 从值的存储形式的角度看，bool类型值的占用空间是最小的（之一），只占用一个字节。
- 从值的表示形式的角度看，bool类型的值只有两个——true和false。并且，这两个值都是预定义常量。
- 把bool类型作为值类型更有利于判断字典类型值中是否存在某个键。例如，如果我们在向m的值添加键值对的时候总是以true作为其中的元素的值，那么索引表达式

  ```
  m["a"]
  ```

  的结果值就总能够直接体现出在m的值中是否包含键为"a"的键值对。但是，如果m的类型是map[interface{}]byte的话，那么我们只有通过

  ```
  v, ok := m["a"]
  ```

才能确切地得出上述判断的结果。虽然在向map[interface{}]byte类型的m的值添加键值对的时候，我们可以总以非零值的byte类型值作为其中的元素的值，但是我们在做判断的时候依然需要编写更多的代码：

```
if v := m["a"]; v != 0 { // 如果“m”中不存在以“a”作为键的键值对
    // 省略若干条语句
}
```

而对于map[interface{}]bool类型的m的值来说，如此即可：

```
if m["a"] { // 如果“m”中不存在以“a”作为键的键值对
    // 省略若干条语句
}
```

现在，HashSet类型的基本结构已经被确定。我们下面需要考虑初始化HashSet类型值的问题了。由于字典类型值的零值为nil，所以我们不能简单地使用new函数来创建一个HashSet类型值。换句话说，与HashSet类型声明处在同一个代码包中的表达式

```
new(HashSet).m
```

的求值结果会是nil。因此，我们需要编写一个专门用于创建和初始化HashSet类型值的函数。这个函数的声明如下：

```
func NewHashSet() *HashSet {
    return &HashSet{m: make(map[interface{}]bool)}
}
```

可以看到，我们使用make函数对字段m进行了初始化。注意，函数NewHashSet的结果声明的类型是*HashSet而不是HashSet。这是因为，我们希望在这个结果值的方法集合中包含调用接收者类型为HashSet或*HashSet的所有方法。至于这么做的好处，我们在后面编写Set接口类型的时候再予以说明。

2. 基本功能

现在，我们就需要为HashSet类型编写方法了。不过，在这之前我们先需要明确一下它都需要提供哪些功能。HashSet类型应该提供的基本功能如下。

- ❑ 添加元素值。
- ❑ 删除元素值。
- ❑ 清除所有元素值。
- ❑ 判断是否包含某个元素值。
- ❑ 获取元素值的数量。
- ❑ 判断与其他HashSet类型值是否相同。
- ❑ 获取所有元素值，即生成可迭代的快照。
- ❑ 获取自身的字符串表示形式。

上述功能中的绝大部分都是在其他编程语言的Set类型上已经提供的功能。作为一个可用和好用的Set类型，我们当然需要它们。

首先需要编写的是向HashSet类型值中添加元素值的方法，其声明如下：

```
func (set *HashSet) Add(e interface{}) bool {
    if !set.m[e] {
        set.m[e] = true
        return true
    }
    return false
}
```

方法Add会返回一个bool类型的结果值，以表示添加元素值的操作是否成功。如果当前的m的值中还未包含以e的值为键的键值对，那么就将键为e（代表的值）、元素为true的键值对添加到m的值当中并返回true。否则，就直接返回false。

在这里需要注意的是，Add方法的声明中的接收者类型是*HashSet。这里将其类型设置为*HashSet而不是HashSet，主要原因是减少复制接收者值时对系统能够资源的耗费。我们在上一章中说过，方法的接收者值只是当前值的一个复制品。所以，当Add方法的接收者的类型为HashSet的时候，对它的每一次调用都需要对当前值（当前的HashSet类型值）进行一次复制。虽然，在HashSet类型中只有一个引用类型的字段，但是这终归是一种开销。并且，我们还未考虑HashSet类型中的字段可能会变得更多的情况。当Add方法的接收者的类型为*HashSet的时候，对它进行调用时复制的当前值（当前的*HashSet类型值）只是一个指针值。在大多数情况下，一个指针值占用的内存空间总会比它指向的那个其他类型的值所占用的内存空间小。指针值所占用的内存空间的大小与且只与当前计算机的计算架构中的字长（32比特或64比特）相对应。也就是说，无论一个指针值指向的那个其他类型值所需的内存空间有多么大，它所占用的内存空间总是不变的。因此，从节约内存空间的角度出发，建议尽量将方法的接收者类型设置为相应的指针类型。关于指针的更多知识请参见3.2.8节。

从HashSet类型值中删除元素值的操作是非常简单的。因为我们是用字典值作为HashSet类型的内部支持的，所以我们调用delete函数就可以达到删除元素值的目的。删除元素值的方法的声明如下：

```
func (set *HashSet) Remove(e interface{}) {
    delete(set.m, e)
}
```

编写实现清除所有元素值功能的方法会用到一个小技巧。由于Go语言本身并没有提供可以清除字典值中的所有键值对的方法和内建函数，所以我们需要自己编码完成这一功能。迭代出其中的所有键值对并逐一删除它们当然是不可取的。这样做可能会在并发访问和修改的情况下引发问题，并且不一定总能把所有的键值对都删除掉。最干脆和简洁的方法就是为字段m重新赋值。依此实现的Clear方法如下：

```
func (set *HashSet) Clear() {
    set.m = make(map[interface{}]bool)
}
```

对字段m赋值的效果的达成也得益于Clear方法的接收者类型*HashSet。如果接收者类型是HashSet，那么该方法中的这条赋值语句的作用只是为当前值的某个复制品中的字段m赋值而已，

而当前值中的字段m则不会被重新赋值。

方法Clear中的这条赋值语句被执行之后，当前的HashSet类型值中的元素就相当于被清空了，就像刚刚被初始化过的值一样。已经与字段m解除绑定的那个旧的字典值由于不再与任何程序实体存在绑定关系而成为了无用的数据。它会在之后的某一时刻被Go语言的垃圾回收器发现并回收。

附属于HashSet类型的Contains方法用于判断其值是否包含某个元素值。它同样只包含一条语句。这也是得益于元素类型为bool的字段m。其声明如下：

```
func (set *HashSet) Contains(e interface{}) bool {
    return set.m[e]
}
```

读者可能会有疑问，Go语言是怎样生成interface{}类型值的hash值的？对于一个interface{}类型值来说，Go语言总能正确地判断出在一个字典值中是否包含与之相对应的键吗？我通过查看Go语言的源代码获知，当我们把一个interface{}类型值作为键添加到一个字典值的时候，Go语言会先获取这个interface{}类型值的实际类型（即动态类型），然后再使用与之相对应的hash函数对该值进行hash运算。所以，interface{}类型值总是能够被正确地计算出hash值。显然，在之后的键查找的过程中也会存在这样的hash运算。但是，请注意，我们在上一章讲字典类型的时候说过，字典类型的键不能是函数类型、字典类型或切片类型。这种限制总是存在的。因此，Contains方法的参数e的值的动态类型一定不能是上面这几种类型，否则就会引发一个运行时恐慌并有如下提示：

```
panic: runtime error: hash of unhashable type <某个函数类型、字典类型或切片类型的名称>
```

现在我们继续编码。与Remove方法类似，被用于获取元素值数量的方法Len也是利用Go语言的内建函数来完成功能的：

```
func (set *HashSet) Len() int {
    return len(set.m)
}
```

合理利用Go语言的内建函数是我们编写Go语言代码的最基本的要求。Go语言内建函数的汇总请参见3.3.5节。

下面是对另一个方法的考虑。两个HashSet类型值相同的必要条件是，它们包含的元素值应该是完全相同的。由于HashSet类型值中的元素的迭代顺序总是不确定的，所以我们也就不用在意两个值在这方面是否一致。因此，刚才所说的那个必要条件也就成为了唯一的充分条件。下面的Same方法用来判断两个HashSet类型值是否相同：

```
func (set *HashSet) Same(other *HashSet) bool {
    if other == nil {
        return false
    }
    if set.Len() != other.Len() {
        return false
    }
```

```
        for key := range set.m {
            if !other.Contains(key) {
                return false
            }
        }
        return true
    }
```

虽然Same方法中的语句稍微多了一些，但是其中的逻辑依然是非常简单和清晰的。我们利用了之前声明的Len方法和Contains方法完成了Same方法中最核心的逻辑。在大多数情况下，这种相同性判断就已经足够了。如果要判断两个HashSet类型值是否是同一个值，就需要利用指针运算进行内存地址的比较。不过我们在这里并不需要这种判断方式。

我们刚刚讲过，HashSet类型值的元素迭代顺序的不确定性。这种不确定性会使我们无法通过索引值获取某一个元素值。并且，我们也已经知道for语句和range子句只能够对数组类型、切片类型、字典类型和通道类型的值起作用。那么我们怎样对一个HashSet类型值进行迭代呢？或者说，我们怎样取出其中的值呢？一个简单可行的解决方案就是先生成一个它的快照，然后再在这个快照之上进行迭代操作。所谓快照，就是目标值在某一个时刻的映像。对于一个HashSet类型值来说，它的快照中的元素迭代顺序是总是可以确定的，这正是由于快照只反映了该HashSet类型值在某一个时刻的状态。另外，我们还需要从元素可迭代且顺序可确定的数据类型中选取一个作为快照的类型。这个类型必须是以单值作为元素的，所以字典类型最先被排除。又由于HashSet类型值中的元素数量总是不固定的，所以也就无法用一个数组类型的值来表示它的快照。因此，快照的类型应该是一个切片类型或者通道类型。我们这里以切片类型为例。

我们为这个被用于生成快照的方法起了一个比较通用的名字——Elements。我们根据前面对Elements方法的描述和分析编写出了它的声明：

```
func (set *HashSet) Elements() []interface{} {
    initialLen := len(set.m)
    snapshot := make([]interface{}, initialLen)
    actualLen := 0
    for key := range set.m {
        if actualLen < initialLen {
            snapshot[actualLen] = key
        } else {
            snapshot = append(snapshot, key)
        }
        actualLen++
    }
    if actualLen < initialLen {
        snapshot = snapshot[:actualLen]
    }
    return snapshot
}
```

之所以我们使用这么多条语句来实现这个方法是因为需要考虑到在从获取字段m的值的长度到对m的值迭代完成的这个时间段内，m的值中的元素数量可能会发生变化。

我们每次调用append函数的时候，都会有一个新的切片值创建出来，并且有时候还会导致新的切片值的底层数组被替换。显然，这会降低生成快照的效率。因此，我们先获取字段m的值的长度，并以此初始化一个[]interface{}类型的变量snapshot来存储m的值中的元素值。在正常情况下，我们仅仅把迭代值按照既定顺序设置到快照值（变量snapshot的值）的指定元素位置上即可。这一过程并不会创建任何新值。如果在迭代完成之前，m的值中的元素数量有所增加，致使实际迭代的次数大于先前初始化的快照值的长度，那么我们再使用append函数向快照值追加元素值。这样做既提高了快照生成的效率，又不至于在元素数量增加时引发索引越界的运行时恐慌。

对于已被初始化的[]interface{}类型的切片值来说，未被显式初始化的元素位置上的值均为nil。如果在迭代完成之前，m的值中的元素数量有所减少，致使快照值的尾部存在若干个没有任何意义的值为nil的元素，那么我们就应该把这些无用的元素值从快照值中去掉。我们使用切片表达式和赋值语句snapshot = snapshot[:actualLen]达到了这一目的。

注意，虽然我们在Elements方法中针对并发访问和修改m的值的情况采取了一些措施。但是由于m的值本身不是并发安全的，所以我们并不能保证Elements方法的执行总会准确无误。要做到真正的并发安全，还需要一些辅助的手段，比如使用在上一章讲字典类型时提到的读写互斥量。

现在我们来编写最后一个提供基本功能的方法。它的功能是获取自身的字符串表示形式。这个方法的声明如下：

```
func (set *HashSet) String() string {
    var buf bytes.Buffer
    buf.WriteString("Set{")
    first := true
    for key := range set.m {
        if first {
            first = false
        } else {
            buf.WriteString(" ")
        }
        buf.WriteString(fmt.Sprintf("%v", key))
    }
    buf.WriteString("}")
    return buf.String()
}
```

这个String方法的签名也算是一个惯用法。代码包fmt中的打印函数总会使用参数值附带的具有如此签名的String方法的结果值作为该参数值的字符串表示形式。当然，前提是那个数据类型声明了这个名为String的方法。所以，如果我们想让自定义类型值的字符串表示形式有更好的可读性，就需要声明这样的一个方法。顺便说一句，在String方法包含的语句列表中，我们使用bytes.Buffer类型值作为结果值的缓冲区，这样可以避免因string类型值的拼接造成的内存空间上的浪费。

至此，我们完整地编写了一个具备常用功能的Set的实现类型。但是，在很多时候，我们需要提供更多的功能来降低客户端代码使用它的成本。

3. 高级功能

在集合代数中有对集合的基本性质和规律的描述，其中包含了对各种集合运算和集合关系的说明。集合的运算包括并集、交集、差集和对称差集。集合的关系包括等于（也就是相同）和真包含。我们在前面编写的Same方法已经实现了对集合相同性的判断。下面，我们关注其余的关系判断功能和运算。

首先，我们来实现集合真包含的判断功能。从名称上看，它与Contains方法在逻辑上有些类似，不过它会更复杂一些。为了不与Contains方法在名称上过于类似，我们需要为这个方法另起一个名字。根据集合代数中的描述，如果集合A真包含了集合B，那么就可以说集合A是集合B的一个超集。因此，我们给这个方法的名称确定为IsSuperset。对于会返回一个bool类型的结果值的方法来说，用以“Is”为开头的动宾短语作为它的名称是非常适合的。IsSuperset方法的声明如下：

```
func (set *HashSet) IsSuperset(other *HashSet) bool {
    if other == nil {
        return false
    }
    oneLen := one.Len()
    otherLen := other.Len()
    if oneLen == 0 || oneLen == otherLen {
        return false
    }
    if oneLen > 0 && otherLen == 0 {
        return true
    }
    for _, v := range other.Elements() {
        if !one.Contains(v) {
            return false
        }
    }
    return true
}
```

只要我们理解了真包含的含义，实现IsSuperset方法并不难。因为我们已经把实现基本功能的方法都编写完成了，在这里只要对它们进行组合使用即可。

现在我们来看集合运算。我们先来了解一下这些集合运算的含义。

- 并集运算是指把两个集合中的所有元素都合并起来并组成一个集合。
- 交集运算是指找到两个集合中共有的元素并把它们组成一个集合。
- 集合A对集合B进行差集运算的含义是找到只存在于集合A中但不存在于集合B中的元素并把它们组成一个集合。
- 对称差集运算与差集运算类似但有所区别。对称差集运算是指找到只存在于集合A中但不存在于集合B中的元素，再找到只存在于集合B中但不存在于集合A中的元素，最后把它们合并起来并组成一个集合。

与IsSuperset方法相同，我们可以利用HashSet已有的方法来编写实现这些集合运算的方法。

我们先为这些方法确定名称，实现并集、交集、差集、对称差集运算的方法的名称分别为Union、Intersect、Difference、SymmetricDifference。请读者模仿前面已经编写完成的方法的声明自行编写出它们的声明，并满足如下4点要求。

- 它们都接受一个名为other且类型为*HashSet的参数值。
- 在方法中不得修改接收者set的值和参数other的值。
- 它们的结果的类型都应该为*HashSet。
- 尽可能地利用已有的附属于*HashSet类型的方法。

另外，需要注意，由于参数值other和其中的字段m都可能为nil，所以我们应该考虑到每个实现高级功能的方法在这种情况下的不同处理方式。比如我们前面提到过的卫述语句。

在完成了这些方法的声明之后，读者应该首先去测试它们的功能和性能。关于怎样编写单元测试程序，请读者参看第5章。在我们使用单元测试程序对HashSet类型及其方法进行了全面的测试之后，就可以放心大胆地对它们进行修改和重构了（希望读者已经根据我们的要求编写了实现那些高级功能的方法）。

4. 进一步重构

我们在本节所实现的HashSet类型提供了一些必要的集合操作功能。但是，我们在不同应用场景下可能会需要使用功能更加丰富的集合类型。我们可以对HashSet类型进行扩展（注意，不是继承）以满足我们特定的要求。我们对HashSet类型的扩展往往可以通过将它嵌入到新类型的声明中来实现。比如，我们可以使一个嵌入了HashSet类型的新类型实现sort.Interface接口类型，以使它具有对元素排序的能力（参考3.2.6节）。又比如，我们可以创建一个元素类型固定的集合类型。这需要用到代码包reflect中声明的程序实体（请参看代码包reflect的文档和3.2.7节中的方案）。

当我们有了多个集合类型的时候，就应该在它们之上抽取出一个接口类型以标识它们共有的行为方式。我们可以把这个接口类型就取名为Set。依照HashSet类型的声明，我们可以这样来声明Set接口类型：

```
type Set interface {
    Add(e interface{}) bool
    Remove(e interface{})
    Clear()
    Contains(e interface{}) bool
    Len() int
    Same(other Set) bool
    Elements() []interface{}
    String() string
}
```

注意，Set中的Same方法的签名与附属于HashSet类型的Same方法有所不同。因为我们不能在接口类型的方法的签名中包含它的实现类型。这就需要我们对HashSet类型的Same方法稍作改动：

```
func (set *HashSet) Same(other Set) bool {
    // 省略若干条语句
}
```

可以看到，我们只是修改了这个Same方法的签名。这样做的目的是让*HashSet类型成为Set接口类型的一个实现。

也许有些读者认为应该在Set接口类型的声明中加入实现高级功能的方法的声明，比如：

```
IsSuperset(other Set) bool
Union(other Set) Set
Intersect(other Set) Set
Difference(other Set) Set
SymmetricDifference(other Set) Set
```

但是，这些代表集合操作的方法应该适用于所有实现了Set接口类型的数据类型（以下简称实现类型），不是吗？这些实现了高级功能的方法（以下简称高级方法）中的核心逻辑，应该通过对那些实现了基本功能的方法（以下简称基本方法）的组合使用来实现。每个实现类型的不同之处都应该体现在它们的基本方法的实现中。在高级方法中，我们应该屏蔽掉（或者说透明化）这些不同。因此，我们完全可以把这些高级方法抽离出来，并使之成为独立的函数，以面向所有的实现类型。并且，我们也不应该在每个实现类型中重复地实现这些高级方法。读者可以试着把之前声明的这些高级方法修改为独立的、面向所有集合类型的函数。我们在这里给出改造后的IsSuperset方法的声明：

```
// 判断集合 one 是否是集合 other 的超集
func IsSuperset(one Set, other Set) bool {
    if one == nil || other == nil {
        return false
    }
    oneLen := one.Len()
    otherLen := other.Len()
    if oneLen == 0 || oneLen == otherLen {
        return false
    }
    if oneLen > 0 && otherLen == 0 {
        return true
    }
    for _, v := range other.Elements() {
        if !one.Contains(v) {
            return false
        }
    }
    return true
}
```

我们在前面重点讲述的与集合相关的数据类型和函数的参考实现，都会被放在goc2p项目中的代码包basic/set的源码文件中。不过我还是建议读者在自己实现它们之后再去与参考实现进行比较。说不定你的实现会更好。

4.5 实战演练——Ordered Map

我们已经知道，字典类型的值有一个共同特点，即其中的元素值的迭代顺序是不确定的。但

是在一些应用场景下，我们是需要固定的元素迭代顺序的。

我们在前面的章节中多次提到过，如果要使元素可排序就需要让数据类型实现sort.Interface接口类型。该接口类型中的方法Len、Less和Swap的含义分别是获取元素的数量、比较相邻元素的大小以及交换它们的位置。我们在基于数组类型或切片类型的自定义数据类型之上可以非常轻松地实现这几个方法。但是，对于基于字典类型的扩展数据类型来说，实现它们可就不那么容易了。因为字典类型值中的元素值是无序的。我们没有任何方法可以确定它们的位置以及与它们相邻的元素值。

因此，要想自定义一个有序字典类型，仅基于Go语言的字典类型是不可能实现的。我们应该使用一个元素有序的数据类型值作为辅助。依据这一思路，我声明了一个名为OrderedMap的结构体类型：

```
type OrderedMap struct {
    keys []interface{}
    m map[interface{}]interface{}
}
```

可以看到，该类型的基本结构中，除了一个字典类型的字段，还有一个切片类型的字段。为了让OrderedMap类型实现sort.Interface接口类型，我们需要为它添加如下几个方法：

```
func (omap *OrderedMap) Len() int {
    return len(omap.keys)
}

func (omap *OrderedMap) Less(i, j int) bool {
    // 省略若干条语句
}

func (omap *OrderedMap) Swap(i, j int) {
    omap.keys[i], omap.keys[j] = omap.keys[j], omap.keys[i]
}
```

这样，*OrderedMap类型（注意，不是OrderedMap类型）就是一个sort.Interface接口类型的实现类型了。可以看到，我们在这些方法中操作的实际上都是OrderedMap类型中的字段keys的值。在Len方法中，我们以keys字段的值的长度作为结果值。而在Swap方法中，我们使用平行赋值语句交换的两个元素值也都是在keys字段的值中的。这就意味着，我们会使用字段keys的值的元素迭代顺序全权代表字段m的值的元素迭代顺序。

为了达到这个目的，我们就必须要在添加和删除字段m的值中的键值对的时候对字段keys的值进行完全同步的操作。在编写相关方法之前，我们先来关注一下刚刚提到却被省略实现的Less方法。

方法Less的功能是比较相邻的两个元素值的大小并返回判断结果。我们在上一章讲值的可比性与有序性的时候介绍过各种数据类型值的比较方法。已知，只有当值的类型具备有序性的时候，它才可能与其他的同类型值比较大小。Go语言规定，字典类型的键类型的值必须是可比的（即可判定两个该类型的值是否相等）。然而，在具有可比性的数据类型中只有一部分同时具备有序

性。也就是说，我们只依靠Go语言本身对字典类型的键类型的约束是不够的。在Go语言中，具备有序性的预定义数据类型只有整数类型、浮点数类型和字符串类型。

类型OrderedMap中的字段keys是[]interface{}类型的。也就是说，我们总是需要比较两个interface{}类型值的大小。这显然是不可行的，因为接口类型的值只具有可比性而不具备有序性。所以，我们刚才声明的OrderedMap类型是不可用的。如果把keys字段的元素类型改为某一个具体的数据类型（整数类型、浮点数类型或字符串类型），虽然可以轻松编写出比较各个元素值大小的代码，但是这样做却会使OrderedMap类型的应用价值大打折扣。

总之，我们还需要重新审视一下将要创建的这个类型。

首先，我们先要对OrderedMap类型的主要功能需求进行收集和整理。实际上，这些需求都集中在对字段keys的值的操作上，如下所示。

- 字段keys的值中的元素值应该都是有序的。我们应该可以方便地比较它们之间的大小。
- 字段keys的值的元素类型不应该是一个具体的类型。我们应该可以在运行时再确定它的元素类型。
- 我们应该可以方便地对字段keys的值进行添加元素值、删除元素值以及获取元素值等操作，就像对待一个普通的切片值那样。
- 字段keys的值中的元素值应该可以被依照固定的顺序获取。
- 字段keys的值中的元素值应该能够被自动地排序。
- 由于字段keys的值中的元素值总是已排序的，所以我们应该能够确定某一个元素值的具体位置。
- 既然我们可以在运行时决定字段keys的值的元素类型，那么也应该可以在运行时获知这个元素类型。
- 我们应该可以在运行时获取到被用于比较keys的值中的不同元素值的大小的具体方法。

根据上面这些需求，我们有了这样一个接口类型声明：

```
type Keys interface {
    sort.Interface
    Add(k interface{}) bool
    Remove(k interface{}) bool
    Clear()
    Get(index int) interface{}
    GetAll() []interface{}
    Search(k interface{}) (index int, contains bool)
    ElemType() reflect.Type
    CompareFunc() func(interface{}, interface{}) int8
}
```

在Keys接口类型中嵌入sort.Interface接口类型，就意味着Keys类型的值一定是可排序的。Add、Remove、Clear和Get这4个方法使得我们可以对Keys的值进行添加、删除、清除和获取元素值的操作。GetAll方法让我们可以获取到一个与Keys类型值有着相同的元素值集合和元素迭代顺序的切片值。Search、ElemType和CompareFunc方法分别体现了需求列表中第6项、第7项和第8项所描述的功能。其中，ElemType方法返回一个reflect.Type类型的结果值。我们在之前提到过

reflect代码包，但是并没有对它进行说明。实际上，reflect包中的程序实体为我们提供了Go语言运行时的反射机制。通过它们，我们可以编写出一些代码来动态的操纵任意类型的对象。比如，其中TypeOf函数用于获取一个interface{}类型的值的动态类型信息。在本小节，我们会展示它的用法。

细心的读者可能会发现，在Keys接口类型的声明中并没有体现出需求列表中的第1、2、5项所描述的功能。不用着急，我们会在Keys接口类型的实现类型中实现它们。既然Keys接口类型的值必是sort.Interface接口的一个实现，那么我们使用sort代码包中的程序实体应该不难实现元素自动排序的功能。因此，我们编码的重点就落在了实现第1项和第2项中的功能上。

为了能够动态地决定元素类型，我们不得不在这个Keys的实现类型中声明一个[]interface{}类型的字段，以作为存储被添加到Keys类型值中的元素值的底层数据结构：

```
container   []interface{}
```

由于Go语言本身并没有对自定义泛型的提供支持，所以只有这样我们才能够用这个字段的值存储某一个数据类型的元素值。但是，我们前面提到的那个问题又出现了——接口类型的值不具备有序性，不可能比较它们的大小。不过，也许把这个问题抛出去并让使用这个Keys的实现类型的编程人员来解决它是一个可行的方案。因为他们应该知道添加到Keys类型值中的元素值的实际类型并知道应该怎样比较它们。所以，我们还应该有这样的一个字段：

```
compareFunc func(interface{}, interface{}) int8
```

这是一个函数类型的字段。就像我们刚才说的，Keys的实现的使用者应该知道需要对作为该函数参数的那两个元素值进行怎样的类型转换以及怎样比较它们。这个函数返回一个int8类型的结果值。我们在这里做出如下规定。

- 当第一个参数值小于第二个参数值时，结果值应该小于0。
- 当第一个参数值大于第二个参数值时，结果值应该大于0。
- 当第一个参数值等于第二个参数值时，结果值应该等于0。

现在，通过把比较两个元素值大小的问题抛给使用者，我们既解决了需要动态确定元素类型的问题，又明确了比较两个元素值大小的解决方式。不过还有一个问题，由于container字段是[]interface{}类型的，所以我们常常不能够很方便地在运行时获取到它的实际元素类型（比如在它的值中还没有任何元素值的时候）。因此，我们还需要一个明确container字段的实际元素类型的字段。这个字段的值所代表的类型也应该是当前的Keys类型值的实际元素类型。

综上所述，这个Keys接口类型的实现类型的声明应该是这样的：

```
type myKeys struct {
    container   []interface{}
    compareFunc func(interface{}, interface{}) int8
    elemType    reflect.Type
}
```

其中compareFunc和elemType字段的值应该是相对应的。比如，如果我们想使用一个*myKeys类型的值来存储int64类型的元素值，那么我就应该这样来初始化它：

```
int64Keys := &myKeys{
    container: make([]interface{}, 0),
    compareFunc: func(e1 interface{}, e2 interface{}) int8 {
        k1 := e1.(int64)
        k2 := e2.(int64)
        if k1 < k2 {
            return -1
        } else if k1 > k2 {
            return 1
        } else {
            return 0
        }
    },
    elemType: reflect.TypeOf(int64(1))}
```

注意，compareFunc字段的值中的那两个类型断言表达式的目标类型一定要与elemType字段的值所代表的类型保持一致。在这里，elemType字段的值所代表的类型其实就是调用reflect.TypeOf函数时传入的那个参数值的类型，即int64。这与前面在e1和e2上应用的类型断言表达式中的类型字面量是相对应的。只有存在这样的对应关系才能够保证在对变量int64Keys的使用过程中不会出现问题。我们会在编写myKeys类型的方法的过程中逐渐体现出如此设计的真正含义。

我们已经在前面的内容中多次讲过怎样实现sort.Interface接口类型中的那几个方法。不过，在这里，由于元素值之间的比较方法是由int64Keys的值的创建者确定的，所以其中的Less方法的实现会稍有不同。这些被用于实现sort.Interface接口类型的方法的声明如下：

```
func (keys *myKeys) Len() int {
    return len(keys.container)
}

func (keys *myKeys) Less(i, j int) bool {
    return keys.compareFunc(keys.container[i], keys.container[j]) == -1
}

func (keys *myKeys) Swap(i, j int) {
    keys.container[i], keys.container[j] = keys.container[j], keys.container[i]
}
```

在Less方法中，我们把比较两个元素值的操作全权交给了compareFunc字段所代表的那个函数（以下简称compareFunc函数）。如果当前值是按照我们上面所说的正确的方式来初始化的话，那么这种比较方式应该总是有效的。另外，请注意，这3个方法的接收者类型都是*myKeys，所以实现sort.Interface接口类型的类型是*myKeys而不是myKeys。

现在我们来看Add方法怎样实现。在真正向字段container的值添加元素值之前，我们应该先判断这个元素值的类型是否符合要求。这需要使用到字段elemType的值，因为它代表了可接受的元素值的类型。由于我们在很多地方都会用到这种判断，所以我们应该把实现这一功能的代码独立为一个方法，像这样：

```
func (keys *myKeys) isAcceptableElem(k interface{}) bool {
    if k == nil {
        return false
```

```
    }
    if reflect.TypeOf(k) != keys.elemType {
        return false
    }
    return true
}
```

因为Add方法的参数的类型是interface{}类型的，所以isAcceptableElem方法的参数类型也应该是interface{}的。我们先使用reflect.TypeOf函数确定参数k的实际类型，再让它与当前值的elemType字段的值进行比较。由于reflect.Type是一个接口类型，所以我们使用比较操作符!=来判定它们的相等性是合法的。

在Add方法中，我们首先使用isAcceptableElem方法来判定元素值的类型是否可被接收。如果结果是否定的，那么我们就直接返回false。如果结果是肯定的，那么我们就向container字段的值添加这个元素值。在添加之后，我们应该对container的值中的元素值进行一次排序。这需要用到sort代码包中的排序函数sort.Sort。sort.Sort函数的声明是这样的：

```
func Sort(data Interface) {
    // 省略若干条语句
}
```

函数sort.Sort的签名中的参数类型Interface其实就是接口类型sort.Interface。由于这两个程序实体处在同一个代码包中，所以在该参数类型的名称中并不用加入所属代码包名称和“.”（也就是限定前缀）。

值得一提的是，sort.Sort函数使用的排序算法是一种由三向切分的快速排序算法、堆排序算法和插入排序算法组成的混合算法。虽然快速排序是最快的通用排序算法，但在元素值很少的情况下它比插入排序要慢一些。而堆排序的空间复杂度是常数级别的，且它的时间复杂度在大多数情况下只略逊于其他两种排序算法。所以在快速排序中的递归达到一定深度的时候，切换至堆排序来节约空间是值得的。

这样的算法组合使得sort.Sort函数的时间复杂度在最坏的情况下是O(N*logN)的，并且能够有效地控制对空间的使用。但是，请注意，它并不提供稳定性的保证。稳定性是指在排序过程中能够保留数组（这里是切片值）中重复元素的相对位置。如果我们对稳定性没有特殊要求，那么选用sort.Sort函数提供的排序算法往往是最佳选择。

我们在这里选择使用sort.Sort函数对Add方法的接收者值（实际上是字段container的值）中的元素值进行排序是在算法特性和代码量之间进行权衡的结果。如果我们在使用过程中发现它的某些方面（时间复杂度、空间复杂度、稳定性等）并没有满足要求，也可以使用其他排序算法（组合）来替换对sort.Sort函数。

另一方面，由于*myKeys类型是sort.Interface接口类型的一个实现类型，所以我们可以直接使用Add方法的接收者值来作为sort.Sort的参数值。

按照上面的描述和分析，我们编写出了Add方法的声明：

```
func (keys *myKeys) Add(k interface{}) bool {
    ok := keys.isAcceptableElem(k)
```

```
    if !ok {
        return false
    }
    keys.container = append(keys.container, k)
    sort.Sort(keys)
    return true
}
```

正是有了isAcceptableElem方法的保证，我们才能够放心大胆地使用sort.Sort函数对当前值进行排序。sort.Sort函数会通过对keys的值的Len、Less和Swap方法的调用来完成排序。而在Less方法中，我们是通过那个compareFunc函数对相邻的元素值进行比较的。

这样一来，我们就真正地把elemType和compareFunc函数间接地关联了起来。换句话说，如果一个值能够通过isAcceptableElem方法的检查并被添加到container的值当中，那么这个元素值就肯定能够在compareFunc函数中被正确地比较。因此，我们在初始化myKeys类型值的时候，就必须保证这两个字段的值在类型设定方面的一致性。否则，在我们比较两个元素值的过程中就会引发运行时恐慌。

我们在从container中删除一个指定元素值之前先要找到它所处的位置。所以，我们在实现Remove方法之前先来看看Search方法应该怎样编写。在Search方法中，我们要搜索参数k代表的值在container中对应的索引值。由于k的类型是interface{}的，所以我们同样需要先使用isAcceptableElem方法对它进行判定。如果结果是否定的，我们就在把该方法的结果声明中的contains赋值为false之后直接返回。如果结果是肯定的，那么我就需要在container中搜索该元素值。我们可以通过调用sort.Search函数来实现搜索元素值的核心逻辑。sort.Search函数的声明如下：

```
func Search(n int, f func(int) bool) int {
    // 省略若干条语句
}
```

由于sort.Search函数使用二分查找算法在切片值中搜索指定的元素值。这种搜索算法有着稳定的O(logN)的时间复杂度，但它要求被搜索的数组（这里是切片值）必须是有序的。因此，我们必须确保container字段的值中的元素值是已被排过序的。幸好我们在添加元素值的时候保证了这一点。

从上面的声明可知，sort.Search函数有两个参数。第一个参数接受的应该是欲排序的切片值的长度，而第二个参数接受的是一个函数值。这个函数值的含义是：对于一个给定的索引值，判定与之对应的元素值是否等于欲查找的元素值或者应该排在欲查找的元素值的右边。由此，参数f的值应该是这样的：

```
func(i int) bool { return keys.compareFunc(keys.container[i], k) >= 0 }
```

与Less方法相同，我们在这里也是通过compareFunc函数对两个元素值进行比较的。

这个参数f的值到底意味着什么呢？假设我们有这样一个切片值：

```
[]int{1, 3, 5, 7, 9, 11, 13, 15}
```

且要查找的元素值是7。依据二分查找算法，sort.Search函数内部会在第三次折半的时候使用7的索引值3作为函数f的参数值。这时，函数f的结果值应该是true。同时，由于已经没有可以被

折半的目标子序列了，所以sort.Search函数的执行会结束并返回7的索引值3作为它的结果值。显然，这个结果值对应的元素值就是我们要查找的。

另一种情况，我们要查找的元素值根本就不在这个切片值里，比如是6或8。这时，sort.Search函数的执行也会在f(3)被求值之后结束，且它的结果值会是4或3。但是，这两个结果值对应的元素值都不是我们要查找的。

总之，sort.Search函数的结果值总会在[0, n]的范围内，但结果值并不一定就是欲查找的元素值所对应的索引值。因此，我们还需要在得到调用sort.Search函数的结果值之后再进行一次判断。代码如下：

```
if index < keys.Len() && keys.container[index] == k {
    contains = true
}
```

其中index代表了sort.Search函数的结果值。我们需要先检查结果值是否在有效的索引范围之内，然后还要判断它所对应的元素值是否就是我们要查找的。

经过前面这一系列的分析，相信读者已经能够自己实现*myKeys类型的Search函数了。现在就把它编写出来吧。

等这个函数被实现之后，我们再来看Remove函数。首先，我们需要调用*myKeys类型的Search函数，以获取欲删除的元素值对应的索引值和它是否被包含在container中的判断结果。如果第二个结果值是false，那么就直接忽略剩余的操作并直接返回false，否则就从container中删除掉这个元素值。从切片值中删除一个元素值有很多种方式，比如使用for语句、copy函数或append函数，等等。我们在这里选择用append函数来实现，因为它可以在不增加时间复杂度和空间复杂度的情况下使用更少的代码来完成功能，且不降低可读性。下面这行代码实现了删除一个元素值的功能：

```
keys.container = append(keys.container[0:index], keys.container[index+1:]...)
```

这行代码充分地使用了切片表达式和append函数。首先，我们使用切片表达式

```
keys.container[0:index]
```

和

```
keys.container[index+1:]
```

分别把container字段的值中的、在预删除元素值之前和之后的子元素序列提取出来。然后，我们再把这两个元素子序列拼接起来。还记得吗？append是一个可变参函数。所以，我们可以在第二个参数值之后添加“...”以表示把第二个参数值中的每个元素值都作为传给append函数的独立参数。这样，我们就把第二个子序列中的所有元素值逐个追加到了第一个子序列的尾部。最后，我们把拼接后的元素序列赋值给了container字段。

根据上面的描述，请读者自行编写出Remove函数。

通过在上一小节中对HashSet类型的实现，我们已经了解了Clear方法的编写手法。其声明如下：

```
func (keys *myKeys) Clear() {
    keys.container = make([]interface{}, 0)
}
```

又由于container字段本身就是切片类型的，所以Get方法也是相当好实现的。它的声明如下：

```
func (keys *myKeys) Get(index int) interface{} {
    if index >= keys.Len() {
        return nil
    }
    return keys.container[index]
}
```

方法GetAll的编写方式与*HashSet类型的Elements方法基本一致。唯一要注意的地方就是切片值的第一个迭代变量（左边的）代表了元素值的索引值，而第二个迭代变量（右边的）才代表了元素值本身。GetAll方法的声明如下：

```
func (keys *myKeys) GetAll() []interface{} {
    initialLen := len(keys.container)
    snapshot := make([]interface{}, initialLen)
    actualLen := 0
    for _, key := range keys.container {
        if actualLen < initialLen {
            snapshot[actualLen] = key
        } else {
            snapshot = append(snapshot, key)
        }
        actualLen++
    }
    if actualLen < initialLen {
        snapshot = snapshot[:actualLen]
    }
    return snapshot
}
```

至于ElemType和CompareFunc方法的实现就更不用多说了，我们直接把字段elemType和compareFunc字段的值分别作为它们的结果值就可以了：

```
func (keys *myKeys) ElemType() reflect.Type {
    return keys.elemType
}

func (keys *myKeys) CompareFunc() CompareFunction {
    return keys.compareFunc
}
```

作为一个可选的方法，String方法被用于生成可读性更好的接收者值的字符串表示形式。读者可以仿照*HashSet类型的String方法完成这一方法的编写。

至此，我们已经完成了myKeys类型以及相关方法的编写。不过按照Go语言的惯例，我们还应该编写一个用于初始化*myKeys类型值的函数。由于当前只有一个Keys接口类型的实现，所以我们就把这个函数定名为NewKeys，并把它的结果的类型设定为Keys。下面就是这个函数的声明：

```
func NewKeys(
    compareFunc func(interface{}, interface{}) int8,
    elemType reflect.Type) Keys {
    return &myKeys{
```

```
        container:   make([]interface{}, 0),
        compareFunc: compareFunc,
        elemType:    elemType,
    }
}
```

可以看到，在NewKeys函数的参数声明列表中没有与container字段相对应的参数声明。原因是container字段的值总应该是一个长度为0的[]interface{}类型值。因此它不必由NewKeys函数的调用方提供。另外，NewKeys函数的compareFunc参数和elemType参数之间的关系，也应该满足我们在讲怎样初始化myKeys类型值的时候所提及的约束条件。最后，由于只有*myKeys类型的方法集合中才包含了Keys接口类型中声明的所有方法，所以在NewKeys函数中返回的是一个*myKeys类型值，而不是一个myKeys类型值。

好了，我们已经编写完成了OrderedMap类型所需要用到的最核心的数据类型Keys和myKeys。并且，在这个过程中，我们不仅对Go语言的很多基础知识进行了复习，还了解到了一些与元素排序和运行时反射有关的知识。下面，我们再回过头来看OrderedMap类型。

由于有了Keys接口类型，OrderedMap类型的声明被修改为：

```
type myOrderedMap struct {
    keys     Keys
    elemType reflect.Type
    m        map[interface{}]interface{}
}
```

是的，我们更改了该类型的名称，这是因为我们要声明一个接口类型来描述有序字典类型所提供的功能。OrderedMap更适合作为这个接口类型的名称。OrderedMap接口类型的声明如下：

```
type OrderedMap interface {
    // 获取给定键值对应的元素值。若没有对应元素值则返回nil。
    Get(key interface{}) interface{}
    // 添加键值对，并返回与给定键值对应的旧的元素值。若没有旧元素值则返回(nil, true)。
    Put(key interface{}, elem interface{}) (interface{}, bool)
    // 删除与给定键值对应的键值对，并返回旧的元素值。若没有旧元素值则返回nil。
    Remove(key interface{}) interface{}
    // 清除所有的键值对
    Clear()
    // 获取键值对的数量
    Len() int
    // 判断是否包含给定的键值
    Contains(key interface{}) bool
    // 获取第一个键值。若无任何键值对则返回nil。
    FirstKey() interface{}
    // 获取最后一个键值。若无任何键值对则返回nil。
    LastKey() interface{}
    // 获取由小于键值toKey的键值所对应的键值对组成的OrderedMap类型值。
    HeadMap(toKey interface{}) OrderedMap
    // 获取由小于键值toKey且大于等于键值fromKey的键值所对应的键值对组成的OrderedMap类型值。
    SubMap(fromKey interface{}, toKey interface{}) OrderedMap
    // 获取由大于等于键值fromKey的键值所对应的键值对组成的OrderedMap类型值。
    TailMap(fromKey interface{}) OrderedMap
    // 获取已排序的键值所组成的切片值
```

```
    Keys() []interface{}
    // 获取已排序的元素值所组成的切片值
    Elems() []interface{}
    // 获取已排序的键值对所组成的字典值
    ToMap() map[interface{}]interface{}
    // 获取键的类型
    KeyType() reflect.Type
    // 获取元素的类型
    ElemType() reflect.Type
}
```

我们要使*myOrderedMap类型成为OrderedMap接口类型的实现类型。虽然OrderedMap接口类型中的方法声明很多，但是有了之前编写HashSet类型和myKeys类型的经验，我们实现myOrderedMap类型的这些方法应该难度不大。读者能试着把这些方法的完整声明编写出来吗？请记得再编写一个NewOrderedMap函数，并把初始化好的*myOrderedMap类型值作为结果值返回。

别担心，完整的myOrderedMap类型以及相关方法的声明连同本小节所提及的所有代码都被放到了goc2p项目的basic/omap代码包中。在必要时读者可以把它们作为参考。但是，千万不要只动眼不动手。

另外，有些读者可能会认为，这样实现出来的有序字典类型的时间复杂度和空间复杂度都比较高。当然，我们也可以使用基于B树（及其衍生数据结构）或跳跃表来实现有序字典类型。在通过这两个小节的复习和训练之后，读者应该在Go语言基础语法的运用上已经基本没有什么障碍了。那么读者是否可以试着使用Go语言来实现更高效的有序字典类型呢？

4.6 本章小结

本章我们先对Go语言的代码块和作用域进行了说明，这两个概念对于我们进一步理解Go语言程序的组织方式来说非常重要。

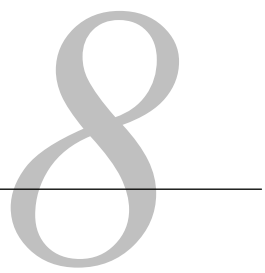
紧接着，我们对Go语言所支持的一些基本流程控制方式进行了详述。当前流行的很多编程语言都支持这些流程控制方式。这包括了if语句、switch语句、for语句、goto语句和标记语句。虽然这些流程控制语句被很多编程语言所支持，但是Go语言在它们的表现和语义的细节上却有着它自己的特点。

除了这些被广泛支持的流程控制语句之外，Go语言还有一些独特的、杀手级的流程控制方式。defer语句就是其中之一。我们可以利用该语句轻松地完成针对函数执行的收尾工作。另外，在Go语言中，报告错误状态的方式有两种。对于普通的错误状态，我们常常通过返回error类型的结果值来表达，而对于致命的和暂不可恢复的错误状态，我们往往需要引发一个运行时恐慌。

总之，本章介绍的知识是我们编写更复杂的Go语言程序的基础。搞懂这些知识对于我们编写出正确、有效的Go语言程序来说大有好处。如果你真正地理解了本章最后的那两个完整示例，并能够根据要求完善它们的话，那么就可以说你已经对Go语言程序的基本语法和编写方式足够熟悉了。

第 8 章

同　步



我们在之前多次提到，Go语言除了为应用程序开发者提供了自己特有的并发编程模型和工具之外，还提供了传统的同步工具。它们都在Go语言的标准库代码包sync和sync/atomic中。这些工具使我们有了第二种选择。它们很简单，也很直观。如果读者仔细阅读过我们在第6章介绍的多进程和多线程编程的话，应该还会记得原子操作、互斥量、条件变量等名词。在Go语言中，这些名词都被沿用了。当然，它们从概念和用法上也都是非常相似的。下面，我们就来介绍这些同步工具。

8.1　锁的使用

在本节，我们对Go语言所提供的与锁有关的API进行说明。这包括了互斥锁和读写锁。我们在第6章描述过互斥锁，但却没有提到过读写锁。这两种锁对于传统的并发程序来说都是非常常用和重要的。

1. 互斥锁

互斥锁是传统的并发程序对共享资源进行访问控制的主要手段。它由标准库代码包sync中的Mutex结构体类型代表。sync.Mutex类型（确切地说，是*sync.Mutex类型）只有两个公开方法——Lock和Unlock。顾名思义，前者被用于锁定当前的互斥量，而后者则被用来对当前的互斥量进行解锁。

类型sync.Mutex的零值表示了未被锁定的互斥量。也就是说，它是一个开箱即用的工具。我们只需对它进行简单声明就可以正常使用了：

```
var mutex sync.Mutex
mutex.Lock()
```

在我们使用其他编程语言（比如C或Java）的锁类工具的时候，可能会犯的一个低级错误就是忘记及时解开已被锁住的锁，从而导致诸如流程执行异常、线程执行停滞甚至程序死锁等一系列问题的发生。然而，在Go语言中，这个低级错误的发生几率极低。其主要原因是有defer语句的存在。

我们一般会在锁定互斥锁之后紧接着就用defer语句来保证该互斥锁的及时解锁。请看下面这个函数：

```
var mutex sync.Mutex

func write() {
    mutex.Lock()
    defer mutex.Unlock()
    // 省略若干条语句
}
```

函数write中的这条defer语句保证了在该函数被执行结束之前互斥锁mutex一定会被解锁。这省去了我们在所有return语句之前以及异常发生之时重复的附加解锁操作的工作。在函数的内部执行流程相对复杂的情况下，这个工作量是不容忽视的，并且极易出现遗漏和导致错误。所以，这里的defer语句总是必要的。在Go语言中，这是很重要的一个惯用法。我们应该养成这种良好的习惯。

对于同一个互斥锁的锁定操作和解锁操作总是应该成对地出现。如果我们锁定了一个已被锁定的互斥锁，那么进行重复锁定操作的Goroutine将会被阻塞，直到该互斥锁回到解锁状态。请看下面的示例：

```
func repeatedlyLock() {
    var mutex sync.Mutex
    fmt.Println("Lock the lock. (G0)")
    mutex.Lock()
    fmt.Println("The lock is locked. (G0)")
    for i := 1; i <= 3; i++ {
        go func(i int) {
            fmt.Printf("Lock the lock. (G%d)\n", i)
            mutex.Lock()
            fmt.Printf("The lock is locked. (G%d)\n", i)
        }(i)
    }
    time.Sleep(time.Second)
    fmt.Println("Unlock the lock. (G0)")
    mutex.Unlock()
    fmt.Println("The lock is unlocked. (G0)")
    time.Sleep(time.Second)
}
```

我们把执行repeatedlyLock函数的Goroutine称为G0。而在repeatedlyLock函数中，我们又启用了3个Goroutine，并分别把它们命名为G1、G2和G3。可以看到，我们在启用这3个Goroutine之前就已经对互斥锁mutex进行了锁定，并且在这3个Goroutine将要执行的go函数的开始处也加入了对mutex的锁定操作。这样做的意义是模拟并发地对同一个互斥锁进行锁定的情形。当for语句被执行完毕之后，我们先让G0小睡1秒钟，以使运行时系统有充足的时间开始运行G1、G2和G3。在这之后，解锁mutex。为了能够让读者更加清晰地了解到repeatedlyLock函数被执行的情况，我们在这些锁定和解锁操作的前后加入了若干条打印语句，并在打印内容中添加了我们为这几个Goroutine起的名字。也由于这个原因，我们在repeatedlyLock函数的最后再次编写了一条“睡眠”语句，以此为可能出现的其他打印内容再等待一小会儿。

经过短暂的执行，标准输出上会出现如下内容：

```
Lock the lock. (G0)
The lock is locked. (G0)
Lock the lock. (G1)
Lock the lock. (G2)
Lock the lock. (G3)
Unlock the lock. (G0)
The lock is unlocked. (G0)
The lock is locked. (G1)
```

从这8行打印内容中，我们可以清楚地看出上述4个Goroutine的执行情况。首先，在repeatedlyLock函数被执行伊始，对互斥锁的第一次锁定操作便被进行并顺利地完成。这由第一行和第二行打印内容可以看出。而后，在repeatedlyLock函数中被启用的那3个Goroutine在G0的第一次“睡眠”期间开始被运行。当相应的go函数中的对互斥锁的锁定操作被进行的时候，它们都被阻塞住了。原因是该互斥锁已处于锁定状态了。这就是我们在这里只看到了3个连续的Lock the lock. (G<i>)而没有立即看到The lock is locked. (G<i>)的原因。随后，G0“睡醒”并解锁互斥锁。这使得正在被阻塞的G1、G2和G3都会有机会重新锁定该互斥锁。但是，只有一个Goroutine会成功。成功完成锁定操作的某一个Goroutine会继续执行在该操作之后的语句。而其他Goroutine将继续被阻塞，直到有新的机会到来。这也就是上述打印内容中的最后3行所表达的含义。显然，G1抢到了这次机会并成功锁定了那个互斥锁。

实际上，我们之所以能够通过使用互斥锁对共享资源的唯一性访问进行控制，正是因为它的这一特性。这有效地对竞态条件进行了消除。

互斥锁的锁定操作的逆操作并不会引起任何Goroutine的阻塞。但是，它的进行有可能引发运行时恐慌。更确切地讲，当我们对一个已处于解锁状态的互斥锁进行解锁操作的时候，就会已发一个运行时恐慌。这种情况很可能会出现在相对复杂的流程之中——我们可能会在某个或多个分支中重复的加入针对同一个互斥锁的解锁操作。避免这种情况发生的最简单、有效的方式依然是使用defer语句。这样更容易保证解锁操作的唯一性。

虽然互斥锁可以被直接的在多个Goroutine之间共享，但是我们还是强烈建议把对同一个互斥锁的成对的锁定和解锁操作放在同一个层次的代码块中。例如，在同一个函数或方法中对某个互斥锁的进行锁定和解锁。又例如，把互斥锁作为某一个结构体类型中的字段，以便在该类型的多个方法中使用它。此外，我们还应该使代表互斥锁的变量的访问权限尽量地低。这样才能尽量避免它在不相关的流程中被误用，从而导致程序不正确的行为。

互斥锁是我们见到过的众多同步工具中最简单的一个。只要遵循前面提及的几个小技巧，我们就可以以正确、高效的方式使用互斥锁，并用它来确保对共享资源的访问的唯一性。下面我们来看看稍微复杂一些的锁实现——读写锁。

2. 读写锁

读写锁即是针对于读写操作的互斥锁。它与普通的互斥锁最大的不同就是，它可以分别针对读操作和写操作进行锁定和解锁操作。读写锁遵循的访问控制规则与互斥锁有所不同。在读写锁管辖的范围内，它允许任意个读操作同时进行。但是，在同一时刻，它只允许有一个写操作在进行。并且，在某一个写操作被进行的过程中，读操作的进行也是不被允许的。也就是说，读写锁

控制下的多个写操作之间都是互斥的，并且写操作与读操作之间也都是互斥的。但是，多个读操作之间却不存在互斥关系。

在这样的互斥策略之下，读写锁可以在大大降低因使用锁而对程序性能造成的损耗的情况下完成对共享资源的访问控制。

在Go语言中，读写锁由结构体类型sync.RWMutex代表。与互斥锁类似，sync.RWMutex类型的零值就已经是立即可用的读写锁了。在此类型的方法集合中包含了两对方法，即：

```
func (*RWMutex) Lock
func (*RWMutex) Unlock
```

和

```
func (*RWMutex) RLock
func (*RWMutex) RUnlock
```

前一对方法的名称和签名与互斥锁的那两个方法完全一致。它们分别代表了对写操作的锁定和解锁。以下简称它们为写锁定和写解锁。而后一对方法则分别表示了对读操作的锁定和解锁。以下简称它们为读锁定和读解锁。

需要特别注意的是，写解锁在进行的时候会试图唤醒所有因欲进行读锁定而被阻塞的Goroutine。而读解锁在进行的时候只会在已无任何读锁定的情况下试图唤醒一个因欲进行写锁定而被阻塞的Goroutine。若对一个未被写锁定的读写锁进行写解锁，就会引发一个运行时恐慌，而对一个未被读锁定的读写锁进行读解锁却不会如此。

无论锁定针对的是写操作还是读操作，我们都应该尽量及时对相应的锁进行解锁。对于写解锁，我们自不必多说。而读解锁的及时进行往往更容易被我们忽视。虽说读解锁的进行并不会对其他正在进行中的读操作产生任何影响，但它却与相应的写锁定的进行关系紧密。注意，对于同一个读写锁来说，施加在它之上的读锁定可以有多个。因此，只有我们对互斥锁进行相同数量的读解锁，才能够让某一个相应的写锁定获得进行的机会，否则就会继续使进行后者的Goroutine处于阻塞状态。由于sync.RWMutex和*sync.RWMutex类型都没有方法让我们获得已进行的读锁定的数量，所以这里是很容易出现问题的。还好我们可以使用defer语句来尽量避免此类问题的发生。再次强调，无论是写解锁还是读解锁，操作不及时都会对使用该读写锁的流程的正常执行产生负面影响。

除了我们在前文所述的那两对方法之外，*sync.RWMutex类型还拥有另外一个方法——RLocker。这个RLocker方法会返回一个实现了sync.Locker接口的值。sync.Locker接口类型包含了两个方法：Lock和Unlock。其实，*sync.Mutex类型和*sync.RWMutex类型都是该接口类型的实现类型。而我们在调用*sync.RWMutex类型值的RLocker方法之后所得到的结果值就是这个值本身。只不过，这个结果值的Lock方法和Unlock方法分别对应了针对该读写锁的读锁定操作和读解锁操作。换句话说，我们在对一个读写锁的RLocker方法的结果值的Lock方法或Unlock方法进行调用的时候，实际上是在调用该读写锁的RLock方法或RUnlock方法。这样的操作适配在实现上并不困难。我们自己也可以很容易的编写出这些方法的实现。通过读写锁的RLocker方法获得这样一个结果值的实际意义在于，我们可以在之后以相同的方式对该读写锁中的“写锁”和“读锁”进行操作。这为

相关操作的灵活适配和替换提供了方便。

3. 锁的完整示例

我们下面来看一个与上述锁实现有关的示例。在Go语言的标准库代码包os中有一个名为File的结构体类型。os.File类型的值可以被用来代表文件系统中的某一个文件或目录。它的方法集合中包含了很多方法，其中的一些方法被用来对相应的文件进行写操作和读操作。

假设，我们需要创建一个文件来存放数据。在同一个时刻，可能会有多个Goroutine分别进行对此文件的写操作和读操作。每一次写操作都应该向这个文件写入若干个字节的数据。这若干字节的数据应该作为一个独立的数据块存在。这就意味着，写操作之间不能彼此干扰，写入的内容之间也不能出现穿插和混淆的情况。另一方面，每一次读操作都应该从这个文件中读取一个独立、完整的数据块。它们读取的数据块不能重复，且需要按顺序读取。例如，第一个读操作读取了数据块1，那么第二个读操作就应该去读取数据块2，而第三个读操作则应该读取数据块3，以此类推。对于这些读操作是否可以被同时进行，这里并不做要求。即使它们被同时进行，程序也应该分辨出它们的先后顺序。

为了突出重点，我们规定每个数据块的长度都是相同的。该长度应该在初始化的时候被给定。若写操作实际欲写入数据的长度超过了该值，则超出部分将会被截掉。

当我们拿到这样一个需求的时候，首先应该想到使用os.File类型。它为我们操作文件系统中的文件提供了底层的支持。但是，该类型的相关方法并没有对并发操作的安全性进行保证。换句话说，这些方法都不是并发安全的。我只能通过额外的同步手段来保证这一点。鉴于这里需要分别对两类操作（即写操作和读操作）进行访问控制，所以读写锁在这里会比普通的互斥锁更加适用。不过，关于多个读操作要按顺序且不能重复读取的这个问题，我们需还要使用其他辅助手段来解决。

为了实现上述需求，我们需要创建一个类型。作为该类型的行为定义，我们先编写了一个这样的接口：

```
// 数据文件的接口类型。
type DataFile interface {
    // 读取一个数据块。
    Read() (rsn int64, d Data, err error)
    // 写入一个数据块。
    Write(d Data) (wsn int64, err error)
    // 获取最后读取的数据块的序列号。
    Rsn() int64
    // 获取最后写入的数据块的序列号。
    Wsn() int64
    // 获取数据块的长度
    DataLen() uint32
}
```

其中，类型Data被声明为一个[]byte的别名类型：

```
// 数据的类型
type Data []byte
```

而名称wsn和rsn分别是Writing Serial Number和Reading Serial Number的缩写形式。它们分别代表了最后被写入的数据块的序列号和最后被读取的数据块的序列号。这里所说的序列号相当于一个计数值，它会从1开始。因此，我们可以通过调用Rsn方法和Wsn方法得到当前已被读取和写入的数据块的数量。

根据上面对需求的简单分析和这个DataFile接口类型声明，我们就可以来编写真正的实现了。我们将这个实现类型命名为myDataFile。它的基本结构如下：

```
// 数据文件的实现类型。
type myDataFile struct {
    f       *os.File     // 文件。
    fmutex  sync.RWMutex // 被用于文件的读写锁。
    woffset int64        // 写操作需要用到的偏移量。
    roffset int64        // 读操作需要用到的偏移量。
    wmutex  sync.Mutex   // 写操作需要用到的互斥锁。
    rmutex  sync.Mutex   // 读操作需要用到的互斥锁。
    dataLen uint32       // 数据块长度。
}
```

类型myDataFile共有7个字段。我们已经在前面说明过前两个字段存在的意义。由于对数据文件的写操作和读操作是各自独立的，所以我们需要两个字段来存储两类操作的进行进度。在这里，这个进度由偏移量代表。此后，我们把woffset字段称为写偏移量，而把roffset字段称为读偏移量。注意，我们在进行写操作和读操作的时候，会分别增加这两个字段的值。当有多个写操作同时要增加woffset字段的值的时候就会产生竞态条件。因此，我们需要互斥锁wmutex来对其加以保护。类似地，rmutex互斥锁被用来消除多个读操作同时增加roffset字段的值时产生的竞态条件。最后，由上述需求可知，数据块的长度应该是在初始化myDataFile类型值的时候被给定的。这个长度会被存储在myDataFile类型值的dataLen字段中。它与DataFile接口中声明的DataLen方法是对应的。下面我们就来看看被用来创建和初始化DataFile类型值的函数NewDataFile。

关于这类函数的编写，读者应该已经驾轻就熟了。NewDataFile函数会返回一个DataFile类型的值，但是实际上它会创建并初始化一个*myDataFile类型的值并把它作为其结果值。这样可以通过编译的原因是，后者会是前者的一个实现类型。NewDataFile函数的完整声明如下：

```
func NewDataFile(path string, dataLen uint32) (DataFile, error) {
    f, err := os.Create(path)
    if err != nil {
        return nil, err
    }
    if dataLen == 0 {
        return nil, errors.New("Invalid data length!")
    }
    df := &myDataFile{f: f, dataLen: dataLen}
    return df, nil
}
```

可以看到，我们在创建*myDataFile类型值的时候，只需要对其中的字段f和dataLen进行初始化。这是因为woffset字段和roffset字段的零值都是0，而在未进行过写操作和读操作的时候，它

们的值理应如此。对于字段fmutex、wmutex和rmutex来说，它们的零值即为可用的锁。所以我们也不必再对它们进行显式地初始化。

把变量df的值作为NewDataFile函数的第一个结果值体现了我们的设计意图。但要想使*myDataFile类型真正成为DataFile类型的一个实现类型，我们还需要为*myDataFile类型编写出已在DataFile接口类型中声明的所有方法。其中最重要的当属Read方法和Write方法。

我们先来编写*myDataFile类型的Read方法，该方法应该按照如下步骤实现。

(1) 获取并更新读偏移量。

(2) 根据读偏移量从文件中读取一块数据。

(3) 把该数据块封装成一个Data类型值并将其作为结果值返回。

其中，前一个步骤在被执行的时候应该由互斥锁rmutex保护起来。因为，我们要求多个读操作不能读取同一个数据块，并且它们应该按顺序地读取文件中的数据块。而第二个步骤，我们也会用读写锁fmutex加以保护。下面是这个Read方法的第一个版本：

```
func (df *myDataFile) Read() (rsn int64, d Data, err error) {
    // 读取并更新读偏移量
    var offset int64
    df.rmutex.Lock()
    offset = df.roffset
    df.roffset += int64(df.dataLen)
    df.rmutex.Unlock()

    //读取一个数据块
    rsn = offset / int64(df.dataLen)
    df.fmutex.RLock()
    defer df.fmutex.RUnlock()
    bytes := make([]byte, df.dataLen)
    _, err = df.f.ReadAt(bytes, offset)
    if err != nil {
        return
    }
    d = bytes
    return
}
```

可以看到，在读取并更新读偏移量的时候，我们用到了rmutex字段。这保证了可能同时运行在多个Goroutine中的以下两行代码

```
offset = df.roffset
df.roffset += int64(df.dataLen)
```

的执行是互斥的。这是我们为了获取到不重复且正确的读偏移量所必需采取的措施。

另一方面，在读取一个数据块的时候，我们适时地进行了fmutex字段的读锁定和读解锁操作。fmutex字段的这两个操作可以保证我们在这里读取到的是完整的数据块。不过，这个完整的数据块却并不一定是正确的。为什么会这样说呢?

请想象这样一个场景。在我们的程序中，有3个Goroutine来并发的执行某个*myDataFile类型值的Read方法，并有2个Goroutine来并发的执行该值的Write方法。通过前3个Goroutine的运行，

数据文件中的数据块被依次地读取了出来。但是，由于进行写操作的Goroutine比进行读操作的Goroutine少，所以过不了多久读偏移量roffset的值就会等于甚至大于写偏移量woffset的值。也就是说，读操作很快就会没有数据可读了。这种情况会使上面的df.f.ReadAt方法返回的第二个结果值为代表错误的非nil且会与io.EOF相等的值。实际上，我们不应该把这样的值看成错误的代表，而应该把它看成一种边界情况。但不幸的是，我们在这个版本的Read方法中并没有对这种边界情况做出正确的处理。该方法在遇到这种情况时会直接把错误值返回给它的调用方。该调用方会得到读取出错的数据块的序列号，但却无法再次尝试读取这个数据块。由于其他正在或后续执行的Read方法会继续增加读偏移量roffset的值，所以当该调用方再次调用这个Read方法的时候只可能读取到在此数据块后面的其他数据块。注意，执行Read方法时遇到上述情况的次数越多，被漏读的数据块也就会越多。为了解决这个问题，我们编写了Read方法的第二个版本：

```
func (df *myDataFile) Read() (rsn int64, d Data, err error) {
    // 读取并更新读偏移量
    // 省略若干条语句

    //读取一个数据块
    rsn = offset / int64(df.dataLen)
    bytes := make([]byte, df.dataLen)
    for {
        df.fmutex.RLock()
        _, err = df.f.ReadAt(bytes, offset)
        if err != nil {
            if err == io.EOF {
                df.fmutex.RUnlock()
                continue
            }
            df.fmutex.RUnlock()
            return
        }
        d = bytes
        df.fmutex.RUnlock()
        return
    }
}
```

在上面的Read方法展示中，我们省略了若干条语句。原因在这个位置上的那些语句并没有任何变化。为了进一步节省篇幅，我们在后面也会遵循这样的省略原则。

第二个版本的Read方法使用for语句是为了达到这样一个目的：在其中的df.f.ReadAt方法返回io.EOF错误的时候，继续尝试获取同一个数据块，直到获取成功为止。注意，如果在该for代码块被执行期间，一直让读写锁fmutex处于读锁定状态，那么针对它的写锁定操作将永远不会成功，且相应的Goroutine也会被一直阻塞。因为它们是互斥的。所以，我们不得不在该for语句块中的每条return语句和continue语句的前面都加入一个针对该读写锁的读解锁操作，并在每次迭代开始时都对fmutex进行一次读锁定。显然，这样的代码看起来很丑陋。冗余的代码会使代码的维护成本和出错几率大大增加。并且，当for代码块中的代码引发了运行时恐慌的时候，我们是很难及时对读写锁fmutex进行读解锁的。即便可以这样做，那也会使Read方法的实现更加

丑陋。我们因为要处理一种边界情况而去掉了defer df.fmutex.RUnlock()语句。这种做法利弊参半。

其实，我们可以做得更好。但是这涉及了其他同步工具。因此，我们以后再来对Read方法进行进一步的改造。顺便提一句，当df.f.ReadAt方法返回一个非nil且不等于io.EOF的错误值的时候，我们总是应该放弃再次获取目标数据块的尝试，而立即将该错误值返回给Read方法的调用方。因为这样的错误很可能是严重的（比如，f字段代表的文件被删除了），需要交由上层程序去处理。

现在，我们来考虑*myDataFile类型的Write方法。与Read方法相比，Write方法的实现会简单一些。因为后者不会涉及边界情况。在该方法中，我们需要进行两个步骤：获取并更新写偏移量和向文件写入一个数据块。我们直接给出Write方法的实现：

```
func (df *myDataFile) Write(d Data) (wsn int64, err error) {
    // 读取并更新写偏移量
    var offset int64
    df.wmutex.Lock()
    offset = df.woffset
    df.woffset += int64(df.dataLen)
    df.wmutex.Unlock()

    //写入一个数据块
    wsn = offset / int64(df.dataLen)
    var bytes []byte
    if len(d) > int(df.dataLen) {
        bytes = d[0:df.dataLen]
    } else {
        bytes = d
    }
    df.fmutex.Lock()
    df.fmutex.Unlock()
    _, err = df.f.Write(bytes)
    return
}
```

这里需要注意的是，当参数d的值的长度大于数据块的最大长度的时候，我们会先进行截短处理再将数据写入文件。如果没有这个截短处理，我们在后面计算的已读数据块的序列号和已写数据块的序列号就会不正确。

有了编写前面两个方法的经验，我们可以很容易的编写出*myDataFile类型的Rsn方法和Wsn方法：

```
func (df *myDataFile) Rsn() int64 {
    df.rmutex.Lock()
    defer df.rmutex.Unlock()
    return df.roffset / int64(df.dataLen)
}

func (df *myDataFile) Wsn() int64 {
    df.wmutex.Lock()
    defer df.wmutex.Unlock()
```

```
    return df.woffset / int64(df.dataLen)
}
```

这两个方法的实现分别涉及到了对互斥锁rmutex和wmutex的锁定操作。同时，我们也通过使用defer语句保证了对它们的及时解锁。在这里，我们对已读数据块的序列号rsn和已写数据块的序列号wsn的计算方法与前面示例中的方法是相同的。它们都是用相关的偏移量除以数据块长度后得到的商来作为相应的序列号（或者说计数）的值。

至于*myDataFile类型的DataLen方法的实现，我们无需呈现。它只是简单地将dataLen字段的值作为其结果值返回而已。

编写上面这个完整示例的主要目的是展示互斥锁和读写锁在实际场景中的应用。由于还没有讲到Go语言提供的其他同步工具，所以我们在相关方法中所有需要同步的地方都是用锁来实现的。然而，其中的一些问题用锁来解决是不足够或不合适的。我们会在本节的后续部分中逐步对它们进行改进。

从普通的互斥锁和读写锁的源码中可以看出，它们是同源的。读写锁的内部是用互斥锁来实现写锁定操作之间的互斥的。我们可以把读写锁看作是互斥锁的一种扩展。除此之外，这两种锁实现在内部都用到了操作系统提供的同步工具——信号灯。互斥锁内部使用一个二值信号灯（只有两个可能的值的信号灯）来实现锁定操作之间的互斥，而读写锁内部则使用一个二值信号灯和一个多值信号灯（可以有多个可能的值的信号灯）来实现写锁定操作与读锁定操作之间的互斥。当然，为了进行精确的协调，它们还使用到了其他一些字段和变量。由于篇幅原因，我们就不在这里赘述了。如果读者对此感兴趣的话，可以去阅读sync代码包中的相关源码文件。

8.2 条件变量

我们在第6章讲多线程编程的时候，详细说明过条件变量的概念、原理和适用场景。因此，本小节仅对sync代码包中与条件变量相关的API进行简单的介绍，并使用它们来改造我们之前实现的*myDataFile类型的相关方法。

在Go语言中，sync.Cond类型代表了条件变量。与互斥锁和读写锁不同，简单的声明无法创建出一个可用的条件变量。为了得到这样一个条件变量，我们需要用到sync.NewCond函数。该函数的声明如下：

```
func NewCond(l Locker) *Cond
```

我们在第6章中说过，条件变量总是要与互斥量组合使用。因此，sync.NewCond函数的唯一参数是sync.Locker类型的，而具体的参数值既可以是一个互斥锁也可以是一个读写锁。sync.NewCond函数在被调用之后会返回一个*sync.Cond类型的结果值。我们可以调用该值拥有的几个方法来操纵对应的条件变量。

类型*sync.Cond的方法集合中有3个方法，即Wait方法、Signal方法和Broadcast方法。它们分别代表了等待通知、单发通知和广播通知的操作。

方法Wait会自动地对与该条件变量关联的那个锁进行解锁，并且使调用方所在的Goroutine

被阻塞。一旦该方法收到通知，就会试图再次锁定该锁。如果锁定成功，它就会唤醒那个被它阻塞的Goroutine。否则，该方法会等待下一个通知，那个Goroutine也会继续被阻塞。而方法Signal和Broadcast的作用都是发送通知以唤醒正在为此而被阻塞的Goroutine。不同的是，前者的目标只有一个，而后者的目标则是所有。

我们在6.3.2节中详细地描述过这些操作地行为和意义。读者可以在需要时回顾其中的内容。

在上一小节，我们在*myDataFile类型的Read方法和Write方法的实现中使用到了读写锁fmutex。在Read方法中，我们对一种边界情况进行了特殊处理，即如果*os.File类型的f字段的ReadAt方法在被调用后返回了一个非nil且等于io.EOF的错误值，那么Read方法就忽略这个错误并再次尝试读取相同位置的数据块，直到读取成功为止。从这个特殊处理的具体流程上来看，似乎使用条件变量来作为辅助手段会带来一些好处。下面我们就来动手试验一下。

我们先在结构体类型myDataFile增加一个类型为*sync.Cond的字段rcond。为了快速实现想法，我们暂时不考虑怎样初始化这个字段，而直接去改造Read方法和Write方法。

在Read方法中，我们使用一个for循环来达到重新尝试获取数据块的目的。为此，我们添加了若干条重复的语句、降低了程序的性能，还造成了一个潜在的问题——在某个情况下读写锁fmutex不会被读解锁。为了解决这一系列新生的问题，我们使用代表条件变量的字段rcond。Read方法的第三个版本如下：

```
func (df *myDataFile) Read() (rsn int64, d Data, err error) {
    // 读取并更新读偏移量
    // 省略若干条语句

    //读取一个数据块
    rsn = offset / int64(df.dataLen)
    bytes := make([]byte, df.dataLen)
    df.fmutex.RLock()
    defer df.fmutex.RUnlock()
    for {
        _, err = df.f.ReadAt(bytes, offset)
        if err != nil {
            if err == io.EOF {
                df.rcond.Wait()
                continue
            }
            return
        }
        d = bytes
        return
    }
}
```

在这里，我们假设条件变量rcond与读写锁fmutex中的“读锁”相关联。可以看到，我们让defer df.fmutex.RUnlock()语句回归了，并删除了所有return语句和continue语句前面的针对fmutex的读解锁操作。这都得益于新增在continue语句前面的df.rcond.Wait()。添加这条语句的意义在于：当发现由文件内容读取造成的EOF错误时，要让当前Goroutine暂时放弃fmutex的“读

锁”并等待通知的到来。放弃fmutex的“读锁”也就意味着Write方法中的数据块写操作不会受到它的阻碍了。在写操作完成之后，我们应该及时向条件变量rcond发送通知以唤醒为此而等待的Goroutine。请注意，在某个Goroutine被唤醒之后，应该再次检查需要被满足的条件。在这里，这个需要被满足的条件是在进行文件内容读取时不会造成EOF错误。如果该条件被满足，那么就可以进行后续的操作了。否则，应该再次放弃“读锁”并等待通知。这也是我们依然保留for循环的原因。

这里有两点需要特别注意。

- 一定要在调用rcond的Wait方法之前锁定与之关联的那个“读锁”，否则就会造成对rcond.Wait方法的调用永远无法返回。这种情况会导致流程执行的停滞，甚至整个程序的死锁！导致这种结果的原因与条件变量和读写锁的内部实现方式有关（结果也许并不应该是这样，我已经向Go语言官方提交了一个issue，Go语言官方已经接受了这个issue，并承诺将会在Go 1.4版本中改进它）。另外，假设与条件变量rcond关联的是某个读写锁的“写锁”或普通的互斥锁，那么对rcond.Wait方法的调用将会引发一个运行时恐慌。原因是，该方法会先对与之关联的锁进行解锁，而试图解锁未被锁定的锁就会引发一个运行时恐慌。
- 一定不要忘记在读操作完成之前解锁与条件变量rcond关联的那个“读锁”，否则对读写锁的写锁定操作将会阻塞相关的Goroutine。其根本原因是，条件变量rcond的Wait方法在返回之前会重新锁定与之关联的那个“读锁”。因此，在结束这个从文件中读取一个数据块的流程之前，我们应该调用fmutex字段的RLock方法。那条defer语句就起到了这个作用。

我们对Read方法的这次改进使得它的实现变得更加简洁和清晰了。不过，要想使其中的条件变量rcond真正发挥作用，还需要Write方法的配合。换句话说，为了让rcond.Wait方法可以适时地返回，我们要在向文件写入一个数据块之后及时向rcond发送通知。添加了这一操作的Write方法如下：

```
func (df *myDataFile) Write(d Data) (wsn int64, err error) {
    // 省略若干条语句
    var bytes []byte
    // 省略若干条语句
    df.fmutex.Lock()
    defer df.fmutex.Unlock()
    _, err = df.f.Write(bytes)
    df.rcond.Signal()
    return
}
```

由于一个数据块只能由某一个读操作读取，所以我们只是使用条件变量的Signal方法去通知某一个为此等待的Wait方法，并以此唤醒某一个相关的Goroutine。这可以免去其他相关的Goroutine中的一些无谓操作。

与Wait方法不同，我们在调用条件变量的Signal方法和Broadcast方法之前无需锁定与之关联的锁。随之，相应的解锁操作也是不需要的。在这个Write方法中的锁定操作和解锁操作与

df.rcond.Signal()语句之间并没有联系。

我们一直在说，条件变量rcond是与读写锁fmutex的“读锁”关联的。这是怎样做到的呢？读者还记得我们在上一节提到读写锁的RLocker方法吗？它会返回当前读写锁中的“读锁”。这个结果值同时也是sync.Locker接口的实现。因此，我们可以把它作为参数值传给sync.NewCond函数。所以，我们在NewDataFile函数中的声明df变量的语句的后面加入了这样一条语句：

```
df.rcond = sync.NewCond(df.fmutex.RLocker())
```

在这之后，我们就可以像前面那样使用这个条件变量了。

随着对*myDataFile类型和NewDataFile函数的改造的完成，我们也将结束本节。Go语言提供的互斥锁、读写锁和条件变量都基本遵循了POSIX标准中描述的对应的同步工具的行为规范。它们简单且高效。我们可以使用它们为复杂的类型提供并发安全的保证。在一些情况下，它们比通道更加适用。在只需对一个或多个临界区进行保护的时候，使用锁往往会对程序的性能损耗更小。

在下一节中，我们将会介绍对程序性能损耗更小的同步工具——原子操作。同样地，我们会使用这一工具进一步改造*myDataFile类型及其方法。

8.3　原子操作

我们已经知道，原子操作即是进行过程中不能被中断的操作。针对某个值的原子操作在被进行的过程当中，CPU绝不会再去进行其他的针对该值的操作。无论这些“其他的操作”是否为原子操作都会是这样。为了实现这样的严谨性，原子操作仅会由一个独立的CPU指令代表和完成。只有这样才能够在并发环境下保证原子操作的绝对安全。

Go语言提供的原子操作都是非侵入式的。它们由标准库代码包sync/atomic中的众多函数代表。我们可以通过调用这些函数对几种简单的类型的值进行原子操作。这些类型包括int32、int64、uint32、uint64、uintptr和unsafe.Pointer类型，共6个。这些函数提供的原子操作共有5种：增或减、比较并交换、载入、存储和交换。它们分别提供了不同的功能，且适用的场景也有所区别。下面，我们就根据这些种类对Go语言提供的原子操作进行逐一的讲解。

1. 增或减

被用于进行增或减的原子操作（以下简称原子增/减操作）的函数名称都以“Add”为前缀，并后跟针对的具体类型的名称。例如，实现针对uint32类型的原子增/减操作的函数的名称为AddUint32。事实上，sync/atomic包中的所有函数的命名都遵循此规则。

顾名思义，原子增/减操作即可实现对被操作值的增大或减小。因此，被操作值的类型只能是数值类型。更具体地讲，它只能是我们在前面提到的int32、int64、uint32、uint64和uintptr类型。例如，我们如果想原子地把一个int32类型的变量i32的值增大3的话，可以这样做：

```
newi32 := atomic.AddInt32(&i32, 3)
```

我们将指向i32变量的值的指针值和代表增减的差值3作为参数传递给了atomic.AddInt32函数。之所以要求第一个参数值必须是一个指针类型的值，是因为该函数需要获得被操作值在内存中的存放位置，以便施加特殊的CPU指令。也就是说，对于一个不能被取址的数值，我们是无法进行原子

操作的。此外，这类函数的第二个参数的类型被操作值的类型总是相同的。因此，在前面那个调用表达式被求值的时候，字面量3会被自动转换为一个int32类型的值。函数atomic.AddInt32在被执行结束之时，会返回经过原子操作后的新值。不过不要误会，我们无需把这个新值再赋给原先的变量i32。因为它的值已经在atomic.AddInt32函数返回之前被原子地修改了。

与该函数类似的还有atomic.AddInt64函数、atomic.AddUint32函数、atomic.AddUint64函数和atomic.AddUintptr函数。这些函数也可以被用来原子地增/减对应类型的值。例如，如果我们要原子地将int64类型的变量i64的值减小3话，可以这样编写代码：

```
var i64 int64
atomic.AddInt64(&i64, -3)
```

不过，由于atomic.AddUint32函数和atomic.AddUint64函数的第二个参数的类型分别是uint32和uint64，所以我们无法通过传递一个负的数值来减小被操作值。那么，这是不是就意味着我们无法原子地减小uint32或uint64类型的值了呢？幸好，不是这样。Go语言为我们提供了一个可以迂回的达到此目的办法。

如果我们想原子的把uint32类型的变量ui32的值增加NN（NN代表了一个负整数），那么我们可以这样调用atomic.AddUint32函数：

```
atomic.AddUint32(&ui32, ^uint32(-NN-1))
```

对于uint64类型的值来说也是这样。调用表达式

```
atomic.AddUint64(&ui64, ^uint64(-NN-1))
```

表示原子的把uint64类型的变量ui64的值增加NN（或者说减小-NN）。

这种方式之所以可以奏效，是因为它利用了二进制补码的特性。我们知道，一个负整数的补码可以通过对它按位（除了符号位之外）求反码并加一得到。我们还知道，一个负整数可以由对它的绝对值减一并求补码后得到的数值的二进制表示来代表。例如，如果NN是一个int类型的变量且其值为-35，那么表达式

```
uint32(int32(NN))
```

和

```
^uint32(-NN-1)
```

的结果值就都会是11111111111111111111111111011101。由此，我们使用^uint32(-NN-1)和^uint64(-NN-1)来分别表示uint32类型和uint64类型的NN就顺理成章了。这样，我们就可以合理地绕过uint32类型和uint64类型对值的限制了。

以上是官方提供一种通用解决方案。除此之外，我们还有两个非通用的方案可供选择。首先，需要明确的是，对于一个代表负数的字面常量来说，它们是无法通过简单的类型转换将其转换为uint32类型或uint64类型的值的。例如，表达式uint32(-35)和uint64(-35)都是不合法的。它们都不能通过编译。但是，如果我们事先把这个字面量赋给一个变量然后再对这个变量进行类型转换，那么就可以得到Go语言编译器的认可。我们依然以值为-35的变量NN为例，下面这条语句可以通过编译并被正常执行：

```
fmt.Printf("The variable: %b.\n", uint32(NN))
```

其输出内容为：

```
The variable: 11111111111111111111111111011101.
```

可以看到，表达式uint32(NN)的结果值的二进制表示与前面的uint32(int32(NN))表达式以及^uint32(-NN-1)表达式的结果值是一致的。它们都可以被用来表示uint32类型的-35。因此，我们也可以使用下面的调用表达式来原子的把变量ui32的值减小-NN：

```
atomic.AddUint32(&ui32, uint32(NN))
```

不过，这样的编写方式仅在NN是数值类型的变量的时候才可以通过编译。如果NN是一个常量，那么也会使表达式uint32(NN)不合法并无法通过编译。它与表达式uint32(-35)造成的编译错误是一致的。在这种情况下，我们可以这样来达到上述目的：

```
atomic.AddUint32(&ui32, NN&math.MaxUint32)
```

其中，我们用到了标准库代码包math中的常量MaxUint32。math.MaxUint32常量表示的是一个32位的、所有二进制位上均为1的数值。我们把NN和math.MaxUint32进行按位与操作的意义是使前者的值能够被视为一个uint32类型的数值。实际上，对于表达式NN&math.MaxUint32来说，其结果值的二进制表示与前面uint32(int32(NN))表达式以及^uint32(-NN-1)表达式的结果值也是一致的。

我们在这里介绍的这两种非官方的解决方案是不能混用的。更具体地说，如果NN是一个常量，那么表达式uint32(NN)是无法通过编译的。而如果NN是一个变量，那么表达式NN&math.MaxUint32就无法通过编译。前者的错误在于代表负整数的字面常量不能被转换为uint32类型的值。后者的错误在于这个按位与运算的结果值的类型不是uint32类型而是int类型，从而导致数据溢出的错误。相比之下，官方给出的那个解决方案的适用范围更广。

有些读者可能会有这样的疑问：为什么如此曲折地实现这一功能？直接声明出atomic.SubUint32()函数和atomic.SubUint64()函数不好吗？我的理解是，不这样做是为了让这些原子操作的API可以整齐划一，并且避免在扩充它们的时候使sync/atomic包中声明的程序实体成倍增加。（我向Go语言官方提出了这个问题，并引发了一些讨论，他们也许会使用投票的方式来选取更好一些的方案。）

注意，并不存在名为atomic.AddPointer的函数，因为unsafe.Pointer类型值之间既不能被相加也不能被相减。

2. 比较并交换

有些读者可能很熟悉比较并交换操作的英文称谓——Compare And Swap，简称CAS。在sync/atomic包中，这类原子操作由名称以“CompareAndSwap”为前缀的若干个函数代表。

我们依然以针对int32类型值的函数为例。该函数名为CompareAndSwapInt32。其声明如下：

```
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
```

可以看到，CompareAndSwapInt32函数接受3个参数。第一个参数的值应该是指向被操作值的指针值。该值的类型即为*int32。后两个参数的类型都是int32类型。它们的值应该分别代表被

操作值的旧值和新值。CompareAndSwapInt32函数在被调用之后，会先判断参数addr指向的被操作值与参数old的值是否相等。仅当此判断得到肯定的结果之后，该函数才会用参数new代表的新值替换掉原先的旧值。否则，后面的替换操作就会被忽略。这正是“比较并交换”这个短语的由来。CompareAndSwapInt32函数的结果swapped被用来表示是否进行了值的替换操作。

与我们前面讲到的锁相比，CAS操作有明显的不同。它总是假设被操作值未曾被改变（即与旧值相等），并一旦确认这个假设的真实性就立即进行值替换。而使用锁则是更加谨慎的做法。我们总是先假设会有并发的操作要修改被操作值，并使用锁将相关操作放入临界区中加以保护。我们可以说，使用锁的做法趋于悲观，而CAS操作的做法则更加乐观。

CAS操作的优势是，可以在创建互斥量和不形成临界区的情况下完成并发安全的值替换操作。这可以大大地减少同步对程序性能的损耗。当然，CAS操作也有劣势。在被操作值被频繁变更的情况下，CAS操作并不那么容易成功。有些时候，我们可能不得不利用for循环以进行多次尝试。示例如下：

```
var value int32
func addValue(delta int32) {
    for {
        v := value
        if atomic.CompareAndSwapInt32(&value, v, (v + delta)) {
            break
        }
    }
}
```

可以看到，为了保证CAS操作的成功完成，我们仅在CompareAndSwapInt32函数的结果值为true时才会退出循环。这种做法与自旋锁的自旋行为相似。addValue函数会不断地尝试原子地更新value的值，直到这一操作成功为止。操作失败的缘由总会是value的旧值已不与v的值相等了。如果value的值会被并发地修改的话，那么发生这种情况是很正常的。

CAS操作虽然不会让某个Goroutine阻塞在某条语句上，但是仍可能会使流程的执行暂时停滞。不过，这种停滞的时间大都极其短暂。

请记住，如果想并发安全地更新一些类型（更具体地讲是前文所述的那6个类型）的值，我们总是应该优先选择CAS操作。

与此对应，被用来进行原子的CAS操作的函数共有6个。除了我们已经讲过的CompareAndSwapInt32函数之外，还有CompareAndSwapInt64、CompareAndSwapPointer、CompareAndSwapUint32、CompareAndSwapUint64和CompareAndSwapUintptr函数。这些函数的结果声明列表与CompareAndSwapInt32函数的完全一致。而它们的参数声明列表与后者也非常类似。虽然其中的那3个参数的类型不同，但其遵循的规则是一致的，即第二个和第三个参数的类型均为与第一个参数的类型（即某个指针类型）紧密相关的那个类型。例如，如果第一个参数的类型为*unsafe.Pointer，那么后两个参数的类型就一定是unsafe.Pointer。这也是由这3个参数的含义决定的。

3. 载入

在前面展示的for循环中，我们使用语句v := value为变量v赋值。但是，要注意，在进行读取

value的操作的过程中，其他对此值的读写操作是可以被同时进行的。它们并不会受到任何限制。

在7.1节的最后，我们举过这样一个例子：在32位计算架构的计算机上写入一个64位的整数。如果在这个写操作未完成的时候，有一个读操作被并发地进行了，那么这个读操作很可能会读取到一个只被修改了一半的数据。这种结果是相当糟糕的。

为了原子地读取某个值，sync/atomic代码包同样为我们提供了一系列的函数。这些函数的名称都以“Load”为前缀，意为载入。我们依然以针对int32类型值的那个函数为例。

我们下面利用LoadInt32函数对上一个示例稍作修改：

```
func addValue(delta int32) {
    for {
        v := atomic.LoadInt32(&value)
        if atomic.CompareAndSwapInt32(&value, v, (v + delta)) {
            break
        }
    }
}
```

函数atomic.LoadInt32接受一个*int32类型的指针值，并会返回该指针值指向的那个值。在该示例中，我们使用调用表达式atomic.LoadInt32(&value)替换掉了标识符value。替换后，那条赋值语句的含义就变为：原子地读取变量value的值并把它赋给变量v。有了“原子地”这个词的修饰就意味着，在这里读取value的值的同时，当前计算机中的任何CPU都不会进行其他针对此值的读或写操作。这样的约束是受到底层硬件的支持的。

注意，虽然我们在这里使用atomic.LoadInt32函数原子地载入value的值，但是其后面的CAS操作仍然是有必要的。因为，那条赋值语句和if语句并不会被原子地执行。在它们被执行期间，CPU仍然可能进行其他针对value的值的读或写操作。也就是说，value的值仍然有可能被并发地改变。

与atomic.LoadInt32类似的函数有atomic.LoadInt64、atomic.LoadPointer、atomic.LoadUint32、atomic.LoadUint64和atomic.LoadUintptr。

4. 存储

与读取操作相对应的是写入操作。而sync/atomic包也提供了与原子的值载入函数相对应的原子的值存储函数。这些函数的名称均以“Store”为前缀。

在原子地存储某个值的过程中，任何CPU都不会进行针对同一个值的读或写操作。如果我们把所有针对此值的写操作都改为原子操作，那么就不会出现针对此值的读操作因被并发地进行而读到修改了一半的值的情况了。

原子的值存储操作总会成功，因为它并不会关心被操作值的旧值是什么。显然，这与前面讲到的CAS操作是有着明显的区别的。因此，我们并不能把前面展示的addValue函数中的调用atomic.CompareAndSwapInt32函数的表达式替换为对atomic.StoreInt32函数的调用表达式。

函数atomic.StoreInt32会接受两个参数。第一个参数的类型是*int 32类型的，其含义同样是指向被操作值的指针值。而第二个参数则是int32类型的，它的值应该代表欲存储的新值。其他同类函数也会有类似的参数声明列表。

5. 交换

在sync/atomic代码包中还存在着一类函数。它们的功能与前文所讲的CAS操作和原子载入操作都有些相似之处。这样的功能可以被称为原子交换操作。这类函数的名称都以“Swap”为前缀。

与CAS操作不同，原子交换操作不会关心被操作值的旧值。它会直接设置新值。但它又比原子载入操作多做了一步。作为交换，它会返回被操作值的旧值。此类操作比CAS操作的约束更少，同时又比原子载入操作的功能更强。

以atomic.SwapInt32函数为例。它接受两个参数。第一个参数是代表了被操作值的内存地址的*int32类型值，而第二个参数则被用来表示新值。注意，该函数是有结果值的。该值即是被新值替换掉的旧值。atomic.SwapInt32函数被调用后，会把第二个参数值置于第一个参数值所表示的内存地址上（即修改被操作值），并将之前在该地址上的那个值作为结果返回。其他的同类函数的声明和作用都与此类似。

至此，我们快速且简要地介绍了sync/atomic代码包中的所有函数的功能和用法。这些函数都被用来对特定类型的值进行原子性的操作。如果我们想以并发安全的方式操作单一的特定类型（int32、int64、uint32、uint64、uintptr或unsafe.Pointer）的值的话，应该首先考虑使用这些函数来实现。请注意，原子的减小一些特定类型（确切地说，是uint32类型和uint64类型）的值的实现方式并不那么直观。在Go语言官方对此进行改进之前，我们应该按照他们为我们提供的特定方法来进行此类操作。

6. 应用于实际

下面，我们就使用刚刚介绍的知识再次对在前面示例中创建的*myDataFile类型进行改造。在*myDataFile类型的第二个版本中，我们仍然使用两个互斥锁来对与roffset字段和woffset字段相关的操作进行保护。*myDataFile类型的方法中的绝大多数都包含了这些操作。

首先，我们来看对roffset字段的操作。在*myDataFile类型的Read方法中有这样一段代码：

```
// 读取并更新读偏移量
var offset int64
df.rmutex.Lock()
offset = df.roffset
df.roffset += int64(df.dataLen)
df.rmutex.Unlock()
```

这段代码的含义是读取读偏移量的值并把它存入到局部变量中，然后增加读偏移量的值以使其他并发的读操作能够被正确、有效地进行。为了使程序能够在并发环境下有序地对roffset字段进行操作，我们给这段代码加上了互斥锁rmutex。

字段roffset和变量offset都是int64类型的。后者代表了前者的旧值。而字段roffset的新值即为其旧值与dataLen字段的值的和。实际上，这正是原子的CAS操作的适用场景。我们现在用CAS操作来实现该段代码的功能：

```
// 读取并更新读偏移量
var offset int64
for {
    offset = df.roffset
    if atomic.CompareAndSwapInt64(&df.roffset, offset,
```

```
            (offset + int64(df.dataLen))) {
            break
        }
    }
```

根据roffset和offset的类型，我们选用atomic.CompareAndSwapInt64来进行CAS操作。我们在调用该函数的时候传入了3个参数，分别代表了被操作值的地址、被操作数的旧值和欲设置的新值。如果该函数的结果值是true，那么我们就退出for循环。这时，变量offset即是我们需要的读偏移量的值。另一方面，如果该函数的结果值是false，那么就说明在从完成读取到开始更新roffset字段的值的期间内，有其他并发操作对该值进行了更改。当遇到这种情况，我们就需要再次尝试。只要尝试失败，我们就会重新读取roffset字段的值并试图对该值进行CAS操作，直到成功为止。具体的尝试次数与具体的并发环境有关。

我们在前面说过，在32位计算架构的计算机上写入一个64位的整数也会存在并发安全方面的隐患。因此，我们还应该将这段代码中的offset = df.roffset语句修改为offset = atomic.LoadInt64(&df.roffset)。

除了这里，在*myDataFile类型的Rsn方法中也有针对roffset字段的读操作：

```
df.rmutex.Lock()
defer df.rmutex.Unlock()
return df.roffset / int64(df.dataLen)
```

我们现在去掉施加在上面的锁定和解锁操作，转而使用原子操作来实现它。修改后的代码如下：

```
offset := atomic.LoadInt64(&df.roffset)
return offset / int64(df.dataLen)
```

这样，我们就在依然保证相关操作的并发安全的前提下去除了对互斥锁rmutex的使用。对于字段woffset和互斥锁wmutex，我们也应该如法炮制。读者可以试着按照上面的方法修改与之相关的Write方法和Wsn方法。

在修改完成之后，我们就可以把代表互斥锁的rmutex字段和wmutex字段从*myDataFile类型的基本结构中去掉了。这样，该类型的基本结构会显得精简了不少。

通过本次改造，我们减少了*myDataFile类型及其方法对互斥锁的使用。这对该程序的性能和可伸缩性都会有一定的提升。其主要原因是，原子操作由底层硬件支持，而锁则由操作系统提供的API实现。若实现相同的功能，前者通常会更有效率。读者可以为前面展示的这3个版本的*myDataFile类型的实现编写性能测试，以验证上述观点的正确性。

总之，我们要善用原子操作。因为它比锁更加简单和高效。不过，由于原子操作自身的限制，锁依然常用且重要。

8.4　只会执行一次

现在，让我们再次聚焦到sync代码包。除了我们介绍过的互斥锁、读写锁和条件变量，该代码包还为我们提供了几个非常有用的API。其中一个比较有特色的就是结构体类型sync.Once和它

的Do方法。

与代表锁的结构体类型sync.Mutex和sync.RWMutex一样，sync.Once也是开箱即用的。换句话说，我们仅需对它进行简单的声明即可使用，就像这样：

```
var once sync.Once
once.Do(func() { fmt.Println("Once!") })
```

如上所示，我们声明了一个名为once的sync.Once类型的变量之后，立刻就可以调用它的指针方法Do了。

该类型的方法Do可以接受一个无参数、无结果的函数值作为其参数。该方法一旦被调用，就会调用被作为参数传入的那个函数。从这一点看，该方法的功能实在是稀松平常。不过，重点并不在这里。

我们对一个sync.Once类型值的指针方法Do的有效调用次数永远会是1。也就是说，无论我们调用这个方法多少次，也无论我们在多次调用时传递给它的参数值是否相同，都仅有第一次调用是有效的。无论怎样，只有我们第一次调用该方法时传递给它的那个函数会被执行。请看下面的示例：

```
func onceDo() {
    var num int
    sign := make(chan bool)
    var once sync.Once
    f := func(ii int) func() {
        return func() {
            num = (num + ii*2)
            sign <- true
        }
    }
    for i := 0; i < 3; i++ {
        fi := f(i + 1)
        go once.Do(fi)
    }
    for j := 0; j < 3; j++ {
        select {
        case <-sign:
            fmt.Println("Received a signal.")
        case <-time.After(100 * time.Millisecond):
            fmt.Println("Timeout!")
        }
    }
    fmt.Printf("Num: %d.\n", num)
}
```

在onceDo函数中，我们利用for语句连续3次异步地调用once变量的Do方法。这3次调用传给Do方法的参数值，都是变量fi所代表的匿名函数值。这个函数值的功能是先改变num变量的值，再向非缓冲的sign通道发送一个true。变量num的值可以表示出once的Do方法被有效调用的次数，而通道sign则被用来传递代表了fi函数被执行完毕的信号。请注意，为了能够精确地表达出fi函数是在哪一次（或哪几次）调用once.Do方法的时候被执行的，我们在这里使用了闭包。在每

次迭代之初，我们赋给fi变量的函数值都是对变量f所代表的函数值进行闭包的一个结果值。我们使用变量ii作为f函数中的自由变量，并在闭包的过程中把for代码块中的变量i的值加1后再与该自由变量绑定在一起。这样就生成了为当次迭代专门定制的函数fi。迭代中生成的fi函数在每次被执行的时候都会修改变量num的值。这些新的值不会出现重复，并且非常有助于我们倒推出所有的曾经赋给自由变量ii的值。这样，我们就可以知道哪个（或哪些）fi函数被真正地执行了。

函数onceDo中的第二条for语句的作用是等待之前的那3个异步调用的完成。读者可能已经发现，这两条for语句的预设迭代次数是一致的。在第二条for语句中，我们使用了select语句，并且为针对sign通道的接收操作设定了超时时间（100毫秒）。这是为了当永远无法从sign通道中接收元素值的时候不至于造成永久的阻塞。select语句中的每个case在被执行时都会打印出相应的内容。这有助于我们观察程序的实际运行情况。最后，我们还会打印出num变量的值。据此，我们可以判断在前面几次传递给Do方法的fi是否都被执行了。

在执行onceDo函数之后，我们会看到如下打印内容：

```
Received a signal.
Timeout!
Timeout!
Num: 2.
```

上面的打印内容表明，在成功从sign通道接收了一个元素值之后，出现了两次接收操作超时的情况。我们不用考虑在对sign通道的接收操作开始之时匿名函数fi还没有被执行完毕的情况。因为100毫秒的时间已经足够执行它很多次的了。因此，这两次接收操作超时应该是当时没有正在为此等待的对sign通道的发送操作导致的（注意，sign是一个非缓冲通道）。综上所述，我们可以初步判断，传递给once.Do方法的匿名函数fi只被执行了一次。并且，这仅有一次的执行的对象是在我们第一次调用该方法时传递给它的那个fi函数。

依据最后一行打印内容，我们可以证实上述判断。num变量的值为2意味着它只被修改了一次，并且是在自由变量ii为1的时候被修改的。这就可以证实，只有在for循环的第一次迭代时传递给once.Do方法的那个fi函数被执行了。这也符合sync.Once类型及其指针方法Do的语义。

请注意，这个仅被执行一次的限制只是针对单个sync.Once类型值来说的。换句话说，每个sync.Once类型值的指针方法Do都可以被有效地调用一次。

这个sync.Once类型的典型应用场景就是执行仅需执行一次的任务。例如，数据库连接池的初始化任务。又例如，一些需要持续运行的实时监测任务，等等。

在一探sync.Once类型及其指针方法Do的内部实现之后，我们会有所发现：它们所提供的功能正是由前面讲到的互斥锁和原子操作来实现的。这个实现并不复杂。其使用的技巧包括卫述语句、双重检查锁定，以及对共享标记的原子读写操作。在熟知了本章讲述的这些同步工具之后，我们是否也能轻易设计出这样简单且有效的解决方案呢？

总之，sync.Once类型及其方法实现了“只会执行一次”的语义。我们在需要完成只需或只能执行一次的任务的时候应该首先想到它。

8.5 WaitGroup

我们在第6章多次提到过sync.WaitGroup类型和它的方法。sync.WaitGroup类型的值也是开箱即用的。例如，在声明

```
var wg sync.WaitGroup
```

之后，我们就可以直接正常使用wg变量了。该类型有3个指针方法，即Add、Done和Wait。

类型sync.WaitGroup是一个结构体类型。在它之中有一个代表计数的字段。当一个sync.WaitGroup类型的变量被声明之后，其值中的那个计数值将会是0。我们可以通过该值的Add方法增大或减少其中的计数值。例如：

```
wg.Add(3)
```

或

```
wg.Add(-3)
```

虽然Add方法接受一个int类型的值，并且我们也可以通过该方法减少计数值，但是我们一定不要让计数值变为负数。因为这样会立即引发一个运行恐慌。这也代表着我们对sync.WaitGroup类型值的错误使用。

除了调用sync.WaitGroup类型值的Add方法并传入一个负数之外，我们还可以通过调用该值的Done来使其中的计数值减一。也就是说，下面这3条语句与wg.Add(-3)的执行效果是一致的：

```
wg.Done()
wg.Done()
wg.Done()
```

使用该方法的禁忌与Add方法的一样——不要让值中的计数值变为负数。例如，这段代码中的第5条语句会引发一个运行时恐慌：

```
var wg sync.WaitGroup
wg.Add(2)
wg.Done()
wg.Done()
wg.Done()
```

我们现在知道，使用sync.WaitGroup类型值的Add方法和Done方法可以变更其中的计数值。那么变更这个计数值有什么用呢？

当我们调用sync.WaitGroup类型值的Wait方法的时候，它会去检查该值中的计数值。如果这个计数值为0，那么该方法会立即返回，且不会对程序的运行产生任何影响。但是，如果这个计数值大于0，那么该方法的调用方所属的那个Goroutine就会被阻塞。直到该计数值重新变为0之时，为此而被阻塞的所有Goroutine才会被唤醒。

这个类型的值一般被用来协调多个Goroutine的运行。假设，在我们的程序中启用了4个Goroutine，分别是G1、G2、G3和G4。其中，G2、G3和G4是由G1中的代码启用并被用于执行某些特定任务的。G1在启用这3个Goroutine之后要等待这些特定任务的完成。在这种情况下，我们有两个方案。

第一个方案是使用前文讲到的通道来传递任务完成信号。例如，我们在启用G2、G3和G4之前声明这样一个通道：

```
sign := make(chan byte, 3)
```

然后，在G2、G3和G4执行的任务完成之后，立即向该通道发送代表了某个任务已被执行完成的元素值：

```
go func() { // G2
    // 省略若干条语句
    sign <- 2
}()

go func() { // G3
    // 省略若干条语句
    sign <- 3
}()

go func() { // G4
    // 省略若干条语句
    sign <- 4
}()
```

最后，在启用这几个Goroutine之后，我们还要在G1执行的函数中添加类似以下的代码，以等待相关的任务完成信号：

```
for i := 0; i < 3; i++ {
    fmt.Printf("G%d is ended.\n", <-sign)
}
// 省略若干条语句
```

这样的方法固然是有效的。上面的这条for语句会等到G2、G3和G4都被运行结束之后才会被执行结束，继而其后面的语句才会得以执行。sign通道起到了协调这4个Goroutine的运行的作用。

不过，对于这样一个简单的协调工作来说，使用通道是否过重了？或者说，通道sign是否被大材小用了？通道的实现中包含了很多专为并发安全的传递数据而建立的数据结构和算法。原则上说，我们不应该把通道当作互斥锁或信号灯来说用。在这里使用它并没有体现出它的优势，反而会在代码易读性和程序性能方面打一些折扣。

该需求的第二个方案就是使用sync.WaitGroup类型值。对应的代码如下：

```
var wg sync.WaitGroup
wg.Add(3)

go func() { // G2
    // 省略若干条语句
    wg.Done()
}()

go func() { // G3
    // 省略若干条语句
    wg.Done()
}()
```

```
go func() { // G4
    // 省略若干条语句
    wg.Done()
}()

wg.Wait()
fmt.Println("G2, G3 and G4 are ended.")
```

可以看到，我们在启用G2、G3和G4之前先声明了一个sync.WaitGroup类型的变量wg，并调用其值的Add方法以使其中的计数值等于将要额外启用的Goroutine的个数。然后，在G2、G3和G4的运行即将结束之前，我们分别通过调用wg.Done方法将其中的计数值减去1。最后，我们在G1中调用wg.Wait方法以等待G2、G3和G4中的那3个对wg.Done方法的调用的完成。待这3个调用完成之时，在wg.Wait()处被阻塞的G1会被唤醒，它后面的那条语句也会被立即执行。

不论是Add方法还是Done方法，它们在被执行的时候都会在增大或减小其所属值中的那个计数值之后对它进行判断。如果该计数值为0，那么该方法就会唤醒所有已为此而被阻塞的Goroutine（如果有的话）。这些Goroutine即是在从该计数值最近一次变为正整数到此时（即重新变为0）的时间段内执行该sync.WaitGroup类型值的Wait方法的Goroutine。

显然，我们的第二个方案更加适合这里的应用场景。它在代码的清晰度和性能损耗方面都会更胜一筹。

在这里，我们可以总结出一些使用一个sync.WaitGroup类型值的方法和规则。

- 对一个sync.WaitGroup类型值的Add方法的的第一次调用，应该发生在对该值的Done方法进行调用之前。因为如果先调用了Done方法，那么就会使该值中的计数值小于0，继而引发运行时恐慌。由于这两个方法通常不会在同一个Goroutine中被调用，所以调用Add方法的时机还应该提前到将会调用该值的Done方法的那个或那些Goroutine被启用之前。
- 对一个sync.WaitGroup类型值的Add方法的第一次调用，同样应该发生在对该值的Wait方法进行调用之前。如果在我们调用Wait方法的时候该值的计数值等于0，那么该方法将会直接返回而不会阻塞调用方所属的Goroutine。这往往是与我们的期望相悖的。
- 在一个sync.WaitGroup类型值的生命周期内，其中的计数值总是由起初的0变为某个正整数（或先后变为某几个正整数），然后再回归为0。我们把完成这样一个变化曲线所用的时间称为一个计数周期，如图8-1所示。

  如图8-1所示，计数值的每次变化都是由对其所属值的Add方法或Done方法的调用引起的。一个计数周期总是从对其所属值的Add方法的调用开始的，并且也总是以对其所属值的Add方法或Done方法的调用为结束标志的。我们若在一个计数周期之内（不包含计数值等于0的两端）调用其所属值的Wait方法，则会使调用方所在的Goroutine被阻塞，直至该计数周期结束的那一刻。

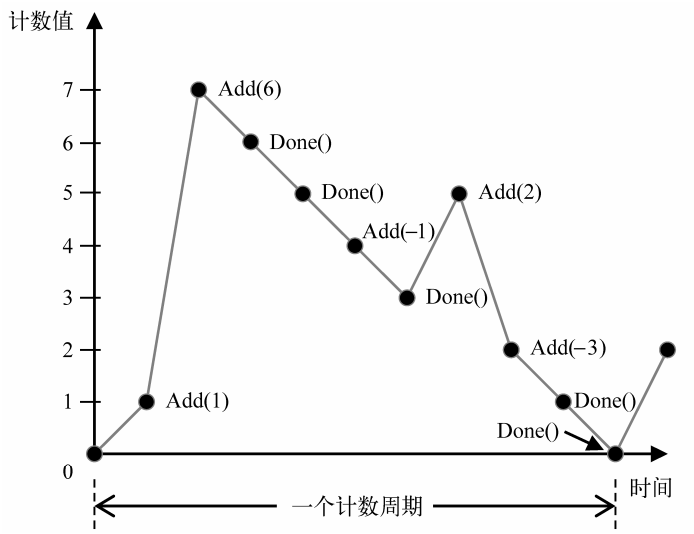


图8-1　sync.WaitGroup类型值的计数值的变化曲线示意

- sync.WaitGroup类型值是可以被复用的。也就是说，此类值的生命周期可以包含任意个计数周期。一旦一个计数周期结束，我们在前面对该值的那些方法的调用所产生的作用就会消失。也就是说，它们不会影响到后续计数周期中的该值的计数值以及参与改变该计数值的各方。换句话讲，一个sync.WaitGroup类型值在其每个计数周期中的状态和作用都是独立的。

最后，值得说明的是，在sync.WaitGroup类型及其方法中也用到了在前面章节中提到的互斥锁、原子操作和信号灯机制。这使得我们总是可以在任意个Goroutine中并发地调用同一个sync.WaitGroup类型值的那些方法。也就是说，它们都是并发安全的。

本节所讲的sync.WaitGroup类型提供了一种方式，使我们可以对多个Goroutine的运行进行简单的协调。这得益于它提供的那几个以计数值为基础的方法，以及它的并发安全特性。只要理解了每个方法对计数值的操纵方式以及意义，我们就可以用好该类型的值了。我们刚刚说明的那些使用方法和规则，对理解该类型及其方法应该是非常有帮助的。

8.6　临时对象池

本节要讲解的是sync.Pool类型。我们可以把sync.Pool类型值看作是存放可被重复使用的值的容器。此类容器是自动伸缩的，高效的，同时也是并发安全的。为了描述方便，我们也会把sync.Pool类型的值称为临时对象池，而把存于其中的值称为对象值。至于为什么要加“临时“这两个字，我们稍后再解释。

我们在用复合字面量初始化一个临时对象池的时候，可以为它唯一的公开字段New赋值。该字段的类型是func() interface{}，即一个函数类型。可以猜到，被赋给字段New的函数会被临时

对象池用来创建对象值。不过，实际上，该函数几乎仅在池中无可用对象值的时候才会被调用。

类型sync.Pool有两个公开的方法。一个是Get，另一个是Put。前者的功能是从池中获取一个interface{}类型的值，而后者的作用则是把一个interface{}类型的值放置于池中。

通过Get方法获取到的值是任意的。如果一个临时对象池的Put方法未被调用过，且它的New字段也未曾被赋予一个非nil的函数值，那么它的Get方法返回的结果值就一定会是nil。我们稍后会讲到，Get方法返回的不一定就是存在于池中的值。不过，如果这个结果值是池中的，那么在该方法返回它之前就一定会把它从池中删除掉。

这样一个临时对象池在功能上与一个通用的缓存池有几分相似。但是实际上，临时对象池本身的特性决定了它是一个个性非常鲜明的同步工具。我们在这里说明它的两个非常突出的特性。

第一个特性是，临时对象池可以把由其中的对象值产生的存储压力进行分摊。更进一步说，它会专门为每一个与操作它的Goroutine相关联的P都生成一个本地池。在临时对象池的Get方法被调用的时候，它一般会先尝试从与本地P对应的那个本地池中获取一个对象值。如果获取失败，它就会试图从其他P的本地池中偷一个对象值并直接返回给调用方。如果依然未果，那它只能把希望寄托于当前的临时对象池的New字段代表的那个对象值生成函数了。注意，这个对象值生成函数产生的对象值永远不会被放置到池中。它会被直接返回给调用方。另一方面，临时对象池的Put方法会把它的参数值存放到与当前P对应的那个本地池中。每个P的本地池中的绝大多数对象值都是被同一个临时对象池中的所有本地池所共享的。也就是说，它们随时可能会被偷走。

临时对象池的第二个突出特性是对垃圾回收友好。垃圾回收的执行一般会使临时对象池中的对象值被全部移除。也就是说，即使我们永远不会显式地从临时对象池取走某一个对象值，该对象值也不会永远待在临时对象池中。它的生命周期取决于垃圾回收任务下一次的执行时间。

请读者阅读一下这段代码：

```
package main

import (
    "fmt"
    "runtime"
    "runtime/debug"
    "sync"
    "sync/atomic"
)

func main() {
    // 禁用GC，并保证在main函数执行结束前恢复GC
    defer debug.SetGCPercent(debug.SetGCPercent(-1))
    var count int32
    newFunc := func() interface{} {
        return atomic.AddInt32(&count, 1)
    }
    pool := sync.Pool{New: newFunc}

    // New 字段值的作用
```

```
    v1 := pool.Get()
    fmt.Printf("v1: %v\n", v1)

    // 临时对象池的存取
    pool.Put(newFunc())
    pool.Put(newFunc())
    pool.Put(newFunc())
    v2 := pool.Get()
    fmt.Printf("v2: %v\n", v2)

    // 垃圾回收对临时对象池的影响
    debug.SetGCPercent(100)
    runtime.GC()
    v3 := pool.Get()
    fmt.Printf("v3: %v\n", v3)
    pool.New = nil
    v4 := pool.Get()
    fmt.Printf("v4: %v\n", v4)
}
```

在这里，我们使用runtime/debug代码包的SetGCPercent函数来禁用、恢复GC以及指定垃圾收集比率（详见7.1节中的相关说明），以保证我们的演示能够如愿进行。

我们把这段代码存放在gocp项目的sync1/pool代码包的文件pool_demo.go中，并使用go run命令运行它：

```
hc@ubt:~/golang/goc2p/src/sync1/pool$ go run pool_demo.go
```

而后，我们会在标准输出上看到如下内容：

```
v1: 1
v2: 2
v3: 5
v4: <nil>
```

请读者注意第3行和第4行的内容，也就是我们在手动地进行垃圾回收之后的输出内容。在把nil赋给pool的New字段之前，即使手动地执行了垃圾回收，我们也是可以从临时对象池获取到一个对象值的。而在这之后，我们却只能取出nil。读者应该可以依据我们刚刚描述的那两个特性想明白如此输出的原因。

看到这里，读者可能会隐约地感觉到，我们在使用临时对象池的时候应该依照一些方式方法，否则就会很容易迈入陷坑。实际情况确实如此。

首先，我们不能对通过Get方法获取到的对象值有任何假设。到底哪一个值会被取出是完全不确定的。这是因为我们总是不能得知操作临时对象池的Goroutine在哪一时刻会与哪一个P相关联，尤其是在比上述示例更加复杂的程序的运行过程中。在这种情况下，我们也就无从知晓我们放入的对象值会被存放到哪一个P的本地池中，以及哪一个Goroutine执行的Get方法会返回该对象值。所以，我们给予临时对象池的对象值生成函数所产生的值，以及通过调用它的Put方法放入到池中的值，都应该是无状态的或者状态一致的。从另一方面说，我们在取出并使用这些值的时候，也不应该以其中的任何状态作为先决条件。这一点非常重要。

第二个需要注意的地方实际上与我们前面讲到的第二个特性紧密相关。临时对象池中的任何对象值都有可能在任何时候被移除掉，并且根本不会通知该池的使用方。这种情况常常会发生在垃圾回收器即将开始回收内存垃圾的时候。如果这时临时对象池中的某个对象值仅被该池引用，那么它还可能会在垃圾回收的时候被回收掉。因此，我们也就不能假设之前放入到临时对象池的某个对象值会一直待在池中，即使我们没有显式地把它从池中取出。甚至一个对象值可以在临时对象池中待多久，我们也无法假设。除非我们像前面的示例那样手动地控制GC的启停。不过，我们并不推荐这种方式。这会带来一些其他问题。

依据我们刚刚讲述的临时对象池特性和使用注意事项，读者应该可以想象得出临时对象池的一些适用场景（比如作为临时且状态无关的数据的暂存处），以及一些不适用的场景（比如用来存放数据库连接的实例）。如果我们在做实现技术的选型的时候把临时对象池作为了候选之一，那么就应该好好想想它的“个性”是不是符合你的需要。如果真的适合，那么它的特性一定会为你的程序增光添彩，无论在功能上还是在性能上。而如果它被用在了不恰当的地方，那么就只能适得其反了。

8.7 实战演练——Concurrent Map

我们在本章前面的部分中对Go语言提供的各种传统同步工具和方法进行了逐一的介绍。在本节，我们将运用它们来构造一个并发安全的字典（Map）类型。

我们已经知道，Go语言提供的字典类型并不是并发安全的。因此，我们需要使用一些同步方法对它进行扩展。这看起来并不困难。我们只要使用读写锁将针对一个字典类型值的读操作和写操作保护起来就可以了。确实，读写锁应该是我们首先想到的同步工具。不过，我们还不能确定只使用它是否就足够了。不管怎样，让我们先来编写并发安全的字典类型的第一个版本。

我们先来确定并发安全的字典类型的行为。还记得吗？依然，这需要声明一个接口类型。我们在第4章带领读者编写过OrderedMap接口类型及其实现类型。我们可以借鉴OrderedMap接口类型的声明，并编写出需要在这里声明的接口类型ConcurrentMap。实际上，ConcurrentMap接口类型的方法集合应该是OrderedMap接口类型的方法集合的一个子集。我们只需从OrderedMap中去除那些代表有序Map特有行为的方法声明即可。既然是这样，为何不从这两个自定义的字典接口类型中抽出一个公共接口呢？

这个公共的字典接口类型可以是这样的：

```
// 泛化的Map的接口类型
type GenericMap interface {
    // 获取给定键值对应的元素值。若没有对应元素值则返回nil。
    Get(key interface{}) interface{}
    // 添加键值对，并返回与给定键值对应的旧的元素值。若没有旧元素值则返回(nil, true)。
    Put(key interface{}, elem interface{}) (interface{}, bool)
    // 删除与给定键值对应的键值对，并返回旧的元素值。若没有旧元素值则返回nil。
    Remove(key interface{}) interface{}
    // 清除所有的键值对。
    Clear()
```

```
    // 获取键值对的数量。
    Len() int
    // 判断是否包含给定的键值。
    Contains(key interface{}) bool
    // 获取已排序的键值所组成的切片值。
    Keys() []interface{}
    // 获取已排序的元素值所组成的切片值。
    Elems() []interface{}
    // 获取已包含的键值对所组成的字典值。
    ToMap() map[interface{}]interface{}
    // 获取键的类型。
    KeyType() reflect.Type
    // 获取元素的类型。
    ElemType() reflect.Type
}
```

然后，我们把这个名为GenericMap的字典接口类型嵌入到OrderedMap接口类型中，并去掉后者中的已在前者内声明的那些方法。修改后的OrderedMap接口类型如下：

```
// 有序的Map的接口类型。
type OrderedMap interface {
    GenericMap // 泛化的Map接口
    // 获取第一个键值。若无任何键值对则返回nil。
    FirstKey() interface{}
    // 获取最后一个键值。若无任何键值对则返回nil。
    LastKey() interface{}
    // 获取由小于键值toKey的键值所对应的键值对组成的OrderedMap类型值。
    HeadMap(toKey interface{}) OrderedMap
    // 获取由小于键值toKey且大于等于键值fromKey的键值所对应的键值对组成的OrderedMap类型值。
    SubMap(fromKey interface{}, toKey interface{}) OrderedMap
    // 获取由大于等于键值fromKey的键值所对应的键值对组成的OrderedMap类型值。
    TailMap(fromKey interface{}) OrderedMap
}
```

我们要记得在修改完成后立即使用go test命令重新运行相关的功能测试，以此确保这样的重构没有破坏任何现有的功能。

有了GenericMap接口类型之后，我们的ConcurrentMap接口类型的声明就相当简单了。由于后者没有任何特殊的行为，所以我们只要简单地将前者嵌入到后者的声明中即可：

```
type ConcurrentMap interface {
    GenericMap
}
```

下面我们来编写该接口类型的实现类型。我们依然使用一个结构体类型来充当，并把它命名为myConcurrentMap。myConcurrentMap类型的基本结构如下：

```
type myConcurrentMap struct {
    m        map[interface{}]interface{}
    keyType  reflect.Type
    elemType reflect.Type
    rwmutex  sync.RWMutex
}
```

有了编写myOrderedMap类型（还记得吗？它的指针类型是OrderedMap的实现类型）的经验，写出myConcurrentMap类型的基本结构也是一件比较容易的事情。可以看到，在基本需要之外，我们只为myConcurrentMap类型加入了一个代表了读写锁的rwmutex字段。此外，我们需要为myConcurrentMap类型添加的那些指针方法的实现代码实际上也可以以myOrderedMap类型中的相应方法为蓝本。不过，在实现前者的过程中，要注意合理运用同步方法以保证它们的并发安全性。下面，我们就开始编写它们。

首先，我们来看Put、Remove和Clear这几个方法。它们都属于写操作，都会改变myConcurrentMap类型的m字段的值。

方法Put的功能是向myConcurrentMap类型值添加一个键值对。那么，我们在这个操作的前后一定要分别锁定和解锁rwmutex的写锁。Put方法的实现如下：

```
func (cmap *myConcurrentMap) Put(key interface{}, elem interface{}) (interface{}, bool) {
    if !cmap.isAcceptablePair(key, elem) {
        return nil, false
    }
    cmap.rwmutex.Lock()
    defer cmap.rwmutex.Unlock()
    oldElem := cmap.m[key]
    cmap.m[key] = elem
    return oldElem, true
}
```

该实现中的isAcceptablePair方法的功能是检查参数值key和elem是否均不为nil，并且它们的类型是否均与当前值允许的键类型和元素类型一致。在通过该检查之后，我们就需要对rwmutex进行写锁定了。相应地，我们使用defer语句来保证对它的及时写解锁。与此类似，我们在Remove和Clear方法的实现中也应该加入相同的操作。

与这些代表着写操作的方法相对应的，是代表读操作的方法。在ConcurrentMap接口类型中，此类方法有Get、Len、Contains、Keys、Elems和ToMap。我们需要分别在这些方法的实现中加入对rwmutex的读锁的锁定和解锁操作。以Get方法为例，我们应该这样来实现它：

```
func (cmap *myConcurrentMap) Get(key interface{}) interface{} {
    cmap.rwmutex.RLock()
    defer cmap.rwmutex.RUnlock()
    return cmap.m[key]
}
```

这里有两点需要特别注意。

- 我们在使用写锁的时候，要注意方法间的调用关系。比如，一个代表写操作的方法中调用了另一个代表写操作的方法。显然，我们在这两个方法中都会用到读写锁中的写锁。如果使用不当，我们就会使后者被永远锁住，而前者中的流程也会永远停在那里。当然，对于代表写操作的方法调用代表读操作的方法的这种情况来说，也会是这样。请看下面的示例：

```
func (cmap *myConcurrentMap) Remove(key interface{}) interface{} {
    cmap.rwmutex.Lock()
    defer cmap.rwmutex.Unlock()
    oldElem := cmap.Get()
```

```
        delete(cmap.m, key)
        return oldElem
    }
```

可以看到，我们在Remove方法中调用了Get方法。并且，在这个调用之前，我们已经锁定了rwmutex的写锁。然而，由前面的展示可知，我们在Get方法的开始处对rwmutex的读锁进行了锁定。由于这两个锁定操作之间的互斥性，所以我们一旦调用这个Remove方法就会使当前Goroutine永远陷入阻塞。更严重的是，在这之后，其他Goroutine在调用该*myConcurrentMap类型值的一些方法（涉及到写锁定或读锁定的那些方法）的时候也会立即被阻塞住。

我们应该避免这种情况的发生。这里有两种解决方案。第一种解决方案是，把Remove方法中的oldElem := cmap.Get()语句与在它前面的那两条语句的位置互换，即变为：

```
oldElem := cmap.Get()
cmap.rwmutex.Lock()
defer cmap.rwmutex.Unlock()
```

这样可以保证在解锁读锁之后才会去锁定写锁。相比之下，第二种解决方案更加彻底一些，即消除掉方法间的调用。也就是说，我们需要把oldElem := cmap.Get()语句替换掉。在Get方法中，体现其功能的语句是oldElem := cmap.m[key]。因此，我们把后者作为前者的替代品。若如此，那么我们必须保证该语句出现在对写锁的锁定操作之后。这样，我们才能依然确保其在锁的保护之下。实际上，通过这样的修改，我们升级了Remove方法中被用来保护从m字段中获取对应元素值的这一操作的锁（由读锁升级至写锁）。

- 对于rwmutex字段的读锁来说，虽然锁定它的操作之间不是互斥的，但是这些操作与相应的写锁的锁定操作之间却是互斥的。我们在上一条注意事项中已经说明了这一点。因此，为了最小化对写操作的性能的影响，我们应该在锁定读锁之后尽快的对其进行解锁。也就是说，我们要在相关的方法中尽量减少持有读锁的时间。这需要我们综合地考量。

依据前面的示例和注意事项说明，读者可以试着实现Remove、Clear、Len、Contains、Keys、Elems和ToMap方法。它们实现起来并不困难。注意，我们想让*myConcurrentMap类型成为ConcurrentMap接口类型的实现类型。因此，这些方法都必须是myConcurrentMap类型的指针方法。这包括马上要提及的那两个方法。

方法KeyType和ElemType的实现极其简单。我们可以直接分别返回myConcurrentMap类型的keyType字段和elemType字段的值。这两个字段的值应该是在myConcurrentMap类型值的使用方初始化它的时候给出的。

按照惯例，我们理应提供一个可以方便地创建和初始化并发安全的字典值的函数。我们把它命名为NewConcurrentMap，其实现如下：

```
func NewConcurrentMap(keyType, elemType reflect.Type) ConcurrentMap {
    return &myConcurrentMap{
        keyType:  keyType,
        elemType: elemType,
        m:        make(map[interface{}]interface{})}
}
```

这个函数并没有什么特别之处。由于myConcurrentMap类型的rwmutex字段并不需要额外的初始化，所以它并没有出现在该函数中的那个复合字面量中。此外，为了遵循面向接口编程的原则，我们把该函数的结果的类型声明为了ConcurrentMap，而不是它的实现类型*myConcurrentMap。如果将来我们编写出了另一个ConcurrentMap接口类型的实现类型，那么就应该考虑调整该函数的名称。比如变更为NewDefaultConcurrentMap，或者其他。

待读者把还未实现的*myConcurrentMap类型的那几个方法都补全之后（可以利用NewConcurrentMap函数来检验这个类型是否是一个合格的ConcurrentMap接口的实现类型），我们就开始一起为该类型编写功能和性能测试了。

参照我们之前为*myOrderedMap类型编写的功能测试，我们可以很快地照猫画虎地创建出*myConcurrentMap类型的功能测试函数。这些函数和本小节前面讲到的所有代码都被放到了goc2p项目的basic/map1代码包中。其中，接口类型ConcurrentMap的声明和myConcurrentMap类型的基本结构及其所有的指针方法均在库源码文件cmap.go中。因此，我们应该把对应的测试代码放到cmap_test.go文件中。

既然有了很好的参照，我并不想再赘述*myConcurrentMap类型的功能测试函数了。我希望读者能够先独立的编写出来并通过go test命令的检验，然后再去与cmap_test.go文件中的代码对照。

另外，在myConcurrentMap类型及其指针方法的实现中，我们多处用到了读写锁和反射API（声明在reflect代码包中的那些公开的程序实体）。它们执行的都是可能会对程序性能造成一定影响的操作。因此，针对*myConcurrentMap类型的性能测试（或称基准测试）是很有必要的。这样我们才能知道它的值在性能上到底与官方的字典类型有怎样的差别。

我们在测试源码文件cmap_test.go文件中声明两个基准测试函数——BenchmarkConcurrentMap和BenchmarkMap。顾名思义，这两个函数是分别被用来测试*myConcurrentMap类型和Go语言官方的字典类型的值的性能的。

在BenchmarkConcurrentMap函数中，我们执行这样一个流程。

(1) 初始化一个*myConcurrentMap类型的值，同时设定键类型和元素类型均为int32类型。

(2) 执行迭代次数预先给定（即该函数的*testing.B类型的参数b的字段N的值）的循环。在单次迭代中，我们向字典类型值添加一个键值对，然后再试图从该值中获取与当前键值对应的元素值。

(3) 打印出一行提示信息，包含该值的键类型、元素类型以及长度等内容。

下面是该函数的实现：

```
func BenchmarkConcurrentMap(b *testing.B) {
    keyType := reflect.TypeOf(int32(2))
    elemType := keyType
    cmap := NewConcurrentMap(keyType, elemType)
    var key, elem int32
    fmt.Printf("N=%d.\n", b.N)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        b.StopTimer()
        seed := int32(i)
        key = seed
```

```
            elem = seed << 10
            b.StartTimer()
            cmap.Put(key, elem)
            _ = cmap.Get(key)
            b.StopTimer()
            b.SetBytes(8)
            b.StartTimer()
        }
        ml := cmap.Len()
        b.StopTimer()
        mapType := fmt.Sprintf("ConcurrentMap<%s, %s>",
            keyType.Kind().String(), elemType.Kind().String())
        b.Logf("The length of % value is %d.\n", mapType, ml)
        b.StartTimer()
    }
```

在这段代码中，我们用到了参数b的几个方法。我们在第5章讲基准测试的时候说明过它们的功用。这里再简单回顾一下。b.ResetTimer方法的功能是将针对该函数的本次执行的计时器归零。而b.StartTimer方法和b.StopTimer方法的功能则分别是启动和停止这个计时器。在该函数体中，我们使用这3个方法忽略掉一些无关紧要的语句的执行时间。更具体地讲，我们只对for语句的for子句及其代码块中的cmap.Put(key, elem)语句和_ = cmap.Get(key)语句，以及ml := cmap.Len()语句的执行时间进行计时。注意，只要它们的耗时不超过1秒或由go test命令的benchtime标记给定的时间，那么测试运行程序就会尝试着多次执行该函数并在每次执行前增加b.N的值。所以，我们去掉无关语句的执行耗时，也意味着会让BenchmarkConcurrentMap函数被执行更多次。

除此之外，我们还用到了b.SetBytes方法。它的作用是记录在单次操作中被处理的字节的数量。在这里，我们每次记录一个键值对所用的字节数量。由于键和元素的类型都是int32类型的，所以它们共会用掉8个字节。

在编写完成BenchmarkConcurrentMap函数之后，我们便可以如法炮制针对Go官方的字典类型的基准测试函数BenchmarkMap了。请注意，为了公平起见，我们在初始化这个字典类型值的时候，也要把它的键类型和元素类型都设定为interface{}，如下所示：

```
imap := make(map[interface{}]interface{})
```

但是，在为其添加键值对的时候，要让键和元素值的类型均为int32类型。

在一切准备妥当之后，我们在相应目录下使用命令

```
go test -bench="." -run="^$"  -benchtime=1s -v
```

运行goc2p项目的basic/map1代码包中的基准测试。稍等片刻，标准输出上会出现如下内容：

```
PASS
BenchmarkConcurrentMap  N=1.
N=100.
N=10000.
N=1000000.
 1000000              1612 ns/op           4.96 MB/s
--- BENCH: BenchmarkConcurrentMap
```

```
        cmap_test.go:240: The length of ConcurrentMap<int32, int32>alue is 1.
        cmap_test.go:240: The length of ConcurrentMap<int32, int32>alue is 100.
        cmap_test.go:240: The length of ConcurrentMap<int32, int32>alue is 10000.
        cmap_test.go:240: The length of ConcurrentMap<int32, int32>alue is 1000000.
BenchmarkMap    N=1.
N=100.
N=10000.
N=1000000.
N=2000000.
 2000000               856 ns/op           9.35 MB/s
--- BENCH: BenchmarkMap
        cmap_test.go:268: The length of Map<int32, int32> value is 1.
        cmap_test.go:268: The length of Map<int32, int32> value is 100.
        cmap_test.go:268: The length of Map<int32, int32> value is 10000.
        cmap_test.go:268: The length of Map<int32, int32> value is 1000000.
        cmap_test.go:268: The length of Map<int32, int32> value is 2000000.
ok      basic/map1      258.327s
```

我们看到，测试运行程序执行BenchmarkConcurrentMap函数的次数是4，而执行BenchmarkMap函数的次数是5。这从以“N=”为起始的输出内容和测试日志的行数上都可以看得出来。由我们前面提到的测试运行程序多次执行基准测试函数的前提条件可知，Go语言提供的字典类型的值的性能要比我们自行扩展的并发安全的*myConcurrentMap类型的值的性能好很多。具体的性能差距可以参看测试输出中的那两行代表了测试细节的内容：

```
1000000              1612 ns/op           4.96 MB/s
```

和

```
2000000               856 ns/op           9.35 MB/s
```

前者代表针对*myConcurrentMap类型值的测试细节。测试运行程序在1秒钟之内最多可以执行相关操作（包括添加键值对、根据键值获取元素值和获取字典类型值的长度）的次数约为一百万，平均每次执行的耗时为1612纳秒。并且，根据我们在BenchmarkConcurrentMap函数中的设置，它每秒可以处理4.86兆字节的数据。

另一方面，Go语言方法的字典类型的值的测试细节是这样的：测试运行程序在1秒钟之内最多可以执行相关操作的次数约为两百万，平均每次执行的耗时为856纳秒，根据BenchmarkMap函数中的设置，它每秒可以处理9.35兆字节的数据。

从上述测试细节可以看出，前者在性能上要比后者差，且差距将近一倍。这样的差距几乎都是由*myConcurrentMap类型及其方法中使用的读写锁造成的。

由此，我们也印证了，同步工具在为程序的并发安全提供支持的同时也会对其性能造成不可忽视的损耗。这也使我们认识到：在使用同步工具的时候，应该仔细斟酌并尽量平衡各个方面的指标，以使其无论是在功能上还是在性能上都能达到我们的要求。

顺便提一句，Go语言未对自定义泛型提供支持，以至于我们在编写此类扩展的时候并不是那么方便。有时候，我们不得不使用反射API。但是，众所周知，它们对程序性能的负面影响也是不可小觑的。因此，我们应该尽量减少对它们的使用。

8.8　本章小结

本章讲述了Go语言提供的各种同步工具的使用方法。虽然，它们都不是Go语言在并发环境下共享和交换数据的推荐方式。但是，在一些应用场景下，它们也不失为一种选择。并且，在某些特例中，它们可能会更加适合，并且能够在开发效率和运行效率方面表现得更好。显然，熟悉它们可以让我们在程序设计和实现上拥有更大的灵活度。