

TURING

图灵原创

Go

并发编程 实战

郝林◎著



人民邮电出版社
POSTS & TELECOM PRESS

“《Go并发编程实战》这本书对Go语言并发编程的探讨之深入、讲解之细腻是它的一大亮点。同时，这本书也非常适合作为Go语言的入门教材，即便是对Go语言了解不深甚至从未接触的人也能从中获益。另外，书中的示例也非常有价值，它们贴切地展现了用Go语言进行编程的方法和技巧。总之，《Go并发编程实战》是一份难得的Go语言学习资料。”

——许式伟，七牛云存储 CEO

“Go语言作为优秀的开源编程语言，已逐渐成为云计算时代的必学语言之一。《Go并发编程实战》不但对基本的Go语言编程方法和技巧进行了深入的阐释，还独树一帜地对Go语言的内部机制和原理进行了清晰的描述。这些都是学好和用好Go语言的极佳资料。推荐Go语言爱好者以及对Go语言感兴趣的技术人员都阅读这本书。”

——杜玉杰，中国OpenStack社区（COSUG）发起人，OpenStack
基金会董事，企业级云计算联盟（ECA）副秘书长

“Go语言是服务端编程领域非常热门的语言，市面上关于Go语言的图书并不少见，但都没有像《Go并发编程实战》这样，把Go语言最精髓的部分——并发编程讲解得如此深入浅出，明白透彻。……不管你是第一次接触Go语言，还是已经非常熟悉它了，如果想了解Go语言更多的技术内幕，这本书都值得仔细阅读，相信读者能够从中受益匪浅。”

——郭理靖，京东云平台开放云事业部总监

“郝林的这本书非常注重对Go编程细节的清晰阐释，这在国内原创技术书中是不多见的。书中示例选取精心得当，深入浅出，完全没有那种看完仍需要参悟的感觉。这是一本Go学习者真正需要的图书。”

——程显峰，蓝海讯通COO，《MongoDB权威指南》译者

“在《Go并发编程实战》一书中，作者由浅入深地对Go并发技术进行了剖析，并辅以翔实的案例，让读者真正了解和掌握Go并发编程，成为多核时代和云计算时代的开发尖兵。”

——陈佳桦，51CTO签约讲师，Gogs项目创始人

图灵社区：iTuring.cn

热线：(010) 51095186 转 600

分类建议 计算机/程序设计/Go语言

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-37398-4



9 787115 373984 >

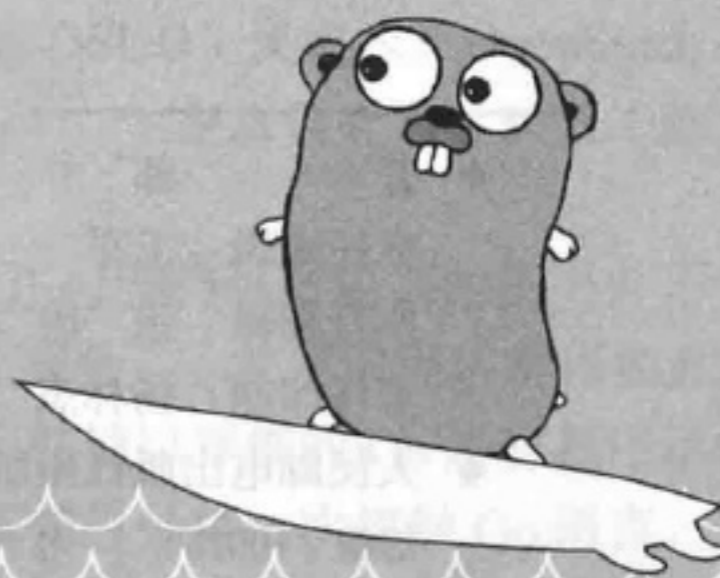
ISBN 978-7-115-37398-4

定价：89.00元

Go

并发编程实战

郝林◎著



人民邮电出版社
北京

图书在版编目 (C I P) 数据

Go并发编程实战 / 郝林著. -- 北京 : 人民邮电出版社, 2015.1
(图灵原创)
ISBN 978-7-115-37398-4

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

中国版本图书馆CIP数据核字(2014)第246517号

内 容 提 要

本书全面介绍了 Go 语言的特点、安装部署环境、工程规范、工具链、语言语法、并发编程模型以及在多个编程实战中的应用,重点阐述了 Go 语言并发编程模型和机制。本书共分为四个部分,介绍了 Go 语言编程环境搭建、Go 语言基础编程、Go 语言并发编程方法及其原理,以及使用 Go 语言开发的应用系统的案例讲解。

本书适用于有一定计算机编程基础的从业者以及对 Go 语言编程感兴趣的爱好者,非常适合作为 Go 语言编程进阶教程。

-
- ◆ 著 郝 林
责任编辑 王军花
执行编辑 张 霞
责任印制 杨林杰
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
三河市海波印务有限公司印刷
 - ◆ 开本: 800×1000 1/16
印张: 35.75
字数: 845千字
印数: 1-3 000册
 - 2015年1月第1版
2015年1月河北第1次印刷
-

定价: 89.00元

读者服务热线: (010)51095186转600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京崇工商广字第 0021 号

专家推荐

“并发编程的支持无疑是 Go 语言最大的亮点。但是，尽管 Go 语言大幅降低了并发编程的门槛，但至今大部分开发者对如何运用该语言编写高并发程序的认知仍然有限。我很高兴能有一本专门探讨 Go 语言并发编程的书。《Go 并发编程实战》这本书对 Go 语言并发编程的探讨之深入、讲解之细腻是它的一大亮点。同时，这本书也非常适合作为 Go 语言的入门教材，即便是对 Go 语言了解不深甚至从未接触的人也能从中获益。另外，书中的示例也非常有价值，它们贴切地展现了用 Go 语言进行编程的方法和技巧。总之，《Go 并发编程实战》是一份难得的 Go 语言学习资料。”

——许式伟，七牛云存储 CEO

“Go 语言作为优秀的开源编程语言，已逐渐成为云计算时代的必学语言之一。《Go 并发编程实战》不但对基本的 Go 语言编程方法和技巧进行了深入的阐释，还独树一帜地对 Go 语言的内部机制和原理进行了清晰的描述。这些都是学好和用好 Go 语言的极佳资料。推荐 Go 语言爱好者以及对 Go 语言感兴趣的技术人员都阅读这本书。”

——杜玉杰，中国 OpenStack 社区（COSUG）发起人，OpenStack 基金会董事，
企业级云计算联盟（ECA）副秘书长

“Go 语言是服务端编程领域非常热门的语言，市面上关于 Go 语言的图书并不少见，但都没有像《Go 并发编程实战》这样，把 Go 语言最精髓的部分——并发编程讲解得如此深入浅出，明白透彻。本书系统地梳理了并发编程的概念和原理，并辅以详细的 Go 语言程序示例，非常容易让读者对并发、线程、信号等概念有清晰的理解。不管你是第一次接触 Go 语言，还是已经非常熟悉它了，如果想了解 Go 语言更多的技术内幕，这本书都值得仔细研读，相信读者能够从中受益匪浅。”

——郭理靖，京东云平台开放云事业部总监

“Go 语言从诞生之日起就充满了争议，但是社区对它的热情却越来越高，今年 InfoWorld 最佳的开源项目 Docker 就是用 Go 写的，可见 Go 闪烁着越来越耀眼的光芒。郝林的这本书非常注重对 Go 编程细节的清晰阐释，这在国内原创技术书中是不多见的。书中示例选取精心得当，深入浅出，完全没有那种看完仍需要参悟的感觉。这是一本 Go 学习者真正需要的图书。”

——程显峰，蓝海讯通 COO，《MongoDB 权威指南》译者

“Go 语言之所以被称为 21 世纪的 C 语言，不仅在于它精简的语法和高效的开发，更在于它具有原生支持和易于使用的高并发的特性。而越是简单的技术就越能够生成千变万化的组合，想要用到极致，需要对它有深刻的见解。在《Go 并发编程实战》一书中，作者由浅入深地对 Go 并发技术进行了剖析，并辅以翔实的案例，让读者真正了解和掌握 Go 并发编程，成为多核时代和云计算时代的开发尖兵。”

——陈佳桦，Go 语言签约讲师，Gogs 项目创始人

推 荐 序

我很幸运，在三年前就开始接触 Go 语言。由于那时候资料匮乏，我基本上是通过读取官方的源码包学习过来的。那个时候官方有一个三天入门系列，基本上花几个小时就可以完成 Go 语言的入门，后面就是靠自己在 Go 源码包中不断地深入学习。由于我之前是 Web 开发者，所以我就从本职工作出发写了《Go Web 编程》一书。它主要介绍了 Go 语言如何与 Web 开发结合起来，只花了很小的篇幅去介绍 Go 语言本身。至今已经过去了两年多的时间。在之后的这段时间里，我自己也在不断思考，是不是需要再写一本实战类的图书介绍 Go 语言。让我很惊讶和兴奋的是，郝林赶在我之前写出了这本《Go 并发编程实战》。

郝林的《Go 并发编程实战》一书不仅清楚地解释了 Go 语言的各个知识点，而且包含了很多案例和高层次的解读，还阐述了很多软件工程方面的设计和开发技巧。这本书最大的看点在于并发编程，这也是 Go 语言最大的特色。作者花了大量篇幅详细介绍了 Go 并发编程的核心要素——Goroutine 和 Channel 的概念、原理、基本用法和高级技巧，以及编写并发程序的过程中对各种同步工具的运用等问题。在讲述这些知识的过程中，作者还展示和细致讲解了各种各样的代码实例，尤其是包含了像载荷发生器和网络爬虫框架这样的实用程序。这对读者真正理解上述知识点是非常有帮助的。从基础知识到高级应用，再到实战，作者如此缜密的构思真是完美啊。这本书不仅让你知其然，而且还让你知其所以然。在我通读全书之后，不禁感慨作者的阐述是如此地全面和夯实。因此，这本书不仅适合新手入门 Go 语言，而且让我们这样有几年 Go 语言编程经验的读者也受益匪浅。

这本书围绕着并发编程做了大量的介绍，但前几章的编程基础也介绍得相当详细。我们知道，Go 语言的语法很简洁，关键字只有 25 个，但是表达能力超强。作者通过 5 个章节把 Go 语言的语言细节介绍得非常清楚，而且还介绍了很多底层的实现细节，贯穿于语言层面和源码层面，从而让读者对 Go 语言的实现有更加深刻的理解。我们学习的过程一般都是先进行基础知识的学习，接着开始动手写代码。这本书就是按这样的顺序编写的，从知识学习到实战应用的构思非常好，也让我学习到了很多知识，非常感谢作者能写出这样的一本书来。如果在三年前就有这样一本书的话，我相信在它的帮助下我可以更加深入地理解 Go 语言，而不需要天天深埋在源码中，研究各种设计思路和语言细节，这样也许我就有更多时间去写出更多的开源项目。作为一名 Go 语言程序员，我们需要不断地深入学习 Go 的各种细节，这样才能使用它编写出正确和高效的程序。而此书对于 Go 语言细节的讲解非常透彻，还通过各种实例演示了其使用方法及开发技巧。这是一本深度和广度俱佳的 Go 语言的实战图书。我在此郑重推荐给每一位学习 Go 语言的读者。

建议读者不仅要细读这本书，而且还要深入理解作者给出的一些实例，这样才能掌握 Go 语言的设计思想。

最后，感谢郝林邀请我写作这篇推荐序，能够为这样一本书写序，是我莫大的荣幸。

谢孟军

2014 年 10 月 20 日

前言

很高兴你能选择这本书。希望这本书能够让你成为 Go 粉。很多 Go 语言爱好者都喜欢称自己是 Gopher。

Go 编程语言（或称 Golang，以下简称 Go 语言）是云计算时代的 C 语言。它的诞生是为了让程序员有更高的生产效率，并让程序在有并行计算支持的环境下更快速地运行。2009 年 11 月 10 日，Go 语言正式成为开源编程语言大家庭的一员。在本书截稿之时，它的最新版本是 1.3。因此，本书的内容和相关代码将会围绕 Go 语言的 1.3 版本展开。

这里所说的开源是指开放源代码。开源不仅仅意味着分享和免费。对于软件开发人员来讲，它更多的是代表着一种协同工作的方式。它可以使不同公司、不同城市甚至不同国家的技术爱好者们聚集起来，以共同分享和使用源代码的方式协同完成一个任务或目标。开源的意义并不在于目标大小，而在于对某个技术领域、某个应用行业甚至整个产业的启示和推动力。开源的编程语言会更多、更好地凝聚众人的力量，并使它更快地奔向成功。

Go 语言每半年就会发布一个大版本升级（比如从 1.1 升级到 1.2、从 1.2 升级到 1.3，等等）。这样的更新速度是很多其他编程语言望尘莫及的。其中，开源无疑起到了很大的推动作用。

Go 语言虽然在 2009 年才真正成为开源编程语言，在 2012 年才发布第一个正式版——1.0 版本，但是它很早就进入了中国软件开发者的视野。这不单单是因为它是由几位教父级软件开发先驱开发、由 Google 公司发布的开源编程语言，更是因为它是一门拥有诸多先进特性、拥有高性能和生产效率、为云计算时代和软件工程而生的多范式编程语言。Go 语言虽然年轻，但已经足以用于软件生产。

目前来看，Go 语言在中国发展的最大不足就是其开源社区非常松散，爱好者不论是在地理位置上还是在互网络中都比较分散。当然，这也是由于 Go 语言的年轻所导致的。目前，在中国没有一个可以主导语言爱好者学习和交流的 Go 语言社区，语言爱好者的数量也相对较少。它不像 Python 那样，在中国有一个统一的、分支众多的语言爱好者联盟，Python 语言爱好者可以很轻易地找到志同道合的组织和同行，并积极地进行分享和交流。Go 语言在这方面有所缺失的很大一部分原因是在中国使用 Go 语言的软件开发者太少。因此，我非常希望能通过本书使更多的同行了解和使用 Go 语言，并用它来创造更多的价值。我也希望大家能够协力共建中国的 Go 语言技术社区。

通过本书，我会带领大家进入 Go 语言的世界，领略 Go 语言的魅力，为大家打开一扇门，也力求使大家对使用 Go 语言开发软件产生更多的兴趣。

本书结构

本书共分为四个部分。

第一部分，将会带领你进入 Go 语言的世界，让你对 Go 语言有个基本的了解，这一部分包含两章。

第 1 章，会简要介绍 Go 语言为大家带来的优秀特性和新鲜活力，以及与其他主流编程语言相比的优缺点。

第 2 章，将对编写和运行 Go 语言程序所需的所有前期工作进行说明。这包括安装和配置方法、工程结构以及标准命令。通过对这些知识的学习和实践，相信读者就可以为编写 Go 代码和建立 Go 项目做好准备了。

第二部分，会详细阐述 Go 语言的基础编程知识，这一部分包含了 3 章。

第 3 章，首先会介绍 Go 语言中的关键字、运算符、类型、表达式等最基本的概念。掌握这些知识，读者可以了解到 Go 语言在代码构建方面的言简意赅。然后，展示 Go 语言基本数据结构及其主要操作方法和技巧。Go 语言的基本数据结构的操作方式是非常脚本化的，使用起来也非常简单和方便。最后，我将加入一些相对高级一点的主题，比如基本数据结构的初始化方式、可比性与有序性，以及类型转换方式等，从中我们会看到 Go 语言的设计者们在灵活性和简洁性之间做出的取舍。

第 4 章，我们一起来学习 Go 语言的流程控制方法。这些方法已经足以体现出 Go 语言的先进和强大（当然，我们在后面还会看到更棒的特性）。这一章不但会介绍 Go 语言中基本的逻辑控制方法，还会深入阐述 Go 语言特有的程序编写方式，以及这些特有方式所体现出的 Go 语言程序设计哲学。

第 5 章，将详细介绍 Go 语言程序的测试和文档的编写。

第三部分，将集中笔墨讲解 Go 语言中最强悍的部分——并发编程的概念和知识，这一部分同样包含了 3 章的内容。

第 6 章，先简要介绍与多进程编程和多线程编程有关的知识。这些知识会作为理解 Go 语言并发编程模型的先导内容展现给大家。看过以后，大家应该可以对并发编程有一个比较清晰的理解。同时，还会指出以内存共享为基础的并发编程方式所带来的一些问题。最后，通过剖析多核时代（基于多 CPU 核心的计算时代）的并发编程需求，引出 Go 语言先进的并发编程模型。大家会看到 Go 语言为之做出的努力。

第 7 章，会向大家展示 Go 特有的编程要素——Goroutine（也可称为 Go 程）的魅力。这一章不但会介绍怎样利用 Goroutine 编写可并发运行的程序，还会深入阐述 Goroutine 背后的运作机理和执行过程，使读者知其然更知其所以然。此外，还会介绍标准库的 `runtime` 代码包中与 Goroutine 相关的一些 API。这一章的另一个重点是对 Go 语言并发编程中不可或缺的部分——Channel 的介绍。Channel 与 Goroutine 堪称绝配，它的用法相当灵活。这一章在介绍了 Channel 的基本概念和使用规则之后，还会介绍多种 Channel 的实际用法，并通过丰富的示例来说明每种使用方法和技巧。最后，还描述了标准库的 `time` 代码包中的 API 是如何通过 Channel 来实现其

强大功能的。

第 8 章，将是对 Go 语言提供的传统同步和互斥方法的介绍。这包括锁、条件变量、原子操作、WaitGroup、临时对象池，等等。这些同步方法的提供者大都是标准库的 sync 代码包。虽然 Go 语言官方并不推荐使用这些方式来控制和协调 Go 语言编写的并发程序，但不容忽视的是，它们确实有一些应用场景中是简单有效的。

第四部分仅包含一章，即第 9 章。

第 9 章，包括了一个基本囊括本书前三个部分所涉及的全部概念和知识的完整示例。我会带领大家一步一步地编写这个示例。在这个过程中，我会进一步阐述 Go 语言的哲学和理念，以及我在多年编程生涯中的一些见解和感悟。大家可以通过对这个示例的学习来巩固我们之前讲到的 Go 语言知识，并加深对 Go 语言并发编程的理解。

在本书的附录中，将向大家简单介绍目前在国内外比较活跃的一部分 Go 语言开源项目和 Go 语言社区。这会使大家学习 Go 语言的道路变得更加顺畅，也有利于大家找到志同道合的朋友。

目标读者

原则上来讲，任何对计算机编程和 Go 语言感兴趣的人都可以阅读本书。但是，当你学习一门编程语言的时候，往往还是需要有些许基础的。比如，怎样使用文本编辑器，怎样在相应的操作系统中安装软件，等等。并且，要想成为一名高级的 Go 软件工程师，你需要了解的周边知识可能在数量上会比 Go 语言编程本身多出几倍甚至几十倍。这就像摘苹果一样，如果要摘到苹果，就需要徒手爬上果树，或者找到工具帮助你；如果想摘到果树顶端最甜的那个苹果，就需要花费更多的时间和精力，爬过更多的枝叶。希望本书能成为帮助你摘苹果的工具。但是，你还是需要先活动活动手脚的，毕竟最后享用苹果的是你而不是梯子或者其他什么东西。在想有所收获之前，请先潜心地学习和积累。

我在这里列出了一些你可能需要爬过的枝叶。

- ❑ 知道计算机系统是什么，以及它的核心是由哪些部件组成的。比如，CPU 是什么？它是做什么用的？
- ❑ 知道软件是什么，以及它们是怎么设计和开发出来的。比如，操作系统和网络浏览器都是软件吗？
- ❑ 了解一些基本的 Linux 操作系统知识。比如，在命令行（也被称为终端）环境中怎样进入一个目录？怎样查看一个文本文件？
- ❑ 使用过一两门编程语言，真正编写过程序或软件。当然，没有也没关系，因为 Go 语言很适合作为计算机编程的入门语言。

当然，以上这些仅供参考。你在阅读本书的过程中边看边学也完全没问题，甚至可以看完本书再去学习相关知识。采用哪种学习方式完全取决于你自己。

关于示例代码

我会把本书涉及的所有 Go 示例代码（也许不只，可能会有小惊喜）都放到一个名为 goc2p 的项目中。读者可以访问 <https://github.com/hyper-carrot/goc2p> 下载它。该项目的根目录的路径应该被包含在环境变量 GOPATH 中。如果你不了解 Git（一款代码版本控制工具），请在互联网中搜索“git”并获取相关信息。

关于勘误

由于时间水平都比较有限，所以书中难免会出现一些纰漏和错误。如果大家发现了一些问题，请及时指正。我也会尽快在本书后续的版本中加以改正。我专为本书设立的电子邮箱是：hypermind.cn@gmail.com。我的微博是：特价萝卜。我欢迎也希望和大家一起学习和讨论 Go 语言，并共同推动 Go 语言中国社区的发展。

致谢

撰写图书是一项需要大量精力和一定毅力的工作，尤其是编写技术图书，更需要作者对相关知识进行深入的梳理和系统的整合，还需要制作各种图表，编写各种示例。对于个人而言，工作量确实是不小的。但是，这个写作过程也是有趣的，通过写作我也收获了很多。当然，很多收获是来自他人的传授。其中，图灵公司的编辑王军花、张霞以及傅志红老师都给予了我很大的帮助，尤其是在写作技巧和图书结构方面。此外，还有目前中国业内公认的 Go 语言专家许式伟、谢孟军等，我在编写本书的时候经常向他们讨教。在此，我对这些帮助过我的专家和同行表示由衷的感谢。同时，我也要感谢我的家人。没有他们的支持和理解，我不可能在有限的业余时间里完成这本书。

目 录

第一部分 Go 语言的世界

第 1 章 初识 Go 语言	2
1.1 Go 语言特性一瞥	2
1.2 Go 语言的优劣	3
1.3 怎样学习 Go 语言	4
1.4 本章小结	5
第 2 章 Go 语言环境搭建	6
2.1 安装和设置	6
2.1.1 Linux	6
2.1.2 Windows	9
2.2 工程结构	10
2.2.1 工作区	10
2.2.2 GOPATH	11
2.2.3 源码文件	11
2.2.4 代码包	14
2.3 标准命令概述	17
2.4 本章小结	18

第二部分 编程基础

第 3 章 词法与数据类型	20
3.1 基本词法	20
3.1.1 标识符	21
3.1.2 关键字	22
3.1.3 字面量	23
3.1.4 类型	24
3.1.5 操作符	26
3.1.6 表达式	33
3.2 数据类型	43

3.2.1 基本数据类型	44
3.2.2 数组	48
3.2.3 切片	52
3.2.4 字典	61
3.2.5 函数和方法	64
3.2.6 接口	72
3.2.7 结构体	76
3.2.8 指针	84
3.2.9 数据初始化	87
3.3 数据的使用	90
3.3.1 赋值语句	90
3.3.2 常量与变量	93
3.3.3 可比性与有序性	101
3.3.4 类型转换	108
3.3.5 内建函数	114
3.4 本章小结	118

第 4 章 流程控制方法	119
4.1 基本流程控制	119
4.1.1 代码块和作用域	119
4.1.2 if 语句	121
4.1.3 switch 语句	124
4.1.4 for 语句	129
4.1.5 goto 语句	137
4.2 defer 语句	141
4.3 异常处理	145
4.3.1 error	146
4.3.2 panic 和 recover	149
4.4 实战演练——Set	154
4.5 实战演练——Ordered Map	163
4.6 本章小结	173

第5章 程序测试和文档	174
5.1 程序测试	174
5.1.1 功能测试	174
5.1.2 基准测试	180
5.1.3 样本测试	187
5.1.4 测试运行记录	189
5.1.5 测试覆盖率	193
5.2 程序文档	201
5.3 本章小结	205

第三部分 并发编程

第6章 并发编程综述	208
6.1 并发编程基础	208
6.1.1 串行程序与并发程序	209
6.1.2 并发程序与并行程序	209
6.1.3 并发程序与并发系统	210
6.1.4 并发程序的不确定性	210
6.1.5 并发程序内部的交互	210
6.2 多进程编程	211
6.2.1 进程	211
6.2.2 关于同步	217
6.2.3 管道	222
6.2.4 信号	228
6.2.5 Socket	238
6.3 多线程编程	260
6.3.1 线程	261
6.3.2 线程的同步	268
6.4 多线程与多进程	285
6.5 多核时代的并发编程	286
6.6 Go语言的并发编程	290
6.6.1 线程实现模型	290
6.6.2 调度器	299
6.6.3 更多的细节	311
6.7 本章小结	315
第7章 Goroutine 和 Channel	316
7.1 Goroutine 的使用	316
7.1.1 go 语句与 Goroutine	316

7.1.2 Goroutine 的运作过程	321
7.1.3 runtime 包与 Goroutine	322
7.1.4 Happens Before	326
7.2 Channel	327
7.2.1 Channel 是什么	328
7.2.2 单向 Channel	335
7.2.3 for 语句与 Channel	342
7.2.4 select 语句	344
7.2.5 非缓冲的 Channel	352
7.2.6 time 包与 Channel	358
7.3 实战演练——载荷发生器	363
7.3.1 参数和结果	364
7.3.2 基本结构	365
7.3.3 初始化	369
7.3.4 启动和停止	376
7.3.5 调用器和功能测试	389
7.4 本章小结	401

第8章 同步	402
8.1 锁的使用	402
8.2 条件变量	411
8.3 原子操作	414
8.4 只会执行一次	420
8.5 WaitGroup	423
8.6 临时对象池	426
8.7 实战演练——Concurrent Map	429
8.8 本章小结	436

第四部分 编程实战

第9章 一个网络爬虫框架的设计和实现	438
9.1 网络爬虫与框架	438
9.2 功能需求和分析	440
9.3 总体设计	441
9.4 详细设计	443
9.4.1 基本数据结构	443
9.4.2 接口的设计	449
9.5 中间件的实现	459

9.5.1 通道管理器	460	9.7.3 请求缓存	521
9.5.2 实体池	470	9.7.4 摘要信息的类型	524
9.5.3 停止信号	477	9.8 一个使用演示	530
9.5.4 ID 生成器	480	9.8.1 再看调度器参数	530
9.6 处理模块的实现	482	9.8.2 开启调度器	535
9.6.1 网页下载器	483	9.8.3 调度器监控函数	542
9.6.2 分析器	488	9.9 当前的不足和解决思路	552
9.6.3 条目处理管道	494	9.10 本章小结	555
9.7 调度器的实现	498	附录 Go 语言的学习资源	557
9.7.1 基本结构	499		
9.7.2 主要的函数和方法	502		

Part 1

第一部分

Go 语言的世界

作为本书的第一部分，我会先带领你从宏观上粗略地了解 Go 语言的一些特点，其中包括它与其他编程语言相比的优势。当然，我的描述会很客观。不过，如果你有不同意见尽可以和我探讨。

在经过一些概述之后，我们会一起来为编写 Go 语言程序做准备。这包括在不同的操作系统上安装和配置 Go 语言（别担心，这很简单），了解 Go 语言的工程结构和标准命令。

与 C++、Java、C# 等一些主流的编译型通用编程语言相比，学习 Go 语言的门槛并不高。不过，Go 语言确实也有它自己的一些规则。只要我们遵循了这些规则，就可以非常畅快地编写和运行 Go 语言程序了。稍后你就会了解到这一点。下面，就让我们一起快步进入 Go 语言的世界。

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。

想要操作计算机，就必须获得操作系统的支持。同样，想要用Go语言编写程序，就必须掌握Go语言编译器及相关工具的知识。本章将详细介绍编写Go程序所必需的软件和工具的获取方式、安装步骤和使用方法。

2.1 安装和设置

本节所安装和配置的对象是Go语言编译器及其工具。

Go语言项目的下载地址是<http://golang.org/dl/>，其中包含了针对不同平台的二进制安装包。

如果你的系统是FreeBSD 7及更高版本、Linux 2.6.23及更高版本（需带有glibc库）、Mac OS X 10.6及更高版本（有时需要有Xcode软件），或Windows 2000及更高版本，那么恭喜，你的系统符合Go语言开发的最低要求，赶紧动手安装吧！

这里将只以Ubuntu 12.10 32bit和Windows 7 64bit系统为例进行介绍。Go语言的安装和配置过程都非常简单，读者应该可以很容易地举一反三，找出适合自己系统的一套方法。

2.1.1 Linux

本节我们将以Ubuntu 12.10 32bit为例，介绍Go语言的安装和配置。

首先，从前述地址中下载名为go1.3.linux-386.tar.gz的文件：

```
hc@ubt:~$ mkdir ~/package
hc@ubt:~$ cd ~/package
hc@ubt:~/package$ wget http://golang.org/dl/go1.3.linux-386.tar.gz
```

这里建立的~/package目录，即用户home目录下的package。进入该目录后，使用wget命令通过HTTP/HTTPS协议来下载目标文件。当然，只要拥有读写权限，用户可自行决定将文件下载到哪个目录。

接下来，解压安装文件：

```
hc@ubt:~/package$ tar xzf go1.3.linux-386.tar.gz
```

tar命令用于将多个文件打包为一个存档文件，以及将存档文件解包。

zxf是tar命令的参数。参数z表示在进行tar解包之前，首先进行gzip解包，因为目标文件不仅是一个tar文件，更是一个用gzip压缩过的文件（这类文件一般会有“.gz”后缀）。参数x表示当前需要tar命令来解包，而非打包。参数f用来指定要操作的tar存档文件。另外，如果想在tar命令执行过程中查看被处理的文件列表，则可加入参数v。

待目标文件解压后，我们可在~/package目录下发现新增的“go”子目录。该目录中包含多个文件夹和文件，下面对其中比较重要的文件夹进行简要说明。

- ❑ api文件夹：存放Go API检查器的辅助文件。其中，gol.1.txt、gol.2.txt、gol.3.txt和gol.txt文件分别罗列了不同版本的Go语言的全部API特征；except.txt文件中罗列了一些（在不破坏兼容性的前提下）可能会消失的API特性；next.txt文件则列出了可能在下一个版本中添加的新API特性。
- ❑ bin文件夹：存放所有由官方提供的Go语言相关工具的可执行文件。默认情况下，该目录会包含go、godoc和gofmt这3个工具。本书将在2.3节中给出这3个工具的简单介绍，并给出免费的Go命令教程的网址。
- ❑ doc文件夹：存放Go语言几乎全部的HTML格式的官方文档和说明，方便开发者在离线时查看。
- ❑ misc文件夹：存放各类编辑器或IDE（集成开发环境）软件的插件，辅助它们查看和编写Go代码。有经验的软件开发者定会在该文件夹中看到很多熟悉的工具。
- ❑ pkg文件夹：用于在构建安装后，保存Go语言标准库的所有归档文件。pkg文件夹包含一个与Go安装平台相关的子目录，我们称之为“平台相关目录”。例如，在针对Linux 32bit操作系统的二进制安装包中，平台相关目录的名字就是linux_386；而在针对Windows 64bit操作系统的安装包中，平台相关目录的名字则为windows_amd64。

Go源码文件对应于以“.a”为结尾的归档文件，它们就存储在pkg文件夹下的平台相关目录中。

值得一提的是，pkg文件夹下有一个名叫tool的子文件夹，该子文件夹下也有一个平台相关目录，其中存放了很多可执行文件。关于这些可执行文件的用途，读者可参见附属于本书的Go命令教程。2.3节提供了该教程的存放地址。

- ❑ src文件夹：存放所有标准库、Go语言工具，以及相关底层库（C语言实现）的源码。通过查看这个文件夹，可以了解到Go语言的方方面面。本书的后续章节会适时地对其中的部分文件进行说明。
- ❑ test文件夹：存放测试Go语言自身代码的文件。通过阅读这些测试文件，可大致了解Go语言的一些特性和使用方法。

浏览过go文件夹后，继续安装Go语言，我们需要把go文件夹放到指定的目录中去。Go语言官方建议把go文件夹复制到/usr/local目录下，但也可以将其复制到其他地方。这里，我们遵照官方的建议：

```
hc@ubt:~/package$ mv ./go /usr/local
```

其中，`mv`命令用于将目录或文件（命令中的第一个参数）移动到目标目录（命令中的第二个参数）中去。

非root用户没有对/usr/local目录的写权限（通常是这样，也强烈建议这么做），所以需要在mv命令前添加sudo命令：

```
hc@ubt:~/package$ sudo mv ./go /usr/local
```

sudo允许普通用户以超级用户（root）的身份执行命令。当然，如果想让一个用户可以使用sudo命令，还需编辑一下/etc/sudoers文件。至于具体操作方法，读者可查阅相关文档。

这里，我们只希望当前用户对go目录及其子目录或文件具有读、写和执行的权限，若希望其他用户也有相应的权限，可使用chmod命令进行设置。chmod命令的用法请参看相关文档。

接下来设置环境变量。为了使环境变量永久生效，需要编辑/etc/profile文件：

```
hc@ubt:~/package$ vim /etc/profile
```

同样，如果当前用户不是root，需要在最前端加上sudo。打开Vim的编辑界面后，使用shift + g命令将光标移至文件末尾，输入字母“i”转为插入模式。添加如下两行命令：

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

其中，export命令用于设置环境变量。第一行的作用是设置名为GOROOT的环境变量，其值将指向移动后的go目录的地址。第二行的作用是，在环境变量PATH的末尾加入Go可执行文件的目录，该目录在前文已作介绍，默认情况下，它包含go、godoc和gofmt这3个可执行文件。

设置完环境变量PATH后，保存/etc/profile文件，并退出Vim。为使上述配置立即生效，需再执行如下命令：

```
source /etc/profile
```

其中，source命令是bash shell的内置命令，用于在当前shell环境下执行脚本。

其实，Go语言还有两个隐含的环境变量——GOOS和GOARCH。

❑ GOOS代表程序构建环境的目标操作系统，可笼统地理解为Go语言安装到的那个操作系统的标识，其值可以是darwin、freebsd、linux或windows。

❑ GOARCH则代表程序构建环境的目标计算架构，可笼统地理解为Go语言安装到的那台计算机的计算架构的标识，其值可以是386、amd64或arm。

一般情况下，不用对这两个环境变量进行显式设置。在对应于不同操作系统和计算架构的Go语言安装包中，已经把这两个环境变量设置为了常量，即标准库代码包runtime中的常量GOOS和GOARCH。前面所提到的“平台相关目录”，其实就是用\${GOOS}_\${GOARCH}的方式来命名的。在本章的后面还可以看到，Go归档文件的存放路径也是根据“平台相关目录”来指定的。

至此，我们已经在Linux操作系统中安装并配置好了Go语言。下面简单测试一下安装是否成功：在命令行中输入命令go（并回车），屏幕会立即输出该命令的用法说明，如图2-1所示。

```
hc@ubt:~$ go
Go is a tool for managing Go source code.

Usage:

    go command [arguments]

The commands are:

    build      compile packages and dependencies
    clean      remove object files
    doc        run godoc on package sources
    env        print Go environment information
    fix        run go tool fix on packages
    fmt        run gofmt on package sources
    get        download and install packages and dependencies
    install    compile and install packages and dependencies
    list       list packages
    run        compile and run Go program
    test       test packages
    tool       run specified go tool
    version    print Go version
    vet        run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

    gopath     GOPATH environment variable
    packages   description of package lists
    remote     remote import path syntax
    testflag   description of testing flags
    testfunc   description of testing functions

Use "go help [topic]" for more information about that topic.

hc@ubt:~$
```

图2-1 go命令用法说明

2.1.2 Windows

本节以Windows 7 64bit为例，介绍Go语言的安装和设置，这与在Linux下安装Go极其相似。

首先，下载与Windows 64bit操作系统相对应的二进制安装文件：<http://golang.org/dl/go1.3.windows-amd64.zip>。

用任意一款压缩软件将安装文件解压到一个目录中。Go语言官方建议把go文件夹复制到C盘的根目录下，但这种绿色安装软件实在没必要跟臃肿的Windows挤在一起。因此，我们可以选择诸如D:\dev的这类目录。将go文件夹复制过去后，Go语言的根目录即为D:\dev\go。

其次，与在Linux下安装Go语言一样，需要添加系统环境变量GOROOT，并在系统环境变量PATH后追加;%GOROOT%\bin。

最后，打开命令提示符并输入go，如果出现了命令用法说明，则安装和配置成功。

2.2 工程结构

Go是一门推崇软件工程理念的编程语言，它为开发周期的每个环节都提供了完备的工具和支持。Go语言高度强调代码和项目的规范和统一，这集中体现在工程结构或者说代码体制的细节之处。

Go也是一门开放的语言，它本身就是开源软件。更重要的是，Go语言可以让程序开发者很容易地通过`go get`命令从各种公共代码库（比如著名的代码托管网站Github）中下载开源代码并使用它们。这除了得益于Go语言自带命令的强大之外，还应该归功于Go语言工程结构的严谨和完善。在本节，我们就对Go语言的工程结构进行详述。

2.2.1 工作区

Go代码必须放在工作区中。工作区其实就是一个对应于特定工程的目录，它应包含3个子目录：`src`目录、`pkg`目录和`bin`目录，下面逐一说明。

- `src`目录：用于以代码包的形式组织并保存Go源码文件。这里的代码包，与`src`下的子目录一一对应。例如，若一个源码文件被声明为属于代码包`logging`，那么它就应当被保存在`src`目录下名为`logging`的子目录中。当然，我们也可以把Go源码文件直接放于`src`目录下，但这样的Go源码文件就只能被声明为属于`main`代码包了。除非用于临时测试或演示，一般还是建议把Go源码文件放入特定的代码包中。有关Go代码包，请参考2.2.4节的内容。顺便提一句，Go语言的源码文件分为3类：Go库源码文件、Go命令源码文件和Go测试源码文件。
- `pkg`目录：用于存放经由`go install`命令构建安装后的代码包（包含Go库源码文件）的“.a”归档文件。该目录与GOROOT目录下的`pkg`功能类似。区别在于，工作区中的`pkg`目录专门用来存放用户（也就是程序开发者）代码的归档文件。构建和安装用户源码的过程一般会以代码包为单位进行，比如`logging`包被编译安装后，将生成一个名为`logging.a`的归档文件，并存放在当前工作区的`pkg`目录下的平台相关目录中。有关`go install`命令，请参考2.3节的内容。
- `bin`目录：与`pkg`目录类似，在通过`go install`命令完成安装后，保存由Go命令源码文件生成的可执行文件。在Linux操作系统下，这个可执行文件一般是一个与源码文件同名的文件。而在Windows操作系统下，这个可执行文件的名称是源码文件名称加`.exe`后缀。

注意 这里有必要明确一下Go语言的命令源码文件和库源码文件的区别。所谓命令源码文件，就是声明为属于`main`代码包，并且包含无参数声明和结果声明的`main`函数的源码文件。这类源码文件可以独立运行（使用`go run`命令），也可被`go build`或`go install`命令转换为可执行文件。而库源码文件则是指存在于某个代码包中的普通源码文件。

2.2.2 GOPATH

我们需要将工作区的目录路径添加至环境变量GOPATH中。否则，即使处于同一工作区（事实上，未被加入到环境变量GOPATH中的目录不应该被称为工作区），代码之间也无法通过绝对代码包路径完成调用。在实际开发环境中，工作区往往有多个。这些工作区的目录路径都需要添加至GOPATH。以Linux操作系统为例，若有如下两个工作区：

```
~/golang/lib
~/golang/goc2p
```

则需要修改/etc/profile文件，并加入设置环境变量GOPATH的内容：

```
export GOPATH=$HOME/golang/lib:$HOME/golang/goc2p
```

之后，保存/etc/profile文件，并用source命令使配置生效。当然，我们也可以修改当前用户的home目录下的.bash_profile文件，但是添加的内容都是一样的。

注意

- GOPATH 中不要包含环境变量 GOROOT 的值（即 Go 的安装目录路径），以此将 Go 语言本身的工作区同用户工作区严格地分开；
- 通过 Go 工具中的代码获取命令 go get，可将指定项目的源码下载到我们在环境变量 GOPATH 中设定的第一个工作区中，并在其中完成构建和安装的过程。

本书约定，在\$HOME/golang/lib目录中存放第三方代码库，而在\$HOME/golang/goc2p目录中存放本书的示例和附属代码库。

2.2.3 源码文件

本书为涉及的Go代码示例建立了项目goc2p，以求对它们进行更加规范的组织（2.2.2节已将goc2p项目的根目录路径添加到了环境变量GOPATH中）。该项目的源码文件全部存储在src子目录中，部分结构如下：

```
/home/hc/golang/goc2p/src:
basic/
  set/
    set_test.go
    set.go
cnet/
  ctcp/
    base.go
    tcp.go
    tcp_test.go
helper/
  ds/
    shows.go
```

```
logging/  
  tag.go  
  base.go  
  logger_test.go  
  console_logger.go  
  log_manager.go
```

goc2p项目的代码包中，目前仅4个包含源码文件的代码包，分别为basic/set、cnet/ctcp、helper/ds和logging。其中，代码包helper/ds中含有一个命令源码文件showds.go，可以直接通过go run命令运行。上面的目录树结构示意就是由它生成的。其余3个包只含有库源码文件和测试源码文件（文件名以“_test.go”结尾）。因此，该项目包含了Go源码文件的所有3个种类，即命令源码文件、库源码文件和测试源码文件，下面对它们进行详细说明。

1. 命令源码文件

如果一个源码文件被声明为属于main代码包，且该文件代码中包含无参数声明和结果声明的main函数，则它就是命令源码文件。命令源码文件可通过go run命令直接启动运行。

同一个代码包中的所有源码文件，其所属代码包的名称必须一致。如果命令源码文件和库源码文件处于同一个代码包中，那么在该包中就无法正确执行go build和go install命令。换句话说，这些源码文件也就无法被编译和安装。因此，命令源码文件通常会单独放在一个代码包中。这是合理的，因为一般情况下，一个程序模块或软件的启动入口只有一个。

同一个代码包中可以有多命令源码文件，可通过go run命令分别运行它们。但这种情况会使go build和go install命令无法编译和安装该代码包。所以一般情况下，也不建议把多个命令源码文件放在同一个代码包中。

当代码包中有且仅有一个命令源码文件时，在文件所在目录中执行go build命令，即可在该目录下生成一个与目录同名的可执行文件；而若使用go install命令，则可在当前工作区的bin目录下生成相应的可执行文件。例如，代码包helper/ds中只有一个源码文件showds.go，且它是命令源码文件，则相关操作和结果如下：

```
hc@ubt:~/golang/goc2p/src/helper/ds$ ls  
showds.go  
hc@ubt:~/golang/goc2p/src/helper/ds$ go build  
hc@ubt:~/golang/goc2p/src/helper/ds$ ls  
ds showds.go  
hc@ubt:~/golang/goc2p/src/helper/ds$ go install  
hc@ubt:~/golang/goc2p/src/helper/ds$ ls ../../../../bin  
ds
```

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

2. 库源码文件

通常，库源码文件声明的包名会与它实际所属的代码包（目录）名一致，且库源码文件中不包含无参数声明和无结果声明的main函数。下面来安装（包含编译过程）basic/set包，其中含

有若干库源码文件：

```
hc@ubt:~/golang/goc2p/src/basic/set$ ls
set.go  set_test.go
hc@ubt:~/golang/goc2p/src/basic/set$ go install
hc@ubt:~/golang/goc2p/src/basic/set$ ls ../../../../pkg
linux_386
hc@ubt:~/golang/goc2p/src/basic$ cd ../../../../pkg/linux_386/basic
hc@ubt:~/golang/goc2p/pkg/linux_386/basic$ ls
set.a
```

这里，我们通过基本在basic/set目录下执行go install命令，成功地安装了basic/set包，并生成一个名为set.a的归档文件。归档文件的存放目录由以下规则产生。

- ❑ 安装库源码文件时所生成的归档文件会被存放到当前工作区的pkg目录中。goc2p项目的set包所属的工作区的根目录是~/golang/goc2p。因此，上面所说的pkg目录即~/golang/goc2p/pkg。
- ❑ 根据被编译时的目标计算架构，归档文件会被放置在pkg目录下的平台相关目录中。例如，我的安装操作是在Linux 32bit环境下进行的，对应的平台相关目录就是linux_386，归档文件set.a一定会被放到~/golang/goc2p/pkg/linux_386目录中的某个地方。
- ❑ 存放归档文件的目录的相对路径与被安装代码包的上一级代码包的相对路径是一致的。第一个相对路径是相对于工作区的pkg目录下的平台相关目录而言的，而第二个相对路径是相对于工作区的src目录而言的。如果被安装代码包没有上一级代码包（也就是说它的父目录就是工作区的src目录），那么它的归档文件就会被直接存放到当前工作区的pkg目录的平台相关目录下。例如，basic包的归档文件basic.a总会被直接存放到~/golang/goc2p/pkg/linux_386目录下，而basic/set包的归档文件set.go则会被存放到~/golang/goc2p/pkg/linux_386/basic目录下。

3. 测试源码文件

测试源码文件是一种特殊的库文件，可以通过执行go test命令运行当前代码包下的所有测试源码文件。成为测试源码文件的充分条件有两个。

- ❑ 文件名需要以“_test.go”结尾。
- ❑ 文件中需要至少包含一个名称以“Test”开头或“Benchmark”开头、拥有一个类型为“testing.T”或“testing.B”的参数的函数。类型“testing.T”和“testing.B”分别对应功能测试和基准测试所需的结构体。

当我们在一个代码包中执行go test命令时，该代码包中的所有测试源码文件就会被找到并运行。我们依然用basic/set包做例子：

```
hc@ubt:~/golang/goc2p/src/basic/set$ go test
PASS
ok      basic/set  0.086s
```

我们使用go test命令在basic/set包中找到并运行了测试源码文件set_test.go，并调用了其中所有的测试函数。命令行的回显信息表示我们通过了测试，并且运行测试源码文件中的测试程序

共花费了0.086秒。

最后，还有一点需要注意，存储Go代码的文本文件需要以UTF-8编码存储。如果源码文件中出现了非UTF-8编码的字符，则在运行、编译或安装时，Go会抛出“illegal UTF-8 sequence”的错误。

因力求对源码文件的讲解的连贯性和完整性，本小节也涉及了很多关于Go语言代码包和命令的内容，但并没有立即深入探讨。下一小节，我们就对Go语言的代码包进行专门的描述。

2.2.4 代码包

对于大多数计算机编程语言来说，代码包都是组织代码最有效和最直观的方式。如前文所述，Go语言中的代码包是对代码进行构建和打包的基本单元。在本节，我们继续以goc2p项目的src目录为辅助，对代码包进行说明。

1. 包声明

细心的读者可能已经发现，在goc2p项目中，src目录的每个代码包中的源码文件名称看似都与包名没什么联系。实际上，在Go语言中，代码包中的源码文件名可以是任意的。比如，在logging包中，没有任何源码文件的名称与包名相同。这些任意名称的源码文件都必须以包声明语句作为文件中代码的第一行。比如，basic/set包中的所有源码文件都要先声明自己属于basic/set包：

```
package set
```

package是Go语言中用于包声明语句的关键字。Go语言规定包声明中的包名为代码包路径的最后一个元素。比如，basic/set包的包路径为basic/set，而包声明中的包名则为set。但有一个例外，前文提到过，不论命令源码文件存放在哪个代码包中，它都必须声明为属于main包。

2. 包导入

goc2p项目目前包含了4个存在源码文件的代码包：basic/set、helper/ds、cnet/ctcp和logging。并且，goc2p库的目录已被添加到环境变量GOPATH中。那我们怎么在其他源码文件中导入（或称依赖）它们呢？请看下面的例子：

```
import basic/set
import helper/ds
import cnet/ctcp
import logging
```

代码包的导入使用代码包导入路径。代码包导入路径就是代码包在工作区的src目录下的相对路径。比如，代码包ctcp的绝对目录路径是~/golang/goc2p/src/cnet/ctcp，而~/golang/goc2p是被包含在环境变量GOPATH中的工作区目录路径，那么其代码包导入路径就是cnet/ctcp。

当导入多个代码包时，需要用圆括号括起它们，且每个代码包名独占一行。在调用被导入代码包中的函数或使用其中的结构体、变量或常量时，需要使用包路径的最后一个元素加“.”的方式指定代码所在的包。

如果我们有两个包logging和go_lib/logging，且有一个源码文件需要导入这两个包：

```
import (
```

```

    "logging"
    "go_lib/logging"
)

```

则这句代码`logging.NewSimpleLogger()`就会引起冲突，Go语言无法知道`logging`代表的是哪一个包。所以，在Go语言中，如果在同一个源码文件中导入多个代码包，那么代码包路径的最后一个元素不可以重复。

如果用上面这段代码包导入代码，那么在编译代码时，Go语言会抛出“`logging redeclared as imported package name`”的错误。如果这种代码包确实有必要导入，那我们又该怎么做呢？当有这类重复时，我们可以给它们起个别名来区分，比如：

```

import (
    la "logging"
    lb "go_lib/logging"
)

```

如此导入之后，我们就可以调用包中的代码了：

```
var logger la.Logger = la.NewSimpleLogger()
```

这里不必给每个引起冲突的代码包都起一个别名，只要能区分它们就可以了。

如果我们想直接调用某个依赖包的程序，就可以用“.”来代替别名，如下所示：

```

import (
    . "logging"
    lb "go_lib/logging"
)

```

看到那个“.”了吗？之后，在当前源码文件中，我们就可以直接进行代码调用了：

```
var logger Logger = NewSimpleLogger()
```

细心的读者可能观察到函数`NewSimpleLogger()`的名称首字母是大写的。这好像与其他编程语言的编码规范不太一致。

Go语言把变量、常量、函数、结构体和接口统称为程序实体，而把它们的名字统称为标识符。标识符可以是任何Unicode编码可以表示的字母字符、数字以及下划线“_”。并且，首字母不能是数字或下划线。

实际上，标识符的首字母的大小写控制着对应程序实体的访问权限。如果标识符的首字母是大写的，那么它所对应的程序实体就可以被本代码包之外的代码访问到，也可以称其为可导出的。否则，对应的程序实体就只能被本包内的代码访问。当然，还有以下两个额外条件。

- 程序实体必须是非局部的。局部程序实体的意思是，它被定义在了函数或结构体的内部。
- 代码包所在的目录必须被包含在环境变量GOPATH中的工作区目录中。

举个例子，如果代码包`logging`中有一个叫作`getSimpleLogger`的函数。那么光从这个函数的名字上我们就可以看出，这个函数是不能被包外代码调用的。

回到代码包导入的问题上来。代码包导入还有另外一种情况。如果我们只想初始化某个代码包而不需要在当前源码文件中使用那个代码包中的任何代码，就可以用“_”来代替别名，如下所示：

```
import (
    _ "logging"
)
```

这种情况下，我们只触发了对代码包logging的初始化操作。符号“_”就像一个垃圾桶，它在Go语言代码中使用很广泛，在后续章节中我们可以看到。

一个代码包怎样被初始化呢？我们下面开始说明这个问题。

3. 包初始化

在Go语言中，可以有专门的函数负责代码包初始化。这个函数需要无参数声明和结果声明，且名称必须为init，如下所示：

```
func init() {
    println("Initialize...")
}
```

Go语言会在程序真正执行前对整个程序的依赖进行分析，并初始化相关的代码包。也就是说，所有的代码包初始化函数都会在main函数（命令源码文件中的入口函数）之前执行完成，而且只会执行一次。并且，当前代码包中的所有全局变量的初始化会在代码包初始化函数执行前完成。这就避免了在代码包初始化函数对某个变量进行赋值之后又被该变量声明中赋予的值覆盖掉的问题。

例如，我们有如下源码文件：

```
package main // 命令源码文件必须在这里声明自己属于main包

import ( // 引入了代码包fmt和runtime
    "fmt"
    "runtime"
)

func init() { // 包初始化函数
    fmt.Printf("Map: %v\n", m) // 先格式化再打印
    // 通过调用runtime包的代码获取当前机器所运行的操作系统以及计算架构
    // 而后通过fmt包的.Sprintf方法进行字符串格式化并赋值给变量info
    info = fmt.Sprintf("OS: %s, Arch: %s", runtime.GOOS, runtime.GOARCH)
}

var m map[int]string = map[int]string{1: "A", 2: "B", 3: "C"}
    // 非局部变量，map类型，已被显式赋值

var info string // 非局部变量，string类型，未被显式赋值

func main() { // 命令源码文件必须有的入口函数
    fmt.Println(info) // 打印变量info
}
```

我们把它命名为initpkg_demo.go，并保存到goc2p项目的basic/pkginit包中。现在我们来运行这个文件：

```
hc@ubt:~/golang/goc2p/src/basic/pkginit$ go run initpkg_demo.go
Map: map[1:A 2:B 3:C]
```

OS: linux, Arch: 386

关于每行代码的用途，在源码文件中已经作了基本的解释，这里我们再解释一下这个小程序的输出。

输出的第一行是对变量m格式化后的结果。这就意味着，在函数init的第一条语句执行时，变量m已经被初始化并赋值了。这验证了一条规则：当前代码包中所有全局变量的初始化会在代码包初始化函数执行前完成。

输出的第二行是对变量info格式化后的结果。变量info被定义时并没有被显式赋值，因此它被赋予类型string的零值——“”（空字符串）。之后，变量info在代码包初始化函数init中被赋值，并在入口函数main中被输出。可见，所有的包初始化函数都会在main函数之前执行完成。

在同一个代码包中，可以存在多个代码包初始化函数，甚至代码包内的每一个源码文件都可以定义多个代码包初始化函数。Go语言编译器不能保证同一个代码包中的多个代码包初始化函数的执行顺序。如果确实要求按特定顺序执行的话，可以考虑使用Channel进行控制。Channel是Go语言并发编程模型中的一员，在本书第三部分，我们会详细介绍它。

最后需要说明的是，Go语言认可两个特殊的代码包名称——all和std。all代表了环境变量GOPATH中包含的所有工作区中的所有代码包，而std则代表了Go语言标准库中的所有代码包。

2.3 标准命令概述

Go语言中包含了大量用于处理Go语言代码的命令和工具。其中，go命令就是最常用的一个，它有许多子命令。这些子命令都拥有不同的功能，如下所示。

- build: 用于编译给定的代码包或Go语言源码文件及其依赖包。
- clean: 用于清除执行其他go命令后遗留的目录和文件。
- doc: 用于执行godoc命令以打印指定代码包。
- env: 用于打印Go语言环境信息。
- fix: 用于执行go tool fix命令以修正给定代码包的源码文件中包含的过时语法和代码调用。
- fmt: 用于执行gofmt命令以格式化给定代码包中的源码文件。
- get: 用于下载和安装给定代码包及其依赖包。
- install: 用于编译和安装给定的代码包及其依赖包。
- list: 用于显示给定代码包的信息。
- run: 用于编译并运行给定的命令源码文件。
- test: 用于测试给定的代码包。
- tool: 用于运行Go语言的特殊工具。
- version: 用于显示当前安装的Go语言的版本信息。

我们在执行这些命令的时候可以通过附加一些额外的标记来定制命令的执行过程。下面是一些比较通用的标记。

- -a: 用于强行重新构建所有涉及的Go语言代码包（包括Go语言标准库中的代码包），即使

它们已经是最新的了。

- ❑ `-n`: 使命令仅打印在执行期间使用到的所有命令，而不真正执行它们。
- ❑ `-v`: 用于打印出命令执行过程中涉及的Go语言代码包的名字。这些代码包一般包括我们给定的目标代码包，有时还会包括该代码包直接或间接依赖的代码包。
- ❑ `-wrok`: 打印出命令执行时生成和使用的临时工作目录的名字，且命令执行完成后不对它进行删除。
- ❑ `-x`: 打印出命令执行期间使用到的所有命令。

我们也可以暂且把这些标记看作命令的特殊参数。上面展示的这些特殊参数可以添加到命令名称和命令的真正参数中间。用于构建、安装、运行和测试Go语言代码包或源码文件的命令都支持这些特殊参数。我们会在本书后面的内容中再次提及和使用它们。

除了这些go命令的子命令之外，Go语言还自带了很多有用的命令和工具。比如上面提到的`godoc`命令和`gofmt`命令。我们还可以通过执行`go tool`命令来运行一些特殊的Go工具。比较常用的Go语言的特殊工具有以下几个。

- ❑ `fix`: 可以把给定代码包的所有Go语言源码文件中的旧版本代码修正为新版本。它是我们升级Go语言版本后会使用到的工具。
- ❑ `vet`: 用于检查Go语言源码中静态错误的简单工具。我们可以使用它检测一些常见的Go语言代码编写错误。
- ❑ `pprof`: 用于以交互的方式访问一些性能概要文件。命令将会分析给定的概要文件，并根据要求提供高可读性的输出信息。这个工具可以分析的概要文件包括CPU概要文件、内存概要文件和程序阻塞概要文件。这些内含Go语言运行时信息的概要文件可以通过标准库代码包`runtime`和`runtime/pprof`中的程序来生成。
- ❑ `cgo`: 用于帮助Go语言代码使用C语言代码库，以及使Go语言代码可以被C语言代码引用。

在这里，我们并未深入说明这些命令和工具，本书也不会涉及它们的高级用法。我们只会在用到的时候顺带着进行一些简要的说明。不过，作为本书的一个附属品，我把Go语言命令的详细说明（Go命令教程）放在了著名的代码托管网站GitHub上。具体网址是https://github.com/hyper-carrot/go_command_tutorial。只要我们用Go语言开发软件就会用到这些命令，所以该教程会尽量做到详实。我也会持续地维护它，以便读者在需要时免费查阅。

2.4 本章小结

本章所有内容都是为真正编写Go语言程序做准备的。我们了解了安装和配置Go语言的基本方法，还知晓了Go语言程序的各种组织形式。如果我们对这一章的内容都熟知了，那么应该就可以轻易地摆弄后续章节中的各种示例了。不过，即使你忘记了这些基础内容也没有关系，翻回来回顾一下就是了。本章的内容都很简单，轻而易举就可以回忆起来。

Part 2

第二部分

编程基础

在第一部分，我们已经了解了怎样安装和配置 Go 语言的开发环境，以及 Go 语言的基本工程结构。现在，你应该能够编写并运行一个只向标准输出打印一行内容的命令源码文件了。

在第二部分，我们开始学习 Go 语言的语法，包括基本词法、数据类型和各种流程控制方法等。

在数据类型方面，我们会了解到针对 Go 语言的基本数据类型（如 `byte`、`uint64`、`string` 等）以及各种高级数据类型（如数组、切片、函数、结构体等）的展现、命名、声明、赋值、比较、转换等一系列操作方法。

在控制流程方面，Go 语言拥有很多独到之处。即使是在其他编程语言中常见的 `if` 语句、`switch` 语句和 `for` 语句，Go 语言也提供了很多便捷和有意思的使用方式。更别说它特有的 `defer` 语句和 `go` 语句（本书第三部分会详细介绍）了。

好了，现在关上我们身后刚刚打开的那扇门，在 Go 语言的世界里继续前行吧。

本章主要探讨Go语言的基本数据类型及其操作方法。在这之前，我们会先介绍一些Go语言的词法知识以及使用Go语言编写程序的必备概念。此外，在讲解上述内容的过程中，我们还会涉及稍微高级一些的主题，比如怎样比较两个类型相同或不同的值，怎样把一个值从一个类型转换为另一个类型。

现在就让我们开始吧。

3.1 基本词法

在Go语言中，词法指的是代码的构成法则。通俗地讲，词法规定了我们敲入怎样的字符才能够编写出Go语言编译器认可的代码。

Go语言的语言符号又称为词法元素，共包括5类：标识符（identifier）、关键字（keyword）、操作符（operator）、分隔符（delimiter），以及字面量（literal）。它们是组成Go语言代码和程序的最基本单位。一般情况下，空格符、水平制表符、回车符和换行符都会被忽略，除非它们作为多个语言符号之间的分隔符的一部分。另外，在Go语言中我们并不需要显式地插入分号。在必要时，Go语言会自动为代码插入分号以进行语句分隔。

Go语言代码由若干个Unicode字符组成，也正是这些字符代表和形成了各种各样的Go语言符号。我们只要将这些Go语言符号按照Go语言的语法规则进行排列，就能够编写出Go语言编译器认可的程序。因此，我们学习使用Go语言编写程序的第一步，就是了解Go语言的语言符号及其使用方法。

这里简要介绍一下Unicode编码规范。Unicode编码规范是一种在计算机上使用的字符编码方式。它为世界上已存在的各种语言的每个字符都设定了统一且唯一的二进制编码。因此，它能够满足跨语言、跨平台地转换和处理文本的要求。关于Unicode编码的详细说明，请参见相关的文献和资料，或者直接访问其官方网站<http://www.unicode.org>并查询文档。

在这里，读者只需要记住一条简单的规则：Go语言的所有源代码都必须由Unicode编码规范的UTF-8编码格式进行编码。换句话说，我们编写的Go语言源码文件必须是UTF-8编码格式的。

本节会重点介绍Go语言的标识符、关键字和操作符，而字面量是表示各种数据类型及其值的重要方法，本节会先予以简单介绍，然后在3.2节讲解Go语言复合数据类型时，会对复合字面

量进行重点介绍。关于分隔符的知识，我们也会在本章后续部分中予以介绍。

3.1.1 标识符

下面我们来对Go语言中最灵活的语言符号——标识符进行说明。一个标识符可以代表一个变量或一个类型。也就是说，我们可以把标识符看作是变量或类型的代号或名称。Go语言的标识符是由若干字母、下划线“_”和数字组成的字符序列。字符序列的第一个字符必须为字母。这里所说的字母是广义的，只要能够由Unicode编码，就符合要求。

在Go语言代码中，每一个标识符都必须在使用前进行声明。一个声明将一个非空的标识符与一个常量、类型、变量、函数或代码包绑定在一起。在同一个代码块中，不允许重复声明同一个标识符（这有一个例外情况，请参见3.3.1节）。在一个源码文件和一个代码包中的标识符都需要遵循此规则。一个已被声明的标识符的作用域与其直接所属的代码块的范围相同。因此，也可以说，每一个有效的标识符都代表了特定范围内的程序实体。

严格来讲，代码包声明语句并不算是一个声明。因为代码包名称并不会出现在任何一个作用域中。代码包声明语句的目的是为了鉴别若干源码文件是否属于同一个代码包，或者指定导入代码包时的默认代码包引用名称。

很多时候，我们需要访问其他代码包中的变量或类型。这时就需要用到限定标识符。我们可以把限定标识符看作是把代码包名称作为前缀的标识符。代码包名称和标识符本身之间需要用英文句点“.”分隔。例如，当我们需要访问代码包os中名为O_RDONLY的常量时，就需要这样写：os.O_RDONLY。

我们刚才提到，一个限定标识符代表了对另一个代码包中的某个标识符的访问。这需要有两个前提条件。第一，这里所说的另一个代码包必须被事先导入。代码包的导入操作需要由Go语言的导入语句来实现。第二，这个在另一个代码包中的标识符必须是可导出的。

一个可导出的标识符也需要满足两个前提条件。第一，标识符名称中的第一个字符必须大写。Go语言是根据标识符名称中的第一个字符的大小写来确定这个标识符的访问权限的。具体的规则是：当标识符名称的第一个字符为大写时，其访问权限为“公开的”，这意味着该标识符可以被任何代码包中的任何代码通过限定标识符访问到；当标识符名称的第一个字符为小写时，其访问权限就是“包级私有的”，也就是说，只有与该标识符同在一个代码包的代码才能够访问到它。第二，标识符必须是被声明在一个代码包中的变量或者类型的名称，或者是属于某个结构体类型的字段名称或方法的名称。

另外，在Go语言中还存在着一类特殊的标识符，叫作预定义标识符。这类标识符随Go语言的源码一同出现。它们是在Go语言源码中被声明的。这类标识符包括以下几种。

- 所有基本数据类型的名称。
- 接口类型error。
- 常量true、false和iota。
- 所有内建函数的名称，即append、cap、close、complex、copy、delete、imag、len、make、new、panic、print、println、real和recover。

这些特殊标识符会在本章陆续出现，我们会逐渐地熟悉它们。

现在我们来稍稍总结一下。在声明一个变量或自定义一个类型的时候，我们会创建标识符以表示变量或类型的名称。如果我们要引用其他代码包中的代码，那么就需要使用限定标识符。我们在声明一个基本数据类型的变量或者使用Go语言内建函数时还会用到预定义标识符。总之，标识符无处不在。它是我们编写Go语言代码时最常用到的语言符号。

顺便提一下，我们在Go语言代码中还会碰到一个叫作空标识符的标识符。它由一个下划线_表示。它一般被用在不需要引入一个新绑定的声明中。举个简单的例子，如果在代码中存在一个变量x，但是却不存在任何对它的使用。这样的代码会使编译器报错。如果我们不想因此引起一个编译错误的话，就需要在变量x的声明代码后的某个位置添加这样一行代码：

```
_ = x
```

这个方法可以绕过编译器检查，使它不产生任何编译错误。这是因为这段代码确实使用到了变量x。不过，它没有在变量x上进行任何操作，也没有将它赋值给任何其他变量。更常用的一个例子是，在导入语句（或者称导入声明）中，当我们只想执行一下某个代码包中的初始化函数，而不需要使用这个代码包中的任何程序实体的时候，可以这样编写这个导入语句：

```
import _ "runtime/cgo"
```

其中，“runtime/cgo”代表了一个标准库代码包的标识符。这段代码引发了导入这个代码包所需的所有操作（包括执行其中的所有初始化函数），但是却没有真正地把它绑定到一个具体的名称上。也正因为如此，在当前的源码文件中，我们无法对这个代码包中的任何程序实体进行调用（我们无法通过一个标识符访问到它）。以上就是空标识符“_”的作用。它只会导致赋值或导入操作的相关准备工作的进行而已。

在上面的内容中，我们提及了常量、变量、字段和方法等名词，本章的后续部分将详细地进行说明。

3.1.2 关键字

关键字是指被编程语言保留而不让编程人员作为标识符使用的字符序列。因此，关键字也称为保留字。每个编程语言都有自己的关键字。一个编程语言的关键字可以侧面体现出这门语言的简洁程度、功能特性甚至特有哲学。

现在来看看Go语言中的关键字。从使用角度看，Go语言的关键字可以分为3类，包括用于程序声明的关键字、用于程序实体声明和定义的关键字，以及用于程序流程控制的关键字，如表3-1所示。

表3-1 Go语言中的关键字

类 别	关 键 字
程序声明	import, package
程序实体声明和定义	chan, const, func, interface, map, struct, type, var
程序流程控制	go, select, break, case, continue, default, defer, else, fallthrough, for, goto, if, range, return, switch

可以看到，Go语言的关键字并不多，只有25个。这也体现了Go语言简约的设计风格。

其中，用于程序声明的关键字只有两个——`import`和`package`。这两个关键字我们已经很熟悉了。在本书第2章已经对它们进行过详细的说明。

在Go语言中，程序实体的声明和定义是建立在其数据类型体系之上的。用于各种程序实体声明和定义的关键字并不多，只有8个。其中大多数关键字用来声明和定义Go语言的复合数据类型，包括`chan`、`func`、`interface`、`map`和`struct`，共5个。它们分别与Go语言的复合数据类型Channel（通道）、Function（函数）、Interface（接口）、Map（字典）和Struct（结构体）相对应。除此之外，关键字`type`用于自定义数据类型，它与上述5个关键字可以连用。关键字`var`可以用于声明任何Go语言数据类型的变量，而关键字`const`则用于声明常量和常量表达式。它们的具体用法都会在后面讲到。

Go语言有许多种程序流程控制方法，我们在使用这些方法时会用到相应的关键字。这些关键字也是在Go语言所有关键字中比重最大的一部分，共有15个，绝大部分在第4章中都会讲到，除了`go`和`select`。这两个关键字主要用于Go语言并发编程，因此我们会在本书第三部分中讲到它们。它们恰恰也是Go语言所特有的关键字。尤其是`go`，将会在本书后续内容中展现其强大的威力。

3.1.3 字面量

简单来说，字面量就是表示值的一种标记法。但是，在Go语言中，字面量的含义要更加广泛一些。

我们常常会在Go语言代码中用到的字面量有以下3类。

□ 用于表示基础数据类型值的各种字面量。这是编写Go语言代码时最常用到的一类字面量。

例如，表示浮点数类型值的`12E-3`。我们会在3.2节对它们进行说明。

□ 用于构造各种自定义的复合数据类型的类型字面量。例如，下面的字面量用于表示一个名称为`Person`的自定义结构体类型：

```
type Person struct {  
    Name    string  
    Age     uint8  
    Address string  
}
```

我们在编写Go语言代码的时候经常会根据需要创建出各种各样的自定义数据类型。因此，类型字面量在我们的代码中也会频繁出现。我们会在3.1.4节讲各种复合类型的时候说明相应类型的字面量的编写方法。

□ 用于表示复合数据类型的值的复合字面量。更确切地讲，它会被用来构造类型`Struct`（结构体）、`Array`（数组）、`Slice`（切片）和`Map`（字典）的值。复合字面量一般由值的字面类型和由花括号“`{`”和“`}`”括起来的复合元素的列表组成。这里的字面类型指的是隶属于结构体、数组、切片或字典类型的自定义数据类型的名称。而一个复合元素可以是一个单一表达式，也可以是一个键值对。如果是单一表达式，那么表达式的结果值必须

可以被赋给对应的字段、元素或字面类型中的键类型；如果是键值对，那么键就应该是结构体类型字面量中定义的一个字段的名称，或者是数组或切片类型字面量中的一个有效的索引值，再或者是字典类型字面量中定义的键类型的一个值。对于被用来表示字典类型值的复合字面量来说，每一个复合元素都必须是键值对形式的。例如，下面的字面量用于表示上面那个名称为Person的结构体类型的值：

```
Person{Name: "Robert Hao", Age: 32, Address: "Beijing, China"}
```

这个复合字面量中的类型名称是Person，意味着它代表了一个Person类型的值。在这个类型名称后面的是用花括号“{”和“}”括起来的3个作为复合元素的键值对。注意，对复合字面量的每次求值都会导致一个新的值被创建。因此，上面示例中的复合字面量每被求值一次就会创建出一个新的Person类型的值。我们还要注意的是，Go语言不允许在一个此类的复合字面量中为多个复合元素指定同一个字段名称或其他常量形式的键。例如，下面这3个复合字面量就都是错误的。

```
Person{Name: "Robert Hao", Age: 32, Name: "Unknown"} // 表示结构体类型值
map[string]string{"Name": "Robert Hao", "Age": "32", "Age": "32"} // 表示字典类型值
[]string{0: "0", 1: "1", 0: "-1"} // 表示切片类型值
```

这些字面量都无法通过编译，因为在它们的花括号中的键都有重复。

与复合字面量相关的更多编写方法和技巧将会在下一节陆续呈现。在这里，我们只是宏观地对字面量方面的知识进行概述。字面量是非常重要的一类语言符号。除非只是写写打印问候语句的Go语言程序，否则我们基本上都要用到它们。

3.1.4 类型

一个类型确定了一类值的集合，以及可以在这些值上施加的操作。类型可以由类型名称或者类型字面量指定。类型分为基本类型和复合类型，基本类型的名称可以代表其自身。比如，以下代码

```
var bookName string
```

就声明了一个类型为string、名称为bookName的变量。类型string是基本类型中的一个。其他基本类型有bool、byte、rune、int/uint、int8/uint8、int16/uint16、int32/uint32、int64/uint64、float32、float64、complex64和complex128，共18个。我们在之前讲过，基本类型的名称都属于预定义标识符。因此，基本类型也可以称为预定义类型。除了bool和string之外的其他基本类型也叫作数值类型。对于基本类型而言，我们可以通过由类型名称和圆括号括起来的字面量组成的表达式，把这个字面量转换为该类型的值。比如，表达式uint(123)会将字面量123转换为uint类型的值。我们会在后面经常用到这类表达式。关于数据类型转换的更多内容，我们会在后面陆续展开。

除了基本类型之外，Go语言还有若干复合类型，包括我们已经提到过的Array（数组）、Struct（结构体）、Function（函数）、Interface（接口）、Slice（切片）、Map（字典）和Channel（通道），

以及我们还未曾提及的`Pointer`（指针），共有8个。请读者记住这些官方英文名称和中文译名之间的对应关系，因为我们此后会穿插地使用这些名称。

复合类型一般由若干（也包括零）个其他已被定义的类型组合而成。比如：

```
type Book struct {
    Name      string
    ISBN      string
    Press     string
    PageNumber uint16
}
```

以上代码是一个复合类型`struct`的声明，名称是`Book`。它由3个`string`类型的字段（`Name`、`ISBN`和`Press`）和一个`uint16`类型的字段（`PageNumber`）组合而成。

从另一个角度来讲，Go语言中的类型又可以分为静态类型和动态类型。一个变量的静态类型是指在变量声明中示出的那个类型。绝大多数类型的变量都只拥有静态类型。唯独接口类型的变量例外，它除了拥有静态类型之外，还拥有动态类型。这个动态类型代表了在运行时与该变量绑定在一起的值的实际类型。这个实际类型可以是实现了这个接口类型的任何类型。接口类型的变量的动态类型可以在执行期间变化，因为所有实现了这个接口类型的类型的值都可以被赋给这个变量。但是，这个变量的静态类型永远只能是它声明时被指定的那个类型。也就是说，接口类型的变量的静态类型永远会是这个接口类型。这与Go语言的接口实现方式有很大关系。我们会在讲述接口类型的时候详细说明它。

下面我们再来说说类型的潜在类型。每一个类型都会有一个潜在类型。如果这个类型是一个预定义类型（也就是基本类型），或者是一个由类型字面量构造的复合类型，那么它的潜在类型就是它自身。例如，`string`类型的潜在类型就是`string`类型。又例如，在上面的示例中我们自定义的那个名为`Book`的类型的潜在类型就是`Book`。如果一个类型并不属于上述情况，那么这个类型的潜在类型就是在类型声明中的那个类型的潜在类型。这句话说起来比较拗口，我们来举几个例子吧。如果我们使用关键字`type`声明一个自定义类型：

```
type MyString string
```

实际上，我们可以把类型`MyString`看作`string`类型的一个别名类型，那么`MyString`类型的潜在类型就是`string`类型。Go语言基本数据类型中的`rune`类型也是如此。它可以看作是`uint32`类型的一个别名类型，其潜在类型就是`uint32`。需要注意的是，类型`MyString`和类型`string`是两个不相同的类型。例如，我们不能把属于其中一个类型的值赋给另一个类型的变量。但是，别名类型与它的源类型的不同仅仅体现在名称上，它们的内部结构却是一致的。所以，用于类型转换的表达式`MyString("ABC")`和`string(MyString("ABC"))`都是合法的。并且，更重要的是，这种类型转换并不会创建新的值。

一个类型的潜在类型具有可传递性。也就是说，如果存在如下类型声明：

```
type iString MyString
```

那么类型`iString`的潜在类型会与`MyString`类型的相同，即为`string`类型。

下面，我们声明这样一个类型：

```
type MyStrings [3]string
```

注意，类型MyStrings的潜在类型并不是[3]string。因为[3]string既不是一个预定义类型，也不是一个由类型字面量构造的复合类型，而是一个元素类型为string的数组类型。根据上面的定义，我们应该把[3]string类型的潜在类型作为类型MyStrings的潜在类型，而[3]string类型的潜在类型是string类型。因此，类型MyStrings的潜在类型就是string类型。

我们颇费口舌地阐明了潜在类型的概念。那么在现实场景中它又有什么意义呢？我们还是以数组类型为例。在Go语言中，有如下规则：一个数组类型的变量的声明中的类型决定了在这个变量中可以存放哪一个类型的元素。现在我们使用潜在类型这个概念替换一部分文字，即一个数组类型的潜在类型决定了在该类型的变量中可以存放哪一个类型的元素。这样说是不是更简洁且容易理解呢？潜在类型和类型、变量、函数一样，都是Go编程语言中的基础概念。我们一旦搞明白了它们，在今后提及更复杂的定义和概念的时候，就可以更加轻松地接受和领会了。

在上面的说明中，我们已经展示了几段关于类型声明的代码。Go语言的类型声明方式多种多样，对于复合类型来讲尤为如此。但是，这些类型声明语句的总体结构是基本一致的。一个类型声明总会以关键字type作为开始，并且需要用一个自定义标识符作为这个新类型的名称。最后，它还会包含一个基本类型的名称或者一个复合类型的定义。关于Go语言各个数据类型的详细讲解将在下一节陆续呈现。

3.1.5 操作符

操作符就是用于执行特定算术或逻辑操作的符号。在一些编程语言中，这些符号称为操作符。操作符总是会与其操作对象结合起来以表达既定的语义。这些操作对象统称为操作数。

在Go语言中，操作数代表了一个表达式中的基本值。Go语言的操作数可以是字面量，也可以是用于代表常量、变量或函数名称的限定或非限定标识符，还可以是方法（Method，函数的一种）表达式或者带圆括号的表达式。我们在下一小节中会讲到方法表达式。在这里，我们先来了解一下其他形式的操作数。在读完本节之后，相信读者对上述大部分操作数都会有足够的认知。

为了容易理解，本小节只使用表示基本数据类型值的字面量（操作数的有效形式之一）来辅助说明各种操作符的用法。另外，本小节还会多次提到表达式这个名词。表达式是下一小节的主题，我们在这里可以暂时把表达式理解为若干操作符和操作数的组合。

现在，让我们重新关注操作符。Go语言中的操作符并不多。从操作符与操作数的结合方式来看，Go语言的操作符可以被分为一元操作符和二元操作符。所谓一元操作符就是仅需要一个操作数的操作符，包括+、-、!、^、*、&和<-。而二元操作符就是需要两个操作数的操作符。在Go语言中没有三元操作符，所以除了一元操作符以外的操作符都必定是二元操作符。此外，除了!和<-之外的一元操作符也可以作为二元操作符来使用。下面我们通过表3-2来看看Go语言的操作符都有哪些。

表3-2 Go语言中的操作符

符号	说 明	示 例
	表示逻辑或操作。它是二元操作符，同时也属于逻辑操作符	true false //表达式的结果是true
&&	表示逻辑与操作。它是二元操作符，同时也属于逻辑操作符	true && false //表达式的结果是false
==	表示相等判断操作。它是二元操作符，同时也属于比较操作符	"abc" == "abc" //表达式的结果是true
!=	表示不等判断操作。它是二元操作符，同时也属于比较操作符	"abc" != "Abc" //表达式的结果是true
<	表示小于判断操作。它是二元操作符，同时也属于比较操作符	1 < 2 //表达式的结果是true
<=	表示小于或等于判断操作。它是二元操作符，同时也属于比较操作符	1 < =2 //表达式的结果是true
>	表示大于判断操作。它是二元操作符，同时也属于比较操作符	3 > 2 //表达式的结果是true
>=	表示大于或等于判断操作。它是二元操作符，同时也属于比较操作符	3 >= 2 //表达式的结果是true
+	表示求和操作。它既是一元操作符又是二元操作符，同时也属于算术操作符。若作为一元操作符，此操作符不会对原值产生任何影响	+1 //表达式的结果是1 1 + 2 //表达式的结果是3
-	表示求差操作。它既是一元操作符又是二元操作符。若作为一元操作符，则表示求反操作。同时，它也属于算术操作符	-1 //表达式的结果为-1（1的相反数） 1 - 3 //表达式的结果是-2
	表示按位或操作。它是二元操作符，同时也属于算术操作符	5 11 //表达式的结果是15
^	表示按位异或操作。它既是一元操作符又是二元操作符。若作为一元操作符，则表示按位补码操作。同时，它也属于算术操作符	5 ^ 11 //表达式的结果是14 ^5 //表达式的结果是-6
*	表示求乘积操作。它既是一元操作符又是二元操作符。同时，它也属于算术操作符和地址操作符。若作为地址操作符，则表示取值操作	*p //若p为指向整数类型值2的指针类型值，则表达式的结果即为2 2 * 5 //表达式的结果是10
/	表示求商操作。它是二元操作符，同时也属于算术操作符	10 / 5 //表达式的结果是2
%	表示求余数操作。它是二元操作符，同时也属于算术操作符	12 % 5 //表达式的结果是2
<<	表示按位左移操作。它是二元操作符，同时也属于算术操作符	4 << 2 //表达式的结果是16
>>	表示按位右移操作。它是二元操作符，同时也属于算术操作符	4 >> 2 //表达式的结果是1
&	表示按位与操作。它既是一元操作符又是二元操作符。同时，它也属于算术操作符和地址操作符。若作为地址操作符，则表示取址操作	&v //表达式的结果为标识符v所代表的值在内存中的地址 5 & 11 //表达式的结果是1
&^	表示按位清除操作。它是二元操作符，同时也属于算术操作符	5 &^ 11 //表达式的结果是4
!	表示逻辑非操作。它是一元操作符，同时也属于逻辑操作符	!b //若b的值为false，则表达式的结果为true
<-	表示接收操作。它是一元操作符，同时也属于接收操作符	<- ch //若ch代表了元素类型为byte的通道类型值，则此表达式从ch中接收byte类型值的操作

如表3-2所示，Go语言的操作符一共有21个。在该表的说明中，我们提到了一些与操作符有关的名词，包括算术操作符、比较操作符、逻辑操作符、地址操作符和接收操作符。实际上，这5个名词就是根据操作符的功能对Go语言操作符进行分类的结果。为了便于记忆和使用，我们下面就以功能分类来对Go语言操作符进行详细的讲解。

1. 算术操作符

算术操作符是用于操作数值类型值的符号。算术操作符对一个或两个操作数进行操作，并产生与第一个操作数具有相同类型的结果。其中的4个标准算术操作符可以作用于整数、浮点数和复数类型的操作数。它们是+、-、*和/。其中，标准操作符+还可以用于操作字符串类型的值，作为字符串连接符来使用。比如：

```
"Hello, " + "Golang" + "!" //表达式的结果是"Hello, Golang!"
```

注意，这样的字符串连接操作只会创建并使用一个新的字符串值来保存操作结果，而不会改变任何操作数的值。

除了标准算术操作符之外，其余的Go语言算术操作符都只能用于操作整数类型，包括%、&、|、^、&^、<<和>>，共7个。

需要说明的是，算术操作符/和%共同实现了舍尾除法（truncated division）。简单来说，舍尾除法将两个数值相除之后的结果分解为了商数和余数。有下面的公式：

$$\text{被除数} = \text{除数} \times \text{商数} + \text{余数}$$

余数的绝对值一定会小于除数的绝对值。实际上，对两个数值进行舍尾除法后得到的商数总是通过将除法结果向零取整而得出的。下面举几个例子：

- 设被除数为5、除数为2，则除法结果为2.5。从结果2.5中可以分解出的商数2和余数为1。因为 $5 = 2 \times 2 + 1$ 。
- 设被除数为5、除数为-2，则除法结果为-2.5。从结果-2.5中可以分解出的商数-2和余数1，因为 $5 = -2 \times -2 + 1$ 。
- 设被除数为-5、除数为2，则除法结果为-2.5。从结果-2.5中可以分解出商数-2和余数-1，因为 $-5 = 2 \times -2 - 1$ 。
- 设被除数为-5、除数为-2，则除法结果为2.5。从结果-2.5中可以分解出商数2和余数-1，因为 $-5 = -2 \times 2 - 1$ 。

由此，我们可以总结出这样一个规律：对两个数值进行舍尾除法后得到的余数的正负符号，总是会与被除数的正负符号相同。标准算术操作符%也会遵循此规则。除此之外，我们还需要注意，除数不能为零！否则程序会引发一个运行时的恐慌（panic）。不过，如果被除数是浮点数类型或复数类型，那么恐慌也不一定会发生，这取决于具体的实现方式。

除了/和%，其他非标准算术操作符都用于二进制位运算。我们下面对它们进行解释。

算术操作符&和|分别实现了按位与和按位或的操作。按位与的含义是，在同位上的两个二进制数只要有一个为0，则此位的结果就为0，否则此位的结果为1。而按位或的含义是，在同位上的两个二进制数只要有一个为1，则此位的结果就为1，否则此位的结果为0。例如：

```
7      & 13      = 5      // 十进制数的按位与操作
00000111 & 00001101 = 00000101 // 对应的二进制数运算示意
```

再如：

```
7      | 13      = 15      // 十进制数的按位或操作
00000111 | 00001101 = 00001111 // 对应的二进制数运算示意
```

为了简便和清晰，我们下面会以仅由8位二进制位就可以表示的数值（即byte类型的数值）作为示例数据。

算术操作符`^`实现了按位异或操作。按位异或的含义是，若在同位上的两个二进制数的值相同，则此位的结果就为0，否则，此位的结果为1。例如：

```
7      ^ 13      = 10      // 十进制数的按位异或操作
00000111 ^ 00001101 = 00001010 // 对应的二进制数运算示意
```

算术操作符`&^`实现了按位清除操作。按位清除的含义是，根据第二个操作数的二进制值对第一个操作数的二进制值进行相应的清零操作。具体来讲，如果第二个操作数的某个二进制位上的数值为1，那么就把第一个操作的对应二进制位上的数值设置为0。否则，第一个操作数的对应二进制位上的数值不变。当然，这不会改变第一个操作数的原值，只会根据两个操作数的二进制值计算出结果值。我们来看下面这个例子：

```
7      &^ 13      = 2      // 十进制数的按位清除操作
00000111 &^ 00001101 = 00000010 // 对应的二进制数运算示意
```

如果我们将两个操作数互换，那么：

```
13     &^ 7       = 8      // 十进制数的按位清除操作
00001101 &^ 00000111 = 00001000 // 对应的二进制数运算示意
```

显然，按位清除是一种不具交换性的二元运算。也就是说，如果将参与按位清除运算的两个操作数互换位置，那么运算结果将会不同。作为特殊情况，如果两个操作数的值相同，那么其按位清除的运算结果就会为0。

算术操作符`<<`实现了按位左移的操作。我们先来看最简单的情况。以按位左移一位为例，它是指把二进制值上的每一位数值都用右边相邻位上的数值代替，最右边的二进制位（也称为最低比特位）上的数值会被设置为0。例如，2（二进制值为00000010）在按位左移一位之后得到4（二进制位00000100）。从表面上看，这就像将所有二进制位向左移动了一位。这也是“左移”一词的由来。算术操作符`<<`是二元操作符，它的第一个操作数就是需要被“左移”的数值，而第二个操作数就是“左移”操作的次数。正因为此，第二个操作必须是一个正整数。例如：

```
8      << 3 = 64      // 十进制数的按位左移操作
00001000 << 3 = 01000000 // 对应的二进制数运算示意
```

读者可能已经意识到了，每左移一位就相当于在当前操作数的基础上乘以2。这是因为我们是在二进制位上进行操作的。

如果读者理解了`<<`操作符的含义，那么`>>`操作符代表的操作就很容易理解了。算术操作符`>>`实现了按位右移的操作。没错，它代表了一个与`<<`操作符完全相反的运算，即“右移”N次且“最高比特位”补零。其中，N即是第二个操作数的值。示例如下：

```
8      >> 3 = 1       // 十进制数的按位右移操作
00001000 >> 3 = 00000001 // 对应的二进制数运算示意
```

类似地，每右移一位就相当于在当前操作数的基础上除以2，移位后得到的值即是其商数。

在Go语言的算术操作符中，有一些操作符可以作为一元操作符来使用，包括`+`、`-`、`^`、`*`和`&`。其中，需要特别说明的是操作符`^`。

如果与一元操作`^`联结的唯一操作数的类型是无符号的整数类型，这一操作就相当于对这个操作数和其整数类型的最大值进行二元的按位异或操作。例如：

```
^uint8(1)          = 254      // 无符号整数的一元按位异或操作
00000001 ^ 11111111 = 11111110 // 对应的二进制数运算示意
```

其中，内置函数`uint8`会将一个整数字面量转换为一个`uint8`类型的值。这确保了一元操作符`^`的唯一操作数一定是一个无符号整数类型的值。

如果与一元操作`^`联结的唯一操作数的类型是有符号的整数类型，那么这一操作就相当于对这个操作数和-1进行二元的按位异或操作。例如：

```
^1                  = -2       // 有符号整数的一元按位异或操作
00000001 ^ 11111111 = 11111110 // 对应的二进制数运算示意
```

注意，以上示例中的操作数的二进制值都是以补码形式表示的。并且，在上例中，我们并没有把操作数1的类型显式地转换为代表一个字节宽度的有符号整数数据类型`int8`（通过调用内置函数`int8`可以做到这一点）。这是因为在默认情况下整数字面量是有符号的。

除此之外，操作符`*`和`&`仅在被当作二元操作符使用时才属于算术操作符。如果我们把它们作为一元操作符使用，那么它们就属于地址操作符。关于地址操作符我们稍后再作介绍。

最后，需要特别注意的是，除了`<<`和`>>`的其他算术操作符只能对两个类型完全相同的数值类型的操作数进行操作。但是，只要有一个操作数是字面量就不存在这种约束。例如，表达式`uint(2) + int(3)`不会通过编译，而表达式`uint(2) + 3`却是合法的。后者中的3会被隐式地进行类型转换。具体细节请参看3.3.4节。

2. 比较操作符

Go语言的比较操作符用于比较两个操作数并得出无类型的布尔值。它们是`==`、`!=`、`<`、`<=`、`>`和`>=`，共6个。其中，操作符`==`、`!=`又称为相等操作符，而其余的4个操作符也称为排序操作符。这些操作符所实现的操作非常好理解。不过需要注意的是，参与操作的两个操作数所属的数据类型必须是互相兼容的。

关于针对各个数据类型及其值的具体比较规则，请读者查阅3.3.3节。

3. 逻辑操作符

Go语言的逻辑操作符包括`||`、`&&`和`!`，分别代表逻辑或、逻辑与和逻辑非操作。与它们联结的操作数只能是布尔值或者产出布尔值的表达式。逻辑操作符与操作数组合而成的表达式的结果也是布尔值。其中，操作符`||`和`&&`是二元操作符，也就是说它们都需要两个操作数。以操作符`||`为例：

```
b1 || b2
```

标识符`b1`和`b2`都是布尔值或者产出布尔值的表达式。只要它们中有一个为`true`，那么上述示例中的表达式的结果就为`true`。也正因为如此，如果`b1`被求值（或者说其计算结果）为`true`，那么Go语言会忽略对`b2`的求值（也可以说不会计算`b2`）。再来看操作符`&&`：

```
b3 && b4
```

同样，标识符**b3**和**b4**也都是布尔值或产出布尔值的表达式。与操作符**||**相反，只要它们中有一个为**false**，那么上述示例中的表达式的结果就为**false**。并且，只要**b3**被求值为**false**，那么Go语言就会忽略对**b4**的求值。

逻辑操作符**!**为一元操作符。如果其唯一操作数被求值为**false**，那么它与这个操作数组成的表达式的结果就是**true**，反之亦然。

我们可以把任意多个逻辑操作符和操作符进行串联并组成更复杂的表达式：

```
b1 || b2 || !b3 && b4
```

请记住，在一般情况下，当多个逻辑操作符和操作数进行串联时，Go语言总是从左到右依次对它们进行求值。也就是说，上述示例中的表达式相当于：

```
((b1 || b2) || !b3) && b4
```

Go语言会先对表达式**b1 || b2**进行求值，然后把这个求值结果与表达式**!b3**的求值结果做逻辑或运算，最后再把这第三个求值结果与**b4**做逻辑与运算。当然，我们也可以使用圆括号显式地改变求值的顺序。比如：

```
b1 || (b2 || !b3) && b4
```

这种情况下，Go语言会把**b1**与表达式**b2 || !b3**的求值结果做逻辑或运算，最后将这第3个结果与**b4**做逻辑与运算。显然，以不同的顺序运算多个相同的子表达式很可能会得到不同的最终结果。

读者很可能已经注意到了表达式**!b3**。它总是会被最先运算，除非我们显式地使用圆括号改变其运算顺序，比如**!(b3 && b4)**。这种可以被优先运算的特权与操作符的优先级有关，我们稍后再作解释。

4. 地址操作符

我们之前已经提到，操作符*****和**&**在作为一元操作符时即属于地址操作符。这时，操作符*****又被称为取值操作符，而操作符**&**又被称为取址操作符。

对于某个类型的操作数**v**，我们可以使用表达式**&v**来得到指向存放此操作数的内存地址的指针类型值。显然，这要求操作数**v**必须是可寻址的。也就是说，Go语言必须为了存储这个操作数专门开辟一段内存空间。为了满足这个要求，操作数**v**必须是下列几个种情况之一。

- **v**是一个变量的名称。这时，如果我们要对变量**v**进行取址操作应该这样编写表达式：**&v**。
- **v**是一个取值操作符和操作数组成的表达式。这种情况我们稍后再讨论。
- **v**是一个可寻址的数组类型值的索引表达式。对于至少包含了一个元素的数组类型值**a**，其有效的索引操作就是**a[0]**。这时，如果我们要对**a**的第一个元素进行取址操作，就应该这样编写表达式：**&a[0]**。
- **v**是一个切片类型值的索引表达式。对于至少包含了一个元素的切片类型值**s**，其有效的索引操作就是**s[0]**。这时，如果我们要对**s**的第一个元素进行取址操作就应该这样编写表达式：**&s[0]**。
- **v**是一个可寻址的结构体类型值中的一个字段选择器。对于包含了名称为**F**的字段的结构

体类型值`s`，我们可以这样访问这个字段`s.F`。`s.F`即为字段选择器的表现形式——选择表达式。这时，如果我们要对结构体类型值`s`的字段`F`进行取址操作，就应该这样编写表达式：`&s.F`。

- `v`可以是一个复合字面量。这种情况是作为可寻址能力要求的一个特例存在的。复合字面量代表了一个自定义复合类型的值，而这个值在被赋给一个变量之前，是没有任何指针类型值与它相对应的。因此，它是不可能被寻址的。但是，一个复合字面量仍然可以被取址。这是因为，我们在使用取址操作符`&`对一个复合字面量进行取址操作的时候，Go语言会为这个复合字面量所代表的值专门生成一个指针类型值。

前面的第2种情况提到的取值操作符即是`*`。作为一元操作符的`*`只能与指针类型的操作数进行联结。具体来说，对于代表了某个指针类型值的标识符`p`，表达式`*p`表示了这个指针类型值指向的那个值。注意，如果值`p`为空值（即`nil`），那么表达式`*p`在被求值时就会引发一个运行时的恐慌。

读者可能已经意识到，一元操作符`*`和`&`之间存在着一定的对立关系。对于表达式`*p`，我们可以再使用操作符`&`取到这个表达式所表示的那个被指向值的指针类型值。也就是说，表达式`&(*p)`的结果与指针类型值`p`相同。反过来讲也是一样的，对于可寻址的操作数`v`，我们同样可以使用表达式`*(&v)`来表示它。

关于指针的更多知识，请见3.2.8节。

5. 接收操作符

Go语言的接收操作符只有一个，即`<-`，只作用于通道类型的值。对于通道类型的值`ch`，表达式`<-ch`的含义是从此通道中接收一个值。前提是通道的方向必须允许接收操作，并且该操作的结果的类型必须与通道元素的类型之间存在可赋予的关系。这个表达式会被阻塞直到通道中有一个值可用。至于什么时候会有一个值可用，我们会在第7章说明。使用接收操作符时，以下两点需要特别注意。

- 从一个通道类型的空值（即`nil`）接收值的表达式将会永远被阻塞。
- 从一个已被关闭的通道类型值接收值会永远成功并立即返回一个其元素类型的零值。

一个由接收操作符和通道类型的操作数所组成的表达式可以直接被用于变量赋值或初始化，例如：

```
v1 := <-ch
v2 = <-ch
```

再如：

```
v, ok = <-ch
v, ok := <-ch
```

上面例子中的特殊标记`=`用于将一个值赋给一个已被声明的变量或常量。而特殊标记`:=`则用于在声明一个变量的同时对这个变量进行赋值，且只能在函数体内使用。我们在3.3.1节讲赋值语句的时候会对它们进行更详细的说明。

当我们同时对两个变量进行赋值或初始化时，第二个变量将会是一个布尔类型的值。这个值代表了接收操作的成功与否。这样我们就可以通过它来判断一个通道是否已被关闭了。如果此值

为false，那么就说明这个通道已经被关闭。

我们在第7章讲解通道类型的时候会再次遇到接收操作符。

6. 操作符优先级

当一个表达式中存在多个操作符时，就涉及操作先后顺序的问题。在Go语言中，一元操作符拥有最高的优先级，而二元操作符的优先级共有5个，如表3-3所示。

表3-3 Go语言的二元操作符的优先级

优 先 级	操 作 符
5	* / % << >> & &^
4	+ - ^
3	== != < <= > >=
2	&&
1	

在此表中，以数字来表示操作符的优先级，数字越大就意味着优先级越高。如果在一个表达式中出现了处于相同优先级的多个操作符，且这些操作符之间仅存在操作数，那么就会按照从左到右的顺序进行操作。比如，表达式`a << 4 * b & c`等同于`((a << 4) * b) & c`。当然，我们可以使用使用圆括号显式地改变原有的操作顺序，例如表达式`a << (4 * b) & c`等同于`((a << (4 * b)) & c)`，即子表达式`4 * b`会先被求值。

最后需要注意的是，`++`和`--`是语句而不是表达式，因而它们不存在于任何操作符优先级层次之内。例如，表达式`*p--`等同于`(*p)--`。

操作符是组成表达式的元素之一，我们需要熟知它们的用法。有时对操作符使用不当会造成不可预知的程序错误，并且不容易排查。因此，请读者对本小节所讲的内容进行重点记忆。

3.1.6 表达式

表达式代表了把操作符和函数作用于操作数的计算方法。在Go语言中，表达式是构成具有词法意义的代码的最基本元素。Go语言的表达式有很多种，我们在之前讲到的限定标识符、复合字面量、操作符和操作数都包含在内。在本小节中，我们会对Go语言中最常用和最重要的表达式进行说明。认识和理解这些表达式会对我们编写合格和优秀的Go语言代码有非常大的帮助。

1. 基本表达式

基本表达式一般作为复杂的表达式的一部分。它也是高级表达式的操作对象。基本表达式有如下几种形成方式。

- 使用操作数来表示基本表达式。例如，我们把用于表示切片类型值的复合字面量`[]int{1, 2, 3, 4, 5}`当作操作数，并使用这个操作数作为一个基本表达式。我们可以在这个基本表达式的基础上编写一个更复杂一些的表达式`[]int{1, 2, 3, 4, 5}[2]`。这个表达式用于取出数组中索引为2的元素。所以，此表达式的结果值即为3。顺便提一句，这种“更复杂一些”的表达式称为索引表达式。

- 使用类型转换作为基本表达式。例如，有一个类型为`uint8`的变量`v1`，我们想把它与一个`int`类型的变量`v2`相加。此时，我们需要这样编写表达式：`int(v1) + v2`。我们在这里使用了表达式`int(v1)`，而并没有直接使用变量名称`v1`。这是因为操作符`+`只能作用于两个类型完全相同的值。对于表达式`int(v1) + v2`来说，`int(v1)`就是它内部的一个基本表达式。
- 使用内建函数调用作为基本表达式。例如，有一个数组类型的变量`v3`，我们想知道它包含的元素的个数。这时，我们就需要使用Go语言的内建函数`len`，并编写调用这个内建函数的代码`len(v3)`以完成需求。这个内建函数调用代码`len(v3)`就是一个基本表达式。
- 一个基本表达式和一个选择符号可以组成另外一个基本表达式。选择符号是由英文句号“.”和一个标识符组合而成的符号。例如，如果在一个结构体类型中存在字段`f`，那么我们就可以在这个结构体类型的变量`x`上应用一个选择符号来访问这个字段`f`。这个表达式为`x.f`。其中，`.f`就是一个选择符号。注意，前提是这个变量`v`的值不能是`nil`。在Go语言中，`nil`用来表示空值。
- 一个基本表达式和一个索引符号可以组成另外一类基本表达式——索引表达式。索引符号由狭义的表达式和外层的方括号“[”和“]”组成。这里所说的狭义的表达式是指仅由操作符和操作数组成的表达式。我们在前面讲操作符的时候提到过这种表达式。我们在前面提到过的表达式`[]int{1, 2, 3, 4, 5}[2]`就是索引表达式。更复杂一点的例子如`[]int{1, 2, 3, 4, 5}[1+2]`。
- 一个基本表达式和一个切片符号可以组成一个基本表达式。切片符号由2个或3个狭义的表达式和外层的方括号组成，这些表达式之间由冒号“:”分隔。切片符号的作用与索引符号类似，但不同的是，索引符号只能用于取出数组或切片中的一个元素或者字符串中的一个字节，而切片符号则可以取出其中的一个或多个元素或字节。我们也可以这样理解：索引符号针对的是一个点，切片符号针对的是一个范围。举个简单的例子，如果我们想要取出一个切片`[]int{1, 2, 3, 4, 5}`的第二个到第四个元素，那么可以使用切片符号编写出这样的基本表达式：`[]int{1, 2, 3, 4, 5}[1:4]`。
- 一个基本表达式和一个类型断言符号可以组成一个基本表达式。类型断言符号以一个英文句号“.”为前缀，并后跟一个被圆括号括起来的类型名称或类型字面量。类型断言符号用于判断一个变量或常量是否为一个预期的类型，并根据判断结果采取不同的响应。例如，如果我们要判断一个`int8`类型的变量`num`是否是`int`类型，可以这样编写表达式：`interface{}(num).(int)`。这也许与我们想象的代码编写方式不太一样。实际上，这一表达式的结果值也可能会让初识Go语言的读者疑惑。当然，这涉及一个特定的用法，我们稍后会对它进行详细说明。
- 一个基本表达式可以和一个调用符号组成另外一个基本表达式。调用符号只针对于函数或者方法。因此，与调用符号组合的基本表达式一般不是一个代表代码包名称（或者其别名）的标识符就是一个代表结构体类型的方法的名称的标识符。调用符号由一个英文句号“.”为前缀和一个被圆括号括起来的参数列表组成，多个参数之间用逗号“,”分隔。例如，基本表达式`os.Open("/etc/profile")`表示对代码包`os`中的函数`Open`的调用。

至此，我们列出了Go语言中所有的基本表达式的基础概念和组成形式。下面我们将会对上述的几种基本表达式进行有针对性的说明。

2. 选择符号和选择表达式

首先需要记住的是，选择符号的应用对象不能是一个代表代码包名称（或者其别名）的标识符。例如，之前提到过的`os.O_RDONLY`是一个限定标识符，而不是一个包含选择符号的表达式。这正是因为`os`代表的是一个代码包。换个角度说，只有当一个基本表达式`x`不代表一个代码包的时候，我们才能在它之上应用一个选择符号，就像我们前面提到的基本表达式`x.f`。这类表达式又被称为选择表达式。可以看到，选择表达式与限定标识符在表现形式上非常相近，它们唯一的区别就在于应用对象。

基本表达式`x.f`中的`f`既可以是一个字段的名称，也可以是一个方法的名称。它被我们使用选择符`.`调用了。也正因为此，`x`必须代表一个至少拥有字段或方法的某个类型的值。在Go语言中，符合此要求的类型有结构体类型和接口类型。另外，`f`不能是空标识符。还记得空标识符吗？空标识符用“`_`”来表示。

选择符号可以用于调用任意深度的类型值的字段或者方法。最简单的情况就是，如果`x`代表了某个结构体类型的变量，那么选择表达式`x.f`就表示了深度为0的字段或者方法。如果`f`是`x`的一个字段的同时也代表了某个结构体类型的值，并且在这个类型上有一个字段`f2`，那么我们可以通过选择表达式`x.f.f2`访问到这个深度为1的某个结构体类型的值（由`x`的字段`f`代表）的字段`f2`。以此类推。也就是说，表达式最左边的那个值的深度为0，而每个选择符前缀（表现为英文句号“`.`”）右边的那个标识符代表的值，都比其左边的标识符代表的那个值的深度更深一层。

现在，我们对选择表达式中值的深度的含义已经有所了解了。下面让我们来看看一些与选择表达式有关的规则。

- ❑ 对于一个类型`T`或者对应的指针类型`*T`的值`x`，选择表达式`x.f`表示类型`T`的最浅深度（即深度为0）的字段或者方法。这里有两个前提条件：`T`不能是接口类型；类型`T`必须要有名称为`f`的字段或者方法。如果不满足这两个前提条件，那么选择表达式`x.f`就是非法的。
- ❑ 对于一个接口类型`I`的变量`x`，选择表达式`x.f`表示赋给`x`的那个值（实现了接口类型`I`的那个类型的值）的方法`f`。如果接口类型`I`的方法集合中不包含名称为`f`的方法，那么选择表达式`x.f`就是非法的。

请注意，除了上述两种情况之外的任何选择表达式都是非法的！

- ❑ 如果`x`是一个与某个结构体类型对应的指针类型的变量，并且它的值为`nil`，那么针对表达式`x.f`的赋值和求值都会引起一个运行时恐慌。不论`f`代表的是那个结构体类型的一个字段，还是它拥有的某个方法，都会是这样。
- ❑ 如果`x`是一个接口类型的变量且它的值为`nil`，那么针对表达式`x.f`的调用和求值都会引起一个运行时恐慌。这里的前提条件是，`f`必须代表的是该接口类型的一个方法。

上面提到的运行时恐慌属于Go语言异常处理机制的一部分。我们会在下一章讲到它。

在上面的规则中，我们提到了在与某个结构体类型对应的指针类型的变量上也能够找到在那个结构体类型中声明的字段。实际上，这涉及了选择符的一个特性。我们可以把这个特性叫作自

动解引用。具体来说，如果x是与一个结构体类型对应的指针类型的值，那么表达式x.f就是表达式(*x).f的一个速记法。不论f代表的是一个字段还是一个方法。注意，这里的操作符*是一个取值操作符，且表达式*x的结果值是指针类型值x指向的那个结构体类型值。如果f代表了一个字段且也是一个与结构体类型对应的指针类型值，那么表达式x.f.f2就是表达式(*(*x).f).f2的一个速记法，不论f2代表的是一个字段还是一个方法。以此类推。如果x包含了一个类型为*A的匿名字段，并且A是一个拥有字段或方法f的结构体类型，那么x.f就是选择表达式(*x.A).f的一个速记法。关于结构体类型的匿名字段的说明，请参看下一节。从表面上看，这里的表达式x.f代表的是一个深度为0的字段或方法，但是实质上它却代表了一个深度为1的字段或方法。这是因为，虽然类型为*A的匿名字段在其所属的结构体类型的声明中并没有一个显式的名称，但是它却具有与x的其他字段相同的深度。这从它代表的表达式(*x.A).f上就有所体现。

3. 索引符号和索引表达式

一个索引表达式由一个基本表达式和一个索引符号组成，形如a[x]。索引表达式a[x]会被求值为由x索引的a中的那个元素。被索引符号操作的基本表达式a代表的可以是数组、切片、字符串或字典类型的值。对于字典类型的值，标识符x则代表了字典的键。对于其他类型的值，x则代表了索引值。

与索引表达式有关的具体规则如下。

- ❑ 如果a不是一个字典类型的值。索引值x必须是一个int类型的值，或者是无类型的整数字面量。同时，x必须大于等于0且小于a的长度。否则针对该表达式的赋值和求值都会引起一个与越界有关的运行时恐慌。顺便提一句，a的长度的范围就是int类型可以表示的非负的取值范围。
- ❑ 如果A是一个数组类型，那么对于A及其对应的指针类型*A则有：作为索引值的常量必须在上一条规则描述的范围内。如果*A类型的变量a的值为nil（注意，数组类型的值不可能为nil）或者索引值x超出了规定的范围，就会引起一个运行时恐慌。索引表达式a[x]代表了数组类型的变量a的值中与索引值x对应的那个元素类型值，且a[x]的类型与数组类型A的元素类型相同。
- ❑ 如果S是一个切片类型，那么对于与其类型值绑定的变量a则有：如果切片类型的值为nil或者索引值x超出了范围，那么就会引起一个运行时恐慌。索引表达式a[x]代表了切片类型的变量a的值中与索引值x对应的那个元素类型值，且a[x]的类型与切片类型S的元素类型相同。
- ❑ 如果T是一个字符串类型，那么对于与其类型值绑定的变量a则有：作为常量的索引值必须在同是常量的字符串类型值的有效长度范围内。如果索引值x超出了有效范围，那么就会引起一个运行时恐慌。索引表达式a[x]代表了字符串类型的变量a的值中与索引值x对应的那个字节类型（即byte）值，且a[x]的类型即是字节类型。注意，不能对a[x]进行赋值操作！因为字符串类型值是不能改变的。
- ❑ 如果M是一个字典类型，那么对于与其类型值绑定的变量a则有：键x的类型必须是可以赋值给字典类型M的键的类型的。也就是说，键x的类型的值永远可以通过类型推断符号判

定为字典类型M的键的类型。如果a中包含了一个以x为键的键值对，那么表达式a[x]就代表了a中的、与键x对应的那个值，且a[x]的类型与字典类型M的元素的类型相同。如果字典类型的变量a的值为nil，或者其中不包含以x为键的键值对，那么表达式a[x]的求值结果就会是字典类型M的元素的类型的零值。

注意，除上述规则中描述的情况之外的任何表达式都是非法的！

关于上面所说的“字典类型的变量a的值为nil，或者其中不包含以x为键的键值对”的情况，还存在一个有歧义的地方。具体来说，如果表达式a[x]的求值结果是该字典类型的元素类型的零值，那么我们无法辨别得到这个零值的真实原因是由于该字典类型值内没有与键x对应的键值对，还是由于其中的与键x对应的值本身就是这个零值。对于此，Go语言还允许这样的赋值语句：

```
v, ok := a[x]
```

请注意，上面的索引表达式的结果是一对值，而不是单一值。第一个值的类型就是该字典类型的元素类型，而第二个值则是布尔类型的。在这个示例中，与变量ok绑定的布尔值代表了在字典类型的a中是否包含了以x为键的键值对。如果在a中包含这样的键值对，那么赋给变量ok的值就是true，否则就为false。这样，我们就可以很容易地消除上述歧义了。

最后，需要特别注意一点，虽然当字典类型的变量a的值为nil时，求值表达式a[x]并不会发生任何错误，但是在这种情况下对a[x]进行赋值却会引起一个运行时恐慌。

4. 切片符号和切片表达式

切片符号可以操作字符串、数组、数组的指针以及切片类型的值。对于这样一个类型的值a，切片表达式形如a[x:y:z]。a是切片符号[x:y]的操作对象。其中，x代表了切片的元素下界索引，y代表了切片的元素上界索引，而z则代表了切片的容量上界索引。对于它们有以下约束：

$0 \leq \text{元素下界索引} \leq \text{元素上界索引} \leq \text{容量上界索引} \leq \text{操作对象的容量}$

如果元素下界索引、元素上界索引或容量上界索引不满足此约束，那么切片表达式在被求值的时候，就会造成一个越界错误并引发一个运行时恐慌。

切片类型值的长度和容量是不同的。作为切片表达式的求值结果的那个切片类型值（以下简称新切片值）的长度一定等于元素上界索引 - 元素下界索引。我们可以直接编写针对新切片值的索引表达式以获取对应位置上的元素值，只要满足

$0 \leq \text{索引值} < (\text{元素上界索引} - \text{元素下界索引})$

即可。

切片类型值的容量体现了该值的最大有效索引范围。如果我们不在切片表达式中显式地指定容量值，那么新切片值的容量就一定等于

$\text{操作对象的容量} - \text{元素下界索引}$

的求值结果，否则其容量就一定等于

$\text{容量上界索引} - \text{元素下界索引}$

的求值结果。

对于切片表达式中的那3个索引值：如果它是一个常量，那么它在字面上就必须是一个int类型的非负值。如果索引值是由一个表达式代表的，那么就不存在这个限制，此表达式的求值结果只要是一个整数值就可以通过编译。

如果元素下界索引、元素上界索引和容量上界索引都是常量，那么它们就必须在字面上满足前面所说的那个约束条件。

对于元素下界索引和元素上界索引来说，容量上界索引显得并不那么重要，且总是可以被省略。但是，它也有特殊用途。在3.2.3节中，我们再将切片类型值的容量和容量上界索引的用法进行详细讲解。下面我们只关注切片表达式中的元素下界索引和元素上界索引。

对于字符串类型的值来说，切片表达式的作用是截取子字符串，并且它的求值结果也会是一个字符串类型的值。对于数组、数组的指针和切片类型的值来说，切片表达式的作用是截取一段切片。这种情况下，切片表达式的求值结果会是一个切片类型的值，并且这个切片类型值的元素类型会与切片符号的操作对象的元素类型一致。注意，如果切片符号的操作对象是数组类型的值，那么这个值必须是可寻址的。另外，当在一个空值（nil）上应用切片符号就会引发一个运行时恐慌。指向数组的指针类型和切片类型的值都可能为nil。

下面，我们通过一系列示例来增强对以上描述的理解。

我们以元素类型为int的切片类型值[]int{1, 2, 3, 4, 5}为例。切片表达式[]int{1, 2, 3, 4, 5}[1:3]的求值结果是由操作对象[]int{1, 2, 3, 4, 5}中的第二个元素到第三个元素组成的元素类型为int的切片类型值，即[]int{2, 3}。眼尖的读者可能会有个疑问：在这个例子中，元素上界索引为3。它对应于操作对象中的第四个元素。但是，切片表达式截取出的切片类型值中却不包含第四个元素。这是为什么呢？

要解释这个原因，需要从索引的真正含义开始讲起。对于字符串类型值中的每一个字节以及数组和切片类型值中的每一个元素来说，都有一个索引值相对应。那么，元素与索引值之间到底是怎样对应起来的呢？我们先来看图3-1。

切片类型的变量a，值为：[]int{1, 2, 3, 4, 5}

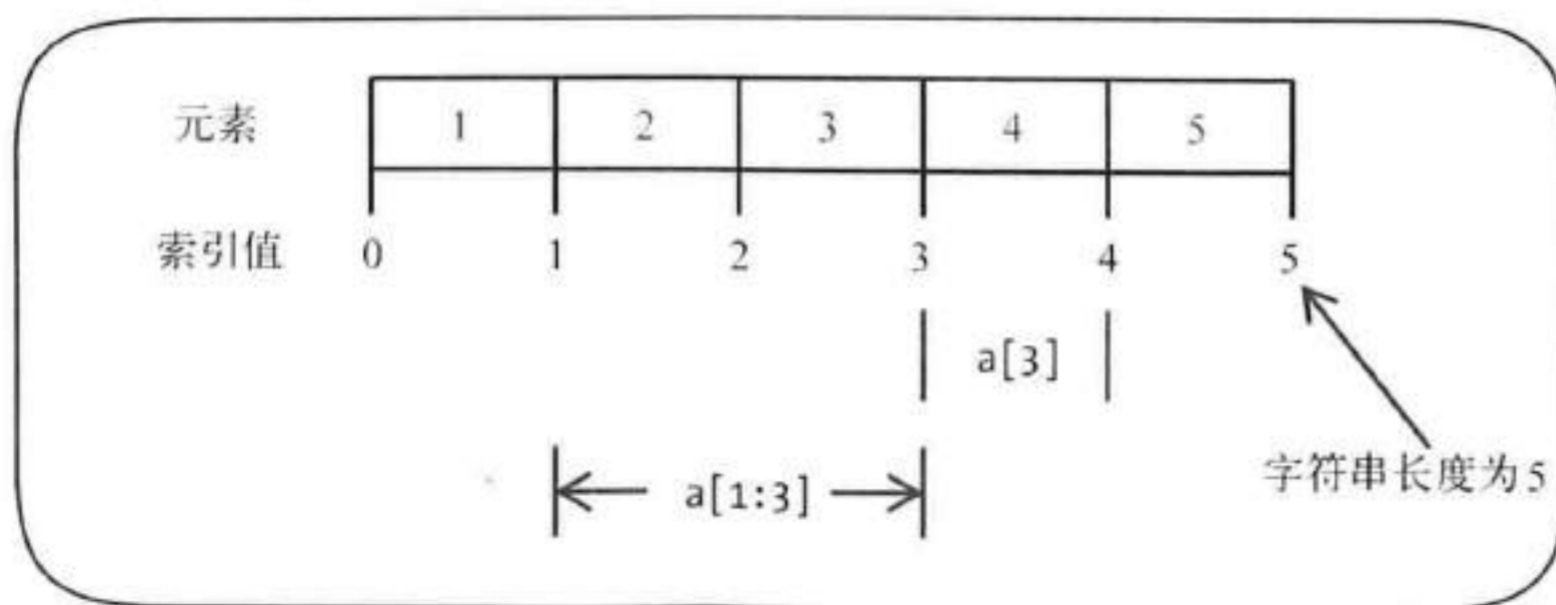


图3-1 切片与索引值

图中示意了元素类型为int的切片类型值[]int{1, 2, 3, 4, 5}中的元素与索引值的对应关系。可以看到，索引出现在了切片类型值的第一个元素的左边、最后一个元素的右边，以及每对相邻

元素的中间位置上。最左边的索引的值为0，并与第一个元素相对应。相应地，紧挨在每个索引右边的元素就是与该索引对应的那个元素，除了最后一个索引。实际上，没有与最后一个索引对应的元素。但是，这个索引的值恰恰是该切片类型值的长度。因此，索引表达式`a[3]`的求值结果为该切片类型值的第四个元素4。然而，切片表达式中的索引值的作用略有不同。切片表达式中的元素下界索引和元素上界索引精确地限定了切片操作的范围。具体来说，切片表达式`a[1:3]`意味着索引1到索引3之间的所有元素都会被截取出来并构成一个新的切片类型值，即`[]int{2, 3}`。因此，与元素下界索引对应的（或者说紧挨在元素下界索引右边的）那个元素永远不会被包含在这个新生成的切片类型值中。这也意味着，对于这个作为切片表达式求值结果的值来说，其长度永远等于元素上界索引与元素下界索引的差。

对于一个字符串类型的值来说，上述规则也同样适用，如图3-2所示。

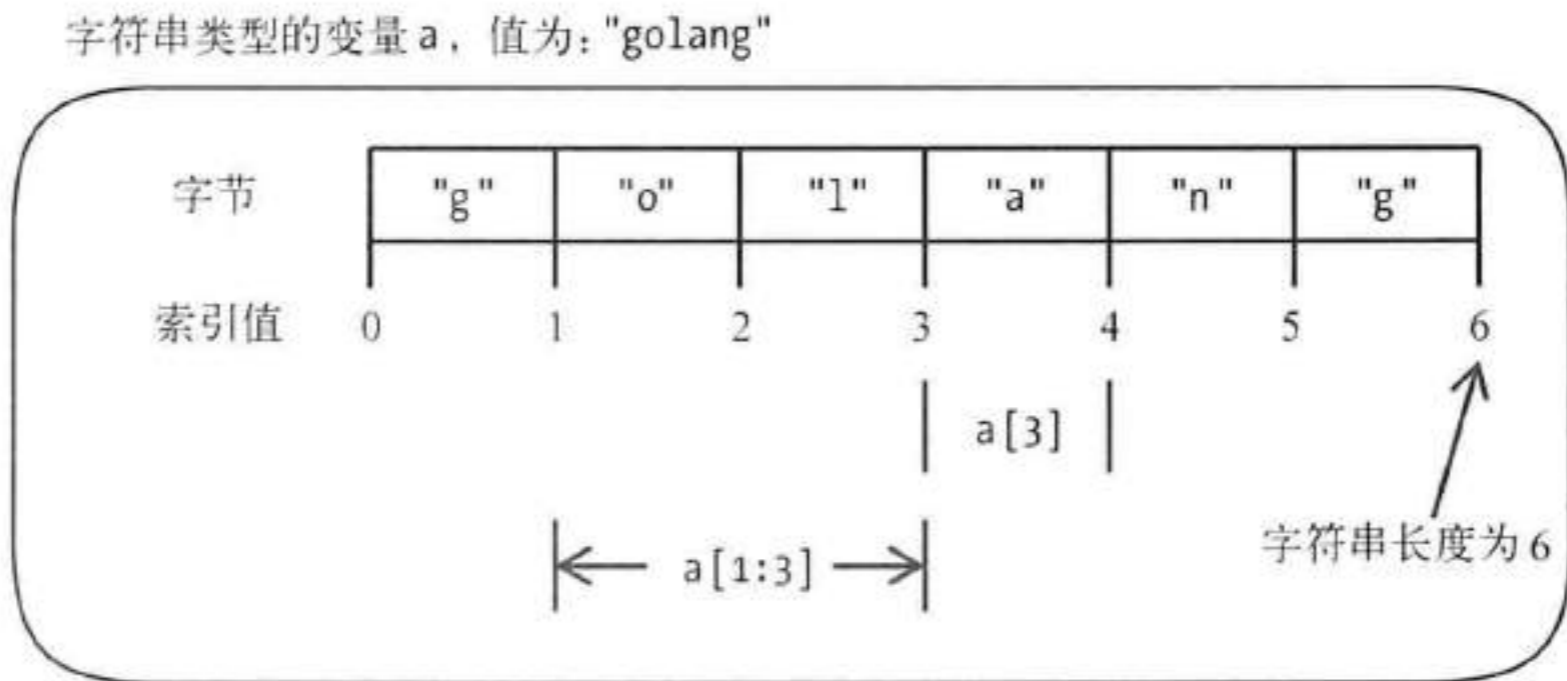


图3-2 字符串与索引值

对于值为"golang"的字符串类型的变量a来说，索引表达式`a[3]`的求值结果为字节类型值97。由于字符串类型值"golang"中的每个字符都是单字节字符，因此这个字节类型值也就是"golang"中的第四个字符"a"的UTF-8编码值的十进制表示形式。此外，针对一个字符串类型值的切片表达式的求值结果也会是一个字符串类型的值。具体地说，切片表达式`a[1:3]`的求值结果是字符串类型值"ol"。当然，只有当在元素上界索引左边的字符全部为单字节字符时，切片表达式的结果才能够与相应位置的字符对应起来。举个例子，对于值为"Go并发编程实战"的字符串类型的变量a来说，切片表达式`a[1:3]`的求值结果就是字符串类型值"o0"。这显然与我们直觉上的结果不太一致。这是因为，UTF-8编码格式会以3个字节来表示一个中文字符，而切片操作是针对字节进行的。切片表达式`a[1:3]`意味着截取了字符串类型值"Go并发编程实战"中的第二个字节和第三个字节。其中，第二个字节即代表了单字节字符"o"，而第三个字节则是三字节字符"并"中的第一个字节。切片操作将截取出的多个字节类型值转换为一个字符串类型值，并将其作为最终的结果值。由于被截取出的第二个字节属于非法的UTF-8编码值，所以被转换成了字符"0"。根据上面的描述，如果我们这样编写切片表达式：`a[1:5]`，那么得到的求值结果中就会包含一个完整的中文字符。切片表达式`a[1:5]`的求值结果是字符串类型值"o并"。

好了，我们已经了解了关于切片表达式的很多细节。下面介绍与切片符号有关的两个语法糖。

同样以元素类型为`int`的切片类型的变量`a`为例, 设`a`的值为`[]int{1, 2, 3, 4, 5}`。切片表达式`a[:3]`的含义是截取在索引3之前的(左边的)所有元素、构成新的元素类型为`int`的切片类型值, 并将其作为求值结果。这等同于切片表达式`a[0:3]`。这是因为切片符号中的元素下界索引的默认值为0。当我们未在元素下界索引的位置上插入任何值的时候, Go语言会使用0作为元素下界索引。对应地, 元素上界索引的默认值为操作对象的长度值或容量值。因此, 当我们未在元素上界索引的位置上插入任何值的时候, Go语言会使用操作对象的长度值作为元素上界索引。例如, 切片表达式`a[3:]`的含义是截取在索引3之后的(右边的)所有元素、构成新的元素类型为`int`的切片类型值并将其作为求值结果。对于切片符号的操作对象`[]int{1, 2, 3, 4, 5}`来说, 这等同于切片表达式`a[3:5]`。

最后, 如果`a`代表的是一个切片类型的值, 那么切片表达式`a[:]`就等同于复制`a`所代表的值并将这个复制品作为此表达式的求值结果。否则, 切片表达式`a[:]`就意味着将会有有一个包含了指向`a`的第一个元素的指针的切片类型值被创建。关于这一区别, 我们会在3.2.3节讲切片类型的时候详细说明。

5. 类型断言

对于一个求值结果为接口类型值的表达式`x`和一个类型`T`, 对应的类型断言为:

`x.(T)`

这也属于表达式的一种。其作用是判断“`x`不为`nil`且存储在其中的值是`T`类型的”这一假设是否成立。

如果`T`不是一个接口类型, 那么`x.(T)`将会判断`x`的动态类型是否与类型`T`一致。也就是说, 这是一个关于“类型`T`是否为`x`的动态类型”的判断。我们在本节讲类型的时候已经对动态类型有过说明。简单来说, 一个变量的动态类型就是在运行期间存储在其中的值的实际类型。这个实际类型必须是该变量所声明的那个类型的一个实现类型, 否则就根本不可能在该变量中存储这一类型的值。因此, 在我们当前的上下文中, 类型`T`必须是`x`的类型的一个实现类型。又由于在Go语言中只有接口类型可以被其他类型实现, 所以`x`的求值结果必须是一个接口类型的值。举个例子, 表达式`int(123).(int)`会引发一个编译错误, 相应的提示信息是

```
invalid type assertion: 123.(int) (non-interface type int on left)
```

其大意是: 表达式`int(123)`的求值结果是`int`类型的, 而`int`类型并不是一个接口类型。现在, 我们把这个表达式改写为`interface{}(123).(int)`。这个表达式会顺利通过编译, 因为表达式`interface{}(123)`的含义是将字面量123转换为`interface{}`类型的值, 它的结果值是接口类型的。这正好符合上面的要求。实际上, `interface{}`是一个特殊的接口类型, 代表空接口。所有类型都是它的实现类型。我们在3.2.6节讲接口类型的时候还会接触到它。

如果`T`不是一个接口类型且类型`T`不是`x`的类型的一个实现, 那么类型断言`x.(T)`就是失败的, 或者说它所判断的假设就是不成立的。这会引发一个运行时恐慌。例如, 表达式`interface{}(uint(123)).(int)`在被求值时就会引发一个运行时恐慌, 且相应的错误提示信息是

```
interface conversion: interface is uint, not int
```

这是因为123在被转换为接口类型之前是uint类型的，而不是int类型的。

对于另一种情况，即T是一个接口类型，那么表达式x.(T)将会判断x的动态类型是否实现了接口类型T。这一判断内容几乎与上一种情况相反，且更容易被理解。在这种情况下，x的类型是否为一个接口类型并不重要，重要的是x的类型是否完全定义或实现了接口类型T中所定义的方法集合。关于接口类型、实现和方法集合方面的知识，请参看3.2.6节。

无论属于哪种情况，如果断言成功就说明x的结果值（对于一个变量x来说，其结果值就是存储在其中的那个值）的类型就是T或者它的实现类型。否则，在这个类型断言表达式被求值时就会引发一个运行时恐慌。注意，只有在程序运行期间，x的动态类型才能够被获知，而在编译期间能够确定的只有T所代表的类型。这也是在程序运行期间才能够确定类型断言是否成功的原因。

类型断言可以被用在很多地方。在对变量的赋值或初始化的时候，我们也可以使用类型断言，例如：

```
v, ok := x.(T)
```

在这条赋值语句中使用了一种特殊用法。当使用类型断言表达式同时对两个变量进行赋值时，如果类型断言成功，那么赋给第一个变量的将会是已经被转换为T类型的表达式x的求值结果，否则赋给第一个变量的就是类型T的零值。但是，更关键的地方在于布尔类型的第二个结果。在这里，它会被赋给变量ok。ok的值体现了类型断言的成功（此时值为true）与否（此时值为false）。注意，在这种使用场景下，即使类型断言失败也不会引发运行时恐慌。

6. 调用

如果有函数类型F的值f，那么表达式f(a1, a2, a3)就表示了对函数f的调用，同时会以a1、a2和a3作为参数传递给函数f。这其中的参数的数量、排列顺序和各自的类型都由函数类型F的声明来决定。在调用时，每个参数都可以用一个表达式来代表，但前提条件是该表达式的求值结果只有一个，并且其类型即是对应的参数的类型，或是该类型的实现类型。表达式f(a1, a2, a3)的类型即是F定义中的结果（确切地讲，是返回参数）的类型。与此对应，我们之前常说的一个表达式的类型实际上就是该表达式的求值结果的类型。

方法可以说是函数的一种。简单地说，方法比函数多了一个接收者。这个接收者可以是该方法所属的结构体类型的值，或者与该结构体类型对应的那个指针类型的值。在这种情况下，我们在调用一个结构体的方法的时候，必须先要声明一个该结构体类型的变量，或者初始化一个该结构体类型的值，然后再在这个变量或者值之上，使用前面讲过的选择符号来调用其拥有的方法。例如，有一个无参数的方法m，它所属的结构体的类型是S，并且有该结构体类型的变量s，那么表达式s.m()就是对方法m的调用。注意，s.m()有效的前提条件是，在类型S的方法集合中必须包含方法m，并且参数（如果有的话）的数量、排列顺序和各自的类型也需要完全对应。如果x是可寻址的，并且与类型S对应的那个指针类型的方法集合中包含了方法m（名称和参数的列表都要完全对应），那么调用表达式s.m()也是有效的。在这种情况下，s.m()即是调用表达式(&s).m()的速记法。

在函数或方法调用表达式中，函数值和参数是以通常的顺序被求值的。在这里，我们先重点

解释一下“通常的顺序”这个词。

“通常的顺序”意味着在求值一个表达式、赋值语句或者返回语句中包含的操作数的时候，所有的函数调用、方法调用和通信操作（由接收操作符及其操作数组成）都会按照词法上的从左到右的顺序被求值。例如：

```
y[f()], ok = g(h(), i()+x[j()], <-c), k())
```

在这一赋值语句中，函数调用和通信操作的求值顺序如下：`f()`、`h()`、`i()`、`j()`、`<-c`、`g()`、`k()`。注意，Go语言并不会对它们（函数调用、方法调用和通信操作）与其他表达式之间以及其他表达式之间的求值顺序作出任何保证。例如，在上面的示例中，我们无从知晓针对`y`和`x`的索引表达式会在什么时候被求值。又例如：

```
a := 10
f := func() int { a = a * 2; return 5 }
x := []int{a, f()}
```

在上面的示例中，我们将整数字面量赋给了变量`a`，然后将一个匿名函数（由函数字面量表示）赋给了变量`f`。最后，将一个元素类型为`int`的切片类型值赋给了变量`x`。注意，这个切片类型值中包含了两个元素，即是在前面被声明和初始化的变量`a`和`f`。根据上面所说的“通常的顺序”规则，变量`x`的值可能是`[]int[10, 5]`，也可能是`[]int[20, 5]`。这是因为`a`和`f()`之间的求值顺序并不固定。

此外，在默认情况下，在单一表达式中的浮点数操作的求值顺序仅依赖于操作符的优先级顺序。不过，我们可以显式地使用圆括号来改变默认的求值顺序。

当函数或方法调用表达式中所有作为参数的标识符或表达式都被求值之后，它们的值会被传递给函数。而后，被调用的那个函数开始执行。当函数执行完毕并返回时，其返回参数（即结果）的值就会被传给调用该函数的代码。

最后，值得一提的是，作为函数调用表达式的一个特例，我们可以把一个函数或者方法的结果直接作为参数传递给另一个函数或者方法。例如：

```
f(g(a1, a2, a3))
```

此表达式表明，在调用函数`g`之后，把它的所有结果都作为参数传递给了函数`f`。为了让上面的表达式合法，需要满足下面几个条件。

- ❑ 函数`g`的结果与函数`f`的参数在数量上必须相同。并且按照从左到右的顺序，函数`g`的结果必须能够分别赋值给函数`f`的参数。也就是说，在对应的位置上，函数`g`的结果的类型需要与函数`f`的参数的类型一致，或者为函数`f`的参数的类型的一个实现。
- ❑ 对函数`f`的调用表达式中除了函数`g`的调用表达式之外不能再出现其他参数，即`f(a0, g(a1, a2, a3))`和`f(g(a1, a2, a3), a4)`都是不合法的。
- ❑ 函数`g`必须至少有一个结果。
- ❑ 如果函数在参数列表的最后是一个可变长参数，那么Go语言会在把函数`g`的结果作为函数`f`的普通参数传递给`f`之后，再尝试将剩余的结果（如果有的话）传递给函数`f`的可变长参数。

最后，在Go语言中没有单独的方法类型和方法字面量，因此方法调用表达式除了需要满足上述的规则和约束之外再无其他。

7. 可变长参数

如果函数f可以接受的参数的数量是不固定的，那么函数f就是一个能够接受可变长参数的函数，简称为可变参函数。在Go语言中，在可变参函数的参数列表的最后总会出现一个可变长参数，这个可变长参数的类型声明形如...T。可变长参数可以用于接受数量不定但类型均为T或其实现类型的参数值。它等同于一个元素类型为T的切片类型的参数。对于函数f的每一次调用，被传递给可变长参数的值实际上都是包含了实际参数、元素类型为T的切片类型值。Go语言会在每次调用函数f的时候创建一个这样的切片类型值，并用它来存放这些实际参数。这个切片类型值的长度就是在当前调用表达式中与可变长参数绑定的实际参数的数量。下面举个例子，如果可变参函数appendIfAbsent的声明如下（由于篇幅原因，在这里省略了它的函数体）：

```
func appendIfAbsent(s []string, t ...string) []string
```

那么就可以这样编写针对它的调用表达式：

```
appendIfAbsent([]string{"A", "B", "C"}, "C", "B", "E")
```

其中，与可变长参数t绑定的切片类型值为[]string{"C", "B", "E"}。其中包含了实际参数"C"、"B"和"E"。

我们也可以直接把一个元素类型为T的切片类型值赋给...T类型的可变长参数。为此，我们需要在欲赋予可变长参数的那个切片类型值（或者代表这个值的变量）的后面追加一个特殊符号...。例如，我们可以把上面的调用表达式改为：

```
appendIfAbsent([]string{"A", "B", "C"}, []string{"C", "B", "E"}...)
```

或者，如果有一个元素类型为string的切片类型的变量s的话，那么就是这样：

```
appendIfAbsent([]string{"A", "B", "C"}, s...)
```

对于将切片类型的变量赋给可变长参数的情况，Go语言不会专门创建一个切片类型值来存储其中的实际参数。因为，这样的切片类型值已经存在了，正如我们刚刚看到的那样，可变长参数t的值就是变量s的值。

至此，我们已经对Go语言中的绝大多数表达式进行了介绍和说明。我们在编写Go语言代码的时候离不开表达式。读者在本书的后续部分也会经常看到它们的身影。它们一般会作为各种Go语言语句的组成部分。另外，我们在后面讲复合数据类型的时候，还会对可作为表达式的一些类型字面量和复合字面量进行一番探讨。

3.2 数据类型

在本节，我们会集中精力学习Go语言的数据类型的概念、表现形式和各种操作方式。本节会涉及Go语言中除Channel类型之外的绝大多数数据类型。此外，在本节的最后，我们还会讲到初始化这些数据类型值的各种方法。

3.2.1 基本数据类型

Go语言的基本数据类型并不多，并且大部分都与整数相关。Go语言把整数进行了比较细致的划分。下面我们通过表3-4来概览一下Go语言定义的所有基本数据类型。

表3-4 Go语言的基本数据类型

名 称	宽度(字节)	零值	说 明
bool	1	false	布尔类型。其值不为真即为假。真用常量true表示，假由常量false表示
byte	1	0	字节类型。它也可以看作是一个由8位二进制数代表的无符号整数类型
rune	4	0	rune类型。它是由Go语言定义的特有的数据类型，专用于存储Unicode编码。它也可以被看作是一个由32位二进制数代表的无符号整数类型
int/uint	-	0	有符号整数类型/无符号整数类型。其宽度与平台有关
int8/uint8	1	0	由8位二进制数代表的有符号整数类型/无符号整数类型
int16/uint16	2	0	由16位二进制数代表的有符号整数类型/无符号整数类型
int32/uint32	4	0	由32位二进制数代表的有符号整数类型/无符号整数类型
int64/uint64	8	0	由64位二进制数代表的有符号整数类型/无符号整数类型
float32	4	0.0	由32位二进制数代表的浮点数类型
float64	8	0.0	由64位二进制数代表的浮点数类型
complex64	8	0.0	由64位二进制数代表的复数类型。它由float32类型的实部和float32类型的虚部联合表示
complex128	16	0.0	由128位二进制数代表的复数类型。它由float64类型的实部和float64类型的虚部联合表示
string	-	""	字符串类型。一个字符串类型代表了一个字符串值的集合。而一个字符串值实质上是一个字节序列。注意：字符串类型的值是不可变的，即一旦创建其内容就不可被改变

上表列出了Go语言的全部18种基本数据类型。下面我们分别进行说明。

1. 布尔类型

布尔类型代表了布尔真值的集合。简单来说，布尔真值用于指示一个陈述在什么程度上是真的。在最简单且使用最广泛的情形中，布尔真值非真即假。因此，在Go语言中，布尔真值的集合也只有两个元素，由预定义标识符true和false来表示。布尔类型的值只可能是它们中的一个。Go语言使用bool来表示布尔类型。布尔类型的值也简称为布尔值。

2. 数值类型

在Go语言中，可以代表数值的基本类型众多。它们的不同基本上仅仅体现在其值所用的字节数量和代表名称的标识符上。我们也把为了存储某个类型的值而需要使用的比特/字节的数量称为这个类型的宽度。为了避免可移植性问题，几乎所有的Go语言数值类型的宽度都被直接体现在了它的名称上。比如，我们从名称上就可以获知数值类型int8和uint16的宽度。这些数值类型名称最后面的数字即代表了这个类型所使用的比特(bit)的数量，而1个字节等于8个比特。因此，int8类型的值需要使用1个字节，而uint16类型的值需要使用2个字节。

不过这也有例外，代表字节的类型`byte`以及专用于存储Unicode编码字符的类型`rune`在名称中并未体现出它们所用的字节数量。但实际上我们可以把它们看作其他数值类型的别名类型。类型`byte`可以被看作类型`uint8`的别名类型，而类型`rune`可以被看作是`uint32`的别名类型。此外，类型`int`和`uint`的名称中也没有任何关于宽度的信息。因为它们的宽度并不是唯一的。在386计算架构下，它的宽度为32比特，即4个字节。在amd64（有时也称为x86-64）计算架构下，它们的宽度为64比特，即8个字节。

注意，虽然Go语言一直标榜`int`类型和`uint`类型的实际宽度会根据计算架构的不同而不同，但在其1.0版本或更早版本中，类型`int`和`uint`在所有计算架构下都是由4个字节表示的。直到Go语言的1.1版本，在Go语言官方的编译器（`gc`）以及gcc的Go语言编译器（`gccgo`）中才真正使得这两个类型在amd64计算架构下的宽度为8个字节。

Go语言用数值类型的宽度来区分它们。显然，宽度这个属性对于一个数值类型来讲非常重要。至少对于Go语言是这样。那么这些宽度意味着什么呢？我们先来看表3-5。

表3-5 数值类型宽度的含义

字节（byte）数	比特（bit）数	数值范围
1	8	8位无符号二进制数可以表示的数值的范围是0~255。8位有符号二进制数可以表示的数值的范围是-128~127
2	16	16位无符号二进制数可以表示的数值的范围是0~65535。16位有符号二进制数可以表示的数值的范围是-32768~32767
4	32	32位无符号二进制数可以表示的数值的范围是0~4294967295，约等于从0到42.94亿。32位有符号二进制数可以表示的数值的范围是-2147483648~2147483647，约等于从-21.47亿到21.47亿
8	64	64位无符号二进制数可以表示的数值的范围是0~18446744073709551615，约等于从0到1844亿亿。64位有符号二进制数可以表示的数值的范围是-9223372036854775808~9223372036854775807，约等于从-922亿亿到922亿亿

从表3-5我们可以看到，宽度每翻一番，都会使数值类型的可表示范围存在若干指数级的增长。并且，Go语言又把不同宽度的数值类型分为了有符号和无符号的。相对于其他编程语言（比如Java语言和Python语言）来讲，Go语言对数值类型进行了更加细致的划分。这主要是因为Go语言自称是一个用于系统编程的通用编程语言。它更加关注对系统资源的高效利用。Go语言希望编程人员能够根据实际情况选用匹配度最高的数值类型。这需要对数值类型的当前使用场景进行更细致的评估。有时候，这使我们不得不在最大数值范围和最小资源占用之间进行权衡和妥协。这明显是系统级编程语言的特征之一。不过，Go语言也为我们提供了第二种选择。我们可以直接使用`int`类型或`uint`类型，而无需为选择哪一种宽度的数值类型伤脑筋。虽然这可能会造成一定的系统资源浪费，但是也让我们可以把精力集中到程序设计本身上去。在实际编程过程中，我们往往会根据实际需要，混合使用这些数值类型。

下面我们来讨论数值类型的表示法。这里依然会使用“字面量”这个词。在这之前，我们提到过类型字面量。而这里的字面量狭义地指代一种表示值的标记方法。

在Go语言中，整数可以由整数字面量表示。具体的表示方法有3种：十进制表示法、八进制

表示法和十六进制表示法。我们用以“0”为前缀的字面量表示八进制的整数，而用以“0x”或“0X”为前缀的字面量表示十六进制的整数。另外，为了表示十六进制的整数，我们使用从“a”到“f”（或从“A”到“F”）这6个英文字母来代表从10到15的数字。比如，十进制字面量56代表整数56，八进制字面量056代表整数46，而十六进制字面量0x56则代表整数86。再比如，整数78若分别使用十进制字面量、八进制字面量和十六进制字面量表示，则为78、0116和0x4e。

在Go语言中，与浮点数对应的数值类型有float32和float64。这两种类型分别用4个字节和8个字节的二进制数来代表浮点数。这两种类型的值都可以由浮点数字面量来表示。通常，一个浮点数字面量由整数部分和小数部分组成。在这两部分之间需要用小数点（英文句点“.”）分隔。浮点数字面量的整数部分和小数部分均由十进制数组成，比如56.78。

另外，在浮点数字面量中还可以添加指数部分。指数部分一般由“e”或“E”后跟一个带有正负号的十进制数表示，比如“E+2”和“e-3”。指数部分应该被放在浮点数字面量的最后（最右边）。其含义是把“e”或“E”左边的数值乘以10的N次方或者除以10的N次方。其中N由“e”或“E”右边的十进制数代表，而这个十进制数的正负号决定了需要执行的是乘法还是除法。比如，浮点数字面量12E+2代表浮点数1200.0（12乘以10的2次方），而浮点数字面量12e-3则代表浮点数0.012（12除以10的3次方）。

浮点数字面量还有若干简写方法。当一个浮点数的小数部分为0时，我们可以把这个0省略掉。比如，浮点数1200.0的浮点数字面量可以被简写为1200.。更进一步地，我们还可以把小数点省略掉，即1200。如果一个浮点数的整数部分为0，我们也可以把这个0省略掉。比如，浮点数0.012的浮点数字面量可以被简写为.012。但要注意，这里的小数点就不能再简化了。

这里强调一点，浮点数字面量中的各个部分只能由十进制数表示，而不能由八进制数和十六进制数表示。也就是说，浮点数字面量056.78和56.78都代表浮点数56.78。

现在我们来说说复数。复数可以由Go语言的类型complex64和complex128代表。complex64类型值的实部和虚部各由一个float32类型值表示，而complex128类型值的实部和虚部各由一个float64类型值表示。复数类型的值可以由复数字面量表示。实部和虚部的浮点数的表示方法即为浮点数字面量。在表示虚部的浮点数字面量的最后（最右边）需要追加小写字母“i”。在实部和虚部之间需要由加号“+”分隔。当一个复数的实部或虚部为0时，可以在其复数字面量表示中省略为0的部分。例如，复数字面量12e+2 + 43.4e-3i、0.1i和1E3都是合法的。它们分别代表了复数1200+0.0434i、0+0.1i和1000+0i。

最后，我们来讨论Go语言的一个特有的数值类型rune的值的表示方法。类型rune的值由rune字面量代表。rune字面量可以表示一个rune类型的常量。我们在本节开始处的表格中提到过，rune类型专用于存储经过Unicode编码的字符。因此，一个rune常量即是一个Unicode编码值。Unicode编码值也可以被叫作Unicode代码点。Unicode代码点的惯用表示方式是使用十六进制表示法来表示与它对应的数字值，并使用“U+”作为前缀。比如，英文字母字符“A”的Unicode代码点就是U+0041。一个rune字面量由外层的单引号和内层的一个或多个字符组成。在包裹字符的单引号中不能出现单引号“'”和换行符“\n”。这样的—个rune字面量就可以由与它对应的Unicode代码点来表示。

更确切地说，我们可以用5种方式来表示一个rune字面量，具体如下。

- ❑ 该rune字面量所对应的字符。比如：'a'、'ä'或'—'。当然，这个字符必须是Unicode编码规范所支持的。
- ❑ 使用“\x”为前导并后跟两位十六进制数。这种方式可以表示宽度为一个字节的值，即一个ASCII编码值。
- ❑ 使用“\”为前导并后跟三位八进制数。这种表示法也只能表示有限宽度的值，即它只能用于表示对应数值在0和255之间的值。因此，它与上一个表示法的表示范围是一致的。
- ❑ 使用“\u”为前导并后跟四位十六进制数。它只能用于表示两个字节宽度的值。这种方式即为Unicode编码规范中的UCS-2表示法。不过，UCS-2表示法在不久之后就会被废止。
- ❑ 使用“\U”为前导并后跟八位十六进制数。这种方式即为Unicode编码规范中的UCS-4表示法。UCS-4表示法已经成为Unicode编码规范和相关国际标准中的规范编码格式。

这些表示法会分别用于不同情况下的rune字面量的表示。这些表示法虽然都可以把一个rune字面量表示为一个整数，但是它们却有不同的表示范围。正如上面描述的那样。

我们在本节开始处说过，Go语言的所有源代码都必须由Unicode编码规范的UTF-8编码格式进行编码。UTF-8编码格式是一种可变长度的Unicode编码方式。它可以用来对Unicode编码规范支持的任何字符进行编码。根据字符的不同，一个字符可以被UTF-8编码格式编码为1到4个不等的字节。这些字节序列都可以表示为一个或多个整数值。例如，英文字符“A”会被UTF-8编码格式编码为一个字节。其Unicode代码点是U+0041，也就是rune字面量'\u0041'。中文字符“一”会被UTF-8编码格式编码为3个字节（0xE4、0xB8和0x80）。其Unicode代码点是U+4E00，即为rune字面量'\u4E00'。

rune字面量可以支持一类特殊的字符序列——转义符。转义符的表示方式是在“\”后面追加一个特定的单字符，参见表3-6。

表3-6 转义符说明

转义符	Unicode代码点	说 明
\a	U+0007	告警铃声或蜂鸣声
\b	U+0008	退格符
\f	U+000C	换页符
\n	U+000A	换行符
\r	U+000D	回车符
\t	U+0009	水平制表符
\v	U+000B	垂直制表符
\\	U+005C	反斜杠
\'	U+0027	单引号。仅在rune字面量中有效
\"	U+0022	双引号。仅在string字面量中有效

注意，在rune字面量中，除了在上面表格中出现的转义符之外的以“\”为前导的字符序列都是不合法的。当然，在上表中的转义符“\”也不能在rune字面量中出现。

3. 字符串类型

在Go语言中，字符串类型属于预定义类型。字符串类型代表了一个字符串值的集合。在底层，一个字符串值即是一个字节的序列。长度为0的序列与一个空字符串相对应。字符串的长度即是底层字节序列中字节的个数。一个字符串常量的长度在编译期间就能够确定。

字符串字面量就是上面所说的字符串常量。它代表了一个连续的字符序列。其中，每一个字符都会被隐含地以Unicode编码规范的UTF-8编码格式编码为若干字节。字符串字面量有两种表示格式：原生字符串字面量和解释型字符串字面量。

原生字符串字面量是在两个反引号“```”之间的字符序列。在两个反引号之间，除了反引号之外的其他字符都是合法的。在两个反引号之间的所有内容都看作是这个原生字符串字面量的值。其内容是由若干非解释型字符组成的。所谓非解释型字符，就是在编译期间就可以确定的字符。在原生字符串字面量中，不存在任何转义符，所有的内容都是所见即所得的。这也包括换行符。需要注意的是，原生字符串字面量中的回车符会被编译器移除。

解释型字符串字面量是被两个双引号“`”`”包含的字符序列。解释型字符串中的转义字符都会被成功转义。值得注意的是，在字符串字面量中，转义符“`\`”是不合法的，而转义符“`\\`”却是合法的。这与rune字面量刚好相反。在字符串字面量中可以包含rune字面量。我们可以在字符串字面量中加入任意数量的以各种方式表示的rune字面量。

不过，在这之中也会有一些限制。以3个八进制数（形如“`\101`”）和2个十六进制数（形如“`\x41`”）表示的rune字面量只能用于表示所属字符串字面量中的单字节字符。所谓单字节字符，就是其经过UTF-8编码格式编码后的字节序列的大小为1的字符。由于单字节字符的UTF-8编码值和ASCII编码值是相同的。所以，单字节字符也可以理解为ASCII编码标准所支持的字符。除上述两个表示法之外的其他方法表示的rune字面量都可以用于代表其所属字符串字面量中的单个字符。当然，这样的字符也都是经过UTF-8编码格式编码的，且一个字符可能对应多个字节。

举个例子，在解释型字符串字面量中，rune字面量“`\101`”和“`\x41`”都代表了单字节字符“A”。而rune字面量“`\u4E00`”和“`\U00004E00`”都与Unicode字符“一”相对应。中文字符“一”的Unicode代码点为U+4E00。它会被UTF-8编码格式编码为3个字节，即“`\xE4\xB8\x80`”。

字符串字面量与rune字面量的本质区别是在于它们所代表的Unicode字符的数量上。具体地讲，rune字面量仅用于代表一个Unicode字符，无论这个Unicode字符会被UTF-8编码格式编码为几个字节。而字符串字面量则用于代表一个由若干个Unicode字符组成的序列。

最后需要注意，字符串值是不可变的！也就是说，我们不可能改变一个字符串的内容。我们对字符串的操作只会返回一个新字符串，而不是改变原字符串并返回。

以上，就是我们需要了解的Go语言基本数据类型的相关知识。

3.2.2 数组

一个数组就是一个由若干相同类型的元素组成的序列。在Go语言中，数组被称为Array。

1. 类型表示法

我们在声明一个数组类型的时候需要指明它的长度和元素类型。例如，下面的示例声明了一个长度为n、元素类型为T的数组类型：

```
[n]T
```

可以看到，在声明的左侧的是被方括号括起来的数组长度，而右侧则是数组的元素类型。用于表示数组类型的声明是类型字面量的一种。

注意，数组的长度是数组类型的一部分。只要类型声明中的数组长度不同，即使两个数组类型的元素类型相同，它们也还是不同的类型。例如，数组类型[2]string和[3]string就是两个不同的类型，虽然它们的元素类型都是string。更重要的是，一旦我们在数据类型的声明中确定了它们的长度，就无法在任何时候改变它。也就是说，所有属于这个类型的数组的长度都是固定的。

在数组类型声明中所标识的长度可以由一个非负的整数字面量代表，也可以由一个表达式代表。如果是表达式，那么该表达式的结果值必须是一个int类型的非负值。例如：

```
[2*3*4]byte
```

这个类型字面量表示了一个元素类型为byte的数组类型。在方括号之中的是一个代表了数组长度的表达式。还要注意，这个表达式中只能出现整数字面量和代表了某个常量的标识符。

数组类型声明中的元素类型可以是任意一个有效的Go语言数据类型。也就是说，它可以是一个预定义数据类型、复合数据类型，或者我们自定义的数据类型。甚至，我们还可以把一个类型字面量作为数组的元素类型。例如：

```
[5]struct { name, address string } // "struct { ... }"是用于自定义匿名结构体类型的类型字面量
```

这意味着，虽然数组的元素类型只能是单一数据类型，但是因为这个单一数据类型可以是一个复合数据类型，所以我们可以使用数组构造出更多样的数据结构，而不只是把它当作包含若干相同类型元素的有序列表。

2. 值表示法

数组类型的值（以下简称为数组值）可以由复合字面量来表示。这个复合字面量会由表示数组类型的类型字面量和被花括号“{”和“}”括起来若干代表元素值的字面量或表达式组成，在多个元素值之间使用逗号“,”分隔。例如，字面量：

```
[6]string{"Go", "Python", "Java", "C", "C++", "PHP"}
```

表示了一个长度为6、元素类型为string的数组值，且已包含了6个元素值。

注意，上面的数组值中的每个元素值都会隐含的与一个索引值相对应。这些索引值标示了相应元素值在数组值中的位置。这些索引值一定都是非负的整数。默认情况下，在花括号中的第一个元素值会与索引值0相对应，之后的每个元素值的索引值都是在前一个元素值的索引值的基础上再加1。在上面的示例中，元素值“Go”的索引值是0，而元素值“C”的索引值是3。

我们也可以在编写这类复合字面量的时候指定元素值的索引值。在这种情况下，索引值和对应的元素值是以键值对的形式表示的。索引值为键，元素值为值，且它们之间以冒号“:”分隔，形如0: “Go”。我们可以把上面的复合字面量改写为：

```
[6]string{0: "Go", 1: "Python", 2: "Java", 3: "C", 4: "C++", 5: "PHP"}
```

这个字面量也体现了在默认情况下的各个元素值与索引值的对应关系。

当然，我们也可以打乱它们默认的对对应关系，例如：

```
[6]string{2: "Go", 1: "Python", 5: "Java", 4: "C", 0: "C++", 3: "PHP"}
```

或者，只显式地指定一部分元素值的索引值：

```
[6]string{5: "Go", 0: "Python", "Java", "C", "C++", 4: "PHP"}
```

索引值的指定方式是非常灵活的。但是需要满足下面两个条件。

- 指定的索引值必须在该数组的类型所体现的有效范围之内，即大于等于0并且小于数组类型中声明的长度。在上面的示例中，索引值只能是0、1、2、3、4或5。例如，数组值

```
[6]string{6: "Go", "Python", "Java", "C", "C++", "PHP"}
```

就是不合法的，因为我们为元素值"Go"指定的索引值不在索引值的有效范围之内。此外，需要特别注意，我们指定的索引值也不能导致后续元素值的索引值超出范围。例如，数组值

```
[6]string{"Go", "Python", "Java", "C", 5: "C++", "PHP"}
```

也是不合法的，虽然我们为元素值"C++"指定的索引值5并没有在索引值的有效范围之外。这是因为，根据元素值索引的递增规则推算，元素值"C++"的下一个（紧挨在它右边的）元素值"PHP"隐含对应的索引值是6，这显然超出了索引值的有效范围。

- 指定的索引值不能与其他元素值的索引值重复，不论其他元素值的是隐含对应的还是显示对应的。例如，数组值

```
[6]string{0: "Go", "Python", 1: "Java", "C", "C++", "PHP"}
```

是不合法的。这同样可以通过使用上面的索引值推算规则来检查。我们为元素值"Go"指定了索引值0。根据推算，元素值"Python"的隐含索引值为1。同时，我们又为元素值"Java"指定了索引值1。显然，出现了重复的索引值。这是不允许的。

现在，我们再来看此类复合字面量中用于表示数组值长度的那个整数。与用于表示数组类型的类型字面量相同，这里的整数可以由一个表达式代表，同时也必须符合相关的规则。但是，这里的长度还必须满足一个额外的条件，那就是，它必须大于或等于花括号中元素值的实际数量。例如，下面的数组值是合法的：

```
[8]string{"Go", "Python", "Java", "C", "C++", "PHP"}
```

可以看到，这个复合字面量中表示的数组值长度比元素值的实际数量大。这种情况下，在此数组值中未指定的元素将会被填充为元素类型（这里是string类型）的零值。因此，这个数组值等同于下面的复合字面量：

```
[8]string{0: "Go", 1: "Python", 2: "Java", 3: "C", 4: "C++", 5: "PHP", 6: "", 7: ""}
```

为了更加清楚地体现填充的两个元素值，我们按照索引值推算规则为这个复合字面量中的每个元素值都显式地指定了索引值。可以看到，被填充的两个元素的值均为string类型的零值，且

它们的索引值分别6和7。

当然，我们还可以通过显式地指定索引值来改变被填充元素值的位置。例如，数组值

```
[8]string{1: "Go", "Python", 4: "Java", "C", "C++", "PHP"}
```

等同于

```
[8]string{0: "", 1: "Go", 2: "Python", 3: "", 4: "Java", 5: "C", 6: "C++", 7: "PHP"}
```

总之，当在方括号中的整数值与花括号中元素值的实际数量不同的时候，数组值的长度由前者指定。不过，我们也可以忽略掉这个在方括号中的整数值。请看下面的数组值：

```
[...]string{"Go", "Python", "Java", "C", "C++", "PHP"}
```

在这个复合字面量中的方括号中只有一个特殊标记...。这表示我们在这里并不显式地指定数组值的长度，而让Go语言编译器为我们计算该值所包含的元素值的数量并以此确定这个长度的值。这种情况下，此数组值的长度完全由其中元素值的数量代表。因此，上面这个数组值的长度为6。这种表示方法在我们能够一次性确定数组中的全部元素值的时候是很有用的。它可以避免由于指定的长度和元素值的实际数量不相符而导致的多余零值元素或编译错误。

3. 属性和基本操作

数组类型属于值类型。因此，一个数组类型的变量在被声明之后就会拥有一个非空值。这个值所包含的元素值的数量与其类型中所声明的长度一致，并且其中的每个元素值都是其类型的元素类型的零值。在Go语言中，一个数组即是一个值。数组类型的变量即代表了整个数组，而并不代表一个指向数组的第一个元素值的指针，这有别于C语言中的数组。这就意味着，当我们将一个数组值赋给一个变量或者传递给一个函数的时候，会隐含地创建出此数组值的一个备份。为了避免这种隐含的备份，我们可以通过取址操作符获取到这个数组值的指针，并把这个指针用在变量赋值操作和函数参数传递的操作当中。

我们已经知道，数组值的长度是它所属的类型的一部分。我们可以使用Go语言的内建函数len来获取这个长度。例如，调用表达式

```
len([...]string{"Go", "Python", "Java", "C", "C++", "PHP"})
```

的结果值是6，即为其中的数组值的长度。由此，我们可以判断出这个数组的类型是[6]string。

在本章中，我们还会陆续看到很多Go语言的内建函数。我们会在它们出现时简单地介绍它们的用法。在本章的末尾，我们还会专门开辟一个小节来汇总和说明这些内建函数。

数组值中的每个元素值都有一个对应的索引值，用于标示出元素值的在数组中的位置。我们可以通过索引值访问数组值中的每一个元素。这在我们讲述索引表达式的时候已经有所说明。例如，索引表达式

```
[...]string{"Go", "Python", "Java", "C", "C++", "PHP"}[0]
```

的值就是"Go"，而索引表达式

```
[...]string{"Go", "Python", "Java", "C", "C++", "PHP"}[5]
```

的值则是的值就是"PHP"。再次强调，一个数组值的索引值的有效范围在0和数组类型中声明的长

度再减1的整数之间。

索引值除了可以让我们访问到数组值中对应的元素之外，还可以被用于改变对应的元素。我们首先使用赋值语句将上面示例中的数组值赋给变量array1:

```
array1 := [6]string{"Go", "Python", "Java", "C", "C++", "PHP"}
```

还记得特殊标记:=吗？它被用于在声明一个变量的同时对这个变量进行赋值。现在，我们要把与索引值2对应的元素修改为字符串类型值Clojure。对应的赋值语句如下：

```
array1[2] = "Clojure"
```

在上面的示例中，索引表达式成为了赋值语句的一部分。这条语句被执行之后，变量array1的值就会变更为

```
[6]string{"Go", "Python", "Clojure", "C", "C++", "PHP"}
```

注意，如果变量array1的值为nil，那么索引表达式在被求值时就会引发一个运行时恐慌。而当索引值不在其有效范围的时候，如果索引值由整数字面量代表，或用于表示该索引值的表达式中只包含了整数字面量和代表了某个常量的标识符（也就是说，索引值在编译期间就可以被确定），那么这一索引表达式会在编译期间造成一个编译错误，否则它会在程序运行期间引发一个运行时恐慌。

最后，数组值中元素的顺序会以它们的索引值为依据。索引值越小，对应元素的位置就越靠前。这在我们使用for语句对数组值进行迭代的时候就会体现出来。下一章，我们会对for语句进行详细的说明。

当需要详细的规划程序所用的内存的时候，数组类型是非常有用的。使用数组类型值可以完全避免耗时费力的内存二次分配操作，因为它的长度是不可变的。数组类型是切片类型的根基，数组值也为切片类型值提供了底层支持。我们马上就会讲到切片类型及其值的相关知识。

3.2.3 切片

切片可以看作是对数组的一种包装形式，其官方称谓是Slice。切片包装的数组称为该切片的底层数组。反过来说，切片是针对其底层数组中某个连续片段的描述符。

切片类型为了实现针对其底层数组中某个连续片段的操作提供了比数组类型更加通用、强大和便捷的接口。在编写Go语言代码的过程中，我们一般会使用切片类型值（以下简称切片值）而不是数组值来满足我们对数组的需要，除非确实需要明确的设定长度。

1. 类型表示法

用于表示一个切片类型的类型字面量由一对中间没有任何内容的方括号和代表其元素类型的标识符组成。对于一个元素类型为T的切片类型来说，它的类型字面量就是

```
[]T
```

可以看出，长度并不是切片类型的一部分。它不会出现在表示切片类型的类型字面量中。同时，切片的长度是可变的。因此，相同类型的切片值可能会有不同的长度。

与数组类型声明一致，切片类型声明中的元素类型也可以是任意一个有效的Go语言数据类型。例如，类型字面量

```
[]rune
```

用于表示元素类型为`rune`的切片类型。我们同样可以把一个匿名结构体类型作为切片类型的元素类型：

```
[]struct { name, department string }
```

2. 值表示法

切片值的表示与数组值非常地类似。它也是复合字面量的一种。例如：

```
[]string{"Go", "Python", "Java", "C", "C++", "PHP"}
```

可以看出，在描述所包含的元素的方式上，切片值与数组值毫无区别。唯一的区别就在于其中的类型字面量。上面示例中的切片值的类型为`[]string`。

同样的，切片值中的每个元素都有对应的索引值。它们的特性与数组值中的索引值相同。唯一不同的是，切片值中的索引值只需要满足不出现重复的要求即可，而不再受到切片值长度的限制。因为，在切片值所属的类型中根本就没有关于长度的规定。所以，下面的切片值是合法的：

```
[]string{8: "Go", 2: "Python", "Java", "C", "C++", "PHP"}
```

它等同于下面的复合字面量：

```
[]string{0: "", 1: "", 2: "Python", 3: "Java", 4: "C", 5: "C++", 6: "PHP", 7: "", 8: "Go"}
```

当然，切片值的长度还是需要在`int`类型所能表示的非负值范围之内的。

3. 属性和基本操作

切片类型的零值为`nil`。在初始化之前，一个切片类型的变量值为`nil`。

切片类型中虽然没有关于长度的声明，但是值确实是有长度的。这些切片值的长度准确地体现了它们所包含的元素值的实际数量。我们可以使用内建函数`len`来获取切片值的长度。例如，调用表达式

```
len([]string{8: "Go", 2: "Python", "Java", "C", "C++", "PHP"})
```

的结果值是9。这个切片值实际上包含了6个被明确指定的`string`类型值和3个被填充的`string`类型的零值`""`。

注意，在切片类型的零值（即`nil`）上应用内建函数`len`将会得到0。

除了长度之外，切片值还有一个很重要的属性——容量。在对这个属性进行说明之前，我们先来说说切片值的底层实现方式。一个切片值总会持有一个对某个数组值的引用。事实上，一个切片值一旦被初始化，就会与一个包含了其中元素值的数组值相关联。这个数组值被称为引用它的切片值的底层数组。

多个切片值可能会共用同一个底层数组。例如，如果我们把一个切片值复制成多个，或者针对其中的某个连续片段再切片成新的切片值，那么这些切片值所引用的都会是同一个底层数组。对切片值中的元素值的修改，实质上就是对其底层数组上的对应元素的修改。从这个角度看，切

片值类似于指向底层数组的指针。反过来讲，对作为底层数组的数组值中元素值的改变，也会体现到引用该底层数组且包含该元素值的所有切片值上。

切片值的容量与它所持有的底层数组的长度有关。我们可以使用内建函数`cap`来获取它。例如，调用表达式

```
cap([]string{8: "Go", 2: "Python", "Java", "C", "C++", "PHP"})
```

的结果值是9。在这个特例中，切片值的容量就等于它的长度。但是在很多情况下不会是这样。

这需从切片值的底层数据结构讲起。一个切片值的底层数据结构中包含了一个指向底层数组的指针类型值、一个代表了切片长度的`int`类型值和一个代表了切片容量的`int`类型值。如图3-3所示。

切片值的底层数据结构：

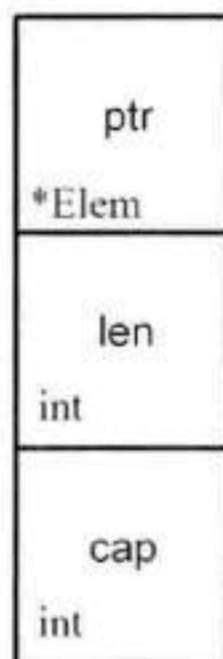


图3-3 切片值的底层数据结构

在切片值中存储着指向其底层数组的指针。这个指针体现了它们之间的引用关系。我们在使用复合字面量初始化一个切片值的时候，首先创建的是这个切片值所引用的底层数组。这个底层数组与这个切片值有着相同的元素类型、元素值及其排列顺序和长度。因此，这时的切片值的长度和容量一定是相同的。

与内建函数`len`一样，对切片类型的零值应用内建函数`cap`也会得到0。

在上一节讲切片表达式的时候我们提到过，可以使用切片表达式从一个数组值或者切片值上“切”出一个连续片段，并生成一个新的切片值。例如：

```
array1 := [...]string{"Go", "Python", "Java", "C", "C++", "PHP"}
slice1 := array1[:4]
```

在这个示例中，变量`slice1`的值的底层数组实际上就是变量`array1`的值，它们的关系如图3-4所示。

再次重申，切片表达式的作用并不是复制数组值中某个连续片段所包含的元素值，而是创建一个新的切片值。在这个切片值中包含了指向这个连续片段中第一个元素值的指针。因此，使用切片表达式从数组值中获取片段的效率是非常高的。我们也常常通过修改切片值中的元素值来改变其底层数组的值。

变量 slice1 的值:

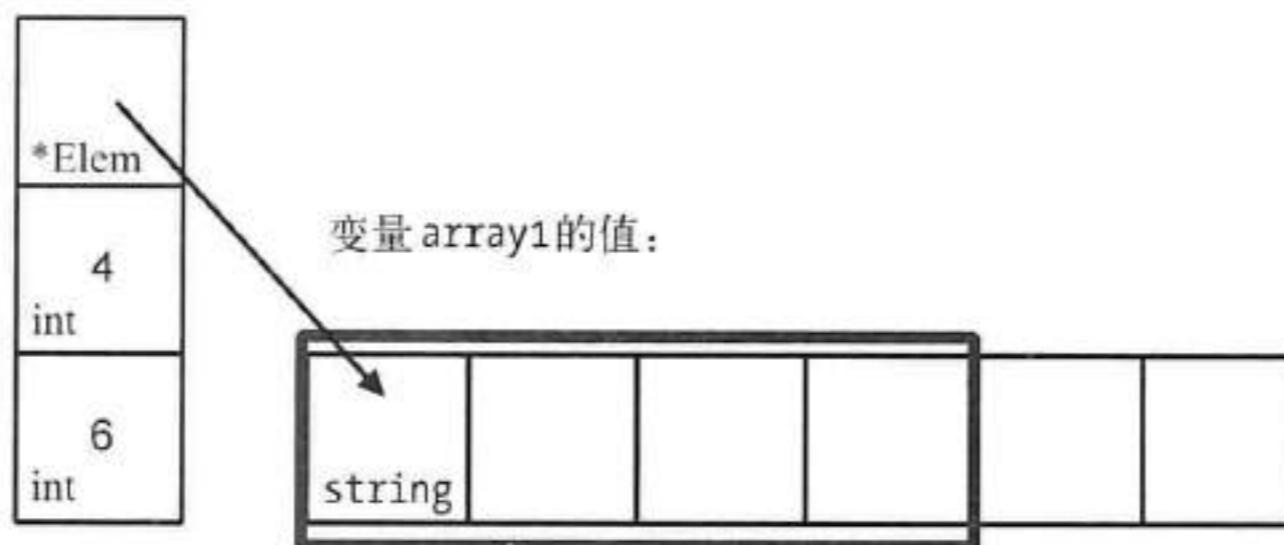


图3-4 在数组值上切出一个切片值1

我们从图3-4中可以获知，变量slice1的值的长度为4、容量为6。很多读者可能会由此假设：一个切片值的容量可能就是其底层数组的长度。但是事实并非如此。为了否定这个假设，我们再创建一个切片值。代码如下：

```
slice2 := array1[3:]
```

变量slice2的值的底层数组也是变量array1的值，它们的关系如图3-5所示。

变量 slice2 的值:

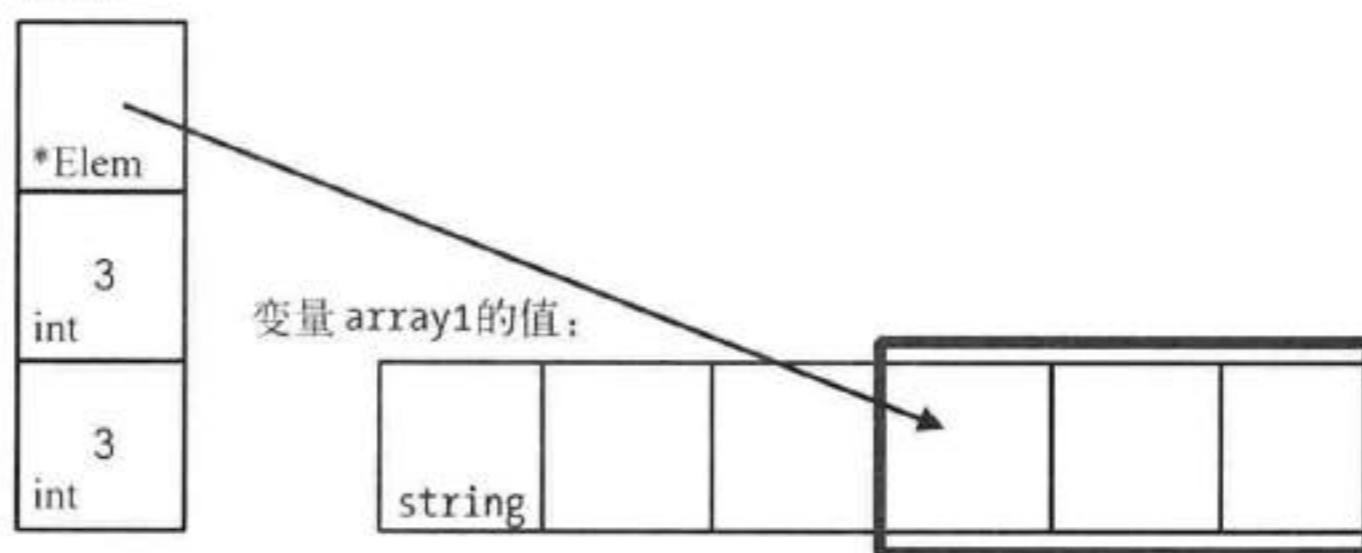


图3-5 在数组值上切出一个切片值2

从图3-5可知，slice2的值的容量与array1的值的长度并不相等。实际上，一个切片值的容量是从其中的指针指向的那个元素值到底层数组的最后一个元素值的计数值。slice2的值中的那个指针指向了array1的值中的第4个元素，而从这个元素到array1的值中的最后元素的元素计数值是3。因此，slice2的值的容量就是3。由此看来，切片值的容量的含义是其能够访问到的当前底层数组中的元素值的最大数量。

我们可以把切片值想象成朝向其底层数组的一个窗口。这个窗口是我们查看底层数组中的元素值的途径。这个值的长度就是我们当前可以看到的底层数组中的元素值的数量，而它的容量则表示了我们最多能够看到多少个当前底层数组中的元素值。

因此，我们可以很方便地对这个窗口进行扩展，以查看更多底层数组元素。但是，我们并不能直接通过再切片的方式来扩展窗口。例如，对于原始的slice1的值来说，索引表达式

```
slice1[4]
```

会引起一个运行时恐慌。因为其中的索引值超出了这个切片值当前的长度，这是不允许的。正确的扩展窗口的方式如下：

```
slice1 = slice1[:cap(slice1)]
```

上面的代码通过再切片的方式把slice1的窗口扩展到了最大，这样就能够看到最多的底层数组元素值了。这时，slice1的值的长度等于其容量。

注意，一个切片值的容量是固定的。也就是说，我们能够看到的底层数组元素的最大数量是固定的。我们不能把切片值的窗口扩展到其容量之外。因此，下面这段代码会引起一个运行时恐慌：

```
slice1 = slice1[:cap(slice1)+1]
```

另外，一个切片值的窗口只能向一个方向扩展。这个方向也就是我们已经在上面演示过的，即索引值递增的方向。因此，我们不能使用再切片的方式把窗口向索引值递减的方向扩展。以变量slice2为例，切片表达式

```
slice2[-2:]
```

是错误的，会引起一个运行时恐慌。记住，与索引值一致，切片值也不允许由负整数字面量代表。

那么我们怎样突破这种限制呢？怎样随意扩展切片值的容量呢？答案是创建一个新的切片值。别担心，切片值的创建成本非常低廉。我们刚刚说过，一个切片值仅包含了一个指针类型值和两个int类型值。我们可以使用Go语言的内建函数append对切片值进行扩展。笼统地讲，append函数会将指定的若干元素值追加到原切片值的末端（有最大索引值的元素值的那一端）。如果需要更大的容量，它还会对原切片值进行扩容。append函数可以接受一个切片类型的参数和一个可变长参数。我们在上一节讲表达式的时候已经介绍过可变长参数。一个可变长参数就是一个切片类型的参数，并且与它绑定的值的数量可以是任意的。append函数的第一个参数应该与将要被扩展的切片值绑定。而它的可变长参数，也就是第二个参数，应该与作为扩展内容的一个或多个元素值绑定，并且这些元素值的类型必须与其第一个参数的元素类型相同。另外，append函数是有结果的。这个结果的类型与其第一个参数的类型完全一致。

我们以被扩展之前的变量slice1为例。这时的slice1的值还只是包含了底层数组array1中排在最前面的（索引值最小的）那4个元素值。现在，我们使用append函数来扩展slice1的值：

```
slice1 = append(slice1, "Ruby", "Erlang")
```

上述语句被执行后，切片类型变量slice1的值及其底层数组（数组变量array1的值）的状态如图3-6所示。

可以看出，slice1的值的长度已经由原来的4增长到了6。这与它的容量是相同的。但是，由于这个值的长度还没有超出它的容量，所以也就没必要再创建一个新的底层数组出来。这时的slice1的值为：

```
[]string{"Go", "Python", "Java", "C", "Ruby", "Erlang"}
```

变量slice1的值:

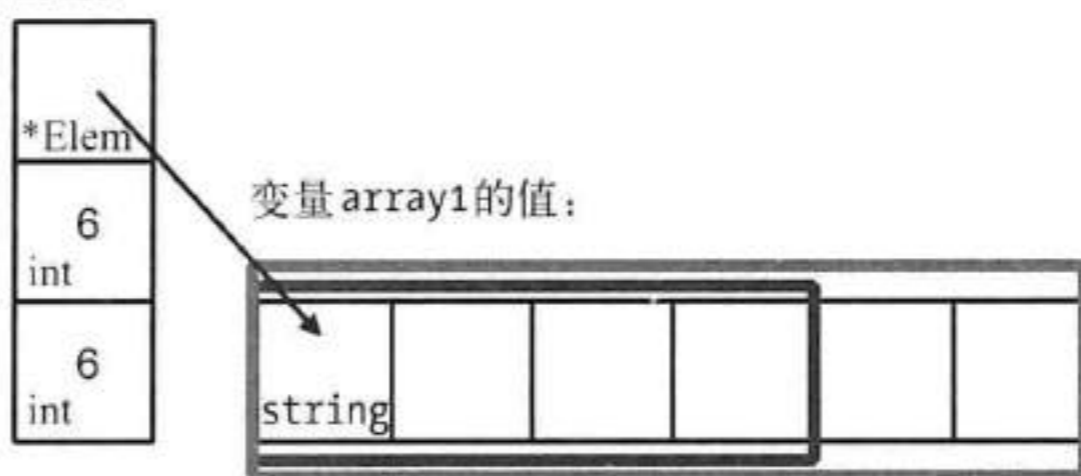


图3-6 扩展切片值1

注意新增的（最右边的）那两个元素值。它们实际上体现的是底层数组中的最右边两个元素值在被改变后的值。也就是说，array1的值中的对应位置上的那两个元素值"C++"和"PHP"已经被变更为了"Ruby"和"Erlang"。现在的array1的值为：

```
[6]string{"Go", "Python", "Java", "C", "Ruby", "Erlang"}
```

记住，切片值相当于朝向其底层数组的一个窗口，它准确地体现了其底层数组中的某个连续片段。它们在对应位置上的元素值在任何时候都是完全一致的。

另外，还有一点需要注意，如果我们想改变slice1的值，那么我们必须将append函数的结果再次赋给变量slice1。举个反例，下面这段代码

```
slice3 := append(slice1, "Ruby", "Erlang")
```

并不会改变的slice1的值，而是声明并初始化了一个新的变量slice3。slice3的值如下：

```
[]string{"Go", "Python", "Java", "C", "Ruby", "Erlang"}
```

注意，array1的值中的第5个和第6个元素同样被改变了。变量slice1、slice3和array1的值之间的关联如图3-7所示。

变量slice1的值:

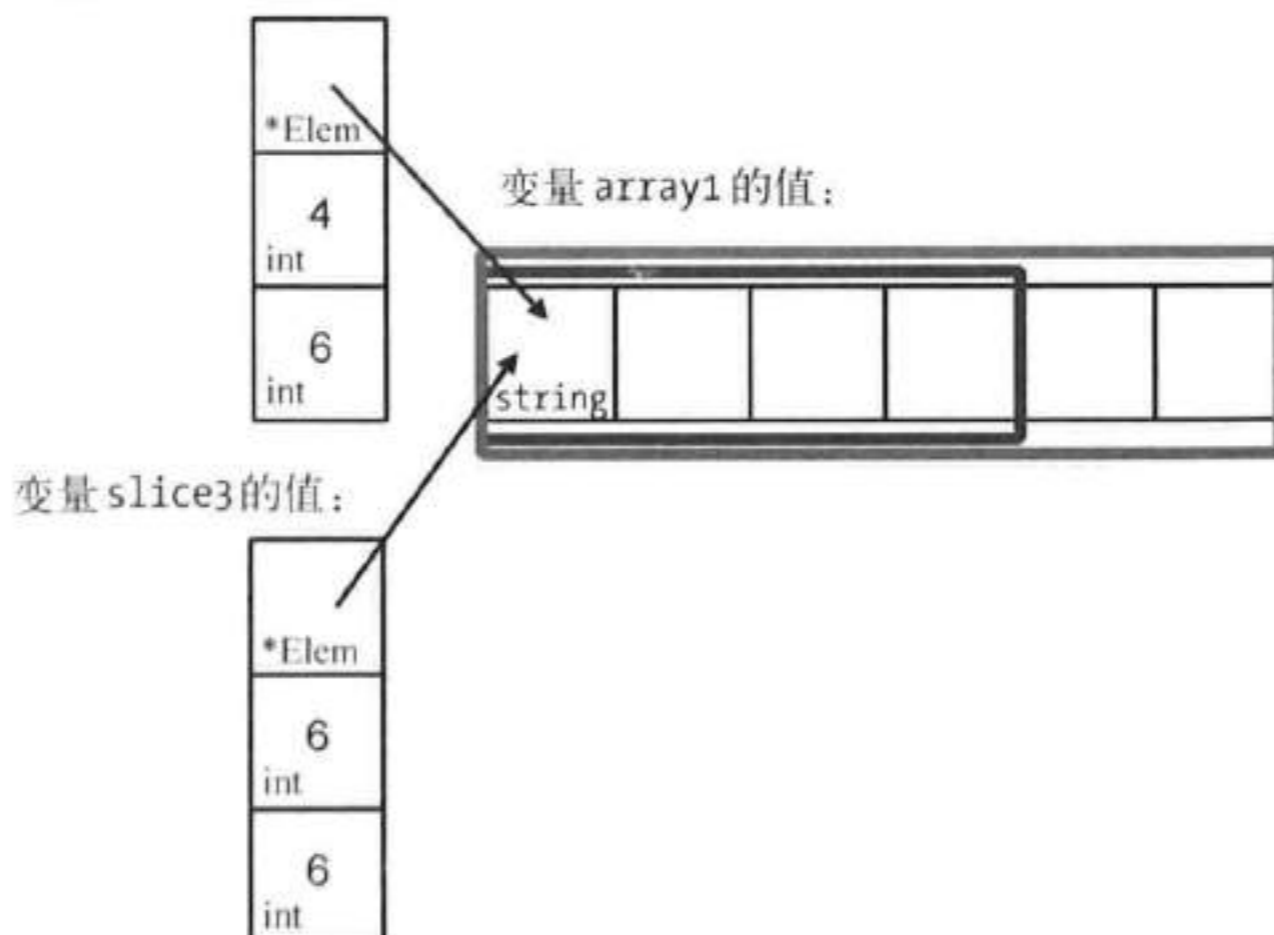


图3-7 扩展切片值2

可以看到，`slice1`的值和`slice3`的值的底层数组是相同的。但不同的是，`slice1`的值的长度依然为4，而`slice3`的值长度为6。`slice3`的值包含了`array1`的值中所有的元素。换句话说，这时的`slice3`的值是`array1`的值的完整体现。

从这个示例可知，`append`函数并不是在原切片值之上进行扩展的，而且是创建了一个新的切片值。在无需扩容的情况下，这个切片值会与原切片值共用一个底层数组，且其中的指针类型值和容量值都会与原切片值保持一致，正如前面讲述的`slice1`和`slice3`的值。而作为扩展内容的“Ruby”和“Erlang”，会被分别绑定到底层数组中的第5个和第6个元素上。这两个元素也正是处在`slice3`的值比`slice1`的值多出的那两个元素位置上。在这些内部的操作完成之后，新创建的切片值被赋给了变量`slice3`。

下面来看看需要扩容的情况。我们忽略掉前面那条包含了变量`slice3`的语句。也就是说，变量`slice1`（已经被扩展了一次）的值为：

```
[ ]string{"Go", "Python", "Java", "C", "Ruby", "Erlang"}
```

再次对变量`slice1`的值进行扩展，代码如下：

```
slice1 = append(slice1, "Lisp")
```

执行这条语句之后，变量`slice1`的值的长度就超出了它的容量。这时将会有一个新的数组值被创建并初始化。这个新的数组值将作为在`append`函数新创建的切片值的底层数组，并包含原切片值中的全部元素值以及作为扩展内容的所有元素值。这个底层数组的长度总是大于需要存储的元素值的总和。新切片值中的指针将指向其底层数组的第一个元素值，且它长度和容量都与其底层数组的长度相同。这与我们直接使用复合字面量来初始化切片值时的内部操作流程有很多相似之处。最后，这个新的切片值会被赋给变量`slice1`。

我们在前面说过，内建函数`append`的第二个参数是一个可变长参数，前面的示例中先后对变量`slice1`的值扩展了一个和两个元素。作为利用可变长参数的另一个示例，我们还可以使用`append`函数把两个元素类型相同的切片值连接起来。例如：

```
slice1 = append(slice1, slice2...)
```

这种将切片值直接传递给可变长参数的方式我们在讲表达式的时候也已经介绍过。当然，我们也可以把数组值作为第二个参数传递给`append`函数。

最后，即使切片类型的变量的值为零值`nil`，也会被看作是长度为0的切片值，所以我们可以 在值为`nil`的切片类型的变量之上应用函数`append`来追加元素值。像这样：

```
slice2 = nil
slice2 = append(slice2, slice1...)
```

或者：

```
var slice4 [ ]string
slice4 = append(slice4, slice1...)
```

上面示例中的第一条语句用于声明（不包含初始化）一个变量。它总是以关键字`var`作为开始，并后跟变量的名称和类型。未被初始化的变量的值为`nil`。我们在下一节讲变量和常量的时

候会再对变量的声明进行详细介绍。此外，变量slice4的值是完全独立的，因为其底层数组还未与其他切片值共享。

好了，如果读者认为已经真正地理解了前面所讲的关于切片值的扩展方法的内容，我们就来看一个更加复杂的用法。

我们在上一节讲切片表达式的时候说过，还可以在切片表达式中添加第三个索引——容量上界索引。如果该索引被指定，那么作为切片表达式的求值结果的那个切片值（以下简称新切片值）的容量就不再是该切片表达式的操作对象的容量与该表达式中的元素下界索引之间的差值了，而是容量上界索引与元素下界索引之间的差值。又因为它们之间存在以下关系：

$$0 \leq \text{元素下界索引} \leq \text{元素上界索引} \leq \text{容量上界索引} \leq \text{被操作对象的容量}$$

所以，指定容量上界索引的目的就是为了缩小新切片值的容量。那么，这有什么意义呢？它最重要的意义在于允许更加灵活的数据隔离策略。例如，我们有这样一个数组值：

```
var array2 [10]int = [10]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

并根据这个数组值创建了一个切片值：

```
slice5 := array2[2:6]
```

显然，变量slice5的值的底层数组即是array2的值。slice5的值的容量为8，即array2的值的长度10与切片表达式array2[2:6]中的元素下界索引2之间的差值。我们已经知道，切片值的容量就是能通过它访问和修改的底层数组元素值的最大数量。在这个例子中，我们可以直接通过在slice5的值之上应用索引表达式来访问和修改array2的值中对应索引值在[2,6)范围内的元素值。并且，在对slice5的值进行再切片（即slice5[:cap(slice5)]）之后，还可以访问和修改array2的值中对应索引值最大的那两个元素值。

如果把slice5的值作为数据载体传递给了另一个程序，是不是就意味着那个程序就可以随意地更改array2的值中的某些元素值了呢？答案当然是肯定的。这等于暴露了程序中的部分实现细节，并公开了一个可以间接修改程序内部状态的方法。在很多情况下，这并不是我们想要的。

当然，我们可以通过生成并传递一个slice5的值的副本来避免此类问题。但是，如果我们采用的是折中方案呢？如果我们就是想让另一个程序可以访问和修改array2的值中对应索引值在[2,8)范围内的元素值呢？在无法指定（在Go 1.2之前，能在切片表达式中出现的只有元素下界索引和元素上界索引）或不指定容量上界索引的情况下，这是不可能的。因为，slice5的值的容量上界索引不是由我们自己来控制的。它总是等于array2的值中最后一个元素位置的索引值再加上1，即array2的值的长度值。也就是说，array2的值中对应索引值在[2,10)范围内的元素值，总是可以被slice5的值的持有者访问和修改。

对容量上界索引的设定使得我们对切片值容量的精细控制成为了可能。例如，如果我们这样声明变量slice5：

```
slice5 := array2[2:6:8]
```

那么就可以使slice5的值的持有者只能访问和修改array2的值中对应索引值在[2,8)范围内的元素值。即使通过

```
slice5 = slice5[:cap(slice5)]
```

把slice5的窗口扩展到最大，我们也不可能通过它访问到array2的值中对应索引值大于等于8的那些元素值。此时，slice5的值的容量为6（容量上界索引与元素下界索引的差值）。对于再切片操作来说，被操作对象的容量是一个不可逾越的限制。因此，slice5的值对其底层数组（array2的值）的“访问权限”也就得到了严格的控制。另外，如果在slice5的值之上的扩展超出了它的容量，如下所示：

```
slice5 = append(slice5, []int{10, 11, 12, 13, 14, 15}...)
```

那么它原有的底层数组就会被替换。这样也就彻底切断了通过slice5访问和修改其原有底层数组中的元素值的途径。

总之，这种通过容量上界索引对切片值的容量进行设定的方式，对于精细控制切片值对其底层数组的“访问权限”来说是极其有效的。与普通的切片方式（比如array2[2:6]）相比，它提供了更好的可控性。

关于切片表达式中的这3个索引，还有一个限制：当我们在切片表达式中指定容量上界索引的时候，元素上界索引是不能够省略的。但是，在这种情况下元素下界索引却是可以省略的。例如，切片表达式

```
slice5[:3:5]
```

是合法的，而切片表达式

```
slice5[0::5]
```

则会造成一个编译错误。

现在，停顿一会儿，请读者把切片表达式以及那3个索引的用法和作用记在心里。

最后，我们来看看怎样批量复制切片值中的元素。首先，新声明并初始化两个切片值：

```
sliceA := []string{"Notepad", "UltraEdit", "Eclipse"}
sliceB := []string{"Vim", "Emacs", "LiteIDE", "IDEA"}
```

可以看到，变量sliceA的值中包含了3个元素，而变量sliceB的值中包含了4个元素。现在，使用Go语言的内建函数copy，将变量sliceB的值中的元素复制到sliceA的值中。代码如下：

```
n1 := copy(sliceA, sliceB)
```

内建函数copy的作用是把源切片值（第二个参数值）中的元素值复制到目标切片值（第一个参数值）中，并且返回被复制的元素值的数量。这个结果的类型是int的。copy函数的两个参数的元素类型必须一致，且它实际复制的元素值的数量将等于长度较短的那个切片值的长度。例如，上述示例中的语句被执行后，变量sliceA的值被修改为：

```
[]string{"Vim", "Emacs", "LiteIDE"}
```

由于sliceA的值的长度是3，所以copy函数并没有复制sliceB的值中的第四个元素。因此，前面示例中的变量n1的值为3。

注意，不像append函数那样，copy函数会改变与其第一个参数绑定的那个值。

如果我们把上面示例中的传递给copy函数的两个参数交换位置，那么改变的就将会是变量sliceB的值，改变后的值为：

```
[]string{"Notepad", "UltraEdit", "Eclipse", "IDEA"}
```

变量n1的值依然会为3。因为变量sliceA的值中只有3个元素可被复制。

Go语言的切片类型相当于其他编程语言中的动态数组类型，其扩展机制也与那些动态数组类型非常类似。在Go语言中，切片类型的应用场景非常广泛。它比数组类型更灵活、更强大，但同时也更难于理解。希望读者通过对本小节的阅读和学习能够真正地理解Go语言的切片类型。

3.2.4 字典

在Go语言中，字典类型的官方称谓是Map，它是哈希表（Hash Table）的一个实现。哈希表是一个实现了关联数组的数据结构，是计算机科学领域最有用的数据结构之一。关联数组是用于代表键值对的集合的一种抽象数据类型。在一个键集对集合中，一个键最多能够出现一次。与这个抽象数据结构相关联的操作有4个。

- 向集合中添加键值对。
- 从集合中删除键值对。
- 修改集合中已存在的键值对的值。
- 查找一个特定键所对应的值。

哈希表可以通过一个哈希函数快速地建立起键值对的内部关联，并在此基础上实现上述操作。此外，在哈希表中的键值对之间是没有顺序关系的。哈希表的实现多种多样。Go语言的字典类型的内部特性属于Go语言运行时系统的实现细节，至今未出现在Go语言的规范中。

1. 类型表示法

在Go语言中，一般称键值对为键-元素对，并把字典类型值（以下简称字典值）中的每个键值都看作与其对应的元素值的索引。不过，我们在本书中仍然使用“键值对”这个名词，因为它更加通用。键值对代表了键的值和对应的元素的值构成的结对。所以，我们也说一个键值对是由一个键值和一个元素值组合而成的。

如果一个字典类型中的键的类型为K，且元素的类型为T，那么用于表示这个字典类型的类型字面量就是：

```
map[K]T
```

可以看到，一个字典类型的键类型和元素类型都是需要在其声明中指定的。字典类型声明中的元素类型可以是任意一个有效的Go语言数据类型。但是，它的键类型不能是函数类型、字典类型或切片类型。因为键的类型必须是可比较的，也就是说，键的值必须可以作为比较操作符=和!=的操作数。如果字典类型的键类型是接口类型，那么就要求在程序运行期间，该类型的字典值中的每一个键值的动态类型都必须是可比较的，否则在进行相应操作的时候会引发运行时异常。

下面举几个例子。这些用于表示字典类型的类型字面量都是合法的：

```
map[int]string
```

```
map[string]struct { name, department string }
map[string]interface{}
```

而下面这几个类型字面量就是不合法的：

```
map[[]int]string
map[map[int]string]string
```

在3.3.3节讲可比性与有序性的时候，我们会对数据类型的可比较性作详细论述。

2. 值表示法

字典值可以由复合字面量来表示。这个复合字面量会由表示字典类型的类型字面量和被花括号“{”和“}”括起来的若干键值对组成，且在多个键值对之间使用逗号“,”分隔。键值对中的键值和元素值之间需要用冒号“:”分隔。

例如，下面表示的是一个类型为`map[string]bool`的值：

```
map[string]bool{"Vim": true, "Emacs": true, "LiteIDE": true, "Notepad": false}
```

当然，我们也可以这样表示一个不包含任何键值对的空字典值：

```
map[string]bool{}
```

3. 属性和基本操作

与指针类型和切片类型一样，字典类型是一个引用类型。与切片值相同，一个字典值总是会持有一个针对某个底层数据结构值的引用。这意味着，如果将一个字典值传递给一个会改变它的函数，那么这个改变对于函数的调用方来说也是可见的。作为对比，数组类型并不是引用类型。因此，一个数组值在作为参数被传递给某个函数并在此函数内部被改变之后，该函数的调用方并不能看到它的变化。其根本原因是，任何函数都只会拿到调用方传递给它的参数值的一个复制品。在很多编程语言中，这种传递参数值的方式常被称为“传值”。而与其对应的是以“传引用”的方式传递参数值。请记住，在Go语言中，只有“传值”而没有“传引用”。函数内部对参数值的改变是否会在该函数之外体现出来（或者说是否会反映到该参数值的源值上），只取决于这个被改变的值的类型是值类型还是引用类型。

也正因为字典类型是一个引用类型，它的零值是`nil`。一个值为`nil`的字典类型的变量类似于一个长度为0的空字典。对它进行读取操作的时候并不会引起任何错误，但是对它的写操作（添加或删除键值对）将会引发一个运行时恐慌。而一个未被初始化的字典类型的变量的值就是`nil`。

一个字典值的长度代表了它当前所包含的键值对的数量。我们可以使用内建函数`len`来获取一个字典值的长度。正如之前所说，在一个值为`nil`的字典类型的变量上应用`len`函数会得到0。

我们可以随时将一个键值对添加到一个字典值中，只要这个字典类型的值不是`nil`。这个添加键值对的操作需要用到左侧为索引表达式的赋值语句。我们在这里声明并初始化一个字典类型的变量，如下所示：

```
editorSign := map[string]bool{"LiteIDE": true, "Notepad": false}
```

变量`editorSign`是一个字典类型，它的元素类型是布尔类型，而键类型是字符串类型。现在，我们使用下面的赋值语句将一个由键值“Vim”和元素值`true`组成的键值对添加到变量`editorSign`的值中：

```
editorSign["Vim"] = true
```

这很像是通过索引值把一个元素设置到一个数组值的指定位置上。其实，我们也可以把键的值想象成字典值中的元素值的“索引值”。因为与数组值中的索引值类似，我也可以使用键值对字典值中的某个元素值进行“定位”。但不同的是，字典值中的键的类型可以是多种多样的，字典值中的键的值可以是任意的，只要它们的类型符合该字典值的类型声明即可。在上面的示例中，如果在`editorSign`的值中已存在了键为“Vim”的键值对，那么这个赋值语句的作用就相当于更新该字典值中键为“Vim”的键值对的元素值。否则，这个键值为“Vim”、元素值为`true`的键值对就会被添加到`editorSign`的值中。

我们也可以通过索引表达式在一个字典值中查找并获取与指定键值对应的那个元素值。例如：

```
sign1 := editorSign["Vim"]
```

上面的变量`sign1`的值将会是`editorSign`中与键值“Vim”对应的那个元素值。但是，当`editorSign`的值中没有键为“Vim”的键值对时，变量`sign1`将会被赋予`editorSign`的元素类型的零值，即`false`。显然，这存在歧义。我们不知道`false`真是在`editorSign`中的键为“Vim”的键值对中的那个元素值，还是意味着在`editorSign`中根本就不存在这个键值对。这种情况的解决方案我们在上一节讲表达式的时候已经说过，可以通过如下方式来消除这个歧义：

```
sign1, ok := editorSign["Vim"]
```

关于变量`sign1`的赋值依然遵循我们刚刚描述的规则，而变量`ok`将会是布尔类型的。它的值表明了`editorSign`的值中是否存在键为“Vim”的键值对。

删除字典值中的某个键值对需要用到Go语言的内建函数`delete`。我们依然以变量`editorSign`为例。如果要从`editorSign`的值中删除掉以“Vim”为键的键值对需要这样编写代码：

```
delete(editorSign, "Vim")
```

内建函数`delete`需要两个参数。第一个参数就是我们要改变的那个字典值，而第二个参数就是我们要删除的键值对中的那个键的值。`delete`函数会很“安静”。它没有结果，也不会删除并不存在的键值对的时候产生错误或者引发运行时恐慌。

最后，需要注意：字典值并不是并发安全的！Go语言官方认为，在大多数使用字典值的地方并不需要多线程场景下的安全访问控制。为了少数的并发使用场景而强制要求所有的字典值都满足互斥操作将会降低大多数程序的速度，这是得不偿失的。我认为这样确实是合情合理的。

对一个非并发安全的字典值进行不受控制的并发访问很可能会导致程序行为的错乱。不过，我们可以很容易地扩展Go语言官方的字典类型来保证并发安全性。这需要使用标准库代码包`sync`中的结构体类型`RWMutex`。从名称上我们就可以猜到这是一个读写互斥量。它常常用于多线程环境下的并发读写控制。我们会在本书第8章中详细讲解它，并且还会使用它构造出一个并发安全的字典类型。

关于Go语言的字典类型，我们就暂时介绍到这里。

3.2.5 函数和方法

在Go语言中，函数类型是一等类型。这意味着可以把函数当作一个值来传递和使用。例如，函数类型的值（以下简称为函数值）既可以作为其他函数的参数，也可以作为其他函数的结果（之一）。另外，我们还可以利用函数类型的这一特性生成闭包。总之，作为一等类型的函数类型可以使程序更加灵活和稳固。下面我们就来进行讨论。

1. 类型表示法

函数类型指代了所有可以接受若干参数并能够返回若干结果的函数。声明一个函数类型总会以关键字`func`作为开始。紧跟在关键字`func`之后的应该是这个函数的签名，包括了参数声明列表和结果声明列表。参数声明列表在左，结果声明列表在右，中间由空格“ ”分隔。参数声明列表必须由圆括号括起来，多个参数声明之间需用逗号“,”来分隔。

参数声明的一般写法是参数名称在前，参数类型在后，中间以空格“ ”分隔。例如，我们这样声明一个名称为`name`、类型为`string`的参数：

```
name string
```

如果有一个参数列表，除了上述的名称为`name`的参数之外，还包括一个名称为`age`、类型为`int`的参数。那么，这个参数列表应该这样编写：

```
(name string, age int)
```

注意，在同一个参数声明列表中的所有参数名称都必须是唯一的。

如果相邻两个参数属于同一数据类型，那么我们只需要写一次参数类型。例如，我们向上面的参数声明列表中添加一个名称为`seniority`、类型为`int`的参数：

```
(name string, age, seniority int)
```

这形同于：

```
(name string, age int, seniority int)
```

另外，我们也可以在函数类型声明的参数声明列表中略去所有参数的名称：

```
(string, int, int)
```

当然我们不推荐这种做法，因为它的可读性很差。我们应该尽量让阅读它的人轻易猜出其含义。

还记得可变长参数吗？我们可以再向这个参数声明列表中追加一个名称为`informations`、类型为`...string`的可变长参数：

```
(name string, age int, seniority int, informations ...string)
```

注意，可变长参数必须是参数列表中的最后一个。所以，可变长参数也常常被称为“最后的参数”。

另一方面，函数类型声明的结果声明列表中一般包含若干个结果声明。结果声明列表的编写规则与参数声明基本一致。不过，它们之间存在两点区别。第一，只存在可变长参数的声明而不存在可变长结果的声明；第二，如果结果声明列表中只有一个结果声明且这个结果声明中并不包含结果的名称，那么就可以忽略掉它的圆括号，如下所示：

```
func (name string, age int, seniority int, informations ...string) bool
```

其中bool就是这个函数类型的唯一结果的类型声明。该结果声明独自组成了该函数类型的结果声明列表。

如果我们需要命名这个结果，就应该这样编写：

```
func (name string, age int, seniority int, informations ...string) (done bool)
```

我们将这个函数类型的唯一结果命名为了done。注意，这时的结果声明列表就必须被圆括号括起来了。命名的结果是很有用的。其名称可以作为附属于该函数类型声明的文档的一部分。阅读代码的人可以根据结果的名称大概猜出该结果的含义。这样，我们在编写这个函数类型的实现的时候，就会更加明确地知道需要返回怎样的结果了。

我们之所以说一个函数类型可以有一个结果声明的列表，是因为Go语言的函数类型可以有多个结果。这是Go语言的先进特性之一。不知道大家在用其他编程语言编写程序的时候是否遇到过这种情况。你需要使用整数来表示函数体内操作的结果。例如，使用-1来表示操作失败、使用0来表示操作成功，再使用大于0的某个整数来表示受影响的数据的数量，也许还会使用小于-1的某个整数表示操作失败的原因。比如：

```
func (name string, age int, seniority int) (result int)
```

可能你已经习惯这种“多合一”的表述方式了。但是现在让我来告诉你在Go语言程序中可以怎样做，请看下面这个函数类型声明：

```
func (name string, age int, seniority int) (effected uint, err error)
```

为函数声明多个结果可以让每个结果的职责更加单一。这既易于理解又方便使用。更值得称赞的是，我们可以利用这一特性将错误值作为结果（之一）返回给调用它的代码，而不是把错误抛（throw）出来，然后再不得不在调用它的地方编写若干代码来抓（catch）住这个错误。在上面这个函数类型声明中，第二个结果声明就体现了这样的错误值传递方式。

这样一来，我们既可以非常清晰地可能出现的错误值提供一个单独的传递渠道，又可以用一种非常安静的方式来传递它。单独的传递渠道可以使函数非常清晰地表达出错误发生的可能性，但是又不会像throw-catch模式那样迫使外层代码掺杂一些有时并不必要的错误处理代码。这很合理，不是吗？我们会在下一章详细讨论Go语言的错误处理机制。

函数类型的多个结果声明的另一个好处是，可以利用它从不同的角度来体现函数的内部操作的结果。例如：

```
func (name string, age int, seniority int) (done bool, id uint, synchronized bool)
```

假设上面声明的函数类型专用于保存某项数据，它的3个结果的作用如下。

- ❑ done：用于表示数据是否被成功保存。
- ❑ id：数据被保存后的ID。此ID可以被用来检索数据。
- ❑ synchronized：用于表示此数据是否已被同步到相关系统中。

这样，该函数的调用方会更加清晰明了地获知具体的操作结果。同时，处理这些操作结果的代码也会更加简单和扁平化。

我们从Go语言的函数类型的声明方式上就可以看出，Go语言的函数是非常灵活多样的。再加之函数类型是Go语言中的一等类型，所以函数在Go语言程序中的用途相当广泛。下面，我们来看看怎么编写函数类型的实现。

2. 值表示法

函数类型的零值是nil。因此，未被初始化的函数类型的变量的值就是nil。我们在一个未被初始化的函数类型的变量上应用调用表达式会引发一个运行时恐慌。

函数类型的值被分为两类：命名函数值和匿名函数值。在很多时候，我们称命名函数值为命名函数，称匿名函数值为匿名函数。虽然可以这样称呼，但是我们应该牢记它们都是值的一种。

我们先来讨论命名函数。命名函数的声明一般由关键字func、函数名称、函数的签名（由参数声明列表和结果声明列表）和函数体组成。其中，函数体就是由花括号“{”和“}”括起来的若干条Go语言语句的合称。如果在函数的签名中包含了结果声明列表，那么在该函数的函数体中的任何可到达的流程分支的最后一条语句都必须是终止语句。终止语句有很多种，比如以关键字return或goto开始的语句，又或者仅包含针对内建函数panic的调用表达式的语句。其中的内建函数panic用于产生一个运行时恐慌。关于终止语句的详细说明参见下一章。在此，我们仅以关键字return开始的终止语句为例。假设有这样一个用于取模运算的Module函数：

```
func Module(x, y int) int {  
    return x % y  
}
```

该函数的参数声明列表包含了两个参数：x和y。它们都是int类型的。同时，Module函数还有一个未命名的结果声明，也是int类型的。正因为存在这个结果声明，所以该函数体内的最后一条语句必须是终止语句。函数Module在它的函数体内仅包含了一条语句。这条语句由关键字return和一个由求余操作符和两个操作数组成的表达式。这两个操作数正是Module函数的两个参数。Module函数将它的两个参数作为操作数进行求余（也就是取模）操作，并将结果返回给调用方。注意，在关键字return之后（右边）的结果必须在数量上与该函数的结果声明列表中的内容完全一致，且在对应位置的结果的类型上存在可赋予的关系，否则将不能通过编译。顺便提一句，以关键字return开始的语句称为return语句，跟在return之后的内容称为return的参数。

我们在前面说过，在声明一个函数类型的时候可以给它的结果命名。同样地，我们在编写函数的时候也可以给它的结果命名。我们可以给上述的Module函数的结果命名，如下所示：

```
func Module(x, y int) (result int) {  
    return x % y  
}
```

注意，在为这个唯一的结果命名之后就必须用圆括号将它括起来了。

实际上，为函数的结果命名会使它们能够以常规变量的形式存在，就像函数的参数那样。当结果被命名，它们在函数被调用时就会被初始化为对应的数据类型的零值。如果这样的函数的函数体中有一条不带任何参数的return语句，那么在执行到这条return语句的时候，作为结果的变量的当前值就会被返回给函数调用方。因此，我们可以稍微改造一下Module函数的函数体中的语句，使它与命名结果的风格相适应：

```
func Module(x, y int) (result int) {
    result = x % y
    return
}
```

在Module函数被调用时，变量result被初始化为int类型的零值0。当该函数的函数体中的第一条语句被执行时，变量result被赋予了表达式x%y的结果值。当该函数体中的无参数的return语句被执行时，result的当前值就会作为结果被返回给函数调用方。

对函数结果的命名可以使函数体内的代码更加简单和清晰，其中的哪一条语句赋操作了哪一个函数结果变得一目了然。我们也可以非常方便地使用编辑器的代码高亮功能找到这些语句。在函数体包含很多语句的时候，这种惯用法所体现出的便捷性会更为突出。

顺便提一下，命名函数的声明还可以省略掉函数体。这意味着，该函数会由外部程序（如汇编语言程序）实现，而不会由Go语言程序实现。

再来说匿名函数。匿名函数由函数字面量表示。函数字面量也是表达式的一种。顾名思义，匿名函数没有名字。在声明的内容上，匿名函数与命名函数的区别也只是少了一个函数名称。也就是说，函数字面量仅由关键字func、函数的签名和函数体组成。我们稍加改动就可以把前面声明的Module函数改写成匿名函数：

```
func (x, y int) (result int) {
    result = x % y
    return
}
```

函数字面量和函数类型声明很像，它比函数类型声明多了一个函数体。因此，函数字面量也可以看作是对某个函数类型的即时实现。

一个函数字面量可以被赋给一个变量，也可以被直接调用。这也充分体现了Go语言把函数作为值的这一特性。下面我们就来看看函数都有哪些属性以及怎样操作它。

3. 属性和基本操作

函数类型也是Go语言的数据类型之一。因此，我们可以把函数类型作为一个变量的类型。例如，我们可以这样声明一个变量：

```
var recorder func (name string, age int, seniority int) (done bool)
```

之后，所有符合这个函数类型的实现都可以被赋给变量recorder，如下所示：

```
recorder = func(name string, age int, seniority int) (done bool) {
    // 省略若干条语句
    return
}
```

注意，被赋给变量recorder的函数字面量必须与recorder的类型拥有相同的函数签名。

熟悉面向对象编程的读者可能会意识到，这很像“面向接口编程”原则的一种实现方式。对设计模式有所了解的读者或许也可以从这一小段代码上联想到策略模式。正因为Go语言中函数类型是一等类型，我们才能使用它实现程序的更细粒度的灵活性，而不用像Java语言那样，必须先要创建一个类（Class）再考虑实现某个接口的问题。对于那些为了一定的灵活性而不得不编

写各种样板代码的语言来说，Go语言会让我们感到非常地得心应手。

在上面的示例中，我们将一个函数字面量赋给了变量`recorder`。由于我们可以在一个函数类型的变量上直接应用调用表达式来调用它，所以下面这段代码是合法的：

```
done := recorder("Harry", 32, 10)
```

我们把调用变量`recorder`（实为对它代表的那个函数的调用）后得到的结果值又赋给了新的变量。需要注意的是，被赋值的变量在数量上必须与函数的结果声明列表中的内容完全一致，且在对应位置的变量和结果的类型上存在可赋予的关系。这条规则同样适用于对命名函数进行调用并赋值的情况。

我们可以把函数类型的变量的值看作是一个函数值。所有的函数值都可以被调用，函数字面量也不例外。我们可以在函数字面量被编写出来的时候直接调用它，例如：

```
func(name string, age int, seniority int) (done bool) {
    // 省略若干条语句
    return
}("Harry", 32, 10)
```

函数既然可以作为变量的值，那么也就可以像其他值那样在函数之间传递。换句话说，一个函数既可以作为其他函数的参数，也可以作为其他函数的结果。

我们来举一个例子。现在要声明一个可以对一段文本进行加密的函数，同时，要求可以根据不同的应用场景实时地、频繁地对加密算法进行变更。根据上述需求，我们就不应该只声明一个加密函数，而应该声明一个能够生成加密函数的函数，然后在程序运行期间，根据不同的要求使用这个函数来生成需要的加密函数。

首先，我们应该确定可以向这个生成函数的函数提供加密算法的方式。最简单也是最直观的方式就是把加密算法封装成一个函数并作为参数传递进去。因为，在Go语言中，函数是封装一段代码的最小单元。此外，所有用于封装加密算法的函数都应该是同一个函数类型的，这有利于加密算法的无缝替换。因此，我们应该首先声明这样一个函数类型：

```
type Encipher func(plaintext string) []byte
```

在上一节我们已经使用过关键字`type`，它专门用于声明自定义数据类型。这里声明的`Encipher`类型实际上就是函数类型`func(plaintext string) []byte`的一个别名类型。

这个函数接受一个`string`类型的参数，并且返回一个元素类型为`byte`的切片类型的结果。其实这代表了一类比较通用的加密算法的输入数据和输出数据。

在有了这个用于封装加密算法的函数类型之后，我们就可以声明那个可以生成加密函数的函数了。其声明如下：

```
func GenEncryptionFunc(encrypt Encipher) func(string) (ciphertext string) {
    return func(plaintext string) string {
        return fmt.Sprintf("%x", encrypt(plaintext))
    }
}
```

可以看到，函数`GenEncryptionFunc`的签名中包括了一个参数声明和一个结果声明。其中，参

数声明中的参数类型就是我们刚刚定义的那个用于封装加密算法的函数类型。它后面的结果声明同样表示了一个函数类型的结果。这个函数类型正是GenEncryptionFunc函数所生成的加密函数的类型。它接收一个string类型的明文作为参数，并返回一个string类型的密文作为结果。之所以密文是string类型的，是因为要考虑到能够把密文作为文本保存的需求。

在GenEncryptionFunc函数的函数体内直接返回了符合加密函数类型的匿名函数。这个匿名函数的函数体内也只包含了一条语句，这条语句做了两件事。首先，它调用名称为encrypt的函数，把作为该匿名函数的参数的明文加密。然后，它使用标准库代码包fmt中的Sprintf函数，把encrypt函数的调用结果转换成了字符串。这个字符串的内容实际上是用十六进制数表示的加密结果。而这个加密结果实际上是[]byte类型的。

对于这个被GenEncryptionFunc函数返回的匿名函数来讲，其中的标识符encrypt并不是在它的函数体内定义的。它是一个外来的标识符，是GenEncryptionFunc函数中参数的名称。而这个参数代表了待定的加密算法函数。只有当我们调用GenEncryptionFunc函数的时候，这个匿名函数中的标识符encrypt才能够具有特定的意义——代表了某个加密算法函数。在这之后，对GenEncryptionFunc函数的调用结果恰恰就是基于传递给它的那个加密算法函数（由参数encrypt代表）生成的加密函数。

每一次调用GenEncryptionFunc函数时，传递给它的那个加密算法函数都会一直被对应的加密函数引用着。只要生成的加密函数还可以被访问，其中的加密算法函数就会一直存在，而不会被Go语言的垃圾回收器回收。

熟悉函数式编程范式的读者可能会发现，这里恰恰实现了闭包。闭包这个词源自于通过“捕获”自由变量的绑定对函数文本执行的“闭合”动作。在上面的示例中，加密函数中的标识符encrypt代表的就是自由变量，它在调用它的加密函数被生成的时候，与GenEncryptionFunc函数的参数encrypt的值进行了绑定，从而使得这个加密函数变得完整。我们也可以说，通过“捕获”自由变量encrypt的绑定使GenEncryptionFunc函数返回的加密函数“闭合”了。这就是闭包的典型应用。

为了使读者更加宏观和清晰地理解GenEncryptionFunc函数所涉及的一些概念，我制作了一幅图，见图3-8。

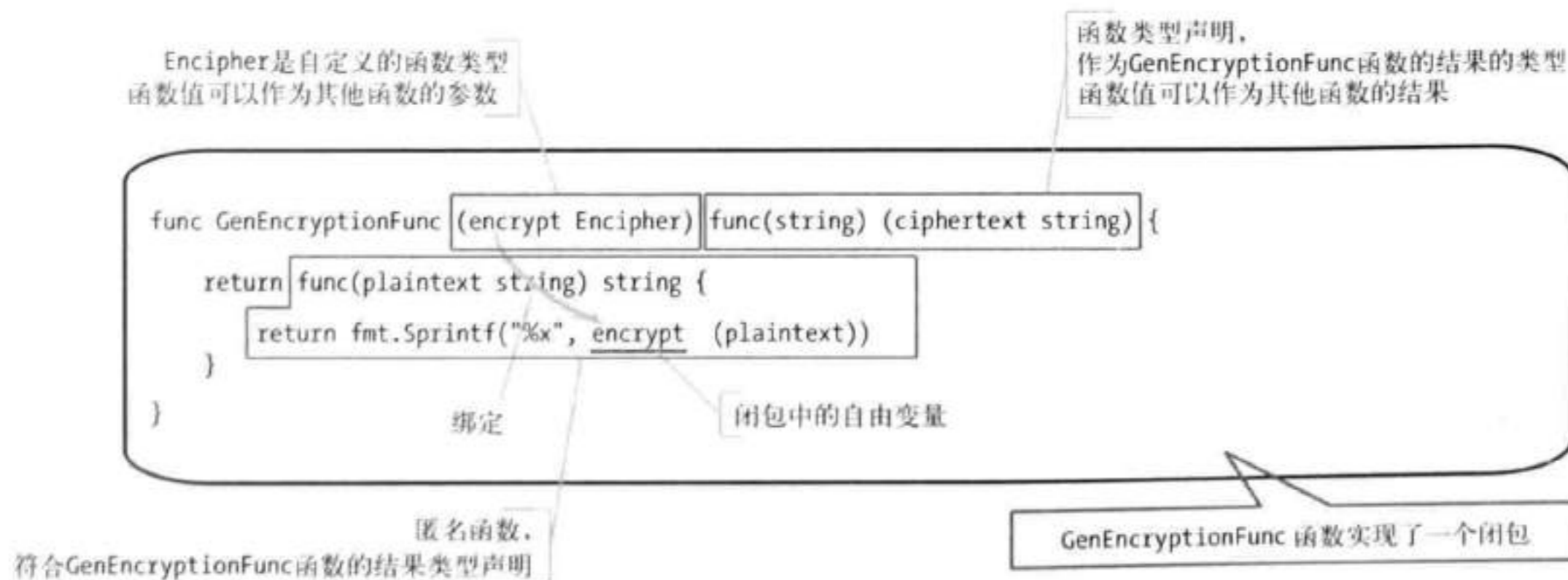


图3-8 GenEncryptionFunc函数

实际上，只有当函数类型是一等类型并且其值可以作为其他函数的参数或结果的时候，我们才能够编写出实现闭包的代码。为什么我们可以在Go程序和Python程序中轻松地实现闭包，而在使用Java语言（1.8版本之前）编写的程序中却不能这样做？这就是根本原因。

函数类型是Go语言中非常重要的一个数据类型。它是Go语言支持函数式编程范式的重要体现，也是我们编写函数式风格代码的主要手段。此外，函数还可以附属于任何自定义的数据类型，或者与接口类型和结构体类型相结合作为针对某个或某些数据类型的操作方法。下面我们就对这个函数类型的重要演进形式——方法进行专门的论述。

4. 方法

方法就是附属于某个自定义的数据类型的函数。具体地说，一个方法就是一个与某个接收者关联的函数。因此，在方法的签名中不但包含了函数签名，还包含了一个与接收者有关的声明。也就是说，方法的声明包含了关键字func、接收者声明、方法名称、参数声明列表、结果声明列表和方法体。其中的接收者声明、参数声明列表和结果声明列表被统称为方法签名，而方法体可以在某些情况下被忽略。一般情况下，一个接收者声明由被圆括号括起来的两个标识符组成。这两个标识符之间被空格“ ”分隔。左边的标识符代表了接收者的值在当前方法中的名称，而右边的标识符则代表了接收者的类型。前者又称为接收者标识符。下面我们来看一个例子：

```
type MyIntSlice []int

func (self MyIntSlice) Max() (result int) {
    // 省略若干条语句
    return
}
```

在这个示例中，我们首先自定义了一个数据类型MyIntSlice。我们可以把这个自定义类型看作[]int类型的一个别名类型。在这之后，我们还声明了一个方法。在这个名称为Max的方法中，接收者声明为(self MyIntSlice)。其中，右边的标识符明确地表示了该方法所属的数据类型，即MyIntSlice。而左边的接收者标识符则代表了MyIntSlice类型的值在方法Max中的名称，这为我们在该方法中使用这个值提供了前提条件。

下面，我们来看一看与方法声明中的接收者声明有关的几条编写规则。

- 接收者声明中的类型必须是某个自定义的数据类型，或者是一个与某个自定义数据类型对应的指针类型。但不论接收者的类型是哪一种，接收者的基本类型都会是那个自定义数据类型。例如，方法声明

```
func (self *MyIntSlice) Min() (result int)
```

中的接收者的类型是*MyIntSlice，而其基本类型是MyIntSlice。接收者的基本类型既不能是一个指针类型，也不能是一个接口类型。

- 接收者声明中的类型必须由非限定标识符代表。也就是说，方法所属的数据类型的声明必须与该方法声明处在同一个代码包内。
- 接收者标识符不能是空标识符“_”，并且必须在其所在的方法签名中是唯一的。
- 如果接收者的值（由接收者标识符代表）未在当前方法的方法体内被引用，那么我们就

可以将这个接收者标识符从当前方法的接收者声明中删除掉。注意，虽然这条规则同样适用于方法声明和函数声明中的参数声明，但是并不推荐这么做，原因已经在前面说明。

我们常常把接收者类型是某个自定义数据类型的方法叫作该数据类型的值方法，而把接收者类型是与某个自定义数据类型对应的指针类型的方法叫作该数据类型的指针方法。可见，一个方法总是与其接收者的基本类型相关联的。

还要注意，对于一个接收者的基本类型来说，它所包含的方法的名称之间不能有重复。如果这个接收者的基本类型是一个结构体类型，那么还需要保证它包含的字段和方法的名称之间不能出现重复。

一个方法的类型与从其声明中去掉接收者声明之后的函数的类型相似。例如，方法

```
func (self *MyIntSlice) Min() (result int)
```

的类型是

```
func Min() (self *MyIntSlice, result int)
```

也就是说，我们把接收者声明中的两个标识符原样搬到参数声明列表的首位，就可以得到该方法的类型了。

但要注意：形如上述方法的类型表示的函数的值只能算是一个函数，而不能叫作方法。也就是说，这样的函数并没有与任何自定义数据类型相关联。

我们之前多次提到，接收者的类型可以是一个自定义的数据类型，也可以是一个与某个自定义数据类型对应的指针类型。那么在接收者的基本类型确定的情况下，我们应该怎样选择接收者的类型呢？实际上，这也是一个在值方法和指针方法之间做选择的问题。这里有两条很重要的规则。

- ❑ 在某个自定义数据类型的值上，只能够调用与这个数据类型相关联的值方法，而在指向这个值的指针值上，却能够调用与其数据类型关联的值方法和指针方法。从另一个角度讲，自定义数据类型的方法集合中仅包含了与它关联的所有值方法，而与它相对应的指针类型的方法集合中却包含了与它关联的所有值方法和所有指针方法。
- ❑ 在指针方法中一定能够改变接收者的值，而在值方法中，对接收者的值的改变对于该方法之外一般是无效的。这是因为，以接收者标识符代表的接收者的值实际上也是当前方法所属的数据类型的当前值的一个复制品。对于值方法来说，由于这个接收者的值就是一个当前值的复制品，所以对它的改变并不会影响到当前值。而对于指针方法来说，这个接收者的值则是一个当前值的指针的复制品。因此，依据这个指针来对当前值做变更，就等于直接对该值进行了改变。

对于上面的第一条规则，我们在实际编程过程中可能会遇到这种情况：虽然自定义数据类型的方法集合中不包含与它关联的指针方法，但是我们仍然能够通过这个类型的值调用到它的指针方法。其实，我们在3.1.6节讲调用表达式的时候已经说明原因。还记得由 `(&s).m()` 表示的速记法吗？还不清楚的读者可以翻回到前面温习一下。

另外，上面的第二条规则有个例外：接收者的类型如果是引用类型的别名类型，那么在该类

型值的值方法中对该值的改变也是对外有效的。在前面讲字典类型的时候我们说过，切片类型和字典类型都属于引用类型。除此之外，通道类型也属于引用类型，这在第7章会讲到。

请读者牢记上面的规则，我们在自己编写自定义数据类型及其方法或者对其他自定义数据类型的方法进行调用的时候都需要依照它们。

至此，我们几乎介绍了与Go语言的函数和方法相关的全部知识。我们在后面讲接口类型和结构体类型的时候还会再涉及它们。

3.2.6 接口

一个Go语言的接口由一个方法的集合代表。只要一个数据类型（或与其对应的指针类型）附带的方法集合是某一个接口的方法集合的超集，那么就可以判定该类型实现了这个接口。这意味着Go语言对接口的实现是非侵入式的。换句话说，要想实现一个接口，只需要实现其中的所有方法声明即可，而不需要在数据类型上添加任何特殊的标记。此外，一个接口类型的变量，可以与任何实现了这个接口类型的数据类型的值绑定。

一个数据类型可以拥有一个与它关联的方法集合。一个接口的方法集合，就是我们所说的通常意义上的接口——一组操作数据的方式。一个非接口类型的数据类型的方法集合决定了它是否实现了某个或某些接口。在本小节，我们会对Go语言的接口进行说明，并且介绍在一个非接口类型的数据类型上附加方法和实现接口的基本知识。

1. 类型表示法

接口由方法集合代表。相应地，接口类型的声明由若干个方法的声明组成。方法的声明由方法名称和方法签名构成。我们之前说过，方法是函数的一种。因此，这里的方法签名的编写规则和约束与函数签名完全一致。另外，在一个接口类型的声明中不允许出现重复的方法名称。

接口类型是一个很宽泛的概念，它是所有自定义的接口类型的统称。与其他自定义数据类型一样，我们在声明一个自定义的接口类型的时候要以关键字`type`作为开始。除此之外，接口类型声明还包含了接口类型的名称、关键字`interface`和由花括号“{”和“}”括起来的方法声明的集合。我们以标准库代码包`sort`中的接口类型`Interface`为例，其声明如下：

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

为了突出重点，我们去掉了该声明中的注释行。可以看到，这个接口类型的声明中包含了3个方法声明。每个方法声明都独占一行。

顺便提一句，只要一个数据类型实现了代码包`sort`中的接口类型`Interface`，就能够使用这个代码包中的函数对该数据类型的值进行排序。我们在下一章的实战演练环节中会用到它们。

至此，我们已经解释了编写一个接口类型声明所需的一切。如果读者已经理解了我们之前所讲的关于类型和函数的知识的话，那么一定会感觉接口类型的声明编写起来非常简单。

除此之外,我们还可以将一个接口类型嵌入到另一个接口类型中。请看下面的接口类型声明:

```
type Sortable interface {
    sort.Interface
    Sort()
}
```

在接口类型Sortable中,我们嵌入了sort中的接口类型Interface。这是通过在前者的声明中加入代表后者的限定标识符来实现的。这个限定标识符与方法声明一样,需要独占一行。

接口类型Sortable实际上包含了4个方法声明,它们的名称分别为Len、Less、Swap和Sort。其中前3个方法声明其实都是被包含在接口类型sort.Interface中的。这就是在一个接口类型声明中嵌入另一个接口类型声明的作用——将一个接口的方法集合中的方法批量地添加到另一个接口的方法集合中。实际上,在一些编程语言中,这种嵌入常被叫作接口间的继承。所以,也可以说接口类型Sortable继承了接口类型sort.Interface。

Go语言并不提供典型的类型驱动的子类化方法,但是却靠这种嵌入的方式实现了同样的效果。类型嵌入同样体现了非侵入式的风格。它同样适用于结构体类型。请记住,一个接口类型只接受其他接口类型的嵌入。

关于接口类型的嵌入,有一个约束,那就是不能嵌入自身。这包括直接的嵌入和间接的嵌入。直接的嵌入如下:

```
type Interface1 interface {
    Interface1
}
```

而间接的嵌入则像这样:

```
type Interface2 interface {
    Interface3
}

type Interface3 interface {
    Interface2
}
```

错误的接口类型嵌入会造成编译错误。另外,当前接口类型中声明的方法也不能与任何被嵌入其中的接口类型的方法重名,否则也会造成编译错误。

最后值得一提的是interface{}。它是Go语言自身定义的一个特殊的接口类型——空接口。空接口不包含任何方法声明的接口。也正因为如此,Go语言中所有数据类型都是它的实现。

2. 值表示法

严格来说,Go语言的接口类型没有相应的值表示法,因为接口是规范而不是实现。但是,我们在本小节开头说过,一个接口类型的变量可以被赋予任何实现了这个接口类型的数据类型的值。从这个角度讲,接口类型的值可以由任何其他数据类型的值来表示。

3. 属性和基本操作

接口的最基本属性就是它们的方法集合。关于方法集合在接口中起到的作用,我们在前面的内容中已经有所说明。因此,在这里我们重点介绍怎样编写接口类型的实现。

实现一个接口类型的可以是任何自定义的数据类型，只要这个数据类型附带的方法集合是该接口类型的方法集合的超集。

我们现在编写一个自定义的数据类型`SortableStrings`：

```
type SortableStrings [3]string
```

这个自定义数据类型相当于`[3]string`类型的一个别名类型。我们想让这个自定义数据类型实现`sort.Interface`接口类型。这需要我们实现`sort.Interface`中声明的全部方法。这些方法的实现都需要以类型`SortableStrings`为接收者的类型。这些方法的声明如下：

```
func (self SortableStrings) Len() int {
    return len(self)
}

func (self SortableStrings) Less(i, j int) bool {
    return self[i] < self[j]
}

func (self SortableStrings) Swap(i, j int) {
    self[i], self[j] = self[j], self[i]
}
```

有了上面这3个方法声明，`SortableStrings`类型就已经是一个`sort.Interface`接口类型的实现了。下面我们来验证一下。还记得我们在上一节中介绍过的类型断言表达式吗？我们就用它来进行这项验证：

```
_, ok := interface{}(SortableStrings{}).(sort.Interface)
```

注意，要想让这条语句编译通过，首先需要导入代码包`sort`。我们可以看到，赋值语句的右边就是一个类型断言表达式，左边的两个标识符代表了这个表达式的求值结果。但是，我们在这里不关心类型转换后的结果，而只关注类型转换成功与否，所以第一个标识符为空标识符“_”。标识符`ok`代表了一个布尔类型的变量。在这里，这个变量的值一定是`true`，因为`SortableStrings`类型确实实现了接口类型`sort.Interface`中声明的所有方法。

上述方式是验证一个数据类型是否是某个接口类型的实现的最简单也是最直接的方法。

一个接口类型可以被任意数量的数据类型实现。反过来讲，一个数据类型也可以同时实现多个接口类型。上面的自定义数据类型`SortableStrings`也可以实现接口类型`Sortable`，只要我们再编写一个这样的方法声明就可以了：

```
func (self SortableStrings) Sort() {
    sort.Sort(self)
}
```

现在，`SortableStrings`类型在实现了接口类型`sort.Interface`的同时也实现了接口类型`Sortable`。这意味着下面这条语句中的变量`ok2`也会被赋予布尔值`true`：

```
_, ok2 := interface{}(SortableStrings{}).(Sortable)
```

现在，我们把`SortableStrings`类型包含的`Sort`方法中的接收者类型由`SortableStrings`改为`*SortableStrings`。也就是说，我们把这个函数的接收者类型改为了与`SortableStrings`类型对应

的指针类型。这种情况下, `SortableStrings` 类型就不再是接口类型 `Sortable` 的实现了。也就是说, 变量 `ok2` 的值现在应该是 `false`。因为, 方法 `Sort` 不再是一个值方法了, 我们把它变成了一个指针方法。这个时候, 只有与 `SortableStrings` 类型的值对应的指针值才能够通过上面的类型断言, 如下所示:

```
_, ok3 := interface{}(&SortableStrings{}).(Sortable)
```

这条语句执行后, 变量 `ok3` 的值将会是 `true`。这也印证了我们之前讲到过的与值方法和指针方法有关的规则。

不过, `SortableStrings` 类型还有存在一个很大的问题。请看下面的测试代码:

```
ss := SortableStrings{"2", "3", "1"}
ss.Sort()
fmt.Printf("Sortable strings: %v\n", ss)
```

在上面的测试代码中, 我们用到了代码包 `fmt`。这是一个Go语言标准库中的代码包, 其中包含了很多用于将目标字符串打印到各种输出上的函数。因此, 要想让上面的几条语句通过编译, 我们需要先导入代码包 `fmt`。代码包 `fmt` 中的 `Printf` 函数也用于打印字符串, 不过它比 `Print` 函数和 `Println` 函数更加灵活。关于 `fmt` 包及其中函数的更多信息, 请读者查阅Go语言官方文档网站 (<http://godoc.org>) 上的相关信息。

在我们执行上面的测试代码后, 计算机屏幕上会显示这样一行信息:

```
Sortable strings: [2 3 1]
```

其中 `[2 3 1]` 是 `SortableStrings` 类型值的字符串表示。`SortableStrings` 类型是 `[3]string` 类型的别名类型, 因此它沿用了 `[3]string` 类型值的字符串表示方式。从上面的字符串表示来看, 变量 `ss` 的值并没有被排序, 但是我们在打印它之前已经调用过 `Sort` 方法了, 这是怎么回事呢?

我们在上一小节说过, 在值方法中, 对接收者的值的改变在该方法之外是不可见的。在上面的示例中, `SortableStrings` 类型的 `Sort` 方法实际上是通过函数 `sort.Sort` 来对接收者的值进行排序的。`sort.Sort` 函数接受一个类型为 `sort.Interface` 的参数值, 并利用这个值的方法 `Len`、`Less` 和 `Swap` 来修改其参数中的各个元素的位置以完成排序工作。再来看 `SortableStrings` 类型, 虽然它实现了接口类型 `sort.Interface` 中声明的全部方法, 但是这些方法都是值方法。这使得在这些方法中对接收者值的改变并不会影响到它的源值。因为, 它们只是改变了源值的某个复制品。这就是 `Sort` 方法失效的真正原因。这个问题在我们不了解 `sort.Sort` 函数的内部运作机制时是不容易定位的。当我们把 `SortableStrings` 类型的方法 `Len`、`Less` 和 `Swap` 的接收者类型都改为 `*SortableStrings` 之后, 这个问题就会得到解决。但是, 这时的 `SortableStrings` 类型就已经不再是接口类型 `sort.Interface` 的实现了。因此, 前面示例代码中变量 `ok` 的值会变为 `false`。

至此, 上述示例代码已经能够完全体现出接口的实现方式以及值方法与指针方法之间的区别了。这里给读者留下一个小题目。

把本小节中的所有示例代码都放到一个命令源码文件中, 并使用 `go run` 命令运行该文件中的代码。然后, 根据上面的描述改动这些代码, 并利用代码包 `fmt` 中的函数打印出变量 `ok`、`ok2` 和 `ok3` 以及在调用表达式 `ss.Sort()` 被求值前后的 `ss` 的值。

希望读者能够通过体会这些变量的值的变化,更深刻地理解与接口实现、值方法和指针方法有关的概念、规则和实际意义。

现在,我们再对`SortableStrings`的类型声明稍作改动:

```
type SortableStrings []string // 去掉了方括号中的3。
```

这会产生哪些变化呢?请读者先自己试验一下,修改`SortableStrings`的类型声明并再次编译或运行这些代码。

这实际上是将`SortableStrings`由数组类型的别名类型改为了切片类型的别名类型。这使得与之关联的方法无法通过编译。

其中与索引表达式有关的错误,是由于索引表达式不能被应用在指向切片值的指针类型值上。这方面的说明我们在上一节讲索引表达式的时候已经提到。又由于内建函数`len`的参数也不能是指向切片值的指针类型值,所以与`SortableStrings`类型关联的`Len`方法中的代码也会造成一个编译错误。

上面这两个问题的解决方法非常简单,即将方法`Len`、`Less`、`Swap`和`Sort`的接收者类型都由`*SortableStrings`改回`SortableStrings`。不用担心,我们在上一小节讲值方法和指针方法的选择规则的时候说过,对于引用类型的别名类型来说,值方法对接收者值的改变也会反映在其源值上。因此,经过上面的变更之后,`SortableStrings`类型的这些方法的功能并不会受到任何影响。在进行了这些修改之后,读者应该再次运行包含了前面所有示例代码的那个命令源码文件,然后看看标准输出上出现的那些内容。

我们会在下一小节介绍结构体类型,它是Go语言中最灵活的一种数据类型。并且,与某个数据类型的别名类型相比,使用结构体类型来实现接口类型是更常用的一种做法。

3.2.7 结构体

结构体类型既可以包含若干个命名元素(又称为字段),又可以与若干个方法相关联。从面向对象编程的角度看,结构体类型中的字段代表了该类型的属性,而与它关联的方法则可以看作是针对于这些属性的操作。

1. 类型表示法

结构体类型的声明可以包含若干个字段的声明。字段的声明由两个标识符组成,左边的标识符表示了该字段的名称,右边的标识符则代表了该字段的类型,两个标识符之间需用空格“ ”分隔。通常情况下,结构体类型声明中的每个字段声明都独占一行。并且,同一个结构体类型声明中的字段之间不能出现重名的情况。

与函数类型一样,结构体类型也分为命名结构体类型和匿名结构体类型。我们先来讨论命名结构体类型。

命名结构体类型的声明总是以关键字`type`开始,并依次包含结构体类型的名称、关键字`struct`和由花括号“{”和“}”括起来的字段声明列表。请看下面的示例:

```
type Sequence struct {
    len int
    cap int
    Sortable
    sortableArray sort.Interface
}
```

任何数据类型都可以成为结构体类型的字段的类型。当字段名称的首字母是大写字母时，我们就可以在任何位置（包括其他代码包）上通过其所属的结构体类型的值（下简称结构体值）和选择表达式访问到它们。否则，这些字段就是包级私有的。我们只有在该结构体声明所属的代码包中才能够对它们进行访问或者给它们赋值。另外，虽然我们可以把两个类型相同的字段写到同一行中：

```
len, cap int
```

但是为了清晰起见，我并不建议这样做。

如果一个字段声明中只有类型而没有指定名称的话，这个字段就叫作匿名字段。结构体类型Sequence中的Sortable就是一个匿名字段。匿名字段有时也被称为嵌入式的字段或结构体类型的嵌入类型。在形式上，这种嵌入的方式与接口类型间的嵌入很类似。但是，在意义上，这两种嵌入大不相同。

匿名字段的类型必须由一个数据类型的名称或者一个与非接口类型对应的指针类型的名称代表。更重要的是，代表匿名字段类型的非限定名称将被隐含地作为该字段的名称。如果匿名字段类型是一个指针类型的话，那么这个指针类型所指的数据类型的非限定名称就会被作为该字段的名称。所谓非限定名称就是由非限定标识符代表的名称。非限定标识符与我们在上一节讲过的限定标识符的含义相对立，指的是不包含代码包名称和点“.”的标识符。为了更加清晰地说明匿名类型的隐含名称，我们来看这样一个示例：

```
type Anonymities struct {
    T1
    *T2
    P.T3
    *P.T4
}
```

这个名为Anonymities的结构体类型中包含了4个匿名字段。其中，T1和P.T3为非指针的数据类型，它们隐含的名称分别为T1和T3。*T2和*P.T4为指针类型，它们隐含的名称分别是T2和T4。请注意，匿名字段的隐含名称也不能与它所属的结构体类型中的其他字段名称重复。

结构体类型中的嵌入字段比接口类型间的嵌入有着更加复杂的含义。嵌入类型所附带的方法都会无条件地与被嵌入的结构体类型关联在一起，即它们也成为了被嵌入的结构体类型的方法。这意味着，结构体类型自动地实现了它包含的所有嵌入类型所实现的接口类型。但是，请注意，嵌入类型的方法的接收者类型仍然是该嵌入类型，而不是那个被嵌入的结构体类型。当我们在被嵌入的结构体值上调用实际上属于其中某个嵌入类型的方法的时候，这一调用会被自动转发到这个嵌入类型的值上。

现在，我们对Sequence的声明进行一些改动：

```
type Sequence struct {
    Sortable
    sorted bool
}
```

如上，我们几乎去掉了所有的字段，只留下了Sortable。现在，存储和操作可排序序列的功能都交给了匿名字段Sortable。然后，我们又添加了一个布尔类型的字段sorted，并用它来表示序列类型值是否已被排序。

如果我们有一个Sequence类型的值seq，那么就可以直接在这个值上调用Sortable接口类型中包含的那些方法了，如seq.Sort()。

假如Sequence类型中也包含了一个与Sortable接口类型的Sort方法的名称和签名都相同的方法的话，那么调用表达式seq.Sort()就一定是对Sequence类型值自身附带的Sort方法的调用。也就是说，在这种情况下，嵌入类型Sortable的方法Sort被隐藏了。

这种隐藏会带来一些便利。例如，如果我们需要在原有的排序操作上添加一些额外的功能的话，就可以很方便地声明这样一个同名方法：

```
func (self *Sequence) Sort() {
    self.Sortable.Sort()
    self.sorted = true
}
```

这个与Sequence类型关联的Sort方法把排序操作全权委托给了嵌入类型Sortable的Sort方法，并且在排序操作完成后还对Sequence类型的sorted字段进行了赋值。这就达到了对其匿名字段Sortable的Sort方法的功能进行无缝扩展的目的。这在无形中丰富了调用表达式seq.Sort()的含义。熟悉设计模式的读者可能会由此联想到装饰器模式。

注意，如果这两个Sort方法的名称相同但签名不同，那么嵌入类型Sortable的方法Sort也同样会被隐藏。这时，在Sequence的类型值上调用Sort方法的时候，就必须依据被该类型的Sort方法的签名来编写调用表达式。假设Sequence类型附带的那个名为Sort的方法如下所示：

```
func (self *Sequence) Sort(quick sort bool) {
    // 省略若干条语句
}
```

那么调用表达式seq.Sort()就会造成一个编译错误，因为那个Sortable的无参数的Sort方法已经被隐藏了。我们不能对它进行调用，只能通过seq.Sort(true)或seq.Sort(false)来对Sequence的Sort方法进行调用。不过，我们总是可以使用调用表达式seq.Sortable.Sort()来调用嵌入类型Sortable的Sort方法，不论被嵌入类型是否包含了同名的方法。

在上面的描述中，为了简洁，我们一直没有区分嵌入类型是一个非指针的数据类型，还是一个指针类型。实际上这两种情况是有区别的。假设有结构体类型S和非指针类型的数据类型T，则：

- 如果在S中包含了一个嵌入类型T，那么S和*S的方法集合中都会包含接收者类型为T的方法。除此之外，*S的方法集合中还会包含接收者类型为*T的方法。
- 如果在S中包含了一个嵌入类型*T，那么S和*S的方法集合中都会包含接收者类型为T或*T的所有方法。

其中*S和*T分别代表了指向S的指针类型和指向T的指针类型。与指针类型相关的知识我们会在下一小节介绍。

现在我们来讨论另外一个问题。对于嵌入类型的字段来说，我们同样可以像访问被嵌入的结构体类型的字段那样来访问它们。假设，我们有一个名为List的结构体类型，并且在它的声明中嵌入了类型Sequence：

```
type List struct {
    Sequence
}
```

那么对于List类型的值list来说，选择表达式list.sorted就代表着对嵌入的Sequence类型值的字段sorted的访问。但是，如果List类型也有一个名称为sorted的字段的话，那么其中的Sequence类型值的sorted字段就会被隐藏。选择表达式list.sorted只代表对List类型的sorted字段的访问。不论这两个名称为sorted的字段类型是否相同，都会是这样。然而，我们同样可以通过选择表达式list.Sequence.sorted访问到嵌入类型Sequence的值的sorted字段。

对于结构体类型的多层嵌入来讲，上述的规则同样适用。只要记住以下两点。

- ❑ 我们可以在被嵌入的结构体类型的值上像调用它自己的字段或方法那样调用任意深度的嵌入类型值的字段或方法。唯一的前提条件就是这些嵌入类型的字段或方法没有被隐藏。如果它们被隐藏，我们也可以通过链式的选择表达式或调用表达式访问或调用它们，如list.Sequence.sorted。
- ❑ 被嵌入的结构体类型的字段或方法可以隐藏任意深度的嵌入类型的同名字段或方法。这包括，任何较浅层次的嵌入类型的字段或方法都会隐藏较深层次的嵌入类型包含的同名的字段或方法。注意，这种隐藏是可以交叉进行的，即字段可以隐藏方法，方法也可以隐藏字段，只要它们的名称相同即可。

此外，如果在同一嵌入层次中的两个嵌入类型拥有同名的字段或方法，那么涉及它们的选择表达式或调用表达式将会造成一个编译错误。因为编译器不能确定被选择或调用的目标。

好了，对结构体类型的嵌入类型的讨论就到这里。

现在我们来谈谈匿名结构体类型。匿名结构体类型比命名结构体类型少了关键字type和类型名称，它的声明如下：

```
struct {
    Sortable
    sorted bool
}
```

匿名结构体类型在类型特性和声明规则方面与命名结构体类型是完全一致的。我们在前面提到过，可以在数组类型、切片类型或字典类型的声明中，将一个匿名的结构体类型作为它们的元素的类型。除此之外，我们还可以直接将匿名结构体类型作为一个变量的类型，如：

```
var anonym struct {
    a int
    b string
}
```

不过，更常用的做法是在声明以匿名结构体类型为类型的变量的同时对其初始化：

```
anonym := struct {
    a int
    b string
}{0, "string"}
```

与命名结构体类型相比，匿名结构体类型更像是“一次性”的类型。它不具有通用性，因此它常常被用在临时数据存储和传递的场景中。

最后值得一提的是，我们可以在结构体类型声明中的字段声明的后面添加一个字符串字面量标签，以作为对应字段的附加属性，如下所示：

```
type Person struct {
    Name    string `json:"name"`
    Age     uint8  `json:"age"`
    Address string `json:"addr"`
}
```

如上所示，字段的字符串字面量标签一般由两个反引号“```”包裹的任意字符串组成。并且，它应该被添加在与其对应的字段的同一行的最右侧。在通常情况下，这种标签对于使用该结构体类型及其值的代码来说是不可见的。但是，我们可以用标准库代码包`reflect`中提供的函数查看到结构体类型中字段的标签。因此，这种标签常常会在一些特殊应用场景下使用，比如，标准库代码包`encoding/json`中的函数会根据这种标签的内容确定与该结构体类型中的字段对应的JSON节点的名称。

2. 值表示法

结构体值一般由复合字面量来表达。我们之前说过，复合字面量由类型字面量和由花括号“`{`”和“`}`”括起来的若干键值对组成。对于结构体值来讲，键值对中的键就是结构体类型中某个字段的名称，而值（或称元素）就是要赋给该字段的那个值。我们常常把用于表示结构体值的复合字面量简称为结构体字面量。在同一个结构体字面量中，一个字段名称只能出现一次。也就是说，我们只能在结构体字面量中对同一个字段赋值一次。以上面创建的结构体类型`Sequence`为例，我们可以这样来表示它的值：

```
Sequence{Sortable: SortableStrings{"3", "2", "1"}, sorted: false}
```

类型`SortableStrings`实现了接口类型`Sortable`，因此我们可以把一个`SortableStrings`类型的值赋给`Sortable`字段。另外，我们将`false`赋给了字段`sorted`。

编写结构体字面量的方法不止一种。我们还可以忽略掉字段的名称，也就是说不添加结构体字面量中的键值对的键。不过，在这种情况下会有两个限制。

- ❑ 如果想要省略掉其中某个或某些键值对的键，那么其他的键值对的键也必须省略。也就是说，要么给出其中的每个值所对应的字段的名称，要么省略掉所有的字段名称。例如：

```
Sequence{SortableStrings{"3", "2", "1"}, sorted: false}
```

是不合法的，因为我们只指定了部分的值所对应字段的名称。这会造成编译错误。

- ❑ 多个字段值之间的顺序应该与结构体类型声明中的字段声明的顺序一致，并且不能够省略

掉对任何一字段的赋值。这种限制对于不省略字段名称的字面量来说是不存在的。例如：

```
Sequence{sorted: false, Sortable: SortableStrings{"3", "2", "1"}}
```

和

```
Sequence{Sortable: SortableStrings{"3", "2", "1"}}
```

都是合法的结构体字面量。未被明确赋值的字段的值将会被其类型的零值填充。但是，

```
Sequence{false, SortableStrings{"3", "2", "1"}}
```

和

```
Sequence{SortableStrings{"3", "2", "1"}}
```

都是不合法的。它们都会使Go语言编译器报错。

此外，我们也可以在结构体字面量中不指定任何字段的值。例如，我们可以这样表示Sequence类型的值：

```
Sequence{}
```

这种情况下，此值中的两个字段都会被赋予它们所属类型的零值。当然，我们也可以在之后改变这些字段的值，不过这需要在字段访问权限允许的情况下进行。也就是说，当字段名称的首字母是小写字母时，我们只能在结构体类型声明所属的代码包中访问到该类型的值中的字段，或者对它们进行赋值。这对于结构体字面量来说也是一样的。当结构体字面量处于其类型声明所属的代码包之外时，我们是不能对其中的名称首字母为小写字母的字段进行初始化的。

与数组类型相同，结构体类型属于值类型。结构体值的零值就是我们刚刚提到的那个不为任何字段赋值的结构体字面量。

3. 属性和基本操作

一个结构体类型的属性就是它所包含的字段和与它关联的方法。在访问权限允许的情况下，我们可以使用选择表达式访问结构体值中的字段，也可以使用调用表达式调用结构体值关联的方法。关于它们，我们刚刚已经介绍得足够多了。

需要强调的是，在Go语言中，只存在嵌入而不存在继承的概念。因此，我们不能把在前面声明的那个List类型的值赋给一个Sequence类型的变量。这样的赋值语句会造成一个编译错误。

另外，在一个结构体类型的别名类型的值上，我们既不能调用那个结构体类型的方法，也不能调用与那个结构体类型对应的指针类型的方法。这也是由于在Go语言中没有继承这种说法。别名类型也不是被它“别名”的那个数据类型（也可以称之为别名类型的源类型）的子类型。但是，别名类型内部的结构会与它的源类型一致。比如我们在前面提到过的结构体类型SortableStrings。还记得吗？它的最新版本的声明是这样的：

```
type SortableStrings []string
```

可以确定的是，类型SortableStrings的内部结构是与元素类型为string的切片类型是一致的。正因为如此，我们才可以把一个SortableStrings类型的值像下面这样转换为一个[]string类型的值：

```
[]string(SortableStrings{"4", "5", "6"})
```

反之亦然：

```
SortableStrings([]string{"4", "5", "6"})
```

对于一个结构体类型的别名类型来说，它拥有源类型的全部字段。但是，就像刚才说得那样，这个别名类型并没有继承与它的源类型关联的任何方法。下面举个例子。

如果我们没有把Sequence类型嵌入到List类型当中，而是把List类型作为Sequence类型的一个别名类型，那么它的声明将会是这样：

```
type List Sequence
```

这时，List类型的值的表示方法与Sequence类型的值的表示方法无异：

```
List{SortableStrings{"4", "5", "6"}, false}
```

如果有一个List类型的值list，那么选择表达式list.sorted访问的就是这个List类型的值的sorted字段。当然，我们也可以通过选择表达式list.Sortable访问这个值的嵌入字段Sortable。但是，这个List类型目前却不包含任何方法。

现在我们翻回来看结构体类型Sequence。正因为别名类型存在这样的局限性，我们才在Sequence类型中嵌入了接口类型Sortable，而不是直接将Sequence类型声明为一个接口类型Sortable的某个实现类型（如SortableStrings类型）的别名类型。不过这样做的原因不只上面这一个。比如，嵌入字段Sortable能够用于存储所有实现了该接口类型的数据类型的值。这样的类型结构设计使得Sequence类型可以在一定程度上模拟出泛型类型的一些特点。

泛型是强类型编程语言可以有的一种特性。它允许我们在编写代码时定义一些可变部分。这些可变部分就是泛型的参数，或者说是泛型类型的类型参数。泛型的参数可以代表一类程序实体。比如说，在声明SortableStrings类型的时候，可以通过设定类型参数来指定哪些类型的值可以作为SortableStrings类型值的元素值，然后在初始化SortableStrings类型值的时候再去确定这个具体的类型。这样就可以避免为了支持不同类型的元素而编写多个Sortable接口类型的实现了。如此一来，SortableStrings类型就应该被更名为SortableSlice。因为它允许使用方自己选择具体的元素类型，而不只是把其元素类型固定为string类型。

然而，非常遗憾，虽然Go语言中很多预定义类型都属于泛型类型（比如数组类型、切片类型、字典类型和通道类型），但它却不支持自定义的泛型类型。比如，我们不能使一个自定义的结构体类型成为泛型类型。因此，我们只能在声明它们的时候就指定好一切，而不能出现任何可变的分。

为了使Sequence类型能够部分模拟泛型类型的行为特征，只向它嵌入Sortable接口类型是不够的。我们需要对Sortable接口类型进行扩展。记住，不论从修改最小化还是可维护性方面来看，扩展一个接口类型远远要比直接对这个接口类型进行修改要好得多。同时，这也符合“对修改关闭，对扩展开放”的面向对象设计原则。因此，我们应该创建一个新的接口类型，并将Sortable接口类型嵌入其中。这个新的接口类型的声明如下：

```
type GenericSeq interface {
```

```

Sorttable
Append(e interface{}) bool
Set(index int, e interface{}) bool
Delete(index int) (interface{}, bool)
ElemValue(index int) interface{}
ElemType() reflect.Type
Value() interface{}
}

```

可以看到，接口类型GenericSeq中声明了用于添加、修改、删除、查询元素，以及获取元素类型的方法。并且，一个数据类型要想实现GenericSeq接口类型，也必须实现Sorttable接口类型。

现在，我们将嵌入到Sequence类型的Sorttable接口类型改为GenericSeq接口类型。新版本的Sequence类型的声明如下：

```

type Sequence struct {
    GenericSeq
    sorted bool
    elemType reflect.Type
}

```

在这个类型声明中，我们还添加了一个reflect.Type类型（即标准库代码包reflect中的Type类型）的字段elemType。我们要用elemType字段来缓存GenericSeq字段中存储的值的元素类型。

另外，为了能够在改变GenericSeq字段存储的值的及时对字段sorted和elemType的值进行修改，我们还创建了几个与Sequence类型关联的方法，它们的声明如下：

```

func (self *Sequence) Sort() {
    self.GenericSeq.Sort()
    self.sorted = true
}

func (self *Sequence) Append(e interface{}) bool {
    result := self.GenericSeq.Append(e)
    // 省略部分代码
    self.sorted = false
    // 省略部分代码
    return result
}

func (self *Sequence) Set(index int, e interface{}) bool {
    result := self.GenericSeq.Set(index, e)
    // 省略部分代码
    self.sorted = false
    // 省略部分代码
    return result
}

func (self *Sequence) ElemType() reflect.Type {
    // 省略部分代码
    self.elemType = self.GenericSeq.ElemType()
    // 省略部分代码
    return self.elemType
}

```

仔细的读者可能会发现，这些方法分别与接口类型`GenericSeq`或`Sortable`中声明的某个方法有着相同的方法名称和方法签名。也就是说，我们通过这种方式隐藏了`GenericSeq`字段中存储的值的这些同名方法，并达到了对它们进行无缝扩展的效果。之所以说无缝，是因为这对于方法的调用方来说完全是透明的。方法调用方不用修改任何代码就可以获得这种扩展所带来的一切好处。这也是我们让这些方法的签名分别与其隐藏的那个方法的签名完全一致的原因。

到这里，我们描述了新版本的`Sequence`类型以及相关的接口类型和方法的一部分。这部分代码运用到了我们至此讲述的很多知识。当然，在初始化`Sequence`类型值的时候，我们还需要用到一个`GenericSeq`接口类型的实现类型。不过，我们就不在此赘述了。关于这个实现类型以及`Sequence`类型的完整实现代码，请参见与本书配套的`goc2p`项目中的`seq.go`文件。这个文件在该项目的`src/basic`目录中。

结构体类型是Go语言中最复杂的一个数据类型，尤其是在与类型嵌入有关的一些概念和规则方面。同时，结构体类型也是我们在实际开发过程中最常用的一种数据类型。在大多数场景中，它比那些预定义数据类型的别名类型更适合作为接口类型的实现。另外，它还是Go语言支持面向对象编程的主要体现。

3.2.8 指针

指针是一个代表着某个内存地址的值。这个内存地址往往是在内存中存储的另一个变量的值的起始位置。Go语言对指针的支持介于Java语言和C/C++语言之间。它既没有像Java语言那样取消了代码对指针的直接操作的能力，也避免了C/C++语言中由于对指针的滥用而造成的安全和可靠性问题。

Go语言的指针类型指代了指向一个给定类型的变量的指针。它常常被称为指针的基本类型。指针类型是Go语言的复合类型之一。

1. 类型表示法

我们可以通过在任何一个有效的数据类型的左边插入符号`*`来得到与之对应的指针类型。例如，一个元素类型为`int`的切片类型所对应的指针类型是`*[]int`，而我们前文所提到的结构体类型`Sequence`所对应的指针类型是`*Sequence`。注意，如果代表类型的是一个限定标识符（如`sort.StringSlice`），那么表示与其对应的指针类型的字面量应该是`*sort.StringSlice`，而不是`sort.*StringSlice`。

此外，在Go语言中还有一个专门用于存储内存地址的类型`uintptr`。实际上，`uintptr`类型与`int`类型和`uint`类型一样，也属于数值类型。它的值是一个能够保存一个指针值的32位或64位（与程序运行在怎样的计算架构之上有关）无符号整数。也可以说，它的值是指针类型值（以下简称指针值）的位模式（bit pattern）形式。我们在对指针值进行操作的时候会用到`uintptr`类型。

2. 值表示法

如果一个变量`v`的值是可寻址的，那么我们可以使用取址操作符`&`取出与这个值对应的指针值。也就是说，表达式`&v`就代表了指向变量`v`的值的指针值。关于取址操作符`&`的用法说明请参看

3.1.5节。

这里需要特别解释一下“可寻址的”这个词的含义。如果某个值确实被存储在了计算机内存中，并且有一个内存地址可以代表这个值在内存中存储的起始位置，那么我们就可以说这个值以及代表它的变量是可寻址的。

3. 属性和基本操作

指针类型理所当然地属于引用类型。它的零值是nil。

现在来看看在Go语言中都可以对指针值进行哪些操作。提到对指针值的操作就不得不从标准库代码包unsafe讲起。从名称上就可以看出，代码包unsafe中提供的都是不安全的操作。这些操作绕过了（或者说违反了）Go语言的类型安全机制。因此，我们必须仔细审查使用了代码包unsafe中的程序实体的那些代码，以确保它们的类型安全性。

在代码包unsafe中，有一个名为ArbitraryType的类型。从类型声明上看，它是int类型的一个别名类型。但是，正如其名，它实际上可以代表任意的Go语言表达式的结果类型。事实上，它也并不算是unsafe包的一部分。在这里声明它仅出于代码文档化的目的。

除了上面这个文档性质的类型之外，unsafe包还声明了一个名为Pointer的类型。unsafe.Pointer类型代表了ArbitraryType类型的指针类型。这里有4个与unsafe.Pointer类型相关的特殊转换操作。

- 一个指向其他类型值的指针值都可以被转换为一个unsafe.Pointer类型值。例如，如果有一个float32类型的变量f32，那么我们可以这样将与它的对应的指针值转换为一个unsafe.Pointer类型的值：

```
pointer := unsafe.Pointer(&f32)
```

其中，在特殊标记:=右边的就是用于进行转换操作的调用表达式。取址表达式&f32的求值结果是一个*float32类型的值。

- 一个unsafe.Pointer类型值可以被转换为一个与任何类型对应的指针类型的值。下面的代码用于将pointer的值转换为与指向int类型值的指针值，并赋值给变量vptr：

```
vptr := (*int)(pointer)
```

注意，在特殊标记:=右边的这个调用表达式中，左边的圆括号被用于优先运算，即让Go语言把*int看作一个类型，或者说一个整体。而右边的圆括号及其中的内容就代表着需要将变量pointer所代表的值转换为右边圆括号中的那个类型的值。更加需要注意的是，*int类型与*float32类型在内存中的布局是不同的！如果我们在它们之上直接进行类型转换（对应的表达式为(*int)(&f32)）是行不通的，会造成编译错误。当我们使用unsafe.Pointer类型作为中转类型的时候，上面的转换操作看起来好像没什么问题。但是，我们一定会在使用取值表达式*vptr的时候遇到问题。对于内存上的同一段数据，把它分别作为int类型的值和float32类型的值来解析所得出的结果会是完全不同的。因此，在这种情况下，对取值表达式*vptr的求值肯定会产生一个不正确的结果。另外，在有些时候，它还会引发一个运行时恐慌。比如，如果我们把对变量vps的赋值语句改为

```
vptr := (*string)(pointer)
```

那么对取值表达式*vptr的求值就会引发一个运行时恐慌。

□ 一个unsafe.Pointer类型值可以被转换为一个uintptr类型的值。例如：

```
uptr := uintptr(pointer)
```

□ 一个uintptr类型的值也可以被转换为一个unsafe.Pointer类型值。例如：

```
pointer2 := unsafe.Pointer(uptr)
```

也正因为存在这些特殊转换操作，unsafe.Pointer类型使程序绕过Go语言的类型系统并在任意的内存地址上进行读写操作成为了可能。这是非常危险的！必须万分小心地使用它！

另外，我们还可以利用上述特殊转换操作以及unsafe包中声明的Offsetof函数进行有限的指针运算。

我们以之前提到过的结构体类型Person为例，它的声明是这样的：

```
type Person struct {
    Name    string `json:"name"`
    Age     uint8  `json:"age"`
    Address string `json:"addr"`
}
```

现在，我们来初始化它的值，并把它的指针值赋给变量pp：

```
pp := &Person{"Robert", 32, "Beijing, China"}
```

然后，我们利用上述特殊转换操作中的第一条和第三条获取这个结构体值在内存中的存储地址：

```
var puptr = uintptr(unsafe.Pointer(pp))
```

变量puptr的值就是存储上面那个Person类型值的内存地址。由于类型uintptr的值实际上是一个无符号整数，所以我们可以对该类型的值上进行任何算术运算。例如：

```
var npp uintptr = puptr + unsafe.Offsetof(pp.Name)
```

这里我们用到了unsafe包中的Offsetof函数。unsafe.Offsetof函数会返回作为参数的某字段（由相应的选择表达式表示）在其所属的结构体类型之中的存储偏移量。换句话说，该函数的结果值就是在内存中从存储这个结构体值的起始位置到存储其中某字段的值的起始位置之间的距离。这个存储偏移量（或者说距离）的单位是字节，它的值的类型是uintptr。实际上，同一个结构体类型的值在内存中的存储布局是固定的。也就是说，对于同一个结构体类型和它的同一个字段来说，这个存储偏移量总是相同的。

我们现在知道了存储上面那个Person类型值的内存地址，也知道了它的存储起始位置到其中的Name字段值的存储偏移量。依此，我们把它们相加就会得到存储这个结构体值中的Name字段值的内存地址。在上例中，我们把表示这个内存地址的值赋给了uintptr类型的变量npp。

在获得了这个存储Name字段值的内存地址之后，我们可以利用上述特殊转换操作中的第二条和第四条将它还原成指向这个Name字段值的指针类型值。代码如下：

```
var name *string = (*string)(unsafe.Pointer(npp))
```

这样一来，我们就可以很方便地通过取值表达式`*name`获取到这个Name字段的值了。它就是我们之前初始化那个结构体值的时候赋给它的Name字段的字符串值"Robert"。

这里有一个恒等式可以对上述示例中的一些操作进行很好的总结：

```
uintptr(unsafe.Pointer(&s)) + unsafe.Offsetof(s.f) == uintptr(unsafe.Pointer(&s.f))
```

原则上，我们只要获得了存储某个值的内存地址，就可以通过一定的算术运算得到存储在其他内存地址上的值甚至程序。只要能够获取到它们，对它们进行修改就不是一件难事。

在Go语言中的指针运算是相对繁琐的。这也体现了Go语言只对指针运算提供有限支持的策略。除非确实有必要，否则我们不提倡使用`uintptr`类型和`unsafe`包中声明的那些程序实体进行指针运算。但是，与某个数据类型对应的指针类型确实是会常常用到的。在很多应用场景中，它们也是被推荐使用的（如有必要请回顾3.2.5、3.2.6和3.2.7节）。

3

3.2.9 数据初始化

这里的数据初始化是指对某个数据类型的值或变量的初始化。在Go语言中，几乎所有的数据类型的值都可以使用字面量来进行表示和初始化。我们前面讲各个数据类型的时候都分别进行过详细说明。在大多数情况下，我们使用字面量就可以满足初始化值或变量的要求。

除了前面介绍过的这种数据初始化方式之外，Go语言还为我们提供了两个专门用于数据初始化的内建函数`new`和`make`。

这两个函数所做的事情是不同的。同时，它们所针对的数据类型也是不同的。这可能会令人迷惑，不过好在其中的规则还是非常简单的。

1. new

让我们先来看看`new`函数。`new`函数用于为值分配内存。但是与其他编程语言中的`new`（比如Java语言中的关键字`new`）不同的是，它并不会去初始化分配到的内存，而只会清零它。因此，调用表达式`new(T)`被求值时，所做的是为T类型的新值分配并清零一块内存空间，然后将这块内存空间的地址作为结果返回。而这个结果就是指向这个新的T类型值的指针值。它的类型为`*T`。实际上，这个`new`函数返回的`*T`类型值总会指向一个T类型的零值。例如，调用表达式`new(string)`的求值结果指向的是一个`string`类型的零值`""`，而调用表达式`new([3]int)`的求值结果指向的则是一个`[3]int`类型的零值`[3]int{0, 0, 0}`。

正因为有这种干净的内存分配策略，使得我们可以在用内建函数`new`创建某个数据类型的新值之后立刻就可以拿来使用，而不用担心在这个值中会遗留某些初始化的痕迹。我们以标准库代码包`bytes`中的结构体类型`Buffer`为例。`bytes.Buffer`是一个尺寸可变的字节缓冲区。它的零值就是一个立即可用的空缓冲区。因此，调用表达式`new(bytes.Buffer)`的求值结果就是一个指向一个空缓冲区的指针值。之后，我们就可以立即在这个缓冲区上进行读写操作了。显然，这与我们的期望相符。我们当然希望一个新的缓冲区中不包含任何残留数据，即使这些残留数据是对它初始化的时候留下的。相似的，标准库代码包`sync`中的结构体类型`Mutex`也是一个可以`new`后即用的数

据类型。它的零值就是一个处于未锁定状态的互斥量。

内建函数`new`的这种特性为我们提供了一个关于自定义数据类型的可参考的设计规则。例如，在我们自定义一个结构体类型的时候就要考虑到，在其中的每个字段的值都分别为对应类型的零值的时候，这个结构体值就应该已经处于可用的状态。这样，我们在`new`它的时候就能够得到一个立即可用的值的指针值，而不需要再做额外的初始化。

当然，在感觉一个类型的零值还无法让它变得可用的时候，我们可以使用相应的字面量来达到分配内存空间并初始化值的目的。前面我们已经讲过，我们可以在字面量中灵活的指定新值中的每一个元素的值。但是要注意，字面量所代表的是该类型的值，而不是指向该类型值的指针值。因此，我们在将它们与调用`new`函数的调用表达式做等价替换的时候，还需要在字面量的前面加入取址操作符`&`以表示指向该类型值的指针值。

2. `make`

内建函数`make`所做的事情与`new`大有不同。`make`函数只能被用于创建切片类型、字典类型和通道类型的值，并返回一个已被初始化的（即非零值的）的对应类型的值。这么做的原因是与上面这3个复合类型的特殊结构有关的。我们在之前说过，它们都是引用类型。在它们的每一个值的内部都会保持着一个对某个底层数据结构值的引用。如果不对它们的值进行初始化，那么其中的这种引用关系是不会被建立起来的，同时相关的内部值也会不正确。在这种情况下，该类型的值也就不能够被使用，因为它们是不完整的，还处于未就绪的状态。这就意味着，在创建这3个引用类型的值的时候，必须将内存空间分配和数据初始化这两个步骤绑定在一起。也正是为了保证这些值的可用性，切片类型、字典类型和通道类型的零值都是`nil`，而不是那个未被初始化的值。因此，当我们`new`这3个引用类型并想创建它们的值的时候，得到的却是一个指向空值`nil`的指针值。

除此之外，内建函数`make`所接受的参数也与`new`函数有所不同。`make`函数除了会接受一个表示目标类型的类型字面量之外，还会接受一个或两个额外的参数。

对于切片类型来说，我们可以在把新值的长度和容量也传递给`make`函数。例如，调用表达式

```
make([]int, 10, 100)
```

创建了一个新的`[]int`类型的值，这个值的长度为10、容量为100。当然，我们也可以省略掉最后一个参数，即不指定新值的容量。这种情况下，该值的容量会与其长度一致。示例如下：

```
s := make([]int, 10)
```

变量`s`的类型是`[]int`的，而长度和容量都是10。

在使用`make`函数初始化一个切片值的过程中，该值会引用一个长度与其容量相同且元素类型与其元素类型一致的数组值。这个数组值就是该切片值的底层数组。该数组值中的每个元素都是当前元素类型的零值。但是，切片值只会展现出数量与其长度相同的元素。因此，调用表达式`make([]int, 10, 100)`所创建并初始化的值就是`[]int{0 0 0 0 0 0 0 0 0 0}`。

我们在使用`make`函数创建字典类型的值的时候，也可以指定其底层数据结构的长度。但是，该字典值只会展示出我们明确“放入”的键值对。例如，调用表达式

```
make(map[string]int, 100)
```

所创建和初始化的值会是`map[string]int{}`。虽然我们也可以忽略掉那个用于表示底层数据结构长度的参数（像这样：`make(map[string]int)`），但是我还是建议：应该在性能敏感的应用场景下，根据这个字典值可能包含的键值对的数量以及“放入”它们的时间，仔细地设置该长度参数。

我们最后提一下对于通道类型（`Channel`）的值的初始化。我们到此为止还没有对通道类型进行过任何说明。不过，在本书的第7章中，我们会详细讲解这个特殊的引用类型。

我们可以这样使用`make`函数创建一个通道类型的值：

```
make(chan int, 10)
```

其中的第一个参数表示的是通道的类型，而第二个参数则表示该通道的长度。与字典类型相同，第二个参数也可以被忽略掉。但是，忽略它的含义却与针对字典类型的情况有着很大的不同。关于这些知识，我们会在后面专门讲解。

请记住，`make`函数只能被应用在引用类型的值的创建上。并且，它的结果是第一个参数所代表的类型的值，而不是指向这个值的指针值。如果我们想要获得该指针值的话，只能在调用`make`函数的表达式的求值结果之上应用取址操作符`&`，像这样：

```
m := make(map[string]int, 100)
mp := &m
```

到目前为止，我们已经介绍了3种创建值的方法，即使用字面量、调用内建函数`new`和调用内建函数`make`。它们适用于不同的应用场景。

当然，在某些应用场景中，我们可以有多种选择。例如，在创建一个切片类型的值的时候，我们既可以使用字面量也可以使用`make`函数。这种选择的结果往往取决于我们是否需要定制切片值中的某个或某些元素值。又例如，如果我们能够保证一个结构体类型的值在其中字段的值均为零值的情况下就能够处于可用状态的话，那么仅使用`new`函数来初始化它与使用字面量进行初始化是基本等价的。不过要注意，这两种方法产生的结果的类型是不同的。

下面我们来总结一下在本小节中描述的一些规则，以便读者参考。

- 字面量可以被用于初始化几乎所有的Go语言数据类型的值，除了接口类型和通道类型。接口类型没有值，而通道类型的值只能使用`make`函数来创建。如果需要指向值的指针值，那么可以在表示该值的字面量之上进行取址操作。
- 内建函数`new`主要被用于创建值类型的值。调用`new`函数的表达式的结果值将会是指向被创建值的指针值，并且被创建值会是其所属数据类型的零值。因此，`new`函数不适合被用来创建引用类型的值。其直接的原因是引用类型的值的零值都是`nil`，是不可用的。
- 内建函数`make`仅能被用于某些引用类型（切片类型、字典类型和通道类型）的值的创建。它在创建值之后还会对其进行必要的初始化。与`new`函数不同，调用`make`函数的表达式的结果值将会是被创建的值本身，而不是指向它的指针值。

请读者记住这些规则，这在我们根据具体场景来选择值的初始化方法的时候会非常有用。

3.3 数据的使用

前面已经对Go语言的绝大多数数据类型以及相关的概念和操作方法进行了全面、详细的介绍。在本节中，我们将着重说明与这些数据类型及其值有关的应用惯例和最佳实践。

3.3.1 赋值语句

在我们对赋值语句进行说明之前先来了解一下与赋值有关的规则。

如果值x可以被赋给类型为T的变量，那么它们至少需要满足以下条件中的一个。

- 如果值x的类型就是T，那么x理所应当的可以被赋给T类型的变量。
- 如果值x的类型是V，那么V和T应该具有相同的潜在类型，并且它们之中至少有一个是未命名的类型。注意，这里所说的未命名的类型并不是指结构体类型中的匿名字段，而是指未被署名的数据类型。例如，字面量

```
struct {
    a int
    b string
}{0, "string"}
```

所代表的值的类型是

```
struct {
    a int
    b string
}
```

而这个类型就是一个未命名的类型。它的潜在类型与结构体类型

```
type Anonym struct {
    a int
    b string
}
```

相同。因此，上面的那个值可以被赋给类型为Anonym的变量。

- 类型T是一个接口类型，且值x的类型实现了T。这种情况下，x就可以被赋给类型为T的变量。
- 如果值x是一个双向通道类型的值，而T也是一个通道类型。那么x的类型V和T应该具有相同的元素类型，并且它们之中至少有一个是未命名的类型。关于此种情况，我们在第8章讲通道类型的时候再详细说明。在这里，我们可以暂且认为这与值[]string{"1", "2", "3"}和类型[]string之间的关系类似。
- 如果值x预定义标识符nil，那么它可以被赋给切片类型、字典类型、函数类型、接口类型、指针类型和通道类型的变量。也就是说，只要变量不是值类型的，它就可以被赋予空值nil。
- 如果值x是一个由某个数据类型的值代表的无类型的常量，那么它就可以被赋给该数据类型的变量。例如，字符串字面量"ABC"可以被赋给string类型的变量，以及整数字面量123

可以被赋给int类型的变量。显然，我们可以把无类型的常量理解为字面量。

- 所有值都可以被赋给空标识符“_”。空标识符有时也被称为占位标识符。它只起到占位的作用，不会与任何值建立绑定关系。

赋值语句是建立在上述赋值规则的基础之上的。现在让我们来看看赋值语句的编写方法。

赋值语句一般由左右分立的两个表达式列表和处于中间的一个赋值操作符组成，例如：

```
var ints = []int{1, 2, 3}
```

表达式列表中的多个表达式之间需要有逗号“,”作为分隔符。在大多数情况下，左右两边的表达式的数量必须是相同的。当左边表达式的数量大于1时，就形成了多个赋值操作同时进行的情况。这种情况常常被称为平行赋值。

赋值操作符肯定会包含符号=。并且，作为可选项，在符号=的左边可以插入一个可被用于二元运算的算术操作符，记为“op=”，例如+=、*=或&=。

在赋值操作符左边的那个表达式列表中的每个表达式的结果值都必须是可寻址的，例如：

```
ints[1], ints[2] = (ints[1] + 1), (ints[2] + 2)
```

在上面这个示例中，表达式(ints[1] + 1)的唯一结果值会被赋给ints[1]（也就是切片值ints中与索引值1对应的位置），而表达式(ints[2] + 2)的唯一结果值会被赋给ints[2]（也就是切片值ints中与索引值2对应的位置）。

当然，如果我们并不需要对赋值操作符右边的某个表达式的结果值进行绑定，可以在赋值操作符左边的相应位置上应用空标识符“_”。例如，如果我们不想对表达式(ints[2] + 2)的结果值进行任何绑定，那么可以这样做：

```
ints[1], _ = (ints[1] + 1), (ints[2] + 2)
```

可以看到，我们用圆括号“(”和“)”括起了上面示例中的某些表达式。实际上，每个表达式都可以被圆括号括起来。当赋值语句中的表达式相对复杂的时候，我们就可以采用这种方式以使赋值语句的表达更加清晰。另外，在表达式的内部也可以出现任意对圆括号，例如：

```
ints[1], ints[2] = (ints[1] * (ints[0] + 1)), (ints[2] * (ints[0] + 2))
```

如果说=是基本的赋值操作符的话，那么在左边插入了算术操作符的赋值操作符就属于非基本的赋值操作符。非基本的赋值操作符的语义是这样的：赋值语句

```
x op= y
```

等效于

```
x = x op y
```

显然，包含了非基本赋值操作符的赋值语句就是普通赋值语句的一种简写形式。也正因为这种等效性，非基本的赋值操作符并不能被用于平行赋值。也就是说，在这类赋值操作符的左右两边只能各存在一个表达式。并且，这两个表达式都应该是单值表达式。所谓单值表达式，就是其结果值的数量为1的表达式。例如，有这样一个函数func Save(p Person) (id int, done bool)，那么针对它的调用表达式Save(Person{})就不是一个单值表达式。

作为非基本赋值操作符的例子，有：

```
i1 += 1 // 等效于: i1 = i1 + 1
```

和

```
i2 &= 2 << 3 // 等效于: i2 = i2 & (2 << 3)
```

对于普通赋值语句（以`=`为赋值操作符的赋值语句）来说，在赋值操作符两边的表达式的数量可以不相等。当在赋值操作符的右边只有一个表达式且该表达式是一个多值表达式（与单值表达式相对应）的时候，在赋值操作符的左边的表达式可以有多个。对于右边的那个唯一的表达式，有下面4种情况。

- 此表达式是一个调用会返回多个结果的函数或者方法的表达式。这时，在赋值操作符的左边的表达式的数量应该等于该函数或方法的结果的数量。例如，以调用表达式 `Save(Person{})` 作为右边表达式的赋值语句可以是 `id, done := Save(Person{})`。
- 此表达式是一个应用于字典值之上的索引表达式。这时，在赋值操作符的左边可以有一个或两个表达式。我们在前面讲字典类型的时候说过，这类索引表达式可以有两个结果值。例如：`v, ok := map["k1"]`。
- 此表达式是一个类型断言表达式。这时，在赋值操作符的左边可以有一个或两个表达式。我们已经知道，类型断言表达式也可以有两个结果值，如：`v, ok := x.(string)`。
- 此表达式是一个由接收操作符和通道类型值组成的表达式。这时，在赋值操作符的左边可以有一个或两个表达式。与上面的第二种和第三种情况相同，这类表达式的结果值也可以有两个，如：`v, ok := <-ch`。

再次强调，除了上述的4种情况，无论赋值语句中的赋值操作符是否为`=`，在赋值操作符两边的表达式的数量必须一致。并且，无论表达式处在赋值操作符的左边还是右边，它都必须是一个单值表达式。

赋值语句的执行分为两个阶段。第一个阶段，在赋值操作符左边的索引表达式和取址表达式的操作数以及在赋值操作符右边的表达式，都会按照通常的顺序被求值。我们在3.1.6节中说明调用表达式的时候已经对“通常的顺序”这个词进行过详细的解释。第二个阶段，赋值会以从左到右的顺序进行。例如，以下3条赋值语句：

```
i := 1
s := []string{"A", "B", "C"}
i, s[i-1] = 2, "Z"
```

在被执行之后，变量`s`的值是`[]string{"Z", "B", "C"}`而不是`[]string{"A", "Z", "C"}`。这是因为对表达式`i-1`的求值会先于赋值的执行，在赋值真正开始的时候，第二个被赋值对象已经被确定为`s[0]`了。

关于赋值顺序的规则主要是针对平行赋值的场景。我们已经在前面的示例中多次展示过了。不过，有一些与平行赋值有关的特例还需要我们了解一下。例如，我们可以使用平行赋值来交换两个变量的值：

```
a, b = b, a
```

这样很方便，不是吗？我们并不需要一个临时变量来辅助完成这一值交换过程。当然，这两个变量的类型应该符合我们本小节最开始处说的赋值规则。

又例如，如果有一个切片类型的变量`x`，它是这样被初始化的：

```
x := []int{1, 2, 3}
```

下面这条赋值语句又对这个切片值的第一个元素进行了赋值：

```
x[0], x[0] = 1, 2
```

在这条赋值语句中，变量`x`所代表的切片值的第一个元素先被赋予了值1，然后又被赋予了值2。因此，在这条赋值语句被执行之后，索引表达式`x[0]`的求值结果应该是2。这也充分说明了在平行赋值时的赋值顺序。

由于平行赋值永远是从左向右进行的，所以即使靠右的赋值引发了运行时恐慌，它左边的赋值也依然会生效。例如：

```
x[2], x[3] = 4, 5
```

因为`x`代表的切片值只有3个元素，所以索引表达式`x[3]`在被求值的时候会引发一个运行时恐慌。注意，在这个运行时恐慌发生之前，`x[2]`所代表的元素的值已经被变更为了4。所以，不管这个运行时恐慌是否会被“恢复”，变量`x`的值已经被改变了。

最后，再次强调：在赋值语句中，每个右边的表达式的结果值必须是可以被赋给与其相对应的左边的表达式的类型的，即使这些值由无类型的常量代表也一定会是这样。

3.3.2 常量与变量

从数学的角度讲，常量与变量是代表事物量的一对概念。在事物的特定运动过程中，某量若保持不变则被称常量，否则被称为变量。从编程语言的角度讲，常量一旦被声明它的值就不能被改变，而对于变量却没有这种限制。不过，对于像Go语言这样的强类型编程语言来说，我们可以改变一个变量的值但却不能改变它的类型。

1. 常量

在Go语言中，常量总会在编译期间被创建，即使它们作为局部变量被定义在了函数内部。也正因为如此，常量只能由字面常量或常量表达式来赋值。常量表达式是能够且总会在编译期间被求值的。而其他的表达式只能在程序运行期间被求值，所以它们并不能被赋给常量。

Go语言的常量可以被划分为布尔常量、`rune`常量（也被称为字符常量）、整数常量、浮点数常量、复数常量和字符串常量。其中，字符常量、整数常量、浮点数常量和复数常量又被统称为数值型常量。数值型常量可以代表任意精度的数值。数值型常量永远不会出现溢出的情况。

一个常量的值一般由布尔字面量、`rune`字面量、整数字面量、浮点数字面量、复数字面量或字符串字面量表示。换句话说，由相应的字面量表示的基本数据类型的值都可以被称为常量值，不论它们出现在什么地方、充当怎样的角色。由字面量表示的常量值也被简称为字面常量。例如，布尔字面量`true`是一个字面常量，而`rune`字面量`'好'`也是一个字面常量。又例如，内建函数`len`在被调用后会返回一个`int`类型的值作为结果，这个结果值也是一个字面常量。

常量可以是有类型的也可以是无类型的。由字面量表示的常量，如`true`、`false`、`"A"`和`iota`，以及由仅以无类型的常量作为其操作数的常量表达式的结果值都属于无类型的常量。顺便提一句，这里的`iota`是一个预定义标识符。我们稍后再对它进行说明。

常量可以被显式地给定类型。我们可以通过常量声明或者数据类型转换将一个无类型的常量类型化。常量也可以被隐含地给定类型。例如，当我们将一个基本数据类型的值赋予一个变量或者将它作为操作数置于一个表达式中的时候。注意，我们不能把一个字面常量与一个不相称的类型关联到一起。例如，字面常量`10.0`只可以被给定为任何一个整数类型或者任何一个浮点数类型。又例如，字面常量`2147483648.0`可以被给定为`float32`类型、`float64`类型或`uint32`类型。但是，它不能被给定为`int32`类型和`string`类型。因为，这个字面常量既不在`int32`类型可表示的范围内，也没有被双引号包裹起来。

最后，值得一提的是，虽然从语言规范上来说，数值型常量可以有任意的精度，但编译器却只会使用一个有限精度的内部表示方法来实现它们。对于每一个实现，都必须满足下列条件。

- 整数字面量至少要用256个比特位来表示。
- 浮点数字面量的小数部分至少要用256个比特位来表示，而其指数部分至少要用32个比特位来表示。对于复数常量的实部和虚部中的相应部分也是如此。
- 若不能精确地表示一个整数常量，则要给出一个错误。
- 若由于溢出而不能表示一个浮点数常量或复数常量，则要给出一个错误。
- 若由于精度限制而不能表示一个浮点数常量或复数常量，则这个值会被四舍五入为一个可表示的最相近的常量。

这些要求对于字面常量和常量表达式的求值结果来说都是强制性的。

2. 常量表达式

我们在前面多次提到了常量表达式。那么什么是常量表达式呢？常量表达式就是仅以常量作为操作数的表达式。

无类型的布尔常量、数值常量和字符串常量都可以作为常量表达式的操作数。如果一个二元操作的操作数是两个不同种类的无类型的数值型常量，那么对于非布尔操作（不包含比较操作符的操作）来说其操作结果的种类将会遵循下面所展示的优先级顺序（从低到高）：整数、`rune`、浮点数、复数。例如，表达式

```
2 + 3.0
```

的结果是一个无类型的浮点数常量`5.0`。又例如，常量表达式

```
15 / 4.0
```

的结果是一个无类型的浮点数常量`3.75`。而常量表达式

```
'w' + 1
```

的结果是一个无类型的`rune`常量`'x'`。

不过这也有例外，那就是当常量表达式中的操作符是移位操作符`<<`或`>>`的时候，其结果的种类是固定的。例如，常量表达式

```
1 << 3.0
```

和

```
1.0 << 3
```

的结果都是一个无类型的整数常量8。

实际上，操作数为无类型常量的移位操作的结果总会是一个无类型的整数常量。而对于比较操作来说，其结果总会是一个无类型的布尔常量。例如，常量表达式

```
"A" > "C"
```

的结果是一个无类型的布尔常量false。

常量表达式总会被正确地求值。中间值和作为表达式结果的常量值自身都会有足够的精度。这个精度可以比Go语句中预定义的那些类型所支持的精度更高。例如，常量表达式

```
1 << 100
```

的结果是一个无类型的整数常量1267650600228229401496703205376。这个常量值实际上已经超出了Go语言中任何一个整数类型（即使是uint64类型）所能表示的范围了。

对于有类型的常量来说，它的值必须永远能够被精确地表示为其类型的值。从这方面讲，有类型的常量几乎就与变量无异了，除了它只能被赋值一次这个限制之外。

下面我们来讨论怎样对无类型或有类型的常量进行声明。

3. 常量的声明

常量声明会将字面常量或常量表达式与标识符绑定在一起。在上一小节讲赋值语句的时候，我们只关注了对变量的赋值，而常量也是需要被赋值之后才能够被使用的。与变量不同的是，对常量的赋值必须与其声明同时进行。并且，我们只可以对常量赋值一次。

一条常量声明语句总会以关键字const开始。例如：

```
const untypedConstant = 10.0
```

上面这条语句声明了一个名为untypedConstant的常量，并把它与浮点数常量10.0绑定在了一起。这个常量属于无类型常量。对于一个无类型的常量的声明来说，被声明的常量仍然会是无类型的，并且常量标识符代表了赋值操作符=右边的字面常量或常量表达式的结果值。就上例来讲，常量标识符untypedConstant表示了一个浮点数常量，因为在赋值操作符右边的是一个浮点数字面量。

当然，我们可以声明一个有类型的常量，如下：

```
const typedConstant int64 = 1024
```

我们总是应该把常量的类型放在其名称和赋值操作符的中间，并且使用空格“ ”来分隔它们。对于一个有类型的常量的声明来说，赋值操作符右边的字面常量或常量表达式与该类型之间必须满足我们在上一小节开始处说明的赋值规则。

在常量声明语句中还可以包含平行赋值。请看下面的示例：

```
const tc1, tc2, tc3 int64 = 1024, -10, 88
```

这条赋值语句将标识符tc1、tc2和tc3分别与字面常量1024、-10和88绑定在了一起。注意，

在这3个常量标识符和赋值操作符之间，只有一个被用于表示类型的字面量`int64`。在包含平行赋值的常量声明语句中，如果类型被给定，那么所有的常量的类型都应该与这个被给定的类型一致。相应地，赋值操作符右边的字面常量或常量表达式的结果值也都必须可以被赋给这个类型。实际上，在此处插入一个以上的类型字面量也是不被允许的。如果在包含平行赋值的常量声明语句中未给定类型，那么赋值操作符右边的多个字面常量或常量表达式的结果值的种类都会是彼此独立的，即它们的种类都可以是任意的。例如：

```
const utc1, utc2, utc3 = 6.3, false, "C"
```

在上面这条常量声明语句中，标识符`utc1`、`utc2`和`utc3`分别表示了一个浮点数常量、一个布尔常量和一个字符串常量。

我们可以把对多个常量的声明拆分多行。这会与上面展示的那种编写方法稍有不同。拆分的一个最明显的好处是，我们只需要写一次`const`了。我们可以把上面那条常量声明语句改写成这样：

```
const (  
    utc1 = 6.3  
    utc2 = false  
    utc3 = "C"  
)
```

由于只有一个`const`，所以我们需要把后面的几行内容用圆括号“(”和“)”括起来，以表示它们都是对常量的声明和赋值。在这个圆括号中的每一行都可以被称为一个常量声明。在圆括号中的常量声明内，我们同样可以使用平行赋值：

```
const (  
    utc1, utc2 = 6.3, false  
    utc3      = "C"  
)
```

在带圆括号的常量声明语句中，有时候我们并不需要显式地对所有的常量进行赋值。被省略了赋值的常量实际上还是有值的，只不过这个值是被隐含地赋予的。它们的值及其类型都会与在其上面的、最近的且被显式赋值的那个常量相同。也正因为如此，在带圆括号的常量声明语句中，对第一个被声明的常量的赋值是永远不能够被省略的。请看下面的例子：

```
const (  
    utc1, utc2 = 6.3, false  
    utc3      = "C"  
    utc4  
    utc5  
)
```

常量`utc4`和`utc5`的值及其类型都会与在它们上面被声明的常量`utc3`相同。显然，这个语法糖是为了让我们在声明拥有相同值的多个常量的时候能够少敲一些代码。有两个与此有关的约束需要读者注意。

- 如果有一个未被显式赋值的常量，那么与它同一行的常量（如果有的话）的赋值也都必须被省略。
- 在未包含显式赋值的那一行常量声明中的常量标识符的数量必须与在它上面的、最近的且包含显式赋值的那一行常量声明中的常量标识符的数量相等。

例如，常量声明语句

```
const (
    utc1, utc2, utc3 = 6.3, false, "C"
    utc4, utc5
)
```

是不合法的，会造成一个编译错误。原因是，在省略赋值的那一行常量声明中有两个常量标识符，而在此行上面的未省略赋值的那一行常量声明中却有3个常量标识符。在这两行中的常量标识符的数量是不同的。这违背了上述的第二个约束。不过，这一问题的解决方法极其简单，如下：

```
const (
    utc1      = 6.3
    utc2, utc3 = false, "C"
    utc4, utc5
)
```

现在让我们更近一步。我们不但可以为这样隐含地对多个常量赋予同一个值，而且还可以更加方便地对多个常量分别赋予一系列连续的值。这会用到我们在本小节之前提到的预定义标识符 `iota`。示例如下：

```
const (
    Sunday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)
```

为了弄清楚上面这条常量声明语句中的每个常量的值，我们必须先来解释一下 `iota` 的含义和作用。在常量声明语句中，`iota` 代表了连续的、无类型的整数常量。它第一次出现在一个以 `const` 开始的常量声明语句中的时候总会表示整数常量0。

在同一条常量声明语句中，`iota` 在第二个包含它的常量声明中会表示为整数常量1，在第三个包含它的常量声明中会表示为2，在第四个包含它的常量声明中会表示为3，以此类推。也就是说，随着在同一条常量声明语句中包含 `iota` 的常量声明的数量的增加，`iota` 所表示的整数值也会递增。

我们先来看看一个最简单的例子：

```
const x = iota
const y = iota
```

常量 `x` 和 `y` 的值都是整数常量0，因为这两个 `iota` 分别出现在了这两条常量声明语句中。它们之

间并不存在递增关系。现在，我们再在后面加上一条常量声明语句：

```
const x = iota
const y = iota

const (
    a = iota
    b
    c
)
```

在上面的第三条常量声明语句中，我们使用了常量隐式赋值。与常量x和y相同，常量a的值也是0，原因我们刚刚已经说过了。而常量b和c的值分别是1和2。因为它们所在的行分别包括了在当前的常量声明语句中第二个和第三个包含iota的常量声明。这样的一条常量声明语句使常量a、b、c的值之间存在了连续的递增关系。

其实，我们可以利用iota进行更加灵活的常量隐式赋值。例如：

```
const (
    u = 1 << iota
    v
    w
)
```

常量u、v和w的值分别是1、2和4。实际上，这3个常量的值分别是常量表达式1 << 0、1 << 1和1 << 2的结果值。

现在来看更复杂的。如果iota同时遇上了常量隐式赋值和平行赋值会怎么样呢？示例如下：

```
const (
    e, f = iota, 1 << iota
    g, h
    i, j
)
```

首先，我们来看常量e和f。它们的值分别是0和1（常量表达式1 << 0的结果值）。在这里，读者可能会有疑惑。iota明明出现了两次，但为什么它在字面常量iota和常量表达式1 << iota中代表的都是0呢？这里确实是一个比较容易混淆的地方。请记住，在同一条常量声明语句中，iota代表的整数常量的是否递增取决于是否又有一个常量声明包含了它，而不是它是否又在常量声明中出现了一次。由此可知，常量g和h的值分别是1和2（常量表达式1 << 1的结果值），而常量i和j的值分别是2和4（常量表达式1 << 2的结果值）。

最后一个关于iota的小技巧是，我们可以利用空标识符“_”来跳过iota表示的递增序列中的某个或某些值，像这样：

```
const (
    e, f = iota, 1 << iota
    _, _
    g, h
    i, j
)
```

注意，在这条常量声明语句中的第二个常量声明是`_`，`_`。也就是说，我们使用了两个空标识符跳过了`iota`的值为1的情况，而到了第三个常量声明那里，`iota`的值已被递增为了2。因此，在这种情况下，常量`g`和`h`的值分别是2和4，常量`i`和`j`的值分别是3和8。

现在稍微总结一下。我们在对常量进行声明的时候必须同时对它进行赋值。对一个常量的赋值只能进行一次，且只有字面常量和常量表达式可以作为它的值。另外，我们可以利用隐式赋值、平行赋值和`itoa`对常量进行非常灵活和复杂的赋值操作。

4. 变量

变量与常量的最主要的区别是它在被声明之后可以被赋值任意次。并且，我们可以使用普通的表达式给变量赋值，而不仅仅局限于常量表达式。因为，对于变量来说，它的值是在程序运行期间才被计算出来的。

5. 变量声明

一个变量声明可以将一个标识符与一个变量值绑定在一起。前提条件是这个变量值与该变量的类型之间必须要满足赋值规则。

变量声明语句总是会以关键字`var`开始：

```
var v int64 = 0
```

这与有类型的常量的声明非常类似。比如，我们也可以省略变量的类型：

```
var v = 0
```

如果变量的类型未被显式地指定，那么它将会由变量值推导得出。如果在省略类型的同时，赋值操作符右边的表达式的求值结果是一个字面常量，那么该变量的类型将会根据这个字面常量的种类被推导出来。具体规则如表3-7所示：

表3-7 字面常量与变量类型的对应关系

字面常量的种类	变量的类型
布尔常量	<code>bool</code>
字符常量	<code>rune</code>
整数常量	<code>int</code>
浮点数常量	<code>float64</code>
复数常量	<code>complex128</code>
字符串常量	<code>string</code>

实际上，在上述情况下，Go语言的运行时程序会根据字面常量的种类将其转换为对应的数据类型，然后再赋给相应的变量。

另外，常量声明中的大多数语法糖在变量声明中也是受支持的。比如，我们也可以对多个变量进行平行赋值：

```
var v1, v2 = 0, -1
```

或者把对多个变量的声明拆分成多行，并用圆括号“(”和“)”括起来：

```
var (  
    v1 = 0  
    v2 = -1  
)
```

但是，请注意，隐式赋值在变量声明中是不可用的。

变量声明与常量声明另一个不同是：我们在声明一个变量的时候可以不对它进行赋值。也就是说，我们可以不对一个新声明的变量的值进行初始化。如果初始的显式赋值被省略，那么变量的值将会是与该变量的类型相对应的零值。但要注意，在这种情况下，变量的类型就不可以被省略了。此外，如果是平行赋值的话，我们要么省略其中所有变量的初始赋值，要么就必须对所有变量进行初始赋值。例如：

```
var v3, v4, v5 float64
```

和

```
var v3, v4, v5 float64 = 3, 4, 5
```

如果变量声明中包含了对变量的显式赋值，那么在一些情况下赋值操作符两边的表达式的数量也可以是不相等的。这个问题我们在前面讲赋值语句的时候已经说明过。

6. 局部变量

与常量相同，变量声明可以作为源码文件中的顶级元素，也可以成为函数体内容的一部分。前者可以被称为全局变量，后者可以被称为某个函数的局部变量。局部变量有时也被称为本地变量。下面举一个例子，有一个命令源码文件的内容是这样的：

```
package main  
  
import "fmt"  
  
var v6 bool  
  
func main() {  
    var v6 bool = true  
    fmt.Printf("v6: %v\n", v6)  
}
```

在这个源码文件中有两个名为v6的布尔类型的变量。上面的变量v6是一个全局变量，而下面的变量v6则是main函数的局部变量。也正因为它们的作用域不同，所以即使同名也不会引起编译错误。

实际上，在函数体内部，局部变量会遮蔽与它同名的全局变量。因此，在上面的main函数体中的第二行代码将会在标准输出上打印字符串v6: true。显然，这行代码中的v6指的是局部变量v6。由于我们在声明全局变量v6的时候并没有对它进行初始化，因此它的值是布尔类型的默认值false。

我们在函数体内部声明变量的时候可以采用一种更加简短的声明方式，如：

```
v6 := true
```

我们在前面的内容中已经多次见到过这种声明方式。这种声明方式又被称为短变量声明。短

变量声明被用于声明并初始化局部变量。在短变量声明中，我们并不需要也不能显式地给定变量的类型。也就是说，在这种声明中的变量的类型总是根据它的值推导得到的。另外，短变量声明也不需要以`var`开始。因为特殊标记`:=`只会被用于对变量的声明和初始化的语句中，所以并不会产生任何歧义。短变量声明与普通变量声明一样，也支持平行赋值，例如：

```
v7, v8 := "Go", 1.2
```

一般情况下，在`:=`左边的标识符在当前的上下文环境中（或者说语句块中）都应该是新的。也就是说，它们所代表的都应该是新的变量。不过这也有例外情况。这种例外情况只会出现在我们刚刚演示过的包含平行赋值的短变量声明中。请看下面的示例：

```
v8, v9 := 2.0, false
```

在这个短变量声明中，`v8`是我们在之前已经声明且初始化过的变量，而`v9`则是一个新的变量。这时，我们在这个短变量声明中对变量`v8`进行了重声明。我们可以把这种重声明理解为对在当前上下文环境中的已存在变量的又一次赋值。注意，我们赋给被重声明的变量的那个值与该变量的类型之间也必须满足我们之前说过的赋值规则。

重声明仅会出现在短变量声明中。我们不能用普通的声明方式重新声明一个在当前上下文环境中已存在的变量，不论该变量是一个全局变量还是一个局部变量。

至此，我们强调一下针对短变量声明的约束条件。

□ 短变量声明仅能够在函数体内部声明变量的时候使用。

□ 在短变量声明中的`:=`的左边的标识符至少要有一个代表在当前上下文环境中的新变量。

注意，空标识符“`_`”代表的并不是新的变量，即使它在当前上下文环境中并没有出现过。

最后，值得一提的是，短变量声明可以出现在`if`、`for`和`switch`等语句的初始化器中，并被用来声明仅存在于这些语句块中的本地临时变量。我们会在后面讲解相关语句的时候具体说明短变量声明的这种用途。

如果我们在当前上下文环境中声明了某个局部变量但没有使用它，那么就会造成一个编译错误。注意，对变量的赋值并不算是对它的使用。之所以会造成编译错误，是因为Go语言认为这种情况是一种对计算资源的浪费，甚至预示着一个更加严重的问题的出现。

虽然作为一种变通方案，我们可以通过把未使用的局部变量赋给一个空标识符，以使编译器不报错，但是我强烈建议读者仅把这种屏蔽错误的方式作为一种在程序开发期间的临时方案，并且一定要在该段程序投入生产之前删除掉这种赋值语句以暴露出该编译错误，并真正地解决它。

我们上面所讲的常量和变量是在Go语言程序中最常出现的两种程序实体。希望读者通过阅读本小节的内容已经对它们有所了解，并能够在实际代码中合理地运用它们。

3.3.3 可比性与有序性

到目前为止，我们几乎讲解了所有的Go语言数据类型，并涉及了表示方法、基本属性和操作方式、初始化方法和赋值规则等各个方面。在这些内容中，我们少有提及不同数据类型之间以及它们的值之间的关系。比如，怎样判断两个数据类型是否相等。又比如，怎样比较两个具有相

同数据类型的值。在本小节，我们就专门讲解这方面的知识。在有了前面讲到的与数据类型有关的知识作为铺垫之后，我们可以更加顺利地进入这些主题。

1. 类型的恒等

我们先来看看怎样判断两个数据类型是否恒等。

对于两个命名类型来说，要判断它们是否恒等是非常容易的，即如果它们的名称不同，那么它们肯定是不恒等的；如果它们的名称相同且源于相同的类型声明，那么它们就是恒等的。由此可知，别名类型与它的源类型是两个完全不同的类型。另外，一个命名类型和一个匿名类型总是不恒等的。

需要我们仔细分辨、也是我们需要重点讲述的是两个匿名类型之间的恒等判断。如果两个匿名类型的类型字面量是相同的，我们就可以说它们是恒等的。也就是说，两个恒等的匿名类型会有相同的字面结构，并且在它们之中的对应位置上的组成部分都需要具有相同的类型。当然，对于不同的数据类型来说，具体判断标准也会有所不同。

各个数据类型的恒等判断方法的规则如下。

- 对于两个数组类型，如果它们的长度一致且元素的数据类型是恒等的，那么它们就是恒等的。例如，下面两个数组类型就是恒等的：

```
[4]string  
[4]string
```

而下面两个数组类型就是不恒等的：

```
[4]string  
[3]string
```

再次强调，数组类型的长度是类型声明的一部分，也是类型的一部分。

- 对于两个切片类型，如果它们的元素的数据类型恒等，那么它们就是恒等的。例如，下面两个切片类型是恒等的：

```
go  
[]string  
[]string
```

还记得吗？切片类型的长度并不会存在于类型声明之中，并且两个同一切片类型的值的长度也不一定会相同。

- 对于两个结构体类型来说，如果它们之中的字段声明的数量是相同的，并且在对应位置上的字段具有相同的字段名称（如果有的话）和恒等的数据类型，那么这两个结构体数据类型就是恒等的。例如，有这样两个变量：

```
var a1 struct {  
    f1 sort.Interface  
    f2 int64  
}  
var a2 struct {  
    f1 sort.Interface  
    f2 int64  
}
```

从字面上看，变量a1和变量a2的类型显然是恒等的。注意，如果其中一个结构体类型声明中有匿名字段，那么在另一个结构体类型的声明中的对应位置上的字段声明也必须不包含名称。否则，它们就是不相等的，即使在对应位置上的这两个字段的类型是恒等的。现在我们稍微改动一下变量a1的类型的声明，像这样：

```
var a1 struct {
    sort.Interface
    f2 int64
}
```

变量a1的类型的声明中的第一个字段是匿名字段，而变量a2的类型的声明中的第一个字段的名称仍然是f1。这使得变量a1和变量a2的类型变得不恒等。另外，我们知道，可以在结构体类型声明中的字段声明上添加标签。如果某个字段声明是有标签的，那么这个标签也应该作为恒等判断的一个依据。我们还是对变量a1的类型的声明进行修改，如下：

```
var a1 struct {
    f1 sort.Interface `json:"list"`
    f2 int64
}
```

字段声明中的标签实际上就是一个字符串字面量，所以我们可以很容易地分辨出它们的异同。读者可以试着修改变量a2的类型的声明，并使此类型与变量a1的类型恒等。

- 对于两个指针类型，如果它们的基本类型（也就是它们指向的那个类型）恒等，那么它们就是恒等的。这条规则相当简单，读者可以试写一下两个恒等的指针类型。
- 对于两个函数类型，如果它们包含了相同数量的参数声明和结果声明，并且在对应位置上的参数或结果的类型都是恒等的，那么它们就是恒等的。例如，函数类型

```
func(name string, dept string, isManager bool) (id int, done bool)
```

和

```
func(appName string, targetOs string, authRequired bool) (id int, done bool)
```

是恒等的。注意，函数类型的恒等判断并不会以参数和结果的名称为依据，而只关心它们的数量、顺序和类型。另外，如果其中一个函数类型是可变参函数，那么另一个函数类型也应该是这样，否则它们肯定就是不恒等的。例如，函数类型

```
func(string, ...int) bool
```

和

```
func(string, []int) bool
```

就是两个不恒等的类型。

- 对于两个接口类型，如果它们拥有相同的方法集合，那么它们就是恒等的。相同的方法集合的意思是，两个接口类型所包含的方法声明的数量必须相同，并且对于一个接口类型中包含的每一个方法声明都能够在另一个接口类型中找到与它完全相等的方法声明。方法是函数的一种。因此，我们可以依据针对函数类型的恒等判断方法来判断多个接口

类型中的方法声明是否恒等。不过要注意，由于在接口类型中声明的方法都是有名称的，所以名称相同也是方法声明恒等的必要条件之一。另外，两个接口类型中的方法声明的顺序是无关紧要的。例如，接口类型

```
type Ia interface {
    Name() string
    Age() int
}
```

和

```
type Ib interface {
    Age() int
    Name() string
}
```

是恒等的。

- 对于两个字典类型，如果它们具有恒等的键类型和元素类型，那么它们就是恒等的。
- 对于两个通道类型，如果他们具有恒等的元素类型，并且方向相同，那么它们就是恒等的。我们在第7章讲解通道类型的时候再来解释它们的判等问题。

只要读者真正理解了前面这些规则，就可以比较轻松地判断出两个数据类型是否恒等。这些类型判等方法可以让我们加深对Go语言类型系统的理解，也可以作为我们编写相关代码时的理论基础。另外，在编译器报出与类型有关的错误的时候，我们也能够借此快速地定位和解决问题。

最后，值得一提的是，如果两个数据类型在不同的代码包中，即使它们满足了上述相关规则也是不恒等的。

2. 数据的可比性与有序性

以上，我们实际上是阐述了Go语言的数据类型之间的可比性。现在，我们来关注数据类型的值之间的可比性和有序性。这方面知识在我们编写Go语言代码的过程中会更有现实意义。

可比性的意思是可比较的，但是并不意味着可以比较大小，而是可以判断相等与否。我们可以判断两个具有可比性的值是否相等。而有序性才有可以比较大小的含义。如果我们可以确定两个值的先后顺序，那么也就知道了它们谁大谁小。只有具有可比性的值才可以被作为比较操作符`=`和`!=`的操作数，而只有具有有序性的值才可以被作为比较操作符`<`、`<=`、`>`和`>=`的操作数。

对于绝大多数数据类型来说，它们的值在这方面的性质都是一致的。在大多数情况下，我们也只能判断或比较两个类型相同的值的相等或大小，否则就会造成编译错误。下面是各个数据类型的值的相关特性的说明。

- 布尔类型值具有可比性。我们在3.2节中提到过，布尔值只有`true`和`false`两种可能。显然，我们很容易判断两个布尔值是否相等。但是，我们却无法比较两个布尔值的大小。
- 整数类型值具有可比性，也具有有序性。这是显而易见的，就像在基础数学中论述的那样。
- 浮点数类型值具有可比性，也具有有序性。这被定义在IEEE-754标准中。IEEE-754标准是一个针对于二进制浮点数的算术标准。

- 复数类型值具有可比性。实际上，判断两个复数类型值是否相等的结果是通过分别对它们的实部和虚部上的值进行比较而得出的。
- 字符串类型值具有可比性，也具有有序性。对于两个字符串类型值判断相等或比较大小的方法就是对它们中的每个对应位置上的字节进行判断或比较。这就相当于对多对整数类型值依次进行判断或比较，直到可以得出结果为止。例如，字符串类型值

"abc"

与

"abC"

是不相等的，且前者小于后者。

- 指针类型值具有可比性。如果两个指针类型值指向了同一个变量，或者它们都为空值nil，那么就可以判定它们是相等的。有一点需要注意，分别指向两个不同的、大小为零（zero-size）的变量的指针值可能是相等的也可能是不相等的。大小为零的变量是指无任何字段的结构体值或无任何元素的数组值。大小为零的变量可能会有相同的内存地址，因为它们在本质上很可能是没有区别的。比如，两个元素类型都为int类型的且都不包含任何元素的数组类型值之间从本质上并没有任何区别，即使与它们绑定的变量的名称并不相同。因此，分别与两个大小为零的变量对应的指针类型值很可能指向了同一个值。下面我们来看一段代码：

```
numArray := [3]int{1, 23, 456}
p1 := &numArray
p2 := &numArray
fmt.Printf("%v\n", p1 == p2)
```

变量numArray代表了一个数组类型的值，而变量p1和p2则分别代表了指向这个变量的指针。最后一行代码将在标准输出上打印出判断p1和p2是否相等的结果。当我们真正运行这段代码之后，会发现这个结果是true。

- 通道类型值具有可比性。如果两个通道类型值的元素类型和缓冲区大小都一致，那么就可以被判定为相等。另外，如果两个通道类型的变量的值都是nil，那么它们也是相等的。
- 接口类型值具有可比性。如果两个接口类型值拥有相等的动态类型和相同的动态值，那么就可以判定它们是相等的。如果我们有一个接口类型的变量，那么在这个变量中就只能存储实现了该接口类型的类型的值。我们把存储在该变量中的那个值的类型叫作该变量的动态类型，而把这个值叫作该变量的动态值。另外，如果两个接口类型的变量的值都是空值，那么它们也是相等的。请看下面的若干声明：

```
type Ic interface {
    Code() string
}
```

```
type Sc struct {
    code string
}
```

```
}
```

```
func (self Sc) Code() string {
    return self.code
}
```

结构体类型Sc是接口类型Ic的一个实现类型。如果有如下两个Ic类型的变量：

```
var ic1 Ic = Sc{code: "A"}
var ic2 Ic = Sc{code: "A"}
```

那么调用表达式

```
fmt.Printf("%v\n", ic1 == ic2)
```

在被求值之后，标准输出上就会出现true。

- 非接口类型X的值x可以与接口类型T的值t判断相等，当且仅当接口类型T具有可比性且类型X是接口类型T的实现类型。如果值t的动态类型与类型X恒等并且值t的动态值与值x相等，那么就可以说值t和值x就是相等的。我们以上一个示例为基础，并新增加一个变量声明：

```
var sc1 Sc = Sc{code: "A"}
```

变量sc1与前两个变量不同，它的类型是Sc的。由于Sc类型是Ic类型的实现类型，所以变量sc1可以与变量ic1或ic2做相等判断。又由于变量ic1和ic2的动态类型和动态值都分别与变量sc1的类型和值相等，所以调用表达式

```
fmt.Printf("%v\n", ic1 == sc1)
```

和

```
fmt.Printf("%v\n", ic2 == sc1)
```

都将在标准输出上打印出true。显然，这一条规则描述的是在特定情况下的不同数据类型的值之间的可比性。注意，除上述情况之外，我们不能对两个不同数据类型的值做相等判断。这会造成一个编译错误。

- 如果一个结构体类型中的所有字段都具有可比性，那么这个结构体类型的值就具有可比性。如果两个结构体值中的对应的字段值是相等的，那么这两个结构体类型值就是相等的。下面我们来举一个反例。首先，有这样一个结构体类型声明：

```
type Sd struct {
    ints []int
}
```

和两个变量声明：

```
sd1 := Sd{ints: []int{0, 1}}
sd2 := Sd{ints: []int{0, 1}}
```

而后，在调用表达式

```
fmt.Printf("%v\n", sd1 == sd2)
```

被编译的时候就会造成一个编译错误。原因就是，结构体类型Sc中包含了一个切片类型的字段，而切片类型的值是不具有可比性的。这不满足结构体类型值具有可比性的前提条件。

- 数组类型值具有可比性，当且仅当其元素类型的值具有可比性。如果两个数组类型值在对应位置上的值都是相等的，那么这两个数组类型值就是相等的。下面依旧是一个反例：

```
slices1 := [3][[]int{[]int{0, 1}}
slices2 := [3][[]int{[]int{0, 1}}
fmt.Printf("%v\n", slices1 == slices2)
```

变量slices1和slices2都代表了元素类型为[]int的数组类型的值。这没有问题。但是在第三行的调用表达式会造成一个编译错误。因为，这两个值的类型的元素类型都是不具有可比性的。从而这两个数组类型的值也不具有可比性。

至此，我们知道了各个数据类型是否具有的可比性和有序性。在绝大多数情况下，我们对两个不具有可比性的值做相等判断时会造成一个编译错误。同样地，我们对两个不具有有序性的值比较大小也会造成一个编译错误。

一个例外情况是，我们在判断两个具有相同接口类型的值是否相等的时候，如果它们的动态类型不具有可比性就会引发一个运行恐慌。比如，两个接口类型的变量的动态类型是切片类型或字典类型的别名类型。例如，如果有一个结构体类型是这样的：

```
type Se []int

func (self Se) Code() string {
    return ""
}
```

那么这个实现了接口类型Ic的类型的值就可以被赋给Ic类型的变量，如：

```
var ic3 Ic = Se{1, 2}
var ic4 Ic = Se{1, 2}
```

但是，当我们判断变量ic3和ic4是否相等的时候却会引发一个运行时恐慌。

又比如，两个被判断的值的类型虽然都实现了同一个接口类型，但这个实现类型却是一个包含了切片类型的字段的结构体类型。也就是说，实现了这个接口类型的结构体类型是不具有可比性的。如果我们为前面声明过的结构体类型Sd添加这样一个方法：

```
func (self Sd) Code() string {
    return ""
}
```

那么就使得Sd也成为了接口类型Ic的一个实现类型。现在我们再声明两个Ic类型的变量并用Sd类型的值对它们进行初始化：

```
var ic5 Ic = Sd{ints: []int{0, 1}}
var ic6 Ic = Sd{ints: []int{0, 1}}
```

如果我们对这两个变量进行相等判断，那么必会引发一个运行时恐慌。因为结构体类型Sd的值不具有可比性。

我们前面所说的这种例外情况，同样适用于下面这两种应用场景：以接口类型为元素类型的数组类型的值，以及以接口类型为其中某个字段的类型的结构体类型的值。下面我们分别举例说明。

首先，我们需要基于前面示例中的代码。其中，结构体类型`Se`是接口类型`Ic`的一个实现类型，但它的值已被确定不具有可比性。

对于第一种应用场景，我们现在有这样一个数组类型`[10]Ic`，并且有两个此数组类型的变量：

```
ica1 := [10]Ic{Se{10, 20}}
ica2 := [10]Ic{Se{10, 20}}
```

在判断变量`ica1`和`ica2`是否相等的时候就会引发一个运行时恐慌，而罪魁祸首就是其中的`Se`类型的值。

对于第二种应用场景，我们首先新声明一个结构体类型：

```
type Sf struct {
    ic Ic
}
```

在结构体类型`Sf`中有一个类型为`Ic`的字段`ic`。现在，我们再声明两个`Sf`类型的变量并初始化它们：

```
sf1 := Sf{ic: Se{100, 200}}
sf2 := Sf{ic: Se{100, 200}}
```

在判断变量`sf1`和`sf2`是否相等的时候也会引发一个运行时恐慌。其导火索同样是不具有可比性的`Se`类型的值。

最后，需要注意的是，切片类型、字典类型和函数类型的值是不具有可比性的。然而，作为特例，这些值是可以与空值`nil`进行判等的。

至此，我们基本上对所有的Go语言数据类型及其值的可比性和有序性都进行了全面详细的说明。在实际编码过程中，我们应该遵循上述规则，并使它们成为我们进行各种判断和比较的有力工具。如果在程序编译或运行过程中出现了相关的编译错误或运行时恐慌，我们应该对照上面的这些规则进行排查和纠错。

3.3.4 类型转换

类型转换的意思是把一个类型的值转换成为另一个类型的值。在这里，我们可以把这个值原来的类型称为源类型，而把这个值被转换后的类型称为目标类型。类型转换可以用表达式来表示。如果`T`是值`x`的目标类型，那么相应的类型转换表达式就是这样的：

`T(x)`

其中，`x`也可以是一个表达式，不过这个表达式的结果值只能有一个。

如果代表目标类型的字面量始于操作符`*`或`<-`，或者它是没有结果声明列表的函数类型，那么往往需要用圆括号“(”和“)”把它们括起来，以避免歧义的产生。例如，如果`v`是一个变量的名称，那么表达式

`*string(v)`

的含义就是先将变量v代表的值转换为string类型的值，然后再获取指向它的指针类型值。也就是说，这个表达式等同于表达式`*(string(v))`。如果我们想把变量v的值转换为指针类型`*string`的值，就需要这样编写表达式：

`(*string)(v)`

又例如，表达式

`<-chan int(v)`

的含义是先将变量v代表的值转换为chan int类型的值，然后再从此通道类型值中接收一个int类型的值。也就是说，该表达式等同于表达式`<-(chan int(v))`。如果我们想要把变量v的值转换为通道类型`<-chan int`的值，就需要这样编写表达式：

`(<-chan int)(v)`

再例如，表达式

`func()(v)`

会被Go语言理解为任何无参数声明但有一个结果声明的匿名函数。因此，如果我们想把变量v的值转换为函数类型`func()`的值，就需要这样编写表达式：

`(func()(x))`

但是，如果我们想把v的值转换成一个有结果声明的函数的类型的话，就不需要使用圆括号将目标类型括起来。例如，表达式`func() int(x)`等同于`(func() int)(x)`。这并不会产生任何歧义。

另一方面，对于常量x，如果它能够被转换为类型T的值，那么它们肯定符合下列情况中的一种。

- x可以被类型T的值代表。例如，iota可以表示一个大于等于零的整数常量。因此，它可以被uint类型的值代表。类型表达式`uint(iota)`是合法的，它的结果值会是一个uint类型的常量。又例如，常量表达式`"Go" + "lang"`的求值结果是一个字符串常量。因此，它可以被string类型的值代表。类型表达式`string("Go" + "lang")`是合法的，它的结果值就是string类型的常量`"Golang"`。
- x是一个浮点数常量，T是一个浮点数类型，并且x在（根据IEEE-754标准中描述的向偶数舍入规则）被舍入之后可以被类型T的值代表。这种情况下，表达式`T(x)`的求值结果就应该是一个被舍入之后的浮点数常量。例如，如果有一个浮点数常量0.49999998，那么类型转换表达式`float32(0.49999998)`的求值结果就是一个float32类型的常量0.5。
- 如果x是一个整数数常量，并且T是一个string类型，那么将会遵循一套规则来决定类型转换的结果。这一套规则同样适用于非常量的值。对于这种情况，我们会在后面专门予以论述。

对于非常量x，如果它能够被转换为类型T的值，那么它们肯定符合下列情况中的一种。

- 值x可以被赋给类型T的变量。例如，有如下声明：

```
type Computer interface {
    CpuType() string
}

type Laptop struct {
    cpuType string
}

func (self Laptop) CpuType() string {
    return self.cpuType
}
```

则类型转换表达式`Computer(Laptop{cpuType: "Intel Core i5"})`是合法的，且它的求值结果会是一个`Computer`类型的值。因为，类型`Laptop`是接口类型`Computer`的一个实现类型。

- 值x的类型和类型T的潜在类型是相等的。例如，我们之前提到过这样一个类型声明：

```
type MyString string
```

类型转换表达式`MyString("Mine")`是合法的。因为，类型`MyString`的潜在类型就是`string`类型。

- 值x的类型和类型T都是未命名的指针类型，并且它们的基本类型（指向的那个值的类型）的潜在类型是相等的。例如，如果有一个`string`类型的变量`str1`，那么类型转换表达式`(*MyString)(ampstr1)`就是合法的，且它的求值结果会是一个`*MyString`类型的值。
- 值x的类型和类型T都是整数类型或都是浮点数类型。例如，如果有一个`uint32`类型的变量`i32`，那么类型转换表达式`int64(i32)`就是合法的。又例如，如果有一个`float32`类型的变量`f32`，那么类型转换表达式`float64(f32)`就是合法的。
- 值x的类型和类型T都是复数类型。例如，如果有一个`complex64`类型的变量`comp64`，那么类型转换表达式`complex128(comp64)`就是合法的。
- 值x是一个整数类型值或是一个元素类型为`byte`或`rune`的切片类型值，且T是一个`string`类型。例如，类型转化表达式`string([]byte{'a'})`是合法的，且它的求值结果就是`string`类型值`"a"`。
- 值x是一个`string`类型值，且T是一个元素类型为`byte`或`rune`的切片类型。例如，类型转化表达式`[]rune("golang")`是合法的。

另外，Go语言并没有为指针类型与整数类型值之间的转换提供语言级别的支持，但是标准库代码包`unsafe`的某些程序实体却在一定约束条件下提供了这样的功能。在上一节讲指针的时候，我们已经对它们做过了介绍。

以上的内容就是针对于数据类型及其值转换的一般规则。然而，对于数值类型之间以及它们与`string`类型之间的类型转换还有一些特殊的规则。这些类型转换会改变值的表现形态，并导致一些额外的程序运行时间的消耗。相对应地，所有其他类型转换只会改变值的类型而不会改变值的表现形态。下面我们就来说明这些特殊的类型转换规则。

1. 数值类型之间的转换

首先，我们强调一下数值类型值与数值常量的区别。我们在前面讲常量的时候说过：可以通过常量声明或者数据类型转换把一个无类型的常量类型化。举个例子，数值常量1024是无类型的，但是当我们把它赋给一个int类型的变量：

```
var number int = 1024
```

或者把它向int类型转化：

```
int(1024)
```

之后，它就变成了一个数值类型值，具体类型为int。

对于非常量的数值类型值，有如下规则。


- 当我们把一个整数类型值从需要较少二进制位表示的整数类型转换到需要较多二进制位表示的整数类型（比如从int8类型转换到int16类型）的时候：如果这个整数类型值是有符号的，那么该符号位上的（最左边的）那个二进制值将作为扩展项填充在转换过程中新增的那些二进制位上，否则将会把0作为扩展项进行填充。注意，这种扩展方式是针对整数类型值的补码而言的。例如，int16类型值-32767的十六进制表示是0xffff。它的补码是0x8001。此补码最左边的二进制位上的二进制值是1。如果我们要把这个int16类型值转换为int32类型值，就需要用最左边的这个值1填充在高位一侧新增的那16个二进制位上。也就是说，类型转换之后的补码是0xffff8001。我们在这个补码之上再求补码以得其原码，即0x80007fff。此原码表示的就是十进制数-32767，与类型转换前的那个数组相等。
- 当我们把一个整数类型值从需要较多二进制位表示的整数类型转换到需要较少二进制位表示的整数类型（比如从int16类型转换到int8类型）的时候，需要把多余的若干个较高位置的二进制值截掉，而只保留与目标类型所需二进制位数相当的若干个较低位置的二进制值。我们仍以int16类型值-32767为例。如果我们要把它转换为一个int8类型值，就需要对其补码0x8001截取较低8位的二进制值，得到0x01。由于此值的最左边的二进制位上是0，所以它本身就是类型转换后的数值的原码，即十进制数1。注意，对于非常量的整数类型值来说，这种类型转换总会得到一个有效的数值。但是，对于整数常量来说，这样的类型转换就会造成一个编译错误。例如，类型转换表达式int8(-32767)会使编译器报错，因为整数常量-32767超出了int8类型所能表示的数值范围。这个区别很重要，请读者牢记。
- 当把一个浮点数类型值向整数类型值进行转换的时候，该浮点数类型值的小数部分将被抹去（截断至零）。这一条规则很好理解。例如，如果有一个float32类型的变量f32且其值为-32767.345，那么类型表达式int32(f32)的求值结果就是-32767。但要注意，如果浮点数类型值在被抹去小数位之后超出了目标整数类型的表示范围，那么该值还会被截短。这个截短操作将遵循上一条规则。也就是说，在类型表达式int8(f32)被求值的过程中会首先把float32类型值-32767.345的小数部分去掉，然后再将其中较高的24位的二进制值截掉，最终得到结果1。

- 当把一个整数或浮点数转换为一个浮点类型的值或者把一个复数转换为一个复数类型的值的时候，该值将会被依据目标类型的精度进行舍入操作。例如，在float32类型的变量x中存储的值可能会超出IEEE-754标准中规定的32位（二进制值代表的）浮点数的精度。但是，类型转换表达式float32(x)的求值结果一定会是x的值向32位浮点数的精度转化之后的值。相似地，算术表达式x + 0.1的结果值可能会超出32位浮点数的精度，但是类型转换表达式float32(x + 0.1)的求值结果却不会这样。

在非常量的浮点数类型值或复数类型值的类型转换中，当目标类型的精度不能够满足被转换的值的需要的时候，虽然转换会成功，但其结果将是不确定的，这依赖于不同平台的Go语言的具体实现。

2. 与string类型相关的转换

我们现在来专门论述怎样从其他值转换到string类型值，以及怎样从string类型值转换到其他值。这将包括我们在前面讲类型转换通用规则时未详细解释的那一项。下面是具体规则。

- 当把一个有符号整数值或无符号整数值向字符串类型转换的时候，将会产生出一个字符串类型值。注意，被转换的整数值应该是一个有效的Unicode代码点的代表。这种情况下，在作为结果的字符串类型值中的就是与那个Unicode代码点对应的字符。当然，在底层，这个字符串类型值是由该Unicode代码点的UTF-8编码值表示的。如果被转换的整数值不能代表一个有效的Unicode代码点，那么转换结果将会是"\ufffd"，即Unicode字符“”。例如，类型转换表达式


```
string(0x4e2d)
```

的求值结果是"中"，其UTF-8编码为\xe4\xb8\xad。又例如，类型转换表达式

```
string('国')
```

的结果值是"国"，其UTF-8编码为\xe5\x9b\xbd。作为反例，类型表达式

```
string(-1)
```

的结果值就是""，因为整数值-1并不能代表一个有效的Unicode代码点。另外，在这种类型转换中，如果目标类型是一个string类型的别名类型，那么我们可以将它视同为string类型。例如，若有string类型的别名类型MyString，则类型转换表达式MyString(0x4e2d)的求值结果与string(0x4e2d)的相同。

- 当把一个元素类型为byte的切片类型值向字符串类型转换时，将会产生出一个字符串类型值。这个字符串类型值实际上就是由被转换的切片类型值中的每个字节类型值依次组合而成的。如果切片类型值为nil，那么类型转换的结果将会是""。例如，类型转换表达式

```
string([]byte{'g', '\x6f', '\x6c', '\x61', 'n', 'g'})
```

的结果值是"golang"。解释一下，由于使用"\x"为前导并后跟两位十六进制数可以表示宽度为一个字节的值，因此一个字节类型的值也就可以由这种方法表示。'\x6f'、'\x6c'、'\x61'分别代表了字符'o'、'l'和'a'。并且，在前缀"\x"后面的分别是与这三个字符相对应的ASCII编码值的十六进制表示形式。另外，如果源类型是一个[]byte类型的别名类

型，那么我们可以将它视同为[]byte类型。

- 当把一个元素类型为rune的切片类型值向字符串类型转换时，将会产生出一个字符串类型值。这个字符串类型值实际上就是依次串联每个rune类型值后的结果。如果切片类型值为nil，那么类型转换的结果将会是""。例如，类型转换表达式

```
string([]rune{0x4e2D, 0x56fd}))
```

的结果值是"中国"。其中，0x4e2D是字符'中'对应的Unicode代码点的十六进制字面量，而0x56fd则是字符'国'对应的Unicode代码点的十六进制字面量。同样地，如果源类型是一个[]rune类型的别名类型，那么我们可以将它视同为[]rune类型。

- 当把一个字符串类型值向[]byte类型转换时，其结果将会是把该字符串类型值按字节拆分后的结果。对于""来说，转换后的结果一定是[]byte类型的空值nil。这种类型转换与前面第2项规则描述的类型转换互为逆转换。例如，类型表达式

```
[]byte("hello")
```

的结果值是[]byte{104 101 108 108 111}。在这个[]byte类型值中的每个元素都是对应字符的ASCII编码值的十进制表示形式。在这种类型转换中，如果目标类型是一个[]byte类型的别名类型，那么我们可以将它视同为[]byte类型。

- 当把一个字符串类型值向[]rune类型转换时，其结果将会是把该字符串类型值按字符拆分后的结果。对于""来说，转换后的结果一定是[]rune类型的空值nil。这种类型转换与前面第3项规则描述的类型转换互为逆转换。例如，类型表达式

```
[]rune("中国")
```

的结果值是[]rune{20013, 22269}。在这个[]rune类型值中的每个元素都是对应字符的Unicode代码点的十进制表示形式。在这种类型转换中，如果目标类型是一个[]rune类型的别名类型，那么我们可以将它视同为[]rune类型。

其实，对于上面这5项规则，我们并不需要逐个记忆，宏观地去看待它们会感觉非常好理解。

一个字符串类型值可以被拆解为多个字节或多个字符。字符串类型以及rune类型（可以看成代表字符的类型）的值都会被用Unicode编码规范中的UTF-8编码格式编码成一个字节序列并进行储存。还记得吗？UTF-8这种编码方式会把一个字符编码为一个或多个字节。因此，对于同一个字符串类型值来说，与它对应的字节序列和字符序列中的元素并不一定是一一对应的。也就是说，字节序列中的单个字节并不一定能代表一个完整的字符。我们仍以字符串类型值"中国"为例。与它对应的[]byte类型值是：

```
[]byte{228, 184, 173, 229, 155, 189}
```

这个字节序列中的前三个元素代表了字符'中'的UTF-8编码值，而后三个元素则代表了字符'国'的UTF-8编码值。另一个方面，与"中国"对应的[]rune类型值是：

```
[]rune{20013, 22269}
```

这个字符序列中的第一个元素代表了字符'中'的Unicode代码点，而第二个元素则代表了字符'国'

'的Unicode代码点。

记住了吗？在正常情况下，与字符串类型值对应的字节序列中的每一个元素都代表了某个字符的全部或部分UTF-8编码值，而与字符串类型值对应的字符序列中的每一个元素则都代表了某个字符的Unicode代码点。另外，对于每一个ASCII编码可表示的字符来说，它的Unicode代码点和UTF-8编码值与其ASCII编码值都分别是一致的，且它们都可以由一个字节类型值代表。例如上面提及的字符'o'、'l'和'a'。正因为如此，对于一个只包含了ASCII编码可表示字符的字符串类型值来说，与它对应的字节序列和字符序列中的元素值必定也是一一对应的。

再来说byte类型值和rune类型值的表示方式。虽然我们在上一节讲基本数据类型的时候已经对它们进行了详细的说明，但是在这里还是再稍稍重申一下。

byte类型值和rune类型值都属于整数值的一种。所有整数值都可以由十进制字面量、八进制字面量和十六进制字面量来代表。我们在讲rune类型的时候说过，此类型的值有5种表示方法。实际上，任何整数类型（包括byte类型）的值都可以由这5种方法表示。反过来讲，我们可以把使用任意一种方式表示的rune字面量赋给任何整数类型的变量，只要该rune字面量对应的Unicode代码点不超出那个整数类型的表示范围。例如，int16类型的变量nation可以这样被赋值：

```
var nation int16 = '国' // '国' == 0x56fd == 22269
```

又例如，与字符串类型值"golang"相对应的切片类型值可以是：

```
[]byte{'g', '\x6f', 0x6c, '\u0061', '\156', '\U00000067'}
```

3. 别名类型值之间的转换

在本小节的最后，我们稍微解释一下别名类型值之间的关系以及类型转换。我们在前面提到过，类型MyString是string类型的别名类型。并且，如果把一个整数值分别转换为这两个类型的值，将会得到相同的结果。实际上，如果忽略掉它们不同的名称，类型MyString和它的源类型string是相同的。无论从类型值的内存布局还是从数据操作的角度看都是这样。例如，我们可以把一个字符串字面量赋给MyString类型的变量：

```
var ms MyString = "中国"
```

又例如，我们也在MyString类型的值之上应用切片操作：

```
ms[1]
```

在某个数据类型和它的别名类型之间以及同一个数据类型的多个别名类型之间的类型转换是合法的。并且，在这种类型转换的过程中并不会创造出新的值，而仅仅是变换了一下那个已存在的值的所属类型。

类型转换是我们在编程过程中经常会用到的语言级功能之一。在很多应用场景下，它会与类型断言表达式一起使用。了解类型转换的操作方法和实现细节会使我们对不同数据类型之间的异同和关系有更加深刻的认识。通过对本小节的阅读和相关知识的学习，你真正理解类型转换了吗？

3.3.5 内建函数

在前面的章节中，我们提到过一些Go语言的内建函数。在本小节，我们会将与它们相关的

知识进行汇总，并补齐之前没有讲过的一些内建函数的相关内容。

所谓内建函数，就是Go语言内部预定义的函数。调用它们的方式与调用普通函数并无差异，并且在使用它们之前也不需要导入任何代码包。但是，我们并不能把内建函数当作值来使用，因为它们并不像普通函数那样有隶属的Go语言数据类型。下面，我们就来对它们进行逐一讲解。

1. close函数

内建函数close只接受通道类型的值（以下简称为通道）作为参数。示例代码如下：

```
ch := make(chan int, 1)
close(ch)
```

调用这个close函数之后，会使作为参数的通道无法再接受任何元素值。若试图关闭一个仅能接受元素值的通道，则会造成一个编译错误。在通道被关闭之后，再向它发送元素值或者试图再次关闭它的时候，都会引发一个运行时恐慌。此外，试图关闭一个为nil的通道值也会引发一个运行时恐慌。

我们试图调用close函数关闭一个通道，并不会影响到在此调用之前已经发送的那些元素值。它们会被正常接收（如果存在接收操作的话）。但是，在此调用之后，所有的接收操作都会立即返回一个该通道的元素类型的零值。我们在讲接收操作符的时候说过，接收表达式可以有两个返回值，其中的第二个返回值代表了通道是否已经被关闭。这样就可以有效避免接收到的值为该元素类型的零值的时候的歧义。

2. len函数与cap函数

相信读者已经对len函数和cap函数都比较熟悉了。这两个函数都只接受一个参数，并会返回一个int类型的结果。现在我们来总结一下它们的用法。

在表3-8和表3-9中，我们分别列举了这两个函数的所有应用场景及其作用。

表3-8 len函数的使用方法

参数类型	结 果	备 注
string	string类型值的字节长度	(无)
[n]T 或 *[n]T	数组类型值的长度。实际上，它等于n	n代表了数组类型的长度。T代表了数组类型的元素类型
[]T	切片类型值的长度	T代表了切片类型的元素类型
map[K]T	字典类型值的长度，也就是其中已包含的键的数量	K代表了字典类型的键类型。T代表了字典类型的元素类型
chan T	通道类型值当前包含的元素的数量	T代表了通道类型的元素类型

表3-9 cap函数的使用方法

参数类型	结 果	备 注
[n]T 或 *[n]T	数组类型值的长度，与n相等	n代表了数组类型的长度。T代表了数组类型的元素类型
[]T	切片类型值的容量	T代表了切片类型的元素类型
chan T	通道类型值的容量	T代表了通道类型的元素类型

对于一个切片类型值来说，它的长度和容量的关系如下：

```
0 <= len(s) <= cap(s)
```

我在讲切片类型的时候说过，一个切片值的容量就是它拥有的那个底层数组的长度。这个底层数组的长度必定不会小于该切片值的长度。

另外，需要注意的是，为nil的切片类型值、字典类型和通道类型值的长度都是0。值为nil的切片类型值和通道类型值的容量也都是0。如果读者还记得这3种数据类型的零值都是nil的话，就应该能够领会上面这两条规则的原因了。

如果s是一个string类型的常量，那么表达式len(s)和cap(s)也都等同于常量。这意味着，len(s)所代表的值在编译期间就会被计算出来。类似的，如果s是一个表达式，且其类型是数组类型或指向数组类型的指针类型，那么只要该表达式中不包含通道接收操作和函数调用操作，它就不会被求值。因为s的类型中已经包含了它的长度信息。在对表达式len(s)和cap(s)进行求值的时候并不要求得s的结果值而只需要从s的类型中取得其长度即可。因此，在这种情况下，这两个表达式也会等同于常量。

3. new函数和make函数

这两个函数我们在上一节已经详细说明过。因此我们就不在此重复描述了。读者可以参看3.2.9节。

4. append函数和copy函数

内建函数append和copy都被用于辅助在切片类型值之上的操作。我们在讲切片类型的时候已经解释过它们的用法和作用。对这两个内建函数还不了解的读者请参见3.2.3节。

5. delete函数

内建函数delete专用于删除一个字典类型值中的某个键值对。它接受两个参数，第一个参数是作为目标的字典类型值，而第二个参数则是能够代表要删除的那个键值对的键。调用表达式会像这样：

```
delete(m, k)
```

这里有两点需要注意。

- 第二个参数k与m的键的类型之间必须满足赋值规则。
- 当m的值是nil或者k所代表的键值对并不存在于m中的时候，delete(m, k)不会做任何操作。

也就是说，在没有可删除的目标的时候，删除操作将被忽略。这种删除失败不会被反馈。

6. complex函数、real函数和imag函数

这3个内建函数都是专用于操作复数类型值的。complex函数被用于根据浮点数类型的实部和虚部来构造复数类型值。例如：

```
var cplx128 complex128 = complex(2, -2)
```

内建函数real和imag则分别被用于从一个复数类型值中抽取浮点数类型的实部部分和浮点数类型的虚部部分。例如：

```
var im64 = imag(cplx128)
var r64 = real(cplx128)
```

对于这3个内建函数，它们各自的参数类型和结果类型都是有关联的。对于`complex`函数来说，两个参数的类型必须是同一种浮点数类型，并且其结果类型必须与参数类型对应。比如，如果两个参数的类型是`float32`类型的，那么结果类型就是`complex64`类型的。而如果两个参数的类型是`float64`类型的，那么结果类型则必是`complex128`类型的。这应该很好理解，两个32位的值理应需要64位的空间来表示，而两个64位的值也需要128位的空间来表示。相应地，`real`和`imag`函数中的参数类型和结果类型的对应关系恰恰与之相反。因此，就有了这样的一个恒等式：

```
z == complex(real(z), imag(z))
```

其中，`z`是一个复数类型的变量。

注意，如果`complex`函数的两个参数都没有显式的类型，那么该函数的结果的类型将会是`complex128`类型的。此外，如果`complex`函数的参数都是常量，那么它的结果值也必是常量，这对于`real`函数和`imag`函数也是一样。

7. `panic`函数和`recover`函数

内建函数`panic`和`recover`分别被用于报告和处理运行时恐慌。

函数`panic`只接受一个参数。这个参数可以是任意类型的值。我们想要生成和报告一个运行时恐慌的时候可以直接调用这个函数并传递给它一个用以描述恐慌细节的值。按照惯例，`panic`函数的实际参数的类型常常是接口类型`error`的某个实现类型。不论怎样，`panic`函数的参数都应该足以表示恐慌发生时的异常情况。

函数`recover`不接受任何参数，但是返回一个`interface{}`类型的结果值。我们知道，`interface{}`就是空接口。所有的数据类型都是它的实现类型。这意味着，`recover`函数的结果值可能是任何类型的。实际上，这是与`panic`函数的那个唯一参数相对应的。它们都是`interface{}`类型的。如果运行时恐慌的报告是通过调用`panic`函数来进行的话，那么之后调用`recover`函数所得到的结果值就应该是先前`panic`函数在被调用时接受的那个参数值。不过，`recover`函数的结果值也有可能是`nil`。如果是`nil`，那么肯定属于下面情况中的一种。

- ❑ 传递给`panic`函数的参数值就是`nil`。
- ❑ 运行时恐慌根本就没有发生。狭义地讲，`panic`函数没有被调用。
- ❑ 函数`recover`并没有在`defer`语句中被调用。

请放心，我们在任何情况下任何位置上调用`recover`函数都不会产生任何副作用。不过，如果我们不用它来处理运行时恐慌，那么对它的调用也就没有任何意义了。

不论怎样，`panic`函数和`recover`函数之间肯定存在着某种联系的，不仅仅是参数值和结果值相对应那么简单。那么它们之间的联系到底是什么呢？我们应该怎样搭配使用这两个函数呢？又怎样让它们在异常处理流程中发挥应有的作用呢？在这里，我先不回答这些问题。在下一节中，我会对异常报告和处理的更多细节进行专门的介绍。

8. `print`函数和`println`函数

这两个内建函数的作用显而易见，都是把参数值在标准输出上打印出来且不返回任何结果值。它们都属于可变参函数。不过它们略有不同。

函数`print`的作用是依次（即从左到右）打印出传递给它的参数值，每个参数值对应的打印内容都由它们的具体实现决定。而函数`println`函数则会在`print`函数打印的内容的基础上再在每个参数之间加入空格“ ”，并在最后加入换行符。示例如下：

```
print("A", 12.4, 'R', "C")
println("A", 12.4, 'R', "C")
```

上面两个调用表达式被求值之后，在标准输出上会出现这样的内容：

```
A+1.240000e+00182CA +1.240000e+001 82 C
```

读者可以试着识别一下，哪些内容是`print`函数打印出来的，而哪些内容是`println`函数打印出来的。

对于这两个函数，有以下几点需要注意。

- ❑ 它们接受的参数只能是有限的数据类型的值。并且，在这些受支持的数据类型当中，大部分都是Go语言的基础数据类型。
- ❑ 这两个函数针对于每种受支持的数据类型的打印格式都是固定的，我们无法自定义。
- ❑ Go语言并不保证会在以后的版本中一直保留这两个函数。

鉴于以上这几点，建议尽量不要在我们的程序使用这两个函数，尤其是用于生产环境的程序。我们应该使用标准库代码包`fmt`中的函数`Print`和`Println`来替代它们。

至此，Go语言中预定义的内建函数就全部介绍完了。在实际编程过程中，它们中的绝大部分都会被经常用到。甚至可以说，有的内建函数是至关重要的，如`new`和`make`。我们应该在使用它们之前确保已经真正地理解了它们，以求对它们的合理运用。

3.4 本章小结

本章先介绍了编写Go语言程序必须要理解的基本词法，主要涉及标识符、关键字、字面量、类型、操作符和表达式。

而后，我们逐一地讲解了Go语言中的绝大多数预定义数据类型的展现和使用的方法。同时，我们对于怎样定义自己的数据类型的问题也进行了非常详尽的说明。除此之外，我们还深入论述了数据初始化方法、赋值方式、可比性与有序性以及类型转换等一些相对高级的主题。

对于上述这些内容，我都附上了大量的示例，并穿插了对相关的使用惯例和最佳实践的说明。我希望大家在了解Go语言的编程基础的同时，也能够真正理解这些附加内容。这对于我们编写出更加优秀的Go语言程序是很有好处的。

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语句可以使我们的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。`++`的作用是把它的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`--`的作用则是把它的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 索引值	0x47	0x6f	0x6c	0x61	0x6e	0x67	0xe7	0x88	0xb1	0xe5	0xa5	0xbd	0xe8	0x80	0x85
	0	1	2	3	4	5	6			9			12		
[]rune类型值 索引值	'G'	'o'	'l'	'a'	'n'	'g'	'爱'			'好'			'者'		
	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生出一个值。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。

现在我们来再考虑一个问题，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，并且附属于此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，不用担心，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知。

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{}]`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{}]`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，但是这终究是一种开销。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。

现在我们来查看集合运算。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`。

现在我们来查看`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 $O(N \cdot \log N)$ 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 $O(\log N)$ 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。

从软件工程的角度看，软件开发离不开软件测试。这两个环节永远是相辅相成的。从程序设计的角度来说，程序测试也与程序开发紧密的联系在一起。测试不论在软件工程理论还是在程序设计方法论中，都拥有着举足轻重的地位。测试是程序质量的有力保障手段，也是程序维护者手边最好的辅助工具。请想象一下，我们在优化一段代码之后，总是担心它在功能上是否依然如初、它的性能是否已经有所提高。在我们更新或增加了程序的功能之后，我们如何在正式运行这个程序之前就能最大限度地保证它们是正确的，以及怎样知道对程序的改动是否对已有的程序产生了影响。单元测试（或者说程序员测试方法）就是为解决此类问题而诞生的。

在本章，我们会先就Go语言的程序测试的编写方法和高级运行方法展开讨论。之后，作为一个知识补充，我们会简要地说明一下怎样为Go语言程序添加文档。虽然后者所占用的篇幅较小，但却不能说它不重要。如果你学过软件工程那门课程的话就应该知道，程序加文档才能被称为软件。即便我们编写的只是一小段程序，也应该积极地为它添加文档。这是养成良好习惯的重要一环。

5.1 程序测试

现代计算机编程语言基本上都有用于进行单元测试的框架，比如，Java语言的单元测试框架JUnit、C++语言的单元测试框架CppUnit，或是Python语言的单元测试框架PyUnit。这些单元测试框架都是某个或某些使用这些编程语言的程序设计者开发出来的。换句话说，它们都属于第三方的程序框架。

Go语言作为新兴的和先进的现代计算机编程语言，为程序测试提供了大量的开箱即用的工具。在第2章中，我们提及过`go test`命令（附属于本书的Go命令教程对它进行了详尽的介绍）。它不仅仅可以对代码包进行测试，还可以对个别源码文件进行测试，只要存在针对这些测试的测试源码文件。除此之外，Go语言还在标准库中提供了一个专门用于测试的代码包`testing`。这个代码包提供了我们编写测试源码文件所需要的一切。

本节会就Go语言的程序测试的编写方法和高级运行方法展开讨论。

5.1.1 功能测试

测试源码文件总应该与被它测试的源码文件处于同一个代码包内。我们在编写测试源码文件

的时候，总是会用到标准库代码包testing中的API。testing包为Go语言的代码包提供自动化测试支持。它的目标是与go test命令协同使用，以自动执行目标代码包中的任何测试函数。

1. 编写功能测试函数

在测试源码文件中，针对其他源码文件中的程序实体的功能测试程序总是以函数为单位的。被用于测试程序实体功能的函数的名称和签名形如：

```
func TestXxx (t *testing.T)
```

其中，Xxx应该是大写字母开头的若干字母和数字的组合。通常情况下，我们会把Xxx替换成被测试的程序实体的名称。

我们可以利用*testing.T类型的参数t上的一些方法对功能测试的过程进行记录和控制。我们使用t的值上的方法记录的信息会在测试结束之后（不论成败）一并打印到标准输出上。好了，我们现在就开始了解怎样使用这些API。

2. 常规记录

参数t上的Log和Logf方法一般用于记录一些常规信息，以展现测试程序的运行过程以及被测试程序实体的实时状态。t.Log方法与fmt.Println函数的使用方法是类似的，而t.Logf方法则与fmt.Printf函数的使用方法是类似的。例如，调用语句

```
t.Log("Test tcp listener & sender (serverAddr=", "127.0.0.1:8080", ")...")
```

和

```
t.Logf("Test tcp listener & sender (serverAddr= %s )...", "127.0.0.1:8080")
```

执行之后，都会使得在测试结束之后有这样一行内容出现在标准输出上：

```
tcp_test.go:26: Test tcp listener & sender (serverAddr= 127.0.0.1:8080 )...
```

其中，作为前缀的tcp_test.go:26:代表了调用语句所在的测试源码文件的名称以及出现的行号。

3. 错误记录

参数t上的Error和Errorf方法被用于错误信息。当被测试的程序实体的状态不正确的时候，我们就应该调用t.Error或t.Errorf方法，及时对当前的错误状态进行记录。例如：

```
actLen := len(s)
if actLen != explen {
    t.Errorf("Error: The length of slice should be %d but %d.\n", explen, actLen)
}
```

当s的值的长度与预期的长度不一致的时候，我们就可以调用t.Errorf方法，来记录这种错误状态。

调用t.Error方法相当于先后对t.Log和t.Fail方法进行调用，而调用t.Errorf方法则相当于先后对t.Logf和t.Fail方法进行调用。我们会在后面再次提到t.Fail方法。

4. 致命错误记录

参数t上的Fatal和Fatalf方法被用于记录致命的被程序实体的状态错误。所谓致命错误是指

使得测试无法继续进行的错误。例如：

```
if listener == nil {  
    t.Fatalf("Listener startup failing! (addr=%s)!\n", serverAddr)  
}
```

当代表了网络监听器的listener没有被启动成功的时候，我们就无法执行后续的与数据收发相关的测试了。因此，这里适合调用t.Fatalf以报告致命错误。

调用t.Fatal方法相当于先后对t.Log和t.FailNow方法进行调用，而调用t.Fatalf方法则相当于先后对t.Logf和t.FailNow方法进行调用。我们会在后面再次提到t.FailNow方法。

5. 失败标记

如果我们需要标记当前测试函数中的测试是失败的，那么就需要用到t.Fail方法。对t.Fail方法的调用不会终止当前测试函数的执行。但是，此函数的测试结果已经被标记为失败了。

6. 立即失败标记

方法t.FailNow与t.Fail不同的地方是，它在被调用时会立即终止当前测试函数的执行。这会使得当前的测试运行程序会转而去执行其他的测试函数。需要注意的是，我们只能在运行测试函数的Goroutine中调用t.FailNow方法，而不能在我们在测试代码创建出的Goroutine中调用它。不过，我们在其他的Goroutine中调用t.FailNow方法也不会造成什么错误，只是它不会产生任何效果而已。我们会在第7章详细讲解Goroutine的概念，现在可以暂且把它理解为可被并发执行的代码块的载体。

7. 失败判断

在调用t.Failed方法之后，我们会获得一个bool类型的结果值。它代表当前的测试函数中的测试是否已被标记为失败。

8. 忽略测试

调用t.SkipNow方法就意味着标记当前测试函数为已经被忽略的并且立即终止该函数的执行。当前的测试运行程序会转而去执行其他测试函数。与t.FailNow方法相同，t.SkipNow方法也只能在运行测试函数的Goroutine中被调用。

调用t.Skip方法相当于先后对t.Log和t.SkipNow方法进行调用，而调用t.Skipf方法则相当于先后对t.Logf和t.SkipNow方法进行调用。

方法t.Skipped的结果值会告知我们当前的测试是否已被忽略。

9. 并行运行

方法t.Parallel的调用会使当前的测试函数被标记为可并行运行的。这会使测试运行程序可以并发地执行它以及其他可并行运行的测试函数。我们会在后面看到这个方法的作用。

上面讲述的这些方法都会很容易在功能测试函数中使用。还记得我们在上一章的实战环节中编写的结构体类型HashSet、myKeys和myOrderedMap吗？读者何不现在就试着为它们编写测试呢？

10. 功能测试的运行

Go语言提供了专用于程序测试命令——go test命令。现在我们简单地说明一下它的基本用法。我们以goc2p项目的代码包cnet/ctcp为例。这个代码包中仅包含一个名为tcp_test.go的测试源

码文件。这个测试源码中有两个测试函数。一个是名为TestPrimeFuncs的功能测试函数，一个是名为BenchmarkPrimeFuncs的基准测试函数。我们下面使用go test命令运行cnet/ctcp包中的测试。示例如下：

```
hc@ubt:~/golang/goc2p/src$ go test cnet/ctcp
ok      cnet/ctcp      2.017s
```

可以看到，测试通过了，且总耗时约2秒。

如果我们只想运行代码包中部分测试的话，有两种方式可以选择。第一种方式就是在go test命令后面以测试源码文件及其测试的源码文件为参数，而不是代码包，像这样：

```
go test envir_test.go envir.go
```

而第二种方式是使用标记-run。-run标记的值应为一个正则表达式。名称与此正则表达式匹配的功能测试函数，才会在当次的测试运行过程中被执行。例如：

```
hc@ubt:~/golang/goc2p/src$ go test -run=Prime cnet/ctcp
ok      cnet/ctcp      2.016s
```

在这个示例中，该代码包的测试源码文件envir_test.go中的功能测试函数TestPrimeFuncs会被执行。但是当我们把正则表达式换为“Prima”后，就不会有任何功能测试函数被执行了。因为，在cnet/ctcp包并没有名称与之匹配的功能测试函数。示例如下：

```
hc@ubt:~/golang/goc2p/src$ go test -run=Prima cnet/ctcp
ok      cnet/ctcp      0.012s
```

从之前的示例中我们可以获知，在正常执行功能测试函数TestPrimeFuncs的情况下需要2秒多的时间。但在这个示例中，测试的总耗时是0.012s。我们也由此可以推断功能测试函数TestPrimeFuncs并没有被执行。不过，在查看测试结果的时候，光凭我们的经验推断测试过程是远远不够的，也是不严谨的。我们需要的是确切的测试过程信息。

我们已经知道，可以通过方法t.Log和t.Logf来记录测试过程。但是，在默认情况下，使用它们打印的信息不会被显示出来的。为此，我们需要标记-v。-v可被译为冗长模式。它的作用是，在测试运行结束后打印出所有在测试运行过程中被记录的日志。当然，我们也可以使用标准库的代码包fmt中的一些函数来打印测试日志。但是用这种方式记录的日志会在测试运行过程中被显示出来，不论我们是否使用了-v标记。这会使打印出来的信息在格式和内容上都比较混乱。所以，强烈建议在测试源码文件中使用方法参数t的值上的方法来记录日志。下面我们来看示例：

```
hc@ubt:~/golang/goc2p/src$ go test -v cnet/ctcp pkgtool
=== RUN TestPrimeFuncs
--- PASS: TestPrimeFuncs (2.00 seconds)
    tcp_test.go:24: Test tcp listener & sender (serverAddr=127.0.0.1:8080)...
PASS
ok      cnet/ctcp      2.012s
=== RUN TestGetGoroot
--- PASS: TestGetGoroot (0.00 seconds)
=== RUN TestGetAllGopath
--- PASS: TestGetAllGopath (0.00 seconds)
=== RUN TestGetSrcDirs
```

```

--- PASS: TestGetSrcDirs (0.00 seconds)
=== RUN TestAppendIfAbsent
--- PASS: TestAppendIfAbsent (0.00 seconds)
PASS
ok      pkgtool 0.010s

```

我们看到，在这次的打印信息中显示了非常详细的测试日志。这些信息中的第1行至第5行是关于代码包cnet/ctcp的。我们来稍加解释一下。第1行是开始运行测试函数TestPrimeFuncs的标识。第2行表示测试函数TestPrimeFuncs中的测试已经通过，且测试耗时为2秒。第3行是在这个测试函数中的测试日志信息。如果我们记录了多个日志，那么就会在这里出现同样行数的信息。第4行表明代码包cnet/ctcp中的所有测试均已通过。第5行表示代码包cnet/ctcp的测试运行总耗时为2.012秒。读者可以试着解释一下剩下的测试日志。它们是与测试代码包pkgtool有关的日志内容。

11. 关于测试运行时间

有这样一种测试场景，我们在一个测试函数中包含一段了耗时较长的代码，并且我们需要严格规定执行这个测试函数的耗时上限。可被用于go test命令的标记-timeout能够满足这种需求。如果我们在执行go test命令时加入了这个标记，且在达到其值所代表的时间上限时测试还未结束，那么就会引发一个运行时恐慌。-timeout标记的值是类型time.Duration可接受的时间表示法。这种表示时间的方法的主要思想是使用包含一个带符号的十进制数字和一个代表时间单位的字符串后缀的组合来代表时间。并且，多个这样的字符串组合被可以联合起来使用。但是它们需要以时间单位从大到小为顺序从左到右的排列起来。例如，1h20s代表1小时20秒，2h45m代表2小时45分钟，200ms代表200毫秒，等等。有效的时间单位请见表5-1。

表5-1 有效的时间单位

时间单位	字符串表示法
纳秒	"ns"
微秒	"us" 或 "μs"
毫秒	"ms"
秒	"s"
分钟	"m"
小时	"h"

我们已经知道，运行代码包cnet/ctcp中的功能测试函数的执行耗时大约在2秒左右。我们将通过-timeout标记将测试耗时上限设置为100毫秒，并运行测试。示例如下：

```

hc@ubt:~/golang/goc2p/src$ go test -timeout 100ms cnet/ctcp
panic: test timed out

.....

FAIL    cnet/ctcp    0.112s

```

由于篇幅的关系，我们忽略了一些调用栈信息。但仅从上面显示的信息中，我们就可以获知

测试失败的原因正是由于发生了一个运行时恐慌。并且这个运行时恐慌中包含的信息是“test timed out”，即测试超时。这就是-timeout标记所起到的作用。

如果我们并不想强制性地设置测试函数执行耗时的上限，而只是想让测试尽快结束，那么可以试试标记-short。使用-short标记意味着告诉之后要运行的测试尽量缩短它们的运行时间。至于怎样缩短运行时间，就是由测试函数自己去定义和实现了。代码包testing中有一个名为Short的函数。这个函数在被调用后会返回一个类型bool的值。这个值表明了我们是否在执行go test命令的时候加入了-short标记。如果这个函数返回的bool值为true，那么我们就可以根据具体情况，去剪裁测试代码从而缩短测试运行时间了。例如，我们可以在一个功能测试函数中写一段类似这样的代码：

```
if testing.Short() {
    multiSend(serverAddr, "SenderT", 1, (2 * time.Second), showLog)
} else {
    multiSend(serverAddr, "SenderT1", 2, (2 * time.Second), showLog)
    multiSend(serverAddr, "SenderT2", 1, (2 * time.Second), showLog)
}
```

这段代码摘自测试源码文件tcp_test.go中的测试函数TestPrimeFuncs，但也做了些修改。我们把关注点放在函数multiSend上。在上面的代码中，我们根据函数testing.Short()的返回的结果值做了不同的策略。如果结果值为true，则只调用一次multiSend函数，否则调用两次。这是一种很简单的缩短测试运行时间的方法。当然，如果你有需要，可以制定更复杂、更有效的策略。在知道了应对-short标记的代码编写方法之后，让我们来看看上述代码示例的原型的真实测试结果：

```
hc@ubt:~/golang/goc2p/src$ go test cnet/ctcp
ok      cnet/ctcp      2.169s
hc@ubt:~/golang/goc2p/src$ go test -short cnet/ctcp
ok      cnet/ctcp      1.148s
```

可以看到，使用-short标记后，测试运行时间大约缩减了一半。

12. 测试的并发执行

在多核计算时代，我们在很多时候都想要更充分地利用CPU资源来获得更快的计算速度。如果我们的功能测试运行在拥有多核CPU或者多CPU的计算机上，那么我们就可以使用并发的方式来执行测试。通过-parallel标记，我们能够设置允许并发执行的功能测试函数的最大数量。但是，想成为能够被并发执行的功能测试函数需要具备一个先决条件。这个先决条件是：在功能测试函数的开始处加入代码t.Parallel()。在调用t.Parallel方法的时候，执行功能测试函数的测试运行程序会阻塞在这里，并等待其他同样满足并发执行条件的测试函数。当所有需要并发执行的测试函数都被清点且阻塞后，命令程序会根据-parallel标记的值，全部或者部分地并发执行这些功能测试函数中的在语句t.Parallel()之后的那些代码。

实际上，-parallel标记的默认值是我们通过标准库的代码包runtime的函数GOMAXPROCS设置的值。这个函数的作用我们可以暂且认为是设置Go语言并发处理的最大数量。从这一点上可以看出，即使我们给予的-parallel标记的值大于这个Go语言最大并发处理数，真正能够并发执行

的功能测试函数的数量也不会比它更多。所以，在通常情况下，我们其实并不需要在命令中加入`-parallel`标记，让它的实际值为默认值就好了。但需要注意的是，Go语言最大并发处理数的默认值为1。所以，如果我们想要某些测试函数中的代码被并发地执行，要做的就是测试源码文件的`init`函数中设置适当的Go语言最大并发处理数，并在这些测试函数中加入语句`t.Parallel()`。关于`runtime.GOMAXPROC`函数的知识，我们将在第6章详细讲解。

至此，我们基本介绍了定制化运行Go语言功能测试的全部方法。现在，读者可以用这些方法来运行前不久编写的针对`HashSet`、`myKeys`和`myOrderedMap`的测试源码文件，并体会`-timeout`、`-short`和`-parallel`标记的效果和作用。

5.1.2 基准测试

所谓基准测试（Benchmark Test，常被简称为BMT）是指，通过一些科学的手段实现对一类测试对象的某项性能指标进行可测量、可重复和可比对的测试。因此，在很多时候，基准测试已被狭义地称为性能测试。

在Go语言中，基准测试也是由`go test`启动的，并且也是应该被放在相应的测试源码文件中的。

1. 编写基准测试函数

与功能测试相同，针对其他源码文件中的程序实体的基准测试程序也是以测试函数为单位的。一个基准测试函数的名称和签名类似于：

```
func BenchmarkXxx(b *testing.B)
```

显然，它与功能测试函数有两点不同。首先，它的名称总是以“Benchmark”开始的。其次，它的参数类型是`*testing.B`而不是`*testing.T`。为了在名称上加以区别，我们一般把这个`*testing.B`类型的参数命名为`b`。在类型`*testing.B`之上也有很多有用的方法。比如，`*testing.B`类型也有`Log*`、`Error*`、`Fatal*`、`Fail*`和`Skip*`这几个系列的方法，并且它们的用法和作用也分别与`*testing.T`类型的同名方法相同。下面，我们重点关注`*testing.B`类型所特有的一些方法。

2. 关于计时器

在`*testing.B`类型中，与定时器相关的方法有3个它们是`StartTimer`、`StopTimer`和`ResetTimer`。这3个方法被用于操纵基准测试函数的计时器。该计数器的作用是计算当前基准测试函数的执行时间。

调用`b.StartTimer`方法意味着开始对当前的测试函数的执行进行计时。它总会在开始执行基准测试函数的时候被自动地调用。因此，这个方法被暴露出来的意义在于：计时器在被停止之后重新启动。相对应地，我们调用`b.StopTimer`方法可以使当前测试函数的计时器停止。例如，我们用这样的一个测试源码文件：

```
package bmt

import (
    "testing"
    "time"
```

```
)

func Benchmark(b *testing.B) {
    customTimerTag := false
    if customTimerTag {
        b.StopTimer()
    }
    time.Sleep(time.Second)
    if customTimerTag {
        b.StartTimer()
    }
}
```

我们把这个文件命名为**bmt_test.go**并把它临时放到某一个工作区的**testing/bmt**代码包中。之后，我们这样来运行其中的基准测试：

```
hc@ubt:~/golang/goc2p/src/testing/bmt$ go test -bench="." -v
testing: warning: no tests to run
PASS
Benchmark          1    1002811545 ns/op
ok      _/home/hc/golang/goc2p/src/testing/bmt    1.135s
```

现在，我们把其中的**customTimerTag**变量的值改为**true**。然后再来运行测试：

```
hc@ubt:~/golang/goc2p/src/testing/bmt$ go test -bench="." -v
testing: warning: no tests to run
PASS
Benchmark 20000000000      0.00 ns/op
ok      _/home/hc/golang/goc2p/src/testing/bmt    6.168s
```

显然，这两个输出内容在最后两行上有些不同。输出内容中的倒数第二行的信息很明显地被划分为了3个部分。这3个部分分别代表当前测试函数的名称、操作次数以及操作平均耗时。这里的操作次数是当前的基准测试函数被执行的次数，而操作平均耗时是当前基准测试函数的平均执行时间。

在反复运行**testing/bmt**代码包中的测试之后，我们可以观察到：当**customTimerTag**为**true**时，基准测试函数**Benchmark**可以被执行多次，而当**customTimerTag**为**false**时，此基准测试函数往往只能获得一次执行机会。这是因为在**testing**包中有这样的一个限制：在基准测试函数单次执行时间超过指定值（默认为1秒，也可以由标记**-benchtime**自定义）的情况下，只执行该基准测试函数一次。反过来讲，测试运行程序会在不超过这个执行时间上限的情况下尽可能多次地执行一个基准测试函数。

那为什么在**customTimerTag**被赋予**true**之后，基准测试函数**Benchmark**就可以被执行20亿次之多呢？其中的奥秘就在于**b.StartTimer**和**b.StopTimer**这两个方法上。我们已经知道，测试运行程序执行基准测试函数的次数的决策依赖于该函数的执行时间。我们在调用语句**time.Sleep(time.Second)**的之前和之后，分别停止和重启了**Benchmark**函数的计时器。这相当于不把**time.Sleep(time.Second)**语句的执行时间算在**Benchmark**函数的执行时间之内。这种情况下，执行**Benchmark**函数的时间已经基本可以忽略不计了（由上面示例中的**0.00 ns/op**可知）。所以，测试运行程序在**Benchmark**函数的累积执行时间未达到时间上限之前就会连续不断地重复执行它。

另一方面，调用语句`time.Sleep(time.Second)`的作用是让当前的测试程序“休息”一秒。所以，当`customTimerTag`的值为`false`的时候，`Benchmark`函数的单次执行时间就肯定会大于一秒。因此，测试运行程序也就不会对`Benchmark`函数执行第二次。

此外，方法`b.ResetTimer`在被调用时，会重置当前基准测试函数的计时器。也就是说，把该函数的执行时间重置为0。这相当于把当前函数中在`b.ResetTimer`语句之前的所有的语句的执行时间都从该函数的执行时间中减去。这与我们刚刚演示的联合使用`b.StartTimer`和`b.StopTimer`方法的那种用法有些类似。但是它的灵活性显然不如后者高。

3. 关于内存分配统计

方法`b.ReportAllocs`的含义是判断在启动当前测试的`go test`命令的后面是否有`-benchmem`标记。它会返回一个`bool`类型的结果值。我们会稍后解释`-benchmem`标记的作用。

方法`b.SetBytes`接受一个`int64`类型的值。它被用于记录在单次操作中被处理的字节的数量。当我们在基准测试函数中调用了这个方法。那么在测试结果信息中会出现类似于1.2 MB/s的内容。例如，如果我们把前面的那个基准测试函数`Benchmark`的声明改为：

```
func Benchmark(b *testing.B) {
    customTimerTag := false
    if customTimerTag {
        b.StopTimer()
    }
    b.SetBytes(12345678)
    time.Sleep(time.Second)
    if customTimerTag {
        b.StartTimer()
    }
}
```

那么再次执行这一测试的情形就会是这样的：

```
hc@ubt:~/golang/goc2p/src/testing/bmt$ go test -bench="." -v
testing: warning: no tests to run
PASS
Benchmark          1    1001788419 ns/op    12.32 MB/s
ok      _/home/hc/golang/goc2p_book/src/testing/bmt    1.083s
```

在针对`Benchmark`函数的操作信息的那一行信息中多出了一个部分——12.32 MB/s。它的含义是每秒被处理的字节的数量（以MB为单位）。这个数量其实就等于测试运行程序在执行（可能是多次）`Benchmark`函数的过程中每秒调用`b.SetBytes`方法的次数乘以传入的那个整数值。

请读者想象这样一个场景。在基准测试函数`Benchmark`中测试的是一个向文件系统中写入数据的函数。在写入成功后，我们会调用`b.SetBytes`方法并把真正写入的字节数作为参数传入。从而，通过测试结果信息中的xxx MB/s，我们就可以获知该函数每秒能向文件系统写入多少兆字节（MB）的数据了。

总之，`b.SetBytes`方法能够帮助我们从输入输出（IO）的角度统计出被测试的程序实体的实际性能。

4. 基准测试的运行

在上一小节的示例中，`go test`命令只运行了`cnet/ctcp`包中的功能测试。也就是说，它只会执行测试源码文件`tcp_test.go`中的功能测试函数`TestPrimeFuncs`。那么我们怎样运行基准测试呢？下面让我们来看一下可以使`go test`命令运行基准测试的那些标记，见表5-2。

表5-2 `go test`命令的基准测试标记说明

标记名称	标记描述
<code>-bench regexp</code>	在默认情况下， <code>go test</code> 命令不会运行任何基准测试，但可以使用该标记以执行匹配“ <code>regexp</code> ”处的正则表达式所代表的基准测试函数。“ <code>regexp</code> ”可以被替换成任何正则表达式。默认情况下，如果需要运行所有的基准测试函数，可以这样写：“ <code>-bench.</code> ”或“ <code>-bench=.</code> ”
<code>-benchmem</code>	在输出内容中包含基准测试的内存分配统计信息
<code>-benchtime t</code>	用来间接地控制单个基准测试函数的操作次数。这里的“ <code>t</code> ”指的是执行单个测试函数的累积耗时上限。“ <code>t</code> ”处的内容使用的是类型 <code>time.Duration</code> 可接受的时间表示法。“ <code>t</code> ”的默认值是 <code>1s</code> ，也就是1秒

在执行`go test`命令时加入上表中的标记，我们就可以运行针对代码包`cnet/ctcp`运行基准测试了。现在我们来具体操作一下：

```
hc@ubt:~/golang/goc2p/src$ go test -bench=Prime cnet/ctcp
PASS
BenchmarkPrimeFuncs      1      3009172100 ns/op
--- BENCH: BenchmarkPrimeFuncs
      tcp_test.go:49: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)...
ok      cnet/ctcp      7.020s
```

在上面的示例中，我们使用了`-bench`标记，并指定了测试函数名的正则表达式。这里，我们只运行名称中包含字符串“`Prime`”的基准测试函数。从打印信息上看，`go test`命令运行了一个叫作`BenchmarkPrimeFuncs`的测试函数。这个测试函数正是我们之前说的在测试源码文件`tcp_test.go`中的那个基准测试函数。注意，示例中有这么一行信息：`BenchmarkPrimeFuncs 1 3009172100 ns/op`。我们刚刚讲过这行信息的含义。

结构体类型`testing.B`的字段`N`可以被用来设置对基准测试函数中的某一个代码块的重复执行次数，像这样：

```
for i := 0; i < b.N; i++ {
    //测试代码
}
```

上面`for`语句块中的测试代码就是我们刚才所说的“基准测试函数中的某一个代码块”。这间接地使得基准测试函数中的某一个部分代码的执行次数可配置。我们要把需要进行基准测试的代码或者会主导性能指标的代码放在上述的`for`语句块中。这样测试运行程序就能够通过每次改变基准测试结构体类型`testing.B`的字段`N`的值来控制该代码块的执行次数。当然，我们还可以更灵活地使用这个字段`N`。比如，我们可以依据它做一些额外的运算，以达到更灵活地控制`N`的值的

效果。为了进行这种控制，我们就需要用到刚才提到的标记`-benchtime`了。在之前的表格中我们已经有所说明，这个标记的含义是指定单个基准测试函数的累积执行时间上限。测试运行程序会在

当前函数的累积执行时间达到上限之前，一次又一次地执行基准测试函数。而且，在每次执行之前，它还会把N的值设置到更大，即增加对“基准测试函数中的某一个代码块”的执行次数。为了更清楚地看到命令程序的行为，我们在基准测试函数中增加一行打印语句以打印相应的执行次数。这里为了让关于操作次数的日志信息更加突出，我们使用fmt包中的相关函数来打印日志。但要记住，这种打印日志的方式是不推荐的。现在请看下面的示例：

```
hc@ubt:~/golang/goc2p/src$ go test -bench=Prime -benchtime 20s cnet/ctcp
PASS
BenchmarkPrimeFuncs
Iterations (N): 1
Iterations (N): 10
Iterations (N): 1000
Iterations (N): 100000
Iterations (N): 500000
      500000      109318 ns/op
--- BENCH: BenchmarkPrimeFuncs
      tcp_test.go:49: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)...
      tcp_test.go:49: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)...
      tcp_test.go:49: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)...
      tcp_test.go:49: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)...
      tcp_test.go:49: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)...
ok      cnet/ctcp      70.569s
```

从测试源码文件tcp_test.go中代码打印的信息来看（请看打印信息中以“tcp_test.go”开始的那几行），命令程序执行了5次基准测试函数BenchmarkPrimeFuncs。并且，每次设置的N的值都在增加（请看打印信息中以“Iterations”开始的那几行）。起初，N的值为1。测试运行程序在每次执行该基准测试函数之前，都会先去检查该函数的累积执行时间是否已经超过了默认的累积执行时间上限或标记-benchtime的值。如果超过了则不会再执行该测试函数。如果没超过，则会根据上一次执行的操作次数和操作平均耗时来计算并设置N的值，然后再次执行该测试函数。至于计算N的值的具体算法，我们在此就不赘述了。如果读者对这一算法有兴趣，可以查看标准库的testing包的源码文件benchmark.go中的相关代码。-benchtime的默认值为1s。这也是在默认情况下BenchmarkPrimeFuncs函数总是只被执行一次的原因。具体地讲，在我的机器上，测试函数BenchmarkPrimeFuncs的单次执行时间大约为3秒，测试运行程序在第二次调用前获知它的累积执行时间已经大于1秒，所以并不会第二次执行它。

如果我们还想在看到基准测试函数的操作次数和操作平均耗时的同时获得这个过程中的内存分配情况，就需要用到-benchmem标记。示例如下：

```
hc@ubt:~/golang/goc2p/src$ go test -bench=Prime -benchmem cnet/ctcp
PASS
BenchmarkPrimeFuncs      1      5002927361 ns/op      50000 B/op
      129 allocs/op
--- BENCH: BenchmarkPrimeFuncs
      tcp_test.go:49: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)...
ok      cnet/ctcp      7.020s
```

测试依然通过，并且打印信息中多了两项数据。“50000 B/op”的意思是每次操作分配的字

节的平均数为50000个。“129 allocs/op”的意思是每次操作分配内存的次数平均为129次。

`go test`命令还可以接受一个可自定义测试运行次数并在测试运行期间改变Go语言最大并发处理数的标记,此标记记作`-cpu`。`-cpu`标记的值可以是一个整数列表,多个整数之间用英文半角逗号分隔。以运行功能测试为例,在测试运行程序执行测试的时候会对应这个整数列表中的每一个整数值做如下操作。

- 设置Go语言最大并发处理数,也就是调用`runtime.GOMAXPROCS`函数并把对应的整数作为参数传入。
- 运行目标代码包内的所有功能测试。

对于基准测试来说,测试运行程序也会执行相同的操作。

测试运行程序对`-cpu`标记的处理方式与我们在前面中提到过的`-parallel`标记刚好相反。`-parallel`标记默认使用Go语言最大并发处理数,而`-cpu`标记却会直接设置它。但是,由`-cpu`标记引发的Go语言最大并发处理数的设置操作并不会影响`-parallel`标记的默认值。因为`-parallel`标记的值是在测试运行程序初始化的时候被设置的。如果在`go test`命令中没有显式地加入`-parallel`标记,则它的值会被设置为测试运行程序初始化时刻的Go语言最大并发处理数。在这个时候,测试运行程序还没有把`-cpu`标记的值(如果有的话)解析成整数数组,就更不用说使用这个数组中的整数设置Go语言最大并发处理数了。

我们现在来试用一下这个标记。在下面的示例中我们只关注基准测试。

```
hc@ubt:~/golang/goc2p/src$ go test -bench=Prime -cpu=1,2,4,8,12,16,20 cnet/ctcp
PASS
BenchmarkPrimeFuncs          1          3007172000 ns/op
--- BENCH: BenchmarkPrimeFuncs
    tcp_test.go:53: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)... [GOMAXPROCS=1,
NUM_CPU=4, NUM_GOROUTINE=5]
BenchmarkPrimeFuncs-2        1          1000057200 ns/op
--- BENCH: BenchmarkPrimeFuncs-2
    tcp_test.go:53: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)... [GOMAXPROCS=2,
NUM_CPU=4, NUM_GOROUTINE=5]
BenchmarkPrimeFuncs-4         5           999657180 ns/op
--- BENCH: BenchmarkPrimeFuncs-4
    tcp_test.go:53: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)... [GOMAXPROCS=4,
NUM_CPU=4, NUM_GOROUTINE=5]
    tcp_test.go:53: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)... [GOMAXPROCS=4,
NUM_CPU=4, NUM_GOROUTINE=5]
BenchmarkPrimeFuncs-8         1          1001057300 ns/op
--- BENCH: BenchmarkPrimeFuncs-8
    tcp_test.go:53: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)... [GOMAXPROCS=8,
NUM_CPU=4, NUM_GOROUTINE=5]
BenchmarkPrimeFuncs-12        1          1001057200 ns/op
--- BENCH: BenchmarkPrimeFuncs-12
    tcp_test.go:53: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)... [GOMAXPROCS=12,
NUM_CPU=4, NUM_GOROUTINE=5]
BenchmarkPrimeFuncs-16        1          1006057500 ns/op
--- BENCH: BenchmarkPrimeFuncs-16
    tcp_test.go:53: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)... [GOMAXPROCS=16,
NUM_CPU=4, NUM_GOROUTINE=5]
```

```
BenchmarkPrimeFuncs-20      1      1000057200 ns/op
--- BENCH: BenchmarkPrimeFuncs-20
      tcp_test.go:53: Benchmark tcp listener & sender (serverAddr=127.0.0.1:8081)... [GOMAXPROCS=20,
      NUM_CPU=4, NUM_GOROUTINE=5]
ok      cnet/ctcp      28.252s
```

我们看到，测试运行程序执行基准测试函数BenchmarkPrimeFuncs的次数是7。这与-cpu标记的值1,2,4,8,12,16,20中的7个数字相对应。其实，测试运行程序在对代码包cnet/ctcp进行基准测试之前，也进行了功能测试，即调用了7次功能测试函数TestPrimeFuncs。只不过我们不想在测试结果信息中看到这些记录而已。如果读者想查看有关功能测试的记录信息的话，可以在执行上述go test命令时加入标记-v。

现在，我们再来仔细查看一下测试记录。“BenchmarkPrimeFuncs-2”中的2代表在执行测试函数BenchmarkPrimeFuncs之前设置的Go语言最大并发处理数量为2。其后第二行的测试记录的末尾包含了一些运行时环境信息：“[GOMAXPROCS=2, NUM_CPU=4, NUM_GOROUTINE=5]”。我们来简单解释一下。对于“GOMAXPROCS”，我们已经比较熟悉了。它代表Go语言最大并发处理数，在测试记录生成的那一时刻它的值为2。“NUM_CPU”代表当前计算机的CPU总内核数，即所有CPU的核心数相加得出的总和。这里的值为4。至于“NUM_GOROUTINE”，读者可以暂时把它理解为当前时刻的并发程序的数量。Goroutine是Go语言并发编程模型的核心概念之一。本书第7章会专门讨论它。纵观上面的测试结果信息，我们可以看到：随着Go语言最大并发处理数的增加，操作平均耗时会相应减少。但在Go语言最大并发处理数大于当前计算机CPU总内核数之后，操作平均耗时反倒增加了。这是为什么呢？读者可以先想一想这个问题，我在第6章再进行解释。

至此，我们介绍了两个与并发处理有关的标记——“-parallel”和“-cpu”。让我们通过表5-3来总结一下它们的用法。

表5-3 与并发处理有关的标记

标记名称	使用示例	说 明	
-parallel	-parallel 4	功能：	设置可并发执行的功能测试函数的最大数量
		默认值：	调用runtime.GOMAXPROCS(0)后的结果，即Go语言最大并发处理数量
		先决条件：	功能测试函数需要在开始处调用结构体testing.T类型的参数值的Parallel方法
		生效的测试：	功能测试
-cpu	-cpu 1,2,4	功能：	根据标记的值，迭代的设置Go语言并发处理最大数并执行全部功能测试或全部基准测试。迭代的次数与标记值中的整数个数一致
		默认值：	“”，即空字符串
		先决条件：	无
		生效的测试：	功能测试和基准测试

最后，还有一点需要注意。这两个标记的作用域都是代码包。换句话说，它们只能用于控制某一个代码包内的测试的流程。如果我们使用go test命令启动了多个代码包的测试，那么每个代码包中的功能测试永远是可并发执行的，而基准测试永远是串行执行的。如果我们把针对某一

个代码包的所有测试的运行过程看成一个整体的话，那么若在执行`go test`命令时加入了`-bench`标记，则针对各个代码包的测试运行过程会被串行地执行，否则它们将被并发地执行。但无论怎样，打印测试记录和结果信息的动作都是严格按照跟在`go test`命令后面的作为参数的代码包从左到右的顺序执行的。

5.1.3 样本测试

除了功能测试和基准测试之外，我们还可以向测试源码文件添加样本测试函数。样本测试函数的解析和执行由`go test`命令提供支持。但是，我们编写它却不需要使用`testing`代码包的API。

1. 编写样本测试函数

样本测试函数的名称需要以“`Example`”作为开始。并且，在这类函数的函数体的最后还可以有若干个注释行。这些注释行的作用是，比较在该测试函数被执行期间，标准输出上出现的内容是否与预期的相符。要想使这些注释行能被正确地解析，需要满足下面这几个条件。

- ❑ 这些注释行必须出现在函数体的末尾，且在它们和作为当前函数体结束符的“`}`”之间没有任何代码。否则，注释行对于样本测试来说就是无效的。
- ❑ 在第一行注释中，紧跟在单行注释前导符`//`之后的永远应该是`Output:`。否则这些注释行对于样本测试来说就是无效的。
- ❑ 在`Output:`右边的内容以及后续注释行中的内容都分别代表了标准输出中的一行内容。我们在编写这些注释行的时候应该遵守这一规定。

例如，我们有这样一个测试源码文件：

```
package et

import (
    "fmt"
)

func ExampleHello() {
    fmt.Println("Hello, Golang~")
    // Output: Hello, Golang~
}
```

在这个测试源码文件中仅包含了一个名为“`ExampleHello`”的函数。由于该函数的名称是以“`Example`”开头的，所以它可以被识别为一个样本测试函数。在这个函数的函数体中有一行注释。这个注释行处在该函数体的最后且其内容是以`Output:`开始的。所以，这个注释行可以被识别为一个被用于样本测试的注释行（以下简称样本注释行）。如果该测试函数在被执行的过程中向标准输出打印的内容就是`Output:`右边的内容`Hello, Golang~`，那么该测试函数中的测试就是通过的，否则就是失败的。下面我们就来看看运行这类测试的真实效果。

2. 样本测试的运行

我们把上述示例中的代码临时存放到某一个工作区的`testing/et`代码包中的`et_test.go`文件中，并在当前目录下使用`go test`命令启动对代码包`testing/et`的测试：

```
hc@ubt:~/golang/goc2p/src/testing/et$ go test -v
=== RUN: ExampleHello
--- PASS: ExampleHello (40.743us)
PASS
ok      _/home/hc/golang/goc2p/src/testing/et    0.082s
```

可以看到，ExampleHello函数中的测试是通过的。调用fmt.Println函数所产生的打印内容并不会真正地出现在标准输出上，而只会被用于与样本注释行中的内容做对比。如果我们稍微变动一下其中的样本注释行的内容，像这样：

```
// Output: Hello, Erlang~
```

那么，测试结果就会大为不同：

```
hc@ubt:~/golang/goc2p/src/testing/et$ go test -v
=== RUN: ExampleHello
--- FAIL: ExampleHello (39.421us)
got:
Hello, Golang~
want:
Hello, Erlang~
FAIL
exit status 1
FAIL    _/home/hc/golang/goc2p/src/testing/et    0.059s
```

测试结果信息明确地告诉我们，该样本测试函数欲向标准输出打印的实际内容Hello, Golang~与我们期望的内容Hello, Erlang~并不相同。

样本注释行可以不止一行。比如，我们把样本测试函数ExampleHello修改成这样：

```
func ExampleHello() {
    for i := 0; i < 3; i++ {
        fmt.Println("Hello, Golang~")
    }

    // Output: Hello, Golang~
    // Hello, Golang~
    // Hello, Golang~
}
```

由于欲打印的内容与预期相符，所以这一测试仍然是通过的。

但是，请注意，在样本测试函数的函数体末尾的多个样本注释行必须是连续的。也就是说，在它们之间不能间隔任何行，即使是空行也不行。还是那个原则，命令程序只会把在样本测试函数的函数体中的紧挨着当前函数体结束符“}”的注释行视为样本注释行。如果我们把样本测试函数ExampleHello修改成这样：

```
func ExampleHello() {
    for i := 0; i < 3; i++ {
        fmt.Println("Hello, Golang~")
    }

    // Output: Hello, Golang~
    // Hello, Golang~
```

```
// Hello, Golang~
}
```

命令程序只会检查处在ExampleHello函数的函数体结束符“}”的前一行的那个注释行。但是，由于这个注释行中的内容不是以Output:开始的，所以它并不能被称为一个样本注释行。

如果一个样本测试函数中没有任何样本注释行，那么这个函数仅仅会被编译而不会被执行。

3. 样本测试函数的命名

按照惯例，根据被测试的程序实体的种类，我们应该遵循这样的样本测试函数命名规则。

- ❑ 当被测试对象的是整个代码包时，样本测试函数的名称应该是Example，即直接以样本测试函数名的统一前缀作为函数名称。
- ❑ 当被测试对象的是一个函数时，对于函数F，样本测试函数的名称应该是ExampleF。
- ❑ 当被测试对象的是一个类型时，对于类型T，样本测试函数的名称应该是ExampleT。
- ❑ 当被测试对象的是某个类型中的一个方法时，对于类型T的方法M，样本测试函数的名称应该是ExampleT_M。
- ❑ 如果需要在样本测试函数的名称上添加后缀，那么需要用下划线“_”把该后缀与名称的其他部分隔开。并且，该后缀的首字母必须小写。例如，针对类型T的方法M且需要加入后缀“basic”的样本测试函数的名称应该是ExampleT_M_basic。

我们为什么要在这里特别强调样本测试函数的命名呢？你会在5.3节中找到答案。

以上内容就是我们编写样本测试函数所需要的全部知识。编写样本测试函数很简单，它几乎与编写普通的函数没什么区别。但是，这样的测试函数却很有用。通过简单的输出比对就能够判断出被测试对象的有效性和正确性。与功能测试函数和基准测试函数相比，样本测试函数更加轻快和干净。并且，样本测试函数还能够向被测试程序实体的使用者完整展示该程序实体的基本用法和使用技巧。

5.1.4 测试运行记录

在做软件系统运维工作的过程中，我们常常需要监控和记录软件系统本身和运行环境的健康状况。这里说的环境包括计算机硬件环境和计算软件环境。在硬件环境方面，我们主要考察计算机的负载状况，比如CPU使用率、内存使用率、磁盘使用情况，等等。在软件系统方面，往往包括内存分配、并发处理数量及死锁等情况。但是，在大多数情况下，当发现因软件系统的问题而导致的计算机负载过高、计算机暂时失去响应甚至宕机的时候，往往已经给系统使用者和维护者造成了损失。正因为如此，当今业界中才存在着大量的商用的或开源的性能测试工具。它们的功用都是尽量提前发现问题。但是，这些工具的测试对象往往都是整个软件系统或者其中可独立运行的子系统。而对于软件系统中的某个模块或者某段程序的性能测试，却只能通过软件开发者来进行。然而，软件开发者进行此项工作的大多数方式都非常简陋，甚至会把监控的代码直接插入到软件系统的程序中。这不但会污染到将会用在正式运行环境中的代码，更会增加这些正式代码的出错几率。

Go语言测试框架的亮点之一就是提供了一系列可以在测试运行时记录性能的方法。对于我们编写的测试函数和测试源码文件来说,它们都是非侵入式的。这些方法虽说是测试框架提供的,但是我们依然可以通过在执行`go test`命令时后跟标记的方式来启用和定制它们。下面,我们就来逐一讲解这些方法。

1. 收集资源使用情况

在测试运行时的资源使用情况监控方面,Go语言测试框架为我们提供了3个可用的标记。如表5-4所示:

表5-4 与收集资源使用情况有关的标记

标记名称	标记描述
<code>-cpuprofile cpu.out</code>	记录CPU使用情况,并写到指定的文件中,直到测试退出。 <code>cpu.out</code> 作为指定文件的文件名可以被其他任何名称代替
<code>-memprofile mem.out</code>	记录内存使用情况,并在所有测试通过后将内存使用概要写到指定的文件中。 <code>mem.out</code> 作为指定文件的文件名可以被其他任何名称代替
<code>-memprofilerate n</code>	此标记控制着记录内存分配操作的行为,这些记录将会被写到内存使用概要文件中。 <code>n</code> 代表着分析器的取样间隔,单位为字节。也就是说,每当有 <code>n</code> 个字节的内存被分配时,分析器就会取样一次

现在我们通过执行带`-cpuprofile`标记的`go test`命令来运行标准库的`net`代码包中的测试。示例如下:

```
hc@ubt:~/golang/goc2p/pprof$ go test -cpuprofile cpu.out net
ok      net      23.456s
```

我相信很多读者会去尝试着在运行多个代码包测试的时候使用`-cpuprofile`标记。但是很遗憾,`go test`命令程序不允许我们这么做。请看这个示例:

```
hc@ubt:~/golang/goc2p/pprof$ go test -cpuprofile cpu.out net runtime
cannot use test profile flag with multiple packages
```

对于我们稍后会讲到的`-memprofile`标记和`blockprofile`标记,这一约束同样存在。

回到主题。在一个代码包的测试运行完成后,我们会发现在命令执行的当前目录中会出现一个用于运行测试的可执行文件`net.test`。我们可以通过执行这个文件运行相应的测试。另外,在目标代码包的所在目录中也会出现一个名为`cpu.out`的文件。我们可以使用`go tool pprof`命令来交互式的对这个概要文件进行查阅,如:

```
hc@ubt:~/golang/goc2p/pprof$ go tool pprof ./net.test $GOROOT/src/pkg/net/cpu.out
Welcome to pprof! For help, type 'help'.
(pprof) top10
Total: 394 samples
      190 48.2% 48.2%      190 48.2% net.neverEnding.Read
       14  3.6% 51.8%        35  8.9% selectgo
         9  2.3% 54.1%      129 32.7% net.(*ioSrv).ExecIO
         9  2.3% 56.3%         9  2.3% runtime.casp
         8  2.0% 58.4%        17  4.3% runtime.MCache_Alloc
         7  1.8% 60.2%        12  3.0% runtime.cgocall
         7  1.8% 61.9%        36  9.1% runtime.mallocgc
```

```

5   1.3%  63.2%      5   1.3% siftup
4   1.0%  64.2%    141 35.8% net.(*netFD).Read
4   1.0%  65.2%      4   1.0% runtime.MHeap_LookupMaybe
(pprof)

```

关于go tool pprof命令的具体使用方法，大家可以参看我在Github上免费提供的Go命令教程，这里就不再赘述了。

标记-cpprofile相当于一个开关。它决定了在测试运行期间是否对CPU使用情况进行取样操作。这个取样操作的间隔是固定的，即每10毫秒会进行一次取样。实际上，当-cpprofile标记有效时，运行测试的程序会通过标准库代码包runtime/pprof中的API来控制该操作的启动和停止。用来启动CPU使用情况记录操作的函数名为pprof.StartCPUProfile。用来停止CPU使用情况记录操作的函数名为pprof.StopCPUProfile。

现在，让我们看看-memprofile标记。当这个标记有效时，测试运行程序会在运行测试的同时记录它们对内存的使用情况。这里所说的内存使用情况其实就是程序运行期间的堆内存的分配情况。

我们可以使用-memprofrate标记来设置分析器的取样间隔，单位是字节。它的值越小就意味着取样效果越好，因为取样间隔会更短。实际上，在testing包内部，-memprofrate标记的值会赋给runtime包中的int类型的变量MemProfileRate。这个变量的默认值为512 * 1024，即512K字节。如果设置该值为0，则代表停止取样。

下面我们来搭配使用这两个标记：

```

hc@ubt:~/golang/goc2p/pprof$ go test -memprofile mem.out -memprofrate 10 net
ok      net      22.917s

```

在命令执行完成后，同样会有两个文件被生成出来。一个是执行该命令时所在的目录下的可执行文件net.test。虽然这个文件在上面记录net包测试的CPU使用情况时已经被生成过，但它还是会被重新生成并替代原来的文件。另一个是目标代码包所在目录下的概要文件mem.out。同样也可以利用go tool pprof命令来对概要文件进行查询和分析。像这样：

```

hc@ubt:~/golang/goc2p/pprof$ go tool pprof ./net.test $GOROOT/src/pkg/net/mem.out
Adjusting heap profiles for 1-in-10 sampling rate
Welcome to pprof! For help, type 'help'.
(pprof) top10
Total: 1.0 MB
1.0  92.9%  92.9%      1.0  92.9% cnew
0.0   3.6%  96.5%      0.0   3.6% newdefer
0.0   0.9%  97.4%      0.0   0.9% resizefintab
0.0   0.6%  98.0%      0.0   0.8% time.NewTimer
0.0   0.6%  98.6%      0.0   0.6% runtime.malg
0.0   0.3%  98.9%      0.0   0.3% proresize
0.0   0.3%  99.2%      0.0   0.3% itab
0.0   0.1%  99.4%      0.0   0.1% net.(*anOp).Init
0.0   0.1%  99.5%      0.0   0.1% net.newFD
0.0   0.1%  99.6%      0.0   0.1% addtimer
(pprof)

```

需要注意的是，如果我们要获得最好的取样效果，可以将-memprofrate标记的值设置为1。

在这样的设置下，每当有一个字节的内存被分配，分析器就进行一次取样。这样固然可以使我们在分析概要文件的时候得到最精准的结果，但是同时也会付出高昂的代价。另外，为了得到内存分配情况的所有记录，我们还可以将-memprofilerate标记的值设置为1的同时，将环境变量GOGC设置为“off”。这样做可以使Go语言的垃圾回收器处于不可用状态。强烈建议只应该在运行测试且确实有必要时候去设置这个环境变量。因为这会让Go程序运行在一个没有垃圾回收器的环境中。在这种情况下，可用的内存只会不断的减少而不会增加。当没有可用内存时程序就会崩溃，甚至更糟。

2. 记录程序阻塞事件

绝大多数现代计算机编程语言都支持多线程编程。而在多个线程的执行调度过程中可能会产生线程阻塞。线程阻塞是由于线程因等待某个条件的触发而导致的执行暂停。当大量线程同时等待同一个触发条件或者多个线程间互为触发条件时，会造成资源的过度消耗和程序执行的停滞，更有甚者，可能会导致操作系统的不稳定以及计算机宕机。因此，对线程阻塞情况的记录是非常有必要的。

在Go语言的程序测试过程中，我们不但可以对被测试的程序使用CPU和内存的情况进行记录，还可以对程序中的Goroutine阻塞事件予以记录。我们可以通过在执行go test命令时加入标记-blockprofile和-blockprofilerate来达到记录阻塞事件的目的。经过我们上面对收集资源使用情况时需要用到的那4个标记的讲解，相信读者已经能够大概猜出这2个标记的作用了。对于它们的确切描述如表5-5所示。

表5-5 与记录程序阻塞事件有关的标记

标记名称	标记描述
-blockprofile block.out	记录Goroutine阻塞事件，并在所有测试通过后将概要信息写到指定的文件中。block.out作为指定文件的文件名可以被其他任何名称代替
-blockprofilerate n	这个标记用于控制记录Goroutine阻塞事件的时间间隔，n的单位为次，默认值为1

下面我们来试用一下：

```
hc@ubt:~/golang/goc2p/pprof$ go test -blockprofile block.out -blockprofilerate 100 net
ok      net      24.564s
```

当上述命令执行完毕后，我们就可以查看Goroutine阻塞情况的概要信息了。如下：

```
hc@ubt:~/golang/goc2p/pprof$ go tool pprof ./net.test $GOROOT/src/pkg/net/block.out
Welcome to pprof! For help, type 'help'.
(pprof) top10
Total: 64.755 seconds
 64.505 99.6% 99.6% 64.505 99.6% runtime.chanrecv1
  0.250  0.4% 100.0%  0.250  0.4% runtime.chansend1
  0.000  0.0% 100.0%  0.000  0.0% sync.(*Mutex).Lock
  0.000  0.0% 100.0%  0.000  0.0% runtime.chanrecv2
  0.000  0.0% 100.0%  0.000  0.0% sync.(*Cond).Wait
  0.000  0.0% 100.0% 64.755 100.0% gosched0
  0.000  0.0% 100.0%  0.000  0.0% io.(*PipeReader).Read
  0.000  0.0% 100.0%  0.000  0.0% io.(*PipeWriter).Write
```

```
0.000 0.0% 100.0% 0.000 0.0% io.(*pipe).read
0.000 0.0% 100.0% 0.000 0.0% io.(*pipe).write
(pprof)
```

与-memprofilerate标记相同，-blockprofilerate标记的值也是通过标准库代码包runtime中的API——函数SetBlockProfileRate——传递给Go运行时系统的。当我们传入的参数为0时，就相当于彻底取消记录操作。当我们传入的参数为1时，每一个阻塞事件都将被记录。实际上，-blockprofilerate标记的默认值就是1。这意味着，即使我们只加入-blockprofile而没有加入-blockprofilerate标记，每一个阻塞事件也都会被记录并被保存到概要文件中。

在本小节中，我们介绍了怎样在运行测试程序的同时对程序运行的情况进行监控的操作方法。这些操作都是通过在执行go test命令的同时加入某个或某些标记来启动的。实际上，这些标记所对应的功能都是通过标准库代码包runtime和runtime/pprof中的API实现的。我们可以直接使用这些API，并以此非常灵活地对程序运行时的资源使用情况进行记录。

5.1.5 测试覆盖率

在执行go test命令的时候，我们还可以加入一些标记，以使在测试结果信息中包含与测试覆盖率有关的内容。这里的测试覆盖率是指，作为被测试对象的代码包中的代码有多少在刚刚执行的测试中被使用到。换句话说，如果执行的该测试致使当前代码包中的80%的语句都被执行了，那么该测试的测试覆盖率就是80%。

表5-6中罗列了3个与测试覆盖率有关的标记。

表5-6 go test命令可接受的与测试覆盖率有关的标记

标记名称	使用示例	说 明
-cover	-cover	启用测试覆盖率分析
-covermode	-covermode=set	自动添加-cover标记并设置不同的测试覆盖率统计模式。支持的模式共有以下3个。 <input type="checkbox"/> set：只记录语句是否被执行过 <input type="checkbox"/> count：记录语句被执行的次数 <input type="checkbox"/> atomic：记录语句被执行的次数，并保证在并发执行时也能正确计数，但性能会受到一定影响 这几个模式不可以被同时使用 在默认情况下，测试覆盖率的统计模式是set
-coverpkg	-coverpkg bufio,net	自动添加-cover标记并对该标记后罗列的代码包中的程序进行测试覆盖率统计。在默认情况下，测试运行程序会只对被直接测试的代码包中的程序进行统计。该标记意味着在测试中被间接使用到的其他代码包中的程序也可以被统计。另外，代码包需要由它的导入路径指定，且多个导入路径之间以逗号“,”分隔
-coverprofile	-coverprofile cover.out	自动添加-cover标记并把所有已通过的测试的覆盖率的概要写入指定的文件中

这些标记是从Go语言的1.2版本开始被加入到其工具链中的。在这之前，恐怕我们只能够使

用像GNU Gcov这类的工具了。虽然这类工具很通用，但是对于Go语言来说，使用它们还是太繁琐了。并且，它们的兼容性也是一个问题。

不同于这些传统的测试覆盖率分析工具，Go语言提供的方式避免了对程序的二进制文件的动态调试步骤。我们说传统工具使用起来繁琐，动态调试也是其中的一个原因。Go语言的这种独特方法的核心思想很简单，即在代码包的源代码被编译成二进制文件之前重写它们，然后再编译和测试已经被修改了源代码的程序并进行统计分析。重写代码的工作很容易地被安插在编译测试的流程中，因为这个流程是由go命令直接控制的。

下面我们就来看看这些标记的具体使用方法。最简单的用法就是在go test命令后加入-cover标记以开启测试覆盖率统计。示例如下：

```
hc@ubt:~/golang/goc2p/src$ go test -cover cnet/ctcp
ok      cnet/ctcp    2.065s    coverage: 71.2% of statements
```

我们在对goc2p项目的代码包cnet/ctcp进行功能测试的时候加入了-cover标记。在最终的测试结果信息中新增了这次测试的覆盖率信息，即有71.2%的语句被测试到了。看来cnet/ctcp包中的测试代码还有进一步改善的余地。

我们可以在go test命令后加入-covermode标记。但是，无论我们设置哪一个测试覆盖率统计模式，在测试结果信息中的相关内容的格式都会与使用-cover标记时输出的相同。那这个标记到底有什么作用呢？稍后我们再解释。

标记-coverpkg使得我们可以获得间接被使用的代码包中的程序在测试期间的执行率。例如，在代码包cnet/ctcp中使用了bufio和net包中的程序，因此我们可以这样来获取针对它们的测试覆盖率：

```
hc@ubt:~/golang/goc2p/src$ go test cnet/ctcp -coverpkg=bufio,net
ok      cnet/ctcp    2.091s    coverage: 17.4% of statements in bufio, net
```

但是要注意，正如测试结果信息中显示的那样，在使用-coverpkg标记来指定要被统计的代码包之后，未被指定的代码则肯定不会被统计，即使是被直接测试的那个代码包。

标记-coverprofile会使测试运行程序把测试覆盖率的统计信息写入到指定的文件中。该文件会被放在执行go test命令时的那个目录下。示例如下：

```
hc@ubt:~/golang/goc2p/src$ go test cnet/ctcp -coverprofile=cover.out
ok      cnet/ctcp    2.078s    coverage: 71.2% of statements
hc@ubt:~/golang/goc2p/src$ ls
basic  cnet  cover.out  helper  logging  pkgtool
```

在这个被测试运行程序生成的概要文件中包含的信息会受到通过-covermode标记设置的统计模式的影响。那我们怎样查看这个概要文件中的内容呢？Go语言为我们提供了一个很给力的工具，这就是cover工具。

所谓的cover工具是Go语言自带的特殊工具之一。它被存放在\$GOROOT/pkg/tool/\$GOOS_\$GOARCH/目录中。与其他特殊工具（fix、vet、pprof和cgo等）一样，它也被分发为一个独立的文件。并且，我们总是可以通过go tool cover命令来运行它。无论在哪一个操作系统下，它

的主文件名都为cover。所以，我们也可以称它为cover工具。

该工具有如下两个功能。

- 根据指定的规则重写某一个源码文件中的代码，并输出到指定的目标上。
- 读取测试覆盖率的统计信息文件，并以指定的方式呈现。

我们先来看第一个功能。当我们运行附带了测试覆盖率相关标记的go test命令之后，测试运行程序就会使用cover工具在被测试的代码包中的非测试源码文件被编译之前重写它们。

重写的方式会由运行go test命令时的-covermode标记所指定的测试覆盖率统计模式决定。在默认情况下，这个统计模式为set。我们先来说明在set统计模式下cover工具对被测试的源代码的重写方式。

首先，有这样一个源码文件ct_test.go：

```
package ct

func TypeCategoryOf(v interface{}) string {
    switch v.(type) {
    case bool:
        return "boolean"
    case int, uint, int8, uint8, int16, uint16, int32, uint32, int64, uint64:
        return "integer"
    case float32, float64:
        return "float"
    case complex64, complex128:
        return "complex"
    case string:
        return "string"
    }
    return "unknown"
}
```

这个源码文件被临时放置到某一个工作区的testing/ct代码包中。其中的函数TypeCategoryOf的功能是根据参数值的类型的不同而返回不同的字符串标识作为结果值。这小段代码的开始由一个switch语句来控制流程。为了简单，我们只关心基本数据类型的参数值。如果参数值是非基本类型的，那么我们就统一返回"unknown"。

在我们运行go test -cover命令之后，测试运行程序会cover工具会把TypeCategoryOf函数重写成这样：

```
func TypeCategoryOf(v interface{}) string {
    GoCover.Count[0] = 1
    switch v.(type) {
    case bool:
        GoCover.Count[2] = 1
        return "boolean"
    case int, uint, int8, uint8, int16, uint16, int32, uint32, int64, uint64:
        GoCover.Count[3] = 1
        return "integer"
    case float32, float64:
        GoCover.Count[4] = 1
```

```

        return "float"
    case complex64, complex128:
        GoCover.Count[5] = 1
        return "complex"
    case string:
        GoCover.Count[6] = 1
        return "string"
    }
    typeCategory = "other"
    // 省略若干条语句
    GoCover.Count[1] = 1
    return typeCategory
}

```

这种重写的方式非常明了：原先的源代码中的每一个流程分支上都会被安插一个计数器（由GoCover.Count代表）。每当某个流程分支被执行的时候，相应的计数器就会被赋值为1。这样的赋值方式使得计数器只能表示该分支是否被执行过。

其中的变量GoCover所代表的是一个匿名的结构体类型的值。它的值被用于存储每个分支的计数值、每个分支的起始和结束位置在当前源码文件中的行数以及每个分支中的语句的条数等信息。我们在这里暂且称它为执行计数变量。执行计数变量的名称是可以通过-var标记告知给cover工具的。

注意，go test命令的测试目标可以是一个或多个代码包，而cover工具每次只能重写的一个源码文件。因此，如果被测试的源码文件有多个，那么go test命令会对每一个文件都运行一次cover工具。由于在被重写的每一个源码文件的源代码之上都需要添加一份执行计数变量的声明和初始化的代码，所以为了使这些变量的名称不重复，go test命令针对不同的源码文件传递给cover工具的-var标记的值也是不同的。-var标记的值总是在GoCover_前缀的基础上再追加代表了当前变量的编号的整数。执行计数变量在统计测试覆盖率的过程中起到了关键的作用。正是因为有了它们，测试运行程序才可能对每一个被测试的源码文件中的每一个分支的执行情况进行统计。

我们回到统计模式的话题。在count和atomic统计模式下，cover工具对被测试程序的重写方式与在set统计模式下相差无几。对于count统计模式，这样的语句

```
GoCover.Count[0] = 1
```

会被写为

```
CoverVar.Count[0]++
```

这相当于对分支的每一次执行都进行了记录。这就比只记录分支是否被执行要精细多了。而对于atomic统计模式，它会变为

```
_cover_atomic_.AddUint32(&CoverVar.Count[1], 1)
```

其中_cover_atomic_是代码包sync/atomic的别名，它在被重写后的源代码的开始处被这样导入：

```
import _cover_atomic_ "sync/atomic"
```

atomic统计模式一般只在存在并发执行的应用场景下才被使用，因为原子操作的执行会带来一定的性能损耗。而set和count统计模式则比它通用得多，它们对原先的程序的执行成本的影响也小很多。

顺便提一下，我们可以通过-mode标记把需要使用的测试覆盖率统计模式直接传递给cover工具。-mode标记在用法和含义上都与go test命令的-covermode标记一模一样。实际上，go test命令会把-covermode标记的值原封不动地作为运行cover工具时提送给它的-mode标记的值。但是，-mode标记并没有默认值。因此我们在使用cover工具时对某个源代码文件进行重写的时候必须添加-mode标记。

在默认情况下，被重写的源代码会输出到标准输出上。但是我们也可以使用-o标记把这些代码存放到指定的文件中。

下一步，在go tool cover命令以及所有标记和设置值之后的应该是要被重写的源码文件的路径。该路径可以是相对的也可以是绝对的。

总之，我们可以像这样使用cover工具对某一个源码文件进行重写：

```
go tool cover -mode=set -var="GoCover" -o dst.go src.go
```

实际上，go test命令也是这样运行cover工具的。其中，-mode标记的值就是我们在运行go test命令的时候添加的-covermode标记的值。如果未添加这个标记，那么go test命令会把-mode标记的值设置为set。而go test命令设置-var的值的方式我们在前面已经解释过了。

请记住，我们在使用cover工具重写一个源码文件时必备的两个参数是-mode标记和要被重写的那个源码文件的路径。

现在让我们来关注cover工具的第二个功能。我们之前用

```
go test cnet/ctcp -coverprofile=cover.out
```

命令为cnet/ctcp代码包生成了一个测试覆盖率的概要文件。现在我们使用cover工具来查看这个概要文件。先看一个示例：

```
hc@ubt:~/golang/goc2p/src$ go tool cover -func=cover.out
cnet/ctcp/base.go: Content          0.0%
cnet/ctcp/base.go: Err              100.0%
cnet/ctcp/base.go: NewTcpMessage    100.0%
cnet/ctcp/tcp.go:  Read             80.0%
cnet/ctcp/tcp.go:  Write            100.0%
cnet/ctcp/tcp.go:  Init              80.0%
cnet/ctcp/tcp.go:  Listen            50.0%
cnet/ctcp/tcp.go:  Close            85.7%
cnet/ctcp/tcp.go:  Addr              0.0%
cnet/ctcp/tcp.go:  NewTcpListener    100.0%
cnet/ctcp/tcp.go:  Init              88.9%
cnet/ctcp/tcp.go:  Send              83.3%
cnet/ctcp/tcp.go:  Receive           100.0%
cnet/ctcp/tcp.go:  Addr              0.0%
cnet/ctcp/tcp.go:  RemoteAddr        0.0%
cnet/ctcp/tcp.go:  Close            85.7%
cnet/ctcp/tcp.go:  NewTcpSender      100.0%
total: (statements)              71.2%
```

标记-`func`可以让`cover`工具把概要文件中包含的每个函数的测试覆盖率概要信息打印到标准输出上。这相当于是我们在运行`go test -cover`命名之后看到的在测试结果信息最后的那个测试覆盖率信息

```
coverage: 71.2% of statements
```

的明细单。

在这段输出内容中，除了最后一行，每一行的内容都包括了3项信息，分别是：函数所在的源码文件的相对路径、函数名称，以及函数中被测试到的语句的数量占该函数的语句总数的百分比。而最后一行内容中的百分比则是被测试代码包中的所有被测试到的语句的数量占该代码包中的语句总数的百分比。这与在前面的测试结果信息中所展示的百分比是一致的。

我们通过这段输出内容已经能够非常清晰地看到每个函数的测试覆盖率了。我们也可以以此来改进我们的测试程序。但是，有时候，我们更希望看到更加图形化的信息来直观的反应统计情况。`cover`工具也可以满足我们的这个需求。这需要用到`-html`标记。如果我们使用这样的命令：

```
go tool cover -html=cover.out
```

来查看概要文件，那么该命令会立即返回并且在标准输出上也并不会出现任何内容。取而代之的是当前操作系统的默认网络浏览器会被启动并显示`cover`工具刚刚根据概要文件生成的HTML格式的页面文件。显示效果如图5-1所示。



图5-1 网络浏览器显示HTML格式的测试覆盖率概要文件1

在这个HTML页面中，被测试到的语句以绿色显示，未被测试到的语句以红色显示，而未参加测试覆盖率计算的语句则用灰色表示。在它的左上角，我们还可以通过下拉框选择被测试代码包中的不同源码文件以查看它们的测试覆盖率情况。

这个HTML页面文件展示的是在`set`统计模式下生成的概要文件。而在`count`或`atomic`统计模式下，概要文件中除了会记录各个语句是否被测试到还会记录每条被测试到的语句都各被执行了多

少次，而对应的HTML页面文件也会稍有不同。请看下面的示例：

```
hc@ubt:~/golang/goc2p/src$ go test cnet/ctcp -covermode=
count - coverprofile=cover.out
ok      cnet/ctcp    2.074s   coverage: 71.2% of statements
hc@ubt:~/golang/goc2p/src$ go tool cover -html=cover.out
hc@ubt:~/golang/goc2p/src$
```

可以看到，我们在运行go test命令的时候加入了-covermode和-coverprofile标记。根据这两个标记的值，go test命令以count统计模式对被测试代码包中所有的非测试源码文件进行重写，并且会在测试完成后把收集到的测试覆盖率统计信息写到cover.out文件中。

而后，我们使用cover工具生成和查看与该概要文件对应的HTML页面。效果图参见图5-2。



图5-2 网络浏览器显示HTML格式的测试覆盖率概要文件2

我们看到，这个HTML页面文件中的内容和颜色都要比上一个更加丰富一些。由于依据的概要文件是在count统计模式下生成的，所以该HTML页面还可以体现出每条被测试到的语句被执行的次数。我们在此页面的最上一行的内容上就可以看到明显的变化，即提示在页面中使用了不同亮度的绿色来体现语句被执行次数的多少。我们浏览此行下面的代码，可以发现这样的展现效果相当的直观。

当我们把鼠标的光标停留在绿色的语句上面的时候，在光标附近还会出现代表了该语句被执行次数的数字，如图5-3所示。

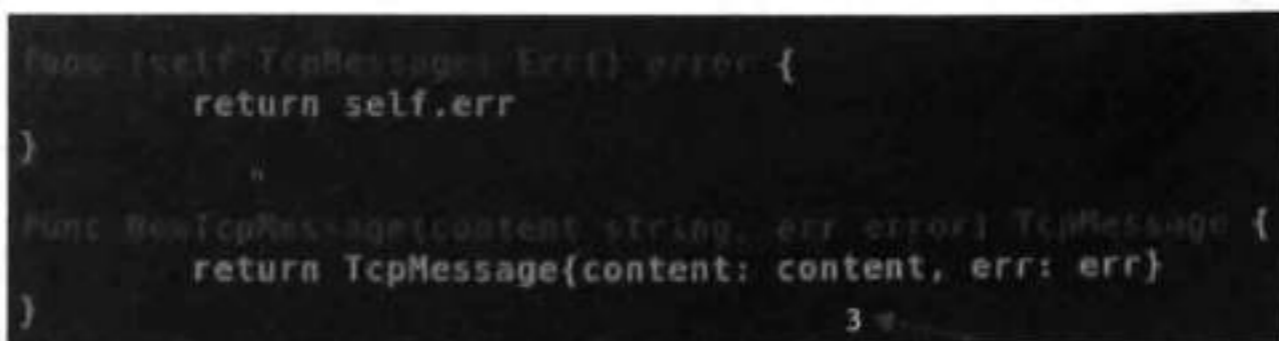


图5-3 网络浏览器显示HTML格式的测试覆盖率概要文件3

Go语言官方透露，由于光标停留后显示数字的这种显示不太直观，因此在之后的Go语言版本中，这种显示方式会被替换为在每行代码的最左边直接显示对应语句的被执行次数。这样更有利于了解函数的执行情况，同时也为性能分析提供了非常有用的第一手材料。

读者可能会有个疑问。根据概要文件中的内容显示，cover工具和测试运行程序对测试覆盖率的统计粒度都是语句级别的。但是，前面对它们的行为的描述来看，它们好像都是以代码行作为执行计数的对象的。事实的确是这样，它们的计数最小单位是代码行而不是语句。它们模糊了语句和代码行之间的区别，并以代码行来代表语句这个概念。但实际上这两者并不是一一对应的。这么做的好处有两个。

□ 被计数对象的划分清晰且统一。比如，有这样一行代码：

```
func1() && func2() && func3()
```

在func1()的结果值为true、func1()和func2()的结果值都为true以及3个函数调用表达式的求值结果均为true的情况下，这行代码是否可以被算作被执行过的呢？在cover工具看来，只要一行代码中的一部分被执行了，那么这行代码就可以被视为已经被执行。这样就免去了过细的分析过程。

□ 避免了复杂的词法分析。请看下面的代码：

```
func1(func2(func3()))
```

当这条嵌套的调用语句被执行时，应该增加几次计数呢？根据我们上面对cover工具の説明，读者应该可以猜到——增加一次计数。至于在这几个函数的内部都有哪些分支被执行了，以及每个分支的单次计数值都是多少，就是其他执行计数变量需要关注的事情了。

以代码行为单位的计数方式势必会使测试覆盖率统计存在一定的误差。但是，这项工作原本就不需要很高的精确度。至今的测试覆盖率统计工具基本上都没有做到精确无误。因为做到过分精确所需要投入的成本太高，而现实意义却微乎其微。

我们的关注重点应该是怎样从近似的测试覆盖率统计数据中找到测试程序的遗漏，并进行及时填补以提供测试的质量。

程序测试是软件开发过程中不容忽视的组成部分。并且，像这样以代码包和源码文件为单位的单元测试更应该受到我们的重视。我们需要依靠这样的测试来保障我们的程序质量，并把它们作为修改和重构程序时的辅助工具。而测试覆盖率统计信息能为我们的测试程序查缺补漏，并为我们的程序测试策略提供现实的依据。因此，go test命令的与测试覆盖率有关的标记以及cover工具都非常有用并且十分重要。我们应该很好地掌握它们并从中获得切实的好处。

最后，为了方便起见，我们下面再通过表5-7总结一下cover工具可接受的标记。

表5-7 cover工具可接受的标记

标记名称	使用示例	说 明
-func	-func=cover.out	根据概要文件（即cover.out）中的内容，输出每一个被测试函数的测试覆盖率概要信息
-html	-html=cover.out	把概要文件中的内容转换成HTML格式的文件，并使用当前操作系统中的默认网络浏览器查看它

(续)

标记名称	使用示例	说 明
-mode	-mode=count	被用于设置测试概要文件的统计模式。详见go test命令的-covermode标记
-o	-o=cover.out	把重写后的源代码的输出到指定文件中。如果不添加此标记，那么重写后的源代码会输出到标准输出上
-var	-var=GoCover	设置被添加到原先的源代码中的额外变量的名称

5.2 程序文档

程序文档是让程序使用者了解程序行为和特性的最有效的途径之一（另一个途径是“代码即文档”）。

在Go语言中，我们可以很方便地使用godoc命令在本机启动一个可被用于查看本机所有工作区中的所有代码包文档的Web服务。启动命令如下：

```
godoc -http=:9090 -index
```

关于godoc命令及其用法的说明，请大家参看我放到Github上的Go命令教程。

既然生成文档和显示文档的工作Go语言已经为我们做好了，那么我们需要做的就是写好程序的注释，为程序文档提供优秀的素材。

1. 编写程序注释

Go语言在注释风格中融入了C语言和C++语言的基因。我们既可以使用C++语言风格的行注释：

```
// 行注释
```

也可以使用C语言风格的块注释：

```
/*  
    块注释  
*/
```

行注释我们应该已经很熟悉了。它已经在我们的示例中出现过很多次了。行注释能且只能占用单独一行来存放注释的内容。它是最基本的注释方式，并可以出现在源码文件中的任何地方。

块注释是更加灵活的一种注释方式。它可以占据任意行存放注释内容。块注释更适合作为一个代码包的文档性注释。

需要注意的是，无论是哪一种风格的注释都不能出现嵌套的情况。

2. 代码包的注释

每个代码包都应包含一段独立的代码包注释。这段注释应该对当前代码包的功能和用途进行总体性的介绍。按照惯例，代码包注释应该被存放到当前代码包目录下的doc.go文件中。在这个文件中，应该有与包中其他源码文件相同的代码包声明语句，并在该声明语句之上以块注释的方式插入代码包注释。当然，在代码包中只有一个源码文件或者代码包注释比较短的情况下，把代码包注释插入到与当前代码包同名的源码文件中的代码包声明语句之上也未尝不可。顺便说一句，当代码包中仅有一个源码文件的时候，建议源码文件的主文件名与当前代码包的名称相同。

代码包注释总是会出现在godoc命令生成的对应文档页面的首要位置上。也就是说，代码包注释会作为该代码包的文档的第一段说明出现。所以，我们不但应该花时间编写它，还应该让读过它的人能够对当前代码包有足够正确和清晰的认识。在必要时，我们应该向代码包注释中加入一些列表、注释和相关文档的指引。例如，在标准库中，代码包fmt的代码包注释被放在了\$GOROOT/src/fmt/doc.go中。与其相对应的文档页面如图5-4所示。



图5-4 代码包fmt的文档页面1

在这个文档页面中，在“Overview”一栏中的就是fmt包的代码包注释。

3. 程序实体的注释

看到上图中的那个紧挨在“Overview”栏上面的蓝色的（意味着带有链接）的目录了吗？其中的“Index”栏包含的是当前代码包中的所有可导出的程序实体的文档的索引。在我们点击其中一个索引（如“type State”）之后，页面会定位到与该索引相对应的程序实体文档的锚点处，如图5-5所示。

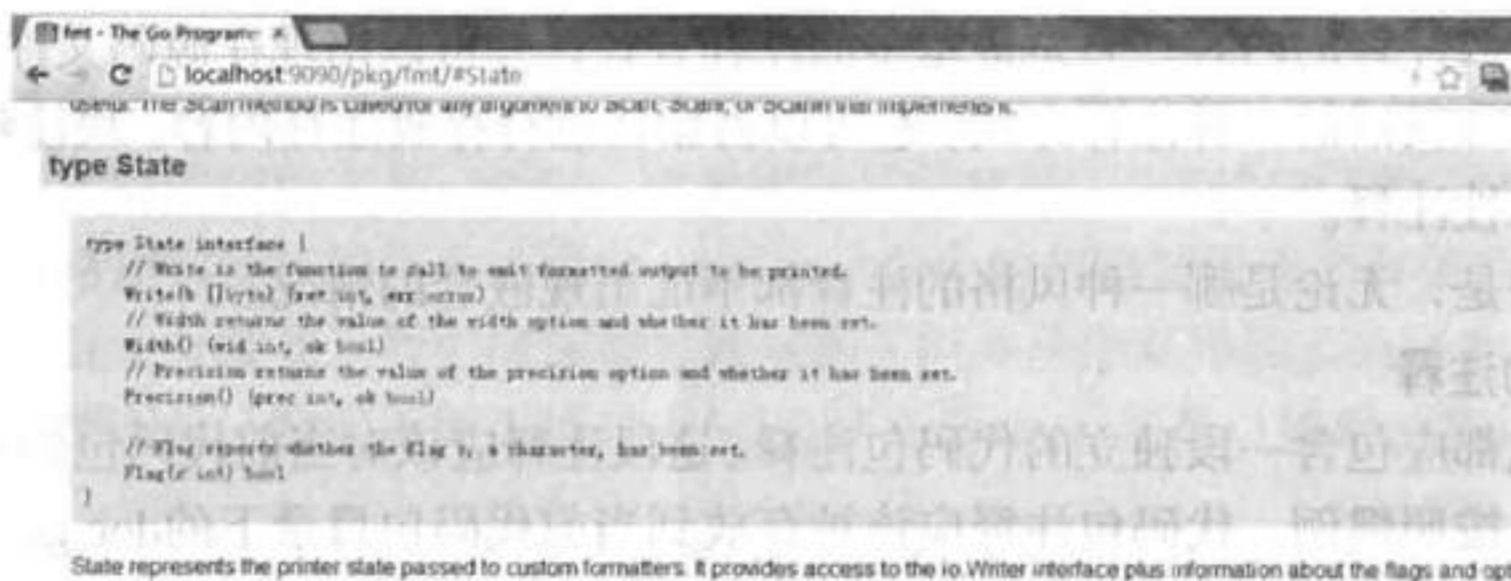


图5-5 代码包fmt的文档页面2

现在，我们把鼠标放在页面中“type State”的“State”处并点击。页面会跳转到声明fmt.State的源代码处（见图5-6高亮的部分）。

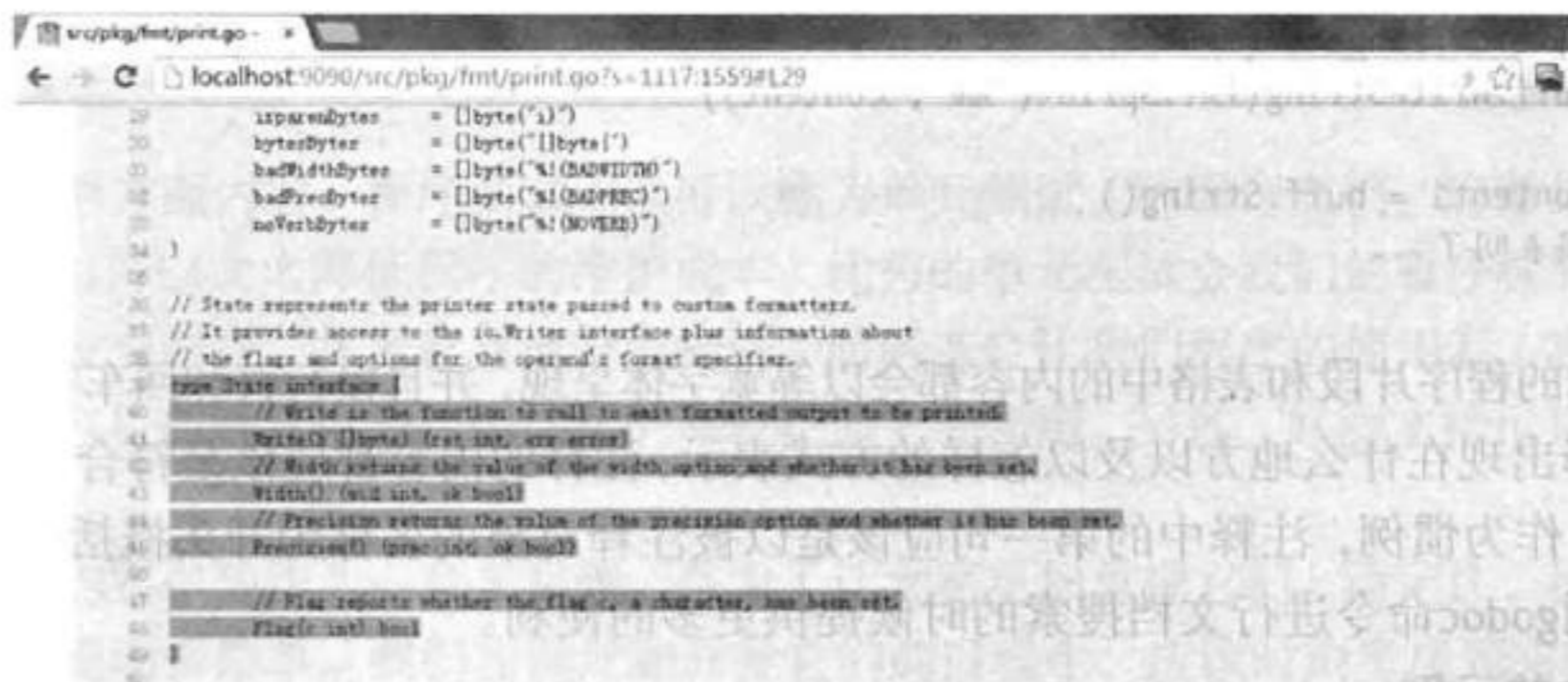


图5-6 代码包fmt的文档页面3

纵观上面展示的一系列配图，我们可以了解到，程序实体的文档即是它的声明代码以及紧挨在上面的行注释。这也部分体现了“代码即注释”的思想。如果我们仔细考量程序实体的各部分命名和声明的手法（比如与`fmt.State`接口类型声明中的每个方法声明对应的那些注释），那么不论是对于它的使用者还是对于它的文档的阅读者来说都会有很大的帮助。

重申一下，程序实体文档中的文字性说明由紧挨在它的声明之上的那几个行注释呈现。

4. 常量和变量的注释

由于Go语言语法允许一次声明一组常量或变量，所以为这样的程序实体声明编写注释会稍有不同。请看如下示例：

```
// IP address lengths (bytes).
const (
    IPv4len = 4
    IPv6len = 16
)
```

这段代码摘自代码包`net`中的源码文件`ip.go`。对于这样一组声明来说，我们应该把对它们的描述统一放在关键字`const`（或关键字`var`）之上且紧邻的注释行中，而不是分别为每一个常量（或变量）声明添加注释。

Go语言的注释就是普通的文本。因此，我们不要试图使用一些非文本的字符为注释添加多余的格式，比如用HTML标签`
`表示换行、用很多星号组成一个横幅，等等。即使添加了这些字符，`godoc`也会以普通文本的方式把它们呈现出来。那么，怎样控制注释中的格式呢？实际上，`godoc`命令会根据上下文来决定是否对注释进行格式化。因此，我们要做的就是让注释在普通文本状态下就显示良好。比如，统一使用制表符来表示缩进、使用空行来分隔标题与主体内容以及各个段落，以及手动的为一些长文本进行换行（`godoc`命令会忽略它们，但阅读起来就会舒服很多），等等。另外，`godoc`命令并不会用等宽字体来展现文档。但是也有一个例外。我们可以使用更多的相同数量的制表符（与上下文相比）来表示一个程序片段或表格，像这样：

```
/*
    下面是一段代码：
    var buff bytes.Buffer
*/
```

```

for _, e := range s {
    buff.WriteString(fmt.Sprintf("%v", content))
}
var content1 = buff.String()
这段代码表明了...
*/

```

这样表示的程序片段和表格中的内容都会以等宽字体呈现，并且换行符和回车符也不会被忽略。

无论注释出现在什么地方以及以怎样的方式表示，注释的内容都应该是符合英语或中文语法规则的句子。作为惯例，注释中的第一句应该是以被注释对象的名称为开头概括性语句。这样能够在我们使用godoc命令进行文档搜索的时候提供更多的便利。

5. 文档中的示例

在代码包的文档页面中还可以包含有针对性的示例代码。实际上，这些示例代码是godoc命令程序自动从代码包中的测试源码文件中取得的。具体的获取目标是，符合命名规则的样本测试函数中的代码。因为这样的示例代码可以与文档页面中的某个程序实体的文档对照起来（紧跟在该程序实体的文档后面出现）。例如，接口类型`net.Listener`的示例代码被放在`net`代码包下的样本测试函数`ExampleListener`中。而在`net`包的文档页面中，它们的展现效果如图5-7所示。

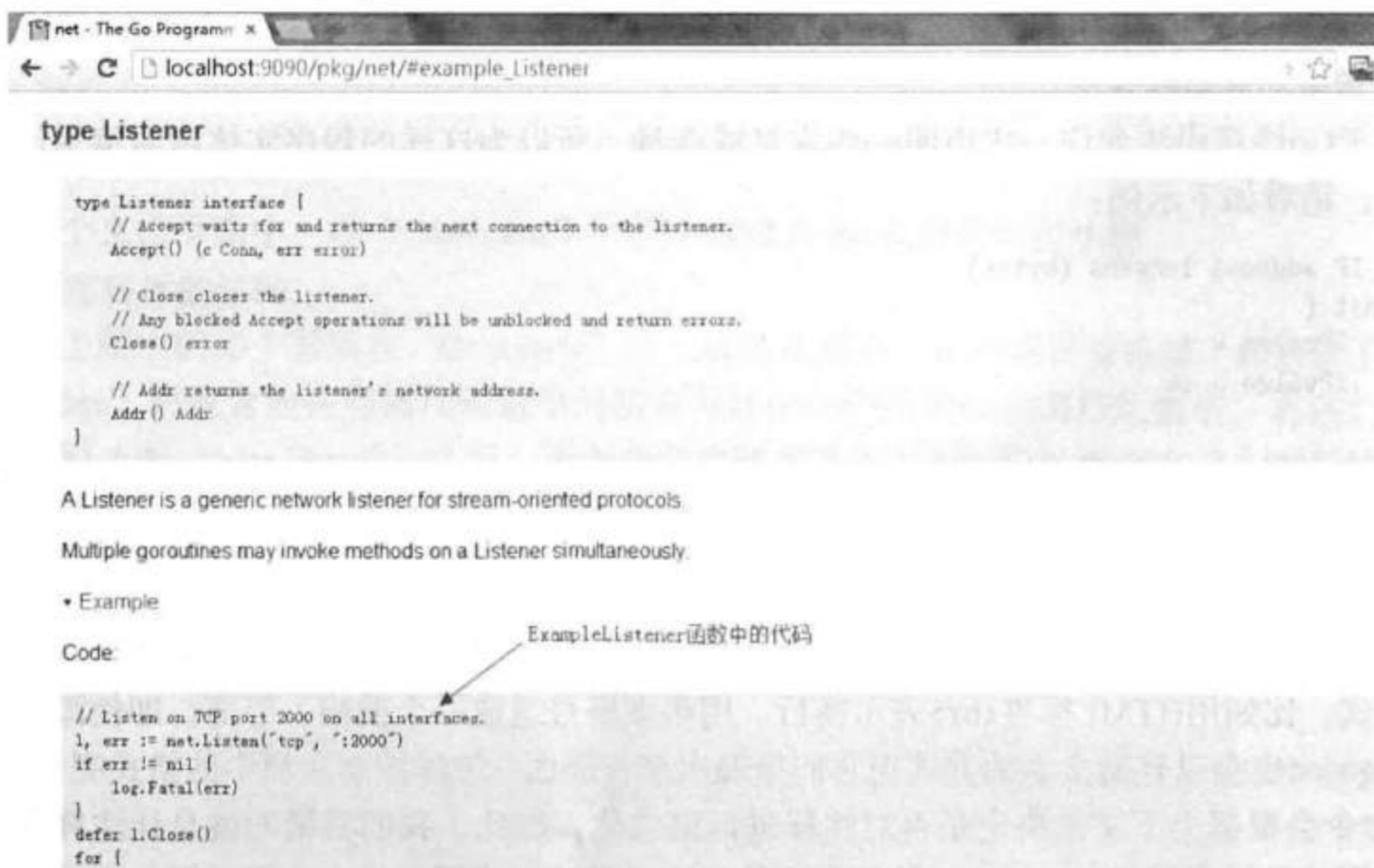


图5-7 代码包`net`的文档页面

这也正是我们在上一节强调样本测试函数的命名规则的原因。合格的样本测试函数既可以用于程序测试，又有利于文档中的程序示例的合理展示。

至此，我们已经介绍了与Go语言程序文档的展示和编写有关的所有内容。希望通过阅读这些内容，读者能够更积极地为自己的程序提供文档，并使其内容更加规范和翔实。

5.3 本章小结

本章讨论了两方面内容：程序测试（也可以称为单元测试）和程序文档。前者是程序正确性的有力保障，同时可以大大降低程序的维护成本。优秀的单元测试会为我们的程序保驾护航。它可以让我们及早地发现错误，并更容易地定位错误。而后者会让我们程序的使用者（甚至是几个月之后的自己）更容易理解其中的代码，并更加轻松地上手使用。另外，好的文档也是高质量程序的体现和标志。

我们把这些内容单独作为一章来呈现，是为了让大家更加重视它们。请记住，程序设计和开发不只是编写代码那么简单。我们在设计和开发它们的过程中，应该时刻考虑到它们的可读性、可维护性和可扩展性。而程序测试和程序文档恰恰是可以帮助我们更容易地做到或提高这些方面的辅助工具。

Part 3

第三部分

并发编程

终于到了为大家讲解怎样用 Go 语言编写并发程序的时候了。并发编程是本书的主题，更是当今开发服务器端软件必会涉及的一个重要话题，尤其是在互联网领域。本部分共分为 3 章。

我们会从并发编程的历史开始讲起。从多进程编程到多线程编程，再到 Go 语言的并发编程模型，我们会讲述它们的特点并进行对比。Go 语言对多进程编程实际上也是有支持的。在这方面，我们会重点讲解它对 Socket 的强力支持。虽然 Go 语言拥有自己的并发编程模型，但是它的根基还是操作系统的线程模型。所以，我们还是会在第 6 章讲述一些多线程编程方面的知识，同时也会指出当今主流的并发编程模型的不足之处。当然，我们会在该章的最后详细地讲述 Go 语言的并发编程模型。

在第 7 章，我们会重点讲解 Go 语言并发编程模型中与应用程序联系最紧密的两个组件（或者说工具）——Goroutine 和 Channel 的使用方法和技巧。在这一章的最后，我们还会用代码说话，带领大家编写一个真实完整、开箱即用的示例程序。

虽然 Go 语言并不推荐以同步的方式在并发环境中共享数据和建立通讯，但是它还是提供了一些同步工具让我们选用，第 8 章会介绍这部分内容。这些同步工具在一些应用场景下还是很有用的。同样的，我们也会以一个示例程序结束这一章的内容。

好了，现在让我们进入主题。

如果你是一个跟得上节奏的软件开发者的话，那你一定听说过甚至实践过并发编程。它通常代表了更快的程序执行速度和更复杂的程序设计方式。并发编程的含义并不单一，它可能是基于一个独立程序的（比如多线程编程）、一台计算机的（比如多进程编程），又或者是基于一个网络的（比如分布式计算）。我们在这里所说的并发编程特指在单台计算机的环境下可以使计算机指令同时发生并相互协作的计算机语言级别的并发计算技术。

并发编程是一种现代计算机编程技术，绝大多数的现代编程语言都在不同程度上对它进行了支持。在本章，我们讲述的重点是帮助读者深入地了解并发编程，并且对目前主流的并发编程模型有所理解和比较，以及在多核时代的大背景下怎样才能更好地进行并发编程。当然，我们还会带领读者对Go语言的并发编程模型探究一番。

6.1 并发编程基础

并发这个概念由来已久。其主要思想是使多个任务可以在同一时间执行以便能够更快地得到结果。几乎在计算机问世的时候，并发就已经出现在它的世界中了。

并发编程的思想来自于多元程序（multiprogramming，或称多任务）操作系统。多元程序操作系统允许同时运行多个程序。在早期的单用户计算机系统的操作系统中，任务是被一个接一个地读取、寻找资源并运行的。各个任务的执行完全是串行的，只有在一个任务被运行完成之后另一个任务才会被读取。而多元程序操作系统则允许终端用户同时运行多个程序。当一个程序暂时不需要使用CPU的时候，系统会把该程序挂起或中断，以使其他程序可以使用CPU。

最早支持并发编程的计算机编程语言是汇编语言。不过那时并不没有任何理论基础来支持这种编程方式。这使得一个细微的编程错误就会使程序变得非常不稳定。并且，对这种程序的测试也是几乎不可能的。

在20世纪60年代末，多元程序操作系统已变得非常臃肿和脆弱。对系统资源的无限制地抢夺成为了频繁发生的程序死锁现象的导火索。连其缔造者都不得不公开发表言论说，这是一场软件危机。

不过，早在20世纪60年代中叶，计算机科学家就迈出了深入探索并发编程的第一步。在不到15年的时间里，他们逐步把并发编程思想凝炼成了理论，同时开发出了一套关于它的描述方法。之后，他们把这套理论融入到了编程语言当中，并用这些编程语言来编写操作系统模型。在

20世纪70年代，第一本关于操作系统和并发编程原则的简明参考书问世了。这意味着一个新的编程理论正式形成。正如上面介绍的那样，发展并发编程思想的最初动机来源于开发可靠的操作系统的强烈欲望。但在并发编程理论形成不久之后，它就被公认为一个可以不仅被应用于操作系统开发的通用编程理论。

经过多年的演进和变化，并发程序的编写早已没有先前那么复杂了。大多数现代软件设计技术都可以很方便地产生出支持并发的程序。在它们看来，只有一个例程（routine）的并发程序其实就相当于一个串程序。这种转换是平滑的和透明的。总体来看，编程人员感觉编写并发程序会更加困难的原因有两个。

- 缺乏既非常适合开发应用程序又对并发编程有良好支持的编程语言。

- 感觉（仅仅是感觉）并发编程的理论太难了。

第一个原因的确是存在的。不过，现在许多善于开发应用程序的编程语言都在致力于降低用它们编写并发程序的门槛。至于第二个原因，我认为这不应该成为专业编程人员躲避编写并发程序的借口。况且，由于越来越多的编程语言对并发编程具有良好的支持，我们在学习计算机编程的时候，更加不可避免地接触到并发编程。毫不夸张地讲，如果我们想要真正理解一门编程语言以及了解怎样才能编好程序，那么学习并发编程这一步也是必不可少的。更不用说，作为软件运行的基础——计算机硬件也越来越向着并行化发展。

在本章中，我们会对当今主流的一些并发编程模型进行阐述和对比。在此基础上，我们会专门针对Go语言的并发编程模型进行详细的说明。不过在这之前，让我们先来简要介绍一下并发编程理论中的一些基础概念。

6.1.1 串程序与并发程序

一个串程序特指一个只能被顺序执行的指令列表。而一个并发程序则是被并发的执行的两个或两个以上的串程序的统称。并发程序允许其中的串程序运行在一个或多个可共享的CPU之上，同时也允许每个串程序都运行在专为它服务的CPU上。前一种方法也被称为多元程序。多元程序由来自荷兰的图灵奖得主Edsger Wybe Dijkstra在1968年提出。多元程序由操作系统内核支持并提供多个串程序复用多个CPU的方法。这种方法被称为多元处理。多元处理是指计算机中的多个CPU共用一个存储器（即内存），并且在同一时刻可能会有数个串程序分别运行在不同的CPU之上。它是由美国计算机科学家Anita K. Jones和Peter M. Schwarz在1980年提出的。无论从理论还是实践角度看，多元程序和多元处理都是串程序得以并发甚至并行运行的基础支撑。在现代计算机系统中，这两种方式已经得到了很好的融合。

6.1.2 并发程序与并行程序

在一些参考文献和图书中，常常把并发和并行这两个概念混淆在一起。但是，它们实际上有着很清晰的区别。并发程序是指可以被同时发起执行的程序。而并行程序则是被设计成可以在并行的硬件上执行的并发程序。换句话说，并发程序代表了所有可以实现真正的或者可能的并发行为的程序。它是一个比较宽泛的概念。这其中包含了并行程序。并行程序是并发程序中的一种。

6.1.3 并发程序与并发系统

首先, 并发程序属于程序。即使它被划分为许多部分(可以是规模更小的程序), 只要这些部分之间是紧密地关联在一起的, 并且可以被看作一个概念上的整体, 那么它就属于一个程序, 也可以称之为一个内聚的软件单元。另一个方面, 程序与程序之间可以通过协商一致的协议进行通讯, 并且它们之间是松耦合的。它们可以被看作是一个系统, 而不是程序。并发程序和并发系统中的并发的含义是一致的。但是, 并发系统更有可能是并行的, 因为其中的多个程序一般可以同时在不同的硬件环境上运行。因此, 并发系统也常常被称为并行系统。与并发系统同义的一个更流行的词是分布式系统。

本章以及全书所关注的并发编程意在编写并发程序, 而不是实现并发系统。

6.1.4 并发程序的不确定性

一个串行程序中的所有活动的先后顺序都是固定的, 而一个并发程序中的活动只是部分有序的, 也就是说其中一些活动的发生顺序并没有被明确地指定。这一特性被称为不确定性。这种不确定性导致了并发程序的每次运行的活动执行路径都是不同的, 即便是在输入数据相同的前提下。

6.1.5 并发程序内部的交互

我们已经知道, 并发程序内部会被划分为多个部分, 每个部分都可以被看作是一个串行程序。在这些串行程序之间可能会存在交互的需求。比如, 多个串行程序可能都要对一个共享的资源进行访问。又比如, 它们需要相互传递一些数据。在这种情况下, 我们就需要协调它们的执行。这就涉及了同步。同步的作用是避免在并发访问共享资源时可能存在的冲突, 以及确保在互相传递数据时能够顺利地接通。我们会在本章后续部分详细地介绍各种同步的方法。

根据同步的原则, 程序如果想使用一个共享资源, 就必须先请求该资源并获取到对它的访问权。当程序不再需要某个资源的时候, 它应该释放该资源, 即放弃对它的访问权。一个程序对资源的请求不应该导致其他正在访问该资源的程序中断, 而应该等到那个程序释放该资源之后再行请求。也就是说, 在同一时刻, 某个资源应该只被一个程序占用。

传递数据是并发程序内部的另一种交互方式。它也被称为并发程序内部的通讯。实际上, 协调这种内部通讯的方式不只同步这一种。我们也可以用异步的方式对通讯进行管理。这种方式使得数据可以不加延迟地发送给数据接收方。即使在数据接收方还没有立即为接收该数据做好准备的时候, 也不会造成数据发送方的等待。数据会被临时存放在一个被称为通讯缓存的数据结构中。通讯缓存是一种特殊的共享资源, 它可以同时被多个程序使用。数据接收方可以在准备就绪之后按照数据被存入通讯缓存的顺序从通讯缓存中接收它们。我们稍后还会看到对这两种交互方式的描述和对比。

好了, 上面这些知识作为讲解并发编程模型的铺垫已经足够了。在这之后, 我会对当今的主流并发编程技术进行介绍。

6.2 多进程编程

在现代操作系统中，我们可以很方便地编写出多进程的程序。在多进程程序中，如果多个进程之间需要协作完成任务，那么进程间通讯的方式就是需要重点考虑的事项之一。这种通讯常被叫作IPC（Interprocess Communication）。不同版本的Unix及其衍生系统中所支持的IPC方法都不尽相同。同时，针对IPC制定的标准也不止一个。因此，为了简单和统一，我们在讨论IPC的概念和使用方法的时候只针对Linux操作系统。我们会详细地描述Go语言能直接操纵的那些IPC方法。

在Linux操作系统中，我们可以使用的IPC方法有很多种。从处理机制的角度看，它们可以被分为三大类，即基于通讯的IPC方法、基于信号的IPC方法以及基于同步的IPC方法。其中，基于通讯的IPC方法又被分为以数据传送为手段的IPC方法和以共享内存为手段的IPC方法。前者包括了管道（Pipe）和消息队列（Message Queue）。管道可以被用来传送字节流，而消息队列可以被用来传送结构化的消息对象。以共享内存为手段的IPC方法主要以共享内存区（Shared Memory）为代表。它是最快的一种IPC方法。基于信号的IPC方法就是我们常说的操作系统的信号（Signal）机制。它是唯一的一种异步的IPC方法。在基于同步的IPC方法中，最重要的就是信号灯（Semaphore）。

在本节，我们会详细地描述Go语言支持的IPC方法。它们是管道、信号和Socket。不过，在介绍它们之前，我们还需要先了解一些基本的概念。

6.2.1 进程

我们在介绍具体IPC方法之前，理所当然地要对进程本身进行解说。这对于我们更加深刻地理解各种IPC方法的概念和使用也是很有好处的。

1. 进程的定义

进程是Unix衍生操作系统（包括Linux操作系统）的根本，因为所有的代码都是在进程中执行的。我们通常把一个程序的执行称为一个进程。反过来讲，进程被用于描述程序的执行过程。因此，程序与进程成为了一对相依的概念。它们分别描述了一个程序的静态形式和动态特征。除此之外，进程还是操作系统进行资源分配的一个基本单位。

2. 进程的衍生

了解Unix/Linux系统编程的读者都应该知道，一个进程可以使用系统调用fork创建若干个新的进程。前者被称为后者的父进程，后者被称为前者的子进程。每个子进程都是源自它的父进程的一个副本。它会获得父进程的数据段、堆和栈的副本，并与父进程共享代码段。每一份副本都是独立的，子进程对属于它的副本的修改对其父进程和兄弟进程（同父进程）都是不可见的，反之亦然。全盘复制父进程的数据是相当低效的一种做法。Linux操作系统内核（以下简称内核）使用写时复制（Copy On Write，常被简称为COW）等技术来提高进程创建的效率。当然，刚被创建的子进程也可以通过系统调用exec把一个新的程序加载到自己的内存中，而原先在其内存中的数据段、堆、栈以及代码段就会被替换掉。在这之后，子进程执行的就会是那个刚刚被加载进来的新程序。

在Unix/Linux操作系统中，每一个进程都有父进程。所有的进程共同组成了一个树状结构。

内核启动进程作为进程树的根并负责系统的初始化操作。它是所有进程的祖先。然而，这个进程也是有父进程的——就是它自己。如果某一个进程先于它的子进程结束，那么这些子进程将会被内核启动进程“收养”，成为它的直接子进程。

3. 进程的标识

为了管理进程，内核必须对每个进程的属性和行为进行详细的记录，包括进程的优先级、状态、虚拟地址范围以及各种访问权限，等等。更具体地说，这些信息都会被记录在每个进程的进程描述符中。进程描述符并不是一个简单的符号，而是一个非常复杂的数据结构。被保存在进程描述符中的进程ID（常被称为PID）是进程在操作系统中的唯一标识。进程ID为1的进程就是我们之前所说的内核启动进程。进程ID是一个非负整数且总是被顺序的编号。新被创建的进程的ID总是前一个被创建的进程的ID递增的结果。进程ID也是可以被重复使用的。当进程ID已达到其最大限值时，内核会从头开始查找已被闲置的进程ID并使用最先找到的那一个作为新进程的ID。另外，进程描述符中还会包含当前进程的父进程的ID（常被称为PPID）。

在Go语言中，我们可以使用标准库代码包os提供的API来查看当前进程的PID和PPID，像这样：

```
pid := os.Getpid()
ppid := os.Getppid()
```

注意，PID并不传达与进程有关的任何信息。它只是一个用来唯一标识进程的数字而已。进程的属性信息只被包含在内核中的、与PID对应的进程描述符中。而PPID在实际操作中也没有太多用处。不过它确实体现了两个进程之间的亲缘关系。我们可以利用这一点做一些事情。比如，顺藤摸瓜地查找守护进程的踪迹。

进程ID对内核以外的程序非常有用。内核可以高效地把进程ID转换成对应进程的描述符。我们可以shell命令kill终止某个进程ID所对应的进程，还可以通过进程ID找到对应的进程并向它发送信号。这在本节的后面会讲述到。

4. 进程的状态

在Linux操作系统中，每个进程在不同时刻都可能会有不同的状态。这些进程可能的状态共有6个，分别是：可运行状态、可中断的睡眠状态、不可中断的睡眠状态、暂停状态或跟踪状态、僵尸状态和退出状态。下面我们分别对这几种状态进行简要的说明。

- ❑ 可运行状态（TASK_RUNNING，简称为R）：如果一个进程处在该状态，那么说明它将要、立刻或正在CPU上运行。运行的时机是不确定的。这会由进程调度器来决定。
- ❑ 可中断的睡眠状态（TASK_INTERRUPTIBLE，简称为S）：当进程正在等待某个事件（比如网络连接或信号灯）的发生时会进入此状态。这样的进程会被放入对应事件的等待队列中。当事件发生时，对应的等待队列中的一个或多个进程就会被唤醒。
- ❑ 不可中断的睡眠状态（TASK_UNINTERRUPTIBLE，简称为D）：此种状态与可中断的睡眠状态的唯一区别就是它是不可被打断的。这意味着处在此种状态的进程不会对任何信号作出响应。更确切地讲，发送给处于不可中断状态的进程的信号直到该进程从此状态转出才会被传递过去。进程处于此种状态通常是由于在等待一个特殊的事件。比如在等待同步的I/O操作（磁盘I/O等）的完成。I/O是Input/Output的缩写，在这里可以理解为对

输入输出信息的处理。

- 暂停状态或跟踪状态 (TASK_STOPPED或TASK_TRACED, 简称为T): 向进程发送SIGSTOP信号就会使该进程转入暂停状态, 除非该进程正处于不可中断的睡眠状态。向正处于暂停状态的进程发送SIGCONT信号会使该进程转向可运行状态。处于被跟踪状态的进程会暂停并等待跟踪它的进程对它进行操作。例如, 我们使用调试工具GDB在某个程序中设置一个断点, 而后对应的进程在运行过程中会在断点处停下来并等待被操作。这时, 此进程就处于跟踪状态。跟踪状态与暂停状态非常类似。但是, 向处于跟踪状态的进程发送SIGCONT信号并不能使它被恢复。只有当调试进程进行了相应的系统调用或者退出之后, 它才能够被恢复。
- 僵尸状态 (TASK_DEAD-EXIT_ZOMBIE, 简称为Z): 处于此状态的进程即将要结束。该进程占用的绝大多数资源也都被回收。不过还有一些信息未被删除, 比如退出码以及一些统计信息。保留这些信息是考虑到该进程的父进程可能需要它们。由于此时的进程主体已经被删除而只留下了一个空壳, 故此状态常被称为僵尸状态。
- 退出状态 (TASK_DEAD-EXIT_DEAD, 简称为X): 在进程退出的过程中, 有可能连退出码和统计信息都不需要被保留。造成这种情况的原因可能是显式地让该进程的父进程忽略掉SIGCHLD信号 (当一个进程消亡的时候, 内核会为其父进程发送一个SIGCHLD信号以告知此情况), 也可能是该进程已经被分离。分离的含义是让子进程和父进程分别独立运行。分离后的子程序将不会再使用和执行与父进程共享的代码段中的指令, 而是加载并运行一个全新的程序 (我们讲“进程的衍生”的时候提到过)。在这些情况下, 该进程在退出的时候就不会转入僵尸状态, 而会直接转入退出状态。处于退出状态的进程会立即被干净利落地结束掉。它所占用的系统资源也会被操作系统自动回收。

进程在其生命周期内可能会产生一系列的状态变化。简单地说, 进程的状态只会在可运行状态和非可运行状态之间转换。图6-1展示了一般情况下的进程状态转换。

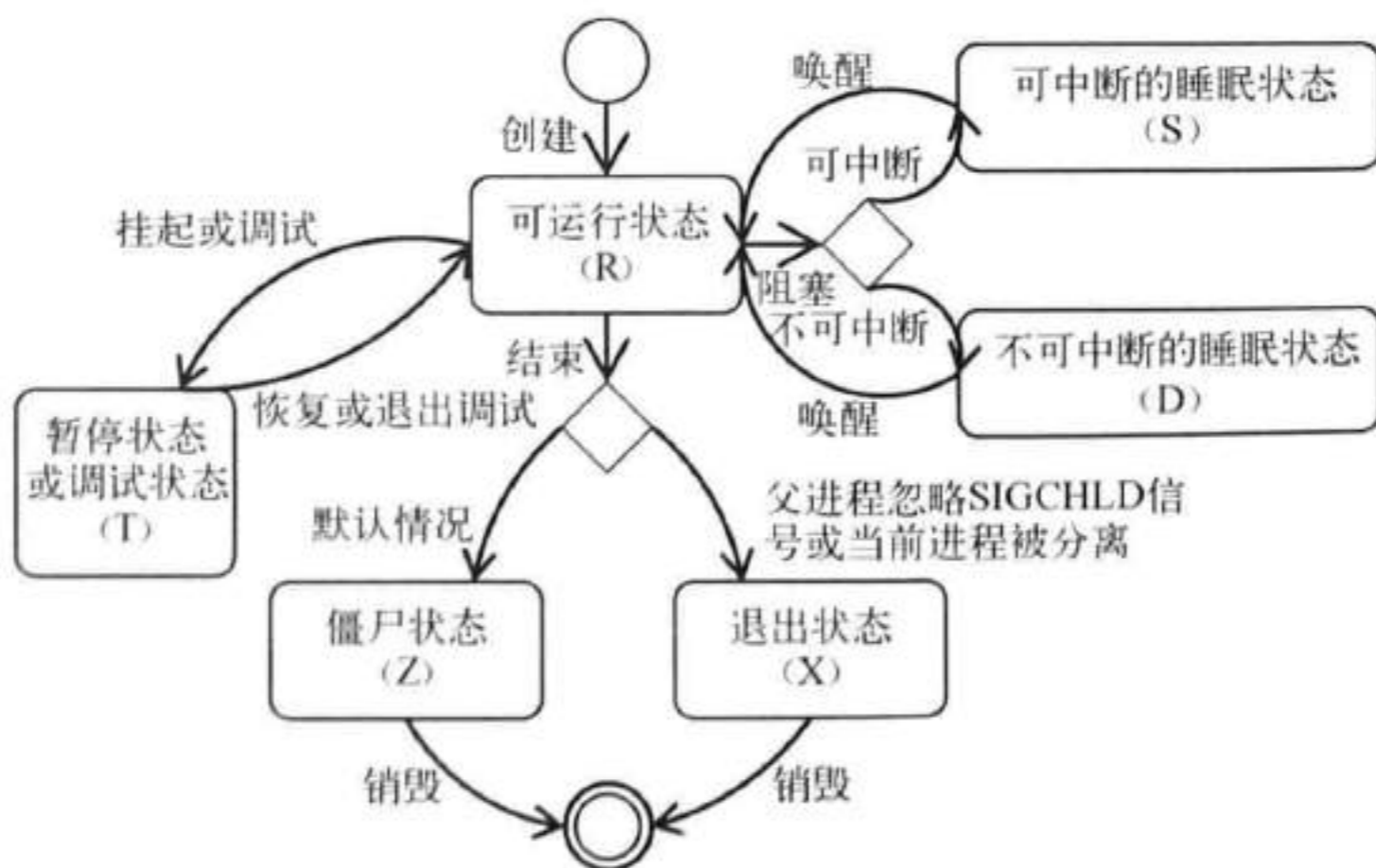


图6-1 Linux操作系统进程的状态转换

虽然暂停状态和调试状态极为相似，但是也可以把它们看成两个不同的状态。因此，Linux操作系统中的进程有7种可能的状态的说法也是正确的。

5. 进程的空间

一个用户进程（或者说我们的程序的执行实例）总会生存于用户空间中。这些进程可以做很多事，但是却不能与其所在的计算机的硬件进行交互。内核可以与硬件交互，但是它却生存在内核空间中。用户进程无法直接访问内核空间。用户空间和内核空间体现了Linux操作系统对物理内存的划分。换句话说，这两个空间指的都是操作系统在内存上划分出的一个范围。它们共同瓜分了操作系统能够支配的内存区域，如图6-2所示。



图6-2 Linux操作系统对虚拟内存的划分

内存区域中的每一个单元都是有地址的。这些地址是由指针来标识和定位的。通过指针来寻找内存单元的操作也被称为内存寻址。指针是一个正整数，由若干个二进制位表示。具体的二进制位的数量由计算机（更确切地说是CPU）的字长所决定。因此，在32位计算机中可以有效标识2的32次方个内存单元，而在64位计算机中可以有效标识2的64次方个内存单元，正如图6-2所示。

这里所说的地址并非物理内存中的真实地址，它们被称为虚拟地址。而由虚拟地址来标识的内存区域又被称为虚拟地址空间，有时也被称为虚拟内存。回顾图6-2，用户空间虚拟地址的范围是从0到`TASK_SIZE`，而内核空间则占据了其余虚拟地址所代表的空间。`TASK_SIZE`相当于这两个空间的分界线。它实际上是一个特定的常数。它的值由所在的计算机的体系结构决定。在不同的计算机体系结构下，`TASK_SIZE`的值可能是不同的。注意，虚拟内存的最大容量与实际可用的物理内存的大小是无关的。内核和CPU会负责维护虚拟内存与物理内存之间的映射关系。

内核会为每个用户进程分配的是虚拟内存而不是物理内存。每个用户进程被分配到的虚拟内存总是在用户空间中的，而内核空间被留给内核专用。另外，每个用户进程都会认为分配给它的虚拟内存就是整个用户空间。一个用户进程不可能操纵另一个用户进程的虚拟内存，因为后者的虚拟内存对于前者来说是不可见的。换句话说，这些进程的虚拟内存几乎是彼此独立、互不干扰的。这是由于它们基本上被映射到了不同物理内存之上。

内核会把进程的虚拟内存划分为若干页（page）。而物理内存单元的划分由CPU负责。一个

物理内存单元被称为一个页框（page frame）。不同的进程的大多数页都会与不同的页框相对应，如图6-3所示。

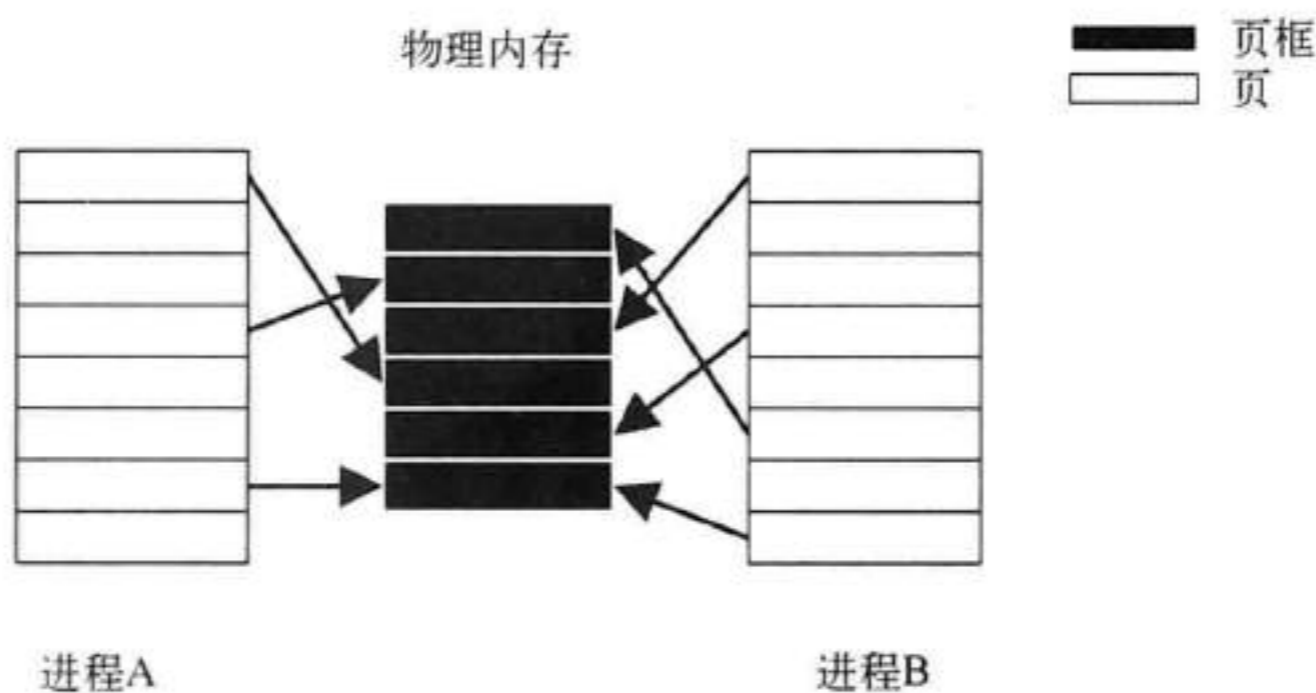


图6-3 进程的虚拟内存与物理内存

图中的进程A和进程B的大多数页都分别与物理内存中的不同页框相对应。但是，进程A的页7与进程B的页8共享了同一个页框（即最下面的一个页框）。这种页框的共享是被允许的。实际上，这正是作为IPC方法之一的共享内存区的基础。另外，我们看到，不论进程A还是进程B都有一些页没有与任何一个页框对应。这也是有可能的。这也许是由于该页没有数据或者数据还不需要被使用，也许是该页已经被换出至磁盘（确切地说，是Linux文件系统中的swap分区）中。

6. 系统调用

我们在前面说过，用户进程生存在用户空间中且无法直接操纵计算机的硬件，但是在内核空间中的内核却可以做到。用户进程无法直接访问内核空间，也无法随意指使内核去做它能做到的一些事。但是为了使用户进程能够使用操作系统更底层的功能，内核会暴露出一些接口以供它们使用。这些接口是用户进程使用内核功能（包括操纵计算机硬件）的唯一手段，也是用户空间和内核空间之间的一座桥梁。用户进程使用这些接口的行为被称为系统调用，但在很多时候“系统调用”这个词也指代内核提供的这些接口。注意，虽然系统调用也是由函数代表的，但它与普通的函数是有明显的区别的。系统调用是向内核空间发出的一个明确的请求，而普通的函数只是定义了如何获取一个给定的服务。更重要的是，系统调用会导致内核空间中的数据的存取和指令的执行，而普通函数却只能在用户空间中有所作为。当然，如果在一个函数的函数体中包含了系统调用，那么它的执行也将涉及对内核空间的访问。但是这种访问仍然是通过函数体内的系统调用来进行的。另外，系统调用是内核的一部分，而普通的函数却不是。

说到系统调用就不得不提及另外一对概念——内核态和用户态。为了保证操作系统的稳定和安全，内核依据由CPU提供的、可以让进程驻留的特权级别建立了两个特权状态。它们就是内核态和用户态。在大部分时间里CPU都处于用户态。这时CPU只能对用户空间进行访问。换言之，CPU在用户态下运行的用户进程是不能与内核接触的。当用户进程发出一个系统调用的时候，内核会把CPU从用户态切换到内核态，而后会让CPU执行对应的内核函数。CPU在内核态下是有限访问内核空间的。这就相当于使用户进程通过系统调用使用到了内核提供的功能。当内核函数

被执行完毕后，内核会把CPU从内核态切换回用户态，并把执行结果返回给用户进程。图6-4大致地描述了系统调用过程中的CPU状态切换和流程控制。

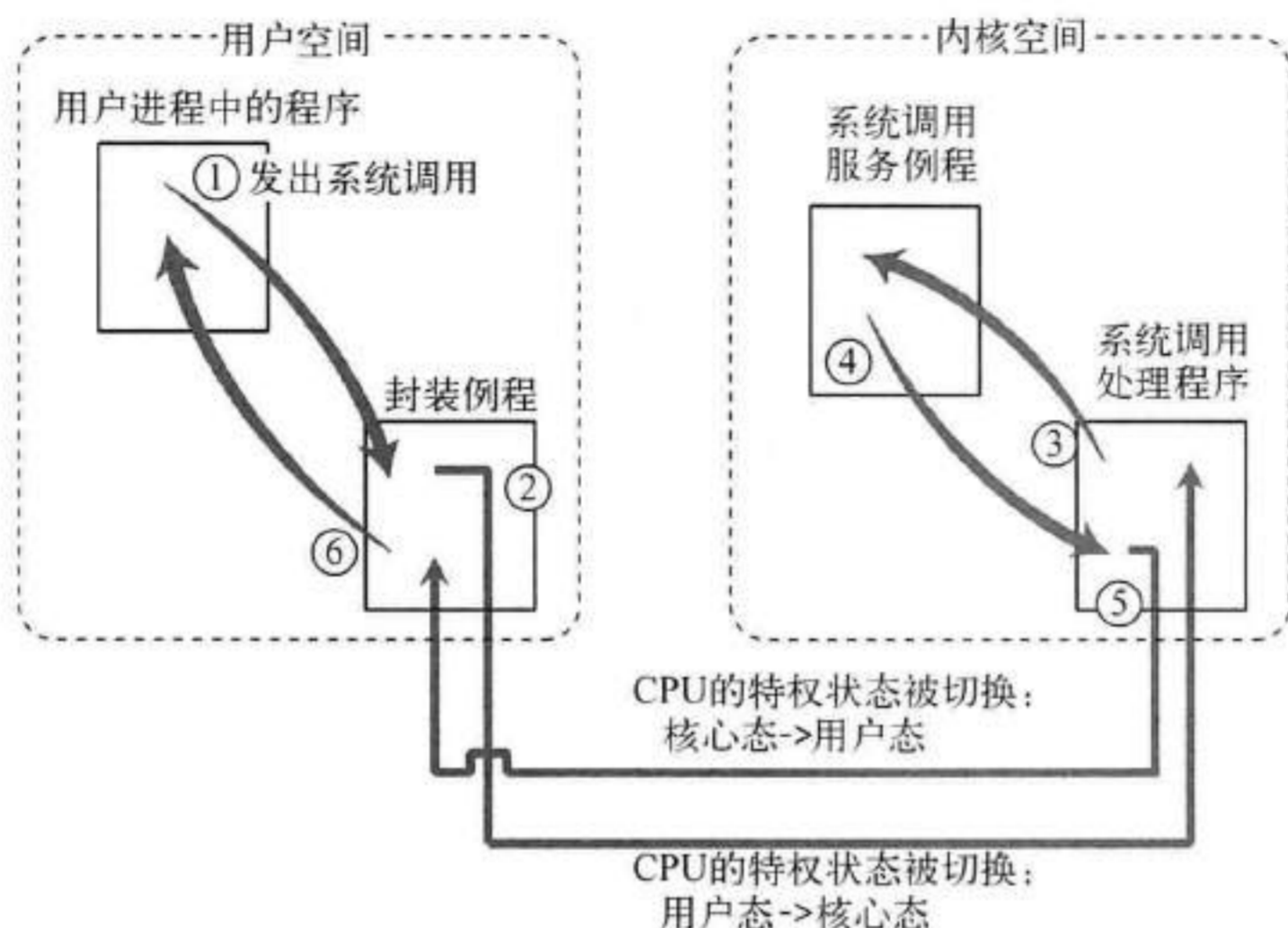


图6-4 关于系统调用过程的示意图

这幅示意图描绘的流程比我们刚刚叙述得更加细致一些。从中我们可以看出一个系统调用从开始到结束的较完整流程。其中，封装例程与系统调用是一一对应的。实际上，它就是我们所说的内核暴露给用户进程的接口。另外，我们可以把图中的系统调用处理程序和系统调用服务例程看作是内核为了响应用户进程的系统调用而执行的一系列函数。我们在上面的叙述中把它们统称为内核函数。最后，再强调一下，只有当CPU被切换至核心态之后才可以执行内核空间中的函数，而在内核函数被执行完毕后，CPU状态也会被及时地切换回用户态。

7. 进程的切换和调度

与其他分时操作系统一样，Linux操作系统也可以凭借CPU的威力，快速地在多个进程之间进行切换（也被称为进程间的上下文切换），以产生多个进程在同时运行的假象。每个进程也都会认为自己独占了CPU。这就是多任务操作系统这个称谓的由来。不过，无论切换速度如何，在同一时刻正在运行的进程也仅仅会有一个。当然，切换CPU正在运行的进程是需要付出代价的。例如，内核此刻要换下正在CPU上运行的进程A，并让CPU开始运行进程B。在换下进程A之前，内核必须要及时保存进程A的运行状态。另一方面，假设进程B不是第一次被运行，那么在让进程B被重新运行之前，内核必须要保证已经依据之前保存的相关信息，把进程B恢复到之前被换下时的运行状态。当然，需要运行的进程往往不只两个。但是在处理流程上是相通的。我们把这种在进程换出换入期间所必须要做的任务统称为进程切换。进程切换主要是由内核来完成的。除了进程切换之外，为了使各个生存着的进程都有被运行的机会、让它们共享CPU，内核还要考虑把哪一个进程应该作为下一个被运行的进程、应该在哪一时刻进行切换，以及被换下的进程需要在哪一时刻再被换上，等等。解决类似问题的方案和任务被统称为进程调度。

进程切换和进程调度是程序并发执行的基础。没有它们，程序的并发执行就无从谈起，我们所说的并发编程（单个计算机环境下的并发编程）也就没有任何现实意义了。不过，我们并不打算对它们进行详细的说明。因为这会非常耗费篇幅，并且也偏离了我们的主题。但是，我们确实应该关注随之而来的一些问题及其解决方案。请读者接着往下看。

6.2.2 关于同步

内核对进程的合理切换和调度使得多个进程可以被有条不紊地并发运行。在很多时候，多个进程之间需要相互配合并合作完成一个任务。这就需要有进程间通讯机制（IPC）的支持。不过在详细讲解各种IPC方法之前，我们先来了解一下进程之间在通讯过程中可能发生的干扰。这种干扰主要集中在有共享数据的情况下。不论是多CPU、多进程还是我们之后要提到的多线程，只要它们之前存在数据共享，就一定会牵扯到同步问题。不管被共享的数据是被存储在内存中、磁盘上，还是其他被共用的数据介质上，都会是这样。所以，我们接下来要讨论的这个问题是具有普遍意义的。其中的一些概念和论点可以适用于很多场景。

首先，我们考虑一个看似简单的应用场景——计数器。这个计数器由进程A创建并与进程B共享。进程A和进程B实际上执行了相同的程序。这个程序的任务是把符合某些条件的数据从数据库迁移到磁盘上。程序总是按照固定顺序从数据库中查询数据，并使用计数器记录的已被查询的数据的最大行号作为依据。下面是程序的具体任务步骤。

- (1) 读取计数器的值。
- (2) 从数据库中查询数据。如果我们用 c 来代表计数器的值，那么查询的范围就是行号在 $[c, c+100000)$ 范围内的数据。也就是说，每次查询10万条数据。
- (3) 遍历并筛选出符合条件的数据，并组成新的数据集合。
- (4) 将新数据集合存储到指定目录的文件中。该文件的名称总是有一致的主名称data并以递增的序号为后缀。例如，data1、data2，等等。
- (5) 把计数器的值加100000。也就是说，计数器的新值就是下次要查询的数据的首行行号。
- (6) 检查数据是否已被全部读完。如果是则直接退出，否则跳转回1。

进程A和进程B会被并发地运行。它们会各自循环往复地迁移它们认为的下一个数据集合，直到数据被全部迁移完毕。

这会出现问题吗？答案是肯定的。我们已经知道，每个进程在每次对指定数据集合的迁移过程中都需要完成上述6个步骤。由于内核会对各个进程切换和调度，因此不能保证进程在迁移每个数据集合的过程中都不被打断。也就是说，进程A和进程B的运行是互相穿插在一起的。这种穿插或者说切换的粒度会比我们上面罗列的步骤的粒度还要小很多。不过，为了清晰，我们假设进程切换的粒度与以上步骤的粒度相同。下面，我们在这个假设的基础上叙述一种可能的进程调度过程。

- (1) 内核使CPU运行进程A。
- (2) 进程A读取计数器的值1，并依此查询并筛选了数据，得到了新的数据集合。
- (3) 内核认为进程A已经运行了足够长的时间，所以它把进程A换下并让CPU开始运行进程B。

- (4) 进程B读取计数器的值1，并依此查询并筛选了数据，得到了新的数据集合。注意，这个数据集合与进程A刚刚得到的那个数据集合完全一样。
- (5) 进程B把得到的数据集合写入名称为data1的文件，并在写入完成后关闭文件。
- (6) 内核把进程B换下并让CPU开始运行进程A。
- (7) 进程A把得到的数据集合写入名称为data1的文件，并在写入完成后关闭文件。
- (8) 进程A把计数器的值更新为100001。
- (9) 内核把进程A换下并让CPU开始运行进程B。
- (10) 进程B把计数器的值更新为100001。

上述进程调度过程如图6-5所示。

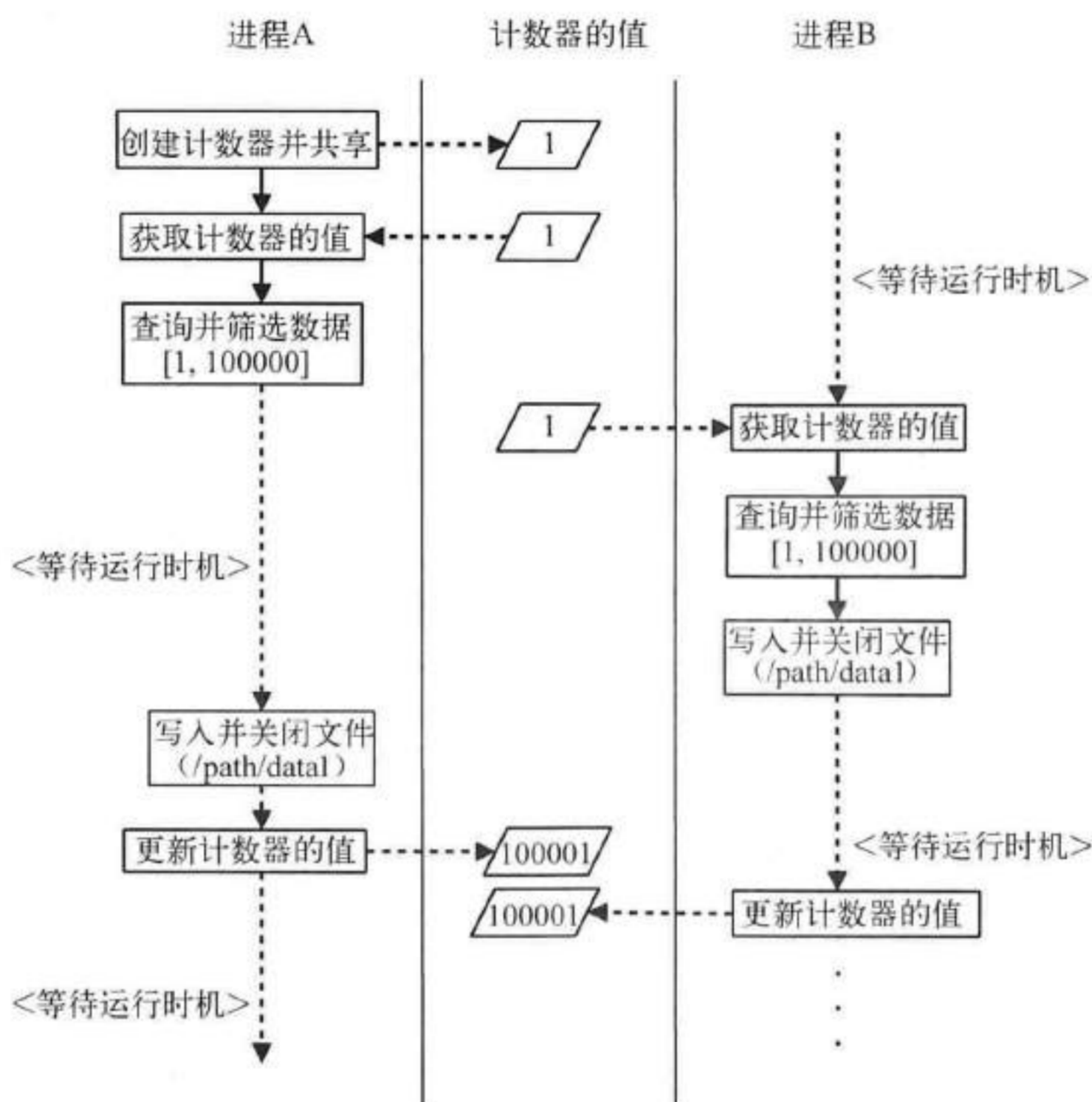


图6-5 初始流程下的进程调度过程

好了，到这里我们已经看出了一个很明显的问题，即进程A和进程B在做重复的事。更大的问题是，它们造成了双倍的资源消耗，并导致了事倍功半的结果。这是由于同一个进程对计数器的值的读取和更新之间的时间跨度太大了，以至于计数器只起到了任务进度记录的作用，而没有起到在两个进程之间协调的作用。既然这是由于时间跨度大的两个操作引起的，那么我们就把这个时间跨度缩小到最小，看看会不会解决此问题。我们把前面所说的程序的具体任务步骤中的第5步上移至第2步。也就是说，我们让一个进程在读取计数器的值之后马上更新它。这样，之后CPU

运行的另一个进程就会去查询并处理后面的数据行了。

但是，如此真的能够彻底的解决上述问题吗？非常遗憾，答案是不能。请想象一下这样的进程调度过程，如果内核在进程A已经读取却还未更新计数器的值的时候让CPU转而运行进程B，会发生什么？请看图6-6。进程B在得到计数器的值1之后把该值更新为了100001。但是注意，它仍然会去做进程A即将要去做的事（查询行号在 $[c, c+100000)$ 范围内的数据、筛选并保存到文件data1）。当进程A重新获得运行时机的的时候也依旧会从查询行号在 $[c, c+100000)$ 范围内的数据开始。为了突出重点，我们省略掉了所有的“写入并关闭文件”的步骤。

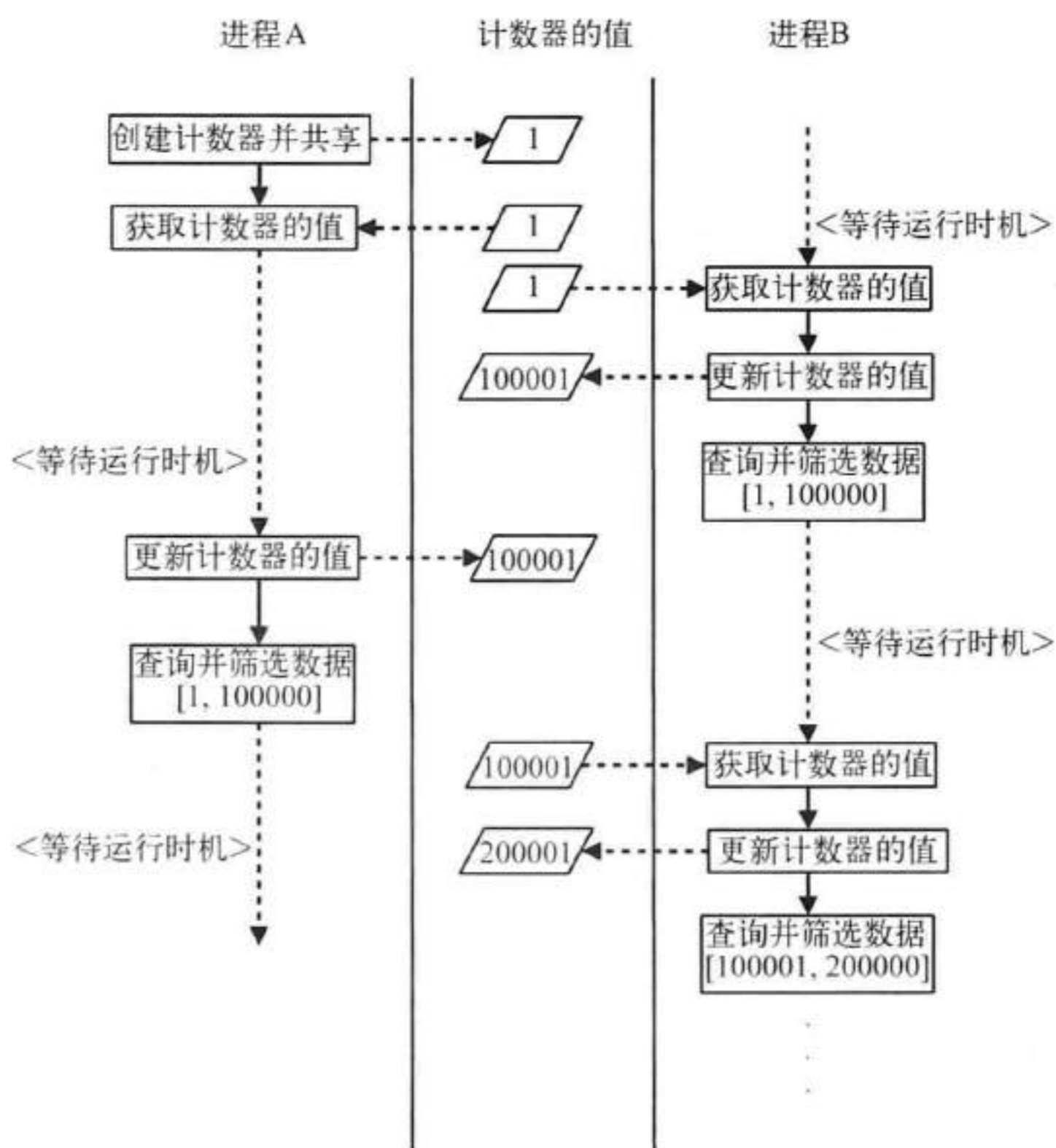


图6-6 改进流程下的进程调度过程

显然，从第二个进程调度过程来看，我们对流程的更改并没有起到什么作用。上述问题仍会出现。内核是无法理解程序中各个语句的语义的，因此也就无法保证总是会在合适的时机切换进程。但是，新的流程确实要比之前的流程好得多。因为它大大减小了上述问题出现的概率。

做过此类工作的编程人员可能会提出一个新的解决方案，那就是在更新计数器的值的那一刻之前再去读取一下计数器的值，以保证更新不会重复。但是我们很难实现这种重新进行操作的方法，因为我们无法判断在进程获取计数器值和更新计数器值期间该进程是否被切换过。并且，即使这种方法可以被实现，它也不会解决现有问题。更糟糕的是，它会使事情变得更加复杂。一个

可能的进程调度过程如图6-7所示。

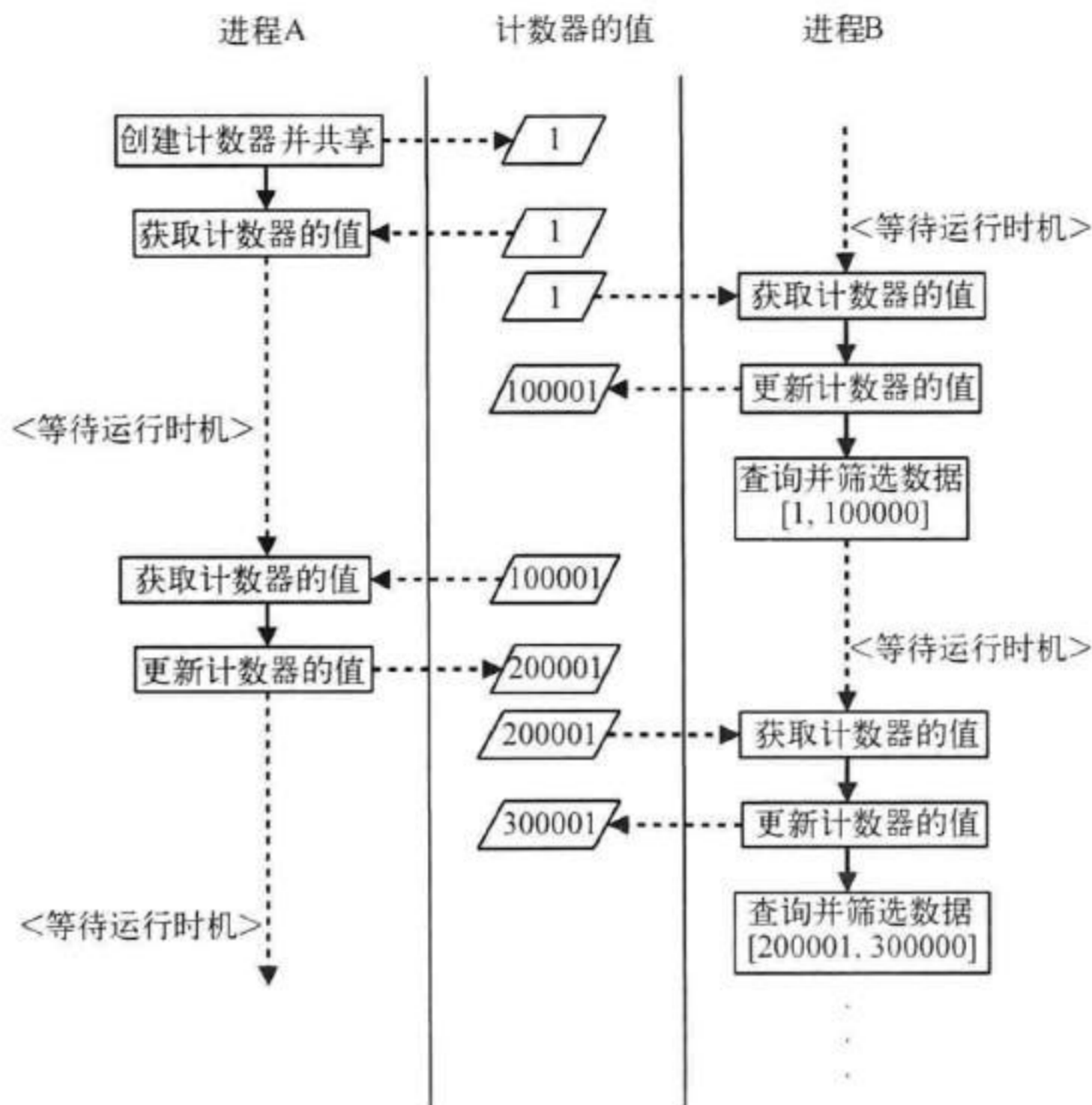


图6-7 更糟糕的流程下的进程调度过程

从图6-7中可以看出，重复的计数器值获取操作导致了行号在[100001, 200000]范围内的数据没有被处理。这个问题比重复处理数据的那个问题更为严重。

那到底怎样才是正确的解决方法呢？在揭晓答案之前，让我们先来熟悉一下相关的概念。前述的3个版本的程序流程都存在不同程度的问题。其中，第一个版本的程序流程的问题最为明显。它不但会导致计数器值的重复设置以及数据的重复处理，可能还会使文件中的数据造成错乱。想象一下，如果两个进程同时（或者说交错地）把大量数据写入到同一个文件中，会造成怎样的后果。

当几个进程同时对同一个资源进行访问的时候，就很可能造成互相的干扰。这种互相干扰通常被称为竞态条件（race condition）。竞态条件通常在编码和测试过程中难以被察觉到。我们在前面列举的可能的进程调度过程都属于特例。它们的发生可能不会那么频繁。但是，一旦发生就绝对会造成程序运行结果的错误。更为重要的是，排查这种错误是比较困难的。正因为它们的发生并不频繁，所以场景重现变得非常不容易。这需要满足一系列特定的条件才有可能做到。找到并消除一个竞态条件可能会让程序运维人员耗上几个小时甚至几天的时间，尤其是在对底层的运行机制不了解的情况下。

相比于其他现代编程语言，Go语言的并发编程模型更加成熟和先进。它的目标就在于大幅减少编程人员产生竞态条件的可能。它尽可能多地把复杂的并发处理逻辑埋藏在它的运行时系统之下，让编程人员能够腾出精力和时间去解决真正的业务问题。关于Go语言的并发编程模型的详细介绍我们放在了本章的最后。

现在回到本小节的主题。造成竞态条件的根本原因在于进程在进行某些操作的时候被中断了。虽然进程在被再次运行的时候其状态会恢复如初，但是外界环境很可能已经在这极短的时间内被改变甚至面目全非了，就像我们改进后的第二个版本的程序流程那样。从访问计数器的方面看，它几乎已经能够成为有效的解决方案了。但就是由于应用程序对进程调度的不可控性使得竞态条件仍然可能发生。反过来说，如果能够保证计数器的值的获取并更新是一个原子操作的话，那么竞态条件就不会发生。更具体地，如果进程A在获取并更新计数器的值的过程中不被中断，那么进程B就会如我们所愿地去处理行号在[100001, 200000]范围内的数据了。

在这里，我们把执行过程中不能被中断的操作称为原子操作（atomic operation），而把只能被串行化的访问或执行的某个资源或某段代码称为临界区（critical section）。在第二个版本的程序流程中，每个进程对计数器的获取和更新操作都应该被看作是一个单一的、不应该被中断的操作。因此，它们应该组成一个原子操作。并且，这两个操作应该被串行化的执行，即在一个进程对计数器的值的获取并更新操作还未完成之前其他进程不得介入。因此，体现计数器的获取操作和更新操作的代码应该共同形成为一个临界区。顺便提一句，所有的系统调用都属于原子操作。我们不用担心它们的执行会被中断。

我们可以看出，原子操作和临界区这两个概念看起来有些相似。但是它们有一个明显的不同。原子操作是不能被中断的，临界区对是否可以被中断却没有强制的规定。只要保证一个访问者在临界区中的时候其他访问者不会被放进来就可以了。这也意味着它们的强度是不同的。

原子操作必须由一个单一的汇编指令代表，并且需要得到芯片级别的支持。当今的CPU中都提供了对原子操作的支持。即使是在多核CPU或多CPU的计算机系统中，原子操作也是可以被保证的。这使得原子操作能够做到绝对的并发安全，并且比其他同步机制要快很多。不过，读者可能会考虑这样一个问题：如果一个原子操作的执行总是无法结束而我们又无法中断它，那该怎么办？实际上，这也是内核只提供针对二进制位和整数的原子操作的原因。原子操作只适合细粒度的简单操作，就像前面讲述的对计数器的值的获取并更新操作那样。Go语言也在CPU和各个操作系统的底层支撑之上提供了对原子操作的支持。它们由标准库代码包sync/atomic中的一些函数代表。我们会在第8章详细说明它们。

相比之下，让要求被串行化执行的若干代码形成临界区的这种做法更加通用。保证只有一个进程或线程在临界区之内的这种做法有一个官方称谓——互斥（mutual exclusion，简称mutex）。实现互斥的方法必须确保排他原则（exclusion principle），并且这种保证不能依赖于任何计算机硬件（包括CPU）。也就是说，互斥方法必须有效且通用。时至今日，互斥方法的实现方式非常多样，有的只停留在理论层面，而有的已经成为了各个操作系统的标配。作为IPC方法之一的信号灯就属于后者。在Go语言的sync代码包中也包含了几个提供了互斥方法的类型。我们同样会在第8章讲解它们。

好了，我们对同步的介绍暂时就到这里。不过，在后面讲解多线程编程的时候，我们还会重返这一主题。

6.2.3 管道

管道（pipe）是一种半双工的（或者说单向的）通讯方式。它只能被用于父进程与子进程以及同祖先的子进程之间的通讯。例如，我们在使用shell命令的时候常常会用到管道：

```
hc@ubt:~$ ps aux | grep go
```

shell为每个命令都创建一个进程，然后把左边的命令的标准输出用管道与右边的命令的标准输入连接起来。管道的优点在于它的简单，而缺点则是只能单向通讯以及对通讯双方关系上的严格限制。

对于管道，Go语言是支持的。通过标准库代码包os/exec中的API，我们可以执行操作系统命令并在此之上建立管道。我们可以像这样创建一个exec.Cmd类型的值：

```
cmd0 := exec.Command("echo", "-n", "My first command from golang.")
```

变量cmd0的值与操作系统命令

```
echo -n "My first command from golang."
```

是对应的。在exec.Cmd类型之上有一个名为Start的方法。我们可以使用这个方法启动一个操作系统命令，像这样：

```
if err := cmd0.Start(); err != nil {
    fmt.Printf("Error: The command No.0 can not be startup: %s\n", err)
    return
}
```

但是为了创建一个能够获取此命令的输出的管道，我们需要在上面这条if语句之前加入这样几条语句：

```
stdout0, err := cmd0.StdoutPipe()
if err != nil {
    fmt.Printf("Error: Can not obtain the stdout pipe for command No.0: %s\n", err)
    return
}
```

变量cmd0的值的StdoutPipe方法会返回一个输出管道。在这里，我们把代表这个输出管道的值赋给了变量stdout0。它的类型是io.ReadCloser。这是一个接口类型并扩展了接口类型io.Reader。这样，在我们启动该命令之后就可以调用stdout0的值的Read方法来获取这个命令的输出：

```
output0 := make([]byte, 30)
n, err := stdout0.Read(output0)
if err != nil {
    fmt.Printf("Error: Can not read data from the pipe: %s\n", err)
    return
}
fmt.Printf("%s\n", output0[:n])
```

方法Read会把读出的输出数据存入调用方传递给它的字节切片（这里是output0的值）中并返回一个int类型值和一个error类型值。如果命令的输出小于output0的值的长度，那么变量n的值就代表了命令实际输出的字节的数量。否则，n的值就等于output0的值的长度。后一种情况常常意味着我们并没有完全读出输出管道中的数据。这时，我们通常需要再去读取一次或多次（可以使用for语句进行循环读取）。如果输出管道中再没有可以被读取的数据了，那么Read方法返回的第二个结果值就会是变量io.EOF的值。我们可以依此来判断数据是否已经被读完。像这样：

```
var outputBuf0 bytes.Buffer
for {
    tempOutput := make([]byte, 5)
    n, err := stdout0.Read(tempOutput)
    if err != nil {
        if err == io.EOF {
            break
        } else {
            fmt.Printf("Error: Can not read data from the pipe: %s\n", err)
            return
        }
    }
    if n > 0 {
        outputBuf0.Write(tempOutput[:n])
    }
}
fmt.Printf("%s\n", outputBuf0.String())
```

为了达到效果，我们故意将作为Read方法参数的字节切片的长度设置得很小。另外，为了收集每次迭代读到的输出内容，我们这些内容依次存放入到一个缓冲区中，并在最后将此缓冲区中的内容打印出来。

不过，一个更加方便的方法是，一开始就使用带缓冲的读取器（以下简称缓冲读取器）从输出管道中读取数据，像这样：

```
outputBuf0 := bufio.NewReader(stdout0)
output0, _, err := outputBuf0.ReadLine()
if err != nil {
    fmt.Printf("Error: Can not read data from the pipe: %s\n", err)
    return
}
fmt.Printf("%s\n", string(output0))
```

由于stdout0的值也是io.Reader类型的，所以我们可以把它作为bufio.NewReader函数的参数。这个函数会返回一个bufio.Reader类型的值。它就是我们刚刚提到的缓冲读取器。在默认情况下，该读取器会携带一个长度为4096的缓冲区。缓冲区的长度代表了我们一次可以读取的字节的最大数量。由于cmd0代表的命令只会输出一行内容，所以我们可以直接用outputBuf0的ReadLine方法来读取它。这个方法的第二个bool类型的结果值表明了当前行是否还未被读完。如果为false，我们依然可以利用for语句来读出剩余的数据。不过在这里我们并不需要这样做，所以我们将第二个结果赋给了空标识符“_”。另外，我们总是需要先检查err的值，看看是否有错误发生。如果没有任何错误，那么我们就可以放心地处理output0的值了。

使用带缓冲区的读取器的好处是我们可以非常方便和灵活地读取需要的内容,而不是只能先把所有内容都读出来再做处理。读者可以考虑一下,如果我们不使用缓冲读取器,那么从`stdout0`的值中读取一行内容的代码应该怎样编写。显然,这省去了我们自己的一些工作量。缓冲读取器提供的功能远比我们在这里展示得强大得多。请详见`bufio`代码包的文档。

好了,言归正传。我们已经知道,管道是一个单向数据通道。它可以把一个命令的输出作为另一个命令的输入。当然,我们也可以使用Go语言代码做到这一点。假设我们有如下两个`exec.Cmd`类型值:

```
cmd1 := exec.Command("ps", "aux")
cmd2 := exec.Command("grep", "apipe")
```

现在,我们在`cmd1`代表的命令之上建立一个输出管道,然后启动这个命令:

```
stdout1, err := cmd1.StdoutPipe()
if err != nil {
    fmt.Printf("Error: Can not obtain the stdout pipe for command: %s\n", err)
    return
}
if err := cmd1.Start(); err != nil {
    fmt.Printf("Error: The command can not be startup: %s\n", err)
    return
}
```

这与我们操纵`cmd0`时的代码几乎相同。在这之后,我们通过`StdinPipe`方法在`cmd2`之上建立一个输入管道,并把与`cmd1`连接的输出管道中的数据全部写入到这个输入管道中:

```
outputBuf1 := bufio.NewReader(stdout1)
stdin2, err := cmd2.StdinPipe()
if err != nil {
    fmt.Printf("Error: Can not obtain the stdin pipe for command: %s\n", err)
    return
}
outputBuf1.WriteTo(stdin2)
```

变量`cmd2`的值的`StdinPipe`方法会返回两个结果值。第一个结果值就是与该命令连接的输入管道。它是一个`io.WriteCloser`接口类型的值。这个接口类型扩展了`io.Writer`接口类型。正因为如此,它可以被作为缓冲读取器`outputBuf1`的`WriteTo`方法的参数。这个方法会把所属值中缓存的数据全部写入到参数值代表的写入器中。这样就等于把第一个命令的输出内容通过管道传递给了第二个命令。

不过这还不算完。我们还需要启动`cmd2`并关闭与它连接的输入管道,以完成数据的传递。另外,为了获取到`cmd2`的输出结果,我们还需要附加两行代码。请看下面的示例:

```
var outputBuf2 bytes.Buffer
cmd2.Stdout = &outputBuf2
if err := cmd2.Start(); err != nil {
    fmt.Printf("Error: The command can not be startup: %s\n", err)
    return
}
err = stdin2.Close()
if err != nil {
```

```
    fmt.Printf("Error: Can not close the stdio pipe: %s\n", err)
    return
}
```

我们初始化了一个缓冲区outputBuf2，并把它赋给了cmd2的Stdout字段。这样，命令cmd2被启动后的所有输出内容就都会被写入到该缓冲区中。之后，我们启动了cmd2，并关闭了stdin2。

为了获取到cmd2的所有输出内容，我们需要等到它运行结束后，再去查看缓冲区outputBuf2中的内容。因此，我们还需要调用cmd2的Wait方法。像这样：

```
if err := cmd2.Wait(); err != nil {
    fmt.Printf("Error: Can not wait for the command: %s\n", err)
    return
}
```

方法Wait会一直阻塞到其所属的命令完全运行结束为止。这样，我们再去处理outputBuf2中的内容就完全没有问题了。

这个基于cmd1和cmd2的示例模拟出了操作系统命令

```
ps aux | grep apipe
```

的执行效果。不过，cmd2的输出会与直接运行这个操作系统命令得到的输出有所不同。因为该示例程序相当于在自身运行的过程当中又运行了上面的这个操作系统命令。

我把上面这些关于管道的示例代码都放到了goc2p项目的multiproc/apipe代码包中的命令源码文件中。读者可以使用go run命令运行其中的命令源码文件，并比较和分析刚才所说的在最终输出上的不同。

我们上面所讲的管道也被叫作匿名管道，与此相对的是命名管道（named pipe）。与匿名管道不同的是，任何进程都可以通过命名管道交换数据。实际上，命名管道以文件的形式存在于文件系统中。使用它的方法与使用文件很类似。Linux操作系统支持使用shell命令创建和使用命名管道。例如：

```
hc@ubt:~$ mkfifo -m 644 myfifo1
hc@ubt:~$ tee dst.log < myfifo1 &
[1] 3485
hc@ubt:~$ cat src.log > myfifo1
```

在上面的示例中，我们先使用命令mkfifo在当前目录创建了一个命名管道myfifo1，然后又使用这个命名管道和命令tee把src.log文件中的内容写到了dst.log文件中。为了简单，我们只是使用命名管道搬运了数据。实际上，在此基础上我们可以实现诸如数据的过滤或转换，以及管道的多路复用等功能。注意，命名管道默认是阻塞式的。更具体地说，只有在对这个命名管道的读操作和写操作都已准备就绪之后，数据才会开始流转。相对于匿名管道，命名管道最大的优势就是通讯双方可以毫不相关。并且，我们可以使用它建立非线性的连接以实现数据的多路复用。但要注意，命名管道仍然是单向的。又由于我们可以在命名管道之上实现多路复用，所以有时候也需要考虑多个进程同时向命名管道写数据的情况下的操作原子性问题。

在Go语言标准库代码包os中包含了可以创建这种独立管道的API。创建一个命名管道的代码非常简单，如下：

```
reader, writer, err := os.Pipe()
```

函数Pipe会返回两个结果值。第一个结果值是代表了该管道输出端的*os.File类型值，因第二个结果是代表了该管道输入端的*os.File类型值。既然它们都是*os.File类型的，那么我们就可以在它们之上调用*os.File类型包含的所有方法。不过，os.Pipe方法返回的前两个结果值只是让我们用来传递数据的渠道而已。在底层，Go语言使用系统函数来创建管道，并把它的两端封装成两个*os.File类型的值。假设，有这样的两段代码：

```
n, err := writer.Write(input)
if err != nil {
    fmt.Printf("Error: Can not write data to the named pipe: %s\n", err)
}
fmt.Printf("Written %d byte(s). [file-based pipe]\n", n)
```

和

```
output := make([]byte, 100)
n, err := reader.Read(output)
if err != nil {
    fmt.Printf("Error: Can not read data from the named pipe: %s\n", err)
}
fmt.Printf("Read %d byte(s). [file-based pipe]\n", n)
```

如果它们是被并发运行的，那么我们在reader之上调用Read方法就可以按顺序获取到之前通过调用writer的Write方法写入的数据。为什么强调是并发运行？因为命名管道默认会在其中一端还未就绪的时候阻塞另一端的进程。Go语言在这里提供给我们的命名管道的行为特征也是如此。所以，如果我们顺序地执行这两段代码，那么程序肯定会被永远阻塞在语句

```
n, err := writer.Write(input)
```

或

```
n, err := reader.Read(output)
```

出现的地方。具体被阻塞在哪儿取决于调用表达式writer.Write(input)和reader.Read(output)哪一个先被求值。

另外，我们已经知道，管道都是单向的。因此，我们不能反过来使用reader或writer。也就是说，我们在reader之上调用Write方法或者在writer的Read方法之后获取的第二个结果值都将是一个非nil的error类型值。其中的信息会告诉我们，这样的访问是不被允许的。另外，不论我们在哪一方调用Close方法，都不会影响到另一方的读取或写入数据的操作。

实际上，我们在exec.Cmd类型值之上调用StdinPipe或StdoutPipe方法后得到的输入管道或输出管道也是通过os.Pipe函数生成的。只不过，在这两个方法内部又对刚刚生成的管道做了少许的附加处理。输入管道的输出端会在所属命令启动后就被立即关闭，而输入端则会在所属命令运行结束之后被关闭。而输出管道的两端的自动关闭的时机与前面刚好相反。不过要注意，有些命令会等到输入管道被关闭之后才结束运行。所以，在这种情况下，我们就需要在数据被读取之后尽早地手动关闭输入管道。在前面的示例中，我们已经有过类似的演示：

```
if err := cmd2.Start(); err != nil {
    // 省略若干条语句
```

```

}
err = stdin2.Close()
// 省略若干条语句
if err := cmd2.Wait(); err != nil {
    // 省略若干条语句
}

```

请读者在必要时依照上面这样的操作顺序。由于输出管道实际上也是由`os.Pipe`函数生成的，所以我们在使用某个`exec.Cmd`类型值上的输出管道的时候也需要有所注意。例如，我们不能在读完输出管道中的全部数据之前调用该值的`Wait`方法。又例如，只要我们建立了对应的输出管道就不能使用`Run`方法来启动该命令，而应该使用`Start`方法。

由于通过`os.Pipe`函数生成的管道在底层是由系统级别的管道来支持的，所以我们在使用它的时候，要注意操作系统对管道的限制。例如，匿名管道会在管道缓冲区被写满之后使写数据的进程阻塞，以及我们已经在前面说过的命名管道会在其中一端为就绪前阻塞另一端的进程，等等。

我们在前面讲过，命名管道可以被多路复用。所以，当有多个输入端同时写入数据的时候我们就不得不需要考虑操作原子性的问题。操作系统提供的管道是不提供原子操作支持的。为此，Go语言在标准库代码包`io`中提供了一个被存于内存中的、有原子性操作保证的管道（以下简称内存管道）。我们生成它的方法与之前的很相似：

```
reader, writer := io.Pipe()
```

函数`io.Pipe`返回两个结果值。第一个结果值是代表了该管道输出端的`*PipeReader`类型值，第二个结果只是代表了该管道输入端的`*PipeWriter`类型值。`*PipeReader`类型和`*PipeWriter`类型分别对管道的输出端和输入端做了很好的操作限制，即在`*PipeReader`类型的值上我们只能使用`Read`方法从管道中读取数据，而在`*PipeWriter`类型的值上我们则只能通过`Write`方法向管道写入数据。这样就有效避免了管道使用者对管道的反向使用。另一方面，我们在使用`Close`方法关闭管道的某一端之后，另一端在写数据或读数据的时候会得到一个预定义的`error`类型值。不过我们也可以通过调用`CloseWithError`来自定义另一端将会得到的`error`类型值。

另外，还需要注意，与`os.Pipe`函数生成的管道相同的是，我们仍然需要并发的运行被用来在内存管道的两端进行操作的代码。

在内存管道的内部是通过充分使用`sync`代码包中提供的API来从根本上保证操作的原子性的。所以，我们可以在它之上放心地并发写入和读取数据。另外，由于这种管道并不是基于文件系统的，并没有作为中介的缓冲区，所以通过它传递的数据只会被复制一次。这也就更进一步地提高了数据传递的效率。

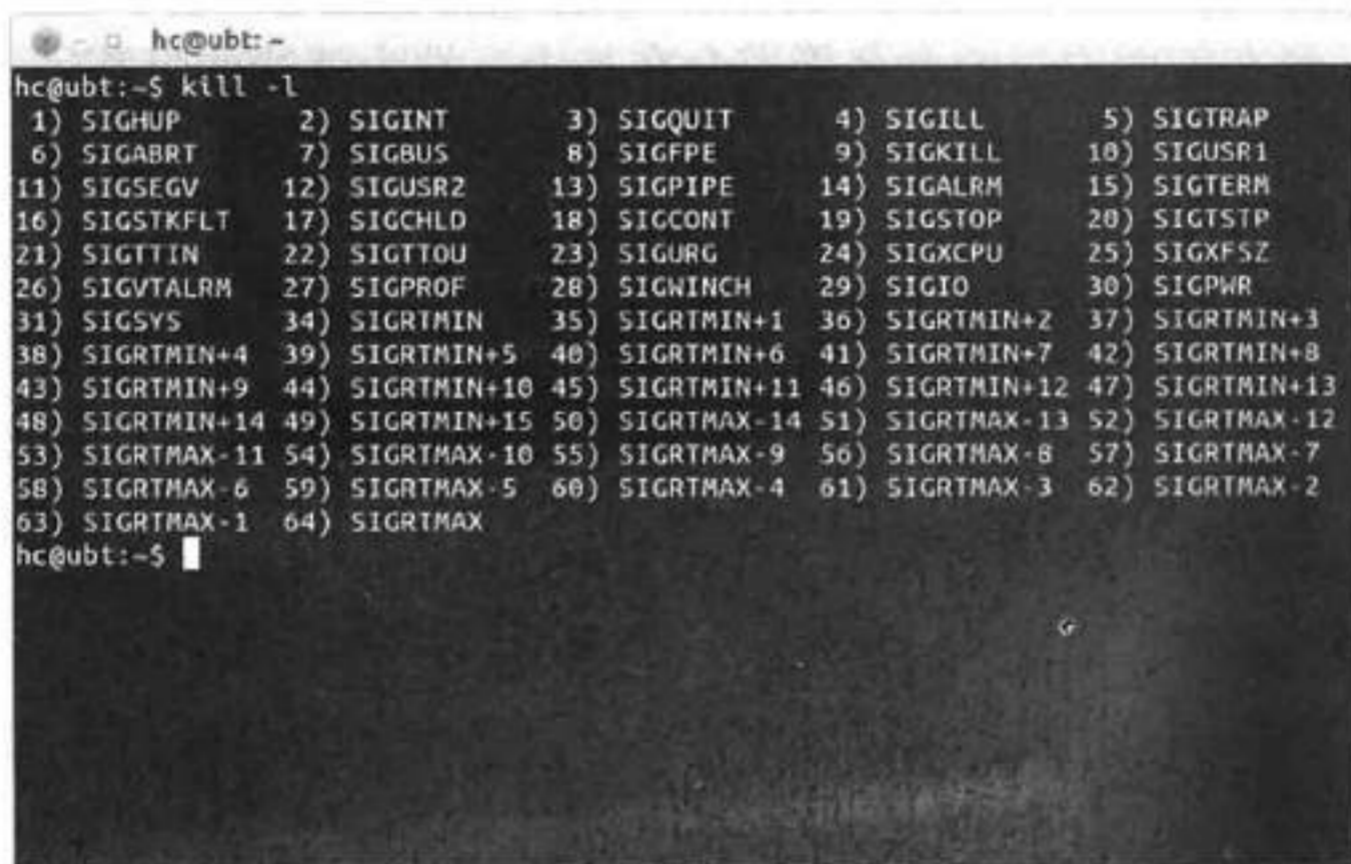
我们上面所展示的关于命名管道以及内存管道的示例代码都被集中放置在了`goc2p`项目的`multiproc/npipe`代码包中，以供读者参考和取用。虽然它们的使用方法都非常简单，但是其中的一些运用技巧（尤其是命名管道）还是值得我们特别记忆的。

至此，我们介绍了系统级别的管道的概念和基本用法，以及Go语言标准库中与系统管道对应的若干API的使用方法和技巧。另外，我们还简单地说明了Go语言特别提供的一种基于内存的同步管道的创建和使用方法。

6.2.4 信号

操作系统信号（Signal，以下简称信号）是IPC中唯一一种异步的通讯方法。它的本质是用软件来模拟硬件的中断机制。信号被用来通知某个进程有某个事件发生了。例如，在命令行终端下按下某些快捷键就会挂起或停止正在运行的程序。又例如，我们通过kill命令杀死某个进程的操作也有信号的参与。

每一个信号都有一个以“SIG”为前缀的名字。例如，我们会在稍后看到的SIGINT、SIGQUIT以及SIGKILL，等等。但是，在操作系统内部，这些信号都由正整数代表。这些正整数被称为信号编号。在Linux操作系统的命令行终端下，我们可以使用kill命令来查看当前系统所支持的信号，如图6-8所示。



```

hc@ubt:~$ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS     8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM     15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT    19) SIGSTOP     20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH   29) SIGIO        30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
hc@ubt:~$

```

图6-8 Linux操作系统支持的信号

可以看到，Linux操作系统支持的信号有62种（注意，没有编号为32和33的信号）。其中，编号从1到31的信号属于标准信号（也被称为不可靠信号），而编号从34到64的信号属于实时信号（也被称为可靠信号）。对于同一个进程来说，每种标准信号只会被记录并处理一次。并且，如果发送给某一个进程的标准信号的种类有多个，那么它们被处理的顺序也是完全不确定的。而实时信号解决了标准信号的这两个问题，即多个同种类的实时信号都可以被记录在案，并且它们可以按照信号的发送顺序被处理。虽然实时信号在功能上更为强大，但是已成为事实标准的标准信号也无法被替换掉。因此，这两大类信号一直共存着。

为了贴紧主题，我们下面仅仅会涉及使用Go语言开发信号处理程序所必需的知识。关于信号的完整概念和知识请读者参阅有关的文档和图书。

简单来说，信号的来源有键盘输入（比如按下快捷键Ctrl-c）、硬件故障，系统函数调用和软件中的非法运算。进程响应信号的方式有3种：忽略、捕捉和执行默认操作。

Linux操作系统对每一个标准信号都有默认的操作方式。针对不同种类的标准信号，其默认的操作方式一定会是以下操作中的一个：终止进程、忽略该信号、终止进程并保存内存信息、停

止进程、若进程已停止就恢复。

对于绝大多数标准信号而言，我们可以自定义当进程接收到它们之后应该进行怎样的处理。这种自定义信号响应的唯一方法是，进程要告知操作系统内核：当某种信号到来时，需要执行某某操作。在程序中，这些作为信号响应的自定义操作往往是由函数来代表的。

go命令会对其中的一些以键盘输入为来源的信号作出响应。这是由于go命令使用了在标准库代码包os/signal中的被用于处理信号的API。更具体地讲，go命令指定了需要被处理的信号并用一种很优雅的方式（使用到了通道类型的变量）来监听信号的到来。

下面我们从os.Signal接口类型开始讲起。该类型的声明如下：

```
type Signal interface {
    String() string
    Signal() // to distinguish from other Stringers
}
```

从os.Signal接口类型的声明可知，其中的Signal方法的声明并没有实际意义。它只是作为os.Signal接口类型的一个标识。因此，在Go语言标准库中，此接口类型的所有实现类型的Signal方法都是空方法（方法体中没有任何语句）。

所有此接口类型的实现类型的值都应该可以代表一个操作系统信号。理所当然，其中每一个操作系统信号都是需要由操作系统支持的。换句话说，它们都是依赖于操作系统的。

在Go语言的标准库中，已经包含了与不同操作系统的信号相对应的程序实体。在标准库代码包syscall中，已经为不同的操作系统所支持的每一个标准信号都声明了一个相应的同名常量（以下简称信号常量）。这些信号常量的类型都是syscall.Signal的。syscall.Signal是os.Signal接口类型的一个实现类型，同时也是一个int类型的别名类型。这就意味着，每一个信号常量都隐含着一个整数值。而信号常量的整数值与它所代表的信号在所属操作系统中的编号是一致的。

另外，如果我们查看syscall.Signal类型的String方法的源代码，还会发现一个包级私有的、名为signals的变量。在这个数组类型的变量中，每个索引值都代表了一个标准信号的编号，而对应的元素则是针对该信号的一个简短的描述。这些描述会分别出现在那些信号常量的字符串表示形式中。

好了，在了解了这些基础之后，我们就可以尝试使用os/signal代码包中的API来接受和处理操作系统的信号了。

代码包os/signal中的Notify函数用来把操作系统发给当前进程的指定信号通知给该函数的调用方。我们先来看看该函数的声明：

```
func Notify(c chan<- os.Signal, sig ...os.Signal)
```

函数signal.Notify的第一个参数是通道类型的。虽然我们还没有正式讲通道类型，但是在这里还是有必要简单解释一下这个参数。这个参数的类型是chan<- os.Signal。这表示，在该参数中只能传递os.Signal类型的值（以下简称信号值）。并且，在函数signal.Notify中，只能向该通道类型值放入信号值，而不能从该值中取出信号值。这一约束是由在关键字chan右边的接收操作符<-代表的。signal.Notify函数会把当前进程接收到的指定信号放入参数c代表的通道类型值

(以下简称signal接收通道)中。这样,调用方代码就可以从这个signal接收通道中按顺序的获取到操作系统发来的信号并进行相应的处理了。

函数signal.Notify的第二个参数是一个可变长的参数。这意味着我们在调用signal.Notify函数的时候,可以在第一个参数值之后再附加任意个os.Signal类型的参数值。参数sig代表的参数值应该包含我们希望自行处理的所有信号。在接收到我们希望自行处理的信号之后,os/signal包中的程序(以下简称signal处理程序)会把它封装成syscall.Signal类型的值并放入到signal接收通道中。当然,我们也可以只为第一个参数绑定实际值。在这种情况下,signal处理程序会把我们的意图理解为想要自行处理所有信号,并把接收到的几乎所有的信号都逐一进行封装并放入到signal接收通道中。下面我们来看一个例子(假设当前操作系统是Linux):

```
sigRecv := make(chan os.Signal, 1)
sigs := []os.Signal{syscall.SIGINT, syscall.SIGQUIT}
signal.Notify(sigRecv, sigs...)
for sig := range sigRecv {
    fmt.Printf("Received a signal: %s\n", sig)
}
```

在这个示例中,我们先创建了调用signal.Notify函数所需的两个参数的值。变量sigRecv的值就是signal接收通道。我们用内建函数make创建了它。它的元素类型是os.Signal且长度是1。我们希望自行处理SIGINT信号和SIGQUIT信号。所以,变量sigs代表的[]os.Signal类型的切片值包含了syscall.SIGINT和syscall.SIGQUIT两个元素。在我们调用signal.Notify函数之后,立即试图用for语句从signal接收通道中获取信号值。只要sigRecv的值中存在元素值,for语句就会把它们按顺序地接收并赋给迭代变量sig。否则,for语句就会被阻塞,并等待新的元素值被发送到sigRecv的值中。顺便提一句,在sigRecv代表的通道类型值被关闭之后,for语句会立即被退出执行,所以我们不用担心程序会一直在这里循环往复。不过,我们在实际使用for语句迭代通道类型值的时候,不应该如此简单地处理。关于这种用法的细节我们会在第7章介绍。

注意,signal处理程序在向signal接收通道发送值的时候,并不会因为通道已满而产生阻塞。因此,signal.Notify函数的调用方必须保证signal接收通道会有足够的空间缓存并传递接收到的信号。我们可以创建一个足够长的signal接收通道。但是,一个更好的方法是,只创建一个长度为1的signal接收通道,并且时刻准备从该通道中接收信号。我们在上面的示例中也是这么做的。

这个示例中的信号处理代码非常简单,即只是把从signal接收通道中接收的信号的字符串表现形式打印出来而已。在实际的场景中,这样做是比较危险的。因为我们忽略了当前进程本该处理的信号。为什么这么说呢?我们在前面说过,应该把想自行处理的信号追加在传递给signal.Notify函数的第一个参数值的后面。那么,如果当前进程接收到了我们不想自行处理的信号会怎样做呢?答案是,执行由操作系统指定的默认操作。所以,如果我们指定了想要自行处理的信号但又没有在接收到信号时执行必要的处理动作,就相当于使当前进程忽略了这些信号。以SIGINT信号为例。SIGINT信号即中断信号,一般被用来停止一个已经失去控制的程序。如果我们在运行一个Go语言程序的过程中按下快捷键Ctrl-c,那么此程序的运行会被停止。然而,如果在被运行的这个Go语言程序中含有上面示例中的那段代码的话,无论我们按下多少次Ctrl-c都

不能让它停下来，而仅仅会使标准输出上多出几行信息。试想一下，如果上面那段代码被修改为这样（注意第二行代码）会怎样：

```
sigRecv := make(chan os.Signal, 1)
signal.Notify(sigRecv)
for sig := range sigRecv {
    fmt.Printf("Received a signal: %s\n", sig)
}
```

如果被运行的Go语言程序中包含了这段代码，那么发给该进程的所有信号几乎都会被忽略掉。这样做而导致的后果可能是很悲剧的。

不过，幸好在类Unix操作系统下有两种信号既不能被自行处理也不会被忽略，它们是：SIGKILL和SIGSTOP。对它们的响应只能是执行系统默认操作。这种策略的最根本的原因是：它们向系统的超级用户提供了使进程终止或停止的可靠方法。系统不允许任何程序消除或改变与这两个信号所对应的处理动作。即使我们在程序中这样调用signal.Notify函数：

```
signal.Notify(sigRecv, syscall.SIGKILL, syscall.SIGSTOP)
```

也不会改变当前进程对SIGKILL信号和SIGSTOP信号的处理动作。这种保障，不论对于应用程序还是操作系统来说，都是非常有必要的。

对于其他信号，我们除了能够自行处理它们之外，还可以在之后的任意时刻恢复针对它们的系统默认操作。这需要使用到os/signal包中的Stop函数。它的声明如下：

```
func Stop(c chan<- os.Signal)
```

在函数signal.Stop的声明中只有一个参数声明。并且，这个参数声明与signal.Notify函数的第一个参数声明完全一致。这并不是巧合，而是有意为之。

函数signal.Stop会取消掉在之前调用signal.Notify函数的时候告知signal处理程序需要自行处理若干信号的行为。只有我们把当初传递给signal.Notify函数的那个signal接收通道作为调用signal.Stop函数时的参数值，才能如愿以偿地取消掉之前的行为，否则调用signal.Stop函数不会起到任何作用。在对signal.Stop函数的调用完成之后，作为其参数的signal接收通道将不会再被发送任何信号。这里存在一个副作用，即在之前示例中的那条被用于从signal接收通道接收信号值的for语句将会被一直阻塞。为了消除这种副作用，我们可以在调用signal.Stop函数之后使用内建函数close关闭该signal接收通道，就像下面这样：

```
signal.Stop(sigRecv)
close(sigRecv)
```

signal接收通道sigRecv被关闭之后，被用于从它那里接收信号值的for语句就会被退出执行。

在很多时候，我们可能并不想完全取消掉自行处理信号的行为，而只是想取消对一部分信号的自行处理。为了达到这个目的，我们只需再次signal.Notify函数，并重新设定与其参数sig绑定的、以os.Signal为元素类型的切片类型值（以下简称信号集合），只要作为第一个参数的signal接收通道相同就可以。在我们对signal.Notify函数的调用完成后，signal处理程序会发送给signal接收通道的信号的种类也会发生相应的改变。这完全取决于我们传递给signal.Notify函数的

os.Signal类型值都有哪些。

有些读者可能会有疑问：如果signal接收通道不同又会怎样？答案是这样的：如果我们先后调用了两次signal.Notify函数，但是两次传递给该函数的signal接收通道不同，那么signal处理程序会视为这两次调用毫不相干。它会分别看待这两次调用时所设定的信号的集合。

我们把前面比较散碎的示例整理成关于信号的第一个完整示例，并把这个示例存放在一个单独的函数（以下简称示例函数）中。由于这个完整示例中的信息量比较大，所以我们下面分阶段来展示和讲解。

第一个阶段的代码如下：

```
sigRecv1 := make(chan os.Signal, 1)
sigs1 := []os.Signal{syscall.SIGINT, syscall.SIGQUIT}
fmt.Printf("Set notification for %s... [sigRecv1]\n", sigs1)
signal.Notify(sigRecv1, sigs1...)

sigRecv2 := make(chan os.Signal, 1)
sigs2 := []os.Signal{syscall.SIGQUIT}
fmt.Printf("Set notification for %s... [sigRecv2]\n", sigs2)
signal.Notify(sigRecv2, sigs2...)
```

我们先后调用了两次signal.Notify函数，并且每次传递给它的signal接收通道并不相同。为了清晰起见，我们也同样初始化了两个信号集合。第一次调用时设定的信号集合中包含了SIGINT信号和SIGQUIT信号，而第二次调用时的信号集合中只有SIGQUIT信号。如此一来，如果当前进程接收到的是SIGQUIT信号，那么signal处理程序会把它封装之后先后发送给signal接收通道sigRecv1和sigRecv2。而如果接收到的是SIGINT信号，那么signal处理程序只会把封装好之后的信号发送给signal接收通道sigRecv1。也就是说，signal处理程序是分别处理不同的signal接收通道以及相应的信号集合的。

第二个阶段，我们要分别用两条for语句从signal接收通道sigRecv1和sigRecv2中接收信号值。由于这两条for语句都会被阻塞，所以我们不得不让它们并发执行。这需要用到我们还没有正式讲过的go语句。go语句与defer语句的组成很类似，即包含关键字、单条语句或函数以及调用符号。与go语句有关的知识我们在下一章再细说。在这里，我们只需要知道它会被并发的执行其中的单条语句或函数就可以了。此外，我们需要示例函数在这两段被并发执行的程序都执行完毕之后再退出执行。因此，这里还用到了标准库代码包sync中的类型WaitGroup。请看这个阶段的代码：

```
var wg sync.WaitGroup
wg.Add(2)
go func() {
    for sig := range sigRecv1 {
        fmt.Printf("Received a signal from sigRecv1: %s\n", sig)
    }
    fmt.Printf("End. [sigRecv1]\n")
    wg.Done()
}()
go func() {
    for sig := range sigRecv2 {
        fmt.Printf("Received a signal from sigRecv2: %s\n", sig)
```

```

    }
    fmt.Printf("End. [sigRecv2]\n")
    wg.Done()
}()

```

简单来说，我们会先调用`sync.WaitGroup`类型值`wg`的`Add`方法添加一个值为2的差量。然后，在每段并发程序的最后再调用`wg`的`Done`方法。这个方法的作用可以被视为使差量减1。在该示例函数的最后，我们再调用这个值的`Wait`方法。该方法会被一直阻塞直到差量变为0。这就相当于实现了我们刚刚描述的那个功能。

到了第三个阶段，我们不想从`signal`接收通道`sigRecv1`接收并自行处理信号了。换句话说，我们不再需要让`signal`处理程序向`signal`接收通道`sigRecv1`中发送信号了。不过，我们并不想马上这样做而是要先等待两秒钟。这使得我们能够有时间测试被删减之前的信号自行处理流程。这一功能可以通过标准库代码包`time`的`Sleep`函数来实现。第三阶段的代码如下：

```

fmt.Println("Wait for 2 seconds... ")
time.Sleep(2 * time.Second)
fmt.Printf("Stop notification... ")
signal.Stop(sigRecv1)
close(sigRecv1)
fmt.Printf("done. [sigRecv1]\n")

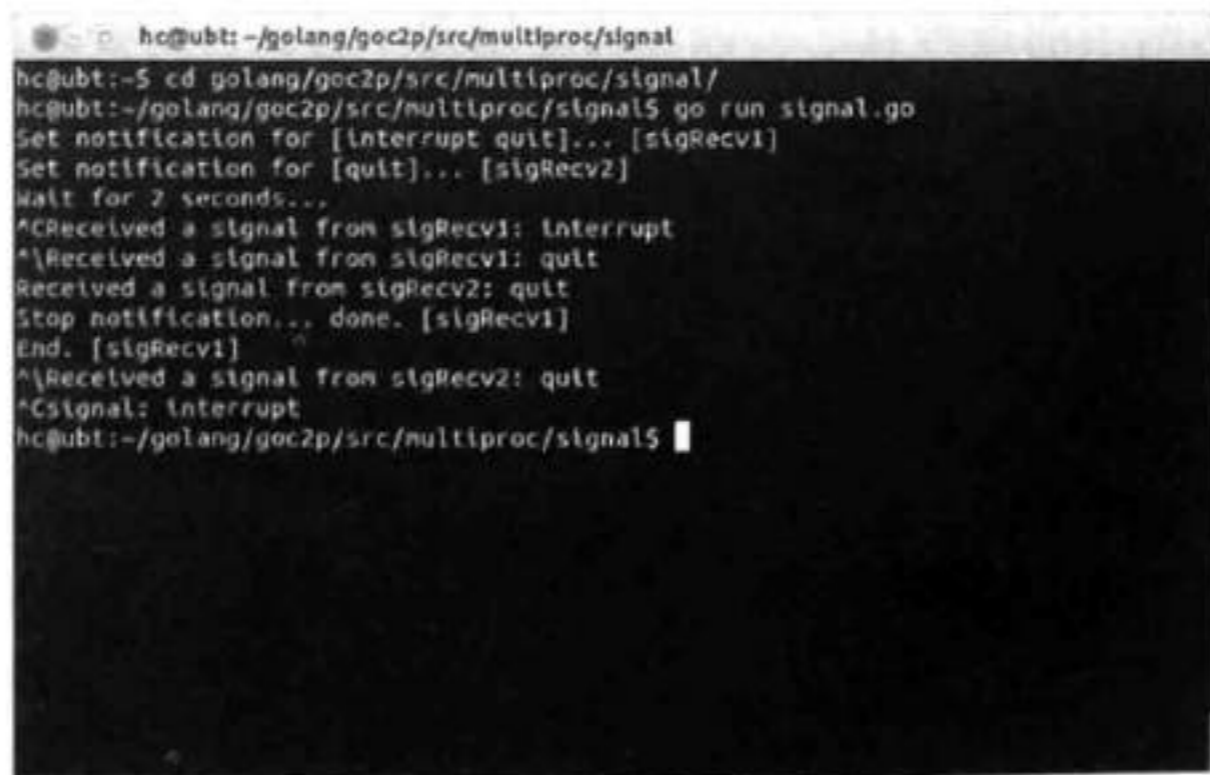
```

最后一个阶段只包含一条语句：

```
wg.Wait()
```

我们刚刚讲过这条语句的作用。这会避免该示例函数被提前退出执行。那样的话，我们就无法完整地演示信号自行处理的全过程了。

我把这个示例函数命名为`sigHandleDemo`并放到`goc2p`项目的`multiproc/signal`代码包的命令源码文件`signal.go`中。然后，我在该文件的`main`函数中添加了针对该示例函数的调用语句。最后，打开一个命令行终端，进入到`multiproc/signal`代码包所在的目录，使用`go run`命令运行命令源码文件`signal.go`。图6-9展现了这一演示的效果。



```

hc@ubt: ~/golang/goc2p/src/multiproc/signal
hc@ubt:~/golang/goc2p/src/multiproc/signal$ go run signal.go
Set notification for [interrupt quit]... [sigRecv1]
Set notification for [quit]... [sigRecv2]
Wait for 2 seconds...
^CReceived a signal from sigRecv1: interrupt
^CReceived a signal from sigRecv1: quit
Received a signal from sigRecv2: quit
Stop notification... done. [sigRecv1]
End. [sigRecv1]
^CReceived a signal from sigRecv2: quit
^Csignal: interrupt
hc@ubt:~/golang/goc2p/src/multiproc/signal$

```

图6-9 第一个信号示例的演示效果

注意，图中的这行内容

```
Stop notification... done. [sigRecv1]
```

是在第三个阶段的代码被执行后被打印出来的。这意味着，程序完成了取消掉与signal接收通道sigRecv1对应的信号自行处理行为以及关闭signal接收通道sigRecv1的操作。在这之前，我们无论从键盘上键入Ctrl-c（会导致向当前进程发送SIGINT信号）还是Ctrl-\（会导致向当前进程发送SIGQUIT信号），都只会由我们自己提供的代码进行处理。并且，我们按下Ctrl-\之后会有两行内容被打印出来，即

```
^\Received a signal from sigRecv1: quit  
Received a signal from sigRecv2: quit
```

这意味着，signal接收通道sigRecv1和sigRecv2都被发送了与SIGQUIT信号对应的信号值，并且与之相应的for语句也从中接收到了该信号值。但是，当sigRecv1被关闭之后，由这行内容

```
End. [sigRecv1]
```

可以看出，相应的for立即被退出执行。在这之后，我们再次按下Ctrl-\只会使sigRecv2被发送信号值。而我们再按下Ctrl-c则导致当前进程直接被停止。这是由于与sigRecv2对应的信号集合中并没有SIGINT信号。

我们纵观os/signal代码包中的这两个函数的行为特征就能够看出，它们都是以signal接收通道为唯一标识来对相应的信号集合进行处理的。在signal处理程序的内部，存在一个包级私有的字典（以下称为信号集合字典）。该信号集合字典被用于存放以signal接收通道为键、以信号集合的变体为元素的键值对。当我们调用signal.Notify函数的时候，signal处理程序就会在信号集合字典中查找相应的键值对。如果键值对不存在，就向信息集合字典添加这个新的键值对，否则就更新该键值对中的信息集合的变体。前者相当于向signal处理程序注册一个信号接收的申请，而后者则相当于更新该申请。signal接收通道作为函数调用方接收信号的唯一途径，也理所应当成为了这些申请的标识。也许读者已经猜到，当我们调用signal.Stop函数的时候，signal处理程序会删除掉信息集合字典中以该函数的参数值（某个signal接收通道）为键的键值对。

当接收到一个发送给当前进程且已被标识为应用程序想要自行处理的操作系统信号之后，signal处理程序会对它进行封装，然后遍历信息集合字典中的所有键值对，并查看它们的元素中是否包含了该信号。如果该信号被包含，那么就会立即把它发送给作为键的signal接收通道。这也进一步地解释了当我们多次调用signal.Notify函数且以不同的signal接收通道作为其参数值的时候所发生的事情。

总之，signal接收通道在Go语言提供的操作系统信号通知机制中起到了举足轻重的作用。我们能否合理地处理操作系统信号，也基本在于signal接收通道的初始化和使用的方式。

好了，我们现在已经对如何开始和停止自行处理接收到的信号有了足够多的了解。不过，Go语言程序能做的可不止这些。我们还可以编写向一个进程发送信号的程序。这需要用到标准库代码包os中的一些API。更具体地说，这主要依靠结构体类型os.Process和相关的函数和方法。

首先，我们可以使用os.StartProcess函数启动一个进程，或者使用os.FindProcess函数查找

一个进程。这两个函数都会返回一个`*os.Process`类型的值（以下简称进程值）和一个`error`类型值。然后，我们可以调用该进程值的`Signal`方法来向该进程发送一个信号。进程值的`Signal`方法接受一个`os.Signal`类型的参数值并会返回一个`error`类型值。

我们现在就以第一个信号示例为依托，来演示怎样编写向进程发送信号的Go语言程序。我把这些程序存放到了`mysignal.go`文件中的`sigSendingDemo`函数中。

为了演示第二个示例，我们不能直接使用`go run`命令运行`mysignal.go`文件。正确的做法是，先使用`go build`命令编译该文件，然后再执行刚刚被生成在当前目录的可执行文件`mysignal`。至于为什么要这么做，我们一会儿再解释。

执行`mysignal`文件会使操作系统生成一个进程。在这里，我们称这个进程为演示进程。好了，我们就以演示进程作为信号发送的目标。在第二个示例中，我们打算完成这样几个操作。

- (1) 执行一系列操作系统命令并获得演示进程的进程ID。当然，前提是演示进程已经被生成。
- (2) 根据演示进程的ID初始化一个进程值。
- (3) 使用该进程值之上的API向对应的进程发送一个SIGINT信号。
- (4) 在标准输出上打印出演示进程已接收到信号的凭证。

还记得吗？我们在第一个示例中已经实现了第4个操作。注意，这两个与信号有关的示例是在同一个命令源码文件中的。换句话说，发送信号的程序和自行处理信号的程序都会被包含在与演示进程对应的程序中。因此，第二个示例中的代码要做的就是给自己的进程发送一个SIGINT信号。而第一个示例中的代码会在收到该信号之后向标准输出打印一行内容。当我们在命令行终端下执行`mysignal`文件之后，这两个示例中打印的所有内容都会出现在当前的标准输出上。

好了，第二个示例的功能需求我们已经了解了。现在我们开始编写代码。首先要做的是获取当前进程的进程ID。这完全可以由Linux操作系统命令（或者说shell命令）来实现，但需要用到多个命令和匿名管道。这一系列命令最终被确定为：

```
ps aux | grep "mysignal" | grep -v "grep" | awk '{print $2}'
```

这其中用到了`ps`命令、`grep`命令和`awk`语言。这行命令由4个独立的shell命令组成。它们之间由匿名管道连接。这行命令的含义就是找到依据示例程序而生成并启动的进程的信息，然后找到信息中的进程ID。读者可以自己实验一下这行命令。当然，找到这个进程ID的前提是已经在其他命令行终端下或开发环境中生成并执行了`mysignal`文件。

我们依照此行shell命令创建一个`*exec.Cmd`类型值（以下简称命令值）的切片值，像这样：

```
cmds := []*exec.Cmd{
    exec.Command("ps", "aux"),
    exec.Command("grep", "mysignal.go"),
    exec.Command("grep", "-v", "grep"),
    exec.Command("awk", "{print $2}"),
}
```

由于我们在上一小节已经介绍了很多关于创建和使用命令值的知识，所以读者应该能看明白上面的这几行代码。

为了按顺序地执行前面的那行shell命令的并得到演示进程的进程ID，我们需要使用上一小节

所讲到的相关知识来编写一些代码。这些代码被封装在了一个名为`runCmds`的函数中。该函数的声明如下：

```
func runCmds(cmds []*exec.Cmd) ([]string, error)
```

这个函数接受一个代表了命令值列表的切片值作为参数，并返回一个代表了进程ID列表的`[]string`类型值和一个`error`类型值。它的函数体中的代码作者就不在此展示了。这会是一个很好的练习题。读者可以借此再复习一下怎样用Go语言程序实现串联命令的管道。

我们调用`runCmds`函数获得进程ID列表，像这样：

```
output, err := runCmds(cmds)
if err != nil {
    fmt.Printf("Command Execution Error: %s\n", err)
    return
}
```

由于`os.FindProcess`只接受一个`int`类型的参数值，所以我们还需要把`output`变量的值中的`string`类型值都转换为`int`类型值。这种转换非常容易，因为我们仅仅使用标准库代码包`strconv`中的`Atoi`函数就可以做到。

假设我们已经完成了上述转换并把结果赋给了`[]int`类型的变量`pids`。然后，我们使用`for`语句在`pids`之上进行迭代，并把每次迭代出来的值都赋给迭代变量`pid`。对于每一个`pid`的值，我们都可以使用代码

```
proc, err := os.FindProcess(pid)
```

得到进程值。在这里，我们把获取到的进程值赋给了变量`proc`。然后通过调用它的`Signal`方法给该值对应的进程发送信号，像这样：

```
err = proc.Signal(syscall.SIGINT)
```

顺便说一句，如果在本示例中向演示进程发送的是`SIGKILL`信号，那么我们调用进程值的`Kill`方法也可以达到相同的目的。

我们现在来解答为什么非要生成可执行文件`mysignal`，而不是直接使用`go run`命令来运行命令源码文件`mysignal.go`的问题。`go run`命令程序中会执行一系列动作，为最后的Go语言程序的运行做准备。粗略地讲，这包括了依赖查找、编译、打包、链接这几个步骤。当这些步骤完成之后，会有一个与被运行的命令源码文件的主文件名同名的可执行文件被生成在相应的临时工作目录中。对于命令源码文件`mysignal.go`来说，该可执行文件的名称就是`mysignal`。实际上，这与使用`go build`命令生成的可执行文件是一致的。在最后，命令程序会执行可执行文件`mysignal`。但是，需要注意，为了执行`mysignal`而产生的进程是一个全新的进程。它与代表了`go run mysignal.go`命令的那个进程毫不相干。也就是说，这两个进程是相互独立的。它们都拥有自己的进程ID。

因此，在这种情况下，我们使用前面的那行`shell`命令会把这两个进程的ID都输出出来。又由于进程信息列表的排列顺序问题，与`go run mysignal.go`命令对应的那个进程的信息往往会出现现在前面。所以，我们以顺序遍历进程ID列表的话，第二个示例中发送的信号会先到达与这个命令程序对应的进程。请注意，我们使用`go run`命令运行`mysignal.go`，命令程序会生成并执行可执

行文件mysignal，然后该可执行文件所产生的输出会通过该命令程序打印到标准输出上。也就是说，在该命令程序被挂起、停止或终止之后，mysignal中的程序所打印的内容也再不会出现在标准输出上了。因此，演示进程在接收到SIGINT信号之后的打印内容自然就不会被展示出来了。

为了消除这种影响，我们才有了前面的那个需要先编译后再执行可执行文件的要求。不过，这个要求限制了我们演示示例的方式。这种限制可能会让人生厌。实际上，一个更好的方法是使用shell命令

```
grep -v "go run"
```

来过滤掉原先进程信息列表中的与go run命令对应的进程。这样，被用于查找演示进程的shell命令就变成了这样：

```
ps aux | grep "mysignal" | grep -v "grep" | grep -v "go run" | awk '{print $2}'
```

而对应的命令值的切片值的声明也会变为：

```
cmds := []*exec.Cmd{
    exec.Command("ps", "aux"),
    exec.Command("grep", "mysignal"),
    exec.Command("grep", "-v", "grep"),
    exec.Command("grep", "-v", "go run"),
    exec.Command("awk", "{print $2}"),
}
```

好了，我们用这条声明cmds的语句替换掉原来的语句就可以了。

至此，sigSendingDemo函数中的代码我们也已经编写完毕了。现在我们要修改一下mysignal.go文件中的main函数，以使得sigHandleDemo函数和sigSendingDemo函数可以被并发的执行。这样，我们就可以看到比较好的演示效果了。我们依然运用go语句和time.Sleep函数来达到这一目的。main函数的完整声明如下：

```
func main() {
    go func() {
        time.Sleep(5 * time.Second)
        sigSendingDemo()
    }()
    sigHandleDemo()
}
```

在main函数中，我们并发的执行了sigSendingDemo函数。不过在执行它之前先等待了5秒钟以确保sigHandleDemo函数中的流程已经在

```
wg.Wait()
```

语句处阻塞。最后，我们来运行这个包含了信号自行处理功能和信号发送功能的完整示例。其演示效果如图6-10所示。



```
hcg@ubuntu:~/golang/goc2p/src/multiproc/signal$ go run nysignal.go
Set notification for [interrupt quit]... [sigRecv1]
Set notification for [quit]... [sigRecv2]
Wait for 2 seconds...
Stop notification...done. [sigRecv1]
End. [sigRecv1]
Run command: /bin/ps aux ...
Run command: /bin/grep nysignal ...
Run command: /bin/grep -v grep ...
Run command: /bin/grep -v go run ...
Run command: /usr/bin/awk (print $2) ...
Target PID(s):
[11441]
Send signal 'quit' to the process (pid=11441)...
Received a signal from sigRecv2: quit
```

图6-10 完整的信号示例的演示效果

请看在该命令行终端中最后出现的那两行内容。倒数第二行内容是第二个示例中的代码打印出来的。这表示即将要向演示进程发送SIGINT信号。倒数第一行内容是第一个示例中的代码打印出来的，表示演示进程已经接收到SIGINT信号并自行处理了。

看到下面的那个正在高亮的光标了吗？这表明演示进程并未结束执行。signal接收通道sigRecv2也没有被关闭，被用于从中接收信号值的for语句还一直处于阻塞状态。

经过本小节的一番讲解，相信读者已经基本掌握了使用Go语言自行处理和发送操作系统信号的方法。我们可以通过这些非常方便和灵活地使用操作系统信号。例如，我们可以在进程被终止前释放所持的系统资源和持久化一些重要数据。又例如，我们可以在当前进程中有效的控制其他相关进程的状态。

信号与管道都被称为基础的IPC方法。由于当今的主流操作系统对它们都有所支持，因此Go语言作为一种跨平台的计算机编程语言，自然也就把操纵它们的方法囊括在了标准库中。Go语言为我们提供了关于它们的更高层次的抽象方法和API。这使得我们可以在任何可以安装了Go语言的操作系统下用相同的种方式使用这些系统级别的功能。这也是Go语言为我们带来的便利之一。不过，需要注意的是，在基于数据传递的解决方案中，保证数据的原子性是非常重要的。然而，管道并不提供这种原子性保证。即使是Go语言标准库中提供的相关API也没有附加这种保证。

6.2.5 Socket

Socket，常被译为套接字。它也是一种IPC方法。但是与我们之前讲述的那几种IPC方法不同的是，它是通过网络连接来使两个或更多的进程建立通讯并相互传递数据的。这使得进行通讯的双方是否在同一台计算机上变得无关紧要。实际上，这是Socket的目标之一——使通讯端的位置透明化。

注意，本小节的内容会涉及一些TCP/IP协议栈的知识。但由于篇幅原因，我们并没有在这里的展开它们。读者若需要进一步了解它们，请参阅有关的文档和教程。

1. Socket的基本特性

在当今的大多数操作系统中都包含了Socket接口的实现。在主流以及更多的编程语言中也都有自己的基于Socket的API。当然，Go语言也不例外。

我们从操作系统提供的Socket接口开始讲起。在Linux操作系统中，存在一个名为socket的系统调用。其声明如下：

```
int socket(int domain, int type, int protocol);
```

该系统调用的功能是创建一个Socket实例。它接收3个参数。这3个参数分别代表了这个Socket的通讯域、类型和所用协议。

每个Socket都必将存在于一个通讯域当中。Socket的通讯域决定了该Socket的地址格式和通讯范围，参见表6-1。

表6-1 Socket的通讯域

通讯域	含义	地址形式	通讯范围
AF_INET	IPv4域	IPv4地址（4个字节）， 端口号（2个字节）	在基于IPv4协议的网络中的任意两台计算机之上的两个应用程序
AF_INET6	IPv6域	IPv6地址（16个字节）， 端口号（2个字节）	在基于IPv6协议的网络中的任意两台计算机之上的两个应用程序
AF_UNIX	Unix域	路径名称	在同一台计算机上的两个应用程序

由上表可知，Linux操作系统提供的Socket的通讯域有3个，即AF_INET、AF_INET6和AF_UNIX。它们分别代表了IPv4域、IPv6域和Unix域。这3个域的标识符都以“AF_”为前缀。“AF”是address family的缩写，意为地址族。这也暗示了每个域的Socket地址格式的不同。另外，我们还可以了解到，IPv4域和IPv6域的通讯是在网络范围内的，而Unix域的通讯则是在单台计算机范围内的。

Socket的类型有很多，包括SOCK_STREAM、SOCK_DGRAM、面向更底层的SOCK_RAW，以及针对某个新兴数据传输技术的SOCK_SEQPACKET。这些Socket类型的相关特性如表6-2所示。

表6-2 Socket类型的特性

特 性	Socket类型			
	SOCK_DGRAM	SOCK_RAW	SOCK_SEQPACKET	SOCK_STREAM
数据形式	数据报	数据报	字节流	字节流
数据边界	有	有	有	没有
逻辑连接	没有	没有	有	有
数据有序性	不能保证	不能保证	能够保证	能够保证
传输可靠性	不具备	不具备	具备	具备

表6-2呈现了不同Socket类型的5个特性。

数据形式有两种：数据报和字节流。以数据报为数据形式意味着数据接收方的Socket接口程序可以意识到数据的边界并会对它们进行切分。这样就省去了接收方的应用程序寻找数据边界和切分数据的工作量。以字节流为数据形式的数据传输实际上传输的是一个字节接着一个字节的串。我们可以把它想象成一个很长的字节数组。一般情况下，字节流并不能体现出其中的哪些字

节属于哪个数据包。因此，Socket接口程序是无法从中分离出独立的数据包的。这一工作只能由应用程序去完成。然而，SOCK_SEQPACKET类型的Socket的接口程序却截然不同。数据发送方的Socket接口程序可以忠实地记录数据边界。这里的数据边界就是应用程序每次发送的字节流片段之间的分界点。这些数据边界信息会随着字节流一同被发往数据接收方。数据接收方的Socket接口程序会根据数据边界把字节流切分成（或者说还原成）若干个字节流片段并按照需要依次传递给应用程序。

面向有连接的Socket之间在进行数据传输之前必须要先建立逻辑连接。在连接被建立好之后，通讯双方可以很方便地互相传输数据。并且，由于连接已经暗含了双方的地址，所以在传输数据的时候不必再指定目标地址。从另一个角度看，两个面向有链接的Socket之间一旦建立连接，那么它们发送的数据就只能被发送到连接的另一端。然而，面向无连接的Socket则完全不同。这类Socket在进行通讯时无需建立连接。它们传输的每一个数据包都是独立的，并且会直接被发送到网络上。在这些数据包中都含有目标地址，因此每个数据包都可能被传输至不同的目的地。此外，在面向无连接的Socket之上的数据流只能是单向的。也就是说，我们不能使用同一个面向无连接的Socket实例既发送数据又接收数据。

数据传输的有序性和可靠性与Socket是否面向连接有很大的关系。正因为逻辑连接的存在，通讯双方才有条件通过一些手段（比如基于TCP协议的序列号和确认应答，等等）来保证从数据发送方发送的数据能够及时、正确、有序地到达数据接收方，并被接收方接受。

最后要注意，SOCK_RAW类型的Socket提供了一个可以直接通过底层（TCP/IP协议栈中的网络互联层）传送数据的方法。为了保证安全性，应用程序必须具有操作系统的超级用户的权限才能够使用这种方式。并且，该方法的使用成本也相对较高，因为应用程序一般需要自己构建数据传输格式（像TCP/IP协议栈中的TCP协议的数据段格式和UDP协议的数据报格式那样）。因此，应用程序一般极少使用这种类型的Socket。

我们在调用系统调用socket的时候，一般会把0作为它的第三个参数值。其含义是让操作系统内核根据第一个参数和第二个参数的值自行决定Socket所使用的协议。这也意味着Socket的通讯域和类型与所用协议之间是存在对应关系的。这来通过表6-3来了解一下这种对应关系。

表6-3 Socket所用协议的默认选择

决定因素	SOCK_DGRAM	SOCK_RAW	SOCK_SEQPACKET	SOCK_STREAM
AF_INET	UDP	IPv4	SCTP	TCP或SCTP
AF_INET6	UDP	IPv6	SCTP	TCP或SCTP
AF_UNIX	有效	无效	有效	有效

在表6-3中，TCP（Transmission Control Protocol，中文译作传输控制协议）、UDP（User Datagram Protocol，中文译作用户数据报协议）和SCTP（Stream Control Transmission Protocol，中文译作流控制传输协议）都是TCP/IP协议栈中的传输层协议，而IPv4和IPv6则分别代表了TCP/IP协议栈中的网络互连层协议IP（Internet Protocol，中文译作网际协议）的第4个版本和第6个版本。“有效”表示该通讯域和类型的组合会使内核选择某个内部的Socket协议。“无效”则表示该通讯域和类

型的组合是不合法的。在Go语言提供的Socket编程API中也会涉及这些组合，并有一些专用的字符串字面量来表示它们。

现在我们来查看系统调用socket的返回值。在没有发生任何错误的情况下，系统调用socket会返回一个int类型的值。该值是作为socket唯一标识符的文件描述符。在得到该标识符之后，我们就可以调用其他系统调用来进行各种相关操作了，比如，绑定和监听端口、发送和接收数据以及关闭Socket实例，等等。不过，由于篇幅原因，我们就不在这里介绍那些系统调用的用法了。

注意，我们一直在说通过系统调用来使用操作系统提供的Socket接口。这就意味着，Socket接口程序与TCP/IP协议栈的实现程序一样，是Linux操作系统内核的一部分。

2. 基于TCP/IP协议栈的Socket通讯

我们已经知道，Socket接口既可以提供网络中的不同计算机上的多个应用程序间的通讯支持，也可以成为单台计算机上的多个应用程序间通讯的手段。虽然如此，但是我们使用Socket接口的绝大多数情况都是为了在网络中的进行通讯。这样的通讯是基于TCP/IP协议栈的。

图6-11表明了Socket接口与TCP/IP协议栈以及操作系统内核的关系。

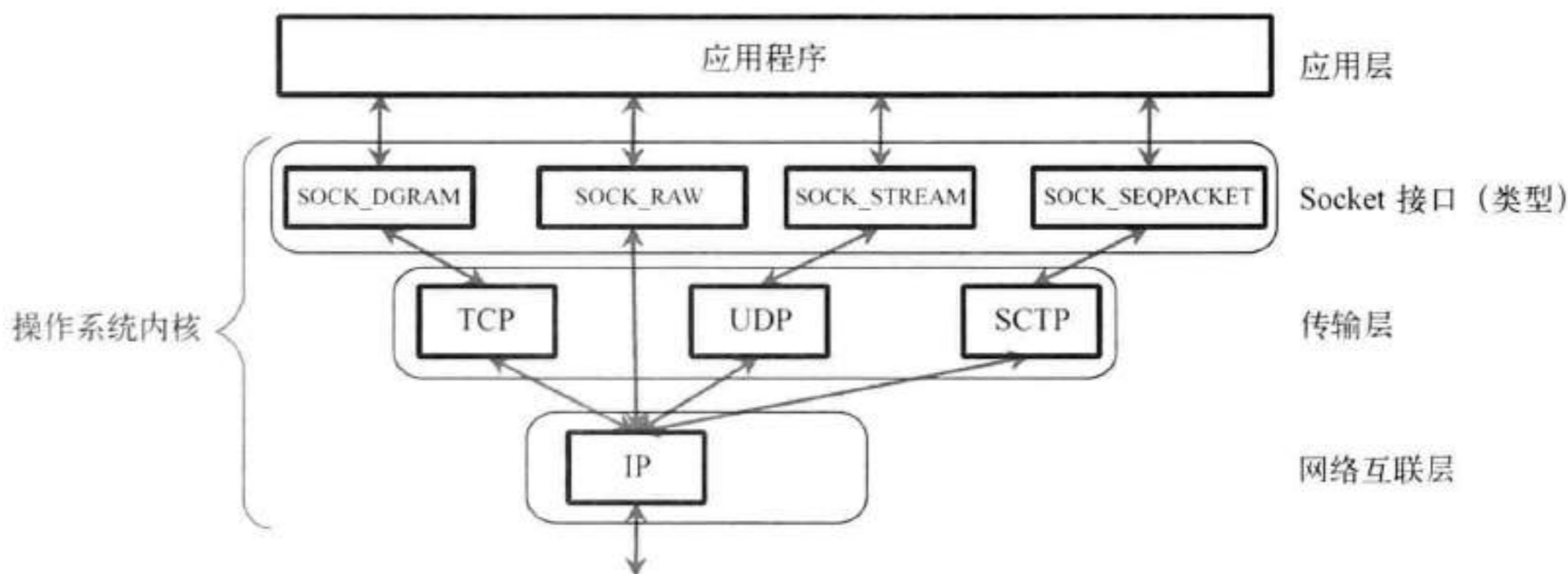


图6-11 Socket接口与TCP/IP协议栈

通过基于TCP/IP协议栈的Socket接口，我们不但可以建立和监听TCP连接和UDP连接，甚至还可以直接与网络互联层的IP协议实现程序进行通讯。不过，我们并不打算详细描述后者。因为，绝大多数应用程序需要的仅仅是与传输层的程序打交道。

在本小节中，我们会利用Go语言提供的Socket编程API来编写一个较完整的示例，以试图让读者学会使用它们。这个示例包含了两个在概念上独立的程序，即服务端程序和客户端程序。服务端程序会在一个给定的端口上监听TCP连接，而客户端程序则会试图与这个服务端程序建立TCP连接并进行通讯。为了让读者对基于TCP/IP协议栈的Socket通讯有一个宏观上的认识，我们绘制了一张流程图，如图6-12所示。它展现了TCP服务端和TCP客户端通过操作系统的Socket接口建立TCP连接并进行通讯的一般情形。其中不但涉及了我们在前面提到过的系统调用socket，还包含了一些我们并没有讲到的系统调用。

图6-12所示的只是一个极其简单的通讯流程。服务端程序在创建Socket实例、绑定本地地址、

监听地址之后开始等待连接的接入。在之后的某个时刻，客户端程序也创建了一个Socket实例并试图与服务端程序建立TCP连接。服务端程序接收到客户端程序发出的TCP连接请求并随即与它建立连接。客户端程序向服务端程序发送了请求数据，服务端程序在接收到并处理了该请求之后也向客户端程序发送了响应数据。客户端程序接收到了它想要的响应数据，并关闭了TCP连接。这时，客户端程序所在的操作系统的内核会通知服务端程序。服务端程序在接到通知之后会立即关闭相对应的TCP连接。在实际的应用场景中，通讯双方会进行多次数据交互。也就是说，图6-12中在圆角框之内的子流程一般会循环很多次。

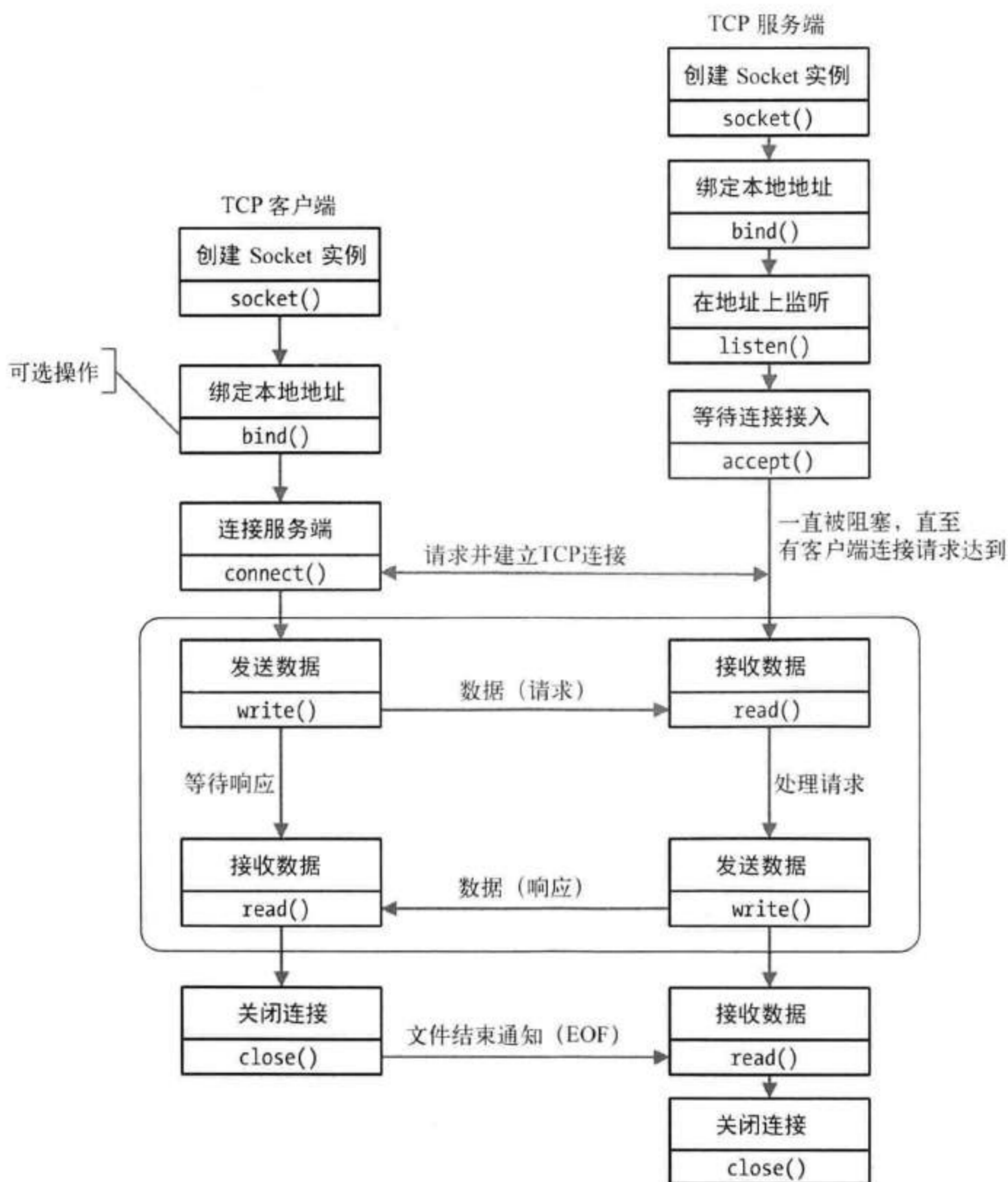


图6-12 基于TCP/IP协议栈的Socket通讯的一个简单流程

为了使用Go语言程序实现上面所说的服务端程序和客户端程序，我们主要会使用到标准库代码包net中的API。首先，我们会用到这个函数：

```
func Listen(net, laddr string) (Listener, error)
```

函数net.Listen被用于获取一个监听器。它接受两个string类型的参数。第一个参数的含义是以何种协议来在给定的地址上监听。我们在稍前的内容中已经介绍过Socket可能使用的协议。在Go语言中，这些协议由一些字符串字面量来表示，如表6-4所示。

表6-4 代表Socket协议的字符串字面量

字面量	Socket协议	备 注
"tcp"	TCP	无
"tcp4"	TCP	网络互联层协议仅支持IPv4
"tcp6"	TCP	网络互联层协议仅支持IPv6
"udp"	UDP	无
"udp4"	UDP	网络互联层协议仅支持IPv4
"udp6"	UDP	网络互联层协议仅支持IPv6
"unix"	有效	可看作是在通讯域为AF_UNIX且类型为SOCK_STREAM的时候内核采用的默认协议
"unixgram"	有效	可看作是在通讯域为AF_UNIX且类型为SOCK_DGRAM的时候内核采用的默认协议
"unixpacket"	有效	可看作是在通讯域为AF_UNIX且类型为SOCK_SEQPACKET的时候内核采用的默认协议

函数net.Listen的第一个参数的值所代表的必须是面向流的协议。TCP和SCTP都属于面向流的传输层协议。但不同的是，TCP协议实现程序无法记录和意识到任何消息边界，也无法从字节流分离出消息，而SCTP协议实现程序却可以做到这些的。后者使得应用程序无需再在发送的字节流的中间加入额外的消息分隔符，也无需再去查找所谓的消息分隔符并据此对字节流进行切分。保存消息边界的这种做法有利有弊，因此TCP协议和SCTP协议也各自适用于不同的场景。不过，二者皆适用的情况也是存在的。

解释一下，消息是数据包在TCP/IP协议栈的应用层中的称谓。消息边界与我们前面所说的数据边界的含义基本相同。这两者的不同之处在于，消息边界仅仅针对消息，而数据边界针对的对象的范围更广。还要注意，数据段是TCP协议实现程序为了使数据流满足网络传输的要求而做的分段，与这里所说的被用于区分独立消息的消息边界毫不相关。

综上所述，net.Listen函数的第一个参数的值必须是tcp、tcp4、tcp6、unix和unixpacket中的一个。它们代表的都是面向流的协议。其中，tcp4和tcp6分别仅与基于IPv4协议的TCP协议和基于IPv6协议的TCP协议相对应，而tcp则表示Socket所用的TCP协议会（或者说应该）兼容这两个版本的IP协议。另外，unix和unixpacket分别代表了两个通讯域为Unix域的内部的Socket协议。遵循它们的Socket实例仅被用于在本地计算机上的不同应用程序之间的通讯。

对于基于TCP协议的Socket来说，net.Listen函数的第二个参数laddr的值代表了当前程序在网络中的标识。laddr是Local Address的简写形式。它的格式是“host:port”。其中，“host”代表IP地址或主机名，而“port”则代表当前程序欲监听的端口号。例如，127.0.0.1:8085。注意，在

“host”处的内容必须是与当前计算机对应的IP地址或主机名，否则在调用该函数的时候会造成一个错误。另外，如果在“host”处的是主机名，那么该API中的程序（以下简称API程序）会先通过DNS（Domain Name System，中文译作域名系统）找到与该主机名对应的IP地址。因此，若“host”处的主机名没有在DNS中注册，那么也同样会造成一个错误。

好了，现在我们可以迈出构建一个基于TCP协议的服务端程序的第一个步了，像这样：

```
listener, err := net.Listen("tcp", "127.0.0.1:8085")
```

函数net.Listen返回两个结果值。第一个结果值是net.Listener类型的。它就是我们欲获取的监听器。第二个结果值是一个error类型值。它代表可能出现的错误。当然，和往常一样，我们需要先判断变量err是否为nil。若判断结果为真，则说明以给定的协议在给定的地址上的监听无法开始。这时，我们往往应该先去检查传递给net.Listen函数的两个参数值的合法性。否则，我们就可以开始等待客户端的连接请求了，代码如下：

```
conn, err := listener.Accept()
```

当我们调用net.Listener类型值的Accept方法的时候，流程会被阻塞，直到某台计算机上的某个应用程序与当前程序建立了一个TCP连接。此时，Accept方法会返回两个结果值。第一个结果值是代表了当前TCP连接的net.Conn类型值，而第二个结果值依然是一个error类型值。我们依旧要先对第二个结果值进行检查。

为了让这个不为人知的服务端程序具有意义，我们在继续编写服务端程序之前先来了解一下怎样才能与一个服务端程序建立TCP连接，并实现一个客户端程序。

代码包net中的Dial函数可被用于向网络中的某个地址发送数据，它的声明如下：

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

函数net.Dial也接受两个参数。其中，network与net.Listen函数的第一个参数net含义非常类似。它比后者拥有更多的可选值。因为，发送数据之前不一定要先建立连接。像UDP协议和IP协议就都是面向无连接型的协议。因此，udp、udp4、udp6、ip、ip4和ip6都可以作为参数network的值。其中，udp4和udp6分别代表了仅基于IPv4协议的UDP协议和仅基于IPv6协议的UDP协议，而udp所代表的UDP协议则在IP协议的版本上没有任何限制。另外，unixgram也是network参数的可选值之一。与unix和unixpacket相同，unixgram也代表了一个基于Unix域的内部Socket协议。但不同的是，后者是以数据报作为传输形式的。

函数net.Dial的第二个参数address的含义与net.Listen函数的第二个参数laddr完全一致。如果我们想与前面刚刚开始监听的服务端程序连接的话，那么这个参数的值就应该是该服务端的地址，即为127.0.0.1:8085。因此，这个参数的名称address其实也可由raddr（Remote Address）代替。名称laddr和raddr都是相对的。前者指的是当前程序所使用的地址（本地地址），而后者则指的是参与通讯的另一端所使用的地址（远程地址）。我们会在net代码包的函数或方法声明中经常见到这两个参数名称。

有的读者可能会问：客户端自己的地址在哪里给出呢？答案是根本不用给出。端口号可以由应用程序指定，也可以由操作系统内核动态分配。就使用net.Dial建立Socket连接的客户端程序

而言，它占用的端口号是由操作系统内核动态分配的。另一方面，客户端程序的地址中的“host”一定是本地计算机的主机名或IP地址，这也会由操作系统内核为我们指定。当然，我们也可以自己去指定当前程序的地址，不过这就需要使用另外的函数建立连接了。我们后面再探讨这个问题。

调用net.Dial函数的代码类似于：

```
conn, err := net.Dial("tcp", "127.0.0.1:8085")
```

函数net.Dial返回两个结果值。一个是net.Conn类型值，另一个是error类型值。同样的，若参数值不合法，则第二个结果值会不为nil。此外，对基于TCP协议的连接请求来说，当在远程地址之上并没有程序正在监听的时候，也会使net.Dial函数返回一个非nil的error类型值。

我们都知道，网络中是存在延时现象的。因此，在收到另一方的有效回应（无论连接成功或失败）之前，发送连接请求的一方往往会等待一段时间，在上面的示例中则表现为流程在调用net.Dial函数的那行代码上一直阻塞。在超过这个等待时间之后，函数的执行就会结束并返回相应的error类型值。因此，这类等待时间也常被称为超时（timeout）时间。不同操作系统对基于不同协议的连接请求的超时时间都有不同的设定。例如，在Linux操作系统内核中，把基于TCP协议的连接请求的超时时间设定为75秒。与其他超时时间相比，这已经算是很短了。在很多应用场景中，固定不变的超时时间往往无法满足需求。我们总是希望掌控能够掌控的一切，在编写代码时也不例外。因此，操作系统内核为我们提供了改变这类超时时间的接口。同时，在Go语言的net代码包中也存在相应的API。对于net.Dial函数来说，可同时设定超时时间的函数为net.DialTimeout。它的声明如下：

```
func DialTimeout(network, address string, timeout time.Duration) (Conn, error)
```

该函数与net.Dial函数的唯一区别就是可以同时连接请求的超时时间进行设定。我们可以看到，net.DialTimeout函数声明中的最后一个参数是被专门用于设定超时时间的。它的类型是time.Duration，单位是纳秒。但是，我们设定的超时时间一般会比纳秒级别高好几个数量级。不过不用担心，在标准库代码包time中，预先声明了与常用的时间单位相对应的time.Duration类型的常量。time.Duration类型是int64类型的一个别名类型。所以，不严谨地说，time.Duration相当于一个数值类型。在设定time.Duration类型的值的时候，我们可以直接使用它们来拼凑需要的时间，而不用再去计算诸如1小时48分73秒等于多少纳秒之类的问题。例如，常量time.Nanosecond代表1纳秒，它的值就是1。而常量time.Microsecond代表1微秒，其值为1000 * Nanosecond，也就是1000纳秒。以此类推。当我们想表示一个时间为2秒的time.Duration类型值时可以这样编写：

```
2 * time.Second
```

如果我们在请求TCP连接的同时想把超时时间设定为2秒，我们可以这样调用net.DialTimeout函数：

```
conn, err = net.DialTimeout("tcp", "127.0.0.1:8085", 2*time.Second)
```

注意，我们在这里设定的超时时间不一定是net.DialTimeout函数的执行总耗时。因为，如果远程地址的“host”部分是主机名而不是IP地址的话，API程序在向服务端发送连接请求之前还

需要向DNS询问与该主机名对应的IP地址。而这一操作也是有超时时间的。这个超时时间也等于我们传递给`net.DialTimeout`函数的第三个参数值。因此，在最糟糕的情况下，`net.DialTimeout`函数的执行总耗时等于我们设定的超时时间的两倍。

至此，我们讲述的API足以让我们在服务端程序和客户端程序之间建立起TCP连接。不过，看起来我们在这里并没有使用操作系统内核提供的API创建Socket实例。的确，这一操作已经被隐含在Go语言提供的Socket API程序中了。此外，在服务端，与本地地址绑定的操作也被隐含在`net.Listen`函数背后的程序中了。这种在API上的简化是很值得称赞的。虽然这隐藏了处于底层的Socket接口的相关细节，但好在我们通过前面内容已经对Socket接口有所了解。

在前面，我们在通过调用`net.Listen`函数得到一个`net.Listener`类型值之后，又调用该值的`Accept`方法以等待客户端连接请求的到来。当收到客户端的连接请求之后，服务端会与客户端建立TCP连接（三次握手）。当然，这个连接的建立过程是两端的操作系统内核共同协调完成的。当成功建立连接后，我们会通过从`Accept`方法得到一个代表了该TCP连接的`net.Conn`类型值。这就是说，不论服务端程序还是客户端程序，当TCP连接建立完成之后都会得到一个`net.Conn`类型值。在这之后，通讯两端就可以分别利用各自获得的`net.Conn`类型值交换数据了。下面我们就来说说API程序在`net.Conn`类型之上提供的功能。

首先需要说明的是，Go语言的Socket编程API程序在底层获取的是一个非阻塞式的Socket实例。这就是说，我们使用Socket接口在一个TCP连接上的数据读取操作也都是非阻塞式的。在应用程序试图通过系统调用`read`从Socket的接收缓冲区中读取数据的时候，即使接收缓冲区中没有任何数据，操作系统内核也不会使系统调用`read`进入阻塞状态，而是直接返回一个错误码为“EAGAIN”的错误。但是，应用程序并不应该视此为一个真正的错误，而是应该忽略该错误然后稍等片刻之后再去尝试读取。另外，如果在读取数据的时候接收缓冲区有数据，那么系统调用`read`就会携带这些数据立即返回。即使当时的接收缓冲区中只包含了一个字节的数据也会是这样。这一特性被称为部分读（partial read）。另一方面，在应用程序试图向Socket的发送缓冲区中写入一段数据的时候，即使发送缓冲区已被填满系统调用`write`也不会被阻塞，而是直接返回一个错误码为“EAGAIN”的错误。同样地，应用程序应该忽略该错误并稍后再尝试写入数据。如果发送缓冲区中有少许剩余空间但不足以放入这段数据，那么系统调用`write`会尽可能地写入一部分数据然后返回已写入的字节的数量。这一特性被称为部分写（partial write）。应用程序应该每次调用`write`之后都去检查该结果值，并在发现数据未被完全写入时继续写入剩下的数据。在非阻塞式的Socket接口之下，除了`read`和`write`之外，系统调用`accept`也会显现出一致的非阻塞风格。它不会被阻塞以等待新连接的到来，而会直接返回错误码为“EAGAIN”的错误。有些读者可能会立即发问：前面说`net.Listener`类型值的`Accept`方法会在被调用时阻塞直至新连接的到来，它们与这里所说的非阻塞式的行为并不相符啊？！别急，请读者继续看接下来的说明。

Go语言的Socket编程API程序在一定程度上充当了前面所说的应用程序的角色。它为我们屏蔽了相关系统调用的“EAGAIN”错误。这使得有些Socket编程API调用起来像是阻塞式的。但是，我们应该明确，它在底层使用的是非阻塞式的Socket接口。另外，需要注意的是，Go语言的Socket编程API程序同样为我们屏蔽了非阻塞式Socket接口的部分写特性。相关API直到把所有数据全部

写入到Socket的发送缓冲区之后才会返回，除非在写入的过程中发生了某种错误。但是，它却保留了非阻塞式Socket接口的部分读特性，并把它们呈现给了它的使用者（我们编写的应用程序）。这样做是合理的。因为，在TCP协议之上传输的数据是字节流形式的。数据接收方无法意识到数据的边界（也可以说消息边界）。所以，Socket编程API程序也就无从判断函数调用返回的时机。把数据切分和分批返回的任务交给调用方程序也算是最好的选择了。部分读是需要我们在程序中做一些额外的处理的。我们会在后面进一步说明这个问题。

好了，现在让我们重新关注net.Conn类型。它是一个接口类型。在它的方法集合中包含了8个方法。它们定义了我们可以从一个连接上做的所有事情。接下来，我们就逐一地对它们进行说明。

1. Read方法

方法Read被用来从Socket的接收缓冲区中读取数据。下面是该方法的声明：

```
Read(b []byte) (n int, err error)
```

该方法接受一个[]byte类型的参数。该参数的值相当于一个被用来存放从连接上接收到的数据的“容器”。它的长度完全由应用程序来决定。Read方法会最多从连接中读取数量等于该参数值的长度的若干字节，并把它们依次放置到该参数值中的相应元素位置（索引值从0到len(b)-1）上。该参数值中的相应位置上的原元素值将会被替换。不过，即使是这样，我们也应该让“容器”保持绝对地干净。换句话说，传递给Read方法的参数值应该是一个不包含任何非零值元素的切片值。在一般情况下，Read方法只有在把参数值填满之后才会返回。但是，在有些情况下，Read方法在未填满参数值之前就返回了。这可能是由相关的网络数据缓存机制导致的。我们在前面已经说明过这一问题。不管是什么原因，如果Read方法未填满参数值，而该参数值的靠后部分又存在遗留元素值的话，我们就需要特别小心。好在Read方法返回的第一个结果值可以帮助我们从中识别出真正的数据部分。结果n代表了本次操作实际读取到的字节的个数。我们也可以把它理解为Read方法向参数值中填充的字节的个数。我们可以这样来使用它：

```
b := make([]byte, 10)
n, err := conn.Read(b)
content := string(b[:n])
```

我们通过对[]byte类型的结果n的切片来抽取出接收到的数据。即使n的值为0，这样做也不会有任何问题。但是，我们仍然需要通过检查作为结果之一的error类型值来判断函数的执行是否正常结束。读者应该已经非常熟悉这种做法了。不过，我们在这里对错误的检查会稍微复杂一些。

如果Socket编程API程序在从Socket的接收缓冲区中读取数据的时候发现TCP连接已经被另一端关闭了，那么就会立即返回一个error类型值。这个error类型值与io.EOF变量的值是相等的。我们在前面多次接触过io.EOF变量。它的值象征着文件内容的完结。相应地，该值在这里意味着在该TCP连接上再无可被读取的数据。也可以说，该TCP链接已经无用，可以被关闭了。因此，如果Read方法的第二个结果值与io.EOF变量的值相等，那么我们就应该中止后续的数据读取操作，并关闭该TCP连接。请看下面的代码：

```
var dataBuffer bytes.Buffer
b := make([]byte, 10)
```

```

for {
    n, err := conn.Read(b)
    if err != nil {
        if err == io.EOF {
            fmt.Println("The connection is closed.")
            conn.Close()
        } else {
            fmt.Printf("Read Error: %s\n", err)
        }
        break
    }
    dataBuffer.Write(b[:n])
}

```

上面这几行代码较完整地展现了一个在代表TCP连接的`net.Conn`类型值之上读取数据的流程。首先，我们声明一个`bytes.Buffer`类型值，并以此来存储将会接收到的所有数据。通过不带任何子句的`for`语句，我们编写出了一个可以被无限循环执行的代码块。在这个代码块中，我们总是先在变量`conn`的值上调用`Read`方法以读取从网络上接收到的数据，并在确定未发生任何错误之后把数据追加到`dataBuffer`的值中。这可以解决我们前面提到的非阻塞式的`Socket`接口的部分读特性所带来的问题。另一方面，对于非`nil`的`error`类型值，我们还有第二层判断。如果它等于`io.EOF`变量的值，那么就说明当前连接已经被正常关闭，而不是有真正的错误发生。这时，我们可以打印提示信息，然后在本端也执行关闭连接的操作。否则，我们就应该打印出错误信息。无论第二层判断的结果如何，我们都会终止当前的`for`语句的执行。当然，在发生读取错误的时候，是否需要终止循环应该根据具体的应用场景来决定。我们在这里展示的是最简单的情况。另一个可能需要调整地方是，我们一般不会对连接被关闭之前无休止地从连接上读取数据。作为一个处在TCP/IP协议栈的应用层的程序，应该负责切分数据并生成有实际意义的消息。即使在最简单的情况下，应用层程序也应该知道怎样在接收到的字节流上进行切分。我们可以按照自己的要求去编写实现切分操作的程序。不过，还有一个更简便的方法。我们可以利用标准库代码包`bufio`中的API实现一些较复杂的数据切分操作。`bufio`是Buffered I/O的缩写。顾名思义，`bufio`代码包中的API提供了与带缓存的I/O操作有关的支持。比如，通过包装不带缓存的I/O类型值的方式增强它们的功能。我们在前面讲管道的时候已经介绍过`bufio.NewReader`函数的用法。它接收一个`io.Reader`类型的参数值。由于`net.Conn`类型实现了接口类型`io.Reader`中唯一的方法`Read`，所以它是该接口类型的一个实现类型。因此，我们可以使用`bufio.NewReader`函数来包装变量`conn`，像这样：

```
reader := bufio.NewReader(conn)
```

在这之后，我们就可以通过调用`reader`变量的值之上的`ReadBytes`方法来依次获取经过切分之后数据了。`ReadBytes`方法接受一个`byte`类型的参数值。该参数值应该是通讯两端协商一致的那个消息边界。一个关于`ReadBytes`方法的用法示例如下：

```
line, err := reader.ReadBytes('\n')
```

一般情况下，在每次调用`ReadBytes`方法之后，我们都会得到一段以该消息边界为结尾的数

据。当然，在很多时候，消息边界的定位并不是查找一个单字节字符那么简单。比如，HTTP协议中规定，在HTTP消息的头部信息的末尾一定是连续的两个空行，即字符串“\r\n\r\n”。在获取到HTTP消息的头部信息之后，相关程序会通过其中的名为“Content-Length”的信息项的值得到HTTP消息的数据部分的长度。这样，一个HTTP消息就可以被切分出来了。为了满足这些较复杂的需求，bufio代码包为我们提供了一些更高级的API，例如bufio.NewScanner函数、bufio.Scanner类型及其方法，等等。

2. Write方法

方法Write被用来向Socket的发送缓冲区写入数据。下面是该方法的声明：

```
Write(b []byte) (n int, err error)
```

该方法背后的API程序为我们屏蔽了很多非阻塞式Socket接口的细节。这使得我们可以简单地调用它而不用再做其他额外的处理，除了需要应对可能会发生的操作超时异常。

同样地，我们也可以使用代码包bufio中的API来使这里的写操作更加灵活。net.Conn类型的Write方法的声明与io.Writer接口类型中的唯一方法Write的声明完全一致。所以，net.Conn类型的值可以作为bufio.NewWriter函数的参数值，像这样：

```
writer := bufio.NewWriter(conn)
```

与前面示例中的变量reader类似，writer的值可以被看作是针对变量conn代表的TCP连接的缓冲写入器。我们可以调用其上的以“Write”为名称前缀的方法分批次地向其中的缓冲区写入数据，也可以调用它的ReadFrom方法直接从其他io.Reader类型值中读出并写入数据，还可以通过调用Reset方法以达到重置和复用它的目的。在向其写入全部数据之后，我们应该调用它的Flush方法，以保证其中的所有数据都被真正地写入到了它代理的对象（在这里，这一对象就是由变量conn代表的TCP连接）中。此外，我们应该留心该缓冲写入器的缓冲区容量（默认是4096个字节）。因为，在我们调用以“Write”为名称前缀的方法的时候，如果作为参数值的数据的字节数量超出了此容量，那么该方法就会试图把这些数据的全部或一部分直接写入到它代理的对象中，而不会先在缓冲写入器自己的缓冲区中缓存这些数据。有时候，这并不是我们希望的。为解决此类问题，我们可以通过调用bufio.NewWriterSize函数来初始化一个缓冲写入器。该函数与bufio.NewWriter函数非常类似，但它让我们可以自定义将要生成的缓冲写入器的缓冲区容量。

3. Close方法

方法Close会关闭当前的连接。它不接受任何参数并返回一个error类型值。在调用该方法之后，对该连接值（由示例中的conn变量代表的值）上的Read方法、Write方法或Close方法的任何调用都会使它们立即返回一个error类型值。代表该error类型值的变量已经被预置在了net代码包中，其提示信息是：

```
use of closed network connection
```

另外，如果我们在调用Close方法的时候，Read方法和/或Write方法正在被应用程序调用且还未执行结束，那么它们也会立即结束执行并返回非nil的error类型值。即使它们正处于阻塞状态也会是这样。

4. LocalAddr和RemoteAddr方法

单从名称上来看，我们就能猜到这两个方法的作用了。它们都不接受任何参数并返回一个net.Addr类型的结果。这个结果的值代表了参与当前通讯的某一端的应用程序在网络中的地址。显然，LocalAddr方法返回的是代表了本地地址的net.Addr类型值，而RemoteAddr方法返回的则是代表了远程地址的net.Addr类型值。net.Addr类型是一个接口类型。在它的方法集合中有两个方法——Network和String。Network方法会返回当前连接所使用的协议的名称。例如，在我们所说的这个应用场景中，这条语句

```
conn.LocalAddr().Network()
```

会使我们得到"tcp"这个string类型值。String方法返回相应的地址。这个地址与我们前面所说的各个通讯域下的地址的表现形式和格式是对应的。对于IPv4域来说，这个地址的格式就是"host:port"。我们前面讲到的那个基于TCP协议的服务端程序的地址就是"127.0.0.1:8085"。这与我们获取监听器时给定的那个地址是一致的。当一个客户端连接到来时，我们可以通过如下语句获取该连接的另一端的应用程序的网络地址：

```
conn.RemoteAddr().String()
```

另一方面，对于客户端程序，如果我们在与服务端程序通讯的时候未指定本地地址，那么这条语句：

```
conn.LocalAddr().String()
```

会让我们得到操作系统内核为该客户端程序分配的网络地址。

5. SetDeadline、SetReadDeadline、SetWriteDeadline方法

这3个方法都只接受一个time.Time类型值，并会返回一个error类型值。方法SetDeadline会设定在当前连接上的I/O（包括但不限于读和写）操作的超时时间。注意，这里的超时时间是一个绝对时间！也就是说，如果在SetDeadline方法调用语句之后的相关I/O操作在到达此超时时间的时候还没有完成，那么它们就会被立即结束执行并返回一个非nil的error类型值。这个error类型值由一个被预置在net代码包中的包级私有变量代表。它的Error方法的返回值是"i/o timeout"。注意，当我们以循环的方式不断尝试从一个连接上读取数据的时候，如果想要设定超时时间，那么就需要在每次迭代中的读取数据操作之前都设定一次。这正是因为我们的超时时间是一个绝对时间，并且它会对之后的每个I/O操作都起作用。如果在超时时间达来的时候循环语句仍在执行过程中的话，那么在后面的迭代中执行的I/O操作都会失败并返回代表超时的error类型值。请看下面的示例：

```
b := make([]byte, 10)
conn.SetDeadline(time.Now().Add(2 * time.Second))
for {
    n, err := conn.Read(b)
    // 省略若干条语句
}
```

我们通过调用time.Now函数获得代表了当前绝对时间的time.Time类型值，然后调用该值的Add方法在当前绝对时间之上加上了2秒的相对时间。这就意味着，我们把超时时间设定为2秒之

后的那一时刻。假设，在之后的for语句块的第二次迭代完成的时候逝去的时间将近2秒，并for语句块第三次迭代开始的时候已经达到了超时时间。这时，在第三次迭代中的读操作会立即失败。并且，后面的迭代中的读操作也必定会相继失败。这样的流程设计显然是不正确的。如果我们把上面的代码改成这样：

```
b := make([]byte, 10)
for {
    conn.SetDeadline(time.Now().Add(2 * time.Second))
    n, err := conn.Read(b)
    // 省略若干条语句
}
```

那么只要Read方法的执行能够在2秒内结束，就不会有超时错误出现。这是由于我们在每次迭代的读操作开始之前都先对超时时间进行了延伸。

如果我们不再需要设定超时时间了，那就应该及时取消掉它，以免干扰后续的I/O操作。这一操作可以通过调用同样的方法来实现。如果给予SetDeadline方法的参数值为time.Time类型的零值，超时时间就会被取消掉。由于time.Time是一个结构体类型，所以我们用time.Time{}来表示它的零值。因此，下面代码会取消掉之前对超时时间的设定：

```
conn.SetDeadline(time.Time{})
```

读者肯定已经猜到了，SetReadDeadline方法和SetWriteDeadline方法的功能也是设定之后的I/O操作的超时时间。但不同的是，它们仅分别针对于读操作和写操作。这里说的读操作与连接值的Read方法的调用对应，而写操作则与连接值的Write方法的调用对应。对于写操作的超时，有一个问题需要明确。即使一个写操作（也就是对Write方法的调用）超时了，也不一定代表写操作完全没有成功。因为，在超时之前，Write方法背后的程序可能已经将一部分数据写到Socket的发送缓冲区了。也就是说，即使Write方法因操作超时而被迫结束执行并返回，它的第一个结果值也可能大于0。这时，该结果值就代表了在操作超时之前被真正写入的数据的字节数量。

另外，我们对SetDeadline方法的调用相当于先后以同样的参数值对SetReadDeadline方法和SetWriteDeadline方法进行调用。如果我们想统一设定所有相关的I/O操作的超时时间，那么使用SetDeadline方法肯定是便捷的。但当我们更需要更细致的控制操作超时的时候，就需要用到后两个方法了。总之，这3个方法为我们提供了不同粒度的I/O操作超时时间控制方法。最后，要记住，它们仅针对在当前连接值之上的I/O操作。

好了，我们现在对net.Conn接口类型上的所有方法也都有所了解了。现在，我们通过一个较完整的示例把这些知识和用法贯穿起来。

该示例包含服务端程序和客户端程序。它们以网络和TCP协议作为通讯的基础。服务端程序的功能可以被概括为：接收客户端程序的请求、计算请求数据的立方根，并把对结果的描述返回给客户端程序。下面是对服务端程序的功能需求更详细的描述。

- ❑ 需要根据事先约定好的数据边界把接收到的请求数据切分成数据块。
- ❑ 仅接受可以由int32类型表示的请求数据数据块。对于不符合要求的数据块，要生成错误信息并返回给客户端程序。并且，发送给客户端程序的每块响应数据都应该带有约定好

的数据边界。

- ❑ 对于每个符合要求的数据块，需要依次计算它们的立方根、生成结果描述并返回给客户端程序。
- ❑ 需要鉴别闲置的通讯连接并主动关闭它们。闲置连接的鉴别依据是：在过去的10秒钟内，没有任何数据经该连接被传送到服务端程序。这可以非常有效地减少相关资源的消耗。

客户端程序的功能相对简单一些，可以被概括为：向服务端程序发送若干个代表了int32类型值的请求数据，接收服务端程序返回的响应数据并记录它们，下面是一些细节。

- ❑ 发送给服务端程序的每块请求数据都应该带有约定好的数据边界。
- ❑ 需要根据事先约定好的数据边界把接收到的响应数据切分成数据块。
- ❑ 在获得所有期望的响应数据之后，应该及时关闭连接以节省资源。
- ❑ 需要严格限制耗时，从开始向服务端程序发送请求数据到接收到所有期望的响应数据，其耗时不应该超过5秒钟，否则应该在报告超时错误之后关闭连接。这实际上是对服务端程序的响应速度的检验。

除上述需求之外，我们还希望把服务端程序和客户端程序放置在同一个命令源码文件中。所以，在实现它们的时候，我们不得不使用一些Go语言提供的并发和同步的手段，以使得它们能够并发地运行和适时地结束。别担心，这些方法我们在前面都已经使用过。并且，在后面的两章中，我们也会详细地讲解它们。

首先，我们在goc2p项目中专门建立了一个命令源码文件。这个源码文件在该项目中的相对路径是src/multiproc/socket/tcpsock.go。由于通讯两端的程序都在这一个源码文件中，所以我们完全可以把服务端程序所用的网络协议和地址声明为常量并存放在该文件中。此外，对于作为数据边界的分界符也应该是统一的。注意，如果通讯两端的程序是完全分离的（通常如此），那么最好把这类共用信息存放到第三方的存储介质中并对相关程序提供可访问的接口。

根据上面的描述，我们首先声明了3个常量：

```
const (
    SERVER_NETWORK = "tcp"
    SERVER_ADDRESS = "127.0.0.1:8085"
    DELIMITER      = '\t'
)
```

可以看出，为了简单起见，我们还是使用单字节字符作为数据边界。

我们为服务端程序和客户端程序各声明了一个入口函数，它们的名称是serverGo和clientGo。它们都是不接受任何参数且没有任何结果值的函数。这样做比把它们的代码都堆在命令源码文件的main函数中要好得多。这也是我们遵循单一职责原则的一个表现。另外，独立的入口函数让我们可以任意地选择两端程序的执行方式（并发或串行）。

下面，我们来编写serverGo函数的函数体。首先要做的就是根据给定的网络协议和地址创建一个监听器，代码如下：

```
var listener net.Listener
listener, err := net.Listen(SERVER_NETWORK, SERVER_ADDRESS)
```

```

if err != nil {
    printLog("Listen Error: %s\n", err)
    return
}
defer listener.Close()
printLog("Got listener for the server. (local address: %s)\n", listener.Addr())

```

注意，我们在这段代码中加入了一条defer语句，并用它来保证在serverGo函数结束执行之前关闭监听器。这样我们就不用在每条返回语句之前都添加一条listener.Close()语句了。并且，我们也不用担心万一发生运行时恐慌的时候监听器不能被关闭。另外，读者可能已经注意到，在这段代码中有一个名为printLog函数。这个函数实际上使我们为了更好地记录日志而编写的一个辅助函数。它的调用方法与fmt.Printf函数完全一致。这样做是为了分离将来很可能发生变化的日志记录操作。当前，我们仅仅简单地把日志打印到标准输出。但如果我们日后想完善日志的记录格式或者更改日志的记录方式（比如把日志记录到文件或者数据库），那么仅仅改动printLog函数的函数体中的代码就可以了。把可能频繁变化和基本不变的代码分离开来是非常重要的代码编写及优化手段。它可以有效地避免在程序维护过程中的散弹式修改。printLog函数的声明如下：

```

func printLog(format string, args ...interface{}) {
    fmt.Printf("%d: %s", logSn, fmt.Sprintf(format, args...))
    logSn++
}

```

我们通过连用fmt代码包中的两个函数很方便地实现了在原有的日志记录项之上添加内容的功能。其中logSn是我们在当前源码文件中声明的一个包级私有的变量。它代表了每个日志记录项的序号。这纯属是为了我们在后面讲解的日志的时候能够方便一点。

在成功获得到监听器之后，我们就可以开始等待客户端的连接请求了。请看下面的代码：

```

for {
    conn, err := listener.Accept() // 阻塞直至新连接到来
    if err != nil {
        printLog("Accept Error: %s\n", err)
    }
    printLog("Established a connection with a client application. (remote address: %s)\n",
        conn.RemoteAddr())
    go handleConn(conn)
}

```

请注意for代码块中的最后一条语句。这条语句是一条go语句。它是我们所说的Go语言提供的并发手段之一。go handleConn(conn)语句意味着要启动一个新的Goroutine（或称Go程）来并发的执行handleConn函数。在服务端程序中，这通常是非常有必要的。为了快速、独立地处理已经建立的每一个连接，我们应该尽量让这些处理过程被并发地执行。否则，当我们处理已建立的第一个连接的时候，后续连接就只能排队等待，尽管它们可能已经达到很长时间了。这相当于完全串行地处理众多连接，这样做的效率是非常低下的。并且，只要对其中的某一个连接的处理因某些原因被阻塞了，后续的所有连接就都无法得到处理。这时，服务端程序就等于完全丧失了主要功能。这是非常糟糕的情况。如果阻塞状态永远不被改变，那么这种糟糕的状况也一定会延续下去。因此，对于服务端程序而言，采用并发的方式处理连接是必然的选择。

既然每一个连接都是由handleConn函数处理的，那么我们就来看看怎样编写它的实现。它的简单声明（不包含其函数体）如下：

```
func handleConn(conn net.Conn)
```

它仅接受一个代表了连接的net.Conn类型值。由于我们可以把响应数据通过这个net.Conn类型值传递给客户端程序，所以handleConn函数无需再返回结果。另一个更客观的原因是，handleConn函数作为go语句的一部分，即使它返回了结果值也不会有任何意义。实际上，go语句中的函数向调用方传递结果值的方式是与众不同的。我们在后面的章节再说明这种独特的方式。

函数handleConn首先要做的肯定是试图从连接中读取数据。注意，这类读取操作应该处在循环之中。也就是说，服务器端程序应该不断的尝试从已建立的连接中读取数据。这样才能保证尽量及时地处理和响应请求。请看下面的这段代码：

```
for {
    conn.SetReadDeadline(time.Now().Add(10 * time.Second))
    strReq, err := read(conn)
    if err != nil {
        if err == io.EOF {
            printLog("The connection is closed by another side. (Server)\n")
        } else {
            printLog("Read Error: %s (Server)\n", err)
        }
        break
    }
    printLog("Received request: %s (Server)\n", strReq)
    // 省略若干条语句
}
```

上面的for代码块并不是handleConn函数的函数体中的全部。我们暂时只在这里展示它的一部分。for代码块中的第一条语句的作用是实现上面所说的关闭闲置连接的功能需求的一部分。其中的SetReadDeadline函数的调用方法我们应该已经很熟悉了。超时错误的发生就意味着当前连接已经可以被判定为闲置连接。这时，我们会记录日志并通过break语句退出当前的for语句块的执行。至于关闭连接的操作，我们在后面会看到。现在接着往下看，第二条语句中read函数也是我们编写的一个辅助函数。该函数的功能是从连接中读取一段以数据分界符为结尾的数据。它的完整声明如下：

```
func read(conn net.Conn) (string, error) {
    readBytes := make([]byte, 1)
    var buffer bytes.Buffer
    for {
        _, err := conn.Read(readBytes)
        if err != nil {
            return "", err
        }
        readByte := readBytes[0]
        if readByte == DELIMITER {
            break
        }
        buffer.WriteByte(readByte)
    }
}
```

```

    }
    return buffer.String(), nil
}

```

我们把`readBytes`变量的值的长度初始化为1的原因是，防止从连接值中读出多余的数据从而对后续的读取操作造成影响。我们从连接上每读取出一个字节的数据都要检查它是否是数据分界符。如果不是，就继续读取下一个字节。如果是就停止读取并返回结果。这样就不会把当前字节流中的第一个数据分界符后面的数据提前读取出来。如果提前读取发生了，那么我们下一次调用`read`函数的时候就无法得到一个完整的数据块了。另外，为了暂存当前数据块中的字节，我们用到了一个`bytes.Buffer`类型值。这通常比使用一个`[]byte`类型值来存储一个不定长的字节流更加实用和高效。还记得吗？如果当前连接已经被关闭，那么连接值的`Read`方法在被调用之后会返回一个与`io.EOF`变量的值相等的错误值。因此，鉴于`read`函数中对该`Read`方法返回的错误值的处理方式，我们在调用`read`函数之后，也应该对其返回的错误值做一样的相等性判断。这一点已经在前面示例中的`for`代码块中展示出来了。

认真读过前面内容的读者可能会想到通过调用`bufio.NewReader`函数得到一个针对当前连接的缓冲读取器。如果能想到这一点真的很好。不过对于当前的场景来说，缓冲读取器是不适合的。为什么这么说呢？简单来说，这是由于我们把`conn.Read`封装在了`read`函数中。我们先看看使用一个使用了缓冲读取器`read`函数版本是什么样子的，代码如下：

```

// 千万不要使用这个版本的read函数！
func read(conn net.Conn) (string, error) {
    reader := bufio.NewReader(conn)
    readBytes, err := reader.ReadBytes(DELIMITER)
    if err != nil {
        return "", err
    }
    return string(readBytes[:len(readBytes)-1]), nil
}

```

这很诱人。因为这个版本的`read`函数减少了一多半的代码。但是，这里面却埋藏了一个陷阱。这与缓冲读取器中的缓存机制有关。在很多时候，它会读取比足够多更多一点的数据到其中的缓冲区中。这就产生了我们前面提到的提前读取的问题。当然，如果我们每次都从同一个缓冲读取器中读取数据块的话，肯定是没有问题的。但是，在这里，我们对`read`函数的每一次调用都会导致一个新的针对当前连接的缓冲读取器被创建出来。我们实际上是在使用不同的缓冲读取器试图从同一个连接上读取的数据。这显然会造成一些问题，因为没有任何机制来协调它们的读取操作。本应留给后面的缓冲读取器读取的数据却被提前读取到了前面的缓冲读取器的缓冲区中。并且，由于我们不会再使用前面的这些缓冲读取器读取数据，所以这些被提前读取的数据实际上是被废弃了。这不但会导致一些数据块的不完整，甚至还可能会使一些数据块被漏掉。对于像本示例中的这种长度很短的小数据块而言更是如此。综上所述，我们决不能使用这个版本的`read`函数！不过，如果我们确实需要使用缓冲读取器也不是没有办法。方法很简单，即我们删除掉`read`函数，直接在`for`代码块之前初始化一个缓冲读取器，并且保证在`for`循环中总是使用同一个缓冲读取器来读取数据。这不但可以规避之前提到的所有问题，也可以避免多次创建缓冲读取器所带来的资

源浪费。读者可以沿着这一思路尝试对现有的这个for代码块进行重构。

如果读者刚才去重构for语句块了，那么真的值得一赞。现在，我们接着看for语句块中的第二部分：

```
for {
    // 省略若干条语句
    i32Req, err := convertToInt32(strReq)
    if err != nil {
        n, err := write(conn, err.Error())
        if err != nil {
            printLog("Write Error (written %d bytes): %s (Server)\n", err)
        }
        printLog("Sent response (written %d bytes): %s (Server)\n", n, err)
        continue
    }
    f64Resp := cbrt(i32Req)
    respMsg := fmt.Sprintf("The cube root of %d is %f.", i32Req, f64Resp)
    n, err := write(conn, respMsg)
    if err != nil {
        printLog("Write Error: %s (Server)\n", err)
    }
    printLog("Sent response (written %d bytes): %s (Server)\n", n, respMsg)
}
```

这部分的代码实现的功能是检查数据块是否可以被转换为一个int32类型值，如果能被转换就立即计算它的立方根，否则就向客户端程序发送一条错误信息。其中，convertToInt32函数实现了尝试转换数据块的功能，而cbrt被用于计算立方根。它们的函数体中涉及了一些与Socket无关的代码包的使用。因此，我们在这里略过对它们的讲解。不过，我们有必要简要说明一下write函数。它的声明如下：

```
func write(conn net.Conn, content string) (int, error) {
    var buffer bytes.Buffer
    buffer.WriteString(content)
    buffer.WriteByte(DELIMITER)
    return conn.Write(buffer.Bytes())
}
```

有了编写read函数的经验，我们编写write函数会很轻松。同样是使用一个bytes.Buffer类型值暂存数据，不过这次存储的是将要发送出去的数据，而不是已经接收到的数据。bytes.Buffer类型针对不同形式的数据提供了不同的写入方法，这很方便。注意，我们在每次发送的数据的后面都要追加一个数据分界符。这样才能形成一个两端程序均可识别的数据块。另外，bytes.Buffer类型值的Bytes方法会把其中存储的所有数据以字节切片的形式返回给调用方。我们正好可以用这个字节切片作为conn.Write方法的参数值。由于write函数的结果声明列表与conn.Write方法的完全相同，所以在write函数的最后我们直接返回后者的结果就可以了。

回到for代码块中。在数据块转换出错的情况下，我们直接把错误信息发送给了客户端程序，并根据发送的结果记录了日志。之后，我们要做的就是读取下一个数据块并试图转换它。这就是紧接在后面的continue语句所起到的作用——放弃执行后面的语句并开始下一次迭代。而如果数

据转换成功，我们就会计算数据块代表的int32类型值的立方根。然后就是生成结果描述并把它发送给客户端程序。发送操作同样用到了write函数。

至此，handleConn函数的主体（那个for代码块）已经被我们实现了。不过，还要注意，当它执行结束的时候应该把连接关闭。它被结束执行可能是由于主体已经执行结束，也可能是某些代码引发了一个运行时恐慌。不论怎样，把当前连接及时关闭掉都是一件很重要的事情。这也捎带着满足了关闭闲置连接的需求。还记得吗？当当前连接被判断为闲置连接的时候，read函数会返回非nil的错误值，并且那个for代码块中唯一的一条break语句会被执行。在这种情况下，我们肯定会用到defer语句，像这样：

```
defer conn.Close()
```

为了最大程度地保证连接的及时关闭，我们应该把这条defer语句放置在handleConn函数的函数体的开始处。

好了，handleConn函数和serverGo函数的编写已基本完成。下面我们来编写clientGo函数。clientGo函数的简单声明是这样的：

```
func clientGo(id int)
```

该函数接受一个名为id的、int类型的参数。接受这样一个参数只是因为要在运行多个客户端程序的场景下在日志中区分它们。我们不用太过在意它。clientGo函数应该首先试图与服务端程序建立连接，代码如下：

```
conn, err := net.DialTimeout(SERVER_NETWORK, SERVER_ADDRESS, 2*time.Second)
if err != nil {
    printLog("Dial Error: %s (Client[%d])\n", err, id)
    return
}
defer conn.Close()
printLog("Connected to server. (remote address: %s, local address: %s) (Client[%d])\n",
    conn.RemoteAddr(), conn.LocalAddr(), id)
time.Sleep(200 * time.Millisecond)
```

可以看到，如果连接不成功，那么就在记录日志之后直接返回。这也就意味着客户端程序的执行的结束。要使连接成功，最基本的条件就是连接操作应该在服务端程序已经启动的情况下进行。由于客户端程序和服务端程序处在同一个命令源码文件中，所以需要一点小技巧。这个我们稍后再讲。

下面那一条defer语句保证了在clientGo函数的执行将要结束的时候当前的连接会被关闭。这对于两端的程序都是有好处的。另外，解释一下，让客户端程序“睡眠”200毫秒纯属是为了两端程序记录的日志看起来更清晰一些。因为它们会出现在同一台计算机的标准输出上。

现在一切准备就绪，我们开始编写发送请求数据的代码。为了让两端的程序在体现出它们的功能和整体流程之后就结束执行，我们把客户端程序发送的请求数据块的数量定为5个。这需要声明一个变量，以便在发送数据块和接收数据块的时候都以此作为迭代次数。另外，为了满足检验服务端程序的响应速度的需求，我们还要在发送和接收操作开始之前设置一下超时时间。据此，有两行代码需要首先被编写出来，像这样：

```
requestNumber := 5
conn.SetDeadline(time.Now().Add(5 * time.Millisecond))
```

发送数据块的代码并不难编写，我们已经在实现serverGo函数的时候编写过类似的代码了。用来发送数据块的for代码块如下：

```
for i := 0; i < requestNumber; i++ {
    i32Req := rand.Int31()
    n, err := write(conn, fmt.Sprintf("%d", i32Req))
    if err != nil {
        printLog("Write Error: %s (Client[%d])\n", err, id)
        continue
    }
    printLog("Sent request (written %d bytes): %d (Client[%d])\n", n, i32Req, id)
}
```

其中，标准库代码包rand中的函数Int31可以生成一个随机的int32类型值。使用这样的随机值，既可以省去我们专门设计一个整数序列的时间，也可以更充分地体现出服务端程序的效能。读者还记得fmt.Sprintf函数吗？它可以把任意个任意类型的值转换为具有给定格式的string类型值。在很多时候，这比使用其他转换方法更加简单方便。在这里，我们用它来把一个int32类型值转换成一个string类型值。至于此for代码块中的其他语句，应该就不用更多的解释了吧？如果读者忘记了它们的含义和作用，那就再看一眼前面对handleConn函数的说明吧。

在把所有的5个请求数据块都发送出去之后，客户端程序应该开始着手接收响应数据块的事情了。实现这一功能的代码与服务端程序中接收请求数据块的代码如出一辙。我们要把这些代码放置在一个for代码块里面。这是因为我们需要在接收到所有预期的响应数据块之后，及时关闭当前连接并结束客户端程序的执行。这里所说的for代码块是这样的：

```
for j := 0; j < requestNumber; j++ {
    strResp, err := read(conn)
    if err != nil {
        if err == io.EOF {
            printLog("The connection is closed by another side. (Client[%d])\n", id)
        } else {
            printLog("Read Error: %s (Client[%d])\n", err, id)
        }
        break
    }
    printLog("Received response: %s (Client[%d])\n", strResp, id)
}
```

在编写完serverGo、clientGo以及相关函数之后，我们还需要考虑一件事情，那就是怎么样协调服务端程序、客户端程序以及main函数的执行。只要main函数的执行结束了，当前的这个与命令源码文件tcpsock.go对应的进程也就会随即消失。所以，我们要让main函数等待serverGo函数和clientGo函数都执行完毕之后再结束执行。这样，我们就要使用到前面提到过的sync.WaitGroup类型值了。为了让服务端程序和客户端程序都能使用到该值，我们要把该值声明为一个全局变量。当然，为了遵循开放封闭原则，该变量应该是包级私有的。据此，这个变量的声明如下：

```
var wg sync.WaitGroup
```

现在，我们开始编写main函数的函数体。为了让服务端程序和客户端程序能够并发地运行，我们应该分别使用go语句执行serverGo函数和clientGo函数。并且，客户端程序运行的时机应该在服务端程序开始运行并已准备好接收新连接之后。因此，我们应该让这两条go语句的执行之间有一点时间间隔。500毫秒的间隔时间在这里是足够的。根据上面的简单分析，main函数的第一个版本是这样的：

```
func main() {
    go serverGo()
    time.Sleep(500 * time.Millisecond)
    go clientGo(1)
}
```

要使我们前面声明的变量wg能够真正起到作用，我们还需要对现有的serverGo函数、clientGo函数以及第一个版本的main函数进行改造。下面是改造后的main函数：

```
func main() {
    wg.Add(2)
    go serverGo()
    time.Sleep(500 * time.Millisecond)
    go clientGo(1)
    wg.Wait()
}
```

如果我们只运行一个服务端程序和一个客户端程序的话，我们调用wg的Add方法的时候应该以2作为参数。这表示main函数只需等待上述两个程序运行完毕后即可。注意，该调用语句必须出现在这两个程序被运行之前。另外，我们在这里说的“等待”的操作是由调用语句wg.Wait()代表的。

如果我们不对serverGo函数和clientGo函数做出修改，当main函数被执行的时候，它将会永远在wg.Wait()语句处被阻塞。至于原因，我们在第一次接触sync.WaitGroup类型值的时候（6.2.4节）已经有所说明。我们要在serverGo函数和clientGo函数的函数体的最前面都加入一条语句：

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

这样，在这两个函数都被执行结束的时候main函数即可从wg.Wait()语句处继续往下执行了。

至此，在tcpsock.go文件中的示例的编码工作全部完成。现在我们来运行一下这个示例。在运行该示例之后标准输出上将出现如下内容：

```
1: Got listener for the server. (local address: 127.0.0.1:8085)
2: Connected to server. (remote address: 127.0.0.1:8085, local address: 127.0.0.1:51036) (Client[1])
3: Established a connection with a client application. (remote address: 127.0.0.1:51036)
4: Sent request (written 11 bytes): 1298498081 (Client[1])
5: Sent request (written 11 bytes): 2019727887 (Client[1])
6: Sent request (written 11 bytes): 1427131847 (Client[1])
7: Sent request (written 10 bytes): 939984059 (Client[1])
8: Sent request (written 10 bytes): 911902081 (Client[1])
9: Received request: 1298498081 (Server)
10: Sent response (written 44 bytes): The cube root of 1298498081 is 1090.972418. (Server)
11: Received request: 2019727887 (Server)
12: Sent response (written 44 bytes): The cube root of 2019727887 is 1264.050100. (Server)
13: Received request: 1427131847 (Server)
14: Sent response (written 44 bytes): The cube root of 1427131847 is 1125.869444. (Server)
15: Received request: 939984059 (Server)
```

```
16: Sent response (written 42 bytes): The cube root of 939984059 is 979.580571. (Server)
17: Received request: 911902081 (Server)
18: Sent response (written 42 bytes): The cube root of 911902081 is 969.726809. (Server)
19: Received response: The cube root of 1298498081 is 1090.972418. (Client[1])
20: Received response: The cube root of 2019727887 is 1264.050100. (Client[1])
21: Received response: The cube root of 1427131847 is 1125.869444. (Client[1])
22: Received response: The cube root of 939984059 is 979.580571. (Client[1])
23: Received response: The cube root of 911902081 is 969.726809. (Client[1])
24: The connection is closed by another side. (Server)
```

从这24个日志记录项中，我们可以清晰地看到两个程序共同完成的示例流程。第1~3个日志记录项反应出了服务端程序的启动过程，以及它与唯一的一个客户端程序的连接过程。第4~8个日志记录项表示该客户端程序连续向服务端程序发送了5个请求数据块。第9个和10个日志记录项表示服务端程序接到了第1个请求数据块，并在进行相应处理后向客户端程序发送了相应的结果描述。在这之后的8项日志记录则体现了服务端程序对之后到达的4个请求数据块的处理情况。而证明客户端程序已收到全部的5个结果描述的是第19~23个日志记录项。最后一项日志记录是服务端程序发出的，它表明了客户端程序在收到所有结果描述之后主动地关闭了与服务端程序建立的连接。综上所述，这些日志记录项所体现出的流程细节均正如我们所愿。

读者应该能从这个完整示例中学习到怎样使用Go语言提供的Socket编程API编写能够相互通讯的程序。虽然我们的示例将服务端程序和客户端程序置于同一个进程之中，但是在绝大多数应用场景中通讯两端的程序是由不同的进程代表的。在很多情形下，它们往往不是在同一台计算机上甚至不在同一个子网络中的两个程序。可以说，使用Socket接口的程序可以在网络中的任何地方与另一个同类程序进行通讯。并且，这些程序可以是由不同的编程语言编写的。

在Go语言标准库中，一些实现了某种网络通讯功能的代码包都是以net代码包所提供的Socket编程API为基础的。其中最有代表性的就是net/http代码包。它以此为基础实现了TCP/IP协议栈的应用层协议HTTP，并为我们提供了非常好用的API。这些API让我们可以非常方便地编写出满足一般性需求的Web应用程序。

除net代码包之外，标准库代码包net/rpc中的API为我们提供了在两个Go语言程序之间建立通讯和交换数据的另一个种方式。这种方式被称为远程过程调用（Remote Procedure Call）。这个代码包中的程序也是基于TCP/IP协议栈的。它们也使用到了net包以及net/http包提供的API。

6.3 多线程编程

多线程编程是一种比多进程编程更加灵活和高效的并发编程方式。绝大多数现代操作系统也已经对它提供了支持。在Unix的世界中，POSIX标准中定义的线程及其属性和操作方法已经被广泛认可和遵循。Linux操作系统作为Unix世界中的一员当然也提供了以POSIX标准中定义的线程（以下简称POSIX线程）为中心的各种系统调用。在Linux操作系统中，最贴近POSIX线程标准的线程实现被称为NPTL（Native POSIX Threads Library）。除了更加贴近POSIX标准，POSIX线程的出现的主要目的是对Linux操作系统原有的线程实现进行大幅改进。自2.6版本的Linux操作系统内核起，NPTL已经逐渐成为了其默认的线程实现。

这里顺带地解释一下POSIX。POSIX是Portable Operating System Interface of Unix的缩写，中译名为可移植性操作系统接口。它是由美国的电气电子工程师学会（IEEE）为了提高运行在各种类Unix操作系统下的应用程序的可移植性而开发的一套规范。该规范之后被美国国家标准协会（ANSI）和国际标准化组织（ISO）标准化。

为了更加通用，我们在这里只对POSIX线程及其相关概念进行介绍。这也是Go语言并发编程模型在Linux操作系统下真正使用的内核接口。

在本节中，我们首先会对POSIX线程的基本定义和概念进行说明。然后，我们将讨论一些与线程有关的关键问题，比如线程间的同步方法、线程安全性以及线程本地存储，等等。

Go语言的并发编程模型在底层是由操作系统所提供的线程库支撑的。所以，我们很有必要在讲解该并发编程模型之前对多线程编程进行一番介绍。撰写本小节的目的不但在于让读者对这个当今最主流的并发编程方法有所了解，更是为了向读者提供理解Go语言并发编程模型所必需的基础知识。

6.3.1 线程

一个线程可以被看作是在某个进程中的一个控制流。一个进程至少会包含一个线程，因为其中至少会有一个控制流持续运行。因而，一个进程的第一个线程会随着它的启动而被创建。这个线程被称为该进程的主线程。当然，一个进程也可以包含多个线程。这些线程都是由当前进程中已存在的线程所执行的相应系统调用（更确切的说，是pthread_create函数）创建出来的。拥有多个线程的进程可以并发的执行多个任务，并且即使某个或某些任务被阻塞也不会影响到其他任务的正常执行。这可以大大改善程序的响应时间和吞吐量。另一方面，线程不可能独立于进程存在。一个线程必属于某一个进程。它的生命周期不可能逾越其所属进程的生命周期。

一个进程中的所有线程都拥有自己的线程栈，并以此存储自己的私有数据。这些线程的线程栈都被包含在操作系统内核分配给其所属进程的虚拟内存地址中。然而，一个进程中的很多资源都会被其中的所有线程共享。这些被线程共享的资源包括在当前进程的虚拟内存地址中存储的代码段、数据段、堆、信号处理函数，以及当前进程所持有的文件描述符，等等。正因为如此，同一进程中的多个线程运行的一定是同一个程序（当然，具体的控制流程和执行的函数可能会不同），并且在它们之间共享数据也是非常轻松和自然的事情。此外，创建一个新线程也不会像创建一个新进程那样耗时费力，因为在其所属进程的虚拟内存地址中存储的代码、数据和资源都不需要被复制。

操作系统内核提供了若干个系统调用以使应用程序能够管理当前进程中的所有线程。此外，应用程序还可以通过相应的系统功能协调这些线程的运行。这些系统功能是由一些同步原语代表的。

下面，我们就对这些与线程紧密相关的知识进行更详细的阐述。

1. 线程的标识

和进程一样，每个线程也都有属于它自己的ID。这类ID也被称为线程ID或者TID。但与进程不同，线程ID在系统范围内可以不是唯一的，但在其所属进程的范围内必须是唯一的。不过，Linux操作系统的线程实现则确保了每个线程ID在系统范围内的唯一性。另外，当线程已不复存在之后，其线程ID是可以被其他线程复用的。

线程的ID是由操作系统内核分配和维护的。应用程序一般无需对它过多的关注。如果应用程序依赖线程ID,那么将会给它的移植带来困扰。不过,在我们对应用程序进行调试的时候,线程ID是非常有用的。它们可以帮助我们区分不同的线程。

2. 线程间的控制

我们已经知道,系统中的每个进程都有它的父进程,而由某个进程创建出来的进程都被称为该进程的直接子进程。与这种家族式的树状结构不同,同一个进程中的任意两个线程的关系都是平等的。也就是说,它们之间并不存在层级关系。任何线程都可以对其所属进程中的其他线程进行有限的管理。这里所说的有限的管理主要有以下4种。

- 创建线程:主线程在其所属进程启动的时候被创建。因此,它的创建并不在此论述的范围之内。这里仅指对其他线程的创建。我们已经说过,任何线程都可以通过调用系统调用 `pthread_create` 来创建新的线程。为了言简意赅,我们自此把调用系统调用或函数的线程简称为调用线程。在创建新线程的时候,调用线程需要给定新线程将要执行的函数以及传入该函数的参数值。由于代表该函数的参数被命名为 `start`,因此我们通常称这个函数为 `start` 函数。`start` 函数是可以有返回值的。我们可以在其他线程中通过与新线程的连接得到在该新线程中执行的 `start` 函数的返回值。如果新线程创建成功,调用线程会得到新线程的ID。
- 终止线程:线程可以通过多种方式终止其所属进程中的其他线程。其中一种方式就是调用系统调用 `pthread_cancel`。``pthread_cancel` 函数的作用是取消掉给定的线程ID代表的线程。更明确地讲,它会向目标线程发出一个请求,要求它立即终止执行。但是,该函数只是发送请求并立即返回,而不会等待目标线程对该请求做出响应。至于目标线程什么时候对此请求做出响应、做出怎样的响应,则取决于另外的因素(比如目标线程的取消状态及类型)。在默认情况下,目标线程总是会接受线程取消请求,但等到时机成熟(执行到某个取消点)的时候目标线程才会去响应线程取消请求。
- 连接已终止的线程:此操作由系统调用 `pthread_join` 代表。该函数会一直等待(或者说阻塞)与给定的线程ID对应的那个线程的终止,并把该线程执行的 `start` 函数的返回值告知给调用线程。如果目标线程已经处于终止状态,那么该函数会立即返回。这就像是把调用线程放置在了目标线程的后面,当目标线程把流程控制权交出时,调用线程会接过流程控制权并继续执行 `pthread_join` 函数调用之后的代码。这也是把这一操作称为“连接”的缘故之一。实际上,如果一个线程是可被连接的,那么在它终止之时就必须被连接,否则它就会变成一个僵尸线程。僵尸线程不但会导致系统资源的浪费,还会使其所属进程的可创建线程数量被无意义地减少。
- 分离线程:将一个线程分离意味着,它不再是一个可被连接的线程。而在默认情况下,一个线程总是可以被其他线程连接的。分离操作的另一个作用是让操作系统内核在目标线程终止时自动进行清理和销毁工作。注意,分离操作是不可逆的。也就是说,我们无法使一个不可被连接的线程变回到可被连接的状态。不过,对于一个已处于分离状态的线程执行终止操作仍然会起作用。分离操作由系统调用 `pthread_detach` 代表。它接受一个代表了线程ID的参数值。

当然，一个线程对自身也可以进行两种控制：终止和分离。线程终止自身的方式有很多种。在线程执行的start函数中执行return语句会使该线程随着start函数的执行结束而终止。需要注意的是，如果在主线程中执行了return语句，那么当前进程中的所有线程都会被终止。另外，在任意线程中调用系统调用exit也会达到这种效果。另一种终止自身的方式是，显式地调用系统调用pthread_exit。与执行return语句或调用exit函数不同，如果在主线程中调用了pthread_exit函数，那么只有主线程自己会被终止，而其他线程仍然会照常运行。这是很重要的区别。线程分离自身与分离其他线程的方式并无不同，即调用pthread_detach函数。区别仅在于调用线程传递给该函数的线程ID是自己的ID还是其他线程的ID。

3. 线程的状态

从前面的论述中可知，一个线程在从被创建到被终止的完整生命周期中也经常会在多个状态之间切换。由于线程只是进程中的一个控制流，所以对进程的状态描述几乎都适用于线程。不过，正如前面所说的那样，线程的状态及其切换规则还是有它的特点的，如图6-13所示。

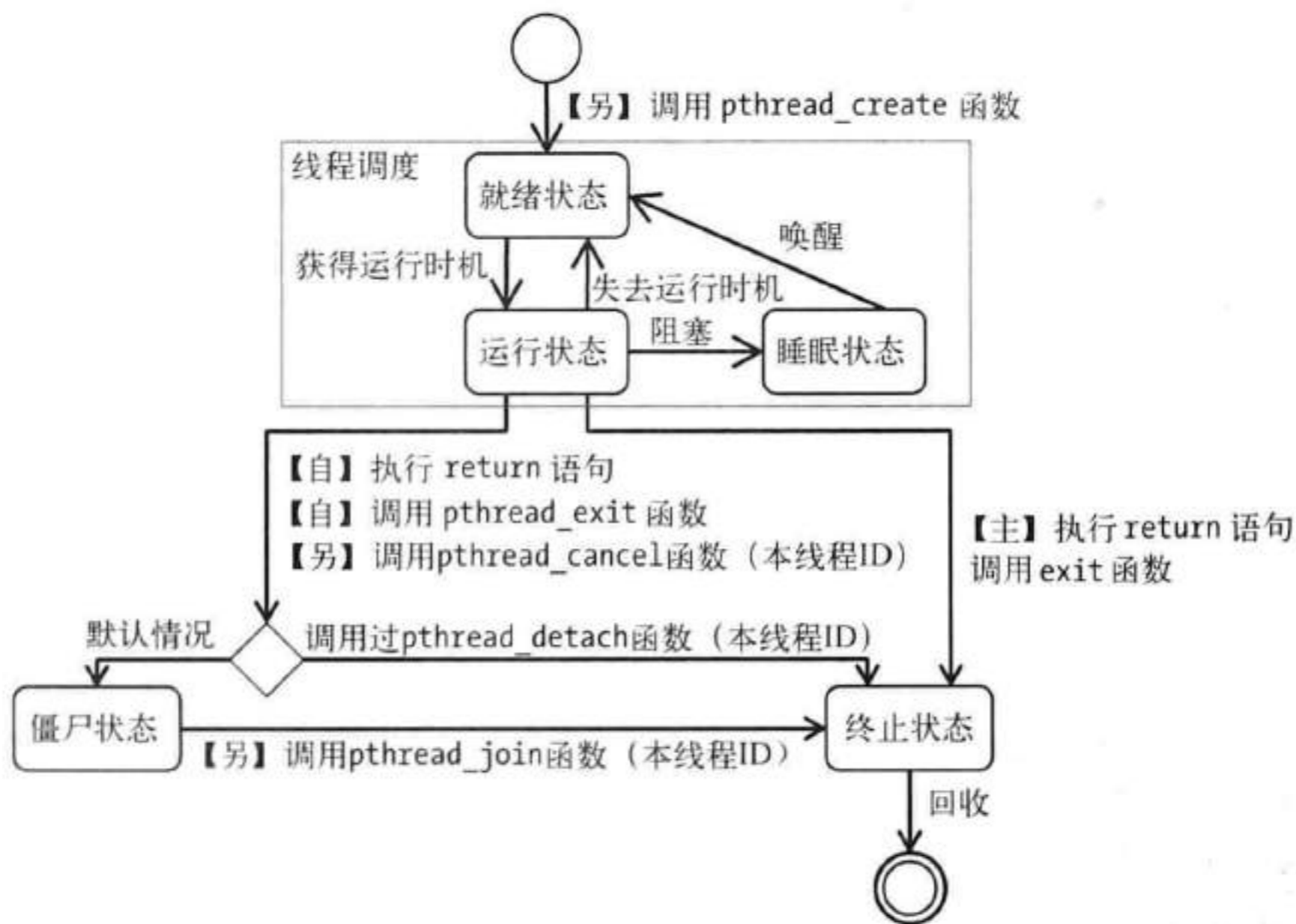


图6-13 Linux内核线程的状态转换

图6-13是以系统调用的视角来描述线程在不同状态之间的转换的。在图6-13中，一些系统调用操作描述或语句执行操作描述的左侧被插入了一个前缀。这些前缀都由中文的方括号“【”和“】”括起来以示强调。前缀“【另】”代表描述的操作是由当前进程中的其他线程执行的。前缀“【自】”代表描述的操作是由当前线程执行的。而前缀“【主】”则代表描述的操作是由主线程执行的。如果在操作描述的左侧没有前缀，那么就说明该操作可能由当前进程中的任何线程执行。另外，请注意，当其他线程调用pthread_cancel函数或pthread_join函数，以及任一线程调用pthread_detach函数的时候，传递给它们的参数值代表的都应该是当前线程的ID。这样，它们的

执行才会对当前线程起作用。

如图6-13所示,线程在被创建出来之后就会进入就绪状态。处于就绪状态的线程会等待被运行的时机。一旦该线程被真正地运行,它就会由就绪状态转换至运行状态。正在运行的线程可能会由于某些原因被阻塞,进而由运行状态转换至睡眠状态。这里可能的原因包括但不限于等待未完成的I/O操作、等待还未接收到的信号、等待获得互斥量,以及等待某个条件变量。后两个原因都属于因同步而产生的线程阻塞。我们在后面会专门对它们进行说明。当阻塞线程等待的那个事件或条件发生或满足时,该线程会被唤醒。也就是说,它会从睡眠状态转出。但是,它并不会直接进入运行状态,而是先进入就绪状态以等待运行时机。如果CPU正处于空闲状态,那么它会被立即运行。此外,处于运行状态的线程有时也会因CPU被其他线程抢占而失去运行时机,从而转回至就绪状态并等待下一个运行时机。操作系统内核的调度器会按照一定的算法和策略使线程在这3个状态之间转换。线程在其生命周期的大部分时间里都会处于就绪状态、运行状态或睡眠状态之中。

在当前线程自我终结或者其他线程向当前线程发出取消请求且取消时机已到之后,当前线程就会试图进入终止状态。不过,如果当前线程之前没有被分离过,并且此时并没有其他线程与它连接,那么当前线程就会进入到僵尸状态而非终止状态。当且仅当有其他线程与之连接之后,当前线程才会从僵尸状态转换至终止状态。处于终止状态的线程才会被操作系统内核回收。不过,有两种操作可以直接使当前线程进入终止状态,而不管它是否已经被分离。在任意线程中调用`exit`函数以及在主线程中执行`return`语句,都不但会使其所属进程中的所有线程立即终止,还会结束该进程的运行。

4. 线程的调度

在前面的论述中,我们只是一笔带过了线程调度方面的内容。实际上,在线程的生命周期中,操作系统内核对线程的调用是非常核心的部分。正因为有了调度器的实时调度和切换,才能给我们一种众多线程被并行运行的幻觉。调度器会把时间划分成极小的时间片并把这些时间片分配给不同的线程,以使众多线程都能有机会在CPU上运行。一个线程什么时候能够获得CPU时间,以及它能够在CPU上被运行多久,都属于调度器的工作范畴。线程调度(也被称为线程间的上下文切换)是一项非常复杂的工作。因此,我们在这里只对线程调度的最基本的规则和策略进行阐述。

线程的执行总是趋向于CPU受限或I/O受限。换句话说,一些线程需要花费一定的时间使用CPU进行计算,而另外一些线程则会花费一些时间等待相对较慢的I/O操作的完成。一个被用于计算16位整数的14次方根的线程属于前者,而一个等待人类用户通过敲击键盘提供输入数据的线程则属于后者。但是,在通常情况下,一个线程的趋向性并不总是那么清晰。因此,调度器往往需要去猜测它们。这是非常困难的任务。调度器会依据它对线程的趋向性的猜测把它们分类,并让I/O受限的线程具有更高的动态优先级以优先使用CPU。这倒不是为了讨好与I/O受限的线程进行交互的数据输入人。实际上,调度器根本无法判断I/O操作的数据输入方是人类、磁盘还是网络中的另一个应用程序。真正的原因是,调度器认为I/O操作往往会花费很长的时间,应该让它们尽早地开始执行。事实上也确实如此。这也是为了让众多线程运行得更加有效率。在人决定下一个要敲击的按键、磁盘在磁道中定位簇或者网卡从网络中接收数据帧的时候,CPU可以腾出手来为其他线程服务。这些时间已经可以让CPU见缝插针地完成很多事情了。

注意，我们刚刚所说的线程的动态优先级是可以被调度器实时调整的。而与之相对应的线程的静态优先级则只能由应用程序指定。如果应用程序没有显式地指定一个线程的静态优先级，那么它将被设定为0。调度器并不会改变线程的静态优先级。线程的动态优先级就是调度器在其静态优先级的基础上调整得出的。它在线程的运行顺序上起到了关键的作用。而线程的静态优先级则决定了线程单次能够在CPU上运行的最长时间，亦即调度器分配给它的时间片的大小。后面我们会看到线程的时间片所起到的作用。

所有等待使用CPU的线程会被按照动态优先级从高到低的顺序排入到与该CPU对应的运行队列中。因此，下一个被运行的线程总是动态优先级最高的那一个。实际上，每一个CPU的运行队列中都包含两个优先级阵列。其中的一个被用于存放正在等待运行的线程。我们暂且称之为激活的优先级阵列。而另一个则被用于存放已经运行过但还未完成的线程。我们暂且称之为过期的优先级阵列。更确切地讲，优先级阵列是一个由若干个链表组成的数组。一个链表只会包含具有相同优先级的线程，而一个线程也只会放到与它的优先级相对应的那一个链表中。当一个线程被放入某个优先级阵列的时候，它实际上是被放到了与它的优先级相对应的那个链表的末尾处，如图6-14所示。

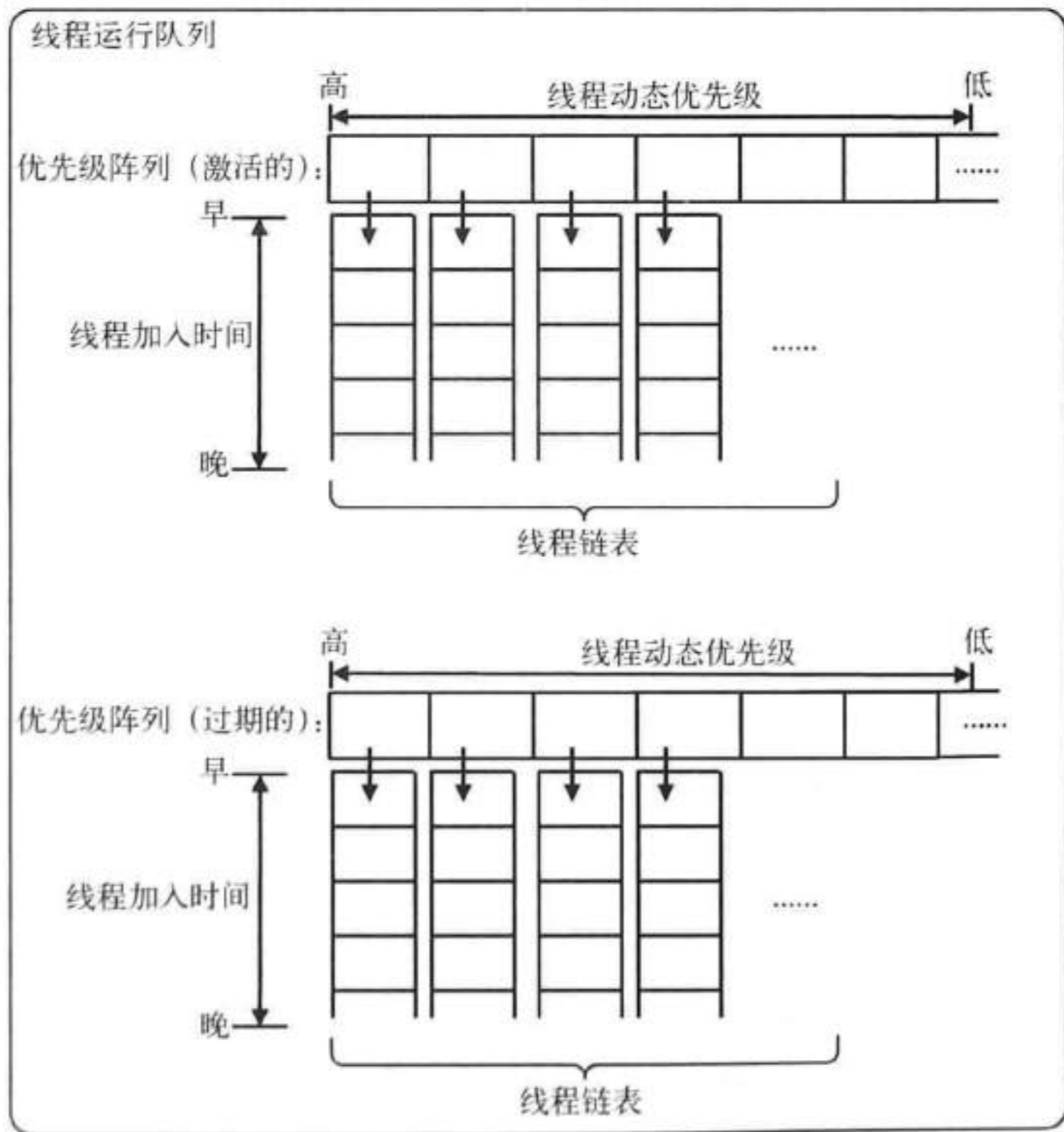


图6-14 线程运行队列的内部结构示意图

下一个被运行的线程总是会从激活的优先级阵列中选出。如果调度器发现某个线程已经占用

了CPU很长时间（该时间只会小于或等于给予该线程的时间片），并且激活的优先级阵列中还有优先级与它相同的线程在等待运行，那么调度器就会让那个等待的线程在CPU上运行。被换下的线程会被排入过期的优先级阵列。当激活的优先级阵列中没有待运行的线程的时候，调度器就会把这两个优先级阵列的身份互换，即之前的激活的优先级阵列成为新的过期的优先级阵列。而之前的过期的优先级阵列则会成为新的激活的优先级阵列。这样，之前被放入过期的优先级阵列的线程就又有机会被运行了。

当然，线程不会总是在就绪状态和运行状态之间徘徊，它还有可能被阻塞而进入睡眠状态。处于睡眠状态的线程不能够被调度和运行。换句话说，它们会从运行队列中被移除。线程的睡眠状态也可以被细分为可中断的睡眠状态和不可中断的睡眠状态。不过，我们在这里并不打算区分它们。相信读者已经通过阅读上一节对它们有所了解。

线程会因等待某个事件或条件的发生而被加入到对应的等待队列中，并随即进入睡眠状态。当事件或条件发生时，内核会通知对应的等待队列中的所有线程。这些线程会因此而被唤醒并从等待队列转移至适当的运行队列中。调度器往往会稍稍调高被唤醒的线程动态优先级。这算是一个小小的额外恩惠，以使这类线程能够更早地被运行。

如果当前计算机上有多个CPU，那么平衡它们之间的负载也将会是调度器的职责之一。调度器会尽量使一个线程在一个特定的CPU上运行。这有很多好处，比如维护高速缓存的高命中率以及高效使用就近的内存，等等。然而，有时候一个CPU需要运行太多的线程以至于造成了多CPU之间的负载的不平衡。也就是说，一些CPU过于忙碌或另一些CPU则被闲置。在这种情况下，调度器会把一些原本在较忙碌的CPU上运行的线程迁移至其他较空闲的CPU上运行。由于内核会为每个CPU都建立一个运行队列，所以线程的这种迁移并不困难。事实上，每个运行队列中都会保存对应CPU的负载系数。调度器可以根据这一系数了解并调整各个CPU的负载。当然，CPU负载平衡的调度逻辑相当复杂，负载系数仅仅是冰山一角。但由于篇幅原因，我们只介绍这么多。

总体来说，操作系统内核的调度器就是使用若干策略对众多线程在CPU上的运行进行干涉，以使得操作系统中的各个任务都能够有条不紊地进行，同时还要兼顾效率和公平性。从线程的角度来看，调度器是通过协调各个线程的状态来达到调度目的的。单台计算机上的资源是很有限的，尤其是以CPU为代表的计算资源。因此，操作系统内核对线程的调度非常重要。随着开发者们对操作系统内核的不断改进和完善，调度器日趋强大，同时也愈加复杂。但是，它的总体目标和基本规则原则是未曾变化的。相信读者通过以上介绍已经对线程调度有了一个宏观上的认识。

5. 线程实现模型

线程的实现模型主要有3个，分别是：用户级线程模型、内核级线程模型和两级线程模型。它们之间最大的差异就在于线程与内核调度实体（Kernel Scheduling Entity，简称KSE）之间的对应关系上。顾名思义，内核调度实体就是可以被内核的调度器调度的对象。在很多文献和书中，它也被称为内核级线程。它是操作系统内核的最小调度单元。下面，我们就来说说这3个线程实现模型的特点以及优劣。

□ 用户级线程模型：此模型下的线程是由用户级别的线程库全权管理的。线程库并不是内核的一部分。它只被存储在进程的用户空间之中。进程中的线程的存在对于内核来说是

无法感知的。显然，这些线程也不是内核的调度器的调度对象。对线程的各种管理的和协调完全是用户级程序的自主行为，与内核无关。应用程序在对线程进行创建、终止、切换或同步等操作的时候，并不需要让CPU从用户态切换到内核态。从这方面讲，用户级线程模型确实在线程操作的速度上存在优势。并且，由于对线程的管理完全不需要内核的参与，所以使得程序的移植性更强一些。但是，这一特点导致在此模型下的多线程并不能够被真正地并发运行。例如，如果线程在I/O操作过程中被阻塞，那么其所属进程也会被阻塞。这正是由线程无法被内核调度造成的。在调度器的眼里，进程是一个无法再被分割的调度单元，无论其中存在多少个线程。另外，即使计算机上存在多个CPU，进程中的多个线程也无法被分配给不同的CPU运行。对于CPU的负载均衡来说，进程的粒度太粗了。因而让不同的进程在不同的CPU上运行的意义也微乎其微。显然，线程的所谓优先级也会形同虚设。同一个进程中的所有线程的优先级只能由该进程的优先级来体现。同时，线程库对线程的调度完全不受内核控制，它与内核为进程设定的优先级是没有关系的。正因为用户级线程模型存在这些严重的缺陷，所以现代操作系统都不是使用这种模型来实现线程的。但是，在早期，以这种模型作为线程的实现方式的案例确实存在。由于包含了多个用户级线程的进程只与一个KSE相对应，因此这种线程实现模型又被称为多对一（M：1）的线程实现。

- 内核级线程模型：该模型下的线程是由内核负责管理的。它们是内核的一部分。应用程序对线程的创建、终止和同步都必须通过内核提供的系统调用来完成。进程中的每一个线程都与一个KSE相对应。也就是说，内核可以分别对每一个线程进行调度。由此，内核级线程模型又被称为一对一（1：1）的线程实现。一对一线程实现消除了多对一线程实现的很多弊端。在这样的实现之下，可以真正实现线程的并发运行。因为这些线程完全是由内核来管理和调度的。正如前文所述，内核可以在不同的时间片内让CPU运行不同的线程。内核在极短的时间内快速切换和运行各个线程使得它们看起来像正在被同时运行。即使进程中的一个线程由于某种原因进入到了阻塞状态，其他线程也不会受到影响并可以正常地运行。这也使得内核在多个CPU上进行负载平衡变得容易和有效。当然，如果一个线程与被阻塞的线程之间存在同步关系，那么它也可能会受到牵连。但是，这是一种应用级别的干预，并不属于线程本身的特质。同时，内核对线程的全权接管使操作系统在库级别几乎无需为线程管理做什么事情。这与用户级别线程模型形成了鲜明的对比。但是，内核线程的管理成本显然要比用户级线程高出很多。线程的创建会使用到更多的内核资源。并且，像创建线程、切换线程，同步线程这类操作所花费的时间也会更多。如果一个进程包含了大量的线程，那么它会给内核的调度器造成非常大的负担，甚至会影响到操作系统的整体性能。因此，采用内核级线程模型的操作系统对一个进程中可以创建的线程的数量都有直接或间接的限制。尽管内核级线程模型有资源消耗较大、调度速度较慢等缺点，但是与用户级线程的实现方式相比它还是有较大的优势的。很多现代操作系统都是以内核级线程模型实现线程的，包括Linux操作系统。实际上，Linux操作系统的最新线程库实现（NPTL）为最小化内核级线程模型的劣势付出了巨大的努力。

这也使得在Linux操作系统中使用线程更加高效。

- 两级线程模型：两级线程模型的目标是取前两种模型之精华，并去二者之糟粕。它也被称为多对多（M：N）的线程实现。与其他模型相比，两级线程模型提供了更多的灵活性。在此模型下，一个进程可以与多个KSE相关联。这与内核级线程模型是相似的。但与内核级线程模型不同的是，进程中的线程（以下称之为应用程序线程）并不与KSE一一对应。这些应用程序线程可以被映射到同一个已关联的KSE上。首先，已被加载到进程的虚拟内存中的实现两级线程模型的线程库会通过操作系统内核创建多个内核级线程。然后，它再通过这些内核级线程对应用程序线程进行调度。大多数此类线程库都可以为实际运行的应用程序线程动态地分配若干个内核级线程。这样的设计显然使线程的管理工作更加复杂一些，因为这需要内核和线程库的共同努力和协作才能正确、有效地进行。但是，也是由于这样的设计，内核资源的消耗才得以大大减少，同时也使线程管理操作的效能有了不少的提高。因为两级线程模型的实现的复杂性，它往往不会被操作系统内核的开发者采纳。但是，这样的模型却可以很好地在编程语言层面上实现并发挥出其应有的作用。就拿Go语言来说，其并发编程模型就与两级线程模型在理念上非常类似，只不过它的具体实现方式更加高级和优雅一些。在Go语言的并发编程模型中，不受操作系统内核管理的独立控制流并不被叫作应用程序线程或者线程，而被称为Goroutine（也可以被称为Go程）。3种线程实现模型如图6-15所示。

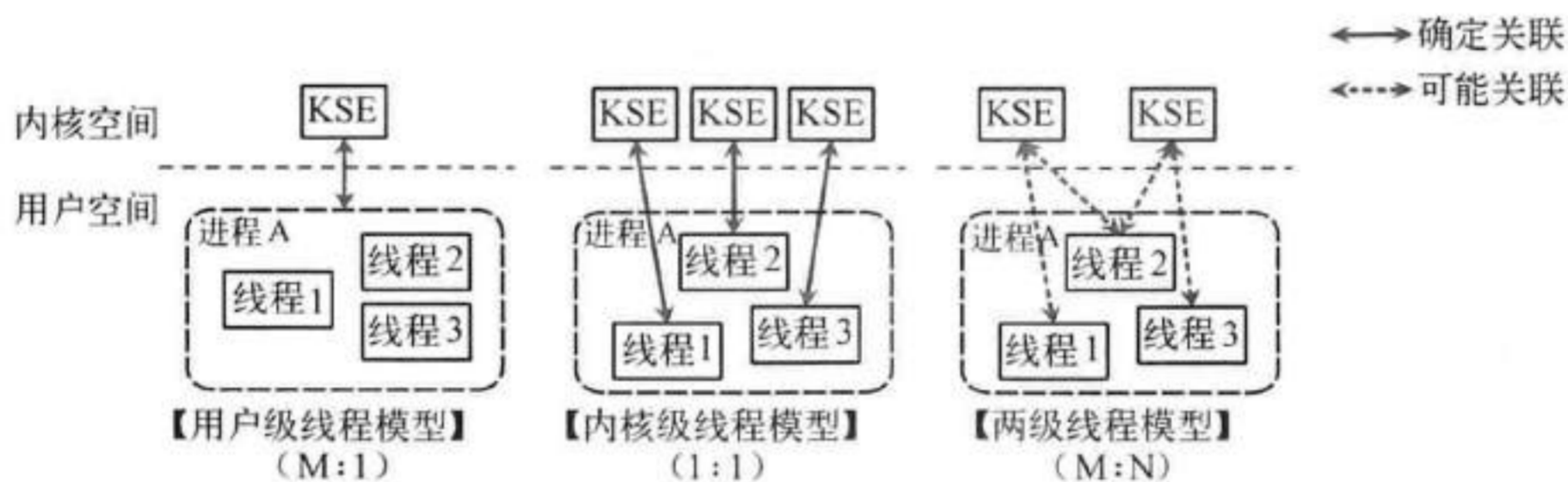


图6-15 3种线程实现模型

至此，我们已经讨论很多与线程及其实现模型有关的概念和知识。下面，我们会讨论一些更加具体的主题。它们在多线程编程的过程中是十分常用的，同时也是非常重要的。在我们使用Go语言进行并发编程的时候也会常常涉及它们。因此，了解它们既有利于我们对底层实现的理解，也会让我们在实际的编程过程中更容易作出正确的决策。

6.3.2 线程的同步

同步，永远是多线程编程中最核心和最重要的话题之一。在上一节，我们提及过一些与线程中的同步有关的概念和工具，比如临界区、原子操作以及互斥量，等等。在本小节，我们会对它们以及其他相关内容进行深入且详细的介绍。

总的来说,在多个线程之间采取同步措施,无非是为了让它们更好地协同工作或者维持共享数据的一致性。以后者作为目的的同步较为常见,但在以前者作为控制流管理手段的程序中同步的意义更大。而在实际的场景中,同步的目的则更可能是两者兼而有之的。就像我们在上一节讲解同步概念的时候举的那个计数器的例子一样。

1. 共享数据的一致性

包含多个线程的程序(以下简称多线程程序)多以共享数据作为在线程之间传递数据的手段。由于一个进程所拥有的相当一部分虚拟内存地址都可以被该进程中的所有线程所共享,因此这些被共享的数据也大多是以内存空间作为载体的。如果两个线程同时读取同一块被共享的内存但获取到的数据却是不同的,那么程序很可能就会出现某种错误。这是因为,共享数据的一致性往往代表着某种约定,而只有在该约定成立的前提下,多线程程序中的各个线程才能够使相应的流程被正确地执行。换句话说,如果操作共享数据的实际结果总是与我们约定的(或称期望的)操作结果相符,那么就可以说该共享数据的一致性得到了保证。而共享数据的一致性保证则是多线程程序中的各个控制流得以正确执行的前提。当然,即使这种约定是成立的,线程在执行流程的过程中也不一定不会出错。不过那就属于另外的情况了,并不在这里的讨论范围之内。支撑共享数据一致性的约定,一般会被直接地或间接地包含在我们对多线程程序的设计方案之中。因此,这种一致性的保证也关系到我们的程序设计方案能否被正确地实施。在上一节讲同步概念的时候,我们用了一定的篇幅描述了因共享数据的不一致而可能导致的种种错误。在多线程编程的过程中,我们总是要想方设法地保证共享数据的一致性,除非该共享数据永远只可会被同一个线程访问。

实际上,保证共享数据的一致性的最简单且最好的方法,就是使该数据成为一个不变量。例如,我们在程序中声明的常量就是一个绝对的不变量。常量是不可能被改变的,也就不可能出现不一致的情况。这是由编程语言来保证的。因此,无论当前程序中有多少个可能访问该常量的线程,我们都不需要采取任何措施。但是,把计数器变成一个常量是不现实的。一个可以并需要被改变的计数器只能被看作一个变量。我们需要通过额外的手段,来保证被多个线程所共享的变量的一致性。这才有了临界区这个概念。我们已经知道,临界区是只能被串行化地访问或执行的某个资源或某段代码。因而临界区也常被称为串行区域。保证临界区有效的最佳方式就是利用同步机制。在针对多线程程序的同步机制中包含了很多同步方法,包括我们之前提及过的原子操作和互斥量,也包括我们还未曾讲到的条件变量。在后面的内容中,我们会着重介绍两种可以帮助线程同步对共享数据的使用的方法(以下简称线程同步方法):互斥量和条件变量。

2. 互斥量

在同一时刻,只允许一个线程处于临界区之内的约束被称为互斥(mutex)。每一个线程在进入临界区之前都必须先锁定某个对象。只有成功锁定对象的线程才会被允许进入到临界区之内,否则就会被阻塞。这个对象被称为互斥对象或互斥量。

由上面这段描述可知,互斥量有两种可能的状态,即已锁定状态和未锁定状态。互斥量每次只能被锁定一次。也就是说,处于已锁定状态的互斥量不能被再次锁定。除非它已被解锁,否则任何线程都不能对它进行二次加锁。如果对一个已被锁定的互斥量进行加锁操作,那么这个操作必定会失败。成功锁定互斥量的线程会成为该互斥量的所有者。只有互斥量的所有者才能对该互

斥量进行解锁。从这个角度讲，多个线程对同一个互斥量的争相锁定也可以被看作是对该互斥量的所有权的争夺。从而，锁定可以被看作是对互斥量的获取，解锁也可以被看作是对互斥量的释放。对于互斥量来说，这两对术语具有相同的含义。在相关的资料和图书中，它们也会成对地出现。在本书中，我们主要使用“锁定”和“解锁”这对术语。

另一方面，线程在离开临界区的时候，必须要对相应的互斥量进行解锁。这样，其他为进入该临界区而被阻塞的线程才会被唤醒并有机会再次尝试锁定该互斥量。而在这些线程中，只可能有一个线程成功锁定该互斥量。注意，对同一个互斥量的锁定和解锁操作应该成对地出现。我们既不应该对一个互斥量进行重复锁定，也不应该对一个互斥量进行多次解锁。与后者相比，前者有时会造成严重的后果。

为了合理、安全地使用共享数据，我们应该把操作同一个共享数据的代码都置于一个或多个临界区之内，并使用同一个互斥量对它们进行保护。需要注意的是，在使用互斥量的过程中，我们必须遵循既定的使用规则。我们刚刚已经讲述了一些基本原则。不过，在后面，我们还会分别列举一些正例和反例，并借此进一步对相关规则进行说明。互斥量的使用会涉及操作系统提供的线程库中的一些函数。我们并不打算讲解这些函数，所以下面的示例主要以示意图的形式展现。

我们的第一个示例改编自那个与计数器有关的例子（参见6.2.2节）。我们将会当前的示例中给出针对那个计数器的同步问题的解决方案，如图6-16所示。

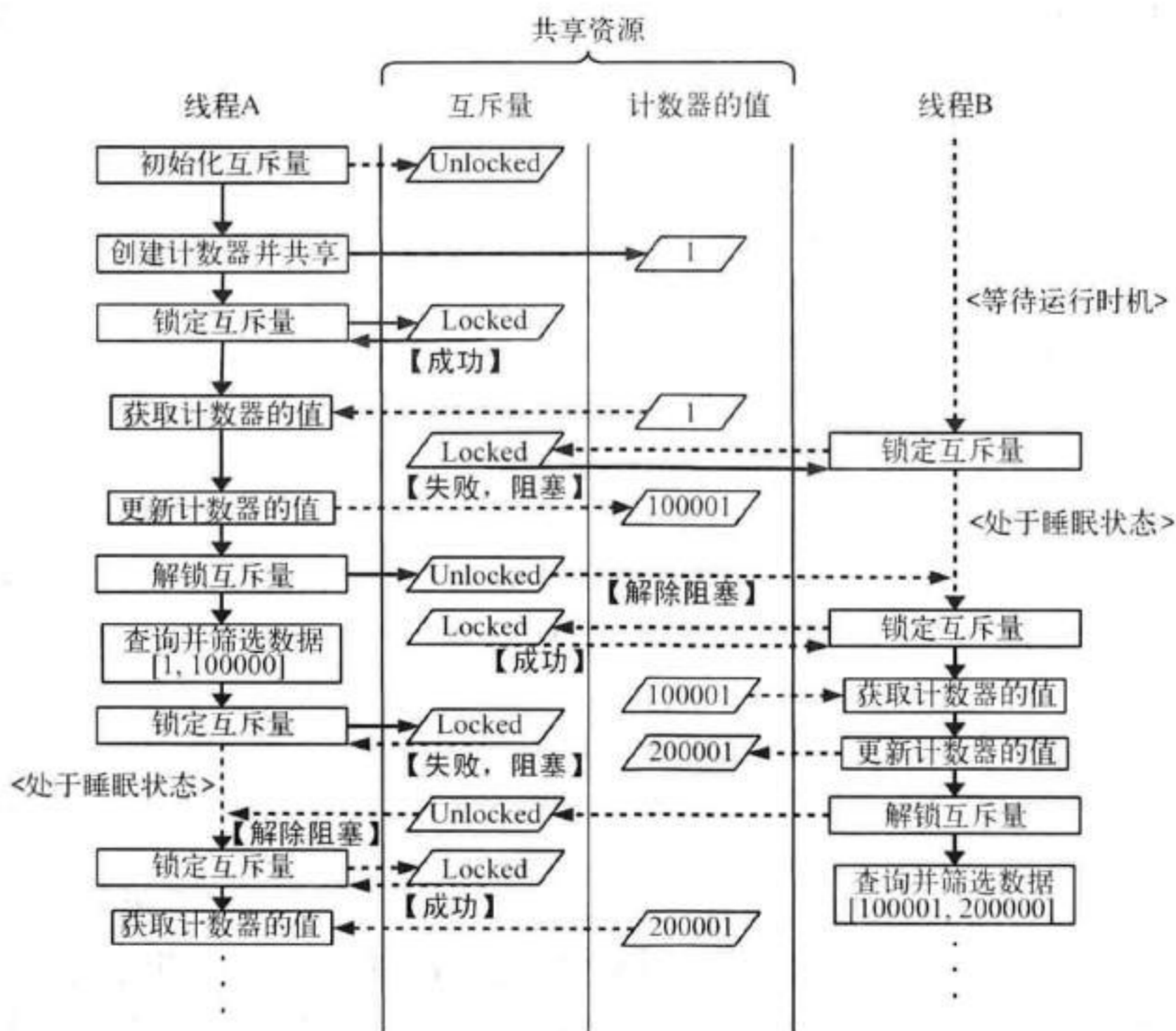


图6-16 互斥量保护下的计数器操作

图6-16展示了两个线程共同使用一个计数器的情形。在这一过程中，我们使用互斥量作为线程间同步的工具。这幅图的信息量有些大。不过别担心，我们下面对其中的要点逐一进行解释。

首先，互斥量与计数器一样也属于共享资源。互斥量必须能够被使用相应的共享资源的线程访问到。因此，代表互斥量的变量或常量一般不是局部的。不过，为了尽量少地暴露程序的实现细节，我们应该在满足上述要求的前提下最小化互斥量的访问权限。

其次，初始化互斥量的操作总是应该在任何线程真正使用它之前进行。从图6-16我们也可以看到，线程A执行的第一个操作即是初始化互斥量。经过初始化的互斥量会处于未锁定状态。注意，如果多个线程将要执行的代码中都包含了对同一个互斥量的初始化操作，那么我们必须保证该互斥量只会被初始化一次。操作系统的线程库中专门提供了满足此要求的函数。在Go语言中，我们也有很多可选择的实现方式。

在初始化互斥量并创建计数器之后，线程A开始使用计数器。它会获取并更新计数器的值。不过，在这之前，线程A会先试图锁定那个已被初始化过的互斥量。这是当前示例与源示例之间最主要也是最重要的区别。线程A欲锁定互斥量，而互斥量当时又处于未锁定状态，因此锁定操作会成功完成。从图6-16中可知，在这一操作完成后，互斥量处于已锁定状态（Locked）。请读者注意图中由中文的方括号“【”和“】”括起来的注解。在成功锁定互斥量之后，线程A开始放心大胆地对计数器的值进行操作。

注意，在线程A对计数器的值进行获取操作之后、更新操作之前，线程B被运行并准备使用计数器。这与源示例中发生的情形一样。不过，线程B在使用计数器之前也不得不先试图锁定互斥量。但是由于当时互斥量已处于锁定状态，而它又不能重复锁定，所以线程B对它的锁定操作会失败。并且，锁定操作的失败将导致线程B被阻塞并进入睡眠状态。直到线程A完成对计数器的值的获取和更新操作并解锁互斥量之后，线程B才会被唤醒并退出睡眠状态。被唤醒之后，线程B会立即再次试图锁定互斥量。由于这时的互斥量已处于未锁定状态（Unlocked），所以线程B的锁定操作会成功。之后，线程B开始操作计数器的值并处理相关数据。

互斥量对于每个想要锁定它的线程都是平等的。因此，线程A在处理完数据之后又立即开始争夺互斥量。但是这次它对互斥量的锁定也会失败，因为该互斥量已经被线程B锁定。这使得线程A被阻塞，直到线程B解锁互斥量。

由于我们对互斥量的合理使用，线程A和线程B都不会干扰到对方对计数器的使用。因而，原本存在于它们之间的竞态条件已被消除，使用互斥量的目的达到了。即使有更多的线程参与到其中也会是如此。另外，我们用互斥量保护的临界区中包含且仅包含了对计数器的值的获取、相加和更新操作。因此，这组操作的执行看上去具有了原子性。更确切地说，它们的执行是伪原子性的。之所以说是伪原子性，是因为它们在执行过程中是可以被中断的，并且它们既不与单一的汇编指令相对应也没有芯片级别的支持。实际上，能够使用得上真正的原子操作的应用场景并不多。所以，互斥量作为一个可以保证共享数据的一致性的工具常常会是首选。

对于这个示例，再强调两点。第一，对互斥量的初始化必须要保证唯一性。这永远是正确使用互斥量的第一个必要条件。第二，线程在离开临界区的时候必须要及时解锁互斥量，以免造成不必要的性能损耗甚至死锁。关于第二点，我们稍后还会举例说明。

上面的示例忽略了一个细节，那就是线程执行的程序在筛选数据之后要做的事情。在上一节我们说过，程序在每次数据筛选之后要把得到的数据集合写入文件，并在写入完成后关闭文件。注意，线程A和线程B执行的是相同的程序。因此，我们完全有必要对其中的文件操作进行同步，以免文件中的内容发生混乱。至此，我们在线程A和线程B执行的程序中使用了两个互斥量。在这里，我们把为了同步计数器操作的互斥量称为互斥量 α ，而把为了同步文件操作的互斥量称为互斥量 β 。互斥量 α 和互斥量 β 分别保护了两个毫不相干的临界区，而这两个临界区又分别仅包含了对于两个完全独立的共享资源的操作。也就是说，在这两个互斥量的作用域之间不存在任何的重叠。图6-17描绘了线程A和线程B争相进入两个在互斥量的保护之下的临界区的情形。其中，临界区 α 指代由互斥量 α 保护的临界区，而临界区 β 指代由互斥量 β 保护的临界区。

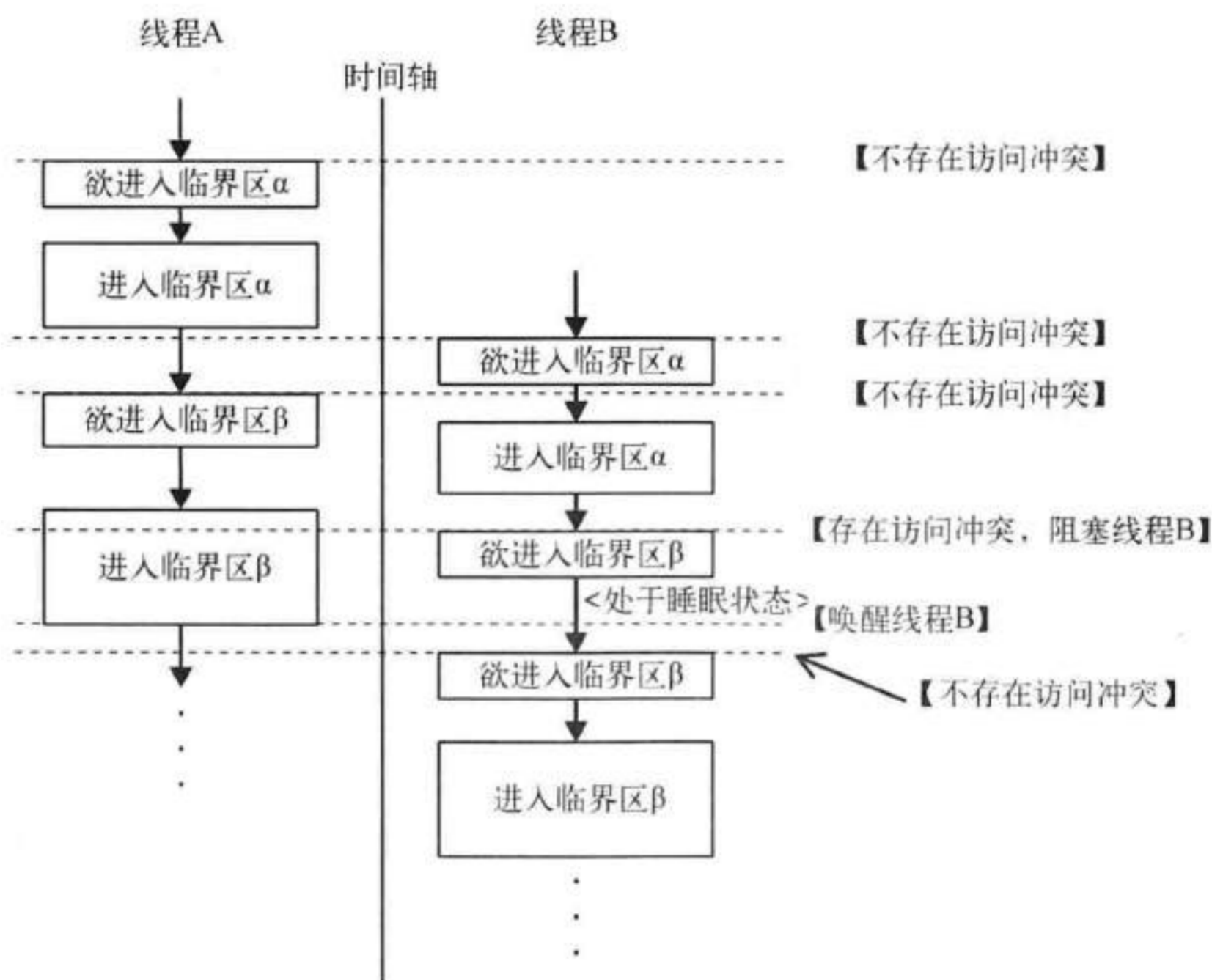


图6-17 互斥量保护下的临界区

从图6-17中，我们可以清晰地看到互斥量 α 和互斥量 β 所起到的作用。在线程B欲进入临界区 β 的时候，由于线程A正在临界区 β 之内执行，所以线程B被迫进入睡眠状态。在线程A离开临界区 β 的时候，线程B被唤醒并再次尝试进入临界区 β 。

图6-17描绘的同时使用多个互斥量的情形已经算是相当简单的了。因为互斥量 α 和互斥量 β 的作用域的互不重叠。加之线程在离开临界区之前总会及时解锁相应的互斥量，所以程序不会因使用这两个互斥量而产生死锁。并且，它们对程序的复杂度的负面影响也是非常有限的。不过，因为锁定和解锁互斥量也需要时间，所以使用互斥量本身确实会稍许降低程序的性能。但如果我们在这里将两个临界区合二为一并且只使用一个互斥量来保护的话，又可能会使线程等待进

入临界区的时间大大增加。相比之下，只使用一个互斥量会比使用两个互斥量更多的降低了程序性能。当然，这只是对于我们当前描述的这个程序来说的。使用互斥量的过程中总会存在着博弈和权衡。

在一般情况下，应该尽量少地使用互斥量。每个互斥量保护的临界区应该在合理范围内并尽量地大。但是，如果发现多个线程会频繁地出入某个较大的临界区，并且它们之间经常存在访问冲突，那么就应该把这个较大的临界区切分成若干个较小的临界区，并使用不同的互斥量加以保护。此举的意义是让等待进入同一个临界区的线程数变少，从而降低线程被阻塞的几率，并减少其处于睡眠状态的时间。这样就可以从一定程度上提高程序的整体性能。

请注意，如果在切分之后由不同的互斥量保护的临界区中包含了对同一个共享资源的同一种操作，那么此次临界区切分就是不成功的。我们要么重新考量，要么放弃切分。因为这种不对应的关系使互斥量的所谓的保护失去了意义。即使只是包含了对同一个共享资源的不同操作，我们也应该仔细考虑，怎样保证这些不同的操作在同时被执行的时候不会给共享资源或程序的正常流程造成破坏。例如，在前述的那个计数器的例子中，我们就不能把获取计数器的值的操作和更新计数器的值的操作分别置于两个临界区中。

除此之外，我们应该特别注意，尽量不要让不同的互斥量所保护的临界区重叠。因为这会大大增加死锁发生的几率。图6-18展示了因互斥量的使用不当而造成的死锁。

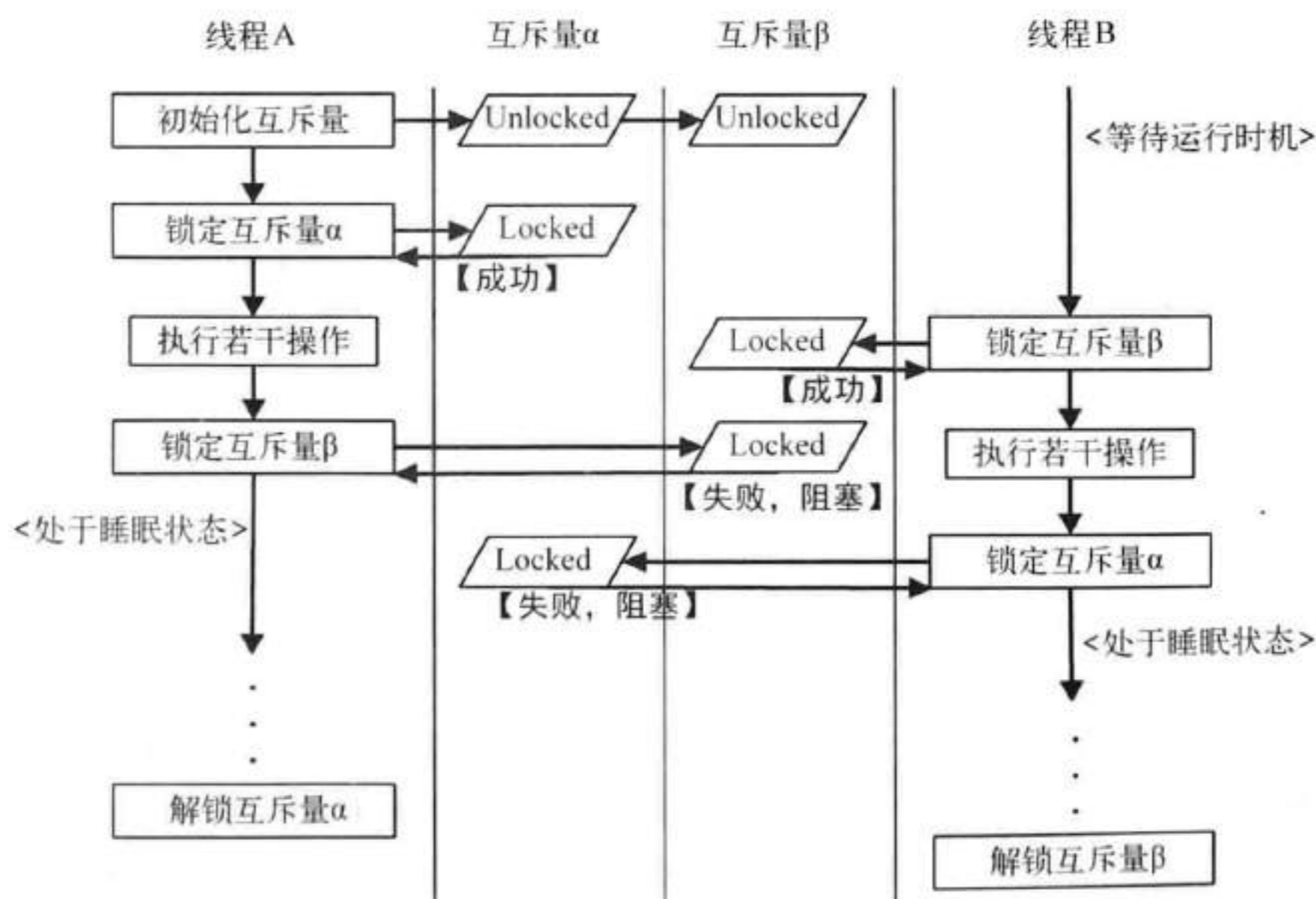


图6-18 互斥量的使用不当造成的死锁

如图6-18所示，线程A和线程B在一开始就分别锁定了一个互斥量 α 和互斥量 β 。而后，在它们还未释放自己所持的互斥量的情况下，就想去锁定已被另一方锁定的互斥量。线程A在成功锁定互斥量 β 之前不会解锁互斥量 α ，而线程B在成功锁定互斥量 α 之前也不会解锁互斥量 β 。

实际上,线程A永远不能解锁互斥量 α ,线程B也永远不能解锁互斥量 β 。它们相持不下并使另一方和自己永远处于睡眠状态。如此一来,死锁就产生了。并且,如果当前进程只包含了线程A和线程B,那么整个进程的运行就会停滞。即使它们不是当前进程所包含的全部线程,也会使该进程在功能和性能上都大打折扣。这同样是我们不希望看到的。

对于运行已停滞的进程,我们能做的只有强制重新启动它。这样不但会使所有的运行时数据丢失,还可能会造成各种不一致的状态。作为并发程序的设计者,我们应该坚决避免可预见的死锁的发生。我们已经知道,如果可以保证不同的互斥量所保护的临界区之间不存在任何重叠,那么就可以避免因互斥量的使用不当而造成的死锁。但是,如果出于某种原因,我们无法对此作出保证,或者必须要使用临界区重叠的多个互斥量,那又该怎么办呢?

在这种情况下,我们有两种通用的解决方法。其中的一个方法需要利用到操作系统提供的线程库的功能。它被叫作试锁定和回退。其核心思想是,如果在执行一个代码块的时候需要先后(顺序不定)锁定两个互斥量,那么在成功锁定其中一个互斥量之后应该使用“试锁定”的方法来锁定另一个互斥量。如果“试锁定”第二个互斥量的操作不成功,那么就把已锁定的第一个互斥量解锁,并重新对这两个互斥量进行锁定和“试锁定”。如果需要锁定的互斥量多于两个,那么应该总是先锁定其中的一个,然后再按照上面的方法“试锁定”其他互斥量并在必要时进行“回退”。这里的“试锁定”指的是操作系统的线程库所提供的一个函数。它会尝试对一个互斥量进行锁定。但是,若锁定失败则函数会直接返回一个错误码,而不是被阻塞在那里。这一点很重要,是避免产生死锁的关键。当然,“回退”环节也是很有必要的。因为,如果“试锁定”失败,就说明有其他线程已经先于当前线程进入或试图进入这个受多个互斥量保护的代码块。这时,当前线程应该知趣地退出对这些互斥量的争夺,并从头尝试锁定它们。这是为了避免因“试锁定”的使用而破坏互斥量和临界区的原本含义。虽然从理论上来看,这种方法几乎可以应对所有临界区重叠的情况,但是它却也大大增加了程序的复杂性。尤其是在分别对多个互斥量锁定的操作之间夹杂着其他操作的时候。我们在从头对多个互斥量进行锁定的同时往往还要考虑到其他状态的“回退”。而后者的复杂程度也决定着“回退”环节的可行性。另外,在使用试锁定和回退方法时,我们对多个互斥量解锁的操作的顺序最好要与锁定它们的顺序完全相反。特别是,一定要最后解锁那个被第一个锁定的互斥量。这可以大大减少“回退”的次数。

另一种通用解决方法更加廉价。它不需要用到更多的线程库函数,也不会像第一种方法那样对程序的复杂度有那么大的影响。它被称作“固定顺序锁定”。顾名思义,它的思路是在需要先后对多个互斥量进行锁定的场景下,总以固定不变的顺序锁定它们。就前面的示例而言,如果线程A和线程B总是先锁定互斥量 α 、再锁定互斥量 β ,那么对它们的锁定就不会造成死锁的发生。因为在成功锁定互斥量 α 之前,线程永远无法执行对互斥量 β 的锁定操作。从而避免了它们分别持有另一方想要锁定的互斥量的情况。另一方面,互斥量 α 和互斥量 β 的解锁操作的顺序与其锁定操作的顺序之间没有强制性的约束。这取决于具体的流程设计。在一般情况下,解锁操作的顺序是与其锁定操作的顺序相反的。这是因为我们在很多时候,需要保证在一个线程完全离开这些重叠的临界区之前,不会有其他同样需要锁定那些互斥量的线程进入到那里。这可能是由于这些临界区中的操作之间具有一些固有的关联性,也可能是由于需要保证这些操作的完整性。除上述

情况之外，对多个互斥量的解锁操作的顺序就显得不那么重要了。另外，如果在程序的某处还存在着单独使用其中的某个互斥量的情况，我们还应该仔细考量，看看是否可以通过“缩短”对它的锁定操作和解锁操作之间的“距离”来减小因使用它而给程序性能带来的负面影响。不过，请记住，我们总是应该先实现功能，然后再在必要的时候对程序的性能进行优化，否则很可能会降低程序逻辑的清晰度甚至失去原有设计的优势。

显然，这两种方法都能够有效地解决死锁的问题。但是，在解决问题的同时，我们也需要付出一些代价。前一种方法虽然更加通用但却会使程序更加复杂。虽然后一种方法既简单又实用，但是它却由于固定的操作顺序而降低了程序的灵活性。同时，它在适用场景方面也有些许限制。例如，在不能确定线程对多个临界区的访问顺序的情况下，我们就无法使用“固定顺序锁定”的方法来预防死锁。当然，我们不一定要进行这种非黑即白的设计决策。混用这两种方法有时候也会是不错的选择。比如，在访问顺序可控的临界区之上使用“固定顺序锁定”的方法，而在访问顺序不可控或者需要更大灵活性的地方使用“试锁定-回退”方法。不过，无论怎样，它们都属于下策，并且都是一种对不优雅或者不得以的设计的补救措施。我们应该总能够想到，保持共享数据的独立性是预防因使用互斥量而导致的死锁的最佳方法。如果共享数据之间的关联无法消除，那么我们应该尽可能地使多个临界区之间没有重叠。仅当这两点都无法得到保证的时候，我们才应该考虑使用“试锁定-回退”和“固定顺序锁定”的方法。

死锁几乎是我们在使用互斥量时需要特别注意的唯一问题。同时，它也是并发程序设计当中的最严重的问题之一。我们应该想尽一切办法避免它的发生。

互斥量简单而高效，并且适用于绝大部分的共享数据的同步场景。互斥量的实现会使用到机器语言级别的原子操作，并仅在锁定冲突的才会涉及系统调用的执行。这使得互斥量比其他同步方法（比如信号灯）的速度要快很多。

3. 条件变量

在我们可用的同步方法集中，还有一个可以与互斥量相提并论的同步方法——条件变量。与互斥量不同，条件变量的作用并不是保证在同一时刻仅有一个线程访问某一个共享数据，而是在对应的共享数据的状态发生变化时，通知其他因此而被阻塞的线程。条件变量总是与互斥量组合使用。互斥量为共享数据的访问提供互斥支持，而条件变量可以就共享数据的状态的变化向相关线程发出通知。当线程成功锁定互斥量从而访问到共享数据的时候，共享数据的状态并不一定正好满足它的要求。下面我们通过一个示例来了解条件变量的适用场景。

假设有一个容量有限的数据块队列和若干个会操作该队列的线程。我们可以把这里所说的数据块想象成具有明显边界的字节序列。其中的一些线程会生产数据块并把它们添加到数据块队列中，而另一些线程会消费数据块并把它们从数据块队列中删除。显然，这是一个经典的生产者-消费者问题。因为会有多个线程并发的访问这个数据块队列，所以我们应该想到把向数据块队列添加数据块的操作（以下简称添加操作）和从数据块队列中获取并删除数据块的操作（以下简称获取操作）都置于临界区之中，并用同一个互斥量加以保护。这样，我们就能保证在执行添加操作的线程（以下简称生产者线程）未完成添加操作之前，其他生产者线程和执行获取操作的线程（以下简称消费者线程）都无法进行相应的操作。同时也可以保证，一个数据块只会被某一个消

费者线程取走。因而，我们使用互斥量保证了队列中的每一个数据块的正确和完整。但是，即使有了互斥量的保护也可能发生以下两种情况。

第一种情况是生产者线程获得到互斥量，但却发现数据块队列已满，无法再添加新的数据块。这时，生产者线程可能会在临界区内等待，直到数据块队列有空余空间以容纳新的数据块。其中的等待行为往往是通过循环的判断数据块队列的已满状态来实现的。一旦判断的结果为假，循环就会立即被结束执行，紧随其后的就是添加操作，如图6-19所示。

生产者线程这样做会导致一个严重的问题。如果在当前线程第一次判断队列是否已满的时候结果就为真，那么它会一直循环的执行获取队列状态和判断队列是否已满这两个步骤，直到永远。由于当前线程在完成添加操作之前不会解锁互斥量，所以会使任何消费者线程都无法取走队列中的数据块。如果队列中的数据块不会被取走，判断队列是否已满的结果又怎么可能为假呢？注意，这样就产生了一个死循环！再次强调，我们应该尽力避免可预知的死锁的发生！由于上述流程的问题很明显，所以可以很容易改进它，如图6-20所示。

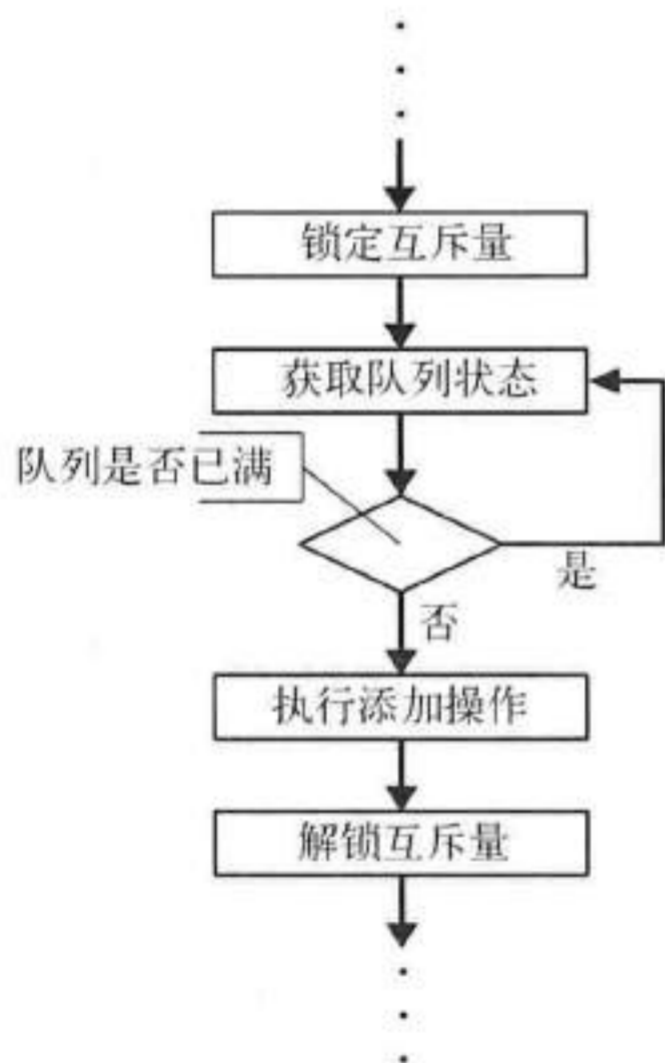


图6-19 生产者线程添加数据块的流程1

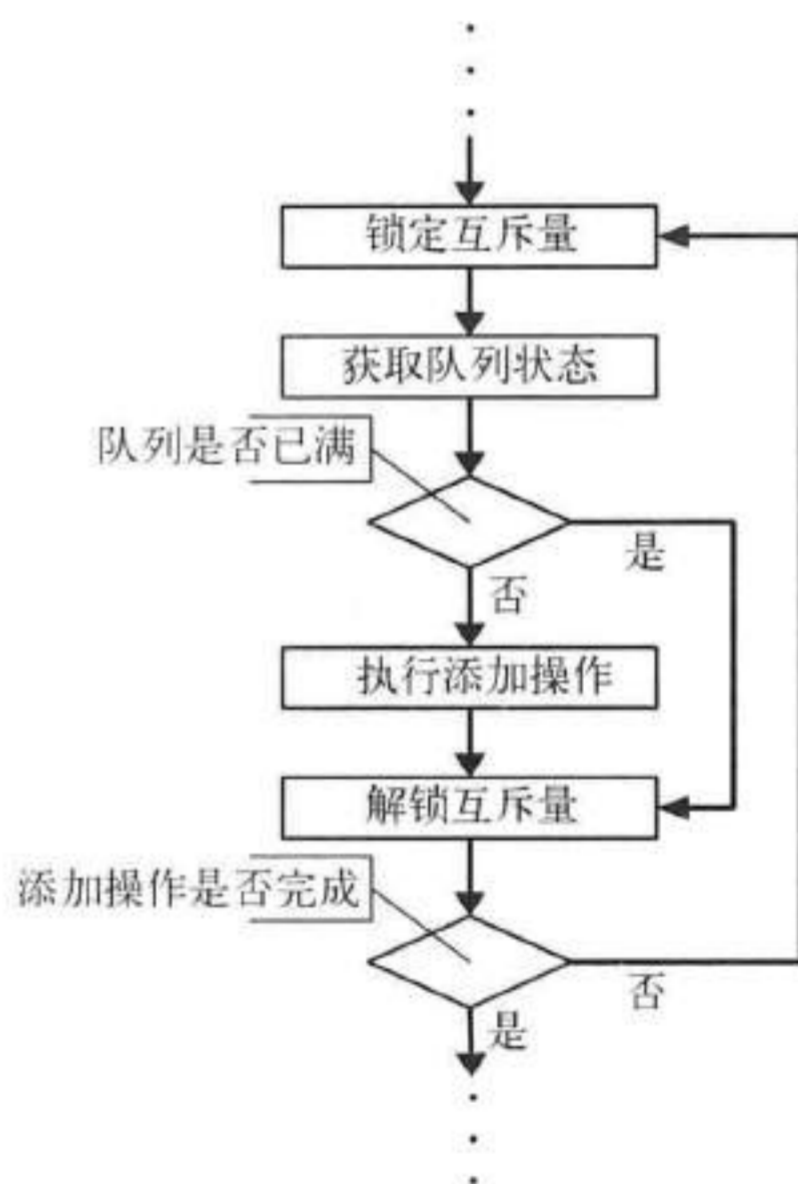


图6-20 生产者线程添加数据块的流程2

从图6-20中可知，我们现在把锁定和解锁互斥量的操作与其他相关操作一起放到了循环体里面。这使得前面造成的那个死锁问题被消除了。因为无论队列是否已满、添加操作是否可以进行，互斥量都会被解锁。不过，这个进行过改进的流程还是存在缺陷的。如果队列长时间处于已满状态，则这里的循环体会被执行很多次。其中包括针对互斥量的那两个操作。这样的循环无疑会造成CPU资源的浪费。另外，如果生产者线程有很多，那么当判断队列是否已满的结果为假的时候添加操作就一定能够成功吗？请读者带着这些问题接着往下看。顺便提一句，如果添加操作

在被执行的时候引发了一个异常并使该流程被意外终结，那么互斥量就永远不会被解锁。这又应该怎样解决？我们在第8章讲Go语言中的互斥量的时候再讨论这个问题。

第二种情况是消费者线程获得互斥量，但却发现数据块队列为空。这时，消费者不得不迭代的检查队列的状态，并在队列中存在数据块的时候再尝试获取它。相应的流程如图6-21所示。

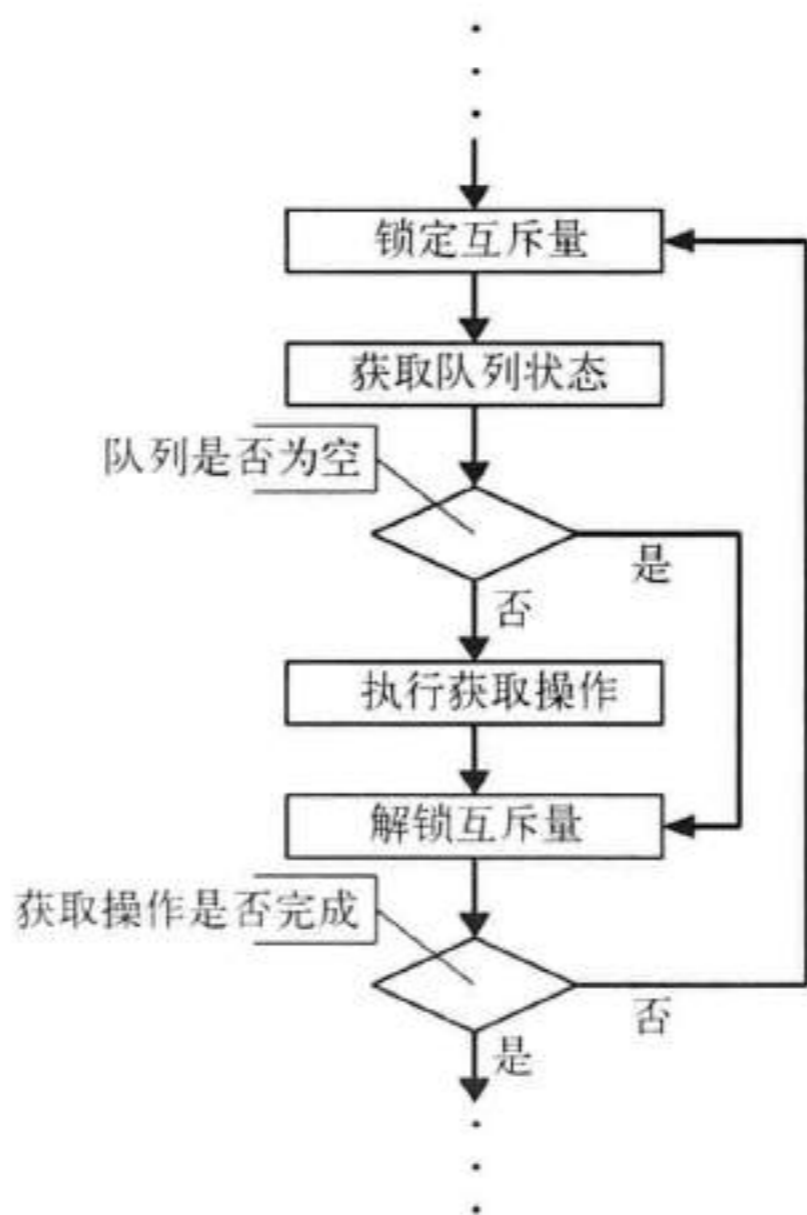


图6-21 消费者线程获取数据块的流程

显然，这一流程与前面那个经过改进的生产者线程的流程非常类似，并且也存在着相同的缺陷。我们就不再在这里进行重复的描述了。

如果添加操作和获取操作能够在条件不满足时自行阻塞，并且一旦条件满足就立即进行相应的操作就好了，不是吗？这样的话，我们就不用在外部使用互斥量了。但遗憾的是，我们在这里虚构出来的数据块队列并没有这样的能力。幸好，我们能够使用条件变量来解决这个问题。更确切地说，条件变量恰恰是解决这类问题的利器。

与互斥量相同，我们在使用一个条件变量之前必须创建和初始化它。同样地，条件变量的初始化必须要保证唯一性。另外，条件变量在被真正使用之前还必须要与某个互斥量进行绑定。这与它的具体操作有关。我们可以在一个条件变量之上进行的操作有3种。

- 等待通知 (wait): 等待通知的意思是阻塞当前线程，直至收到该条件变量发来的通知。
- 单发通知 (signal): 单发通知的意思是让该条件变量向至少一个正在等待它的通知的线程发送通知，以表示某个共享数据的状态已经改变。与其他操作相同，这里的英文名称signal也是相应函数的名称的一部分。但是，请注意，这里的signal与我们在上一节所讲的信号是不同的两个概念。为了避免混淆，我们把条件变量发送的信号称为通知。

□ 广播通知 (broadcast): 广播通知的意思是指让条件变量给正在等待它的通知的所有线程都发送通知, 以表示某个共享数据的状态已经改变。

实际上, 等待通知的操作并不是简单的阻塞当前线程。在该操作被执行的时候会先解锁与该条件变量绑定在一起的那个互斥量, 然后再使当前线程阻塞。这里隐藏着两个细节。第一个细节是, 只有在当前的共享数据的状态不满足条件时, 才应该执行等待通知操作, 而检查共享数据的状态需要受到互斥量的保护。也就是说, 检查共享数据状态的操作和等待通知操作都需要在相应的临界区内进行。因此, 等待通知操作中包含的解锁互斥量的操作是会造成任何问题的。它没有违反互斥量的基本使用原则, 即不能对一个互斥量进行重复解锁。第二个细节与等待通知操作中包含解锁互斥量操作的原因有关。如果等待通知操作在阻塞当前线程之前不对互斥量进行解锁, 那么其他线程也无法进入相应的临界区。这与我们在前面讲述的生产者线程的最初流程是相似的。如果当前线程因等待共享数据状态的改变而被阻塞, 而其他的线程也因互斥量的阻挡而无法改变共享数据的状态, 那么会立即形成死锁。因此, 在阻塞当前线程之前, 等待通知操作必须先解锁互斥量。更重要的是, 等待通知操作所包含的解锁互斥量的操作和阻塞当前线程的操作共同形成了一个原子操作。也就是说, 在等待通知操作使当前线程被阻塞之前, 任何线程都无法锁定相应的互斥量。这样做的原因与其他线程在进行相关操作时所处的环境是有关系的, 如图6-22所示。我们在稍后讲单发通知操作的时候再说明它。

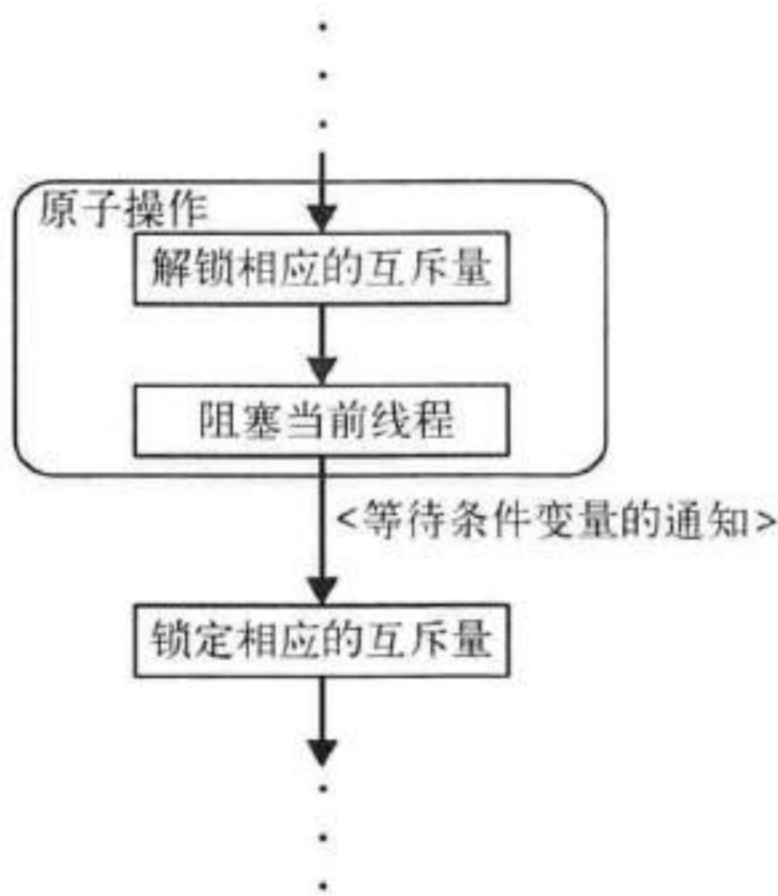


图6-22 条件变量的等待通知操作的内部流程

等待通知操作的精妙之处不止于此。当等待通知操作因收到条件变量发送的通知而唤醒当前线程之后, 会首先重新锁定与该条件变量绑定在一起的那个互斥量。如果该互斥量已经被其他线程抢先锁定, 那么当前线程会再次进入到睡眠状态。为什么要重新锁定那个互斥量? 其主要原因是条件变量的设计者认为, 线程在执行等待通知操作的地方被唤醒之后一般会立即访问共享数据。事实也确实如此。这倒不是由于当收到条件变量的通知时会立即对共享数据进行操作, 而是因为当前线程在继续运行之后, 应该马上再次检查共享数据的状态以判断其是否满足条件。为什

么要这么做？请看下面的这个反例。我们在生产者线程向队列添加数据块的流程中加入对条件变量的运用，并且当该线程在执行等待通知操作处被唤醒之后，不再次检查队列的状态而直接向队列添加数据块，其流程如图6-23所示。

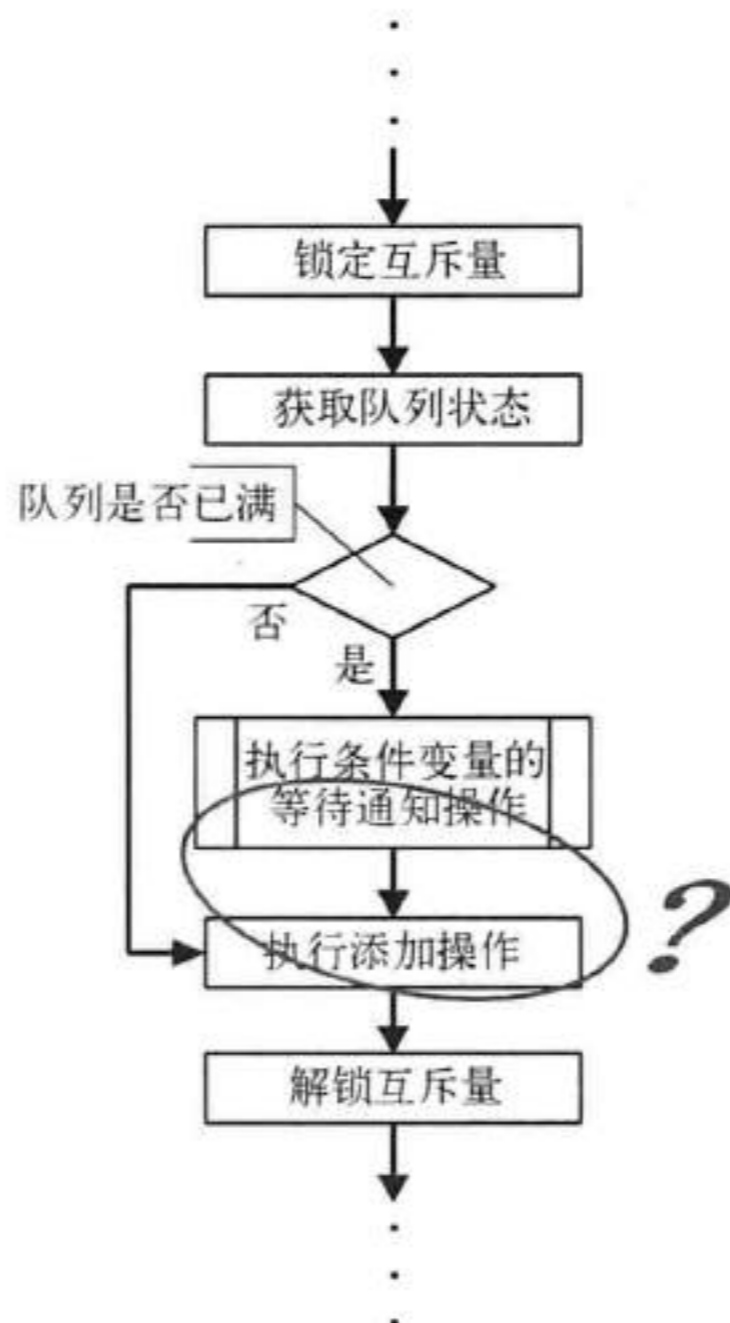


图6-23 不正确地使用条件变量的生产者线程添加数据块的流程

在线圈之下的部分就是此流程的问题所在。在当前线程被唤醒之后直接执行了添加操作。如果生产者线程只有一个的话，可能不会出现什么问题。但是在真实的场景中往往没有这么简单。在队列已满的情况下，可能会有多个生产者线程会相继因执行等待通知操作而进入到睡眠状态。由于不知道在队列有空余空间的时候会有多少个生产者线程接收到该条件变量的通知，所以我们总是应该考虑多个生产者线程同时接收到通知的情况。当这种情况发生时，如果一个生产者线程被唤醒并直接执行添加操作，那么就有可能使该操作失败。因为可能会有其他生产者线程抢先向队列添加了数据块，致使该队列又处于已满的状态。如此一来，我们可能需要再次尝试添加数据块。如果再次尝试，为了保证成功率，我们必然还要先检查队列的状态。这样做的效率是很低的。因为线程始终会执行成功率低下的添加操作。如果不再次尝试，那么就等于甘愿承受数据块添加操作的失败。显然，无论是否再次尝试，都不是正确且高效的方案。其本质在于，我们错过了再次检查队列状态的最佳时机。如果在执行添加操作之前再次检查队列的状态，并保证仅在条件满足时才执行添加操作，那么就可以保证添加操作的成功。请看图6-24所示的流程图。

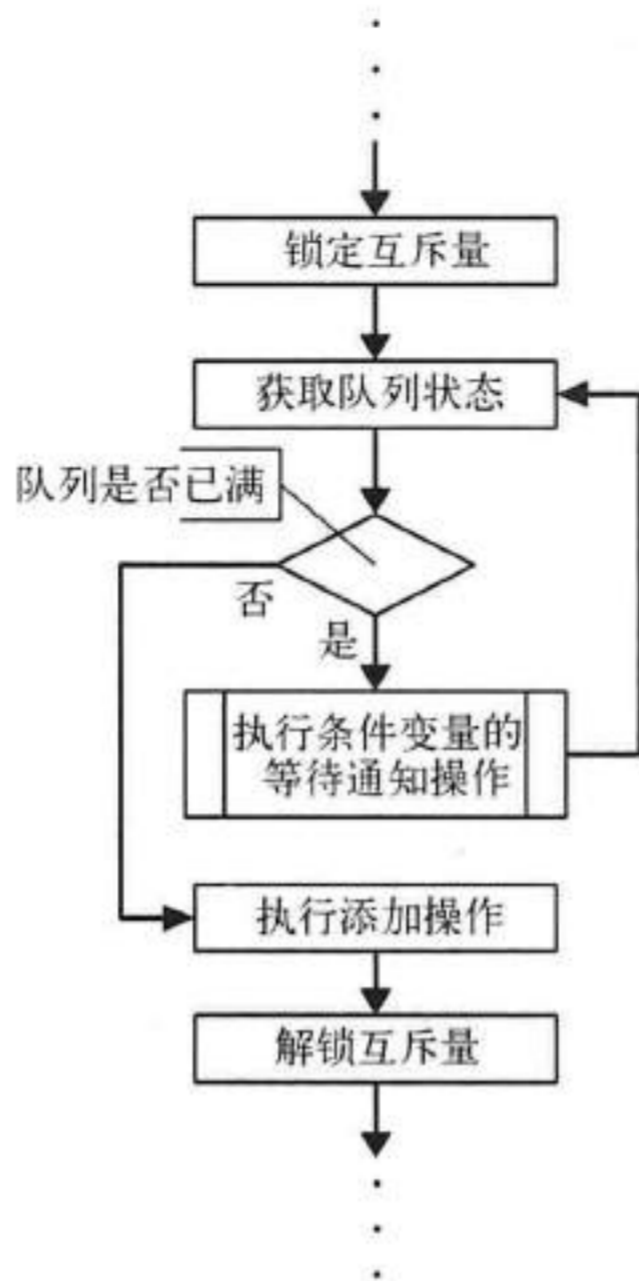


图6-24 使用条件变量的生产者线程添加数据块的流程1

我们使用了一个循环体让线程在被唤醒时总是重新检查队列的状态。这样的话，即使被其他线程抢先了一步，当前线程仍然可以再次利用等待通知操作重新等待操作时机。如果再次确认了队列的不满状态，那么就可以放心地执行添加操作了。所以此流程是正确和高效的。注意，在有些多CPU的计算机系统中，即使没有接收到条件变量的通知，线程也有可能被唤醒。所以，我们总是应该依据此流程运用等待通知操作。

下面我们来说说条件变量的单发通知操作和广播通知操作。这两个操作的作用都是向因执行相同条件变量的等待通知操作而被阻塞的线程（以下简称等待线程）发送通知。但不同的是，前者只保证至少会唤醒一个等待线程，而后者则必然会唤醒所有的等待线程。这就决定了这两项操作的适用场景。如果我们明知道等待线程都在等待共享数据的同一个状态，并且在某个等待线程被唤醒并执行相应操作之后，共享数据的状态就不再满足等待线程的条件了，那么再使用广播通知操作来通知等待线程肯定就是低效的。前文所述的数据块队列的例子就是这样的典型案例。所有的生产者线程都在等待队列非满的状态。如果我们使用广播的方式来发送通知，虽然所有已在等待的生产者线程都会接收到该通知，但是一旦某个生产者线程被唤醒并抢先向队列添加了一个数据块，那么该队列的状态就又会回到已满的状态。据此，我们只要通知一个生产者线程以示意可以向队列添加新的数据块就可以了。通知更多的生产者线程只会让它们白白浪费CPU的资源，从而使程序的整体性能降低。如果一个等待线程接收到通知但未能及时就此作出应有的响应，那么这个通知就属于过剩的通知。这种情况下，因接收到通知而唤醒等待线程的这个动作也被称为伪唤醒。请记住，它们会对程序的运行带来负面影响。

为了及时地向生产者线程告知等待队列的非满状态,我们需要对原有的消费者线程获取数据块的流程进行改进。改进后的流程如图6-25所示。

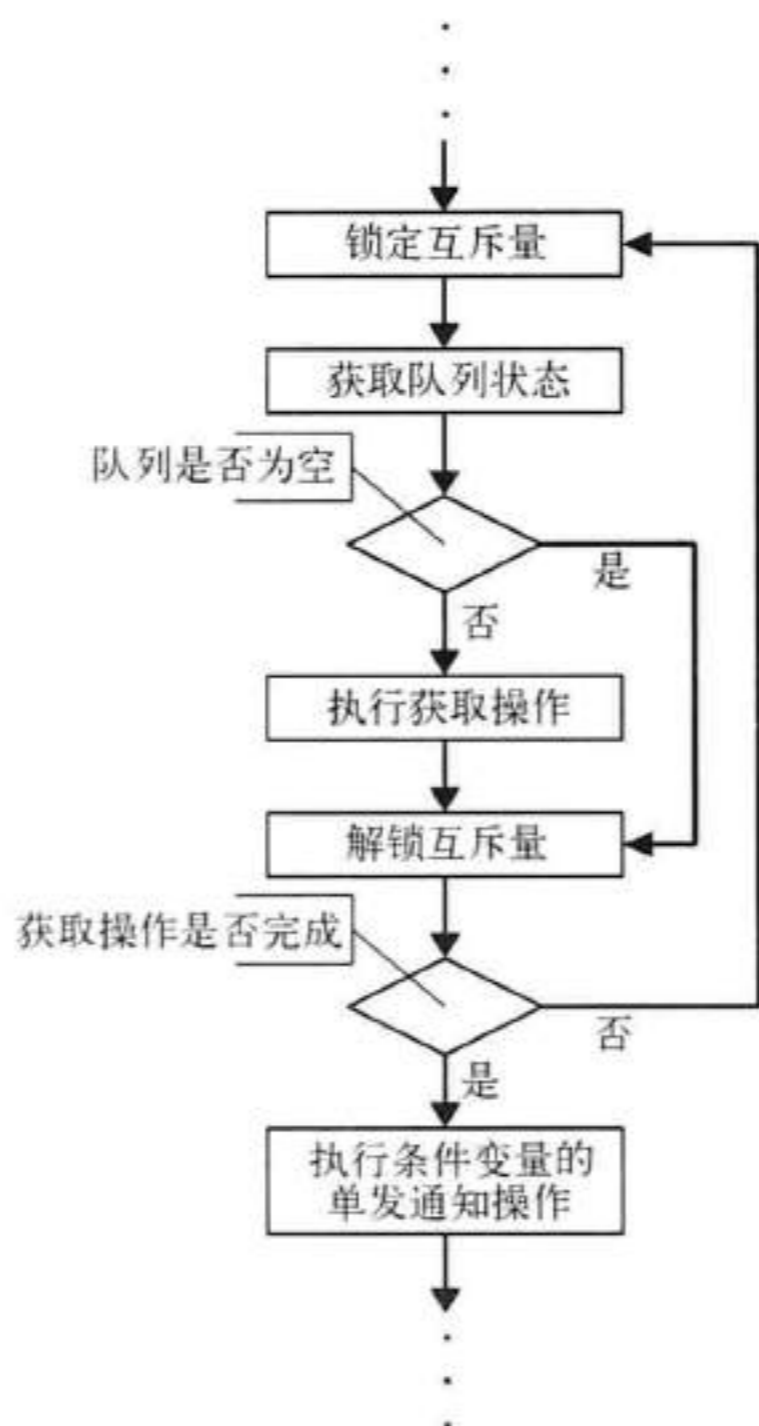


图6-25 使用条件变量的消费者线程获取数据块的流程1

可以看到,我们只是在确认获取操作的执行完成之后添加了一个执行条件变量的单发通知操作的步骤。这意味着,每当队列中的一个数据块被消费之后,相应条件变量的单发通知操作都会被执行一次。注意,条件变量的通知具有即时性。通知只是负责向等待线程发送一个信号以告知共享数据的状态发生了某种变化,而不会存储相关信息。在通知被发送的时候,如果没有任何线程正在等待此条件变量的通知,那么该通知就会被无视。并且,它也绝不会被传递到在它被发送之后才开始等待它的线程那里。换句话说,通知稍纵即逝,并且不会留下任何痕迹。因此,我们丝毫不用担心这样频繁的通知发送操作会给生产者线程的正常流程带来什么不利影响。另外,细心的读者可能会发现,消费者线程是在对互斥量进行解锁之后才执行条件变量的单发通知操作的。实际上,这两个操作之间的顺序是无关紧要的。我们完全可以颠倒它们的执行顺序。并且,在互斥量的保护下执行单发通知操作通常更加安全。

现在我们来解释一下前文埋下的那个问题。等待通知操作在被执行的时候会先解锁互斥量再阻塞当前线程。但是,由于这两个步骤共同组成了一个原子操作,所以在当前线程被阻塞之前其他线程无法锁定该互斥量。我们依然以在生产者线程中执行的等待通知操作为例。在生产者线程执行等待通知操作的过程中,消费者线程无法从队列那里获取数据块,并且也无法执行之后的单

发通知操作。如果双方的操作被同时发起，那么消费者线程执行这些操作的时间势必会被推后一点。这点时间是极短的，几乎可以忽略不计。然而，为了这极短的时间而错过了本应接收到的通知则是非常不值得的。从另一个角度看，即使在生产者线程正在执行等待通知操作的过程中同一个条件变量发送的通知会被传递给它，它也无法做出任何响应。其原因是这时的生产者线程还没有为接收通知做好准备。它此时还未进入到睡眠状态并开始等待通知。更重要的是，在通知达到之后生产者线程依然会被阻塞。因此，这时达到的通知不会起到任何作用，并且还可能会引发一些冲突。可见，把等待通知操作中的两个步骤合为一个原子操作是非常有必要的。

到目前为止，我们仅讲到了生产者线程应该怎样利用条件变量来等待队列状态的更新并及时做出响应，以及消费者线程应该在什么时候让该条件变量向正在等待通知的生产者线程发送通知。但是，不要忘了在这个示例中还存在另外一种对称的情况，那就是消费者线程需要利用条件变量等待空队列的非空状态的出现并试图从队列那里获取新的数据块，同时发送者线程也应该在每次数据块添加操作完成之后让该条件变量发送通知给正在等待的消费者线程。

注意，为了向特定的等待线程告知队列已处于非满状态或非空状态，我们需要分别创建两个条件变量。为了加以区分，我们把被用来关注队列的非满状态的条件变量称为条件变量 I，而把被用来关注队列的非空状态的条件变量称为条件变量 II。由于在生产者线程和消费者线程中的对数据块队列的操作都由同一个互斥量保护的，所以条件变量 I 和条件变量 II 也必须都与这个互斥量进行绑定，这样才能达到预期的效果。

为了在添加操作完成后向消费者线程告知队列已处于非空状态，我们还需要对前述的那个生产者线程执行的流程稍作修改，新的流程图如图6-26所示。

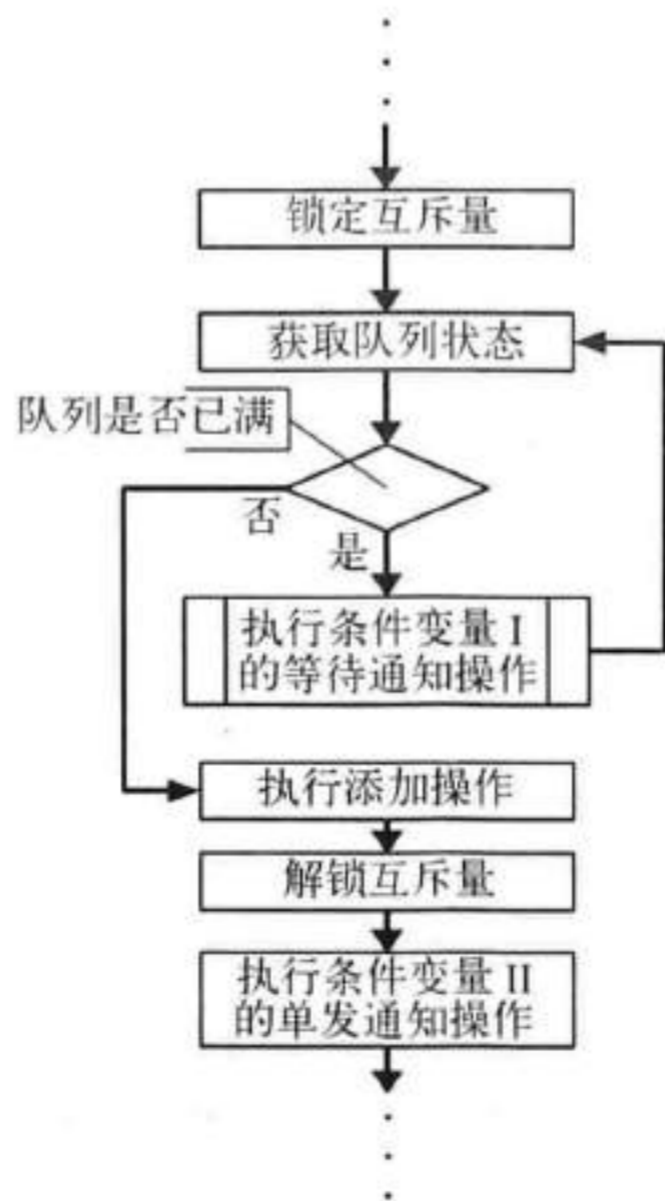


图6-26 使用条件变量的生产者线程添加数据块的流程2

对前面的消费者线程的流程的改造也是必须的。这与先前对生产者线程的流程的改进方式是非常类似的。我们需要让消费者线程在发现队列未处于非空状态时等待条件变量II的通知。并且，让消费者线程在被唤醒时重新检查队列的状态，并在必要时再次等待条件变量II的通知总是应该的。这样既可以避免在新的数据块被其他消费者线程抢先取走的情况下数据块获取操作的失败，又可以使之尽早地从队列中获取到数据块，如图6-27所示。

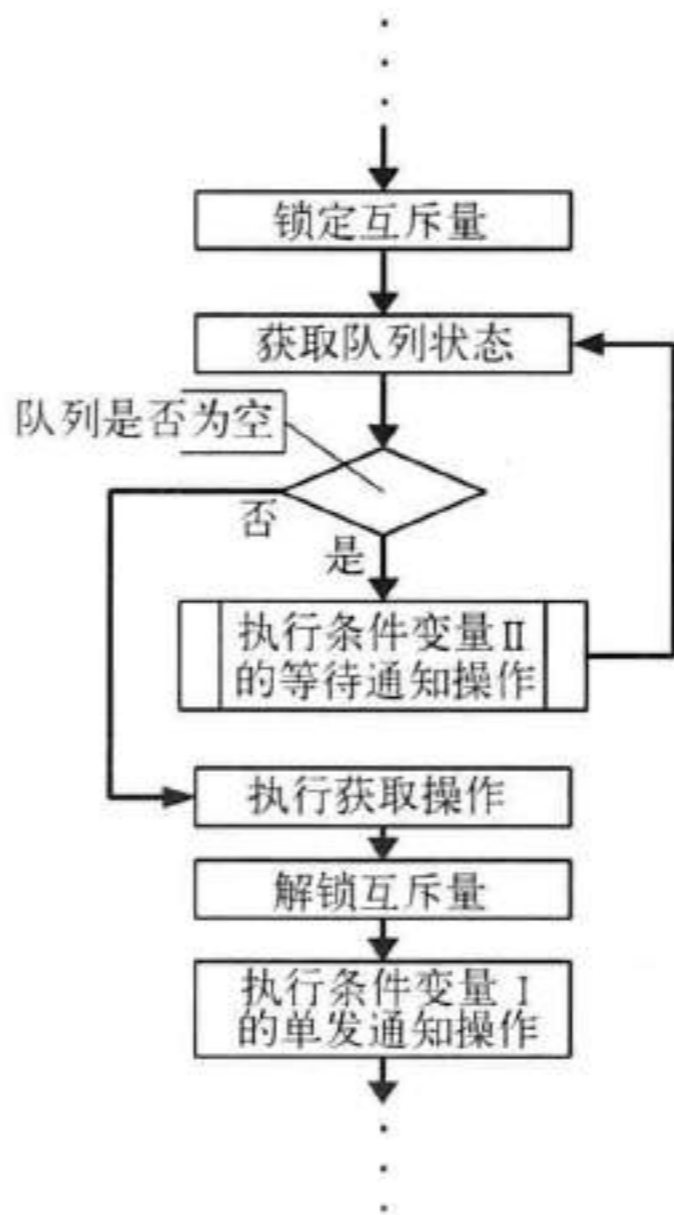


图6-27 使用条件变量的消费者线程添加数据块的流程2

可以看到，经过几番改进之后，消费者线程添加数据块的流程依然与生产者线程添加数据块的流程非常相似。现在，这两类线程都利用条件变量实现了更高级的同步效果。这使得它们的运行效率和协作能力都得到了进一步的提高。

在我们真正地理解了条件变量的单发通知操作的特点和使用方法之后，广播通知操作就不难理解了。实际上，从易用性的角度来看，广播通知操作更胜一筹。只要等待线程可以处理好过剩的通知，那么广播通知总是能够达到目的的。而且在某些应用场景下，广播通知操作是更适合的。例如，如果多个等待同一个条件变量的通知的线程所期望的共享数据的状态存在多样性，那么以广播的方式发送通知就是最好的选择。准备发送通知的线程（以下简称发送线程）无法得知有哪些线程正在等待共享数据的当前状态，更不会知道在执行单发通知操作之后哪一个线程会接收到该通知。因此，发送线程只得执行广播通知操作以向所有的等待线程告知共享数据的状态已发生变化，且不会（也不可能）去关心哪些等待线程会对这一状态变化感兴趣。等待线程在被唤醒后会去重新检查共享数据的状态，并自行决定是对此做出响应还是继续等待下一个通知。

到这里，我们已经详细说明了多线程编程当中的两个非常重要的同步工具。互斥量使作为被

同步对象的若干操作可以被有条不紊地执行，并且不会产生任何竞态条件。而条件变量则会使这些操作执行得更有效率。Go语言作为原生支持并发编程的语言提供了与它们有着类似功能的API。有了本小节所讲知识的铺垫，读者若要使用那些API将会是非常容易的事情。

4. 线程安全性

在本小节的最后，我们来简要地了解一下线程安全性这个概念。如果有一个代码块，它可以被多个线程并发地执行，且总能够产生预期的结果，那么该代码块就是线程安全的（thread-safe）。例如，如果代码块中包含了对共享数据的更新操作，那么这个代码块通常就是非线程安全的。但是，如果该代码块中的类似操作都处于临界区之中，那么这个代码块就是线程安全的。

经常被置于线程安全问题之中的代码块就是函数。这不仅仅是因为函数是最常用的独立代码块，更是因为函数的线程安全性有着更多的含义。让函数具有线程安全性的最有效方式就是使其可重入（reentrant）。如果某个进程中的所有线程都可以并发的对一个函数进行调用，并且无论它们调用该函数的实际执行情况怎样，该函数都可以产生预期的结果，那么就可以说这个函数是可重入的。更通俗地讲，如果多个线程并发的调用该函数与它们以任意顺序依次地调用它所产生的效果总是相同的，那么该函数就是一个可重入函数。

如果一个函数把共享数据的值作为其返回的结果或者包含于其返回的结果中，那么该函数就肯定不是一个可重入函数。因为，如果其他线程在该结果返回的过程中更新了相关的共享数据，那么函数的调用方得到的结果就很可能不是该函数真正想返回的那个结果。这种情况是无法通过使用互斥量来解决的。因此，为了使函数可重入，我们必须也只能杜绝在该函数的返回结果中掺杂任何共享数据。当然，如果这些共享数据都是完全不可被更新的话，就不会存在这样的隐患。一条更加通用的规则是，任何内含了对共享数据进行操作的代码的函数都可以被视为不可重入的函数。

使函数成为可重入函数是实现其线程安全性的最直接的一种方式，并且也是最简单和最高效的方式。我们应该尽量编写可重入的函数。但是，如果我们无法办到这一点，或者该函数中需要调用某些不可重入的函数，那么就需要采取另一种办法——使用互斥量把相关的代码保护起来。

当然，为了实现线程安全的函数，把其中的所有代码都置于临界区之中是可行的。但是，这往往也是最低效的一种方法。我们应该仔细地函数体中查找出操作共享数据的代码并用互斥量把它们保护起来。如果可能的话，我们还应该把这些代码从函数体中分离出来。这样有利于在之后施加保护措施。分离的方式可以是把它们聚集在一起，也可以是把它们独立成一个或多个函数。或者，更进一步地，把它们抽象并组装成一个数据结构并使这个数据结构具有线程安全性。当然，是否分离这些代码以及以怎样的方式分离它们取决于很多因素。最简单的方式就是把每条涉及操作共享数据的语句都用互斥量保护起来。这通常也是我们最初采用的方式。

无论怎样，互斥量的使用总会耗费一定的系统资源和时间。正如前文所说，使用互斥量的过程中总会存在着某些博弈和权衡。如果在一个代码块中仅包含对共享数据的访问操作而不包含对它们的更新操作，那么在这个代码块之内就可以不使用互斥量。但一个必要的前提是，必须在真正使用共享数据之前就把它们完全复制到当前线程的线程栈之中。也就是说，线程需要自己维护一份其需要使用到的共享数据的副本。对于函数来说，这样的副本是作为其局部变量存在的。某

一个线程对某个函数的第一次调用会致使该函数中的局部变量陆续被创建在该线程的线程栈中。在不同线程的线程栈中，因调用相同的函数而被创建的同名局部变量之间是完全独立的，并且不会相互干扰。一般情况下，这样的函数（即只包含对共享数据的访问操作的函数）是可重入的，也是线程安全的。因为它们的执行效果与是否并发地执行它们无关。不过，有两个地方还需要我们注意。

第一，如果当前进程中还并发地执行了更新相关共享数据的代码的话，情况就会有些不同。这些函数的执行效果可能会受到这些更新操作的影响，无论这些函数是否被并发的执行。例如，在线程A正在更新一个较复杂的共享数据（无法仅用一条语句完成对它的更新）但还未完成的时候，线程B访问了该共享数据并把它值赋给了一个局部变量。这时，线程B中的那个局部变量的值就只代表了该共享数据在被更新过程中的一个中间状态。这是不正确的。虽然这种情况的发生需要一定的条件，但是我们还是应该把它纳入到考虑范围之内。在这种情况下，是否可以省略掉对互斥量的使用，还需要依靠我们对程序的运行时状况的预判和评估。

第二，如果某个局部变量的值由共享数据本身提供，并在该共享数据中还包含了对其他共享数据的引用，那么就不能说该局部变量所属的函数是可重入的。因为被这个函数间接使用到的共享数据的改变也有可能影响到该函数的实际执行效果。这种情况极易成为一个漏洞，造成一些看起来匪夷所思的错误，并且非常难以被察觉和定位。这也是我们在前面强调必须对共享数据进行完全复制的原因。另外，若局部变量的值是指向某个共享数据的指针而不是其本身，也会使所谓的共享数据副本形同虚设。因此，如果我们确实需要在线程内创建一个共享数据的副本，就要对它进行深层的复制。也就是说，要把该共享数据中引用到的所有其他共享数据全部复制一份，并进行重新组装。有时候，这可能是非常费时、费力，以及耗费系统资源的。是否真的需要在涉及结构复杂的共享数据的时候依然使用这种方式使函数实现线程安全性还有待商榷。这又是一个需要权衡的地方。

再次强调，在你还未深入考虑过或者还没有最终确定应该使用哪一种方式保证线程安全性的时候，请使用互斥量保护好那些涉及共享数据操作（包括访问和更新）的代码。

我们在进行多线程编程的时候总会遇到同步的问题。只要处理好这些问题才能够让我们的程序正确并高效地运行。本小节为读者抛出了一些常见的问题以及可以利用的工具。在Go语言中，一些容易导致错误的问题被它的运行时系统有效地处理了，并在操作系统的多线程支持之上为我们提供了更加先进的并发编程模型。这为我们提供了极大的便利，使我们在编写Go语言程序的时候可以更少的在这些非业务性的问题上耗费精力。但是，Go语言依然通过标准库为我们保留了那两个最重要且使用最广泛的同步工具——互斥量和条件变量，以备我们的不时之需。

6.4 多线程与多进程

作为一种相对较新的并发编程方式，多线程编程有着明显的优势。这些优势成为不断诱惑着我们采用这种方式进行并发编程的重要因素之一。不过，对于多线程编程来说，它的优势之处很可能也隐含着一些劣势。这种微妙的差别使得我们在二者选其一的时候总是需要仔细的考量一番。

我们已经知道，在多个线程之间交换数据是非常简单和自然的事情。而在多个进程之间，我们只能通过一些额外的手段（比如管道、消息队列、信号灯和共享内存区）在它们之间传递数据。显然，使用这些额外的手段会增加开发的成本。不过，线程间交换数据虽然简单但却由于可能发生竞态条件而不得不使用一些同步工具（比如互斥量和条件变量）加以保护。这些与业务逻辑无关的代码会增加程序的复杂度。并且，使用这些同步工具本身也是需要注重方式方法的。如果使用不当，可能造成程序性能的大幅下降甚至发生死锁。这其中充满着各种博弈。因此，在进行多线程编程的时候，我们要仔细地划分临界区，并在对实际情况进行评估之后，再在原有代码上添加互斥量和条件变量的相关操作，以求同步的正确和高效。

并发编程方式的选择对程序的开发维护成本和运行性能都有着深远的影响。总体来说，多线程作为更加现代的并发编程方式在系统资源的利用和程序性能的提高方面都更具有优势。但是，在某些情况下（比如对信号的处理、同时运行多套不同的程序以及包含多个需要超大内存支持的任务等），传统的多进程编程方式可能会更加适合。

选择并发编程方式的一个不可被忽视的因素就是我们所使用的编程语言。一些编程语言更适合编写多进程程序，而另一些编程语言则对多线程有着强大的支持。Go语言是两者兼而有之但更倾向于后者的。它对多线程技术的使用比现有的其他编程语言更加先进和充分。

虽然每种编程语言都会倾向于某种并发编程方式，但是它们的最终目标都是在开发维护成本和运行性能这两个方面上进行权衡，以满足应用开发者的要求。需要注意的是，程序的开发维护成本的降低和运行性能的提高在很多时候是相互对立的。它们之间常常存在着此消彼长的态势。所以，让程序在可容忍的开发维护成本下被高效的并发运行并不是一件容易的事。我们在编写并发程序的时候总是应该仔细考量并小心对待。

Go语言的先进特性可以让我们非常快速地开发应用程序。同时，它的简约编程风格和工程哲学也让应用程序的维护成本总能保持在较低的水平上。更重要的是，Go语言程序的运行性能已经与传统的系统级别编程语言相差无几了。Go语言天生就是被用来编写并发程序的。它完全可以胜任适合采用多线程编程方式的所有应用场景。并且，一些不适合在多线程程序中执行的任务（比如信号处理）它也可以很容易地完成。总之，使用Go语言编写并发程序能够帮助我们更快速和轻松地达到目标。

6.5 多核时代的并发编程

几十年来，计算机硬件工业一直与摩尔定律相吻合：每18个月台式计算机的运行速度就会翻一倍。除了对算法和软件架构的改进，软件开发者们还依赖摩尔定律让他们的软件跑得更快。然而，由于单CPU的时钟频率越来越难提高，制造商们转而把精力放在增加CPU的核心数量上。这使得软件开发者们有机会让他们的程序真正并行地运行起来。图6-28展示了在单核CPU和多核CPU上运行并发程序的区别。

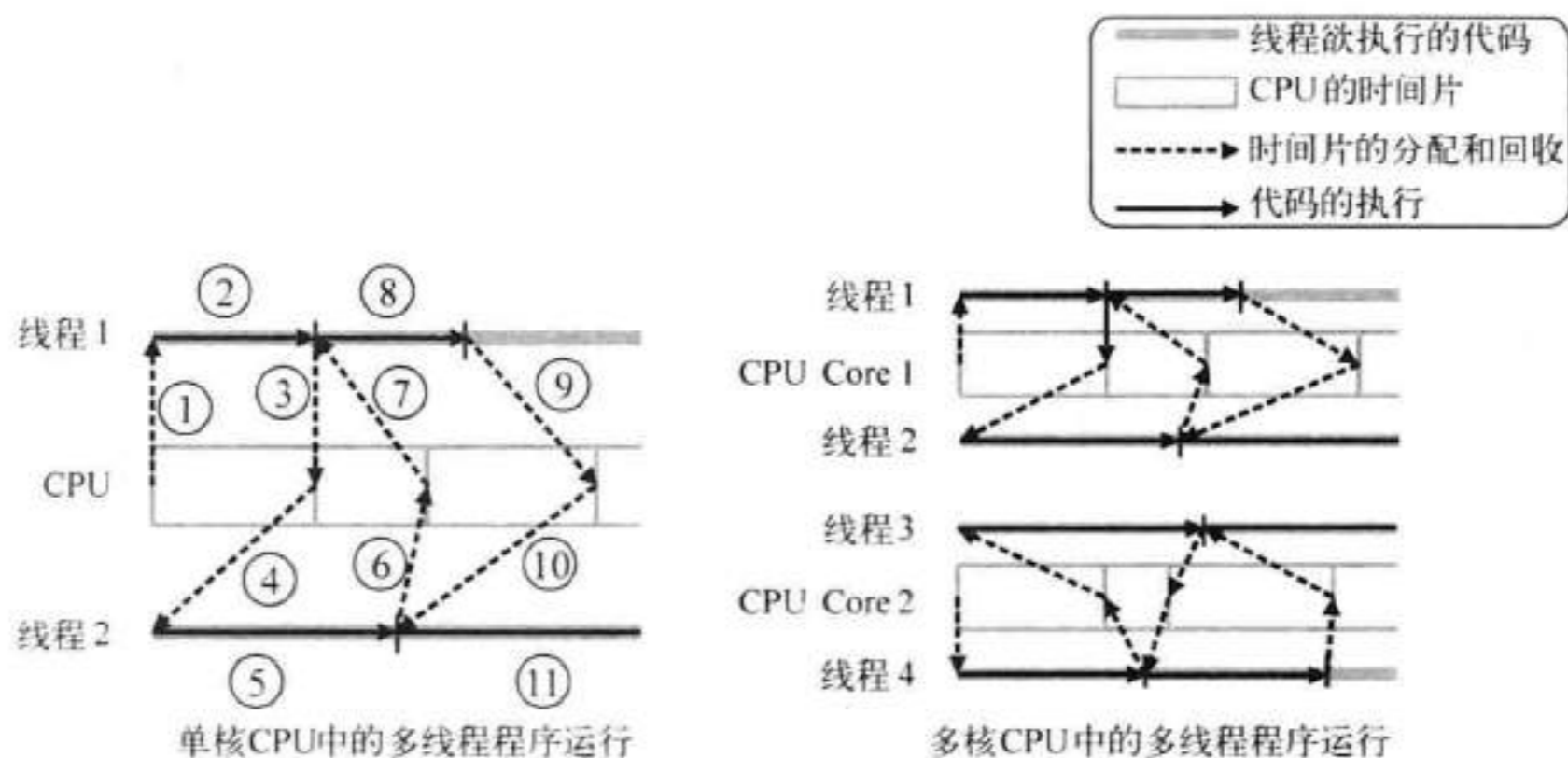


图6-28 在单核CPU和多核CPU上运行并发程序的区别

如图6-28所示，当在单核CPU上运行多线程程序的时候，每一时刻CPU只可能运行一个线程。但由于操作系统内核会根据调度策略切换CPU运行的线程，所以一般情况下我们会感觉多个线程在同时运行。此处的同时运行实际上只能算作是并发运行。这就是该图左半部分向我们展示的内容。在该图的右边，一个双核心的CPU之上运行着一个拥有4个线程的进程。其中，线程1和线程2在CPU核心1上运行，线程3和线程4在CPU核心2上运行。在同一时刻，在线程1和线程2中只会有一个被运行，而在线程3和线程4中也只会有一个被运行。我们也可以说，这4个线程是被并发运行的。但是，在同一时刻，CPU核心1和CPU核心2上会分别运行着某一个线程。此时，我们可以说这两个线程正在被并行的运行。我们由此可以看出并发运行和并行运行的区别。并发运行是指多个任务被同时发起（或者说开始）运行，但是在同一时刻这些任务不一定都处于运行状态。这取决于CPU核心或者CPU的数量。相比之下，并行运行指的是在同一时刻可以有多个任务真正地同时运行。并行运行的必要条件是多CPU核心或/和多CPU的计算环境。由此可见，并行运行是并发运行的一个更高级的层次。或者说，并行运行的一个必要条件就是并发运行。

通过上面的图示和说明，我们可以得出一个结论：让程序真正在多核CPU上并行的运行起来的前提是采用某种并发编程方式来编写程序。在这个前提下，操作系统内核能够通过调度使多个进程或线程并行的运行于不同的CPU核心之上。这可以更加充分地利用计算机硬件以进一步提高程序的运行性能。因此，当代软件开发者的一个主要的开发或维护任务就是让程序被更加高效地并发运行。这里的高效是指，在保证程序的正确性和可伸缩性的前提下提升程序的响应时间和吞吐量。

在这里，我们再一次提到了这两个重要的程序性能测量指标——响应时间和吞吐量。响应时间是指从计算请求被递送到计算结果部分可用之间的实际耗时。对于一个长期运行的程序（比如各种服务端程序）来说，响应时间是非常重要的性能指标。吞吐量是指在一个时间单元（比如秒或分钟）之内程序完成并输出结果的计算任务的数量。采用多进程或多线程编程是可以使此指标提升的主要方法。不过，这也取决于实际运行程序的计算机硬件的性能（比如CPU的时钟频率和核心数量）。

然而,与并发编程关系更加紧密的是程序的正确性和可伸缩性。正确性是指程序的行为应该与程序设计者的意图严格一致。即使在它被并发地运行的时候也应该如此。因为进程或线程间的上下文切换可能发生在任何时刻,所以并发程序应该保证那些被并发执行的操作的有效性和完整性。这一般可以通过我们之前讲到的各种同步方法和原子操作来实现。并发程序的可伸缩性主要体现在增加CPU核心数量的情况下,其运行速度不会受到负面的影响。乍一看,这应该是理所当然的。因为CPU核心数量的增加意味着计算机硬件性能的增强。这种增强理应使程序运行得更快。但是,事实上,并发程序运行速度的提升曲线会随着CPU核心数量的增加而趋于平缓。这种趋于平缓的态势主要取决于并发程序中的原子操作和同步方法的数量和执行耗时。对同一个程序来说,其中的原子操作和同步方法的数量越多、执行耗时越长,其运行速度的提升曲线趋于平缓的态势就越明显。为什么会这样?其根本原因就在于这些操作和方法的复杂性。这里的复杂性倒不是说我们使用它们会有多复杂,而是说它们的实现会比较复杂。就互斥量来说,为了消除竞态条件,它让多个线程不能同时执行临界区中的代码。换句话说,各个线程只能串行的执行这些被同步的代码。这可以使并发程序得以正确地运行。可是,这种在并发运行过程中的串行化执行是需要付出代价的。这需要操作系统内核和计算机硬件(主要指CPU)的共同努力才能够完成。这涉及一些必要的底层协调工作,尤其是当CPU核心不止一个的时候。这种协调工作本身就会对CPU运行程序的效率产生不可小视的负面影响。类似的协调工作越多,这种负面影响就会越大。在绝大多数情况下,CPU核心数量的增加也意味着在执行原子操作和同步方法的时候需要更多的协调工作。显然,这对并发程序的运行性能是有害的。这也是我们之前建议应该尽量编写可重入的函数的原因之一。其目的就是消除函数中的同步方法。不过,让并发程序的运行速度丝毫不受到因CPU核心数量的增加而产生的负面影响是不可能的。我们只能尽力而为之。

说到这里,读者可能会意识到,程序的正确性和可伸缩性之间有时候会存在一定的矛盾。事实也确实是这样。当并发程序中包含了对共享数据的操作的时候,保证这些操作的并发安全总是必须的。这也是保证并发程序的正确性的重要方法之一。其中,最直截了当的保证手段就是使用编程语言或者某些函数库提供的原子操作和同步方法。当然,这些操作和方法最终还是由操作系统和计算机硬件来支持的。使用这些保证并发安全的手段就意味着给程序的可伸缩性施加了更强的约束。这种约束与程序的运行性能是成反比的。由此可见,如果想在多CPU核心的计算环境中进一步提升并发程序的运行性能,以更加高效的方式来实现代码块的并发安全性应该是我们努力的方向。显然,减少对原子操作和各种同步方法的使用是最简单和有效的。但是,我们通常很难做到这一点。因为保证程序的正确性永远是第一要务。因此,以更加得当的方式使用这些操作和方法就至关重要了。下面,就此问题给出几条建议。

- 控制临界区的纯度。临界区中仅应包含操作共享数据的代码。也就是说,尽量不要把无关代码囊括其中,尤其是各种相对耗时的I/O操作(注意,调用打印函数也会引发I/O操作)。夹杂无关代码只会让临界区的界限模糊并进一步影响程序的运行性能。
- 控制临界区的粒度。由于粒度过细的临界区会增加底层协调工作的发生次数,所以有时候我们会粗化临界区。如果存在相邻的多个临界区,并且它们内部都是操作同一个共享数据的代码,那么就应该合并它们。若在它们之间夹杂着一些无关代码,则应该试着调

整这些代码的位置，即把它们放在合并后的临界区的前面或后面。如果实在无法调整，我们就需要在临界区的纯度和粒度之间进行权衡。简单地说，如果其间没有长耗时的无关代码，那么就把它合并在一起，否则就只能放弃合并。总之，我们应该优先考虑减少粒度过细的临界区。

- 减少临界区中代码的执行耗时。提高临界区的纯度可以减少临界区中代码的执行耗时。但是，如果操作共享数据的代码本身执行起来就很耗时，那又该怎么办呢？这分为两种情况。第一种情况，临界区中包含了对几个共享数据的操作代码。在这种情况下，无论这些操作不同共享数据的代码之间是否存在强关联，我们都可以考虑把它们拆分到不同的临界区中，并使用不同的同步方法加以保护。这里不存在粒度过细的问题，因为它们针对的是不同的共享数据。不过这需要注意另外一些问题（详见6.3节中讲到的因互斥量的使用不当而造成的死锁问题）。第二种情况，临界区中仅包含了操作同一个共享数据的代码。这时我们往往不能通过调整临界区的方式达到减少耗时的目的。因为粒度过细的临界区反而会增加额外的时间消耗。所以，正确的做法应该是检查其中的业务逻辑和算法并加以改进以减少耗时。
- 避免长时间持有互斥量。线程长时间持有某个互斥量所带来的危害是明显的。同样明显的是，在减少临界区中代码的执行耗时的同时可以减少线程持有相应互斥量的时间。不过，有时候使用另一个方法同样可以起到很好的作用。这个方法就是使用条件变量。条件变量会适时地对互斥量进行解锁和锁定操作，所以线程持有互斥量的时间会大大减少。在临界区中的代码会等待共享数据的某个状态的情况下，使用条件变量往往会达到非常好的效果。
- 优先使用原子操作而不是互斥量。这样做的理由是，使用互斥量一般会比使用原子操作所造成的程序性能损耗大很多。并且，随着CPU核心数量的增加，这一差距会被进一步拉大。当我们的共享数据的结构非常简单（比如基础数据类型的数值）的时候，建议使用原子操作来代替附加了互斥量操作的代码。原子操作会直接利用硬件级别的原语来保证操作的成功和数据的并发安全。不过，遗憾的是，对于结构稍复杂一些的共享数据，原子操作就无能为力了。因此，这条建议的适用范围是比较有限的。

上述这些建议的最终目标都是在不失去程序的正确性的前提下最大限度地提高程序的可伸缩性。为什么要提高程序的可伸缩性？其原因是，对于程序的运行性能的提升来说，为运行它的计算机添加更多的CPU核心（或者更换一台拥有更多CPU核心的计算机）往往会更加直接、简单，甚至廉价。退一步说，它起码是提升程序运行性能的一个有力手段。而使用多进程以及多线程编程则是另一个有力的手段。不过后者需要编程人员掌握一定的技巧。在没有这样的人员支持或者条件不充分的情况下，增强计算机硬件的性能通常会首选。这也是云计算和弹性计算在当下如此火热的主要原因之一。话说回来，我们应该尽量让施加在并发程序上的这种“软提升”和“硬提升”在效果上产生叠加而不是抵消。在多核时代，怎样更好地利用并行计算提升程序的运行性能已经是一个应用程序开发者必须要考虑的问题了。值得庆幸的是，Go语言有能力祝你一臂之力。

6.6 Go 语言的并发编程

在本节，我们将会对Go语言下的并发编程及其模型进行介绍。这些内容会让读者在阅读后面几章的时候更加顺畅。

之前我们说过，Go语言在操作系统提供的内核线程之上搭建了一个特有的两级线程模型。由此，也就引出了Goroutine这个特有名词。Go语言的开发者们之所以专门创建了这样一个名词，是因为他们认为已经存在的线程、协程、进程等术语都传达了错误的含义。为了与它们有所区别，Goroutine这个词才得以诞生。

那么，Goroutine所代表的正确的含义是什么？Go语言打出的标语是这样的：

不要用共享内存的方式来通信。作为替代，应该以通信作为手段来共享内存。

更确切地讲，把数据放在共享内存区中供多个线程中的程序访问的这种方式虽然在基本思想上非常简单，但是却使并发访问控制变得异常复杂。只有做好了各种约束和限制，才有可能会使这种看似简单的方法得以正确地实施。但是，正确性往往不是我们唯一想要的。我们常常还需要足够的可伸缩性。然而，一些同步方法的使用让这种需求的达成变得困难了许多。就像我们在上一节所讲的那样。

Go语言不推荐以共享内存区的方式传递数据。作为替代，我们应该优先使用Channel。Channel主要被用来在多个Goroutine之间传递数据，并且还会保证其过程的同步。不过，作为另一种可选方法，Go语言依然提供了一些传统的并发访问控制方法（互斥量、条件变量，等等）。

在后面的几章中，我们会分别对Goroutine、Channel和Go语言提供的传统并发访问控制方法进行介绍。但是，在这之前，我们先要对Go语言的并发编程模型进行必要的讲解。下面，我们一起来探究Go语言构建的这个两级线程模型的内部机理。不过，如果你还不想关注较底层的模型和实现，那么可以跳过这一节直接阅读下一章。等到你想要或不得不了解它们的时候再翻回来查阅也不迟。

6.6.1 线程实现模型

以我们目前对两级线程模型的了解，Goroutine可以被看作是Go语言特有的“应用程序线程”。但是，实际上，Goroutine背后的支撑体系可远没有这么简单。

说起Go语言的线程实现模型，有3个必知的核心元素。它们支撑起了这个线程实现模型的主框架，其简要说明如下。

- M: Machine的缩写。一个M代表了一个内核线程。
- P: Processor的缩写。一个P代表了M所需的上下文环境。
- G: Goroutine的缩写。一个G代表了对一段需要被并发执行的Go语言代码的封装。

可以看到，这些核心元素的表示相当精炼（只需一个字母），含义也非常明确。请读者记住这3个字母，我们在后面会以它们代表对应的元素。

简单来说，一个G的执行需要M和P的支持。一个M在与一个P关联之后就形成了一个有效的G运行环境（内核线程+上下文环境）。每个P都会包含一个可运行的G的队列（runq）。该队列中

的G会被依次传递给与本地P关联的M并获得运行时机。在这里，我们把运行当前程序的那个M称为当前M，而把与当前M关联的那个P称为本地P。后面我们会以此为参考进行描述。

从宏观上看，M、P和G之间的联系如图6-29所示。但是它们的实际关系要比这幅图所展示的复杂很多。不过我们先不用理会这里所说的复杂关系。让我们再把焦点扩大一些，看看它们与内核调度实体（KSE）之间的关系是怎样的，如图6-30所示。

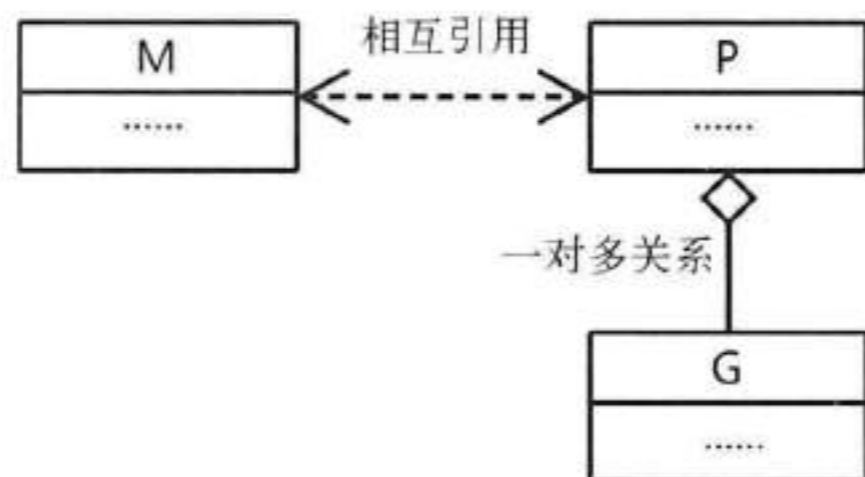


图6-29 Go语言的线程实现模型中的3个核心元素

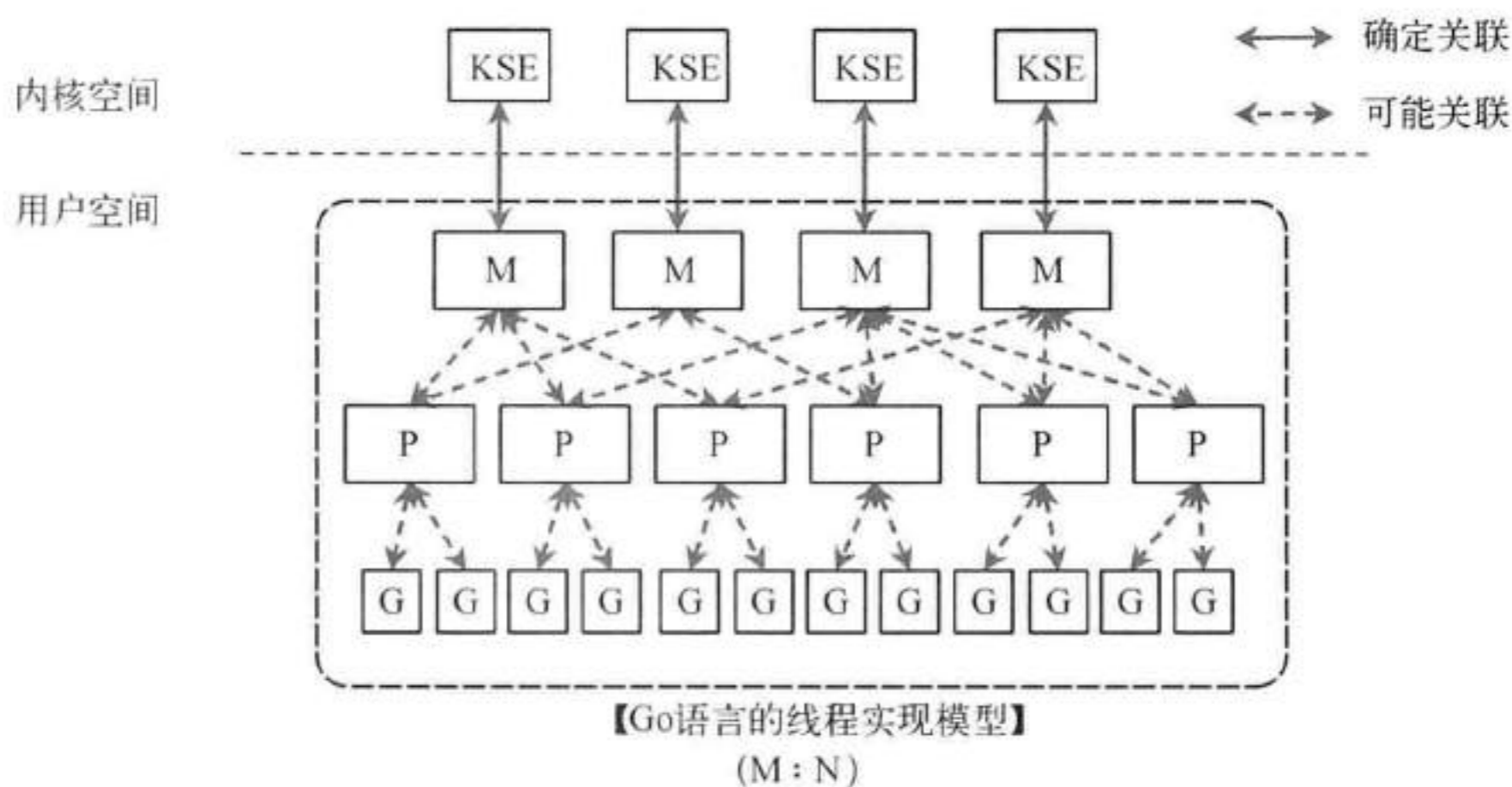


图6-30 M、P、G与KSE的关系

可以看到，M与KSE之间总是一对一的。一个M能且仅能代表一个内核线程。Go语言的运行时系统（runtime system）用它来代表一个内核调度实体。M与KSE之间的关联是非常稳固的。也就是说，在一个M的生命周期内，它会且仅会与一个KSE产生关联。相比之下，M与P以及P与G之间的关联都是易变的。它们之间的关系会在实际调度的过程中被改变。其中，M与P之间也总是一对一的，而P与G之间则是一对多的（还记得我们刚刚说过的P中的待运行的G的队列吗？）。注意，由于M、P和G之间的关系在实际调度过程中的多变性，所以图6-30中所示的可能关联仅能作为一般性的示意。此外，M与G之间也会建立关联，因为一个G终归会由一个M来负责运行。但是，相比之下，这种关联并不是这一模型中的重要关系。所以，为了突出重点，我们在前面两幅图中并没有体现出这种关联。

至此，我们已经知道了这些核心实体之间可能存在的关系。Go语言的运行时系统会对这些实体的实例进行实时管理和调度。我们在下一小节会专门对此进行介绍。现在，让我们再次聚焦，看一看在这些实体内部都有哪些细节值得关注。

1. M

一个M代表了一个内核线程。在大多数情况下，创建一个M的原因都是由于没有足够的M来关联P并运行其中的可运行的G。不过，在运行时系统执行系统监控或垃圾回收等任务的时候也会导致新的M的创建。M的部分结构如图6-31所示。

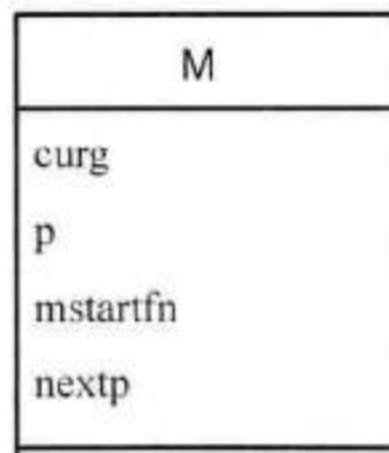


图6-31 M的结构（部分）

M结构中的字段众多。我们在这里只是挑选了对于我们初步认识M最重要的4个字段。其中，字段curg会存放当前M正在运行的那个G的指针，字段p会指向与当前M相关联的那个P，而字段mstartfn则代表了我们马上就会讲到的M的起始函数。在M被调度的过程中，这3个字段是最能体现它的即时情况的。而另外的字段nextp则会被用于暂存与当前M有潜在关联关系的P。我们可以把调度器将某个P赋给某个M的nextp字段的操作称为对M和P的预联。在有些时候，运行时系统会把刚刚被重新启用的M和已与它预联的那个P关联在一起。这就是nextp字段所起到的作用。

图6-31从侧面体现出了M与P和G之间可能建立的主要关联。请读者首先记住它，并带着它理解后面的内容。

M在被创建之初会被加入到全局的M列表（runtime.allm）中。紧接着，它的起始函数和准备关联的P（大多数情况下是导致此M创建操作的那个P）会被设置。最后，运行时系统会为它专门创建一个新的内核线程并与之相关联。这样，这个新的M就为执行G做好了准备。其中，起始函数仅当运行时系统要用此M执行系统监控或垃圾回收等任务的时候才会被设置。而这里的全局M列表其实并没有什么特殊的意义。运行时系统在需要的时候会通过它获取到所有M的信息。同时它也可以防止M被当作垃圾回收掉。

在新的M被创建完成之后会先进行一番初始化工作。其中包括了对自身所持的栈空间以及信号处理方面的初始化。在这些初始化工作都完成之后，该M的起始函数会被执行（如果存在的话）。注意，如果这个起始函数代表的是系统监控任务的话，那么该M会一直在那里执行而不会继续后面的流程。否则，在初始函数被执行完毕之后，当前M将会与那个准备与它关联的P完成关联。至此，一个并发执行环境才真正形成。在这之后，M开始寻找可运行的G并运行之。这一过程可以被看作是调度的一部分。我们在下一小节再细说。

运行时系统所管辖的M（或者说runtime.allm中的M）有时候会被停止，比如在运行时系统准

备开始执行垃圾回收任务的时候。运行时系统在停止M的时候，会在对它的属性进行必要的重置之后，把它放入调度器的空闲M列表（`runtime · sched.midle`）。这很重要，因为在需要一个未被使用的M的时候，运行时系统会先尝试从该列表中获取。

注意，M本身是无状态的。M是否空闲仅以它是否存在于调度器的空闲M列表中为依据。虽然运行时系统可以通过全局M列表获取到所有的M，但是却无法得知它们的状态（因为它们没有状态）。

单个Go程序所使用的M的最大数量是可以被设置的。在我们使用命令运行Go程序的时候，一个引导程序先会被启动。这个引导程序会为Go程序的运行建立必要的环境。引导程序会对M的最大数量进行初始设置。这个初始值是10000。也就是说，一个Go程序最多可以使用10000个M。这就意味着，在最理想的情况下，同时可以有10000个内核线程被同时运行。请注意，这里说的是最理想的情况。由于操作系统内核对进程的虚拟内存的布局的控制以及大小的限制，如此量级的线程可能很难共存。从这个角度看，Go语言本身对于线程数量的限制几乎可以被忽略。

除了上述的初始设置之外，我们也可以在Go程序中对限制进行设置。为了达到此目的，我们需要调用标准库代码包`runtime/debug`包中的`SetMaxThreads`函数并提供新的M最大数量。`runtime/debug.SetMaxThreads`函数在被执行完成后，会把旧的M最大数量作为结果值返回。非常重要的一点是，如果我们在调用`runtime/debug.SetMaxThreads`函数时给定的新值比当时M的实际数量还要小的话，运行时系统就会发起一个运行时恐慌。所以，我们要小心使用这个函数。请记住，如果我们真的需要设置M的最大数量，那么越早调用`runtime/debug.SetMaxThreads`函数就越好。对于它的设定值，我们也要仔细地斟酌。

2. P

P是使G能够在M中运行的关键。Go语言的运行时系统会适时地让P与不同的M建立或断开关联，以使P中的那些可运行的G能够在需要的时候及时获得运行时机。这与操作系统内核在CPU之上实时的切换不同的进程或线程的情形是类似的。

通过调用函数`runtime.GOMAXPROCS`，我们可以改变单个Go程序可以间接拥有的P的最大数量。除此之外，我们还可以在运行Go程序之前设置环境变量`GOMAXPROCS`的值来对Go程序可以拥有的P的最大数量做出预先设定。P的最大数量相当于是对可以被并发运行的用户级别的G的数量做出限制。（关于什么是用户级别的G，我们会在6.6.3节予以说明。）我们已经知道，每个P都需要关联一个M才能使其中的可运行的G得到执行。但是这却不意味着环境变量`GOMAXPROCS`的值会限制住M的总数量。当M因系统调用的进行而被阻塞（更确切地说，是它运行的G进入了系统调用）的时候，运行时系统会将该M和与之关联的P分离开来。这时，如果这个P的可运行G队列中还有未被运行的G，那么运行时系统就会找到一个空闲M，或创建出一个新的M，并与该P关联以满足这些G的运行需要。如果我们在Go程序中创建的大部分Goroutine中都包含了很多需要间接地进行各种系统调用（比如各种I/O操作）的代码的话，那么即使环境变量`GOMAXPROCS`的值被设定为1，也可能会有多个M被创建出来。所以，实际的M总数量很可能会比环境变量`GOMAXPROCS`所指代的数量多。由此可见，Go程序真正使用的内核线程的数量并不会因此而受到限制。

在Go程序开始被运行的时候，我们在前面提到的引导程序也会对P的最大数量进行设置。P的最大数量的默认值是1。因此，在默认情况下，无论我们在程序中用go语句启用出多少个Goroutine，它们都只会被塞入同一个P的可运行G的队列中。不过要注意，正如前文所说，这并不意味着只会有一个与内核线程——对应的M去运行它们。当环境变量GOMAXPROCS的值大于0的时候，引导程序会认为我们要对P的最大数量进行设置。它会先检查一下我们设置的值的有效性。如果该值不大于运行时系统对此设定的硬性上限值，那么此值就被认为是有效的，否则该值就会被这个硬性限制取代。也就是说，最终的P最大数量值绝不会比引导程序中的这个硬性上限值大。该硬性上限值是2的8次方，即256。这个硬性上限值为256的原因是Go语言目前还不能保证在数量比256更多的P同时存在的情形下Go程序仍能保持高效。也就是说，这个硬性上限值并不是永久的。它在以后可能会被改变。

注意，虽然我们可以在应用程序中随意地调用runtime.GOMAXPROCS函数，但是它的执行会暂时使所有的P都相继进入停止状态并试图阻止任何用户级别的G的运行。只有在新的P最大数量被设定完成之后，运行时系统才会开始陆续恢复它们。这对于程序的性能是非常大的损耗。所以，我们最好只在Go程序的main函数的开始处调用runtime.GOMAXPROCS函数。当然，在Go程序中不对它进行调用而只预先设置环境变量GOMAXPROCS是最好不过的了。

在确定P的最大数量之后，运行时系统会根据这个数值初始化全局的P列表（runtime·allp）。与全局M列表类似，该列表中包含了当前运行时系统创建的所有P。随后，运行时系统会把调度器的可运行G队列（runtime·sched.runq）中的所有G均匀的放入到全局P列表中的各个P的可运行G队列当中。至此，运行时系统需要用到的所有P都已就绪。至于这里的调度器的可运行G队列的用途以及其中的G是从哪里得来的，我们在后面会陆续说明。

与空闲M列表类似，在运行时系统中也存在着一个调度器的空闲P列表（runtime.sched.pidle）。当一个P不再与任何M关联的时候，运行时系统就会把它放入到该列表，而当运行时系统需要一个空闲的P关联某个M的话，会从此列表中取出一个。由此我们也可知这个空闲P列表的准入条件。注意，即使P进入到了空闲P列表中，它的可运行G列表也不一定是空的。这两者之间没有必然的联系。

与M不同，P本身是有状态的。一个P可能具有的状态如下。

- **Pidle**：此状态表明当前P未与任何M存在关联。
- **Prunning**：此状态表明当前P正在与某个M关联。
- **Psyscall**：此状态表明当前P中的被运行的那个G正在进行系统调用。
- **Pgctest**：此状态表明运行时系统正在进行垃圾回收。在运行时系统进行垃圾回收的时候，会试图把全局P列表中的都置于此状态。
- **Pdead**：此状态表明当前P已经不会再被使用。当我们在Go程序运行的过程中通过调用runtime.GOMAXPROCS函数减少P最大数量的时候，多余的P就会被运行时系统置于此状态。

P的初始状态是Pgctest，虽然运行时系统并不会在这时进行垃圾回收。不过，P处于这一初始状态的时间会非常短暂。在紧接着的初始化和填充P中的可运行G队列之后，运行时系统会将其状态设置为Pidle并放入到调度器的空闲P列表中。此空闲P列表中的所有P都会由调度器根据实

际情况进行取用。图6-32描绘了P在各个状态之间进行流转的具体情况。

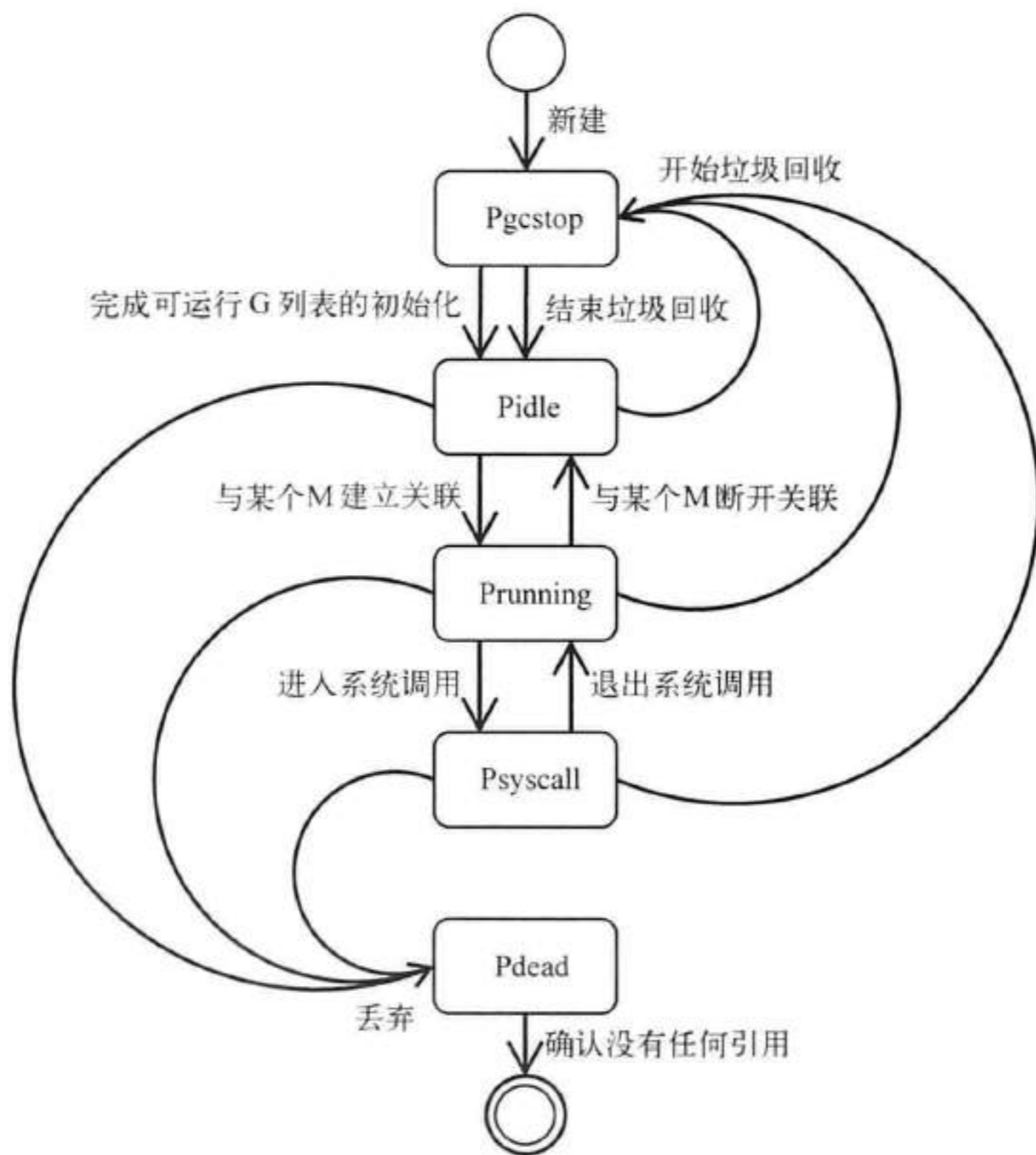


图6-32 P的状态转换

我们可以看到，除了Pdead之外的其他状态的P都会在运行时系统欲进行垃圾回收的时候被置于Pgcstop状态。但是，等到垃圾回收结束之后，它们并不会被恢复至原有状态，而会被统一地转换为Pidle状态。这就意味着它们会被重新调度。这是合理的。因为对于原有状态各异的众多P来说，这样做是最简单和有效的，当然也是最公平的。另一方面，除了Pgcstop状态，处于其他状态的P都可能由于全局P列表的缩小而被认为是多余的并被置于Pdead状态。不过，我们并不担心其中的G会失去归宿。因为，在P被转换为Pdead状态之前，它的可运行G队列中的G都会被转移至调度器的可运行G队列中，而它的自由G列表（马上就会讲到）中的G也都会被转移到调度器的自由G列表中。

我们已经知道，每个P中都一个可运行G队列。不过正如我们刚才所述，它们还都包含了一个自由G列表（gfree）。其中包含了一些已经被运行完成的G。随着被运行完成的G的增多，该列表可能会很长。如果它增长到了一定的程度，运行时系统会把其中的部分G转移到调度器的自由G列表（runtime·sched.gfree）中。另一方面，当我们使用go语句欲启用一个G的时候，运行时系统会先试图从相应P的自由G列表中获取一个现成的G来封装我们提供的函数，仅当获取不到这样

一个G的时候才有可能去创建一个新的G。考虑到由于相应P的自由G列表为空而获取不到自由G的情况,运行时系统若在这个过程中发现其中的自由G太少,则会先尝试从调度器的自由G列表中转移过来一些G。这样,只有在调度器的自由G列表也弹尽粮绝的时候才会有新的G被创建。这在很大程度上提高了G的复用率。顺便提一句,当一个P被运行时系统认为不会再被使用(会被置于Pdead状态)的时候,其中的自由G列表中的所有G会都被转移至调度器的自由G列表中。

在P的结构中,可运行G队列和自由G列表是最重要的两个成员。至少对于我们这些Go语言的使用者来说是这样。它们间接地体现了运行时系统对相应的G的调度情况。下面我们就对模型中离我们最近的G进行介绍。

3. G

一个G就相当于一个Goroutine(或称Go程),也与我们使用go语句欲并发执行的一个匿名或命名的函数相对应。我们作为编程人员只是使用go语句向Go语言的运行时系统告知了(或提交了)一个并发执行任务,而Go语言的运行时系统则会按照我们的要求并发地执行并完成这一任务。

Go语言的编译器会把我们编写的go语句(go关键字和其后的函数的统称)变成对一个运行时系统中的函数调用,并把go语句中的那个函数(以下简称go函数)及其参数都作为参数传递给这个运行时系统中的函数。这也是我们应该了解的第一件与go语句有关的事。其实它并不神奇,只是代表了我们向运行时系统递交的一个并发任务而已。

运行时系统在接到这样一个调用之后,会先检查一下go函数及其参数的合法性,紧接着会试图从本地P的自由G列表和调度器的自由G列表获取可用的G(我们刚刚讲过)。如果没有获取到则只好新建一个G了。与M和P相同,运行时系统也持有一个G的全局列表(runtime·allg)。新建的G会在第一时间被加入到该列表中。类似地,该列表的主要作用也是集中存放当前运行时系统中的所有G的指针。无论将会封装当前的这个go函数的G是否是新的,运行时系统都会对它进行一次初始化。其中包括了关联go函数以及设置该G的状态和ID等步骤。在初始化完成后,这个G会被放入到本地P的可运行G队列中。如果时机成熟,调度会立即进行以使这个G尽快被运行。不过,即使这里不立即调度,我们 also 无需担心该G不能被及时运行,因为运行时系统总是在不停地为了及时运行各个可运行的G而忙碌着。

每个G都会由运行时系统根据其实际状况设置不同的状态,其可能的状态如下。

- ❑ Gidle: 在当前G被创建但还完全未被初始化的时候会处于此状态。
- ❑ Grunnable: 表示当前G是可运行的,并且正在等待被运行。
- ❑ Grunning: 表示当前G正在被运行。
- ❑ Gsyscall: 表示当前G正在进行系统调用。
- ❑ Gwaiting: 表示当前G正在因某个原因而等待。
- ❑ Gdead: 表示当前G已被运行完成。

我们之前讲过,在运行时系统想用G封装我们通过go语句递交的go函数的时候,会先对这个G进行初始化。其中的一步就是初始化这个G的状态,而这个状态总会是Grunnable。也就是说,一个G真正开始被使用是在其状态被设置为Grunnable之后。图6-33展示了G在其生命周期内

的状态流转情况。

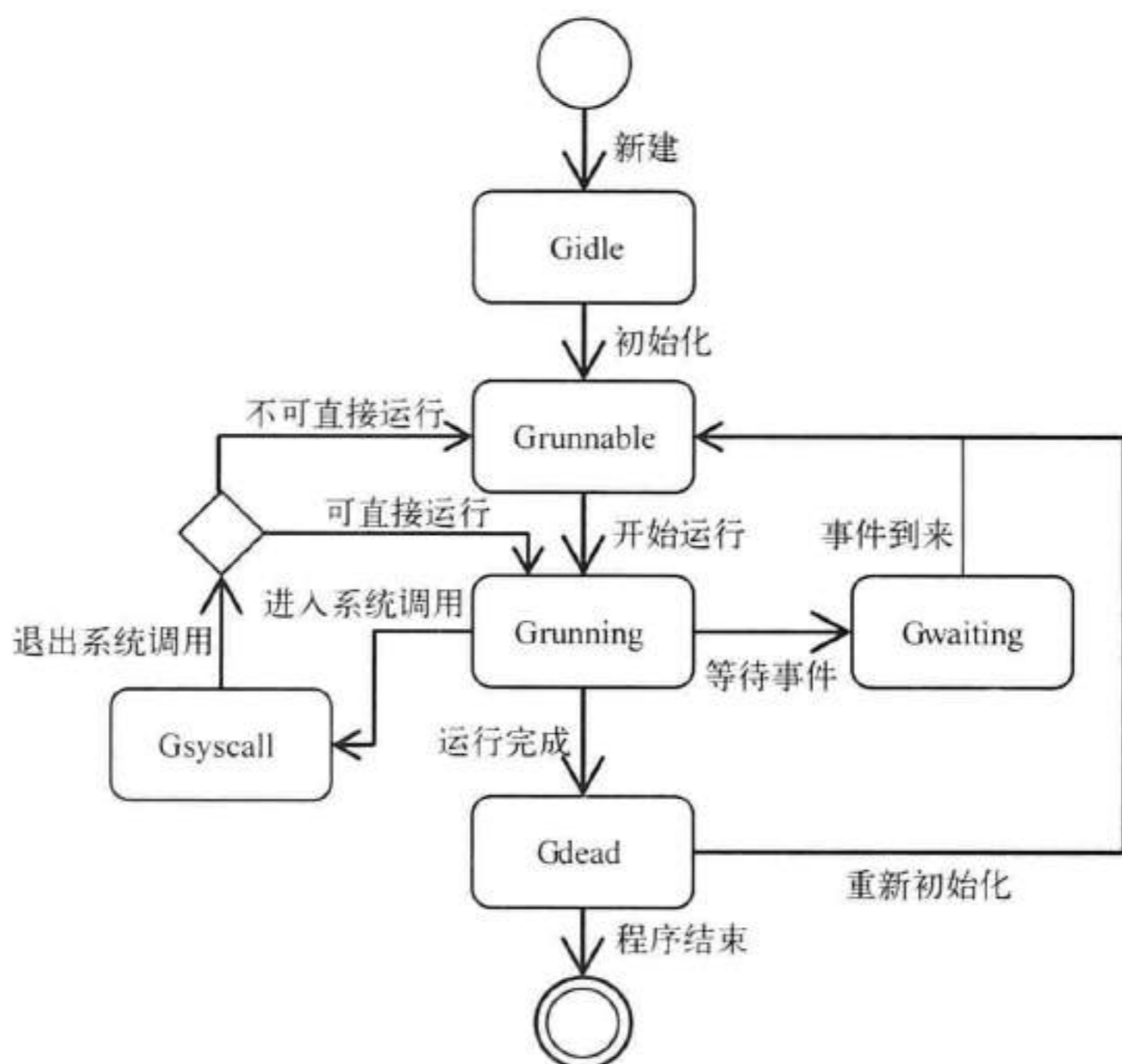


图6-33 G的状态转换

一个G在被运行的过程当中，是否会等待某个事件以及会等待什么样的事件，完全由其封装的go函数决定。例如，如果这个函数中包含了对通道类型值的操作，那么在执行到对应的代码的时候这个G就有可能进入Gwaiting状态。这可能是在等待从通道类型值中接收值，也可能是在等待向通道类型值发送值。又例如，涉及网络I/O的时候也会导致相应的G进入Gwaiting状态。此外，操纵定时器（`time.Timer`）和调用`time.Sleep`函数同样会造成相应G的等待。在事件到来之后，G会被“唤醒”并被转换至Grunnable状态。待时机到来时，它会再次被运行。

G在退出系统调用的时候的状态转换要比上述情况复杂一些。运行时系统先会尝试直接运行这个G，仅当无法直接运行的时候，才会把它转换为Grunnable状态并放入到调度器的自由G列表中。显然，对这样一个G来说，在其退出系统调用之时就被立即继续运行是再好不过的了。运行时系统当然会为此做出一些努力。不过，即使努力失败了，该G也还是会在实时的调度过程中被发现并运行。

最后，值得一提的是，进入死亡状态（Gdead）的G是可以被重新初始化并使用的。相比之下，P在进入死亡状态（Pdead）之后则只能面临被销毁的结局。由此也可以说明Gdead状态与Pdead状态所表达的含义是截然不同的。处于Gdead状态的G会被放入本地P或调度器的自由G列表，这为它们的重用提供了条件。

至此，我们基本了解到一个G在运行时系统中的流转方式和时机，这也展现了一条go语句背

后所蕴藏的玄机。

4. 核心元素的容器

我们刚刚讲述的是Go语言的线程实现模型中的3个核心元素——M、P和G。同时我们也多次提到了承载这些元素的实例的容器——各种队列和列表。我们现将这些容器归纳一下，如表6-5所示。

表6-5 M、P和G的容器

中文名称	源码中的名称	作用域	简要说明
全局M列表	runtime.allm	运行时系统	被用于存放所有M的列表
全局P列表	runtime.allp	运行时系统	被用于存放所有P的列表
全局G列表	runtime.allg	运行时系统	被用于存放所有G的列表
调度器的空闲M列表	runtime · sched.midle	调度器	被用于存放空闲M的列表
调度器的空闲P列表	runtime · sched.pidle	调度器	被用于存放空闲P的列表
调度器的可运行G队列	runtime · sched.runq	调度器	被用于存放可运行G的队列
调度器的自由G列表	runtime · sched.gfree	调度器	被用于存放自由G的列表
P的可运行G队列	runq	本地P	被用于存放当前P中的可运行G的队列
P的自由G列表	gfree	本地P	被用于存放当前P中的自由G的列表

在这些容器中，全局的那3个列表存在的主要目的都分别是为了统计运行时系统中的所有M、P或G。相比之下，最应该值得我们关注的是那些非全局的容器，尤其是与G相关的那4个容器。

与G有关的非全局容器有可运行G队列、调度器的自由G列表、本地P的可运行G队列以及本地P的自由G列表。运行时系统创建出的任何G都会存在于全局G列表中。而其余的4个列表则只会存放在当前作用域内的具有特定状态的G。注意，这里的两个可运行G列表中的G都拥有几乎平等的运行机会。在运行时系统调度的过程中会先后对它们进行检查，并会立即运行第一个被发现的可运行的G。由于这种平等性的存在，所以我们无需关心哪类可运行的G会进入到哪一个队列中。不过，可以顺便提一下，从Gsyscall状态和Ggcstop状态转出的G，都会被放入调度器的可运行G队列，而被运行时系统初始化的G，都会被放入本地P的可运行G队列。至于从Gwaiting状态转出的G，除了因进行网络I/O而陷入等待的G之外，都会被放到本地P的可运行G队列中。此外，我们之前说过，对runtime.GOMAXPROCS函数的调用，可能会导致运行时系统清调度器的可运行G队列。其中的所有G都会被均匀地放入到全局P列表中的各个P的可运行G队列当中。另一方面，在G转入Gdead状态之后，首先会被放入本地P的自由G列表，而在运行时系统需要用自由G封装go函数的时候，也会先尝试从本地P的自由G列表中获取。调度器的自由G列表只是起到了一个暂存自由G的作用。这方面内容我们在前面已有所描述。

与M和P相关的非全局容器分别是调度器的空闲M列表和调度器的空闲P列表。这两个列表都被用于存放暂时不被使用的元素的实例。在运行时系统有需要的时候，会从中获取相应元素的实例并重新启用它。

在本小节中，我们一直把实现和操纵Go语言的线程实现模型的内部程序统称为运行时系统。实际上，我们应该把它叫作调度器。调度器拥有自己的结构，也依此提供了一些很重要的功能。

其实，其中的大部分功能我们在本小节都已经接触到了。比如，对各类元素实例之间的关联的管理、对各个元素实例状态的转换以及它们在不同核心元素容器间的转移，等等。只不过，我们是围绕着各种模型元素（M、P、G以及各种容器）来对它们进行介绍的。这可能会让读者对调度器负责执行的流程依然感到有些模糊。不过别担心，在下一小节，我们就对调度器的结构以及与之相关的几个重要流程进行介绍。

6.6.2 调度器

我们在上一节讲过，两级线程模型中的一部分调度任务会由操作系统内核之外的程序承担。在Go语言中，其运行时系统中的调度器会负责这一部分调度任务。调度的主要对象是M、P和G的实例，调度的辅助设施是我们在上一小节的最后讲到的各种容器。其实每个M（即每个内核线程）在运行过程中都会按需执行一些调度任务。不过为了更加容易理解，我们把这些调度任务统称为调度器的调度行为。在本节，我们会了解到这些调度行为的核心流程。

1. 基本结构

调度器有它自己的数据结构。这一数据结构的主要目的就是为了更加方便地管理和调度各个核心元素的实例。在这些字段中，有我们已经熟知的空闲M列表、空闲P列表、可运行G队列和自由G列表。下面，我们再来讲解另外的一些字段，如表6-6所示。

表6-6 调度器的字段（部分）

字段名称	数据类型	用途简述
gcwaiting	uint32	作为垃圾回收任务被执行期间的辅助标记、停止计数和通知机制
stopwait	int32	
stopnote	Note	
sysmonwait	uint32	作为系统监测任务被执行期间的停止计数和通知机制
sysmonnote	Note	

在这张表中，我们只罗列了几个比较重要的字段。它们都与运行时系统执行的垃圾回收任务有关。

字段gcwaiting、stopwait和stopnode都是运行时系统中的垃圾回收器在进行垃圾回收时的辅助协调手段之一。由调度器的字段gcwaiting的值，我们可以知道垃圾回收器是否已经开始准备或正在进行垃圾回收。stopwait字段是为了对还未被停止调度的P进行计数。当该计数为0的时候，就说明调度工作已被完全停止。这时，垃圾回收器会立即开始执行垃圾回收任务。而stopnode字段就是被用来向垃圾回收器告知调度工作已经完全被停止的通知机制的重要部分。

这些辅助协调手段存在的意义在于保证所有的P在垃圾回收期间都处于Pgstop状态。这样有助于最大化垃圾回收的效果。更明确地说，Go语言的垃圾回收器的做法是：先停止一切调度工作（包括停止对M和P的调度，等等），然后进行垃圾回收，最后待垃圾回收完成之后再重启调度工作。这意味着Go语言的垃圾回收任务是在“Stop the world”的环境下被执行的。“Stop the world”即指运行时系统要放下手头所有工作并专心（无其他并发任务）执行垃圾回收任务。

垃圾回收器在准备执行垃圾回收任务的时候会先把调度器的gcwaiting字段的值设置为1。这是为了要告诉调度器，它已经开始准备执行垃圾回收任务。垃圾回收器会利用stopnode字段将自身阻塞住，以等待调度器完全停止调度。调度器在发现gcwaiting字段的值被设置为1之后，会积极响应，并陆续停止正在进行的调度工作。待所有的调度工作均已停止（此时作为计数器的stopwait字段的值变为0）之后，调度器会利用stopnode字段向垃圾回收器发送通知。后者在收到通知后才会真正开始垃圾回收。我们无需关心stopnode字段的数据类型Node到底是怎样的类型。不过，顺便提一句，此通知机制在底层是由信号灯实现的。

现在，我们再来看调度器的另外两个字段——sysmonwait和sysmonnote。它们与前面那一组字段的用途类似，只不过它们针对的是系统监测任务（我们稍后会介绍它）。在垃圾回收器进行垃圾回收的时候，被持续执行的系统监测任务也需要被暂停。而这两个字段的作用就是及时地暂停和恢复系统监测任务的执行。sysmonwait字段是表示系统监测任务是否已被暂停的标记，而sysmonnote字段则是被用来向执行系统监测任务的程序发送通知的。

系统监测任务是被持续执行的。更确切地说，它被置于了无尽的循环之中。在每次迭代之初，相关程序（或称系统监测器）会先检查调度器的gcwaiting字段的值。若发现其值为1，则说明垃圾回收器已经开始准备或正在执行垃圾回收任务。这时，系统监测器会先将调度器的sysmonwait字段的值设置为1以表示系统监测任务已被暂停。然后利用sysmonnote字段阻塞自身以等待垃圾回收的完成。在调度工作被重启之后，调度器若发现其sysmonwait字段值为1则会利用sysmonnote字段向系统监测器发送通知。系统监测器在收到该通知之后会立即执行当次迭代的后续流程并继续进行之后的迭代。

我们看到，调度器的这5个字段都是为了辅助垃圾回收的执行而存在的。垃圾回收任务被执行期间的“Stop the world”环境是由它们帮助构建的。

2. 一轮调度

我们已经知道，引导程序会为Go程序的运行建立必要的环境。在引导程序完成它的工作之后，Go程序的main函数才会被真正地执行。引导程序会在最后让调度器进行一轮调度，这样才能够让main函数所在的G马上有机会被运行（封装main函数的G总是Go语言运行时系统创建的第一个G）。我们现在就来看看调度器在一轮调度中都做了哪些工作。为了让读者对此能够先有一个宏观的了解，我们根据调度器在进行一轮调度的时候所执行的流程绘制了一幅流程图，如图6-34所示。

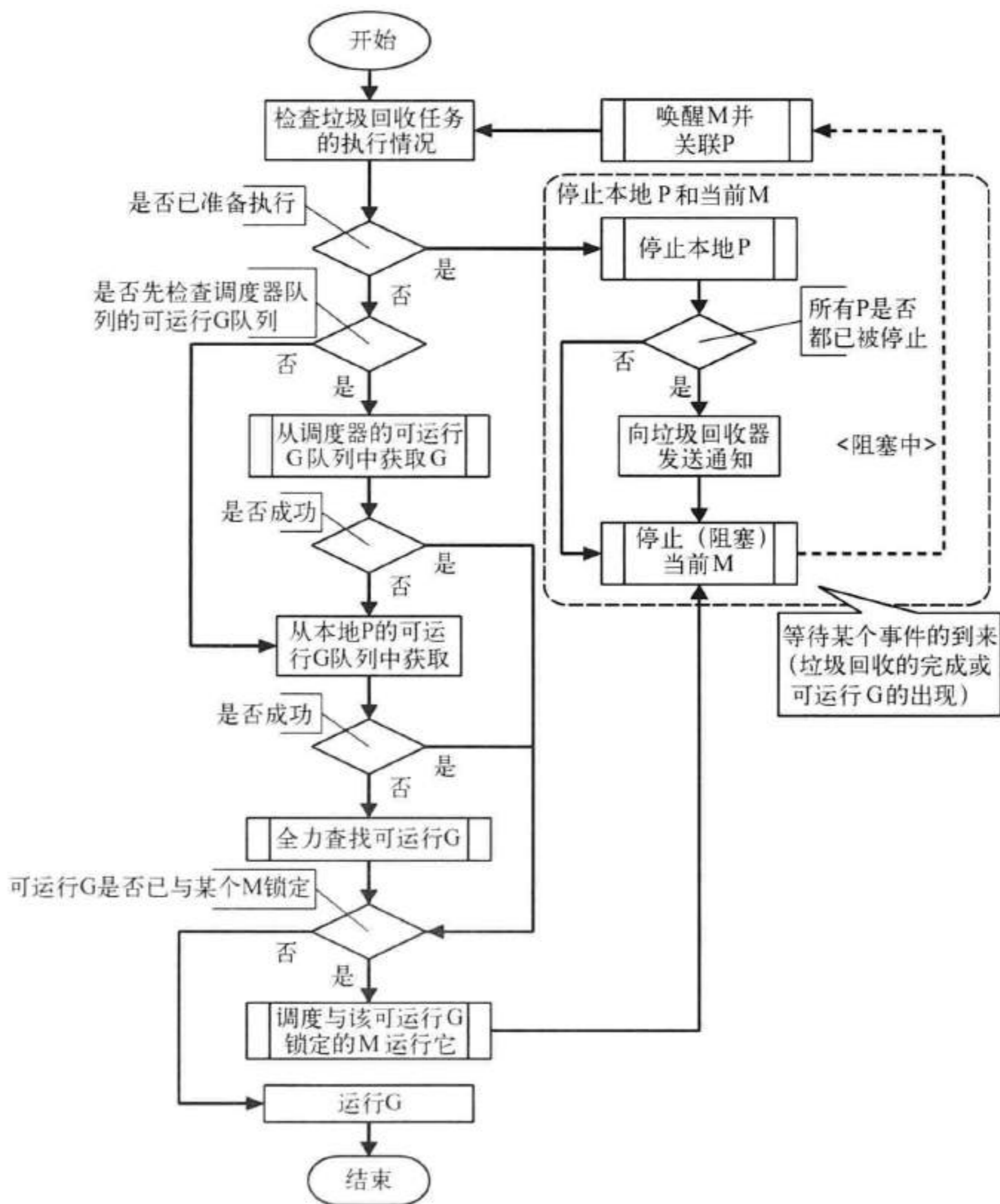


图6-34 一轮调度的总体流程

注意，为了突出重点，此流程图中只描绘了其中的一些重要步骤。本小节后续出现的流程图也会是如此。

在调度器的一轮调度总体流程中，一共有5个子流程。如，调度器在从它自己的可运行G队列中获取G的时候，并不只是查找那么简单。如果找到了一个可运行G，调度器还会把该G转移至本地P的可运行G队列中。又如，在发现垃圾回收器已经准备开始垃圾回收的时候，调度器会积极响应并停止本地P和当前M。停止这两者的工作都需要通过几个步骤来完成。我们会在解释一轮调度的总体流程的过程中适当地描述这些子流程。

作为响应垃圾回收任务的执行而进行的停止调度工作的一部分，调度器在开始进行一轮调度的时候会先检查它的`gcwaiting`字段的值。如果发现该值为1，那么调度器会立即停止本地P和当前M。在这里，停止本地P的工作需要两步，即断开本地P与当前M之间的关联和把这个（曾经的）本地P的状态置为`Pgcstop`。与其他停止P的流程一样，在调度器停止当前的P之后，总是要检查一下是否所有的P都已被停止。如果答案是肯定的，那么它就会立刻利用它的`stopnode`字段向垃圾回收器发送通知。这使得通知总会非常及时地被送达。无论怎样，当前M也是需要被停止的。停止当前M的步骤会稍稍复杂一些，如下所示。

- (1) 重置当前M的一些属性。
- (2) 把当前M放入到调度器的空闲M列表中。
- (3) 阻塞当前M。被阻塞的M会等待调度器的唤醒。

在垃圾回收结束之后，调度器会从它的空闲M列表中取出M并将它们唤醒。被唤醒的M会立即与一个可运行G队列不为空的P进行关联。随后，调度器会再为它进行新一轮的调度。正如图6-34所示。

只要错开了垃圾回收任务的执行时期，调度器就会试图在本地P的可运行G列表中查找可以被运行的G。有时候为了公平性，调度器也会先检查它自己的可运行G队列，并仅当该队列中无可运行的G的时候再从本地P的可运行G队列中获取。倘若从队列中不到可运行的G，那么调度器就会进入全力查找可运行G的子流程。鉴于这个子流程的复杂性，我们稍后会专门对它进行讲解。可以肯定的是，如果经过一番努力之后还是无法找到任何可运行G，该子流程就会被暂停，并且直到有可运行G出现才会继续下去。也就是说，这个全力查找可运行G的子流程的结束就意味着当前M抢到了一个可运行的G。

在得到一个可运行G之后，调度器在让当前M运行它之前还会判断一下该G是否已与某个M锁定。这里所说的锁定的含义是G中的一个标记被设置为某个M以表示它必须由该M运行。在M的结构中实际上也存在一个对应的标记并被用来表示某个G只能由它运行。这两个标记一定会被一起设置或重置。这样，在设置它们之后，相应的M和G就等于被捆绑在了一起。在Go程序中，我们可以通过调用`runtime.LockOSThread`函数，把当前的Goroutine与当时执行它的那个M捆绑在一起，也可以通过调用`runtime.UnlockOSThread`函数，把与当前Goroutine有关的捆绑解除。一个M只能与一个G捆绑，反之亦然。所以，如果我们多次调用`runtime.LockOSThread`函数，那么仅有最后一次调用是有效的。也就是说，之前的调用所产生的结果会被最后一次调用覆盖掉。另一方面，即使当前的Goroutine没有与任何M捆绑，我们调用`runtime.UnlockOSThread`函数也不会产生任何副作用。它会直接返回。

那么在什么时候才有必要把一个G和某个M捆绑在一起呢？答案是大多数情况下是完全没有必要的。如果我们没有编写过使用了某些C语言函数库（通过`cgo`）的Go语言程序的话，可能无法体会到其中的真正含义。有些C语言的函数库（比如OpenGL）会用到线程本地存储技术。也就是说，它们会把一些数据存储在当前的内核线程的私有缓存中。所以，如果我们让调度器任意选择运行它们的M（内核线程）的话，就意味着会丢失掉其存储在之前运行它的那个内核线程的私有缓存中的那些数据。这样往往会造成不可预估的问题。因此，让包含了此类操作的G只被同一

个M运行是非常有必要的。

回到刚才的话题。如果调度器发现它找到的这个可运行G已经与某个M锁定在了一起，那么它就会让与该G锁定的那个M去运行这个G。然后，停止当前的M（即让当前M继续等待其他可运行G）。在这之后的某个时刻，这个M会被唤醒。调度器同样会为它重新进行一轮调度。另一种情况，如果调度器发现它找到的可运行G未与任何M锁定，那么它就会直接让当前M去运行这个G。至此，调度器在一个M中的一轮调度才真正完成。

一轮调度是调度器中的核心流程。运行时系统在调度过程中会经常使用到它。比如，调度器在让某个G等待之后会进行一轮调度。又比如，在垃圾回收结束之时一轮调度流程也会被执行。再比如，在某个G退出系统调用的时候，运行时系统也会启动一轮调度的流程。除此之外，我们对runtime代码包中的一些函数的调用也会导致该流程的执行。例如，我们在为了让其他Goroutine有机会被运行而调用runtime.Gosched函数的时候，就相当于手动地让调度器进行了新一轮的调度。这也是其他Goroutine能够得到运行机会的真正原因。又例如，runtime.Goexit函数会终结调用它的那个Goroutine。调度器在结束那个Goroutine的运行之后，会立即进行一轮调度以使其他等待运行的Goroutine获得机会。

我们下面介绍的一些流程也会与一轮调度有关。请读者继续往下看。

3. 全力查找可运行的G

我们刚才提到，调度器在没有从相关队列中找到可运行G的时候，会进入全力查找可运行G的子流程。我们现在就来简要介绍一下这个子流程。该子流程会做如下的获取可运行G的尝试。

- (1) 从本地P的可运行G队列中获取G。
- (2) 从调度器的可运行G队列中获取G。
- (3) 从网络I/O轮询器（netpoller）处查找已经就绪的G。这样的G可以被当作可运行的G。
- (4) 在条件许可的情况下，从另一个P的可运行G队列中偷取可运行的G。
- (5) 再次尝试从调度器的可运行G队列中获取G。
- (6) 尝试从所有P的可运行G队列中获取G。
- (7) 再次尝试从网络I/O轮询器处查找已经就绪的G。

其中，从网络I/O轮询器处查找已经就绪的G是一个较复杂的过程。简单地说，网络I/O轮询器是Go语言为了在操作系统提供的异步I/O接口之上实现自己的阻塞式I/O而编写的一个子程序。它所选用的异步I/O接口都是可以对网络I/O的状态进行高效轮询的利器（比如poll和kqueue）。当一个Goroutine试图在一个网络连接上进行读或写操作的时候，底层程序会让网络I/O轮询器在它们准备好之后通知该Goroutine。在这之前，这个Goroutine会被迫转入等待状态（即Gwaiting状态），然后调度器会使它与运行它的那个M分离。在网络I/O轮询器从底层程序那里得知准备就绪的消息之后，会立即通知为此等待的Goroutine。因此，这里所说的从网络I/O轮询器处查找已经就绪的G的意思就是，获取这些已经接收到通知的Goroutine。它们既然已经可以进行网络读写操作了，那么调度器理应让它们从等待状态转出并调度某些M去运行它们。

无论在进行哪一次尝试的时候找到了可运行的G，调度器都会立即中止这个子流程并把找到的G返回给父流程。而万一不幸的事情发生了，即在做出如此多的尝试之后依然找不到可运行的

G, 调度器就会停止当前的M。当有可运行G出现时, 这个M会被唤醒。随后调度器会重新执行该子流程的全部或部分。假如永远找不到一个可运行的G或者即使有可运行的G出现, 也都被其他M中执行的调度程序抢走了, 那么该M中的这个全力查找可运行G的子流程就会被一直执行下去, 永远不会退出。这就是我们在前面所说的: 全力查找可运行G的子流程的结束就意味着当前的M抢到了一个可运行的G。

4. 启用或停止M

我们在本节中多次提到调度器有时会停止当前M。至于停止当前M的原因, 我们也提到过一些, 例如垃圾回收任务的执行和等待新的可运行G的到来, 等等。在调度器停止某个M之前一定会把它放入到自己的空闲M列表中, 而调度器准备唤醒的M一定是从它的空闲M列表中取出的。调度器只会停止当前M, 但却可以根据需要启用其他M。这一停一启充分体现了调度器对M的调度行为, 如图6-35所示。

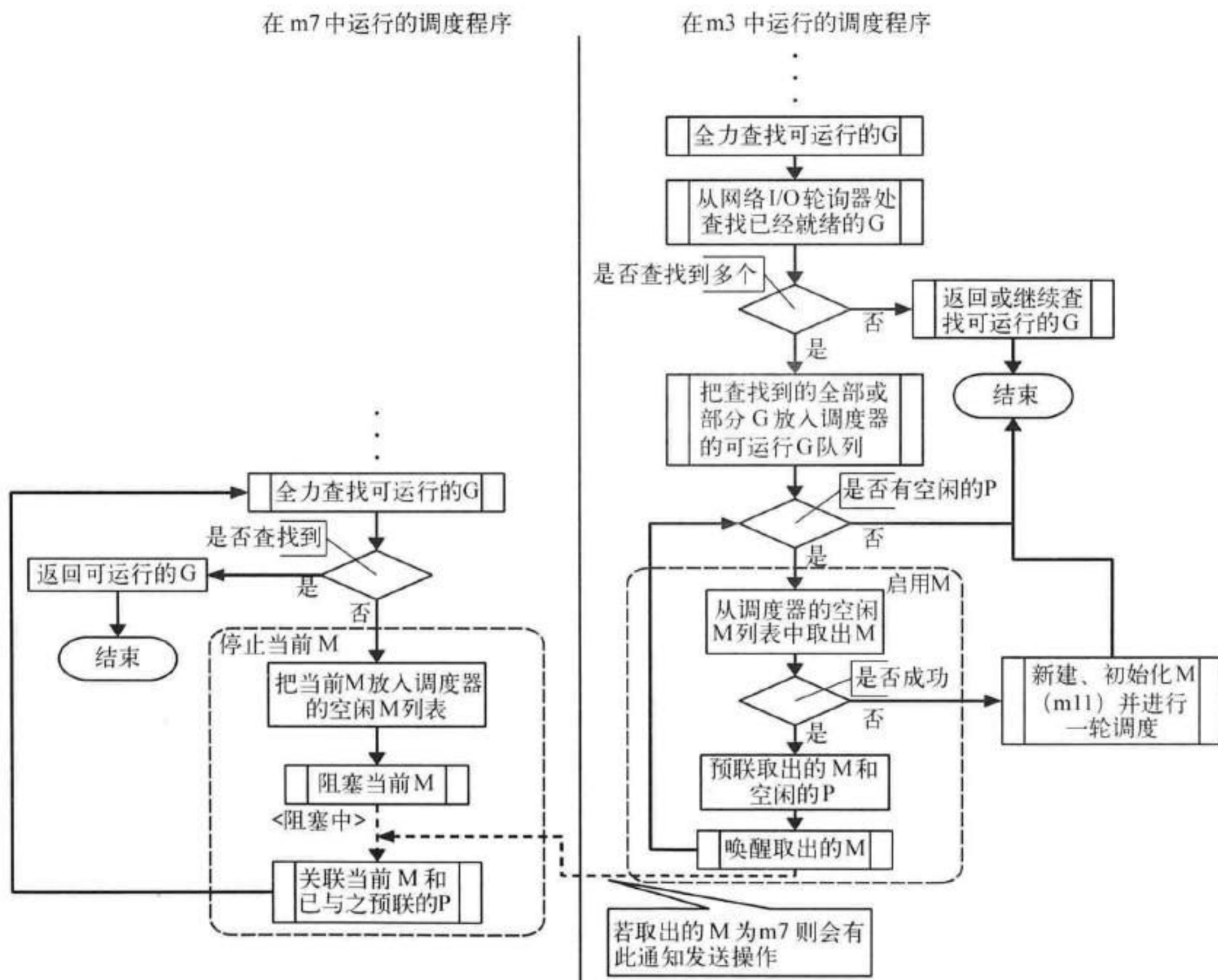


图6-35 启用或停止M

图6-35所描绘的流程是以调度器全力查找可运行G为背景的。我们放大了其中的一些部分。比如，若在全力查找之后仍未找到可运行的G，调度器会停止当前的M。又比如，在全力查找可运行G的过程中，调度器会从网络I/O轮询器处查找已经就绪的G。是否找到了G以及找到了多少个G决定了其之后的走向。总之，图6-35向我们展示了调度器停止当前M的一般方法以及该M被唤醒的时机。注意，这并不是调度器启停M的唯一场景，而只是众多类似的调度场景中的一例。

下面，我们对此图所展示的流程进行一些必要的解释。如图6-35所示，停止当前M总体来说需要两个步骤。在运行在m7中的调度程序发现无论如何也找不到可运行的G的时候，会把m7放入到调度器的空闲列表中，然后阻塞它以等待在其他M中运行的调度程序在发现多个可运行的G的时候向m7发送的通知。

另一方面，在m3中运行的调度程序在全力查找可运行G的过程中发现网络I/O轮询器处有多个已就绪的G。于是它把这些G作为可运行的G放入到了调度器的可运行G队列中。然后，调度程序会在有空闲的P（或者说有可用的上下文环境）的前提下从调度器的空闲M列表中取出一个M，并在预联这个M和那个空闲的P之后利用通知机制唤醒这个刚刚被取出的空闲M。如果这个M恰好是m7，那么m3中的调度器程序就会向m7发送通知以唤醒它。在m7被唤醒后，调度程序会立即关联m7和在m3中已与它预联的那个空闲的P。这时，m7已经有了新的上下文环境。随后，m7中的调度程序会重新全力查找可运行的G。当然，重新全力查找并不意味着一定会查找到可运行的G。因为它们可能已经被在其他M上运行的调度程序抢走了。实际上，这种情况并不是偶尔才发生的。也正因为如此，这个全力查找可运行G的流程才会如此往复，直至抢到一个可运行的G为止。

我们在讲解调度器的一轮调度流程的时候说过，如果调度器发现找到的可运行G已经与某个M锁定，那么调度器就会让那个M来运行这个G。也就是说，调度器会唤醒与这个可运行的G锁定在一起的那个M。为什么说唤醒而不用启用？这是因为，调度程序在发现当前M已与某个G锁定的时候会停掉当前M，并等待那个与之锁定的G变得可被运行。相关的流程如图6-36所示。

为了突出重点，该图隐藏了一些非关键的步骤。我们以调度程序处理一个刚刚退出系统调用的G的流程为例。这其中包含了我们已经知道的一些步骤和流程，比如一轮调度流程、停止M的流程，等等。如图所示，这个流程是在m2中被执行的。而在m6中，我们放大了一轮调度流程中的末尾部分，即根据判断被找到的可运行G是否已与某个M锁定的结果决定后续的操作（启用被锁定的M或者直接运行那个可运行的G）。

启用或停止被锁定的M的流程要比普通的M启停流程稍微复杂一些。我们先来看在m2中被执行的流程。在得到一个退出系统调用的G之后，运行在m2中的调度程序会立即尝试继续运行它。若存在空闲的P，那么调度程序就会直接运行这个G。否则，调度程序就会判断m2是否已被锁定。若结果是肯定的，则进入停止被锁定的M的子流程。如果答案是否定的，由于找不到一个空闲的P（或者说没有一个可用的上下文环境），调度器只能停止m2以等待一个空闲的P。还记得吗？一个代表了内核线程的M在与一个代表了上下文环境的P结合之后才能去运行一个G。

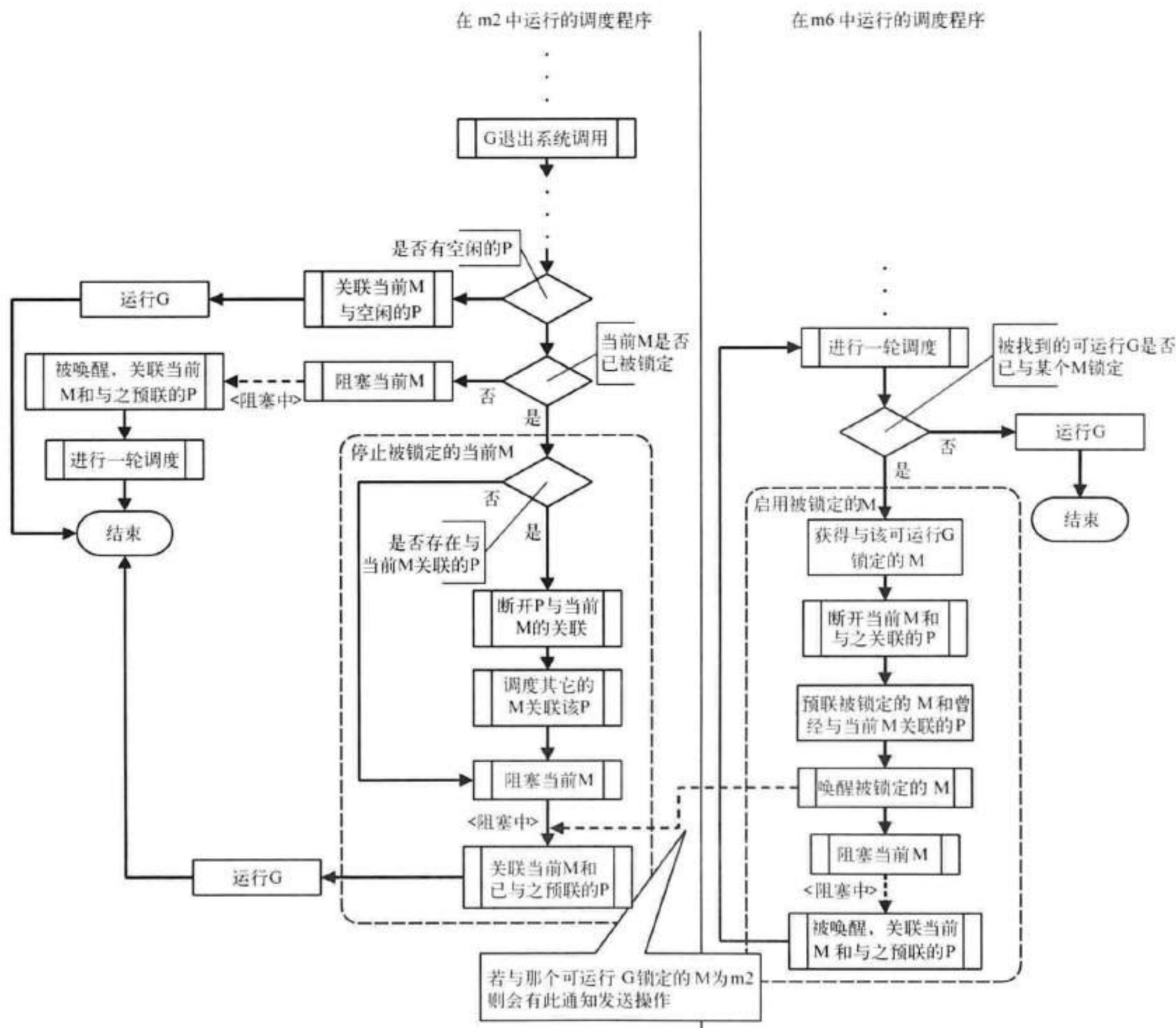


图6-36 启用或停止被锁定的M

我们继续关注m2已被锁定的情况。在停止被锁定的M的子流程中，调度程序会先断开与m2关联的P并促使它与其他M产生关联（如果存在这样一个P的话）。这样，即使停止了m2也不会浪费相关的上下文环境。随后，m2会被停止以等待运行与之锁定的G的时机。另一方面，如果m6中的调度程序发现它找到的可运行G已与某个M锁定了，它就会进入到启用被锁定的M的流程中。调度程序会先获取与该可运行G锁定在一起的那个M。然后，它会断开与m6关联的P，并把该P与这个被锁定的M预联。这既是为了拿掉当前M（即m6）的上下文环境，也是为了预设被锁定的M（这里是m2）的上下文环境。其背后的原因是，调度器会在唤醒m2之后停掉m6，m6不应再与任何P有关联。在m2被唤醒之后，其中的调度程序会把它与那个曾经属于m6的P关联在一起。至此，运行在不同的M中的调度程序共同完成了对上下文环境（也就是P）的转移。这一步非常关键。在这之后，m2就可以运行与它锁定的那个G了。

在高并发的Go程序中，启停M的流程在调度器中经常会被执行。因为并发量越大，调度器对M、P和G的调度就越频繁。各个Goroutine总是会通过这样或那样的途径使用到操作系统的提供的各种接口，也会经常使用到Go语言本身提供的各种组件（比如Channel和Timer，等等）。这些操作都直接或间接的涉及了启停M的流程。由此可见，此流程可以算得上是调度器乃至运行时系统中的核心流程了。

5. 系统监测任务

我们在讲解调度器的字段的时候提到过系统监测任务。那时我们着重解释了系统监测任务是怎样配合垃圾回收任务而执行的。现在我们来专门介绍这一任务。此任务本身并不复杂，而且其中涉及的一些子流程我们在前面已经详细的说明过了。当然，其余的部分也是值得介绍的。我们先来了解一下系统监测任务的总体流程，如图6-37所示。

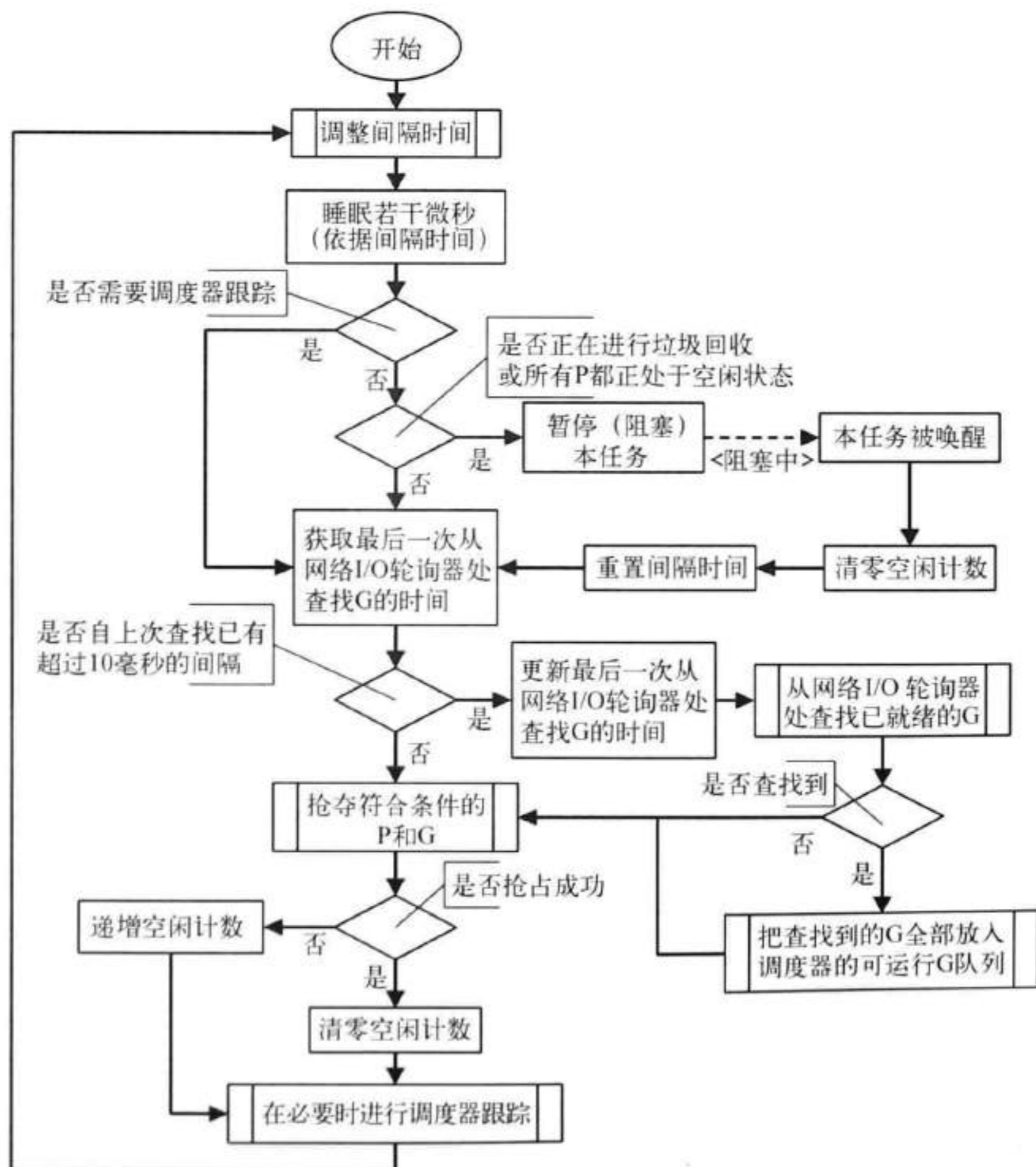


图6-37 系统监测任务的总体流程

概括来说，这个系统监测任务做了如下3件事。

- 在必要的时候，从网络I/O轮询器处查找已就绪的G，并把它们放入调度器的可运行G队列。
- 抢夺符合条件的P和G。
- 在必要的时候，进行调度器跟踪并打印出相关信息。

该任务做第一件事是为了帮助调度器从网络I/O轮询器那里及时的找回一个可以被运行的G。做第二件事的目的是周期性地为调度工作查缺补漏。而做第三件事则完全是为了调试（它会打印出一些调度过程的信息）。当然，除此之外，系统监测任务中还包括了一些其他操作。比如，根据上次的监测情况（由空闲计数和间隔时间代表）决定本次监测的延迟时间。又比如，在垃圾回收正在进行或所有P都处于空闲状态的时候暂停，并在接收到通知后继续执行。这其中最值得一提的当属抢夺符合条件的P和G这个子流程了。

在抢夺P和G的子流程中，所有的P都会被检查。程序会先查看P的状态。如果它的状态为Psyscall或Prunning，那么程序会对它进行进一步检查并在必要时进行相应的调度。图6-38展示了程序对一个P的检查和处理的流程。

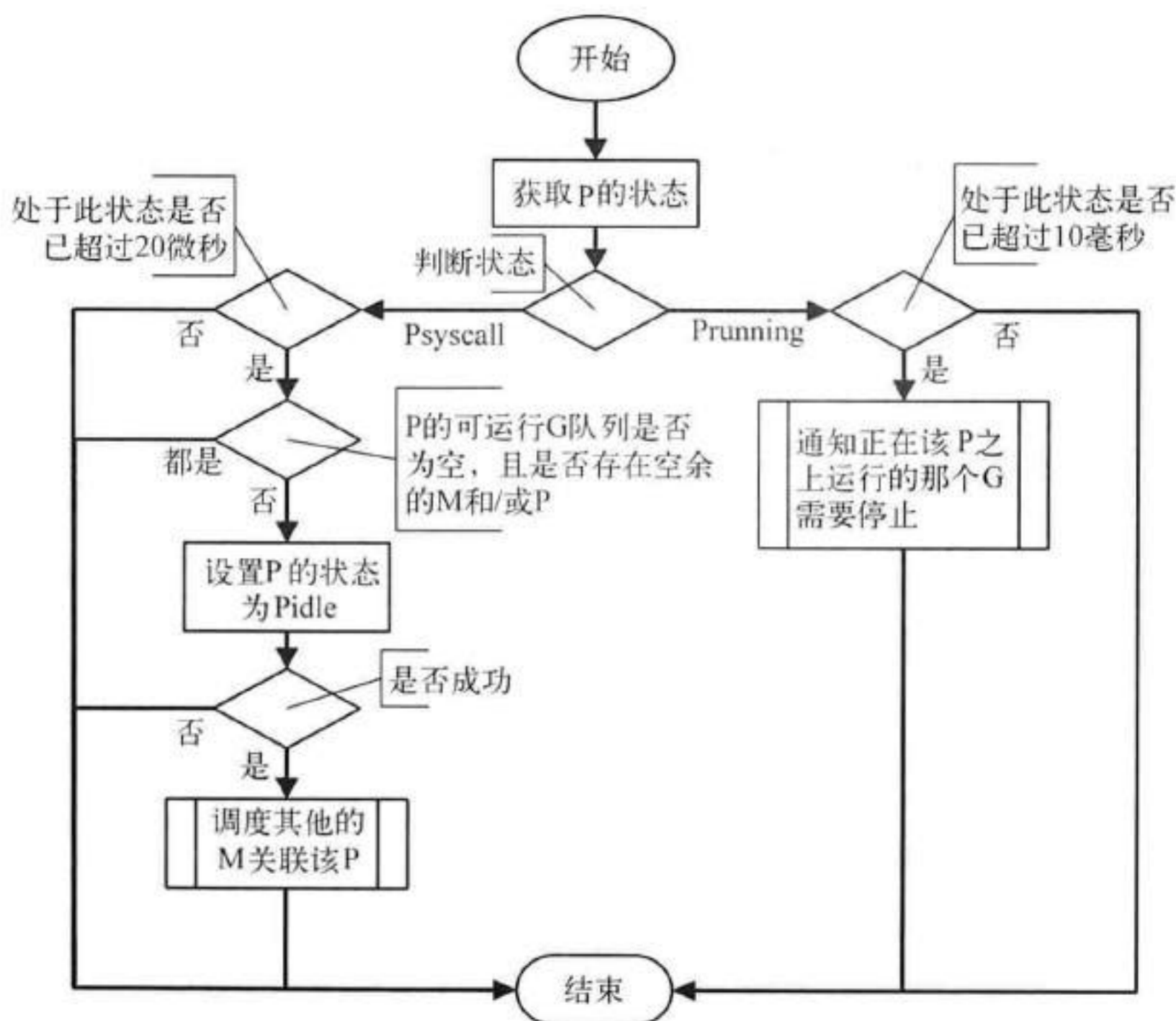


图6-38 抢夺P和G的流程

从图6-38中我们可以看出，当P的状态为Psyscall且处于该状态已超过20微秒的时候，程序会尝试拿走该P并让其他的M与它关联。这是为了让该P的可运行G队列中的G能够尽快被运行。不过，这还需要以其他条件的满足为前提。如果这个P的可运行G队列已经空了，那么让其他M与它

关联也就没有什么意义了。不过，若此时已经没有任何空余的M和P了，那么还是应该对这个P进行调度。这里所说的空余的M是指M未被停止但它还没找到可运行的G，而空余的P即是指空闲的P。那么没有空余的M和P意味着什么呢？实际上，这就意味着现有的未被停止的M和所有的P都已无事可做。这时使用这个处于Psyscall状态的P作为补充是合理的。

另一方面，如果P的状态为Prunning且处于该状态已超过10毫秒，那么程序会尝试停止正在该P代表的上下文环境之上运行的那个G。也就是说，程序会阻止一个G被某个M运行过长的时间。这也是为了公平起见。注意，这里的运行过长的时间指的是，G持续的被M运行（中途既没有进入系统调用也没有被阻塞）超过了10毫秒。不过，即使G被持续的运行了10毫秒，并且程序也向这个G发送了通知，这个G也不一定会停止运行。且不说这个通知不一定能够被正确地传递给这个G，就算这个G及时地得到了这个通知，它也可能会将该通知忽略掉。因此，程序仅会也仅能履行告知义务，而既不保证通知的正确达到也不保证作为目标的G会做出响应。这也是该程序仅能作为辅助调度手段的原因之一。

至此，我们已经全面地了解了系统监测任务中的主要流程和重要细节。此任务既是调度程序的有力补充，也是我们了解调度过程的主要手段。此外，细心的读者可能会发现，此系统监测任务永远不会结束（在流程图中也没有“结束”节点）。它会一直被循环地运行下去。它就像调度器的守护者一样，实时地监测着调度过程。最后，值得一提的是，系统监测任务是在一个单独的M中被运行的。但是，调度器并没有把它封装在G中。

6. 变更P的最大数量

我们在上一小节中其实已经介绍过与此变更操作相关的一些步骤，比如，调整全局P队列的大小、把多余的P的状态置为Pdead，以及重新分配可运行的G给全局P队列中的所有P，等等。我们现在按照实际的顺序把这些操作步骤串接起来，使读者能够看到这一变更操作的全貌。

当我们在Go程序中调用runtime.GOMAXPROCS函数的时候，它会先进行下面两项检查以确保变更的合法和有效。

- 如果我们传入的参数值（以下简称新值）比运行时系统对此设定的硬性上限值（即256）大，那么前者会被后者替代。也就是说，无论我们传入的新值有多大，最终的值也不会超过256。这是运行时系统对自身的保护。
- 如果新值不是正整数或者与存储在运行时系统中的P最大数量值（以下简称旧值）相同，那么该函数就会忽略此变更而直接返回旧值。

如果通过了这两项检查，该函数会先通过调度器停止一切调度工作，然后暂存新值、重启调度工作，最后将旧值作为结果返回。在调度工作真正被重启之前，调度器如果发现有新值被暂存，那么就会进入到P最大数量的变更流程中。

在此变更流程中，旧值也会先被获取。如果发现旧值或新值不合法，那么调度器就会发起一个运行时恐慌，流程也会随即终止。不过由于runtime.GOMAXPROCS函数中的前期检查，此流程中的这个分支在这里永远不会被执行到。在通过对旧值和新值的检查之后，调度器会依据新值对全局P列表进行重新初始化。更确切地说，是对全局P列表中的前N个P进行重新初始化。这里的N即为新值。如果全局P列表中的P的数量不够，调度器则会新建相应数量的P并把它们追加到全局

P列表中。新的P的状态为Pgcstop以表示它还不能被使用。顺带说一下，全局P列表中所有P的可运行G队列的初始长度都会是128。当前的策略是，此长度可以根据实际需要翻倍增长。但是，这种策略可能会在今后被改变。调度器也许会通过对其中的可运行G的适当调度来避免P的可运行G队列的无限增长。这会比当前的策略更安全，也更可控。

在对全局P列表的初始化完成之后，调度器会把全局P列表中的所有P（包括将要丢弃但还未丢弃的P）的可运行G队列中的G全部取出，并依次放入到调度器的可运行G队列中。然后，调度器的可运行G队列中的所有可运行G会被均匀地依次放入到已被重新初始化的那些P的可运行G队列中。至此，所有可运行G的重新分配工作完成。这也是在为丢弃多余的P做准备。对于这些多余的P，调度器会释放它们的本地缓存、将它们的自由G列表中的所有G都转移到调度器的自由G列表中，最后把它们的状态都置为Pdead。之所以不能直接销毁它们，是因为它们可能会被正在进行系统调用的M引用。如果某个P被这样的M引用但却被销毁了，那么就会在该M完成系统调用的时候造成错误。

然后，调度器会把当前M与全局P列表中的第一个P关联（别忘了，调度程序也是在M中被运行的），并把剩余的P全部放入到调度器的空闲P列表中。正如我们前面所讲的，在P与M关联或被放入空闲P列表之前，它的状态都会先被置为Pidle。最后，存储在运行时系统中的P最大数量的值会被变更为新值。

图6-39展示了此变更流程。

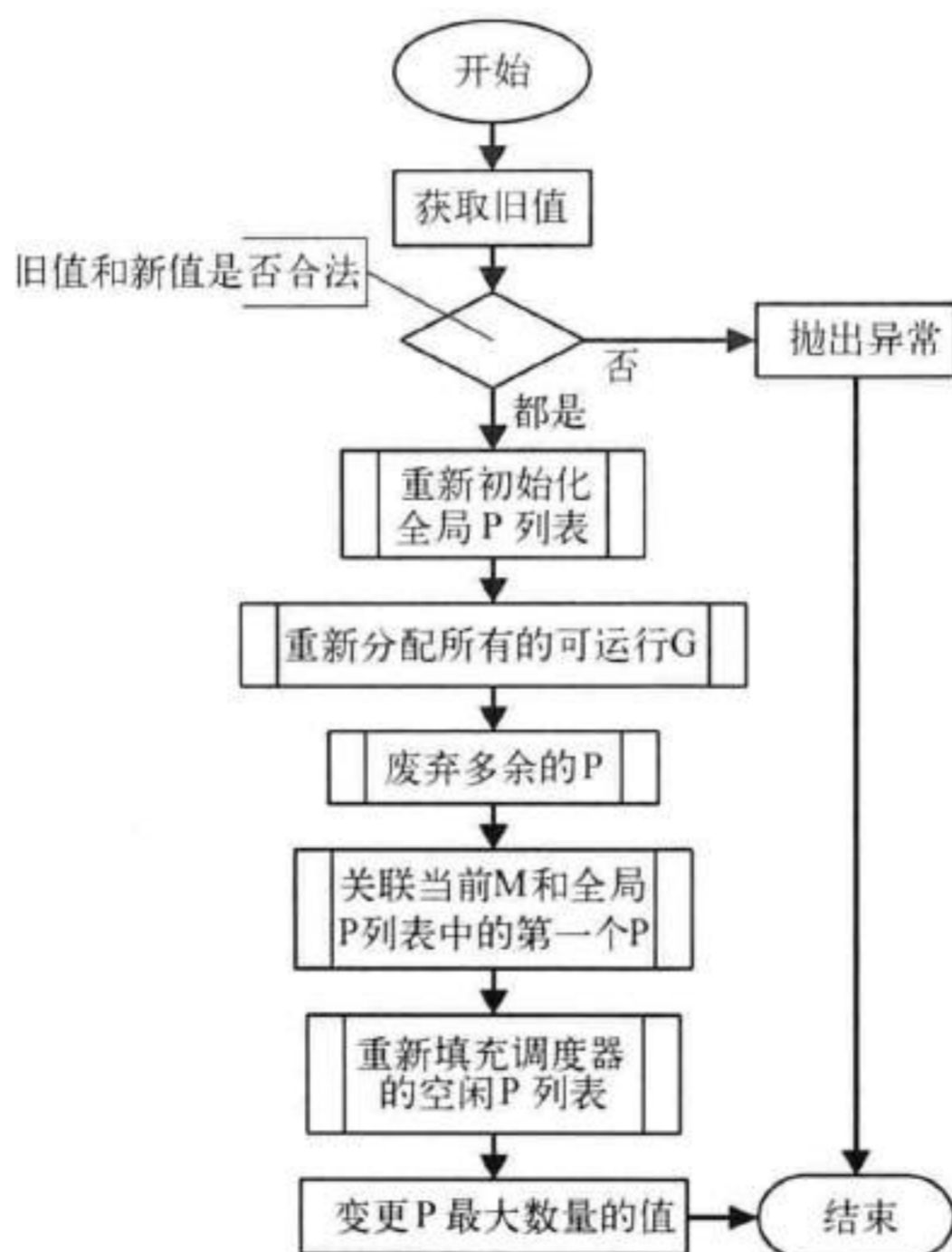


图6-39 P最大数量的变更流程

从图6-39中我们也可以看到，此变更流程内含了3类操作，即重建、废弃和更新。这3类操作都是由调度器来完成的。

在本小节中，我们介绍了调度器中的5个非常关键的流程，即一轮调度、全力查找可运行的G、启用或停止M、系统监测任务和变更P的最大数量。一轮调度流程是调度器中最核心的流程，没有之一。而全力查找可运行的G则是专属于一轮调度流程的子流程。它非常重要，以至于我们为它专门设置了一个标题。启用或停止M的流程也与前两者有着千丝万缕的联系。它让我们了解到了调度器启停M的方法和规律。系统监测任务的本质是为调度器的调度工作查缺补漏以使其对M、P和G的调度更加合理和高效。我们可以通过调用`runtime.GOMAXPROCS`函数改变运行时系统中的P的最大数量。它是我们对Go程序的性能进行调优的最直接的方法之一。不过需要注意，对`runtime.GOMAXPROCS`函数的调用会引起调度工作的短暂时停止。对于对响应时间敏感的Go程序来说，即使是如此短暂的停止，也可能会给程序的性能带来影响。所以，我们需要知道和记住使用此函数的正确方式（在上一小节有介绍）。希望对这些流程的讲解能够使读者对Go语言程序的并发运行机制有更深入的理解。

6.6.3 更多的细节

在本小节，我们会对与调度任务有关的一些细节进行简短的介绍。了解这些细节可以让我们对调度器及其运行过程的认识更加清晰一些。

1. g0和m0

运行时系统中的每个M都会拥有一个特殊的Goroutine——g0。它不是由Go程序中的代码（确切地说是go语句）间接生成的，而是运行时系统在初始化M期间创建并分配给该M的。g0内含了各种调度、垃圾回收和栈管理等程序。

除了g0之外，其他由M运行的G都可以被视作用户级别的G。用户级别的G可以被简称为用户G，而g0则可以被称为系统G。在通常情况下，M会运行用户G。不过，g0也会时不时地被切换和运行以执行前面说到的那些管理性质的任务。这就是我们在前面提到的每个M都会运行调度程序的根本原因。与用户G不同，g0不会被阻塞，也不会被包含在任何G队列或列表中。同时，它的栈也不会垃圾回收进行期间被扫描。由此也可见g0的特殊性。

除了每个M都有属于它自己的g0之外，还存在一个`runtime.g0`。`runtime.g0`被用于执行引导程序。它是在Go程序所间接拥有的第一个内核线程中被运行的。这个内核线程也被称为`runtime.m0`。`runtime.m0`和`runtime.g0`都是被静态分配的，因此引导程序也无需为它们分配内存。

2. 调度器锁和原子操作

其实，在我们本节介绍的很多流程中都用到了调度器锁。但是为了描述的简洁，我把对调度器锁的加锁和解锁操作从流程中去掉了。但是这并不意味着这类操作不重要。

我们已经知道，每个M都有可能执行调度任务。这些任务的执行在时间上可能会重叠，即称并发的调度。因此，调度器会在查询或更改它自己的字段以及运行时系统中的全局变量的之前和之后分别对调度器锁进行加锁和解锁操作。它所涉及的代码非常多，以至于遍及绝大多数的调度

流程。不过，从其源码上看，各个临界区的数量的大小都已被合理地控制。除此之外，调度器在必要时会对它的字段或全局变量进行原子的查询或更改操作，无论这些操作是否已在临界区中。

调度器在自身的并发执行上做了很多有效的约束和控制，兼顾正确性与可伸缩性。这也是我们在上一节讲多线程编程的时候所提倡的。这非常值得我们学习。

3. 调度器跟踪

我们在上一小节说过，系统监测器会在必要的时候打印出调度器跟踪信息。实际上，我们可以通过设置操作系统的一个环境变量来对此进行控制。这个环境变量的名字是GODEBUG。它控制着调试信息的输出。这其中包含了调度器跟踪信息。我们如果想让Go程序在被运行的同时打印出调试跟踪信息，就需要在此之前设置好这个环境变量。

环境变量GODEBUG的值可以由若干个键值对组成。键和值之间需要用等号“=”分隔，而多个键值对之间需要用英文逗号“,”分隔。目前，可以出现在此环境变量中的键有3个，其中两个是与调度器跟踪信息有关的。它们是schedtrace和scheddetail。

当我们设置键schedtrace的值为X的时候，就意味着系统监测器会每X毫秒打印一个单行信息到操作系统的错误输出上。这一行信息中包含了调度器状态的概要。如果我们在有效设置schedtrace（即其值X大于0）的前提下将scheddetail的值设定为1，那么系统监测器就会每X毫秒向操作系统的错误输出上打印一个多行信息。其中包括了调度器以及所有现存的M、P和G的状态。

我们现在来举例说明，操作系统依然为Ubuntu 12.10 32bit。下面是一个极其简单的命令源码文件：

```
package main

import (
    "time"
)

func main() {
    for i := 0; i < 10; i++ {
        go func() {
            time.Sleep(5 * time.Second)
        }()
        time.Sleep(time.Second)
    }
}
```

我们把这个命令源码文件就命名为schedtrace.go。存放它的目录无关紧要。它的目的非常明确，就是要创建出10个Goroutine并让它们并发的运行一段时间。我们在运行它之前，需要先设置好环境变量。若要让系统监测器每2秒打印出一个单行信息，环境变量GODEBUG应该这样被设置：

```
export GODEBUG=schedtrace=2000
```

好了，现在可以运行这个源码文件了。执行的命令及其输出如下：

```
hc@ubt:~$ go build schedtrace.go
hc@ubt:~$ ./schedtrace
SCHED 0ms: gomaxprocs=1 idleprocs=0 threads=4 idlethreads=0 runqueue=1 [0]
SCHED 2010ms: gomaxprocs=1 idleprocs=1 threads=5 idlethreads=2 runqueue=0 [0]
```

```
SCHED 4019ms: gomaxprocs=1 idleprocs=1 threads=5 idlethreads=2 runqueue=0 [0]
SCHED 6026ms: gomaxprocs=1 idleprocs=1 threads=5 idlethreads=2 runqueue=0 [0]
SCHED 8030ms: gomaxprocs=1 idleprocs=1 threads=5 idlethreads=2 runqueue=0 [0]
SCHED 10032ms: gomaxprocs=1 idleprocs=1 threads=5 idlethreads=2 runqueue=0 [0]
```

每一个单行的调度跟踪信息的格式都是相同的。我们以最后一行为例解释一下其中的内容。SCHED表示此行信息是调度器跟踪信息，紧随其后的是10032ms。它说明该行信息是在Go程序运行后10.032秒的时候被生成出来的。在冒号“:”之后的内容即概括的描述了调度器内部的情况。这些内容其实就是以空格“ ”分隔的5个键值对。其中的5个键gomaxprocs、idleprocs、threads、idlethreads和runqueue分别代表了P最大数量（也就是P的总数量）、空闲P的数量、M的总数量、空闲M的数量，以及调度器的可运行G队列中的G的数量。而在单行信息最后的、以方括号“[”和“]”括起来的数字则表示了唯一的那个P的可运行G队列中的G的数量。为什么只有一个P？那是因为默认情况下的P最大数量为1。如果P最大数量大于1，那么在这个方括号中的就会是以空格“ ”分隔的多个数字。数字的数量与P最大数量的值相同。它们出现的顺序则与它们被生成的顺序一致。例如，若P最大数量为3则一个单行的调度跟踪信息会像这样：

```
SCHED 10020ms: gomaxprocs=3 idleprocs=3 threads=6 idlethreads=3 runqueue=0 [0 0 0]
```

请注意，它与前面展示的调度跟踪信息是不同的。如果我们想深究调度器内部的具体运作情况，那么就需要修改一下环境变量GODEBUG的值，像这样：

```
export GODEBUG=schedtrace=2000,scheddetail=1
```

在这之后，当我们再次运行Go程序的时候就会看到比上面多得多的输出内容。我们依然以最后一个调度跟踪信息为例，如下：

```
SCHED 10017ms: gomaxprocs=3 idleprocs=3 threads=6 idlethreads=3 runqueue=0 gcwaiting=0 nmidlelocked=1
nm spinning=0 stopwait=0 sysmonwait=0
P0: status=0 schedtick=10 syscalltick=4 m=-1 runqsize=0/128 gfreecnt=2
P1: status=0 schedtick=12 syscalltick=4 m=-1 runqsize=0/128 gfreecnt=0
P2: status=0 schedtick=6 syscalltick=8 m=-1 runqsize=0/128 gfreecnt=1
M5: p=-1 curg=-1 mallocing=0 throwing=0 gcing=0 locks=0 dying=0 helpgc=0 spinning=0 lockedg=-1
M4: p=-1 curg=-1 mallocing=0 throwing=0 gcing=0 locks=0 dying=0 helpgc=0 spinning=0 lockedg=-1
M3: p=-1 curg=2 mallocing=0 throwing=0 gcing=0 locks=0 dying=0 helpgc=0 spinning=0 lockedg=-1
M2: p=-1 curg=-1 mallocing=0 throwing=0 gcing=0 locks=0 dying=0 helpgc=0 spinning=0 lockedg=-1
M1: p=-1 curg=-1 mallocing=0 throwing=0 gcing=0 locks=1 dying=0 helpgc=0 spinning=0 lockedg=-1
M0: p=-1 curg=4 mallocing=0 throwing=0 gcing=0 locks=0 dying=0 helpgc=0 spinning=0 lockedg=-1
G1: status=4(sleep) m=-1 lockedm=-1
G2: status=3() m=3 lockedm=-1
G10: status=4(sleep) m=-1 lockedm=-1
G4: status=3(stack growth) m=0 lockedm=-1
G11: status=4(sleep) m=-1 lockedm=-1
G6: status=6(sleep) m=-1 lockedm=-1
G7: status=6(sleep) m=-1 lockedm=-1
G8: status=6(sleep) m=-1 lockedm=-1
G9: status=4(sleep) m=-1 lockedm=-1
G12: status=4(sleep) m=-1 lockedm=-1
G13: status=4(sleep) m=-1 lockedm=-1
```

乍一看，如此多的信息让人有些无所下手。但实际上，它们却是非常规整和有规律的。

在以SCHED开始的第一行内容中多出了5个键值对。这些额外的键的含义如下。

- gcwaiting: 表示垃圾回收任务是否正在被准备或执行。0代表否, 1代表是。
- nmidlelocked: 表示已被锁住且空闲的M的数量。
- nm spinning: 表示正在自旋的M的数量。简单来讲, 未被停止且还没有找到可运行G的M都被认为是正在自旋的M。
- stopwait: 表示在垃圾回收准备期间还未被停止的P的数量。
- sysmonwait: 表示系统监测器是否正在被阻塞。0代表否, 1代表是。

可以看到, 其中的大部分键所代表的状态都与垃圾回收任务相关。但实际上, 它们与所有的在被执行期间对调度工作进行停止 (“Stop the world”) 和重启 (“Start the world”) 的任务都是相关的。只不过垃圾回收任务是其中最主要的一个任务。其他任务还有改变P最大数量的任务, 等等。这一点请读者注意。

从第二行开始, 每一行都表示了一个核心元素 (M、P或G) 的内部状态。

以P0、P1和P2开始的内容分别表示了3个P的内部状态。在每一行内容中都包含了6个键值对。它们的含义如下。

- status: 代表了当前P的状态。我们知道, P的状态共用5个。这里用正整数0至4分别代表Pidle、Prunning、Psyscall、Pgstop和Pdead状态。
- schedtick: 表示当前P中的G被运行的次数。
- syscalltick: 表示当前P中的G完成系统调用的次数。
- m: 表示与当前P关联的M的ID。若未关联则值为-1。
- runqsize: 表示当前P的可运行G队列中G的数量及其长度。例如, 0/128代表该队列的长度为128但其中没有G。
- gfreecnt: 表示当前P的自由G队列中G的数量。

以M开始的内容分别代表了现存的6个M的内部状态。每行的键都有10个。它们的含义如下。

- p: 表示与当前M关联的P的ID。若未关联则值为-1。
- curg: 表示正在当前M上运行的G的ID。若没有则值为-1。
- mallocing: 表示是否正在为当前M上运行的程序分配内存。0代表否, 1代表是。
- throwing: 表示在当前M上运行的运行时系统是否抛出了异常。0代表否, 1代表是。另外, -1代表有异常被抛出但不会打印运行时的栈信息。
- gcing: 表示垃圾回收任务是否正在被执行。0代表否, 1代表是。如前文所述, 更广泛地讲, 此值代表了调度工作是否已被停止。
- locks: 表示在当前M上运行的运行时系统持有的 (象征性的) 锁的数量, 被用于一些内部的关键任务的执行计数和协调。
- dying: 表示在当前M上运行的程序是否已经引发了运行时恐慌。0代表否, 1代表是。
- helpgc: 表示当前M是否需要执行垃圾回收任务。0代表不需要, 否则代表需要。其中, -1代表当前M是调度器专为进行垃圾回收而创建的, 而大于0的整数则表示当前M即将或已经开始执行垃圾回收任务且此整数即为专用序号 (并发执行此任务的M可能有多个)。

□ spinning: 表示当前M是否正在自旋。

□ lockedg: 与当前M锁定的G的ID。若没有则值为-1。

最后, 以G开始的内容则分别代表了现存的某个G的内部状态。其中的3个键的含义如下。

□ status: 代表了当前G的状态。这里用从0至6的正整数表示G的7种状态, 即Gidle、Grunnable、Grunning、Gsyscall、Gwaiting、Gmoribund_unused和Gdead。其中, Gmoribund_unused状态至今未被使用, 所以我们在前面也没有对它进行介绍。另外, 紧随在状态之后的圆括号“(”和“)”会包含该G处于此状态的原因。该原因会以一个短语的方式呈现。

□ m: 表示正在运行当前G的M的ID。若没有则值为-1。

□ lockedm: 表示与当前G锁定的M的ID。若没有则值为-1。

至此, 我们已经对调度跟踪信息中的所有键都做了必要的说明。此后, 我们就可以轻松地看懂这些内容了。

当我们想了解Go程序中的各个Goroutine的运作情况的时候, 就可以设置环境变量GODEBUG、运行Go程序并查看其输出的调度跟踪信息。这些信息可以让我们清楚地看到Go语言的调度器的实时调度操作。通过对这些信息的分析, 我们就能够发现Go程序本身以及其运行过程中的一些问题。另外, 让Go程序在被运行的同时打印调度跟踪信息, 只需要设置一下环境变量, 而并不需要对程序本身进行任何修改。这样的好处是显而易见的。

在本小节, 我们对Go语言的并发编程模型及其调度器的一些细节进行了进一步的介绍。显然, 在编程过程中, 对我们最有用处的应该是对调度跟踪信息的解读。不过, 对m0和g0的简短说明也让我们更深入地了解了Go语言的运行时系统引导Go程序运行以及执行一些管理任务(包括调度、垃圾回收, 等等)的方式方法。由于运行时系统一般会使用多个M并发的执行这些任务, 所以它自己也会用到各种操作系统提供的同步方法。我们在前面也已对此进行了强调。

6.7 本章小结

本章为大家讲述了并发编程的基础知识以及一些主流的并发编程模型和编程方法。希望读者在阅读这些内容之后能够对并发编程有所了解并明白其中的一些重要逻辑。此外, 在本章最后的Go语言并发编程模型方面的内容需要大家认真理解和思考。我想这些内容对每一个有意深入学习Go语言编程的人都是非常有用的。并且, 它们也是后面所涉及内容的引导和必备知识。

在上一章，我们详细地探讨了主流的并发编程方式以及Go语言特有的并发编程模型。接下来，我们会开始着手使用Go语言编写并发程序。这会用到我们已经提到过很多次的Goroutine，以及Go语言特有数据类型Channel（也就是我们前文提及的通道类型）。本章会专门对它们的使用方法和技巧进行讲解。

7.1 Goroutine 的使用

上一章已经对Go语言的并发编程模型中与我们最接近的核心元素G（即Goroutine）进行了详细的讲解。其中包括了G在模型中的位置和作用、生命周期、状态转换和调度方法等内部运作细节。我们说过，读者可以跳过这些内容而直接阅读本章。所以，如果你在阅读本节的过程中需要了解一些细节，那么上一章的最后一节一定是你最应该查阅的。

说到Goroutine，就不得不提到Go语言特有的关键字go。它是我们启用Goroutine的唯一途径。接下来，我们就开始学习使用go关键字并编写go语句。

7.1.1 go语句与Goroutine

一条go语句意味着一个函数或方法的并发执行。go语句是由go关键字和表达式组成的。对表达式的详细讲解请参见第3章。简单来说，表达式就是用于描述针对若干操作数的计算方法的式子。Go语言的表达式有很多种，其中包括了调用表达式。调用表达式所表达的即是对函数或方法的调用。其中，函数可以是命名的，也可以是匿名的。可以明确地讲，能够被称为表达式语句的调用表达式是我们创建go语句时唯一可以合法使用的表达式。还记得吗？针对如下函数的调用表达式不能被称为表达式语句：`append`、`cap`、`complex`、`imag`、`len`、`make`、`new`、`real`、`unsafe.Alignof`、`unsafe.Offsetof`和`unsafe.Sizeof`。在这11个函数中，前8个函数是Go语言的内建函数，而最后3个函数则是标准库代码包`unsafe`中的函数。

可见，go语句的编写规则并不复杂。它与defer语句的编写规则有很多相似之处。接下来，我们要真正地编写几条go语句。

我们使用go关键字和一个针对内建函数`println`的调用表达式组成了一条go语句：

```
go println("Go! Goroutine!")
```

22,如果在go关键字后面的是针对匿名函数的调用表达式,那么go语句就会像这样:

```
go func() {
    println("Go! Goroutine!")
}()
```

注意,无论是否需要传递参数值给匿名函数,我们都不要忘了最后的那对圆括号。它们代表了对函数的调用行为。否则,这就成了一个函数字面量,而不是go语句需要的调用表达式了。另外还有一点需要注意,在go关键字后面的调用表达式是不能被圆括号括起来的。这些都与defer语句的构建规则相同。

Go语言对go语句中的函数或方法及其参数的求值顺序并没有任何特别之处。但是,反直觉的是对它们的执行方式。Go语言的运行时系统对go语句中的函数或方法(以下简称go函数)的执行是并发的。更确切地说,当go语句被执行的时候,其中的go函数会被单独地放入到一个Goroutine中。在这之后,该go函数的执行会独立于当前Goroutine的运行。在一般情况下,在当前Goroutine中的、在某条go语句后面的那些语句并不会等到相应的go函数被执行完成之后才被执行。甚至,在该go函数真正被执行之前,运行时系统往往就已经开始执行后面的语句了。

另一方面,当go函数被执行完毕的时候,相应的Goroutine也会暂时进入到死亡状态(Gdead)。这标志着该Goroutine的一次运行的完成。此外,作为go函数的函数或方法是可以有结果声明的。但是,它们返回的结果值会在它们被执行完成的时候被丢弃。也就是说,即使它们返回了结果值也是没有任何意义的。这些结果值不会被传送到任何地方。那么,如果我们想把go函数中的结果值或者其他值传递给其他程序(或者说在其他Goroutine中的程序)的话,应该怎样去做呢?不要着急,我们在讲Channel的时候就会揭晓这个答案。

我们现在已经基本知晓了Go语言的运行时系统对go函数的执行方式。下面来看几个例子。假设有这样一个命令源码文件:

```
package main

func main() {
    go println("Go! Goroutine!")
}
```

当我们使用go命令去运行这个源码文件的时候,标准输出上会出现什么内容呢?读者可能会认为该程序输出的内容会是:

```
Go! Goroutine!
```

但是,实际上,这行内容并不会出现。这是为什么呢?我们刚刚说过,运行时系统会并发地执行go函数。正因为如此,运行时系统在使用一个Goroutine封装go函数并把它放入到相应的队列中之后,会立即继续执行在相应的go语句后面的语句。至于这个新的可运行G什么会被运行,就要看调度器的实时调度情况了(请见上一章中的解释)。而在本例中,go语句之后没有任何语句。因此,main函数此时即被执行完毕。这也意味着该Go程序的运行的结束。可是,这个时候,main函数中的那个go函数还没来得及被执行。换句话说,封装这个go函数的那个Goroutine还没有来得及被调度并运行。这种情况几乎总是会发生。所以我们不要对这种并发执行的先后顺序有任何假

设,也不要指望main函数所在的G总是最后一个被运行完毕。如果我们确实希望如此,就必须通过额外的手段去实现。

Go语言为我们提供了很多这里所说的额外手段。其中,最简陋的一个手段是使用time包中的Sleep函数,像这样:

```
package main

import (
    "time"
)

func main() {
    go println("Go! Goroutine!")
    time.Sleep(time.Millisecond)
}
```

函数time.Sleep的作用是让调用它的Goroutine暂停(进入Gwaiting状态)一段时间。在这里,我们让main函数所在的Goroutine暂停了1毫秒。在理想的情况下,运行该源码文件会如我们所愿地在标准输出上打印出Go! Goroutine!。但是,请注意,情况并不总是这样的。调度器的实时调度是我们无法控制的,所以上例所示的这个手段是非常不保险的。我们不应该在这种情形下使用time.Sleep函数。在这里,用调用语句runtime.Gosched()替换对time.Sleep函数的调用是一种更保险的方式。我在之前说过, runtime.Gosched函数的作用是让其他Goroutine有机会被运行。这种手段在这里施展是再适合不过的。但是,实际的情况往往要比这复杂得多。那时, runtime.Gosched函数就会变得不适用。当然,我们依然可以实现我们的需求。至于实现的具体方式,我会在后面披露。

下面我们来看更复杂一些的例子,如下:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    name := "Eric"
    go func() {
        fmt.Printf("Hello, %s.\n", name)
    }()
    name = "Harry"
    runtime.Gosched()
}
```

请读者试想一下,在我们运行内容如上的命令源码文件之后,标准输出上打印出怎样的内容? 是Hello, Eric.还是Hello, Harry.? 通过多次试验得知,答案是后者。这进一步说明了这种执行的并发性。在赋值语句name = "Harry"被执行之后,它上面的go函数才得以执行。更有甚者,如果我们在这里不在最后加入表达式语句runtime.Gosched(),那么go函数根本就没有被执行的机

会。我们在前面已经说明了造成这种情况的原因。现在，我们把main函数中的最后两条语句互换一下位置，像这样：

```
runtime.Gosched()
name = "Harry"
```

那么，标准输出上会打印出内容吗？或者说，打印出的内容会与之前有什么不同？由runtime.Gosched的作用可知，打印出的内容会是：

```
Hello, Eric.
```

因为我们在改变变量name的值之前就给那个go函数被执行的机会了。或者说，Go语言运行时系统应我们的要求抢先执行了该go函数。

现在情况变得更复杂了。我们要同时向多个人问候。问候目标的名单如下：

```
names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
```

要同时问候这5个人，最简单的方式就是连续编写出5条go语句。不过，这样好像太繁琐了，会写出很多冗余代码。既然我们把名单作为一个切片类型值呈现，那么我们用for语句来实现同时的（或者说并发的）问候应该会更好。代码如下（我们仅仅改造了一下前面示例中的main函数）：

```
func main() {
    names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
    for _, name := range names {
        go func() {
            fmt.Printf("Hello, %s.\n", name)
        }()
    }
    runtime.Gosched()
}
```

请读者运行一下这个源码文件。标准输出上的新增内容可能会让你感到诧异。它是这样的：

```
Hello, Mark.
Hello, Mark.
Hello, Mark.
Hello, Mark.
Hello, Mark.
```

我们的朋友Mark可能要应接不暇了。这到底是怎么回事？要弄清楚这个问题，我们首先要知道go函数中所使用的标识符name到底代表了什么。在运用本书第4章中讲到的知识对此进行分析之后可知，这个标识符name其实就是在该go语句外层的for语句中声明的那个迭代变量name。这会有什么问题的吗？这里的细节是：随着迭代的进行，每一次被获取出的迭代值（这里是名单中的单个名字）都会被赋给相应的迭代变量（这里是name）。也就是说，迭代变量name会依次被赋予"Eric"、"Harry"、"Robert"、"Jim"和"Mark"这5个值。注意，"Mark"是最后一个被赋给变量name的值。隐约感觉出问题所在了吗？事实上，在这里被并发执行的5个go函数（确切地讲，是被5个Goroutine分别封装的同一个函数）中，name的值都是"Mark"。这是因为它们都是在for语句被执行完毕之后才被执行的，而name在那时指代的值已经是"Mark"了。这也有for语句非常简单、瞬间就可以被执行为完成的原因在里面。不过，即使for语句很复杂，这种情况也有可能发生。还是那句话，不要对go

函数的执行时机做任何假设，除非你确实能做出让这种假设成为绝对事实的保证。

现在，我们来考虑一下解决上面问题的方案。有两种思路。第一种思路是，让5个go函数在每次迭代完成之前被执行完毕。按照这种思路，我们使用go语句发出问候有一点画蛇添足了。因为顺序的执行这些代码就可以达到目的，而且会简单得多。不过，这样实在就算不上是同时问候了，即使for语句的每次迭代都会以极快的速度完成。那么应该怎么办呢？其实很简单，我们在每次迭代完成之前给予之前的go函数一个被执行的机会就可以了。我们把上面的for修改成下面这样：

```
for _, name := range names {
    go func() {
        fmt.Printf("Hello, %s.\n", name)
    }()
    runtime.Gosched()
}
```

使用这一方案解决本例中的这个问题是简单而有效的。但是，如果我们的go函数比较复杂，并且在那条打印语句之前还有很多其他语句，那么这个方案就不一定会带来正确的结果。为了看清此问题，我们需要对go函数稍加修改，如下：

```
go func() {
    time.Sleep(10 * time.Nanosecond)
    fmt.Printf("Hello, %s.\n", name)
}()
```

为了不喧宾夺主，我们只使用针对time.Sleep函数的调用来代表执行若干条语句所需的时间。假设这些语句的执行总共需要耗费10纳秒。这已经是一个非常短暂的时间了。可是，这依然会使我们的需求无法实现。在我的计算机上运行这个源码文件后，不会打印出任何内容。也许，在你的计算机上可以打印出内容，但也绝对不是正确的结果。因为10纳秒足以让如此简单的for循环完成若干次迭代了。如此一来，我们可能会少问候一个或几个人，而另外的人可能会被问候几次。显然，这样的解决方案并不总是可行的。它会受到go函数以及for语句的执行时间的影响。

下面我们来考量第二种思路。如果我们在go函数中使用的name的值不会随外部变量的变化的影响，那么就可以既保证go函数的独立执行，又不用担心它们的正确性受到破坏。显然，如果这样的设想被实现了，那么这里的go函数就可以被称作可重入函数。

我们已经知道，go函数可以有结果声明（虽然这没有任何意义）。但是却还没有提到，go函数也和普通的函数一样可以有参数声明。如果把迭代变量name的值作为参数传递给go函数，那么也就实现了我们上面的设想。

能够如此轻易地实现该设想的根本原因是，name变量的类型string是一个非引用类型。我们在把一个值作为参数传递给函数或方法的时候，该值会被复制。对于引用类型（比如切片类型和字典类型）的值来说，由于它类似于指向真正数据的指针，所以即使它被复制了，之后在外部对该值的修改也会被反映到该函数或方法的内部。而对于非引用类型的值来说，这种修改就不会对函数体内部的操作产生影响。因为这样的两个值已经被完全分离了。

言归正传，实现此设想的main函数会是这样：

```
func main() {
    names := []string{"Eric", "Harry", "Robert", "Jim", "Mark"}
    for _, name := range names {
        go func(who string) {
            fmt.Printf("Hello, %s.\n", who)
        }(name)
    }
    runtime.Gosched()
}
```

请注意，我们为go函数添加了一个参数声明。该参数的名称为who。相应地，我们在go函数中不再使用外部变量name，而仅仅使用参数who。因为有了这样一个参数声明，所以我们在编写对它的调用表达式的时候，就需要在最后的圆括号“(”和“)”中放入参数值。在这里，我们把变量name的值作为参数值传递给go函数。在我们分析这条for语句的迭代之后会发现，在每次迭代的起始，name变量都会被赋予names的某一个元素值，紧接着这个元素值会被传入go函数。在传入的过程中，该值会被复制并在go函数中由参数who指代。此后，name的值的改变与go函数完全无关。我们运行包含此main函数的源码文件之后总会得到正确的结果。由此，根据第二种思路产出的解决方案是完全可行并且总是正确的。

再次强调，无论哪一种解决方案，它们最多只能保证go函数执行的正确性，而无法保证这些go函数总会先于main函数被执行完成。后者相当于保证多个Goroutine的执行顺序，属于同步的范畴。

通过这一系列的示例，我们已经对go语句的使用方法和技巧有了足够的了解。作为一个理论补充，我们将会在下小节简述封装main函数的Goroutine从“诞生”到“死亡”的全过程。

7.1.2 Goroutine的运作过程

在上一章，我们已经讲述了很多与Go语言的并发编程模型、运行时系统和调度器有关的知识。其中，我们详细地解释了Goroutine的状态以及它在这些状态之间的转换规则和时机。同时，我们也从调度器的角度说明了一个Goroutine是怎样被调度和运行的。如果读者还没有看过这些内容，我强烈建议读者在阅读本小节的内容之前先去了解一下它们。

我们说过，封装main函数的Goroutine是Go语言运行时系统创建的第一个Goroutine（也可被称为主Goroutine）。主Goroutine是在runtime.m0上被运行的。我们在上一章中讲过，封装了引导程序的runtime.g0就是在runtime.m0被运行的。实际上，在runtime.m0在运行完runtime.g0中的引导程序之后，会接着运行主Goroutine。

主Goroutine所做的事情并不是执行main函数那么简单。它首先要做的，是设定每一个Goroutine所能申请的栈空间的最大尺寸。在32位的计算机系统下，这个最大尺寸为250MB，而在64位的计算机系统中，此尺寸为1GB。如果有某个Goroutine申请的栈空间总尺寸大于了这个限制，那么运行时系统就会发起一个“栈溢出”（stack overflow）的运行时恐慌。这在正在运行的Go程序上的表现就是发生一个运行时恐慌。随即，该Go程序的运行也会被终止。

在设定好Goroutine的最大栈尺寸之后，主Goroutine会启动系统监测器。我们已经知道，系统监测器的作用就是对调度器的工作进行查缺补漏。这也是让系统监测器的启动先于main函数的执行的原因之一。

此后，主Goroutine会进行一系列的初始化工作。由于这些工作的重要性和特殊性，主Goroutine会在此期间与当前M（即runtime.m0）锁定在一起。这里所涉及的工作内容如下。

- ❑ 创建一个特殊的defer语句，以执行主Goroutine退出时必要的善后处理。实际上，这里的善后处理即是指主Goroutine与当前M的解锁操作。因为，主Goroutine也可能会非正常地结束，所以这一点很有必要。
 - ❑ 检查当前M是否是runtime.m0。如果不是，那么就说明之前的程序出现了某种问题。这时，主Goroutine会立即抛出异常。这也意味着Go程序启动的失败。
 - ❑ 创建定时垃圾回收器（scavenger）。主Goroutine会创建一个专门的Goroutine来封装这个定时垃圾回收器，并把它放入到当前M的可运行G队列中。注意，此Goroutine即是Go语言运行时系统创建的第二个Goroutine。不过，创建它的方式与我们使用go语句创建一个用户级别的Goroutine的方式几乎无二。顺便提一下，定时垃圾回收器会定时（当前是2分钟一次）的执行垃圾回收任务，并在必要时促使一些M协助它进行一些垃圾回收工作。
 - ❑ 执行main包的init函数。我们在第2章讲过，每个代码包都可以有若干个代码包初始化函数。这些代码包初始化函数都必须是无任何参数声明和结果声明且名称为init的函数。当然，对于main包来说也是如此。在一个可运行的Go程序中，只可能有一个被用于启动Go程序的、属于main包的命令源码文件。因此，main包的init函数与main函数一样，指的是存在于这个源码文件中的同名函数。
 - ❑ 对之前创建的那个特殊的defer语句进行最后的检查和设置，并在必要时抛出异常。
- 如果上述初始化工作成功完成，那么主Goroutine就会去执行main函数。在执行完main函数之后，它还会检查是否有Goroutine发生了运行时恐慌，并进行必要的处理。最后，主Goroutine会结束自己以及当前进程的运行。

以上就是主Goroutine从始至终的运行过程。在main函数被执行期间，运行时系统会根据我们编写的go语句复用或新建Goroutine来封装go函数。这些Goroutine都会被放入到相应的P的可运行G队列中，然后等待调度器的调度。这样的等待时间通常会很短暂。但是有时如此短的时间也是不容忽视的。就像我们在上一小节举例说明的那样，它可能会使Goroutine错过甚至永远失去运行时机。

7.1.3 runtime包与Goroutine

我们已经知道，Go语言的标准库中有一个名为runtime的代码包。其中的程序实体提供了各种可以使应用程序与Go语言运行时系统进行交互的功能。我们在前面的章节中已经提及过很多这样的API。在本小节，我们主要说明那些可以获取到Goroutine信息或者能够直接或间接地控制Goroutine的运行的API。为了汇总它们，我们也会把一些已经讲过的函数罗列在这里。不过，对于这样的函数，我们只会进行概括性的描述。

1. runtime.GOMAXPROCS函数

通过调用runtime.GOMAXPROCS函数,应用程序可以在运行期间设置运行时系统中的P的最大数量。但由于这样做会引起“Stop the world”,所以我强烈建议应用程序应该尽量早地(在main函数的开始处,甚至在main包的init函数中)调用它。并且,请记住,最好的设置P最大数量的方式是在运行Go程序之前设置好操作系统的环境变量GOMAXPROCS,而不是在程序中调用runtime.GOMAXPROCS函数。

最后,请记住,无论我们传递给该函数怎样的整数值,运行时系统中的P最大数量总会在1~256的范围内。

2. runtime.Goexit函数

函数runtime.Goexit被调用之后会立即使调用它的Goroutine的运行被终止,但其他Goroutine并不会受此影响。runtime.Goexit函数在终止调用它的Goroutine的运行之前会先执行该Goroutine中所有还未被执行的defer语句。我们知道,defer语句的执行会被延迟至它所在的函数或方法被执行完毕之前。所以runtime.Goexit函数中的这一善后处理非常重要。

该函数会把被终止运行的Goroutine置于Gdead状态,并将其放入调度器的自由G列表。这样,调度器可以在有需要时重新启用此Goroutine。最后,应用程序对runtime.Goexit函数的调用还会触发调度器的一轮调度流程。

3. runtime.Gosched函数

我们在前面的示例中多次用到了runtime.Gosched函数。该函数的作用是暂停调用它的Goroutine的运行。调用它的Goroutine会被重新置于Grunnable状态,并被放入到调度器的可运行G队列中。这也是使用“暂停”这个词的原因。因为经过调度器的调度,该Goroutine不久就会再次被运行。这样做完全是为了让其他Goroutine立即有被运行的机会。

4. runtime.NumGoroutine函数

函数runtime.NumGoroutine在被调用后会返回运行时系统中的处于特定状态的Goroutine的数量。这里的特定状态是指Grunnable、Grunning、Gsyscall和Gwaiting。处于这些状态的Goroutine即被看作是活跃的或者说正在被调度的。注意,如果定时垃圾回收器所在的Goroutine的状态也在此范围内的话,那么也会被纳入到该计数当中。

5. runtime.LockOSThread函数和runtime.UnlockOSThread函数

我们在上一章说明过这两个函数的功能。前者使调用它的Goroutine与当前运行它的M锁定在一起,而后者则会解除这样的锁定。多次调用前者不会造成任何问题,但是只有最后一次调用的效果会被保留下来。即使在之前没有调用过前者,对后者的调用也不会产生任何副作用。对后者的多次调用也会是这样。

6. runtime/debug.SetMaxStack函数

函数runtime/debug.SetMaxStack的功能是约束单个Goroutine所能申请的栈空间的最大尺寸。我们已经知道,在main函数以及main包的init函数真正被执行之前,主Goroutine会对此进行默认的设置。默认的设定值对于绝大多数程序来说是适合的。所以,确实需要调用该函数的场景极少。

该函数接收一个int类型的参数。该参数的含义是欲设定的栈空间最大字节数。它在被执行

完毕的时候会把之前的设定值作为结果返回。这有利于我们对该项设置的记录和恢复。

在对该函数的调用完成之后,如果运行时系统在为某个Goroutine增加栈空间的时候发现其栈空间所占用的总字节数已经超过了相关的设定值,那么就会发起一个运行时恐慌并终止程序的运行。

可以看出,该函数的作用主要是预防因执行了某些有问题的代码(比如无限的递归)而导致的栈空间的无限增长。不过,需要注意的是,该函数并不会像runtime.GOMAXPROCS函数那样对传入的参数值进行检查和纠正。所以,我们应该在调用它的时候保持足够的警惕。尤其是,即使我们设定了一个过小的值,相关的问题也一般不会对程序的运行初期就显现出来。因为运行时系统仅会在增长Goroutine的栈空间的时候才会对它占用的总字节数进行检查。这样,错误设置就像给程序埋下了一个定时炸弹。其造成的后果想必也是无法忽略的。

7. runtime/debug.SetMaxThreads函数

函数runtime/debug.SetMaxThreads的作用是对Go语言的运行时系统所使用的内核线程的数量(更确切地说,是M的数量)进行设置。在引导程序中,该数量被设置为了10000。这对于操作系统和Go程序来说都已经是一个足够大的值了。

该函数接受一个int类型的值,也会返回一个int类型的值。前者代表欲设定的新值,而后者则代表之前设定的旧值。我在前面说过,如果调用此函数时给定的新值比运行时系统当前正在使用的M的数量还要小的话,该调用就会引发一个运行时恐慌。另一方面,在对此函数的调用完成之后,我们设定的新值就会立即发挥作用。每当运行时系统新建一个M(即向操作系统索取一个新的内核线程)的时候,就会检查它当前所持的M的数量。如果该数量大于了M最大数量的现有设定,那么运行时系统就会发起一个同样的运行时恐慌,并终止Go程序的运行。

如果在运行时系统需要一个M去运行G的时候却发现现有的M都正在忙碌(可能它们正在进行系统调用、cgo调用或正在等待与其锁定在一起的那个G),那么它就会新建一个M以满足使用需要。虽然有些繁琐,但我们确实可以在一定程度上对这种需要进行不精确的预测。此后,我们就可以通过调用runtime/debug.SetMaxThreads函数来限制M的实际数量,以确保操作系统不会因Go程序对内核线程的无节制使用而被拖垮。换句话说,此设置可以让Go程序在计算机因此宕机之前被终止。当然,如果我们能够确定M的数量会在一个合理的范围内的话,不进行此设置也是完全可以的。

8. 与垃圾回收相关的一些函数

由于运行时系统在进行垃圾回收的时候会促使所有调度工作停止,所以说我们对垃圾回收的控制也会间接地影响到Goroutine的运行。

确切地讲,垃圾回收任务包括了两项工作,即垃圾收集和垃圾清扫。前者发现垃圾并记录它们的位置,后者清除垃圾并把它们所占用的内存归还给操作系统。在这之中,只有垃圾收集工作会导致“Stop the world”,并可能会征调一些M来并发的执行辅助任务。而垃圾清扫工作则每次仅会由某一个M来运行,并且不会对调度工作产生任何明显的影响。

下面,我们就介绍几个可以发起或控制这两项工作的函数。

❑ 函数runtime.GC会让运行时系统进行一次强制性的垃圾收集。所谓的强制性的垃圾收集就

是不论怎样都要进行一次垃圾收集操作。相对应地，非强制性的垃圾收集只会在一定的条件下才进行垃圾收集操作。更具体地说，这个条件是运行时系统自上次垃圾收集之后新申请的堆内存的单元（也被称为堆内存单元增量）达到指定的数值。这个数量是可以由应用程序控制的。这需要用到的`runtime/debug.SetGCPercent`函数。

- 函数`runtime/debug.SetGCPercent`被用于设置一个比率（以下称垃圾收集比率）。前面所说的指定的堆内存单元增量与前一次垃圾收集时的堆内存的单元数量和此垃圾收集比率有关，具体计算公式如下：

$$\text{<触发垃圾收集的堆内存单元增量>} = \text{<上一次垃圾收集完成后的堆内存单元数量>} * (\text{<垃圾收集比率>} / 100)$$

可以看到，垃圾收集比率就是应该触发垃圾收集的堆内存单元增量相对于之前的堆内存单元总数量的一个百分比。我们还可以把通过此公式计算出的数值简称为增量下限值。只有在堆内存的单元增量达到了这个增量下限值的情况下，运行时系统发起的非强制性的垃圾收集才不会被忽略。增量下限值会在每次垃圾收集完成之后被重新计算。也就是说，我们在Go程序被运行期间通过调用`runtime/debug.SetGCPercent`函数对垃圾收集比率的修改是可以影响到之后的所有垃圾收集任务的执行的。

我们在调用`runtime/debug.SetGCPercent`函数之后会得到一个结果值。这个结果值即是之前的垃圾收集比率。如果我们没有对此比率进行过任何显式的设置，那么这个结果值就会等于预设值。运行时系统内部对此比率的预设值是100。该预设值是可以被改变的。Go语言为我们提供的改变此预设值的唯一途径是设置操作系统的环境变量`GOGC`。如果我们在Go程序被运行之前（更确切地讲，是在第一次垃圾收集被发起之前）对此环境变量进行了设置，那么上述的预设值就会是我们指定的值。`GOGC`的有效值是什么整数和字符串`off`。这里有两点需要注意。第一点，如果我们为`GOGC`设置了一个无效的值，那么垃圾收集比率的预设值就会被设定为0。这意味着在默认情况下，非强制性的垃圾收集总会被进行。第二点，如果我们把`GOGC`的值设置为了负整数或`off`，那么就会导致垃圾回收器忽略一切垃圾收集操作。这与我们调用`runtime/debug.SetGCPercent`函数并传入一个负整数的效果是一样的。另一方面，如果该函数的结果值是一个负整数，那么就说明垃圾收集操作在此前的一段时间内是被忽略的。长时间的忽略垃圾收集操作对于在非测试环境中运行的程序来说是非常危险的。我们应该特别注意这一点。

有些时候，垃圾收集操作也会被自动地忽略。例如，引导程序还未被执行完成的时候。又例如，运行时恐慌正在爆发的时候。不过，这些情景都是合理且短暂的，并不会对垃圾回收任务的执行产生明显的影响。

上述两个函数都是与垃圾收集工作有关的。它们并不会发起或影响到垃圾清扫工作。在默认情况下，垃圾清扫工作会由垃圾回收器定时地进行。但是，我们可以通过调用`runtime/debug.FreeOSMemory`函数手动地进行一次垃圾清扫。当然，该函数在清扫垃圾之前会先试图把垃圾都收集起来。此处的垃圾收集操作相当于我们对`runtime.GC`函数进行了一次调用。前文所说的外部设置也会影响到这一操作的有效性。而之后的垃圾清扫操作的意义仅在于，尽最大可能地将程序已

经不用的内存归还给操作系统。

本小节讲到的这些API都可以让我们在一定程度上了解、微调和变更Go语言的运行时系统的行为。通过它们，我们可以根据实际情况对Go程序的运行环境进行调整和优化。对于本小节讲到的所有设置，Go语言的运行时系统都给予了默认值。因此，我们应该仅在程序性能不能满足需要的时候再去考虑调整它们。这里的唯一例外是对P最大数量的设置，因为它的默认值是1。这会影响Go程序在多核CPU或多CPU的计算环境下的运行性能。我们应该根据计算环境中逻辑CPU的数量（也就是所有CPU的核数的总和）来设置它。不过要注意，过多的P会对调度器的效率产生负面影响。

7.1.4 Happens Before

在本节的最后，我们来说说Go语言中的“happens before”原则。该原则为多Goroutine程序运行的正确性提供了可以依照的准则。更具体地讲，“happens before”原则描述了使（针对全局变量的）读写操作的结果在全局（即程序中的所有Goroutine）可见的充分条件。

这里有两个基本的描述方法需要了解。

- 如果事件1先于事件2发生，那么就可以说事件2后于事件1发生。
- 如果事件1未先于且未后于事件2发生，那么就可以说事件1和事件2是同时发生的。

对于只有一个Goroutine（即仅有主Goroutine）的程序来说，“happens before”原则并没有什么特别之处。因为其中任何的读写操作的实际执行顺序都一定不会改变程序原本的意图。即使编译器或运行时系统为了优化程序性能而对其进行了代码重排，也同样会保持这一点。但是，对于拥有多个Goroutine的程序来说就完全不同了。因为所有的Goroutine都是被并发地运行的，所以这些在共享内存上的操作可能会使不同的Goroutine对它们的感知有所不同（比如观察到了不同的操作执行顺序）。从而导致某个或某些Goroutine对另一个Goroutine的意图或行为的错误理解。最终，程序在被运行的过程当中就可能会发生不可预知且排查困难的异常。我们在讲解多进程和多线程编程的时候也讨论过类似的问题。

然而，如果程序可以满足“happens before”原则中的那些充分条件的话，那么这些问题就不会发生。Go语言的运行时系统会对此做出保证。下面我们就来看看“happens before”原则中都有哪些内容。

首先，如果要想一个对变量v的写操作w所产生的结果能够被对该变量的读操作r观察到，那么需要同时满足如下两个条件。

- 读操作r未先于写操作w发生。
- 没有其他对此变量的写操作后于写操作w且先于读操作r发生。

第一个条件应该很好理解。写操作是不可能被在它之前发生的读操作观察到的。而第二个条件的意思是：如果在写操作w发生之后又有其他写操作作用于该变量，那么之后发生的读操作r读到的就必定不是写操作w所产生的结果。因为在它们之间发生的写操作会覆盖掉w的结果。

此外，为了使对变量v的读操作r能够观察到特定的写操作w对v的改变，那么就要保证w是唯一允许r观察的写操作。也就是说，若要保证r能够观察到w所产生的结果，就需要同时满足如下

两个条件。

- 写操作w一定要发生在读操作r之前。
- 任何其他对共享变量v的写操作都只能发生在w之前或r之后。

相比于第一对充分条件，第二对充分条件的约束力更强。因为它们要求不能有与w或r同时发生的其他写操作。更具体地讲，第二对充分条件既不保证与r同时发生的写操作能被r观察到，也不保证与写操作w或读操作r同时发生的其他写操作不会被r观察到。换言之，Go语言不对同时发生的针对同一个变量的读写操作所产生的相互作用和最终结果做出任何假设。

对于单Goroutine的程序来说，这两对充分条件是等价的。因为在这样的程序中不存在并发的情况，也不可能有两个操作同时发生。这时，这两对充分条件可以被凝练为一句话，即读操作能且仅能观察到在它之前发生且离它最近的那个针对相同变量的写操作对该变量产生的结果。

相应地，对于多Goroutine的程序来说，应用程序在采取必要的同步措施之前肯定是无法满足上述充分条件的。因此，采用同步方法是实现对共享变量的安全访问的唯一途径。除了互斥量和条件变量之外，Go语言还提供了Channel这种既能实现操作同步又能满足通讯需要的高级方法。我们会在后面讲到它们。

注意，对一个已被声明的变量的第一次初始化操作形同于对该变量的写操作。这意味着，只有访问变量的操作在初始化它的操作之后发生才算是满足上述的“happens before”条件。

此外，针对长度超过一个机器字长的值的读写操作相当于多个对长度为一个机器字长的值的读写操作，并且其读或写的顺序是无法在语言层面保证的。举个例子，我们在32位计算架构的计算机上写入一个64位的整数，相当于分别写入两个32位的整数。因为在这样的计算机上，一条CPU运算指令最多只能修改一个长度为32位（与该计算机的字长相同）的数据。对于一个64位的数据来讲，只能通过两条指令分别修改它的高32位和低32位。虽然这两条指令分别写入的部分数据在逻辑上有高低之分，但执行它们的顺序却是不确定的。在这种情况下，如果存在与这个写操作并发的读操作，那么很有可能会读取到只修改了一半的数据。这比读取到一个旧数据更加糟糕。因此，我们更应该对这些操作进行同步。这也是Go语言的“happens before”原则不对同时发生的读写操作所产生的结果做任何假设的原因之一。

我们在使用同步方法的时候，应该使相应的操作满足上述的“happens before”条件。只有这样，Go语言才能够帮助我们实现对共享数据的安全访问。这是在编程过程中必须要注意的。在后面的内容中，我们会陆续介绍Go语言提供的各种同步方法是怎样辅助我们满足“happens before”条件的。

7.2 Channel

我们在本节会专门讲述Go语言所提倡的“应该以通信作为手段来共享内存”的最直接和最重要的体现——Channel。Channel也就是我们前面多次提到过的通道类型。它是Go语言预定义的数据类型之一。

Go语言鼓励使用与众不同的方法来共享值。这个方法就是使用一个通道类型值在不同的

Goroutine之间传递值。Go语言的Channel就像是一个类型安全的通用型管道。（实际上，Channel的设计灵感来源于Tony Hoare在1985年首次公开的专著*Communicating Sequential Processes*中的论述。）

Channel提供了一种机制。它既可以同步两个被并发执行的函数，又可以让这两个函数通过相互传递特定类型的值来进行通信。虽然在有些时候使用共享变量和传统的同步方法也可以实现上述用途。但是，作为一个更高级的方法，使用Channel可以使让我们更加容易地编写清晰、正确的程序。

下面，我们就开始介绍与Channel有关的各方面知识。

7.2.1 Channel是什么

在Go语言中，Channel即指通道类型。有时，我们也用它来直接指代可以传递某种类型的值的通道。通道即是某一个通道类型的值，是该类型的一个实例。

1. 类型表示法

与切片类型和字典类型相同，通道类型是引用类型之一。一个泛化的通道类型的声明应该是这样的：

```
chan T
```

其中，关键字chan是代表了通道类型的关键字，而T则表示了该通道类型的元素类型。通道类型的元素类型约束了可以经由此类通道传递的元素值的类型。例如，我们可以声明这样一个别名类型：

```
type IntChan chan int
```

别名类型IntChan代表了元素类型为int的通道类型。又例如，我们可以直接声明一个chan int类型的变量：

```
var intChan chan int
```

在被初始化之后，变量intChan就可以被用来传递int类型的元素值了。

以上展示了最简单的通道类型声明方式。这样的声明意味着该通道类型是双向的。也就是说，我们既可以向此类通道发送元素值，也可以从它那里接收元素值。此外，我们还可以声明单向的通道类型。这需要用到接收操作符<-。下面是只能被用来发送值的通道类型的泛化表示：

```
chan<- T
```

只能被用来发送值的意思是，我们只能向此类通道发送元素值，而不能从它那里接收元素值。接收操作符<-形象的表示了元素值可能的流向。我们把这样的单向通道类型简称为发送通道类型。当然，我们也可以声明只能从中接收元素值的通道类型，形如：

```
<-chan T
```

注意，这次接收操作符<-是在关键字chan的左边。这依然很形象，不是吗？相似地，此类的单向通道类型可以被简称为接收通道类型。

虽然在这样的类型声明中既有空格“ ”也有操作符<-，但是我们应该把它视为一个整体。它代表了一个类型，而不是某些复杂的东西。还记得吗？我们在第3章讲类型转换的时候提到过这样一个表达式：

```
<-chan int(v)
```

在这个容易让我们产生疑惑的表达式中，chan int是一个整体，代表了一个通道类型。要注意，操作符<-并不是该类型的一部分。因此，该表达式的含义是，先将变量v的类型转换为chan int，然后再从该值中接收一个元素值。它相当于：

```
<-(chan int(v))
```

它与表达式

```
*string(v) // 相当于 *(string(v))
```

遵从类似的规则。我们应该在不添加作为辅助的圆括号的时候就知晓它们所表达的真正含义。正因为前面的那个原始的表达式具有这样的含义，所以当我们想把变量v的类型转换成一个接收通道类型的时候就应该这样表示：

```
(<-chan int)(v)
```

在这里，附加的圆括号是必须的，也是必要的。否则，这个表达式就会被Go语言的编译器理解成之前的那种含义。

2. 值表示法

正因为通道类型是一个引用类型，所以一个通道类型的变量在被初始化之前它的值一定是nil。这也是此类型的零值。

与其他类型不同的是，通道类型的变量是被用来传递值的，而不是存储值的。所以，通道类型并没有对应的值表示法。它的值具有即时性，是无法用字面量来准确表达的。

3. 属性和基本操作

基于通道的通讯是在多个Goroutine之间进行同步的重要手段。而针对通道的操作本身也是同步的。在同一时刻，仅有一个Goroutine能向一个通道发送元素值，同时也仅有一个Goroutine能从它那里接收元素值。在通道中，各个元素值都是严格按照被发送至此的先后顺序排列的。最早被发送至通道的元素值会被最先接收。因此，通道相当于一个FIFO（先进先出）的消息队列。此外，通道中的元素值都具有原子性。它们是不可被分割的。通道中的每一个元素值都只可能被某一个Goroutine接收。已被接收的元素值会立刻被从通道中删除。

4. 初始化通道

我们已经知道，引用类型的值都需要使用内建函数make来初始化。通道类型也不例外。请看下面的调用表达式：

```
make(chan int, 10)
```

这个表达式初始化了一个通道类型的值。传递给make函数的第一个参数表明此值的具体类型是元素类型为int的通道类型，而第二个参数则指出该值在同一时刻最多可以容纳10个元素值。

也就是说，如果我们发送给该通道的元素值未被取走，那么该通道最多可以暂存（或者说缓冲）10个元素值。

当然，我们可以在初始化一个通道的时候省略第二参数值，像这样：

```
make(chan int)
```

还记得吗？我们在初始化一个字典类型值的时候也可以这样做。但是，这两个类似的做法所起到的作用（或者说达到的效果）是截然不同的。

对于字典类型的值来说，是否传递第二个参数只会影响到该值的初始长度。其初始长度与第二个参数值相同或为0。由于字典类型的值的长度是可以自动增长的，所以这通常不会造成什么不同，除了在对程序性能非常敏感的情况下。而对于通道类型的值来说，传与不传第二个参数值会对被初始化的这个通道的特性产生非常大的影响。

最根本的原因是，一个通道类型的值的缓冲容量是固定不变的。它可同时容纳的元素值的最大数量永远等于在它被初始化时给定的第二个参数值。如果第二个参数值被省略了，那么就表示被初始化的这个通道无法缓冲任何元素值。发送给它的元素值必须被立刻取走。Go语言的运行时系统会依据我们的初始化方式来确定通道的行为。我们会在下一小节详细论述这一点。

实际上，我们把初始化时给定了第二个参数的通道称为缓冲通道，而把初始化时未给定第二个参数的通道称为非缓冲通道。我们下面仅会讲解操作缓冲通道的方法。关于非缓冲通道的操作方法，我们会在后面专门说明。

5. 接收元素值

接收操作符`<-`不但可以被作为通道类型声明的一部分，也可以被用来对通道进行操作（发送或接收元素值）。假如有这样一个通道类型的变量：

```
strChan := make(chan string, 3)
```

内建函数`make`在被调用后会返回一个已被初始化的通道类型值作为结果。所以，这样的赋值语句使变量`strChan`成为了一个双向通道的代表。该通道的元素类型为`string`、容量为3。

如果我们此时要从该通道中接收元素值的话，应该这样编写代码：

```
elem := <-strChan
```

其中的接收操作符`<-`让我们很轻易就可以理解到这条语句所代表的含义——把`strChan`中的一个元素值赋给变量`elem`。不过，此时进行这类操作会使当前Goroutine被阻塞在这里。因为现在通道`strChan`中还没有任何元素值。当前Goroutine会被迫进入Gwaiting状态，直到`strChan`中有新的元素值可取时才会被唤醒。

我们在讲`<-`操作符的时候说过，像下面这样来编写这条赋值语句也是可以的：

```
elem, ok := <-strChan
```

与前面的写法相同，当该通道中没有任何元素值时，当前Goroutine会被阻塞于此。如果在进行接收操作之前或过程当中该通道被关闭了，那么该操作会立即被结束，并且变量`elem`会被赋予该通道的元素类型的零值。采用上述两种写法都会是如此。由于相应元素类型的零值也可以被发送到通道中，所以当我们接收到这样一个元素值的时候就无从判断它所代表的含义，即

它确实是通道中缓冲的一个元素值还是被用来表示该通道已经被关闭的标识。这时，第二种编写方法的优势就显现出来了。在特殊标记`:=`左边的第二个变量（这里是变量`ok`）被赋予的值会体现出实际的情况。在此语句中，变量`ok`必定会是一个布尔类型的值。当接收操作因通道关闭而被结束时，该值会为`false`（代表了操作失败），否则会为`true`。这样，我们就可以很容易地据此做出上述判断了。

我们把这些在特殊标记`=`或`:=`的右边仅能是接收表达式的赋值语句称为接收语句。在其中的接收操作符`<-`右边的不仅仅可以是代表通道的标识符，还可以是任意的表达式。只要这个表达式的类型是通道类型即可。我们把这样的表达式称为通道表达式。

最后，还有一点需要注意，试图从一个未被初始化的通道类型值（即值为`nil`的通道类型的变量）那里接收元素值会造成当前Goroutine的永久阻塞！

6. Happens before

为了能够从通道接收元素值，我们应该先向它发送元素值。理所当然，一个值在被从通道中取出之前必须先存在于该通道内。更加正式地讲，如果一个通道是带缓冲的，那么：

- 针对此通道的发送操作会被阻塞，直到被发送的元素值完全被复制到通道的缓冲区中。换句话说，通道缓冲元素值的这个动作的完成一定发生在相应的发送操作完成之前。
- 针对此通道的接收操作也会被阻塞，直到当前Goroutine真正从中获取到一个元素值。换句话说，当前Goroutine（进行接收操作的那个Goroutine）接收到某个元素值的这个结果一定会形成在相应的接收操作完成之前。
- 对于同一个元素值来说，把它发送给某个通道的操作一定会在从该通道接收它的操作完成之前完成。也就是说，在通道完全持有某一个元素值的副本之前，任何Goroutine都不可能从它那里接收到这个元素值。这一保证也是在前两个保证的基础上做出的。

以上就是与缓冲通道有关的“happens before”原则。Go语言的运行时系统会保证它们的有效性。这也是通道作为一个并发环境下的通讯工具应该具有的特性。

7. 发送元素值

通过了解上述原则，我们应该会对怎样操作一个缓冲通道有了更加清晰的认识。现在我们来讲讲向通道发送元素值的具体操作方法。这一操作是通过发送语句来完成的。

发送语句由通道表达式、接收操作符`<-`和代表元素值的表达式（以下简称元素表达式）组成。其中，元素表达式的类型必须与通道表达式的类型的元素类型之间存在可赋予的关系。也就是说，前者的值必须可以被赋给类型为后者的变量。

对接收操作符`<-`两边的表达式的求值会先于发送操作真正被执行之前。在对这两个表达式的求值完成之前，发送操作一定会被阻塞。并且，在这之后，发送操作是否得以进行还需要取决于其他因素。这在后面就会说明。

如果我们想向通道`strChan`发送一个元素值“a”的话，应该这样做：

```
strChan <- "a"
```

接收操作符`<-`左边是将要接纳元素值的通道，而右边则是欲发送给该通道的那个元素值。在此表达式被求值之后，通道`strChan`中就缓冲了元素值“a”。下面我们再向它发送两个元素值：

```
strChan <- "b"
strChan <- "c"
```

现在，strChan中已经缓冲了3个元素值。这已经是它能够容纳元素值的最大数量了。我们已经知道，一个通道的缓冲容量是固定的。因此，在这之后，当某个Goroutine中的向通道strChan发送值的操作被执行的时候，该Goroutine会被阻塞在那里，直到该通道中有足够的空间容纳该元素值为止。在我们再从该通道中接收一个元素值之后，那个Goroutine就会被立即唤醒并且完成那个发送操作。

注意，与接收操作相同，当我们向一个值为nil的通道类型的变量发送元素值的时候，当前Goroutine也会被永久地阻塞！除此之外，如果我们试图向一个已被关闭的通道发送元素值，那么会立即引发一个运行时恐慌。即使发送操作正在因通道缓冲已满而被阻塞，这个通道的关闭同样会使该操作引发一个运行时恐慌。这一点需要特别注意。我们肯定不希望因通道的关闭而使正常流程被迫中断。为了避免这样的流程中断，我们可以在select代码块中进行发送操作。这会在后面专门说明。

我们已经知道，针对通道的发送操作和接收操作都可能造成相关Goroutine的阻塞。如果有多个Goroutine因向同一个通道发送元素值而被阻塞，那么当该通道中有多余的缓冲空间的时候，最早被阻塞的那个Goroutine会最先被唤醒。也就是说，这里的唤醒顺序与发送操作的开始顺序相同。这对于接收操作来说也是如此。一旦通道中有了新的元素值，那么最早因从该通道接收元素值而被阻塞的那个Goroutine会最先被唤醒。无论是发送操作还是接收操作，运行时系统每次只会唤醒一个Goroutine。

关于这里的发送操作还有一点需要注意。那就是，在我们向通道发送一个值之后，该通道将会得到该值的一个副本，而非该值本身。当这个副本形成之后，我们对那个原来的值的任何修改都不会影响到通道中相应的副本。例如，我们有这样两个结构体声明：

```
type Person struct {
    Name    string
    Age     uint8
    Address Addr
}

type Addr struct {
    city      string
    district string
}
```

注意，在结构体Person中有一个名为Address的字段。这个字段的类型也同时是一个结构体——Addr。现在，我们声明并初始化一个以Person为元素类型的通道，像这样：

```
var personChan = make(chan Person, 1)
```

这个由变量personChan代表的通道的容量是1。然后，我们创建一个Person类型的值并把它发送给personChan：

```
p1 := Person{"Harry", 32, Addr{"Beijing", "Haidian"}}
fmt.Printf("p1 (1): %v\n", p1)
personChan <- p1
```

可以看到，p1所代表的那个人大致住址是北京市海淀区。在发送操作完成之后，我们对p1的住址稍作改变：

```
p1.Address.district = "Shijingshan"
fmt.Printf("p1 (2): %v\n", p1)
```

第一行代码表明Harry已从北京市海淀区搬到了北京市石景山区。接下来，我们从personChan中的那个唯一的元素值，看看它是否会随着p1的值的改变而被改变：

```
p1_copy := <-personChan
fmt.Printf("p1_copy: %v\n", p1_copy)
```

为了展现效果，我们在这小段代码中加入了3条打印语句。在运行这些代码之后，标准输出上会出现如下内容：

```
p1 (0): {Harry 32 {Beijing Haidian}}
p1 (1): {Harry 32 {Beijing Shijingshan}}
p1_copy: {Harry 32 {Beijing Haidian}}
```

显而易见，通道中的元素值丝毫没有受到外界的影响。这说明了，在发送过程中进行的元素值复制并非浅表复制，而属于完全复制。这也保证了我们使用通道传递的值的不变性。

8. 关闭通道

从代码实现上来讲，关闭一个通道的操作实际上是非常简单的。通过对内建函数close的调用就能够实现。例如，如果我们想关闭通道strChan，那么如此编写一个调用表达式就可以了：

```
close(strChan)
```

我们在前面讲过，不合时宜地关闭一个通道可能会给针对它的发送操作和接收操作带来问题。这样可能会对它们所在的Goroutine的正常流程的执行产生影响。因此，我们应该在保证安全的情况下进行关闭通道的操作。这会涉及一些技巧。比如我们会在后面讲到的for语句和select语句。不过在讲解这些高级方法之前，我们应该先明确一点：无论怎样都不应该在接收端关闭通道。因为在那里我们无法判断发送端是否还会向该通道发送元素值。如果非要这样做，那么就应该使用一些辅助手段来避免发送端引发运行时恐慌。然而，我们在发送端调用close以关闭通道却不会对接收端接收该通道中已有的元素值产生任何影响。这也是通道非常优秀的特性之一。示例如下：

```
func main() {
    ch := make(chan int, 5)
    sign := make(chan byte, 2)
    go func() {
        for i := 0; i < 5; i++ {
            ch <- i
            time.Sleep(1 * time.Second)
        }
        close(ch)
        fmt.Println("The channel is closed.")
        sign <- 0
    }()
    go func() {
```

```

    for {
        e, ok := <-ch
        fmt.Printf("%d (%v)\n", e, ok)
        if !ok {
            break
        }
        time.Sleep(2 * time.Second)
    }
    fmt.Println("Done.")
    sign <- 1
}()
<-sign
<-sign
}

```

在该示例中，我们分别启用了两个Goroutine来对通道进行发送操作和接收操作。发送操作共有5次，每次操作的间隔是1秒。在所有发送操作都完成之后，我们会立即关闭该通道。另一方面，接收操作会持续地进行，每次操作的间隔是2秒。在通过接收语句中的第二个被赋值的变量得知该通道已被关闭之后，我们会结束包含它的for循环，并打印Done.。上述发送操作和接收操作的不同间隔时间的意义在于，接收端在没有将通道中已有的元素值全部接收完毕之前，该通道就会被关闭。

在我们运行这段代码之后，标准输出上会被打印出如下内容：

```

0 (true)
1 (true)
2 (true)
The channel is closed.
3 (true)
4 (true)
0 (false)
Done.

```

很明显，运行时系统并没有在通道ch被关闭之后立即把false作为相应接收操作的第二个结果，而是等到接收端把已在通道中的所有元素值都接收到之后才这样做。这确保了在发送端关闭通道的安全性。

由此，更确切地讲，调用close函数的作用是告诉运行时系统不应该再允许任何针对被关闭的通道的发送操作，该通道即将被关闭。虽然我们调用close函数只是让相应的通道进入关闭状态而不是立即阻止对它的一切操作，但是为了简化概念我们仍然笼统地称在对close函数的调用返回之后该通道就已经被关闭了。不过，读者应该将这其中的真正含义铭记于心。

对于同一个通道，运行时系统只允许我们关闭一次。因此，试图关闭一个已被关闭的通道会引发一个运行时恐慌。而若在调用close函数时的参数值是一个值为nil的通道类型的变量则也会是如此。

顺便提一下，我们在这段代码中还另外声明并初始化了一个通道sign。该通道在这里的作用是推迟主Goroutine被运行完成的时间。这利用到了通道本身的特性，也是我们在主Goroutine中启用的那两个Goroutine能够被正常运行完成的关键。我们在下一小节还会碰到这种惯用法。

如果读者需要了解与内建函数`close`有关的更多知识，请查阅本书的第3章。

9. 通道的长度与容量

内建函数`len`和`cap`也是可以作用在通道之上的。它们的作用分别是获取通道的当前实际缓冲的元素值的数量（即长度）和通道可容纳元素值的最大数量（即容量）。通道的容量是在初始化该通道的时候已经设定的，并且在之后也不会被改变。而通道的长度则会随着实际的情况实时变化。

这两者的区别也决定了我们对它们的使用方法的不同。对于通道的容量来说，如果我们需要重复使用它，就应该先使用一个变量将它缓存起来。然而，对于通道长度，我们决不能这样做。因为我们是无法确定它变化的时间以及具体数值的。因此，我们应该在每次要获取通道的长度的时候都调用一次`len`函数。

除此之外，我们可以通过的容量来判断它是否带有缓冲。若其容量为0，那么它肯定就是一个不带缓冲的通道，否则它就应该是一个带缓冲的通道。

至此，我们已经介绍了通道类型及其值的表示法、基本属性，以及操作缓冲通道的基本方法。通道类型的表示法非常独特，其中必须包含关键字`chan`、空格“ ”和代表元素类型的字面量。然而，通道类型的值（之后也会简称为通道）却因其即时性而无法被表示。我们最应该记住的通道的特性就是它的自同步和其中元素值的原子性。这些特性在针对通道的接收操作和发送操作中体现得最为明显。在本节的后续部分，我们会陆续介绍各种与通道相关的规则以及惯用法。依照这些规则和惯用法，我们就能够充分发挥出这一独特的通讯工具的强大威力，并可以用它构建出非常复杂但依然稳固、清晰的并发程序。

7.2.2 单向Channel

我们在上一小节讲通道类型声明的时候提到过单向通道这个概念。但没有讲到它们的应用场景。单向通道可分为接收通道和发送通道。需要注意的是，无论哪一种单向通道，都不应该出现在一个变量的声明中。这是为什么呢？请试想一下，如果我们声明并初始化了这样一个变量：

```
var uselessChan chan<- int = make(chan<- int, 10)
```

那么应该怎样去使用它呢？显然，一个只进不出的通道没有任何意义。虽然这行代码可以通过编译，但它其实没有什么可用之处。那么单向通道的应用场景又在哪里呢？

实际上，单向通道常常由双向通道变换而来。我们可以用这种变换来约束某个函数或某个使用方程序对通道的使用方式。例如，我们在上一章讲信号的时候介绍过，`os/signal.Notify`函数的声明是这样的：

```
func Notify(c chan<- os.Signal, sig ...os.Signal)
```

该函数的第一个参数就是一个单向通道类型的。从表面上看，调用它的程序需要传入一个只能发送而不能接收的通道。然而，我们不应该传给它这样一个值。实际上，这个参数声明表达的含义是，`os/signal.Notify`函数内部只会对该通道发送元素值，而不会从该通道接收元素值。因为，试图从一个发送通道中接收元素值会造成一个编译错误。从另一方面看，调用该函数的程序

应该只从这个通道中接收元素值，向其发送元素值只会造成纯粹的干扰。也就是说，参数c是函数内部向函数调用方传递系统信号的一个途径。这个途径的方向应该是单一的，通讯双方会分别使用到不同的操作（即发送或接收）。综上所述，我们在调用此函数的时候应该传入一个双向的通道，并且要自觉遵守该函数声明中隐含的约定。双向通道在被传递给该函数的过程中会被自动地转换为相应参数声明所示的单向通道。

函数os/signal.Notify以这样的声明方式向我们传达了它的第一个参数的真正意义。它是利用Go语言现有的语法规则做到这一点的。同时，这也是“代码即注释”这种编程风格的一个体现。这种Go语言特有的代码编写手法是值得我们学习和效仿的。

请想象一下，如果这样一个声明被包含在接口声明中会起到什么样的作用？例如：

```
type SignalNotifier interface {
    Notify(c chan<- os.Signal, sig ...os.Signal)
}
```

接口类型SignalNotifier的声明中包含了与os/signal.Notify函数完全一样的方法声明。声明一个接口类型的意义就在于会有一到多个自定义数据类型来实现它。它是对某一类数据类型的归纳和抽象。因此，这里的参数c的声明明确表达了一个实现规则，即所有实现该接口的数据类型的Notify方法内部只能向c发送元素值。这就相当于利用语法级别的约束来避免实现类型对参数c的实际值进行错误的操作。

现在，我们再来对SignalNotifier接口类型的声明稍作改变。改变后的声明是这样的：

```
type SignalNotifier interface {
    Notify(sig ...os.Signal) <-chan os.Signal
}
```

可以看到，我们把Notify方法中的第一个参数声明去掉了，然后为它添加了一个看起来与前者有些类似的结果声明。请注意接收操作符<-与关键字chan的位置关系。在结果声明中的是一个接收通道，而非发送通道。与前一个版本的Notify方法的声明恰恰相反，此方法声明的约束目标是方法的调用方，而非方法的实现方。Notify方法的调用方只能从其结果值中接收元素值，而不能向其发送元素值。

这两个版本的Notify方法传递系统信号的方式是相同的——使用单向通道。并且，系统信号在其中的单向通道中的传递方向也是相同的——从方法内部传至方法调用方。这表明它们所体现的功能完全一致。

这两个方法声明的真正不同在于对其中的单向通道的使用方式，即它们分别对单向通道一端的使用者进行了约束。这使得它们分别适用于不同的应用场景。前一个版本的方法声明更合适存在于接口类型中，因为它可以作为该接口的实现规则之一。后一个版本的声明更适用于结构体的方法和独立函数，原因是它隐含并约束了对函数或方法的结果值的使用方式。但是，这并不是绝对的。比如，在os/signal.Notify函数的声明中，参数c的类型就隐含了函数调用方对该通道的使用规则。虽然此规则是可以轻易被破坏的，但是这对于函数调用方来说没有任何好处。因此，这样是可以达到约束目的的。

当然，我们在同一个函数声明中连用上述两种方式也是可以的。例如，我们需要批量地更改一些人员信息（由在上一小节编写好的Person类型的值代表）中的住址。这其中有几个具体要求。

- 我们事先并不知道这些人员都有哪些。
- 住址的更改方式可以由使用者灵活掌握，并能够实时地替换。
- 经过处理的人员信息应该自动地被传递给使用者。

这样一个批处理任务应该怎样实现呢？请读者先自己思考一下。

如果你确实经过了思考，请继续往下看。我们先按照上述要求定义一个接口：

```
type PersonHandler interface {
    Batch(origs <-chan Person) <-chan Person
    Handle(orig Person)
}
```

接口类型PersonHandler中包含了两个方法声明。方法Batch被声明为了实现批量处理人员信息功能的方法。其中的参数声明可以满足第一个具体要求。需要被更改住址的人员信息可以经由参数origs传递至方法内部。而其中的结果声明则满足了第三个要求。方法调用方只要持有该方法的结果值就可以实时地接收到已被处理的人员信息。注意，在Batch方法的声明中的这两个通道分别对该方法和该方法的调用方使用它的方式进行了约束。至于第二个要求，我们是通过该接口中的另一个方法来实现的。这会在后面的Batch方法的实现中展现出来。

由于我们这里只关注与单向通道的使用有关的部分，所以下面仅讨论PersonHandler接口中的Batch方法的实现。不过在这之前，我们应该先展示一下该接口的实现类型的声明：

```
type PersonHandlerImpl struct{}
```

这个声明很简单。它其实就是一个不包含任何字段的结构体类型。为了让这个类型真正实现接口类型PersonHandler，我们还要声明这样两个方法：

```
func (handler PersonHandlerImpl) Batch(origs <-chan Person) <-chan Person {
    // 省略若干条语句
}

func (handler PersonHandlerImpl) Handle(orig Person) {
    // 省略若干条语句
}
```

好了，现在我们就开始填充PersonHandlerImpl类型的Batch方法的方法体。首先，既然需要被更改的人员信息会通过单向通道origs传递进来，那么我们就应该不断地试图从该通道中接收它们。不过，我们应该处理通道已被关闭的情况。因为这象征着所有的人员信息都已被传递完毕，也预示着批处理流程执行的结束。所以，我们应该这样来编写Batch方法体的第一个版本：

```
for {
    p, ok := <-origs
    if !ok {
        break
    }
}
```

接下来，我们再来考虑作为结果值的那个单向通道。由于这个单向通道不来自于Batch的参数，所以该方法应该自行创建并初始化它。又由于Batch方法的返回并不意味着这一批处理过程的完成(既然该方法的结果值是一个单向通道，那么就应该充分体现出这种结果传输方式的优势，即传输应该是异步的)。综上所述，Batch方法体的第二个版本如下：

```
dests := make(chan Person, 100)
for {
    p, ok := <-origs
    if !ok {
        close(dests)
        break
    }
    handler.Handle(p)
    dests <- p
}
```

如上所示，我们先调用make函数并初始化了一个元素类型为Person、容量为100的双向通道dests。然后，每当从单向通道origs中接收到一个有效的人员信息p的时候，我们都会使用本类型的Handle方法来处理这个p。最后，我们会及时地把刚刚被处理完成的p发送给dests。

可以看到，for代码块的执行会一直持续到通道origs被关闭的时候。所以，在此之前，对Batch方法的调用是无法结束的。方法调用方也许会长时间得不到该方法的结果值。并且，如果是这样的话，我们使用通道作为Batch方法的结果值的做法也失去了意义。那么我们应该怎样做呢？

答案是启用一个Goroutine来执行for语句块。只有这样才能真正地实现已处理p的异步传输。请看下面的代码：

```
func (handler PersonHandlerImpl) Batch(origs <-chan Person) <-chan Person {
    dests := make(chan Person, 100)
    go func() {
        for {
            p, ok := <-origs
            if !ok {
                close(dests)
                break
            }
            handler.Handle(p)
            dests <- p
        }
    }()
    return dests
}
```

我们之前说过，基于通道的通讯是在多个Goroutine之间进行同步的重要手段。通道既能够被用来在多个Goroutine之间传递数据，又能够在数据传递的过程中起到同步的作用。在上面的这个Batch方法的第三个版本中，通道的这两个重要用途都被很好地展现了出来。如果我们把Batch方法的调用方所在的Goroutine称为G1，并把在该方法中新启用的Goroutine称为G2，那么利用通道在它们之前传递数据的示意如图7-1所示。

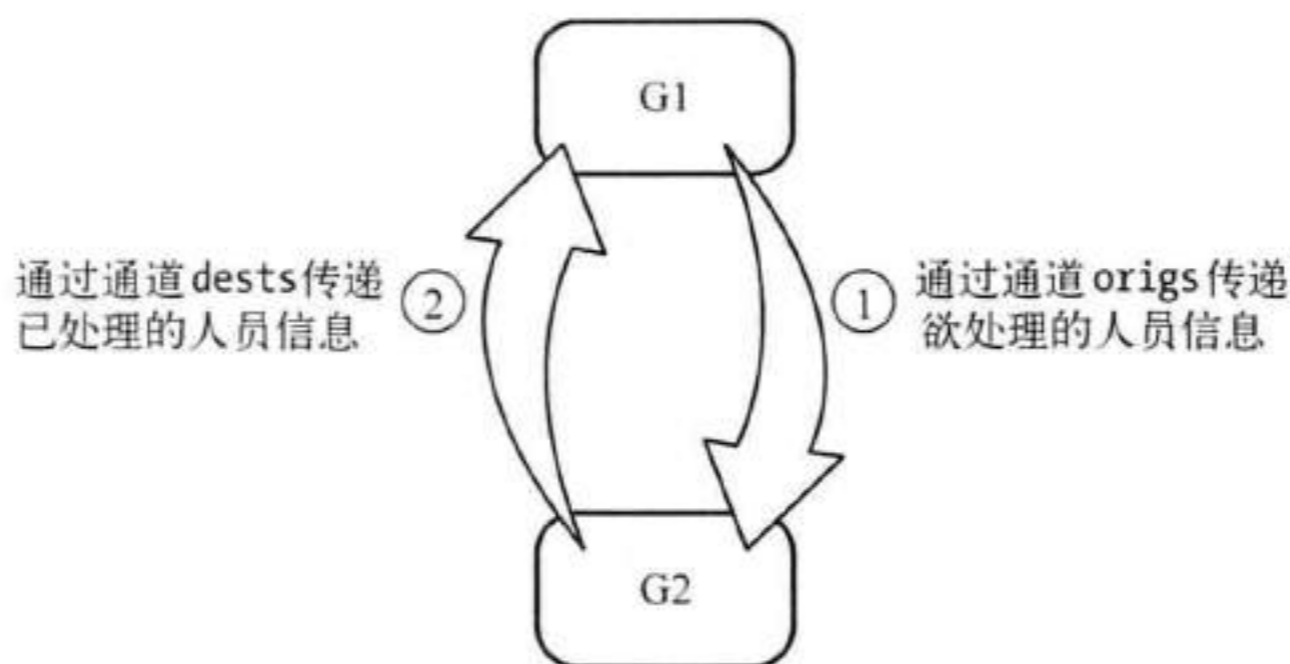


图7-1 通道的作用示意

至此,通过对Goroutine和通道的使用,我们以异步的方式实现了对若干人员信息的批量处理。读者应该能够从这个示例中初步感受到通道的威力。这种实现方式要比使用其他同步手段(比如互斥量或条件变量)强很多。实际上,我们可以轻易地使用通道模拟出前面提到的大部分同步方法的应用流程。读者如果感兴趣的话,可以试着写出这些模拟代码。

更进一步地,如果我们要更完整地实现批量更改人员信息的流程的话,G1中的代码就可能会是这样:

```
handler := getPersonHandler()
origs := make(chan Person, 100)
dests := handler.Batch(origs)
fetchPerson(origs)
sign := savePerson(dests)
<-sign
```

其中, `getPersonHandler` 函数会返回一个 `PersonHandler` 类型的结果值。这个值的动态类型可以是我们在前面部分实现了的 `PersonHandlerImpl` 类型,也可以是其他实现了 `PersonHandler` 接口的类型。函数 `fetchPerson` 的功能是从某处取出欲处理的人员信息并把它们发送给 `origs` 通道,而 `savePerson` 函数的功能则是从 `dests` 通道中取出已变更的人员信息并把它们存储到某处。此外, `savePerson` 函数还会返回一个通道 `sign`。该通道的作用是,在批处理完全执行结束之前阻塞 G1。由于在通道中没有任何元素值的时候,对缓冲通道的接收操作会阻塞当前的 Goroutine,所以实现此功能也是非常容易的。以下是上述3个函数的声明:

```
func getPersonHandler() PersonHandler
func fetchPerson(origs chan<- Person)
func savePerson(dest chan Person) chan byte
```

请读者自行实现这3个方法。必要时,读者可以参照图7-2所示的流程图。

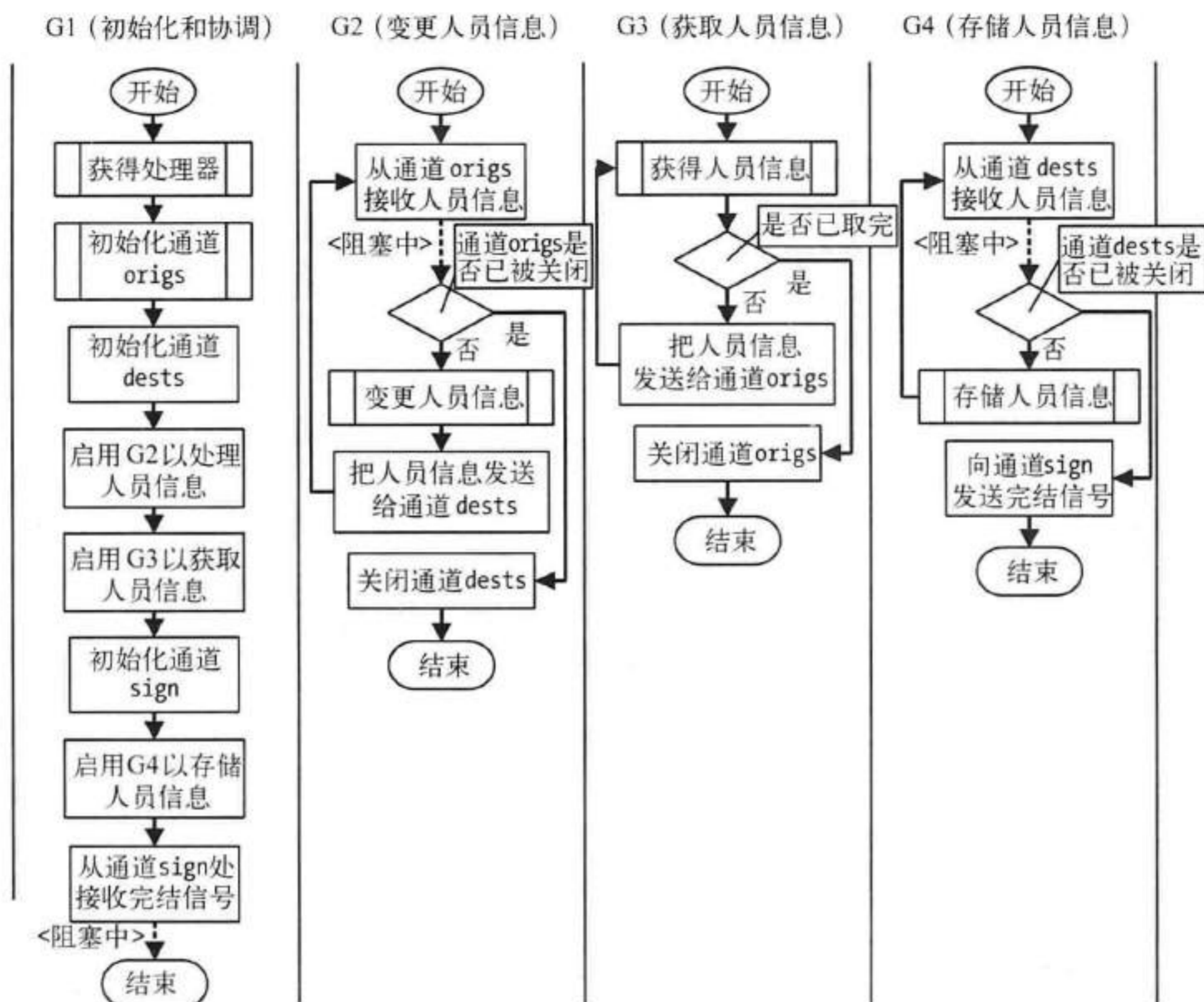


图7-2 批量处理人员信息的流程

在这幅流程图中，我们以Goroutine为界划分出了4个子流程。为什么又多出了两个Goroutine呢？这主要是为了让此批处理流程实现完全的异步化。所以，建议异步地获取和存储人员信息，并使用现有的两个通道origs和dests满足它们与其他Goroutine之间的同步和传递数据的需要。我们称用于获取人员信息的Goroutine为G3，而用于存储人员信息的Goroutine为G4。它们分别与fecthPerson函数和savePerson函数的功能相对应。可以看到，G3和G4的流程与G2的流程非常相似。所以把它们编写出来应该并不困难。

或许，读者应该先动手实现出普通版本的fecthPerson函数和savePerson函数。也就是说，不让它们启用Goroutine，而只是单纯地向通道origs发送原始的人员信息，以及从通道dests接收已被处理的人员信息。即先再想办法让这个批处理程序正确地运行起来。其中，人员信息的获取操作和存储操作可以用模拟的函数代替。

这两个函数的普通版本是否可以实现批处理流程的异步化呢？请读者先去编写它们，然后再来思考这个问题。

为了让读者看得更清楚，我们再通过图7-3描述一下人员信息经由两个通道在多个执行步骤之间的流转情况。

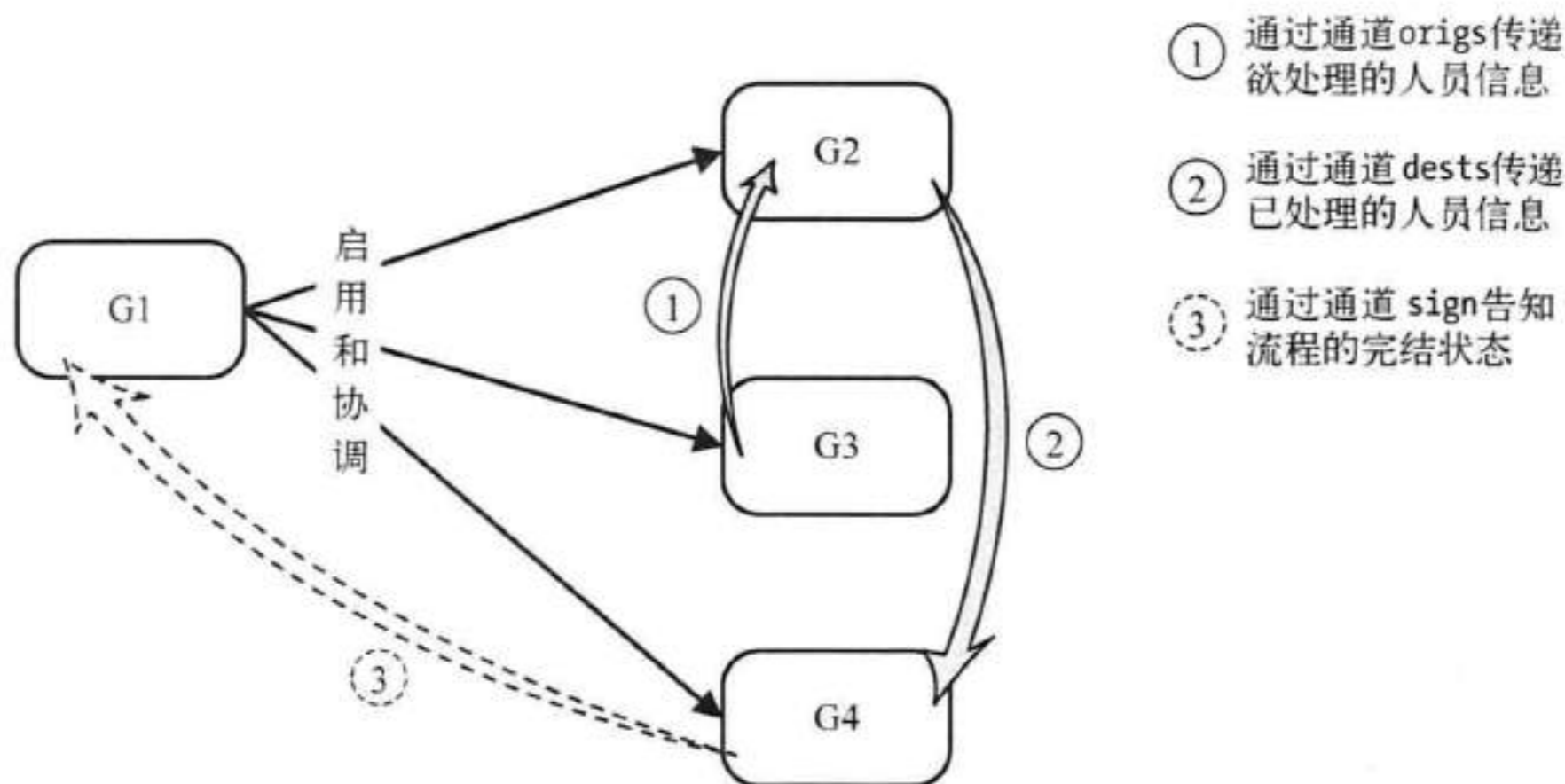


图7-3 人员信息的流转示意

我们应该让每一个人员信息可以独立地穿过整个的批处理程序。在穿过的过程中，批处理程序中的多个执行步骤会先后对此人员信息进行处理。然而，普通版本的`fetchPerson`函数和`savePerson`函数会让已经被异步化的`Batch`方法所做的努力变得毫无意义。为什么要把所有需要处理的人员信息都获取到之后再对它们进行变更呢？这样会增长大多数人员信息的处理时间，也会使整个批处理流程笨重许多。

总之，对于运行在多核CPU的计算机上的程序来说，使用并发编程的手法总会比普通的串行编程更高效一些。但是，我们需要仔细考量，并发编程所带来复杂性是否值得我们这样做。在Go语言的并发程序中，通道会成为联系各个Goroutine的纽带。它可以正确、清晰和高效地实现Goroutine之间的同步和实时的数据传递。我们借着介绍单向通道的机会，通过前面的这个示例展示了双向通道和单向通道的典型应用场景和使用方式。

最后，请注意，在这个已被异步化的批处理流程中，原先参与人员信息传递的G1已经退居幕后了。它现在的主要职责是初始化各种资源以及对G2、G3和G4的运行进行协调。在并发的程序中，我们总是应该构建一段可以统揽全局、协调各个部件的程序。

其实，这个批处理流程中的几个子流程以及前面所示的`Batch`方法的实现代码都依然存在着很大的优化空间。比如，我们可能会需要并发地处理或存储多个人员信息。但是，鉴于本小节的主题和篇幅，我暂时就讨论到这里。一个该流程的较完整的参考实现存放在`goc2p`项目的代码包`chan1/oneway`中，文件名为`phandler.go`。不过，强烈建议读者在实现了自己的版本并正确运行之后再去看它。

经过前面的练习和思考之后，相信读者对双向和单向的通道的基本使用会有一个比较深刻的理解。请记住，直接初始化一个单向通道毫无意义。单向通道的真正含义是在一定作用域（某个代码块）内约束使用者对它的使用方式，也即暗示和控制相应数据的流转方向。

单向通道往往由双向通道转换而来。那么，单向通道是否可以被转换回双向通道呢？请记住这样一句话，通道所允许的数据传递方向是它的类型的一部分。对于两个通道类型而言，方向的

不同就意味着它们类型的不同。也就是说，元素类型相同的双向通道、发送通道和接收通道都属于不同的类型。所以，我们不能像下面这样转换通道的类型：

```
var ok bool
ch := make(chan int, 1)
_, ok = interface{}(ch).(<-chan int)
_, ok = interface{}(ch).(chan<- int)

sch := make(chan<- int, 1)
_, ok := interface{}(sch).(chan int)

rch := make(<-chan int, 1)
_, ok := interface{}(rch).(chan int)
```

在上面这段代码中，每一个类型转换表达式的结果都会是否定的，即变量ok的值总会是false。

因此，我们只能利用函数声明来约束通道的方向。比如，利用函数的参数声明把函数调用方所持的双向通道转换为单向通道，并提供给函数内部使用。又比如，利用函数的结果声明把函数内部所持的双向通道转换为单向通道，并提供给函数调用方使用。

注意，即使利用函数声明转换通道类型，也无法把单向通道转换为双向通道。并且，通过这种方式也不可能改变单向通道的方向。尝试这样做只会造成编译错误。请看下面的这个示例：

```
ch1 := make(chan int, 1)
f := func(c chan<- int) chan int {
    return c // 这里会造成编译错误
}
ch2 := f(ch1)
```

变量ch1是元素类型为int的双向通道。我们把它作为参数值传给了变量f代表的匿名函数。在这个匿名函数的内部，它变成了一个发送通道。然后，我们想以相似的方式让它变回双向通道，但是却失败了。该匿名函数中的return语句会造成一个编译错误。由此看来，我们只能用这种方式附加对通道的使用约束，而不能解除它。这难免让人有些遗憾，但是这也确实避免了因太过灵活而导致的混乱局面。

好了，通过认真阅读本小节的内容，你是否对单向通道的概念、使用和含义真正理解了呢？我想，答案应该是肯定的。

7.2.3 for语句与Channel

我们在讲for语句的时候已经提到过，使用该语句及其range子句可以持续地从一个通道中接收元素值。因此，我们在本小节只是再简单汇总一下与该用法有关的知识。

首先，我们来看一下这种用法的基本表现形式：

```
var ch chan int
// 省略若干条语句
for e := range ch {
    fmt.Printf("Element: %v\n", e)
}
```

我们先声明了一个双向通道，然后试图使用for语句接收其中的元素值。在单次的迭代中，range子句会尝试从通道ch中接收一个元素值，并把它赋给唯一的迭代变量e。注意，range子句的迭代目标不能是一个发送通道。与从发送通道中接收元素值的行为一样，这样做会造成一个编译错误。

我们在前面说过，从一个还未被初始化的通道中接收元素值，会导致当前Goroutine被永久地阻塞。当然，使用for来进行接收操作也不会例外。因此，上面的代码存在一个明显的缺陷。下面是改进的版本：

```
var ch chan int
// 省略若干条语句
if ch != nil {
    for e := range ch {
        fmt.Printf("Element: %v\n", e)
    }
}
```

这样确实可以避免因通道初始化问题导致的Goroutine阻塞。但是这种接收方式与普通的接收操作一样，当通道（注意，我们在本小节依然只针对缓冲通道）中没有任何元素值的时候，当前Goroutine依然会陷入阻塞。阻塞的具体位置会在其中的range子语句处。

语句for会不断地尝试从通道中接收元素值，直到该通道被关闭。在相关的通道被关闭后，若通道中已无元素值或当前的Goroutine正阻塞于此，则这条for语句的执行会立即结束。而当此时的通道中还有遗留的元素值时，运行时系统会等for语句把它们全部接收后再结束该语句的执行。当然，在结束执行该for语句之前，当前的Goroutine会先被唤醒。

经过上述说明，我们可以很容易地把已在上一小节中实现的Batch方法改造成这样：

```
func (handler PersonHandlerImpl) Batch(origs <-chan Person) <-chan Person {
    dests := make(chan Person, 100)
    go func() {
        for p := range origs {
            handler.Handle(p)
            dests <- p
        }
        fmt.Println("All the information has been handled.")
        close(dests)
    }()
    return dests
}
```

在这个版本的Batch方法中，for语句不断尝试接收通道origs中的元素值。每接收到一个元素值，for代码块中的代码都会立即对它进行处理，并通过dests通道把它传递给下一道工序。若在某次迭代开始时发现origs通道已被关闭且已无元素值可接收，那么运行时系统就会立即结束这条for语句的执行，并继续执行在它后面的语句。这次改造使得该Goroutine（G2）中的代码的数量减少了三分之一，并且在流程的实现上也更加清晰了。

在需要循环的接收通道中的元素值的场景下，我们总是应该优先使用for语句来实现。

7.2.4 select语句

在本小节，我们要介绍一个仅能被用于发送和接收通道中的元素值的专用语句——select语句。一个select语句在被执行的时候会选择执行其中的某一个分支。在表现形式上，select语句与switch语句非常类似，但是它们选择分支的方法是完全不同的。

1. 组成和编写方法

在select语句中，每个分支依然以关键字case开始。但与switch语句不同的是，跟在每个case后面的只能是针对某个通道的发送语句或接收语句。我们在7.2.1节中专门介绍过这两种语句。select语句的另一个特点是，在select关键字右边并没有像switch语句那样的switch表达式，而是直接后跟花括号“{”。这也与它选择分支的方法有关。下面是select语句的典型用法的一个示例：

```
var ch1 = make(chan int, 10)
var ch2 = make(chan string, 10)
// 省略若干条语句
select {
case e1 := <-ch1:
    fmt.Printf("1th case is selected. e1=%v.\n", e1)
case e2 := <-ch2:
    fmt.Printf("2th case is selected. e2=%v.\n", e2)
default:
    fmt.Println("default!")
}
```

针对这条select语句中的每一个case，我们都初始化了一个通道。通道的类型不受任何约束。也就是说，在select语句中操作的那些通道的元素类型和容量都可以是任意的。另外，select语句也可以包含default case（也可以被称为默认分支）。如果select语句中的所有case都不满足选择条件且存在default case，那么default case就会被执行。

2. 分支选择规则

在开始执行select语句的时候，所有跟在case关键字右边的发送语句或接收语句中的通道表达式和元素表达式都会先被求值。求值的顺序是自上而下、从左到右的。无论它们所在的case是否有可能被选择都会是这样。我们通过下面的示例就可以证实这一点。

首先，我们需要准备如下几个变量：

```
var ch3 chan int
var ch4 chan int
var chs = []chan int{ch3, ch4}
var numbers = []int{1, 2, 3, 4, 5}
```

其中，变量chs和numbers分别代表了包含了有限元素的通道列表和整数列表。下面的select语句使用到了它们：

```
select {
case getChan(0) <- getNumber(2):
    fmt.Println("1th case is selected.")
case getChan(1) <- getNumber(3):
    fmt.Println("2th case is selected.")
}
```

```
default:
    fmt.Println("default!")
}
```

这其中包含了两个通道表达式和两个元素表达式。通道表达式由对getChan函数的调用表达式代表，而元素表达式则由对getNumber函数的调用表达式代表。这两个函数的声明如下：

```
func getNumber(i int) int {
    fmt.Printf("numbers[%d]\n", i)
    return numbers[i]
}

func getChan(i int) chan int {
    fmt.Printf("chs[%d]\n", i)
    return chs[i]
}
```

我们在这两个函数中分别添加了打印语句。通过这些被打印出的内容，我们可以得知这些表达式被求值的顺序。当上面那个select语句被执行时，标准输出上会出现如下的内容：

```
chs[0]
numbers[2]
chs[1]
numbers[3]
default!
```

上面内容的最后一行表示在本次执行select语句的过程中被选中并执行的是default case。不过，虽然前面那两个case都没有被选中，但是它们的表达式都被求值了。从前4行内容可以看出，它们的求值顺序正如我们前面所说的那样。也就是说，运行时系统会自上而下地求值每个case的表达式。并且，同一个case的多个表达式会以从左到右的顺序被求值。

这其中的通道是可以为nil的，不管代表它的是标识符还是表达式。但是，这样的话，它所属的case就会永远被无视，就像select语句中根本就没有包含它一样。例如，在下面的select语句中，被选中并执行的永远会是default case：

```
var ch5 chan int
var ch6 chan string
select {
case ch5 <- 1:
    fmt.Println("1th case is selected.")
case ch6 <- "1":
    fmt.Println("2th case is selected. ")
default:
    fmt.Println("default!")
}
```

这实际上是由这两种操作的特性决定的。还记得吗？针对值为nil的通道类型的变量的发送操作和接收操作都会使当前的Goroutine被永久地阻塞。因此，根据针对select语句的分支选择规则，这类case才会给予我们这样的表象。下面，我们来说说select语句分支选择的具体规则。

在执行select语句的时候，运行时系统会自上而下地判断每个case中的发送或接收操作是否可以立即进行。这里的立即进行的意思是当前Goroutine不会因此操作而被阻塞。所以，对这个

是否可立即进行的判定还需要依据通道的具体特性（缓冲或非缓冲）和那一时刻的具体情况来进行。

当发现第一个满足选择条件的case时，运行时系统就会执行该case所包含的语句。这也意味着其他case会被忽略。如果同时有多个case满足条件，那么运行时系统会通过一个伪随机的算法决定哪一个case将会被执行。例如，下面的代码会随机的向一个通道发送5个范围在[1,3]的整数：

```
chanCap := 5
ch7 := make(chan int, chanCap)
for i := 0; i < chanCap; i++ {
    select {
        case ch7 <- 1:
        case ch7 <- 2:
        case ch7 <- 3:
    }
}
for i := 0; i < chanCap; i++ {
    fmt.Printf("%v\n", <-ch7)
}
```

这段代码被执行后，打印在标准输出上的内容可能会是：

```
3
2
2
1
2
```

另一方面，如果被执行的select语句中的所有case都不满足选择条件并且没有default case的话，那么当前Goroutine就会一直被阻塞于此，直到某一个case中的发送或接收操作可以被立即进行为止。注意，如果这样的select语句中的所有case右边的通道都是nil，那么当前Goroutine就会永远地被阻塞在这条select语句上！我们的程序中永远不要出现像下面这样的代码：

```
var ch8 chan int
var ch9 chan string
select {
case ch8 <- 1:
    fmt.Println("1th case is selected.")
case ch9 <- "1":
    fmt.Println("2th case is selected. ")
}
```

如果当前程序只有主Goroutine且包含了这段代码的话，那么就会导致死锁的发生。

所以，在通常情况下，default case对于select语句来说总是有必要的。select语句只能包含一个default case，且这个特殊的case可以被放置在该语句的任何位置上：

```
ch10 := make(chan int, 10)
ch10 <- 1
select {
default:
    fmt.Println("default!")
case e10 := <-ch10:
```

```
    fmt.Printf("1th case is selected. e10=%v.\n", e10)
}
```

无论default case被放置在哪儿，都不会影响到我们在上面说明的分支选择规则。因此，这段代码被执行后，下面的内容依然会被打印到标准输出上。

```
1th case is selected. e10=1.
```

3. 更多惯用法

我们已经知道，接收操作符<-可以从一个通道中接收一个元素值，并可以通过与=或:=联接把该元素值赋给一个或两个变量。如果同时对两个变量赋值，那么第二个变量便会指明当前通道是否已被关闭。在case关键字右边的接收语句当然也支持这种赋值形式。请看下面的代码：

```
ch11 := make(chan int, 1000)
// 省略若干条语句
select {
case e, ok := <-ch11:
    if !ok {
        fmt.Println("End.")
        break
    }
    fmt.Printf("%d\n", e)
}
```

依据接收表达式的第二个结果值判断通道的关闭状态的这种方式我们应该已经很熟悉了。如果发现通道已被关闭，我们就打印提示内容并退出当前的select语句。其中的break语句的作用就是立即使当前select语句结束执行，无论当前的case中是否还存在未被执行的语句。

在真正的应用程序中，我们常常需要把select语句放到一个单独的Goroutine中去。现在，我们专门启用一个Goroutine来执行上面这条select语句：

```
go func() {
    select {
    case e, ok := <-ch11:
        if !ok {
            fmt.Println("End.")
            break
        }
        fmt.Printf("%d\n", e)
    }
}()
```

这样可以完全避免因select语句的执行而可能导致的死锁现象。我们再来关注这条select中的接收操作。由于select语句在被执行的时候只会对其中的某一个通道操作一次，所以上面这段代码最多只能从ch11通道中接收到一个元素值。为了连续地接收元素值，我们应该把select语句包含在一条for语句中，如下所示：

```
go func() {
    for {
        select {
        case e, ok := <-ch11:
            if !ok {
```

```

        fmt.Println("End.")
        break
    } else {
        fmt.Printf("%d\n", e)
    }
}
}
}()

```

上面这条for语句不包含任何子句。这就意味着被它包裹的select语句会被一次接一次地执行。当通道ch11中未包含任何元素值的时候，当前Goroutine会被阻塞在这条select语句上。直觉上，由于有了变量ok，我们应该可以在ch11通道被关闭后及时退出这条for语句。但是，当我们真正执行这条for语句的时候，标准输出上可能会出现这样的内容：

```

.
.
.
47
48
49
End.
End.
End.
End.
.
.
.

```

我们用原点“.”表示还有很多内容没有在这里展示。显然，在我们关闭ch11通道之后，那个for并没有被结束。break语句只是让当前的select语句结束执行而已。由于它的外层还有for语句，所以流程控制权马上又会重新由select语句掌管，如此往复。最终，这个Goroutine会被一直运行下去而无法结束，直到当前程序被终结。看来我们需要两条break语句结束这个循环。但是，对于被用来退出外层for循环的break语句来说，必须要有一个标志来触发它的执行。代码如下：

```

go func() {
    var e int
    ok := true
    for {
        select {
        case e, ok = <-ch11:
            if !ok {
                fmt.Println("End.")
                break
            } else {
                fmt.Printf("%d\n", e)
            }
        }
        if !ok {
            break
        }
    }
}()

```

可以看到，我们在for语句的前面先声明了两个变量。它们分别被用来存放通道ch11中的元素值和关闭标志。对于第二条break语句来说，仅有变量ok是必要的。那么我们为什么还要声明变量e呢？请注意，上面代码中的那个唯一的case后面，接收语句已经变成了这样：

```
e, ok = <-ch11 // 在前一段示例代码中，它是这样的：e, ok := <-ch11
```

还记得吗？特殊标记:=和=代表着完全不同的赋值方式。前者会声明变量并给变量赋值，而后者则只能给已经声明的变量赋值。如果我们不事先声明变量e，那么这里被用来赋值的特殊标记只能依然是:=。这意味着，运行时系统会把它左边的e和ok都当作代表新变量的标识符来看待。

我们在这里再复习一下之前讲过的概念。如果在当前的代码块中已经存在了变量e，那么使用:=声明在内层代码块中的变量e就会遮蔽外层的同名变量。即使这两个同名变量的类型不同也会是这样。当然，在内层代码块中的那个变量的作用域之外并不存在这种遮蔽现象。如果这两个同名变量的作用域是完全相同的，那么自后面的变量被声明的那时起，之前声明的那个同名变量就会被永远地遮蔽。这种对已有标识符的复用也被称为变量的重声明。我们在第3章讲变量的时候对此做过介绍。

现在回到正题。按照我们的意愿，case右边的ok应该指代在for语句前面声明的那个变量，而不应该是一个新的变量。因为如果是新的变量，我们在case代码块中对变量ok的赋值将不会影响到该代码块之外的那个ok。其原因是，以case代码块为界，ok所指代的变量是不同的。在这种情况下，作用域更大的那个ok变量的值会一直是false。也就是说，for循环仍然永远不会退出。这就是我们在for语句的前面先声明变量e，并在case后使用=为它和ok赋值的根本原因。

在我们像前面那样使用两条break语句并正确地对ok变量进行声明和赋值之后，for语句的执行就可以在通道ch11被关闭之后立即结束了。其打印出的内容如下所示：

```
.
.
.
47
48
49
End.
```

在有些时候，我们并不想等到通道被关闭之后再退出循环，因为对于一些流程来说那样就太迟了。但是，针对通道的接收操作（以及发送操作）并不没有超时这一概念。所以我们只能使用某些辅助手段来达到此目的。

我们已经熟知了针对缓冲通道的各个操作的所有特性。那么我们是否可以利用它们的特性来满足这个需求呢？我们可以创建并初始化一个辅助的通道，并利用它模拟出操作超时的行为。请看下面的代码：

```
timeout := make(chan bool, 1)
go func() {
    time.Sleep(time.Millisecond)
    timeout <- false
}()
```

我们声明并初始化了通道`timeout`，并把它作为超时触发器使用。之所以`timeout`的容量为1，是因为在超时的触发和实施方面都没有并发的需要。另外，使用单独的Goroutine来进行触发超时的操作是理所应当的，只有这样才不会影响主流程的执行。这个Goroutine在被运行之后，会先延迟1毫秒的时间，然后触发超时。这里所说的触发超时实际上就是给`timeout`通道发送一个元素值。那么，这个元素值是怎样让超时得以实施的呢？请看下面的代码：

```
go func() {
    var e int
    ok := true
    for {
        select {
        case e, ok = <-ch11:
            if !ok {
                fmt.Println("End.")
                break
            } else {
                fmt.Printf("%d\n", e)
            }
        case ok = <-timeout:
            fmt.Println("Timeout.")
            break
        }
        if !ok {
            break
        }
    }
}()
```

这段代码由实现前一个需求的代码修改而来。我们只是在`select`语句中新增了一个`case`。这个`case`的作用就是接收“超时信号”并执行超时子流程。在`for`循环进行期间，一旦通道`timeout`中有了新的元素值，第二个`case`几乎马上就会被执行。因为前一段代码会向`timeout`通道发送`false`，所以我们可以在这里直接将这个值赋给变量`ok`以表明超时已被触发、当前流程应被中断。该`case`会使用`break`语句结束当前的`select`语句的执行。又由于这时的变量`ok`已被赋值为`false`，所以这条`for`语句也会被结束执行。

至此，操作超时已经被成功的模拟出来了，它会运作良好的。在这行这段代码之后，标准输出上会出现这样的内容：

```
.
.
.
47
48
49
Timeout.
```

有些读者可能会说这模拟的不是真正的操作超时。是的，我在这里只是提供了一种可以灵活地让这个元素值接收流程结束的方法。确切地讲，这应该叫作流程执行超时。那么，对于每一个接收操作而言，实施单个操作的超时是否真的可行呢？

答案当然是肯定的。最简单的方式就是在每次迭代的开始都初始化一次timeout并在一个单独的Goroutine中布置好延时和“超时信号”的触发，就像下面这样：

```
go func() {
    // 省略若干条语句
    for {
        timeout = make(chan bool, 1)
        go func() {
            time.Sleep(time.Millisecond)
            timeout <- false
        }()
        select {
            // 省略若干条语句
        }
        // 省略若干条语句
    }
}()
```

这的确可以实现单个操作的超时。但是那个超时触发器开始计时的时间是不是有点儿早呢？如果这样是不是更好呢：

```
go func() {
    var e int
    ok := true
    for {
        select {
            case e, ok = <-ch11:
                // 省略若干条语句
            case ok = <-func() chan bool {
                timeout := make(chan bool, 1)
                go func() {
                    time.Sleep(time.Millisecond)
                    timeout <- false
                }()
                return timeout
            }():
                fmt.Println("Timeout.")
                break
        }
        if !ok {
            break
        }
    }
}()
```

在上面这段代码中，select语句的第二个case后的接收语句是关键。注意，这条接收语句中的通道表达式是由一个针对匿名函数的调用表达式（在这个case中的接收操作符<-和冒号“:”之间的那一段代码）代表的。该匿名函数使用在其中声明并初始化的timeout通道作为它的结果，并作为当前case语句中的接收语句的组成部分。在经过大约1毫秒的时间后，该接收语句会从timeout通道接收到一个元素值并把它赋给变量ok，从而恰当地执行了针对单个操作的超时子流程，这会适时地使当前的for语句被结束执行。

在这一实现中，我们利用了select语句的两个特性。

- 在运行时系统开始执行select语句的时候，会先对它所有的case中的元素表达式和通道表达式进行求值。这样才使得在运行时系统选择要执行的case之前先制造出一个可用的超时触发器成为了可能。更具体地讲，在这些case被选择之前，第二个case后的接收语句会由下面这行代码替代：

```
ok = <-timeout
```

这与前面的版本如出一辙。只不过这里的timeout通道是在每次开始执行select语句的时候才被声明并初始化出来的。

- 运行时系统在选择select语句的case的时候，只要case有多个，它就肯定不会为某一个case而等待。只有当所有的case后的发送语句或接收语句都无法被立即执行的时候，它才会阻塞住当前的Goroutine。当然，前提是没有default case。在等待期间，只要发现有某一个case后的语句可以被立即执行，那么运行时系统就会立即执行这个case。在本例中，当无法立即从ch11通道中接收元素值的时候，运行时系统会随即判断是否可以立即接收timeout通道中的元素值。因此，一旦第一个case中的接收操作无法在1毫秒之内完成，我们给定的超时子流程就会被执行。

通过上面这一系列的示例和讲解，读者应该对select语句的编写、执行和惯用法都有所了解。注意，这其中的（以及之前的）很多内容都是只针对缓冲通道的。与非缓冲通道相关的各种使用方法和技巧，我们将会在下小节予以揭晓。

7.2.5 非缓冲的Channel

如果我们在初始化一个通道的时候将其容量设置成0，或者直接忽略对容量的设置，那么就会使该通道称为一个非缓冲通道。与以异步的方式传递元素值的缓冲通道不同，非缓冲通道只能同步的传递我们发送给它的元素值。

1. Happens before

非缓冲通道这种同步传递元素值的特性是怎样实现的呢？这还要从与它有关的“happens before”原则说起。与缓冲通道相比，针对非缓冲通道的“happens before”原则有3个特别之处，具体如下。

- 向此类通道发送元素值的操作会被阻塞，直到至少有一个针对该通道的接收操作开始进行为止。
- 从此类通道接收元素值的操作会被阻塞，直到至少有一个针对该通道的发送操作开始进行为止。
- 针对非缓冲通道的接收操作会在与之相对应的发送操作完成之前完成。

前两条规则保证了只有在针对同一通道的发送操作和接收操作都已经开始进行的时候，通过该通道的元素值传递才能够真正的开始。这也是“同步的传递”的真正含义。这两类操作哪一个先进行并不重要，重要的是它们要有配对的机会。只有这样传递动作才能有机会执行。

在进行针对非缓冲通道的发送操作之时，运行时系统会检查是否有针对同一个通道的接收操作正在进行。如果有，那么该接收操作必定正在被阻塞。这时，上述发送操作会唤醒第一个为此而被阻塞的接收操作，并与之配合完成一次元素值的传递动作。反之亦然。

对于第三条规则，我们需要特别注意。它是使非缓冲通道表现得与缓冲通道截然不同的最重要的一点。在缓冲通道中，由于元素值的传递是异步的，所以发送操作在成功向通道发送元素值之后就会立即结束。它不会关心是否有接收操作要接收该元素值，更别提该元素值是否已被成功接收的这个结果了。然而，针对非缓冲通道的操作在这方面的表现正好相反。发送操作在向非缓冲通道发送元素值的时候，会等待能够接收该元素值的那个接收操作。并且，只有确保该元素值被成功接收，它才会真正的完成执行。这进一步印证了非缓冲通道总会进行“同步的传递”的这一特性。

我们可以利用非缓冲通道的这种特性，实现多个Goroutine之间的同步。请看下面的示例：

```
func main() {
    unbufChan := make(chan int)
    go func() {
        fmt.Println("Sleep a second...")
        time.Sleep(time.Second)
        num := <-unbufChan
        fmt.Printf("Received a integer %d.\n", num)
    }()
    num := 1
    fmt.Printf("Send integer %d...\n", num)
    unbufChan <- num
    fmt.Println("Done.")
}
```

这段代码中共有4条打印语句。它们会以怎样的顺序打印相应的内容呢？我们让在主Goroutine中启用的那个Goroutine在运行之初先“睡”上1秒钟。原因是我们想看看针对unbufChan通道的发送操作是否真的会等待一个能够与之配对的接收操作。如果答案是否定的，那么很可能还没等那个新被启用的Goroutine真正开始运行，整个程序就已经被运行结束了。

在执行了该main函数之后，我们发现标准输出上的内容是这样的：

```
Send integer 1...
Sleep a second...
Received a integer 1.
Done.
```

这4行内容实际上体现了Goroutine和非缓冲通道的几个重要特性。先看前两行。第一行内容所对应的打印语句出现在go语句之后，但不是紧挨着这条go语句的那条语句。而第二行内容所对应的则是新启用的Goroutine中的第一条语句。请注意它们的位置关系。这说明新的Goroutine从启用到运行是需要一定的时间的。虽然这个时间很短暂，但是它也足够运行时系统执行主Goroutine中的好几条语句的了。在第二行内容被打印出来之后，运行时系统会继续运行主Goroutine中剩下的语句。如果针对unbufChan通道的发送操作不会被阻塞，那么运行时系统会在执行完这最后两条语句之后直接结束当前程序的运行。如果是这样的话，第三行甚至第二行的内容就应该是

Done.。但是，实际情况并不是这样。这恰恰说明了主Goroutine中的发送操作在等待一个能够与之配对的接收操作。当那个新的Goroutine中的接收操作开始进行的时候，由于配对成功，元素值1才得以经由unbufChan通道被从主Goroutine传递至那个新的Goroutine。

我们再来看最后两行打印内容。如果unbufChan通道是带缓冲的，那么第四行内容Done.一定会先被打印出来。至于原因，我们在前面已经说明过了。但是，对于这里的非缓冲通道unbufChan来说，由于发送操作一定会等到相对应的接收操作完成之后才完成，所以这两行打印内容才会以这样的顺序展现出来。

通过前面这个示例，我们应该已经对非缓冲通道的“happens before”原则有真正的理解了。请记住它在这方面与缓冲通道的不同。

2. 单向的非缓冲通道

单向的非缓冲通道在表现形式上与我们在7.2.2节中讲述的单向通道并没有什么不同。因为在通道的类型字面量中并不会体现出通道是否带有缓冲。不过，正因为这一点，我们才更应该关注作为参数值被传入的单向通道的特性。请看下面的示例：

```
func fetchPerson(origs chan<- Person) {
    origsCap := cap(origs)
    buffered := origsCap > 0
    goTicketTotal := origsCap / 2
    goTicket := initGoTicket(goTicketTotal)
    go func() {
        for {
            p, ok := fetchPerson1()
            if !ok {
                for {
                    if !buffered || len(goTicket) == goTicketTotal {
                        break
                    }
                    time.Sleep(time.Nanosecond)
                }
                fmt.Println("All the information has been fetched.")
                close(origs)
                break
            }
            if buffered {
                <-goTicket
                go func() {
                    origs <- p
                    goTicket <- 1
                }()
            } else {
                origs <- p
            }
        }
    }()
}
```

上面展示了我们在7.2节中提及但并未展现的fetchPerson函数的实现。这个函数的功能是从某处取出欲处理的人员信息，并把它们发送给origs通道。我们在这里并不关心从哪里取出这些

人员信息，所以这里用`fetchPerson1`函数代表人员信息取出操作。为了尽快地将欲处理的人员信息发送给`origs`通道，我们试图启用更多的Goroutine来并发地进行发送操作。顺便提一句，在无法确定这种并发是否会给程序性能带来正面影响的时候，我们常常会先以串行的方式进行相应的操作。

我们怎样确定并发的进行发送操作是否会给程序性能带来正面影响呢？请注意在`fetchPerson`函数体的开始处声明的那个变量`buffered`。它的值表示了通道`origs`是否带有缓冲。如果`buffered`变量的值是`false`，那么我们就没有必要并发地发送取出的人员信息。其根本原因是，非缓冲通道只能同步地传递元素值。在接收操作完成之前，发送操作是无法完成的。即使我们采用并发的方式发送人员信息，这些信息也只会手递手地传送到处理它们的Goroutine中。对于非缓冲通道来讲，在接收方并没有并发地进行接收操作的时候，发送方并发地进行发送操作是没有任何意义的。这种方式反而会降低程序的性能，并给运行时系统带来无谓的负担。更进一步地说，在收发两端都有并发需求的情况下，使用非缓冲通道作为元素值传输介质是不合适的。除非双方都有着非常强烈的同步传递的需要。但是，在一般情况下，这两种需求是相悖的。

我们再来关注上述`fetchPerson`函数中其他值得说明的代码。在我们启用新的Goroutine以向`origs`通道发送人员信息的时候，还涉及了对`goTicket`通道的接收操作和发送操作。那么，`goTicket`通道又是做什么用的呢？`goTicket`通道实际上是我们为了限制该程序启用的Goroutine的数量而声明的一个缓冲通道。由于这种用法不在本小节的讨论范围之内，所以我们在这里只做简要的介绍。在初始化`goTicket`通道的时候，我们会向它发送与其容量相等的元素值。该通道的容量（在这里与它的长度相等）代表了我们可以启用的Goroutine的数量。每当我们启用一个Goroutine的时候，就从该通道中接收一个元素值，以表示可被启用的Goroutine减少了一个。相应地，每当一个被启用的Goroutine的运行即将结束的时候，我们就应该向该通道发送一个元素值，以表示可被启用的Goroutine增加了一个。当相应的Goroutine的数量已经与该通道的容量相等（即该通道中已不存在任何元素值）的时候，新的启用Goroutine的动作（确切地说，是紧挨在`go`语句之前的针对该通道的接收操作）就会被阻塞住。仅当该通道中又存在新的元素值（即之前启用的一些Goroutine即将被运行结束）的时候，新的启用Goroutine的动作才会得以继续。这是使用缓冲通道作为Goroutine票池的典型做法。这里所说的Goroutine票，其实代表了为了启用一个Goroutine而必须持有的一种令牌。我们使用这样的令牌来限制程序启用Goroutine的数量。

另一方面，既然我们需要并发的向`origs`通道人员信息，那么就应该在保证安全的情况下再关闭`origs`通道。也就是说，我们应该在发现已无可取的人员信息之后，检查被启用的相关Goroutine是否都已运行完毕。检查的具体方法就是查看`goTicket`通道长度是否与其容量相等。如果相等，那么就说明`goTicket`中的令牌都已被放回，所有相关的Goroutine都已经运行完毕。只有在确定这一点之后，我们才应该去关闭`origs`通道。不过，如果`origs`通道是非缓冲的（即变量`buffered`的值为`false`），那么我们就没必要做上述检查了。具体请参看`fetchPerson`函数中那条处于内层的`for`语句。

综上所述，单向的非缓冲通道的重点不在单向而依然在非缓冲。在函数或方法接受外部传来的通道类型的参数值的时候，应该先验明它的种类再依此来决定后续的操作。也正因为通道的这种特性不会被反映到类型上，所以我们只能使用`cap`函数作为辅助。

3. for语句与非缓冲通道

使用for语句接收通道中的元素值的时候，并不需要关注通道是否带有缓冲区。因为这种用法仅涉及接收操作。无论是非缓冲通道还是缓冲通道，对它们的接收操作都会被阻塞，直至通道中有元素值可接收为止。唯一的区别是，针对非缓冲通道的接收操作会在相应的发送操作完成之前完成，而这对于缓冲通道来说恰恰相反。

4. select语句与非缓冲通道

在使用select语句向某个非缓冲通道发送元素值的时候，我们应该特别注意。因为，与操作缓冲通道的select语句相比，它被阻塞的概率一般会大很多。其根本原因依然是非缓冲通道会以同步的方式传递元素值。如果运行时系统在该条select语句中选择要执行的case的时候不存在正在为此而等待的可配对操作，相应的case就肯定不会被选中。因为这样的case无法被立即执行。例如：

```
unbufChan := make(chan int)
select {
case unbufChan <- i:
case unbufChan <- i + 10:
}
```

在执行上面的select语句的时候，它所在的Goroutine会被阻塞，直到在其他Goroutine中进行了对应的接收操作，如：

```
<-unbufChan
```

记住，只有存在可配对的操作的时候，传递元素值的动作才可能真正开始。当然，我们为这样的select语句添加default case可以使它不被阻塞。但是这样并不会减小其中的case不被选中的概率。相比之下，这里的default case可能会被经常选中并执行。我们下面来看一个比较完整的示例：

```
func main() {
    unbufChan := make(chan int)
    sign := make(chan byte, 2)
    go func() {
        for i := 0; i < 10; i++ {
            select {
            case unbufChan <- i:
            case unbufChan <- i + 10:
            default:
                fmt.Println("default!")
            }
            time.Sleep(time.Second)
        }
        close(unbufChan)
        fmt.Println("The channel is closed.")
        sign <- 0
    }()
    go func() {
    loop:
        for {
```

```

        select {
        case e, ok := <-unbufChan:
            if !ok {
                fmt.Println("Closed channel.")
                break loop
            }
            fmt.Printf("e: %d\n", e)
            time.Sleep(2 * time.Second)
        }
    }
    sign <- 1
}()
<-sign
<-sign
}

```

在上面的main函数中，我们启用了两个Goroutine分别对非缓冲通道unbufChan进行发送和接收的操作。发送操作会被进行10次，间隔时间为1秒。接收操作会一直被尝试，尝试的间隔至少为2秒。显然，收发操作可配对的情况每2秒会出现一次。执行main函数之后，打印内容如下：

```

default!
e: 11
default!
e: 13
default!
e: 5
default!
e: 17
default!
e: 9
The channel is closed.
Closed channel.

```

正如我们所料，default case会在收发操作无法配对的情况下被选中并执行。在这里，它被选中的概率是50%。

除此之外，我们故意在第一条select语句的两个case中分别向unbufChan通道发送小于10和大于等于10的整数。这使得我们可以很容易地从打印内容中分辨出每当收发操作可配对的时候哪一个case被选中了。实际上，这与在select语句中向缓冲通道发送元素值的情形是一样的。当可配对的情况出现的时候，这两个case是被随机选择的。运行时系统会先检查都有哪些case可以被立即执行，然后在通过伪随机算法选择它们中的一个。

总之，非缓冲通道是无法缓冲任何元素值的。因此，针对它们的收发操作能否被立即执行完全取决于当时是否有可配对的操作。我们在select语句中使用非缓冲通道的时候应该把这种特性作为重要的参考。例如，上面的这个示例给予了我们这样一个启发：使用非缓冲通道能够让我们非常方便地在接收端对发送端的操作频率实施控制。读者可以试着把上面main函数中的第一条select语句的default case去掉并再次执行main函数。看看它在执行效果上（注意各行内容被打印的时间以及间隔）与之前有什么不同，然后尝试解释这种不同。

非缓冲通道最重要的特性都体现在与之相关的“happens before”原则中。读者应该记住我们

在本小节强调的那些与缓冲通道的“happens before”原则不同的部分。在for和select语句中，使用非缓冲通道与使用缓冲通道在形式上面并没有什么不同。但是，需要注意的是，非缓冲通道的特性让我们在使用它们的时候不得不留意操作配对以及由此带来的一些问题。

再次强调，在发送端每次都需要确保元素值已被接收的情况下，使用非缓冲通道是适合的。否则，我们应该选用缓冲通道来实现相关流程的异步化，并以此提高整个程序的性能。

7.2.6 time包与Channel

本小节主要讲解Go语言的标准库代码包time中的一些API的使用。为什么要在这里讲它们？因为，它们都是用通道来实现的。并且，它们还能够帮助我们对针对通道的收发操作进行更有效的控制。

1. 定时器

首先，我们先来看看time包中的结构体类型Timer。顾名思义，该类型的结构体会被作为定时器使用。我们不应该直接使用复合字面量来初始化该类型的变量。因为time.Timer类型中包含了一个包级私有的字段。并且，该字段是需要被显式地初始化的。这使得我们无法使用复合字面量正确的初始化一个time.Timer类型的值。

在time包中，有两个函数可以帮助我们初始化time.Timer类型的值。它们是time.NewTimer函数和time.AfterFunc函数。

函数time.NewTimer的使用非常简单。我们在调用它的时候只传给它一个time.Duration类型的值就可以了。该函数的唯一参数的含义是，自定时器被初始化的那一刻起，距其到期时间需要多少纳秒。虽然这里的单位是纳秒，但是我们可以很方便地拼出我们需要的时间。因为time包已经包含了很多我们常会用到的time.Duration类型的常量。比如，我们可以这样表示3小时36分钟：

```
3*time.Hour + 36*time.Minute
```

如果我们要声明并初始化一个到期时间距此时的间隔为3小时36分钟的定时器的话，就应该这样编写代码：

```
t := time.NewTimer(3*time.Hour + 36*time.Minute)
```

注意，这里的变量t是*time.Timer类型的，而不是time.Timer类型的。这个time.Timer类型的指针类型的方法集合包含了两个方法，即Reset方法和Stop方法。Reset方法被用于重置定时器。也就是说，我们初始化的定时器是可以被复用的。该方法会返回一个bool类型的值。Stop方法被用来停止定时器。该方法也会使用一个bool类型值作为它的结果。如果该结果为false，就说明该定时器在之前已经到期或者已经被停止了。除了这两种情况，该结果都应该为true。Reset方法对其结果值的设定策略与Stop方法如出一辙。不过，Reset方法的返回值与当次重置操作是否成功无关。换句话说，无论怎样，一旦Reset方法被执行结束，就说明该定时器已被重置。与此不同，如果Stop方法的返回值是false，那么就说明此次调用是无效的，即它并没有对定时器现有的状态产生任何改变。

我们刚刚说到了定时器的到期。那么这个到期的事件是怎样被传达的呢？实际上，这个传达

的途径就是通道，即`time.NewTimer`类型的字段`C`。`C`是一个`chan time.Time`类型的缓冲通道。这意味着，一旦触及到期时间，定时器就会向自己的`C`字段发送一个`time.Time`类型的元素值。这个元素值代表了该定时器的绝对到期时间。与之对应，我们在调用`time.NewTimer`函数时传入的那个`time.Duration`类型值就是该定时器的相对到期时间。这两者之间的关系一定是：

〈初始化时的绝对时间〉 + 〈相对到期时间〉 == 〈绝对到期时间〉

通过定时器的字段`C`，我们可以及时得知定时器到期的这个事件的来临，并对此做出响应。例如，有这样一段代码：

```
t := time.NewTimer(2 * time.Second)
now := time.Now()
fmt.Printf("Now time: %v.\n", now)
expire := <-t.C
fmt.Printf("Expiration time: %v.\n", expire)
```

它被执行之后，标准输出上会被打印出如下内容：

```
Now time: 2014-04-01 16:00:10.2329909 +0800 +0800.
Expiration time: 2014-04-01 16:00:12.2331053 +0800 +0800.
```

可以看到，即使我们在初始化定时器之后马上获取了当前时间，它与定时器被初始化的时间还是有少许偏差的。这个偏差在作者的机器上是微秒级的。不过，这个示例也已经能够证实前述的那3个时间的关系了。

当然，在实际的场景中，我们不会如此简单地使用定时器。现在，我们就来看看它的典型使用方式。还记得我们在7.2.4节的最后实现的那个超时触发器吗？它实际上就是一个简易版本的定时器。现在我们就用官方的定时器来替换我们自己编写的那个版本。不过，为了方便对比，我们先重现一下之前的简易版本定时器以及相关的`case`：

```
case ok = <-func() chan bool {
    timeout := make(chan bool, 1)
    go func() {
        time.Sleep(time.Millisecond)
        timeout <- false
    }()
    return timeout
}():
    fmt.Println("Timeout.")
    break
```

下面，是使用官方的定时器的版本：

```
case <-time.NewTimer(time.Millisecond).C:
    fmt.Println("Timeout.")
    ok = false
    break
```

在这两个版本中，`case`之后的通道表达式的作用是类似的。它们都是先声明并初始化一个通道，做好定时向该通道发送元素值的设定，最后将这个通道作为其结果值。不过，后者显然让我们省去了很多的代码。

其实，我们在上面展示的表达式：

```
time.NewTimer(time.Millisecond).C
```

与我们调用time包的After函数的效果相当。如用time.After函数对此进行等价替换的话，那个通道表达式会是这样：

```
time.After(time.Millisecond)
```

实际上，time.After函数的实现代码正如前者。它只能算是一个为我们屏蔽了time.Timer类型的内部实现的快捷函数而已。

正如前文所述，定时器是可以被复用的。因此，像前面的示例那样在case后的接收语句中初始化定时器难免有些浪费。下面的使用定时器的方式会更加节省资源：

```
go func() {
    // 省略若干条语句
    var timer *time.Timer
    for {
        select {
            // 省略若干条语句
            case <-func() <-chan time.Time {
                if timer == nil {
                    timer = time.NewTimer(time.Millisecond)
                } else {
                    timer.Reset(time.Millisecond)
                }
                return timer.C
            }
        }
        // 省略若干条语句
    }
}()
```

可以看到，我们在for语句的前面先声明了一个*time.Timer类型的值，然后在相关case之后声明的匿名函数中尽可能地复用它。虽然这比每次执行select语句时都初始化一个定时器的那种方式要啰嗦一些，但是在for循环的迭代次数很多的情况下，这样做还是很有必要的。虽然这比最初的版本使用的代码还要多一点点，但是它绝对不会浪费资源。

定时器永远不需要向它的C字段发送第二个元素值。这是因为，定时器一旦到期就意味着在我们重置它之前它无法被再次使用。正因为如此，C的容量仅为1就足够了。注意，如果我们在定时器到期之前停止了它，那么该定时器的字段C也就没有机会缓冲任何元素值了。更具体地讲，若调用定时器的Stop方法的结果值为true，那么在这之后再去试图从它的C字段中接收元素是不会有结果的。更重要的是，这样做还会使当前Goroutine永远被阻塞！再次强调，重置已被停止的定时器是使它恢复如初的唯一方法。

另一方面，如果定时器到期了，但由于某种原因我们未能及时从它的字段C中接收那个元素值，那么就意味着我们错过了处理到期事件的时机。不过，在一些场景下，这种情况常常会发生，

同时它也是正常的。前面的示例展示的就是这样的场景。定时器到期但未被处理就意味着其他case被执行了。这正是我们所期望的。但是，在这种情况下。定时器的字段C中会一直缓冲着那个唯一的元素值，即使在该定时器被重置之后也会是这样。因此，我们总是应该及时从定时器的字段C中接收元素值，否则会影响对定时器的复用。

另外，还有一点需要注意，那就是：我们在设定定时器的相对到期时间的时候，一定要让这个纳秒数为一个正整数。否则，定时器在被初始化或重置之时就会立即到期。在这之后，当我们试图从它的字段C接收元素值的时候，就会立即得到它而不会有任何延时。显然，这样的定时器就失去了存在的意义了。

除了上述讲到的API之外，time包还为我们提供了定时器的另一种使用方式。这种使用方式使得定时器在到期事件到来之时不向其C字段发送代表绝对到期时间的元素值，而是执行我们指定的函数。如果说前者是定时器默认的处理到期事件的方法的话，那么后者就可以被称作自定义方法。示例代码如下：

```
var t *time.Timer
f := func() {
    fmt.Printf("Expiration time: %v.\n", time.Now())
    fmt.Printf("C's len: %d\n", len(t.C))
}
t = time.AfterFunc(1*time.Second, f)
time.Sleep(2 * time.Second)
```

我们通过调用time.AfterFunc函数初始化一个定时器。time.AfterFunc函数需要两个参数。第一个参数是定时器的相对到期时间，而第二个参数就是一个我们自定义的函数。该函数可以由任何无参数声明且无结果声明的函数或方法表示。在这样的函数或方法中，我们可以做的事情就很多了。这显然比先尝试从定时器的C字段中接收元素值然后再为此做出响应要方便得多。在执行上面这段代码之后，如下内容会被打印到标准输出：

```
Expiration time: 2014-04-02 11:46:27.9564585 +0800 +0800.
C's len: 0
```

我们在这段代码的最后让当前Goroutine“睡眠”了2秒钟以确保打印内容的完整。这样做的的原因是，time.AfterFunc的调用不会被阻塞。它会以异步的方式在到期事件来临的那一刻执行我们自定义的函数。正如第二行内容显示的那样，定时器的字段C并没有缓冲任何元素值。这也说明了，在给定了自定义函数之后，默认的处理方法（向C发送代表绝对到期时间的元素值）就不会被执行了。除了这个特点之外，我们操纵使用time.AfterFunc函数初始化的定时器并不会有什么特别。它的Reset方法和Stop方法的行为及其结果值的设定策略都不会因此而改变。

我们常常把定时器作为超时触发器使用。从某种角度看，它的作用与time.Sleep函数有些类似。但是，定时器的一个明显的优势是，由于它使用缓冲通道作为告知到期事件的途径，使得我们可以异步地使用它。虽然使用time.Sleep函数和一些附加代码也可以做到这一点，但是这样显然会导致更多的代码和资源的浪费。正如我们在前面的示例中展现的那样。另外，通过time.AfterFunc函数初始化定时器，可以让我们更加灵活地使用它。这种初始化方式适合被用来执行定时任务。只要我们把它与定时器的Reset方法结合起来使用，就可以满足每隔一段时间执

行一次任务的需求了。

2. 断续器

结构体类型`time.Ticker`表示了断续器的静态结构。所谓断续器，就是周期性的传达到期事件的装置。这种装置的行为方式与仅有秒针的钟表有些类似，只不过断续器的传动间隔时间可以不是1秒钟。另外，断续器与定时器也是相似的。但是，定时器在被重置之前只会传达一次到期事件，而断续器会持续工作直到被停止。

断续器传达到期事件的途径也是它的字段`c`，而`c`的类型也是元素类型为`time.Time`的通道类型。每隔一个指定的时间，断续器就会向此通道发送一个代表了当次的绝对到期时间的元素值。值得注意的是，断续器的字段`c`的容量依然是1。如果断续器在向它的字段`c`发送新的元素值的时候发现旧值还未被接收，那么就会取消当次的发送操作。这相当于丢弃了当次的到期事件而不予以传达。更确切地说，后续的发送操作的成功与否都由断续器的字段`c`中的可用缓冲空间来决定。

现在，我们来看看怎样初始化一个断续器，代码如下：

```
var ticker *time.Ticker = time.NewTicker(time.Second)
```

初始化断续器的函数的名称依然以单词“New”开始。`time.NewTicker`函数接受一个`time.Duration`类型的参数。该参数代表了我们前面所说的周期，单位是纳秒。上面这段代码初始化了一个传达到期事件的间隔时间为1秒的断续器，并把它赋给了一个`*time.Ticker`类型的变量。该方法集合中只有一个方法，即`Stop`方法。它的功能是停止断续器。这与定时器的`Stop`方法的功能是完全相同的。一旦断续器被停止，它就不会再向其`c`字段发送任何元素值了。但如果当时此`c`字段中已经有了一个元素值，那么该元素值会一直在那里，直至被接收。

断续器与定时器的适用场景是完全不同的，把它当作超时触发器来使用是不适合的。因为它对到期事件的传达虽然可能被放弃，但绝不会被延误。也就是说，断续器一旦被初始化，它所有的到期时间就都是固定的了。而超时触发器则需要依据每次具体操作（初始化或重置）的开始时间来决定绝对的超时时间。这也是二者的重要区别之一。

固定不变的到期时间恰恰使断续器非常适合被作为定时任务的触发器。例如，在一个定时的执行数据修补任务的程序中，为了避免对其他正常的数据库操作产生影响，我们要求两次任务执行之间的最短间隔时间为10分钟。我们可以这样编写满足这一需求的代码：

```
var ticker *time.Ticker = time.NewTicker(10 * time.Minute)
ticks := ticker.C
// 省略若干条语句
go func() {
    for _ = range ticks {
        if !patch() {
            break
        }
        _, ok := <-ticks
        if !ok {
            break
        }
    }
}()
```

这段代码中的函数`patch`就代表了数据修补任务。无论执行该任务的耗时是怎样的，从前一次任务执行完成到后一次任务执行开始的间隔时间总会大于10分钟。因为我们在那个`for`循环中额外添加了一条接收语句。这条接收语句的执行会增加下一次迭代的执行延时。由于`for`语句在准备执行下一次迭代之前会等待两个到期事件，所以执行数据修补任务的间隔至少是一个周期（即10分钟）。

有时候，我们只需要断续器的`C`字段而不需要调用它的`Stop`函数。在这种情况下，我们就可以使用`time`包为我们提供的一个快捷函数来初始化一个断续器。这个快捷函数就是`time.Tick`函数。`time.Tick`函数接受一个代表到期事件传达周期的参数，并会返回刚被初始化完成的那个断续器的字段`C`作为结果。示例如下：

```
ticks := time.Tick(time.Second)
```

代码包`time`中的定时器（`time.Timer`）和断续器（`time.Ticker`）都充分利用了缓冲通道的异步特性来传达到期事件。我们可以使用它们对程序的流程进行更加灵活的控制。不过，它们对应着不同的流程控制策略。我们可以利用定时器设定某一个操作或任务的超时时间。这相当于对它们的完成时间点进行控制，而断续器常被我们用来设定操作或任务的开始时间点。从这个角度看，它们面向的是两个看似对立又相互关联的方面。通过对它们的组合使用，我们可以实现对时间敏感的流程的有效控制。

7.3 实战演练——载荷发生器

我们用上一章和本章的大部分篇幅讲解了当今主流的并发编程方式，以及Go语言并发编程模型的来龙去脉。现在，终于到了利用这些知识动手编写一个完整的、可以实际应用的并发程序的时候了！

作为经历过全周期的软件项目（尤其是互联网软件项目）的开发者而言，肯定不止一次地有过这样的需求：在开发完成一个可运行的软件并且通过基本的功能测试之后，我们会非常急迫地想获得这个软件的性能数值。换句话说，我们总是需要尽早地对软件进行性能评测。之所以有这样的需求，是因为我们在正式使用该软件之前往往需要搞清楚下面这几个问题。

- ❑ 这个软件到底能跑多快？
- ❑ 在高负载的情况下，该软件是否还能保持正确性。或者说，载荷数量与软件正确性之间的关系是怎样的？
- ❑ 在保证一定的正确性的条件下，该软件的可伸缩性是怎样的？
- ❑ 在软件可以正常工作的情况下，负载与系统资源（包括CPU、内存和各种I/O资源，等等）使用率之间的关系是怎样的？

只要为这些问题找到了答案，我们就能够真正地了解到软件的性能。也只有这样，我们才会知道需要进行怎样的软件设计，以及提供怎样的系统资源，才能够让它在承受一定量的载荷的同时保证正确性。这也是我们分析并定位软件的性能瓶颈所需的重要参考资料。其中，这个载荷的量应该是我们在对性能评测所得到的一系列数值进行统计和分析后得出的。通过对这些数值的掌

握，我们也可以了解到软件在给定的运行环境下的性能。而正确性的比率则应该体现（或者说满足）软件使用者（客户端软件的开发者或者终端用户）对软件的刚性需求。所谓刚性需求，就是关乎软件质量和使用者体验的硬性指标，是软件必须要满足的需求。

我们在本节将要编写的载荷发生器可以被作为软件性能评测的辅助工具。它可以向被测软件发送指定量的载荷，并记录下被测软件处理载荷的结果。这样，我们就可以统计出被测软件在给定的使用场景下的性能数值了。我们编写的载荷发生器应该具有优良的可控性和可扩展性，同时还应该能够输出内含丰富的结果。为了做到这几项，我们应该首先对它的输入、输出和基本结构进行一番设计。

7.3.1 参数和结果

一个程序的输入、输出以及二者之间的对应关系往往可以很好地体现出该程序的功能。在本小节，我们就从这方面着手对载荷发生器进行设计。

1. 重要的参数

为了编写这样一个工具，我们需要先了解一下几个必须且重要的参数。这些参数可以帮助我们营造出一个有利于软件性能评测的负载环境。

首先，一个软件在给定的运行环境下最多能够被多少个用户同时使用总是我们需要获悉的。这就是说，我们在进行性能测试的时候一般需要给定在同一时刻（或在某一个时间段内）向软件发送载荷的数量。在这一方面存在两个专业术语，它们是QPS（Query Per Second，每秒查询量）和TPS（Transactions Per Second，每秒事务处理量）。这两者都是体现服务器软件的性能的指标，其含义都是在一秒钟之内可以正常响应请求的数量的平均值。不同的是，前者针对的是对服务器上的数据的读取操作，而后者针对的则是对服务器上的数据的写入和修改操作。由于载荷的多样性，我们不打算在载荷发生器中区分这两个性能指标。但是，作为载荷发生器的使用者需要明确，在针对软件的某个种类的API进行测试的时候，得出的结果对应的应该是哪一个性能指标。

我们在这里可以把载荷和请求归为同一个事物。它们都代表了软件使用者为了获得某种结果而向为之服务的软件发送的一段数据。我们把每秒发送的载荷的数量（以下简称每秒载荷量）作为参数，其意义是控制载荷发生器向软件发送载荷（或称请求）的频率。这样，我们就可以控制被测软件在一段时间之内的负载情况了。

其次，软件在承受一定量的载荷的情况下对系统资源的消耗也是应该值得我们特别关注的。这与软件的可靠性息息相关。打个比方来说，有两个服务器软件A和B。经性能评测，A的QPS是2000，B的QPS是2200。但是由于B对系统资源的消耗较大以及对系统资源释放的不及时，导致其在接受每秒2000个载荷并持续了200个小时之后宕机了。但是A在接受同样的负载的情况下可以无故障地运行300小时。我们说，虽然B的部分性能数值更佳，但是其可靠性应该是比A差的。虽然实际的软件可靠性还需要通过一些专门的方法去衡量并且与软件的实时性能并没有直接的关系，但是我们还是应该积极地了解软件在持续接受一定量的载荷的情况下能够无差错地运行多久（或者说它可以持续地为并发的请求服务多长时间）。通过明确的设定持续发送载荷的时间（也可

称为负载持续时间),我们就可以了解到这个时间段内软件性能的变化,同时也有机会使用一些方法获得软件对系统资源的使用情况,并以此推断出软件对各种系统资源的依赖情况以及它们与软件性能之间的关系。这也有助于我们查找软件内部可能存在的设计缺陷。

我们需要了解的第三个参数是评判软件正确性的一个重要标准。这个参数就是载荷的处理超时时间(以下简称处理超时时间),即我们可接受的从向软件发出请求到接收到软件返回的响应的最大耗时。设置处理超时时间可以让我们更加精确地计算出在给定载荷量以及持续时间的情况下软件的正确性比率。处理超时与软件处理载荷出错和响应内容错误一样,也表示了软件的不正确。同时,它也应该是我们关心的软件性能指标之一。它与载荷的量和持续时间之间都存在着一定的关联。例如,在我曾经所属的互联网软件开发团队中有这样一条硬性的软件系统性能要求:对于面向终端用户的所有API,其处理超时时间都是200毫秒。如果某个API在承受不高于最高载荷量的80%的负载的情况下造成了处理超时,那么这个API在性能上就是不合格的。这就强迫我们在各个方面(包括但不限于API设计、处理流程设计和数据缓存设计)都要仔细斟酌。比如,若某API持续承受高负载的时间比例过大(如一天中有12个小时连续承受着较高负载),那么我们就应该考虑该API的设计是否合理、软件系统是否需要再被拆分,甚至与之关联的其他系统是否存在问题。

我们在初始化载荷发生器的时候就应该给定上述3个参数,即每秒载荷量、负载持续时间和处理超时时间。载荷发生器应该根据这些参数自行计算出载荷发生以及发送的频率,并控制好并发量。

2. 输出的结果

载荷发生器的输出应该有助于我们统计、分析和汇总出软件在承受给定负载的情况下所表现出的各个性能数值,以及像软件可以承受的最大载荷量以及它可以持续承受一定载荷量的最长时间这样的极限值。据此,我们应该在针对某一个载荷的结果中至少包含三块内容,即请求和响应的内容、响应的状态以及请求(或者说载荷)处理耗时。其中,请求和响应的内容让我们可以精细地检查响应内容的正确性。响应的状态则应该反映出处理此请求的过程中的绝大多数问题,而不仅仅是成功或失败那么简单。至于请求处理耗时,则需要真实地体现从向软件发送请求到接收到软件的响应的精准耗时,并且不应该夹杂任何其他操作的进行时间。

对于每一个载荷所产生的结果来说,都应该至少包含上述3块内容。那么,载荷发生器的输出就应该是按照响应的到达顺序排列的一个结果列表。根据这些结果,我们就可以计算出软件每秒处理载荷的数量(以下简称每秒载荷处理量)。每秒载荷处理量一定小于或等于我们设定的每秒载荷量。软件在处理某些载荷的时候可能会出错、失败或超时。

7.3.2 基本结构

在做好足够的功课之后,我们就可以开始动手编写载荷发生器了。首先,根据前面的分析和设计,我们要确定载荷发生器的基本结构。我们应该用结构体类型声明来表示它的基本结构。在这个声明中,肯定应该包含我们在前面提到的那3个重要的参数:

```
timeoutNs    time.Duration    // 响应超时时间,单位:纳秒。
lps          uint32          // 每秒载荷发送量。
```

```
durationNs time.Duration // 负载持续时间，单位：纳秒。
```

其中两个表示时间的字段的类型均是`time.Duration`。这是为了方便设定超时和定时任务。而`lps`即是`Loads per second`的缩写。这沿用了`QPS`和`TPS`的命名规则。

我们在前面说过，负载发生器的输出应该是一个结果列表。但是，我们在这里不应该使用数组或切片作为收纳结果的容器。原因是，负载发生器是并发的发送载荷的。这意味着软件处理载荷的结果也会并发地被添加到列表中。我们已经知道，数组和切片本身都不是并发安全的。Go语言原生的数据类型中只有通道是并发安全的，并且可以同时由任意个Goroutine使用。它是收纳结果的绝佳容器。因此，我们将这样一个通道类型的字段也加入到载荷发生器的类型声明中：

```
resultCh chan *lib.CallResult // 调用结果通道。
```

其中，`lib.CallResult`是一个专为处理结果声明的数据类型。“Call”的意思是调用，它象征着我们对软件的API的调用。这个调用无所谓是本地的还是远程的。因此，我们也可以称针对单个载荷的处理结果为调用结果。调用结果的类型声明被放在了代码包`loadgen/lib`中。

这里需要特别说明一下，所有与载荷发生器有关的代码都会被放到`loadgen`代码包及其子包中。因此，为了使这些标识符看起来简短一些，我们在后面的内容中会省略掉在这些代码包中声明的程序实体的部分限定符。例如，类型`loadgen/lib.CallResult`会被简写成`lib.CallResult`。实际上，这也符合Go语言的限定标识符的书写规则。

回到正题，`lib.CallResult`类型的基本结构如下：

```
// 调用结果的结构。
type CallResult struct {
    Id      int64      // ID。
    Req     RawReq     // 原生请求。
    Resp    RawResp    // 原生响应。
    Code    ResultCode // 响应代码。
    Msg     string     // 结果成因的简述。
    Elapse  time.Duration // 耗时。
}
```

在`lib.CallResult`结构体的类型声明中，包含了我们前面所说的那3块内容。字段`Req`和`Resp`分别代表了请求内容和响应内容，字段`Code`和`Msg`被用来描述响应的状态，而字段`Elapse`则被用于表明请求处理耗时。最后，字段`Id`被用来标识和区分调用结果。

我们已经看到，在上述类型声明中又包含了两个自定义的类型，即`lib.RawReq`和`lib.RawResp`。它们的声明如下：

```
// 原生请求的结构。
type RawReq struct {
    Id      int64
    Req     []byte
}

// 原生响应的结构。
type RawResp struct {
    Id      int64
    Resp    []byte
}
```

```

    Err    error
    Elapse time.Duration
}

```

这两个类型的声明中也都包含了Id字段。对于同一个载荷而言，其相关的请求、响应和调用结果中的Id字段的值都应该是一致的。这对于我们了解处理载荷的全过程会很有帮助。

类型lib.RawReq的Req字段的作用是容纳原生请求的数据。我们知道，数据的最底层的表现形式就是一个或多个字节。因此，在这里，我们使用[]byte来作为Req字段的类型。相比之下，lib.RawResp类型中除了Id和代表原生响应数据的Resp字段之外，还包含了Err和Elapse字段。Err字段的值会体现在发送请求和接收响应的过程中（或者说在调用被测软件的API的过程中）发生的错误。如果没有发生错误，那么这个字段的值将会是nil。而Elapse字段则被用来表示这个过程的耗时，单位是纳秒。注意，lib.CallResult类型中的Elapse字段的含义与这是一致的。

现在，我们回到载荷发生器的基本结构上来。前面说到，它的resultCh字段的类型是chan *lib.CallResult。由于结构体类型的零值不是nil，所以如果这个通道的元素类型是lib.CallResult的话，就会给我们对其中的元素值的零值判断带来一些困扰。我们使用它的指针类型作为通道的元素类型，既可以消除这种困扰，也可以省去因元素值复制而带来的一些开销。

载荷发生器的类型目前已经拥有代表了必需的参数和被用来收纳调用结果的容器的4个字段。不过，这样显然是不够的。因为我们还需要一些内部字段来对整个过程进行控制，并且还要使载荷发生器具有良好的可扩展性。

我们为载荷发生器指定的响应超时时间和每秒载荷发送量，应该能够作为它生成和发送载荷的频率的依据。也就是说，我们应该可以根据这两个参数的值计算出具体的并发量，并以此指导实际的载荷发送操作。那么这个计算得出的并发量应该被放在哪儿？最好的存放位置当然是载荷发生器的结构内部。因此，我们在其中又声明了这样一个字段：

```

concurrency uint32           // 并发量。

```

一旦有了并发量的具体数值，我们就有了控制载荷发生器使用系统资源的依据。另外，作为一个Go语言的并发程序，它使用Goroutine的个数应该是我们关心的。过少的Goroutine数量会使程序的并发程度不够，从而导致程序不能充分地利用系统资源。而过多的Goroutine数量则可能会使程序的性能不升反降。因为这对于Go语言运行时系统及其依托的操作系统来说都会造成过重的负担。那么怎样合理地控制程序所启用的Goroutine的数量呢？

我们在讲非缓冲通道时举过一个示例，其中提到了一个控制Goroutine数量方法。这涉及一个名为goTicket的变量和一个名为initGoTicket的函数。该函数初始化了一个Goroutine票池。这个Goroutine票池以一个缓冲通道作为载体，并由goTicket变量代表。在那个示例中，这个Goroutine票池处于一种尚未被封装的状态。而在本节的示例中，我们将把它以及相关的操作封装在一个结构体类型中。这个结构体类型是接口类型GoTickets的一个实现。它们的声明都被放在了loadgen/lib代码包中。我们将在下一小节对它们进行展示和讲解。

既然有了这样一个Goroutine票池，我们就会在载荷发生器的相关方法中使用到它。因此，我们也应该把它作为载荷发生器结构中的一个字段：

```
tickets    lib.GoTickets    // Goroutine票池。
```

载荷发生器良好的可控性不应仅仅体现在这一个方面。在载荷发生器运行的过程中，我们应该可以随时停止它。为此，我们应该着手设计被用来传递停止信号的方式。我们说过，在不同的Goroutine之间传递数据最好的方式就是通过通道。而由于载荷发生器在一开始就是被作为并发程序创建的。因此，我们在其结构中添加这样一个字段：

```
stopSign    chan byte        // 停止信号的传递通道。
```

由于被传递的数据只是一个“信号”而已，所以我们应该把该通道的元素类型设定为一个占用空间最小的类型。在Go语言中，bool类型和byte类型都是占用空间最小的基本类型。它们的单值都只会占用一个字节的空間。不过，byte类型比bool类型更加灵活一些。因为它可能的值不只两个。这会对我们今后可能的扩展有好处。据此，我们才选用byte类型作为stopSign通道的元素类型。

至此，我们向载荷发生器的结构中又加入了3个起到控制作用的字段。不过，这还不算完。既然载荷发生器可以有不止一种的状态，那我们理应再为它添加一个状态字段。状态字段的类型可以是数值类型。但是，为了让它的值与普通的数值有所区别，我们专门为此声明了一个类型：

```
// 载荷发生器的状态的类型。
type GenStatus int
```

可以看到，GenStatus类型只是int类型的一个别名类型而已。另外，为了让载荷发生器的状态更有字面意义，我们还声明了3个GenStatus类型的常量：

```
const (
    STATUS_ORIGINAL GenStatus = 0
    STATUS_STARTED   GenStatus = 1
    STATUS_STOPPED   GenStatus = 2
)
```

有了它们，我们设置和判断载荷发生器的状态的时候就方便多了。GenStatus类型和这几个常量的声明也都被放到了loadgen/lib代码包中。

有了lib.GenStatus类型，我们就可以为载荷发生器添加状态字段了：

```
status      lib.GenStatus    // 状态。
```

最后，不要忘记，我们想让载荷发生器具有良好的可扩展性。我们应该让它的使用者可以根据具体需求对它进行适当的扩展和定制。为了达到这个目的，我们应该预先在其结构中添加一个字段，并以此作为载荷发生器的扩展接口。

不过，在添加这个字段之前，我们应该先搞清楚载荷发生器需要提供哪些扩展。首先，载荷发生器的核心功能肯定是控制和协调请求的生成和发送、响应的接收和验证，以及最终结果的递交等一系列的操作。既然由它来进行对这些操作的控制和协调，那么有些具体的操作是否就可以由可定制的组件来做呢？这样我们就可以把核心功能与可扩展的功能（或者说可作为组件的功能）区分开了。并且，如此一来，既可以保证核心功能的稳定，又可以提供较高的可扩展性。

我们已经确定了一定要作为核心功能的部分。现在我们来看看有哪些被控制和协调的操作可

以作为组件功能。显然，我们不知道或者无法预测到被测软件提供API的形式。况且，载荷发生器不应该对此有所约束，它们可以是任意的。因此，与调用被测软件的API有关的功能应该被作为组件功能，这涉及请求的发送操作和响应的接收操作。据此，既然我们组件化了调用被测软件API的功能，那么请求的生成操作和响应的检查操作也都肯定无法由载荷发生器本身来提供。

根据上面的分析，我们编写出了这样一个接口类型来体现可被组件化的功能：

```
// 调用器的接口。
type Caller interface {
    // 构建请求。
    BuildReq() RawReq
    // 调用。
    Call(req []byte, timeoutNs time.Duration) ([]byte, error)
    // 检查响应。
    CheckResp(rawReq RawReq, rawResp RawResp) *CallResult
}
```

虽然说被作为非核心功能，但是该接口类型中的那几个方法所代表的操作也都是载荷发生器在运行过程中不可或缺的。在执行测试流程的过程中，它们肯定需要被用到。因此，我们应该在初始化载荷发生器的时候给定一个lib.Caller接口类型的实现值。并且，在载荷发生器的结构中也应该存在一个该类型的字段，以存放我们在初始化时给定的那个实现值。

到这里，我们根据一系列准备和设计而编写的载荷发生器的结构体就完成了。表示其结构的完整代码如下：

```
// 载荷发生器的实现。
type myGenerator struct {
    caller      lib.Caller      // 调用器。
    timeoutNs   time.Duration   // 响应超时时间，单位：纳秒。
    lps         uint32          // 每秒载荷发送量。
    durationNs  time.Duration   // 负载持续时间，单位：纳秒。
    concurrency uint32          // 并发量。
    tickets     lib.GoTickets   // Goroutine票池。
    stopSign    chan byte       // 停止信号的传递通道。
    status      lib.GenStatus    // 状态。
    resultCh    chan *lib.CallResult // 调用结果通道。
}
```

这个代表了载荷发生器的结构体类型myGenerator被放在loadgen代码包中。在它的结构中，共包含了9个字段。其中，限定标识符lib.Caller表明，该接口类型也被放在了loadgen/lib代码包中。读者可能会有个疑问：代表载荷发生器的结构体类型myGenerator为什么是包级私有的？难道我们不想让loadgen包之外的程序使用它吗？关于这个问题，我们稍后再做解释。

7.3.3 初始化

在完成基本结构的编写之后，我们就要开始考虑载荷发生器应该以怎样的方式被初始化了。是直接通过复合字面量，还是用其他更好的方式？

对于简单、直接的结构体来说，使用复合字面量来初始化肯定会是首选。但是对于稍微复杂

一些的结构体来说，只是简单地为其中的字段赋予相应的值就不能算是充分地初始化了。因为，这样的结构体会对字段的值有所要求，并且有些字段的值是需要通过一些计算步骤才能给出的。我们在上一小节声明的载荷发生器的结构体类型`myGenerator`就属于这种情况。

在Go语言中，一般会使用一个函数来创建和初始化较复杂的结构体。这类函数的名称通常会以“New”作为前缀，然后后跟相关的名称，像这样：

```
func NewMyGenerator() *myGenerator
```

依据面向接口编程的原则，我们不应该直接将`myGenerator`或`*myGenerator`作为上述函数的结果类型。因为这样会使该函数与具体的结构体类型紧密地绑定在一起。如果我们要修改该结构体类型的名称或者完全更换一套载荷发生器的实现，那么调用该函数的所有代码都不得不经被动的变化，这会造成散弹式的修改。我们应该尽力避免此类情况的发生。因此，让这类初始化函数返回一个接口类型的结果是很有必要的。这个接口类型应该可以充分地体现出载荷发生器的行为。声明这个接口类型相当容易。为了使载荷发生器具有易用性，我只需让它暴露寥寥几个方法。请看下面的这个接口类型声明：

```
// 载荷发生器的接口。
type Generator interface {
    // 启动载荷发生器。
    Start()
    // 停止载荷发生器。
    // 第一个结果值代表已发载荷总数，且仅在第二个结果值为true时有效。
    // 第二个结果值代表是否成功将载荷发生器转变为已停止状态。
    Stop() (uint64, bool)
    // 获取状态。
    Status() GenStatus
}
```

方法`Start`当然是需要的。我们用它来启动载荷发生器。`Stop`方法是为了实现我们之前所说的载荷发生器的可控性而存在的。我们应该可以在任何时候停止载荷发生器的运行。该方法的两个结果值分别代表已发载荷总数和停止操作的成功与否。通常情况下，该操作总是会成功的。但是，在载荷发生器还未被启动的情况下，调用`Stop`方法肯定不会起到什么作用。这时，该方法的第二个结果值就应该是`false`。至于`Status`方法，应该真实地反映出载荷发生器的当前状态。该方法的结果类型之所以是`GenStatus`而不是`lib.GenStatus`，是因为接口类型`Generator`的声明与`GenStatus`类型的声明被放在了同一个代码包中。

好了，在有了这样一个接口之后，那个被用于创建和初始化载荷发生器的函数的声明应该改变为：

```
func NewGenerator() lib.Generator
```

我们打算让`*myGenerator`类型成为接口类型`lib.Generator`的一个实现类型。这需要我们为`*myGenerator`类型编写出与`lib.Generator`接口中声明的方法一一对应的3个方法。读者会在后面陆续看到这一实现过程。

显然，`NewGenerator`函数的声明还不完善。为了实现其功能，我们还应该为它添加若干个参数声明。因此，`NewGenerator`函数的第二版声明应该是这样的：

```
func NewGenerator(
    caller lib.Caller,
    timeoutNs time.Duration,
    lps uint32,
    durationNs time.Duration,
    resultCh chan *lib.CallResult) (lib.Generator, error)
```

该函数的5个参数的名称、类型和含义分别与myGenerator类型中的相应字段对应。在这个函数中，我们会根据它们的值对myGenerator类型进行充分的初始化。不过，在这之前，我们应该先对这几个参数的值的有效性进行检查，并在检查不通过时向函数调用方告知错误情况。这也是我们为该函数添加第二个结果声明的原因。如果此处的检查通过了，我们就需要依据这几个参数值创建并初始化一个myGenerator类型的值。当然，使用复合字面量是必须的，就像下面这样：

```
gen := &myGenerator{
    caller:    caller,
    timeoutNs: timeoutNs,
    lps:       lps,
    durationNs: durationNs,
    stopSign:  make(chan byte, 1),
    status:    lib.STATUS_ORIGINAL,
    resultCh:  resultCh,
}
```

我们使用复合字面量创建了一个myGenerator类型的值，同时对其中的7个字段进行了赋值。然后，我们把该值的指针赋给了变量gen。还记得吗？在我们的设计中，*myGenerator类型会是接口类型lib.Generator的一个实现类型。所以，我们需要把gen的值作为NewGenerator函数的第一个结果值。这样能够被编译通过的前提是，*myGenerator类型的方法集合中要包含lib.Generator接口类型中声明的那3个方法。我们会在下一小节展示*myGenerator类型的这3个公共方法的具体实现。

经过上述步骤之后，我们已经对载荷发生器中的7个字段进行了检查和赋值。不过不要忘了，仍有两个字段未被初始化。它们是concurrency字段和tickets字段。concurrency字段的值应该代表相关调用过程的并发执行数量。一个调用过程总体上包含了两个操作。一个是向被测软件发送一个载荷（或者说对被测软件的API进行一次调用）的操作，另一个是等待并从被测软件那里接收一个响应（或者说等待并获取被测软件的API返回的此次调用的结果）的操作。换句话说，一个调用过程即代表了载荷发生器通过一个载荷与被测软件进行的一次交互。因此，这一过程的并发执行数量可以比较真实地反映出被测软件的负载程度。

调用过程的并发执行数量（以下简称并发量）需要根据timeoutNs字段和lps字段的值以及一个公式计算得出。这个公式如下：

$$\text{并发量} \approx \text{单个载荷的响应超时时间} / \text{载荷的发送间隔时间}$$

在此公式中，单个载荷的响应超时时间即是timeoutNs字段的值所表示的时间。一旦一个载荷发送操作的耗时达到了响应超时时间，相应的载荷就会被判定为没有被成功响应的载荷。在这之后，载荷发生器不会再去等待该载荷的响应。反过来讲，在达到这个响应超时时间之前，接收

该载荷响应的操作是不会被强制结束的。假设响应超时时间是5秒钟，并且所有载荷响应接收操作都会用满这个时间，那么在此时间范围内开始的所有的载荷响应接收操作都会被并发地执行。也就是说，如果我们在5秒钟之前开始计数并在达到响应超时时间的此刻停止计数，那么在这个5秒钟的时间范围之内开始的载荷响应接收操作的数量就应该约等于此刻的并发量。在响应超时时间为5秒钟的设定下，如果我们每隔1秒钟向被测软件发送一个载荷，那么这个并发量就是5。而如果我们每隔1毫秒发送一个载荷，那么该并发量就应该是5000。这里所说的发送间隔时间是可以由载荷发生器的lps字段的值计算得出的。concurrency、timeoutNs和lps这3个字段的值之间的关系如下：

$$\text{concurrency} \approx \text{timeoutNs} / (1\text{e}9 / \text{lps}) + 1$$

其中，表达式 $1\text{e}9 / \text{lps}$ 所表示的就是根据使用方对lps的值的设定计算出的载荷发生器发送载荷的间隔时间，单位是纳秒。1e9代表了1秒钟对应的纳秒数。这样，表达式 $\text{timeoutNs} / (1\text{e}9 / \text{lps})$ 的含义就是在响应超时时间代表的某一个时间周期内的并发量的最大值。而最后与之相加的1则代表了在某一个时间周期之初向被测软件发送的那个载荷。

对于一个通用的性能测试软件来说，这已经算是比较准确的换算方式了。因为我们无法得知被测软件对于每一个载荷的响应都会经过多长时间才能够被返回。实际上，这个真实的载荷响应时间也应该是我们通过性能测试得到的数值之一。换句话说，性能测试软件要做的就是，先通过若干个预设的限定值模拟出一定程度的负载，然后再以此来测试并得到被测试软件实际能承受的最大负载。我们在前面讲到的响应超时时间、每秒载荷发送量和负载持续时间，以及经过计算得出的并发量都属于预设的限定值。

我们计算并发量的最大意义是：为约束被并发运行的Goroutine的数量提供支撑。也就是说，我们会依此数值确定载荷发生器的tickets字段所代表的那个Goroutine票池的容量。这个容量也可以被理解为Goroutine票的总数量。Goroutine票池的初始化工作是由lib.NewGoTickets函数来完成的。对该函数的调用如下：

```
tickets, err := lib.NewGoTickets(gen.concurrency)
```

在讲述该函数的内部实现之前，我们先来看看tickets字段的类型lib.GoTickets。它也是lib.NewGoTicket函数的第一个结果的类型。此接口类型的声明如下：

```
// Goroutine票池的接口。
type GoTickets interface {
    // 获得一张票。
    Take()
    // 归还一张票。
    Return()
    // 票池是否已被激活。
    Active() bool
    // 票的总数。
    Total() uint32
    // 剩余的票数。
    Remainder() uint32
}
```

这个接口类型包含了4个方法声明。其中，方法Take和Return是对应的。前者的功能是从票池获得一张票，而后者在被调用之后会向票池归还一张票。读者可能会感觉这里的“获得”和“归还”操作都比较抽象，不太好理解。确实是这样。实际上，Goroutine票池既不会关心使用方从它那里获得的是哪一张票，也不需要知道使用方把哪一张票归还给了它。这里的“票”本身就是一个抽象的概念。就像我们在上一节讲单向的非缓冲通道的时候所说的那样，这里的票其实就相当于程序为了启用一个Goroutine而必须持有的令牌。Goroutine票池只负责增减票的数量并以此真实地体现出正在被运行的专用Goroutine的数量。

我们也可以把Goroutine票池看成一个POSIX标准中描述的多值信号灯。一个POSIX多值信号灯的灯值代表了可用资源的数量。资源使用方获得或归还资源时会及时减少或增大该信号灯的灯值，以便其他使用方实时了解相应资源的使用情况。在该值被减至0之后，所有试图减少该值的程序（或者说进程）都会为此而被阻塞。而当该值重新被增至一个正整数的时候，这些程序就都会被唤醒。不过，只会有与信号灯的数值相同的个数的程序能够成功的对其进行减1操作，而其他程序则会被迫继续等待。Goroutine票池所表现的行为与多值信号灯非常地类似。lib.GoTickets接口的Take方法和Return方法分别对应了多值信号灯上的增1操作和减1操作。

接口类型lib.GoTickets中的后3个方法声明应该很好理解。一旦Goroutine票池被正确地初始化，Active方法返回的结果值就应该是true。而Total方法和Remainder方法在被调用后则分别会返回代表了票的总数和剩余数的结果值。

根据lib.GoTickets接口的类型声明以及我们上面的描述，我们很容易地编写出了该接口的实现类型。与lib.Generator接口的实现类型一样，该类型也是一个指针类型，名为*lib.myGoTickets。我们在3.2节中已经说明过结构体类型与指向它的指针类型之间的区别，也强调了与之相对应的值方法和指针方法之间的若干不同。注意，如果我们想要让*lib.myGoTickets类型成为接口类型lib.GoTickets的一个实现类型的话，就必须为lib.myGoTickets类型编写出与lib.GoTickets接口中声明的方法——对应的5个指针方法。

首先，让我们先来编写lib.myGoTickets类型的基本结构：

```
// Goroutine票池的实现。
type myGoTickets struct {
    total    uint32    // 票的总数。
    ticketCh chan byte // 票的容器。
    active   bool      // 票池是否已被激活。
}
```

其中的total字段的含义就是我们刚刚所说的Goroutine票的总数量。ticketCh字段则代表了承载Goroutine票的容器。之所以使用chan byte作为它的类型，是因为我们要以最简单且最节省系统资源的方式来实现Goroutine票的“获得”和“归还”操作，并使它具有自同步特性。而第三个字段active，正是被用来表示当前的Goroutine票池是否已被正确地初始化的。

根据lib.myGoTickets类型中的这3个字段，我们可以立即编写出它的那5个指针方法（请参照lib.GoTickets类型的声明）。这样，我们才能使*lib.GoTickets类型成为lib.GoTickets接口的一个实现类型。这应该并不困难，请读者自己动手试一试。在必要时，读者可以参看前面的说明文

字和我们在上一节编写的`fetchPerson`函数以及相关的描述。那里提及了使用缓冲通道作为Goroutine票池的典型用法。

希望读者已经自己动手编写完成了`lib.myGoTickets`类型的所有必要的公共方法。不知读者是否考虑到了我们前面所说的正确地初始化的问题。在我编写的版本中,对`lib.myGoTickets`类型值的初始化工作都由它的包级私有的指针方法`init`来进行。这个方法的完整声明如下:

```
func (gt *myGoTickets) init(total uint32) bool {
    if gt.active {
        return false
    }
    if total == 0 {
        return false
    }
    ch := make(chan byte, total)
    n := int(total)
    for i := 0; i < n; i++ {
        ch <- 1
    }
    gt.ticketCh = ch
    gt.total = total
    gt.active = true
    return true
}
```

在该方法中,我们在最前面做了一些前置性的检查,并且在最后面对接收者值(即那个`lib.myGoTickets`类型的当前值)中的字段进行了赋值。然而,在此方法体中间的那几行代码却是最关键的。我们以参数`total`的值作为容量初始化了一个元素类型为`byte`的缓冲通道`ch`。我们本可以把这个缓冲通道直接赋给当前值的`ticketCh`字段,但是却没有这么做。还记得吗?我们在前面说过,`lib.myGoTickets`类型值会用它的`ticketCh`字段的值来作为承载Goroutine票的容器。这就意味着,该通道中缓冲的元素值的个数就代表了还没有被获得和已被归还的Goroutine票的总和。那么,在Goroutine票池被初始化的时候,其中所有的Goroutine票应该都没有被获得。因此,在此时,我们应该让该通道缓冲的元素值的个数与其容量相等。如果我们不这么做,那么之后所有试图从该Goroutine票池中获得Goroutine票的Goroutine都会被阻塞,从而会使所有的载荷发送操作都无法进行下去。这也意味着载荷发生器的主流程的执行的停滞。它永远也不会输出测试结果,并且它也无法被运行结束。

相信读者已经理解了“正确的初始化”的含义。在`lib.myGoTickets`类型的`init`方法被编写完成之后,`lib.NewGoTickets`函数就可以非常方便地创建并初始化一个`lib.GoTickets`类型的值了,像这样:

```
func NewGoTickets(total uint32) (GoTickets, error) {
    gt := myGoTickets{}
    if !gt.init(total) {
        errMsg :=
            fmt.Sprintf("The goroutine ticket pool can NOT be initialized! (total=%d)\n", total)
        return nil, errors.New(errMsg)
    }
}
```

```
    return &gt, nil
}
```

如上所示，我们先用复合字面量`myGoTickets{}`创建了一个`lib.myGoTickets`类型的值。不过，在此时，该值的所有字段的值都还只是相应类型的零值。我们还需要调用这个`lib.myGoTickets`类型值的`init`方法以完成初始化。该值的`init`方法会返回一个`bool`类型的结果值以告知初始化工作的成功与否。如果初始化不成功，那么我们就应该及时生成一个`error`类型值并将其返回给`lib.NewGoTickets`函数的调用方。否则，代表了已初始化的`*lib.myGoTickets`类型值的`>`就应该被返回。无论怎样，我们总是应该在该函数中返回两个结果值。

好了，现在我们编写完成了对`myGenerator`类型中的所有字段的初始化代码。现在，我们也可以只通过调用`NewGenerator`函数就创建出一个立即可用的载荷发生器了。`NewGenerator`函数的完成声明如下：

```
func NewGenerator(
    caller lib.Caller,
    timeoutNs time.Duration,
    lps uint32,
    durationNs time.Duration,
    resultCh chan *lib.CallResult) (lib.Generator, error) {
    logger.Infof("New a load generator...")
    logger.Infof("Checking the parameters...")
    var errMsg string
    if caller == nil {
        errMsg = fmt.Sprintf("Invalid caller!")
    }
    if timeoutNs == 0 {
        errMsg = fmt.Sprintf("Invalid timeoutNs!")
    }
    if lps == 0 {
        errMsg = fmt.Sprintf("Invalid lps(load per second)!")
    }
    if durationNs == 0 {
        errMsg = fmt.Sprintf("Invalid durationNs!")
    }
    if resultCh == nil {
        errMsg = fmt.Sprintf("Invalid result channel!")
    }
    if errMsg != "" {
        return nil, errors.New(errMsg)
    }
    gen := &myGenerator{
        caller:    caller,
        timeoutNs:  timeoutNs,
        lps:        lps,
        durationNs: durationNs,
        stopSign:   make(chan byte, 1),
        cancelSign: 0,
        status:     lib.STATUS_ORIGINAL,
        resultCh:   resultCh,
    }
}
```

```

    logger.Infof("Passed. (timeoutNs=%v, lps=%d, durationNs=%v)",
        timeoutNs, lps, durationNs)
    err := gen.init()
    if err != nil {
        return nil, err
    }
    return gen, nil
}

```

注意，变量`gen`的类型是`*myGenerator`而不是`myGenerator`。如果在初始化过程中没有发生任何错误，那么`NewGenerator`函数就会把`gen`变量的值作为结果值返回给调用方。不过，到现在为止，我们还未说明怎样编写`myGenerator`类型的指针方法`Start`、`Stop`和`Status`。它们是使`*myGenerator`类型成为`Generator`接口的实现类型的关键。当然，我们可以先在这里编写出相应的空方法（即方法体中无任何可作为具体实现的语句的方法）来使编译通过。

眼尖的读者可能会立即发现，我们用字面量初始化这个`myGenerator`类型的值的时候，为一个未曾讲过的字段`cancelSign`赋了值。这里先不对此进行解释，到后面再进行说明。

另外，在这段代码中出现的标识符`logger`代表的是`loadgen`包中声明的一个变量。相关代码如下：

```

var logger logging.Logger

func init() {
    logger = logging.NewSimpleLogger()
}

```

变量`logger`会在代码包初始化函数`init`中被初始化。这用到了`goc2p`项目的`logging`代码包中的函数`NewSimpleLogger`。顾名思义，`logging.Logger`类型的值被用来以各种方式记录日志。这些日志可能会被打印或传送到任何地方。但是，可以肯定的是，由`NewSimpleLogger`函数返回的`logging.Logger`类型值只会将日志打印到标准输出上。我们还会在后面见到使用`logger`变量的代码。与此相关的具体实现，请读者阅读`logging`包中的代码。

在本节后面的部分中，我们会具体讨论载荷发生器的总体流程以及`myGenerator`类型的那3个公共的指针方法的编写方法。

7.3.4 启动和停止

我们会在本小节讨论启动和停止载荷发生器的相关流程，并完成对`myGenerator`类型的编写。

前文说过，我们调用载荷发生器的`Start`方法就可以启动它。在这之后，载荷发生器会按照我们给定的参数向被测软件发送一定量的载荷（或者说调用被测软件的API并传送一定量的请求）。在达到了我们指定的负载持续时间之后，载荷发生器会自动停止载荷发送操作。在从启动到停止的这个时间段内，载荷发生器还会将被测软件对各个载荷的响应（如果有的话）以及载荷发送的最终结果收集起来并发送给我们提供的调用结果通道。

这个流程看起来并不复杂。但实际上，其中包含了很多的细节。比较重要的就是有效控制载荷发送的并发量以及载荷发生器本身使用Goroutine的数量。还好，我们在对载荷发生器进行初始

化的时候已经为此做好了准备。下面我们就来看看启动流程的具体实现方法。

1. 启动的准备

我们之前说过，载荷发生器的`lps`字段的值指明了它每秒向被测软件发送载荷的数量。根据这个值，我们可以很轻易地得到发送间隔时间。相应的表达式为`1e9 / lps`。为了让发送间隔时间能够起到实质性的作用，我们需要使用缓冲通道和断续器。还记得吗？我们在上一节介绍过断续器。它非常适合被作为定时任务的触发器。请看下面的代码：

```
// 设定节流阀
var throttle <-chan time.Time
if gen.lps > 0 {
    interval := time.Duration(1e9 / gen.lps)
    logger.Infof("Setting throttle (%v)...", interval)
    throttle = time.Tick(interval)
}
```

我们给被用来触发定时任务的缓冲通道起了一个看起来很酷的名字——节流阀。为了配合断续器的使用，我们将它的类型设定为`<-chan time.Time`。这是一个单向通道类型。之所以在这里进行通道方向上的限制不会有什么问题，是因为我们仅仅会通过调用`time.Tick`函数为变量`throttle`赋值。我们知道，`time.Tick`函数的结果值就是这个类型的。该结果值是一个可以周期性地传达到期事件的缓冲通道。作为约束，`time.Tick`函数只允许它的调用方从该通道中接收元素值。

如果`lps`字段的值大于0，我们就会算出发送间隔时间并依据这个时间设置断续器。变量`throttle`所代表的节流阀会直接、及时地体现出断续器的行为。

我们暂时把创建好的节流阀放在一边。不用担心，我们已经说明过，断续器的功能不会因代表到期事件的通道元素值未被及时接收而受到影响。

在真正使用节流阀之前，我们还有另外一个准备工作要做，即让载荷发生器能够在运行一段时间之后自己停下来。这里的一段时间就是我们先前给定的负载持续时间。还记得吗？载荷发生器有个被用来传递停止信号的缓冲通道`stopSign`。我们已经在前面对它进行了初始化。它的长度被设定为了1。我们下面就利用`time`包中的`time.AfterFunc`函数来实现定时的向`stopSign`发送停止信号的功能。

首先，为了让这个定时发送操作不阻碍启动流程的执行，我们需要把它放到一个专用的Goroutine中进行。与`time.After`函数相比，使用`time.AfterFunc`函数的优势是可以直接在调用表达式中自定义处理到期事件的操作。因此，实现初始化停止信号的功能的代码应该如下所示：

```
// 初始化停止信号
go func() {
    time.AfterFunc(gen.durationNs, func() {
        logger.Infof("Stopping load generator...")
        gen.stopSign <- 0
    })
}()
```

如上所示，当经过由载荷发生器的`durationNs`字段代表的一段时间之后，该Goroutine中的代码会向停止信号传递通道发送一个元素值以表示停止载荷发生器的流程应该结束了。

为什么使用一个通道来传递停止信号而不是直接结束当前进程呢？主要原因是，我们想要让正在执行中的各个子流程自行结束。这样的结束方法显然会比强制性的结束更加合理和安全。那么stopSign字段是怎样起到让各个子流程自行结束的作用的呢？别着急，我们稍后就会讲到。

到这里，为了启动载荷发生器而进行的所有准备工作都已完成。我们可以改变当前的载荷发生器的状态了：

```
// 设置已启动状态
gen.status = lib.STATUS_STARTED
```

2. 控制流程

在进入已启动的状态之后，载荷发生器才真正开始生成并发送载荷。包含了载荷发送操作和载荷响应接收操作的调用过程应该是被异步执行的。因为只有这样，载荷发生器才能够从总体上管理和控制它们。请看下面的这个函数：

```
func (gen *myGenerator) genLoad(throttle <-chan time.Time, endSign chan<- uint64) {
    callCount := uint64(0)
Loop:
    for ; ; callCount++ {
        select {
            case <-gen.stopSign:
                gen.handleStopSign()
                endSign <- callCount
                break Loop
            default:
        }
        gen.asyncCall()
        if gen.lps > 0 {
            select {
                case <-throttle:
                case <-gen.stopSign:
                    gen.handleStopSign()
                    endSign <- callCount
                    break Loop
            }
        }
    }
}
```

我们使用这个名为genLoad的载荷发生器的指针方法来从总体上控制各个调用流程的执行。该方法接受两个参数，第一个参数就是我们在前面已经准备好的节流阀throttle，而第二个参数endSign则承担着传递停止信号的响应的任务。这个响应不只是被用来表达所有调用过程都已接收到载荷发生器欲停止的通知的。它还被用于向genLoad方法的调用方传递载荷发送的总数。这也是我们把它的元素类型设定为uint64的原因。在本小节的最后，读者会了解到，为了在载荷发生器的Stop方法中正确设置它的第一个结果值，我们需要提升endSign通道的作用域。

在genLoad方法的方法体中，我们使用一个for循环来周期性的向被测软件发送载荷。这个周期的长短是由节流阀控制的。在循环体的最后，如果lps字段的值大于0，那么就表示节流阀是有效并需要使用的。这时，我们利用select语句来等待节流阀中的到期事件。一旦接收到了这样一

个事件，我们就立即开始下一次迭代（即开始生成并发送下一个载荷）。当然，如果在等待节流阀的到期事件的过程中接收到了停止信号，那么我们就应该立即对它进行处理并终止当前的循环。正因为如此，`genLoad`方法中的第二条`select`语句有两个`case`。

针对`stopSign`通道的接收操作也出现在了`for`循环的开始处。这是因为我们要对及时地处理停止信号做进一步的保证。在这条`select`语句中，我们加入了`default case`。原因是我们只想在这里检测一下`stopSign`通道中是否存在停止信号，而不想因此而阻塞迭代的执行。

一旦发现有停止信号，我们就需要立即进行相应的处理。停止信号的处理代码被放在了载荷发生器的`handleStopSign`方法中。它其实非常简单：

```
func (gen *myGenerator) handleStopSign() {
    gen.cancelSign = 1
    logger.Infof("Closing result channel...")
    close(gen.resultCh)
}
```

载荷发生器的`cancelSign`字段也是作为实现可控性的一个部分存在的。它的作用是，在接收到停止信号之后告知正在执行中的各个调用过程，以使它们取消掉还未被进行的操作。这也是`cancelSign`这个名称的由来。注意，`cancelSign`字段的类型不是一个通道类型，而是值占用空间最小的`byte`类型。这是因为我们只会在处理停止信号的时候把它的值从0改变为某一个正整数。由于该字段的特殊用途，所以它的值也永远不会被修改回0。另外，各个调用过程对该字段的值的判断也是极其简单的，只会判断其是否大于0。因此，即使我们的程序会并发地修改它的值也是无关紧要的。在这些前提之下，我们让`cancelSign`字段尽可能地简单化了。我们在后面会了解到调用过程对`cancelSign`的值的判断以及相应的处理。

在`handleStopSign`方法中，我们先把1赋给了载荷发生器的`cancelSign`字段。然后，我们关闭了代表了调用结果通道的`resultCh`字段。这就意味着我们放弃了当前还未被执行完成的调用过程的结果。

在调用`handleStopSign`方法之后，我们向`endSign`通道发送了载荷发送的计数。最后，代表了控制流程的`for`语句的执行宣告结束。

3. 异步地调用

为了让控制流程与各个调用过程分离开来，我们又编写了载荷发生器的`asyncCall`方法。该方法的作用就是异步地执行每一个调用过程。更详细地说，一个调用过程分为5个操作步骤，即生成载荷、发送载荷并接收响应、检查载荷响应、生成调用结果、发送调用结果。其中的前3个操作步骤都会由使用方在初始化载荷发生器时传入的那个调用器中的方法来完成。特别提示，我们至此仍未用到的载荷发生器的`timeoutNs`字段应该在这个调用过程中发挥作用了。

既然说是异步地调用，那么读者应该能想到我们会在`asyncCall`方法中新启用一个Goroutine来完成调用过程。更确切地说，`asyncCall`方法每次被调用之后都会启用一个专用的Goroutine。这里所说的专用的Goroutine就是我们之前讲过的与Goroutine票池中的Goroutine票对应的Goroutine。因此，在`asyncCall`方法中，我们就应该在适当的时候对Goroutine票池中的票进行“获得”和“归还”操作，如下所示：

```
func (gen *myGenerator) asyncCall() {
    gen.tickets.Take()
    go func() {
        // 省略若干条语句
        gen.tickets.Return()
    }()
}
```

我们在启用专用Goroutine之前，从Goroutine票池获得一张Goroutine票。当Goroutine票池中已无票可拿时，`asyncCall`方法所属的Goroutine会被阻塞于此。只有存在多余的Goroutine票的时候，专用Goroutine才会被启用，从而当前的调用过程才会被执行。另一方面，在这个go函数的最后，我们会及时地把票归还给Goroutine票池。这个归还的时机非常重要，既不能提前也不能延后。

好了，现在异步调用的外围框架已经有了。下面我们来看看专用Goroutine需要执行的语句。第一个操作步骤当然是生成载荷。因为有了调用器，所以这里的代码相当简单：

```
rawReq := gen.caller.BuildReq()
```

可以看到，我们仅仅是对调用器的`BuildReq`方法进行了调用并把其返回的原生请求暂存起来而已。

关于发送载荷并接收响应的这个步骤，我们很有必要详细说明一下。首先是对载荷发生器的`timeoutNs`字段的使用。我们已经知道，该字段应该起到辅助载荷发生器实时判断被测软件处理单一载荷是否超时的作用。我们之前讲过，`time`包中的定时器可以被用来设定某一个操作或任务的超时时间。要做到实时的判断超时，最好的方式就是与通道和`select`语句联用。不过，这就需要再启用一个Goroutine来执行发送载荷并接收响应的操作步骤了。那么，我们可以在不额外启用Goroutine的情况下实现实时的超时判断吗？答案是，可以。但是这需要一些技巧。

在讲述这些技巧之前，让我们先来看看联用定时器、通道和`select`语句以及再启用一个Goroutine的做法是怎样的。代表了执行发送载荷并接收响应的操作步骤（以下简称交互操作）的代码都被封装到了载荷发生器的`interact`方法中。该方法的声明如下：

```
func (gen *myGenerator) interact(rawReq *lib.RawReq, rawRespCh chan<- *lib.RawResp)
```

它接受两个参数。第一个参数就是需要被发送给被测软件的原始请求，而第二个参数则是被用来传递原始响应的通道（以下简称原始响应通道）。在从被测软件处接收到响应之后，`interact`方法会把它封装成`*lib.RawResp`类型值并发送给原始响应通道。

我们先要创建并初始化`interact`方法所需的第二个参数值，然后再使用go语句异步地执行该方法。具体代码如下：

```
var result *lib.CallResult
rawRespCh := make(chan *lib.RawResp, 1)
go gen.interact(&rawReq, rawRespCh)
```

前两行代码分别创建了代表了调用结果值的`result`变量和原始响应通道，而第三行代码的功能则是额外启用一个Goroutine来执行载荷发生器的`interact`方法。

在这之后的`select`语句就很好编写了。我们为它添加两个case，一个case试图从原始响应通道中接收元素值，而另一个case则根据`timeoutNs`字段的值创建一个定时器并等待相应的到期事

件的来临。一旦从原始响应通道接收到了元素值，就意味着交互操作的完成。但是，如果在这之前定时器的到期事件抢先到达并被接收了，那么就说明交互操作的执行超时了。此时，载荷发生器就不应该再等待交互操作的执行结果了，而应该直接向调用结果通道发送一个描述了交互操作的超时情况的调用结果值。与此对应的代码如下：

```
select {
case rawResp := <-rawRespCh:
    if rawResp.Err != nil {
        result = &lib.CallResult{
            Id:    rawResp.Id,
            Req:    rawReq,
            Code:   lib.RESULT_CODE_ERROR_CALL,
            Msg:    rawResp.Err.Error(),
            Elapse: rawResp.Elapse}
    } else {
        result = gen.caller.CheckResp(rawReq, *rawResp)
        result.Elapse = rawResp.Elapse
    }
case <-time.After(gen.timeoutNs):
    result = &lib.CallResult{
        Id:    rawReq.Id,
        Req:    rawReq,
        Code:   lib.RESULT_CODE_WARNING_CALL_TIMEOUT,
        Msg:    fmt.Sprintf("Timeout! (expected: < %v)", gen.timeoutNs)}
}
gen.sendResult(result)
```

在这段代码中，调用表达式`gen.caller.CheckResp(rawReq, *rawResp)`的含义是通过调用器的`CheckResp`方法检查原始响应并生成最终的响应结果。当然，只有在及时接收到未携带任何错误的原始响应的情况下，我们才需要对原始响应进行这样的检查。除此之外，我们仅通过复合字面量来部分初始化表示出现了某类错误的响应结果就可以了。

载荷发生器的`sendResult`方法的功能是向调用结果通道发送一个调用结果值。另外，为了更好地定义调用结果值中代表了响应代码的`code`字段的不同值所代表的含义，我们在`loadgen/lib`包中声明了如下的常量：

```
// 保留 1 ~ 1000 给载荷承受者使用。
const (
    RESULT_CODE_SUCCESS           = 0    // 成功。
    RESULT_CODE_WARNING_CALL_TIMEOUT ResultCode = 1001 // 调用超时警告。
    RESULT_CODE_ERROR_CALL        ResultCode = 2001 // 调用错误。
    RESULT_CODE_ERROR_RESPONSE    ResultCode = 2002 // 响应内容错误。
    RESULT_CODE_ERROR_CALEE        ResultCode = 2003 // 被调用方（被测软件）的内部错误。
    RESULT_CODE_FATAL_CALL        ResultCode = 3001 // 调用过程中发生了致命错误！
)
```

因此，我们可以在前一段代码中看到多个诸如`lib.RESULT_CODE_WARNING_CALL_TIMEOUT`的限定标识符。`lib.RESULT_CODE_WARNING_CALL_TIMEOUT`即表示在交互操作的执行已超时的情况下的响应代码。由于这个超时的时间是由载荷发生器的使用者指定的，所以此类情况并不能算是严格意义

上的错误。因此，我们在这里把该响应代码归为警告级别。另一个值得说明的响应代码常量是 `RESULT_CODE_FATAL_CALL`。该常量表示调用过程中的致命错误，即指调用器未预料到的错误。这很可能是调用器自身的错误，不过也不排除存在因调用器对调用过程中可能发生的错误预估不足而没有及时“抓住”错误的这种情况。在载荷发生器中，这类错误会由一个从调用器的某个方法中扩散出来的运行时恐慌表示。我们需要在载荷发生器的 `interact` 方法中加入处理这类错误的代码，不能让运行时恐慌外泄并影响到载荷发生器的控制流程的执行。因此，在调用 `interact` 方法之前插入一条 `defer` 语句是必须的：

```
defer func() {
    if p := recover(); p != nil {
        err, ok := interface{}(p).(error)
        var buff bytes.Buffer
        buff.WriteString("Async Call Panic! ")
        if ok {
            buff.WriteString("error: ")
            buff.WriteString(err.Error())
        } else {
            buff.WriteString("clue: ")
            buff.WriteString(fmt.Sprintf("%v", p))
        }
        buff.WriteString("\n")
        errMsg := buff.String()
        logger.Fatalln(errMsg)
        result := &lib.CallResult{
            Id: -1,
            Code: lib.RESULT_CODE_FATAL_CALL,
            Msg: errMsg
        }
        gen.sendResult(result)
    }
}()
```

我们在第4章讲异常处理的时候详细探讨过运行时恐慌的处理方法。由此可知，`recover` 函数的结果值的动态类型是未知的（其静态类型是 `interface{}`）。又由于在用的调用器很可能是载荷发生器的使用方自行实现的，所以在这里就更增加了不确定性。因此，我们先使用类型断言表达式来判断变量 `p` 的动态类型是否为 `error`，然后再根据判断结果组织不同内容的结果成因简述（即会赋给调用结果值的 `Msg` 字段的值）。在这条 `defer` 语句的最后，我们把这个代表了调用致命错误的调用结果值发送给调用结果通道。

上面这条 `defer` 语句会作为与专用 Goroutine 对应的匿名 go 函数中的第一条语句出现。在它之后的就是前面展示的那两条分别声明变量 `result` 和 `rawRespCh` 的语句、被用来异步执行 `interact` 方法的 go 语句，以及那条最为关键的 `select` 语句。至此，专用 Goroutine 所对应的匿名 go 函数的编写即将完成。不过不要忘了，我们要在这个 go 函数即将被执行结束的时候，把持有的 Goroutine 票归还给 Goroutine 票池：

```
gen.tickets.Return()
```

这条语句实际上已经在前面的示例中出现过了。这里只是再强调一下。

好了，我们已经编写完成了在`asyncCall`方法中使用一个专用的Goroutine执行一个调用过程的全部代码。这个`asyncCall`方法的实现看起来还不错。可是，如果我们再次纵观这段代码的话就会发现，我们实际上是启用了两个Goroutine来执行一个调用过程。一个Goroutine就是我们所说的专用Goroutine。而在专用Goroutine中，我们又启用了另一个Goroutine来异步地执行`interact`方法。显然，这与我们的初衷有些出入。我们本想用Goroutine票池来限制专用Goroutine的总数量，但是实际情况却是，被用来执行调用过程的Goroutine的最大数量可能是Goroutine票池的总容量的两倍。启用额外的Goroutine的主要原因是我们需要实现针对交互操作的实时超时判断。如果我们能够用另一种方式实现这个实时超时判断，那么就能够避免这个额外Goroutine的启用了。为了达到这个目的，就需要对`asyncCall`方法的实现进行必要的改造。或许，还需要修改一下`interact`方法的签名。

我们先来试着改造一下`asyncCall`的方法的`select`语句中的第一个`case`表达式，像这样：

```
case rawResp := <-func() <-chan *lib.RawResp {
    rawRespCh := make(chan *lib.RawResp, 1)
    gen.interact(&rawReq, rawRespCh)
    return rawRespCh
}():
```

可以看到，我们使用匿名函数`func() <-chan *lib.RawResp`来作为该`case`的通道表达式。在这个匿名函数中，我们顺序地执行了`interact`方法，并将相应的原始响应通道作为该匿名函数的结果值。乍一看，这样做似乎也是可以的。但是，还记得吗？我们在上一节讲到，运行时系统会先对`select`语句的各个`case`中的通道表达式和元素表达式进行求值，而且在所有的求值都完成以后才会去考虑选择某一个`case`执行。对于上面的这个被改造后的`case`来说，匿名函数中的`interact`方法会被早早地执行完毕。更重要的是，这条`select`语句的第二个`case`中的那个定时器直到第一个`case`中的通道（即那个匿名函数的结果值）已经缓冲了一个原始响应之后才会被启动。这相当于在执行完`interact`方法之后再去启动定时器。显然，这个定时器根本就起不到它应有的作用。所以，这个解决方案是不符合要求的。

现在让我们从头再来。实际上，我们想要的只是实时超时判断的异步化，而串行的执行交互操作应该是没有问题的。异步地进行实时超时判断其实并不困难。还记得`time.AfterFunc`函数吗？它可以帮助我们自定义在定时器的到期事件来临之时执行的处理函数。我们在前面说过，当定时器的到期事件抢先到来的时候，载荷发生器的`asyncCall`方法就应该直接向调用结果通道发送一个代表了交互操作超时的调用结果值，而不用再去理会交互操作。因此，我们应该这样来自定义超时处理函数以及调用`time.AfterFunc`函数：

```
var timeout bool
timer := time.AfterFunc(gen.timeoutNs, func() {
    timeout = true
    result := &lib.CallResult{
        Id:    rawReq.Id,
        Req:   rawReq,
        Code:  lib.RESULT_CODE_WARNING_CALL_TIMEOUT,
        Msg:   fmt.Sprintf("Timeout! (expected: < %v)", gen.timeoutNs)}
```

```
    gen.sendResult(result)
})
```

可以看到，我们先声明了一个名为`timeout`的`bool`类型的变量（以下简称超时标识）。并且，在作为参数传递给`time.AfterFunc`函数的那个匿名的超时处理函数中，我们一开始就对它进行了设置。一旦超时标识`timeout`的值被设置为`true`，就说明交互操作已经执行超时。我们会在交互操作被执行完成之后去判断它。请看下面的代码：

```
rawRespCh := make(chan *lib.RawResp, 1)
gen.interact(&rawReq, rawRespCh)
rawResp := <-rawRespCh
if !timeout {
    timer.Stop()
    var result *lib.CallResult
    if rawResp.Err != nil {
        // 省略若干条语句
    } else {
        // 省略若干条语句
    }
    gen.sendResult(result)
}
```

这段代码表示了交互操作的串行执行，以及后续的调用结果值的生成过程。后者的实现代码与前面展示过的代码并无不同，因此我们在这里省略掉了一些细节。我们需要关注的其实只有两个地方，即`if`语句的第一行和第二行。在从通道中接收到一个原始响应之后，我们会首先判断超时标识`timeout`的值。仅当其值为`false`的情况下，我们才会进行后续的操作。否则，我们会忽略掉这个原始响应。这正符合我们声明`timeout`变量的初衷。此外，在需要进行后续操作的时候，我们在第一时间就停止了定时器。其意义是及时阻止自定义超时处理函数的执行。如若不然，一个已经无效的、代表交互操作执行超时的调用结果值就会在超时时间到来时被发送到调用结果通道中。

通过对`timeout`变量的赋值和判断以及对`timer.Stop`方法的调用，我们实现了异步化的实时超时判断子流程与作为主线的调用流程之间的同步。同时，我们还避免了额外的Goroutine的启用。这就意味着，对Goroutine票池的使用可以完全起到约束专用Goroutine数量的作用了。

至此，我们对`asyncCall`方法的改造已经基本完成。不过，不知读者是否发现，在经过这样的改造之后，原始响应通道`rawRespCh`已经显得有些多余了。因此，我们对`interact`方法的声明进行了修改：

```
func (gen *myGenerator) interact(rawReq *lib.RawReq) *lib.RawResp
```

然后在`asyncCall`方法中完全去掉了`rawRespCh`通道，即将这几行代码

```
rawRespCh := make(chan *lib.RawResp, 1)
gen.interact(&rawReq, rawRespCh)
rawResp := <-rawRespCh
```

变更为：

```
rawResp := gen.interact(&rawReq)
```

为了便于读者阅读，我们在这里给出载荷发生器的asyncCall方法的完整实现：

```
func (gen *myGenerator) asyncCall() {
    gen.tickets.Take()
    go func() {
        defer func() {
            if p := recover(); p != nil {
                err, ok := interface{}(p).(error)
                var buff bytes.Buffer
                buff.WriteString("Async Call Panic! (")
                if ok {
                    buff.WriteString("error: ")
                    buff.WriteString(err.Error())
                } else {
                    buff.WriteString("clue: ")
                    buff.WriteString(fmt.Sprintf("%v", p))
                }
                buff.WriteString(")")
                errMsg := buff.String()
                logger.Fatalln(errMsg)
                result := &lib.CallResult{
                    Id:    -1,
                    Code:  lib.RESULT_CODE_FATAL_CALL,
                    Msg:   errMsg}
                gen.sendResult(result)
            }
        }()
        rawReq := gen.caller.BuildReq()
        var timeout bool
        timer := time.AfterFunc(gen.timeoutNs, func() {
            timeout = true
            result := &lib.CallResult{
                Id:    rawReq.Id,
                Req:   rawReq,
                Code:  lib.RESULT_CODE_WARNING_CALL_TIMEOUT,
                Msg:   fmt.Sprintf("Timeout! (expected: < %v)", gen.timeoutNs)}
            gen.sendResult(result)
        })
        rawResp := gen.interact(&rawReq)
        if !timeout {
            timer.Stop()
            var result *lib.CallResult
            if rawResp.Err != nil {
                result = &lib.CallResult{
                    Id:    rawResp.Id,
                    Req:   rawReq,
                    Code:  lib.RESULT_CODE_ERROR_CALL,
                    Msg:   rawResp.Err.Error(),
                    Elapse: rawResp.Elapse}
            } else {
                result = gen.caller.CheckResp(rawReq, *rawResp)
                result.Elapse = rawResp.Elapse
            }
            gen.sendResult(result)
        }
    }
}
```

```

    }
    gen.tickets.Return()
  }()
}

```

在`asyncCall`方法中，我们充分地使用了载荷发生器的使用方提供的调用器。在上述代码中，调用器的`BuildReq`方法和`CheckResp`方法各被调用了一次。而对它的`Call`方法的调用则被隐含在了负责进行交互操作的载荷发生器的`interact`方法中。

实际上，我们在载荷发生器的`interact`方法中只做了一件事，那就是对调用器的`Call`方法进行调用，即与被测软件进行一次交互操作。当然，为了真实地记录一次交互操作，我们还需要添加一些其他代码以进行诸如检查原始请求的有效性、记录对调用器的`Call`方法的调用耗时，以及根据`Call`方法的结果值生成原始响应等操作。

4. 启动和停止

让我们再次回到启动载荷发生器的话题上来。在一切准备工作完成之后，载荷发生器的`Start`方法中的代码会调用`genLoad`方法以执行控制流程。同时，它还会把一个名为`endSign`的缓冲通道作为参数传给`genLoad`方法。`genLoad`方法一旦从`stopSign`字段代表的通道处接收到停止信号，就会立即做出响应。它首先会把`cancelSign`字段的值设置为1，然后关闭调用结果通道。最后，它会向`endSign`通道发送一个代表了调用执行计数的元素值。`endSign`通道有两个作用。第一个作用是告知停止信号已被处理完毕，载荷发生器已经可以进入到已停止状态了。而第二个作用就是传递调用执行计数。这个告知和传递的对象有可能是`Start`方法中的代码，也可能是`Stop`方法中的代码。我们稍后会介绍它们对此的不同处理方式。

载荷发生器的停止流程总共涉及了3个被用来代表或传递信号的变量：`stopSign`、`endSign`和`cancelSign`。它们的作用都是在载荷发生器自动或被手动地停止的时候协调和传递其中的某种状态。通道`stopSign`被用来传递由定时器或载荷发生器的`Stop`方法中的代码所发送的停止信号。而`endSign`通道的作用则正如我们刚刚所说的那样。至于`byte`类型的载荷发生器字段`cancelSign`，我们至今还未说明它的真正作用。

为了让读者能够清晰地了解到载荷发生器联用这3个“信号”的方式，我绘制了一幅流程图（见图7-4）。在这幅流程图中，我们以载荷发生器自动停止的情况为例。

在载荷发生器的自动停止流程中，控制流程对`cancelSign`字段的设置会影响到所有已经开始执行的异步调用过程。更具体地说，各个异步调用过程在把调用结果发送给调用结果通道之前会先检查相应的充分条件，并且仅在充分条件被满足的情况下才会真正发送调用结果。具体代码如下：

```

func (gen *myGenerator) sendResult(result *lib.CallResult) bool {
    if gen.status == lib.STATUS_STARTED && gen.cancelSign == 0 {
        gen.resultCh <- result
        return true
    }
    logger.Warn("Ignore result: %s.\n",
        fmt.Sprintf(
            "Id=%d, Code=%d, Msg=%s, Elapse=%v",

```

```

        result.Id, result.Code, result.Msg, result.Elapse))
    return false
}

```

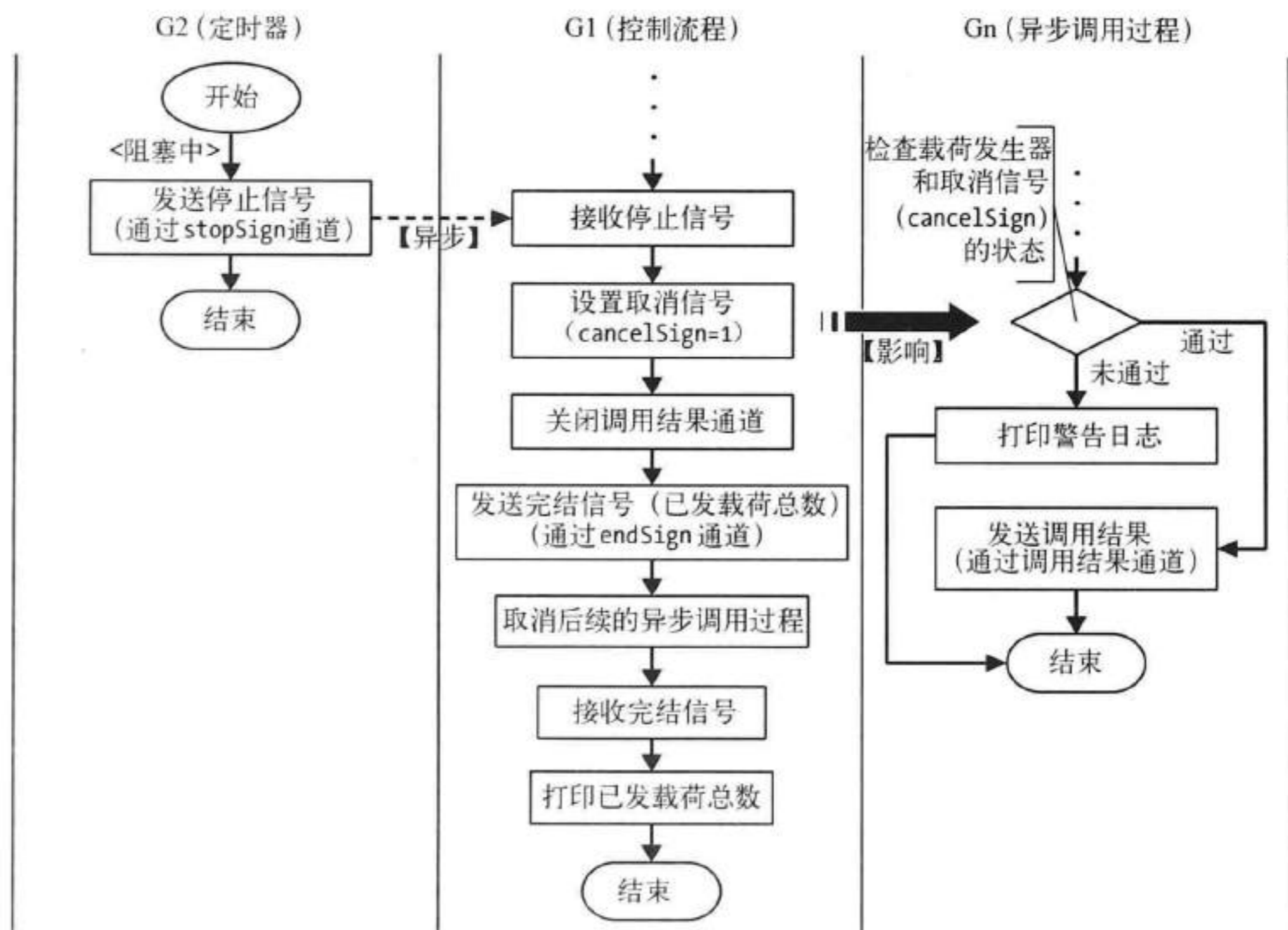


图7-4 载荷发生器的自动停止流程

回顾asyncCall方法的实现，sendResult方法正是在其中被调用的。可以看到，我们刚刚所说的充分条件被满足的情况，即是载荷发生器的状态为lib.STATUS_STARTED且cancelSign字段的值为0。也就是说，把cancelSign字段的值设置为1，就相当于阻止所有还未被执行完成的异步调用过程向调用结果通道发送调用结果。换句话说，对cancelSign字段的设置致使载荷发生器放弃了这些调用结果。

下面我们再来看载荷发生器的手动停止流程。我们可以通过调用载荷发生器的Stop方法来手动地停止它。具体的停止方式与自动停止的方式如出一辙，也是向作为载荷发生器字段之一的stopSign通道发送一个元素值。Stop方法的实现如下：

```

func (gen *myGenerator) Stop() (uint64, bool) {
    if gen.stopSign == nil {
        return 0, false
    }
    if gen.status != lib.STATUS_STARTED {
        return 0, false
    }
}

```

```

    gen.status = lib.STATUS_STOPPED
    gen.stopSign <- 1
    callCount := <-gen.endSign
    return callCount, true
}

```

可以看到，如果stopSign通道还未被初始化，就意味着载荷发生器还未被启动。这时理应忽略掉后续的操作。而在载荷发生器的状态不等于lib.STATUS_STARTED的情况下，我们也不应该向stopSign通道发送停止信号。因为，这样的载荷发生器可能还未被启动，也可能已经被停止。无论怎样，后续的操作都应该被忽略。如果前面这两项检查都通过了，那么Stop方法就会先把当前载荷发生器的状态设置为lib.STATUS_STOPPED。之所以先设置其状态，是为了避免并发地调用Stop方法可能造成的相关操作被重复执行的情况。在设置好状态之后，Stop方法会向stopSign通道发送停止信号，然后再试图从endSign通道处接收代表了调用执行计数的元素值。一旦从中接收到了该计数，那么就说明停止信号已被处理完毕。最后，Stop方法会把调用执行计数和true返回给其调用方。

注意，在Stop方法中被使用的endSign通道是一个载荷发生器的字段，而不是一个局部变量。这就是我们在前面提及的需要将endSign通道的作用域提升的情况。因为如果在启动载荷发生器的时候传递给genLoad方法的参数endSign是一个局部变量，那么我们就不能在Stop方法中使用它了，且Stop方法的调用方也就无法得到确切的调用执行计数了。因此，在myGenerator结构体类型的声明中，我们还应该再加入一个字段的声明：

```
endSign    chan uint64           // 完结信号的传递通道，同时被用于传递调用执行计数。
```

至此，该类型的字段的数量已经增至11个，即除了在上一小节中列罗的那9个字段外，还新增了本小节提及的cancelSign和endSign。

在有了endSign字段之后，我们应该在Start方法中设置载荷发生器的状态为lib.STATUS_STARTED之前对它的endSign字段进行初始化：

```
// 初始化完结信号通道
gen.endSign = make(chan uint64, 1)
```

而在设置已启动状态之后，我们还应该把该字段的值传递给genLoad方法：

```
// 生成载荷
logger.Infof("Generating loads...")
gen.genLoad(throttle, gen.endSign)
```

或者，我们更应该直接修改一下genLoad方法的声明：

```
func (gen *myGenerator) genLoad(throttle <-chan time.Time)
```

并修改上述调用语句：

```
gen.genLoad(throttle)
```

因为该方法中的代码已经可以直接使用作为载荷发生器字段的endSign通道了。

注意，由于genLoad方法所代表的控制流程是被同步执行的，所以在载荷发生器自动停止的流程中，endSign字段的第一个作用是无效的。也就是说，在这样的流程中，它仅起到了传递调

用执行计数的作用。在Start方法的最后（也就是紧随对genLoad方法的调用之后），我们是这样编码的：

```
// 接收调用执行计数
callCount := <-gen.endSign
gen.status = lib.STATUS_STOPPED
logger.Infof("Stopped. (callCount=%d)\n", callCount)
```

在genLoad方法被执行完毕之后，Start方法中代码会试图从endSign通道中接收元素值。这一接收操作总会立即成功。因为在genLoad方法被执行完毕之前，endSign通道中一定会缓冲有一个元素值。在该接收操作完成后，载荷发生器的停止状态会立即被设置。最后，从endSign通道接收到的调用执行计数会被记录到日志中。由logger变量的初始化方式可知，该日志会被打印到标准输出上。

纵观前面对载荷发生器的实现类型myGenerator的代码的展示和描述可知，它的每一个字段都是不可或缺的。通过它们，我们将载荷发生器中的各个组件有机地联系在了一起。根据使用方指定的timeoutNs、lps和durationNs的值，我们设定了载荷发生器的行为。在异步地执行多个调用过程的流程中，我们使用其lib.GoTickets类型的字段tickets严格地限制住了专用Goroutine的数量。不过，为了达到这一目的，我们还比较精细地设计了该流程的具体实现方式。其中涉及了定时器的高级用法。字段stopSign、endSign和cancelSign对于载荷发生器的可控性的实现来说都是十分关键的。有了对它们的联合使用，才使得载荷发生器能够被安全地停止。其中，stopSign通道和endSign通道都充分起到了传递关键数据和状态的作用。而对cancelSign字段的类型的选择则充分体现了我们对直接被多个Goroutine共享的变量的考量。在充分的考量之下，我们决定将该字段的类型设定为byte，而不是某一个通道类型。在编写并发程序的过程中，我们总是应该小心应对，并以最直接、简单和安全的方式来组织代码。此外，在并发程序中，异步的传递结果总是有必要的。使用通道是跨Goroutine传递数据的绝佳选择。在载荷发生器中，我们就是通过它的通道类型的字段resultCh来收集和传递各个异步调用过程的结果的。

好了，我们已经悉数展现和解释了实现一个载荷发生器所需的主要实现代码。这些代码都被放在goc2p项目的loadgen代码包及其子代码包中。读者可以在需要时对照阅读。

在下一小节中，我们会利用之前讲到的知识编写针对载荷发生器的测试源码文件。与此同时，我们还会编写一个lib.Caller接口的实现类型，并使用这个类型来完成对载荷发生器的功能测试。

7.3.5 调用器和功能测试

载荷发生器的调用器应该由使用者给定。这样才能够使载荷发生器具有良好的可扩展性。使用者应该知道怎样生成可以施加于被测软件的载荷、怎样向被测软件发送载荷，以及怎样验证被测软件对载荷做出的响应。这3个行为的声明就组成了我们先前提到的调用器接口loadgen/lib.Caller。

载荷发生器的使用者应该先开发出调用器的实现，然后再使用载荷发生器为被测软件做性能测试。我们下面会简单展示并描述一个调用器的实现过程，然后通过它初始化一个载荷发生器并对被测软件进行测试。注意，我们进行此测试的目的并不是要测试被测软件，而是要以此检验载

荷发生器的功能。因此，我们除了要编写出一个有效的调用器实现之外，还要开发出一个简单的被测软件。只有形成了这样一个闭环之后，我们才能真正地使载荷发生器运行起来，并以此检查其体现的功能是否完全符合我们的设计初衷。

假设有这样一个被测软件，它提供了一个基于网络的API。该API的功能就是根据请求中的参数进行简单且有限的算术运算（针对整数的加减乘除运算），并将结果作为响应返回给请求方。这个被测软件相当简单，所以我们的调用器实现也不会太复杂。读过上一章的读者可能立刻想到使用Go语言的标准库中提供的Socket编程API来实现它们。的确，我们需要的就是这个。

为了贴合本节的主题，我并不想在被测软件的具体实现上耗费笔墨。但是，既然调用器需要与它进行通讯，那么我们还是有必要对请求和响应的结构做一些介绍的。为了简化对它们的组装和解析的操作，我们会使用标准库的encoding/json代码包中的API。这些API既可以让我们非常方便地把某个结构体类型的实例转换成json格式的普通文本数据，又可以把符合格式要求的json数据转换成对应的结构体类型的实例。在确定了这样的数据转换方式之后，我们需要声明两个分别代表请求和响应的结构体类型。请看下面的代码：

```
type ServerReq struct {
    Id      int64
    Operands []int
    Operator string
}

type ServerResp struct {
    Id      int64
    Formula string
    Result  int
    Err     error
}
```

结构体类型ServerReq表示了请求的结构。它包含了3个字段。其中，字段Id的值会唯一标识一个请求。而字段Operands和Operator则分别代表了多个运算数和一个运算符。根据Operands字段的类型，读者应该可以得知这里只允许针对整数的算术运算。虽说Operator字段的类型是string，但是它允许的值是非常有限的，即“+”、“-”、“*”和“/”。这4个值分别代表了加、减、乘和除。如果请求中实际给定的运算符不在此范围之内，那么被测软件肯定不会返回预期的响应。

再来看代表响应结构的结构体类型ServerResp。它的字段Id的值应该唯一标识一个响应。它的值应该和与之对应的请求的Id字段的值一致。字段Result的值需要表示请求中要求的算术运算的结果值。我们已经知道，在Go语言中，两个整数经由“/”运算之后所得到的结果值也会是一个整数。因此该字段的类型是适当的。字段Formula的值会代表相应的运算式子，例如： $2 + 4 + 5 = 11$ 。最后，字段Err是一个error类型的字段。如果被测软件在处理请求的过程中出现了错误，那么该字段的值就会表示相应的错误。

我们把上述两个结构体类型的声明以及被用于测试载荷发生器的调用器和被测软件的实现代码都放到了代码包loadgen/testhelper中。这些代码需要用到loadgen/lib包中的API。为了不引起混淆和编写方便，我们在导入语句中把该代码包的别名设定为了loadgenlib。在本小节后面

的部分中出现的带有loadgenlib.前缀的限定标识符代表的就是loadgen/lib包中的程序实体。

1. 调用器实现的基本结构

在了解了请求和响应的结构之后，我们开始着手编写调用器的实现。调用器需要通过TCP协议与被测软件通讯。因此，我们将这个调用器接口的实现类型命名为TcpComm（TCP Communicator的一种缩写形式）。TcpComm类型的结构非常简单，因为在其中只需要存储被测软件的网络地址：

```
type TcpComm struct {
    addr string
}
```

字段addr的值一般由IP地址（或主机名）和端口号组成，形如："127.0.0.1:8080"。TcpComm类型（更确切地说是*TcpComm类型）需要拥有3个公开的方法声明。这3个公开方法的声明应该与loadgenlib.Caller接口中的方法声明一一对应。我们先来看BuildReq方法。

2. BuildReq方法

该方法应该负责生成并返回一个loadgenlib.RawReq类型的值。从该方法的签名上来看，这个生成规则是需要调用器的编写者自己定义的。读者还记得loadgenlib.RawReq类型的声明吗？它包含了两个字段——int64类型的Id和[]byte类型的Req。注意，在这里，这个Req字段的值就应该代表与某个ServerReq类型值对应的json格式的普通文本数据。下面是*TcpComm类型的BuildReq方法的具体实现：

```
func (comm *TcpComm) BuildReq() loadgenlib.RawReq {
    id := time.Now().UnixNano()
    sreq := ServerReq{
        Id: id,
        Operands: []int{
            int(rand.Int31n(1000)),
            int(rand.Int31n(1000))},
        Operator: func() string {
            op := []string{"+", "-", "*", "/"}
            return op[rand.Int31n(100)%4]
        }(),
    }
    bytes, err := json.Marshal(sreq)
    if err != nil {
        panic(err)
    }
    rawReq := loadgenlib.RawReq{Id: id, Req: bytes}
    return rawReq
}
```

我们已经知道，ServerReq类型和loadgenlib.RawReq类型的Id字段的类型都是int64。因此它们可以被用来存储非常大的有符号整数。为了保持Id的唯一性，我们使用时间戳来作为它的值。这个时间戳需要很高的精度，可以表示到纳秒。因为在高并发的情况下，同一秒种甚至同一毫秒内都可能有很多原始请求被生成出来。time包的函数Now可以返回一个代表了调用它的那个时刻的Time类型值。该值有个名为UnixNano的方法，可以返回一个代表该时刻的纳秒数。这个纳秒数是从1970年1月1日的零时整开始算起的。我们就用它来充当Id的值。

怎样用复合字面量来初始化结构体类型的值，读者应该已经相当熟悉了。但是，这里比较特别的是，我们应该尽可能地体现出ServerReq类型值的随机性。这样才能提供更高的测试覆盖度。因此，对于Operands和Operator这两个字段的赋值，我们都使用了伪随机的算法。

首先来看Operands字段。原则上来说，我们可以为它赋予一个任意长度的[]int类型值。但是，我们在这里偷了一点懒，只是初始化了一个包含两个元素值的[]int类型值。不过，我们使用了math/rand包中的Int31n函数来生成其中的元素值。math/rand.Int31n函数可以在给定的范围内生成一个伪随机数。在这里，这个范围是[0,1000)。

对于Operator字段，我们使用一个针对匿名函数的调用表达式来代表它的值。这个匿名函数的唯一结果的类型是string类型的。它也是该表达式的类型。由于Operator字段的值的允许范围非常有限，所以我们很容易就此进行随机的选择。这里依然使用了math/rand.Int31n函数。

在生成好一个ServerReq类型值之后，我们需要把它转换为json格式的普通文本。这样才能够用它来给loadgenlib.RawReq类型的Req字段赋值。encoding/json代码包的函数Marshal可以实现这种转换。我们只要在调用它的时候把一个结构体类型值作为参数传给它就可以了。encoding/json.Marshal返回两个结果，第一个结果会是代表了转换后的文本的[]byte类型值，而第二个结果则是代表了可能发生的错误的error类型值。如果第二个结果值为nil，那么就说明转换是成功的。在转换失败的情况下我们引发了一个运行时恐慌。因为这种情况不应该发生。在载荷发生器中，这个运行时恐慌会被替换为一个代表了调用致命错误的调用结果。还记得*myGenerator类型的asyncCall方法中的那条defer语句吗？它会负责这一替换。

至于BuildReq方法的最后两条语句就很好理解了。我们在前面的准备工作的基础上生成了一个loadgenlib.RawReq类型值，并将它作为该方法的结果值返回。虽说我们拿到了这样一个值，但是实际上我们只需要其中的Req字段的值。而它的Id字段则只是为了方便载荷发生器的提取和鉴别而添加的。我们在讲解*myGenerator的asyncCall方法的时候展示过相关的代码。

3. Call方法

调用器的Call方法接受两个参数。参数req即代表了请求内容，其类型为[]byte。而time.Duration类型的参数timeoutNs则代表了超时时间。它的值应该与载荷发生器的timeoutNs字段的值一致。把它传给Call方法的含义是告诉调用器要进行超时判断。不过，这并不是强制的。因为载荷发生器已经采取了相应的措施来实时地判断调用超时。

下面，我们就来看看*TcpComm的Call方法的具体实现：

```
func (comm *TcpComm) Call(req []byte, timeoutNs time.Duration) ([]byte, error) {
    conn, err := net.DialTimeout("tcp", comm.addr, timeoutNs)
    if err != nil {
        return nil, err
    }
    _, err = write(conn, req, DELIM)
    if err != nil {
        return nil, err
    }
    return read(conn, DELIM)
}
```

代码包net的方法DialTimeout被用来建立网络通讯。它的特点是可以设定代表超时时间的纳秒数。这使得参数timeoutNs有了用武之地。当已达到超时时间但还未完成通讯的建立的时候，该方法的第二个结果值就会是一个代表了操作超时的error类型值。如果该方法的第二个结果值是nil，那么它的第一个结果值就会是一个代表了通讯连接的net.Conn类型值。我们需要在这里判断第二个结果值并做出相应的处理。如果通讯建立成功，我们就先将请求数据（即我们在BuildReq方法中生成的loadgenlib.RawReq类型值的Req字段的值）写入到连接中，然后在成功后再等待并从连接中读取响应数据。这两个操作分别由write函数和read函数负责。我们在讲Socket的时候已经详细讲解了相关的细节，所以在这里就不展开这两个函数的实现了。

这里还有一点需要注意。我们知道，基于TCP协议的通讯是使用字节流来传递上层给予的消息的。它会根据具体情况为消息分段。但是，这并不意味着参与通讯的另一方可以依此来感知消息的分界。因此，我们需要显式地为请求数据添加结束符。传给write方法和read方法的参数DELIM就代表了这个结束符。这两个方法会使用它来分离单个的请求或响应。在loadgen/testhelper代码包中有该常量的声明：

```
const (
    DELIM = '\n'
)
```

4. CheckResp方法

类型*TcpComm的方法CheckResp的声明是这样的：

```
func (comm *TcpComm) CheckResp(
    rawReq loadgenlib.RawReq,
    rawResp loadgenlib.RawResp) *loadgenlib.CallResult
```

调用CheckResp方法的时机是在载荷发生器接收到被测软件的响应之后。如果原始响应中没有携带任何错误，那么载荷发生器就会调用它来对原始响应进行进一步的检查，并根据检查结果设置其返回的调用结果的Code字段和Msg字段的值。

在CheckResp方法的方法体的开始处，我们需要先对将会被该方法返回的调用结果进行必要的初始化，像这样：

```
var commResult loadgenlib.CallResult
commResult.Id = rawResp.Id
commResult.Req = rawReq
commResult.Resp = rawResp
```

并且，在开始检查原始响应之前，我们必须要把参数rawReq的字段Req的值和rawResp的字段Resp的值转换为相应的结构体类型值。这需要用到json.Unmarshal函数。若以前者为例，则代码如下：

```
var sreq ServerReq
err := json.Unmarshal(rawReq.Req, &sreq)
```

我们需要把rawReq的Req字段的值和刚刚声明的ServerReq类型的变量的指针值作为参数传给json.Unmarshal函数。该函数在被执行完成之后会返回一个error类型值。如果在转换过程中发生了错误，那么代表结果的变量err的值将会是非nil的。若发生这种情况，那么我们会这样做：

```

if err != nil {
    commResult.Code = loadgenlib.RESULT_CODE_FATAL_CALL
    commResult.Msg =
        fmt.Sprintf("Incorrectly formatted Req: %s!\n", string(rawReq.Req))
    return &commResult
}

```

可以看到，我们在对调用结果`commResult`的字段`Code`和`Msg`进行必要的设置后，直接将它作为当前方法的结果返回了。注意，因为`rawReq`的`Req`字段的值就是由相应的`ServerReq`类型值经调用`json.Marshal`函数而得来的，所以此处的转换不应该发生任何错误。所以，如果发生了错误，那么我们会视它为一个致命的调用错误。

对于`rawResp`的字段`Resp`的值的转换，我们使用同样的方法。只不过在出错时，我们为`commResult`的字段`Code`和`Msg`赋予不同的值。相关代码如下：

```

var sresp ServerResp
err = json.Unmarshal(rawResp.Resp, &sresp)
if err != nil {
    commResult.Code = loadgenlib.RESULT_CODE_ERROR_RESPONSE
    commResult.Msg =
        fmt.Sprintf("Incorrectly formatted Resp: %s!\n", string(rawResp.Resp))
    return &commResult
}

```

限定标识符`loadgenlib.RESULT_CODE_ERROR_RESPONSE`代表了响应内容错误时的响应代码。

在上述工作完成之后，`*TcpComm`类型的`CheckResp`方法中的代码就开始对`sresp`变量的正确性进行检查。具体的检查项目如下。

- ❑ 检查`sresp`的`Id`字段的值是否与变量`sreq`的`Id`字段的值相等。也就是说这个原始响应是否与该原始请求相对应。如果不是，那么就说明这里的请求和响应是不匹配的。该项检查未通过。
- ❑ 检查`sresp`的`Err`字段的值是否为非`nil`。如果是，就说明被测软件在处理请求的过程中发生了错误。该项检查未通过。
- ❑ 检查变量`sresp`的`Result`字段的值是否正确。也就是说，它是否为原始请求中的运算符施加于同在其中的若干个运算数之后所得到的结果值。我们在这里用到的方法应该与被测软件采用的运算方法等效。若该字段的值不正确，则不能通过该项检查。

只要某一项检查未通过，那么后续的检查就会被忽略。该方法会立即根据实际情况设置`commResult`的变量`Code`和`Msg`的值，并将`commResult`作为结果值返回。若上述检查都通过了，则说明原始响应`sresp`是完全正确的。这时，该方法同样会设置调用结果的相应字段：

```

commResult.Code = loadgenlib.RESULT_CODE_SUCCESS
commResult.Msg = fmt.Sprintf("Success. (%s)", sresp.Formula)

```

设置完成后，`CheckResp`方法也同样会把`commResult`变量的值作为其结果值返回。

至此，我们完成了对结构体类型`TcpComm`以及为了实现`loadgenlib.Caller`接口而声明的公开方法的编写。下面，我们就使用这个调用器实现来对载荷发生器进行测试。

5. 测试载荷发生器

为了测试载荷发生器的实现类型`*myGenerator`，我们应该首先建立一个测试源码文件。与`myGenerator`类型相关的代码都被放到了`loadgen`代码包的库源码文件`gen.go`中。因此，与之相对应的测试源码文件就应该被命名为`gen_test.go`，且同在`loadgen`代码包中。

我们在第5章讲过，若要编写被用于测试的代码就需要用到`testing`代码包中的API。并且，在测试源码文件中，被用来进行功能测试的函数的名称应该以“Test”为前缀，并接受`*testing.T`类型的参数。因此，我们在`gen_test.go`文件中声明了这样一个函数：

```
func TestStart(t *testing.T)
```

下面我们来编写该函数的函数体。首先，为了让载荷发生器在时机到来之时尽可能快地将载荷发送出去，我们使用`runtime`代码包的`GOMAXPROCS`函数来设置该测试可以使用的P的最大数量。我们在前面说过，设置过大的P最大数量并无益于并发程序的性能。所以，我们只根据当前计算机的CPU核心数量来设置确定该数量。`runtime.GOMAXPROCS`函数的调用代码如下：

```
// 设置P最大数量
runtime.GOMAXPROCS(runtime.NumCPU())
```

注意，在测试载荷发生器之前，我们需要先行启动被测软件。代码如下：

```
// 初始化服务器
server := thelper.NewTcpServer()
defer server.Close()
serverAddr := "127.0.0.1:8080"
t.Logf("Startup TCP server(%s)...\\n", serverAddr)
err := server.Listen(serverAddr)
if err != nil {
    t.Fatalf("TCP Server startup failing! (addr=%s)!\\n", serverAddr)
    t.FailNow()
}
```

为了方便，我们在导入代码包的时候把`loadgen/testhelper`和`loadgen/lib`的别名分别设定为了`thelper`和`loadgenlib`。另外，由于我们在前面并没有展开被测软件的实现代码，所以在这里会对初始化并启动被测软件的过程进行简单的描述。

函数`thelper.NewTcpServer`的功能是创建并初始化一个TCP服务器（即我们所说的被测软件）。紧随其后的`defer`语句保证了在该功能测试方法结束之前该TCP服务器会被关闭。我们指定该服务器的监听IP为127.0.0.1，监听端口为8080，并以此网络地址来启动这个服务器。如果它在被启动的过程中有错误发生，那么我们就调用`t.Fatalf`函数打印出错误信息并使当前的测试立即失败。

在这个TCP服务器被启动成功之后，我们就需要着手调用器的初始化工作了。首先，我们使用`thelper.NewTcpComm`函数来创建和初始化一个基于TCP通讯协议的调用器。我们在前面未提及该函数的原因是它相当地简单。该函数的声明如下：

```
func NewTcpComm(addr string) loadgenlib.Caller {
    return &TcpComm{addr: addr}
}
```

注意，`thelper.NewTcpComm`函数的结果的类型为`loadgenlib.Caller`。这是为了确保`*TcpComm`

是该接口的一个实现类型。

据此，我们应该这样来初始化需要被传给载荷发生器的调用器：

```
// 初始化调用器
comm := thelper.NewTcpComm(serverAddr)
```

当然，除了调用器之外，初始化载荷发生器时还需要其他一些参数。这些参数的声明如下：

```
resultCh := make(chan *loadgenlib.CallResult, 50)
timeoutNs := 3 * time.Millisecond
lps := uint32(200)
durationNs := 12 * time.Second
```

变量`resultCh`代表了调用结果通道。我们把它的容量设置为50，实际上并没有什么特殊的原因。在该功能测试方法中，我们会在启动载荷发生器之后立即不间断地尝试从该通道中接收元素值。因此，原则上说，这里的调用结果通道的容量可以被设置得很小。具体的数值取决于相应的发送操作和接收操作进行的时机和频率。而这里设定的容量值只是一个宽泛的估值而已。

对于在`resultCh`下面的那3个变量，我们应该已经很熟悉了。使用这些参数，我们就可以这样创建出一个载荷加载器：

```
gen, err := NewGenerator(
    comm,
    timeoutNs,
    lps,
    durationNs,
    resultCh)
if err != nil {
    t.Fatalf("Load generator initialization failing: %s.\n",
        err)
    t.FailNow()
}
```

我们在这里使用了`loadgen`包中的`NewGenerator`函数。要记得检查它返回的第二个结果值。如果`err`不为`nil`，那么我们就只好立即停止当前测试了。否则，就可以这样启动载荷发生器`gen`：

```
go gen.Start()
```

我们以异步的方式来执行`gen.Start`方法的原因是，可以避免因调用结果通道`resultCh`被填满而导致的相关Goroutine（即载荷发生器中的那些专用Goroutine）的阻塞。如果我们不在`gen.Start`方法被执行的同时进行相应次数的针对`resultCh`通道的接收操作，那么它们就会被一直阻塞。还记得吗？在专用Goroutine中，如果对调用结果通道的发送操作未完成，那么它就不会将Goroutine票归还给Goroutine票池。如果与此同时Goroutine票池中的Goroutine票被用尽了，那么就会使后续的异步调用过程无法进行。进一步说，载荷发生器会先确保拿到一张Goroutine票，然后再启用一个专用Goroutine来异步地执行调用过程。如果Goroutine票池空了，那么载荷发生器的控制流程就会被永久地阻塞在对它的`asyncCall`方法的某次调用操作上。这样的话，无论是开启节流阀还是发送停止信号都不会使载荷发生器产生任何响应。更糟的是，这也会导致像功能测试函数`TestStart`这样的载荷发生器的使用方也被永久阻塞！

我意识到，这是一个严重的BUG！它即是我们测试载荷发生器（实际上，还没真正开始）的

过程中发现的第一个问题。我们不应该仅仅通过异步的执行`gen.Start`方法来绕过这个问题，而应该立即解决它！

清除这个BUG其实并不困难，我们只需将载荷发生器的控制流程以及后续接收调用执行计数等操作也异步化，即把在它的`Start`方法中的如下代码：

```
// 生成载荷
logger.Infof("Generating loads...")
gen.genLoad(throttle)

// 接收已发载荷总数
genCount := <-gen.endSign
gen.status = lib.STATUS_STOPPED
logger.Infof("Stopped. (genCount=%d)\n", genCount)
```

改为：

```
go func() {
    // 生成载荷
    logger.Infof("Generating loads...")
    gen.genLoad(throttle)

    // 接收调用执行计数
    callCount := <-gen.endSign
    gen.status = lib.STATUS_STOPPED
    logger.Infof("Stopped. (callCount=%d)\n", callCount)
}()
```

这样会使载荷发生器的`Start`方法在被调用后，不会等待控制流程的执行的完成，而是直接返回。我们刚刚发现的那个严重的BUG就这样被解决了。

在清除了这个BUG之后，我们继续编写`TestStart`函数。现在，我们可以直接调用`gen.Start`方法了：

```
// 开始！
t.Log("Start load generator...")
gen.Start()
```

该方法会很快的返回。

在启动载荷发生器之后，我们就要开始对调用结果进行收集和展示了。载荷发生器中启用的每一个专用Goroutine都会在调用过程结束之后向调用结果通道`resultCh`发送调用结果。因此，我们在这里只需做到及时地接收它们即可，如下所示：

```
// 显示调用结果
countMap := make(map[loadgenlib.ResultCode]int)
for r := range resultCh {
    countMap[r.Code] = countMap[r.Code] + 1
    if printDetail {
        t.Logf("Result: Id=%d, Code=%d, Msg=%s, Elapse=%v.\n",
            r.Id, r.Code, r.Msg, r.Elapse)
    }
}
```

这段代码不断地从`resultCh`通道中获取调用结果。其中，`printDetail`是我们在`gen_test.go`文

件中声明的一个bool类型的变量。如果该变量的值为true，那么上面的代码就会打印出接收到的所有调用结果的细节信息。另一方面，这段代码总会以调用结果的Code字段的值作为依据对它们进行分类和计数，并把计数信息存入countMap变量代表的字典中。这样的统计让我们可以方便地展示出各类调用结果的数量：

```
var total int
t.Log("Code Count:")
for k, v := range countMap {
    codePlain := loadgenlib.GetResultCodePlain(k)
    t.Logf("  Code plain: %s (%d), Count: %d.\n",
        codePlain, k, v)
    total += v
}
```

函数loadgenlib.GetResultCodePlain的功能是返回一个string类型值。该值实际上是一个短语。它解释了作为该函数参数的响应代码的含义。在各类调用结果之中，我们最关心的应该是响应代码为loadgenlib.RESULT_CODE_SUCCESS的调用结果的数量及其占调用结果总数的比例。因此，有了下面这段代码：

```
t.Logf("Total load: %d.\n", total)
successCount := countMap[loadgenlib.RESULT_CODE_SUCCESS]
tps := float64(successCount) / float64(durationNs/1e9)
t.Logf("Loads per second: %d; Treatments per second: %f.\n", lps, tps)
```

这里的tps的含义是被测软件平均每秒有效的处理（或称响应）载荷的数量。

现在我们来看看该测试的实际输出。若printDetail变量的值为false，那么在我们通过go test命令执行gen_test.go中的TestStart函数之后，标准输出上会出现如下内容（这里只截取了最后一部分）：

```
--- PASS: TestStart (12.07 seconds)
    gen_test.go:21: Startup TCP server(127.0.0.1:8080)...
    gen_test.go:37: Initialize load generator (timeoutNs=3ms, lps=200, durationNs=12s)...
    gen_test.go:51: Start load generator...
    gen_test.go:65: Code Count:
    gen_test.go:69:   Code plain: Success (0), Count: 2364.
    gen_test.go:69:   Code plain: Call Timeout Warning (1001), Count: 28.
    gen_test.go:73: Total load: 2392.
    gen_test.go:76: Loads per second: 200; Treatments per second: 197.000000.
PASS
ok      loadgen 12.278s
```

从这些输出内容上我们可以看出，在这次对被测软件的性能测试中，它有效处理的载荷的数量的比例为98.5%。更确切地说，有1.5%的载荷的响应没有被测软件及时送回。

好了，在TestStart函数中，我们主要对载荷发生器的启动流程、控制流程和自动停止流程进行了测试。从测试日志上看，载荷发生器表现良好。

另一方面，我们已经知道，载荷发生器还可以被手动的停止。下面我们就来对这部分功能进行测试。该测试由gen_test.go文件中的TestStop函数代表。

在TestStop函数中，被用于初始化服务器、调用器和载荷发生器，以及启动载荷发生器的代

码与TestStart函数中的代码几乎一致。唯一不同的是，被测软件的网络地址由127.0.0.1:8080被改为了127.0.0.1:8081。这是为了彻底地避免两个测试的相互干扰。

除此之外，在TestStop函数的显示调用结果的部分中，我们也稍微做了一些修改：

```
// 显示调用结果
countMap := make(map[loadgenlib.ResultCode]int)
count := 0
for r := range resultCh {
    countMap[r.Code] = countMap[r.Code] + 1
    if printDetail {
        t.Logf("Result: Id=%d, Code=%d, Msg=%s, Elapse=%v.\n",
            r.Id, r.Code, r.Msg, r.Elapse)
    }
    count++
    if count > 3 {
        gen.Stop()
    }
}
```

我们使用变量count来实时地对接收到的调用结果进行计数。并且，我们在第4次迭代即将完成的时候手动地停止载荷发生器。

按照我们的预期，载荷发生器的停止会导致调用结果通道的关闭，从而导致这个for代码块的执行的结束。但是，当我们真正运行该测试的时候却发现，情况并不是这样。相关的测试日志如下：

```
=== RUN TestStop
2014/04/18 16:11:57 [INFO] loadgen.NewGenerator : (gen.go:34) - New a load generator...
2014/04/18 16:11:57 [INFO] loadgen.NewGenerator : (gen.go:35) - Checking the parameters...
2014/04/18 16:11:57 [INFO] loadgen.NewGenerator : (gen.go:66) - Passed. (timeoutNs=3ms, lps=200,
durationNs=12s)
2014/04/18 16:11:57 [INFO] loadgen.(*myGenerator).init : (gen.go:75) - Initializing the load
generator...
2014/04/18 16:11:57 [INFO] loadgen.(*myGenerator).init : (gen.go:87) - Initialized. (concurrency=1)
2014/04/18 16:11:57 [INFO] loadgen.(*myGenerator).Start : (gen.go:215) - Starting load generator...
2014/04/18 16:11:57 [INFO] loadgen.(*myGenerator).Start : (gen.go:221) - Setting throttle (5ms)...
2014/04/18 16:11:57 [INFO] loadgen.(*myGenerator).Start : (gen.go:242) - Generating loads...
2014/04/18 16:11:57 [INFO] loadgen.(*myGenerator).handleStopSign : (gen.go:186) - Closing result
channel...
2014/04/18 16:11:57 [INFO] loadgen.func·006 : (gen.go:256) - Stopped. (callCount=3)
2014/04/18 16:12:07 [INFO] loadgen.func·004 : (gen.go:228) - Stopping load generator...
*** Test killed: ran too long (10m0s).
FAIL    loadgen 600.287s
```

可以看到，无论是载荷发生器的创建、初始化和启动还是载荷的生成和发送都是正常的。我们需要特别关注的是最后5行内容。倒数第五行和第四行的内容表示，在调用结果通道被关闭之后载荷发生器被停止，且停止之时的调用执行计数是3。这里的计数表示了控制流程（由载荷发生器的genLoad方法代表）中被用来异步地执行调用过程的那个循环，在第三次迭代即将结束的时候被终止了。

请注意，倒数第四行日志并不是载荷发生器的Stop方法中的日志记录语句打印出来的（该方

法中没有这类语句), 而是它的Start方法中的最后面的那条语句为之的。

这样是正确的吗? 在回答这个问题之前, 让我们再回顾一下载荷发生器的实现细节。还记得吗? 我们在载荷发生器的Start方法和Stop方法中都包含了针对endSign通道(也就是那个被用来告知停止信号已被处理完毕的通道)的接收操作。但是, 在载荷发生器即将处理完停止信号的时候只会向endSign通道发送一个元素值。如果我们在载荷发生器自动停止之前通过调用其Stop方法来手动停止它, 那么endSign通道中的那个唯一的元素值就会被Start方法中的那个接收操作取走。因为该接收操作是先被进行并阻塞的。这就导致了Stop方法中的那个接收操作被永久阻塞。因为不会再有任何代码向endSign通道发送元素值了。显然, 我们先前在此处的设计并不合理。

由于这样的设计缺陷, 在前面的测试日志中的倒数第四行内容被打印出来之后, 功能测试函数TestStop的执行就被停滞了。载荷发生器的控制流程所在的Goroutine在记录该行日志之后就被运行结束了。而调用它的Stop方法的代码所在的Goroutine却被阻塞了。这个Goroutine即是执行TestStop函数的那一个。

可以看到, 在倒数第四行内容被打印出来的数十秒之后, 倒数第三行内容出现了。这是由于在载荷发生器的Start方法中初始化的定时器执行了我们给定的那个匿名函数。但是, 在这时, 执行这个匿名函数已经没有任何意义了。因为载荷发生器的控制流程的执行已经结束了。而对TestStop函数的执行依然处于停滞状态。当前的功能测试已经无法完成, 直到我们强行终止其所属的进程。如果我们没有这样做, 那么等到该测试的运行时间达到10分钟之后, go test命令会自己结束它并宣告测试失败。这体现在前面展示的测试日志的最后两行内容上。

该问题是在我们对载荷发生器的功能进行测试的过程中发现的第二个问题。我们怎样才能解决它呢? 我们已经知道了这个问题的症结所在, 那就是两种载荷发生器停止方式之间的冲突。它们都以从endSign通道中成功接收到一个元素值作为载荷发生器已经被停止的确认。因此, 最简单的方式就是, 在即将处理完停止信号的时候向该通道连续发送两个相同的元素值(即表示了调用执行计数的值)。这样的解决方案既简单又清晰。

具体的代码修改工作可被分为两步。

(1) 将endSign通道的容量扩充至2。这样, 我们就可以连续的向它发送两个元素值且不用担心会被阻塞了。相关代码在*myGenerator类型的Start方法中。修改后的代码为:

```
// 初始化完结信号通道
gen.endSign = make(chan uint64, 2)
```

(2) 将包含在载荷发生器的genLoad方法中的针对endSign通道的那两条发送语句都删除掉。然后, 在handleStopSign方法的方法体的最后面添加两条与之相同的语句。修改后的handleStopSign方法和genLoad方法如下:

```
func (gen *myGenerator) handleStopSign(callCount uint64) {
    gen.cancelSign = 1
    logger.Infof("Closing result channel...")
    close(gen.resultCh)
    gen.endSign <- callCount
    gen.endSign <- callCount
}
```

```

func (gen *myGenerator) genLoad(throttle <-chan time.Time) {
    callCount := uint64(0)
Loop:
    for ; ; callCount++ {
        select {
        case <-gen.stopSign:
            gen.handleStopSign(callCount)
            break Loop
        default:
        }
        gen.asyncCall()
        if gen.lps > 0 {
            select {
            case <-throttle:
            case <-gen.stopSign:
                gen.handleStopSign(callCount)
                break Loop
            }
        }
    }
}

```

在进行完上述修改之后，我们再使用`go test`运行测试源码文件`gen_test.go`就会得到正常的结果。

好了，我们现在可以说载荷发生器的实现类型`*loadgen.myGenerator`，以及我们自定义的调用器的实现类型`loadgen/testhelper.*TcpComm`在功能上已经是正确的了。作为一款简单的性能测试工具，该程序已经可以被投入使用了。当然，关于该程序是否可以完全正确地模拟极高的并发量，我们还没有进行深入的测试。读者如果有兴趣的话，可以为这个载荷发生器的实现类型编写性能测试。

除此之外，我们还可以为载荷发生器编写出面向命令行以及GUI（Graphical User Interface，图形用户界面）的用户接口。这可以大大增强其易用性。也许，我们需要为此添加一些API以使载荷发生器对扩展更加开放。

总之，我们在本节实现并详述的程序可以被用于简单的性能测试。它向我们展现了Go语言的Goroutine和Channel的常规用法和一些小技巧。从这个角度讲，它还是非常具有参考意义的。读者可以试着修改或扩展它以满足自己的实际需要。

7.4 本章小结

本章讲解了我们基于Go语言的并发编程模型编写程序时最常用到的两个工具——Goroutine和Channel的各种使用方法和技巧。我希望大家能够通过这些描述和说明真正领会到它们的真谛，并可以熟练地使用它们编写并发程序。更进一步地，我们可以将它们与背后的原理和支撑机制结合起来看待。这样一来，我们在编程过程中就更加胸有成竹、游刃有余了。

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了。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>)</i>而没有立即看到The lock is locked. (G<i>)</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于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，我们还需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的话，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1111111111111111111111111111011101。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：

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。

该方法的方法`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。

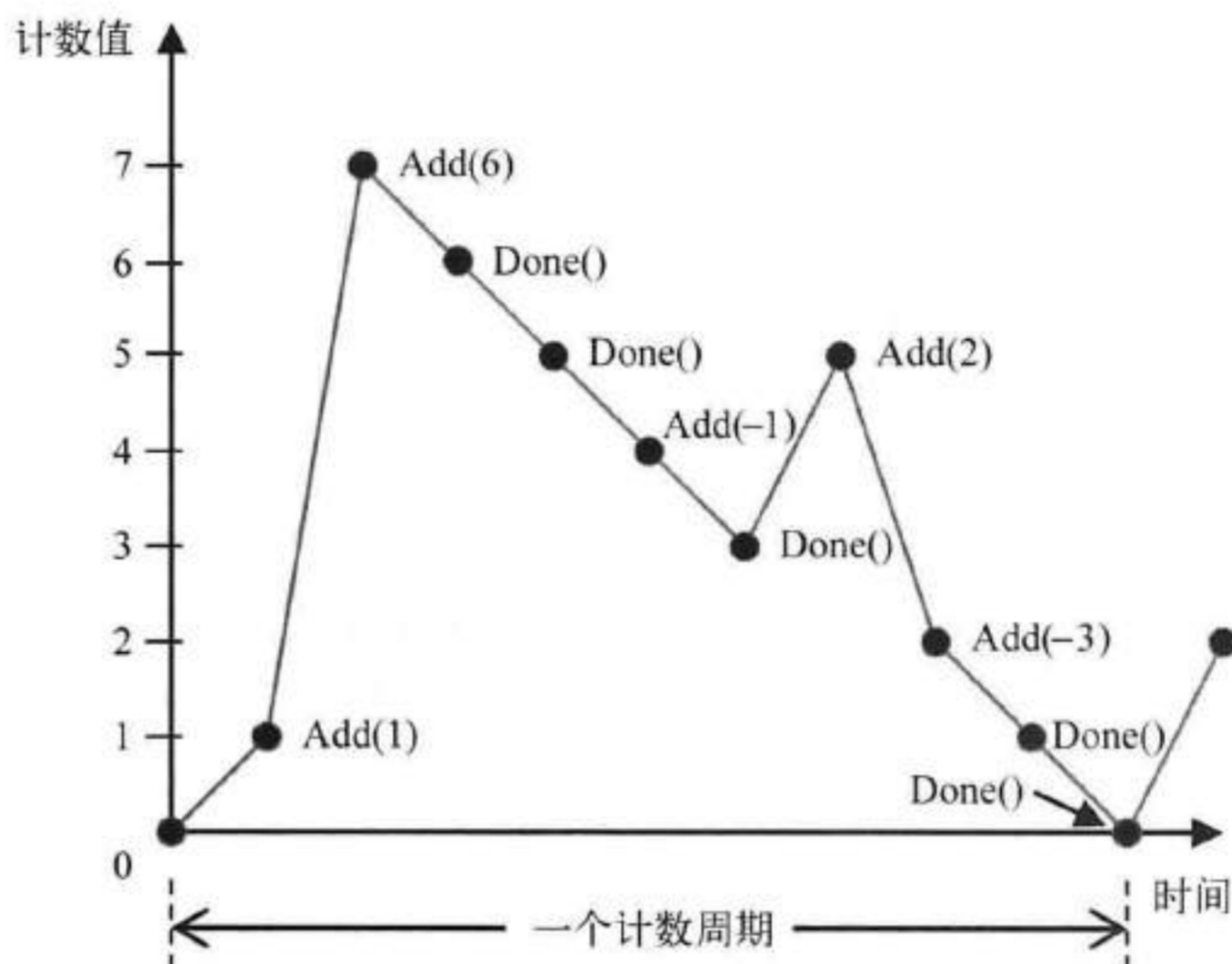


图8-1 sync.WaitGroup类型值的计数值的变化曲线示意

□ 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和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阱。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on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on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。
Elms() []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、Elms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()
    }
    m1 := 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, m1)
    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**m1 := 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。

Part 4

第四部分

编程实战

本部分会通过编写一个完整的示例来使你对 Go 语言编程（尤其是并发编程）的相关知识有更加深刻和透彻的理解。与我们之前展示的示例相比，这个示例的代码量级更大、设计更复杂，同时涉及的 Go 语言编程知识也更多。可以说，它是我们用代码写成的对前面所有内容的一个复习和总结。并且，在我们完成这个示例之后，还可以把它作为开箱即用的工具来使用。下面，就让我们进入具体的章节，并在了解需求之后开始编程。

一个网络爬虫框架的设计和实现

在前面的几个部分中，我们已经对Go语言的特性、开发环境、语法、各种编程方法和技巧等方面进行了大量、详尽的介绍和梳理，并且还着重描述和说明了Go语言在并发编程方面的特性，以及它与早先的并发编程模型相比的优势。我在讲解上述知识的同时，展示了许多示例来辅助和巩固读者对它们的理解。这些示例有些穿插于字里行间，而有些则作为独立的实战演练环节出现。相信读者在仔细阅读这些示例中的代码并按照要求进行练习之后，就能够编写出可以正确甚至高效运行的应用程序了。

作为本书的最后一章，我们将完整地展现一个应用程序的设计、编写和试用的过程，从而将本书讲到的所有Go语言知识和编程技巧从头至尾地贯穿起来。在这个过程中，我会帮助你回忆起Go语言的诸多特性（尤其是并发编程方面的特性）以及应用方法，并设法增强你对它们的记忆和理解。同时，通过这个程序的编写和讲解，我也会让读者真正明白怎样将Go语言用到实处。在读过本章之后，我们会得到一个开箱即用并且易于扩展的工具类程序。由本章的题目可知，这个工具类程序即是一个网络爬虫框架。

之所以选择这个主题，是因为它不会掺杂太多与本书主题关系不大的逻辑和功能。例如，一些面向某一个领域的数据结构和通讯协议的定义，以及其他一些被用于构建复杂的应用系统的细枝末节。这样做可以让我们把焦点一直落在Go语言上。而对于那些更加宏观的应用系统架构设计方面的内容，我们就不在此赘述了。它们的范围太大且与本书主题关联甚微。读者完全可以在熟悉Go语言之后，再通过其他途径了解怎样构建大规模、高复杂度的应用程序或系统，并以此在更大的范围内发挥Go语言的威力。

现在，让我们立即开启这次Go语言编程实战之旅。

9.1 网络爬虫与框架

简单来说，网络爬虫是互联网终端用户的模仿者。它模仿的主要对象有两个，一个是坐在计算机前使用网络浏览器访问互联网内容的人类用户，另一个是网络浏览器。网络爬虫会模仿人类用户输入互联网中某个网站的网络地址，并试图访问该网站上的内容，还会模仿网络浏览器根据

给定的网络地址去下载相应的网络页面（以下简称网页）。在下载对应的网页之后，网络爬虫会根据预设的规则去分析和筛选网页中的内容。这些被筛选出的内容会马上被进行特定的处理，并以某种形式留存下来。与此同时，网络爬虫还会像人类用户点击某个网页中他感兴趣的链接那样，继续访问和下载相关联的其他网页，然后再对这些网页的内容进行同样或类似的处理。

依据上面这几句概述，读者可能已经意识到：网络爬虫实际上就是一个爬取和分析网页的程序。它应该去自动下载使用者需要的网页，并根据使用者的意愿从中提取出相关的链接、筛选出可用的内容，以及分析、统计或存储它们。注意，这其中的关键词是“自动”和“根据意愿”。“自动”的含义是，网络爬虫在被启动后应该自己完成整个网页爬取过程而无需人工干预，并且还能够的过程结束之后自动停止。而“根据意愿”则是说，网络爬虫应该最大限度地允许使用者对其爬取过程进行定制。

乍一看，要做到自动地爬取貌似并不困难。我们只需让网络爬虫根据相关的网络地址不断地下载对应的网页即可。但是，窥探其中，却有很多细节需要我们去进行特别的处理。

- 有效网络地址的判定。
- 有效网络地址的边界定义和校验。
- 重复的网络地址的过滤。
- 非直接的网络地址的发现和提取。

在这些细节当中，有的是比较容易处理的，而有些则需要额外的解决方案。例如，我们都知道，基于HTML的网页中可以包含button标签。若我们想在终端用户点击该按钮的时候使网络浏览器跳转到另一个网页，则可以找到很多种方法。其中，非常常用的一个方法就是为该标签添加onclick属性并把一些JavaScript语言的代码作为它的值。虽然这个方法如此常用，但是我们想让网络爬虫可以从中提取出有效的网络地址却是比较困难的。因为这涉及了对相应的JavaScript语言程序的理解。可以作为onclick属性的值的JavaScript语言代码的书写方法繁多。要想让网络爬虫完全理解它们，恐怕就必须要用到某个JavaScript语言解析器的实现了。

另一方面，由于互联网对人们生活和工作的全面渗透，我们可以通过各种途径了解到的各式各样的网络爬虫实现。它们都有着截然不同且复杂独特的逻辑。这些复杂的逻辑主要针对的是如下几个方面。

- 在根据网络地址组装HTTP请求时需要为其设定的各种各样的头信息和主体内容。
- 对网页中的链接和内容进行筛选时需要用到的各种条件。这里所说的条件包括提取条件、过滤条件和分类条件，等等。
- 处理筛选出的内容时涉及的各种处理方法和步骤。

这些逻辑绝大多数都与网络爬虫使用者当时的意愿有关。换句话说，它们都与具体的使用目的有着紧密的联系。也许它们并不应该作为网络爬虫的核心功能。

经过上面的分析，我发现：我们更应该编写一个容易被定制和扩展的网络爬虫框架，而不是一个针对特定网页爬取目的的网络爬虫实现。这样才能使该应用程序被称为一个实用工具。

好了，既然我们撤去了那些容易喧宾夺主的特定功能，也明确了本章编写的应用程序应该被作为一个网络爬虫框架而存在，那么接下来我们就应该搞清楚这个框架都应该或可以做哪些事。

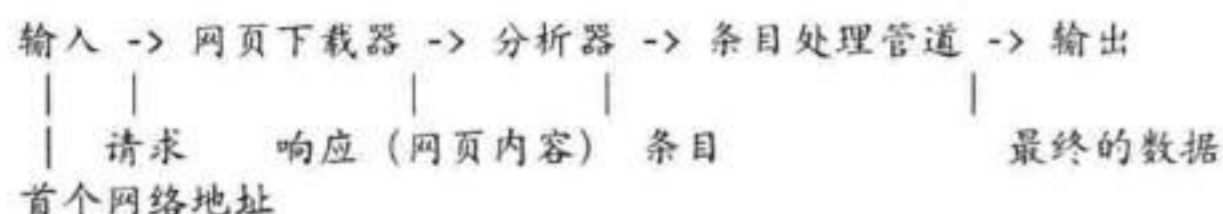
这也能够让我们进一步明确它的功能、用途和意义。

9.2 功能需求和分析

通过阅读上一节，读者应该可以大致了解到我们所说的网络爬虫框架的功用。概括来讲，网络爬虫框架会反复地执行如下步骤直至触碰到停止条件。

- “网页下载器”下载与给定网络地址对应的网页内容。其中，在被用于获取网页内容的“请求”的组装方面，网络爬虫框架应该为用户尽量预留出定制接口。使用者可以使用这些接口自定义“请求”的组装方法。
- “分析器”分析下载到的网页内容，并从中筛选出可用的内容（以下称为“条目”）和需要访问的网页的链接（也就是该网页的网络地址）。其中，在被用于分析和筛选网页内容的规则和策略方面，应该由网络爬虫框架提供灵活的定制接口。换句话说，由于只有使用者自己才知道他们真正想要的是什么，所以应该允许他们对这些规则和策略进行深入的定制。而网络爬虫框架仅需要规定好定制的方式即可。
- “分析器”（也可能是使用者自定义的程序）把筛选出的“条目”发送给“条目处理管道”。同时，它会把发现的新的网络地址和其他一些信息组装成新的请求，然后把这些请求发送给“网页下载器”。在此步骤中，我们应该过滤掉一些不符合要求的网络地址，比如忽略超出有效网络地址边界或重复的网络地址。

读者可能已经注意到，在这几个步骤的描述中，我们使用引号突出展示了几个名词，即网页下载器、请求、分析器、条目和条目处理管道。其中，请求和条目都代表了某类数据，而其他3个名词则代表了处理数据的子程序（或称处理模块）。它们与在前面已经提到过的网页内容（或称对请求的响应）共同描述出了对应于单一网络地址的数据的流转方式。具体如下：



我们在前面说过，分析器除了会从网页内容中筛选出若干个条目之外，还会提取出一些新的请求。这就意味着，我们向网络爬虫输入的一个初始的网络地址往往会使得多个网页（一般是同一个网站中的不同网页）的内容被下载、筛选和处理。因此，网络爬虫程序在完整地执行一个爬取任务的过程中，其中数据的流转应该如图9-1所示。

从图9-1中我们可以清晰地看到每一个处理模块能够接受的输入和可以产生的输出。实际上，我们将要编写的网络爬虫框架就会以此为依据而形成几个相对独立的组件。当然，为了维护这些组件的运行和协作的有效性，该框架中还会存在其他一些子程序。关于这方面，我们在后面的小节中会陆续予以说明。

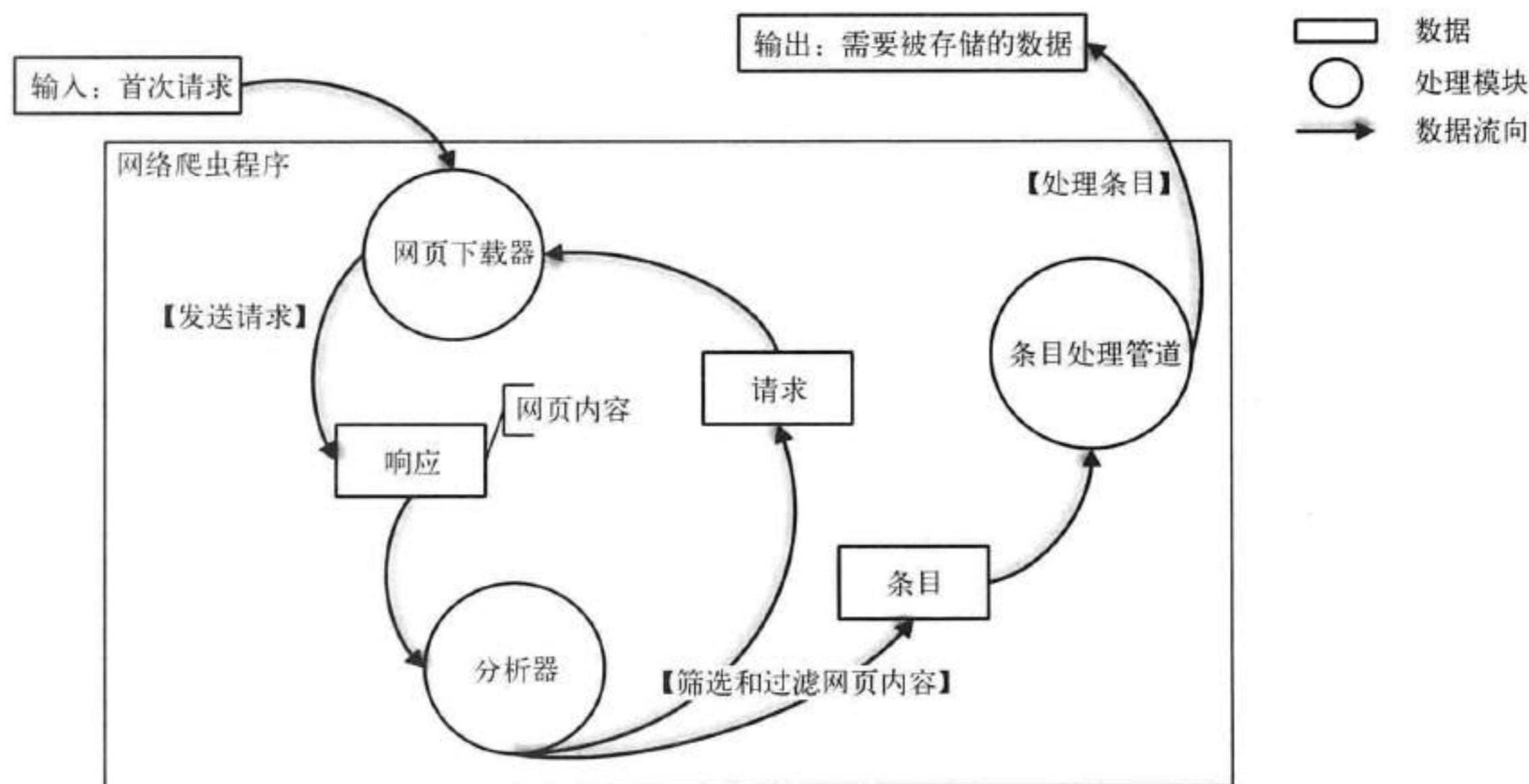


图9-1 起始于首次请求的数据流程图

在这里，我们需要再次强调一下网络爬虫框架与网络爬虫实现的区别。作为一个框架，该程序应该在每个处理模块中给予使用者尽量多的定制方法，而不去涉及各个处理步骤的实现细节。另外，框架应该更多地考虑使用者自定义的处理步骤在被执行过程中可能发生的各种情况和问题，并注意对这些问题的处理方式。这样才能在做到在易于扩展的同时保证框架的稳定性。这方面的思考和策略会体现在该网络爬虫框架的各个阶段的设计和编码实现之中。

好了，下面我们就会根据以上对功能需求和特性方面的分析来对这一程序进行总体设计。

9.3 总体设计

参看在上一节中展示的数据流程图可知，网络爬虫框架的处理模块有3个：网页下载器、分析器和条目处理管道。加之协调和调度这些处理模块的运行的控制模块，我们就可以明晰该框架的模块划分了。我们把这里提到的控制模块称为调度器。下面是这4个模块的各自承担的职责。

- **网页下载器**：接受作为输入的请求，并将该请求转换成HTTP请求；将该HTTP请求发送至与其中的网络地址对应的远程服务器；在HTTP请求发送完成之后，立即等待相应的HTTP响应的到来；在收到HTTP响应之后，将其封装成响应并作为输出返回给网页下载器的调用方。其中，被用于发送HTTP请求和接收HTTP响应的HTTP客户端程序的生成方式，应该可以由网络爬虫框架的使用方自行定义。另外，若在该子流程被执行期间发生了错误，应该立即以适当的方式告知网络爬虫框架的使用方。对于其他模块来讲，也应该是这样。
- **分析器**：接受作为输入的响应，并将该响应还原成HTTP响应；对该HTTP响应的状态和内容进行检查，并根据给定的规则进行筛选以及生成新的请求或条目；将生成的请求或

条目作为输出返回给分析器的调用方。在分析器的职责中，我们可以想到的能够留给网络爬虫框架的使用方自定义的部分并不少。例如，内容筛选的规则、生成请求和条目的方式，甚至是可以接受的HTTP响应状态的定义。在后面，我们会对这些可能自定义的部分进行一些取舍。

- **条目处理管道**：接受作为输入的条目，并对其执行一系列的处理步骤；条目处理管道中可以产出最终的数据；这个最终的数据可以在其中的某个处理步骤中被持久化（不论是本地存储还是发送给远程的存储服务器）以备后用。我们可以把上面所说的一系列处理步骤留给网络爬虫框架的使用方自行定义。这样，网络爬虫框架就可以真正地与条目处理细节脱离开来了。网络爬虫框架应该丝毫不关心这些条目会怎样被处理和持久化。它仅仅负责控制整体的处理流程。我们把负责单个处理步骤的程序称为条目处理器。条目处理器接受条目，并把被处理完成的条目返回给条目处理管道。条目处理管道会紧接着把该条目传递给下一个条目处理器，直至给定的条目处理器序列中的每个条目处理器都依次处理过该条目为止。
- **调度器**：调度器仅接受作为输入的首次请求，并且不会产生任何输出。调度器的主要职责是调度各个处理模块的运行。其中包括维护各个处理模块的实例、在不同的处理模块实例之间传递数据（请求、响应或条目），以及监控所有这些被调度者的状态，等等。有了调度器的支持，各个处理模块得以保持其职责的简洁和专一。而由于调度器是网络爬虫框架中最重要的一个模块，所以我们可能需要再编写出一些工具来支撑起它的功能。其中很多工具的实现是由我们在前几章中实现的代码演变而来的。读者在阅读后面的相应内容时会有所体会。

在弄清楚网络爬虫框架中的各个模块的职责之后，我们会发现该框架是以调度器为核心的。此外，为了并发执行的需要，除调度器之外的其他模块都可以是多实例的。它们应该由调度器创建、维护和调用。反过来讲，这些处理模块的实例会从调度器处接受输入，并在进行相应的处理之后将输出返回给调度器。最后，与另外两个处理模块相比，条目处理管道是比较特殊的。顾名思义，它应该是以流式处理为基础的。它的设计灵感来自于我们之前讲到过的Linux操作系统所提供的管道。我们可以不断地向该管道发送条目，而该管道则会让其中的若干个条目处理器依次处理每一个条目。我们可以很轻易地使用一些同步方法来保证条目处理管道的并发安全性，因此调度器可以只持有该管道的一个实例。不过，我们是否可以并发地向该管道发送条目还要根据那些条目处理器的实现方式来抉择。还记得吗？这些条目处理器是由网络爬虫框架的使用方提供的。

图9-2展示了调度器与各个处理模块的关系。我在图中加入了一个新的元素——中间件。实际上，我们之前所说的被用来支撑调度器的管理功能的那些工具就是中间件的一部分。更明确地讲，中间件不是一个完整的模块，而是一个工具集。调度器会通过特定的工具来实现对各个处理模块的控制，也会使用一些工具在多个处理模块之间传递各种类型的数据。由此，这些工具在网络爬虫框架中的作用可见一斑。它们是调度器与所有处理模块之间的桥梁。

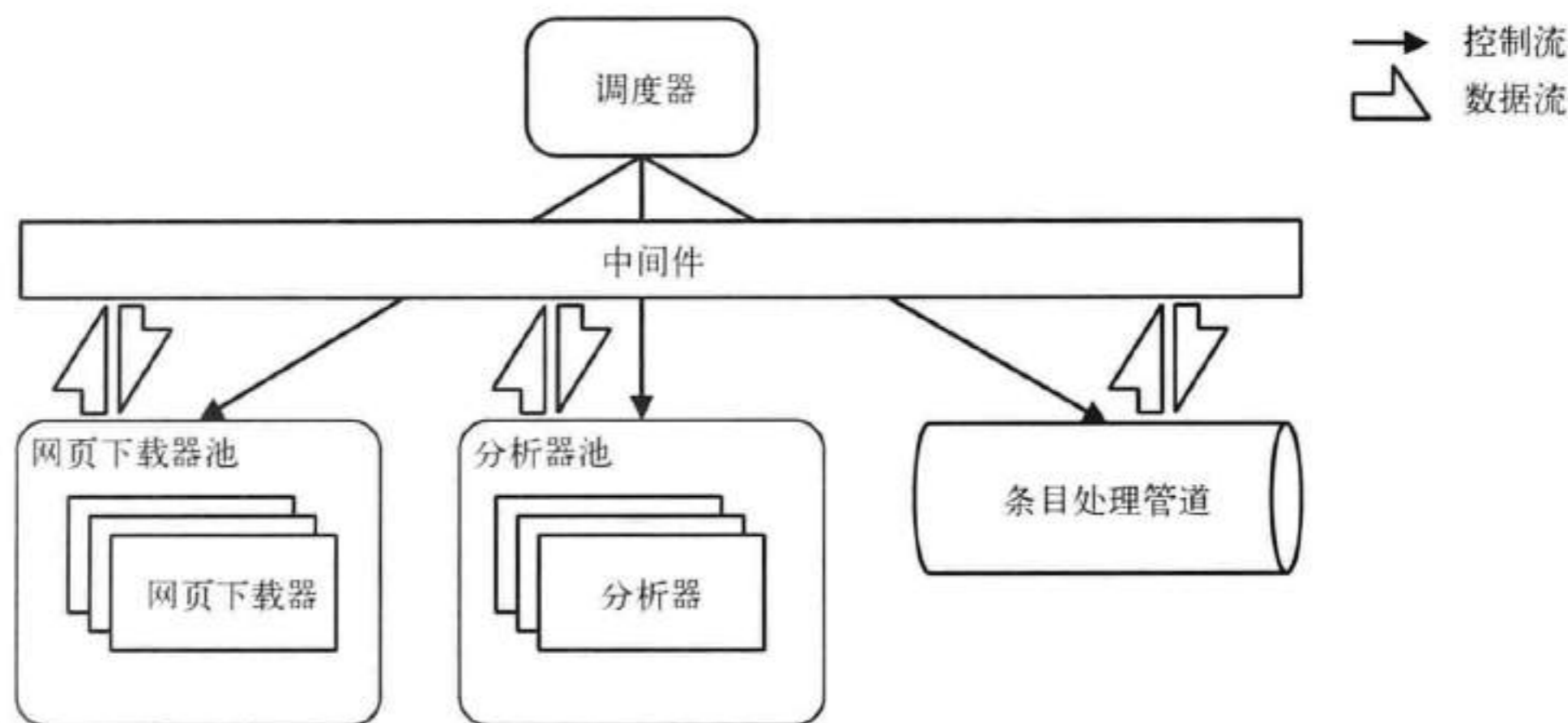


图9-2 调度器与各个处理模块的关系

至此，读者应该对网络爬虫框架的总体设计有了一个宏观上的认识。不过，我们还未提及在这个总体设计之下包含的大量的设计技巧和决策。这些技巧和决策不但与一些通用的程序设计原则有关，还涉及了很多依赖于Go语言自身的编程风格和方式。这也从侧面说明，由于几乎所有编程语言都有着非常鲜明的特点和比较擅长的领域，因此我们在设计一个需要由特定的编程语言实现的软件或程序的时候，多多少少会考虑到这门编程语言自身的特性。也就是说，软件设计不是与具体的编程语言毫不相关的。反过来讲，总会有一门或少数几门编程语言非常适合实现某一类的软件或程序。

我们在后面会看到，就本章所讲的网络爬虫框架而言，Go语言是非常适合作为其实现语言的。不过，在动手实现这个框架之前，我们会先讲一讲其中每一个部分（各个模块以及中间件）的设计方案。

9.4 详细设计

在本节中，我们会逐一讲解网络爬虫框架中的各个模块所涉及的数据结构和接口。在这个过程中，我们还会了解到控制和调度这些模块及其实例时可能需要用到的工具。

9.4.1 基本数据结构

为了承载和封装数据，我们需要先声明一些基本的数据结构。网络爬虫框架中的各个组件都会用到这些数据结构。所以，可以说它们是这一程序的基础。

我们在分析网络爬虫框架的需求的时候提到过这样几类数据：请求、响应、条目。下面我们就逐个讲解它们的声明和设计理念。

请求是被用来承载与某一个网络地址对应的HTTP请求的。它会由调度器或分析器生成并被传递给网页下载器。网页下载器会根据它从远程服务器下载相应的网页。因此，它应该有一个类

型为net/http代码包中的Request类型的字段。不过，为了减少不必要的零值生成（http.Request是一个结构体类型，它的零值不是nil）和实例复制，我们应该把*http.Request作为该字段的类型。下面是base.Request类型的声明的第一个版本：

```
// 请求。
type Request struct {
    httpReq *http.Request // HTTP请求的指针值。
}
```

我们把基本数据结构的声明都放在了goc2p项目下的webcrawler/base代码包中。因此，该类型的限定符是base.。

从我们已经提到的与此相关的功能需求来看，这样的声明已经足够了。不过，我们也说过网络爬虫应该能够在爬取过程结束之后自动停止。那么网络爬虫在对一个网站上的网页爬取到什么程度才能够算是爬取过程的结束呢？对网页爬取程度的一个比较常用的量化方法是计算每个被下载的网页的深度。网络爬虫应该可以根据最大深度的预设值忽略掉对“更深”的网页的下载操作。当所有在该最大深度范围内的网页都被下载完成的时候，就意味着爬取过程即将完成。待这些网页也都被分析和处理完成，就能够判定网络爬虫对爬取过程的执行的结束了。因此，为了记录网页的深度，我们还应该在base.Request类型的声明加入一个字段。综上所述，该类型声明的第二个版本如下：

```
// 请求。
type Request struct {
    httpReq *http.Request // HTTP请求的指针值。
    depth   uint32               // 请求的深度。
}

// 创建新的请求。
func NewRequest(httpReq *http.Request, depth uint32) *Request {
    return &Request{httpReq: httpReq, depth: depth}
}

// 获取HTTP请求。
func (req *Request) HttpReq() *http.Request {
    return req.httpReq
}

// 获取深度值。
func (req *Request) Depth() uint32 {
    return req.depth
}
```

我们希望base.Request类型的值是不可变的。也就是说，在一个base.Request类型的值被创建和初始化之后，当前代码包之外的代码不能对它的任何字段的值进行更改。一般对于这样的需求，我们会通过以下3个步骤来实现它。

(1) 把该类型的所有字段的访问权限都设置为包级私有。也就是说，要保证这些字段的名称的首字母均为小写。

(2) 编写一个创建和初始化该类型的值的函数。由于该类型的所有字段均不能被当前代码包之外的代码直接访问，所以它们自然也就无法为这样的字段赋值。这样也是我们需要在该类型声明所属的代码包内编写这样一个函数的原因。这类函数的名称一般都以“New”为前缀。它们会接受一些参数值并以此为基础初始化一个目标类型的值，然后再把该值返回给函数的调用方。

(3) 编写必要的用来获取某个字段的值的方法。这一步骤并不是必须的。原因可能是有的类型想要隐藏其实现细节，也可能是它的字段的类型（比如切片类型、字典类型、指针类型和一些结构体类型）不允许它们这样做。

显然，我们就是依照上述这3个步骤来编写base.Request类型声明的第二个版本的。注意，NewRequest函数的结果的类型是*base.Request，而不是base.Request。这样做的主要原因是，我们是使用*base.Request类型作为相关方法的接收者的类型的。

在这里，我们再专门说明一下base.Request类型的depth字段。理论上，uint32类型已经可以使depth字段的值足够大了。由于深度值不可能是负数，所以我们也不需要为此而牺牲正整数的部分取值范围。我们传递给调度器的首次请求的深度应该是0。这也是首次请求的一个标识。那么，后续请求的深度应该怎样计算和传递呢？假设网页下载器发出了首次请求并成功接收到了响应，经过分析器的分析，我们在其中找到了若干个新的网络地址并生成了新的请求，那么这两个新请求的深度就为1。比如，这里有两个新的请求：请求A和请求B。如果我们在接收并分析了请求A的响应之后得到了请求C，那么请求C的深度就是2。以此类推。我们可以把首次请求看作是请求A和请求B的父请求。反过来讲，可以把请求A和请求B视作首次请求的子请求。因此，就有了这样一条规则：一个请求的深度等于对它的父请求的深度递增一次后的结果。

在理解了我们刚刚对请求深度计算方法的描述之后，读者可能会发现：只有对某个请求的响应内容进行分析之后，才可能会需要生成新的请求。并且，调度器并不会直接把请求作为参数传递给分析器。这样不符合我们先前对数据流转方式的设计，同时也会使这两个处理模块之间的交互变得混乱。显然，响应也应该携带深度值。一方面，这可以算作是标示响应的深度的一种方式。另一方面，也是更重要的一方面，它可以作为新请求的深度值的计算依据。因此，base.Response类型的声明如下：

```
// 响应。
type Response struct {
    httpResp *http.Response
    depth    uint32
}

// 创建新的响应。
func NewResponse(httpResp *http.Response, depth uint32) *Response {
    return &Response{httpResp: httpResp, depth: depth}
}

// 获取HTTP响应。
func (resp *Response) HttpResp() *http.Response {
    return resp.httpResp
}

// 获取深度值。
```

```
func (resp *Response) Depth() uint32 {
    return resp.depth
}
```

这个类型的声明应该不用我再做解释了。它的各个部分的含义与base.Request类型的基本一致。

除了请求和响应这两个有着对应关系的数据结构之外，我们还需要定义条目的结构。条目的实例需要储存的内容应该会比请求和响应复杂得多。因为对响应的内容进行筛选并生成出条目的规则也是由网络爬虫框架的使用者自己制定的。因此，条目的结构应该足够地灵活。其实例应该可以容纳所有有可能从响应内容中筛选出的数据。基于此，我们编写出了条目的类型声明：

```
// 条目。
type Item map[string]interface{}
```

我们把Item类型声明为字典类型map[string]interface{}的别名类型。这样就可以最大限度地存储多样的数据了。由于条目处理器也应该是由网络爬虫框架的使用者提供的，所以这里并不用考虑字典中的各个元素值是否可以被条目处理器正确理解的问题。

好了，我们需要的3个基本数据类型都在这里了。为了能够用一个类型从整体上标识这3个基本数据类型，我们又声明了base.Data接口类型：

```
// 数据的接口。
type Data interface {
    Valid() bool // 数据是否有效。
}
```

这个接口类型只有一个名为Valid的方法。我们可以通过调用该方法来判断数据的有效性。显然，base.Data接口类型的作用更多的是作为某一类的类型的标签，而不是被用于定义这类类型的行为。为了让代表请求、响应或条目的类型都实现Data接口，我们又在当前的库源码文件中添加了这样几个方法：

```
// 数据是否有效。
func (req *Request) Valid() bool {
    return req.httpReq != nil && req.httpReq.URL != nil
}

// 数据是否有效。
func (resp *Response) Valid() bool {
    return resp.httpResp != nil && resp.httpResp.Body != nil
}

// 数据是否有效。
func (item Item) Valid() bool {
    return item != nil
}
```

这3个方法分别使*base.Request类型、*base.Response类型和base.Item类型实现了base.Data接口类型。这3个类型因base.Data接口类型而被归为了一类。在后面的章节中，我们会了解到这样做还有另外的功效。

至此，实现网络爬虫框架需要用到的基本数据类型均已编写完成。不过，在这里我们还需要一个额外的类型。这个类型是作为error接口类型的实现类型而存在的。它的主要作用是封装爬

取过程中出现的错误，并以统一的方式生成字符串形式的描述。我们已经知道，只要某个类型的方法集合中包含了下面这个方法就等于实现了error接口类型：

```
func Error() string
```

为此，我们首先声明了一个名为CrawlerError接口类型：

```
// 爬虫错误的接口。
type CrawlerError interface {
    Type() ErrorType // 获得错误类型。
    Error() string    // 获得错误提示信息。
}
```

其中，Type方法的结果类型ErrorType只是一个string类型的别名类型而已。另外，由于CrawlerError类型的声明中也包含了Error方法，所以只要某个类型实现了它，就等于实现了error接口类型。先编写这样一个接口类型而不是直接编写出error接口类型的实现类型的原因有两个。第一，我们在编程过程中应该遵循面向接口编程的原则。关于此，我们已经提到过多次。第二是为了扩展error接口类型。读者应该已经熟知，网络爬虫框架拥有多个处理模块。错误类型值应该可以表明该错误是哪一个处理模块抛出的。这也是CrawlerError类型中的Type方法所起到的作用。

下面，就让我们来实现这个接口类型。遵照本书中对实现类型的命名风格，我们声明了结构体类型myCrawlerError：

```
// 爬虫错误的实现。
type myCrawlerError struct {
    errType    ErrorType // 错误类型。
    errMsg     string   // 错误提示信息。
    fullErrMsg string   // 完整的错误提示信息。
}
```

字段errMsg的值应该由初始化myCrawlerError类型值的一方给出。这与传递给errors.New函数的参数值的含义类似。作为附加信息，errType字段的值就应该是该类型的Type方法的结果值。它代表了错误类型。为了便于使用者为该字段赋值，我们还声明了一些常量：

```
// 错误类型常量。
const (
    DOWNLOADER_ERROR    ErrorType = "Downloader Error"
    ANALYZER_ERROR      ErrorType = "Analyzer Error"
    ITEM_PROCESSOR_ERROR ErrorType = "Item Processor Error"
)
```

可以看到，这3个常量的类型都是base.ErrorType类型。它们分别与网络爬虫框架中的3个处理模块相对应。当某个处理模块在被执行的过程中出现了错误时，程序就会使用对应的base.ErrorType类型的常量来初始化一个base.CrawlerError类型的错误值。具体的初始化方法就是使用webcrawler/base代码包中的NewCrawlerError函数。其声明如下：

```
// 创建一个新的爬虫错误。
func NewCrawlerError(errType ErrorType, errMsg string) CrawlerError {
    return &myCrawlerError{errType: errType, errMsg: errMsg}
}
```

这个名称以“New”为前缀的函数的作用是创建和初始化一个CrawlerError类型的值。从该函数的函数体中的代码上我们可以看出，*myCrawlerError类型应该是CrawlerError类型的一个实现类型。*myCrawlerError类型的方法集合中应该包含CrawlerError接口类型中的Type方法和Error方法：

```
// 获得错误类型。
func (ce *myCrawlerError) Type() ErrorType {
    return ce.errType
}

// 获得错误提示信息。
func (ce *myCrawlerError) Error() string {
    if ce.fullErrMsg == "" {
        ce.genFullErrMsg()
    }
    return ce.fullErrMsg
}
```

眼尖的读者可能已经发现，指针方法Error中用到了myCrawlerError类型的fullErrMsg字段。并且，它还调用了一个名为genFullErrMsg的方法。现在我们就来看看它们的作用。下面是genFullErrMsg方法的实现：

```
// 生成错误提示信息，并给相应的字段赋值。
func (ce *myCrawlerError) genFullErrMsg() {
    var buffer bytes.Buffer
    buffer.WriteString("Crawler Error: ")
    if ce.errType != "" {
        buffer.WriteString(string(ce.errType))
        buffer.WriteString(": ")
    }
    buffer.WriteString(ce.errMsg)
    ce.fullErrMsg = fmt.Sprintf("%s\n", buffer.String())
    return
}
```

方法genFullErrMsg同样是myCrawlerError类型的指针方法。它的功能是生成Error方法需要返回的结果值。可以看到，我们没有直接使用myCrawlerError类型值的使用方提供的值（即那个会被赋给errMsg字段的值），而是以它为基础生成了一条更完整的错误提示信息。在这条信息中，明确显示出它是一个网络爬虫的错误，也给出了错误的类型和详情。注意，我们把这条错误提示信息缓存在了fullErrMsg字段中。回顾该类型的Error方法的实现，只有当fullErrMsg字段的值为""时才会调用genFullErrMsg方法，否则会直接把fullErrMsg字段的值作为Error方法的结果值返回。这也是为了避免频繁地拼接字符串给程序性能带来的负面影响。我们在genFullErrMsg方法的实现中使用了bytes.Buffer类型值来作为拼接错误信息的手段。虽然这样做确实可以大大减小这一负面影响，但是由于myCrawlerError类型的值是不可变的，所以缓存错误提示信息还是很有必要的。其根本原因是，对这样的不可变值的缓存永远不会失效。

我们在前面展示的这些类型对于承载数据（不论是正常数据还是错误信息）来说已经足够用了。它们是网络爬虫框架中的基本元素。

9.4.2 接口的设计

这里所说的接口是指网络爬虫框架中的各个模块以及中间件中的重要组件的接口。与我们先前描述的基本数据结构不同，它们的主要职责是定义模块和组件的行为。在定义行为的过程中，我们会对它们应有的功能作进一步的审视，同时也会更多地思考它们之间的协作方式。

下面，我们就开始逐一地设计网络爬虫框架中的这类接口。为了更易于理解，我们会先从那几个处理模块的接口开始，然后再去考虑怎样去定义将会控制和管理这些处理模块的调度器以及它会用到的各种组件的行为。

注意 本小节描述的部分接口会在9.8节中被修改。因此，它们会与随书项目中的相应代码不一致。

1. 网页下载器

网页下载器的功用就是从网络中的目标服务器上下载网页。一个网页在网络中的唯一标识是网络地址。但是，网络地址只能起到定位网页在网络中的位置的作用，而并不是成功下载网页的充分条件。

我们已经知道，HTTP协议是基于TCP/IP协议栈的应用层协议。它是互联网世界的根基之一。因此，在互联网时代诞生的绝大多数语言都会使用不同的方式提供针对该协议的API。当然，Go语言也不例外。Go语言的标准库代码包net/http就提供了这些API。实际上，我们在编写网络爬虫框架的基本数据结构的时候，就使用到了其中的两个类型http.Request和http.Response。并且，不夸张地说，我们将要构建的网络爬虫框架就是以HTTP协议和net/http代码包中的API为基础的。

从网页下载器充当的角色来讲，它的功能只有两个：发送请求并接收响应。因此，我们可以设计出这样一个声明：

```
func Download(req base.Request) (*base.Response, error)
```

函数Download的签名完全体现出了网络下载器应有的功能。不过网络下载器的接口不应只包含这一个声明。因为，我们已经明确表示过，网络爬虫框架应该同时使用若干个网页下载器来提供网页下载能力。进一步讲，这些网页下载器应该被放置于一个网页下载器池中。因此，网页下载器的实例还应该使用某种方式唯一地标识自己。这样的好处很多，至少有助于我们鉴别池中的不同网页下载器实例。

综上所述，我们已经可以给出网页下载器的接口类型声明了：

```
// 网页下载器的接口类型。
type PageDownloader interface {
    Id() uint32 // 获得ID。
    Download(req base.Request) (*base.Response, error) // 根据请求下载网页并返回响应。
}
```

其中，方法Id会把当前的网页下载器实例的唯一标识“ID”作为结果值返回给方法调用方。在一个基于网络爬虫框架的应用程序的实例范围之内，一个ID应该足以唯一地标示一个网页下载器实例。说到这里，我们不得不提到作为中间件中的一个组件的工具——ID生成器。在这里，我

们并不会讨论ID生成器的具体实现，而只会给出它的接口类型：

```
// ID生成器的接口类型。
type IdGenertor interface {
    GetUint32() uint32 // 获得一个uint32类型的ID。
}
```

一个可以有将近43亿（2的32次方，即4 294 967 296）的取值的类型已经完全够用了。这肯定能够满足我们构建一个足够大的网页下载器池的需求。至少对现阶段来说是这样。因此，我们只为接口类型IdGenertor添加一个方法——GetUint32。IdGenertor接口类型的实现类型应该保证：在GetUint32方法的调用次数超过2的32次方之前，每次调用所得到的数值都是不同的。并且，为了使之更加通用，还应该考虑到：在该调用次数超出uint32类型的取值范围之后，GetUint32方法返回的数值又该是怎样的。读者也可以思考一下这个问题，甚至试着编写出一个实现此接口的ID生成器。有了之前我们讲到的那些知识，这应该并不困难。要记得，ID生成器中的所有方法都应该是并发安全的。

在前面，我们多次提到了网页下载器池。那么这个池的功能定义又是怎样的呢？首先，我们应该可以在需要时从该池中取出一个网页下载器。其次，在我们完成对所持网页下载器的使用之后还可以将其归还给该池。最后，我们应该随时能了解到该池的使用状况。据此，我们有了这样一个接口类型声明：

```
// 网页下载器池的接口类型。
type PageDownloaderPool interface {
    Take() (PageDownloader, error) // 从池中取出一个网页下载器。
    Return(dl PageDownloader) error // 把一个网页下载器归还给池。
    Total() uint32 // 获得池的总容量。
    Used() uint32 // 获得正在被使用的网页下载器的数量。
}
```

该接口声明中的注释很清楚地描述了其中每个方法的功能。其中，值得注意的是Total方法。我们在前面并没有明确网页下载器的总容量是否可变。Total方法的结果值会体现出该池总容量的可变性。如果池的总容量可变，那么在不同时刻调用该方法所得到的数值就可能会有所不同。

显然，实现总容量固定的池会比较容易。我们在后面编写池的实现的时候会专门讨论这个问题，并在功能的丰满与程序的复杂度之间进行权衡。

与网页下载器有关的所有接口的声明都被放到了goc2p项目的webcrawler/downloader代码包中。

2. 分析器

分析器的职责是根据给定的规则分析响应，并筛选出请求或条目。同时，它也应该能在单个的基于网络爬虫框架的应用程序中唯一地标识自己。下面是代表了分析器行为的接口类型的声明：

```
// 分析器的接口类型。
type Analyzer interface {
    Id() uint32 // 获得ID。
    Analyze(
        respParsers []ParseResponse,
        resp base.Response) ([]base.Data, []error) // 根据规则分析响应并返回请求和条目。
}
```

接口类型Analyzer与网页下载器接口的声明风格非常类似。不过，作为提高可扩展性的努力的一部分，其中的Analyze方法还要接受一个很重要的参数。它就是元素类型为ParseResponse的切片respParsers。

类型ParseResponse是一个函数类型。它的声明如下：

```
// 被用于解析HTTP响应的函数类型。
type ParseResponse func(httpResp *http.Response, respDepth uint32) ([]base.Data,
[]error)
```

声明这样一个函数类型的意义是让网络爬虫框架的使用者自定义响应分析规则，以及生成相应的请求和条目的方式。该函数类型的参数httpResp代表了目标服务器返回的HTTP响应，而参数respDepth则代表了该响应的深度。这些参数可以让函数实例（也就是符合此类型声明的函数）获取到执行分析和生成过程所需的各种信息。另一方面，它的结果声明列表表明，该函数类型的实例需要把经分析和筛选而生成的若干请求和条目作为结果返回。同时，如果在这个过程中发生了任何错误也要如实地上报。

可以看到，我们把整个的分析、筛选和生成的过程都让使用者自行定义，而不是只让其提供规则。这样做鉴于以下几个原因。

- 对内容的分析和筛选往往与数据的生成方式之间有很强的关联性。若把这几个过程分离开来并由两方分别定义，必然会造成很多不必要的中间数据创建和传递。这样不但会使我们编写出不少冗余的代码、增加相应模块的复杂度，还会使网络爬虫框架的一些API（比如ParseResponse类型）变得不清晰和臃肿。
- 对响应分析和其内容筛选的规则制定是可以有据可依的。因为有一些标准是专门针对于此的。读者可参看W3C网站上关于XPath或XQuery的描述。我们可以直接遵循某一个标准而形成分析和筛选规则的制定方法。所以，把这项任务抛出去并不会给网路爬虫框架的使用者带来太多的不便。另外，在Go语言的标准库中并没有提供实现此类标准（如XPath或XQuery）或便于进行此项任务的代码包。虽然使用第三方的代码包当然是可行的，但是我并不想把此类第三方库作为网络爬虫框架的实现方案的一部分。这样无益于突出重点。况且，有些需要去访问的新的网络地址并不是直接包含在响应的内容中的，而是由相关的JavaScript程序动态生成的。这样的情况过于复杂和多样了。在现阶段，我们把对这些情况的处理留给作为需求方的网络爬虫框架的使用者应该会更好。
- 如果读者对HTTP协议有一定的了解，那么就一定会知道：对于HTTP的请求来说，并不是仅仅需要填充请求主体那么简单。对于请求的头部信息，我们可能需要进行更加灵活多变的设置，包括但不限于对连接方式、缓存机制、字符编码集、Cookie以及授权证书的设定。这些设定往往会根据目标网站、访问身份以及响应的具体内容等因素的不同而不同。因此，如果网络爬虫框架把新请求的生成操作放在其内部，那么这种灵活的设置将很难进行，甚至会导致无法满足使用者的正常需求。当然，我们可以设计更加复杂的ParseResponse函数类型，甚至使用更加重量级的附带若干方法的类型来表现它。但是，从设计和编写该框架的目的来看，我们并不需要把它设计得如此复杂。

总之，在实现这类框架的初期，我们需要把它设计得简单明了一些，并且以可用性作为首要目标。在这之后，我们可以在改进和演化框架的过程中适当地增加它的功能，并以此让它适用于更多的情况。增加一些开箱即用的额外工具会是一个不错的选择。它们可以让使用者在编写针对框架的定制代码的时候减少一些工作量。

好了，让我们回到分析器的接口设计中来。通过Analyzer接口类型的Analyze方法，网络爬虫框架会拿到若干数据（请求或条目）实例和错误实例。这些实例会通过中间件中的工具被递送到不同的模块中。

对于允许有多个实例同时存在的分析器来说，我们也需要用一个池来管理它们。分析器池的功能需求与前面讲述的网页下载器池的功能需求完全一致。前者的声明如下：

```
// 分析器池的接口类型。
type AnalyzerPool interface {
    Take() (Analyzer, error) // 从池中取出一个分析器。
    Return(analyzer Analyzer) error // 把一个分析器归还给池。
    Total() uint32 // 获得池的总容量。
    Used() uint32 // 获得正在被使用的分析器的数量。
}
```

有些读者可能会思考：我们是否可以统一一下这两个池的接口呢？对于Go语言这样不支持自定义泛型的编程语言来说，我们怎样做才能够达到统一的目的？读者可以先考虑一下这两个问题，我们会在后面再次提到它。

与分析器有关的所有接口的声明都放到了goc2p项目的webcrawler/analyzer代码包中。

3. 条目处理管道

顾名思义，条目处理管道的功能就是为条目的处理提供环境，并控制整体的处理流程。具体的处理的步骤由网络爬虫框架的使用者提供。实现单一处理步骤的程序称为条目处理器。这样的设计可以让网络爬虫框架与具体的条目处理步骤分离开来，同时又不至于丧失控制权。下面我们来看看条目处理管道的接口类型：

```
// 条目处理管道的接口类型。
type ItemPipeline interface {
    // 发送条目。
    Send(item base.Item) []error
    // FailFast方法会返回一个布尔值。该值表示当前的条目处理管道是否是快速失败的。
    // 这里的快速失败是指：只要对某个条目的处理流程在某一个步骤上出错，
    // 那么条目处理管道就会忽略掉后续的所有处理步骤并报告错误。
    FailFast() bool
    // 设置是否快速失败。
    SetFailFast(failFast bool)
    // 获得已发送、已接受和已处理的条目的计数值。
    // 更确切地说，作为结果值的切片总会有3个元素值。这3个值会分别代表前述的3个计数。
    Count() []uint64
    // 获取正在被处理的条目的数量。
    ProcessingNumber() uint64
    // 获取摘要信息。
    Summary() string
}
```

该接口类型中最重要的方法就是Send方法。该方法使条目处理管道的使用方可以向它发送条目，以使其中的条目处理器对这些条目进行处理。FailFast方法和SetFailFast对应于条目处理管道的“快速失败”特性。方法的注释对这一特性已有清晰的描述。最后，添加方法Count、ProcessingNumber和Summary更多的是出于程序监控的考虑。我们在后面会看到，Summary方法还会出现在更多接口类型的声明中。

细心的读者可能已经发现，接口类型ItemPipeline中的方法并没有体现出如何设置条目处理器序列。实际上，设计该接口类型的一个很重要的思路是：不能对条目处理管道中的条目处理器序列进行变更。换句话说，我们要求针对条目的处理流程在总体上是不可变的。这种不可变性会使该流程以及对它的控制保持简单。

既然不能改变条目处理器序列，那么我们只能在初始化条目处理管道的时候对它进行设置。由于这个序列的长度是不定的，所以我们需要用一个切片类型的值来代表它。该切片值的类型是[]ProcessItem。类型ProcessItem即是被用来代表条目处理器的类型。其声明如下：

```
// 被用来处理条目的函数类型。
type ProcessItem func(item base.Item) (result base.Item, err error)
```

函数类型ProcessItem接受一个需要被处理的条目，并把被处理后的条目和可能发生的错误作为结果值返回。如果第二个结果值不为nil，则说明在这个处理的过程中发生了一个错误。

我们使用一个函数类型来代表条目处理器的原因是：单一的函数往往可以被安全地并发执行，除非它使用到了某类共享资源。相比之下，实现一个接口类型的方式太多。作为框架，我们几乎无法给出并发安全方面的强制性约束。然而，对于条目处理器来说，保证并发安全性又是非常重要的。所有被送入条目处理管道的条目都会被其中的条目处理器并发地处理。这就让我们不得不尽量对条目处理器的实现进行约束。

由于条目处理管道在用途和设计上不同于网页下载器和分析器，并且有了那个可以使条目处理器的并发安全性基本上得到保障的ProcessItem函数类型，所以我们并不需要所谓的条目处理管道池。网络爬虫框架仅持有条目处理管道的一个实例就足够了。

与条目处理管道有关的所有接口的声明都放到了goc2p项目的webcrawler/itempipeline代码包中。

4. 调度器

调度器属于控制模块而非处理模块。它需要对各个处理模块的运作进行检测和控制。可以说，调度器是网络爬虫框架的核心。正因为如此，我们需要让其提供相应的启动和停止爬取流程的方法。除此之外，出于监控整个流程的目的，我们还应该在这里为使用方提供一些获取实时状态或统计信息的方法。依照这样的思路，我们有了这样一个接口类型声明：

```
// 调度器的接口类型。
type Scheduler interface {
    // 启动调度器。
    // 调用该方法会使调度器创建和初始化各个组件。在此之后，调度器会激活爬取流程的执行。
    // 参数channellen被用来指定数据传输通道的长度。
    // 参数poolSize被用来设定网页下载器池和分析器池的容量。
    // 参数crawlDepth代表了需要被爬取的网页的最大深度值。深度大于此值的网页会被忽略。
```

```

// 参数httpClientGenerator代表的是被用来生成HTTP客户端的函数。
// 参数respParsers的值应为分析器所需的被用来解析HTTP响应的函数的序列。
// 参数itemProcessors的值应为需要被置入条目处理管道中的条目处理器的序列。
// 参数firstHttpRequest即代表首次请求。调度器会以此为起始点开始执行爬取流程。
Start(channellen uint,
      poolSize uint32,
      crawlDepth uint32,
      httpClientGenerator GenHttpClient,
      respParsers []anlz.ParseResponse,
      itemProcessors []ipl.ProcessItem,
      firstHttpRequest *http.Request) (err error)
// 调用该方法会停止调度器的运行。所有处理模块执行的流程都会被中止。
Stop() bool
// 判断调度器是否正在运行。
Running() bool
// 获得错误通道。调度器以及各个处理模块运行过程中出现的所有错误都会被发送到该通道。
// 若该方法的结果值为nil, 则说明错误通道不可用或调度器已被停止。
ErrorChan() <-chan error
// 判断所有处理模块是否都处于空闲状态。
Idle() bool
// 获取摘要信息。
Summary(prefix string) SchedSummary
}

```

接口类型Scheduler的Start方法的作用就是启动调度器。它接受的参数并不少, 共有7个。前6个参数都是被用于初始化网络爬虫框架中的各个组件的。而最后一个参数firstHttpRequest则是在前面提到的首次请求。爬取流程的执行是由它来激活的。它会让网络爬虫框架对首个以及后续网页的下载、分析和处理流程真正地运转起来。

请读者注意Start方法的参数声明列表中的那个名为httpClientGenerator参数。该参数的类型是GenHttpClient。该类型是一个函数类型, 其声明如下:

```

// 被用来生成HTTP客户端的函数类型。
type GenHttpClient func() *http.Client

```

可以看到, 该类函数会生成HTTP客户端的实例。HTTP客户端实例是HTTP请求的发送操作和HTTP响应的接收操作的执行者, 是网页下载器必备的组件。有了这个函数类型, 我们就可以让网络爬虫框架的使用者自己决定怎样创建和初始化HTTP客户端实例。由于http.Client类型的所有字段都是公开的, 所以我们可以有这样的函数中生成定制化程度很高的HTTP客户端实例。当然, 为了保证网页下载子流程的正确执行, 网络爬虫框架也应当提供一个默认的HTTP客户端实例生成方法 (即GenHttpClient函数类型的默认实现)。

接口类型Scheduler的Stop方式的作用是停止调度器以及各个处理模块的运行。停止调度器是比较容易的, 但是中止正在运行的各个处理模块就相对复杂了。这一功能的实现既需要做到统一, 又应该充分分散。前者是说需要有统一的停止信号来标识整个流程的停止。后者指的是所有需要被停止的子流程都需要关注该信号并及时做出响应。这样才能保证爬取流程中的各个环节的状态的一致性。因此, 停止操作的实现几乎涉及了网络爬虫框架的所有组件。我们在实现它的时候会进行详细的讨论。读者也可以先思考一下这一问题的解决方案。

有的读者可能混淆Scheduler接口类型的Running方法和Idle方法的功能。实际上，它们是在不同层次上了解爬取流程执行情况的方法。Running方法仅仅会返回一个代表了当前调度器是否正在运行（已被启动且还未被停止）的bool类型值。而Idle方法则被设计为可以深度检测网络爬虫框架的运行状况的方法。也就是说，它应该对爬取流程的执行情况进行更深层次的了解，比如，检测各个池的使用情况和条目处理管道的内部运作情况。Idle方法应该依据这些信息进行最终的判定，并把判定结果返回给该方法的调用方。请读者回顾一下，在前面讲到的那些处理模块的接口中，哪些方法能够为Idle方法的判定提供依据？

再来说Scheduler接口类型的ErrorChan方法。我们可以调用该方法以获得一个元素类型为error的接收通道。关于这个接收通道的创建过程和其中的元素值的来源，我们会稍后在讲解中间件的相关接口的时候予以介绍。在这里，我们只需要知道从这个接收通道中可以接收到在执行爬取流程的过程中发生的错误的值即可。

最后，Scheduler接口类型的Summary方法有着与ItemPipeline接口类型的Summary基本一致的功能。但不同的是，后者返回的是一个string类型值，而前者返回的则是一个SchedSummary类型的值。SchedSummary类型是专门被用来提供调度器的摘要信息的类型。该类型的声明如下：

```
// 调度器摘要信息的接口类型。
type SchedSummary interface {
    String() string           // 获得摘要信息的一般表示。
    Detail() string          // 获取摘要信息的详细表示。
    Same(other SchedSummary) bool // 判断是否与另一份摘要信息相同。
}
```

其中，String方法是标准的被用来返回类型实例的字符串表示形式的方法。而Detail方法则可以被看作是String方法的增强版。它返回的字符串中会包含更加详细的信息。最后，Same方法被用来判断当前的类型实例与另一个是否相同。在后面我们会看到，这个方法可以为判断打印调度器摘要信息的必要性提供依据。

以上就是所有的与调度器有关的公开的接口。虽然调度器的行为定义看起来如此简单，但实现这些行为却是需要进行很多考量的。相比接口，调度器的实现更加体现了我在设计网络爬虫框架的调度和监控等功能方面的思考。在调度器的实现中，我们使用了很多被称为网络爬虫框架的中间件的工具。从这些工具的设计上，读者也可以更加清晰地了解到网络爬虫框架的运作模式。我们马上就会讲到它们。

与调度器有关的所有接口的声明与它们的实现一起都放在了goc2p项目的webcrawler/scheduler代码包中。不过，其中使用到的中间件工具处于另一个代码包中。

5. 中间件接口概述

从上一节展示的图9-2中，我们可以看到，网络爬虫框架的中间件起到了承上启下的作用。其中的工具都是调度器对各个处理模块进行调度和监控的有力辅助。具体来讲，这些工具包括：ID生成器、通道管理器、实体池和停止信号。

我们在前面讲述网页下载器的接口类型的时候，已经对ID生成器的接口类型IdGenertor进行过介绍。下面我们重点描述与其他3个工具相关的接口类型。

6. 通道管理器

调度器的职责之一就是要在各个处理模块之间传递数据。关于各类型数据的流转方向，读者可以回顾9.2节中展示的图9-1。通过对Go语言的并发编程方式的了解，我们实现这样的功能的时候，肯定会首先想到通道（Channel）类型。实际上，使用通道在各个处理模块之间传递数据是一个最佳方案。其原因不言而喻。由于需要传递的数据有多个种类，我们需要的通道也会有不同的类型。更具体地说，我们会用到的通道的类型共用4个，它们分别对应了请求、响应、条目和错误这4类数据。

通道管理器正是被用来管理上述4类通道的工具。通道管理器只会关心并管理通道，而不会关心网络爬虫框架中的任何流程的执行过程。这从它的接口类型声明上也可以看得出来：

```
// 通道管理器的接口类型。
type ChannelManager interface {
    // 初始化通道管理器。
    // 参数channellen代表通道管理器中的各类通道的初始长度。
    // 参数reset指明是否重新初始化通道管理器。
    Init(channellen uint, reset bool) bool
    // 关闭通道管理器。
    Close() bool
    // 获取请求传输通道。
    ReqChan() (chan base.Request, error)
    // 获取响应传输通道。
    RespChan() (chan base.Response, error)
    // 获取条目传输通道。
    ItemChan() (chan base.Item, error)
    // 获取错误传输通道。
    ErrorChan() (chan error, error)
    // 获取通道长度值。
    Channellen() uint
    // 获取通道管理器的状态。
    Status() ChannelManagerStatus
    // 获取摘要信息。
    Summary() string
}
```

我们可以看到，接口类型ChannelManager中的大部分方法的功能都是获取某类实例或信息。其中有4个方法是分别被用来获取通道管理器中的某一个类型的通道的。这4个方法的结果声明列表中除了相应的通道类型之外，还都包含了一个error类型。这第二个结果的值在哪些情况下会是非nil的呢？读者可以在它们的实现中找到答案。

接口类型ChannelManager的方法Init的功能是对当前的通道管理器进行初始化。在对其初始化的过程中，我们势必要同时对其容纳的各类通道进行初始化。这就需要我们给出这些通道的初始长度。该初始长度由参数channellen代表。另外，参数reset使得通道管理器可以被强行地初始化，即使它已经被初始化过。这样，我们就从接口层面上允许了对通道管理器中的各类通道及其长度的重新设定。

通道管理器的Close方法的作用是关闭通道管理器以及其中的所有通道实例。如果发现该通道管理器已被关闭，那么该方法应该立即返回并将false作为结果值。

我们应该专门设计出一套针对通道管理器的状态标识，以便在其内部和外部都能够实时、清

晰地了解它的状况。为此，就有了这样一个类型以表示通道管理器状态的值：

```
// 被用来表示通道管理器的状态的类型。
type ChannelManagerStatus uint8
```

对于ChannelManagerStatus类型的值，我们预设了3个：

```
const (
    CHANNEL_MANAGER_STATUS_UNINITIALIZED ChannelManagerStatus = 0 // 未初始化状态。
    CHANNEL_MANAGER_STATUS_INITIALIZED   ChannelManagerStatus = 1 // 已初始化状态。
    CHANNEL_MANAGER_STATUS_CLOSED        ChannelManagerStatus = 2 // 已关闭状态。
)
```

这3个常量分别代表了通道管理器的不同状态。CHANNEL_MANAGER_STATUS_UNINITIALIZED表示通道管理器还未被初始化。只要我们没有调用过通道管理器的Init方法，它就会处于这种状态。与之对应，我们对Init方法的初次调用会使该通道管理器的状态转变为CHANNEL_MANAGER_STATUS_INITIALIZED。在这之后，我们再次调用它的Init方法就并不一定会使之状态发生改变了。这由通道管理器当时的状态以及我们传递给该方法的第二个参数值决定。最后，如果通道管理器已被初始化，那么调用它的Close方法必定会使其状态转变为CHANNEL_MANAGER_STATUS_CLOSED。

除了通道管理器的接口类型所表示出的功能需求之外，我们要求它的方法Init、Close以及其他不会改变通道管理器内部状态的方法都应该是并发安全的。这就需要我们在实现它们的时候考虑使用适合的同步工具加以保证。关于这一方面的内容，我们以后再说。

顺便提一下，与网络爬虫框架的中间件有关的所有声明代码分别存放到了goc2p项目的webcrawler/middleware代码包的各个源码文件中。这包括我们稍后会讲到的实体池和停止信号。

7. 实体池

我们已经知道，针对网页下载器池和分析器池的接口之间非常地相似。这也意味着这两者在行为和功能上基本一致。显然，出于避免重复代码和抽象共有特性的目的，我们应该编写一个可以被公用的池。由于池中一般会存放若干个同一个类型的实体（在Go语言中，称为值），因此我们给予它一个专有名称——实体池。下面是为它声明的接口类型：

```
// 实体池的接口类型。
type Pool interface {
    Take() (Entity, error)      // 取出实体。
    Return(entity Entity) error // 归还实体。
    Total() uint32              // 实体池的容量。
    Used() uint32               // 实体池中已被使用的实体的数量。
}
```

该类型中的4个方法的声明与之前讲到的那两个池的基本一致。不过，其中的Entity类型是我们专门为它声明的。

作为一个可以被公用的池，我们应该尽量允许任意类型的元素（当然，一个实体池中的所有元素的类型应该是一致的）。又由于Go语言并不支持自定义泛型，所以我们首先想到的是使用interface{}类型作为相关方法的参数或结果的类型：

```
Take() (interface{}, error)
Return(interface{} Entity) error
```

我们都知道，空接口类型`interface{}`的变量可以被赋予任何类型的值。不过，我们还是希望池中的每个元素值都能够提供唯一标识自己的方法。不知读者是否还记得，我们刚刚讲过接口类型`PageDownloader`和`Analyzer`都包含了这样一个方法：

```
Id() uint32 // 获得ID。
```

这两个类型的值都会以此种方式唯一地标识自己。既然我们主要会用实体池来存放网页下载器或分析器，那么我们就依照它们的ID获取方法来声明`Entity`接口类型：

```
// 实体的接口类型。
type Entity interface {
    Id() uint32 // ID的获取方法。
}
```

这样一来，我们在不变动之前编写好的`PageDownloader`类型和`Analyzer`类型的声明的情况下就可以让它们成为`Entity`的扩展接口类型了。

这里预先透露一点实现细节。对于实体池的实现，我们可以参考之前讲过的Goroutine票池。读者可以先思考一下怎样去做。另外，与通道管理器一样，实体池中的各个方法也应该具有并发安全性。

8. 停止信号

我们在第7章讲述过一个载荷发生器的实现过程。在代表载荷发生器实现的那个结构体类型中有这样一个字段声明：

```
stopSign chan byte // 停止信号的传递通道。
```

字段`stopSign`是一个简单明了的停止信号传输通道。正因为有了这样一个通道，针对载荷发生器的停止流程才得以及时、正确地执行。

不过，我们在网络爬虫框架中使用如此简单的结构来传递停止信号是不合适的。主要原因是，停止信号的发送者应该并不需要关心会有多少个接收者。如果使用上述通道来传递停止信号，那么发送者是无法确定应该向该通道发送几个停止信号的。即使我们在实现了所有处理模块之后可以将这个数量固定下来，也会为今后可能的代码修改埋下隐患。换句话说，一旦我们修改了某个或某些处理模块中被用来处理停止信号的代码，就可能需要重新计算发送方应该发送的停止信号的数量。只要这个数量出现偏差，就非常有可能会使网络爬虫框架在执行停止流程的时候出现问题。因此，我们需要建立一个“一方发送、多方接收”的灵活的停止信号传递机制。这是出于今后对网络爬虫框架扩展的考虑。同时，这样做也可以让我们在编写各种处理停止信号的代码的时候不受任何约束。

为了建立一个灵活的停止信号传递机制，我们编写了下面这个接口类型声明：

```
// 停止信号的接口类型。
type StopSign interface {
    // 置位停止信号。相当于发出停止信号。
    // 如果先前已发出过停止信号，那么该方法会返回false。
    Sign() bool
    // 判断停止信号是否已被发出。
    Signed() bool
}
```

```

// 重置停止信号。相当于收回停止信号，并清除所有的停止信号处理记录。
Reset()
// 处理停止信号。
// 参数code应该代表停止信号处理方的代号。该代号会出现在停止信号的处理记录中。
Deal(code string)
// 获取某一个停止信号处理方的处理计数。该处理计数会从相应的停止信号处理记录中获得。
DealCount(code string) uint32
// 获取停止信号被处理的总计数。
DealTotal() uint32
// 获取摘要信息。其中应该包含所有的停止信号处理记录。
Summary() string
}

```

接口类型StopSign共包含了7个方法。先说前3个方法。停止信号的发出方调用Sign方法以发出信号。当然，这个发出的动作不一定会成功。因为，如果Signed方法的返回值为true，那么就说明停止信号已被发出。在这之后的重复的停止信号发出动作都不会成功。如果发出方想强行地再次发出停止信号，那么只能先调用Reset方法以重置停止信号，而后再调用Sign方法。注意，Reset方法不但会重置停止信号本身，还会清除此前所有的停止信号处理记录。它的功能相当于对其所属的StopSign类型值重新进行初始化。

方法Sign和Reset是专供给停止信号的发出方使用的。而Signed方法则可以被任何一方使用。停止信号的处理方可以适时地通过调用Signed方法以判断信号是否已被发出。如果停止信号已发出，那么处理方就需要及时停止当前流程并进行适当的善后处理。在处理完成后，停止信号的处理方应该调用该停止信号的Deal方法，以表示已经对该信号处理完毕。这样做完全是处于统计和监控的目的。注意，Deal方法应该只在停止信号已被发出且未被重置的情况下才会进行相应的操作。而在其他情况下，此操作应该被忽略。

从调度器的职责以及它与各个处理模块之间的关系来看，内部的停止信号理应由调度器发出。接口类型StopSign的声明中的最后3个方法会对调度器执行监控处理模块的任务有很大帮助。方法DealCount和DealTotal分别被用来获取单一处理方或所有处理方对当前停止信号的处理计数。而Summary方法则可以生成并返回一个综合性的摘要信息，其中包含了全部的处理计数信息。

我们会在讲述StopSign接口类型的实现类型的时候看到，网络爬虫框架中的停止信号处理机制与先前的载荷发生器在这方面的实现方式截然不同。它们使用的被用来保证并发安全的方法也互不相干。这充分说明了，没有任何一个程序设计方案可以被称为银弹。即使对于像通道这样的并发编程利器来说也是这样。

在本小节，我们讲述了网络爬虫框架中的所有公开的接口类型。这些接口类型共同描绘出了网络爬虫框架的总体架构和行为模式。不过，我们在实现这些接口类型的时候仍然会有很多工作要做。在这之中也会涉及对各种设计方案的抉择和取舍。

9.5 中间件的实现

在编写了网络爬虫框架的基本数据结构，以及设计了构建它所需的所有组件的接口之后，我们就可以开始动手实现它了。与接口设计的顺序正好相反，我们会从最基础的组件（即各种中间

件工具)开始写起。下面我们马上开始。

注意 本节描述的部分实现会在9.8节中被修改。因此,它们会与随书项目中的相应代码不一致。

9.5.1 通道管理器

通道管理器会对网络爬虫框架中所涉及的所有通道进行统一的管理。这里所说的管理包括初始化、热替换和关闭等。我们在设计其接口类型的时候,已经知悉了初始化和关闭这两个功能,但是却没有提及过热替换功能。实际上,热替换功能正是设计这个通道管理器的重要目的之一。

1. 热替换

我们在这里所说的热替换的作用是,在替换被用于传输某类数据的通道的时候,使用该通道的各方不会受到丝毫影响。通过这样的隔离方式,调度器就不用担心因通道替换而带来的一些问题了。

为了让读者能够更加清楚地看到这些问题。我们先来进行一个试验。请想象这样一个场景:我们使用两个Goroutine分别对同一个通道进行发送操作和接收操作。首先,我们需要先声明几个变量:

```
var chan1 chan int
var chanLength int = 18
var interval time.Duration = 1500 * time.Millisecond
```

变量chan1代表的是作为操作目标的通道。由于需要多次初始化该通道,所以我们使用chanLength变量来明确它的长度。在这里,我们把这个长度值设定为18。而第三个变量interval表示的则是针对通道chan1的发送操作和接收操作的进行间隔。设置这样一个进行间隔的目的是让我们运行这个试验程序的时候更容易看清其过程。

好了,我们现在把上面的那3行代码放入到一个命令源码文件中并开始编写它的main函数。在main函数中,我们需要做3件事。第一件事是初始化通道chan1:

```
chan1 = make(chan int, chanLength)
```

这也是对该通道的第一次初始化。第二件事,启用一个Goroutine专门向该通道发送元素。请看下面的代码:

```
go func() {
    for i := 0; i < chanLength; i++ {
        if i > 0 && i%3 == 0 {
            fmt.Println("Reset chan1...")
            chan1 = make(chan int, chanLength)
        }
        fmt.Printf("Send element %d...\n", i)
        chan1 <- i
        time.Sleep(interval)
    }
    fmt.Println("Close chan1...")
    close(chan1)
}()
```

在这个专用的Goroutine中，我们会每隔1.5秒向chan1通道发送一个元素值。发送的总数量与该通道的长度值相同。在每次发送之前，我们还会判断当次发送的是第几个元素值。如果符合条件，我们就会先对chan1进行重新初始化。读者不用太在意这个重新初始化的条件。这是为了让重新初始化操作的进行更加有规律而做的设定，并没有其他含义。最后，在所有的发送操作都完成之后，我们还会关闭通道chan1。注意，为了能够清楚地看到程序的运行的过程，我们在程序中适当地添加了一些打印语句。这对我们说清问题会很有帮助。

很显然，我们最初创建的通道值并不会被填满。因为，在第四次迭代之初（由重新初始化条件可知），变量chan1会与当前的值解除绑定，并且与新创建的通道值建立绑定。在这之后，发送语句chan1 <- i实际上是把i的值发送给了新创建的通道值，而我们最初初始化的那个通道值中的元素值的数量就永远不会再增加了。相似地，我们之后赋给chan1变量的每个通道值都最多只会包含3个元素值。这就是我们将要关注的通道值的替换动作。

为了展示上述替换动作所带来的影响，我们还需要定时地从chan1处接收元素值。负责此任务的代码被包含在了函数receive中。我们在main函数中要做的第三件事就是调用这个函数。它只接受一个参数。该参数的值即是chan1的值：

```
receive(chan1)
```

函数receive的完整声明如下：

```
func receive(chan2 chan int) {
    fmt.Println("Receive element from chan1...")
    timer := time.After(30 * time.Second)
Loop:
    for {
        select {
            case e, ok := <-chan2:
                if !ok {
                    fmt.Println("--Closed chan1.")
                    break Loop
                }
                fmt.Printf("Receive a element: %d\n", e)
                time.Sleep(interval)
            case <-timer:
                fmt.Println("Timeout!")
                break Loop
        }
    }
    fmt.Println("--End.")
}
```

在这里，我们同样用到了for语句。不过，为了防止receive函数的执行永远无法结束，我们还用到了标记语句和select语句。在一开始，我们先利用time.After函数创建了一个定时器，超时时间为30秒。然后，我们试图定时且不断地从当前函数的参数chan2中接收元素值。与main函数中的发送操作相同，该接收操作的间隔时间也为1.5秒。注意，我们把该接收操作作为了select语句中的一个case表达式。而另一个case表达式则是针对定时器timer的接收操作。此select语句与外层的for语句和以Loop为标识的标记语句一起构成了带操作超时设置的被用于从通道chan2

接收元素值的程序片段。

如此组合的含义是每隔1.5秒尝试从chan2处接收元素值并把它们打印到标准输出，但只要这些操作耗费的总时间超过了30秒就立刻放弃当前以及后续的所有尝试并结束for语句的执行。此外，如果chan2代表的通道被关闭，程序也会跳出for语句。这样一来，我们就有效地避免了因接收操作的阻塞而导致的程序执行停滞的问题。

在我们使用go run命令运行了当前的命令源码文件（无论包含前述代码的文件叫什么名字）之后，标准输出上会出现如下内容：

```
Receive element from chan1...
Send element 0...
Receive a element: 0
Send element 1...
Receive a element: 1
Send element 2...
Receive a element: 2
Reset chan1...
Send element 3...
Send element 4...
Send element 5...
Reset chan1...
Send element 6...
Send element 7...
Send element 8...
Reset chan1...
Send element 9...
Send element 10...
Send element 11...
Reset chan1...
Send element 12...
Send element 13...
Send element 14...
Reset chan1...
Send element 15...
Send element 16...
Send element 17...
Close chan1...
Timeout!
--End.
```

我们先来关注最关键的部分。请注意这样的内容：Reset chan1...。它是在main函数中的通道类型变量chan1即将被重新初始化的时候被打印出来的。在该内容被第一次打印出来之前，每当针对通道chan1的发送操作被进行一次，都紧接着会有一行代表了针对该通道的接收操作已被执行的内容被打印出来，如下所示：

```
Send element 2...
Receive a element: 2
```

这表明我们的程序正在按照我们的意图运行。一旦chan1通道中出现了新的元素值，receive函数中的程序就会立刻接收并打印它。

但是，当第一个Reset chan1...被打印出来之后，我们就只见发送、不见接收了，如：

```
Reset chan1...
Send element 3...
Send element 4...
Send element 5...
Reset chan1...
Send element 6...
Send element 7...
```

而输出的最后3行

```
Close chan1...
Timeout!
--End.
```

也表明，直到main函数的代码在发送完所有元素值之后关闭了通道chan1，receive函数中的接收操作仍然没有被唤醒（注意，我们把接收表达式<-chan2的结果值赋给了两个变量，读者应该知道这意味着什么）。这是怎么回事呢？

正如我们在前面讲述的那样，一旦在main函数中的针对chan1的重新赋值操作完成，后续的发送操作就会把元素值发送到新的通道值中。而在receive函数中的接收操作却仍然针对的是之前的那个通道值。这样就使本该操作同一个通道值的两个操作不再配对了。接收操作针对的通道中不会再有新的元素值，而那个发送操作向另一个通道发送的元素值也不会被任何接收方接收。其根本原因是，对变量的重新赋值的效果是无法被传递的。当我们想传递这种改变的时候，上述程序是无法让我们如愿的。显然，这不是Go语言本身的问题，而是我们的编码方式的问题。

经过上述试验我们可以看出，针对原有通道的替换动作导致了相应的接收操作的异常，同时其结果也无法被传递。但是，我们的网络爬虫框架需要的恰恰是这种传递，因为被用来传输各类数据的通道可能会在爬取流程的执行过程中被替换。由于这个爬取流程是由网络爬虫框架中的众多组件共同参与执行的，所以必须要让对通道的替换动作瞬时被各个组件觉察到。

我们已经知道，热替换可以解决此类问题。这一设计技巧的灵感仍然来自于面向对象编程世界中的“面向接口编程”原则。该原则主张模块对外提供抽象的接口而不是具体的实现。

依照这个思路，我们需要对前面的代码进行一些改造。首先，我们需要声明一个被用来获取通道的函数：

```
func getChan() chan int {
    return chan1
}
```

这个函数就相当于我们为通道chan1的外部使用方提供的接口。然后，我们需要让receive函数中的代码获取该通道的途径由它的参数chan2转变为对函数getChan的调用。这只需要修改极少的代码：

```
func receive() {
    fmt.Println("Receive element from chan1...")
    timer := time.After(30 * time.Second)
Loop:
    for {
```

```

        select {
        case e, ok := <-getChan():
            // 省略若干条语句
        case <-timer:
            // 省略若干条语句
        }
    }
    fmt.Println("--End.")
}

```

注意，只有select语句中的第一个case表达式被修改了。这意味着，select语句的每次执行都会伴随着对调用表达式getChan()的求值。而该调用表达式的求值结果永远会是通道chan1在当时的值。这就避免了因对chan1的重新初始化而导致的元素值无法被正确传递的问题。

通过这样的改造之后，我们再来运行这个命令源码文件，标准输出上会出现如下内容：

```

Receive element from chan1...
Send element 0...
Receive a element: 0
Send element 1...
Receive a element: 1
Send element 2...
Receive a element: 2
Reset chan1...
Send element 3...
Receive a element: 3
Send element 4...
Receive a element: 4
Send element 5...
Receive a element: 5
Reset chan1...
Send element 6...
Receive a element: 6
Send element 7...
Receive a element: 7
Send element 8...
Receive a element: 8
Reset chan1...
Send element 9...
Receive a element: 9
Send element 10...
Receive a element: 10
Send element 11...
Receive a element: 11
Reset chan1...
Send element 12...
Receive a element: 12
Send element 13...
Receive a element: 13
Send element 14...
Receive a element: 14
Reset chan1...
Send element 15...
Receive a element: 15

```

```

Send element 16...
Receive a element: 16
Send element 17...
Receive a element: 17
Close chan1...
--Closed chan1.
--End.

```

这与我们在前面看到的输出内容有两个不同。

- 无论main函数中的代码对chan1的重新初始化（由输出内容Reset chan1...代表）有多少次，receive函数中的代码都能够及时、正确地从通道中接收元素值。这从

```

Send element 8...
Receive a element: 8

```

和

```

Send element 16...
Receive a element: 16

```

等输出内容上就能看得出来。

- 当main函数中的代码关闭chan1变量代表的通道的时候（由输出内容Close chan1...代表），receive函数中的代码立即就能觉察到并打印出--Closed chan1.。

上述的两个不同表明，收发双方始终操作的是同一个通道。在任何时刻都没有出现操作不配对的情况。这完全得益于作为接口的函数getChan。它能为我们提供热替换功能。

当然，我们在网络爬虫框架中也可以运用这一设计技巧。通道管理器就是一个绝佳的应用场景。实际上，我们先前声明的通道管理器的接口ChannelManager中的方法ReqChan、RespChan、ItemChan和ErrorChan就相当于分别针对于各类通道的接口，而这些通道的具体实现则被隐藏在了通道管理器的内部。需要使用它们的程序仅能通过这些接口来获得对应的通道实例。

据此，我们已经可以给出通道管理器实现的基本结构了：

```

// 通道管理器的实现类型。
type myChannelManager struct {
    channellen uint           // 通道的长度值。
    reqCh       chan base.Request // 请求通道。
    respCh      chan base.Response // 响应通道。
    itemCh      chan base.Item    // 条目通道。
    errorCh     chan error        // 错误通道。
    status      ChannelManagerStatus // 通道管理器的状态。
}

```

结构体类型myChannelManager中的这6个字段是与接口类型ChannelManager中的方法一一对应的。所以在这里我们也不需要再次解释。不过，这只是第一个版本。后面我们还会有所添加。

2. 初始化

一旦有了基本结构，我们需要考虑的一个很重要的方面就是对这些字段的初始化方式。对于myChannelManager来说，我们只给定通道的长度就足够了。我们可以编写出一个名为NewChannelManager的函数，并让其负责创建并初始化通道管理器。该函数只需接受一个代表了通道长度的参数值。

记得吗？ChannelManager类型中包含了一个名为Init的方法。该方法被定义为初始化通道管理器的方法。因此，我们还应该在NewChannelManager函数中调用myChannelManager类型值的Init方法，以达到分清职责和重用初始化代码的目的。所谓分清职责是指，让Init方法全权负责对其所属值的初始化工作，而NewChannelManager函数仅负责校验调用方传来的参数值。这样做也可以把初始化通道管理器的代码集中于一处，以达到复用的目的。

好了，我们现在给出NewChannelManager函数的声明：

```
// 创建通道管理器。
// 如果参数channellen的值为0，那么它会由默认值代替。
func NewChannelManager(channellen uint) ChannelManager {
    if channellen == 0 {
        channellen = defaultChanLen
    }
    chanman := &myChannelManager{}
    chanman.Init(channellen, true)
    return chanman
}
```

可以看到，该函数的函数体中用到了一个名为defaultChanLen的包级私有的全局变量。它代表了默认的通道长度。当然，它只在参数channellen的值为0的时候才起作用，并充当即将被初始化的通道管理器中的通道长度。注意，chanman变量的类型是*myChannelManager，而不是myChannelManager。这意味着我们想把*myChannelManager类型作为ChannelManager接口类型的实现类型。

在校验了参数channellen的值之后，新创建的通道管理器chanman的Init方法将被调用。下面我们来看看这个方法都应该做哪些事。

方法Init首先应该初始化chanman中的所有字段。有了通道长度channellen，这项任务就很容易了。第一个版本的Init方法的声明如下：

```
func (chanman *myChannelManager) Init(channellen uint, reset bool) bool {
    if channellen == 0 {
        panic(errors.New("The channel length is invalid!"))
    }
    chanman.channellen = channellen
    chanman.reqCh = make(chan base.Request, channellen)
    chanman.respCh = make(chan base.Response, channellen)
    chanman.itemCh = make(chan base.Item, channellen)
    chanman.errorCh = make(chan error, channellen)
    chanman.status = CHANNEL_MANAGER_STATUS_INITIALIZED
    return true
}
```

在该方法中，我们进行了非常严格的参数检查。一旦参数channellen的值不符合要求，就会引发一个运行时恐慌。对于一个公开的初始化方法来说，这样做并不过分。把隐患扼杀在摇篮之中总要比困难地排查隐晦的错误好得多。

方法Init需要做的另一件事是避免意外的重复初始化。这项工作同样重要。因为由于使用方的误操作而使通道管理器被意外地重新初始化是一件很糟糕的事情。为此，我们需要再向Init方法中添加两行代码：

```
func (chanman *myChannelManager) Init(channellen uint, reset bool) bool {
    if channellen == 0 {
        panic(errors.New("The channel length is invalid!"))
    }
    if chanman.status == CHANNEL_MANAGER_STATUS_INITIALIZED && !reset {
        return false
    }
    // 省略若干条语句
    return true
}
```

如果通道管理器已处于已初始化状态，那么我们就应该去检查参数reset的值。如果该值为false，那么就说明Init方法的调用方并不想对通道管理器进行重新初始化。这时我们就应该忽略后续的所有操作并直接返回false。

在一般的情况下，这样的逻辑可以有效地避免意外重新初始化的情况发生。但是别忘了，通道管理器是一个公用的工具。它的所有方法都可能被并发地调用。虽然通道本身是并发安全的，但是与它们绑定在一起的变量却不是。我们需要采用一些手段保证针对通道管理器中的那些通道类型的字段的读写操作的并发安全。显然，sync包中的读写锁能够满足这样的需求。

我们需要在myChannelManager类型的基本结构加入一个sync.RWMutex类型的字段：

```
rwmutex    sync.RWMutex    // 读写锁。
```

通道管理器的Init方法会对内部进行初始化，其中就包括了对那4个通道类型的字段赋值。因此，我们需要对刚刚编写的myChannelManager类型的Init方法加以改造。同样是添加两行代码：

```
func (chanman *myChannelManager) Init(channellen uint, reset bool) bool {
    if channellen == 0 {
        panic(errors.New("The channel length is invalid!"))
    }
    chanman.rwmutex.Lock()
    defer chanman.rwmutex.Unlock()
    if chanman.status == CHANNEL_MANAGER_STATUS_INITIALIZED && !reset {
        return false
    }
    // 省略若干条语句
    return true
}
```

注意，在我们调用rwmutex的Lock方法之后，要记得立即编写一条defer语句，并包含针对该读写锁的Unlock方法的调用表达式。这样才能够保证当前函数被退出执行时，该读写锁总会被解锁，因而消除死锁的隐患。

经过了3个版本，我们的Init方法已经基本完善。不过，为了让该方法的接收者类型成为ChannelManager接口类型的实现类型，我们还需要为它继续编写几个方法。

3. 关闭

具体来讲，通道管理器的Close方法需要做3件事：检查状态、关闭所有通道和设置状态。当然，这一切依然要在读写锁的保护下进行。

只有在通道管理器正处于已初始化状态的时候，关闭通道才有意义。所以，一旦通道管理器

未处于此状态就应该直接返回false。还记得吗？Close方法的唯一结果是bool类型的。关于关闭内部通道的方法，我们就不多说了。这需要用到的内建函数close。最后，我们需要把常量CHANNEL_MANAGER_STATUS_CLOSED的值赋给当前的myChannelManager类型值的status字段。还记得吗？CHANNEL_MANAGER_STATUS_CLOSED常量是我们编写ChannelManager接口类型的时候声明的。它代表了通道管理器的已关闭状态。综上所述，我们应该这样编写myChannelManager类型的指针方法Close：

```
func (chanman *myChannelManager) Close() bool {
    chanman.rwmutex.Lock()
    defer chanman.rwmutex.Unlock()
    if chanman.status != CHANNEL_MANAGER_STATUS_INITIALIZED {
        return false
    }
    close(chanman.reqCh)
    close(chanman.respCh)
    close(chanman.itemCh)
    close(chanman.errorCh)
    chanman.status = CHANNEL_MANAGER_STATUS_CLOSED
    return true
}
```

4. 获取通道

我们先前为通道的获取而设计的方法有4个：ReqChan、RespChan、ItemChan和ErrorChan。这4个方法的实现应该是非常相似的。我们在这里以ReqChan方法为例。ReqChan方法的实现中必定包含了对当前myChannelManager类型值的reqCh字段的读操作。因此我们在这里依然会用到读写锁rwmutex。

在ReqChan这样的通道获取方法中，我们需要重点考虑的应该是在当前通道管理器处于不同状态的时候的处理方式。在获取通道的时候，通道管理器的状态理应是已初始化的。当发现不被期望的状态时，我们应该给予通道获取者必要的提示，以帮助它们作判断和采取下一步行动。不过，这部分职责并不应该是通道获取方法应该有的。因此，我们应该专门编写一个方法来做这件事。根据刚刚的描述，一个名为checkStatus的方法诞生了：

```
// 检查状态。在获取通道的时候，通道管理器应处于已初始化状态。
// 如果通道管理器未处于已初始化状态，那么本方法将会返回一个非nil的错误值。
func (chanman *myChannelManager) checkStatus() error {
    if chanman.status == CHANNEL_MANAGER_STATUS_INITIALIZED {
        return nil
    }
    statusName, ok := statusNameMap[chanman.status]
    if !ok {
        statusName = fmt.Sprintf("%d", chanman.status)
    }
    errMsg :=
        fmt.Sprintf("The undesirable status of channel manager: %s!\n",
            statusName)
    return errors.New(errMsg)
}
```

细心的读者可能会发现，我们在checkStatus方法中用到了标识符statusNameMap。从语法上看，它代表的应该是一个字典类型的值。确实，我们在goc2p项目的代码包webcrawler/middleware中有这样一个声明：

```
// 表示状态代码与状态名称之间的映射关系的字典。
var statusNameMap = map[ChannelManagerStatus]string{
    CHANNEL_MANAGER_STATUS_UNINITIALIZED: "uninitialized",
    CHANNEL_MANAGER_STATUS_INITIALIZED:    "initialized",
    CHANNEL_MANAGER_STATUS_CLOSED:         "closed",
}
```

声明这个字典的主要目的是在需要时以更加易读的方式反映出通道管理器的状态。文字（单词、短语等）要比单纯的数字更容易让人理解。

有了checkStatus方法，ReqChan方法的职责就足够单一和清晰了，并且可以很容易地编写出来：

```
func (chanman *myChannelManager) ReqChan() (chan base.Request, error) {
    chanman.rwmutex.RLock()
    defer chanman.rwmutex.RUnlock()
    if err := chanman.checkStatus(); err != nil {
        return nil, err
    }
    return chanman.reqCh, nil
}
```

再次强调，在获取通道的过程中，施加rwmutex的读锁是非常有必要的。这可以保证通道管理器中的那些通道类型的字段的状态一致性。

以ReqChan方法为参照，读者可以自己实现myChannelManager类型的指针方法RespChan、ItemChan和ErrorChan。这非常容易。

5. 获取摘要信息

通道管理器的Summary方法应该返回反映其内部状态的摘要信息。这个摘要信息应该足以让我们了解到通道管理器的所有的运行时细节。

为了实现这样一个方法，我们首先要编写出一个摘要信息的模板。其内容应该就像一道填空题那样。这道题的题干应该留出一些空当以等待填入通道管理器的状态以及其中各类通道实例的实时长度和固定容量。我们把包含上述内容的文字以fmt代码包中的函数能够识别的格式书写出来并赋给一个全局变量：

```
var chanmanSummaryTemplate = "status: %s, " +
    "requestChannel: %d/%d, " +
    "responseChannel: %d/%d, " +
    "itemChannel: %d/%d, " +
    "errorChannel: %d/%d"
```

在确定了摘要信息的模板之后，我们在*myChannelManager类型的Summary方法中对它进行填空即可：

```
func (chanman *myChannelManager) Summary() string {
    summary := fmt.Sprintf(chanmanSummaryTemplate,
```

```

        statusNameMap[chanman.status],
        len(chanman.reqCh), cap(chanman.reqCh),
        len(chanman.respCh), cap(chanman.respCh),
        len(chanman.itemCh), cap(chanman.itemCh),
        len(chanman.errorCh), cap(chanman.errorCh))
    return summary
}

```

其中，我们通过使用字典statusNameMap让摘要信息的可读性更好了。

6. 其他方法

至此，我们离让*myChannelManager类型成为接口类型的实现的目标只有一步之遥。我们还需要为myChannelManager类型添加两个指针方法，即ChannelLen方法和Status方法。相比于前面讲述的方法，这两个方法的实现就太简单了。它们只需分别返回myChannelManager类型中的相应字段的值即可。虽然如此简单，但如果没有了它们，NewChannelManager函数照样通不过编译。

到这里，我们就完成了对通道管理器的一个实现的编写。虽然它所管理的各类通道自身具有并发安全性，但是我们仍然使用了读写锁。很显然，这个读写锁的作用目标是通道管理器本身。它保证了通道管理器的内部状态的一致性。当然，这也有它的字段status的功劳。

通道管理器管理着各类通道。这使得调度器可以轻易地在各个处理模块之间搬移数据。不过，这些容量固定的通道有时候也会带来麻烦。这需要在调度器的实现中加入一些特殊的处理。这在后面会加以论述。总之，这种将通道集中管理的方式有助于各组件代码的简洁、职责的清晰，同时还使各个通道可以被热替换。

9.5.2 实体池

我们为实体池声明的接口类型Pool中包含了4个方法。其中，方法Take和Return的功能分别是“取出”实体和“归还”实体。而方法Total和Used则是被用于向外部提供实体池的真实状态。

我们在前面已经实现过Goroutine票池。它对实体池的实现有很高的参考价值。事实上，我们把前者的“票”改为“实体”即可。下面是具体的实现方法。

首先，我们要确定实体池的基本结构。与Goroutine票池相同，我们使用一个通道作为池中实体的容器。而与之不同的是通道的元素类型。我们在讲实体池接口的时候已经对此有所说明。因此，代表实体容器的字段应该是这样的：

```
container chan Entity // 实体容器。
```

我们已经知道，Entity是一个接口类型。同时它也是实体池中各实体的静态类型。但是，我们只知道实体的静态类型是不够的。我们还需要明确它们的动态类型（也即实际类型）。这样才能够去检查将要入池的实体的类型是否符合要求。

由于Go语言不支持自定义泛型，所以我们不能很自然地约束和检查实体的类型。幸好标准库代码包reflect中的API可以帮助我们满足需求。我们在前面已经接触过reflect包。其中的Type类型可以用来表示某个类型。所以，我们在实体池的基本结构中加入一个这样的字段以表示实体的实际类型：

```
etype    reflect.Type // 池中实体的类型。
```

我们在本书中声明的大多数类型都有着自己的初始化方式。实体池中的各个实体的实际类型可以是任意的，只要它实现了Entity接口。实体池很难也不应该去关注所有这些类型的每一种初始化方式，即使这些方式都非常简易。这就需要我们引导实体池的使用者自己向实体池提供池中实体的初始化方法。这样，实体池在初始化自身的过程中就可以使用此方法创建和初始化所需的实体值了。

由于Go语言视函数为一等类型，所以我们可以直接把一个函数类型作为实体池的字段类型，如下：

```
genEntity func() Entity // 池中实体的生成函数。
```

字段genEntity的值将会代表池中实体的初始化方法。它完全由实体池的使用者提供。我们会在后面看到使用它的方式和时机。

实体池的第四个字段是total。这个uint32类型的字段的值会忠实地反映出使用者对实体池容量的期望。这一期望应该总是能够被实现。

综上所述，我们可以编写出实体池的实现类型的基本结构：

```
// 实体池的实现类型。
type myPool struct {
    total    uint32    // 池的总容量。
    etype    reflect.Type // 池中实体的类型。
    genEntity func() Entity // 池中实体的生成函数。
    container chan Entity // 实体容器。
}
```

实体池也需要自己的初始化方式。为此，我们编写了NewPool函数。其声明如下：

```
// 创建实体池。
func NewPool(
    total uint32,
    entityType reflect.Type,
    genEntity func() Entity) (Pool, error) {
    // 省略若干条语句
}
```

可见，myPool类型中的绝大部分字段的值都由这个函数的参数给定。唯一例外的是container字段。对于外部给定的值，我们应该实行严格校验、及时报错的策略。具体如下。

□ 参数total的类型使得它的值不可能是负整数。但是，0也是不符合要求的。为此，我们需要这样来检查它：

```
if total == 0 {
    errMsg :=
        fmt.Sprintf("The pool can not be initialized! (total=%d)\n", total)
    return nil, errors.New(errMsg)
}
```

当发现参数total的值为0时，我们及时中止了对实体池的初始化，并直接生成和返回相应的错误值。

- 我们并不需要急于检查参数entityType和genEntity的值。因为我们紧接着会在对实体池的container字段的初始化的过程中用到它们。这同样会保证我们刚刚所说的检查策略的实施。

下面我们就来说说怎样初始化container字段。在通常情况下，我们应该在初始化一个池的时候就把它填满。尤其是在实体的创建成本很低或需要提前准备以尽量减少后续操作的响应时间的情况下。

既然是这样，我们就应该在创建和初始化一个chan Entity类型值之后立刻填满它：

```
size := int(total)
container := make(chan Entity, size)
for i := 0; i < size; i++ {
    newEntity := genEntity()
    if entityType != reflect.TypeOf(newEntity) {
        errMsg :=
            fmt.Sprintf("The type of result of function genEntity() is NOT %s!\n",
entityType)
        return nil, errors.New(errMsg)
    }
    container <- newEntity
}
```

在上述代码中，我们使用genEntity参数的值生成实体值，然后把它们陆续发送给container。不过，在进行发送操作之前，我们还需要检查由genEntity生成的实体值的实际类型是否符合要求。这里所说的要求也是由使用者给定的。它由entityType参数的值代表。一旦发现新实体值的实际类型与entityType参数的值所代表的类型不一致，我们会立即忽略掉后续的初始化工作并直接生成和返回相应的错误值给NewPool函数的调用方。

如果执行了这些代码之后仍未出错，那么我们就可以认定调用方给出的这些参数值都是合法的，并可以放心大胆地使用了。在这之后，我们创建一个*myPool类型值，然后将它作为NewPool函数的第一个结果值：

```
pool := &myPool{
    total:    total,
    etype:    entityType,
    genEntity: genEntity,
    container: container,
}
return pool, nil
```

读者把上面这3段代码按照出现顺序拼接起来，就可以得出NewPool函数的函数体了。注意，我们最后返回的第一个结果值是*myPool类型的。读者应该知道这意味着什么。

在明确了myPool类型的基本结构及其初始化方式之后，我们就可以来编写接口类型Pool中声明的那4个方法了。注意，在myPool类型中，这4个方法都应该是指针方法，而不应该是值方法。

首先，Take方法的实现是相当简单的。我们只需要从container字段代表的通道处接收一个元素值（即实体值）并返回给方法的调用方即可。不过，Take方法的结果声明有两个。第二个结果是error类型的。既然存在第二个结果，那么我们就充分利用。我们可以把接收表达式的结

果赋给两个变量。如果第二个结果的值为false，那么就说明container通道已经被关闭。这时，Take方法就应该直接返回表示此问题的错误值。虽然这种情况几乎不可能发生，但是我们仍然有必要对它进行检测和处理。Take方法的完整声明如下：

```
func (pool *myPool) Take() (Entity, error) {
    entity, ok := <-pool.container
    if !ok {
        return nil, errors.New("The inner container is invalid!")
    }
    return entity, nil
}
```

相比之下，Return方法的实现要复杂一些。这是因为，该方法是要将方法调用者传入的Entity类型值发送给container通道。在进行发送操作之前，Return方法必须对该值进行严格的检查。该检查至少分为两步。第一步就是非nil检查。一个有效的实体值必定不为nil。否则，Return方法不予以处理，然后生成并返回一个表示此问题的错误值。如果此步通过，那么就需要检查该值的实际类型是否与实体池的etype字段代表的类型一致。这非常重要。因为Return方法的调用方可能会有意或无意地将一个类型不符的值传递进来。在这种情况下，Return方法应该不予以处理并及时报错。这一步骤保证了实体池中的实体的类型一致性，同时也确保了它的Take方法的调用方能够得到预期类型的值。这两步检查的代码如下：

```
if entity == nil {
    return errors.New("The returning entity is invalid!")
}
if pool.etype != reflect.TypeOf(entity) {
    errMsg := fmt.Sprintf("The type of returning entity is NOT %s!\n", pool.etype)
    return errors.New(errMsg)
}
```

如果调用方传入的Entity类型值通过了上述两个步骤的检查，那么就说明该值是符合要求的。但是，我们的检查却并未结束。要知道，我们无法限制外部对实体池的Return方法的调用次数。并且，即使一个Entity类型值并不是实体池最初在内部生成的，它也仍然可以被传递给Return方法并通过前面的检查。所以，我们还需要考虑一种情况：如果被传给Return方法的值不是最初在NewPool函数中生成的，那么我们应该怎样处理它。如果要阻止这样的值被发送到container通道，那么我们又该怎样去鉴别它们呢？

还记得吗？Entity接口类型中那个唯一的方法声明是：

```
Id() uint32
```

该方法会返回一个能够唯一标识池中实体值的整数。我们可以记录下池中实体值的Id方法的结果值，并利用该值来鉴别某个Entity类型值是否是池中原有的实体值。为了记录下这些ID，我们需要再在myPool类型的基本结构中加入一个字段。该字段的声明如下：

```
idContainer map[uint32]bool // 实体ID的容器。
```

字典idContainer被用来存储池中实体的ID。我们可以通过这样一个字典快捷地辨别一个Entity类型值的有效性（即是否是池中的实体）。

若要让它可用，我们需要在NewPool函数中对它进行初始化。这里所说的初始化并不仅仅指使用make函数创建并初始化一个字典值并将其赋给idContainer字段，还意味着要保持它与container通道的同步。为此，我们要对该函数中的这段代码：

```
size := int(total)
container := make(chan Entity, size)
for i := 0; i < size; i++ {
    newEntity := genEntity()
    // 省略若干条语句
    container <- newEntity
}
```

进行修改。这包括创建和初始化一个map[uint32]bool类型的字典值，以及在把新实体newEntity发送给container通道之后立即将该实体的ID作为键添加到之前创建的字典值中。因此，前面那段代码就被改为：

```
size := int(total)
container := make(chan Entity, size)
idContainer := make(map[uint32]bool)
for i := 0; i < size; i++ {
    newEntity := genEntity()
    // 省略若干条语句
    container <- newEntity
    idContainer[newEntity.Id()] = true
}
```

此外，我们还需要在最后初始化myPool类型值的时候，把这里的idContainer变量赋给该值的idContainer字段：

```
pool := &myPool{
    total:      total,
    etype:      entityType,
    genEntity:  genEntity,
    container:  container,
    idContainer: idContainer,
}
```

在做好上述准备之后，可以编写*myPool的Return方法中的第三步检查了。其代码如下：

```
if _, ok := pool.idContainer[entity.Id()]; !ok {
    errMsg := fmt.Sprintf("The entity (id=%d) is illegal!\n", entity.Id())
    return errors.New(errMsg)
}
```

即使Return方法接受的参数值不为nil、类型符合要求并且也是由NewPool函数中的代码创建并发送给当前实体池的container通道的，这个值也不一定就应该被归还。我们应该考虑到同一个Entity类型值可能会被归还多次的情况。一个Entity类型值仅仅在已被取出并且还未被归还的情况下才应该作为Return方法的参数值。反过来讲，只有这时，Return方法才应该认可并处理它。

为了做到这一点，我们需要更加充分地利用idContainer字段。我们已经知道，该字段代表的字典的元素类型是bool类型的。利用这些字典元素的值，我们就可以表示与之对应的ID所属的

实体的状态。`true`代表未被取出或已被归还，`false`代表已被取出且未被归还。根据这一定义，我们对`Return`方法的参数`entity`的第四步检查的代码如下：

```
if v := pool.idContainer[entity.Id()]; v {
    errMsg := fmt.Sprintf("The entity (id=%d) is already in the pool!\n", entity.Id())
    return errors.New(errMsg)
}
```

可以看到，我们在这里依然使用的是针对`idContainer`字典的索引表达式。因此，我们可以把第三步和第四步检查合并一下：

```
entityId := entity.Id()
v, ok := pool.idContainer[entityId]
if !ok {
    errMsg := fmt.Sprintf("The entity (id=%d) is illegal!\n", entityId)
    return errors.New(errMsg)
}
if v {
    errMsg := fmt.Sprintf("The entity (id=%d) is already in the pool!\n", entityId)
    return errors.New(errMsg)
}
```

在完成了这4步检查之后，我们就可以放心地把`entity`发送给`container`通道了。不过别忘了，为了让最后一步的检查正确无误，我们同时还要修改`idContainer`字典中对应的键值对。相应的代码如下：

```
pool.idContainer[entityId] = true
pool.container <- entity
return nil
```

最后那条`return nil`语句表示相应实体已被归还给实体池。

对于`Take`方法，我们也需要做一些改动，为的是正确表示池中实体的已被取出且还未被归还的状态。改造后的`Take`方法如下：

```
func (pool *myPool) Take() (Entity, error) {
    entity, ok := <-pool.container
    // 省略若干条语句
    pool.idContainer[entity.Id()] = false
    return entity, nil
}
```

至此，通过`myPool`类型的`idContainer`字段，我们可以判断出一个`Entity`类型值是否是池中之物，以及它是否已被取出且未被归还。这使得实体池中的每个实体都有了自己的状态，也使得`Return`方法中对参数值的检查更加严谨了。不过，这又引入了一个新的问题。`myPool`类型的`container`字段是通道类型的。它本身就是并发安全的，所以无论我们怎样操作它都不用顾及同步的问题。而`myPool`类型的`idContainer`字段是字典类型的。Go语言的字典类型自身并未对并发安全性做出任何保证。那么，我们需要使用额外的同步方法来保证这一点吗？显然，答案是肯定的。

我们使用互斥锁来解决这个问题。这个互斥锁需要保证两处代码的串行执行。它们分别在`Take`方法和`Return`方法中。为了实施这项改造，我们需要先为`myPool`类型添加如下字段：

```
mutex      sync.Mutex    // 针对实体ID容器操作的互斥锁。
```

对于Take方法，由于需要被保护的语句有一条，且它紧挨在最后的return语句之前，所以我们直接在需要被保护的语句的前面添加相应的互斥锁操作即可。改造后的Take方法如下：

```
func (pool *myPool) Take() (Entity, error) {
    entity, ok := <-pool.container
    if !ok {
        return nil, errors.New("The inner container is invalid!")
    }
    pool.mutex.Lock()
    defer pool.mutex.Unlock()
    pool.idContainer[entity.Id()] = false
    return entity, nil
}
```

方法Return中的情况要相对复杂一些。因为其中包括了对idContainer字典的读操作和写操作，并且写操作是否需要被执行，取决于读操作的结果值。具体代码如下：

```
v, ok := pool.idContainer[entityId]
if !ok {
    errMsg := fmt.Sprintf("The entity (id=%d) is illegal!\n", entityId)
    return errors.New(errMsg)
}
if v {
    errMsg := fmt.Sprintf("The entity (id=%d) is already in the pool!\n", entityId)
    return errors.New(errMsg)
}
pool.idContainer[entityId] = true
```

注意，这段代码中的一些语句是不需要被保护的，比如errMsg变量的声明语句。

我们直接在这段代码的前面添加相应的互斥锁操作是不妥当的。因为在它之后且在最后那条return语句之前还有针对通道container的发送操作。这样做非常容易锁死相关的Goroutine，并造成爬取流程的停滞不前。

正确的做法是：把需要保护的语句抽离出来并放到一个独立的方法中，以便于我们尽量缩小临界区的大小，以及方便使用defer语句来保证互斥锁的释放。具体步骤是，先在独立的方法中进行针对idContainer字典的读操作，然后根据读操作的结果来决定是否执行写操作。并且，创建出可以表示这一个读写过程的结果的值，并返回给该独立方法的调用方（也就是Return方法）。在Return方法中，我们根据该独立方法的结果值作出选择，是生成并返回相应的错误值还是把给定的实体发送给container通道。

依照这个思路，我们可以编写出如下方法：

```
// 比较并设置实体ID容器中与给定实体ID对应的键值对的元素值。
// 结果值：
//      -1: 表示键值对不存在。
//      0: 表示操作失败。
//      1: 表示操作成功。
func (pool *myPool) compareAndSetForIdContainer(
    entityId uint32, oldValue bool, newValue bool) int8 {
    pool.mutex.Lock()
```

```

    defer pool.mutex.Unlock()
    v, ok := pool.idContainer[entityId]
    if !ok {
        return -1
    }
    if v != oldValue {
        return 0
    }
    pool.idContainer[entityId] = newValue
    return 1
}

```

对于compareAndSetForIdContainer方法，无需再做解释。该方法的注释已经说得很清楚了。

接下来，我们需要对Return方法进行相应的改造。在其中的从entityId := entity.Id()语句后面到该方法体结束的那段代码应变为：

```

casResult := pool.compareAndSetForIdContainer(entityId, false, true)
if casResult == 1 {
    pool.container <- entity
    return nil
} else if casResult == 0 {
    errMsg := fmt.Sprintf("The entity (id=%d) is already in the pool!\n", entityId)
    return errors.New(errMsg)
} else {
    errMsg := fmt.Sprintf("The entity (id=%d) is illegal!\n", entityId)
    return errors.New(errMsg)
}

```

其中，把针对container通道的发送操作提到靠前的位置，是为了能够在条件满足的情况下尽快地归还实体。

现在，实体池类型myPool的“取出”和“归还”功能已经基本完善了。不过，我们还需要让*myPool类型成为Pool接口的实现类型。它还缺少两个方法，即Total和Used。

显然，有了total字段和len函数，实现这两个方法会非常容易。所以我们在这里就不再赘述了。虽然简单，但是它们的作用还是重要的，尤其对于网络爬虫框架的监控机制来说。有了它们，我们就可以实时地了解到实体池的使用情况了。

在实现了myPool类型的指针方法Total和Used之后，我们可以使用go build命令编译webcrawler/middleware代码包。如果没有出现任何编译错误，那么就说明实体池接口及其实现在代码层面上已经完成并正确了（别忘了，与通道管理器相关的代码也在该代码包中，所以也要保证它们正确无误才能通过编译）。不过，它们是否可以按照我们的意愿运行起来，还要看对应的功能测试的结果。对于这些功能测试，我们在这里忽略不讲，读者可以试着去实现它。

9.5.3 停止信号

停止信号的接口类型StopSign中包含的方法声明不少。不过，代表核心操作的方法只有3个，即Sign、Signed和Deal。它们分别代表了发出信号、判断信号和处理信号的功能。它们体现了停止信号的本意。

我们把停止信号的实现类型的命名为myStopSign。构思它的基本结构首先要考虑的就是被用来承载停止信号的字段。我们在讲述其接口类型的时候说过，它不适合用通道类型来代表。因为这会给停止信号的发出方和处理方都带来很大的约束。实际上，一个停止信号只可能有两个状态：未发出和已发出。所以，我们使用一个bool类型的字段就完全可以表示信号本身和停止信号类型的实例的状态了：

```
signed    bool           // 表示信号是否已发出的标志位。
```

字段signed的初始值即是其类型的零值。当停止信号的Sign被调用时，我们就把true赋予该字段。而当Reset被调用时，我们就把该字段的值重新赋为false。

接口类型StopSign的声明表明，停止信号需要为每一个处理方的处理次数进行计数。每一个处理方都会有一个唯一的标识。在调用停止信号的Deal方法的时候，它们应该将相应的标识作为参数值传入。由于这个标识是唯一的，所以我们可以使用一个字典来满足计数的需求。因此，我们为myStopSign类型加入这样一个字段：

```
dealCountMap map[string]uint32 // 处理计数的字典。
```

与实体池的情况类似，我们应该会在一些方法中同时操作多个字段，并且想把处在同一个方法中的多个操作合为一个原子操作。为此，我们应该使用互斥锁来保证相关操作的并发安全性和原子性。我们对于上面这两个字段都有读操作和写操作的需求，比如，停止信号状态的变更和读取，以及处理计数的累加和读取。因此，我们应该使用读写锁来保护这些操作。这样，就有了myStopSign类型的第三个字段：

```
rwmutex    sync.RWMutex    // 读写锁。
```

有了基本结构，我们就可以开始编写被用来初始化停止信号的NewStopSign函数了。该函数的声明如下：

```
// 创建停止信号。
func NewStopSign() StopSign {
    ss := &myStopSign{
        dealCountMap: make(map[string]uint32),
    }
    return ss
}
```

注意，我们在初始化myStopSign类型值得时候，只为其中的一个字段赋了值。这是因为其他两个字段的零值就已经是可用的了。

我们接下来编写停止信号最核心的3个方法。这看起来并不难。Sign方法需要做的最重要的事就是把true赋给signed字段。而Signed方法的职责就是把signed字段的值真实的返回调用方。至于Deal方法，则是要根据参数code的值在dealCountMap字典中添加或修改相应的键值对。注意，对于Sign方法和Deal方法来说，在某种情况下是需要忽略实际的操作的。

先来看Sign方法。我们知道，网络爬虫框架中的各个组件在接收到停止信号之后，应该立即做好当前任务的善后处理并取消后续的所有任务。它们对停止信号的响应都是一次性的，同时也是不可逆的。我们可以通过停止并再次启动调度器来重新创建和初始化网络爬虫框架内的所有

组件。但是，这些组件都会是全新的实例，而那些在停止调度器之前还有效的那些组件实例都会被丢弃。正因为如此，我们没有必要发出停止信号多次。当然，如果停止信号已被重置，那就是另外一回事了。这一特性在Sign方法上的反映就是：当发现signed字段的值已经为true的时候，就应该忽略后续的对signed的赋值操作。其实，要不是为了准确地设定Sign方法的结果值以表示当次调用的成功与否，我们不检查signed字段的值而直接为它赋值也是完全没有问题的。综上所述，Sign方法的声明如下：

```
func (ss *myStopSign) Sign() bool {
    ss.rwmutex.Lock()
    defer ss.rwmutex.Unlock()
    if ss.signed {
        return false
    }
    ss.signed = true
    return true
}
```

之所以在实现如此简单的方法中加入对读写锁的操作，是因为我们不希望在同时发生多次调用的时候出现多于一个的结果值为true的情况。换句话说，无论在什么情况下，我们都希望只有一个发出停止信号的调用被真正处理。因此，对Sign方法中的代码的执行应该是互斥的。

与之形成鲜明对比的是Signed方法。在后者中，我们只需读取signed字段的值并将其作为方法的结果值，而无需添加任何前置和后置的检查和操作。因此，其中的代码也就没有互斥执行的需求。Signed方法的方法体只包含了一行代码，如下：

```
func (ss *myStopSign) Signed() bool {
    return ss.signed
}
```

再来说Deal方法。它的主要操作对象是dealCountMap字段。这势必会用到读写锁。另外，若发现停止信号还未被发出，则后续操作理应被忽略，否则就会导致处理计数的不准确。该方法的声明如下：

```
func (ss *myStopSign) Deal(code string) {
    ss.rwmutex.Lock()
    defer ss.rwmutex.Unlock()
    if !ss.signed {
        return
    }
    if _, ok := ss.dealCountMap[code]; !ok {
        ss.dealCountMap[code] = 1
    } else {
        ss.dealCountMap[code] += 1
    }
}
```

我们要说的停止信号的第4个方法就是Reset方法。这个方法很有用。其重要性仅次于前述的3个方法。我们可以通过调用它来重置停止信号。它是让停止信号从已发出状态转换为未发出状态的唯一方法。

方法Reset的实现也很简单。我们只需要把signed赋值为false，并清零所有的处理计数。也正因为包含了这两项操作，所以我们需要使用读写锁将它们保护起来。其声明如下：

```
func (ss *myStopSign) Reset() {
    ss.rwmutex.Lock()
    defer ss.rwmutex.Unlock()
    ss.signed = false
    ss.dealCountMap = make(map[string]uint32)
}
```

可以看到，我们在这里直接丢弃了dealCountMap字段的原有值，并为它绑定了一个新的值。这也是清零所有处理计数的最便捷的方式。

要想让*myStopSign类型成为StopSign接口类型的实现类型，我们还需要为其编写3个方法，即DealCount、DealTotal和Summary。这3个方法的实现都相当简单，所以我们只在这里提示一下要点。

我们应该使用rwmutex的读锁来保护DealCount方法和DealTotal方法中的代码，因为它们都需要从dealCountMap字典中获取相应的键值对。

至于Summary方法，我们就不用多说了。它的结果值应该忠实地反映出停止信号内部的即时状态，比如表示信号是否已发出的标志位和处理计数的集合。

好了，我们对停止信号的实现的介绍就暂时告一段落。读者在后面会看到，调度器会使用停止信号来实现网络爬虫框架的停止功能。

9.5.4 ID生成器

在本节的最后，我们再来说一说网络爬虫框架的中间件工具集中方法最少的那个工具——ID生成器。

ID生成器的接口类型IdGenertor中只包含了一个方法声明：

```
GetUint32() uint32 // 获得一个uint32类型的ID。
```

不过，为了实现这个接口，我们需要添加多个字段来支持。首先，需要明确的是，ID是非负整数且是递增的。也就是说，我们第一次调用一个ID生成器的GetUint32方法时会得到uint32类型的值0，而第二次调用则应该得到1，以此类推。因此，我们应该用一个字段来表示当前的ID的值，像这样：

```
sn          uint32    // 当前的ID。
```

其次，虽说uint32类型可以被用来表示近43亿个非负整数，但是万一我们对ID生成器的GetUint32方法的调用次数超过了这个数量那又该怎么办呢？这里的解决方案很简单——从头开始。也就是说，当发现前一个ID已是uint32类型所能表示的最大值的时候，我们就把当前的ID重新赋值为0。为了能够识别这种情况，我们还需这样一个字段：

```
ended bool    // 前一个ID是否已经为其类型所能表示的最大值。
```

最后，与大多数中间件工具一样，我们需要使用同步方法来保证其并发安全性。由此，我们需要声明的第3个字段如下：

`mutex sync.Mutex // 互斥锁。`

这3个字段有一个共同的特点：它们的类型的零值即是我们需要分别赋给它们的值。所以，被用于创建并初始化ID生成器的函数`NewIdGenertor`的函数体中只需包含一条语句：

```
return &cyclicIdGenertor{}
```

由于该函数的唯一结果的类型是`IdGenertor`，所以我们要保证`cyclicIdGenertor`的指针类型即为接口类型`IdGenertor`的实现类型。这就意味着，我们需要实现的`GetUint32`方法应为`cyclicIdGenertor`类型的指针方法，而不是值方法。

我们需要在`cyclicIdGenertor`类型的指针方法`GetUint32`中做的最重要的一件事，就是我们刚才说的检查前一个ID是否已为最大值并做出相应处理。这包括两个步骤。其中的一个步骤就是在得到`sn`字段的值之后，立即判断它是否是`uint32`类型所能表示的最大值。如果答案是肯定，那么我们就把`true`赋给`ended`字段。另一个步骤是与前者呼应的。它会在获取到`sn`字段的值之前先检查`ended`字段的值。如果该值为`true`，那么它就会直接将`sn`字段的值重置为0，并将其返回给方法的调用方。这样一来，我们就可以持续不断地从ID生成器中取出可用的ID了。读者不用担心ID可能会重复的问题，因为我们之前编写的实体池所容许的最大容量也不会超过`uint32`类型的取值范围。下面是`GetUint32`方法的完整声明：

```
func (gen *cyclicIdGenertor) GetUint32() uint32 {
    gen.mutex.Lock()
    defer gen.mutex.Unlock()
    if gen.ended {
        defer func() { gen.ended = false }()
        gen.sn = 0
        return gen.sn
    }
    id := gen.sn
    if id < math.MaxUint32 {
        gen.sn++
    } else {
        gen.ended = true
    }
    return id
}
```

若读者把这里的ID生成器用在了别处并且确实会调用ID获取方法超过43亿次，那么建议对这里的ID生成器进行扩展。这应该很容易。我在这里提供一种扩展的思路。

我们把`cyclicIdGenertor`类型的扩展类型命名为`cyclicIdGenertor2`。既然后者是前者的扩展，那么后者的基本结构中就应该包含一个类型为前者的字段：

```
// ID生成器的实现类型2。
type cyclicIdGenertor2 struct {
    base      cyclicIdGenertor // 基本的ID生成器。
    cycleCount uint64                  // 基于uint32类型的取值范围的周期计数。
}
```

我们可以看到，`cyclicIdGenertor2`的基本结构中还存在了第二个字段`cycleCount`。`cycleCount`字段的值代表了一个周期计数。这个周期计数是针对基本的ID生成器而言的。换句话说，该计数

值表示了基本ID生成器base所生成的ID已经第几次触碰到了uint32类型所能表示的最大值。我们利用这个计数值可以将从base处获取到的uint32类型的ID扩展为uint64类型的ID。请看下面的方法声明：

```
func (gen *cyclicIdGenertor2) GetUint64() uint64 {
    var id64 uint64
    if gen.cycleCount%2 == 1 {
        id64 += math.MaxUint32
    }
    id32 := gen.base.GetUint32()
    if id32 == math.MaxUint32 {
        gen.cycleCount++
    }
    id64 += uint64(id32)
    return id64
}
```

可以看到，GetUint64方法每次都会检查base的GetUint32方法返回的uint32类型的ID是否是uint32类型值的上限。如果答案是肯定的，那么就让cycleCount字段的值自增一次。

另外，之所以要在开始处把cycleCount字段的值以2取模，是因为uint64类型允许的最大值是uint32类型允许的最大值的两倍。每当cycleCount字段的值为2的倍数的时候，就说明GetUint64方法上一次返回的结果值已是uint64类型允许的最大值。不过，我们要关注的是已经超出uint32类型的取值范围但却仍在uint64类型的取值范围内的情况。这也是我们总是判断该取模操作的结果值是否等于1的原因。在此种情况下，把base的GetUint32方法返回的ID值与uint32类型允许的最大值相加，就可以得到一个64位的ID值了。并且，如此得到的64位的ID值依然是连续的。这时，uint32类型允许的最大值就相当于一个基数。

另一方面，在把cycleCount字段的值以2取模的结果值为0的情况下，我们并没有设置基数（相当于把基数也设置为0）。由于这时用uint64类型表示的ID值与用uint32类型表示的ID值是相同的，所以我们不用进行任何附加的操作。这也保证了cyclicIdGenertor2类型的指针方法GetUint64返回的ID值总会基于uint64类型的取值范围而循环。

如此，我们编写扩展的ID生成器cyclicIdGenertor2才得以正确地生成64位的ID值，并且其生成规则与原始的ID生成器的生成规则完全一致。

有了ID生成器，网页下载器和分析器都可以非常方便地为它们的实例生成全局唯一的ID。ID生成器自身是并发安全的。这也使得我们在使用它的时候不用有所顾虑。（请注意，扩展的ID生成器cyclicIdGenertor2并不是并发安全的，读者可以为它添加这一特性吗？）

9.6 处理模块的实现

有了前面设计并实现的那些中间件工具的支持，我们再来实现网络爬虫框架中的各个处理模块就会方便很多。在本节，我会带领读者逐一地实现这些处理模块。在这个过程中，读者也会体会到那些中间件工具带来的益处。所谓的“磨刀不误砍柴工”即是如此。

由于我们在对网络爬虫框架进行总体设计的时候，严格遵循了“单一职责”的原则，所以，

这些处理模块在功能上都并不复杂。不过,在实现它们的时候仍然存在一些需要特别处理的地方。下面我们就从相对简单的网页下载器开始讲起。

9.6.1 网页下载器

在本小节,我们将会详细讲解网页下载器以及容纳其集合的池的实现方法。在这个过程中,读者也可以看到一些中间件工具的实际应用。

1. 网页下载器的实现

网页下载器的功能是向目标服务器发送给定的请求以获得相应的网页内容。我们在对网络爬虫框架进行详细设计的时候,提到过代表HTTP客户端的类型`http.Client`。它可以发送HTTP请求并接受HTTP响应。我们的网页下载器恰恰需要使用这样的工具来实现功能。因此,在我们的网页下载器的实现类型中应该包含这样一个字段:

```
httpClient http.Client // HTTP客户端。
```

此外,每个网页下载器的实例都应该拥有一个仅属于自己的ID。所以下面的字段是必须的:

```
id          uint32      // ID。
```

在确定了基本结构之后,我们把网页下载器的实现类型命名为`myPageDownloader`。

按照惯例,我们需要编写一个被用来获取新的网页下载器实例的`NewPageDownloader`函数。这个函数仅仅接受一个`*http.Client`类型的参数`client`。之所以把这个参数的类型设定为`http.Client`类型的指针类型,是因为我们允许该函数的调用方不指定它。也就是说,该参数接受`nil`的赋值。在这种情况下,函数内部会以默认方式创建一个HTTP客户端实例,并把它赋给那个新的`myPageDownloader`类型值的`httpClient`字段。

至于为新的网页下载器实例分配ID的工作,我们就不用劳烦`NewPageDownloader`函数了。因为我们已经编写好了ID生成器。为了持有一个网页下载器专用的ID生成器,我们需要在当前包(也就是`goc2p`项目的`webcrawler/downloader`代码包)的源码文件中声明如下的包级私有的全局变量:

```
// ID生成器。
var downloaderIdGenertor mdw.IdGenertor = mdw.NewIdGenertor()
```

其中,限定标识符`mdw.NewIdGenertor`的前缀`mdw`代表了代码包`webcrawler/middleware`。这样的表示可以成立,是由于我们在导入这个代码包的时候为该代码包起了别名`mdw`。

在声明了变量`downloaderIdGenertor`之后,我们还需要声明这样一个函数:

```
// 生成并返回ID。
func genDownloaderId() uint32 {
    return downloaderIdGenertor.GetUint32()
}
```

添加`genDownloaderId`的函数可以将网页下载器与ID生成器完全解耦。虽然`mdw.IdGenertor`已经代表一个接口了,但是一旦有了上述函数,就可以对网页下载器彻底隐藏ID的具体生成方式了。并且,这种把对外部API的调用代码集中于一处的方法,也会提高之后代码维护工作的效率。

好了，在进行了一番准备之后，我们就可以很方便地编写出NewPageDownloader函数的函数体了。该函数的完整声明如下：

```
// 创建网页下载器。
func NewPageDownloader(client *http.Client) PageDownloader {
    id := genDownloaderId()
    if client == nil {
        client = &http.Client{}
    }
    return &myPageDownloader{
        id:      id,
        httpClient: *client,
    }
}
```

关于变量id和client的赋值方式，我们都已说明。注意，在使用复合字面量创建并初始化http.Client类型值的时候，我们可以不为它的任何字段赋值。如此即是以默认的方式创建一个HTTP客户端实例的方式。

此外，读者应该能够通过其中的return语句看出，我们是想让*myPageDownloader类型成为PageDownloader接口的实现类型。若要满足这个需求，我们就需要为myPageDownloader类型编写两个指针方法——ID和Download。其中，ID方法相当简单。它只需原样返回该类型的id字段的值即可：

```
func (dl *myPageDownloader) Id() uint32 {
    return dl.id
}
```

方法Download会接受一个base.Request类型（即webcrawler/base代码包中的结构体类型Request）的参数，并返回一个*base.Response的结果值和一个error类型的结果值。这个方法本身要做的工作并不复杂。我们把与目标服务器交换数据的工作都交给了httpClient字段代表的HTTP客户端。该HTTP客户端有一个名为Do的方法。该方法即可实现我们刚刚所说的数据交换工作。我们调用该方法并传入一个*http.Request类型的值，待相关操作完成之后，就可以得到一个*http.Response类型的值和一个error类型的值了。前者代表了目标服务器针对当前请求的响应，而后者则代表了可能发生的错误。若后者不为nil，则说明方法Do在被执行的过程中出现了错误。这时，Download方法应该立即将该错误值反馈给它的调用方。请看下面的代码：

```
httpResp, err := dl.httpClient.Do(httpReq)
if err != nil {
    return nil, err
}
```

其中，变量httpReq和httpResp即分别代表了请求和响应。我们在此前并没有声明过httpReq。不过我们马上就会说明。

我们已经知道，base.Request类型和base.Response类型分别相当于对*http.Request类型和*http.Response类型的一层薄薄的封装。HTTP客户端的Do方法所接受和返回的第一个值的类型分别为后两者，而网页下载器的Download方法所接受和返回的第一个值的类型分别为前两者。这就

意味着，这个Download方法还要有一个简单但却重要的职责：完成相关的封装和拆封。我们在编写网络爬虫框架的基本数据类型base.Request和base.Response时已经想到了这一点。因此，履行此职责只需编写相应的调用语句和复合字面量即可。

对于Download方法的参数req的值，调用其HttpReq方法即可获取到被包含在内的HTTP请求。变量httpReq就是这样被声明和赋值的：

```
httpReq := req.HttpReq()
```

再来说对于Download方法的第一个结果值的生成。如果HTTP客户端的Do方法的第二个结果值（那个error类型值）等于nil，那么就说明它与目标服务器的交互操作已经成功地完成了。这时，我们就需要这样来创建并初始化一个base.Response类型值并把它返回给Download方法的调用方：

```
return base.NewResponse(httpResp, req.Depth()), nil
```

我们现在来总结一下。只要我们有了一个HTTP客户端，就可以用它来创建和初始化网页下载器。这个工作由webcrawler/downloader包中的NewPageDownloader函数来完成。NewPageDownloader函数会使用同包中的genDownloaderId函数为其生成的网页下载器创建一个ID，并且还会在我们传入的HTTP客户端不可用时，自己生成一个以确保该网页下载器的可用性。网页下载器的Download方法为实现相应功能会进行3个步骤的操作。首先，它会获取到base.Request类型的参数值中的HTTP请求。其次，它会利用HTTP客户端与目标服务器交互，并得到一个HTTP响应和一个可能不为nil的错误值。最后，Download方法会根据这两个交互结果来生成它自己的结果值。

网页下载器的功能非常简单，但其角色却非常关键。因为网络爬虫要分析的所有内容都会从网页下载器那里获得。在大多数情况下，我们是不会满足于只有一个网页下载器在工作的。这是由于与其他环节相比，网络通讯所需的时间会多很多。它们的耗时往往不在一个数量级上。因此，我们就需要更多的网页下载器并发地与目标服务器交换数据。这就需要用某种方式来创建和管理一个网页下载器的集合。

2. 网页下载器池的实现

显然，我们会使用网页下载器池来做这件事。在9.4节中，我们已经给出了网页下载器池的接口类型。另外，我们之前还实现了一个实体池。它可以作为网页下载器池的内部实现的一部分。实际上，我们再对实体池稍加封装就可以得到一个网页下载器池的实现了。

首先请看网页下载器池的实现类型myDownloaderPool的基本结构：

```
// 网页下载器池的实现类型。
type myDownloaderPool struct {
    pool mdw.Pool // 实体池。
    etype reflect.Type // 池内实体的类型。
}
```

我们可以看到，mdw.Pool类型（即webcrawler/middleware代码包中的Pool类型）的字段pool已位列其中。除此之外，字段etype也是很有用的。它被用来表示实体池内的实体的实际类型。我们在实现myDownloaderPool类型的Take方法的时候会用到它。

按照惯例，函数NewPageDownloaderPool的用途是创建和初始化网页下载器池。下面是它的声明：

```
// 创建网页下载器池。
func NewPageDownloaderPool(
    total uint32,
    gen GenPageDownloader) (PageDownloaderPool, error)
```

该函数接受两个参数。第一个参数即代表了池的总容量，而第二个参数则为网页下载器的生成函数。其中的GenPageDownloader类型的声明如下：

```
// 生成网页下载器的函数类型。
type GenPageDownloader func() PageDownloader
```

有了这两个参数的值，我们就可以创建并初始化一个实体池的实例了。这需要调用mdw.NewPool并传入必要的参数值。

函数mdw.NewPool接受3个参数。首先是代表其总容量的uint32类型值，其次是代表其中实体的类型的reflect.Type类型值，最后是代表了实体生成函数的func() Entity类型值。对于第一个参数的值，我们可以直接使用NewPageDownloaderPool函数接受的第一个参数值来赋予。而对于后两个参数的值，我们就需要依据NewPageDownloaderPool函数接受的第二个参数值转换得出了。

更确切地说，我们会使用实验法得出mdw.NewPool函数需要的第二个参数值。请看下面的代码：

```
etype := reflect.TypeOf(gen())
```

我们先调用了一次网页下载器生成函数gen，并使用reflect.TypeOf函数来获取前者返回的结果值。这里存在一条约定俗成的规则：无论gen函数被调用多少次，它返回的结果值的实际类型都应该是一致的。因此，我们通过这样的方式就可以确定传给mdw.NewPool函数的第二个参数值了。

函数NewPageDownloaderPool的第二个参数是GenPageDownloader类型的，而mdw.NewPool函数的第三个参数的类型字面量却是func() Entity。显然，它们是不同的。我们不能直接使用前者的值为后者赋值。不过，我们却可以通过对前者的简单封装而得到后者：

```
genEntity := func() mdw.Entity {
    return gen()
}
```

这完全得益于Go语言视函数类型为一等类型的特性。

注意，由于PageDownloader接口类型包含了mdw.Entity接口类型中声明的所有方法，所以类型为前者的值也可以被视为后者的值。这符合Go语言的编译规则。

经过上述准备之后，我们就可以调用mdw.NewPool函数了：

```
pool, err := mdw.NewPool(total, etype, genEntity)
```

如果这里的变量err的值不为nil，那么就说明mdw.NewPool无法根据给定的参数值初始化一个实体池。我们应该及时把它传递给该方法的调用方：

```
if err != nil {
    return nil, err
}
```

如果变量err的值为nil，那么就意味着初始化myDownloaderPool类型值的充分条件已经具备。这时，我们仅使用之前得到的那些值就可以轻松地完成任务：

```
dlpool := &myDownloaderPool{pool: pool, etype: etype}
```

由于我们希望把*myDownloaderPool类型作为PageDownloaderPool接口类型的实现类型，所以这里应该直接把dlpool变量的值作为NewPageDownloaderPool函数的第一个结果值返回。为了让这条返回语句能够顺利地通过编译，我们就需要为myDownloaderPool类型编写PageDownloaderPool接口类型中声明的所有方法。并且，这些方法都应该是指针方法。

首先，我们来编写Take方法。该方法的声明如下：

```
func (dlpool *myDownloaderPool) Take() (PageDownloader, error)
```

在它的方法体中，我们首先需要调用dlpool的字段pool的Take方法。这是返回一个网页下载器的第一步。相关代码如下：

```
entity, err := dlpool.pool.Take()
```

在为entity和err这两个变量赋值之后，我们会进行两项检查。第一项检查就是判断err变量的值是否为nil。与NewPageDownloaderPool函数中的做法相同，若此值不为nil就要及时上报。我们在讲解实体池的实现类型的时候，说明过在怎样的情况下它的Take方法的第二个结果值会是非nil的。虽然就目前对它的使用方式来看，这种情况是不可能出现的，但是我们在这里的检查仍然是有必要的。这会为今后的代码变更添加一层保障。

如果实体池pool的Take方法没有返回非nil的错误值，那么我们就要紧接着进行第二项检查。与上一项检查相同，如果实体池工作正常，那么该项检查必定会通过。否则，我们就必须忽略余下的所有操作并返回错误值。第二项检查的代码如下：

```
dl, ok := entity.(PageDownloader)
if !ok {
    errMsg := fmt.Sprintf("The type of entity is NOT %s!\n", dlpool.etype)
    panic(errors.New(errMsg))
}
```

可以看到，此项检查针对的是从实体池中获取的实体的类型。在初始化实体池的时候，我们给予它了一个通过实验得出的代表了实体类型的值，并且还该值赋给了网页下载器池的etype字段。实体池会保证它的Take方法总是按照我们的意愿返回类型与之相应的值。而在作为又一层封装的网页下载器池中，我们只需检查它的Take方法返回的结果值的类型否是与接口类型PageDownloader相匹配。显然，这一点已由NewPageDownloaderPool函数对其参数的类型约束保证了。也正因为如此，一旦发现类型不匹配的情况，就会立即引发一个运行时恐慌。

最后，若没有发生任何异常情况，我们就可以把最终得到的那个PageDownloader类型值（由变量dl代表）返回给方法的调用方了。别忘了还要把nil作为该方法的第二个结果值。

与Take方法相比，myDownloaderPool类型的其他方法实现起来就极其简单了。我们仅仅调用

其内部的实体池的相应方法并返回其返回的结果即可。这些方法的实现如下：

```
func (dlpool *myDownloaderPool) Return(dl PageDownloader) error {
    return dlpool.pool.Return(dl)
}

func (dlpool *myDownloaderPool) Total() uint32 {
    return dlpool.pool.Total()
}
func (dlpool *myDownloaderPool) Used() uint32 {
    return dlpool.pool.Used()
}
```

到这里，网页下载器池的实现已经编写完毕了。我们使用go build命令编译代码包webcrawler/downloader以确保上述代码在语法上正确无误。

网页下载器为网络爬虫框架提供了可以从几乎任何公开的、支持HTTP协议的目标服务器上下载资源的能力。并且，由于有了网页下载器池的支持，我们可以在资源消耗可控的前提下并发地进行上述操作。请注意，这里使用了“几乎”这个词。这是有原因的。有些服务器只为通过其认证或授权的客户端提供服务。这可能需要终端用户提供用户名和密码，或者需要客户端在HTTP请求的头部中加入一些特殊的信息，又或者需要我们使用另外的一套协议与之交互。这些先决条件有的可以通过对网络爬虫框架的定制来满足，而有的则需要专门的流程设计和实现。由于本书主题和篇幅的原因，我们暂时不考虑后者而只关注前者。在后面的讲解中，读者会了解到采用怎样的定制方式才能够使用网络爬虫框架下载到需要认证或授权的数据。比如，下一小节将要讲到的分析器就为此给使用方预留了足够的定制空间。

9.6.2 分析器

本小节会讲解分析器和分析器池。分析器为我们分析响应提供流程上的支持，而分析器池则为这一流程的并行化提供了支持。它们使用到的辅助工具与网页下载器和网页下载器池所用的并没有差别。不过，与后者不同的是，它们会为使用者提供非常大的定制空间。

接口类型Analyzer的Analyze方法接受两个参数。一个代表了作为分析目标的响应，而另一个则代表了分析方法的列表。也就是说，分析需要的所有东西都会在Analyze方法被调用的时候传递给它。因此，我们可以让分析器的基本结构非常简单。请看下面的代码：

```
// 分析器的实现类型。
type myAnalyzer struct {
    id uint32 // ID。
}
```

可以看到，分析器的实现类型myAnalyzer的声明中只包含了一个字段id。它被用来唯一地标识分析器。这与网页下载器中的id字段的用途是相同的。另外，为这个id赋值的时候同样会用到ID生成器，也同样会新建一个函数将当前实现与ID生成器隔离开，这个函数被命名为genAnalyzerId。

分析器实现类型的基本结构很简单，因此用来创建它的新Analyzer函数的实现非常简洁。它的函数体只需包含一个复合字面量和一条return语句。如下：

```
// 创建分析器。
func NewAnalyzer() Analyzer {
    return &myAnalyzer{id: genAnalyzerId()}
}
```

显然，若要使*myAnalyzer类型成为Analyzer接口类型的实现类型，就必须为它实现指针方法Id和Analyze。关于Id方法的实现，我们自不必多说。而Analyze方法的实现就要复杂得多了。

首先，该方法作为myAnalyzer类型的指针方法的声明是这样的：

```
func (analyzer *myAnalyzer) Analyze(
    respParsers []ParseResponse,
    resp base.Response) (dataList []base.Data, errorList []error)
```

我们已经知道，对响应resp的分析以及数据列表和错误列表的产出的工作完全是由参数respParsers中的若干个响应解析函数来完成的。因此，Analyze的主要任务就是把响应依次传递给这些响应解析函数并获取结果，然后再把这些结果汇总并返回给调用它的一方。下面我们就依据这个思路来编写Analyze函数的实现。

作为一个函数，首先应该检查每个参数值的有效性。对于参数respParsers来说，我们直接判断它是否为nil就可以：

```
if respParsers == nil {
    err := errors.New("The response parser list is invalid!")
    return nil, []error{err}
}
```

而参数resp是结构体类型的，因此即使是其零值也不会为nil。我们应该把检查的重点放在它的httpResp字段上。还记得吗？我们可以通过调用它的HttpResp方法来获取该字段的值。具体如下：

```
httpResp := resp.HttpResp()
if httpResp == nil {
    err := errors.New("The http response is invalid!")
    return nil, []error{err}
}
```

在这之后，我们可以记录一些日志以表示将要处理一个有效的HTTP响应：

```
var reqUrl *url.URL = httpResp.Request.URL
logger.Infof("Parse the response (reqUrl=%s)... \n", reqUrl)
```

其中的标识符logger代表了一个logging.Logger类型的值。我们在之前多次提到过logging代码包以及其中的Logger。后者是一个定义了日志记录器行为的接口类型。它们同样存在于goc2p项目之中。我们在后面还会多次提到它们。值得一提的是，为了能够对在网络爬虫框架中使用的日志记录器进行统一配置，我们在webcrawler/base代码包中专门编写了一个被用来获取日志记录器的函数：

```
// 创建日志记录器。
func NewLogger() logging.Logger {
    return logging.NewSimpleLogger()
}
```

在网络爬虫框架的其他代码包中，如果要声明一个代表了日志记录器的变量并为它赋值，那么它们就可以这样编写：

```
// 日志记录器。
var logger logging.Logger = base.NewLogger()
```

当然，前提是当前的源码文件已经导入了webcrawler/base代码包。

这样做有一个明显的好处，那就是：当我们要改变日志记录器的创建和初始化方式的时候，仅需修改base.NewLogger函数的实现即可。这有效地避免了散弹式的修改。请想象一下，如果我们在每个需要用到日志记录器的地方都这样为logger变量赋值：

```
var logger logging.Logger = logging.NewSimpleLogger()
```

那么情况将会怎样？

言归正传。在myAnalyzer类型的指针方法Analyze中，我们调用了logger的Infof方法记录了一条普通的信息。这条信息包含了这段代码将要执行的操作以及与该HTTP响应对应的URL（URL可以让我们很方便地区分响应）。这很重要。因为这样我们就可以从日志中看到哪些时候分析器处理了哪些HTTP响应了。

接下来，我们应该取出参数resp的另外一个字段的值以备后：

```
respDepth := resp.Depth()
```

变量respDepth的值即指该响应包含的网页深度。从该网页的内容中分析出的网络地址及其可能对应的网页的深度应该与前者有递增关系。

现在开始准备通过参数respParsers的值分析HTTP响应。先要建立两个分别可以收集数据（请求或条目）和错误的容器：

```
// 解析HTTP响应。
dataList = make([]base.Data, 0)
errorList = make([]error, 0)
```

然后，使用respParsers参数值中的响应解析函数对httpResp变量的值进行分析。大致的代码如下：

```
for i, respParser := range respParsers {
    // 省略若干条语句
    pDataList, pErrorList := respParser(httpResp, respDepth)
    // 省略若干条语句
}
```

可以看到，我们在for语句依次拿到respParsers中的每个响应解析函数，然后分别调用它们并获取结果。不过，在这之前，我们还有一些工作要做。

还记得吗？Go语言的函数类型的值可以为nil。为了保证调用成功，我们应该首先判断当前的响应解析函数是否为nil。如果答案是肯定的，那么我们就应该立即生成一个可以说明此问题的错误值，然后忽略余下的操作并去获取下一个响应解析函数。相关代码如下：

```
if respParser == nil {
    err := errors.New(fmt.Sprintf("The document parser [%d] is invalid!", i))
```

```

        errorList = append(errorList, err)
        continue
    }

```

我们在错误的描述中加入了这个为nil的响应解析函数在respParsers中的索引。希望这样能够帮助Analyze方法的调用方快速定位问题所在。之后，我们把该错误值追加到先前声明的切片值（由errorList变量代表）中，以使它可以被传递到方法之外。最后，我们使用continue语句让包含它的那条for直接进行下一个迭代。

一旦当前的响应解析函数通过检查，我们就可以放心地把HTTP响应httpResp交给它处理了。待其分析完成之后，我们就会得到两个切片值。它们分别由变量pDataList和pErrorList代表，并分别表示了从该HTTP响应中分析出的数据的列表以及在这个过程中发生的错误的列表。

理所应当，我们应该把每个响应解析函数返回的这样的两个列表都分别追加到dataList和errorList中去。注意，对于这两个追加操作，我们有着不同的要求。

我们把执行追加操作的代码封装在一个名为appendDataList函数中，并在针对pDataList变量的for语句中调用它：

```

if pDataList != nil {
    for _, pData := range pDataList {
        dataList = appendDataList(dataList, pData, respDepth)
    }
}

```

函数appendDataList的声明如下：

```

// 添加请求值或条目值到列表。
func appendDataList(dataList []base.Data, data base.Data, respDepth uint32)
[]base.Data

```

它接受3个参数。第一个参数的值必须是会被Analyze方法返回的数据容器，第二个参数的值应该是pDataList中的某一个数据项，而第三个参数的值则应当是当前被分析的那个响应的深度值。此外，该函数还会返回一个[]base.Data类型的结果。该结果的值即是被追加数据之后的数据容器。

函数appendDataList使用前两个参数值的方式非常明显。不过在这之中，我们还要进行一些特殊的处理。这涉及新的请求的深度。

按照惯例，第一步操作是检查参数data的值的有效性。因为该值是从响应解析函数中返回的。它来自网络爬虫框架之外。我们总是应该先检查这样的值的有效性。如果该值为nil，我们就应该忽略追加操作，并直接把原来的数据容器dataList返回给appendDataList函数的调用方。代码如下：

```

if data == nil {
    return dataList
}

```

还记得吗？起到类似作用的语句有个学名，叫卫述语句。

如果data的值不为nil，我们就继续进行后面的操作。我们检查data代表的新请求中的深度

值。不过，前提是data代表的是请求而不是条目。这就需要我们先进行判断。这会用到类型断言表达式：

```
req, ok := data.(*base.Request)
```

我们试图用类型断言表达式把data的值转换为*base.Request类型的值。由于*base.Request类型代表了请求，并且是base.Data接口类型的实现类型，所以当该值为一个请求的时候，这一类型转换就会成功。成功与否会由ok变量的值来体现。如果ok的值为false，那么就说明类型转换不成功，该数据并不是一个请求。既然该数据不是一个请求，那一定就是一个条目。若为这种情况，我们无需进行任何特殊处理，而直接把该数据追加到数据容器中并返回追加操作的结果即可：

```
if !ok {
    return append(dataList, data)
}
```

如果ok为true，那么变量req就会是一个有效的*base.Request类型值。此时，我们就要通过调用它的Depth方法来检查它的深度值是否是正确的。如果不正确，我们就应该重新创建并初始化一个请求来替代它。请看下面的代码：

```
newDepth := respDepth + 1
if req.Depth() != newDepth {
    req = base.NewRequest(req.HttpReq(), newDepth)
}
```

其中，变量newDepth的值代表了新请求应该具有的深度值。

我们之所以替换这个请求而不是直接改变它，是因为*base.Request类型没有给我们提供这样的方法。并且，我们不应该为此而放宽这种限制。实际上，我们对这些网络爬虫框架的基本数据类型都采用了这样的设计。这样可以让它们的值具有不可变性。如此一来，当我们把一个这样的值传递给一个未处于webcrawler/base代码包中的函数或方法的时候，就不用担心该值会被改变了。这样既可以消除很多方面的顾虑，又可以简化相关代码。

不过，这样也带来一个问题，就是当我们想修改这样的值的时候不得不先对它进行完整的复制。就像我们在前面示例中做的那样。如果存在非常多的类似需求，那么我们就应该仔细考虑是否依然保留这种不可变性。因为如此复制一般会消耗大量的内存，并且给运行时系统的垃圾回收器造成很大的压力。

考虑到这里只是在发现响应解析函数返回的请求的深度不正确的时候才会复制它，我们无需改变现有的设计。

经过这样的检查和必要的修正之后，我们就可以把新请求追加到数据容器并返回它了：

```
return append(dataList, req)
```

另一方面，我们对于每个响应解析函数返回的代表若干错误的切片值pErrorList也需要做类型的处理。区别是，我们不需要对其中的错误值做任何检查。相似地，我们把追加的操作也封装在了一个函数中：

```
// 添加错误值到列表。
func appendErrorList(errorList []error, err error) []error {
    if err == nil {
        return errorList
    }
    return append(errorList, err)
}
```

而在Analyze函数中的那条for代码块的最后，我们这样来调用这个函数：

```
if pErrorList != nil {
    for _, pError := range pErrorList {
        errorList = appendErrorList(errorList, pError)
    }
}
```

以上就是被用来真正处理一个响应的for代码块中的全部代码。一旦这条for语句被执行完成，就意味着对当前的响应的分析处理已经结束。这时，我们就可以将数据容器和错误容器返回给Analyze方法的调用方了：

```
return dataList, errorList
```

至此，我们已经编写完成了Analyze方法和myAnalyzer类型。编译一下，以保证*myAnalyzer类型真正实现了Analyzer接口类型（如果NewAnalyzer函数通过了编译就可以证明）以及不存在其他语法错误。

我们通过以响应解析函数（由ParseResponse类型的值代表）作为参数的方式来接纳使用方对分析器的定制。同时，经过思考的接口设计也为这些定制的行为和网络爬虫框架之间的融合提供了很好的支持。实际上，对响应进行分析的工作是非常复杂的。正因为如此，网络爬虫框架不应该介入到这些复杂的逻辑当中，而应该专心为它们提供良好的执行环境。使用方提供的若干个响应解析函数会被网络爬虫框架中的所有分析器使用。与网页下载器类似，这些分析器会由一个分析器池容纳和管理。

有了编写网页下载器池的经验，我们再来实现分析器池会非常简单。请读者回顾9.2节中的与池的接口设计相关的内容。可以看到，这两个池的接口类型声明极其相似。这也是我们在前面编写实体池并以此来为这两者的实现提供底层支持的原因。

现在是练手的好机会。请读者模仿网页下载器池的实现方式自己编写一个名为myAnalyzerPool的类型，并使该类型成为AnalyzerPool接口类型的实现类型。请记住，虽然myAnalyzerPool类型实现起来会与myDownloaderPool类型非常相近，但是我们应该把它们独立开来，包括像ID获取函数这类的辅助代码。这样，我们今后就可以很方便地单独改变和演进它们了。

网页下载器和分析器会更多地使用不同类型的系统资源。前者主要利用网络I/O完成功能，而后者专注于内部计算。这种需要上的不同会使得它们慢慢向着不同的方向进化。因此，从一开始就在抽出并共享池的本质行为（指的是实体池）的前提下分离这两个应用层面的池的实现是很有好处的。

请读者真正地去实现一个分析器池，不要偷懒。这并不困难。请在写完那些代码并通过编译之后再回到这里继续往下读。

好了，在实现了网页下载器和分析器以及容纳它们的池以后，我们编写网络爬虫框架的处理模块的任务已经完成了大半。最后这个模块与前两模块有很多不同之处。它不需要用池来管理，并且只需一个实例就足够提供必要的功能。它的实例是几乎不可被改变的。另一方面，它提供定制的方式与分析器有一些类似。

9.6.3 条目处理管道

条目处理管道会以流的方式来处理我们发送给它的每一个条目。每一个处理步骤都由一个条目处理器来代表。每个条目处理器都会接受一个条目并返回一个已经经过处理的条目和一个错误值。这些条目处理器的行为是由webcrawler/itempipeline代码包中的ProcessItem函数类型来定义。

根据ProcessItem函数类型以及同一个代码包中的接口类型ItemPipeline对条目处理管道的行为的定义，我们可以得出以下声明：

```
// 条目处理管道的实现类型。
type myItemPipeline struct {
    itemProcessors []ProcessItem // 条目处理器的列表。
    failFast        bool           // 表示处理是否需要快速失败的标志位。
    sent            uint64        // 已被发送的条目的数量。
    accepted        uint64        // 已被接受的条目的数量。
    processed       uint64        // 已被处理的条目的数量。
    processingNumber uint64        // 正在被处理的条目的数量。
}
```

很明显，myItemPipeline类型中的字段itemProcessors代表了条目处理管道将会持有的若干个条目处理器，字段failFast的值会体现出条目处理管道的“快速失败”特性，而后面的4个字段都是为了满足统计的需要而建立的。

从网络爬虫框架的角度看，条目处理管道只会接受输入，而不会产生任何输出。这是由于它接受的每一个条目都已经可以被看作是某一个请求的最终产出的一部分或全部了。所以，那些经过条目处理器处理的条目，对网络爬虫框架本身已经没有任何意义了。它们是否需要被以某种形式输出到别处，完全由使用方提供的那些条目处理器来决定。条目处理管道不关心也无需关心这些额外的输出情况。然而，它却应该时刻关注整个条目处理流程的运作情况，比如调度器向它发送了多少个条目？在这些被发送的条目中有多少条目是有效的？有多少条目经过了所有的处理？以及在某一个时刻有多少条目正在被处理？只要知道了这些数字，我们就可以比较完整地勾勒出条目处理管道的实时运行情况了。这也是myItemPipeline类型的基本结构中的最后那4个字段的存​​在意义。

这4个字段是无需被初始化的。因为它们的零值都是0。这正好符合我们的要求。failFast字段也是如此。原因是我们可以以后通过调用SetFailFast方法改变它的值。因此，唯一需要使用方初始化的就是条目处理器的列表了。在myItemPipeline类型中，它们由itemProcessors字段代表。

我们现在编写一个名为NewItemPipeline的函数，并让它接受一个[]ProcessItem类型的参数

以及返回一个ItemPipeline类型的结果。它的声明如下：

```
func NewItemPipeline(itemProcessors []ProcessItem) ItemPipeline
```

在这个函数的函数体中，我们首先要检查itemProcessors参数的值。由于我们之后不会再去关心条目处理的结果，所以这里对条目处理器列表的检查要更加严格一些。请看下面的语句：

```
if itemProcessors == nil {
    panic(errors.New(fmt.Sprintf("Invalid item processor list!")))
}
innerItemProcessors := make([]ProcessItem, 0)
for i, ip := range itemProcessors {
    if ip == nil {
        panic(errors.New(fmt.Sprintf("Invalid item processor[%d]!\n", i)))
    }
    innerItemProcessors = append(innerItemProcessors, ip)
}
```

一旦发现itemProcessors字段的值为nil，就会立即引发一个运行时恐慌。不但如此，当发现列表中的某一个条目处理器为nil时，也会及时以引发运行时恐慌的方式上报。如果以上检查都通过了，我们就会得到一个与itemProcessors参数同值的变量innerItemProcessors。之所以要完全复制一份条目处理器列表，是因为我们不希望在用它来初始化条目处理管道之后使用者仍然有机会改变它。还记得吗？一个切片值持有的是一个值的引用（它会与一个底层数组相对应）。同时，它也是一类值的容器。所以，在把它作为参数值传递给一个函数或方法之后，我们对其中的某个或某些值的变更，一定也会影响到已被传入到函数或方法中的那个值。对于同样属于引用类型的字典值来说也是如此，而对于数组值来说却恰恰与之相悖。

综上所述，我们使用使用者提供的条目处理器列表的复制品来初始化条目处理管道是安全的。因为这样做使得真正起作用的那个值与外界隔离了。最后，初始化代码非常简单，只有一行：

```
return &myItemPipeline{itemProcessors: innerItemProcessors}
```

以上说明的就是NewItemPipeline函数的实现代码的全部。显然，我们要把*myItemPipeline类型实现为ItemPipeline接口类型的实现类型。这需要我们依据这个接口类型的声明为myItemPipeline类型编写6个对应的指针方法。我们首先来看最为复杂的Send方法的实现。其声明如下：

```
func (ip *myItemPipeline) Send(item base.Item) []error
```

对myItemPipeline类型来说，这个指针方法Send应该具有下面4个功能。

- ❑ 检查item的值的有效性。忽略对无效条目的处理。
- ❑ 依次调用itemProcessors字段中的条目处理器对有效的条目进行处理，并依据failFast字段的值控制处理流程。
- ❑ 收集处理过程中发生的错误，并将相应的值作为结果值返回。
- ❑ 在整个处理的过程中，适时的对字段sent、accepted、processed和processingNumber的值进行设置，以满足记录和统计的需要。

其中，第一、二个功能点表示了Send方法需要执行的流程，而实现第三、四个功能点的代码则会夹杂在其中。

首先，我们来看第一个功能点。还记得吗？`base.Item`类型是一个字典类型的别名类型。因此，它的值可能为`nil`。加之我们对条目中应该包含哪些键值对并无要求，所以这里仅对`item`的值进行一项检查即可。显然，如果条目为`nil`，那么也就没必要对它进行处理了。相关代码如下：

```
errs := make([]error, 0)
if item == nil {
    errs = append(errs, errors.New("The item is invalid!"))
    return errs
}
```

请注意，我们在一开始就要留心收集错误值。这就是声明`errs`变量的原因。实际上，在Send方法执行流程的过程中，只要发生错误，我们就会把对应的错误值追加到`errs`的值中。

接下来是第二个功能点。读者可能已经猜到，实现第二个功能点的代码的主体是一条for语句。它的迭代对象就是`itemProcessors`字段的值。我们需要利用这个for代码块完成对条目的流式处理。已知每个条目处理器都会接受一个条目，也都会返回一个已经经过处理的条目和一个错误值。由此，在发现某一个条目处理器返回了一个非`nil`的错误值的时候，我们应该根据`failFast`值来决定是否中断对该条目的处理。相关代码如下：

```
var currentItem base.Item = item
for _, itemProcessor := range ip.itemProcessors {
    processedItem, err := itemProcessor(currentItem)
    if err != nil {
        errs = append(errs, err)
        if ip.failFast {
            break
        }
    }
    if processedItem != nil {
        currentItem = processedItem
    }
}
```

请注意，其中的变量`currentItem`起着很关键的作用。我们就是利用它让条目依次流经各个条目处理器的。此外，在这个过程中也存在着类似的收集错误值的代码。很明显，在该方法体的最后，我们是要将变量`errs`的值作为结果值返回的。

再来说第四个功能点。读者感觉我们应该在上述代码的哪些位置上插入对`myItemPipeline`类型的最后那4个字段的值的设置代码呢？最明显的当属设置`sent`字段的值的位置了。它应该出现在该方法体的最前面。不过，有一个设置会比它更靠前。这就是对`processingNumber`字段的值的设置。`processingNumber`字段的值代表了正在被条目处理管道处理的条目的数量。所以，Send方法一经调用就应该首先递增它的值。注意，由于Send方法一定会被并发地调用，所以我们必须要使用同步手段来保护这类操作。这其中还包含了对`processingNumber`字段的值的递减操作。由于该操作需要在Send方法即将被执行结束的时候进行，所以我们需要把该操作放在`defer`语句中。总之，我们应该在Send方法的方法体的开始处加入下面这几行代码：

```
atomic.AddUint64(&ip.processingNumber, 1)
defer atomic.AddUint64(&ip.processingNumber, ^uint64(0))
atomic.AddUint64(&ip.sent, 1)
```

可以看到，我们通过调用sync/atomic包中的相应方法来保证递增或递减操作的并发安全。另外，请注意，我们递减uint64类型的字段processingNumber的值的方式。在本书中，这种特殊用法首次出现在8.3节。

现在来看设置accepted字段的值的时机。让Send方法接受并准备处理一个条目的前提是该条目是有效的。因此，这行代码：

```
atomic.AddUint64(&ip.accepted, 1)
```

应该紧跟在对条目进行有效性检查的代码（那条if语句）之后。

最后是对processed字段的值的设置。已被处理的含义是，条目已被所有条目处理器处理过，不论在这个过程中是否有错误发生。注意，如果failFast字段的值为true，那么条目已被处理也意味着在处理过程中未发生任何错误。否则，这二者之间就不存在必然的联系。总之，我们应该把代表该递增操作的代码

```
atomic.AddUint64(&ip.processed, 1)
```

置于前面那条for语句的后面。

好了，以上就是Send方法包含的所有代码。读者应该可以以正确的顺序把它们排列好。下面我们继续讲剩下的5个方法的实现方法。

针对于failFast字段的FailFast方法和SetFailFast方法的实现极其简单。我们在此略过。

方法Count需要返回3个计数值。就myItemPipeline类型而言，它们即是字段sent、accepted和processed的值。这看起来也非常简单。不过有一点需要注意，就是我们一定要使用原子操作来读取它们的值：

```
counts := make([]uint64, 3)
counts[0] = atomic.LoadUint64(&ip.sent)
counts[1] = atomic.LoadUint64(&ip.accepted)
counts[2] = atomic.LoadUint64(&ip.processed)
```

这也同样适用于ProcessingNumber方法的实现。其中的读取processingNumber字段的值的方式应该是完全一致的。

最后是Summary方法。Summary方法返回的值应该反映当前的条目处理管道的一般状态和实时状态。这涉及了myItemPipeline类型中的所有字段。请看我们为此定义的摘要信息的模板：

```
var summaryTemplate = "failFast: %v, processorNumber: %d," +
    " sent: %d, accepted: %d, processed: %d, processingNumber: %d"
```

下面我们来为它填空。需要准备的值共有6个。第一个值很容易得到，直接使用failFast字段的值即可。后面的“processorNumber”指的是条目处理器的数量。所以我们需要使用itemProcessors字段的长度值来填充第二个空当。至于后面的4个空当，我们同样可以直接用字段sent、accepted、processed和processingNumber的值来填充。不过，我们在这里同样要使用原子操作。鉴于我们已经实现了Count方法和ProcessingNumber方法，所以可以不必再去调用

sync/atomic包中的函数。到这里，我们就可以编写出Summary方法的完整声明了：

```
func (ip *myItemPipeline) Summary() string {
    counts := ip.Count()
    summary := fmt.Sprintf(summaryTemplate,
        ip.failFast, len(ip.itemProcessors),
        counts[0], counts[1], counts[2], ip.ProcessingNumber())
    return summary
}
```

以上就是条目处理管道的全部实现代码。最后，我们需要使用go build命令对webcrawler/itempipeline代码包进行编译，以保证代码的合法性并确保*myItemPipeline类型已经实现了接口类型ItemPipeline。

到这里，我们已经讲述了网络爬虫框架中的全部处理模块的接口设计和实现代码。这些处理模块都做到了职责上的单一和清晰。这也是我们设计的方向。它们都专注于做好某一个方面的事情，而毫不关心爬取流程中的其他部分。我们依靠调度器把这些处理模块串联起来，并使整个流程能够被真正地运转起来。

我们在下一节就会专门讲解调度器的实现。读者会从中了解到，调度器不只是使用通道管理器在各个处理模块之间搬运数据那么简单。在它的实现中还包含了很多因地制宜的设计技巧。下面，就让我们进入到下一节。

9.7 调度器的实现

调度器的主要职责是对各个处理模块进行调度，以使它们能够进行良好的协作并共同完成整个爬取流程。从这个职责概述以及调度器的接口声明来看，调度器需要拥有一些字段来支撑或辅助相关功能的实施。有了这些字段，它的各个方法就可以利用所需字段的值来完成自己的功能。

在说明这些字段的声明和方法的实现之前，我们先来看一下其中对网络爬虫框架中的其他代码包的导入方式。调度器本身既会使用到各种中间件工具，也会操纵各个处理模块。当然，网络爬虫框架的基本数据类型肯定也会被用到。也就是说，我们前面讲解的代码所处的代码包都需要被导入进来。由于这些代码包的名称大都不短，所以我们在导入的时候为它们起了别名。下面的代码片段展示了它们的名称和别名之间的对应关系：

```
anlz "webcrawler/analyzer"
base "webcrawler/base"
dl "webcrawler/downloader"
ipl "webcrawler/itempipeline"
mdw "webcrawler/middleware"
```

在调度器的接口声明和实现代码所在的代码包（即webcrawler/scheduler）中，这些被导入的代码包的别名都是统一的。我们在后面会多次提及这些别名。读者应该可以由这些别名联想到与之对应的代码包。

好了，下面就让我们开始编写调度器的实现类型。

注意 本节描述的部分实现会在9.8节中被修改。因此，它们会与随书项目中的相应代码不一致。

9.7.1 基本结构

正如接口类型Scheduler中的Start方法的声明

```
Start(channellen uint,
      poolSize uint32,
      crawlDepth uint32,
      httpClientGenerator GenHttpClient,
      respParsers []anlz.ParseResponse,
      itemProcessors []ipl.ProcessItem,
      firstHttpReq *http.Request) (err error)
```

所示，我们在每次开启调度器的时候都会传给它很多参数值。按照用途来划分，这些参数可以分为3类。第一类参数的值被用来初始化其他类型（如通道管理器、网页下载器池等）的实例。除非有统计的需要，调度器一般不需要存储这类参数的值。第二类参数的值用于校验请求的有效性。由于这类参数会在不同的方法中被用到，所以调度器需要使用相应的字段来留存其值。第三类参数指的就是可以激活整个爬取流程的首次请求。它仅仅为调度器以及爬取流程提供一个起始点，所以调度器并不用记住它。不过，首次请求间接地为校验后续的请求提供了一个依据。这个依据就是目标网络站点的主域名。它一般是指代表了目标网络站点的域名中的主域名部分。不过，它还有可能是一个具体的IP地址。这要看首次请求中包含的远程主机的具体地址是一个域名还是一个IP地址。不论怎样，调度器需要记住它以备后用。

根据以上分析，我们为调度器的实现类型添加的首批字段如下：

```
poolSize      uint32           // 池的尺寸。
channellen    uint             // 通道的总长度（也即容量）。
crawlDepth    uint32           // 爬取的最大深度。首次请求的深度为0。
primaryDomain string           // 主域名。
```

其中，字段poolSize和channellen与第一类参数相对应。它们会分别参与到各类池和通道管理器的初始化过程中去，并且还会是调度器的摘要信息的一部分。字段crawlDepth与第二类参数相对应。调度器会依此过滤掉深度过大的请求。至于primaryDomain字段，会通过解析Start方法的第三类参数firstHttpReq的值而得出。同时，它也是调度器对请求的限制条件之一。

调度器在驱使和协调各个处理模块共同执行爬取流程的过程中，会使用到我们在前面一起设计实现的一些中间件工具，这包括通道管理器和停止信号。当然，实体池也会间接地用到。因为网页下载器池和分析器池都是基于它实现的。而且，调度器会持有这两类池而非池中元素的值（若干网页下载器或分析器的实例）以达到有效节约资源和提高调度效率的目的。据此，我们可以确定调度器实现类型的第二批字段：

```
chanman       mdw.ChannelManager // 通道管理器。
stopSign      mdw.StopSign       // 停止信号。
dlpool        dl.PageDownloaderPool // 网页下载器池。
```

```
analyzerPool  anlz.AnalyzerPool    // 分析器池。
itemPipeline  ipl.ItemPipeline     // 条目处理管道。
```

其中，字段chanman和stopSign分别代表了调度器会用到那两个中间件工具，而后面3个字段则分别代表了网络爬虫框架的3个处理模块。

好了，能够根据调度器接口的Start方法的众多参数分析得出的字段就是这些。调度器的绝大部分方法（如Stop、ErrorChan、Idle和Summary）也都会用到这些字段的值，除了Running方法。Running方法的结果值应该确切地表示出调度器的运行状态。这个状态是不能够从上述那些字段的值上分析或计算出来的。因此，我们需要一个专门的字段来表示这个状态。这就是running字段。其声明如下：

```
running      uint32                // 运行标记。0表示未运行，1表示已运行，2表示已停止。
```

读者可能会问：Running方法的结果的类型是bool类型的，但是running字段的类型却是uint32类型的，这是为什么？

其最主要的目的便于在running字段的值之上实施原子操作。因为我们不能排除调度器的一些方法被并发的调用的可能性。如果多个程序同时调用同一个未被开启的调度器的Start方法，那么该调度器应该只因第一次调用而开启且不理睬后面那些调用。对于Stop方法，也应有类似的限制。

我们知道，Go语言的sync/atomic代码包中没有针对bool类型值的原子操作函数。所以，我们不得不使用一个数值类型而不是bool类型。不过，这也带来一个好处，即running字段的值域变得更大了。虽然Running方法本身并不需要这么大的值域，但是我们可以就此为调度器内部提供更加精细的运行标记。这对我们今后改进和扩展调度器也是有好处的。

到这里，我们一共为调度器的实现类型声明了10个字段。这看起来已经够用，但实际上并不是这样。

不知道读者是否还记得，我们在9.2节中展示的图9-1。在这张图中，我们描述了首次请求对于爬取流程的激活作用，以及各类数据在几个处理模块之间的流转方式。眼尖的读者可能会发现这张图描述的爬取流程存在一个问题：网页下载器和分析器之间形成了一个闭环。请考虑这样一个场景：分析器经过了对某个响应的分析生成了很多请求。当它想把请求传送给网页下载器的时候，发现请求通道已经满了。这可能是由于网页下载器池中的那几个网页下载器处理请求的速度过慢造成的。这个时候，分析器池中的所有分析器都会因此而被阻塞住。只有等到有网页下载器空闲，调度器才会再次从请求通道中接收一个请求并将其分配给这个空闲的网页下载器。一旦请求通道中出现了空余的位置，某一个分析器对请求通道的一个发送操作就会成功。但是，过不了多久，上面这种卡顿的现象就又会发生。

造成上述情况的原因有两个。第一个原因是，网页下载器处理请求的速度要比分析器处理响应的速度慢很多。这主要是因为前者会通过网络I/O与目标网站进行数据交互，而后者却只会使用到本地计算机的CPU和内存。单看多出的网络数据交互这一项，我们就能够预料到这两者处理数据的速度基本不会在同一个数量级上。这涉及交互双方的计算机和操作系统的性能，以及目标网络站点中的网页提供程序（可能是一个单一的小程序，也可能是一个庞大的程序集群）的计算

速度。但是，更主要的往往是在网络上交互数据所耗费的时间。

第二个原因是，网页下载器与分析器单次产出的数据数量有所不同。前者每次接受一个请求，并且仅会返回一个响应；而后者每次只会接受一个响应，但却常常会返回多个请求。我们知道，一个网络站点上的网页之间大都存在关联，不存在链接的网页很少。网页上的每一个直接或间接的链接都可以转化为一个新的请求。这往往会使网页下载器与分析器之间传递的数据的数量存在不小的差距。换句话说，网页下载器传递给分析器的响应的数量往往会比分析器传递给网页下载器的数量少很多。并且，这一差距会随着时间的推移被逐步扩大。

上面这两个原因中的任何一个都可以导致卡顿现象的发生，更何况是它们共同作用。可以预计，卡顿的现象在首次请求被处理之后不久会发生，并且其发生的频率在不久之后就会达到顶峰。最后，分析器处理响应的速度会向网页下载器处理请求的速度靠拢，从而导致爬取流程的各个环节的效率会比对于HTTP请求和响应的那一发一收的操作的效率还要低。这对于网络爬虫框架来说已经是一个非常大的性能问题了。并且，这也是对计算资源的极大浪费。

我们可以考虑将请求通道的容量置于响应通道的容量的数倍，来推迟卡顿现象初次发生的时间。这样甚至可以在一定程度上降低卡顿现象的发生频率。但是，如此却只能缓解问题，而不能解决问题。在这里，我们采用的方案是多增加一个传递请求的介质——请求缓存，如图9-3所示。

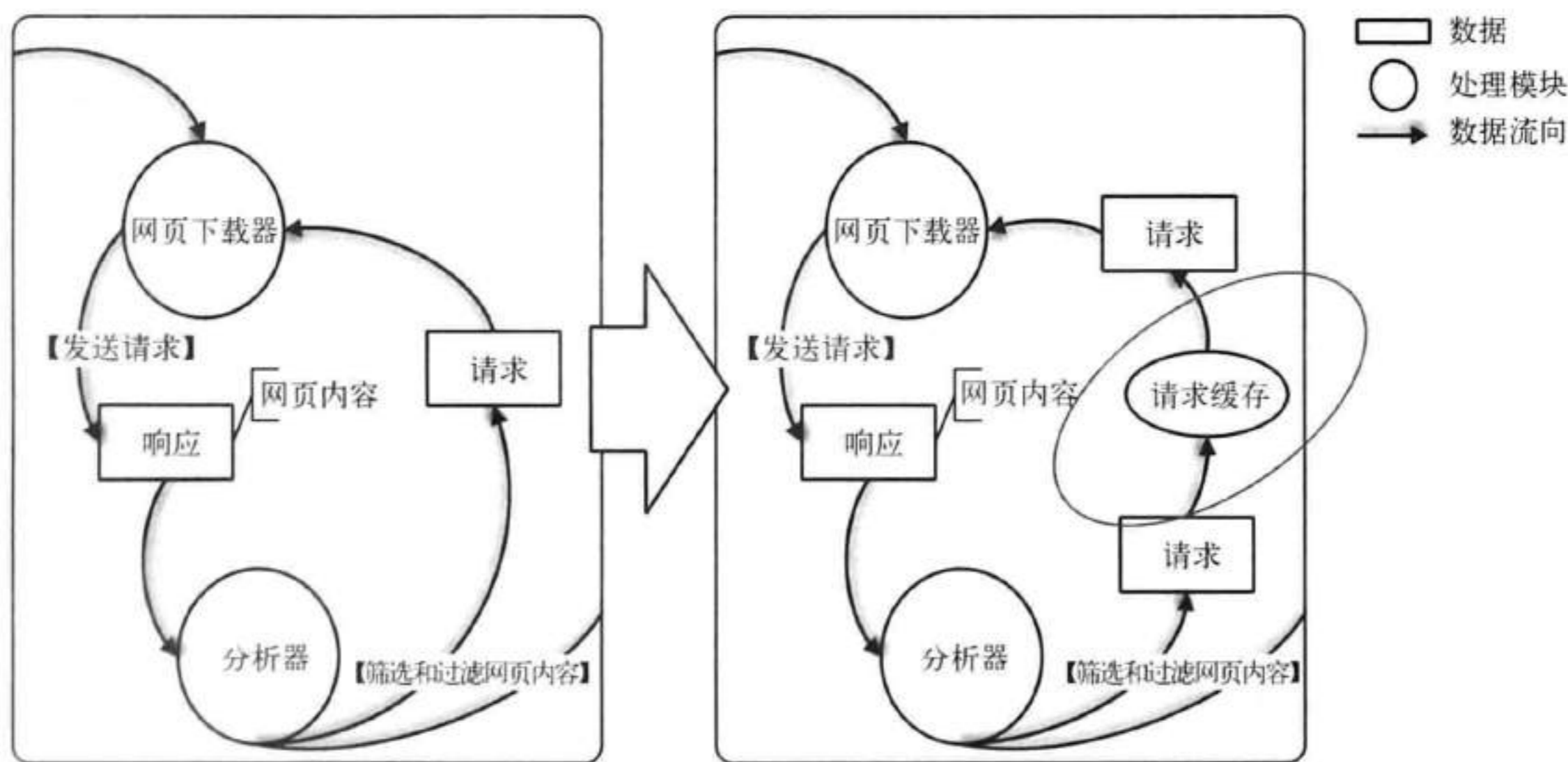


图9-3 对请求传递方式的改进

请注意图中用线圈出的那部分。调度器不会再把分析器返回的响应直接发送给请求通道了，而会把它先存入请求缓存。这样可以防止因请求通道已满而阻碍对后续响应的分析处理。也就是说，不论网页下载器处理请求的速度如何，分析器以及网络爬虫框架中的其他组件的运行效率都不会受到它的影响。另一方面，为了让这些请求能够被及时地处理，调度器会适当地将请求缓存中的请求转移到请求通道中。

我们在下一小节会讲到这里所说的转移的策略和具体实现方式。现在，我们只需要知道这个代表请求缓存的类型被声明在了webcrawler/scheduler代码包中，名称为requestCache。我们也会在后面专门用一个小节来讲解这个类型以及它的具体实现。

既然调度器需要用到请求缓存，那么就需要使用一个字段来表示它。这个字段的声明是这样的：

```
reqCache    requestCache    // 请求缓存。
```

调度器的实现类型的最后一个字段是字典类型的。添加这样一个字段是出于校验请求的有效性的目的。也就是说，它与字段crawlDepth有着类似的作用。我们刚才说过，网络站点上的网页之间大都存在关联。进一步讲，这种关联往往不是单向的，而是网状的。比如，多个网页中可能会存在指向同一个网络地址的链接，还可能会存在指向彼此的网络地址的链接，等等。总之，如果我们把一个网络站点上的所有网页甚至一个规模不太小的网页集合中的链接都抽离出来，就往往会发现其中存在着不少的网络地址重复的链接。显然，我们不应该再对已经处理过的网络链接发送请求并分析其响应。这是完全没有必要的，会白白地浪费系统资源。正因为如此，调度器需要把已经处理过的请求都记录下来，并在发现分析器返回请求的网络地址重复的时候过滤掉它。基于这一需求，就有了下面这个字典类型的字段：

```
urlMap      map[string]bool    // 已请求的URL的字典。
```

以上，就是调度器的实现类型需要的所有字段。我们把调度器实现类型的名称定为myScheduler，并欲使它的指针类型成为调度器接口类型Scheduler的实现类型。

9.7.2 主要的函数和方法

在确定了myScheduler类型的基本结构之后，我们就可以编写调度器的创建函数NewScheduler了。不过，它的函数体非常简单，并没有涉及对那些字段的初始化。因为myScheduler类型的绝大多数字段的值要么是由Start方法的参数值计算得出的，要么只在调度器被开启之后才有必要初始化。而剩下的字段running的零值也已经能够正确地表示出调度器的状态，所以我们并不需要再对它们进行额外的初始化。因此，NewScheduler函数的函数体中只有一行代码，具体如下：

```
// 创建调度器。
func NewScheduler() Scheduler {
    return &myScheduler{}
}
```

下面，我们就开始逐一地为myScheduler类型实现调度器接口类型Scheduler中所声明的那些方法。当然，这些方法都需要被声明为myScheduler类型的指针方法。我们先从最复杂的Start方法开始。

1. 开启调度器

针对于myScheduler类型，Start方法的接收者类型应该是*myScheduler，而代表接收者的标识符被确定为sched。

首先，我们应该为Start方法编写一条defer语句。编写该语句的目的是：即使在使用参数值

初始化调度器的各个字段以及开启调度器的过程中有运行时恐慌被抛出来,调度器也能够及时地恢复它并记录相应的日志。它的声明如下:

```
defer func() {
    if p := recover(); p != nil {
        errMsg := fmt.Sprintf("Fatal Scheduler Error: %s\n", p)
        logger.Fatal(errMsg)
        err = errors.New(errMsg)
    }
}()
```

这属于一种防护措施,可以有效防止使用调度器的程序因从该方法中抛出的运行时恐慌而崩溃。同时,这也让我们在以后调整网络爬虫框架中的一些模块报告错误的方式的时候可以更加灵活一些。

请注意,这段代码中的if代码块中的最后一条语句。在这条语句中,我们把一个新的错误值赋给了变量err。这个变量所指的正是Start方法的签名中的那个命名结果值err。通过这条赋值语句,我们在恢复了本会被抛出的运行时恐慌之后把它转化为了一个可以作为Start方法的结果的错误值。这是以统一、平和的方式报告在Start方法被执行期间所发生的各类异常的有效方式。这是一个很有用的惯用法。

在做好防护措施之后,我们就应该去检查running字段的值。前文提到,当调度器已处于已运行状态的时候,就不应该再去响应后续的对Start方法的调用了。这就需要未被开启的调度器的Start方法被第一次调用的时候要及时修改running字段的值。这里所说的对running字段的值的读和写操作都要尽早进行。这样才能发挥其应有的作用。这也是我们把它放置在defer语句下方的原因。具体代码如下:

```
if atomic.LoadUint32(&sched.running) == 1 {
    return errors.New("The scheduler has been started!\n")
}
atomic.StoreUint32(&sched.running, 1)
```

我们在读写running字段的值的时候都应该像上面那样使用原子操作。注意,正如我们在第8章讲到的那样,即使是如此简单的原子操作,也会对程序性能造成负面影响。我们应该在满足需求的前提下尽量少用或不用。在调度器中,我们只在开启、停止和提供调度器状态的时候才去读写running字段的值。这样的读写操作是必要且至关重要的,并且操作频率已经算是相当低了(不过还要看使用者使用调度器的方式)。

在把running字段的值设置为1之后,我们就需要检查参数值,并根据它们设置相应字段的值了。这是真正开启调度器的前提。我们应该本着使错误尽早暴露并保证调度器以及爬取流程能够正确运转的目标来进行参数值检查。

首先,我们来检查基本数据类型的参数channelLen、poolSize和crawlDepth的值。因为这几个参数的类型都是uint32,所以我们并不担心它们的值会是负数。如此一来,对于channelLen和poolSize的值来说,只要它们的值不等于0就可以通过检查。而对于crawlDepth的值,其类型的约束就已经足够了。相应的代码如下:

```

if channelLen == 0 {
    return errors.New("The channel max length (capacity) can not be 0!\n")
}
sched.channelLen = channelLen
if poolSize == 0 {
    return errors.New("The pool size can not be 0!\n")
}
sched.poolSize = poolSize
sched.crawlDepth = crawlDepth

```

可以看到，一旦参数值通过检查，我们就会把它们赋给相应的字段。在这之后，我们就可以使用它们来初始化一些需要它们的字段了，如下所示：

```

sched.chanman = generateChannelManager(sched.channelLen)
if httpClientGenerator == nil {
    return errors.New("The HTTP client generator list is invalid!")
}
dlpool, err :=
    generatePageDownloaderPool(sched.poolSize, httpClientGenerator)
if err != nil {
    errMsg :=
        fmt.Sprintf("Occur error when get page downloader pool: %s\n", err)
    return errors.New(errMsg)
}
sched.dlpool = dlpool
analyzerPool, err := generateAnalyzerPool(sched.poolSize)
if err != nil {
    if err != nil {
        errMsg :=
            fmt.Sprintf("Occur error when get analyzer pool: %s\n", err)
        return errors.New(errMsg)
    }
}
sched.analyzerPool = analyzerPool

```

在这段代码中，我们使用包级私有的函数`generateChannelManager`、`generatePageDownloaderPool`和`generateAnalyzerPool`分别生成了通道管理器、网页下载器池和分析器池的实例。其中还包含了对`httpClientGenerator`参数的值的非`nil`检查。如果没有出现任何错误，我们就把它们分别赋给了调度器的相应字段。否则，我们就会立即中止开启调度器的流程，并将错误值返回给调用方。

上面所说的那3个函数仅仅是封装了相应的组件初始化函数。至于这些组件初始化函数，我们都在前面讲述过。它们的名称都是以“New”为前缀的。读者应该可以自己编写出那3个函数的函数体。因此，我就不在此赘述了。

接下来，我们继续使用其他参数值来初始化另一个处理模块——条目处理管道。这会用到代表条目处理器列表的`itemProcessors`参数的值。`itemProcessors`是切片类型的。所以我们既要检查它本身，也要检查其中的元素值。在确保它的有效性之后，我们使用它来生成调度器的`itemPipeline`字段的值。具体代码如下：

```

if itemProcessors == nil {
    return errors.New("The item processor list is invalid!")
}

```

```

for i, ip := range itemProcessors {
    if ip == nil {
        return errors.New(fmt.Sprintf("The %dth item processor is invalid!", i))
    }
}
sched.itemPipeline = generateItemPipeline(itemProcessors)

```

其中的generateItemPipeline函数的实现同样非常简单。读者也完全可以自行编写。我们编写这些简单的封装函数的主要原因是：将各种组件初始化函数隔离在调度器的核心实现代码之外。这样可以使代码更加清晰，也可以为今后可能进行的修改提供方便。此类函数的完整实现都被放到了webcrawler/scheduler代码包的库源码文件helper.go中。

到目前为止，还有一个调度器需要用到的中间件工具未被初始化。它就是停止信号。虽说开启调度器的时候用不上它，但是它却可能会影响到爬取流程的正常执行。如果我们发现调度器的stopSign字段的值为nil，就说明这是对当前调度器的Start方法的首次调用。这时，我们应该通过调用mdw.NewStopSign函数为它赋值。否则，就意味着这是一个已被使用过且已被停止的调度器。这时我们就应该调用这个停止信号的Reset方法来重置它。如果不这样做，调度器中的一些代码在执行起初就会因此而被停止。这也是停止信号应有的作用。我们在后面会讲到这方面的内容。

根据前面的描述，我们有了下面这个if代码块：

```

if sched.stopSign == nil {
    sched.stopSign = mdw.NewStopSign()
} else {
    sched.stopSign.Reset()
}

```

现在，myScheduler类型的绝大部分字段都已经被初始化好了，除了reqCache、urlMap和primaryDomain。我们稍后再说primaryDomain字段。先来看其他两个字段。对于urlMap字段，我们直接使用内建函数make来为它的值初始化就好了：

```

sched.urlMap = make(map[string]bool)

```

而对于reqCache字段的值的创建和初始化，我们有专门的函数newRequestCache。该函数被调用后会返回一个requestCache类型的结果值。我们会把这个值赋给reqCache字段。关于这个函数的具体实现，我们会在下一小节同requestCache类型的各个方法一起进行阐述。

在初始化了所有字段之后，我们就要真正进入到开启调度器的环节了。下面这4行代码就代表了这一环节所需要执行的全部动作：

```

sched.startDownloading()
sched.activateAnalyzers(respParsers)
sched.openItemPipeline()
sched.schedule(10 * time.Millisecond)

```

可以看到，上面的每一行代码都是一条针对myScheduler类型的某一个方法的调用语句。其中，startDownloading方法在被调用后会开始等待请求通道中的可用请求。从方法名称上来看，activateAnalyzers方法会激活分析器。更具体地说，它会不断地试图从响应通道处接收响应并将其输送给分析器池中空闲的分析器。我们也可以把它看作是由响应激活的分析过程。openItemPipeline

方法的功能显然是打开条目处理管道。它和前两个方法中的操作模式是一致的，即使用通道中的数据来激活相应的处理模块。至于schedule方法，它所执行的就是我们在上一小节的末尾处所说的“适当地将请求缓存中的请求向请求通道转移”的操作。请注意这里传递给该方法的参数值（即表达式 `10 * time.Millisecond` 的求值结果）。它就代表了执行上述操作的时间间隔。

我们会在后面对这4个方法进行专门的论述。现在我们继续往下看。如果调度器对上面这4个方法的调用都成功了，就意味着网络爬虫框架已经完全运作起来了。这时，我们要做的就是将首次请求放到请求缓存中。不过在这之前，我们还要做一项之前遗留下的任务——为代表了主域名的primaryDomain字段赋值。

我们在前面说过，primaryDomain字段的值应该是由参数firstHttpReq的值分析得出的。从Start方法的签名可知，该参数是*http.Request类型的。而http.Request类型有一个名为Host的字段。若firstHttpReq参数的值是有效的，那么它的Host字段就应该代表了目标网络站点的主机地址。这个主机地址可能是一个IP地址，也可能是一个域名。我们把从这个主机地址中抽离出主域名的相关操作封装在了一个名为getPrimaryDomain的函数中。因此，在Start方法中的相应代码如下所示：

```
if firstHttpReq == nil {
    return errors.New("The first HTTP request is invalid!")
}
pd, err := getPrimaryDomain(firstHttpReq.Host)
if err != nil {
    return err
}
sched.primaryDomain = pd
```

函数getPrimaryDomain会接受一个参数值，并返回两个结果值。参数值是string类型的。它应该为一个IP地址或者一个域名。第一个结果值同样是string类型的。它代表了从参数值中得到的主域名。第二个结果值是error类型的。如果在获取主域名的过程中出现了错误，那么该值就会是非nil的。至于该函数的实现，我们也不细说了。由于从一个完整的域名中抽出主域名的方式有很多，因此这会是一个很有意思的练习题。读者可以先自己编写一个getPrimaryDomain函数的实现，然后再对照我写的实现（也在webcrawler/scheduler包的helper.go文件中）。说不定读者的实现方案会比我的更好。

在给primaryDomain字段赋值之后，我们就可以把firstHttpReq参数的值送到请求缓存中了。注意，请求缓存所缓存的是我们自定义的*base.Request类型（webcrawler/base代码包中的base.Request类型的指针类型）的值，而不是像firstHttpReq这样的值。因此，我们还需要使用base.NewRequest函数把firstHttpReq包装成一个请求缓存能够接受的值。只有这样，我们才能够的把它放入到请求缓存中。具体的代码如下：

```
firstReq := base.NewRequest(firstHttpReq, 0)
sched.reqCache.put(firstReq)
```

字段reqCache的值得方法put执行的就是把请求放入到其所属的请求缓存中的操作。

在这之后，我们也就执行完成了开启调度器所需的所有操作了。最后，我们返回一个nil。还记得吗？Start方法的签名中仅声明了一个error类型的结果。它表示在开启调度器的过程中未发生任何错误。

现在，让我们趁热打铁，说明一下前面一笔带过的那几个被用于开启爬取流程的调度器的方法。

2. 开始下载

方法startDownloading既不接受任何参数值，也不返回任何结果值。它所做的就是不断地试图从请求通道中获取请求并交由空闲的网页下载器处理。其完整声明如下：

```
// 开始下载。
func (sched *myScheduler) startDownloading() {
    go func() {
        for {
            req, ok := <-sched.getReqChan()
            if !ok {
                break
            }
            go sched.download(req)
        }
    }()
}
```

为了让该方法中的代码能够更加清晰地表示这一环节，我们又创建了两个函数来分担一些具体的操作。getReqChan方法被用来从通道管理器处获取请求通道，它的完整声明如下：

```
// 获取通道管理器持有的请求通道。
func (sched *myScheduler) getReqChan() chan base.Request {
    reqChan, err := sched.chanman.ReqChan()
    if err != nil {
        panic(err)
    }
    return reqChan
}
```

我们已经知道，如果通道管理器chanman的ReqChan方法返回的第二个结果值是非nil的，那么就说明该通道管理器未处于已初始化状态。这也就意味着调度器没有正确地使用它。这个异常的状态会导致严重的后果。整个爬取流程会因此不能正常地运转。所以，一旦发现了此类异常，我们就应该引发一个运行时恐慌。如果我们使用mdw.NewChannelManager函数来创建和初始化调度器的chanman字段的值，并且没有在不适当的时候调用chanman的Close方法，那么这里的err变量的值就绝对会是nil。因此，虽然这里报告错误的方式非常严厉，但是我们并不担心它会影响到正常的流程。除非爬取流程真的无法正常运行。

方法startDownloading中会调用的另一个方法是同属于一个调度器的download。顾名思义，该方法的职责是下载与请求对应的网页内容。更具体地说，它会通过某个网页下载器与目标网络站点进行交互，并把交互的结果（即响应或错误）发送给相应的通道。我们知道，调用器所使用的都会是处于网页下载器池中的网页下载器。因此，就有了下面这几行代码：

```

downloader, err := sched.dlpool.Take()
if err != nil {
    errMsg := fmt.Sprintf("Downloader pool error: %s", err)
    sched.sendError(errors.New(errMsg), SCHEDULER_CODE)
    return
}
defer func() {
    err := sched.dlpool.Return(downloader)
    if err != nil {
        errMsg := fmt.Sprintf("Downloader pool error: %s", err)
        sched.sendError(errors.New(errMsg), SCHEDULER_CODE)
    }
}()

```

其中，标识符`sched`代表的是该方法所属的调度器。可以看到，我们调用网页下载器池的`Take`方法以试图从中获取一个空闲的网页下载器，并通过在`defer`语句中调用它的`Return`方法来保证在使用过后及时归还该网页下载器。至于我们第一次见到的标识符`SCHEDULER_CODE`和调度器的方法`sendError`，我们稍后就会给予说明。

只要可以成功地从池中取出空闲的网页下载器，我们就立即使用它来进行相应的下载操作：

```

code := generateCode(DOWNLOADER_CODE, downloader.Id())
respp, err := downloader.Download(req)
if respp != nil {
    sched.sendResp(*respp, code)
}
if err != nil {
    sched.sendError(err, code)
}

```

其中，标识符`DOWNLOADER_CODE`代表了一个常量。实际上，我们为调度器和每一个处理模块都定义了这样一个常量：

```

// 组件的统一代号。
const (
    DOWNLOADER_CODE = "downloader"
    ANALYZER_CODE   = "analyzer"
    ITEMPIPELINE_CODE = "item_pipeline"
    SCHEDULER_CODE  = "scheduler"
)

```

我们在生成某个组件的实例的唯一代号时会用到相应的常量。

另外，在这段代码中我们还遇见了几个陌生的函数和方法。函数`generateCode`的功能是为某个网页下载器、分析器或条目处理管道的实例生成一个全局唯一的代号。这些代号主要是为了在调度器的某处响应停止信号的时候记录下涉及的组件。我们可以看到，在调用调度器的`sendResp`方法和`sendError`方法的时候，相应的代号都被作为参数值传入。请看`sendResp`方法的实现代码：

```

// 发送响应。
func (sched *myScheduler) sendResp(resp base.Response, code string) bool {
    if sched.stopSign.Signed() {

```

```

        sched.stopSign.Deal(code)
        return false
    }
    sched.getRespChan() <- resp
    return true
}

```

其中的调度器的方法`getRespChan`会获得通道管理器持有的响应通道，并在确保无错的情况下将其返回给调用方。它的具体实现与`getReqChan`方法几乎是一致的。

在使停止信号`stopSign`处于已发出状态之后，当前的调度器理应逐一关闭它所使用的内部组件，其中就包括了通道管理器。毫无疑问，通道管理器在被关闭的时候会关闭其管理的所有通道。而向一个已被关闭的通道发送元素值一定会引发一个运行恐慌。正因为有着一系列的连带关系，所以我们在向响应通道发送元素值的之前，一定要检查一下停止信号。如果停止信号已被置位，那么我们就忽略此发送操作。这时，我们在调用停止信号的`Deal`方法的时候，会告诉它哪一个组件因此受到了影响并且忽略了后续操作。

显然，这里的停止信号检查起到了两个作用。第一个作用是响应停止信号并中止当前的流程，而第二个作用就是防止因不必要的运行时恐慌的引发而影响正常的停止流程的执行。

这对于调度器的`sendError`方法来说也是一样的。`sendError`方法的声明如下：

```

// 发送错误。
func (sched *myScheduler) sendError(err error, code string) bool

```

方法`sendError`在检查停止信号之前还做了另外一项工作。那就是封装错误值，即使用`base.NewCrawlerError`函数封装作为参数值传入的错误值，并生成相应的`base.CrawlerError`类型值。这样做有两点好处。一个是使被发送给错误通道的错误值整齐划一了（因为它们都是同一个类型的值），另一个是`base.CrawlerError`类型值的`Error`方法会返回更加丰富的错误提示信息。

该方法中的前几行代码如下：

```

if err == nil {
    return false
}
codePrefix := parseCode(code)[0]
var errorType base.ErrorType
switch codePrefix {
case DOWNLOADER_CODE:
    errorType = base.DOWNLOADER_ERROR
case ANALYZER_CODE:
    errorType = base.ANALYZER_ERROR
case ITEMPIPELINE_CODE:
    errorType = base.ITEM_PROCESSOR_ERROR
}
cError := base.NewCrawlerError(errorType, err.Error())

```

首先，我们需要根据参数`code`的值分析出错误的类型。我们在9.4节中声明了几个`base.ErrorType`类型的常量。在这段代码中名称以“_ERROR”结尾的那些限定标识符代表的正是这些常量。它们中的每一个都与某一个处理模块相对应。我们通过调用`parseCode`方法，从`code`的值中解析出组件的统一代号。然后再依据这些代号为`errorType`变量赋值。请注意，没有与代表调度器的统

一代号SCHEDULER_CODE对应的errorType类型的常量。这是因为存在这样一个约定：如果被传入NewCrawlerError函数的这个base.ErrorType类型值为nil，那么该函数返回的错误值就代表了调度器自身报告的错误。

变量cError即代表了我们将要发送给错误通道的那个base.CrawlerError类型值。当然，在真正发送它之前，我们是一定要检查停止信号的：

```
if sched.stopSign.Signed() {
    sched.stopSign.Deal(code)
    return false
}
```

如果这时的停止信号未处于已发出状态，那么我们就可以放心地发送错误值了：

```
go func() {
    sched.getErrorChan() <- cError
}()
```

其中的方法getErrorChan的实现方式与前面提到的方法getReqChan和getRespChan是非常相似的。除此之外，调度器的方法集合中还包含一个名为getItemChan的方法。它被用来获取当前调度器所持的通道管理器中的条目通道。

请注意，这里把向错误通道发送错误值的操作放在了一个单独的Goroutine中去执行。这是因为错误通道不像通道管理器持有的其他通道那样，会由调度器负责相应的接收操作的执行。我们只能寄希望于调度器的使用者会不断地通过对调度器的ErrorChan方法的调用获得错误通道并从中接收错误值。这样才能够基本避免因错误通道已满而阻塞住执行相应的发送操作的Goroutine的情况。我们决不能因调用方的行为未如所愿而使调度器内部的流程停滞。否则，网络爬虫框架本身的稳定性就是不合格的。

综上所述，我们在这里使用了单独的Goroutine来执行发送操作。如此一来，即使错误通道已满也不会阻碍爬取流程的正常进行。虽然这样做会在错误大量出现的时候使Goroutine的数量陡增，但是这总比爬取流程的执行被停滞要好得多。不过，不论怎样，我们都应该强烈建议调度器的使用者要不断地从错误通道中接收错误值。

好了，在sendError方法中的最后一条语句是return true。虽然我们并不知道发送错误值的操作是真的成功了还是正在被阻塞着，但是这对于爬取流程的执行来说并不重要。

现在回到调度器的download方法。正如前面所讲，该方法会在调用网页下载器的Download方法之后等待其结果值，并在必要时把它们发送给相应的通道。这也是它需要做的最后一部分工作。不过，我们还应该在这个download方法的开始处加入一条defer语句，具体如下：

```
defer func() {
    if p := recover(); p != nil {
        errMsg := fmt.Sprintf("Fatal Download Error: %s\n", p)
        logger.Fatal(errMsg)
    }
}()
```

添加这条defer语句的目的是恢复网页下载器的Download方法可能抛出的运行时恐慌。

Download方法终归是需要通过网络完成任务的。我们要把这样的运行时恐慌转化为日志记录下来。如此可以避免因某次下载的意外失败而影响整个流程的执行。另外，之所以不把它转化为错误值并发送给错误通道，是因为我们想要及时地暴露它。等到调度器的使用方从错误通道接收到它的时候就太晚了。并且，那样会使它与代表普通错误的值混在一起。

再回到调度器的方法startDownloading。startDownloading方法每每在从请求通道中接收到有效的请求之后，都会调用download方法予以处理。请注意，这里针对download方法的调用表达式是被包含在go语句中的。也就是说，它是被异步地调用的。其中的代码会在一个单独的Goroutine中被执行。这样做的目的是以尽可能快的速度从请求通道中取出请求。这对于向请求通道发送请求的一方是很有好处的。因为，如此一来请求通道就几乎不会出现被填满的情况。这也意味着发送方几乎不会因此而被阻塞。但是，这样做的代价是Goroutine的数量可能会增长得快一些。具体的增幅取决于网页下载器池的尺寸以及每一次下载的速度。对于后者，我们并不能控制。这更多的会受到网络速度以及目标网络站点的服务程序的性能的制约。而对于前者，我们可以通过传入调度器的参数值（更具体地说，是参数poolSize的值）来控制。如果这个值给得得当，那么因此而被启用的Goroutine的数量就应该会维持在一个比较合理的范围之内，起码其数量的增幅曲线不会非常陡峭。

3. 激活分析器

调度器的方法activateAnalyzers会紧跟在startDownloading方法之后被调用。下面我们就来讲解它所执行的流程。其中，它与startDownloading方法所做的事情有很多雷同之处。下面就是它的完整声明：

```
// 激活分析器。
func (sched *myScheduler) activateAnalyzers(respParsers []anlz.ParseResponse) {
    go func() {
        for {
            resp, ok := <-sched.getRespChan()
            if !ok {
                break
            }
            go sched.analyze(respParsers, resp)
        }
    }()
}
```

读者是不是感觉似曾相识呢？activateAnalyzers方法会用单独的Goroutine执行其中的所有代码。它会以尽量快的速度从响应通道处接收响应，然后并发地处理它们。

方法analyze会接受两个参数值。第一个参数值由respParsers表示，它代表了分析器所需的响应解析函数的序列。第二个参数由resp表示，它代表了需要被分析的响应。

与调度器的download方法相同，在analyze方法的方法体的开始处，我们会先使用defer语句构筑一道防线。这道防线可以避免因该方法内部抛出的运行时恐慌而影响整个爬取流程的执行。然后，我们会试图从调度器持有的分析器池中取出一个分析器。当然，这一操作可能会被阻塞，直到池中有可用的分析器为止。另外，为了保证在分析完响应之后及时将该分析器归还给分析器

池，我们还要编写一条defer语句。

与上面的描述对应的那几条语句如下：

```
defer func() {
    if p := recover(); p != nil {
        errMsg := fmt.Sprintf("Fatal Analysis Error: %s\n", p)
        logger.Fatal(errMsg)
    }
}()
analyzer, err := sched.analyzerPool.Take()
if err != nil {
    errMsg := fmt.Sprintf("Analyzer pool error: %s", err)
    sched.sendError(errors.New(errMsg), SCHEDULER_CODE)
    return
}
defer func() {
    err := sched.analyzerPool.Return(analyzer)
    if err != nil {
        errMsg := fmt.Sprintf("Analyzer pool error: %s", err)
        sched.sendError(errors.New(errMsg), SCHEDULER_CODE)
    }
}()
```

如果分析器的Take方法未返回非nil的错误值，那么我们就可正常使用从它那里得到的分析器了。还记得吗？调用分析器的Analyze方法可以使用网络爬虫框架的使用方给定的响应解析函数序列来分析指定的响应。另外，为了在必要时响应停止信号，我们还要为这个分析器生成好统一的代号。具体的代码如下：

```
code := generateCode(ANALYZER_CODE, analyzer.Id())
dataList, errs := analyzer.Analyze(respParsers, resp)
```

变量dataList代表的是由响应产出的数据列表。其中的数据可能是请求也可能是条目。变量errs代表的是在分析的过程中发生的错误的列表。当然，我们肯定希望这个列表中什么也没有。

鉴于dataList的各个元素值的类型是不定的，所以我们首先要在迭代地取出它们的时候判定其类型，这样才能够把它们发送给相应的通道。我们以前讲过多种类型判定的方式，其中有一种是与switch语句连用的。由于这里需要判断数据列表中的某一个元素值的类型是否为几种可能类型中的一种，所以我们使用switch语句会更加方便。下面，我们来看看具体的代码是怎样的。

在确保dataList不为nil的前提下，我们使用for语句迭代出其中的每一个元素：

```
if dataList != nil {
    for _, data := range dataList {
        if data == nil {
            continue
        }
        switch d := data.(type) {
        case *base.Request:
            sched.saveReqToCache(*d, code)
        case *base.Item:
            sched.sendItem(*d, code)
        default:
```

```

        errMsg := fmt.Sprintf("Unsupported data type '%T'! (value=%v)\n", d, d)
        sched.sendError(errors.New(errMsg), code)
    }
}

```

我们在这里允许的数据类型只有两种，它们是`*base.Request`和`*base.Item`。如果数据是`*base.Request`类型的，那么我们就把它发送给请求缓存。而如果它是`*base.Item`类型的，我们就应该将其发送给条目通道。但如果该数据不属于这两种类型，那么我们就应该及时生成一个能够表述此问题的错误值，然后把它发送给错误通道。这种情况是有可能的。因为分析器的`Analyze`方法返回的这些数据，都是由网络爬虫框架的使用方指定的某个响应解析函数的执行结果的一部分。我们一直在说，从网络爬虫框架内部的角度看，对这种外来代码的执行结果的检查和防范措施是必需的。另外，顺便提一句，读者注意到那条对`fmt.Sprintf`函数的调用语句中的`%T`了吗？它也是格式化字符串可以包含的合法动词之一。它被用来表示对应参数的实际类型。这样我们就可以从错误的提示信息中准确地了解到这个不被支持的数据类型到底是哪一种类型了。

当处理完作为分析结果的数据之后，我们还要继续检查`errs`变量，因为在产出若干数据的同时也可能会有些错误发生，也可能会有些结果数据因此而没有成型。所以，将这些错误报告给调度器的使用者也是很有必要的。不过，我们略过这部分的具体代码。读者应该可以根据前面的示例代表编写出它们。

现在我们回过头来再看前边展示的`switch`语句中的代码。在那里，我们使用到了两个调度器的方法：`saveReqToCache`和`sendItem`。其中`sendItem`方法的实现方式与前面讲过的`sendResp`方法如出一辙。我就不在这里赘述了。对于`saveReqToCache`方法，我们要在这里详细说明一下。

方法`saveReqToCache`会接受两个参数值并返回一个结果值。其声明如下：

```

// 把请求存放到请求缓存。
func (sched *myScheduler) saveReqToCache(req base.Request, code string) bool

```

我们都知道，请求缓存是为了缓解数据传输上的“请求-响应”闭环所带来的若干问题而诞生的。但是，作为请求的发送方，也应该尽量降低请求缓存的数据传输压力。更具体地说，我们仅应该把合法、合要求的请求放入请求缓存。尽早地对请求进行过滤可以有效地节约后续环节针对请求的传输、检查等一系列的操作成本。

这里需要用到的过滤规则有两类。一类是针对不合法的请求的规则，而另一类则是针对不合要求的请求的规则。第一类规则如下。

- 若请求值中包含的HTTP请求（`*http.Request`类型值）为`nil`，则该请求不合法。
- 若该HTTP请求的URL字段（类型为标准库代码包`net/url`中的URL类型的指针类型的值）为`nil`，则包含它的请求不合法。
- 若该URL的Scheme字段的值不是`http`，则该请求不合法。

前两条规则很好理解。而第三条规则是根据网络爬虫框架目前的能力而定的。现阶段，我们编写的这个网络爬虫框架仅支持基于HTTP协议的请求。所以，为了避免在后面出错，我们在这里就先把未被支持的协议的请求过滤掉了。

根据上述的第一类规则，我们有了下面这段代码：

```
httpReq := req.HttpReq()
if httpReq == nil {
    logger.Warnln("Ignore the request! It's HTTP request is invalid!")
    return false
}
reqUrl := httpReq.URL
if reqUrl == nil {
    logger.Warnln("Ignore the request! It's url is is invalid!")
    return false
}
if strings.ToLower(reqUrl.Scheme) != "http" {
    logger.Warnf("Ignore the request! It's url scheme '%s', but should be 'http'!\n", reqUrl.Scheme)
    return false
}
```

请注意，我们在这里的日志记录都是警告级别的。从另一方面讲，即使欲发送给请求缓存的请求是不合法的，我们也不会把它发送给错误通道。这主要是因为不合法的请求只会影响爬取的效果，而并不代表爬取流程的执行会因此而出问题。

如果请求是合法的，那么我们就需要进一步检查它是否合乎要求。相关规则如下。

- ❑ 如果请求包含的HTTP请求之前已经被处理过，那么该请求就是不合要求的。
- ❑ 如果该HTTP请求的目标网络站点的主机地址与我们从首次请求中得到的主域名不匹配，那么该请求就是不合要求的。
- ❑ 如果请求的深度超过了预设的爬取最大深度，那么该请求就是不合要求的。

对于一条规则，我们需要借助调度器实现类型myScheduler中的urlMap字段来实现。我们在上一小节的最后讲过，urlMap字段是字典类型的，它的值被用来存储已被处理过的请求的URL。如果我们发现该值中已经存在当前请求的URL，那么我们就认为当前请求已经被处理过了，它是不合要求的。相应的代码如下：

```
if _, ok := sched.urlMap[reqUrl.String()]; ok {
    logger.Warnf("Ignore the request! It's url is repeated. (requestUrl=%s)\n", reqUrl)
    return false
}
```

我们把针对urlMap的索引表达式产生的结果值赋给了两个标识符。还记得吗？第一个标识符“_”被称为空标识符。这样做相当于我们把它右边的索引表达式的第一个结果值直接扔掉了。确实，我们在这里只需要用到该表达式的第二个bool类型的结果值。如果局部变量ok的值是true，那么就说明当前请求的URL已存在于urlMap中了。这时，我们就应该忽略对该请求的处理。

第二条规则是针对请求的主域名的。它的关注点在目标网络站点的主机地址上。这个时候，我们在之前已赋值的调度器的primaryDomain字段就可以派上用场了。我们在讲怎样生成该字段的值的时候说过，这个值可能是一个IP地址，也可能是一个域名的主域名部分。不过，无论它是什么，我们都可以使用前面提到的getPrimaryDomain函数来得到它的主域名。首先，我们调用该方法以获得当前的HTTP请求的主域名。然后，我们再去判断当前HTTP请求的主域名是否与primaryDomain字段的值相等。如果不相等，那么就说明该请求是不合要求的。具体代码如下：

```

if pd, _ := getPrimaryDomain(httpReq.Host); pd != sched.primaryDomain {
    logger.Warnf("Ignore the request! It's host '%s' not in primary domain '%s'. (requestUrl=%s)\n",
        httpReq.Host, sched.primaryDomain, reqUrl)
    return false
}

```

在上面这条if语句中，我们同样用到了空标识符“_”。它被用来扔掉getPrimaryDomain函数的第二个结果值。其含义是，即使在获取当前HTTP请求的主域名的过程中发生了错误，我们也不关心。我们只关心该函数的第一个结果值是否与primaryDomain字段的值相等。

第三条规则的实现非常简单。因为我们已经在分析器中对每个新的请求的深度进行了必要的检查和修正。所以，我们在这里只需要判断当前请求的深度是否大于调度器的crawlDepth字段的值即可。具体代码是这样的：

```

if req.Depth() > sched.crawlDepth {
    logger.Warnf("Ignore the request! It's depth %d greater than %d. (requestUrl=%s)\n",
        req.Depth(), sched.crawlDepth, reqUrl)
    return false
}

```

如果一个请求通过了上述的6项检查，那么它就是既合法又合要求的。这就意味着它可以被存放到请求缓存中了。还记得吗？我们在向任何一个通道发送元素值之前都要检查一下停止信号的状态。对于把请求存放到请求缓存的这项操作来说，也应该是这样。只有在此时的停止信号未处于已发出状态的情况下，我们才去执行发送操作，否则就要处理停止信号并忽略后续的操作。相应的代码如下：

```

if sched.stopSign.Signed() {
    sched.stopSign.Deal(code)
    return false
}
sched.reqCache.put(&req)

```

只要上面的最后一条语句执行完毕，我们就可以宣告完成了对当前请求的缓存。这时就应该结束函数saveReqToCache的执行并将true作为结果返回。不过，为了调度器的urlMap字段能够发挥应有的作用，我们一定要记得把缓存成功的请求的URL推入到该字段的值中：

```

sched.urlMap[reqUrl.String()] = true

```

现在回到调度器的analyze方法。我们刚刚已经讲完对分析器的Analyze方法返回的第一个结果值（由变量dataList代表）的处理。接下来，我们会去处理分析器的Analyze方法的第二个结果值。它由变量errs代表。我们应该迭代errs变量的值，并把它包含的每一个error类型值都发送给错误通道（通过调用调度器的sendError方法）。

至此，我们就完成了对某一个响应的分析过程。经过调度器的activateAnalyzers方法中的一次次迭代，analyze方法会被频繁地调用。后者会分析前者从响应通道中接收到响应。与对请求通道中的各个请求的处理形式相同，这一过程是被并发地进行的。

4. 打开条目处理管道

调度器仅持有条目处理管道的一个实例而不需要相应的池。这样更有利用于记录与条目处理

有关的各项计数。打开条目处理管道的任务由调度器的`openItemPipeline`方法负责，它的声明如下：

```
// 打开条目处理管道。
func (sched *myScheduler) openItemPipeline()
```

与开始下载和激活分析器的操作一样，我们对条目处理管道的打开动作应该在一个单独的Goroutine中进行，如下所示：

```
go func() {
    sched.itemPipeline.SetFailFast(true)
    code := ITEMPIPELINE_CODE
    for item := range sched.getItemChan() {
        go func(item base.Item) {
            // 省略若干条语句
        }(item)
    }
}()
```

在这个专用的Goroutine中，我们先把条目处理管道的“快速失败”的状态设置为了`true`。还记得吗？“快速失败”的意思是：只要一个条目被某个条目处理器处理失败了，条目处理管道就不会让排在后面的条目处理器再去处理该条目了。

设置“快速失败”状态以及生成条目处理管道的组件统一代号都属于准备工作。在这之后，我们就要不断尝试从条目管道中接收条目并处理它。利用`for`语句从通道处接收元素值可以保证接收操作的连续性，并且还可以在通道关闭的时候及时结束接收操作。同时，为了可以并发地处理陆续接收到的条目，我们还要使对每一个条目的处理动作都异步化。也就是说，我们要为每一个条目的处理都启用一个专用的Goroutine。不过，一定要注意，我们要把被处理的条目作为参数值传入到相应的`go`函数中。即使这个`go`函数存在于存放条目的变量的作用域之内，也不要再在`go`函数中直接使用它。至于原因，我们在前面也多次提到过：Go语言的运行时系统对`go`函数的执行是有延时的（对于`defer`语句来说更是如此）。在`go`语句被执行的时候，Go语言的运行时系统会把`go`函数分配给一个刚被启用的Goroutine。但该Goroutine并不会立即执行这个`go`函数。`for`语句中的迭代变量（在这里是`item`）会随着迭代的一次又一次地进行而被一次又一次地赋予不同的值。等到`go`函数真正被执行的时候，当初的那个迭代值很可能已经被后续的值覆盖掉了。换句话说，这时与迭代变量绑定的那个值很可能已经不是`go`语句当初被执行时的那个值了。如果读者依然疑惑，还可以回顾4.2节和7.1节中对此问题的描述。

好了，下面我们来编写上面的`for`代码块中的那个`go`函数的函数体。首先，我们依然要构筑一道防线：

```
defer func() {
    if p := recover(); p != nil {
        errMsg := fmt.Sprintf("Fatal Item Processing Error: %s\n", p)
        logger.Fatal(errMsg)
    }
}()
```

然后，我们要把作为参数值传入的条目发送给条目处理管道：

```
errs := sched.itemPipeline.Send(item)
```

最后，我们检查errs变量的值，并在必要时把其中的错误值逐个地发送给错误通道：

```
if errs != nil {
    for _, err := range errs {
        sched.sendError(err, code)
    }
}
```

以上就是openItemPipeline方法中的全部代码。有了对前面的那些方法的讲解作为铺垫，该方法中的这些代码应该很好理解。

5. 调度请求

在调度器的实现类型myScheduler的指针方法Start中，我们先对它的绝大部分字段进行了初始化。然后，我们又通过调用它的方法startDownloading、activateAnalyzers和openItemPipeline完成了对网络爬虫框架的各个处理模块的启动。我们使用各种通道把这些处理模块的输入和输出链接在了一起。我们几乎已经大功告成了。

不过，别忘了，我们在前面只把分析器返回的请求存入了请求缓存，而没有把它们转发给请求通道，而startDownloading方法获取请求的来源却是请求通道。这里存在着一个断链的情况。这也是为什么我们要在Start方法中调用前面那3个方法之后还要调用一个名为schedule的方法的原因。我们依靠schedule方法中的代码把存在于请求缓存中的请求搬运到请求通道中。

可以想象，这样的搬运工作应该在一个for代码块中进行。因为它的进行应该是持续不断的。另外，这项工作的进行不能影响到Start方法中后续任务的执行。因此，schedule方法的实现应该大致是这样的：

```
// 调度。适当地搬运请求缓存中的请求到请求通道。
func (sched *myScheduler) schedule(interval time.Duration) {
    go func() {
        for {
            // 省略若干条语句
        }
    }()
}
```

我们下面要做的就是将for代码块中的代码补全。显然，schedule方法的参数interval的值会决定for代码块中代码的执行时间间隔。这很好控制。而这段代码本身需要做的事情有两个：适当地搬运请求和及时地响应停止信号。这里的“适当”有两层含义。一个是根据请求通道中的空余位置的多少来决定计划搬运请求的数量，另一个是根据请求缓存中存储的请求的多少来决定实际搬运请求的数量。获取请求通道中的空余位置的数量可以通过下面这行代码来实现：

```
remainder := cap(sched.getReqChan()) - len(sched.getReqChan())
```

显然，通道的容量减去其实际包含的元素值的数量就等于空位数量。我们根据这个空位数量来循环地进行搬运。不过，一旦请求缓存空了，我们就应该停止搬运。相应的代码如下：

```

var temp *base.Request
for remainder > 0 {
    temp = sched.reqCache.get()
    if temp == nil {
        break
    }
    sched.getReqChan() <- *temp
    remainder--
}

```

当搬运工作完成后，我们要按给定的时间间隔等待一会儿：

```
time.Sleep(interval)
```

如果不考虑对停止信号的及时响应的話，上面这些就是schedule方法的外层for代码块中的全部代码了。对于响应停止信号的代码，我们早已有了编写套路，如下所示：

```

if sched.stopSign.Signed() {
    sched.stopSign.Deal(SCHEDULER_CODE)
    return
}

```

注意，我们在响应完停止信号之后，不是使用break退出当前的for循环，而是使用return语句直接让所在的go函数退出执行。由于当前函数的函数体中仅包含了这个嵌套多层for循环的代码块，因此return语句会比goto语句更加直接了当。

上面这段被用来响应停止信号的代码应该被放在两处。一处是在外层for循环的开始处，这主要是为了及时制止停止信号被发出后的请求搬运工作。另一处是在把请求真正发送给请求通道之前。根据我们的经验，这样做是很有必要的。

方法schedule的实现代码就是这些。我们在搞清了这个方法的职责和相应规则之后，实现它就很容易了。

好了，到这里，我们已经详细地讲解了开启调度器所需的所有步骤：检查参数值、初始化字段、启动处理模块、适当的调度请求，以及使用首次请求触发整个爬取流程。其中，启动处理模块包含了开始下载、激活分析器和打开条目处理管道这3项任务。我们启动了一些Goroutine来分别完成这些任务。为了便于管理和统计，它们都是在调度器中被启用的。依据它们执行的代码的功能，我们可以把它们划分为几组：处理请求的Goroutine组、处理响应的Goroutine组、处理条目的Goroutine组、处理错误的Goroutine组，以及搬运请求的Goroutine（组）。总体来讲，我们并没有对Goroutine的数量进行限制。但是，通过对网页下载器池和分析器池的容量的指定却可以从侧面对相应的Goroutine的数量有所约束。另一方面，虽然爬取流程中的这些环节的代码都由不同的Goroutine（组）负责执行，但是通过我们为调度器装配的通道管理器仍然使它们成为了一个有机的整体。这种既彼此分散又彼此关联的程序结构的建立，得益于Go语言原生的两大并发编程法宝——Goroutine（也可称为Go程）和Channel（也可称为通道）。

6. 停止调度器

在Start方法被执行结束之后，调度器就真正处于运行状态了。另一方面，由于我们在爬取流程的各个环节中埋下了许多响应停止信号的代码，因此停止调度器的操作将会是非常便捷和有

效的。下面，我们就一起来编写可以发起调度器停止操作的Stop方法的实现。

与我们在前面花费大量篇幅讲解的Start方法相比，Stop方法的实现要简单很多。首先，我们先要基于调度器的running字段的值作出判断，看看是否有必要执行停止操作。显然，仅当调度器已处于运行状态的情况下才需要我们这样做。因此，就有了下面这条if语句：

```
if atomic.LoadUint32(&sched.running) != 1 {
    return false
}
```

当发现调度器未处于已运行状态时，我们会直接将false作为结果值返回给Stop方法的调用方。这会让调用方知道停止操作并没有被进行。若调度器正处于运行状态，那么我们就应该立即让调度器的停止信号处于已发出状态：

```
sched.stopSign.Sign()
```

这会使网络爬虫框架及时忽略掉相应的操作，比如给某个通道发送元素值、搬运请求，等等。在这之后，我们就可以关闭调度器持有的、能够被关闭的所有组件了。这包括了通道管理器和请求缓存。就像下面这样：

```
sched.chanman.Close()
sched.reqCache.close()
```

虽然以停止信号为纽带的停止操作是部分异步的，但是这时我们已经可以宣告调度器已处于停止状态了。我们原子地变更running字段的值，并将true作为Stop方法的结果值返回：

```
atomic.StoreUint32(&sched.running, 2)
return true
```

以上就是Stop方法的方法体中的全部代码。实际上，我们在网络爬虫框架的调度器的停止操作上的设计部分借鉴了Go语言运行时系统中的调度器的做法。不过，后者要更加复杂。

7. 调度器状态判断

对于调度器的使用方来讲，调度器仅有两种状态：未运行和已运行。使用方可以通过调用调度器的Running方法来获知它的状态。Running方法的实现代码要做的仅仅是把内部状态转换为外部状态并返回它。所以，Running方法的全部实现代码只有这些：

```
func (sched *myScheduler) Running() bool {
    return atomic.LoadUint32(&sched.running) == 1
}
```

8. 获得错误通道

我们可以调用调度器的ErrorChan方法以获取错误通道。该方法的唯一结果的类型为<-chan error。这从根本上杜绝了它的调用方向错误通道发送元素值的行为。也正因为此，我们可以放心地将调度器内部的错误通道传给外界。不过，我们在这里还应该传达另外一种信息，那就是错误通道是否可用。那么，在哪些情况下错误通道是不可用的呢？很明显，当包含它的通道管理器未处于已初始化状态的时候，错误通道是不可用的。在这种情况下，通道管理器中的错误通道可能还未被初始化，也可能已被关闭。为了给予ErrorChan方法的调用方以统一的表示，我们这时会返回一个nil。这有利于调用方尽早作出判断。

将上面的描述转换为ErrorChan方法的实现即是：

```
func (sched *myScheduler) ErrorChan() <-chan error {
    if sched.chanman.Status() != mdw.CHANNEL_MANAGER_STATUS_INITIALIZED {
        return nil
    }
    return sched.getErrorChan()
}
```

其中的限定标识符mdw.CHANNEL_MANAGER_STATUS_INITIALIZED代表的是一个常量。我们在讲通道管理器的时候提到过它。它被用来表示通道管理器处于已被初始化状态时的状态值。

9. 是否空闲

调度器的Idle方法会返回一个bool类型的结果值。该结果值被用来表示网络爬虫框架中的所有处理模块是否都正在空闲着。由于调度器会对这些处理模块进行管理和调度，所以它应该是最有发言权的。这也是要求调度器的实现类型有这样一个方法的主要原因。对于网页下载器池和分析器池，我们可以获取到它们中的已被使用的实体（网页下载器或分析器）的数量，并以此作为判断依据。而对于条目处理管道，它正在处理的条目的数量即是我们所需要的。如此一来，我们就可以用下面这3个bool类型的局部变量来分别表示它们的空闲状态了：

```
idleDlPool := sched.dlpool.Used() == 0
idleAnalyzerPool := sched.analyzerPool.Used() == 0
idleItemPipeline := sched.itemPipeline.ProcessingNumber() == 0
```

有了这3个变量，Idle方法的结果值就很好生成了：

```
if idleDlPool && idleAnalyzerPool && idleItemPipeline {
    return true
}
return false
```

10. 获取摘要信息

获取摘要信息的功能由调度器的Summary方法来提供。Summary方法唯一接受的参数prefix完全是为了打印摘要信息时排版的需要。该方法还会返回一个SchedSummary类型的结果值。SchedSummary是一个接口类型。我们在讲调度器的接口类型的时候已经对它进行过说明。我们将通过一个名为NewSchedSummary的函数来创建和初始化一个SchedSummary类型的值。该函数的声明如下：

```
func NewSchedSummary(sched *myScheduler, prefix string) SchedSummary
```

可以看到，该函数接受一个可以代表调度器实例的*myScheduler类型的参数和一个名为prefix的参数。这两个参数的值都可以由myScheduler类型的指针方法Summary中的代码提供。在Summary方法中，我们调用NewSchedSummary函数、传入这两个参数值，并直接把它的结果值作为Summary方法的结果值返回。我们在这里暂时不提NewSchedSummary函数的实现。在本节的最后会有对此的专门描述。

总之，这个Summary方法的完整实现如下：

```
func (sched *myScheduler) Summary(prefix string) SchedSummary {
    return NewSchedSummary(sched, prefix)
}
```

说到这里，我们几乎讲到了所有与调度器的实现类型myScheduler有关的函数和方法。我们实现的那些公开的方法使*myScheduler类型成为了Scheduler接口类型的一个实现类型，也使NewScheduler函数能够顺利地通过编译。

由于myScheduler类型拥有一个requestCache类型的字段reqCache，所以我们在本小节多次提到了该类型的若干方法。我们接下来就会专门论述这个代表了请求缓存的接口类型。此外，我们在后面还会对调度器摘要信息接口SchedSummary的实现类型及相关函数进行讲解。这包括我们刚才并未展开的NewSchedSummary函数。希望读者在阅读后面的章节之前能够真正地理解本小节所讲的内容。它们是调度器乃至整个网络爬虫框架中的最核心的部分。

9.7.3 请求缓存

请求缓存的行为是由webcrawler/scheduler代码包中的接口类型requestCache定义的。这个接口类型的完整声明如下：

```
// 请求缓存的接口类型。
type requestCache interface {
    // 将请求放入请求缓存。
    put(req *base.Request) bool
    // 从请求缓存获取最早被放入且仍在其中的请求。
    get() *base.Request
    // 获得请求缓存的容量。
    capacity() int
    // 获得请求缓存的实时长度，即其中的请求的即时数量。
    length() int
    // 关闭请求缓存。
    close()
    // 获取请求缓存的摘要信息。
    summary() string
}
```

我们在上一小节的示例中可以找到该声明中的许多方法。注意，该接口类型是访问权限是包级私有的。这意味着，我们只是把它以及实现它的类型当作了调度器的内部实现的一部分而已。我们把requestCache接口类型的实现类型命名为reqCacheBySlice。单从这个名称上看，读者可能就会猜测该类型会以切片值作为存储请求的介质。实际上的确如此。

类型reqCacheBySlice的基本结构比较简单，只包含了3个字段。首先是存储缓存中所有请求的[]*base.Request类型的字段cache。然后是被用来确保相关代码的并发安全性的字段mutex，该字段是sync.Mutex类型的。最后，既然有close方法，那么请求缓存必然会有两种状态。我们使用byte类型的字段status来表示这个状态。若status字段的值为0，则表示请求缓存正在运行。而当它的值为1时，就表示请求缓存已被关闭。

在这3个字段中，仅有cache字段需要被初始化。因此，被用来创建请求缓存的newRequestCache函数的完整声明如下：

```
// 创建请求缓存。
func newRequestCache() requestCache {
```

```

    rc := &reqCacheBySlice{
        cache: make([]*base.Request, 0),
    }
    return rc
}

```

请注意，我们把那个切片值的初始长度设为了0。这主要是为了让向其放入请求的代码可以简单一些。当然，这样做的缺点就是会使请求放入操作稍微慢一些。也就是说，我们在这里为了降低实现的复杂性而牺牲了一点性能。读者可以想一想这样做是否值得。

为了让*reqCacheBySlice类型可以成为requestCache接口类型的实现类型，我们还要为它添加一些方法。首先是put方法。在真正地把它的参数值放入到请求缓存之前，我们要进行两项检查。第一项检查的目的是确保参数值是有效的。对于*base.Request类型的值来说，我们首先要判断它是否为nil：

```

if req == nil {
    return false
}

```

实际上，我们只需进行这一项针对参数值的检查，更深入的检查是不应该由请求缓存来做的。读者可以回顾一下myScheduler类型的指针方法saveReqToCache。

第二项检查针对的是请求缓存的状态。显而易见，只有当请求缓存正处于运行状态的时候，把请求放入其中的操作才有意义。实现此逻辑的代码也很简单：

```

if rcache.status == 1 {
    return false
}

```

一旦通过了这两项检查，我们就应该立即锁住互斥锁mutex：

```

rcache.mutex.Lock()
defer rcache.mutex.Unlock()

```

这样做是为了对操作cache字段的值提供并发安全保障。因为，切片类型的值本身并不具备并发安全性。尤其是在切片值因包含的元素值增多而自动增长其容量（即替换它的底层数组）的时候，就更需要在并发环境中得到保护。

我们通过调用内建函数append把put方法的参数值追加到请求的切片cache之中。代码为：

```

rcache.cache = append(rcache.cache, req)

```

在此操作被执行完毕之后，我们就可以告知方法的调用方操作成功了（即返回true）。

与put方法相对应的是get方法。在get方法中，我们的操作步骤是相似的，即检查参数值的有效性、检查请求缓存的状态，并在通过检查后在锁的保护之下操作请求缓存中的切片值。get方法的完整声明如下：

```

func (rcache *reqCacheBySlice) get() *base.Request {
    if rcache.length() == 0 {
        return nil
    }
    if rcache.status == 1 {

```

```

        return nil
    }
    rcache.mutex.Lock()
    defer rcache.mutex.Unlock()
    req := rcache.cache[0]
    rcache.cache = rcache.cache[1:]
    return req
}

```

Go语言并没有提供取走切片值的头部元素值（即索引值0对应的那个元素值）的现成方法。我们自己做的话需要两个步骤。首先是把cache的头部元素值另存起来。然后，使用切片表达式对cache重新赋值。目的是把头部元素值从这个切片值中删除掉。虽然这很简单，但是我们还是希望它能像一个原子操作那样被执行。因此，在这里我们依然使用到了代表互斥锁的mutex字段。

请求缓存的capacity方法和length方法都非常好实现，调用相应的内建函数就可以了。代码如下：

```

func (rcache *reqCacheBySlice) capacity() int {
    return cap(rcache.cache)
}

func (rcache *reqCacheBySlice) length() int {
    return len(rcache.cache)
}

```

在请求缓存的close方法中，我们还是会先检查status字段的值。很显然，如果请求缓存已被关闭了，我们就不用再去关闭它了。这里的“关闭”其实就是把1赋给status字段而已。因此，close方法的方法体中只包含了两条语句：

```

if rcache.status == 1 {
    return
}
rcache.status = 1

```

这两条语句并不需要被执行于锁的保护之下。原因有两个。其一，status字段是byte类型的。对它的值的简单读取或写入都可以仅由一条CPU指令来完成。其二，我们对“一旦请求缓存被关闭，就立刻停止一切请求放入和获取操作”的意愿并没有那么强烈。因为在调度器的各个方法的实现中，已经存在很多对此进行保障的代码了。

最后，我们要实现请求缓存的summary方法。请求缓存的结构简单，需要放入到摘要信息中的内容也并不多。经过汇总，只有3个内部状态需要被放入。第一个状态肯定是请求缓存本身的状态，即status字段的值。而后两个状态与cache字段的值有关，它们分别是该值的长度和容量。因此，我们将下面这个变量的值作为摘要信息的模板：

```

// 摘要信息模板。
var summaryTemplate = "status: %s, " + "length: %d, " + "capacity: %d"

```

有了这个模板，我们在summary方法中只需要做完这道填空题并返回结果即可。summary方法的完整声明如下：

```
func (rcache *reqCacheBySlice) summary() string {
    summary := fmt.Sprintf(summaryTemplate,
        statusMap[rcache.status],
        rcache.length(),
        rcache.capacity())
    return summary
}
```

如上所示，我们通过调用fmt代码包的Sprintf函数做好了填空题并得到了一个结果值，然后再把这个结果值返回给summary方法的调用方。在这个过程中，我们用到了一个字典。这个字典由变量statusMap代表。其声明为：

```
// 状态字典。
var statusMap = map[byte]string{
    0: "running",
    1: "closed",
}
```

声明这个变量的目的很明确，就是为请求缓存的状态值提供可读性较好的描述。如此一来，请求缓存的摘要信息也会更加易读。

以上就是关于请求缓存的全部内容。由于我们只是把它作为了调度器的内部实现的一部分，所以它看起来会稍显简陋一些。不过它已经够用了。

9.7.4 摘要信息的类型

调度器摘要信息的接口类型是SchedSummary。该类型共包含了3个方法的声明。这些方法是String、Detail和Same。前两个方法都被用来提供调度器摘要信息，而第三个方法则被用于判断两份调度器摘要信息是否相同。

我们把接口类型SchedSummary的实现类型命名为mySchedSummary。由于我们要在调度器的摘要信息中囊括其涉及的所有组件的状态，所以mySchedSummary的字段会非常多。

首先是代表摘要信息前缀的字段prefix。其声明如下：

```
prefix          string // 前缀。
```

该字段的存在意义是为每行调度器摘要信息提供统一的前导符。例如，如果prefix字段的值为">>> "，那么由mySchedSummary类型的String方法和Detail方法返回的摘要信息就都应该类似于：

```
>>> Running: false
>>> Pool Size: 2
>>> Channel length: 4
```

其次，我们应该用一些字段来存储调度器本身的状态。比如，下面这几个字段就是如此：

```
running          uint32 // 运行标记。
poolSize         uint32 // 池大小。
channellen       uint   // 通道总长度（或称容量）。
crawlDepth       uint32 // 爬取的最大深度。
```

这些字段的名称和类型都与作为调度器实现的myScheduler类型中的相应字段一模一样。其中，后3个字段还分别与调度器的Start方法的某个参数相匹配。它们会真实地记录下使用方对网络爬虫框架的定制参数。

除此之外，调度器实现类型myScheduler中还包含了很多复合类型的字段。在这些字段的类型中，有一些本身就包含了摘要信息获取方法。比如，通道管理器、请求缓存、条目处理管道等类型。摘要信息类型mySchedSummary无需持有这些调度器字段的值，而只需获取并存储它们的摘要信息。因此，mySchedSummary类型的声明中就有了这些字段：

```
chanmanSummary    string // 通道管理器的摘要信息。
reqCacheSummary   string // 请求缓存的摘要信息。
itemPipelineSummary string // 条目处理管道的摘要信息。
stopSignSummary   string // 停止信号的摘要信息。
```

有的调度器字段的类型虽然不提供摘要信息获取方法，但是我们可以通过调用它们的其他方法来获取到非常有价值的状态信息。这样的调度器字段有两个。它们分别代表了网页下载器池和分析器池。与之相对应的mySchedSummary类型的字段如下：

```
dlPoolLen         uint32 // 网页下载器池的长度。
dlPoolCap         uint32 // 网页下载器池的容量。
analyzerPoolLen   uint32 // 分析器池的长度。
analyzerPoolCap   uint32 // 分析器池的容量。
```

最后，mySchedSummary类型还包含了两个字段。它们的声明如下：

```
urlCount          int    // 已请求的URL的计数。
urlDetail          string // 已请求的URL的详细信息。
```

这两个字段的值可以由myScheduler类型的字段urlMap的值转化而来。添加这两个字段旨在向调度器的摘要信息中加入一些额外的统计数据。

可以看到，调度器摘要信息类型myScheduler的绝大多数字段都是以调度器类型myScheduler中的某个字段为依托的。实际上也本该如此。我们编写调度器摘要信息类型的目的，就是在尽可能封装调度器及其协调或持有的各个组件的状态的前提下，为外界提供简单易用的展现接口。

我们把创建和初始化调度器摘要信息类型值的任务交给了函数NewSchedSummary。这个函数的声明如下：

```
// 创建调度器摘要信息。
func NewSchedSummary(sched *myScheduler, prefix string) SchedSummary
```

可以看到，NewSchedSummary函数接受两个参数。对于它的第二个参数prefix，我们无需再做解释。这里重点关注它的第一个参数sched。该参数即代表了调度器。注意，其类型是*myScheduler类型。之所以不使用它实现的那个接口类型Scheduler，主要是为了在真正使用这个参数的值时省去一步类型转换的操作。由于在该函数的实现中肯定会用到具体的调度器类型（如*myScheduler类型）中的很多字段，所以它也很难被通用化。因此，把sched参数的类型设置为Scheduler也没有什么实际意义。这样反倒会使NewSchedSummary函数的实现变得复杂很多。如果我们以后有了更多调度器的实现类型，那么我们肯定也需要编写相应的调度器摘要信息创建函

数，甚至是一个新的调度器摘要信息的实现类型。也就是说，调度器的实现类型与调度器摘要信息创建函数（甚至是调度器摘要信息的实现类型）之间是一一对应且耦合在一起的。我们目前可以接受这样的耦合。

我们实现NewSchedSummary函数的第一步仍然是参数值检查。因为Go语言的string类型值不会为nil，所以我们在这里只需检查参数sched的值。如果该值为nil，那么我们就直接返回nil。否则，我们就继而开始准备需要赋给mySchedSummary类型的各个字段的值。

实际上，需要我们特别准备值的字段仅有urlCount和urlDetail。其他字段的值只需调用调度器的相应字段的某个方法就可得到。为了生成urlCount字段和urlDetail字段的值，我们需要先调用内建方法len来获得调度器的urlMap字段值的长度。如果该长度为0，那么代表已请求URL计数的字段urlCount的值也将会是0，而urlDetail字段的值将会是"\n"。该值中的换行符完全是为了满足摘要信息排版的需要。若该长度不为0，我们就可以把这个长度直接作为urlCount字段的值，并通过对调度器的urlMap字段值的迭代来拼接出urlDetail字段的值。请看下面的代码：

```
urlCount := len(sched.urlMap)
var urlDetail string
if urlCount > 0 {
    var buffer bytes.Buffer
    buffer.WriteByte('\n')
    for k, _ := range sched.urlMap {
        buffer.WriteString(prefix)
        buffer.WriteString(prefix)
        buffer.WriteString(k)
        buffer.WriteByte('\n')
    }
    urlDetail = buffer.String()
} else {
    urlDetail = "\n"
}
```

可以看到，我们使用标准库代码包bytes中的Buffer类型的值的相应方法来拼接已请求URL的详细信息。这是一个被推荐的用法。当我们要进行大量的字符串拼接操作的时候，总是应该避免使用操作符+来完成，而要使用bytes.Buffer类型值。bytes.Buffer类型值拥有众多的方法可以让我们灵活地读取其中的内容或追加新的内容。此外，该类值还可以把其中的内容输出为字节切片值或字符串。像前面示例中的buffer变量的String方法，就会返回其已存内容的字符串形式。

在完成了上述准备之后，我们就可以创建并初始化一个*mySchedSummary类型的值，并将其作为NewSchedSummary函数的结果值了。代码如下：

```
return &mySchedSummary{
    prefix:      prefix,
    running:     sched.running,
    poolSize:    sched.poolSize,
    channellen:  sched.channellen,
    crawlDepth: sched.crawlDepth,
    chanmanSummary: sched.chanman.Summary(),
```

```

reqCacheSummary:    sched.reqCache.summary(),
dlPoolLen:          sched.dlpool.Used(),
dlPoolCap:          sched.dlpool.Total(),
analyzerPoolLen:    sched.analyzerPool.Used(),
analyzerPoolCap:    sched.analyzerPool.Total(),
itemPipelineSummary: sched.itemPipeline.Summary(),
urlCount:           urlCount,
urlDetail:          urlDetail,
stopSignSummary:    sched.stopSign.Summary(),
}

```

我们已经讲述过上面这个复合字面量中包含的所有的调度器字段以及相关方法。在此，我们就不再一一解释了。在使用复合字面量构建出一个mySchedSummary类型值之后，我们又用地址操作符&得到了指向它的那个指针类型（即*mySchedSummary类型）的值并将其作为结果值返回给了NewSchedSummary函数的调用方。

读者也许已经猜到，我们接下来就要为*mySchedSummary类型添加SchedSummary接口类型中声明的那些方法的实现了。只有这样才能使前者成为后者的实现类型，也才能使NewSchedSummary函数通过编译。

类型mySchedSummary的指针方法String和Detail的实现应该是相似的。这两者的唯一不同仅在于其返回的调度器摘要信息的详细程度。因此，我们应该把可以被它们共用的部分放到一个单独的方法中。我们把这个方法命名为getSummary。它的声明如下：

```

// 获取摘要信息。
func (ss *mySchedSummary) getSummary(detail bool) string

```

可以看到，该方法接受一个bool类型的参数detail。这个参数的值决定了该方法是否会返回较为详细的调度器摘要信息。

与网络爬虫框架中的其他组件的摘要信息生成方式相同，我们首先要确定调度器摘要信息的模板。这个模板是这样的：

```

template := prefix + "Running: %v \n" +
    prefix + "Pool Size: %d \n" +
    prefix + "Channel length: %d \n" +
    prefix + "Crawl depth: %d \n" +
    prefix + "Channels manager: %s \n" +
    prefix + "Request cache: %s\n" +
    prefix + "Downloader pool: %d/%d\n" +
    prefix + "Analyzer pool: %d/d\n" +
    prefix + "Item pipeline: %s\n" +
    prefix + "Urls(%d): %s" +
    prefix + "Stop sign: %s\n"

```

读者如果仔细看的话，会发现该模板中囊括了调度器实现类型myScheduler中的全部基本状态和所有组件的关键信息。在后面，我们将会用到mySchedSummary类型的所有字段的值来给这个模板填空。在该模板中，%d/%d样的内容中的两个%d分别代表的是相应组件的长度和容量。在它们被真实的值取代之后，我们就可以清楚地了解到对应组件的使用情况了。

给模板填空并生成摘要信息的方式是调用fmt.Sprintf函数。这个调用表达式是比较长的：

```

fmt.Sprintf(template,
    func() bool {
        return ss.running == 1
    }(),
    ss.poolSize,
    ss.channellen,
    ss.crawlDepth,
    ss.chanmanSummary,
    ss.reqCacheSummary,
    ss.dlPoolLen, ss.dlPoolCap,
    ss.analyzerPoolLen, ss.analyzerPoolCap,
    ss.itemPipelineSummary,
    ss.urlCount,
    func() string {
        if detail {
            return ss.urlDetail
        } else {
            return "<concealed>\n"
        }
    }(),
    ss.stopSignSummary)

```

除了代表调度器摘要信息模板的`template`变量之外,我们还为`fmt.Sprintf`函数提供了14个参数值。其中比较显眼的是紧随在`template`之后的那个参数值和倒数第二个参数值。我们提供它们的方式都不是使用具体的字段或变量,而是使用针对匿名函数的调用表达式。也就是说,对这两个匿名函数的调用的结果值会分别被视为相应的参数值。在这条对`fmt.Sprintf`函数的调用表达式被求值之前,我们传给它的那些参数会先被求值。对于那两个匿名函数调用表达式来说,也是如此。

第一个匿名函数会返回一个`bool`类型的值,这显然比数值类型的参数值更具有可读性。阅读调度器摘要信息的人一眼就能看出调度器是否正在运行,而不用我们做任何解释。编写第二个匿名函数主要是为了可以根据参数`detail`的值而改变摘要信息中的部分内容。当`detail`的值为`false`时,我们就不会加入已请求URL的详细信息,并转而使用"`<concealed>\n`"代替以表示这部分已被隐藏。

函数`fmt.Sprintf`并不会把它根据模板和后续参数值生成的字符串打印到标准输出,而会把它作为结果值返回给它的调用方。这就使我们能够很方便地获得其内容,并把它作为`getSummary`方法的结果值返回了。

有了`getSummary`方法,我们实现`String`方法和`Detail`方法就轻而易举了。这两个方法的完整声明如下:

```

func (ss *mySchedSummary) String() string {
    return ss.getSummary(false)
}

func (ss *mySchedSummary) Detail() string {
    return ss.getSummary(true)
}

```

除了这两个方法之外，`SchedSummary`接口类型还包含了一个很有特色的方法`Same`。该方法的实现应该可以判断出另一个`SchedSummary`类型值与其所属的值的异同。`*mySchedSummary`类型的该方法的声明如下：

```
func (ss *mySchedSummary) Same(other SchedSummary) bool
```

在该方法的实现中，我们应该先判断参数`other`的值是否为`nil`。因为该方法所属的值肯定不是`nil`，所以它与值为`nil`的参数`other`肯定是不相同的。其次，由于该方法属于`*mySchedSummary`类型，所以若`other`的值的类型不是`*mySchedSummary`，则二者肯定不相同。这两项检查的代码如下：

```
if other == nil {
    return false
}
otherSs, ok := interface{}(other).(*mySchedSummary)
if !ok {
    return false
}
```

只要在这两项检查都通过的情况下，我们才有必要开始进一步的比较。进一步比较的代码如下：

```
if ss.running != otherSs.running ||
    ss.poolSize != otherSs.poolSize ||
    ss.channellen != otherSs.channellen ||
    ss.crawlDepth != otherSs.crawlDepth ||
    ss.dlPoolLen != otherSs.dlPoolLen ||
    ss.dlPoolCap != otherSs.dlPoolCap ||
    ss.analyzerPoolLen != otherSs.analyzerPoolLen ||
    ss.analyzerPoolCap != otherSs.analyzerPoolCap ||
    ss.urlCount != otherSs.urlCount ||
    ss.stopSignSummary != otherSs.stopSignSummary ||
    ss.reqCacheSummary != otherSs.reqCacheSummary ||
    ss.itemPipelineSummary != otherSs.itemPipelineSummary ||
    ss.chanmanSummary != otherSs.chanmanSummary {
    return false
} else {
    return true
}
```

可以看到，除了字段`prefix`和`urlDetail`之外，其他`mySchedSummary`类型值的字段都参与了比较。只要其中的一对字段的值不相等，我们就可以断定这两份调度器摘要信息是不相同的。请注意我们比较的顺序。我们先比较相对容易比较的字段，而把对`string`类型的字段值的比较放在了最后面。并且，我们还根据这3个`string`类型值的长度将相应的表达式也排了序。读者可以想一想为什么要这样做。

好了，在实现了这3个公开的方法之后，我们就完成了对`SchedSummary`接口类型的实现类型`*mySchedSummary`的编写。在下一节，我们就会看到使用该类型的值及其方法的具体方式。同时，随着这一部分的完成，我们也完整地实现了网络爬虫框架的调度器以及所有的组件。我们稍后就来演示怎样使用它们。

9.8 一个使用演示

在本节，我们的主要任务是为网络爬虫框架编写一个使用示例，并演示它如何爬取一个网站上的网页。在这个过程中，我们会发现网络爬虫框架的一些不足，并继续为之添枝加叶。这是一种反哺。我们在软件开发的过程中，应该总是积极地为程序编写使用示例（测试程序也可以被视为使用示例，而且能达到一举多得的效果），并以此来检查和验证我们的程序。

下面，我们就从为调度器准备参数值开始。

9.8.1 再看调度器参数

我们的网络爬虫框架可以接受定制。实际上，我们只有对它进行定制之后，才能让它真正地跑起来。这也是把它叫作框架的原因之一。这里所说的定制完全体现在了调度器的Start方法的参数声明列表上。

1. 参数概览

回顾Start方法，我们需要传递给调度器如下参数。

- ❑ `channelLen`: 通道的长度。也就是调度器中的通道管理器所持的所有通道的长度。它们是统一的，并都经由此参数的值指定。
- ❑ `poolSize`: 池的尺寸。也就是调度器中的网页下载器池和分析器池的容量。目前，它们的容量也是需要被统一给定的。
- ❑ `crawlDepth`: 被爬取的网页的最大深度。此参数的值界定了一个范围，从而明确了一个作为目标的网页的有限集合。
- ❑ `httpClientGenerator`: HTTP客户端生成器。它的值应该是一个函数。它允许我们对网页下载器所使用的HTTP客户端进行定制。
- ❑ `respParsers`: 响应解析函数的序列。分析器会通过调用这些函数来对响应进行分析，并得到作为结果的若干请求和条目。它们是响应分析流程中非常重要的组成部分。
- ❑ `itemProcessors`: 条目处理器的序列。每个条目处理器也都是由一个函数代表的。一个条目会依次被这些条目处理器处理。
- ❑ `firstHttpRequest`: 首次HTTP请求。也就是需要第一个被爬取的网页所对应的HTTP请求。

为前3个参数准备值是很简单的事情，因为它们都是数值类型的。然而，它们的值却左右了网络爬虫框架的爬取效率和时间。

通道的长短可以决定网络爬虫框架中各个处理模块间的协作质量。比如，我们在前面说过，网页下载器下载网页的速度通常会比分析器分析响应的速度慢很多。虽然这个问题已经通过加入请求缓存而基本得到了解决，但是我不应该过分依赖请求缓存，因为被缓存的请求也是需要消耗内存空间的。巨大的请求缓存会给计算机的内存造成很大压力，同时也会直接影响到放入请求的操作的平均速度（这涉及append函数的内部机理，读者可以顺便想一想获得请求操作的速度为什么不会受到直接影响）。如果想要使请求缓存的长度维持在一个合理的范围内，那么我们势必要加大请求通道的长度。但是，过长的通道的存在也是对内存空间的一种浪费。不过这总要

比通道过短带来的问题小得多。相似地，如果条目处理管道完整的处理一个条目的速度比分析器向条目通道发送条目的速度慢太多，那么肯定会使相关Goroutine数量的增长曲线持续陡峭（请参看webcrawler/scheduler代码包中的openItemPipeline函数的实现），甚至最后会给整个程序和计算机的运行带来麻烦。综上所述，我们总是应该仔细地考量并设置这些通道的长度。

对于池的尺寸来说，道理是类似的。过小的池会大大增加数据等待被处理的时间，并会使容纳此类数据的通道时不时地甚至持续地处于满载状态。它所造成的问题最后又基本会归结于同一处。因此，这两方面的参数的设置是有强关联的。我们应该综合地去考虑它们。

2. 参数调整

说到这里，读者可能会想到另外一个问题：我们可不可以更加细致地设置各个通道的长度和各个池的尺寸呢？确实，经过上述分析之后，我们会发觉当前的设置方法显得有些死板了。因为网络爬虫框架中的各个处理环节的情况都不尽相同。例如，网页下载器、分析器和条目处理管道的运行效率有所差别，并且它们大都会受到本身就不确定的外界因素的影响。我们在真正运行程序之前，很难预估它们之间在运行效率上的差异，以及这种差异是否有规律可循。因此，提供更加灵活的配置方式以满足不确定的定制和运行环境是很有必要的。

现在，对调度器以及通道管理器的改造几乎是必须要做的事情了。既然我们要对传给它们的参数进行进一步细化，那么就应该顺带建立一个类型体系。这个类型体系可以让我们在细化参数的同时保持相关接口的简洁。

我们把这个类型体系中被用于定义公共行为的接口类型命名为Args。它的声明如下：

```
// 参数容器的接口。
type Args interface {
    // 自检参数的有效性，并在必要时返回可以说明问题的错误值。
    // 若结果值为nil，则说明未发现问题，否则就意味着自检未通过。
    Check() error
    // 获得参数容器的字符串表现形式。
    String() string
}
```

我们根据上述定义分别针对各个通道和各个池的参数声明了两个类型——ChannelArgs和PoolBaseArgs。它们的基本结构如下：

```
// 通道参数的容器。
type ChannelArgs struct {
    reqChanLen  uint // 请求通道的长度。
    respChanLen uint // 响应通道的长度。
    itemChanLen uint // 条目通道的长度。
    errorChanLen uint // 错误通道的长度。
    description string // 描述。
}

// 池基本参数的容器。
type PoolBaseArgs struct {
    pageDownloaderPoolSize uint32 // 网页下载器池的尺寸。
    analyzerPoolSize       uint32 // 分析器池的尺寸。
    description             string // 描述。
}
```

这两个类型对应的指针类型都应该是Args接口类型的实现类型。为此，我们应该分别为它们实现Check和String这两个指针方法。另外，由于这两个结构体类型的字段都是包级私有的，所以为了方便地对它们进行初始化，我们还要声明两个函数。这两个函数分别被命名为NewChannelArgs和NewPoolBaseArgs。它们的唯一结果的类型应该分别是ChannelArgs和PoolBaseArgs。

除此之外，我们应该把ChannelArgs和PoolBaseArgs设计为值不可变的类型。也就是说，这两个类型的值都应该是只读的。一旦它们被创建并初始化，它们中的字段的值就不能被改变了。因此，我们应该仅为ChannelArgs类型再添加如下方法：

```
// 获得请求通道的长度。
func (args *ChannelArgs) ReqChanLen() uint

// 获得响应通道的长度。
func (args *ChannelArgs) RespChanLen() uint

// 获得条目通道的长度。
func (args *ChannelArgs) ItemChanLen() uint

// 获得错误通道的长度。
func (args *ChannelArgs) ErrorChanLen() uint
```

同时，仅为PoolBaseArgs类型再添加这两个方法：

```
// 获得网页下载器池的尺寸。
func (args *PoolBaseArgs) PageDownloaderPoolSize() uint32

// 获得分析器池的尺寸。
func (args *PoolBaseArgs) AnalyzerPoolSize() uint32
```

在定义好它们的行为、基本结构和一些约束之后，我们就可以编码去实现它们了。具体的实现过程我们就不在这里描述了。读者肯定可以自己搞定。

对上述类型的完整实现都放置在了goc2p项目的webcrawler/base代码包中。

在编写完ChannelArgs类型和PoolBaseArgs类型以及相关的函数和方法之后，我们就可以开始修改通道管理器和调度器的相关代码了。

对于通道管理器来说，我们应该先把接口类型ChannelManager的Init方法的声明由

```
// 初始化通道管理器。
// 参数channellen代表通道管理器中的各类通道的初始长度。
// 参数reset指明是否重新初始化通道管理器。
Init(channellen uint, reset bool) bool
```

变更为

```
// 初始化通道管理器。
// 参数channelArgs代表通道参数的容器。
// 参数reset指明是否重新初始化通道管理器。
Init(channelArgs base.ChannelArgs, reset bool) bool
```

紧接着把它的实现类型myChannelManager的字段

```
channellen uint // 通道的长度值。
```

改为

```
channelArgs base.ChannelArgs    // 通道参数的容器。
```

然后，把myChannelManager类型的指针方法Init的实现修改为：

```
func (chanman *myChannelManager) Init(channelArgs base.ChannelArgs, reset bool) bool {
    if err := channelArgs.Check(); err != nil {
        panic(err)
    }
    // 省略若干条语句
    chanman.channelArgs = channelArgs
    chanman.reqCh = make(chan base.Request, channelArgs.ReqChanLen())
    chanman.respCh = make(chan base.Response, channelArgs.RespChanLen())
    chanman.itemCh = make(chan base.Item, channelArgs.ItemChanLen())
    chanman.errorCh = make(chan error, channelArgs.ErrorChanLen())
    // 省略若干条语句
}
```

并把NewChannelManager函数的实现修改为：

```
// 创建通道管理器。
func NewChannelManager(channelArgs base.ChannelArgs) ChannelManager {
    chanman := &myChannelManager{}
    chanman.Init(channelArgs, true)
    return chanman
}
```

最后，由于通道管理器的ChannelLen方法已经不再适用于当前的情况，并且它的作用是可以被替代的，所以我们索性把它从ChannelManager接口类型及其实现类型中删除。

到这里，我们对通道管理器的修改就完成了。对于调度器，我们同样需要从变更接口开始。接口类型Scheduler的Start方法的原有声明

```
// 开启调度器。
// 调用该方法会使调度器创建和初始化各个组件。在此之后，调度器会激活爬取流程的执行。
// 参数channellen被用来指定数据传输通道的长度。
// 参数poolSize被用来设定网页下载器池和分析器池的容量。
// 参数crawlDepth代表了需要被爬取的网页的最大深度值。深度大于此值的网页会被忽略。
// 参数httpClientGenerator代表的是被用来生成HTTP客户端的函数。
// 参数respParsers的值应为分析器所需的被用来解析HTTP响应的函数的序列。
// 参数itemProcessors的值应为需要被置入条目处理管道中的条目处理器的序列。
// 参数firstHttpRequest即代表首次请求。调度器会以此为起始点开始执行爬取流程。
Start(channellen uint,
    poolSize uint32,
    crawlDepth uint32,
    httpClientGenerator GenHttpClient,
    respParsers []anlz.ParseResponse,
    itemProcessors []ipl.ProcessItem,
    firstHttpRequest *http.Request) (err error)
```

应该被变更为

```
// 开启调度器。
// 调用该方法会使调度器创建和初始化各个组件。在此之后，调度器会激活爬取流程的执行。
// 参数channelArgs代表通道参数的容器。
```

```

// 参数poolBaseArgs代表池基本参数的容器。
// 参数crawlDepth代表了需要被爬取的网页的最大深度值。深度大于此值的网页会被忽略。
// 参数httpClientGenerator代表的是被用来生成HTTP客户端的函数。
// 参数respParsers的值应为分析器所需的被用来解析HTTP响应的函数的序列。
// 参数itemProcessors的值应为需要被置入条目处理管道中的条目处理器的序列。
// 参数firstHttpRequest即代表首次请求。调度器会以此为起始点开始执行爬取流程。
Start(channelArgs base.ChannelArgs,
      poolBaseArgs base.PoolBaseArgs,
      crawlDepth uint32,
      httpClientGenerator GenHttpClient,
      respParsers []anlz.ParseResponse,
      itemProcessors []ipl.ProcessItem,
      firstHttpRequest *http.Request) (err error)

```

相应地，myScheduler类型的字段

```

poolSize      uint32           // 池的尺寸。
channellen    uint             // 通道的总长度（也即容量）。

```

应该被删除，并由字段

```

channelArgs   base.ChannelArgs // 通道参数的容器。
poolBaseArgs  base.PoolBaseArgs // 池基本参数的容器。

```

代替。还需要把它的Start方法的实现修改为：

```

func (sched *myScheduler) Start(
    channelArgs base.ChannelArgs,
    poolBaseArgs base.PoolBaseArgs,
    crawlDepth uint32,
    httpClientGenerator GenHttpClient,
    respParsers []anlz.ParseResponse,
    itemProcessors []ipl.ProcessItem,
    firstHttpRequest *http.Request) (err error) {
    // 省略若干条语句
    if err := channelArgs.Check(); err != nil {
        return err
    }
    sched.channelArgs = channelArgs
    if err := poolBaseArgs.Check(); err != nil {
        return err
    }
    sched.poolBaseArgs = poolBaseArgs
    // 省略若干条语句
    sched.chanman = generateChannelManager(sched.channelArgs)
    // 省略若干条语句
    dlpool, err :=
        generatePageDownloaderPool(
            sched.poolBaseArgs.PageDownloaderPoolSize(),
            httpClientGenerator)
    // 省略若干条语句
    analyzerPool, err := generateAnalyzerPool(sched.poolBaseArgs.AnalyzerPoolSize())
    // 省略若干条语句
}

```

请注意，其中曾让读者自行实现的generateChannelManager函数也应该做出相应的改变。

至此，调度器就修改完成了。不过，由于NewSchedSummary函数以及mySchedSummary类型的实现紧密贴合了刚刚改造完的myScheduler类型的内部状态，所以我们不得不也对它们进行修改。

类型mySchedSummary原有的字段

```
poolSize      uint32 // 池大小。
channellLen   uint   // 通道总长度（或称容量）。
```

应该被新的字段

```
channelArgs   base.ChannelArgs // 通道参数的容器。
poolBaseArgs  base.PoolBaseArgs // 池基本参数的容器。
```

替换掉。这与对myScheduler类型的基本结构所做的修改是相同的。如此一来，mySchedSummary类型的相关方法以及NewSchedSummary函数的实现肯定也要随之改变。不过这种变动是非常小的。其中，新的调度器摘要信息的模板为：

```
template := prefix + "Running: %v \n" +
    prefix + "Channel args: %s \n" +
    prefix + "Pool base args: %s \n" +
    prefix + "Crawl depth: %d \n" +
    prefix + "Channels manager: %s \n" +
    prefix + "Request cache: %s \n" +
    prefix + "Downloader pool: %d/%d \n" +
    prefix + "Analyzer pool: %d/%d \n" +
    prefix + "Item pipeline: %s \n" +
    prefix + "Urls(%d): %s" +
    prefix + "Stop sign: %s \n"
```

正如前文所述，我们对网络爬虫框架的启动参数的调整引起了一系列的修改。这涉及了通道管理器的接口及实现类型、调度器的接口及实现类型，以及调度器摘要信息的实现类型。不过这样做是值得的。现在我们可以通过对启动参数的设置对网络爬虫框架进行更加精细的定制了。不过要注意，如果我们已经发布了这款软件的正式版本（相对于开发版本和测试版本而言），那么这样直接修改其中的接口可不太妥当。这甚至可以说是不合规矩的。当遇到这种情况的时候，我们应该新增一些可以满足我们的新需求的接口方法，而绝不应该去修改那些已经存在的接口方法。在接口设计方面，我们同样应该做到“对修改关闭，对扩展开放”。

不论怎样，我们调整了网络爬虫框架的启动参数以及相关的接口和实现，这还算及时。在下一小节，我们就开始准备使用示例所需的参数值。

9.8.2 开启调度器

现在，一切准备就绪。让我们准备必要的参数，然后开启调度器。

关于这些参数，我们描述得已经够多了。现在直接上代码：

```
// 准备启动参数
channelArgs := base.NewChannelArgs(10, 10, 10, 10)
poolBaseArgs := base.NewPoolBaseArgs(3, 3)
crawlDepth := uint32(1)
httpClientGenerator := genHttpClient
```

```

respParsers := getResponseParsers()
itemProcessors := getItemProcessors()
startUrl := "http://www.sogou.com"
firstHttpReq, err := http.NewRequest("GET", startUrl, nil)
if err != nil {
    logger.Errorln(err)
    return
}

```

在上面这段代码中，我们逐一地生成了调度器的Start方法的7个参数值。我们把那4个通道的总长度都设为了10，而把网页下载器池和分析器池的尺寸都设为了3。最大爬取深度为1意味着我们只爬取首次请求对应的网页以及其中直接链接的那些网页。HTTP客户端生成器由一个名为httpClientGenerator的函数代表。而响应解析函数的序列和条目处理器的序列则分别由getResponseParsers函数和getItemProcessors函数给出。最后，我们首次爬取的目标被确定为搜狗的主页（搜狗的同学请见谅）。

另外，示例中的标识符logger仍然代表日志记录器。其声明是这样的：

```

// 日志记录器。
var logger logging.Logger = logging.NewSimpleLogger()

```

通过它记录的日志只会打印到标准输出上。

现在，我们把焦点聚集在HTTP客户端生成器、响应解析函数序列以及条目处理器序列的生成过程上。最简单的HTTP客户端生成器的实现就是直接使用复合字面量生成一个HTTP客户端的实例并返回，如下所示：

```

// 生成HTTP客户端。
func genHttpClient() *http.Client {
    return &http.Client{}
}

```

如此的HTTP客户端生成函数即是可用的。至于怎样对HTTP客户端进一步定制，我们到后面再说。

负责生成响应解析函数序列的函数getResponseParsers是值得一说的。因为在Go语言的标准库中并没有提供一套可以解析HTML文档的API。而HTML文档则可能是HTTP响应中的主体内容的表现形式。在这里，我们使用了一个名为goquery第三方代码包。该代码包的完整路径为github.com/PuerkitoBio/goquery。它是在公共git仓库Github上的一个开源的代码包。我们通过go get命令就可以从网络上下载并安装它：

```

go get github.com/PuerkitoBio/goquery

```

我们要使用这个代码包中的API来实现一个符合webcrawler/analyzer代码包中的函数类型ParseResponse的声明的函数。该函数的声明如下：

```

// 响应解析函数。只解析“A”标签。
func parseForATag(httpResp *http.Response, respDepth uint32) ([]base.Data, []error)

```

从其名称上可以看出，它只会寻找HTTP响应中HTML文档中的“A”标签，并提取其中的链接地址。这里所说的“A”标签是HTML元素中的一种。它是在某一个HTML文档中包含另一个

HTML文档的链接的重要方法。通常，该标签的“href”属性中会包含一个网络地址（即另一个网页的地址）。请读者参看HTML规范的相关文献，并以此获取关于“A”标签的详细描述。

这里无意对上述第三方代码包做过多说明。不过，为了实现parseForATag函数，我们还是在這裡给出一个简单的示例。在这之前，我们要在这个函数的函数体中加入几段代码。第一段代码如下：

```
// TODO 支持更多的HTTP响应状态
if httpResp.StatusCode != 200 {
    err := errors.New(
        fmt.Sprintf("Unsupported status code %d. (httpResponse=%v)", httpResp))
    return nil, []error{err}
}
```

这段代码的含义是，parseForATag函数只对响应状态为200的响应主体进行解析。否则，就直接返回并携带相应的错误值。这样做的目的主要是保证我们能够对一个可用的HTML进行解析。这基本上只是出于编写简单且易于理解的示例的需要。

下面是第二段代码：

```
var reqUrl *url.URL = httpResp.Request.URL
var httpRespBody io.ReadCloser = httpResp.Body
defer func() {
    if httpRespBody != nil {
        httpRespBody.Close()
    }
}()
```

我们获取与此HTTP响应对应的请求的URL并将其赋予reqUrl，这主要是为了满足可能发生的对相对的网络地址进行转换的需要。而httpRespBody变量指代的就是HTTP响应的主体。这个主体中很可能会包含一个完整的HTML文档。另外，defer语句的功用是保证在parseForATag函数执行结束之前关闭HTTP响应主体。注意，因为我们只想在响应解析函数序列中加入一个响应解析函数，所以才可以做。否则，我们只应该在序列的最后一个函数中关闭HTTP响应主体。

真正解析HTTP响应主体前的最后一段的代码是：

```
dataList := make([]base.Data, 0)
errs := make([]error, 0)
```

无需多说，这两个变量的值将会被作为parseForATag函数的结果值返回。

使用goquery代码包解析HTTP响应主体的第一步就是调用它的NewDocumentFromReader函数。代码如下：

```
// 开始解析
doc, err := goquery.NewDocumentFromReader(httpRespBody)
if err != nil {
    errs = append(errs, err)
    return dataList, errs
}
```

函数goquery.NewDocumentFromReader会返回两个结果值。第一个结果值是代表了HTML文档

的`*goquery.Document`类型值。第二个结果值则是一个错误值。如果在HTTP响应主体中的并不是一个可用的HTML文档，那么第二个结果值就会是非`nil`的。这时，我们应该直接将该错误值追加到`errs`的值中，并返回`dataList`和`errs`。

如果`err`的值为`nil`，那么就说明对这个HTTP响应主体中的HTML文档的初步解析成功。剩下的工作就是找到“A”标签并依此生成的新的请求和条目。这部分工作由下面这一大段代码承担：

```
// 查找“A”标签并提取链接地址
doc.Find("a").Each(func(index int, sel *goquery.Selection) {
    href, exists := sel.Attr("href")
    // 前期过滤
    if !exists || href == "" || href == "#" || href == "/" {
        return
    }
    href = strings.TrimSpace(href)
    lowerHref := strings.ToLower(href)
    // 暂不支持对JavaScript代码的解析。
    if href != "" && !strings.HasPrefix(lowerHref, "javascript") {
        aUrl, err := url.Parse(href)
        if err != nil {
            errs = append(errs, err)
            return
        }
        if !aUrl.IsAbs() {
            aUrl = reqUrl.ResolveReference(aUrl)
        }
        httpReq, err := http.NewRequest("GET", aUrl.String(), nil)
        if err != nil {
            errs = append(errs, err)
        } else {
            req := base.NewRequest(httpReq, respDepth)
            dataList = append(dataList, req)
        }
    }
    text := strings.TrimSpace(sel.Text())
    if text != "" {
        imap := make(map[string]interface{})
        imap["a.text"] = text
        imap["parent_url"] = reqUrl
        item := base.Item(imap)
        dataList = append(dataList, &item)
    }
})
return dataList, errs
```

我们就此做一下简单的解释。调用表达式`doc.Find("a")`的含义是在HTML文档中查找所有的“A”标签。`Find`方法会返回一个`*goquery.Selection`类型的值。该类型的`Each`方法会接受一个符合`func(index int, sel *goquery.Selection)`的函数值。其含意是，对于每一个被找到的“A”标签都应用该函数。其中，参数`sel`的值表示的是代表了与被找到的“A”标签对应的`*goquery.Selection`值，而参数`index`的值则代表了当前“A”标签是第几个被找到的。

在这个匿名函数中，我们首先通过调用`sel`的`Attr`方法得到了这个“A”标签中的“href”属性的值。然后，在经过一番过滤之后，我们在保证这个“href”属性的值为有效的URL且代表了绝对的网络地址的前提下，生成新的HTTP请求并将其追加到`dataList`。而与这个“A”标签相关的其他信息，则被用来生成条目并同样被追加到`dataList`中。当然，如果在这个过程中发生了错误，我们将会忽略后续的步骤并确保相应的错误值被返回。

如果以上这些工作都成功完成，我们依然会返回`dataList`和`errs`。

以上就是`parseForATag`函数中的全部代码。有了它，我们就可以顺利地实现`getResponseParsers`函数了。它的完整声明如下：

```
// 获得响应解析函数的序列。
func getResponseParsers() []analyzer.ParseResponse {
    parsers := []analyzer.ParseResponse{
        parseForATag,
    }
    return parsers
}
```

编写响应解析函数恐怕是定制工作中最复杂的一部分了。不过，它也是最灵活的一部分。它不但可以让我们自由地掌控对响应的解析过程，还使我们可以间接地控制爬取流程的后续执行过程（通过返回特定的请求）。相比之下，条目处理器的编写就显得容易一些了。因为它们所处的位置在爬取流程的末端。不过，这并不代表它们不重要。我们仍然可以在其中做很多事，比如把条目存储到文件系统或者通过网络发送给远程服务器。

由于只出于演示的目的，所以我们在这里只实现一个极简的条目处理器。它的完整声明如下：

```
// 条目处理器。
func processItem(item base.Item) (result base.Item, err error) {
    if item == nil {
        return nil, errors.New("Invalid item!")
    }
    // 生成结果
    result = make(map[string]interface{})
    for k, v := range item {
        result[k] = v
    }
    if _, ok := result["number"]; !ok {
        result["number"] = len(result)
    }
    time.Sleep(10 * time.Millisecond)
    return result, nil
}
```

在`processItem`函数中，我们只是向条目添加了一个代表原有信息数量的键值对而已。请注意，我们在延迟10毫秒之后才返回结果值。这也是为了演示的需要。在下一节我们会看到这样做所产生的效果。

与前面的`getResponseParsers`函数类似，我们只在`getItemProcessors`函数中加入该条目处理器一次：

```
// 获得条目处理器的序列。
func getItemProcessors() []pipeline.ProcessItem {
    itemProcessors := []pipeline.ProcessItem{
        processItem,
    }
    return itemProcessors
}
```

现在，我们为开启调度器所做的准备工作都已完成。剩下的开启调度器以及激活整个爬取流程的代码就很简单了：

```
// 创建调度器
scheduler := sched.NewScheduler()
// 开启调度器
scheduler.Start(
    channelArgs,
    poolBaseArgs,
    crawlDepth,
    httpClientGenerator,
    respParsers,
    itemProcessors,
    firstHttpRequest)
```

我们把上述的演示代码都放入webcrawler/demo代码包的命令源码文件demo.go中。注意，准备启动参数以及创建、开启调度器的那些代码都应该放到这个命令源码文件的main函数中。

现在，让我们使用go run来运行demo.go文件中的代码：

```
hc@ubt:~/golang/goc2p/src/webcrawler/demo$ go run demo.go
hc@ubt:~/golang/goc2p/src/webcrawler/demo$
```

在执行上述命令之后，我们会发现，程序好像很快就运行结束了。这是怎么回事呢？请读者回顾myScheduler类型的Start方法的实现。如果读者仔细读过前面几章的话，肯定很快就能找到答案。

正如我们讲过的那样，调度器的Start方法把调度各个处理模块的工作全部异步化了。该方法在向请求缓存放入首次请求之后就马上会正常地结束执行。然而，虽然对Start方法的执行结束了，但爬取流程的执行却仍在继续，并且刚刚开始。不过，运行demo.go文件中的main函数的主Goroutine可不管这些。一旦完成了对Start方法的调用，它就马上会从Grunning状态转出。主Goroutine的运行完成就意味着整个Go程序的运行的结束，不论是否还有其他Goroutine在运行。这就是我们运行的程序很快就结束了的原因。它甚至还没来得及对首次请求进行处理。

那么我们怎么解决这个问题呢？如果读者马上就能想到使用sync.WaitGroup类型值来延迟主Goroutine的执行结束的话，我会非常地高兴。因为这说明你真的把前几章讲述的知识记住了。那真是太棒了！

我们确实可以给myScheduler类型添加一个sync.WaitGroup类型的字段。还记得吗？我们无须对这种类型的字段进行额外的初始化。假设该字段被命名为wg。我们可以在Start方法中的sched.startDownloading()语句之前添加这样一条语句：

```
wg.Add(4)
```

并在最后那条return语句的前面加入语句：

```
wg.Wait()
```

然后，我们再分别在调度器的startDownloading、activateAnalyzers、openItemPipeline和schedule方法中的最外层的那个go函数的函数体的开始处加入如下语句：

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

如此一来，只有上述4个方法中的go函数都被执行完成之后，Start方法才会被结束执行。在demo.go文件的main函数中，对调度器的Start方法的调用会一直被阻塞（确切地说，是会被阻塞在Start方法中的那条wg.Wait()语句上）。这就意味着主Goroutine会一直等待，直到爬取流程执行结束。不过，请注意，爬取流程实际上永远不会自己结束，即使是在其中流转的数据都已被处理完成的情况下。除非调度器所持的停止信号（由它的stopSign字段代表）被发出。而在我们编写的调度器实现中，只有它的Stop方法被调用，这个停止信号才会被发出。也就是说，我们必须显式地调用调度器的Stop方法。在demo.go文件的main函数中，我们可以另行启用一个Goroutine来调用调度器的Start方法。然后，在主Goroutine中适时的调用调度器的Stop方法，如下所示：

```
go func() {
    scheduler.Start(
        channelArgs,
        poolBaseArgs,
        crawlDepth,
        httpClientGenerator,
        respParsers,
        itemProcessors,
        firstHttpRequest)
}()
// 省略若干条语句，这些语句可以使我们在适当的时候停止调度器。
// 停止调度器
scheduler.Stop()
```

请记住，只有我们在像前面那样为调度器添加了sync.WaitGroup类型的字段以及相应的语句的前提下，这样做才会有意义。

上面这段代码中的注释已经在提示一个新的需要我们考虑的问题了——什么是适当的时候？如果停止过早，那么肯定就会使爬取流程对一些数据的处理被中断。如果停止过晚，虽然看起来不会造成太大问题，但是肯定是不得当的。程序看起来会显得很蹩脚，并且还会白白地浪费系统资源。那么怎样才能适当的时候停止调度器呢？

实际上，我们在设计网络爬虫框架的时候已经考虑到了这个问题，并将解决方案融入到该框架的设计和实现之中了。读者还记得网络爬虫框架中的各个组件的Summary（还有summary）方法，以及调度器的Idle方法吗？

在下一小节，我们就会利用这些方法来实现一个非常有意思的函数。这个函数既可以使我们能够在适当的时候结束爬取流程的执行，又可以实时地监控爬取流程的执行状况。我们把这个函数称为调度器监控函数。

不过，在编写这个函数之前，先让我们把调用Start方法的代码复原，即不另启用一个Goroutine来调用调度器的Start方法，并删除掉对调度器的Stop方法的调用。同时，我们也不用

为调度器添加那个`sync.WaitGroup`类型的字段以及那些相关的语句。请放心，即使不改动这些代码，我们也照样能达到目的。

9.8.3 调度器监控函数

我们的调度器监控函数的主要功能（或者说任务）有3个。

- 在适当的时候停止自身和调度器。
 - 实时监控调度器及其调度或持有的各个组件的运行状况。
 - 一旦调度器及相关组件在运作过程中发生任何错误，及时予以报告。
- 一如既往地，这些功能的具体实现应该是可以被定制的。

1. 确定参数

对于第一个功能，我们需要明确一点：只有在调度器所管辖的处理模块都空闲一段时间之后，调度器才应该被关闭。所以，我们要定时循环地去调用调度器的`Idle`方法，以检查它们是否空闲。如果连续若干次的检查结果均为`true`，那么我们就可以断定再没有新的数据需要被处理了。这时，关闭调度器就是安全的。这里有两个可以灵活掌握的环节：一个是检查的间隔时间，另一个是检查结果连续为`true`的次数。一旦确定了这两个可变量，我们就可以明确在调度器中的处理模块连续空闲多长时间之后才应该关闭调度器。这可以由一个简单的等式来体现：

持续空闲时间 = 检查间隔时间 × 检查结果连续为“真”的次数

另一方面，我们也应该可以决定：当实际情况满足了“持续空闲时间”这个条件之后是否由调度器监控函数来关闭调度器。

调度器监控函数的第二个功能是对运行状况的监控。这些组件的`Summary`或`summary`方法可以提供足够的摘要信息。这就能够让我们实时地了解到它们的运行状况。这里也存在两个可变量：是否需要详细的摘要信息，以及把这些摘要信息记录在哪儿。实际上，该函数的第三个功能也是需要使用到第二个可变量的。

经过上述分析，我们就可以确定调度器监控函数的签名了。我们把该函数命名为`Monitoring`。它的声明如下：

```
// 调度器监控函数。
// 参数scheduler代表作为监控目标的调度器。
// 参数intervalNs代表检查间隔时间，单位：纳秒。
// 参数maxIdleCount代表最大空闲计数。
// 参数autoStop被用来指示该方法是否在调度器空闲一段时间（即持续空闲时间，由intervalNs * maxIdleCount
// 得出）之后自行停止调度器。
// 参数detailSummary被用来表示是否需要详细的摘要信息。
// 参数record代表日志记录函数。
// 当监控结束之后，该方法会向作为唯一返回值的通道发送一个代表了空闲状态检查次数的数值。
func Monitoring(
    scheduler sched.Scheduler,
    intervalNs time.Duration,
    maxIdleCount uint,
    autoStop bool,
```

```
detailSummary bool,
record Record) <-chan uint64
```

读者应该可以从上述声明中找到我们前面分析出的所有可变量。在Monitoring函数的参数声明列表中的参数record就是调用方需要定制的相关信息的记录方式。我们还未曾提及它的类型Record。这个类型的声明很简单，如下：

```
// 日志记录函数的类型。
// 参数level代表日志级别。级别设定：0：普通；1：警告；2：错误。
type Record func(level byte, content string)
```

该函数类型所表达的功能就是根据日志的级别来记录它们。

另外，Monitoring函数在接受灵活定制的同时，还会返回一个通道。在这个函数被执行结束之时，它还会向该通道发送一个数值。这个数值会表示出它对那些处理模块的空闲状态进行检查的总次数。使用方可以通过这个总次数计算出当次爬取流程的总执行时间。不过，这个通道的更重要的作用是作为通知使用方已经可以安全关闭调度器（也表明监控过程已结束）的渠道。

2. 制定监控流程

读者可能会发现，调度器监控函数Monitoring的3个功能之间是没有交集的。因此，我们应该在实现该函数的时候保持这3个功能实现的独立性，并避免它们彼此干扰。这一点，应该可以从它的实现上看起来。Monitoring函数的函数体中的代码是这样的：

```
if scheduler == nil { // 调度器不可能不可用！
    panic(errors.New("The scheduler is invalid!"))
}
// 防止过小的参数值对爬取流程的影响
if intervalNs < time.Millisecond {
    intervalNs = time.Millisecond
}
if maxIdleCount < 1000 {
    maxIdleCount = 1000
}
// 监控停止通知器
stopNotifier := make(chan byte, 1)
// 接收和报告错误
reportError(scheduler, record, stopNotifier)
// 记录摘要信息
recordSummary(scheduler, detailSummary, record, stopNotifier)
// 检查计数通道
checkCountChan := make(chan uint64, 2)
// 检查空闲状态
checkStatus(scheduler,
    intervalNs,
    maxIdleCount,
    autoStop,
    checkCountChan,
    record,
    stopNotifier)
return checkCountChan
```

我们简要地解释一下这段代码体现的监控流程。首先，我们要对被传入函数的参数值进行检

查。其中最重要的是代表调度器实例的scheduler。如果它为nil, 那么这个监控流程就完全没有执行的必要了。我们在发现此情况时, 会把它视为一个致命的错误并引发一个运行时恐慌。除此之外, 我们还需要对检查间隔时间和最大空闲计数的值进行检查。它们的乘积即是我们前面所说的持续空闲时间。这两个值即不能是负数, 也不能是过小的正数。过小的正数会影响到爬取流程的正常执行。所以, 我们分别为它们设定了两个最小值。

我们打算让实现那3个主要功能的代码并发地执行。就像在调度器的实现中所做的那样, 我们需要让它们知道什么时候需要停止。不过, 不同的是, 我们在这里使用一个通道来传递停止信号。我们在前面提到过这种方案。对于组件众多或数量不定的情况, 它并不太适合。但是对于这里的简单场景来说, 这样做会很方便。这个停止信号通道由变量stopNotifier代表。

函数reportError、recordSummary和checkStatus分别代表了Monitoring函数需要实现的那3个主要功能。Monitoring函数会把它们视为自己的组成部分, 并分别调用它们以完成功能。正如刚刚所说, 它们都会启用新的Goroutine来执行其中的代码。另外, 调用这些辅助函数的顺序是与它们对即时性的要求有关的。读者可以思考一下为什么会以这样的顺序调用它们。

对于这段代码, 最后要说明的是变量checkCountChan。它代表的就是被用来传递空闲状态检查总次数的通道。它的值会被传入checkStatus函数, 并被Monitoring函数返回给它的调用方。

3. 报告错误

报告错误的功能由reportError函数负责实现。一旦调度器被开启, 我们就应该通过调用它的ErrorChan方法获取错误通道, 并不断地尝试从中接收错误值。我们已经在前面详述过这样做的原因。

函数reportError接受3个参数。除了代表调度器的scheduler之外, 还有我们代表日志记录方式的record, 以及代表监控停止通知器的stopNotifier。下面是它的完整声明:

```
// 接收和报告错误。
func reportError(
    scheduler sched.Scheduler,
    record Record,
    stopNotifier <-chan byte) {
    go func() {
        // 等待调度器开启
        waitForSchedulerStart(scheduler)
        for {
            // 查看监控停止通知器
            select {
            case <-stopNotifier:
                return
            default:
            }
            errorChan := scheduler.ErrorChan()
            if errorChan == nil {
                return
            }
            err := <-errorChan
            if err != nil {
                errMsg := fmt.Sprintf("Error (received from error channel): %s", err)
            }
        }
    }
}
```

```

        record(2, errMsg)
    }
    time.Sleep(time.Microsecond)
}
}()
}

```

这个函数会启用一个Goroutine来执行其中的代码。在每次迭代的开始，我们都用select语句尝试着从stopNotifier通道接收元素值。只要立即接收到了一个元素值，就马上返回以结束它所在的go函数的执行。其中的default case意味着这个接收操作只是尝试一下而已。如果当时的stopNotifier通道中没有可用的元素值，那么select语句就会立即结束执行。

在这之后，我们调用调度器的ErrorChan方法，并在确保该错误通道不为nil的前提下尝试着从中接收元素值。与针对stopNotifier通道的操作不同，这里的接收操作是阻塞式的。一旦接收到任何元素值，我们就调用record参数代表的函数记录它。不过，如果在接收的过程中恰巧错误通道被关闭了，那么该接收操作就会立即结束并返回一个该通道的元素类型的零值。对于error类型来讲，这个零值就是nil。这也是我们在记录之前要进行非nil检查的原因。

请注意，在reportError函数包含的那个for代码块的最后有一条针对time.Sleep函数的调用语句。之所以添加这样一条语句是为了要避免for循环被迭代得太过频繁可能会带来的一些问题，比如，挤掉其他Goroutine被运行的机会、致使CPU的使用率过高，等等。这是一种保护措施，虽然那些问题不一定会发生。

接收和报告错误的流程就是这样。细心的读者可能会发现，在reportError函数的函数体中还包含了陌生的调用语句waitForSchedulerStart(scheduler)。这条语句上面的注释其实已经说明了它调用的函数的功能。我们都知道，调度器有一个公开的方法Running。该方法会返回一个bool类型值以表示调度器是否正在运行。因此，waitForSchedulerStart函数要做的就是不断地调用调度器的Running方法，直到调度器已被完全开启为止。前者的完整声明如下：

```

// 等待调度器开启。
func waitForSchedulerStart(scheduler sched.Scheduler) {
    for !scheduler.Running() {
        time.Sleep(time.Microsecond)
    }
}

```

这里调用waitForSchedulerStart函数的目的是保证只在调度器被完全开启之后再去做需要做的事。在后面要讲到的另外两个主要功能的实现中，我们也会这样做。

4. 记录摘要信息

函数recordSummary负责记录摘要信息。该函数的声明如下：

```

// 记录摘要信息。
func recordSummary(
    scheduler sched.Scheduler,
    detailSummary bool,
    record Record,
    stopNotifier <-chan byte)

```

可以看到，它接受4个参数。其中的3个参数也是reportError函数所接受的。多出的那个参数是detailSummary。它决定了摘要信息的详细程度。

这个函数同样启用了Goroutine来专门进行相关操作。与reportError函数相同，我们依然要在其中的go函数的开始处等待调度器的完全开启。一旦调度器被开启，我们就要开始为摘要信息的获取、比对、组装和记录做好准备。这里需要先声明如下几个变量：

```
var recordCount uint64 = 1
startTime := time.Now()
var prevSchedSummary sched.SchedSummary
var prevNumGoroutine int
```

其中，变量recordCount和startTime的值会参与到最终的摘要信息的组装过程中去。前者代表了记录的次数，而后者则代表了开始准备记录时的时间。在它们后面声明的两个变量prevSchedSummary和prevNumGoroutine的含义分别是前一次获得的调度器摘要信息和Goroutine数量。它们是否真正需要记录下当次的摘要信息的决定因素。我们每次都会把当前获取到的摘要信息的各个组成部分与前一次的相比对。只有在确定它们不同之后，我们才会对当前的摘要信息予以记录。这主要是为了减少摘要信息对网络爬虫框架本身产生的日志的干扰。

我们应该在停下来之前定时且循环的获取和比对摘要信息。因此，我们把后面的代码放到了一个for代码块中。在迭代的开始，我们仍然需要去检查stopNotifier通道：

```
for {
    // 查看监控停止通知器
    select {
    case <-stopNotifier:
        return
    default:
    }
    // 省略若干条语句
}
```

如果停止信号还没有被发出，那么我们就开始着手获取摘要信息的可变部分。在这份摘要信息中，我们需要关注的可能变化的部分只有两个，即上面所说的Goroutine数量和调度器摘要信息。Goroutine数量代表的是当前的Go运行时系统中正处于运行状态的Goroutine的数量，而调度器摘要信息则体现了调度器当前的状态。获取它们的方式如下：

```
// 获取摘要信息的各组成部分
currNumGoroutine := runtime.NumGoroutine()
currSchedSummary := scheduler.Summary(" ")
```

在得到它们的确切值之后，我们就要分别把它们与变量prevNumGoroutine和prevSchedSummary的值进行比较。这里的比较操作很简单。因为变量currNumGoroutine及prevNumGoroutine都是int类型的，而调度器摘要信息的类型SchedSummary也有可判断相同性的Same方法。如果它们两两相同，那么我们就不再进行后面的组装和记录操作了。因为既然相同的摘要信息已经被记录过了，那么就没有必要再来一次了。否则，我们就开始组装摘要信息。这里所说的组装其实也就是采用我们已多次用到的模板加参数的方式来生成摘要信息。在这个过程中，我们会用到参数detailSummary

的值。在组装完成之后，我们会调用参数record所代表的日志记录函数来记录这个摘要信息。最后，我们把currNumGoroutine变量的值赋给prevNumGoroutine变量，并把currSchedSummary变量的值赋给prevSchedSummary变量。只要这样，后续的比对操作的结果才会正确。除此之外，我们还要递增代表了记录次数的recordCount变量的值。

与刚刚这段描述对应的代码是这样的：

```
// 比对前后两份摘要信息的一致性。只有不一致时才会予以记录。
if currNumGoroutine != prevNumGoroutine ||
    !currSchedSummary.Same(prevSchedSummary) {
    schedSummaryStr := func() string {
        if detailSummary {
            return currSchedSummary.Detail()
        } else {
            return currSchedSummary.String()
        }
    }()
    // 记录摘要信息
    info := fmt.Sprintf(summaryForMonitoring,
        recordCount,
        currNumGoroutine,
        schedSummaryStr,
        time.Since(startTime).String(),
    )
    record(0, info)
    prevNumGoroutine = currNumGoroutine
    prevSchedSummary = currSchedSummary
    recordCount++
}
```

其中，time.Since函数被调用之后会返回一个代表了当前时间与参数值给定时间的时间间隔的time.Duration类型值。调用这个值的String方法就会得到一个描述这个时间间隔的字符串。这个描述字符串会是易读的。

另外，作为摘要信息模板的全局变量summaryForMonitoring的声明是这样的：

```
// 摘要信息的模板。
var summaryForMonitoring = "Monitor - Collected information[%d]:\n" +
    "  Goroutine number: %d\n" +
    "  Scheduler:\n%s" +
    "  Escaped time: %s\n"
```

经此模板生成的摘要信息类似于：

```
Monitor - Collected information[14]:
  Goroutine number: 8
  Scheduler:
    Running: false
    Channel args: { reqChanLen: 10, respChanLen: 10, itemChanLen: 10, errorChanLen: 10 }
    Pool base args: { pageDownloaderPoolSize: 3, analyzerPoolSize: 3 }
    Crawl depth: 0
    Channels manager: status: closed, requestChannel: 0/10, responseChannel: 0/10, itemChannel: 0/10,
    errorChannel: 0/10
```

```

Request cache: status: closed, length: 0, capacity: 1
Downloader pool: 0/3
Analyzer pool: 0/3
Item pipeline: failFast: true, processorNumber: 1, sent: 39, accepted: 39, processed: 39,
processingNumber: 0
Urls(0): <concealed>
Stop sign: signed: true, dealCount: map[scheduler:0]
Escaped time: 10.563s

```

函数recordSummary中的go函数所包含的for代码块基本上就是如此。此外，为了让这个for循环在迭代之间能有一个小小的间隙，我们还需要把下面这条语句放在for代码块的最后：

```
time.Sleep(time.Microsecond)
```

这样做的原因我们已经在前面说过了。实际上，与reportError函数相比，recordSummary函数更有必要加入这条语句。

对摘要信息的获取、比对、组装和记录的操作是独立进行的。它由Monitoring函数启动，并会在接收到停止信号之后结束。发送停止信号的代码存在于checkStatus函数中。因为只有它才知道什么时候应该停止监控。

5. 检查状态

函数checkStatus的主要功能是定时地检查调度器所管辖的各个处理模块是否都处于空闲的状态，并在这种空闲状态持续一段时间之后停止监控（以及停止调度器）。除此之外，该函数还承担着发送停止信号和检查计数值的任务。该函数的职责相对较多，并且其可定制化程度是在Monitoring函数调用的这些辅助函数之中最高的。因此，它接受的参数的数量也比较多，共有7个。这比Monitoring函数的参数数量还要多。checkStatus函数的声明如下：

```

// 检查状态，并在满足持续空闲时间的条件时采取必要措施。
func checkStatus(
    scheduler sched.Scheduler,
    intervalNs time.Duration,
    maxIdleCount uint,
    autoStop bool,
    checkCountChan chan<- uint64,
    record Record,
    stopNotifier chan<- byte)

```

其中的参数我们都介绍过。不过请注意，参数checkCountChan和stopNotifier的类型都是某类发送通道。请读者回顾reportError函数和recordSummary函数的参数声明列表中名为stopNotifier的参数的类型，并体会其中的道理。

函数checkStatus中的所有代码几乎都被包含于其中的go函数之内，除了这条语句：

```
var checkCount uint64
```

变量checkCount代表的就是检查计数。把它声明在go函数之外是为了与go函数中的defer语句相配合。这条defer语句是go函数中的第一条语句。下面这段代码展现了它们的位置关系：

```

var checkCount uint64
go func() {
    defer func() {

```

```

        stopNotifier <- 1
        stopNotifier <- 2
        checkCountChan <- checkCount
    }()
    // 省略若干条语句
}

```

它的作用是保证在go函数的执行即将结束的时候发送停止信号和检查计数值。这个时机非常关键。这里的go函数总会在达到持续空闲时间的时候结束执行。

下面，我们来详述在上面那段代码中被省略的那些语句。编写它们的目的基本上就是准确地判定持续空闲时间的到达。为了让这一判定有据可依，下面这个变量是必需的：

```
var idleCount uint
```

它的值将会代表实际的连续空闲状态的计数。同时，为了记录真实的持续空闲时间，我们还需要声明一个这样的变量：

```
var firstIdleTime time.Time
```

注意，真实的持续空闲时间与理想的持续空闲时间（由变量intervalNs和maxIdleCount的值相乘得出的那个时间）之间肯定是有偏差的。并且，前者肯定会大于后者。因为执行相关的代码也是需要耗时的。

可以肯定的是，我们需要在一个for循环中进行与持续空闲时间判定有关的那些操作。由于这条for语句中的条件判断比较多且复杂，所以我们直接贴出它们然后再进行解释：

```

for {
    // 检查调度器的空闲状态
    if scheduler.Idle() {
        idleCount++
        if idleCount == 1 {
            firstIdleTime = time.Now()
        }
        if idleCount >= maxIdleCount {
            msg :=
                fmt.Sprintf(msgReachMaxIdleCount, time.Since(firstIdleTime).String())
            record(0, msg)
            // 再次检查调度器的空闲状态，确保它已经可以被停止
            if scheduler.Idle() {
                if autoStop {
                    var result string
                    if scheduler.Stop() {
                        result = "success"
                    } else {
                        result = "failing"
                    }
                    msg = fmt.Sprintf(msgStopScheduler, result)
                    record(0, msg)
                }
                break
            } else {
                if idleCount > 0 {

```

```

        idleCount = 0
    }
}
} else {
    if idleCount > 0 {
        idleCount = 0
    }
}
checkCount++
time.Sleep(intervalNs)
}

```

可以看到，我们总是保证checkCount的值会随着迭代一次又一次的进行而递增，同时也会根据使用方的要求让每次迭代之间存在一定的时间间隔。

在这个for代码块的最开始，我们通过调用调度器的Idle方法来判断其中的处理模块是否都处于空闲状态。如果不是，那么我们就在必要时清零idleCount的值。注意，idleCount的值所代表的是连续空闲状态的计数。所以，一旦发现它们未处于空闲状态，我们就要重新进行计数。另一个方面，如果发现它们正处于空闲状态，我们就需要递增idleCount的值。并且，如果发现这是一个新的计数周期的开始，那么就应该把firstIdleTime的值设置为当前时间。只有这样才能在idleCount的值达到最大空闲计数的时候，由firstIdleTime的值准确地计算出真实的持续空闲时间。

在做好计数和起始时间的检查和校正工作之后，我们会马上把idleCount的值与最大空闲计数相比较。如果前者大于或等于后者，那么就可以初步判定调度器中的那些处理模块已经空闲了足够长的时间。这时，我们立刻记录一条基于模板msgReachMaxIdleCount生成的消息。msgReachMaxIdleCount模板的声明如下：

```

// 已达到最大空闲计数的消息模板。
var msgReachMaxIdleCount = "The scheduler has been idle for a period of time" +
    " (about %s)." +
    " Now consider what stop it."

```

可以看到，这条消息建议网络爬虫框架的使用方关闭调度器。不过，使用方可以通过把autoStop参数的值设置为true，来让调度器监控函数自动关闭调度器。这也是这里第二次调用调度器的Idle方法的原因之一。如果这里的调用结果值和autoStop参数的值均为true，那么checkStatus函数就去帮助使用方停止调度器。如果再次调用scheduler.Idle方法时得到的结果值为false，那么idleCount变量的值也会在必要时被清零。对持续空闲状态的计数将重新开始。这是显然一种比较保守的做法，但却可以有效地避免过早的停止调度器。实际上，只要第二次调用该方法的结果值为true，不管autoStop参数的值是怎样的，我们都会退出当前的for代码块。对for代码块的执行的结束就意味着checkStatus函数中的go函数的执行的结束。在这个go函数即将被结束执行之时，向停止信号通道stopNotifier和检查计数通道checkCountChan的发送操作就会被进行。向前者发送的两个元素值会使Monitoring函数的其他两个辅助函数中的go函数陆续结束执行，而向后者发送的那一个元素值则意味着告诉使用方已经可以安全地关闭调度器了。由于这里

向checkCountChan通道发送的一定是那个检查计数值。所以Monitoring函数的使用方还可以由此得到监控停止时的空闲状态检查的次数。

至此，我们展现和说明了checkStatus函数以及Monitoring函数会执行到的所有代码。

6. 使用调度器监控函数

当然，编写Monitoring函数以及相关程序实体的目的就是为了让网络爬虫框架的使用方能够更容易地了解到爬取流程的运转状况，并可以在适当的时候停掉调度器以及爬取流程。我们把以上这些程序实体都存放到了webcrawler/tool代码包的库源码文件monitor.go中。

为了使用调度器监控函数，我们需要先在webcrawler/demo代码包的命令源码文件demo.go中导入webcrawler/tool代码包，然后再在它的main函数中添加一些代码。为了保证调度器在被开启的那一刻起就接受Monitoring函数的监控，我们需要把调用Monitoring函数的语句安插在创建调度器和开启调度器的代码的中间。经修改，main函数中的代码如下：

```
// 创建调度器
scheduler := sched.NewScheduler()

// 准备监控参数
intervalNs := 10 * time.Millisecond
maxIdleCount := uint(1000)
// 开始监控
checkCountChan := tool.Monitoring(
    scheduler,
    intervalNs,
    maxIdleCount,
    true,
    false,
    record)

// 准备启动参数
// 省略若干条语句
// 开启调度器
// 省略若干条语句

// 等待监控结束
<-checkCountChan
```

可以看到，我们把检查间隔时间设置为10毫秒，并把最大空闲计数设置为1000。我们只让调度器监控函数记录基本的摘要信息，并授权它在持续空闲时间被达到之后自动地关闭调度器。为了让主Goroutine等待监控以及调度器的停止，我们还在这段代码的最后加入了对检查计数通道的接收操作。这样，只有监控和调度器都停止之后，主Goroutine才会结束执行。

在我们给予tool.Monitoring函数的参数值列表中，record代表的就是前文所说的日志记录函数。它的完整声明如下所示：

```
func record(level byte, content string) {
    if content == "" {
        return
    }
    switch level {
```

```
    case 0:
        logger.Infofn(content)
    case 1:
        logger.Warnln(content)
    case 2:
        logger.Infofn(content)
    }
}
```

该函数会根据参数level的值的不同，以不同的级别记录日志。

7. 运行演示代码

在经过上面这一番改造之后，我们可以再次运行demo.go文件。运行应该一切正常。调度器监控函数的日志会与网路爬虫框架的日志混在一起。如果你觉得这样很乱，可以实现自己的日志记录函数，以便分开记录这两部分日志。例如，把调度器监控函数的日志记录到文件中，然后在另一个命令行环境下使用tail -f命令实时查看它的增量（仅在类Unix操作系统下有效）。

读者应该在自己的计算机上运行demo.go文件。然后，在改变一些参数的值甚至网络爬虫框架中的代码之后再去运行它，看看在效果上会有什么样的变化。这样可以更快地理解网络爬虫框架的设计和实现，并把Go语言并发编程的相关知识完全掌握在手。

9.9 当前的不足和解决思路

如果读者读懂了我在上一节编写的演示程序的话，那么现在应该已经能够根据自己的需要运用本章所描绘的网络爬虫框架了。

网络爬虫框架的代码向我们展示了Go语言在并发编程方面的威力，以及Go语言的众多特性为软件开发者提供的种种便利。Go语言程序在开发效率和运行效率方面是双赢的。不过，由于思考和开发的时间有限，我们的网络爬虫框架程序难免存在一些不足和缺陷。本节列举了该程序的一些比较明显的问题，并提供了相应的解决思路。当然，问题很可能不只这些。当前的程序也可能并不能满足你的需要。如果是这样，欢迎读者通过任何有效的方式告知我，或者直接向该项目贡献代码。

下面，我们来说一说当前的网络爬虫框架中比较突出的几个问题。

1. 不支持对请求过滤的定制

仔细阅读9.7节的读者肯定还记得，调度器在把请求存入请求缓存之前进行了一番过滤。下面这几类请求在当前会被过滤掉。

- ❑ 包括各种无效的请求。
- ❑ URL的Scheme不是HTTP的请求。
- ❑ URL已被处理过的请求。
- ❑ 主域名与首次请求的主域名不同的请求。
- ❑ 深度大于最大爬取深度的请求。

可是，加入这些过滤条件真的有必要吗？无效的请求肯定无法也不应该被处理。深度大于最大爬取深度的请求也应该被过滤。那么，不基于HTTP协议的请求就不应该被处理吗？从程序功

能的角度看,这样做是狭隘的。但从现有的技术支持情况说,我们不得不这样做。因为现在的网页下载器还不支持基于其他协议的请求。对第3类和第4类请求的过滤是符合我们之前描述的需求的。但是,这是否一定会满足网络爬虫框架的真实使用方的需求呢?答案显然是否定的。甚至,我们在本章的开始说明的那些需求也肯定不是一成不变的。所以,让使用方可以自己定义过滤条件是很有必要的。

由于过滤请求的代码就直接存在于调度器实现的内部,所以我们只需要考虑怎样通过API向调度器传递请求过滤规则,而不必考虑对任何处理模块进行改动。

关于请求规律规则的具体传递方式,我们就不在这里详述了。不过,请注意,如果程序已经发布并成为了正式版本,那么我们就应该添加新的API,而不应该修改现有的API。

2. 默认不支持Cookie

在网络爬虫框架中,我们使用net/http代码包中的Client类型的实例向目标网站发送HTTP请求。与网络浏览器(比如IE、Firefox和Chrome等)这类成熟的软件相比,这样的方式会显得简陋很多。不过,http.Client类型本身也提供了一些方式,以使我们在与目标网络站点交互的时候有更多的选择。这也是我们向网络爬虫框架的使用者提供定制HTTP客户端的方法的主要原因。

对于Cookie这个具体问题,我们可以通过对http.Client类型实例的定制来实现。熟悉互联网软件的读者应该都知道Cookie的含义和用途。它一般会由服务端生成并被存放在客户端。例如,我们使用作为客户端的网络浏览器访问作为服务端的网络站点。在成功使用用户名和密码登录该服务端之后,服务端会向客户端发送一个可以标识客户端身份的Cookie。那么,在该Cookie的有效期之内,当我们再次通过同一个客户端访问该服务端的时候,客户端就会将相应的Cookie回传给服务端。这样,在我们看来,服务端就可以自动地知道我们是哪一个用户了。我们无须再次登录。这个例子虽然略去了很多细节,但是可以说明为网页下载器提供一个带有Cookie存储功能的HTTP客户端的重要性。我们在爬取一个需要登录才能访问的网络站点上的网页的时候,会非常需要这样的功能。

类型http.Client中有一个公开的字段Jar。该字段的类型是http.CookieJar的。http.CookieJar是一个接口类型。通过调用这个Jar字段值的方法,HTTP客户端就可以对Cookie进行存取了。这个接口类型的一个立即可用的实现类型是Jar类型。它的声明存在于标准库代码包net/http/cookiejar中。cookiejar.Jar类型提供了一种基于内存的Cookie存储方案。

我们可以使用cookiejar.New函数创建并初始化一个cookiejar.Jar类型的值。不过,我们必须向该函数传入一个选项集。这个选项集由*cookiejar.Options类型的值代表。虽然叫选项集,但是在它之中只存在一个cookiejar.PublicSuffixList接口类型的选项(或称字段)。该接口类型的用途基本体现在它的PublicSuffix方法上。该方法的功能定义是从一个域名中提取主域名。还记得吗?在前面,我让读者自行编写具有相同功能的函数getPrimaryDomain。也许它就能在这里派上用场。因为我们为http.Client类型值附加存取Cookie的功能的关键就在于实现cookiejar.PublicSuffixList接口类型。

但愿读者已经实现了getPrimaryDomain函数。不过,如果没有,我们还有另一种选择。那就

是使用go.net项目中的现成实现。我们可以通过go get命令下载并安装它：

```
go get code.google.com/p/go.net/publicsuffix
```

在安装好这个代码包之后，我们可以对其中的API简单地封装一下，以编写出自己的cookiejar.PublicSuffixList接口的实现类型：

```
// cookiejar.PublicSuffixList 接口的实现类型。
type myPublicSuffixList struct{}

func (psl *myPublicSuffixList) PublicSuffix(domain string) string {
    suffix, _ := publicsuffix.PublicSuffix(domain)
    return suffix
}

func (psl *myPublicSuffixList) String() string {
    return "Web crawler public suffix list (rev 1.0) power by" +
        'code.google.com/p/go.net/publicsuffix'
}
```

有了myPublicSuffixList类型，我们就可以正常地初始化一个*cookiejar.Jar类型值了，如：

```
options := &cookiejar.Options{PublicSuffixList: &myPublicSuffixList{}}
cj, _ := cookiejar.New(options)
```

最后，在初始化HTTP客户端的时候，我们可以把这样的*cookiejar.Jar类型值赋给HTTP客户端的Jar字段：

```
http.Client{Jar: cj}
```

这样，我们的HTTP客户端就支持Cookie的存取操作了。

3. 关于基于HTTPS协议的请求

我们在说明问题1的时候讲过网络爬虫框架的支持协议单一的问题。对于Scheme为HTTPS的请求来说，现有的HTTP客户端可以应付。但是，如果这个目标网站的数字证书是有问题的，那么与它的交互肯定是无法达成的。因为http.Client类型值的Do方法在这种情况下必定会返回非nil的错误值。

一个简单的解决方案就是去掉安全校验环节。这同样可以通过定制HTTP客户端来实现，如下所示：

```
tr := &http.Transport{
    TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
}
client := &http.Client{Transport: tr}
```

其中的限定标识符tls.Config代表的是标准库代码包crypto/tls中的Config类型。该类型中的InsecureSkipVerify字段的值决定客户端是否忽略校验目标网站的数字证书链。

如果我们确定目标网站是安全可信的，那么这样的解决方案就是可行的。它对于我们这里的应用场景来说也行得通，只是后果需要自负。

4. 未遵循Robots协议

Robots协议也常被称为网络爬虫协议。网络站点会在其根目录下放置一个以ASCII编码的文

本文件robot.txt。这个文本文件就是Robots协议文件。它的内容类似于：

```
User-agent: BadBot
Disallow: /

User-agent: Googlebot
Allow: /*.htm$
Disallow: /*.jpg$
```

该文件会以统一的格式编写。它会告诉爬取其所属网站的网络爬虫程序哪些网页可以爬取，而哪些不能爬取。Robots协议旨在保护网络内容的提供者和生产者的权益。对于这种权益，我们也应该维护。

当然，Robots协议是一种君子协定。它属于业界公认的道德规范的范畴。没有哪一个网络站点会强制网络爬虫程序读取和遵循Robots协议文件。况且，这样做也是不现实的。但是，我们的程序总是应该墨守这样的规范。

网络爬虫框架本身并未直接对自动获取、解析和遵循Robots协议文件提供支持。但是建议网络爬虫框架的使用方通过对该框架的定制完成上述功能。作为辅助，我们可以使用一些现成的第三方代码包，比如开源的github.com/PuerkitoBio/robotstxt.go和github.com/temoto/robotstxt-go。我们同样可以通过go get命令下载和安装它们。

5. 暂不支持分布式计算

很显然，我们编写的网络爬虫框架现在还不能以集群的方式部署和运行。我们现在还并未提供多个使用网络爬虫框架的程序之间的协作机制。当然，网络爬虫框架的使用方可以对该框架进行二次开发并赋予它分布式计算的能力。达到这一目的的方式有很多。比如，使用标准库代码包net/rpc提供的API在多个网络爬虫程序之间建立起指令和数据的收发管道，以使它们能够协同地完成某项任务。又比如，把网络爬虫程序做成独立的Web服务，并把这些Web服务置于各种基于面向服务架构的分布式软件系统当中去。如此即是通过融入另一个软件体系来达到它们彼此以及它们与其他程序之间的通讯和协作的目的。总之，开发者应该总是可以找到实现自己的设想的方式。

在看到了上述的这些问题之后，我们会发现当前的网络爬虫框架还存在很多改进和扩展的空间。它所提供的众多定制方式会让这些改进和扩展更加容易。虽然从软件生命周期的角度看该框架仍处于幼年时期，但是它先进的构架设计一定会使它走得更远。显而易见，对此Go语言起到了很大的积极作用。

9.10 本章小结

非常好！我们已经编写完成了一个规模不小的程序了。你已与我一起参与了该程序的需求分析、总体设计、详细设计和编码实现等多个环节。我希望你能够通过这样一个过程巩固在前面学习到的所有知识和技巧，并且能够把它们真正地运用到今后的编程工作中去。我还希望你能够把本章以及本书中的所有示例代码都下载到你的计算机中，并根据实际需要修改、重构和扩展它们。请记住，编程是一项非常讲究动手和实践的工作。所以，我在这本书中为大家讲述Go语言编程

以及并发编程知识只能算是一个指引，还不足以使你成为Go语言编程熟手甚至高手。你只能通过真正去编写一定量的代码达到此类目标。

希望这本书能够成为你编写Go语言程序时的手边书，也希望书中的示例代码能够对你有所帮助。我把这本书中的示例代码都放到了Github网站上，具体的下载地址是：<https://github.com/hyper-carrot/goc2p>。

知名的 Go 语言开源框架

1. Beego: 一个国产的HTTP框架。我们可以用它来快速地开发各种应用程序。它的官方网址是: <http://beego.me>。
2. Gogs: 一个国产的自助Git托管服务程序。我们可以用它来搭建自己的Git服务器。它的官方网址是: <http://gogits.org>。
3. Docker: 一个软件部署解决方案, 也是一个轻量级的应用容器框架。使用Docker, 我们可以轻松地打包、发布和运行任何应用。现在, Docker已经成为了名副其实的Go语言杀手级应用框架。其官方网址是: <https://www.docker.com>。当然, 你也可以访问非官方的中文网站<http://www.docker.org.cn>和<https://docker.cn>。
4. Skynet: 一个分布式服务框架。它可以帮助我们构建起大规模的分布式应用系统。它的源码被放置在了<https://github.com/skynetservices/skynet>上。
5. NSQ: 一个实时的分布式消息平台。它拥有很高的可伸缩性, 并能够每天处理数以十亿计的消息。它的官方网址是: <http://nsq.io>。
6. etcd: 一个高可用的键值存储系统。它可被用于建立共享配置系统和服务发现系统。它的灵感来自于Apache ZooKeeper。我们可以在<https://github.com/coreos/etcd>上找到它的源码。
7. Groupcache: 著名的内存缓存系统Memcached的作者用Go语言编写的一个与前者功能类似的函数库。作者想用它作为Memcached的替代者。其官方网址是<https://github.com/golang/groupcache>。
8. Gobot: 一个非常有意思的开源项目。它旨在成为下一代自动机工程学框架。换句话说, 我们可以用它来控制机器人! 它已经支持了10个不同的硬件平台。这其中包括已经被国内的计算机硬件发烧友所熟知的Arduino。该开源项目的官方网址是<http://gobot.io>。

国内的 Go 语言社区

1. Golangtc.com: 该社区是众多的Go语言中文社区中比较活跃的一个。我们可以从中获知很多Go语言方面的信息。它的网络地址是: <http://www.golangtc.com>。
2. Go友团: 该社区致力于为Go语言爱好者提供学习与交流的最佳环境。社区的组织者都非常热爱Go语言, 乐于分享, 且技术精湛。其中包括了Beego的创始人。我也在那里为本书开设了专栏。社区主页地址为: <http://golanghome.com>。
3. Go Walker: 这是一个可以在线生成并浏览基于Go语言的项目的API文档及黑客视图的Web服务器, 目前已支持包括 Bitbucket、GitHub、Google Code、Launchpad和Git@OSC在内的五大代码托管平台。其网络地址是: <https://gowalker.org>。
4. 在国内著名的开发者技术社区SegmentFault上开设有Go语言的专栏。网址是: <http://segmentfault.com/t/go>。大家可以去那里提交与Go语言有关的各种问题。

以上, 只是冰山的一角。更多的Go语言资源请读者自行在Google上搜索。